CHAPTER 2

Hashing

Hashing is a cornerstone of cryptographic security. It involves the concept of a *one-way function* or *fingerprint*. Hash functions only work well when a couple of things are true about them:

- They produce *repeatable, unique values* for every input.
- The output value *provides no clues* about the input that produced it.

Some hashing functions are better at satisfying these requirements than others, and we'll talk about some good ones (SHA-256) and some not-so-good ones (MD5, SHA-1) to demonstrate both how they work and why choosing a good one is so terribly important.

Hash Liberally with `hashlib`

WARNING: MD5 Is No Good

We are going to use an algorithm called MD5 for about the first half of the chapter. MD5 is deprecated and **should not be used for any security-sensitive operations**, or really any operations at all, except when you have to interact with legacy systems.

This discussion is for introducing the concept of hashing and for providing historical context. MD5 is nice for that because it produces short hashes, has a rich history, and *gives us something to break*.

When we last left our two favorite spies from East Antarctica, Alice and Bob were working out some codes using simple substitution ciphers. Even though the cipher was very weak, it provided a rudimentary form of message *confidentiality*.

S. J. Nielson and C. K. Monson, *Practical Cryptography in Python*,
https://doi.org/10.1007/978-1-4842-4900-0_2

It did nothing, however, for message *integrity*. If you haven't already guessed, message **confidentiality** means that nobody but the authorized parties can *read* the message. Message **integrity** means that no unauthorized parties can *change* the message without the change being noticed.

It is important to understand the distinction. Even with modern ciphers, just because a message can't be read doesn't mean it can't be altered, even in ways that make sense after decryption.

Also, when Alice and Bob go through customs at the WA border, sometimes their laptops are inspected. It would be nice to know that none of the files have been tampered with during that process.

Fortunately for Alice and Bob, their new technology officer introduces them to something called a "message digest" to "fingerprint" their files and message transmissions. He explains that they can combine their messages' *contents* with message *digests*, then using the two together, they can tell whether part of any message was altered. That sounds like just the thing!

Since they don't know anything about digests, it's time for some training. Let's follow along with their instructor in our own Python interpreter, starting with Listing 2-1.

Listing 2-1. Intro to hashlib

```
>>> import hashlib
>>> md5hasher = hashlib.md5()
>>> md5hasher.hexdigest()
'd41d8cd98f00b204e9800998ecf8427e'
```

Importing a library called `hashlib` seems straightforward enough, but what is `md5`?

The instructor explains that the "MD" in MD5 stands for "message digest." We'll get into some interesting details in just a moment, but for now, a digest like MD5 converts a document of any length (even an empty document) into a large number that takes up a fixed amount of space. It should have at least these features:

- The same document always produces the same digest.
- The digest "feels" random: if you have a digest, it gives you no clues about the document.

In this way, a digest is like a fingerprint and is sometimes called one: it is a small amount of data that stands in for the document's *identity*; every document we might ever care about should have a completely unique digest.

Human fingerprints are similar in other ways. If you have a person at hand, it's easy to produce a (relatively) consistent and unique fingerprint; but if the only thing you have is a fingerprint, it's not so easy to find out whose it is. Digests work the same way: given a document, it's easy to calculate its digest; but given only a digest, it's very hard to find out what document produced it. *Very* hard. The harder, the better, in fact.

The MD5 digest creates a number that always occupies 16 bytes of memory. In our example interpreter session, we asked it to produce a digest for the *empty document*, which is why we didn't add any data to the `md5hasher` before asking it to produce a digest for us. The use of `hexdigest` is shown to demonstrate a more human-readable format for the number, where each of the 16 bytes in the digest is shown as a two-character hexadecimal value.

The instructor, anxious to move on, asks Alice and Bob to hash each of their names (expressed as bytes). To the interpreter, and Listing 2-2!

Listing 2-2. Hash Names

```
>>> md5hasher = hashlib.md5(b'alice')
>>> md5hasher.hexdigest()
'6384e2b2184bcbf58eccf10ca7a6563c'
>>> md5hasher = hashlib.md5(b'bob')
>>> md5hasher.hexdigest()
'9f9d51bc70ef21ca5c14f307980a29d8'
```

For short strings like these, it's not uncommon to combine operations, like Listing 2-3.

Listing 2-3. Combine Operations

```
>>> hashlib.md5(b'alice').hexdigest()
'6384e2b2184bcbf58eccf10ca7a6563c'
>>> hashlib.md5(b'bob').hexdigest()
'9f9d51bc70ef21ca5c14f307980a29d8'
```

"So, Alice, Bob, what did you learn from this?" the instructor asks. When neither one answers, she suggests that they experiment some more. Let's follow along.

Python differentiates between Unicode strings and raw byte strings. A full explanation of the differences is beyond the scope of this book, but for almost all cryptographic purposes, you *must* use bytes. Otherwise you can end up with some very nasty surprises when the interpreter attempts (or refuses) to convert Unicode strings into bytes for you. We forced our string literals to be bytes using the `b''` string syntax. In other examples where user input requires us to start with Unicode strings, we will encode those to bytes ensuring that it is safe to do so.

EXERCISE 2.1. WELCOME TO MD5

Compute more digests. Try computing the MD5 sum of the following inputs:

- `b'alice'` (again)
- `b'bob'` (again)
- `b'balice'`
- `b'cob'`
- `b'a'`
- `b'aa'`
- `b'aaaaaaaaaa'` (ten copies of the letter "a")
- `b'a'*100000` (100,000 copies of the letter "a")

What did you learn about MD5 sums from Exercise 2.1? We will talk about these further in the chapter, but let's jump back to our intrepid Antarcticans.

"Okay, Alice and Bob," the instructor says. "A couple of things. These digest objects don't require the entire input all at once. It can be inserted a chunk at a time using the `update` method," as shown in Listing 2-4.

Listing 2-4. Hash Update

```
>>> md5hasher = hashlib.md5()
>>> md5hasher.update(b'a')
>>> md5hasher.update(b'l')
>>> md5hasher.update(b'i')
>>> md5hasher.update(b'c')
>>> md5hasher.update(b'e')
```

The instructor asks Alice and Bob, "What do you think the output of the `md5hasher.hexdigest()` instruction will be?" Try it out and see if you got it right!

"Great," the instructor says when they've finished. "Your introductory training is almost over. Just one more exercise!"

EXERCISE 2.2. GOOGLE KNOWS!

Do a quick Google search using the following hashes (enter the hashes literally into the Google search bar):

1. 5f4dcc3b5aa765d61d8327deb882cf99
2. d41d8cd98f00b204e9800998ecf8427e
3. 6384e2b2184bcbf58eccf10ca7a6563c

Making a Hash of Education

Within the realm of computer security, the terms "hashing" or "hash function" always refer to *cryptographic* hash functions, unless otherwise stated. There are some very useful non-cryptographic hash functions as well. In fact, you were taught a very simple one in grade school: computing whether a number is even or odd. Let's see how this simple, familiar function illustrates principles that apply to all hash functions.

Hash functions are fundamentally trying to map an enormous (even infinite) number of things onto a (relatively) small set of things. When using MD5, for example, no matter how big our document is, we end up with a 16-byte number. In discrete algebra terms, this means that the *domain* of a hash function is much larger than its *range*. Given a very, very large number of documents, chances are that many of them will produce the same hash.

Hash functions are therefore *lossy*. We lose information going from our source document to a digest or **hash**. This is actually critical to their function, because without a loss of information, there would be a way to go backward from the hash to the document. We really don't want that, and we'll see why soon.

Thus, computing whether a number is even or odd fits this description quite well. No matter how large or interesting the (integer) number, we can squash it down into a single bit of space: 1 for odd, 0 for even. That's a hash! Given any number of any size, we can efficiently produce its "oddness" value, but given its oddness, we would be hard-pressed to figure out which number produced it. We can create a very, very large number of *possible* inputs, but we can't know *which one specifically* was used to make that answer.

An "even or odd" bit is sometimes called a "parity" bit and has often been used as a rudimentary error detection code.

The even/odd hash example illustrates this principle of "squashing" an input down to a fixed size value. This value is *consistent*, meaning you won't get a different value out if you put the same number into it twice. It *compresses* large inputs into a fixed-size space (just one bit!), and it is *lossy*: you can't tell me which number was used as input by examining only the output.

All hash functions, including non-cryptographic hash functions, have the fundamental qualities of **consistency**, **compression**, and **lossiness** and have all kinds of important applications in computer science. These qualities alone, however, are not enough for a hash function to be *cryptographic* or *secure*: for those, a hash function needs a few more properties [11, Chap. 9]:

- Preimage resistance
- Second-preimage resistance
- Collision resistance

We'll talk about each of these important qualities in turn.

Preimage Resistance

Informally, a **preimage** is the set of inputs for a hash function *that produce a specific output*. If we were to apply that to our parity example from earlier, the preimage for an odd parity bit is the (infinite!) set of all odd integers. Similarly, the preimage for an even parity bit is the set of all even integers.

What does this mean for a cryptographic hash? Earlier, we computed that the MD5 hash value `6384e2b2184bcbf58eccf10ca7a6563c` could be generated by the input `b'alice'`. Thus, the preimage of

$$\mathrm{MD5}(x) = \texttt{6384e2b2184bcbf58eccf10ca7a6563c}$$

contains the element `x == b'alice'`.

This is important, so let's state it in more precise terms (using integers in our domain and range—remember, a document is ordered bits and is therefore just a big integer):

Preimage: A *preimage* for a hash function H and a hash value k is the *set of values* of x for which $H(x) = k$.

For cryptographic hash functions, this concept of the preimage is important. If I give you a digest value, there might (should) be infinitely many input numbers that could be used to produce it. Those numbers are the preimage for that digest. Remember, every document is just a large integer number from the computer's point of view. It's all just bytes, and we're just performing a mathematical operation on them. The preimage is therefore just an infinite set of integer numbers.[1]

The idea of **preimage resistance** is basically this: if you hand me a digest and I don't already know how you got it, *I can't even find one element in the preimage for it* without doing a ridiculous amount of work. Ideally I would have to do an *impossible* amount of work.

It's already hard to (in general) find *the entire* preimage; it's way too big. What we're really interested in is making it tough to find *any element* in the preimage unless you already happen to know one. That's where lossiness comes in: the digest should give us *no information whatsoever* about the document that produced it. Without any

[1] If thinking about the domain helps, a good quality for every preimage of a hash function would be that all of its elements are very spread out with unpredictable spacing. That way you are very unlikely to accidentally choose one by guessing (they're really spread out), and given a hit, you're just as unlikely to find any others (unpredictable spacing). That last part is something we'll dive into in just a moment.

information to guide us, the best we can do is random guessing or trying everything until we accidentally land on one that produces the right digest. *That* is preimage resistance.

The process of attempting to find an element in the preimage for a given output is called **inverting** the hash: trying to run it backward to get an input for a given output. Preimage resistance means that finding any inverse is hard.

This is why the even/odd function is a potentially *useful* hash function, but not a *secure* hash function. If I give you an even/odd value, you can readily come up with *something* that matches. I say "even," you say "2," for example. That's not very preimage-resistant, because you just told me an input that produces the given output, and you didn't have to think very hard to do it. In fact, you can describe the entire preimage without much trouble: "all even integers." For cryptographic hash functions, if I tell you `MD5`(x) `= ca8a0fb205782051bd49f02eae17c9ee`, you (ideally) can't tell me what x is unless you can find someone who already knows and is willing to tell you. MD5 is hard to invert.

Now, you could just try random (or ordered) documents to see if any of them produce `ca8a0fb205782051bd49f02eae17c9ee`, and you might get (very!) lucky. That approach is one kind of a **brute-force attack** because you just have to pick your way through every single straw in the haystack to find the needle you are looking for. All you can do is commit to looking at an awful lot of straw, relying on raw stamina to carry you through.

Because consistency is a property of hashes, if you already happen to have an input that maps to a given output, or you can find it by searching Google, for example, then that particular output is trivially inverted. The ASCII text "alice" always maps to "6384e2b2184bcbf58eccf10ca7a6563c" when run through MD5, no matter what, so if you happen to know that those two things go together, you can easily find "alice" from the digest. For that particular output, MD5 is trivially inverted. That doesn't mean MD5 is not preimage-resistant, though: to break that you would need to find an easy way to *always* find an input given an output *without knowing one beforehand*.

That leaves us with brute force again. How long would it take you to "guess" an element of the preimage for an MD5 hash using a brute-force technique (either random guessing or sequential searching)? To answer this, we first need to look at how many possible hash values there are. We know that MD5 always produces a 16-byte digest, and we can use that to figure out how hard it should ideally be to invert MD5. To do so, we'll need some understanding of binary (base-2), decimal (base-10), and hexadecimal (base-16) positive integers (plus 0, but we usually just say "non-negative").

If you already understand those pretty well, feel free to skip to the next section.

Byte into Some Non-negative Integers

Most computers use binary to represent everything. The binary number system is represented in base 2. One nice way to be introduced to it is through counting. Here we have the familiar base-10 (decimal) number on the left and the corresponding binary number on the right:

```
0     0
1     1
2    10
3    11
4   100
5   101
6   110
7   111
8  1000
9  1001
```

How does counting work in this system? We start with 0, which is nice and familiar. Adding 1 gives us 1, which is expected. So far, so good. But then, since we're in base 2, we run out of digits when we try to do it again! Just like there is no single digit representing the number "10" in our decimal system, there is no single digit representing "2" in binary!

What do we do when we run out of digits in base 10? We use place values. The very number "10" shows this: there is "1 ten" and "0 ones" in that number. It's the number that comes after "9."

Binary is similar. When we move up one number from "1," we run out of digits, so we put a "1" in the "twos column" and start over at "0" in the "ones column."

What might seem remarkable is the fact that you can represent every non-negative integer this way, just like you can with decimal. The base value ("base-2," "base-10," "base-16," etc.) tells you how many digits you have to work with and therefore what the place values mean. Here are a few place values in different number systems. Note that people get a little bit careless with these things, using decimal to *talk about them*, but really the number system is arbitrary. When it comes to that, there are ten kinds of people in the world: those who understand binary and those who don't.[2]

[2] An old joke. You're welcome. And we're sorry.

That's a big problem when teaching number systems: what does "10" mean without knowing what base we're operating in? The default is to assume it means "ten" unless the base is explicitly stated, or is *really obvious*, like with hexadecimal, where we see "a"–"f" along with the more common decimal digits. We'll do that here too: if you don't see a base or you can't easily tell what it is, you are looking at decimal.

	Place 3	**Place 2**	**Place 1**	**Place 0**
Binary	8	4	2	1
Decimal	1000	100	10	1
Hexadecimal	4096	256	16	1

Or, put another way:

	Place 3	**Place 2**	**Place 1**	**Place 0**
Binary	2^3	2^2	2^1	2^0
Decimal	10^3	10^2	10^1	10^0
Hexadecimal	16^3	16^2	16^1	16^0

All of these number systems work in the same way: place value is determined by adding one to an exponent on the base.

In decimal, therefore, the number 237 really means $2 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0 = 200 + 30 + 7$.

The same number in hex (we'll use x_h to mean "*x* in hexadecimal") is ed_h, which means $e_h \cdot 10_h^1 + d_h \cdot 10_h^0$. But what does *that* mean? Well, $e_h = 14_d$ in decimal, and $d_h = 13_d$. Since 10_h has a 1 in the sixteens column, we get (in decimal) $14 \cdot 16 + 13 = 237$.

Why do we care about hexadecimal in the first place, other than its relative compactness? Hexadecimal (or "hex") is useful because its place values are multiples of 2 (they're multiples of 2^4, to be exact), so it lines up nicely with binary. Consider the following table with hex on the left and binary on the right:

0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

We ran out of digits in hex at exactly the same time that we needed to go from four columns to five in binary! That's really helpful, because it means we can trivially convert back and forth between a computer's native and sprawling binary numbers to the much more human-friendly and compact hex numbers. People even get good enough at this that they can just translate them on sight. Here's an example with binary on top and hex underneath:

```
101  1100  1010  0011  0111
5     c     a     3     7
```

No matter how big a binary number gets, you can take every four bits and write them as a single hexadecimal digit.

The point of walking through this review of binary is to emphasize once again that *every* sequence of bits in a computer is a *number*. What if those bits are a document? That's a number. What about if they represent an image? That's just a big number.

The “meaning” of those bits is not in the computer, it’s *in our minds*.

We may display the bits a certain way, but *we humans* choose to do that based on what we think they mean. The computer has *no idea* what they really mean. They’re just numbers. Can we store the *meaning itself* somehow? Well, sure, but that would force us to encode the meaning *as a number*, because numbers are all that computers understand. Even their instructions are just numbers.

Philosophize much, do we? It’s actually a pretty important thing to understand if you really want to know how computers work, and we definitely need people in the world who do. Data and code are just big numbers, and computers basically just fetch, store, and do arithmetic on them.

How Hard a Hash!

With that little side trip, we can now answer what we wanted to answer in the first place: how hard would it be, in general, to invert MD5 using brute force? We can take a stab at this by looking at the size of its output. MD5 outputs a value in 16 bytes, which is $16 \cdot 8 = 128$ bits. With n bits we can express 2^n individual values, so MD5 can output a lot of different digests. This many, in fact (in decimal):[3]

340,282,366,920,938,463,463,374,607,431,768,211,456.

Even if you checked 1 million values *per second* (and were guaranteed that nothing you checked produced an output you had seen before), it would still take you about 10^{26} years (100 million billion billion!) to find a suitable input by brute force. By comparison, our sun is only expected to continue sustaining earth life for at most another 5 billion years; your computer would need to keep running for many, many times that long.

If you have a cryptographic algorithm whose only means of being broken is brute force, you have a good algorithm. The trouble is, you don’t necessarily *know that it’s good*. But this gives us an upper bound on how long it would take to find an input that produces a particular hash in MD5. At least it won’t take longer!

[3] In hex, this number is much more tightly related to binary and looks a little more sensible: 10000 0000000000000000000000000000.

Second-Preimage and Collision Resistance

Once preimage resistance is understood, the other two properties are relatively easy to grasp. We wandered a bit into brute force and binary near the end of that last section, so let's quickly review:

> **Preimage resistance** means it is really hard to find a document that produces a particular digest, unless you already know one.

Second-Preimage Resistance

Second-preimage resistance means that if you already have *one* document that produces a particular digest, it's still really hard to find a *different* document that produces that same digest.

In other words, just because you know that

$$\texttt{MD5}(\text{alice}) = \texttt{384e2b2184bcbf58eccf10ca7a6563c},$$

it doesn't mean you can find another value to put into $\texttt{MD5}(\cdot)$ that will give you the same value. You would have to resort to brute force again.

To tie it back to its name, if you have one member of the preimage already, it is not any easier to find a second member of the preimage: there is no exploitable pattern in the preimage.

Collision Resistance

Collision resistance is a bit more subtle than either of the preimage characteristics we just mentioned. Collision resistance means that it's hard to find *any two inputs that produce the same output*: not a *specific* output, just *the same* output.

A classic way of describing this is by using birthdays.[4] Suppose you are in a room full of people and you want to find two of them whose birthday is February 3. How likely is that? Not necessarily very likely, if you really picked it at random.

But now let's say you want to do something else. You want to know whether any two people have *the same birthday*. You don't care what day of the year it falls on, you just want to know whether anyone's birthday overlaps with anyone else's. How likely is

[4] The "birthday problem" is a classic problem in probability theory, of uncertain origin.

that? It turns out that, in general, it is far, *far* more likely. After all, we just removed the constraint of a particular day, and now all we want is a collision on *any day*.

That's the basic idea behind collision resistance. When a hash algorithm is resistant to collision, it is resistant to being able to purposefully create or pick any two inputs that produce the same digest, without deciding what that digest should be beforehand.

MD5 appears to be fairly collision-resistant. One property that helps with this is the fact that small changes in input can produce very large changes in output. Consider Exercise 2.1, where you produced hashes for very similar values like "a" and "aa," or "bob" and "cob." The digests resulting from performing MD5 on these values were not just different, they were *wildly* different:

```
bob: 9f9d51bc70ef21ca5c14f307980a29d8
cob: 386685f06beecb9f35db2e22da429ec9
```

There is no discernible pattern that would tie one to the other. This is due to a property shared by many hashes and cryptographic ciphers called the **avalanche property**: a change to the input, no matter how small, creates a large and unpredictable change in the output. Ideally, 50% of the output bits should be altered for small input changes [11, Chap. 7]. Did we achieve that with "bob" and "cob"? Let's take a look at the digests in binary using some Python to aid our exploration (note that our bit string is quite long, so it is broken over two lines in Listing 2-5).

Listing 2-5. Avalanche

```
>>> hexstring = hashlib.md5(b'bob').hexdigest()
>>> hexstring
'9f9d51bc70ef21ca5c14f307980a29d8'
>>> binstring = bin(int(hexstring, 16))
>>> print("{}\n{}".format(binstring[2:66], binstring[66:]))
1001111110011101010100011011110001110000111011110010000111001010
0101110000010100111100110000011110011000000010100010100111011000
```

The following illustration visualizes the changes in bits when given inputs `b'bob'` and `b'cob'`,

```
MD5(bob):
   9   f   9   d   5   1   b   c   7   0   e   f   2   1   c   a
1001111110011101010100011011110001110000111011110010000111001010
   5   c   1   4   f   3   0   7   9   8   0   a   2   9   d   8
0101110000010100111100110000011110011000000010100010100111011000

MD5(cob):
   3   8   6   6   8   5   f   0   6   b   e   e   c   b   9   f
0011100001100110100001011111000001101011111011101100101110011111
   3   5   d   b   2   e   2   2   d   a   4   2   9   e   c   9
0011010111011011001011100010001011011010010000101001111011001001

Changed Bits:

X_X__XXXXXXXX_XXXX_X_X___X__XX_____XX_XX_______XXXX_X_X__X_X_X_X
_XX_X__XXX__XXXXXX_XXX_X__X__X_X_X____X__X__X___X_XX_XXX___X___X
```

In this example, the difference between the hash of "bob" and "cob" impacted 64 bits out of 128. Not bad! Avalanche is an important property, and we will see it again with ciphers in Chapter 3.

EXERCISE 2.3. OBSERVING AVALANCHE

Compare the bit changes between a wide range of input values.

Avalanche helps collision resistance, because it is hard to produce a document, then come up with predictable changes that will still cause it to produce the same digest. If a small change in a document produces an unpredictable and large change in the digest, then creating collisions on purpose is likely to be a difficult problem, pushing us toward brute force again to solve it.

Remember the birthday analogy from earlier? Finding collisions is not as hard as finding a value in the preimage. Preimage resistance for an n-bit digest means an attacker would expect to compromise your hash after trying 2^n, where it would only take $2^{(n/2)}$ attempts to find a collision. That's not half as many tries, that's a number of tries with half as many *zeros* in it. The difference is astounding. Concretely for MD5, finding a document for a given digest should take about 2^{128} attempts, where finding two documents that collide should only take 2^{64} attempts.

As it happens, MD5's collision resistance is actually nowhere near that good. It has been "broken," meaning that techniques have been discovered for finding collisions in far fewer than the expected 2^{64} attempts. In short, the problem can be solved by something *other than brute force* and in less than an hour [17]. Keep that in mind, and we'll come back to it later on.

Digestible Hash

By this point, you should know enough to create a Python program that computes the MD5 digest[5] of a file. This is a common use for hashes and a good exercise to work through. Remember, you must use Python bytes, not Python Unicode strings, as inputs. If you try to open a Python file with the default mode, it may open it as a text file and read the data as strings, doing implicit decoding. You should, instead, open the file in "rb" mode so that all reads produce raw bytes. For a text file, you might be tempted to read the data as a string and then use the string's `encode` method to convert to bytes, but *depending on configuration, this encoding may not be what you expect and lead to nasty surprises*.

EXERCISE 2.4. MD5 OF A FILE

Write a python program that computes the MD5 sum of the data in a file. You don't need to worry about any of the file's metadata, such as last modified time or even the file's name, only its contents.

You should check your work for Exercise 2.4. If you're using an Ubuntu Linux system, the `md5sum` utility is already installed. Run this utility from the command line with a file as input and see if it produces the same hex digest as your utility.

Speaking of Ubuntu, this is a perfect example of using hashes for file integrity. Visit the web site for Ubuntu releases. At the time of this writing, the web site is `https://releases.ubuntu.com`. If you take a look at the "Bionic Beaver" distribution, for example,

[5] Sometimes called the "MD5 sum," where "sum" is short for "checksum," a name with some interesting (and long) history of its own, from error detection in digital transmissions.

you will find that there are number of files available for download. In particular, there are two ISOs, but they are available directly or through other downloading technologies such as BitTorrent.

There's also a file called MD5SUMS. Take a look. For this distribution, the contents of this file should be as follows:

```
f430da8fa59d2f5f4262518e3c177246 *ubuntu-18.04.1-desktop-amd64.iso
9b15b331455c0f7cb5dac53bbe050f61 *ubuntu-18.04.1-live-server-amd64.iso
```

Once downloaded, you can verify that the data is uncorrupted by running an MD5 sum on the ISO.

How is the MD5 hash value helpful? It will not protect you from somebody that has compromised the Ubuntu web site. If they upload a fake Ubuntu to the web server, they can upload a fake MD5 sum as well.

The MD5 sum *does*, however, make it easier for you to obtain the Ubuntu ISO from other sources and know that it's authentic. For example, suppose that you were about to download the ISO file directly from the Ubuntu web site when a coworker stops by and says that you can use the already-downloaded one they have on a USB drive. You can download the (relatively small) file of MD5 sums from Ubuntu's official site and check them against the (much larger) files on the drive before trusting them.

Looking in the Ubuntu directory, you will also see a file called SHA1SUMS and SHA256SUMS. What are these?

So far, we've only talked about MD5 as a way of teaching some of the principles of hashing. MD5 was a standard approach to cryptographic hashing for a long time too, but it has been broken: people have discovered methods much faster than brute force for inducing collisions, so it is being phased out in favor of other hash functions.

Interestingly enough, being "broken" often means "someone can solve a problem in an order of magnitude less time than brute force." For example, that might mean that preimage values can be found in 2^{127} tries on average instead of 2^{128}. That's still hard, just not as hard as it should be. When looking at articles indicating that something has been broken, it's important to find out *exactly what that means*. Does it mean one of the fundamental properties no longer holds? Does it mean it holds, but isn't as hard to get around? What if it's more than one property? These things matter.

With MD5, researchers have found a way to "break" preimage resistance [12]. They showed that they could find a preimage for an MD5 hash faster than 2^{128} tries. How much faster? Well, their algorithm takes slightly longer than 2^{123} tries or, in decimal, 10,633, 823,966,279,326,983,230,456,482,242,756,608 tries. This attack is considered theoretical because it is still not useful in practice: 2^{123} is still *huge*.

On the other hand, MD5 has been shown to be very, very broken where *collision resistance* is concerned. It is reasonably easy to create two inputs that produce the same MD5 output. This has been shown to enable a practical attack for getting a false certificate for use in TLS, which is used for all kinds of secure Internet communication. We won't get into the details here, because we haven't talked about certificates yet, but we'll revisit this when we get to TLS at the end of the book.

On the other hand, collision resistance is not the same as second-preimage resistance. Remember that second-preimage resistance prevents you for finding a second member of the preimage for an output when you already have the first. Even though MD5's collision resistance is broken, its preimage resistance is not. Returning to our Ubuntu distribution example, if you're getting your distribution from an intermediary, they are not able to create an alternate distribution with the same MD5 digest.

The Ubuntu organization, however, could exploit the MD5 collision resistance weakness to create two separate distributions that have the same MD5 sum. Perhaps, in conjunction with a government, they could sell one distribution with all kinds of tracking software to another government hostile to the first. The MD5 sum could not be used to ensure that the same ISO was being distributed to all parties.

Also, once a cryptographic algorithm is broken in one way, there is increased suspicion that it is broken in other ways as well. So even though nobody has demonstrated a practical attack on MD5's preimage or second-preimage resistance, many cryptographers worry that such vulnerabilities exist.

To reiterate the warning at the beginning of the chapter, **DO NOT USE MD5**. It has been deprecated for over 10 years (decimal), and some of its security flaws have been known for two decades.

The SHA-1 hash is another algorithm that was widely viewed as the replacement for MD5. SHA-1's collision resistance has *also* recently been broken, however, as researchers have shown that it is relatively easy to create two inputs that hash to the same output [13]. So, as with MD5, **DO NOT USE SHA-1**, either.

At the time of this writing, best practice is to use SHA-256. Fortunately, this means very little to you if you are using `hashlib`: just change the hasher, as shown in Listing 2-6.

Listing 2-6. Change to SHA-256

```
>>> import hashlib
>>> hashlib.md5(b'alice').hexdigest()
'6384e2b2184bcbf58eccf10ca7a6563c'
>>> hashlib.sha1(b'alice').hexdigest()
'522b276a356bdf39013dfabea2cd43e141ecc9e8'
>>> hashlib.sha256(b'alice').hexdigest()
'2bd806c97f0e00af1a1fc3328fa763a9269723c8db8fac4f93af71db186d6e90'
```

You should notice that these different hash algorithms have different lengths. MD5, of course, outputs 16 bytes (128 bits). If it isn't obvious, SHA-1's output is 20 bytes (160 bits). And, more simply, SHA-256's output is 32 bytes (256 bits).

If you thought it would take a long time to invert MD5 (find a preimage for a given output), take a look at SHA-1. Because the output is 160 bits, it will take 2^{160} tries, or

1,461,501,637,330,902,918,203,684,832,716,283,019,655,932,542,976

tries to find a preimage. And SHA-256 requires 2^{256} tries, or

115,792,089,237,316,195,423,570,985,008,687,907,853,269,984,665,640,564,039,457, 584,007,913,129,639,936.

Good luck with that!

Pass Hashwords...Er...Hash Passwords

Another common use for hash functions is password storage. When you create an account on a web site, for example, they almost never store your password. Typically, they store a *hash* of the password. That way, if the web site is ever compromised and the password file is stolen, the attacker cannot recover anyone's password.

What does this mean? When you send your password (over a secure channel, via HTTPS), the server doesn't need to store it to check it. When you registered, your password was hashed, and the hash was stored. We'll call that H(password). When you go to log in later, you send a password that we'll call the “proposal”: you're proposing that this is your true password, and the server needs to verify that.

So, you attempt to log in by sending your proposed password over a secure connection, and the server now has two things for you: it can look up H(password) from your username, and it has the proposal that you just submitted. All it has to do is check that H(proposal) = H(password) and let you through if they are the same.

What if you don’t trust the service to not actually store your password? That can be a valid worry, particularly since we’ve seen so many sites with stolen passwords in recent years. Why not use the power of JavaScript to hash your password in the browser, then send *that* to the server? Then the server never even sees your password in memory, let alone in its database!

There are a few big problems with this:

- The code that hashes the password in your browser *came from that server in the first place*, so you still have to trust the service.
- If you don’t have a secure channel for your password, then someone can read it in transit. If you do have a secure channel, then you might as well just send the password. You have to trust the service already.
- If you send a hash successfully, that just became your password. Yes, you can generate it from some other easy-to-remember thing, but now you have to protect that hash value as well. The server has to hash your hash anyway, so that an attacker that makes off with the database can’t just log in using what’s stored there.

In short, if you were to use a hash as your password, the right way is to generate that hash from your password and the site name of interest *using a tool separate from your browser*, and then use the result as your password. This is essentially the same thing as just creating a brand new password and remembering it in a secure place, like a password manager.

Just do that instead. Then the server will never see a password that you use somewhere else, because you created a brand new random one for it.

Far better than trying to solve security with a hash is to use multiple forms of authentication that are proven to make it harder to steal your identity online, typically involving a hardware token attached to your computer.

Most common forms of two-factor authentication don't help and actually make things worse. Secret questions are one of those. It's usually easy to get answers to them, and if it isn't, they're just one more hard thing to remember, unless written down. Plus, now you have *several* things that can be used as a password for the site, which means attackers have more opportunities to get in by guesswork. SMS has been shown to be critically weak and easy to hijack as well, so having a code sent via SMS to your phone is no good.

Properly deployed challenge-response hardware tokens don't have any of these problems. They are something you possess instead of just another thing you know, and they can't be guessed or spoofed by people listening in on the connection or pretending to be some other site's login form. They can't be given over the phone accidentally, and they can't be forged.

Ultimately you need two or more factors for authentication *anyway* for true security. "Fixing passwords" is the wrong place to look for a complete solution.

If server-side hashing is correctly used and if an attacker steals the password file, they will see something like this. From looking at it, can you tell smithj's password?

```
...
smithj,5f4dcc3b5aa765d61d8327deb882cf99
...
```

Look closely. Have you seen that hash value before?

The eagle-eyed reader will remember that hash value from the beginning of the chapter exercises. You were asked to look for that value online. What did you find?

That hash value is the MD5 hash of "password," and yes, that password is still used far too often. But the deeper problem here is that hash values are deterministic: the same input always hashes to the same output. If an attacker has seen the MD5 sum of "password" once, he will be able to look for that same digest in every password file stolen. How can we fix this?

First, let's not assume that we can make people stop using dumb passwords.

Let's assume that they're going to and we need to fix it anyway. We'll start with the digest itself.

Recall that MD5 is *not* broken (in practice) with respect to preimage resistance or second-preimage resistance. So there is no practical attack, at present, for inverting this hash value to the password. Nevertheless, **MD5 is broken and should not be used!** So let's take a look at a new password file.

```
...
smithj,5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
...
```

Any idea what smithj's password is now? Yes, it is still "password" but now it's hashed under SHA-1. That's better, right? Oh yeah, **SHA-1 is broken and should not be used!** Let's try one more time!

```
...
smithj,5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
...
```

There! Finally! We're using a hash algorithm without known vulnerabilities. That's better, but the problem with deterministic hashing is still a problem. If the attacker knows that this hash maps to the SHA-256 of "password," smithj is still compromised.

This is where the idea of a "salt" comes into play. A salt is a publicly known value that is mixed in with the user's password before hashing. By mixing in a salt value, the user's password will not be immediately discernible as it is now.

This salt has to be chosen correctly. It needs to be unique, and it needs to be sufficiently long. One way of doing this is to use `os.urandom` and `base64.b64encode` to generate a strong, random[6] salt:

```
>>> import hashlib
>>> hashlib.md5(b'alice').hexdigest()
'6384e2b2184bcbf58eccf10ca7a6563c'
>>> hashlib.sha1(b'alice').hexdigest()
'522b276a356bdf39013dfabea2cd43e141ecc9e8'
>>> hashlib.sha256(b'alice').hexdigest()
'2bd806c97f0e00af1a1fc3328fa763a9269723c8db8fac4f93af71db186d6e90'
```

[6] The requirement is *uniqueness*, not *randomness*, but randomness provides us with a simple approach that works well for our examples.

Obviously, your salt output will be different from that shown in the code listing and will be different every time you call it.

Once you have the salt, you store it, then mix the password and the salt with concatenation. For example, prepend the password with the salt before hashing. Now, if the attacker gets your password file, it will be impossible to "recognize" the password from any kind of pre-computed table.

They can still try hashing the salt plus "password" to see if anything matches, though. Guesswork is always a strategy, and it's a particularly good one for most people's password choices.

It should be easy to see that the same salt has to be used every time for checking a user's password. But should the same salt be used for multiple users? Could you generate the salt once for an entire web site and just reuse it?

The answer is a very strong "No!" Can you think of why? What will be the impact if two users are using the same salt? At the very least, it means that it is instantly easy to recognize if two users are sharing the *same* password. Thus, best practice is to store the user name and salt along with the password hash.

If our friend smithj has the terribly chosen password, "password", at least it will be stored correctly on our system:

```
...
smithj,cei6LtJVQYSM+n6CtyOO2w==,
    bd51dac1e2fca8456069f38fcce933f1ff30a656320877b596a14a0e05db9567
...
```

We have now walked through the basics of password storage, but there are better algorithms. They are built on the same principle but do additional steps to make it even harder for an attacker to invert the password. One highly recommended algorithms for password storage is called **scrypt** by Colin Percival and described in RFC 7914 [16]. Other popular ones are the newer **bcrypt**[7] (https://pypi.org/project/bcrypt/) and the algorithm considered by some to be its successor: **Argon2** (https://pypi.org/project/argon2/).

Fortunately, using `scrypt` is easy using the `cryptography` module you set up in Chapter 1. Listing 2-7 is an example derived from the `cryptography` module's online documentation. The listing derives the key (hash) to be stored on the file system.

[7] The bcrypt algorithm is quite good and has only one "difficulty" parameter, making it easier to use correctly than approaches with many parameters.

Listing 2-7. Scrypt Generate

```
import os
from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
from cryptography.hazmat.backends import default_backend

salt = os.urandom(16)

kdf = Scrypt(salt=salt, length=32,
                n=2**14, r=8, p=1,
                backend=default_backend())

key = kdf.derive (b"my great password")
```

Both the key and the salt must be stored to disk. The `scrypt` parameters must be fixed or must also be stored. We will walk through these parameters in a moment, but first, verification is depicted in Listing 2-8 (it is presumed that salt and key are restored from disk).

Listing 2-8. Scrypt Verify

```
kdf = Scrypt(salt =salt, length =32,
             n=2**14, r=8, p=1,
             backend=default_backend())
kdf.verify(b"my great password", key)
print("Success! (Exception if mismatch)")
```

Pick Perfect Parameters

With regard to the `scrypt` parameters, let's talk about `backend` first. The `cryptography` module is primarily a wrapper around a lower-level engine. For example, the module can make use of OpenSSL as such an engine. This makes the system faster (because computations aren't being done in Python) and more secure (because it relies on a robust, well-tested library). Throughout this book, we will always rely on `default_backend()`.

The other parameters are specific to `scrypt`. The `length` parameter is how long the key will be once the process is finished. In these examples, the password is processed into an output of 32 bytes. The parameters `r`, `n`, and `p` are tuning parameters that impact how long it will take to compute and how much memory is required. To better protect your password, you want the process to take longer and require more memory, preventing attackers from compromising large chunks of a database at once (every compromise should take a long time).

Fortunately for you, recommended parameters are available. The `r` parameter should be 8, and the `p` parameter should be 1. The `n` parameter can vary based on whether you are doing something like a web site that needs to give a relatively prompt response or something more securely stored that does not need quick responsiveness. Either way, it must be a power of 2. For the interactive logins, 2^{14} is recommended. For the more sensitive files, a number as high as 2^{20} is better.

This is actually an excellent segue into a more general discussion about parameters. A lot of the security in cryptography depends on how parameters are set. Unless you are a cryptography expert, know the exact details of the algorithm, and understand why they are what they are, it may be difficult to choose them properly. It is important that you familiarize yourself with what the parameters mean, at least at a high level, and how they should be used in different contexts. Refer to trusted sources, such as `https://cryptodoneright.org`, for advice and recommendations. Keep an eye on these sources too. What is presumed to be secure can change as new attacks and computational resources are unveiled.

Cracking Weak Passwords

Let's take a look at how attackers try to crack passwords. Unfortunately for smithj, choosing such a bad password means that he will most likely be compromised if the password file gets stolen, since attackers will try common words (including words in *other* stolen databases) against all the hashes anyway. But even less sophisticated methods would probably figure out the password as well.

In this section, we are going to practice cracking weak passwords using the least sophisticated method of all: brute force. This exercise is meant to reinforce why good passwords are so important.

The scenario is this: an attacker has a password file with usernames, salts, and password hashes. What can they do? Well, they could just try all lowercase letter combinations up to a certain length, starting, for example, with "a," "b," "c," and so on.

To make these exercises a little bit easier to start, Listing 2-9 shows some simple code for generating all possible combinations of an alphabet set up to a maximum length.

Listing 2-9. Alphabet Permutations

```
def generate(alphabet, max_len):
    if max_len <= 0: return
    for c in alphabet:
        yield c
    for c in alphabet:
        for next in generate(alphabet, max_len-1):
            yield c + next
```

Calling generate('ab', 2) will generate 'a', 'b', 'aa', 'ab', 'ba', 'bb'. Using helpful sets in the built-in string module, such as

- string.ascii_lowercase
- string.ascii_uppercase
- string.ascii_letters

makes the following exercises fairly easy. Recall that hashing algorithms require bytes as inputs, so don't forget to do an encode operation before passing these generated strings to the hashing function, like this:

string.ascii_letters.encode('utf-8').

ASCII letters encode correctly to bytes, so this will not lead to incorrect hashing or unexpected behaviors.

EXERCISE 2.5. THE POWER OF ONE

Write a program that does the following ten times (so, ten full loops with the time computed):

- Randomly select a single, lowercase letter. This is the "preimage seed."
- Use MD5 to compute the hash of this initial letter. This is the "test hash."

- In a loop, iterate through all possible lowercase one-letter inputs.
  - Hash each letter in the same way as before, and compare against the test hash.
  - When you find a match, stop.
- Compute the amount of time it took to find a match.

How long, on average, did it take to find a match for a random preimage seed?

EXERCISE 2.6. THE POWER OF ONE, BUT BIGGER!

Repeat the previous exercise, but use an increasingly bigger input alphabet set. Try the test with both lowercase and uppercase letters. Then try it with lowercase letters, uppercase letters, and numbers. Finally, try all printable characters (`string.printable`).

- How many total symbols are in each input set?
- How much longer does each run take?

EXERCISE 2.7. PASSWORD LENGTH'S EFFECTS ON ATTACK TIME

Repeat the previous exercise, but this time for two-symbol inputs. Then try it with three and four symbols at a time. How much longer does it take to invert the randomly chosen input?

You will notice that increasing the length of the password and increasing the size of the alphabet both increase the time it takes to invert the hash. Let's look at the math.

When using just lowercase letters, how many possible one-symbol inputs are there? Rather trivially, there are 26 lowercase letters in ASCII, so 26 one-symbol inputs. At worst, it will take 26 hash computations to invert a one-letter password. But, if we have both lowercase *and* uppercase letters, this increases the number of hashes needed to 52. Adding digits increases this to 62. There are 100 characters in `string.printable`. That's nearly a fourfold increase of the worst-case number of hashes required to do brute-force inversion.

What about when we increase the size to two input symbols? How many two-symbol passwords are there using just lowercase letters? If you can have 26 characters for the first symbol and 26 characters for the second symbol, then there are $26 * 26 = 676$ total combinations. That's quite a jump!

Now look what happens if you use two symbols drawn from the 52 uppercase and lowercase letters. The math works out to be 52 ∗ 52 = 2704! Doubling the size of the input set quadrupled the complexity for two-symbol inputs! If we throw in digits, the worst-case computation is 3844 hashes, and for all printable ASCII characters, it is around 10,000 hashes.

Do the math for three, four, and five symbols, and you can easily see why longer passwords matter. Hackers with GPU-enabled rigs are able to invert anything smaller than six characters, and most passwords under eight, so *at a minimum*, passwords should be that long. And for the reasons demonstrated here, choosing from *all* printable letters greatly increases the complexity.

EXERCISE 2.8. MORE HASH, MORE TIME

Choosing a complex-to-invert password is the responsibility of the user, but the systems storing the passwords can also slow down attackers by using a more complicated *hashing function*. Repeat any of the preceding exercises that use MD5, but now use SHA-1 and SHA-256 instead. Record how much longer it takes to get through the brute-force operations. Finally, try out brute force using scrypt. You might not get very far!

One final note. Just because a password is *big* doesn't mean it is *secure*. Attackers will also use large dictionaries to look for known words and phrases, even with various common number or symbol substitutions. A password such as "chocolatecake" is pretty long, but still easily broken. Randomly chosen letters or words are still the best bet. The key is that they are "random," meaning you would never find them in any real writing or common transformations on real writing. Typically, choosing passphrases that are composed of common utterances reduces a successful attack to *seconds* instead of *years*.

Proof of Work

Another area where hashing is used extensively is so-called "proof-of-work" schemes in blockchain technologies. To introduce this, we need to do a very quick overview of how blockchains work.

The basic idea of a blockchain is a "distributed ledger." The system is a *ledger* because it records information related to transactions between participants. It can also store additional information, but the primary operations are transactions. It is a *distributed* ledger because its contents are stored across the set of participants and not in any central location.

The problem is that there is no central location to enforce the correctness of the system. How does the ledger not get corrupted (intentionally or otherwise) by the users? Note that we won't go into the ledger in detail here, but we do want to talk about the blocks that a ledger is composed of.

Every transaction must be stored in a block. There's nothing special about a "block"; it's just a collection of data. Each transaction within the block must be digitally signed by the transactor (we will discuss signatures more in Chapter 5, but for now, simply accept that it means nobody can create a transaction for somebody else without their private key). The overall block structure is protected by a hash. Blocks are copied to the entire set of participants; should anyone try to "lie" about the contents of a block, the data wouldn't verify correctly and their information would be rejected.

How does a new block get created, and how does it get the protective hash? For this part of the discussion, we will use the Bitcoin network blockchain to walk through these concepts. The designer (or designers, the source is actually unknown) of Bitcoin, who goes by "Satoshi Nakamoto," wanted to control how quickly new blocks could be created and also wanted the system to incentivize participation. The solution was to award bitcoins to the "miner" that produced a new block while making the production of the new block very difficult.

Basically, at any given time, various parties known as miners are searching for the next block in the blockchain. Any user of blockchain can request a transaction. They broadcast their desired transaction throughout the blockchain network and miners will pick them up. The miners take some set of requested transactions (there is a limited number per block) and create a candidate block. This candidate block has all the right pieces of information. It has the transactions, the metadata, and so forth. But it isn't the next block in the blockchain until the miner can solve a cryptographic puzzle.

That puzzle is to find a special kind of SHA-256 hash value, specifically a value smaller than a certain threshold. As we discussed earlier, finding an input that produces one particular output would take a really, *really* long time, but finding any output less than a certain value takes a lot *less* time. Making that threshold smaller reduces the number of valid hashes, requiring more work to find a suitable value, and that's how Bitcoin adjusts the difficulty to account for faster hardware or larger computational pools as time goes on. Ultimately, it takes about 10 minutes for the entire Bitcoin network to find a suitable hash. If it takes less than that on average over a period of weeks, the maximum allowed hash value is decreased. Figure 2-1 shows two different example blocks, one with a suitable nonce (a random value that miners are trying to find to produce an acceptable hash) and one without, where the maximum allowed hash value is 2^{236}–1 (20 leading zeros required). For Bitcoin, the very *easiest* that a problem is allowed to be is determined by a maximum value of 2^{224}–1, which would take our little program an average of 2^{12} times longer than before. That translates to 11.3 hours, and the difficulty is *much harder than that* today.

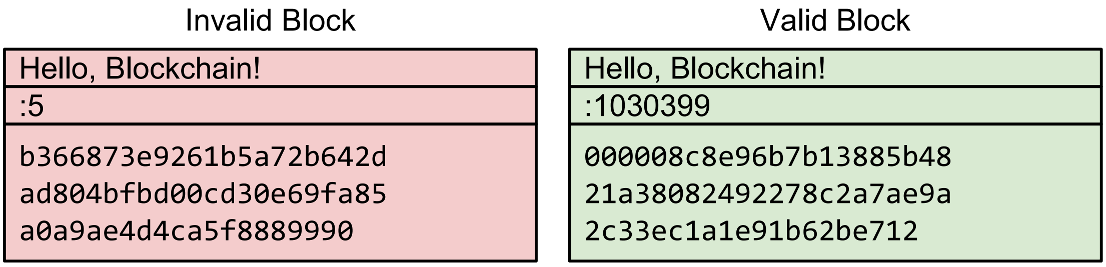

Figure 2-1. *Two block hashes with the same content but different nonce values. A nonce that produces a hash with 20 leading binary zeros (5 leading zeros in hexadecimal) is valid. Requiring 20 leading zeros is the same as requiring that the hash number be less than 2 ∗ ∗ 236.*

Our program definitely won't be beating the network's 10-minute average expectation anytime soon.

Saying that the first few bits must be zero is the same, by the way, as saying the hash value number (the hash is just a number, just like any other string of bits) should be less than some threshold that happens to be a power of 2. Since good hash functions (like SHA-256) produce essentially random hash values, the more structure you impose on the hash, the longer it takes to find one that fits. You can get some intuition for this by thinking about the number of zeros as defining the size of the search space: if you must have a single leading zero, then it's basically a coin flip; it should only take two tries on average to find a suitable hash that starts with a zero bit. If, on the other hand, you need to find a hash with 8 leading zeros, that's a harder problem: 256 different numbers can be represented in 8 bits, so on average it will take 256 attempts to find a suitable value.

That's why this strategy is called "Proof of Work": if you found a suitable hash under the threshold, you had to have done some work (or you broke the hash function, which is deemed to be extremely unlikely, but potentially awesome for you).

This raises an interesting question: how do each of the network participants decide how hard the problem should be? It isn't like there is a central authority telling everyone that the difficulty just went from 11 to 12, for example. That would defeat the whole purpose of the network. The "authority" in the network is tacit agreement between participants to use the same *algorithms* for determining these things. When there is someone on the network doing things differently, their blocks are simply rejected by everyone else and they thus have no incentive to do it wrong. Majority rules.

In the specific case of hashing difficulty, each participant knows the standard algorithm for computing what the number of leading zeros should be and uses that to do mining (or to reject bad proposals from wayward participants looking to compute an easy hash).

You might be asking, however, how a different value of hash is computed when the input data really doesn't change. That's a great question, since hashes are deterministic: they always produce the same output given the same input (they wouldn't be very useful, otherwise!). The answer is that they change one little piece of the input, called the "nonce." It's just a number, and it isn't part of the actual block data: its sole purpose is to enable the proof-of-work concept. When searching for a suitable hash, the participant tries hashing the block with different values for the nonce, typically searching randomly or merely adding 1 to the last value at each attempt. Eventually a suitable hash value is found and the block is sent to all other participants for validation.

Every participant then verifies the block by performing the hash for themselves, checking the leading zeros against their algorithm, and making sure that their answer matches the submitted hash value. If it's good, they accept it and the chain lengthens.

EXERCISE 2.9. PROOF OF WORK

Write a program that feeds a counter into SHA-256, taking the output hash and converting it to an integer (we did this earlier before converting to binary). Have the program repeat until it finds a hash that is less than a target number. The target number should start out pretty big, like 2^{255}. To make this more like blockchain, include some arbitrary bytes to be combined with the counter.

Time to Rehash

We have covered a lot of information about what hashes are and how to use them, including why you should never use MD5 unless you are teaching people that it is broken, and how to use them for more secure password storage and even crypto currency. Hashing is a powerful and important part of cryptography, and we will be seeing it again and again as we move forward.

Now that we have learned a bit about how to digest a document into a safely representative value, it's time to back up a bit and revisit encryption.

CHAPTER 3

Symmetric Encryption: Two Sides, One Key

Symmetric encryption is at the foundation of all modern secure communications. It is what we use to "scramble" messages so that people can only decrypt them if they have access to the same key used to encrypt them. That's what "symmetric" means in this case: one key is used on both ends of the communication channel, to both encrypt and decrypt messages.

Let's Scramble!

Unsurprisingly, the villains[1] of East Antarctica are at it again, causing all kinds of trouble for their neighbors. This time, Alice and Bob are spying on the enemy troops to the west, doing reconnaissance on the size of their snowballs and the accuracy of their throws.

In earlier missions, Alice and Bob used the Caesar cipher from Chapter 1 to protect their messages. As you discovered, this cipher is easy to crack. As a result, the East Antarctica Truth-Spying Agency (EATSA) has equipped them with modern cryptography that uses a key to encode and decode secret messages. This new technology belongs to a class of encryption algorithms called **symmetric ciphers**, because both the encryption and decryption processes use the same shared key. The specific algorithm they are using in this post-et-tu-Brute world is the Advanced Encryption Standard (AES).[2]

[1] ...or heroes, depending on your point of view, Padawan.

[2] The name "Advanced Encryption Standard" is actually more of a title. Many algorithms competed to become the "Advanced Encryption Standard" including many algorithms that are still available today. The original name of the algorithm was *Rijndael*, which is a composite of the last names of the two inventors.

S. J. Nielson and C. K. Monson, *Practical Cryptography in Python*,
https://doi.org/10.1007/978-1-4842-4900-0_3

Alice and Bob don't have a lot of information about the proper care and handling of AES. They have just enough documentation to get encryption and decryption working.

"The docs say we have to create AES keys," Alice says holding one of the manuals. "Apparently, it's fairly easy. We have sample code here."

```
import os
key = os.urandom(16)
```

"Wait… really?" Bob asks. "That's it?"

Alice is right: that's all it takes! An AES key is just random bits: 128 of them (16 bytes' worth) in this example. This will allow us to use AES-128.

With the random key created, how do we then encrypt and decrypt messages? Earlier, we used the Python `cryptography` module to create hashes. It does many other things as well. Let's see how Bob—encouraged by the ease of creating keys—uses it now to encrypt messages with AES.

Bob takes the documentation from Alice and looks at the next section, noting that there are many different *modes* of AES computation. Having to choose between them sounds a bit overwhelming, so Bob picks the one that looks easiest to use.

"Let's use ECB mode, Alice," he says, looking up from the docs.

"ECB mode? What is that?"

"I don't really know, but this is the *Advanced* Encryption Standard. It should all work fine, right?"

Warning: ECB: Not for You

We're going to find out later that ECB mode is *terrible* and should *never be used*. But we'll just follow along for now.

Listing 3-1 has the code they used to create an "encryptor" and "decryptor."

Listing 3-1. AES ECB Code

```
# NEVER USE: ECB is not secure!
from cryptography.hazmat.primitives.ciphers import Cipher,
algorithms, modes
from cryptography.hazmat.backends import default_backend
import os
```

```

key = os.urandom(16)
aesCipher = Cipher(algorithms.AES(key),
                   modes.ECB(),
                   backend=default_backend())
aesEncryptor = aesCipher.encryptor()
aesDecryptor = aesCipher.decryptor()
```

"That's not so bad," Alice says. "What happens now?"

"Apparently, both the encryptor and decryptor have an `update` method. That's pretty much it. The encryptor's update returns the ciphertext."

EXERCISE 3.1. A SECRET MESSAGE

Without looking at additional documentation, try to figure out how the `aesEncryptor.update()` and `aesDecryptor.update()` methods work. Hint: You are going to get some unexpected behavior, so try lots of inputs. Consider starting with `b"a secret message"` and then decrypting the result.

Alice and Bob start trying to figure out the `update` method. Perhaps inspired by the previous chapter on hashing, where they hashed their names, they try encrypting their names in an interactive Python shell. Alice goes first.

The AES example code here uses the key `b"\x81\xff9\xa4\x1b\xbc\xe4\x84\xec9\x0b\x9a\xdbu\xc1\x83"` in case you want to get identical results.

```
>>> aesEncryptor.update(b'alice')
b''
```

"I didn't get any ciphertext," Alice grumbles. "What did I do wrong?"

"I don't know. Let me try," Bob responds.

```
>>> aesEncryptor.update(b'bob')
b''
```

“Me too,” he says, confused. Out of frustration, he tries it several more times.

```
>>> aesEncryptor.update(b'bob')
b''
>>> aesEncryptor.update(b'bob')
b''
>>> aesEncryptor.update(b'bob')
b'\xe7\xf9\x19\xe3!\x1d\x17\x9f\x80\x9d\xf5\xa2\xbaTi\xb2'
```

“Wait!” Alice stops him. “You got something!”

“Weird!” Bob exclaims. “I didn’t do anything different. What happened?”

“Now try decrypting it,” Alice suggests.

```
>>> aesDecryptor.update(_)
b'alicebobbobbobbo'
```

Playing around a bit more, and re-reading docs, Alice and Bob learn what you already discovered from the exercise: the `update` functions for both encryption and decryption always work on 16 bytes at a time. Calling update with fewer than 16 bytes produces no immediate result. Instead, it accumulates data until it has at least 16 bytes to work with. Once 16 or more bytes are available, as many 16-byte blocks of ciphertext as possible are produced. This is illustrated in Figure 3-1.

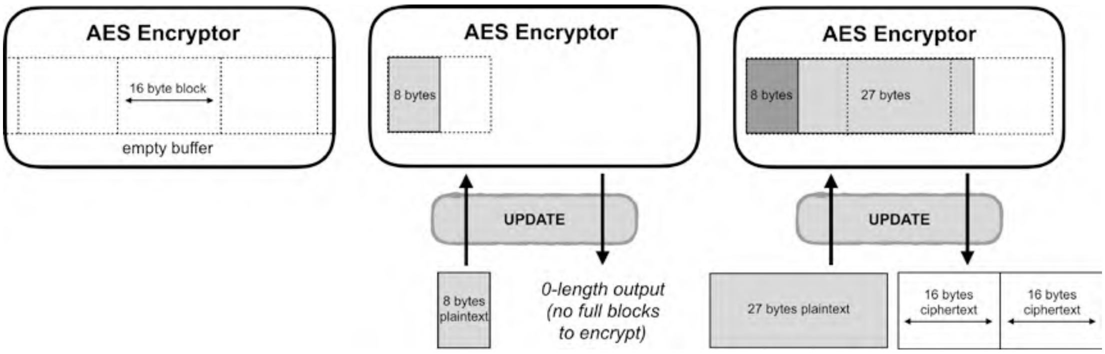

Figure 3-1. *Two calls to the `update` method. The first 8 bytes return nothing because there isn’t a full block of data to encrypt yet.*

EXERCISE 3.2. UPDATED TECHNOLOGY

Upgrade the Caesar cipher application from Chapter 1 to use AES. Instead of specifying a *shift value*, figure out how to get *keys* in and out of the program. You will also have to deal with the 16-byte message size issue. Good luck!

What Is Encryption, Really?

For those who have heard of cryptography, encryption is probably what they have heard about most. Web sites and online services will often mention encryption to reassure you that your information is "secure." They will typically include phrases like "All data transmitted over the Internet is protected by 128-bit encryption, preventing theft."

Don't you feel better already?

Statements like that are really just marketing. They sound nice, but don't usually mean much. That's because "encryption" includes easy-to-break things like Caesar ciphers, it also *isn't enough by itself* to make communications secure. In cryptography, there are *several* properties that contribute to different aspects of security, and they need to work together [11, Chap. 1]. These properties are commonly viewed as the most critical:

1. Confidentiality
2. Integrity
3. Authentication

The encryption we explore in this chapter is all about *confidentiality*. **Confidentiality** means that only folks with the right key are able to read the data. We use encryption to protect messages so that outsiders cannot read them.

Equally important is *integrity*. **Integrity** means that the data cannot be *changed without you noticing*. It is critical to understand that just because something cannot be *read* does not mean it cannot be *usefully altered*. To drive that point home, we are going to do exactly that sort of mischief in this chapter.

Finally, *authentication* relates to knowing the identity of the party with whom you are communicating. **Authentication** typically includes some mechanism to establish *identity and presence,*[3] as well as the ability to tie communication to the established identity.

Hopefully it is obvious that all three of these properties are essential in many forms of communication. Confidentiality will do Alice and Bob little good if Eve can change what the messages *actually say* without them knowing: Eve doesn't need to *read* the messages to cause real problems. Likewise, Alice and Bob will have little success in their covert communications if they aren't sure they have the right person on the other end of the channel.

Keep these ideas in mind as you go through this chapter! Our focus on confidentiality is useful for presentation and it is indeed a critical component of security, but it is not enough. Spending some time with confidentiality by itself will help us to demonstrate how inadequate it is without its friends.

AES: A Symmetric Block Cipher

As mentioned before, the idea behind symmetric encryption is that the same key is used for both encryption and decryption. In the real world, almost all keys to physical locks can be thought of as "symmetric": the same key that locks your door also unlocks it. There are other extremely important approaches to encryption that use distinct keys for each operation, but we'll get to those in later chapters.

Symmetric key encryption algorithms are often divided into two subtypes: **block ciphers** and **stream ciphers**. A block cipher gets its name from the fact that it works on blocks of data: you have to give it a certain amount of data before it can do anything, and larger data must be broken down into block-sized chunks (also, every block must be full). Stream ciphers, on the other hand, can encrypt data one byte at a time.

[3] **Identity and presence** mean loosely "I know who this is, and I know that they consent to my knowing that right now." If you have ever had to dig out your credit card to provide the "CVV code" to a web site that *already has your card on file*, you have run into the concept of presence: the CVV code is meant to be an indication that your card is *there with you* and therefore that *you are around to consent to its use*. This assumes that you are the only one who can hold your own card, a *huge* and easily falsified assumption. Thus, the CVV is an extremely *weak* indication of presence, but ultimately establishment of presence is exactly what it's trying to accomplish.

AES is fundamentally a symmetric key, block cipher algorithm. It is not the only one by any stretch, but it is the only one that we will pay any attention to here. It is used in many common Internet protocols and operating system services, including TLS (used by HTTPS), IPSec, and file-level or full-disk encryption. Given its ubiquity, it is arguably the most important cipher to know how to use properly. More importantly, the principles of correct use of AES transfer easily to correct use of other ciphers.

Finally, even though AES is essentially a block cipher, it (like many other block ciphers) can be used in a way that makes it behave like a stream cipher, so we don't lose any teaching opportunities by excluding native stream ciphers from the discussion. In the past, RC4 was a commonly used stream cipher, but it has been found vulnerable to various attacks and is being replaced by stream modes of AES.

Also, as Bob says, "It's *advanced!*" That ought to be enough for anyone, right?

EXERCISE 3.3. HISTORY LESSON

Do some research online about **DES** and **3DES**. What is the block size for DES? What is its key size? How does 3DES strengthen DES?

EXERCISE 3.4. OTHER CIPHERS

Do a little research about RC4 and Twofish. Where are they used? What kinds of problems does RC4 have? What are some of Twofish's advantages over AES?

Since AES is a pretty good place to start, let's dig in with a little bit of background. We know it's a symmetric key block cipher. Given what we saw of Alice's and Bob's attempts to use it, can you guess the block size?

If you were thinking "16 bytes!" (128 bits), you get a gold star. Tell all your friends![4]

[4] ...over an encrypted channel.

AES has several modes of operation that allow us to achieve different cryptographic properties:

1. Electronic code book (ECB) (***WARNING! DANGEROUS!***)
2. Cipher block chaining (CBC)
3. Counter mode (CTR)

These are not the only modes of operation for AES [11, Chap. 7]. In fact, while CBC and CTR are still used, a newer mode called GCM is now recommended to replace them in many circumstances, and we will examine GCM in detail later in this book. These three modes are, however, very instructive, and together they cover the most important concepts. They will provide a solid foundation on which to build greater understanding of block ciphers in general and AES in particular.

ECB Is Not for Me

Be warned, relying on ECB mode for security is *irresponsibly dangerous* and it should *never* be used. Think of it as being good for testing and educational purposes only. Please, don't ever use it in your applications or projects! Seriously. You have been warned. Don't make us come over there.

By the way, do you see a pattern developing here? Sometimes the best approaches for *explaining* a thing are not at all suitable for *using* it in practice. This seems to apply particularly well to cryptography, which is one reason that we urge people to always use a well established library instead of building their own. The basic principles are simple, but without all of the complex trappings that come with mature libraries and a solid understanding of how to use them, those principles alone will give you *very* poor security, not just "slightly imperfect" security. There is rarely much in the way of middle ground; once the safe is cracked, it doesn't matter how thick its walls are. Cryptographic *concepts* are often simple, but safe and correct *implementation* is usually complex.

With all of those warnings out of the way (not really, there will be more), what is ECB? In a way, ECB is "raw" AES: it treats every 16-byte block of data independently, encrypting each one in exactly the same way using the provided key. As we will see with counter mode and cipher block chaining mode, there are a lot of interesting ways to use that approach as a *building block* for a more advanced (and secure) cipher, but it's really not a good way to go about encryption *by itself.*

The name "electronic code book" hearkens back to earlier days of cryptographic code books, where you would take your (small) key, go to the right page in the book, and use the table on that page to look up the output (ciphertext) that corresponded to each part of your input (plaintext). AES ECB mode can be thought of in that way, but with a mind-bogglingly huge book. The key similarity (ha!) is that once you have the key, *every possible block's encrypted value is known*, and the same is true for decryption; it's like we're looking them up, as visualized in Figure 3-2.

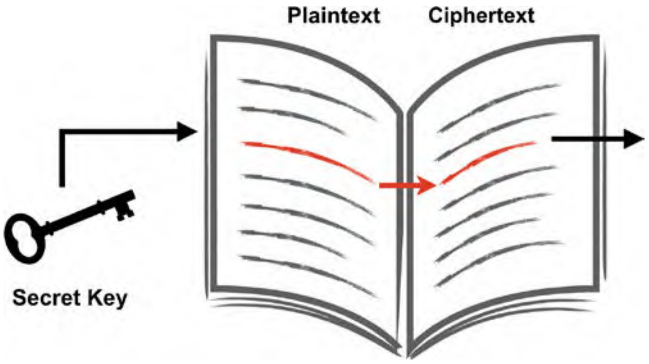

Figure 3-2. *ECB mode is analogous to having a big dictionary of plaintext to ciphertext. Every 16 bytes of plaintext has a corresponding 16-byte output.*

As we will see, the properties of determinism and independence are useful but not sufficient properties for message security. ECB mode is useful because it can be used for testing, for example, to make sure that the AES algorithm is behaving as expected. Some systems will pick a special key, say, all zeros, as a "test key." As part of a self-test, the system will run AES in ECB mode with the test key to see if it encrypts as expected. You will sometimes see tests of this kind called "KATs" (known answer tests).

The National Institute of Standards and Technology (NIST) in the United States publishes a list of KATs that are used for implementation validation. You can download a zip file with these KATs from `https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/aes/KAT_AES.zip`. That archive contains response (.rsp) files that identify expected outputs for given inputs. For example, in the `ECBGFSbox128.rsp` file, the first four ENCRYPT entries are

```
COUNT = 0
KEY = 00000000000000000000000000000000
PLAINTEXT = f34481ec3cc627bacd5dc3fb08f273e6
CIPHERTEXT = 0336763e966d92595a567cc9ce537f5e

COUNT = 1
KEY = 00000000000000000000000000000000
PLAINTEXT = 9798c4640bad75c7c3227db910174e72
CIPHERTEXT = a9a1631bf4996954ebc093957b234589

COUNT = 2
KEY = 00000000000000000000000000000000
PLAINTEXT = 96ab5c2ff612d9dfaae8c31f30c42168
CIPHERTEXT = ff4f8391a6a40ca5b25d23bedd44a597

COUNT = 3
KEY = 00000000000000000000000000000000
PLAINTEXT = 6a118a874519e64e9963798a503f1d35
CIPHERTEXT = dc43be40be0e53712f7e2bf5ca707209
```

That seems useful. Let's test that theory using Listing 3-2.

Listing 3-2. AES ECB KATs

```python
# NEVER USE: ECB is not secure!
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

# NIST AES ECBGFSbox128.rsp ENCRYPT Kats
# First value of each pair is plaintext
# Second value of each pair is ciphertext
nist_kats = [
    ('f34481ec3cc627bacd5dc3fb08f273e6', '0336763e966d92595a567cc9ce537f5e'),
    ('9798c4640bad75c7c3227db910174e72',
    'a9a1631bf4996954ebc093957b234589'),
    ('96ab5c2ff612d9dfaae8c31f30c42168',
    'ff4f8391a6a40ca5b25d23bedd44a597'),
    ('6a118a874519e64e9963798a503f1d35 ',
    'dc43be40be0e53712f7e2bf5ca707209')
]

# 16-byte test key of all zeros.
test_key = bytes.fromhex('00000000000000000000000000000000')

aesCipher = Cipher(algorithms.AES(test_key),
                   modes.ECB(),
                   backend=default_backend())
aesEncryptor = aesCipher.encryptor()
aesDecryptor = aesCipher.decryptor()

# test each input
for index, kat in enumerate(nist_kats):
    plaintext, want_ciphertext = kat
    plaintext_bytes = bytes.fromhex(plaintext)
    ciphertext_bytes = aesEncryptor.update(plaintext_bytes)
    got_ciphertext = ciphertext_bytes.hex()

    result = "[PASS]" if got_ciphertext == want_ciphertext else "[FAIL]"

    print("Test {}. Expected {}, got {}. Result {}.".format(
        index, want_ciphertext, got_ciphertext, result))
```

Assuming that your processor is working correctly, you should see a 4/4 passing score.

EXERCISE 3.5. ALL NIST KATS

Write a program that will read one of these NIST KAT “rsp” files, and parse out the encryption and decryption KATs. Test and validate your AES library on all vectors on a couple of ECB test files.

This all seems very reasonable. So, what's wrong with ECB? Unless you've been completely asleep, you've noticed our dire warnings about it. Why? In a nutshell, because of its independence properties.

Let's return to Alice, Bob, and their nemesis Eve down in Antarctica. Alice and Bob are on a covert mission within the West Antarctica borders. They will send secret messages to each other over radio channels that Eve can monitor. Before they leave, they generate a shared key for encrypting and decrypting their messages, and they keep that key safe during their travels.

We can do that too. We'll start by generating a key. Normally, the key would be random, but we'll just pick one that is easy to remember, then we can also perfectly reproduce the following results. Here is the key:

```
key = bytes.fromhex('00112233445566778899AABBCCDDEEFF')
```

Alice and Bob, being government agents, use a standardized EATSA form for sending each other messages. For example, to arrange a meeting:

```
FROM: FIELD AGENT<codename>
TO: FIELD AGENT<codename>
RE: Meeting
DATE: <date>

Meet me today at <location> at <time>
```

If Alice is telling Bob to meet her at the docks at 11 p.m., the message would be

```
FROM: FIELD AGENT ALICE
TO: FIELD AGENT BOB
RE: Meeting
DATE: 2001-1-1

Meet me today at the docks at 2300.
```

We'll encrypt this message under the key we set out previously. We need to *pad* the message to make sure it is a multiple of 16 bytes long. We can do that by adding extra characters to the end until its length is a multiple of 16, like so.[5]

[5] We take advantage of Python's convenient "negative modulus" behavior, where `-len(msg) % 16` is the same as `16 - (len(msg) % 16)`.

Listing 3-3. AES ECB Padding

```
# NEVER USE: ECB is not secure!
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

# Alice and Bob's Shared Key
test_key = bytes.fromhex('00112233445566778899AABBCCDDEEFF')

aesCipher = Cipher(algorithms.AES(test_key),
                   modes.ECB(),
                   backend=default_backend())
aesEncryptor = aesCipher.encryptor()
aesDecryptor = aesCipher.decryptor()

message = b"""
FROM: FIELD AGENT ALICE
TO: FIELD AGENT BOB
RE: Meeting
DATE: 2001-1-1

Meet me today at the docks at 2300."""

message += b"E" * (-len(message) % 16)
ciphertext = aesEncryptor.update(message)
```

Listing 3-3 shows a straightforward but perhaps not optimal padding. We'll use more standard approaches in the next section. It is, however, good enough for now. When Bob decodes his message, it will simply have a few extra "E" characters at the end.

EXERCISE 3.6. SENDING BOB A MESSAGE

Using either a modification of the preceding program or your AES encryptor from the beginning of the chapter, create a couple of meetup messages from Alice to Bob. Also create a few from Bob to Alice. Make sure that you can correctly encrypt and decrypt the messages.

With their new cryptographic technology at the ready, Alice and Bob begin surveillance in West Antarctica. They meet occasionally to share information and coordinate their activities.

Meanwhile, Eve and her counter-intelligence colleagues learn of the infiltration and soon begin to identify the coded messages. Take a look at several messages from Alice to Bob from Eve's perspective, where all she can see is the ciphertext. Do you notice anything?

Consider these two messages:

```
FROM: FIELD AGENT ALICE
TO: FIELD AGENT BOB
RE: Meeting
DATE: 2001-1-1

Meet me today at the docks at 2300.
```

```
FROM: FIELD AGENT ALICE
TO: FIELD AGENT BOB
RE: Meeting
DATE: 2001-1-2

Meet me today at the town square at 1130.
```

Look at the two ciphertext outputs of these messages side-by-side. Note: even spacing and newlines matter, so make sure to use the format exactly as shown.

Message 1, Block 1	a3a2390c0f2afb700959b3221a95319a
Message 2, Block 1	a3a2390c0f2afb700959b3221a95319a
Message 1, Block 2	0fd11a5dcfa115ba89630f93e09312b0
Message 2, Block 2	0fd11a5dcfa115ba89630f93e09312b0
Message 1, Block 3	87597bf7f98759410ae3e9a285912ee6
Message 2, Block 3	87597bf7f98759410ae3e9a285912ee6
Message 1, Block 4	8430e159229e4bf5c7b39fe1fb72cfab
Message 2, Block 4	8430e159229e4bf5c7b39fe1fb72cfab
Message 1, Block 5	a5c7412fda6ac67fe63093168f474913

Message 2, Block 5	c9b3ccefda71f286895b309d85245421
Message 1, Block 6	dbd386db053613be242c6059539f93da
Message 2, Block 6	699f1cd5adbeb94b80980a0860ead320
Message 1, Block 7	800d3ece3b12931be974f36ef5da4342
Message 2, Block 7	a8ff0ed2ca9b80908757f8c3ecbc9b0d

How many of the 16-byte blocks are identical? Why?

Remember that AES in its raw mode is like a code book. For every input and key, there is exactly one output, independent of any other inputs. Thus, because much of the message header is shared between messages, much of the output is *also* the same.

Eve and her colleagues notice the repeating elements of the messages they see day after day and soon start to figure out what they mean. How do they do this? They might make a good start by guessing. If you saw the same message being sent repeatedly, you could start to guess at some of its contents.

Another way to make progress might be to utilize a deserter or mole within the enemy organization. They could conceivably get Eve a copy of the form or a discarded decoded message. All told, there are many ways for an adversary to learn about the structure and organization of an encrypted message, and you should never assume otherwise. A common error made by those trying to protect information is to assume that the enemy cannot know some detail about how the system works.

Instead, always live by Kerckhoff's principle. This nineteenth-century (long before modern computers) cryptographer taught that a cryptographic system must be secure even if *everything* is known about it, except the key. That means we should find a way for our messages to be secure if the enemy knows just about everything about our system and merely lacks access to the key.

We made this silly example with an overly bureaucratic form, but even in real messages, there is often a significant amount of predictable structure. Consider HTML, XML, or email messages. Those often have huge amounts of predictably positioned, identical data. It would be a terrible thing for an eavesdropper to start learning what's in a message just because it shares protocol headers with every other message.

Even worse, imagine if Eve's team can figure out a way to do what is called a "chosen plaintext" attack. In this attack, they figure out a way to get Alice or Bob to encrypt something on their behalf. Imagine, for example, that they figure out that Alice always calls a meeting with Bob after the Prime Minister of Western Antarctica gives a public speech. Once they know this, they can use political speeches to trigger a message where much of the content is known. Or maybe they manage to slip Bob some false information to send to Alice, encrypted. Once they can control some or all of the plaintext, they can look at the encryption and begin to *create their own code book*.

Eve can also easily create new messages by putting together bits and pieces of old messages. If Eve knows that the first blocks of a ciphertext are the header with a current date, she can take an old message body that directs Bob to an old meeting site and attach it to the new header. Then Bob ends up in the wrong place at the wrong time.

EXERCISE 3.7. SENDING BOB A FAKE MESSAGE

Take two different ciphertexts from Alice to Bob with different meeting instructions on different dates. Splice the ciphertext from the body of the first message into the body of the second message. That is, start by replacing the last block of the newer message with the last block (or blocks if it was longer) of the previous message. Does the message decrypt? Did you change where Bob goes to meet Alice?

All of this may still seem just a bit hypothetical. Perhaps ECB mode isn't really all that bad. Perhaps it is only bad in extreme situations or something like that. Just in case there is a shadow of a doubt remaining, let's do one more test (a pretty fun one) to convince ourselves that ECB mode should never, *ever* be used for real message confidentiality.

In this experiment, you will build a very basic AES encrypting program. It doesn't matter what key is used; feel free to generate a random one, or use a fixed test key. Read in a binary file, encrypt everything *except the first 54 bytes*, and then write it out to a new file. It might look something like Listing 3-4.[6]

Listing 3-4. AES Exercise Example

```
# Partial Listing: Some Assembly Required

ifile, ofile = sys.argv[1:3]
with open(ifile, "rb") as reader:
```

[6] This code listing does not show all the necessary imports, but it requires nothing new over previous listings. In the interest of space, we will regularly leave out details that have been shown in previous examples.

```
    with open(ofile, "wb+") as writer:
        image_data = reader.read()
        header, body = image_data[:54], image_data[54:]
        body += b"\x00"*(16-(len(body)%16))
        writer.write(header + aesEncryptor.update(body))
```

The reason we're not encrypting the first 54 bytes is because this program is going to encrypt the contents of a bit map file (BMP) and the header is 54 bytes in length.[7] Once you have this listing written, in the image editor of your choice, create a large image with text that takes up most of the space. In Figure 3-3, our image simply has the words "TOP SECRET." It is 800x600 pixels.

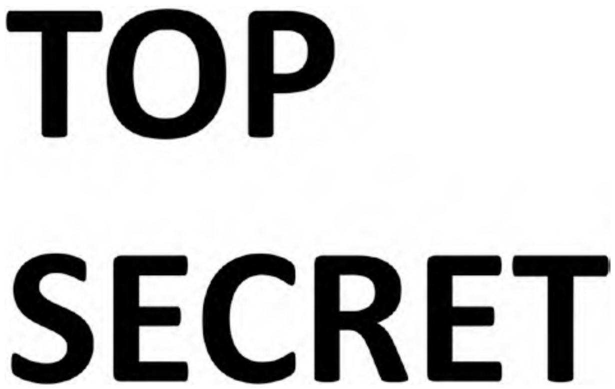

***Figure 3-3.** An image with the text "TOP SECRET." Encrypting it should make it unreadable, right?*

Take your newly created file and run it through your encryption program, saving the output to something like `encrypted_image.bmp`. When finished, open *the encrypted file* in an image viewer. What do you see?

Our encrypted image is shown in Figure 3-4.

[7] In real life, if the header were encrypted, you could just overwrite it with something reasonable based on the file size.

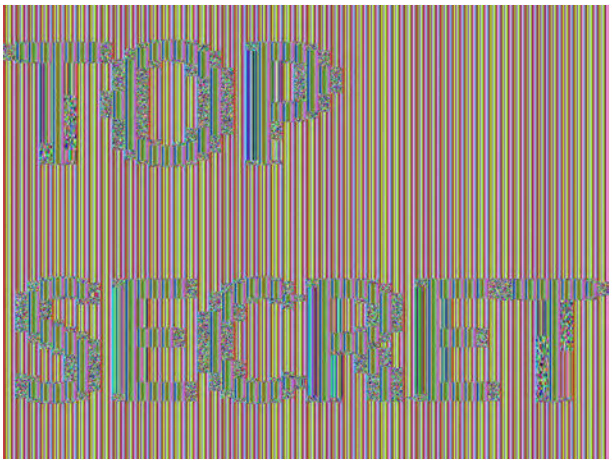

Figure 3-4. *This image was encrypted using ECB mode. This message is not very confidential.*

What happened here? Why is the text of the image still so readable?

AES is a block cipher that operates on 16 bytes at a time. In this image, many 16-byte chunks are the *same*. A chunk of black pixels is encoded with the same bits everywhere. Every time there's a 16-byte block of all black or all white, they encode to the same encrypted output. The *structure* of the image is thus still visible even once the individual 16-byte chunks are encrypted.

Really. Never use ECB. Leave that kind of thing to the "professionals" of the East Antarctica Truth-Spying Agency.

Wanted: Spontaneous Independence

To have an effective cipher, we need to

- Encrypt the same message differently each time.
- Eliminate predictable patterns between blocks.

To solve the first problem, we use a simple but effective trick to ensure that we never send the same *plaintext* twice, which means that we also never send the same *ciphertext* twice! We do this with an "initialization vector," or IV.

An IV is typically a random string that is used as a third input—in addition to the key and plaintext—into the encryption algorithm. Exactly how it is used depends on the mode, but the idea is to prevent a given plaintext from encrypting to a repeatable ciphertext.

Unlike the key, the IV is public. That is, one assumes that an attacker knows, or can obtain, the value of the IV. The presence of an IV doesn't help to keep things *secret* so much as it helps to keep them from being *repeated*, avoiding exposure of common patterns.

As for the second problem, that of being able to eliminate patterns between blocks, we will solve it by introducing new ways to encrypt the message *as a whole*, rather than treating each block as an individual, independent mini-message like ECB mode does.

The details of each solution are specific to the mode being used, but the principles generalize well.

Not That Blockchain

Recall from Chapter 2 that good hash algorithms are expected to have the *avalanche* property. That is, a single change in one input bit will cause approximately half of the output bits to change. Block ciphers should have a similar property, and thankfully, AES does. In ECB mode, however, the avalanche's impact is limited to the block size: if the plaintext is ten blocks long, a change in the very first bit will only change the output bits of the very first block. The remaining nine blocks will remain unchanged.

What if a change in the ciphertext of one block could affect all *subsequent* blocks? Well, it can, and it is quite easy to accomplish. When encrypting, for example, one can XOR the encrypted output of a block with the unencrypted input of the next block. To reverse this while decrypting, the ciphertext is decrypted and then the XOR operation is again applied to the previous ciphertext block to obtain the plaintext. This is called **cipher block chaining (CBC) mode**.

Let's pause here for just a quick moment to review an operation called XOR, often written symbolically as ⊕. We are going to use XOR constantly throughout the book so it's worthwhile to review it. XOR is a binary boolean operator with the following truth table (where we use 0 and 1 instead of "false" and "true').

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Truth tables are useful, showing precisely how functions like XOR behave for all combinations of inputs, but you actually don't need to think about XOR at this level. What is important is that XOR has an amazing inversion property: the XOR operation is its own inverse! That is, if you start with some binary number A and XOR it with B, you can recover A by XORing the output with B again. Mathematically, it looks like this: $(A \oplus B) \oplus B = A$.

Why does this work? If you look at "Input 1" as a control bit, when it is 0, what comes out is simply "Input 2." When "Input 1" is 1, on the other hand, what comes out is the inverse of "Input 2." If you take the outputs and apply XOR with "Input 1" again, it leaves the previously unchanged things unchanged (XOR with 0 again) while flipping the inverted things back to the way they were (XOR with 1 again).

Quite often we XOR not individual bits, but sequences of bits all at once. This is how we will use XOR throughout this book: as an operation between blocks of bits, like this:

$$\begin{array}{cr} & 11011011 \\ \oplus & 10110001 \\ \hline & 01101010 \\ \oplus & 10110001 \\ \hline & 11011011 \end{array}$$

You can see here how applying $\oplus$10110001 twice to 11011011 causes it to reappear.

EXERCISE 3.8. XOR EXERCISE

Because we will use XOR so much, it's a good idea to get comfortable with XOR operations. In a Python interpreter, XOR a few numbers together. Python supports XOR directly using ^ as the operator. So, for example, 5^9 results in 12. What do you get when you try 12^9? What do you get when you try 12^5? Try this out with several different numbers.

EXERCISE 3.9. THE MASK OF XOR-O?

Although this exercise will be even more important in counter mode, it's useful to understand how XOR can be used to *mask* data. Create 16 bytes of plaintext (a 16-character message) and 16 bytes of random data (e.g., using `os.urandom(16)`). XOR these two messages together. There's no built-in operation for XORing a series of bytes, so you'll have to XOR each byte individually using, for example, a loop. When you are done, take a look at the output. How "readable" is it? Now, XOR this output with the same random bytes again. What does the output look like now?

Returning from our XOR interruption to CBC, in this mode we XOR the output of one block of ciphertext with the next *plaintext* block. More precisely, if we call $P[n]$ block n of plaintext and $P'[n]$ block n of "munged, pre-encryption plaintext" (using the XOR operation to accomplish the very scientifically named "munging" process), we first create $P'[n]$ from the previous encrypted block $C[n-1]$, then we encrypt it to make $C[n]$. The formula for creating $P'[n]$ is this:

$$P'[n] = P[n] \oplus C[n-1],$$

From there we can apply AES encryption to $P'[n]$, which is the length of an AES block, to get $C[n]$. When decrypting, then, we don't get the plaintext, we get the "munged, pre-encryption plaintext" $P'[n]$. To get the actual plaintext, we need to reverse the preceding process, which we can do by running it through XOR with the previous encrypted block (recalling that XOR is its own inverse). You can see why this works by performing some basic algebraic manipulations:

$$\begin{aligned} P'[n] &= P[n] \oplus C[n-1] \\ P'[n] \oplus P[n] &= P[n] \oplus P[n] \oplus C[n-1] \\ P'[n] \oplus P[n] &= C[n-1] \\ P'[n] \oplus P'[n] \oplus P[n] &= P'[n] \oplus C[n-1] \\ P[n] &= P'[n] \oplus C[n-1]. \end{aligned}$$

Thus, to get the original plaintext when decrypting, we need only XOR the decrypted block with the previous encrypted block. The very first block, which has no predecessor, is simply XORed with the initialization vector after decryption. That is the essence of

CBC mode: every block is dependent on the blocks that came before. This process is visualized, perhaps a little more intuitively, in Figure 3-5.

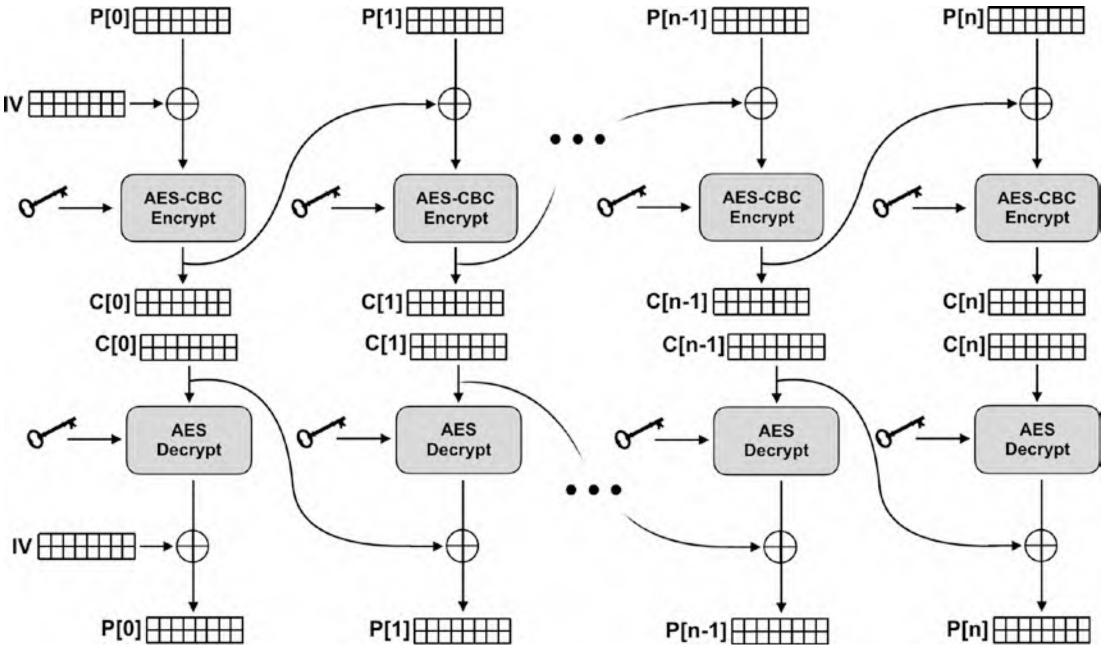

Figure 3-5. *Visual depictions of CBC encryption and decryption. Note that in encryption, the first block of plaintext is XORed with the IV before AES, while in decryption, the ciphertext goes through AES first and is then XORed with the IV to correctly reverse the encryption process.*

In CBC mode, changes to any input block thus affect the output block for all subsequent blocks. This doesn't produce a complete or perfect avalanche property because it does not affect any *preceding* blocks, but even having the avalanche effect moving forward prevents exposing the kinds of patterns that we observe in ECB mode.

Configuration of CBC mode is mostly familiar: we generate a key and then take the extra step of generating an initialization vector (IV). Because the IV is XORed with the first block, AES-CBC IVs[8] are *always* 128 bits long (16 bytes), even if the key size is larger (typically 196 or 256 bits). In the following example, the key is 256 bits and the IV is 128 bits, as it must be (Listing 3-5).

[8] We promise to use more initialisms next time.

Listing 3-5. AES-CBC

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes
from cryptography.hazmat.backends import default_backend
import os

key = os.urandom(32)
iv = os.urandom(16)

aesCipher = Cipher(algorithms.AES(key),
                   modes.CBC(iv),
                   backend=default_backend())
aesEncryptor = aesCipher.encryptor()
aesDecryptor = aesCipher.decryptor()
```

Notice that in this example, `algorithms.AES` takes the key as the parameter while `modes.CBC` takes the IV; AES *always* needs a key, but the use of an IV is dependent on the mode.

Proper Padding

While we are in the business of improving things, let's introduce a better padding mechanism. The `cryptography` module provides two schemes, one following what is known as the PKCS7 specification and the other following ANSI X.923. PKCS7 appends *n* bytes, with each padding byte holding the value *n*: if 3 bytes of padding are needed, it appends `\x03\x03\x03`. Similarly, if 2 bytes of padding are needed, it appends `\x02\x02`.

ANSI X.923 is slightly different. All appended bytes are 0, except for the last byte, which is the length of the total padding. In this example, 3 bytes of padding is `\x00\x00\x03`, and two bytes of padding is `\x00\x02`.

The `cryptography` module provides a padding context that is analogous to the AES cipher context. In the next code listing, `padder` and `unpadder` objects are created for adding and removing padding. Note that these objects also use `update` and `finalize`, since no padding is created from calling the `update()` method. It does, however, return full blocks, storing the rest of the bytes for either the next call to `update()` or the `finalize()` operation. When `finalize()` is called, all remaining bytes are returned along with enough bytes of padding to make a full block size.

Although the API seems straightforward, it doesn't necessarily behave as one might expect.

Listing 3-6. AES-CBC Padding

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os

key = os.urandom(32)
iv = os.urandom(16)

aesCipher = Cipher(algorithms.AES(key),
                   modes.CBC(iv),
                   backend=default_backend())
aesEncryptor = aesCipher.encryptor()
aesDecryptor = aesCipher.decryptor()

# Make a padder/unpadder pair for 128 bit block sizes.
padder = padding.PKCS7(128).padder()
unpadder = padding.PKCS7(128).unpadder()

plaintexts = [
    b"SHORT",
    b"MEDIUM MEDIUM MEDIUM",
    b"LONG LONG LONG LONG LONG LONG",
]

ciphertexts = []

for m in plaintexts:
    padded_message = padder.update(m)
    ciphertexts.append(aesEncryptor.update(padded_message))

ciphertexts.append(aesEncryptor.update(padder.finalize()))

for c in ciphertexts:
    padded_message = aesDecryptor.update(c)
    print("recovered", unpadder.update(padded_message))

print("recovered", unpadder.finalize())
```

Run the code in Listing 3-6 and observe the output. Is it what you expected? It should have looked like this:

```
recovered b''
recovered b''
recovered b'SHORTMEDIUM MEDIUM MEDIUMLONG LO'
recovered b'NG LONG LONG LON'
recovered b'G LONG '
```

Why did it not produce the original messages exactly as specified?

There is technically nothing incorrect with this code, but there is definitely a mismatch between the apparent *intention* of the code and the actual output. This code suggests that the author intended to encrypt each one of the three strings as an independent message. In other words, the probable intention of the code was to encrypt three distinct messages and get back three equivalent messages upon decryption.

That is not what we got. Listing 3-6 is reporting five outputs and two of them are empty.

Let's talk about the `update()` and `finalize()` API one more time. Because of how these methods behave for certain modes (e.g., ECB mode), it can be tempting to think about `update()` as a stand-alone encryptor wherein a plaintext block is provided as input and a ciphertext block is provided as output.

In reality, the API is designed such that the number of calls to `update()` is irrelevant. That is, what is being encrypted is not the input to `\lstinline{update()}, but \emph{the concatenation of every input}` to some number of `\lstinline{update()}` calls, and, of course, output (if any) from a `finalize()` call at the end.

Thus, the program in Listing 3-6 is not encrypting three inputs and producing five outputs, it is processing a single continuous output and producing a single continuous output.

Understanding the `update()` and `finalize()` API is especially important for the padding operations that we've introduced. Padding behavior can appear unusual if you try to think of `update()` as an independent operation. Figure 3-6 illustrates how padding processes the inputs from Listing 3-6. Note that individual calls to `update()` produce no padding. Only the `finalize()` operation will do that.

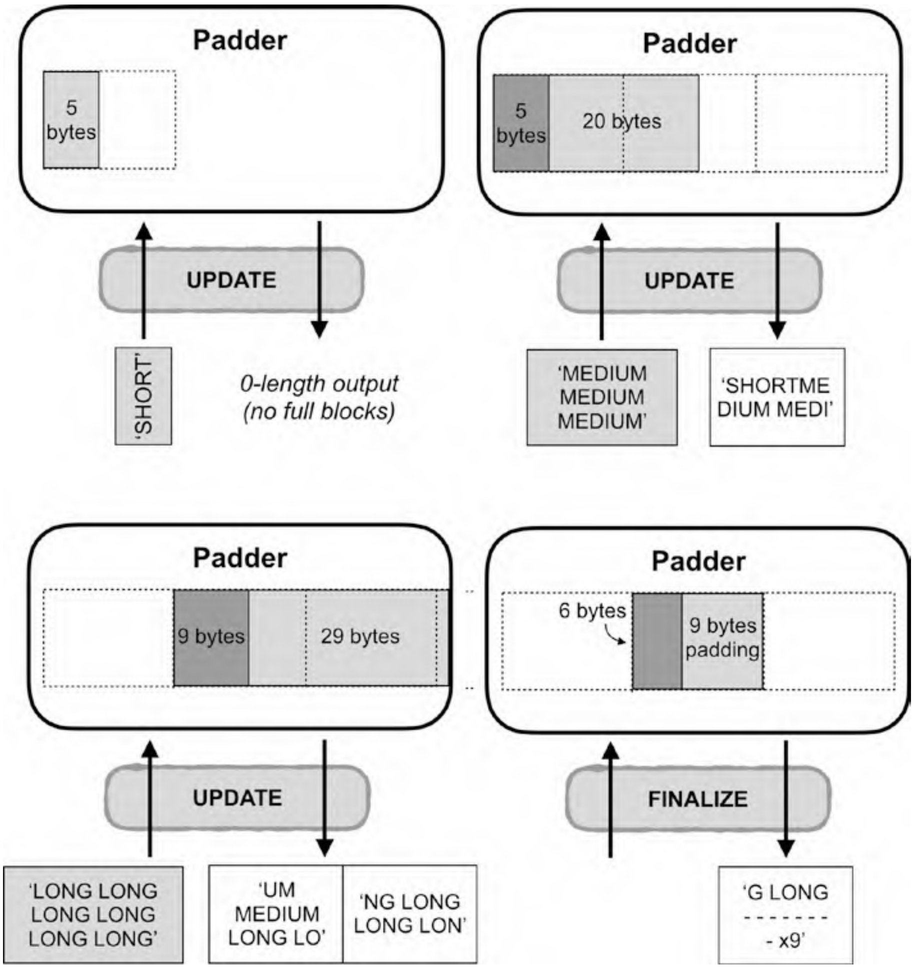

Figure 3-6. *PKCS7 padding does not add any padding until the finalize operation*

Unpadding can be even more jarring. Unlike the padding operations, you can submit a full block to the unpadder and still get *nothing* back. This is because the unpadder has to reserve the last block received in `update()` calls in case it is the last block. Because unpadding requires examining the last block, the unpadder has to be sure it has received all the blocks to know that it has the last one.

Walking through Listing 3-6 one more time illustrates how the effects of these operations are compounded when the padder and encryptor are used together. On the first pass through the loop for encrypting the messages, the input is `SHORT`. Five characters is less than of a block. The padder's `update()` method does not add any padding, so the *padder* buffers these five characters and the `update()` method returns an empty byte string. When this gets passed to the encryptor, there is obviously not a full block so the encryptor's update method also returns an empty byte string. This gets appended to the list of ciphertexts.

On our second pass through the loop, the input is `MEDIUM MEDIUM MEDIUM`. These 20 characters are passed into the padder's internal buffer and are added to the 5 that were there before. The `UPDATE` method now returns the first 16 of those 25 bytes (a full block), leaving the remaining 9 bytes in the internal buffer. The 16 bytes from the padder are encrypted and stored in the list of ciphertexts.

In the final pass, the `LONG LONG LONG LONG LONG LONG` input is added to the padder's internal buffer. These 29 bytes are added to the current 9 bytes in the buffer for a total of 38 bytes. The padder returns the 2 full blocks (of 16 bytes each) leaving the last 6 bytes in its buffer. The two blocks are encrypted, and the two-block output is stored in the list of ciphertexts.

Once the loop exits, the padder's finalize method is called. It takes the last bytes of input, appends the necessary padding, and passes it to the encryption operation. The ciphertext is appended to the list and encryption is over. There are now four ciphertext messages to decrypt. Reversing the process, the first message is, you may recall, the empty buffer. It just passes straight through everything and comes out as an empty message.

But the next recovered text is also empty. That's because the first full block to the unpadder is reserved for the reasons we explained. It produces an empty output fed into the AES decryptor's `update()` method. This generates our second empty output.

The remaining three are more straightforward.

Now that the walk-through is finished, did you notice that we were still using the incorrect terminology? We referred to individual outputs from `update()` methods as individual ciphertexts rather than a snippet of *the* ciphertext. Similarly, we called the output of the decryptor update methods recovered texts rather than part of a single recovered message.

This was intentional. The crucial principle is that *semantics* matter. How we think about our code can be different from how it operates, and this can result in unexpected, and often *insecure*, results. When you use a library (always better than creating your own!), you must understand the API's approach and design. It is critical that you *think* the way the API is designed to be used.

For the `cryptography` library, always think about everything submitted to a sequence of encryption `update()` calls and one `finalize()` call as a single input. Similarly, think about everything that is recovered from a series of decryption `update()` calls and one `finalize()` call as a single output.

And what is going on with the decryption? How did we get *five* outputs instead of four? The first ciphertext in the list was just the empty string so it makes sense that the first "recovered" plaintext was empty. But why was the second one empty too?

Let's look at another way to do this wrong.[9] Suppose we decide to create our own API that will actually work on a message level. That is, every message can be encrypted and decrypted individually and independently. The code is shown in Listing 3-7.

Listing 3-7. Broken AES-CBC Manager

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os

class EncryptionManager:
    def __init__(self):
        self.key = os.urandom(32)
        self.iv = os.urandom(16)

    def encrypt_message(self, message):
        # WARNING: This code is not secure!!
        encryptor = Cipher(algorithms.AES(self.key),
                           modes.CBC(self.iv),
```

[9] Yes, this is a theme in the book. We have found that it is often when things are broken that they are best understood.

```
                           backend=default_backend()).encryptor()
        padder = padding.PKCS7(128).padder()

        padded_message = padder.update(message)
        padded_message += padder.finalize()
        ciphertext = encryptor.update(padded_message)
        ciphertext += encryptor.finalize()
        return ciphertext

    def decrypt_message(self, ciphertext):
        # WARNING: This code is not secure!!
        decryptor = Cipher(algorithms.AES(self.key),
                           modes.CBC(self.iv),
                           backend=default_backend()).decryptor()
        unpadder = padding.PKCS7(128).unpadder()

        padded_message = decryptor.update(ciphertext)
        padded_message += decryptor.finalize()
        message = unpadder.update(padded_message)
        message += unpadder.finalize()
        return message

# Automatically generate key/IV for encryption.
manager = EncryptionManager()

plaintexts = [
    b"SHORT",
    b"MEDIUM MEDIUM MEDIUM",
    b"LONG LONG LONG LONG LONG LONG"
]

ciphertexts = []

for m in plaintexts:
    ciphertexts.append(manager.encrypt_message(m))
```

```

for c in ciphertexts:
    print("Recovered", manager.decrypt_message(c))
```

Run the code and observe the output. Did you get each message individually this time? Good! You probably like this version a lot better!

The API might be more semantically aligned, this time, but the implementation is very broken and incredibly dangerous. Before we tell you what is wrong with it, can you try and see it yourself? Are there any security principles we've talked about in this chapter that we are violating? If it isn't obvious, read on!

A Key to Hygienic IVs

The problem with Listing 3-7 is that it is *reusing the same key and IV* for different messages. Take a look at the constructor where the key and IV are created. Using that single key/IV pair, the offending code re-creates encryptor and decryptor objects in every call to `encrypt_message` and `decrypt_message`. Remember, the IV is supposed to be *different* each time you encrypt, preventing the same data from being encrypted to the same ciphertext! **This is not optional.**

Once again, it is important to understand how an API is built and the security parameters associated with it. Go back and look at Figure 3-5. Remember that in CBC encryption, the algorithm combines the first plaintext block with the IV using the XOR operation before the AES operation is applied. Each subsequent plaintext block is combined with the previous ciphertext block using XOR before AES encryption. With the Python API, each call to `update()` adds blocks to this chain, leaving data less than a full block in an internal buffer for subsequent calls. The `finalize()` method does not actually do any more encrypting, but will raise an error if there is incomplete data still waiting to be encrypted.

Calling the `update()` method over and over is *not* reusing a key and IV because we are appending to the end of the CBC chain. On the other hand, if you create new encryptor and decryptor objects, as we did in Listing 3-7, you are re-creating the chain from the beginning. If you reuse a key and IV here, you will with the same key and IV! This results in *exactly the same output for the same input every time!*

Accordingly, when using the API of Python's `cryptography` module, never give the same key and IV pair to an encryptor more than once (obviously, you give the same key and IV to the corresponding decryptor). In fact, it's probably best to never reuse the same key again, period.

In Listing 3-8 we correct our previous error and only use a key/IV pair once. The encryptor and decryptor objects are moved to the constructor and, instead of having a single encrypt_message() or decrypt_message() call, we use the update/finalize pattern used by the cryptography module.

Listing 3-8. AES-CBC Manager

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os

class EncryptionManager:
    def __init__(self):
        key = os.urandom(32)
        iv = os.urandom(16)
        aesContext = Cipher(algorithms.AES(key),
                            modes.CBC(iv),
                            backend=default_backend())
        self.encryptor = aesContext.encryptor()
        self.decryptor = aesContext.decryptor()
        self.padder = padding.PKCS7(128).padder()
        self.unpadder = padding.PKCS7(128).unpadder()

    def update_encryptor(self, plaintext):
        return self.encryptor.update(self.padder.update(plaintext))

    def finalize_encryptor(self):
        return self.encryptor.update(self.padder.finalize()) + self.encryptor.finalize()

    def update_decryptor(self, ciphertext):
        return self.unpadder.update(self.decryptor.update(ciphertext))

    def finalize_decryptor(self):
        return self.unpadder.update(self.decryptor.finalize()) + self.unpadder.finalize()

# Auto generate key/IV for encryption
manager = EncryptionManager()

plaintexts = [
    b"SHORT",
    b"MEDIUM MEDIUM MEDIUM",
    b"LONG LONG LONG LONG LONG LONG"
]

ciphertexts = []

for m in plaintexts:
    ciphertexts.append(manager.update_encryptor(m))
ciphertexts.append(manager.finalize_encryptor())

for c in ciphertexts:
    print("Recovered", manager.update_decryptor(c))
print("Recovered", manager.finalize_decryptor())
```

Listing 3-8 does not reuse key/IV pairs, but you have probably noticed that we are no longer treating the individual messages as individual messages. Now that we're back to the update() finalize() pattern, we have to treat all the data passed to a single context as a single input. If we want each message treated separately, with a sequence of update() calls and finalize() call *per input*. Alternatively, we can submit all three messages as a single input from the perspective of the encryption and decryption and have an independent mechanism for splitting the single decryption output into messages.

In summary, it is important to carefully understand any cryptography APIs that you use, how they work, and what their requirements (especially security requirements) are. It is also important to understand how easy it can be to create an API that looks like it does the right thing but is actually leaving you vulnerable.

Remember, YANAC (You Are Not A Cryptographer... yet!). Don't roll your own crypto like we are doing in these educational examples.

So why does the `cryptography` module use the update/finalize pattern? Quite often, data needs to be processed in chunks in many practical cryptographic operations. Suppose that you are transmitting data over the network. Do you really want to wait until you have the entire content before you can encrypt it? Even if you were encrypting a local file on the hard drive, it might be impractically large for all-at-once encryption. The `update()` method allows you to feed data to an encryption engine as it becomes available.

The `finalize()` operation is useful for enforcing requirements such as the CBC operation did not leave an incomplete block unencrypted and that the session is over.

Of course, there's nothing wrong with a per-message API so long as a key and IV aren't reused. We will look at strategies for this later.

EXERCISE 3.10. DETERMINISTIC OUTPUT

Run the same inputs through AES-CBC using the same key and IV. You can use Listing 3-7 as a starting point. Change the inputs to be the same each time and print out the corresponding ciphertexts. What do you notice?

EXERCISE 3.11. ENCRYPTING AN IMAGE

Encrypt the image that you encrypted with ECB mode earlier. What does the encrypted image look like now? Don't forget to leave the first 54 bytes untouched!

EXERCISE 3.12. HAND-CRAFTED CBC

ECB mode is just raw AES. You can create your own CBC mode using ECB as the building block.[10] For this exercise, see if you can build a CBC encryption and decryption operation that is compatible with the `cryptography` library. For encryption, remember to take the output of each block and XOR it with the plaintext of the next block before encryption. Reverse the process for decryption.

[10] Never use this for production code! Always use well-tested libraries.

Cross the Streams

Counter mode has a number of advantages to CBC mode and, in our opinion, is significantly easier to understand than CBC mode. Also, while CTR is the traditional abbreviation, "CM" is a really nice set of initials.

Although simple, the concept behind this mode can be a little counter-intuitive at first (yup). In CTR mode, you actually *never use AES* for encryption or decryption of the data. Instead, this mode generates a *key stream* that is the same length as the plaintext and then uses XOR to combine them together.

Recall from earlier exercises in this chapter that XOR can be used to "mask" plaintext data by combining it with random data. The previous exercise masked 16 bytes of plaintext with 16 bytes of random data. This is a real form of encryption called a "one-time pad" (OTP) [11, Chap. 6]. It works great but requires that the *key is the same size as the plaintext*. We don't have the space here to explore the OTP further; the important concept is that using XOR to combine plaintext and random data is a great way to create ciphertext.

AES-CTR mimics this aspect of OTP. But instead of requiring the key to be the same size as the plaintext (a real pain when encrypting a 1TB file), it uses AES and a counter to generate a key stream of almost arbitrary length from an AES key as small as 128 bits.

To do this, CTR mode uses AES to encrypt a 16-byte counter, which generates 16 bytes of key stream. To get 16 more bytes of key stream, the mode increases the counter by one and encrypts the updated 16 bytes. By continually increasing the counter and encrypting the result, CTR mode can produce an almost arbitrary amount of key stream material.[11] Once a sufficient amount of key material is generated, the XOR operation is used to combine them together to produce the ciphertext.

Although the counter is changing by a small amount each time (often just changing by a single bit!), AES has good per-block avalanche properties. Thus, each output block appears completely different from the last, and the stream as a whole appears to be random data.

[11] There are limits but these are beyond the scope of this book.

Note: Random Thoughts

Randomness is actually a huge deal in cryptography. Many otherwise acceptable algorithms have been compromised in practice if they did not have sufficient sources of randomness for keys, among other things. The OTP algorithm we briefly mentioned requires a key that is the same size as the plaintext (no matter how large) and that the entire key be truly random data. AES-CTR mode only requires that the AES key be truly random. The key stream produced by AES-CTR *looks* random, but is actually *pseudo-random*. This means that if you know the AES key, you know the whole key stream no matter how random it appears to be.

Ensuring that you have a sufficiently random source of data is beyond the scope of this book. For our purposes, we will *assume* that `os.urandom()` can return acceptably random data for our needs. In production cryptography environments, you would need to analyze this far more carefully.

Randomness is so important that we will mention it more than once. In fact, we will return to it near the end of this very chapter.

Although AES-CTR is a stream cipher, we can still think about it one block at a time. To encrypt any given block of plaintext, generate the key stream for that block's index and XOR it with the (possibly partial) block. Expressed another way (where the subscript *k* indicates "encrypted with key *k*"):

$$C\left[n\right]=P\left[n\right]\oplus n_k.$$

That's mostly it! The only other slight twist is that we don't want to start with the same counter value every time. So, our IV, which we'll call our "nonce," is used as the starting counter value. To update our definition:

$$C\left[n\right]=P\left[n\right]\oplus\left(IV+n\right)_k.$$

XOR is a really versatile mathematical operation. You can think of it as "controlled bit-flipping": to compute $A \oplus B$, you march down their bits in tandem; when you encounter a 1 in B, you invert the corresponding bit in A, and when you encounter a 0 in

B, you leave that bit in *A* alone. Thinking of it that way, it's easy to see how doing that *twice* simply restores *A* to what it was before.

More formally, as discussed earlier, XOR is its own inverse: $(A \oplus B) \oplus B = A$. Since we created a stream of encrypted blocks by applying XOR to the appropriate value in the key stream, we simply do *exactly the same thing* to decrypt: apply XOR to the encrypted blocks and their corresponding keys:

$$P[n] = C[n] \oplus (IV + n)_k.$$

Of course, nothing happens if you merely XOR with 0 (since $A \oplus 0 = A$, which is where the inverse property comes from), so the keys in the stream need to be composed of random-looking bits, but that is exactly the type of key stream that AES produces.

Figure 3-7 provides a visual representation of AES-CTR operations.

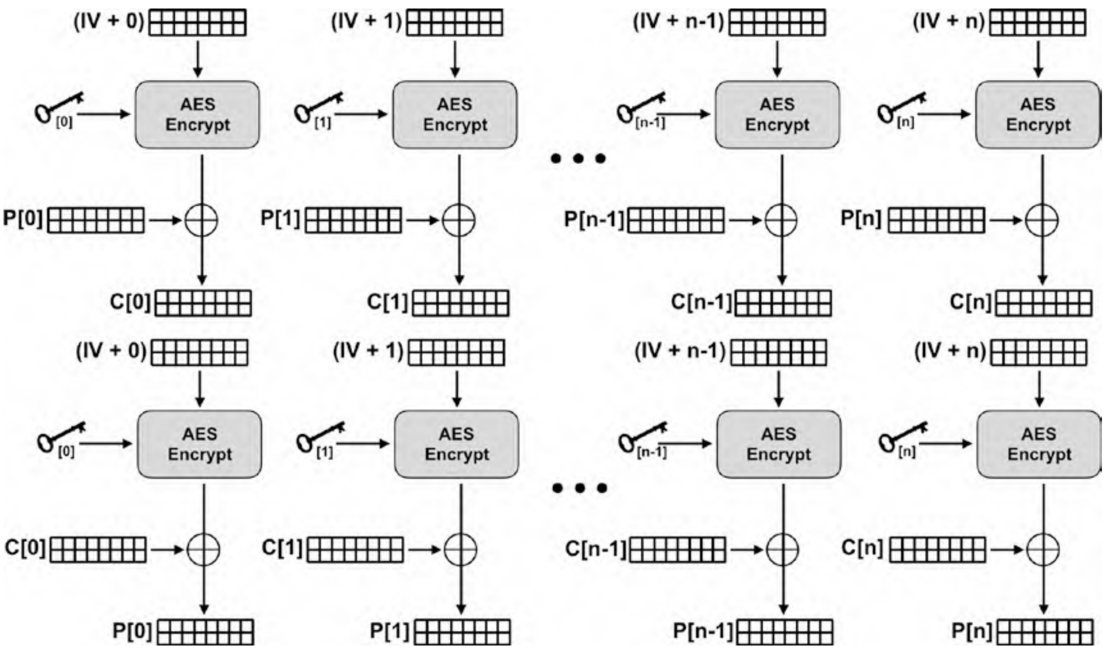

Figure 3-7. *Visual depictions of CTR encryption and decryption. Note that encryption and decryption are the same process!*

Happily, stream ciphers do not require padding! It is quite simple to only XOR a partial block, discarding the later parts of the key that aren't needed.

In general, this approach is much simpler. Padding goes away and blocks can again be encrypted independent of one another.

Let's see it in action in the cryptography module (Listing 3-9).

Listing 3-9. AES-CTR

```python
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import os

class EncryptionManager:
    def __init__(self):
        key = os.urandom(32)
        nonce = os.urandom(16)
        aes_context = Cipher(algorithms.AES(key),
                             modes.CTR(nonce),
                             backend=default_backend())
        self.encryptor = aes_context.encryptor()
        self.decryptor = aes_context.decryptor()

    def updateEncryptor(self, plaintext):
        return self.encryptor.update(plaintext)

    def finalizeEncryptor(self):
        return self.encryptor.finalize()

    def updateDecryptor(self, ciphertext):
        return self.decryptor.update(ciphertext)

    def finalizeDecryptor(self):
        return self.decryptor.finalize()

# Auto generate key/IV for encryption
manager = EncryptionManager()

```

```
plaintexts = [
    b"SHORT",
    b"MEDIUM MEDIUM MEDIUM",
    b"LONG LONG LONG LONG LONG LONG"
]

ciphertexts = []

for m in plaintexts:
    ciphertexts.append(manager.updateEncryptor(m))
ciphertexts.append(manager.finalizeEncryptor())

for c in ciphertexts:
    print("Recovered", manager.updateDecryptor(c))
print("Recovered", manager.finalizeDecryptor())
```

Because no padding is needed, the finalize methods are actually unnecessary except for "closing" the object. They are kept for symmetry and pedagogy.

How do you choose between CTR and CBC modes? In almost all circumstances, counter mode (CTR) is recommended.[12] Not only is it easier, but in some circumstances it is also more secure. As if that wasn't enough, counter mode is also easier to parallelize because keys in the key stream are computed from their index, not from a preceding computation.

[12] You can remember it because it also stands for "choose the right." You can even buy "CTR" rings as a friendly, constant, and, when twisted a bit to our purposes, *cryptographic* reminder.

Why even talk about CBC, then? At the very least it is still in wide use, so you will benefit from understanding it when you encounter it in the wild.

We will introduce other modes later in the book that build on counter mode to make something even better. For now, it is enough to understand the basic characteristics of CBC and CTR modes and how each one works to build a better algorithm from an underlying block cipher.

EXERCISE 3.13. WRITE A SIMPLE COUNTER MODE

As you did with CBC, create counter mode encryption from ECB mode. This should be even easier than it was with CBC. Generate the key stream by taking the IV block and encrypting it, then increasing the value of the IV block by one to generate the next block of key stream material. When finished, XOR the key stream with the plaintext. Decrypt in the same manner.

EXERCISE 3.14. PARALLEL COUNTER MODE

Extend your counter mode implementation to use a thread pool to generate the key stream in parallel. Remember that to generate a block of key stream, all that is required is the starting IV and which block of key stream is being generated (e.g., 0 for the first 16-byte block, 1 for the second 16-byte block, etc.). Start by creating a function that can generate *any* particular block of key stream, perhaps something like `keystream(IV, i)`. Next, parallelize the generation of a key stream up to *n* by dividing the counter sequence among independent processes any way you please, and have them all work on generating their key stream blocks independently.

Key and IV Management

As you have seen, having a library such as `cryptography` makes all kinds of encryption convenient and simple to use. Unfortunately, this simplicity can be deceptive and lead to mistakes; there are many ways to get it wrong. We have already touched briefly on one of them: reuse of keys or IVs.

That kind of mistake falls under the broader category of “Key and IV Management,” and doing it incorrectly is a common source of problems.

Important You must *never* reuse key and IV pairs. Doing so seriously compromises security and disappoints cryptography book authors. Just don't do it. Always use a new key/IV pair when encrypting anything.

Why don't you want to reuse a key and IV pair? For CBC, we already mentioned one of the potential problems: if you reuse a key and IV pair, you will get *predictable output for predictable headers*. Parts of your messages that you might be inclined not to think about at all, because they are boilerplate or contain hidden structure, will become a liability; adversaries can use predictable ciphertext to learn about your keys.

Think about an HTML page, for example. The first characters are often the same across multiple pages (e.g., `"<!DOCTYPE html>\n"`). If the first 16 bytes (an AES block) of HTML pages are the same and you encrypt them under the same key/IV pair, the ciphertext will be the same for each one. You have just leaked data to your enemy, and they can start to analyze your encrypted data for patterns.

If your web site has a large amount of static content or dynamic results that are identically generated, each encrypted page becomes uniquely identifiable. The enemy may not know what each page says, but they can determine the frequency of use and track which parties receive the same pages.

Reusing a key and IV in CBC mode is *bad*.

Reusing a key and IV in counter mode, on the other hand, is *much worse*. Because counter mode is a stream cipher, the plaintext is simply XORed with the key stream. If you happen to know the plaintext, you can *recover the key*: $K \oplus P \oplus P = K$!

"So what?" you might be thinking. "Who cares if they can get the key stream? If they already know the plaintext, why do we care?"

The problem is, under many circumstances an attacker might know some or all of the contents of one of your plaintext messages. If *other* messages are encrypted *with the same key stream*, the attacker can recover *those* messages too!

Bad, bad, bad.

Let's explore this idea a little further. Suppose that you buy something for $100.00 with a credit card at a store. Let's assume a simplified version of the world where the card reader sends a message to your bank to authorize the purchase protected by only AES-CTR encryption.

Imagine that the message to the bank from the credit card reader is XML that looks like this:

```
<XML>
  <CreditCardPurchase>
    <Merchant>Acme Inc</Merchant>
    <Buyer>John Smith</Buyer>
    <Date>01/01/2001</Date>
    <Amount>$100.00</Amount>
    <CCNumber>555-555-555-555</CCNumber
  </CreditCardPurchase>
</XML>
```

The store creates this message, encrypts it, and sends it to the bank. In order to communicate, the store and the bank must share a key. If the programmers who wrote the code were lazy and negligent, they may have created a system with a constant key and IV that are reused on every message, like what we find in Listing 3-10.

Listing 3-10. AES-CTR for a Store

```
# ACME generates a purchase message in their storefront.
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend

# WARNING! Never do this. Reusing a key/IV is irresponsible!
preshared_key = bytes.fromhex('00112233445566778899AABBCCDDEEFF')
preshared_iv = bytes.fromhex('00000000000000000000000000000000')

purchase_message = b"""
<XML>
  <CreditCardPurchase>
    <Merchant>Acme Inc</Merchant>
    <Buyer>John Smith</Buyer>
    <Date>01/01/2001</Date>
    <Amount>$100.00</Amount>
    <CCNumber>555-555-555-555</CCNumber
  </CreditCardPurchase>
</XML>
"""

aesContext = Cipher(algorithms.AES(preshared_key),
                    modes.CTR(preshared_iv),
                    backend=default_backend())
encryptor = aesContext.encryptor()
encrypted_message = encryptor.update(purchase_message)
```

For simplicity, the purchase message was included in the preceding code. Feel free to change it to accept a file or command-line flags that set the buyer's name, purchase price, and so forth. You probably ought to also write the encrypted message to a file.

Back to our scenario, if you are trying to crack this system, you can spend $100.00 at this store, then tap the line and intercept the purchase message transmitted to the bank. If you do this, how much of the plaintext message do you know? You know *all* of it! You know who made the purchase, you know the amount of the purchase, you know the date, and you know your own credit card number.

That means that you can recreate the plaintext message, XOR it with the ciphertext, and recover keystream material. Because the merchant is reusing the same key and IV for the next customer, you can trivially decrypt the message and read the contents. Oops. We feel a news story about a data breach coming on.

EXERCISE 3.15. RIDING THE KEYSTREAM

Put into practice this keystream-stealing attack. That is, encrypt two different purchase messages using the same key and IV. "Intercept" one of the two messages and XOR the ciphertext contents with the known plaintext. This will give you a keystream. Next, XOR the keystream with the other message to recover that message's plaintext. The message sizes may be a little different, but if you're short some keystream bytes, recover what you can.

Even if the attacker does not know *any* of the plaintext and cannot recover a keystream, he or she can still take advantage of messages encrypted with the same key and IV pair. If you have two messages encrypted with the same keystream, you can do the following trick (where *K* is the key stream):

$$
\begin{aligned}
c_1 &= m_1 \oplus K \\
c_2 &= m_2 \oplus K \\
c_1 \oplus c_2 &= (m_1 \oplus K) \oplus (m_2 \oplus K) \\
c_1 \oplus c_2 &= m_1 \oplus m_2 \oplus K \oplus K \\
c_1 \oplus c_2 &= m_1 \oplus m_2
\end{aligned}
$$

What do you get out of having the XOR of the two plaintext messages? Is that readable? It depends. Because plaintext messages often have structure, private data is often extractable or guessable. Take these made-up purchase messages from our example. If you XORed two such messages together, what could you learn?

First of all, any parts that overlap exactly simply reduce to 0. Instantly, you know where the messages are the same and where they diverge. If the attacker were lucky enough that two messages had the same length of name for the buyer, the amount fields would line up as well. This field yields a lot of information when the two are XORed together because there are so few legal characters for this field ("0"–"9" and "."). The XOR of the ASCII characters for the digits leaves open only a few possibilities.

For example, there are only two pairs of digits for which the XOR of their ASCII values is 15. These are "7" and "8" (ASCII values 55 and 56) and "6" and "9" (ASCII values 54 and 57). So, if we know that we have the XOR of two purchase amount field digits and the XOR value is 15, then the two messages each have one of these two pairs of numbers. That's only four possibilities, which will not be very difficult for an attacker to figure out under most circumstances.

You might be surprised how often this vulnerability can show up if you're not careful. One simple example is full-duplex messages. If you have two parties that want to send encrypted messages to each other, they must not use the same key and IV to encrypt each side of the connection. Each side's encryption must be independent of the other. If you think about how CBC and CTR mode work, this will be pretty obvious. If you are going to write messages in both directions, each side needs a separate read key and write key.[13] The read key of the first party will be the write key of the second and vice versa. This way, different messages will not be written under the same key and IV pair.

EXERCISE 3.16. SIFTING THROUGH XOR

XOR together some plaintext messages and look around for patterns and readable data. There's no need to use any encryption for this, just take some regular, human-readable messages and XOR the bytes. Try human-readable strings, XML, JSON, and other formats. You may not find a lot that is instantly decipherable, but it's a fun exercise.

Exploiting Malleability

Some aspects of cryptography are unintuitive at first. For example, an enemy can fail to *read* a confidential message while still being able to *change* it in meaningful, deceptive ways. In this section, we will experiment with altering encrypted messages while not being able to read them.

Counter mode is a really good encryption mode for all of the reasons described earlier. At the risk of being too repetitive, however, it only guarantees *confidentiality*. In fact, because it is a stream cipher, it is trivial to change a small part of the message without changing the rest of it. In counter mode, for example, if an attacker modifies one byte of the ciphertext, it only affects the corresponding byte of plaintext. While that one byte of plaintext will not decrypt correctly, the remaining bytes will remain intact.

Cipher block chaining mode is different because a change to a single byte of ciphertext will affect all subsequent blocks.

[13] Technically, they could use the same key, provided the IVs were different. However, there are a number of ways in practice that IVs might intentionally or accidentally overlap, so it is typically recommended to use different keys and not rely on having different IVs.

EXERCISE 3.17. VISUALIZING CIPHERTEXT CHANGES

To better understand the difference between counter mode and cipher block chaining mode, go back to the image encryption utility you wrote previously. Modify it to first encrypt and then decrypt the image, using either AES-CBC or AES-CTR as the mode. After decryption, the original image should be completely restored.

Now introduce an error into the ciphertext and decrypt the modified bytes. Try, for example, picking the byte right in the middle of the encrypted image data and setting it to 0. After corrupting the data, call the decryption function and view the restored image. How much of a difference did the edit make with CTR? How much of a difference did the edit make with CBC?

HINT: If you can't see anything, try an all-white image. If you still can't see it, change 50 bytes or so to figure out where the changes are happening. Once you find where the changes are happening, go back to changing a single byte to view the differences between CTR and CBC. Can you explain what's happening?

To illustrate this concept of malleability, we are going to let our attacker know some of the plaintext of an encrypted message. This knowledge is going to allow them to change the message en route. What's different this time around is that this vulnerability is *not* dependent on a reused keystream.

If an attacker knows the plaintext behind a keystream-enciphered message, it is easy to extract the keystream from the ciphertext. If the keystream is reused, the attacker can decrypt all messages that used it. Even if it is *not* reused, the attacker can *alter* a message with known plaintext.

Let's revisit our encrypted purchase messages. Suppose that Acme's competitor, Evil LLC, wants to redirect this payment to themselves. They have a tap on the network connection coming out of Acme's store and can intercept and modify the message. When an encrypted form of this message comes along, even though they don't have the key and cannot decrypt it, they can strip out the original message parts that are known and replace them with their own chosen parts.

The part that Evil LLC wants to change is this part:

```
<XML>
  <CreditCardPurchase>
    <Merchant>Acme Inc</Merchant>
```

That data is known and fixed in every payment message. To obtain the keystream, all Evil LLC has to do is XOR this data with the ciphertext. Once this part is XORed, they have the keystream for this many bytes. Then, they create their modified message:

```
<XML>
  <CreditCardPurchase>
    <Merchant>Evil LLC</Merchant>
```

This message has the exact same size as the true message. Because AES-CTR is so malleable, it is easy to XOR this partial message with the extracted keystream and join it to the rest of the still-encrypted message. This process is illustrated in Figure 3-8.

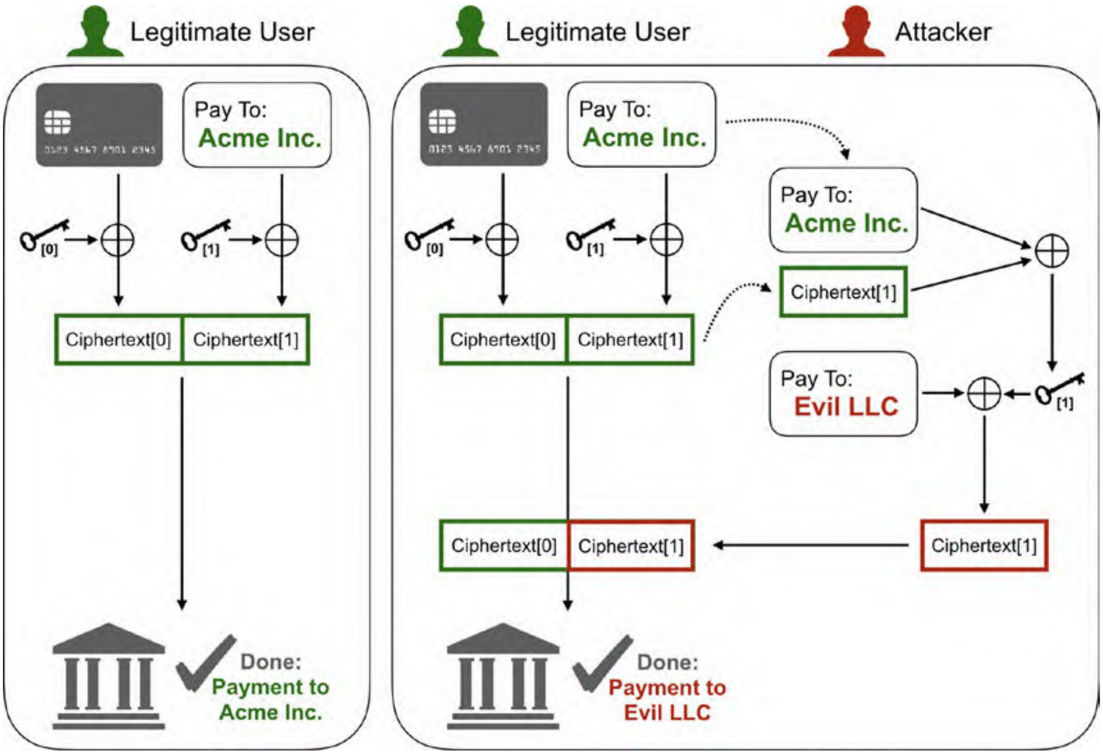

Figure 3-8. *If an attacker knows the plaintext in CTR mode ciphertext, she can extract the keystream to encrypt her own evil message!*

EXERCISE 3.18. EMBRACING EVIL

You work for (or own!) Evil LLC. Time to steal some payments from Acme. Start with one of the encrypted payment messages you created in the earlier exercises. Calculate the size of the header up through the identification of the merchant and extract that many bytes of the encrypted data. XOR the plaintext header with the ciphertext header to get the *keystream*. Once you have this, XOR the extracted keystream with a header identifying Evil LLC as the merchant. This is the "evil" ciphertext. Copy it over the bytes of the encrypted file to create a new payment message identifying your company as the recipient. Prove that it works by decrypting the modified file.

The key lesson here is that encryption is insufficient to protect data by itself. In subsequent chapters, we will use message authentication codes, authenticated encryption, and digital signatures to ensure that data cannot be altered without disrupting communications.

Gaze into the Padding

While CBC mode is less susceptible to alteration than counter mode, it is by no means perfect in that regard. In fact, it is CBC's malleability that made one of the early versions of SSL vulnerable. Remember that CBC mode is a block-based mode and requires padding. An interesting error in the padding specification and the malleability of AES-CBC enabled attackers to execute a "padding oracle attack" and decrypt confidential data.

Let's create that attack right now. It's extremely interesting and educational.

For this little exercise, you will need to write your own padding functions; the ones in the `cryptography` module are too secure. Your functions will follow the very broken SSL 3.0 specification (we'll talk about SSL/TLS more in the last chapter). Basically, *N*–1 bytes of *anything* followed by a single byte that indicates the total length of the padding. Because padding was always required in that specification, it would be added even if the plaintext was a multiple of the block size. This will be important later.

Listing 3-11. SSLv3 Padding

```
def sslv3Pad(msg):
    padNeeded = (16 - (len(msg) % 16)) - 1
    padding = padNeeded.to_bytes(padNeeded+1, "big")
    return msg+padding

def sslv3Unpad(padded_msg):
    paddingLen = padded_msg[-1] + 1
    return padded_msg[:-paddingLen]
```

Let's talk about what we have so far (Listing 3-11). The padding bytes in this scheme are completely ignored *except for the last byte*. It doesn't matter what the bytes are, so long as the last byte is correct. Padding goes at the end of a message, right? Guess which part of a CBC message is the most malleable.

The reason that the last part of the CBC message is more malleable is that it has no impact on any subsequent blocks. It can be changed without messing up anything else. Recall that CBC decryption starts out the same for every single block no matter where it is. The ciphertext block is decrypted by AES with the key. It's only after decryption that it is XORed with the ciphertext from the previous block.

This means that you could substitute *any* block from the CBC chain at the very end of the chain. It will get decrypted at the end just like it would in the middle or the beginning. After decryption, it is XORed with the ciphertext from the previous block.

How is this helpful? Well, suppose that we are fortunate enough to have the original plaintext message be a multiple of 16 bytes long, the AES block length. Because we're using a padding scheme that *always* uses padding, there will be a full block of padding at the end. Since we don't care what bytes are in the padding except for the last one, we can correctly recover the entire message, even if we replace the last block, so long as the very last byte decodes to *15* (the padding length when there is a full block of padding).

Explained another way, when there is a full block of padding at the end, 15 of the 16 bytes are completely ignored. It doesn't matter what they are. If we're going to try to "fool" the decryption, this is a great place to do it, because we only have to get *one* byte correct!

This small change, only caring about the value of the last byte, changes everything! It reduces brute-force guessing to something reasonable. Normally, if you wanted to "guess" a correct AES block, you would have to try all possible combinations of all 16 bytes. You might recall from previous discussion that this works out to a very big number and it is impossible to try every combination for all practical purposes.

But now that we only care about the last byte, we only need to correctly guess *one* byte of data. To repeat, so long as the last byte decrypts to 15, our padding will be "correct." One byte of data has 256 possible values, so if our last byte is randomly selected, then 1 out of 256 times it will correctly decrypt to 15!

You might protest that the data *isn't* random. We are trying to decrypt a specific byte. Very true! But remember that in CBC we XOR the real plaintext with the ciphertext of the previous block! The ciphertext, at least for our purposes here, behaves like random data. For any given key/IV pair, the last byte of ciphertext that will be XORed with our plaintext byte has an equal chance of being any of the 256 possible 1-byte values. If we are lucky, the "random" byte of ciphertext XORed with our plaintext byte will be 15!

If the padding is accepted and decrypts to 15, we can use our knowledge of the previous ciphertext block to get the true plaintext byte.

Actually, recovering the plaintext byte is a little trick and requires that we think through CBC decryption carefully. Remember that the last block of plaintext (e.g., the true padding in the original message) was XORed with the ciphertext from the second-to-last block. This intermediate data was encrypted by the AES algorithm. So, working backward, if we overwrite the final ciphertext block, the CBC operation will first run this block through the AES decryption operation to produce an intermediate value that is then XORed with the preceding ciphertext. If this is difficult to follow, refer back to Figure 3-5.

If the padding is accepted (e.g., the last byte is 15), we know that the *last byte* of the intermediate value decrypted by AES is the XOR of 15 and the last byte of the *previous ciphertext block*. We, of course, have the ciphertext. Now, even without the AES key, we can simply compute the intermediate byte directly (e.g., by taking the XOR of 15 and the last byte of the second-to-last ciphertext block).

But the intermediate value isn't the plaintext byte. Remember, we are decrypting an earlier ciphertext block. That ciphertext block is the AES encryption of the real plaintext XORed with the actual preceding ciphertext (or the IV if it's the first plaintext block). So, when we recover the intermediate last byte, we still have to remove that mixed-in data with an appropriate XOR.

Let's work on putting this into code. First, we need to define our "oracle." In real life, the oracle was the SSLv3 server. If you sent it a message with bad padding, it would send you an error message that the padding was bad. That knowledge is all that is necessary to pull off this attack. For our code in Listing 3-12, we will just have an `accept()` method in an `Oracle` class that indicates whether the padding is valid, performing the same purpose as the server.

Listing 3-12. SSLv3 Padding Oracle

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes
from cryptography.hazmat.backends import default_backend

class Oracle:
    def __init__(self, key, iv):
        self.key = key
        self.iv = iv

    def accept(self, ciphertext):
        aesCipher = Cipher(algorithms.AES(self.key),
                           modes.CBC(self.iv),
                           backend=default_backend())
        decryptor = aesCipher.decryptor()
        plaintext = decryptor.update(ciphertext)
        plaintext += decryptor.finalize()
        return plaintext[-1] == 15
```

This might seem a little weird: we have the key and are using it to create the oracle. Just remember: we're simulating a vulnerable remote server, which *would* have its own key. The attack we write below will proceed without knowledge of the key used here.

Once we have the oracle, it's a pretty easy function to see if we can get lucky and decode the last byte of an arbitrary block in the ciphertext, as in Listing 3-13.

Listing 3-13. Lucky SSLv3 Padding Byte

```
# Partial Listing: Some Assembly Required

# This function assumes that the last ciphertext block is a full
# block of SSLV3 padding
def lucky_get_one_byte(iv, ciphertext, block_number, oracle):
    block_start = block_number * 16
    block_end = block_start + 16
    block = ciphertext[block_start: block_end]

    # Copy the block over the last block.
    mod_ciphertext = ciphertext[:-16] + block
    if not oracle.accept(mod_ciphertext):
        return False, None

    # This is valid! Let's get the byte!
    # We first need the byte decrypted from the block.
    # It was XORed with second to last block, so
    # byte = 15 XOR (last byte of second-to-last block).
    second_to_last = ciphertext[-32:-16]
    intermediate = second_to_last[-1]^15

    # We still have to XOR it with its *real*
    # preceding block in order to get the true value.
    if block_number == 0:
        prev_block = iv
    else:
        prev_block = ciphertext[block_start-16: block_start]

    return True, intermediate ^ prev_block[-1]
```

To repeat: we are counting on the penultimate (second-to-last) block being lucky! As shown in Figure 3-9, we have to be lucky enough that the last byte of the penultimate block will just happen to XOR with our intermediate byte to be 15. This luck that we are counting on is dependent on the key and IV chosen. Once again, for any given key/IV pair, there is a 1-in-256 chance that the penultimate block will "accidentally" XOR with our intermediate plaintext block to give us 15.

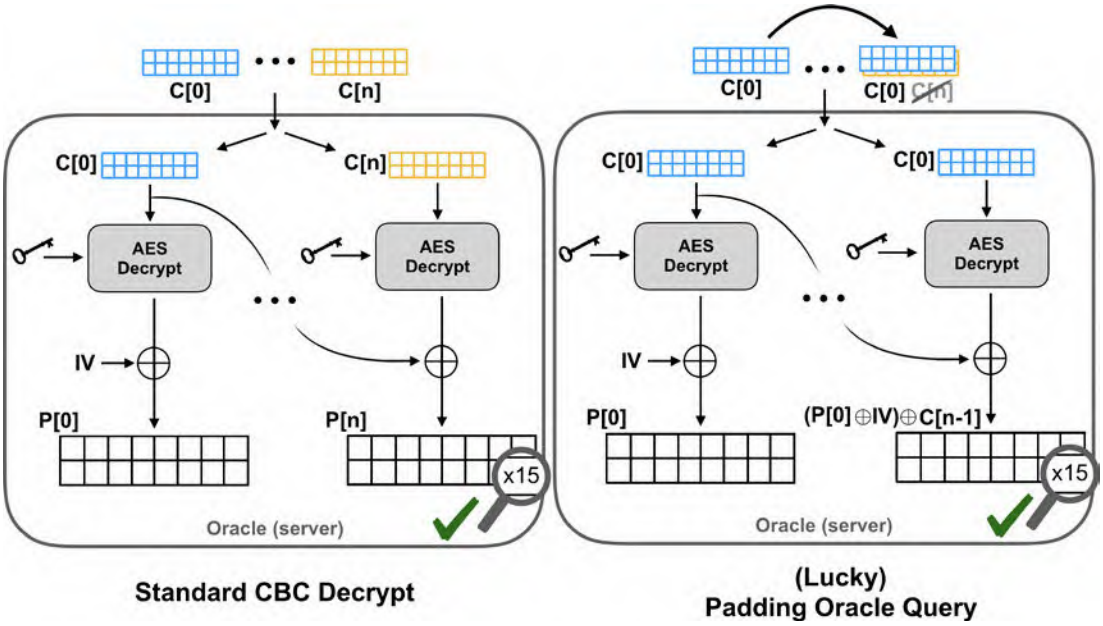

Figure 3-9. *If the first 15 bytes of the padding block are ignored, we can substitute in the second to last block and see if the oracle tells us the padding is correct. If so, we can figure out the last byte in the previous block.*

Is that really all that useful? In the first place, we have to be lucky enough to have a full block of padding. In the second place, we only have a 1-in-256 chance of decoding *a single byte*. That doesn't seem terribly helpful.

Or does it?

Again, cryptography can be very counter-intuitive. Computers don't behave like we would expect them to, and that's where we get into trouble.

While SSLV3 was busy protecting web traffic, it turned out there were a number of ways that a malicious advertisement could generate traffic to an SSL-encrypted web site. But because that advertisement was generating the traffic, its authors could control how long the encrypted message was. Thus, if the attacker was trying to decrypt an encrypted cookie, triggering a GET request of various lengths could control how long the overall message was.

Getting the full block of padding in this case really isn't very difficult as the malicious requester could put arbitrary data in the GET request.

And it is nothing for a computer to make 256 requests over a network. Note that, in the SSLV3 context, the client and server are going to use different keys with every connection (for good reason, as we have seen!). This means that on each connection, the ciphertext will be different! So, if the attacker sends 256 requests, the penultimate block will be different each time, providing a new opportunity to be lucky and have the right "random" number that will provide the needed 15.

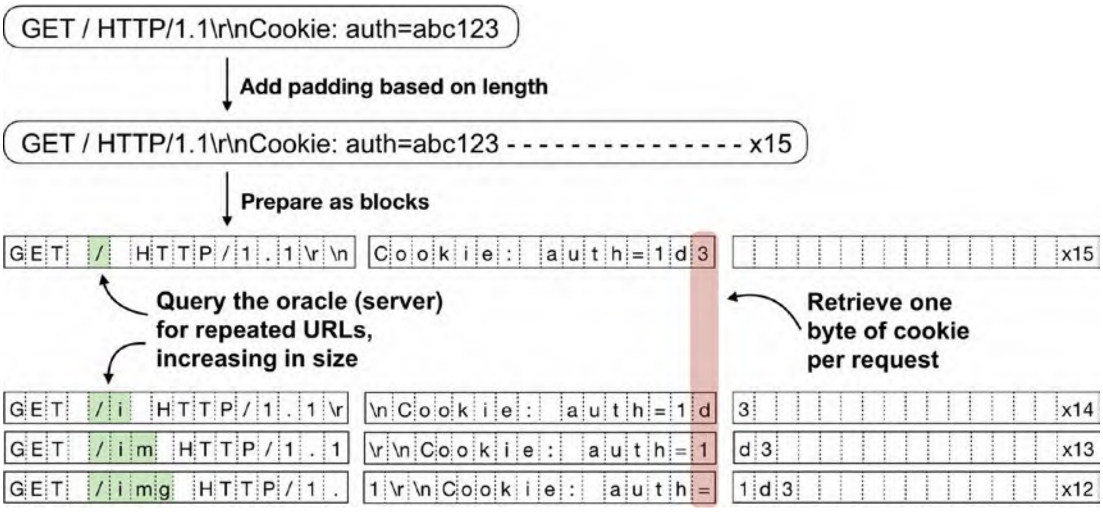
GET / HTTP/1.1\r\nCookie: auth=abc123
Add padding based on length
GET / HTTP/1.1\r\nCookie: auth=abc123 - - - - - - - - - - - - - - - x15
Prepare as blocks
G E T / H T T P / 1 . 1 \r \n
C o o k i e : a u t h = 1 d 3
x15
Query the oracle (server) for repeated URLs, increasing in size
Retrieve one byte of cookie per request
G E T /i H T T P / 1 . 1 \r
\n C o o k i e : a u t h = 1 d
3 x14
G E T /i m H T T P / 1 . 1
\r \n C o o k i e : a u t h = 1
d 3 x13
G E T /i m g H T T P / 1 .
1 \r \n C o o k i e : a u t h =
1 d 3 x12

Figure 3-10. *In order to decrypt a byte that matters, an attacker controls the GET request size so that the cookie is in the right spot. This requires the ability to insert arbitrary requests such as an advertiser within a TLS-secured context.*

It's still just one byte, right? As illustrated in Figure 3-10, the attacker can control the length of the message. Once one byte is decoded, it's pretty straightforward to increase the message length by 1 by inserting a byte earlier in the message, pushing the new byte into the last slot of the arbitrary block. Another 256 tries and that second byte will be decoded too! Wash, rinse, and repeat!

EXERCISE 3.19. RESISTANCE IS FUTILE

Finish the code for the padding oracle attack. We've given you the major pieces, but it will still take some work to put everything together. We will do a few things to try and simplify as much as possible. First, pick a message that is exactly a multiple of 16 bytes in length (the AES block size) and create a fixed padding to append. The fixed padding can be any 16 bytes as long as the last byte is 15 (that's the whole point of the exercise, right?). Encrypt this message and pass it to the oracle to make sure that code is working.

Next, test recovering the last byte of the first block of the message. In a loop, create a new key and IV pair (and a new oracle with these values), encrypt the message, and call the `lucky_get_one_byte()` function, setting block number to 0. Repeat the loop until this function succeeds and verify that the recovered byte is correct. Note that, in Python, an individual byte isn't treated as a byte type but is converted to an integer.

The last step in order to decode the entire message is to be able to make any byte the last byte of a block. Again, for simplicity, keep the message being encrypted a perfect multiple of 16. To push any byte to the end of a block, add some extra bytes at the beginning and cut off an equal number at the end. You can now recover the entire message one byte at a time!

EXERCISE 3.20. STATISTICS ARE ALSO FUTILE

Instrument your padding oracle attack in the previous exercise to calculate how many guesses it took to fully decrypt the entire message and calculate an average number of tries per byte. In theory, it should work out to about 256 tries per byte. But you're probably working with such small numbers that it will vary widely. In our tests on a 96-byte message, our averages varied between about 220 guesses per byte and 290 guesses per byte.

Once again, encryption is about confidentiality, and confidentiality is simply not enough to solve all security problems. In subsequent chapters we will learn how to combine confidentiality and integrity to solve a larger class of problems.

Weak Keys, Bad Management

To conclude this chapter, let's briefly discuss *keys*. Hopefully, it has already become very clear to you how important keys are.

In almost all cryptographic systems, key management is the hardest part. It can be difficult to generate good keys, to share keys, and to manage keys afterward (e.g., keeping them secret, updating them, or revoking them). For now, we'll focus on key generation.

Keys must be drawn from good sources of randomness. We mentioned randomness once already in this chapter in a brief aside, but let's take a second look. For example, the following code is *really wrong*.

```
import random
key = random.getrandbits(16, "big")
```

The random package is a *pseudo-random* number generator and not even a good one at that. Pseudo-random generators are *deterministic*, generating numbers that appear random to humans, but are always the same given a known seed value. Default seeds used to be based on the system time. This may seem reasonable, but it means that if the attacker can guess when the random number generator was seeded, they can completely predict all the random numbers produced. About the only way to make this worse is to hard-code the key or the seed (which is effectively the same thing).

```
import random

# Set the random number generator seed to 0.
r = random.Random(0)
key = r.getrandbits(16, "big")
```

This code will produce the same "random" numbers on every run of the program. This can sometimes be useful for testing, but you must not leave it in production code!

Although Python's default seeding is no longer quite so predictable, it is not suitable for generating secrets like passwords. Instead, always pull from `os.urandom()` or, if using Python 3.6 or later, `secrets.SystemRandom()`. Under most circumstances, this is enough randomness. If you need something stronger, you might need to use different hardware and should consult an expert cryptographer.

In some deployments, a key is not pulled from random numbers. Instead, it is derived from a password. If you are going to derive a key from a password, the password needs to be very secure! In the previous chapter, you learned about brute-force attacks and all of those lessons apply here.

Let's get a feel for the difference in difficulty of guessing a key in these scenarios. How long would it take to try every possible 128-bit (random) key? How many tries is that?

There are 2^{128} different 128-bit keys. That's this many different keys:

340,282,366,920,938,463,463,374,607,431,768,211,456.

If your key is derived from a five-digit pin number, though, you have reduced it to 99,999! It's true that very few passwords will be as hard to brute force as a truly random 128-bit key. After all, you'd need to have a password composed of about 20 random characters to require the same kind of brute-force effort as a 128-bit key. But still, 99,999 is just begging a computer to accept your challenge. You can do better than that!

As a reminder, there are proven algorithms for deriving a key from a password. Make sure you use a good one. In the previous chapter, we used scrypt. There are others that some people feel are even better (such as bcrypt or Argon2). What makes a good derivation function? One characteristic is how long it takes. If someone picks a weak password (e.g., "puppy1"), it won't take the attacker long to figure it out. It might be possible, however, to make it take too long if the derivation function is slow.

In short, don't bother using a good cipher with a bad key. Make sure that your keys are securely generated and adequately resistant to abuse by a determined adversary.

EXERCISE 3.21. PREDICTING TIME-BASED RANDOMNESS

Write an AES encryption program (or modify one of the others you've written for this chapter) that uses the Python random number generator to generate keys. Use the seed method to explicitly configure the generator based on the current time using `time.time()` rounded to the nearest second. Then use this generator to create a key and encrypt some data. Write a separate program that takes the encrypted data as input and tries to guess the key. It should take a minimum time and a maximum time as a range and try iterating between these two points as seed values for random.

Other Encryption Algorithms

In this chapter, we have focused exclusively on AES encryption. There are good reasons for this. AES is the most popular symmetric cipher currently in use. It's used in network communications as well as storing data on disk. And as we will see in Chapter 7, it is the basis for several advanced AEAD (authenticated encryption with associated data).

However, there are other symmetric key ciphers that can be used. Here are a few that are supported by the `cryptography` library:

- Camellia
- ChaCha20
- TripleDES
- CAST5
- SEED

Even though we are always encouraging you to use a well-tested, well-respected third-party library, be aware that libraries often include support for less desirable algorithms for legacy support. In this list of algorithms supported by `cryptography`, a few ciphers are already known to be insecure and are being phased out. For example, while DES is not included in the `cryptography` library's ciphers (GOOD! DES is VERY BAD!), the module does include 3DES (TripleDES). While 3DES is not as broken as DES, it should be retired ASAP. CAST5 fits in this same category.

Another cipher supported by `cryptography` is Blowfish. This algorithm is also not recommended for use, and its stronger successor, Twofish, is not available in the current `cryptography` implementation.

finalize()

This chapter covered a *lot* of material, and we barely scratched the surface. Perhaps the most important principle that you can take away from this chapter is that cryptography is usually far more complicated than perhaps it first seems. The different modes of operation we reviewed have different strengths and weaknesses, some of which we explored by example. We found that even how we approach the APIs to cryptographic operations can have a significant impact on security.

Hopefully, this lesson reinforced the YANAC principle (You Are Not A Cryptographer… yet!). Please remember that these exercises are introductory and educational. Please do not go copying this code into production and don't use the introductory knowledge you have gained to write security-critical operations. Do you really want to risk people's personal information, financial information, or other sensitive data on your newly developed skills?

At the same time, after just one chapter on encryption, you have a broader view of what that word even means. The next time you hear "protected by AES 128-bit encryption," you might wonder whether they're using CTR, CBC, or (heaven forbid!) ECB mode. You might also wonder if they are using their encryption correctly because you already have experienced some of the ways (often unexpected) that symmetric encryption can be broken.

Yes, you've taken your first steps into a cryptography world. Are you ready to take a few more? Then let's talk about *asymmetric* encryption!

CHAPTER 4

Asymmetric Encryption: Public/Private Keys

Asymmetric encryption is one of the most important advances in cryptographic security ever made. It underpins all of the security on the Web, in your Wi-Fi connections, and in secure email and other communication of all kinds. It is ubiquitous, but it is also subtle and easy to implement or use incorrectly, and a lack of correctness means a sometimes drastic reduction in security.

Perhaps you've heard of “public keys,” “public key infrastructure,” and/or “public key encryption.” There are actually multiple operations within asymmetric cryptography and a number of different algorithms. Within this chapter, we are going to focus exclusively on *asymmetric encryption* and specifically using an algorithm known as *RSA*. We are going to leave other asymmetric operations, such as signatures and key exchange, for later chapters.

RSA encryption is, in fact, almost completely obsolete. Why study it? Because RSA is one of the classic asymmetric algorithms and does a good job, in our opinion anyway, of introducing some core concepts that will be helpful when learning about more modern approaches.

A Tale of Two Keys

The East Antarctica Truth-Spying Agency (EATSA) has a new mission for Alice and Bob. Bob is to remain behind in East Antarctica (EA) as Alice's handler and Alice is to get an undercover position in the West Antarctica Government Greasy Spoon (WAGGS). Alice will report back to Bob what the West Antarctica (WA) politicians are eating. EATSA plans to blackmail these politicians over how much hot food they are eating, while their constituents are stuck eating frozen dinners.

S. J. Nielson and C. K. Monson, *Practical Cryptography in Python*,
https://doi.org/10.1007/978-1-4842-4900-0_4

EATSA, however, is concerned about compromised communications. If Alice is captured with a *symmetric* key, the West Antarctica Central Knights Office (WACKO) will be able to use it to decrypt any messages they have intercepted from her to EATSA. That would ruin the entire plan!

EATSA decides to implement a new technology they've been hearing about: asymmetric encryption. Their collective minds are blown when they find out that there are encryption schemes with *two* keys: what is encrypted with one key *can only be decrypted by the other*!

Using this new technology, Bob can send Alice into the field with just one of the two keys (the "public" key). Alice will be able to encrypt messages back to Bob that *not even she can decrypt*! Only Bob, safe within EA territory and in possession of the corresponding "private" key, can decrypt the messages. That sounds perfect—if her key is compromised, it will at least not allow her captors to decrypt what she has written, which is strictly better than before.[1] What could go wrong?

To finish cooking up this scheme, EATSA chooses to use RSA encryption, an asymmetric algorithm that uses very large integers as both keys and messages, and the "modular exponentiation" as the primary mathematical operator for encryption and decryption. The algorithm is simple to understand and, with modern programming languages, relatively easy to implement. It looks in all ways to be the perfect recipe for culinary subterfuge.

Getting Keyed Up

Generating keys in RSA is a little bit tricky, as it requires finding two very large integers with a high likelihood of being *co-prime*. That looked like a lot of math to the agents of EATSA, so they opted to just use existing libraries to do that part. Listing 4-1 shows the package they pulled into Python 3 and the code they wrote that makes use of it.

[1] They can still send fake messages pretending to be her, but that was possible with symmetric encryption as well.

Listing 4-1. RSA Key Generation

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

# Generate a private key.
private_key = rsa.generate_private_key(
     public_exponent=65537,
     key_size=2048,
     backend=default_backend()
)

# Extract the public key from the private key.
public_key = private_key.public_key()

# Convert the private key into bytes. We won't encrypt it this time.
private_key_bytes = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.TraditionalOpenSSL,
    encryption_algorithm=serialization.NoEncryption()
)

# Convert the public key into bytes.
public_key_bytes = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)

# Convert the private key bytes back to a key.
# Because there is no encryption of the key, there is no password.
private_key = serialization.load_pem_private_key(
    private_key_bytes,
    backend=default_backend(),
    password=None)

```

```
public_key = serialization.load_pem_public_key(
    public_key_bytes,
    backend=default_backend())
```

That's not too bad, once you know how it's used. This pattern is the same for any private/public key generation, so even though there are a few constants in there with long names, it definitely seemed like the library was making this easier for EATSA.

See how everything hinges on the private key in RSA? The public key is derived from it. While either key can be used to encrypt (and the other can be used to decrypt), the private key is special because of this property. RSA keys are not only asymmetric because one encrypts and the other decrypts, they are also asymmetric because you can derive an RSA public key from the private key, but not the other way around.

The `private_bytes` and `public_bytes` methods convert large integer keys into bytes that are in a standard network- and disk-ready encoding called a PEM. The corresponding serialization "load" methods can be used to decode these after reading those bytes from disk so that they look like keys again to the encryption and decryption algorithms.

It is possible (and a very good idea) to encrypt the *private key itself*, but we opted not to do that here, which is why no password is used.

RSA Done Wrong: Part One

Alice and Bob are going to help us learn about RSA largely by exploring all the ways to use RSA incorrectly.

The actual encryption and decryption parts looked pretty simple to EATSA, and every library they looked at seemed to have a lot of unnecessary extra stuff making it harder to understand and even (gasp) slowing it down. Not having been taught the YANAC principle, they decided to implement encryption and decryption on their own. Rather than using the third-party library as written, they opted to omit *padding*. This results in a very "raw" or basic form of RSA that will be useful to us in learning about internals even though the results will be very broken.

Warning: Do Not Roll Your Own Encryption

Once again, implementing your own RSA encryption/decryption, rather than using a library, is not a good idea *at all*. Using RSA without padding is especially unsafe and insecure for numerous reasons, just a few of which we'll explore in this section. Although we will be writing our own RSA functions here for educational purposes, **do not under any circumstances use this code for real communication**.

Here is the math for encryption, where c is the ciphertext, m is the message, and the remaining parameters form the public and private keys, to be explained later:

$$c \equiv m^e \pmod{n} \tag{4.1}$$

Similarly, here it is for decryption:

$$m \equiv c^d \pmod{n} \tag{4.2}$$

That doesn't seem too bad, right? Modular exponentiation is a pretty standard operation in large integer math libraries,[2] so there really isn't much to this.

If you're new to this, don't be thrown off by $\equiv$. For simplicity, you can usually just think about it as an equal sign.

The operations in (4.1) and (4.2) can be written concisely in Python using `gmpy2`, a large number math library. The `powmod` function performs the necessary modular exponentiation operation, as shown in Listing 4-2.

Listing 4-2. GMPY2

```
#### DANGER ####
# The following RSA encryption and decryption is
# completely unsafe and terribly broken. DO NOT USE
```

[2] It certainly became popular after PKI was invented.

```
# for anything other than the practice exercise
################
def simple_rsa_encrypt(m, publickey):
    # Public_numbers returns a data structure with the 'e' and 'n' parameters.
    numbers = publickey.public_numbers()

    # Encryption is(m^e) % n.
    return gmpy2.powmod(m, numbers.e, numbers.n)

def simple_rsa_decrypt(c, privatekey):
    # Private_numbers returns a data structure with the 'd' and 'n' parameters.
    numbers = privatekey.private_numbers()

    # Decryption is(c^d) % n.
    return gmpy2.powmod(c, numbers.d, numbers.public_numbers.n)
#### DANGER ####
```

As mentioned before, and perhaps more obvious now, RSA operates on integers, not message bytes. How do we convert messages into integers? Python makes this convenient because its int type has to_bytes and from_bytes methods. Let's make them a little nicer to use in Listing 4-3.

Listing 4-3. Integer/Byte Conversion

```
def int_to_bytes(i):
    # i might be a gmpy2 big integer; convert back to a Python int
    i = int(i)
    return i.to_bytes((i.bit_length()+7)//8, byteorder='big')

def bytes_to_int(b):
    return int.from_bytes(b, byteorder='big')
```

Important Because RSA works on integers, not bytes, the default implementation loses leading zeros. As far as integers are concerned, 01 and 1 are the same number. If your byte sequence begins with any number of zeros, they will not survive encryption/decryption. For our example, we are sending text, so it won't ever be a problem. For binary data transmissions, however, it could be. This problem will be solved with padding.

EATSA now has all of the necessary pieces to create a simple RSA encryption/decryption application. Before looking at their code in Listing 4-4, try creating your own version.

Listing 4-4. RSA Done Simply

```
# FOR TRAINING USE ONLY! DO NOT USE THIS FOR REAL CRYPTOGRAPHY

import gmpy2, os, binascii
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

#### DANGER ####
# The following RSA encryption and decryption is
# completely unsafe and terribly broken. DO NOT USE
# for anything other than the practice exercise
################
def simple_rsa_encrypt(m, publickey):
    numbers = publickey.public_numbers()
    return gmpy2.powmod(m, numbers.e, numbers.n)

def simple_rsa_decrypt(c, privatekey):
    numbers = privatekey.private_numbers()
    return gmpy2.powmod(c, numbers.d, numbers.public_numbers.n)
#### DANGER ####

```

```
def int_to_bytes(i):
    # i might be a gmpy2 big integer; convert back to a Python int
    i = int(i)
    return i.to_bytes((i.bit_length()+7)//8, byteorder='big')

def bytes_to_int(b):
    return int.from_bytes(b, byteorder='big')

def main():
    public_key_file = None
    private_key_file = None
    public_key = None
    private_key = None
    while True:
        print("Simple RSA Crypto")
        print("--------------------")
        print("\tprviate key file: {}".format(private_key_file))
        print("\tpublic key file: {}".format(public_key_file))
        print("\t1. Encrypt Message.")
        print("\t2. Decrypt Message.")
        print("\t3. Load public key file.")
        print("\t4. Load private key file.")
        print("\t5.  Create and load new public and private key files.")
        print("\t6. Quit.\n")
        choice = input(" >> ")
        if choice == '1':
            if not public_key:
                print("\nNo public key loaded\n")
            else:
                message = input("\nPlaintext: ").encode()
                message_as_int = bytes_to_int(message)
                cipher_as_int = simple_rsa_encrypt(message_as_int,
                public_key)
                cipher = int_to_bytes(cipher_as_int)
                print("\nCiphertext (hexlified): {}\n".
                format(binascii.hexlify(cipher)))
        elif choice == '2':
            if not private_key:
                print("\nNo private key loaded\n")
            else:
                cipher_hex = input("\nCiphertext (hexlified): ").encode()
                cipher = binascii.unhexlify(cipher_hex)
                cipher_as_int = bytes_to_int(cipher)
                message_as_int = simple_rsa_decrypt(cipher_as_int,
                private_key)
                message = int_to_bytes(message_as_int)
                print("\nPlaintext: {}\n".format(message))
        elif choice == '3':
            public_key_file_temp = input("\nEnter public key file: ")
            if not os.path.exists(public_key_file_temp):
                print("File {} does not exist.")
            else:
                with open(public_key_file_temp, "rb") as public_key_
                file_object:
                    public_key = serialization.load_pem_public_key(
                                     public_key_file_object.read(),
                                     backend=default_backend())
                    public_key_file = public_key_file_temp
                    print("\nPublic Key file loaded.\n")

                    # unload private key if any
                    private_key_file = None
                    private_key = None
        elif choice == '4':
            private_key_file_temp = input("\nEnter private key file: ")
            if not os.path.exists(private_key_file_temp):
                print("File {} does not exist.")
            else:
                with open(private_key_file_temp, "rb") as private_
                key_file_object:
                    private_key = serialization.load_pem_private_key(
                                     private_key_file_object.read(),
                                     backend = default_backend(),
                                     password = None)
                    private_key_file = private_key_file_temp
                    print("\nPrivate Key file loaded.\n")

                    # load public key for private key
                    # (unload previous public key if any)
                    public_key = private_key.public_key()
                    public_key_file = None
        elif choice == '5':
            private_key_file_temp = input("\nEnter a file name for
            new private key: ")
            public_key_file_temp = input("\nEnter a file name for a
            new public key: ")
            if os.path.exists(private_key_file_temp) or os.path.
            exists(public_key_file_temp):
                print("File already exists.")
            else:
                with open(private_key_file_temp, "wb+") as private_
                key_file_obj:
                    with open(public_key_file_temp, "wb+") as public_
                    key_file_obj:

                        private_key = rsa.generate_private_key(
                                          public_exponent =65537,
                                          key_size =2048,
                                          backend = default_backend()
                                      )
                        public_key = private_key.public_key()

                        private_key_bytes = private_key.private_bytes(
                            encoding=serialization.Encoding.PEM,
                            format=serialization.PrivateFormat.TraditionalOpenSSL,
                            encryption_algorithm=serialization.NoEncryption()
                        )
                        private_key_file_obj.write(private_key_bytes)
                        public_key_bytes = public_key.public_bytes(
                            encoding=serialization.Encoding.PEM,
                            format=serialization.PublicFormat.SubjectPublicKeyInfo
                        )
                        public_key_file_obj.write(public_key_bytes)

                        public_key_file = None
                        private_key_file = private_key_file_temp
        elif choice == '6':
            print("\n\nTerminating. This program will self destruct in 5 seconds.\n")
            break
        else:
            print("\n\nUnknown option {}.\n".format(choice))

if __name__ == '__main__':
    main()
```

Take a few minutes to try this exercise on your own before we walk through it together. Note, by the way, that because a public key can be derived from a private key, loading the private key also loads the public key.

When you are ready, continue reading! You may want to refer back to Listing 4-4 from time to time. Many of our subsequent listings will reuse these imports and function definitions. To save space, we will generally not reprint them so this listing is also useful as a template.

EXERCISE 4.1. SIMPLE RSA ENCRYPTION

Using the preceding application, set up communication from Alice to Bob and then send a few encrypted messages from Alice to Bob for decryption.

Stuffing the Outbox

Once EATSA has built the RSA encryption application, they hand it off to Alice and Bob and order them to begin the mission. Alice will infiltrate WAGGS and send updates to Bob. What do Alice and Bob need to do first?

What's amazing about public/private key pairs is that they don't have to agree on much of anything before they split up in order for Alice to send secure messages to Bob![3] As long as Alice knows where to look, Bob can publish a public key to her *anywhere*. He could send it in a newspaper, recite it to her over the phone, or publicize it on a Goodyear blimp flying around West Antarctica. The key is *public*. It does not matter if the West Antarctica Counter Intelligence sees it: they won't be able to decrypt Alice's messages.

Right?

Alice departs from EATSA headquarters, crosses the border, and makes her way to West Antarctica City where she infiltrates WAGGS. While she's thus engaged in her covert culinary caper, Bob generates a public/private key pair. He hangs onto the private key and publishes the public key for Alice to see.

Let's follow along. Start up an instance of the application that represents Bob's version and select option 5, which generates new paired keys and saves them to disk. Once that's done, you will have two files you can inspect in an editor.

Take a look at the public key file (you chose the name for it when prompted). Its contents should look something like this:

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAuGFr+NV3cMu2pdl+i52J
XkYwwSHgZvAOFyIPsZ/rp6Ts5iBTkpymt7cf+cQCQro4FSw+udVt4A8wvZcppnBZ
h+17ZZ6ZZfjOLCr/3sJw8QfZwuaX5TZxFbJDxWWwsR4jLHsiGsPNf7nzExn7yCSQ
```

[3] Bob won't necessarily know who they're coming from, but that's a separate (and very interesting) problem. At least he'll know he's the only one who can read them.

```
sXLNqc+mLKP3Ud9ta14bTQ59dZIKKDHVGlQ1iLlhjcE1dhOAjWlsdCVfE+L/bSQk
Ld9dWKCM57y5tiMsoqnVjl28XcsSuiOd4QPGITprsXOjb7/p/rzXc9OQHHGyAQzs
WTAbZNaQxf9AY1AhE4wgMVwhnrxJA2g+DpY1yXUapOIH/hpDOsMH56IGcMx9oV/y
SwIDAQAB
-----END PUBLIC KEY-----
```

That's a PEM-formatted public key. Congratulations! Bob can take this key and publish it to a West Antarctica newspaper in the classifieds.

Meanwhile, Alice has been carefully observing what the West Antarctica politicians like to eat. *How un-Antarctican!* she thinks to herself as she watches them eat *hot* dogs and *hot* chocolate. Then, glancing back to the newspaper in her hands, she finds the classified ad she's been looking for! The public key has arrived! She copies it down carefully into a file and now has the ability to encrypt messages for Bob's eyes only.

Following along, let's copy the public key we just generated into a new file. This represents the file that Alice creates after copying the text out of the classified ads. Now launch a new instance of the application that represents Alice's copy of the program. Choose option 3 to load her public key.

Alice needs to send a message back to Bob. That's option 1 in our program. Run it, select option 1, and enter the text "hot dogs" into the plaintext field. Out pops the encrypted message.[4] If you used the preceding public key, you would get the following output:

```
Plaintext: hot dogs

Ciphertext (hexlified): b'56d5586cab1764fae575bc5815115f1c5d759
daddccbd6c9cb4a077026e2616dfca756ffa7733538e66997f06ebbbb853028
3926383a6bb80b7145990a29236d042048eed8eb7607bd35fcafe3dadd5d60a
1f8694192bddedac5728061234ffbb7a407155844a7e79b3dbc9704df0de818
d24acad32ccd6d2afe2d0734199c76e5c5c770fa8c3c208eceae00554aa2f29
9a8510121d388d85f35fa49c08f3e9d7540f22fe5eb4ea15da5f387dbdd0e00
6710aa9031b885094773ef3329cde91dbede53ed77b96483d34daa4fedbf5bc
d95e95b6b482a7decbf47fe2df0e309d706ab9c73ce73a2bdef33b786dd12e9
8a9ce34bbc1847f36e13ae9eea4007b616'
```

[4] The message is displayed as the hex representation of its bytes to make it easy to select and paste elsewhere.

Let’s do it again, but this time for “hot chocolate.” If you do so using the preceding public key we showed you, you would get this output (but go ahead and use your own generated public key):

```
Plaintext: hot chocolate

Ciphertext (hexlified): b'4d1e544e71c4cb15636ef4b0d629294538a05
979db762952cc5f0fc494f71535dff326dbb8543d0f2ace51a2279f65c2a76b
2a5ca5a3ee151e65e516afcb1d4da9ca9871dc7ce1dd4361a3b49def05c5089
99f5fab81b869b251ba8694fb171ab56ca1cde7cef0ac3934da4c28f7bfbb65
b03afa9cff30db974f0bd4fb8dee7fac75c99cd4def94ca8de83d46fffa092a
90642c9cfbfbf07c371f5aa3a62dc997d20e9959fcbec7dd0b434709b679619
ea195008a9a12eaa7462ffdbe8e6f765dd86b21f0f1d9b8b2b523ca7f11785e
fc6da84ec717bd1f0e2191e5a3bef74e489b5e396c49bd8f222ccd89984dbec
8b5e4cbb23ba739637d3307bca4e9f57e7'
```

Again, Alice cannot decrypt these messages, even though she encrypted them herself: she doesn’t have the private key. At least, that’s what the theory tells them.

Confident in her edible espionage, she takes these messages and sends them to Bob via an insecure carrier penguin [15]. Bob receives the message and reloads his application. First, he loads the private key file using option 4 and then chooses option 2 to attempt a decryption. Sure enough, when he copies in the message for Alice, it decrypts correctly:

```
Ciphertext (hexlified): 56 d5586cab1764fae575bc5815115f1c5d759da
ddccbd6c9cb4a077026e2616dfca756ffa7733538e66997f06ebbbb85302839
26383a6bb80b7145990a29236d042048eed8eb760735fcafe3dadd5d60a1f86
94192bddedac5728061234ffbb7a407155844a7e79b3dbc9704df0de818d24a
cad32ccd6d2afe2d0734199c76e5c5c770fa8c3c208eceae00554aa2f299a85
10121d388d85f35fa49c08f3e9d7540f22fe5eb4ea15da5f387dbdd0e006710
aa9031b885094773ef3329cde91dbede53ed77b96483d34daa4fedbf5bcd95e
95b6b482a7decbf47fe2df0e309d706ab9c73ce73a2bdef33b786dd12e98a9c
e34bbc1847f36e13ae9eea4007b616

Plaintext: b'hot dogs'
```

“Hot dogs!” Bob exclaims. “Disgraceful!”

```
Ciphertext (hexlified): 4d1e544e71c4cb15636ef4b0d629294538a05979
db762952cc5f0fc494f71535dff326dbb8543d0f2ace51a2279f65c2a76b2a5c
a5a3ee151e65e516afcb1d4da9ca9871dc7ce1dd4361a3b49def05c508999f5f
ab81b869b251ba8694fb171ab56ca1cde7cef0ac3934da4c28f7bfbb65b03afa
9cff30db974f0bd4fb8dee7fac75c99cd4def94ca8de83d46fffa092a90642c9
cfbfbf07c371f5aa3a62dc997d20e9959fcbec7dd0b434709b679619ea195008
a9a12eaa7462ffdbe8e6f765dd86b21f0f1d9b8b2b523ca7f11785efc6da84ec
717bd1f0e2191e5a3bef74e489b5e396c49bd8f222ccd89984dbec8b5e4cbb23
ba739637d3307bca4e9f57e7
```

```
Plaintext: b'hot chocolate'
```

Bob’s eyes narrow. “Hot chocolate?! Have they no shame?!”

So far, so good! Alice’s messages got to Bob. They were intercepted by agent Eve of WACKO, but she shouldn’t be able to read them, even though she also has the public key. If Alice can’t read her own messages, why should Eve be able to?

What Alice and Bob don’t know is that Eve is about to wreak all kinds of havoc. In the rest of this chapter, we’ll be walking through some of the ways that RSA can be compromised and how to do it right. But first, exercises!

EXERCISE 4.2. WHO GOES THERE? BOB? IS THAT YOU?

Assume the role of Eve and imagine that you know everything about Alice’s and Bob’s operation *except* the private key. That is, suppose you know about the classified ads, the carrier penguins, and even the encryption program.[5] Their scheme is strengthened by using asymmetric encryption, but is still vulnerable to an MITM (man-in-the-middle) attack. How can Eve position herself such that she can trick Alice into sending messages that Eve can decrypt, and Bob into receiving *only* false messages from Eve instead of Alice?

[5] Remember Kerckhoff’s principle? Here it is again!

EXERCISE 4.3. WHAT'S THE ANSWER TO LIFE, THE UNIVERSE, AND EVERYTHING?

We have already talked about chosen plaintext attacks in the previous chapter. The same attack can be used here. Again assume the role of Eve, the WACKO agent. You've intercepted Bob's public key in the newspaper, and you have access to the RSA encryption program. If you suspect you know what Alice is sending in her encrypted messages, explain or demonstrate how you would verify your guesses.

What Makes Asymmetric Encryption Different?

As you learned already in this section, RSA is an example of *asymmetric encryption*. If you haven't heard of asymmetric encryption before now, hopefully the exercises you just walked through have exposed you to the key concepts. Now let's make a few things explicit.

In symmetric encryption, there is a single, shared key that works to both encrypt and decrypt the message. This means that anyone with the power to create an encrypted message has the same ability to decrypt the same message. It is impossible to give somebody the power to decrypt a symmetrically encrypted message without also giving them the ability to encrypt the same kind of messages and vice versa.

In asymmetric cryptography, there is always a private key that must never be disclosed and a public key that can be disclosed widely. Exactly what can be done with the key pair depends on the algorithm. In this chapter we have been focusing on RSA encryption. We'll review RSA's operations in this section as a concrete example but keep in mind that they may not apply to other asymmetric algorithms and operations.

Specifically, RSA supports an asymmetric encryption scheme in which you can use one key to encrypt the message and a different key to decrypt a message. Typically, either key can act in either role: a private key can encrypt messages that can be decrypted by the public key and vice versa. With RSA, of course, one key is clearly the *private* key because the public key can be *derived* from the private key, but not the other way around. It is impossible to have an RSA private key and not *also* have the matching public key. Thus, one key is unambiguously designated as "private" and the other is "public."

The possessor of a properly protected RSA private key and an adequately robust protocol can use asymmetric encryption for two purposes:

1. **Cryptographic dropbox**: Anyone with the public key can encrypt a message and send it to the owner of the private key. Only someone with the private key can decrypt the message.

2. **Signatures**: Anyone with the public key can decrypt a message encrypted by the private key. This obviously is not helpful for confidentiality (anyone can decrypt the message) but it helps to prove *the identity of the sender*, or at least that the sender is in possession of the private key; they wouldn't be able to encrypt a public-key-decryptable message otherwise. This is an example of a cryptographic *signatures*, which we will talk about later.

Note: RSA Encrypts Small Things

The cryptographic dropbox operation we are learning about right now is almost never used to send complete messages in this way. The most common way RSA encryption was used (again, it is being phased out) was to encrypt a symmetric key for transport from one party to another. This is another concept we'll save for a later chapter.

What is really fantastic about the asymmetric nature of RSA encryption is that the two parties do not need to have met each other to begin exchanging messages. In our example, Alice and Bob did not need to create any shared keys together. Alice did not even need to meet or know Bob. So long as Alice had Bob's public key, she can encrypt messages that only Bob can read.

Unfortunately, the ability to encrypt something for only one person is not the only important thing in real life. As demonstrated in the exercises, the advantage of asymmetric encryption is also its weakness. The ability to communicate without any previous interactions also means that, absent additional information, there is no way to know that you are communicating with the *right person*.

If you worked through the earlier exercises, you will have also learned that it is quite simple for WACKO to *both read and alter* the communications between Alice and Bob by deceiving both parties by intercepting messages and keys.

1. They can deceive Alice by intercepting and modifying the public key published in the newspaper. By inserting *their own* public key—which Alice now wrongly assumes is Bob's—they can read all messages sent by Alice intended for Bob. Alice, without additional information, cannot know that the public key has been compromised.

2. They can then deceive Bob by preventing Alice's incorrectly encrypted messages from reaching him and sending him false messages encrypted under the correct public key, which they intercepted. Bob, without additional information, has no way of knowing who is sending the messages.

This is a critical difference between symmetric keys and asymmetric keys. In fact, some cryptographers distinguish between a "secret" symmetric key and a "private" asymmetric key. Two people can share a secret, but only one person knows their own private key. What this means in practice is that a symmetric key, provided that it *remains secret to both parties*, can be used to establish that you are talking to the right person (i.e., the person you created the shared secret key with) while asymmetric keys cannot.[6]

Let's sidestep that problem for now and save it for later, since there is indeed a solution to it that is discussed in the context of *certificates*.

Pass the Padding

Recall from earlier that the EATSA chose to implement RSA without any padding. They really shouldn't have done that; it's a pretty serious mistake. In fact, it's so serious that the `cryptography` module does not even *allow* you to encrypt with RSA without padding!

What, then, is padding, and why is padding such a big deal?

[6] Unless the public key is *also* guaranteed to be secret, but then we've just defeated the purpose of asymmetric keys, in a way, by requiring a secure shared channel for key exchange.

The best way to explain this is to demonstrate how to read messages encrypted with the public key *even if you don't have the private key*, so long as those messages are not padded. Another great exercise is to search the Internet for RSA padding attacks. There are *many* problems with using unpadded plaintext.

Deterministic Outputs

Let's start with the most basic problem. RSA by itself is a *deterministic* algorithm. That means that, given the same key and message, you will always get the same ciphertext, byte for byte. Recall that we had the same problem with symmetric key ciphers like AES. It was essential to use the *initialization vector (IV)* to prevent deterministic outputs. Do you remember why deterministic outputs are so bad?

The problem with deterministic outputs is that they enable passive eavesdroppers, such as Eve, to do some cryptographic reverse engineering. Because the encryption is deterministic, if Eve knows that *m* encrypts to *c* then any time Eve sees *c* she knows what the plaintext is.

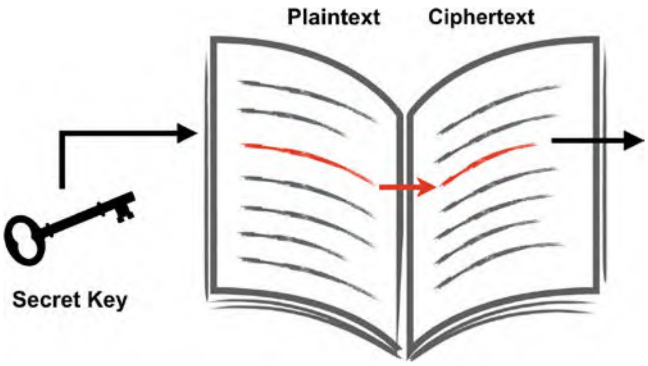

Figure 4-1. *If RSA's outputs are deterministic, an adversary that discovers the mapping between a plaintext and the corresponding ciphertext can record it into a lookup table for later use. Does this figure look familiar?*

Eve has both the public key *and the algorithm* (you can never assume that a cryptographic algorithm is secret). She can encrypt any number of potential messages and store a lookup table of pre-encrypted values. Does Figure 4-1 look familiar? We showed this same image in Chapter 3 to talk about ECB mode for symmetric ciphers and the problems with it.

But a deterministic asymmetric encryption would be worse. Unlike symmetric encryption we have to assume that the adversary *has* the (public) key. In our hypothetical Antarctican conflict, Eve could discover, or simply guess, that Alice is sending messages based on her surveillance of the cafeteria. If she tries encrypting a few hundred words by making a list of things found within the room (e.g., the names of the politicians eating in the cafeteria, topics of conversation, and the food being eaten), as soon as she encrypted "hot dogs" or "hot chocolate," the encrypted values would match up perfectly with what is intercepted in the message back to Bob. For short messages such as these, especially if EA Intelligence always writes words in lowercase, there are less than 300 million messages to try that are 8 characters long. It's not too much trouble to create a table of that many messages to their ciphertext. Using this lookup table, Eve could identify "hot dogs" relatively quickly.

Even if Eve cannot guess the message, there is still all kinds of analysis that can be done. Suppose that Alice continues to send the same message day after day. While Eve may not be able to decrypt the message, she would still be able to confidently state that it was the *same* message. We have considered numerous examples wherein this kind of "information leak" is exploited in previous chapters.

EXERCISE 4.4. BRUTE-FORCE RSA

Write a program that uses brute force to decrypt an RSA-encrypted word that is all lowercase (no spaces) and less than four characters. The program should take a public key and the RSA-encrypted ciphertext as the inputs. Use the RSA encryption program to generate a few words of four or fewer letters and break these codes with your brute-force program.

EXERCISE 4.5. WAITING IS THE HARDEST PART

Modify the brute-force program to try all possible words of five or fewer letters. Measure the time it takes (worst cast) to brute force a four-letter word vs. a five-letter word. About how many times longer does it take and why? How long would it take to try all possible six-letter words?

EXERCISE 4.6. DICTIONARY ATTACKS

It should be pretty clear that it will take longer than your probable attention span to try all possible lowercase ASCII words of length much greater than four or five. But we already saw this same problem in previous chapters. Let's try the same solutions. Modify your brute-force program to take a dictionary as input for trying arbitrary English words.

Chosen Ciphertext Attack

RSA without padding is also vulnerable to something called a "chosen ciphertext attack."[7] This type of attack works when you can get the victim to decrypt *some* ciphertexts of your choosing on your behalf. That may sound counter-intuitive. Why would anyone decrypt anything for you? For example, why would Bob decrypt anything for Eve?

Remember that a lot of computer security is all about psychology, trickery, and human thinking [1, Chap. 2]. What is Bob looking for? Bob is assuming that he is decrypting human-readable messages from Alice. What if he got a message that was not human readable? Suppose, for example, that upon decrypting a message (supposedly from Alice) he got the following output:

```
b'\xe8\xca\xe6\xe8'
```

It's entirely possible that this is just assumed to be due to a transmission error. Those things happen in real life all the time. It could be bit error, or a carrier penguin might have smudged the ink. Bob probably sees a lot of messages that do not decrypt correctly.

What does Bob do? If he does not have very good security controls in place, he might just throw it away. But if Alice can infiltrate the enemy, it can work the other way as well. Which do you think is easier for Eve to get into her hands? Top-secret messages that are being sent up the chain of command for analysis, or "incorrect" messages that get thrown away in the trash? If Eve has a covert agent of her own on the janitorial staff, it might be very possible to get discarded paper or inadequately destroyed data.

[7] Chosen ciphertext attacks (CCA) are way more complicated than we have space to discuss here. Please consider our CCA discussion to be hyper-simplistic. If you want to know more about CCA and indistinguishability under CCA, Dr. Matthew Green has some great blog posts [7].

Let's assume this scenario then: Eve can send arbitrary ciphertext to Bob. For our purposes, Eve cannot see any of the human-readable messages but can recover the supposedly erroneous messages discarded by Bob because they seem to make no sense.

Unfortunately for Alice and Bob, Eve can use this trick to decrypt almost any message Alice sends back to her home base. The mathematics behind this trick are really cool and are used in multiple examples throughout the chapter. So let's pause a minute to talk about *homomorphisms* in encryption.

The basic concept of an encryption homomorphism is that if you perform some kind of computation on the ciphertext, the result is reflected in the plaintext. Not all crypto systems have homomorphic properties, but RSA does to some extent. In RSA we will see that there are ways to do multiplication on the ciphertext that results in multiplications on the plaintext. There are other special homomorphic encryption technologies that exist and are being developed right now that enable third parties to provide services on data *without being able to read it*. You may have heard of some of these; if not, try searching for "homomorphic encryption" online. It's pretty interesting stuff.

While RSA is not a homomorphic encryption scheme, this multiplication property is very interesting (and also powers a number of vulnerabilities). Do you remember from algebra class that $(a^c)(b^c) = (ab^c)$? The same is true for modular exponentiation as shown in the following equation:

$$(m_1)^e (m_2)^e (\bmod n) = (m_1 m_2)^e (\bmod n) \tag{4.3}$$

Does any part of this equation look familiar? Take a look back at (4.1). Do you see it now?

Any time we encrypt a value (m) in RSA, we end up with m^e mod n. On the left-hand side of the (4.3), we have *two* encryptions, one of m_1 and one of m_2, both using the same public exponent e and both modulo the same modulus n.

On the right-hand side, we have a *single* encryption of the value m_1 times m_2. What this equation tells us is that if you take each of these individually encrypted values and multiply them together (mod n), you get the encrypted result of the multiplication!

Restated another way, the product of two ciphertexts (encrypted under the same public key) decrypts to the product of the two plaintexts. Try to do the following exercise on your own before we walk through it.

EXERCISE 4.7. HOMOMORPHIC PROPERTY OF UNPADDED RSA

Use (4.3) to multiply two RSA-encrypted numbers together and decrypt the result to verify the equation.

The code for this exercise is very simple, so definitely try it yourself first. When you're ready, our solution is in Listing 4-5.

Listing 4-5. Solution

```
# FOR TRAINING USE ONLY! DO NOT USE THIS FOR REAL CRYPTOGRAPHY

import gmpy2, sys, binascii, string, time
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa

#### DANGER ####
# The following RSA encryption and decryption is
# completely unsafe and terribly broken. DO NOT USE
# for anything other than the practice exercise
################
def simple_rsa_encrypt(m, publickey):
    numbers = publickey.public_numbers()
    return gmpy2.powmod(m, numbers.e, numbers.n)

def simple_rsa_decrypt(c, privatekey):
    numbers = privatekey.private_numbers()
    return gmpy2.powmod(c, numbers.d, numbers.public_numbers.n)

private_key = rsa.generate_private_key(
      public_exponent=65537,
      key_size=2048,
      backend=default_backend()
)
public_key = private_key.public_key()
```

```

n = public_key.public_numbers().n
a = 5
b = 10

encrypted_a = simple_rsa_encrypt(a, public_key)
encrypted_b = simple_rsa_encrypt(b, public_key)

encrypted_product = (encrypted_a * encrypted_b) % n

product = simple_rsa_decrypt(encrypted_product, private_key)
print("{} x {} = {}".format(a,b, product))
```

If this kind of math doesn't make a lot of sense, don't worry too much about it at this point. Just try to grasp how it is *used* even if you aren't fully sure how it works.

Returning to our current example, suppose that Eve has a ciphertext *c* obtained by the RSA public key encryption of *m*. Without the private key, Eve should not be able to decrypt it. And presumably Bob won't decrypt it for her either. If he will decrypt a *multiple* of it, however, Eve can recover the original.

For our example, let's choose our multiple to just be 2. Eve starts by encrypting 2 using (4.1) and the public key to get c_r.

For clarity, let's call the original ciphertext c_0. If we multiply c_0 and c_r (modulo *n*), we'll get a new ciphertext that we'll call c_1.

$$c_1 = c_0 c_r (\bmod n).$$

From (4.3), this works out to be

$$\begin{aligned} c_1 &= c_0 c_r (\bmod n) \\ &= m^e r^e (\bmod n) \\ &= (mr)^e (\bmod n). \end{aligned}$$

So how does Eve use this? Suppose that Eve has intercepted one of Alice's ciphertexts *c*. Eve takes her computed c_r (again, this is just the value of 2 encrypted under the public key) and then multiplies the two encrypted values together (modulo *n*). Eve sends this new ciphertext c_1 to Bob.

Bob receives c_1 and decrypts it to mr and converts the integers to bytes. He finds that it does not decrypt to anything legible and assumes that something was damaged in transport. Shrugging his shoulders, he crumples up the paper and throws it in the waste basket. Later that night, Eve's agent goes through the trash and finds the crumpled up paper. Creating a quick copy, she sends it by secret carrier back to Eve.

Eve now has mr and needs to extract m. No problem. She chose r to be 2. In familiar arithmetic you would divide by r to extract m. But when doing this arithmetic with modulo operations, you have to use a different inverse operation: r–1 (mod n). Fortunately, there are libraries that compute these kinds of numbers for us, like `gmpy2`.

```
r_inv_modulo_n = gmpy2.powmod(r, -1, n)
```

EXERCISE 4.8. EVE'S PROTEGE

Recreate Eve's chosen ciphertext attack. Create a sample message in Python, as you have done previously, using the public key to encrypt it. Then, encrypt a value of *r* (such as 2). Multiply the two *numeric* versions of the ciphertext together and don't forget to take the answer modulo *n*. Decrypt this new ciphertext and try to convert it to bytes. It shouldn't be anything human readable. Take the numeric version of this decryption and multiply it by the inverse of *r* (mod *n*). You should be back to the original number. Convert it to bytes to see the original message.

Common Modulus Attack

Another problem for RSA without padding is the "common modulus" attack. Recall that the n parameter is the modulus and is included in both the public key and private key. For mathematical reasons beyond the scope of this book, if the same RSA message is encrypted by two different public keys with the same n modulus, the message can be decrypted without the private key.

In the chosen ciphertext example, we walked through the math in some detail both because it can be described relatively easily and because it is critical to multiple attacks. For this example, in the interests of simplicity and conserving space, we won't get into the mathematical details. Instead, use the code in Listing 4-6 to test and explore the attack. If you're interested in the details of the math, you can read "Common Modulus Attacks on Small Private Exponent RSA and Some Fast Variants (in Practice)" by Hinek and Lam.

Listing 4-6. Common Modulus

```
# Partial Listing: Some Assembly Required

# Derived From: https://github.com/a0xnirudh/Exploits-and-Scripts/tree/master/RSA At tacks
def common_modulus_decrypt(c1, c2, key1, key2):
    key1_numbers = key1.public_numbers()
    key2_numbers = key2.public_numbers()

    if key1_numbers.n != key2_numbers.n:
        raise ValueError("Common modulus attack requires a common modulus")
    n = key1_numbers.n

    if key1_numbers.e == key2_numbers.e:
        raise ValueError("Common modulus attack requires different public exponents")

    e1, e2 = key1_numbers.e, key2_numbers.e
    num1, num2 = min(e1, e2), max(e1, e2)

    while num2 != 0:
        num1, num2 = num2, num1 % num2
    gcd = num1

    a = gmpy2.invert(key1_numbers.e, key2_numbers.e)
    b = float(gcd - (a*e1))/float(e2)

    i = gmpy2.invert(c2, n)
    mx = pow(c1, a, n)
    my = pow(i, int(-b), n)
    return mx * my % n
```

Note that in order to test this attack, you will need two public keys with the same modulus (*n* value) and different public exponents (*e* values). Recall that *e* is recommended to always be 65537. But obviously you won't use that for both keys in this example.

How does one create a public key? In all of our examples so far, we either generated new keys or loaded them from disk.

Recall that the *n* and *e* values *define* the public key. Everything else is just wrappers for convenience. The `cryptography` module provides an API for creating a key directly from these values. The RSA private key objects have a method called `private_numbers`, and the RSA public key objects have a method called `public_numbers`. These methods return data structures with data elements such as *n*, *d*, or *e*. These "numbers" objects can also be used to create the key objects.

In Listing 4-7, we generate a private key and then manually create another key with the same modulus and different public exponent.

Listing 4-7. Common Modulus Key Generation

```
# Partial Listing: Some Assembly Required

private_key1 = rsa.generate_private_key(
    public_exponent =65537,
    key_size=2048,
    backend = default_backend()
)
public_key1 = private_key1.public_key()

n = public_key1.public_numbers().n
public_key2 = rsa.RSAPublicNumbers(3, n).public_key(default_backend())
```

Now you should have all the Python code you need to test out this attack.

At this point you might be asking yourself, "how practical is this attack?" In order to carry it out, you have to have *the same message* encrypted under *two keys with the same modulus*. Why would the same message ever be encrypted twice under two different keys and why would two different keys ever have the same modulus?

When dealing with cryptography, you should never rely on this kind of thinking. If there is a way for the cryptography to be exploited, the bad guy will figure out a way to exploit it. Let's start by thinking about how to get the same message encrypted by two different keys.

One possibility is to convince Alice that a new public key has been created and that she needs to switch. If we control the new public key, we can give her a key with *n* and *e* values of our choosing.

But if we can control her key, why would we need to use the common modulus attack? Why not just give her a public key that we created and for which we have the paired private key?

It is true that a new private key/public key pair will allow Eve to decrypt any messages Alice sends in the future. But the common modulus attack will allow Eve to potentially determine some messages sent *in the past*. In our example with Alice infiltrating the cafeteria, the food service probably repeats with some regularity. In fact, as we discussed previously, Eve can already tell if the same message is being resent even if she cannot decrypt it. If Eve observes that the same messages are being sent over and over, the common modulus attack provides a much greater view into the history of what is sent as well as information about messages sent in the future.

EXERCISE 4.9. COMMON MODULUS ATTACK

Test out the code in this section by creating a common modulus attack demo.

EXERCISE 4.10. COMMON MODULUS USE CASES

Write out an additional scenario when the use of the common modulus attack might be useful to an attacker.

The Proof Is in the Padding

As we have just demonstrated, this very raw form of RSA, sometimes referred to as "textbook RSA," is relatively easy to break. There are two critical problems. As we have already seen, one problem with textbook RSA is that the outputs are deterministic. This makes attacks like the common modulus attack, which require encrypting the same message twice, much easier.

Perhaps the bigger problem is how malleable the messages are. We talked about malleability with symmetric encryption in the previous chapter. With RSA we have similar problems, for example, multiplying the RSA ciphertext and getting a decryptable value.

There are also potential problems with trying to encrypt tiny messages, such as some of the small messages we have encrypted in our exercises. In addition to the brute-force methods in the exercises, there are ways to break smaller messages especially with smaller public exponents (e.g., $e = 3$).

To reduce or eliminate these problems, practical uses of RSA always utilizes padding with *random elements*. RSA padding is applied to the plaintext message before encryption by the raw RSA computations we have been working with. The padding ensures that messages are not too small and provide a certain amount of structure that reduces malleability. Also, the randomized elements operate not unlike an IV for symmetric encryption: good randomized padding ensures that each ciphertext produced by the RSA encryption operation, even for the same plaintext, is (with very high probability) unique.

RSA without padding is dangerous enough that the `cryptography` module does not even have a padding-free RSA operation. It should be absolutely clear to you that you must not use RSA for encryption without padding. While the `cryptography` module does not allow this, other libraries do. Significantly, this includes OpenSSL.

At the time of this writing, there are two padding schemes that are typically used. The older scheme is called PKCS #1 v1.5 and the other is OAEP, which stands for Optimal Asymmetric Encryption Padding. Either of these padding schemes can be used with the `cryptography` module as shown in Listing 4-8.

Listing 4-8. RSA Padding

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

def main():
    message = b'test'

```

```
private_key = rsa.generate_private_key(
      public_exponent =65537,
      key_size=2048,
      backend=default_backend()
  )
public_key = private_key.public_key()

ciphertext1 = public_key.encrypt(
    message,
    padding.OAEP(
        mgf = padding.MGF1(algorithm = hashes.SHA256()),
        algorithm = hashes.SHA256(),
        label = None # rarely used. Just leave it 'None'
    )
)

###
# WARNING: PKCS #1 v1.5 is obsolete and has vulnerabilities
# DO NOT USE EXCEPT WITH LEGACY PROTOCOLS
ciphertext2 = public_key.encrypt(
    message,
    padding.PKCS1v15()
)

recovered1 = private_key.decrypt(
ciphertext1,
padding.OAEP(
    mgf=padding.MGF1(algorithm=hashes.SHA256()),
    algorithm=hashes.SHA256(),
    label=None # rarely used.Just leave it 'None'
))

```

```
    recovered2 = private_key.decrypt(
    ciphertext2,
     padding.PKCS1v15()
  )

    print("Plaintext: {}".format(message))
    print("Ciphertext with PKCS #1 v1.5 padding(hexlified): {}".
    format(ciphertext1.hex()))
    print("Ciphertext with OAEP padding (hexlified): {}".
    format(ciphertext2.hex()))
    print("Recovered 1: {}".format(recovered1))
    print("Recovered 2: {}".format(recovered2))

if __name__=="__main__":
    main()
```

If you run this demonstration script repeatedly, you will observe that the ciphertext for both padding schemes causes the output to change *every time*. Consequently, adversaries like Eve cannot execute the chosen ciphertext attack nor the common modulus attack demonstrated earlier in this chapter. She is also unable to use RSA's deterministic encryption to analyze message patterns, frequency, and so forth.

Padding also solves the problem of losing leading zeros during encryption. Padding ensures that the input is always a fixed size: the bit size of the modulus. So, for example, with padding, the input to RSA encryption with a modulus size of 2048 will always be 256 bytes (2048 bits). Because the size of the output is known, it also allows the plaintext to start with leading zeros. Regardless of whether the combined message starts with 0, the known size means that zeros can be affixed until the correct size is reached.

So everything is fine now, right? Alice and Bob will switch to using padding and Eve will be shut out of their communications?

First of all, please note that padding does not solve either of the man-in-the-middle or authentication problems. Eve can still intercept and change the public key, enabling complete decryption of Alice's messages. Bob still cannot tell who is sending him messages. These are problems for another chapter.

Second, the astute reader probably noticed the warning in the source code listing. Just in case you glanced over it without paying attention, we will emphasize it again.

Warning: Say “No” to PKCS #1 v1.5

Do *not* use PKCS #1 v1.5 unless you must do so to be compatible with legacy protocols. It is obsolete and has vulnerabilities (including one we will test in the next section)! For encryption, always use OAEP when possible.

Before moving on from this section, two other comments are in order regarding the use of OAEP:

1. You may have noticed the “label” parameter to OAEP. This is rarely used and can typically be left as `None`. Using a label does not increase security, so ignore it for now.
2. OAEP requires the use of a hashing algorithm. In the example we used SHA-256. Why not SHA-1? Is this related to known weaknesses in SHA-1? No. Actually, there are no known attacks against OAEP that depend on SHA-1’s weaknesses. Because SHA-1 is considered obsolete, it is best to not use it when writing your own code, but if you have to use OAEP with SHA-1 for compatibility reasons or to maintain someone else’s code, it is not known to be less secure than SHA-256 *as of the time of this writing*.

EXERCISE 4.11. GETTING AN UPGRADE

Help Alice and Bob out. Rewrite the RSA encryption/decryption program to use the `cryptography` module instead of `gmpy2` operations.

Exploiting RSA Encryption with PKCS #1 v1.5 Padding

This section is going to be exciting and fun! Eve is not a cryptographer and you—because you are reading this book—are probably not a master cryptographer either. However, you and Eve are going to implement an attack designed by a brilliant cryptographer and use it to break Alice and Bob’s cipher.

This attack is not only *fun*, but it is very real. Not only has it been a real attack in the past, but it even continues to be used today against poorly configured TLS servers. It’s both historical and contemporary at the same time.

The paper in question is "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1" by Daniel Bleichenbacher [2]. You can find this paper online, and some readers may be interested in the mathematics behind the attack. In the sections that follow, we are going to walk through this paper creating an implementation of the attack. At the same time, we will try to give some intuition behind certain key concepts. If you find the in-depth details frustrating or uninteresting, you should be able to ignore most of the explanation and just put together a working RSA cracker from the source code listings. We won't be offended.

There are going to be a lot of code snippets for this example. You should start with Listing 4-9 that initializes a few imports. Don't forget about the dependencies on other functions we've already seen in this chapter. As we work through new snippets, add them to this skeleton.

Listing 4-9. RSA Padding Oracle Attack

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend

import gmpy2
from collections import namedtuple

Interval = namedtuple('Interval', ['a','b'])
# Imports and dependencies for RSA Oracle Attack
# Dependencies: simple_rsa_encrypt(), simple_rsa_decypt()
#               bytes_to_int()
```

Alice and Bob are at it again. This time, though, they're using RSA with padding. But EATSA is still making bad decisions. They decide to use PKCS #1 v1.5 simply because it requires no parameters. Originally they were going to use OAEP, but the East Antarctica Taskforce for Modern Operational RSA Employment and Better Encryption, Especially in the Field (EATMOREBEEF) apparently argued for weeks about the task force *name*. Pressing up against a deadline, and unable to agree about which hashing algorithm should be used for OAEP, and whether "EATMOREBEEF" should be used for the label, they threw up their hands and said, "We're pretty sure PKCS #1 v1.5 is good enough."

Once again, we find Alice in the West Antarctica spying on her neighbors. This time, however, Alice is posing as a CEO for an ice-making company meeting other executives in the ice industry at a conference in West Antarctica City. Sales of ice have melted in the last few years, and the government, facing its own problems with frozen assets and decreased liquidity, has been either unable or unwilling to offer subsidies. Alice's mission is to continue crystallizing the dissent against the current party in power, in an attempt to solidify influence in the next election.

After the conference, Alice needs to send Bob a report of CEOs that she has convinced to donate significantly to the opposition party. Alice transmits the following message using RSA with PKCS #1 v1.5: "Jane Winters, F. Roe Zen, and John White."

Alice whips out a mobile flip phone (they are slowly catching up in technology… no smart phones yet, but they finally did away with carrier penguins). She keys in the message to Bob and it automatically converts it to a number, encrypts it, and transmits it. A few seconds later, her phone vibrates with a new message:

```
Received: OK
```

Elsewhere in the city, Eve watches this communication. She has been tracking Alice since crossing the border. But she cannot decrypt the messages. Alice even came with the public key already installed in the phone so Eve can't give her a fake key either. What can she do?

Fortunately for Eve, she finds out through her own intelligence agency that Alice and Bob are using PKCS #1 v1.5 for the RSA padding. Eve is surprised. After all of the events of the earlier part of this chapter, Eve has been reading up on RSA quite a bit, and she knows that this padding scheme has known vulnerabilities. Why are they using it, she wonders. Did they not get the memo?

Eve has a copy of the Bleichenbacher paper and begins reading. The paper explains that the PKCS #1 v1.5 padding can be broken with an oracle attack similar to the one we saw in the previous chapter.

In this case, Eve needs an oracle to tell her whether or not a given ciphertext (a number) decrypts to something with proper padding. The oracle will not, of course, tell her what the ciphertext decrypted to; all it needs to say is "yes" or "no" with regard to the padding.

Fortunately, Eve has been monitoring EA communications, and it appears that they built an error-reporting system into their technologies. When Alice sends a valid message, she gets back

```
Received: OK
```

But when Eve sends a random number (ciphertext), she almost always gets back

```
Failed: Padding
```

After sending a thousands of random numbers, she did eventually get back one that answered with the OK message. As far as she could tell, it was not a “real” message (human readable, or one that Bob understood), but it did have the correct padding as reported by the automated processing system.

This is Eve’s oracle. It is all she needs to completely decrypt a ciphertext message.

For convenience in writing her attack program, Eve will start by breaking a message encrypted locally with a self-generated private key. Eve will use a pluggable oracle configuration so that when it’s time to attack Bob, she can simply switch out the oracle used to power the attack. The test oracle uses the real private key to decrypt the message and check whether the message has the proper formatting.

Eve starts reading up on PKCS v1.5 and starts playing around with her own experiments. Creating her own key pair, she encrypts messages with the padding and then examines the output. She encrypts the message “test” and then decrypts the message *without removing the padding*. Listing 4-10 shows the key snippet of the code that she used.

Listing 4-10. Encrypt with Padding

```
# Partial Listing: Some Assembly Required

from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend
import gmpy2

# Dependencies: int_to_bytes(), bytes_to_int(), and simple_rsa_decrypt()

private_key = rsa.generate_private_key(
      public_exponent=65537,
      key_size=2048,
      backend=default_backend()
  )
public_key = private_key.public_key()

message = b'test'

```

```
###
# WARNING: PKCS #1 v1.5 is obsolete and has vulnerabilities
# DO NOT USE EXCEPT WITH LEGACY PROTOCOLS
ciphertext = public_key.encrypt(
    message,
    padding.PKCS1v15()
)

ciphertext_as_int = bytes_to_int(ciphertext)
recovered_as_int = simple_rsa_decrypt(ciphertext_as_int, private_key)
recovered = int_to_bytes(recovered_as_int)

print("Plaintext: {}".format(message))
print("Recovered: {}".format(recovered))
```

You can see that she is using the `cryptography` module to create the encryption. But she is using her own `simple_rsa_decrypt` operation for the decryption in order to preserve the padding.

This is what she sees:

```
Plaintext: b'test'
Recovered: b'\x02@&\x1cC\xb1\xe4\x0f\x14\xd9\x93oU
\x07\x1b\xfdC\xe1\xe2K\xeeP\xdd\x8b\x10\xf9cZJ\x0c
42\x8e\xbblZ\xfb\x80\x8b\xfcA?p\xac\xba\xf7I\x9e\x
11\x1cn&t\xb8\x15\xbfo\xfe\xcc\xdf\xe7=\xc2\x9e\x
ca<v\xcd\x9ep\xd8\x1c\xf6b2"\x8c\xc0\x1e\xb8\xdb\x
97\x89\xfauj\x8f``\x99m~,\x18h\xc2k6d~qr-\x0c\xb9\
xfe?\xf9\xf9\xa6o\x05\\ZV\xfd4?\x0e;y\xf3\xd3q\xb2
\x94\xf6\xf8~a\xc1eA\xe4\x14\xce\x82\xdcc\xbf4e\xa
e\xa3<"\xcb,L\xd8\xed\xca}\xeb\x82\xa67\x1a\xd1\xc
7)\x13\xc1D)\xe8\x05h\xbe/\x97\xdf>\xf0\xef\xeb\xe
4Q\xc2\x85(*\xdcE\x9ct\x08c0\xb1\x80la\x94_/2\xd4y
\xc7\x95\x01\x90@\xea\x92\xaa\xb8\x18!\xc7\xff\xab
\x03\xea\x8b\xa3\xb4\xf6\xf2\xd6GH\x98-fM\x1c\x99\
x84\x8d4\xaf"\x95\xa7XR(M\x836\xd4\x17\x99m\xa8\x1
a\xb3\x00test'
```

Eve notices that the actual message is at the end of the padding, consistent with the PKCS #1 v1.5 standard. (From the rest of this section, we will just say "PKCS.")

She does notice that the first byte of the recovered text is 2. That seems weird to her because the standard says that the padding should start with a 0 and a 2. Where did the initial 0 go?

Then Eve remembers! Of course! Because RSA works with *integers* instead of bytes, any leading zeros are wiped out. Fortunately, when RSA padding is used, the size of the bytes is fixed to the key size. Eve decides to update her conversion function with an optional parameter for minimum size,[8] shown in Listing 4-11.

Listing 4-11. Integer to Bytes

```
# Partial Listing: Some Assembly Required

# RSA Oracle Attack Component
def int_to_bytes(i, min_size = None):
    # i might be a gmpy2 big integer; convert back to a Python int
    i = int(i)
    b = i.to_bytes((i.bit_length()+7)//8, byteorder='big')
    if min_size != None and len(b) < min_size:
        b = b'\x00'*(min_size-len(b)) + b
    return b
```

Now properly updated, Eve writes her "fake" oracle that she will use just for testing. The code in Listing 4-12 performs a simple RSA decryption, converts the result to bytes (using the minimum size parameter we just implemented), and checks if the first and second bytes are 0 and 2, respectively. Make sure that the new `int_to_bytes` is working correctly. The old version will always drop the leading zero and the oracle will always report false.

[8] In most sources, because the size is fixed, it is specified as the expected size and the code checks to make sure it isn't too big.

Listing 4-12. Fake Oracle

```
# Partial Listing: Some Assembly Required

# RSA Oracle Attack Component
class FakeOracle:
    def __init__(self, private_key):
        self.private_key = private_key

    def __call__(self, cipher_text):
        recovered_as_int = simple_rsa_decrypt(cipher_text, self.
        private_key)
        recovered = int_to_bytes(recovered_as_int, self.private_key.
        key_size //8)
        return recovered [0:2] == bytes([0, 2])
```

With an oracle in place, Eve prepares to attack the algorithm described in the paper. The algorithm is described in four steps. We will review each one individually and develop the code incrementally.

Step 1: Blinding

Bleichenbacher's algorithm requires the blinding step both for setup and for "blinding" the message. However, the remarks section at the end of the algorithm explains that most of this is not necessary for our situation:

> Step 1 can be skipped if c is already PKCS-conforming (i.e., when c is an encrypted message). In that case, we set $s_0 \leftarrow 1$.

There are three values that get configured in this step. Because we are dealing with an already PKCS-padded encrypted message, we only need to set these values to the prescribed defaults:

$$\begin{aligned} c_0 &\leftarrow c(s_0)^e \pmod n \\ M_0 &\leftarrow [2B, 3B-1] \\ i &\leftarrow 1. \end{aligned}$$

Because $s_0 = 1$, we can reduce the first assignment to

$$c_0 \leftarrow c$$

Obviously, 1 to any power is still just 1, so neither the power nor the modulus has any effect.

The M parameter is going to be a list of lists of intervals (more on intervals in a second). This algorithm consists of repeated steps identified by i. M_0 records a list of intervals identified in the step identified by $i = 1$. In this case, there is only the single interval $[2B, 3B - 1]$.

What is B? As explained earlier in the paper, B is the number of legal values that have the proper padding. It is defined as

$$B = 2^{8(k-2)}.$$

Basically, k is the key size in bytes. So, if we're using a 2048-bit key, k = 256. But why subtract 2?

Let's break it down this way. For RSA with padding, our plaintext size in bytes is always supposed to be the same as the key size. If we're using a 2048-bit key, our padded plaintext must be 2048 bits (256 bytes) as well. That means that there are 2^{2048} possible plaintext values.

That isn't really true, though, is it? We know that the first two bytes must be 0 and 2, and that reduces the number of legal values by 2 × 8 = 16 bits. Thus, B is the maximum number of values for this key size when you account for the first two fixed bytes.

Returning to the intervals, what is $2B$ and $3B$? The intervals in this data structure represent legal values of PKCS numbers in which the actual plaintext message resides. Because the bytes at the beginning are the most significant bytes, the 0 has no impact on the integer number (e.g., 0020 = 20). But the 2 means that any legal number must be at a minimum $2B$ but must be less than $3B$.

Think about it this way. If I told you that a two-digit number must fall between 20 and 30, you would know that there are ten possible values that it could be. Moreover, you know that the minimum value is 2 × 10. This is the same idea.

The way this algorithm works is by narrowing down the legal interval until it is just a single number. That number is the plaintext message!

Eve decides to create a function for each of the steps of the algorithm. Given that there is state data that needs to be shared between these functions (e.g., B, M, etc.), she decides to use a class for storing state. The constructor takes a public key and an oracle.

Remember, the oracle simply takes a ciphertext as input and returns true if the ciphertext decrypts to a proper PKCS-padded plaintext.

Now, Eve writes the code for this step (step 1) of the algorithm. This step requires a ciphertext as input (c) and initializes the values of c_0, B, s, and M. Eve also copies n out of the public key in a convenience function called `_step1_blinding`, as in Listing 4-13.

Listing 4-13. RSA Oracle Attack: Step 1

```
# Partial Listing: Some Assembly Required

class RSAOracleAttacker:
    def __init__(self, public_key, oracle):
        self.public_key = public_key
        self.oracle = oracle

    def _step1_blinding(self, c):
        self.c0 = c

        self.B = 2**(self.public_key.key_size-16)
        self.s = [1]
        self.M = [ [Interval(2*self.B, (3*self.B)-1)] ]

        self.i = 1
        self.n = self.public_key.public_numbers().n
```

The value of B is computed directly from bits rather than converting from bytes. Everything else is computed exactly as described in the paper.

The `Interval` data structure in this code is created using the `collections.namedtuple` factory. Its two values are a (for lower bounds) and b (for upper bounds).

Step 2: Searching for PKCS-Conforming Messages

For this section, we need to dust off our mathematics from about multiplying RSA ciphertexts. Take a quick minute to review (4.3).

Conceptually, step 2 is about searching within the M_{i-1} intervals for new PKCS-conforming messages that are a multiples of the original plaintext message m and some other integer s_i.

Figure 4-2 depicts a (simplified) view of the PKCS-conformant space within all possible RSA ciphertext values. An RSA encryption ranges in output from 0 up to 2^k–1 where k is the key size in bits. Regardless of the key size, every number (in hexadecimal) begins with 1 of 16 digits 0 through `f`. The highlighted slice between 2 and 3 represents RSA ciphertext values that have proper PKCS padding. (This view is overly simplified because, in reality, the correct slice should be from 02 up to 03 out of a range from `00` to `ff`, so it would actually just be 1 slice out of 256.)

The reason the message space is shown as a ring is because we are dealing with modular (wrap-around) arithmetic. If you take two numbers within this space and multiply them together (modulo n), if the product is greater than n, it just wraps around.

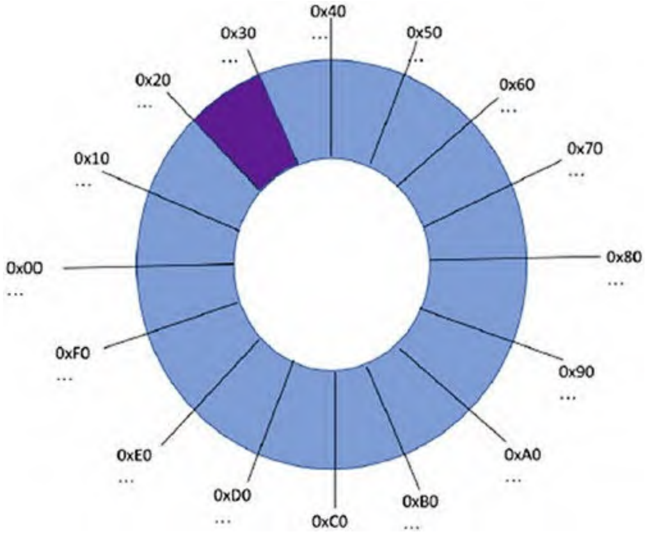

Figure 4-2. *Simplified view of PKCS-conformant space*

This brings us back to multiplying the plaintext message m by another number. In our simplified view in Figure 4-2, m must be inside the highlighted region somewhere. If we use modular multiplication, multiplying m by certain numbers (modulo n) will produce other numbers that have wrapped around that are also within the same region.

Of course, we don't know *exactly* where m is located because all we have is the encrypted version c. All we know is that, because it is PKCS-conformant, it is *somewhere* within the region. Similarly, because we don't know where m is, we also have no idea where a multiple of m will land in the ring. The exception, of course, is that using our oracle, we can determine if the multiple landed back inside the PKCS-conformant region!

Using the oracle, then, we will search for an s_i value that, when multiplied by m (modulo n), is PKCS-conformant and thus within the PKCS-conformant region of the RSA message space. We still won't know where m is, but knowing that it has a multiple that falls within a certain region introduces additional constraints on the interval that contains it. We'll talk more about those constraints and how to use them in step 3. But for now, let's find s_i!

Bleichenbacher breaks up finding s_i into three sub-steps:

1. 2*a* **Starting the search** is for the very first time we do this operation (i.e., when $i = 1$).
2. 2*b* **Searching with more than one interval left** is for rare cases when we have two intervals instead of just one.
3. 2*c* **Searching with one interval left** is for when there is just one interval and i is not 1. This should be all other cases.

Each of these sub-steps requires searching a range of possible s_i values to see if it produces a conformant ciphertext.

Specifically, for each candidate s_i, we encrypt it with RSA to produce c_i.

$$c_i = s_i^e \pmod n.$$

We multiply the encrypted s_i value by our original ciphertext c_0 to create a test cipher c_t. Because c_0 is the encryption of the unknown plaintext m_0[9], we get

$$\begin{aligned} s_t &= c_i c_0 \pmod n \\ &= s_i^e m_0^e \pmod n. \end{aligned}$$

We send c_t to the oracle to test if it is conformant. For our fake oracle, it simply uses the private key to decrypt the c_t and check if the plaintext starts with bytes 0 and 2. (Remember, to break Alice's messages, we won't have a private-key-enabled oracle. Instead, we will send the ciphertext to Bob and check for padding error message responses.)

[9] We were just calling this m, but to tie it to the c_0 value, we will refer to it as m_0.

Because each sub-step needs to be able to check a range of s_i values in this way, Eve decides to create a helper function for performing the search. It takes a starting value and an optional inclusive upper bound (as in Listing 4-14).

Listing 4-14. Find "s"

```
# Partial Listing: Some Assembly Required

# RSA Oracle Attack Component, part of class RSAOracleAttacker
  def _find_s(self, start_s, s_max = None):
    si = start_s
    ci = simple_rsa_encrypt(si, self.public_key)
  while not self.oracle((self.c0 * ci) % self.n):
    si += 1
    if s_max and (si > s_max):
      return None
    ci = simple_rsa_encrypt(si, self.public_key)
  return si
```

Using this helper function, the first two sub-steps are very straightforward. Step 2*a* requires testing all values of $s_i \geq n/(3B)$ until one of them is conformant. Eve encodes this step as shown in Listing 4-15.

Listing 4-15. Step 2a

```
# Partial Listing: Some Assembly Required

# RSA Oracle Attack Component, part of class RSAOracleAttacker
  def _step2a_start_the_searching(self):
    si = self._find_s(start_s=gmpy2.c_div(self.n, 3*self.B))
    return si
```

Notice that the starting s value is computed as $n/(3B)$ using the `c_div` function from the `gmpy2` module. Because we are working with such big numbers, we cannot trust Python's built-in floating point. Many of the values we are computing are just ranges and are not guaranteed to be integers, so fractional values are possible. The `gmpy2` module provides us with fast operations on very large numbers, including floating point.

The `c_div` function itself provides division rounding up toward the ceiling. So, for example, `c_div(3,4)` computes 3/4 and rounds up, returning 1.

Using these RSA concepts, this step searches for values of s_i that multiply c to *another* PKCS-conformant value. Specifically, for a candidate value of s_i, we RSA encrypt it, then multiply it by the original ciphertext. We use the ceiling because s_i must be an integer and must be greater than or equal to the starting value. Whether the starting value is a whole number or not, the next integer (i.e., ceiling) is the starting point for s_i.

Sub-step 2*b* is also quite easy to do. This sub-step deals with rare occurrences where the interval for m_0 gets split in two. When this happens, we iterate s_i forward until we find another conforming value (Listing 4-16).

Listing 4-16. Step 2b

```
# Partial Listing: Some Assembly Required

# RSA Oracle Attack Component, part of class RSAOracleAttacker
    def _step2b_searching_with_more_than_one_interval(self):
    si = self._find_s(start_s=self.s[-1]+1)
    return si
```

We will save every s value we find in the `self.s` array for being able to access these values. In truth, we only ever need the previous value, but we use this idiom to match the way the paper is written.

Finally, the last sub-step, 2*c*, is a bit more complicated. It requires searching for s across a range of possible values. Recall that there is only one interval found in the previous step and we take the lower bound as a and the upper bound as b. Next, we must iterate through r_i values:

$$r_i \geq 2\frac{bs_{i-1} - 2B}{n}.$$

We use these r_i values to bound both sides of the s_i search:

$$\frac{2B + r_i n}{b} \geq s_i < \frac{3B + r_i n}{a}.$$

What we are doing here is picking s_i values within a particular range that will help us continue to narrow down the solution. Bleichenbacher explains in his paper why these bounds work, and we will not repeat his comments here. When we talk about step 3, we will give some further intuition on the entire algorithm that will help to clarify what is happening.

In the meantime, Eve encodes this algorithm as Listing 4-17.

Listing 4-17. Step 2c

```
# Partial Listing: Some Assembly Required

# RSA Oracle Attack Component, part of class RSAOracleAttacker
    def _step2c_searching_with_one_interval_left(self):
        a,b = self.M[-1][0]
        ri = gmpy2.c_div(2*(b*self.s[-1] - 2*self.B),self.n)
        si = None

        while si == None:
            si = gmpy2.c_div((2*self.B+ri*self.n),b)

            s_max = gmpy2.c_div((3*self.B+ri*self.n),a)
            si = self._find_s(start_s=si, s_max=s_max)
            ri += 1
        return si
```

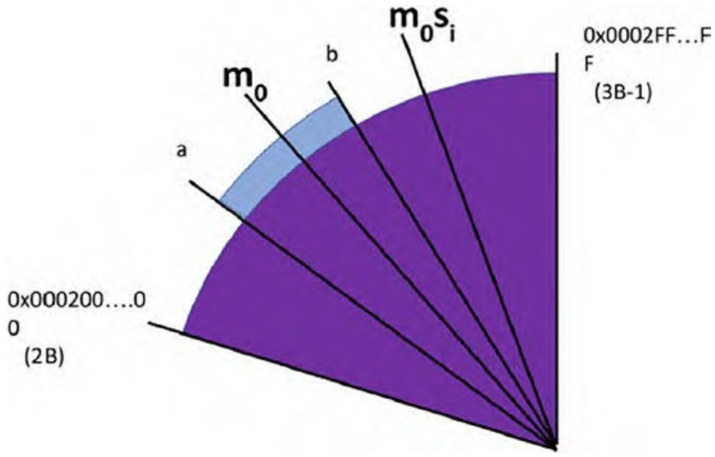

Figure 4-3. *Depiction of Bleichenbacher's attack*

As with previous computations, division is handled using `gmpy2.c_div`. This is very important. If you just use Python's division operators, you are likely to get incomplete results.

Step 3: Narrowing the Set of Solutions

Once an s_i value has been found from step 2, we update our bounds on the location of *m*. Before walking through the math, let's talk about what is going on in this algorithm.

In Figure 4-3 we are again visualizing the slice of the RSA message space ring that contains legitimate PKCS-padded values. The lower bound of this space is numbers beginning with 000200…00 and the inclusive upper bound is 0002FF…FF. The plaintext message m_0 is *somewhere* in here. At the start of the algorithm, we have no idea where.

However, for each s_i value we find that is conformant, we learn of a new value m_0s_i that is within this region as well (wrapping around because of modulo arithmetic). The fact that we know that m_0s_i (modulo *n*) falls within a particular range introduces new constraints on where m_0 can be. We are able to use these constraints to calculate a new interval *a* to *b* within which m_0 *must* be.

Once we update the bounds, we can repeat the process using *new* values of s_i that further tighten the bounds. Eventually, the bounds will restrict m_0 to being a single value. *That* is the plaintext we're looking for!

Hopefully this intuition will help even if the following formulas don't make much sense. Or, it will be helpful if you do try to tackle Bleichenbacher's paper. In any event, we compute the new upper and lower bounds as follows.

For each *a*, *b* interval in the previous M_0 (there will usually be one, but sometimes two), find all integer values of *r* such that

$$\frac{as_i - 3B + 1}{n} \geq r \leq \frac{bs_i - 2B}{n}.$$

For each of these values of *a*, *b*, and *r*, we calculate a new interval. First, we calculate a lower-bound candidate as follows:

$$a_i = \frac{2B + rn}{si}.$$

and an upper-bound candidate

$$b_i = \frac{3B - 1 + rn}{si}.$$

We define a new interval as $[max(a, a_i), min(b, b_i)]$.

The set of all intervals is inserted into M_i. Again, there is typically only one interval.

Eve encodes this step of the algorithm as in Listing 4-18.

Listing 4-18. Step 3

```
# Partial Listing: Some Assembly Required

# RSA Oracle Attack Component, part of class RSAOracleAttacker
    def _step3_narrowing_set_of_solutions(self, si):
        new_intervals = set()
        for a,b in self.M[-1]:
            r_min = gmpy2.c_div((a*si - 3*self.B + 1),self.n)
            r_max = gmpy2.f_div((b*si - 2*self.B),self.n)

            for r in range(r_min, r_max+1):
                a_candidate = gmpy2.c_div((2*self.B+r*self.n),si)
                b_candidate = gmpy2.f_div((3*self.B-1+r*self.n),si)

                new_interval = Interval(max(a, a_candidate), min(b, b_candidate))
                new_intervals.add(new_interval)
        new_intervals = list(new_intervals)
        self.M.append(new_intervals)
        self.s.append(si)

        if len(new_intervals) == 1 and new_intervals[0].a == new_intervals[0].b:
            return True
        return False
```

In this code, note that `r_max` is calculated using `f_div`. This computes division rounding to the floor instead of the ceiling. We use this value because *r* is an integer and must be less than or equal to the value.

Once the intervals are computed, the code adds them to the `self.M` data structure and adds the s_i value to `self.s`.

Finally, it checks to see if we've found a solution. Eve is getting ahead of herself here. This is part of step 4, but it was simply more convenient to put it here.

Step 4: Computing the Solution

As hinted at in previous sections, this algorithm has termination criteria. Hopefully, it is fairly obvious considering the previous discussion. Either

- M_i contains only one interval, or
- The upper and lower bound in the interval of M_i are the same.

In short, we terminate when the interval that bounds the location of *m* is reduced to a single number.

We have already seen Eve's code for checking this condition at the end of step 3. Bleichenbacher's step 4 also deals with a more general problem than ours and includes steps that are unnecessary for when s_0 is 1. Recall that for processing RSA encryption messages where the plaintext was already PKCS-padded, s_0 was set to 1.

Although it's somewhat unnecessary, for sake of completeness and consistency, Eve does create a method for step 4 (Listing 4-19).

Listing 4-19. Step 4

```
# Partial Listing: Some Assembly Required

# RSA Oracle Attack Component, part of class RSAOracleAttacker
    def _step4_computing_the_solution(self):
        interval = self.M[-1][0]
        return interval.a
```

That's it! That's the entire algorithm! Eve combines these steps into Listing 4-20's attack method.

Listing 4-20. Attack!

```
# Partial Listing: Some Assembly Required

# RSA Oracle Attack Component, part of class RSAOracleAttacker
    def attack(self, c):
        self._step1_blinding(c)

        # do this until there is one interval left
        finished = False
        while not finished:
            if self.i == 1:
                si = self._step2a_start_the_searching()
            elif len(self.M[ -1]) > 1:
                si = self._step2b_searching_with_more_than_one_interval()
            elif len(self.M[-1]) == 1:
                interval = self.M[-1][0]
                si = self._step2c_searching_with_one_interval_left()

            finished = self._step3_narrowing_set_of_solutions(si)
            self.i += 1

        m = self._step4_computing_the_solution()
        return m
```

Please note that the `attack()` method's input is the ciphertext, but it must already be in integer form. Don't forget to call `bytes_to_int()` on the ciphertext first!

EXERCISE 4.12. RUN THE ATTACK!

Take the preceding code and run some experiments with breaking RSA encryption with PKCS padding. You should use the `cryptography` module to create the encrypted message, convert the encrypted message to an integer, and then use your attack program (and fake oracle) to break the encryption. To begin with, test your program on RSA keys of size 512. This breaks faster and will enable you to validate your code sooner.

EXERCISE 4.13. TAKING THE TIME

How long does the attack take? Instrument your code with timing checks and a count of how many times the oracle function is called. Run the attack on a suite of inputs and determine the average amount of time required to break keys of sizes 512, 1024, and 2048.

EXERCISE 4.14. STAYING UP TO DATE

Despite the fact that this attack is over 20 years old, it continues to haunt the Internet. Do a little Google searching and find out about the current state of this attack both in terms of prevention and updated variants. Make sure to find out about the ROBOT attack. We'll talk about this one again when we discuss TLS.

Additional Notes About RSA

We've spent a lot of time on RSA in this chapter, and we haven't even gotten into much of how it is actually used in practice. RSA, like most asymmetric ciphers, is almost never used to encrypt messages like we had Alice and Bob do throughout the chapter. When it is used, it is typically used to encrypt a session key for a *symmetric* cipher, or for signatures.

It is, however, critical to understand how asymmetric ciphers work and how they can be broken. Despite all of its weaknesses, RSA is still widely used, often incorrectly. Walking through the exploits and vulnerabilities in this chapter should help put you on the right path.

Here are a few other items for consideration.

Key Management

As with all ciphers, much of their security comes down to correctly creating and safeguarding keys.

When creating an RSA key, make sure to use a library. Do not try to generate the public and private keys yourself. At the same time, keep tabs on any bug reports for the library you do use. For example, some libraries have been found to generate RSA private keys without sufficient randomness, thus producing private keys that were vulnerable to various attacks. You can't possibly anticipate all of the things that will go wrong, or when the library or algorithm you use will be exposed as vulnerable, so you must "maintain" your cryptography by keeping up to date on known vulnerabilities.

Vulnerabilities can be system-specific. The ROCA vulnerability, for example, was largely confined to certain hardware chips.

It is also important to use the proper parameters when creating an RSA key. The key size should typically be at least 2048 bits unless legacy constraints force you to choose something smaller. And the value of the public exponent *e* should always be 65537.

You must also be careful to guard and protect private keys and their secrets. Obviously the private key itself should be stored securely and with appropriate permissions. Your private key should, at the very least, be stored with absolutely minimal permissions on the file system. A very sensitive key might need to be stored offline.

You should also consider storing the private key in encrypted form. This will require a password to decrypt the key which can have its own set of difficulties in a fully automated system. However, properly used, it can reduce the risk of a private key being compromised if an attacker gains access to the host system.

Moreover, the private key is made up of a number of component values. In our examples, we could think of *d* as the private key because that is the value we use to actually decrypt. But in addition to *d*, care must also be taken not to expose the secrets used to generate it. For example, the modulus *n* is not, itself, secret, but the two large primes, *p* and *q*, that generated it are.

There are additional values generated when creating a private key that will compromise security if disclosed. Along with *p* and *q*, these values are not strictly necessary after the key is generated, as everything can be computed from *e*, *d*, and *n*. However, most libraries do keep them as part of the private key both in memory and on disk. You should read your library's documentation about private key generation and follow recommended handling procedures.

One of the weaknesses of asymmetric cryptography is the inability to "revoke" a private key. If Bob's private key is compromised, how does Alice know to stop sending data encrypted under the associated public key? In practice, your RSA keys will probably be used in conjunction with certificates, which can include a hierarchy of certificates and keys allowing some keys to be less sensitive than others and also include an expiration date to limit the exposure of a compromised key. More is said on that elsewhere.

EXERCISE 4.15. FACTORING RSA KEYS

In this section, we recommended using 2048-bit keys. For this exercise, do an Internet search to find out the current size of keys that can easily be factored. For example, do a search for "factoring as a service" and see how much it costs to factor a 512-bit key.

EXERCISE 4.16. ROCA VULNERABLE KEYS

Unless your RSA keys are being generated by certain RSA hardware modules, the keys you have generated for the exercises in this chapter should not be vulnerable to ROCA, but it never hurts to check. For this exercise, visit the online ROCA vulnerability checking site at `https://keychest.net/roca#/` and test a couple of keys.

Algorithm Parameters

If there is one thing that you should take away from this chapter, it is this: *pay special attention to RSA's padding parameter*. As of the time of this writing, you should use the OAEP padding scheme for encryption operations and the PSS padding scheme for signatures. Do *not* use PKCS #1 v1.5 unless it is absolutely necessary for legacy applications.

Quantum Cryptography

We don't have the space to delve into quantum cryptography in this book, but we can't close out our discussion of RSA without mentioning it. When quantum computing arrives, most of our current asymmetric algorithms will become breakable. RSA is already vulnerable to a number of contemporary attacks, but when quantum computing becomes viable, it will be thoroughly broken. Thus, within the next decade or so, RSA will be completely useless.

Really Short Addendum

If there is one thing to get out of this chapter, it is this: parameters matter, and correct implementations are subtle and evolve over time. The intuition for how asymmetric encryption works and can be used is simple to explain, but there are numerous details that can make one implementation safe and another highly vulnerable.

Choose the right tool for the job, and choose the right parameters for the tool.

Oh, and breaking stuff is fun!

CHAPTER 5

Message Integrity, Signatures, and Certificates

In this chapter, we will be talking about "keyed hashes" and how asymmetric cryptography can be used to provide not only message privacy but also message *integrity* and *authenticity* via digital signatures. We will also be talking about how certificates differ from keys and why that distinction is important. Let's dive right into an example and some code!

An Overly Simplistic Message Authentication Code (MAC)

Checking in with Alice and Bob, our East Antarctic espionage duo has had some trouble on their most recent adventure in adversarial territory to the west. Apparently, Eve managed to intercept a few communications sent between them. The messages, encrypted with symmetric encryption, were unreadable, but Eve figured out how to *alter* them, inserting some false instructions and information. Alice and Bob, acting on false information, were almost trapped in an ambush. Fortunately for them, a bunch of ice melted due to global warming, and they managed to swim home to safety!

Quick to learn from their close call, they spent a little time at headquarters drying off and devising new communication mechanisms to prevent the unauthorized modification of their encrypted data.

Eventually, the East Antarctica Truth-Spying Agency (EATSA) discovered a new concept: "message authentication codes" or "MACs."

A MAC, Alice and Bob are told, is any "code" or data transmitted along with a message that can be evaluated to determine if the message has been altered. This is an informal definition for intuition purposes. Be patient while Alice and Bob work through

S. J. Nielson and C. K. Monson, *Practical Cryptography in Python*,
https://doi.org/10.1007/978-1-4842-4900-0_5

this introductory and incorrect starting point. The basic idea for this overly simplistic MAC is this:

1. The sender computes a code C_1 using a function $f(M_1)$ for a given message M_1.
2. The sender transmits M_1 and C_1 to the recipient.
3. The recipient receives the data as M and C, but does not know if they have been modified.
4. The recipient recomputes $f(M)$ and compares the output to C to verify that the message is unaltered.

Suppose that Eve intercepts M_1 and C_1 sent by Alice to Bob. If Eve wants to change the message M_1 to M_2, she must *also* recompute $C_2 = f(M_2)$ and send both M_2 and C_2 to Bob. Otherwise, Bob will detect that something has been changed because $f(M)$ and C will not match.

If you are asking, “So what? Eve can just recompute the MAC, right?” then you are seeing the problem with our overly simplistic setup. We have to assume that Eve has everything *except the key*, but this example also assumes she does not have *f*. We will fix that shortly. Stay tuned!

For now, Alice and Bob are just going to assume that Eve can’t compute, or easily compute, the function *f*. If this assumption is true (which it isn’t in reality), then just about any mechanism for creating a fingerprint will work. The East Antarctican spying agency decides to send the message *hash* as an attachment to the message. Thus, the MAC is a hash in this case.

Let’s dive into some code to see how this simple idea comes together. While we’re at it, we can combine our new fake MAC technology with some symmetric encryption from Chapter 3. This is demonstrated in Listing 5-1.

Listing 5-1. Fake MAC with Symmetric Encryption

```
# THIS IS NOT SECURE. DO NOT USE THIS!!!
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
import os, hashlib
```

```

class Encryptor:
    def __init__ (self, key, nonce):
        aesContext = Cipher(algorithms.AES(key),
                             modes.CTR(nonce),
                             backend=default_backend())
        self.encryptor = aesContext.encryptor()
        self.hasher = hashlib.sha256()

    def update_encryptor(self, plaintext):
        ciphertext = self.encryptor.update(plaintext)
        self.hasher.update(ciphertext)
        return ciphertext

    def finalize_encryptor(self):
        return self.encryptor.finalize() + self.hasher.digest()

key = os.urandom(32)
nonce = os.urandom(16)
manager = Encryptor(key, nonce)
ciphertext = manager.update_encryptor(b"Hi Bob, this is Alice !")
ciphertext += manager.finalize_encryptor()
```

Recall that “counter mode” requires no padding and that in our previous examples the “finalize” functions really didn’t do much. But now, when we finalize our manager, it not only finalizes encryption, it also returns the computed hash as the last few bytes to be appended to the encrypted data. Thus, the final encrypted message has our simple MAC tacked onto the end of it.

EXERCISE 5.1. TRUST BUT VERIFY

Finish out the code of the simple encryption plus hash system and add a decryption operation. The decryption operation should, upon finalization, recompute the hash of the ciphertext and compare it to the hash that was sent over. If the hashes don’t match, it should raise an exception. Be careful! The MAC is not encrypted and should not be decrypted! If you don’t think carefully about this, you might decrypt data that doesn’t exist!

EXERCISE 5.2. EVER EVIL EVE

Go ahead and "intercept" some of the messages encrypted by the code you wrote in this section. Modify the intercepted messages and verify that your decryption mechanism correctly reports an error.

MAC, HMAC, and CBC-MAC

Alice and Bob were told by their support people that any mechanism for authenticating a message is a message authentication code (MAC). As we hinted, this is not a complete definition. A real MAC also requires a *key*.[1]

We've used keys for encryption, but so far we haven't used them for much else. A MAC key, as you might have guessed, isn't really related to encryption at all. Rather, it ensures that the message authentication code can *only* be computed by parties that know the key.

In our example, Alice and Bob had to assume that Eve couldn't compute the function $f(M)$. That, of course, isn't reasonable. Alice and Bob used SHA-256 to derive a fingerprint, so obviously Eve can use it to compute her own authentication code as well. Assuming that she can deterministically alter the ciphertext, as we saw in the previous chapter that she could under certain circumstances, she could insert a new message *and* a new fake MAC.

A real MAC, however, which depends on a key, *cannot* be generated by Eve unless she has compromised the key! Remember, good security means that *everything* can be known *except the key* and it still works right.[2]

A MAC protects the *integrity* of the message. An attacker without a key cannot undetectably alter the data. Furthermore, if the key remains secret, the MAC also provides *authenticity*: the receiver knows that only the other person sharing the key could have sent the MAC because only a person with a key could have generated a legal MAC at all.

While there are many MAC algorithms, we will look at two easy-to-understand approaches: HMAC and CBC-MAC. These algorithms do a good job of teaching how and why a MAC works. They are useful in practice as well.

[1] This is still just an informal definition. Formal definitions exist for the persnickety [11, Chap. 9].

[2] Kerckhoff's principle strikes again!

HMAC

An HMAC is a "hash-based message authentication code." In fact, you already know the most complicated characteristic of an HMAC: hashing. An HMAC is mostly just a hash that is *keyed*.

What does it mean to be "keyed"? To illustrate, let's first review standard cryptographic hashes that are not keyed. For such hashes, if the input doesn't change, neither does the output. They are fully deterministic based only on a single input: the message contents. If you revisit the exercise "GOOGLE KNOWS!" in Chapter 2, you will recall that we can actually enter some hash values into Google and find matching inputs.

Pull up a Python shell and test this one or two more times:

```
>>> import hashlib
>>> hashlib.sha256(b"hello world").hexdigest()
'b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9'
>>> hashlib.sha256(b"happy birthday").hexdigest()
'd7469a66c4bb97c09aa84e8536a85f1795761f5fe01ddc8139922b6236f4397d'
```

The SHA-256 outputs for "hello world" and "happy birthday" are *always* these values on every computer for the rest of eternity. They will *never change*. You can verify this by running the code yourself. The SHA-256 definition demands it. You can also try searching for the hashes online.

To repeat, with an unkeyed algorithm the same input *always* produces the same output.

When an algorithm is *keyed*, it means that the output is dependent on both the input and a key. But how can a hashing algorithm be keyed?

Conceptually, it is actually pretty easy. Because even a minor change to the input of a hashing algorithm completely changes the output, we can have the key be part of the input itself!

While the following example is *not* a real HMAC and is *not* considered sufficiently secure, it illustrates the idea:

```
>>> import hashlib
>>>
>>> password1 = b"CorrectHorseBatteryStaple" # See XKCD 936
>>> password2 = b"LiverKiteWorkerAgainst"
>>>
>>> # This is not really HMAC, it is for illustration ONLY:
```

```
>>> hashlib.sha256(password1 + b"hello world").hexdigest()
'ca7d4abd13bceb305eef2738e3592da77ed826aa1665ba684b80f36bd7522b32'
>>>
>>> hashlib.sha256(password2 + b"hello world").hexdigest()
'b22786bc894c8bb27d1e7e698a9bddfd6b95f35dcd063e37d764fa296216408a'
```

In this example, we used human-readable passwords as the keys. We hashed the input "hello world" two more times, but inserted a different password each time as a *prefix*. Basically we used the key to change what we were hashing. Each password results in a completely different output, meaning that the only way for someone to recreate the output MAC for the message "hello world" is to also *know the password* (or break it through brute force). As with any other cryptographic algorithm, the key/password must be both sufficiently large and sufficiently random.

Speaking of size, it is worth noting that the size of the password is not a factor in how effectively it changes the hash output. Do you recall the avalanche principle? Changing a single *bit* of input to a hash function completely changes the output hash value. You could have a terabyte document, change only a single character of it, and produce a new hash that has no relationship to the unaltered document's hash. Similarly, your password could be a single character, and it would effectively "scramble" the output for any given input, no matter how large. All you need to worry about is that your password length (and randomness) is sufficiently strong to prevent brute-force attacks.

EXERCISE 5.3. BRUTE FORCE AGAIN

You should already have done some brute-force attacks in previous chapters, but it's important to repeat the exercise until you develop intuition for the concept. Using our preceding fake HMAC, have the computer generate a random password of specific sizes and use brute-force methods to find out what it is. To be more specific, assume that you already know what the message is (e.g., "hello world," "happy birthday," or a message of your choosing). Write a program to create a random password of characters, prepend the password to the message, and then print out the MAC (hash). Take the output and iterate through all possible passwords until you find the right one. Start with a simple test of a single-letter character, then try two characters, and so forth. Mix things up by using different sets of characters such as all lowercase, lowercase and uppercase, either case plus numbers, and so forth.

EXERCISE 5.4. BRUTE FORCE FOUR-WORD PASSWORDS

Repeat the previous exercise. But instead of using letters drawn from a source of letters, use words drawn from a source of words. Find or create a text file with a list of common words. It should be at least 2000 words. Using this dictionary, create passwords by picking *n* random words. Attempt to brute force this password by trying every possible combination from the dictionary. Start with *n* = 1 (one-word password) and go up from there.

Even the preceding approach isn't quite good enough, so let's talk about the real HMAC. We have repeatedly said that merely prepending the password is not sufficiently secure. "HMAC" is the official name given to an algorithm defined in a standard document called "RFC 2104." If you haven't ever looked at an RFC before, these are documents from the Internet Engineering Task Force (IETF) that represent standards, best practices, experiments, and discussions for Internet protocols and algorithms. They are all freely available and can be found online. RFC 2104 can be found at `https://tools.ietf.org/html/rfc2104`.

The abstract for the document states:

> This document describes HMAC, a mechanism for message authentication using cryptographic hash functions. HMAC can be used with any iterative cryptographic hash function, e.g., MD5, SHA-1, in combination with a secret shared key.

That part should already make sense. The experiments we already did used SHA-256 and a secret shared key, but we obviously could have used SHA-1 or MD5. As a reminder, though, those hash algorithms are considered "broken" and should not be used except as necessary with legacy applications.

Returning to page 3 of the RFC, we see that once a hashing function *H* is picked, the HMAC over an input text is computed, thus

H(K XOR opad, H(K XOR ipad, text))

Let's take a look at each of these terms. We already know *H*; that's the underlying hash function. The term "text" refers to the input, but does not have to be composed of readable text characters any more than any "plaintext" message needs to be: it can be arbitrary binary data. Oh, and we need to address the commas. Because *H* is a function, you might be tempted to think that this definition is showing a hashing function that takes two parameters. But in this definition in the RFC, the comma can be thought of as concatenation. As in all of our other examples, a hash function only takes a single input.

The term *K* refers to the key, but it can't be just anything. The RFC has a number of requirements for the key that will often require some pre-processing. Most of these requirements are related to the block size of *H*. Recall from Chapter 3 that we used the term "block size" with block ciphers to describe the size of data that the block cipher operates on at one time. AES, for example, has a block size of 16 bytes (128 bits). Hashing algorithms can hash any size of input, so what is the block size of a hash algorithm?

In actuality, hashing functions typically operate on one block at a time, but feed the hash output from one chunk into the hashing computations of the next. SHA-1, for example, has a 64-byte (512-bit) block size, while SHA-256 has a 128-byte (1024-bit) block size. The RFC refers to the block size of *H* as *B* (bytes).

The first requirement for our key is that if it is *shorter* than the block size *B*, it has to be padded with zeros until it is *B* bytes long.

The second requirement is that if the key is *longer* than *B*, it is first reduced by hashing the key with *H*. Don't let this surprise you. We will use *H* multiple times in a single HMAC operation.

In summary, if *K* is too short, it is padded with zeros, and if *K* is too long, *H*(*K*) is used instead.

The eagle-eyed reader will notice that the length of a hash may be *also* be shorter than the block size. SHA-1's hash is 20 bytes long and its block size is 64 bytes. SHA-256's hash is 32 bytes long but its block size is 128 bytes. After reducing the key that is *too long* with the hashing function, it will generally be *too short* and will then require padding.

In the end, we should have a key that is exactly *B* bytes long.

Next, we need to compute $K \oplus$ ipad (XOR). The term "ipad" stands for "inner padding" because this is the inner hashing operation in the HMAC. The RFC defines ipad as "the byte 0x36 repeated B times" and "opad" as "the byte 0x5c repeated B times." The values chosen for ipad and opad were picked arbitrarily. What is most important is that they are different.

The reasons for the pads go beyond the scope of this book, but they give HMAC some extra security in case the underlying hash function is broken. So, for example, these paddings made HMAC-MD5 relatively strong even after MD5 was shown to be broken. That's helpful, but not a good reason to use HMAC-MD5 for new applications. Please don't. HMAC's padding means that HMAC-SHA256 will be a reasonably strong MAC even if someone finds a vulnerability in the SHA-256 hashing function, which can help keep existing uses (that might not be easily upgraded to a better hash function immediately) relatively secure.

The computation of $K \oplus$ ipad is pretty easy because they are the same size. The subsequent value is prepended to the input "text," and the combined data is hashed by H. We have now computed $H(K \oplus (\text{ipad}, \text{text}))$. Again, this is the inner hash computation.

Now, for the outer hash, we compute $K \oplus$ opad. The subsequent value is prepended to the output of the inner hash, and the aggregated bytes are hashed again. The hash of the outer function is the HMAC of the input text keyed on K.

Fortunately for you, cryptographic libraries almost always have HMAC as a primitive.

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes, hmac
>>>
>>> key = b"CorrectHorseBatteryStaple"
>>> h = hmac.HMAC(key, hashes.SHA256(), backend=default_backend())
>>> h.update(b"hello world")
>>> h.finalize().hex()
'd14110a202b607dc9243f83f5e0b1f4a1e59fba572fc5ea5f41d263dd4e78608'
```

Why go to all the trouble of learning how HMAC works on the inside, rather than just learning how to use a supplied library? There are a few reasons. First, it's good to have at least a little bit of an idea of how things work. It helps with intuition and reasoning about when to use it and why.

Second, and perhaps most important, it is to remind you that YANAC (You Are Not A Cryptographer… yet!). You must remember this principle! Use cryptographic libraries as much as possible and do not try to come up with your own "clever" algorithm. Take a look at HMAC again. It's built on some of the same concepts as simply prefixing an input with a key, but has much higher complexity. That complexity comes from deeper and subtler goals, including forward security in the event of a broken hash function. That complexity is not arbitrary; the HMAC operation was based on a research paper by cryptographers that *mathematically proved* certain security properties. Unless you are a cryptographer who publishes your work (often with formal proofs) for public peer scrutiny, test, and debate, then you really should not be creating your own algorithms except for the purposes of education or demonstration.

EXERCISE 5.5. TEST PYTHON'S HMAC

Although you should not roll your own crypto, it doesn't mean you shouldn't verify implementations! Create your own implementation of HMAC following the instructions from RFC 2104 and test some inputs and keys with both your implementation and Python's `cryptography` library's implementation. Ensure that they produce the same outputs!

CBC-MAC

HMAC is a very popular MAC and is used, for example, in TLS, but there are other ways to create MACs. For example, we can take what you learned in Chapter 3 about cipher block chaining (CBC) mode as another way to derive a secure MAC.

Let's quickly introduce some new terminology. A MAC is also sometimes called a "tag." When we create a MAC of a message, we can call it the "tag" of the message; it's like a tag on a gift or a piece of clothing: it's a little bit of information that is attached to the main article. In mathematical notation, a tag is often denoted t. Thus, a MAC over message m_1 produces a tag t_1, and the pair (m_1, t_1) is transmitted to the receiver for verification.

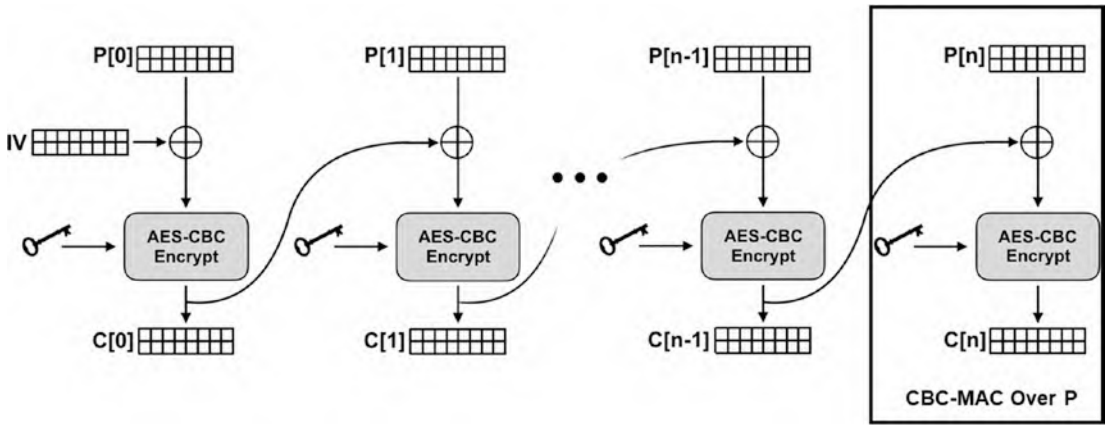

Figure 5-1. *Because all of the message impacts the value of the last encrypted block of data, C[n] is a MAC over all of P... with a few flaws.*

Recall that when encrypting with AES, we were limited to encrypting 128 bits at a time. If we encrypted each 128-bit block independently, there was still information that could "leak" through about the overall data. For example, large image features might still be recognizable. One of the solutions to the problem was to "chain" the encryption so that the input from one block carried over and influenced the encryption of the next. In other words, a change in a bit at the beginning would have a cascading effect all the way down to the very last block.

Put another way, the very last block of ciphertext is determined by the value of *every other block* in the chain: any changes anywhere in the input will be reflected in the last block! That makes the last block of a CBC encryption mode a MAC over the entire data as shown in Figure 5-1.

Hopefully as you have learned by this point in the book, all cryptography comes with limitations and critical parameters. As with HMAC, we will do some naive examples first to see both the basic concepts behind the CBC-MAC algorithm and how naive approaches are exploitable.

Let's start by taking a message and running it through AES-CBC encryption. For security reasons that we will explain shortly, we will fix the initialization vector to zero. In order to have our messages be a multiple of a block size, we will also use the same PKCS7 padding used for encryption. We will need some full block messages MAC'd without padding to simplify the next exercise, so we include a flag for turning padding off.

Listing 5-2. Fake MAC with CBC

```
# WARNING! This is a fake CBC-MAC that is broken and insecure!!!
# DO NOT USE!!!
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os

def BROKEN_CBCMAC1(message, key, pad=True):
    aesCipher = Cipher(algorithms.AES (key),
                       modes.CBC(bytes(16)), # 16 zero bytes
                       backend=default_backend())
    aesEncryptor = aesCipher.encryptor()

    if pad:
        padder = padding.PKCS7(128).padder()
        padded_message = padder.update(message)+padder.finalize()
    elif len(message) % 16 == 0:
        padded_message = message
    else:
         raise Exception("Unpadded input not a multiple of 16!")
    ciphertext = aesEncryptor.update(padded_message)
    return ciphertext[-16:] # the last 16 bytes are the last block

key = os.urandom(32)
mac1 = BROKEN_CBCMAC1(b"hello world, hello world, hello world, hello world", key)
mac2 = BROKEN_CBCMAC1(b"Hello world, hello world, hello world, hello world", key)
```

The code in Listing 5-2, although not secure, does show the basic concept behind the MAC. A piece of data is first padded and then encrypted. No matter how long it is, however, the last block (16 bytes) is determined by all of the preceding input. Change the first letter from an “h” to an “H,” and the MACs are completely different.

Nevertheless, it can be exploited. Recall that a MAC must be *unique* for a given message and key pair. If an attacker can generate the same MAC for a different message with the same key, the MAC algorithm is broken.

It turns out that for this naive version of CBC-MAC, you can do exactly that. Let's do it in code first and see if you can guess what's going on. Note that Listing 5-3 is intended to be combined with Listing 5-2.

Listing 5-3. MAC Prepend Attack

```
# Partial Listing: Some Assembly Required

# Dependencies: BROKENCBCMAC1
def prependAttack(original, prependMessage, key):
    # assumes prependMessage is multiple of 16
    # assumes original is at least 16
    prependMac = BROKEN_CBCMAC1(prependMessage, key, pad = False)
    newFirstBlock = bytearray(original [:16])
    for i in range (16):
        newFirstBlock[i] ^= prependMac[i]
    newFirstBlock = bytes(newFirstBlock)
    return prependMessage + newFirstBlock + original [16:]

key = os.urandom(32)
originalMessage = b"attack the enemy forces at dawn!"
prependMessage = b"do not attack. (End of message, padding follows)"
newMessage = prependAttack(originalMessage, prependMessage, key)
mac1 = BROKEN_CBCMAC1(originalMessage, key)
mac2 = BROKEN_CBCMAC1(newMessage, key)
print("Original Message and mac:", originalMessage, mac1.hex())
print("New message and mac     :", newMessage, mac2.hex())
if mac1 == mac2:
    print("\tTwo messages with the same MAC! Attack succeeded!!")
```

The two MACs produced by Listing 5-3 are *identical*. Our attack prepends another message of our choosing to the original and *also* corrupts the first block. The only restriction on the prepended message is that it must also have the CBC-MAC value for the prepended message under the same key. We turned off padding for this prepended message to make the attack a little easier, but this is only for our convenience and not a prerequisite for the attack to succeed.

Sadly for the attacker, the original message requires modification to the first block; otherwise, the attack could have been even worse. The attacker could then create messages that say "do not attack the enemy forces at dawn!" The attacker also cannot scrub any of the data beyond the first block. In running the code, you probably noticed that "forces at dawn!" was still readable in the new message. Even so, this is still pretty bad: we added an entirely different message without changing the value of the MAC!

For this simple example, where we assume that a human is reading the output, we hope that our message that says the rest of the data is padding will be enough to convince the sender not to read further. In real attacks, transmitted data lengths and other similar mechanisms can often be used to achieve the same effect. If we are successful, we can basically send arbitrary message with the original MAC.

What went wrong? Before we give you an explanation, see if you can figure it out yourself. You might need to revisit how CBC mode works. If you need an additional hint, remember that $A \oplus B \oplus B = A$.

Let's work through it together anyway. Suppose that we have a message M composed of arbitrary blocks of data m_1 through m_n. In the formulas that follow, let E represent the AES encryption operation and let t be the CBC-MAC tag computed over the data:

$$t = E(m_n \oplus E(m_{n-1} \oplus \ldots E(m_2 \oplus E(m_1, k), k) \ldots, k), k)$$

Notice that m_1, the first block of the message, is encrypted by AES under key k and the output is XORed with m_2 before being encrypted.

Suppose that we prepended a message P that was exactly one block in length. How would that change things? The CBC-MAC would obviously produce something different because we're changing the first computation:

$$t_P = E(m_n \oplus E(m_{n-1} \oplus \ldots E(m_2 \oplus E(m_1 \oplus E(P, k), k), k) \ldots, k), k)$$

The outcome is as it should be. Changing the message (i.e., prepending a new block) changed the tag. But what if we already knew the output of the AES encryption of the prepended block $E(P, k)$? Let's call it C. If $E(P, k) = C$, then we can prepend P to the chain without changing the final tag *if* we also corrupt the original first block m_1 to be $m_1 \oplus C$.

$$t = E(m_n \oplus \ldots E(m_2 \oplus E(m_1 \oplus C \oplus E(P, k), k), k) \ldots, k), k)$$

When CBC operates on this corrupted chain, it attempts to XOR the encrypted output of the prepended block (C) into the plaintext of the corrupted first block ($m_1 \oplus C$). But the corrupted first block already has the XOR of C mixed in, the C values cancel! This just reduces to

$$t = E(m_n \oplus E(m_{n-1} \oplus \ldots E(m_2 \oplus E(m_1 \oplus C \oplus C, k), k) \ldots , k), k)$$

Effectively, we have canceled out the input of the prepended block on the final tag! We're back to the original MAC of the message!

$$t = E(m_n \oplus E(m_{n-1} \oplus \ldots E(m_2 \oplus E(m_1, k), k) \ldots , k), k)$$

This example was just for a single block. But it turns out that no matter how long the prepended message is, we only care about the part that will be XORed with m_1 before it is encrypted. In a CBC chain of arbitrary length, the only part that carries over into the next block is the *last encrypted block of the chain*. In other words, the MAC output of the CBC-MAC operation, t, is the only part of a prepended message that would impact what follows it!

Suppose, then, that you have two messages M_1 and M_2 and two corresponding tags t_1 and t_2, both of which were generated under the same key using our broken CBC-MAC algorithm. To create a falsified message, first XOR t_1 with the first block of M_2 to produce M_2'. Now create $M_3 = M_1 + M_2'$ (plus means concatenation). The CBC-MAC of M_3 will also be t_2 because (using $C(\cdot)$ to mean "MAC"):

$$t_2 = E(M_{2,n} \oplus E(M_{2,n-1} \oplus \ldots E(M_{2,1} \oplus t_1 \oplus C(M_1, k), k) \ldots , k), k)$$

As the MAC of M_1 is t_1, it cancels out with the other t_1 and the MAC of what is left is just the MAC of M_2.

A visualization of this attack, and the math we just worked through, is depicted in Figure 5-2.

Importantly, *you do not need the key to do this attack*. In our code example, we had the key ourselves and generated an arbitrary message. This is still an attack, because even the possessor of the shared key should not be able to send two messages with the same MAC.

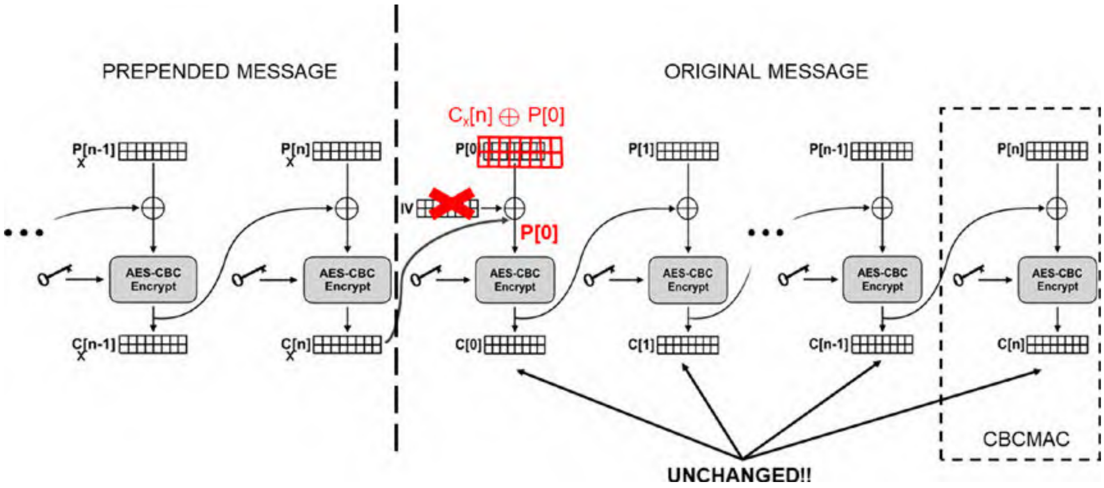

Figure 5-2. *An attacker can prepend a message without changing the (simple) CBC-MAC by corrupting just the first block*

But with this attack, an attacker *without* the key can generate a new message and a falsified tag from two existing messages (e.g., generated by the victim) and corresponding tags.

There are various solutions to this problem, but the only one we'll mention here is to enforce that each message is prepended with the length of the message, as in Listing 5-4.

Listing 5-4. Prepend Message Length

```
# Reasonably secure concept. Still, NEVER use it for production code.
# Use a crypto library instead!
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os

def CBCMAC(message, key):
    aesCipher = Cipher(algorithms.AES(key),
                       modes.CBC(bytes(16)), # 16 zero bytes
                       backend=default_backend())
    aesEncryptor = aesCipher.encryptor()
    padder = padding.PKCS7(128).padder()
```

```

        padded_message = padder.update(message)
        padded_message_with_length = len(message).to_bytes(4, "big") +
        padded_message
        ciphertext = aesEncryptor.update(padded_message_with_length)
        return ciphertext[-16:]
```

To use CBC-MAC securely, there are a few additional caveats:

1. If you are also encrypting the data with AES-CBC, you must not use the same key for both encryption and MAC.
2. The IV should be fixed to zero.

A full explanation of each of these is beyond the scope of this book. Assuming that you follow them, however, the included CBC-MAC code is reasonably secure. We still don't recommend using it because it is *always* dangerous to create your own cryptographic algorithms or even your own implementations of known cryptographic algorithms. Instead, always use algorithms in trusted cryptographic libraries.

The `Cryptography` library that we are using for our example code includes CMAC. This algorithm is an updated and improved CBC-MAC defined in RFC 4493. Either CMAC or HMAC are good choices for a MAC algorithm; HMAC might be faster on most systems without specialized AES encryption hardware.

Using CMAC from the library is straightforward. The following is taken directly from the online documentation:

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import cmac
>>> from cryptography.hazmat.primitives.ciphers import algorithms
>>> c = cmac.CMAC(algorithms.AES(key), backend=default_backend())
>>> c.update(b"message to authenticate")
```

Encrypting and MACing

In many circumstances a message needs to be encrypted and protected from modification. In the first code example in this chapter, Alice and Bob used an unkeyed hash to protect an encrypted message. Obviously, that doesn't work because without a key, anyone can generate the corresponding hash. Now that our intrepid (or dastardly) duo know how to use HMAC and CMAC, they can update their code.

EXERCISE 5.6. ENCRYPT THEN MAC

Update the code from the beginning of the chapter to do a proper MAC by replacing the SHA-256 operation with HMAC or CMAC. Use *two* keys.

Pay attention to when you use MAC and what you use it on in the previous exercise. You will notice that it is the *ciphertext* that the MAC is applied to, not the plaintext. As the name of the exercise implied, this is called *Encrypt-Then-MAC*. There are two other ways of sending an encrypted and authenticated message that have been done in the past.

One is MAC-Then-Encrypt. In this version, the MAC is applied to the *plaintext,* and then both the plaintext and the MAC are encrypted together. This approach was taken by early versions of TLS (which is used for HTTPS connections).

Another approach is called Encrypt-And-MAC. To take this approach, the MAC is again computed over the plaintext, but the MAC itself is not encrypted. It is sent (unencrypted) along with the ciphertext. If you've ever used Secure Shell (SSH or PuTTY), it uses Encrypt-And-MAC.

It is strongly recommended by most cryptographers, with a few dissenters, as there are always some of those, to use Encrypt-Then-MAC[3] over these other two approaches. In fact, certain practical vulnerabilities have been found against certain combinations of MAC-Then-Encrypt. You have already demonstrated one! The padding oracle attack against CBC in the previous chapter only works against MAC-Then-Encrypt scenarios.

There's an even better approach called AEAD (authenticated encryption with additional data) that we will learn about in Chapter 7 that combines encryption and message integrity into a single operation. If, for whatever reason, you need to combine encryption and MAC, make sure you choose Encrypt-Then-MAC (i.e., encrypt the plaintext and then compute a MAC over the ciphertext).

We won't go into the various arguments for why Encrypt-Then-MAC is generally considered better but one point is worth mentioning. As we have talked about in other circumstances, we generally don't want bad guys messing around with our ciphertext. It can be unintuitive because we tend to think about the end goal: protecting the *plaintext*. But bad things happen when the bad guys can change the ciphertext without us being

[3] Are you confused, yet? "Encrypt-*And*-MAC" means to apply them both to the plaintext, while "Encrypt-*Then*-MAC" means to apply the MAC to the ciphertext: *after* encryption.

able to detect it. When you do Encrypt-Then-MAC, the ciphertext should be protected against modification.

EXERCISE 5.7. KNOW THY WEAKNESS

Encrypt-Then-MAC is the recommended approach to combining encryption and MACs. However, it is good to understand all three approaches. If nothing else, if you ever have to maintain code you did not write, or have to be compatible with legacy systems, you may encounter this in the future. Modify your (highly recommended) Encrypt-Then-MAC system to create a MAC-Then-Encrypt variant. Finally, create a MAC-And-Encrypt version as well.

Digital Signatures: Authentication and Integrity

Alice and Bob love sending encrypted messages with HMACs (using Encrypt-Then-MAC). On their current assignment in West Antarctica, they each have four keys. One pair allows them to send encrypted and MAC-protected messages to each other (remember, one key for encryption, one key for MAC generation), and the second pair allows them to send and receive encrypted and MAC-protected messages to and from HQ back in East Antarctica.

Unfortunately, one day Alice is captured as she attempts to infiltrate the West Antarctic Snowball Testing Edifice. Instantly, everything is thrown into disarray as Eve now has access to all of her keys.

This is a terrible compromise. Eve is now able to send messages as though they are from Alice or HQ! Trying to mitigate this loss of confidentiality and authentication is a nightmare. Bob's situation is bad. He needs two new keys to communicate with HQ and perhaps two new keys for communicating with a new partner in the field. This can only be done by returning to HQ, which means pulling him out of the field, potentially wasting time and resources he has spent infiltrating his targets and gathering data. Worse, he can't even be reliably told about what is going on! If he doesn't have first-hand knowledge of Alice's capture, any messages sent by HQ informing him of the event or instructing him to come home can be intercepted and changed.

As bad as things are for Bob, HQ is in far worse shape. They were using the same shared keys for encrypting and tagging all of their messages. Every single agent in the field has the compromised keys lost by Alice. Eve can impersonate HQ to any of them.

And Eve can send messages to HQ as any of the agents, because they did not have their own individual keys for communicating with HQ.

The loss of the shared keys sets EATSA back at least 12 months.

As bad as it is that Eve can read the traffic between HQ and their agents using the encryption keys, it might be worse that she can send messages pretending to be any of these parties by using the MAC keys. To repeat one of our earlier comments, when people first start to learn about cryptography, they typically think about "encryption" as its main purpose or characteristic. As our fictional example illustrates, authentication—knowing who sent a message—is at least as important, and arguably more so.

Even once EATSA manages to get all of their agents' home and is no longer using the old keys (the old keys are thus "revoked"), they have the problem of coming up with a key management system to avoid the same problem in the future. One option they consider is for each agent to have their own individual key. If either HQ or an agent wants to send a message, they use their individual key to tag it.

The problem is MACs require *shared* keys. The receiver of the message must have the same key as the sender. How will they obtain it? Will every agent have every other agent's key? If so, an agent's capture is just as bad as if there was only one key. Worse, nothing keeps an agent from using another agent's key (impersonating them) either by accident or because they go rogue.

Eventually, one of the scientists remembers asymmetric encryption from Chapter 4, specifically that it can be used for something called a *digital signature*. Like message authentication codes, digital signatures are designed to provide authenticity (you can tell who sent the message) and message integrity (the message cannot be changed undetectably). Furthermore, because they use asymmetric encryption, there are *no shared keys*. At the time the EA started playing around with asymmetric encryption, they became very, very focused on encryption of messages (confidentiality) and digital signatures fell off to the side.

It is time to remedy that.

What exactly is a digital signature? First, let's review how asymmetric encryption works for the RSA algorithm we studied in Chapter 4. Unlike symmetric encryption where there is a single shared key between parties, RSA's asymmetric encryption involves a *pair* of keys: the public key and the private key. These keys work as opposites of one other: what one encrypts, the other decrypts. Moreover, the RSA public key can be derived from the private key but not the other way around.

As the name implies, a party should keep the RSA private key private and disclose it to nobody, ever. On the other hand, the RSA public key can and typically should be widely disseminated. This setup enables two very interesting operations.

First, because the RSA public key is held by anyone (and potentially by everyone!), it is easy for anyone in the world to send an encrypted message to the owner of the corresponding RSA private key. Anyone can use the public key to encrypt the message, but *only* the party with the private key can decrypt it.

This is important! The person that sends the encrypted message knows that *only* the party possessing the private key can decrypt the message. This is a different kind of reverse authenticity. The recipient of the message has no idea who sent it, but the sender can be certain (if the keys are secure) that only the intended party can *read* the message. Our introduction to RSA asymmetric encryption in Chapter 4 focused on this use case.

But, the direction of the encryption can be *reversed*: RSA private keys can also be used to *encrypt* messages. The party that has the private key can thus use it to encrypt something that can only be decrypted by the public key. What good would that do? Anybody (everybody!) could have the public key. This encryption certainly won't keep data confidential!

This is true! But, a message sent encrypted under the RSA private key can only have been encrypted by someone *who has that private key*. Even if everyone can *decrypt* it, the fact that it can be decrypted by a particular public key is a *proof that the sender holds the private key*. In other words, if you get a message that you can decrypt using my public key, *you know that it came from me*; nobody else could have encrypted it. That sounds useful!

Let's suppose that the EA wants to publish a manifesto of West Antarctica's crimes to the whole world. First they could disseminate their RSA public key everywhere and then encrypt the document under the associated private key. Now, when they distribute the document, anyone in the world can decrypt it, and that fact proves to them that it came from the EA.

This system is great, but it has a couple of important flaws. First of all, how does the world know that the RSA public key really belongs to the EA (and is not a fake from the WA, for example)? This is a critically important question and we'll get to it a bit later. For now, we will assume that recipients have a legitimate, trusted RSA public key for the intended party.

Another problem is efficiency. RSA encryption is *slow*. Decrypting long documents to verify the sender is not a remotely efficient way of doing things. Worse, some asymmetric algorithms do not have any built-in message integrity. Oh, and while we're talking about RSA's limitations, it can't encrypt something as long as a document anyway.

These latter two problems of efficiency and integrity are fortunately easily addressed. Recall that we are not encrypting for *confidentiality*, but for *proof of origin or authenticity*. Instead of encrypting the message itself, how about encrypting a *hash* of the message?

That is the basic idea of an RSA digital signature over arbitrary data. It consists of two steps. First, hash the data. Second, encrypt the hash with the private key. The encrypted hash is the sender's signature applied to the data. The signature can now be transmitted along with the original (potentially unencrypted) data. When the recipient receives data and a signature, the recipient generates the hash, decrypts the signature with the public key, and verifies that the two hashes (generated and decrypted) are identical.

Here is how cryptographers might represent this. First, for a message M, we generate a hash using a hash function: $h = H(M)$.

Once we have the hash h, we encrypt it under the RSA private key. To depict this operation, we are going to use some notation that is often used in cryptographic protocols. Specifically, we will use $\{\cdot\}$ to indicate RSA-encrypted data. Everything within the braces is plaintext, but the braces indicate that the plaintext is within some cryptographic envelope. The braces will also have a subscript indicating the key. So, for example, the ciphertext C is the plaintext P encrypted under some key K, and this is depicted as $C = \{P\}_K$.

From this point forward in the book, a *shared* key between two parties will be depicted with a subscript indicating both parties. So, for example, a key between Alice and Bob can be depicted as $K_{A,B}$. This would be an example of a symmetric key.

Public keys, such as RSA public keys, will be denoted by a key with just one identifying party. For example, Alice's public key could be denoted K_A and Bob's would similarly be K_B. Because the public key is what is distributed, it is what is named. The private key is denoted instead as the inverse of a public key: $K^{-1}{}_A$ and $K^{-1}{}_B$).

In this chapter, we will also typically use the letter t to represent RSA signatures because a signature is also sometimes called a tag, just like a MAC is. Thus, we represent an R:

$$t_M = \{H(M)\}_{\mathrm{K}^{-1}}$$

When another party with possession of the RSA public key K receives M, $\{H'(M)\}_{\mathrm{K}^{-1}}$, the signature is decrypted by the public key to recover $H'(M)$. The receiving party generates their own $H(M)$, and the signature is considered authentic if $H'(M) = H(M)$.

At the risk of being repetitive, remember that RSA public key encryption is used for different things than private key encryption. Encryption with the RSA public key keeps the message *confidential*: only the private key owner can *read* it. Encrypting with the RSA private key proves *authenticity*: only the owner could have *authored* it.

In the EA spy agency, this seems miraculous! The agency generates an RSA key pair for itself and also has all of the agents generate an RSA key pair. The agency keeps a copy of all the public keys of all the agents, and every agent takes a copy of the agency's public key.

When the agency sends an encrypted message to Alice, they encrypt it under her public key and only Alice will be able to decrypt it. They *also* sign the message with their private key, and Alice can use the agency public key to verify that the message is authentic and uncorrupted. So long as Alice and Bob have a copy of each other's public keys, they can likewise send encrypted and authenticated messages to each other.

This is a big step forward, and it seems pretty great.

It really is, but as has so often been true with the EA's cryptographic experiences, there are complications, caveats, and subtleties. Before we get into that, however, let's help Alice and Bob learn how to send each other some signed communications. For simplicity, we are not going to encrypt them.

Again, the cryptography library comes to our rescue with its signing and verification functions: we do not need, nor should we attempt, to implement digital signatures ourselves. Rather, using our library, we will generate some RSA signatures.

Listing 5-5. Sign Unencrypted Data

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding

private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=default_backend()
)
public_key = private_key.public_key()

message = b"Alice, this is Bob. Meet me at Dawn"
signature = private_key. sign(
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)

public_key.verify(
    signature,
    message,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
print("Verify passed! (On failure, throw exception)")
```

There's probably a bit more in Listing 5-5 than expected, particularly in the padding configuration. Let's walk through it all.

First, we generate a key pair. For RSA, the public key is derivable from the private key, so generating the private key generates the key pair. The API includes a call to obtain the public key from the private key. In this example, both keys are used. In a real example, the signing and verification code would live in completely different programs, and the verification program would only have access to the public key, not the private key.

In Chapter 4 we also learned how to serialize and de-serialize these kinds of RSA keys from disk.

In the next part of the code, we sign the message. You will notice that we are using padding here just as we did for RSA encryption, but it is a different scheme. The recommended paddings for RSA are OAEP for encryption and PSS for signatures. Perhaps that surprises you given that RSA signatures are generated by encrypting a hash. If it's all encryption anyway, why do we need different padding schemes?

The answer is that, because signatures are operating on a hash, there are certain characteristics that must be true about the data. The nature of arbitrary data encryption vs. hash encryption drives the two different padding schemes.

Like the OAEP padding used in Chapter 4, PSS padding function also requires the use of a "mask generation function." At the time of this writing, there is only one such function, MGF1.

Finally, the signature algorithm requires a hashing function. In this example, we are using SHA-256.

The parameters to the verification algorithm should be self-explanatory. Note that the validation function does not return a true or false, rather it raises an exception if the data does not match the signature.

Important Please pay careful attention to this next paragraph. It is very important and somewhat counter-intuitive.

If you wanted to encrypt and sign, should you sign first and then encrypt, or should you encrypt first and then sign? After the discussion in the previous section on Encrypt-Then-MAC, you might be thinking Encrypt-Then-Sign.

But signatures are *not MACs*, and you should generally *not* use Encrypt-Then-Sign. There are two very important reasons.

First, remember that the goal of the signature is not just message *integrity* but also sender *authentication*. Suppose that Alice is sending an encrypted message to Bob, and she encrypts the message *before* signing it. Anyone can intercept the message, strip off the signature, and send the message re-signed under their own key. Oops.

It isn't clear how practical this attack is because the data was encrypted under the *receiver's* public key that everyone already has. The attacker could just send their own encrypted message to Bob (encrypted by Bob's public key) anyway. The attacker can't even decrypt Alice's message to see if he/she wants to take credit for it. But the point is, there is no association between the plaintext and the signature, and there really needs to be: Bob is interested in knowing that the *message he can read* comes from Alice and not someone else. If the encrypted data is signed instead of the plaintext, when Bob receives the ciphertext and the signature, he cannot reliably determine who authored the original message.

In short, if you sign an encrypted message, it is too easy for it to be intercepted and signed by someone else instead, which compromises its authenticity. The signature should be applied to the plaintext.

Second, and far more important, signatures cannot prevent the bad guys from altering the ciphertext. Remember, the number one reason for using Encrypt-Then-MAC was to prevent undetectable alteration of the encrypted data. With Encrypt-Then-Sign, Eve, for example, could intercept a message from Alice to Bob, strip off Alice's signature, *alter the ciphertext*, and then sign the altered data with her own key. What good is this, you might ask? After all, Bob will see that the message is now signed by Eve and not Alice. Why would he trust it?

There are any number of reasons Bob will accept the signature. For example, Eve may have compromised another agent's key. The whole reason for using RSA encryption was to prevent the compromise of one agent's key from compromising the communications of another. But if Eve gets a legitimate signing key, she can strip off Alice's signature, modify the ciphertext, and re-sign with something Bob will accept.

Once this happens, Eve can observe Bob's behavior to learn things about Alice's message. As we used in earlier examples, even Bob throwing away a message is information that Eve can use to her advantage (e.g., she knows that the message she sent to him was unreadable).

Does this sound far-fetched? Well, exactly this kind of vulnerability in Apple's iMessage was discovered by Matt Green. You can read about it on his blog [6]. We won't discuss his attack in detail here other than to say that this kind of attack is actually very practical.

So please, do *not* Encrypt-Then-Sign.

Why is this so different from MACs? Why does Encrypt-Then-MAC work? The fundamental difference comes back to the keys. With a MAC, there is a shared key, typically shared between just the two parties. Nobody should be able to replace a MAC created by a key shared between Alice and Bob because nobody else should have the key. The private key used to create a digital signature, however, is not shared and does not bind any parties together.

What *should* you do? In the first place, there don't seem to be many crypto systems this applies to. If you are using symmetric encryption, it is usually no problem to include a symmetric MAC. If Apple had done this, the iMessage attack we mentioned wouldn't have been possible. Asymmetric encryption is not generally used for bulk encryption. When encrypting a lot of data is necessary, the usual approach is to exchange or create a symmetric key using the asymmetric cryptography and then switch to symmetric algorithms. We will talk about this in the next chapter.

If you absolutely must sign and encrypt without the benefit of a symmetric MAC (e.g., RSA encryption plus some signatures), the plaintext message should be signed and both the plaintext and signature should be encrypted (Sign-Then-Encrypt). Although this means that an attacker can try to mess around with the ciphertext, a good RSA padding scheme like OAEP should make this very difficult.

While there are no known attacks against Sign-Then-Encrypt, some of the most paranoid still Sign-Then-Encrypt-Then-Sign-Again. The inner signature is over the plaintext, proving authorship, and the outer signature is over the ciphertext, ensuring the integrity of the message. One other alternative is something called “signcryption.” Because signcryption isn’t supported by the Python `cryptography` library, we won’t spend any time on it here, but the curious can read this paper about it: `www.cs.bham.ac.uk/~mdr/teaching/modules04/security/students/SS3/IntroductiontoSigncryption.pdf`.

For now we will stick with the slightly less paranoid Sign-Then-Encrypt strategy. Remember, however, that RSA encryption can only encrypt a very limited number of bytes. When OAEP padding is used with SHA-256, the maximum plaintext that can be encrypted is only 190 bytes! If you start encrypting signatures, there may be very little room left for anything else. If your message is too long, you will have to break it up and encrypt it in 190-byte chunks. This is all the more reason to use the combined asymmetric and symmetric operations we will see in the next chapter.

EXERCISE 5.8. RSA RETURNS!

Create an encryption and authentication system for Alice, Bob, and EATSA. This system needs to be able to generate key pairs and save them to disk under different operator names. To send a message, it needs to load a private key of the operator and a public key of the recipient. The message to be sent is then signed by the operator’s private key. Then the concatenation of the sender’s name, the message, and the signature is encrypted.

To receive a message, the system loads the private key of the operator and decrypts the data extracting the sender’s name, the message, and the signature. The sender’s public key is loaded to verify the signature over the message.

EXERCISE 5.9. MD5 RETURNS!

In Chapter 2, we discussed some of the ways that MD5 is broken. In particular, we emphasized that MD5 is still not broken (in practice) for finding the preimage (i.e., working backward). But it *is* broken in terms of finding collisions. This is very important where signatures are concerned because signatures are typically computed over the hash of data and not the data itself.

For this exercise, modify your signature program to use MD5 instead of SHA-256. Find two pieces of data with the same MD5 sum. You can find some examples at or with a quick search of the Internet. Once you have the data, verify that the hashes are the same for the two files. Now, create a signature for both files and verify that they are the same.

One last thing should be mentioned. In some cases, you may not have all of the data to be signed all at once. The `sign` function does not have an `update` method like hashing functions do. It does have an API to submit pre-hashed data, however. This allows you to hash the data that needs to be signed separately. Here is an example drawn from the `cryptography` module documentation:

```
>>> from cryptography.hazmat.primitives.asymmetric import utils
>>> chosen_hash = hashes.SHA256()
>>> hasher = hashes.Hash(chosen_hash, default_backend())
>>> hasher.update(b"data & ")
>>> hasher.update(b"more data")
>>> digest = hasher.finalize()
>>> sig = private_key.sign(
...     digest,
...     padding.PSS(
...         mgf=padding.MGF1(hashes.SHA256()),
...         salt_length=padding.PSS.MAX_LENGTH
...     ),
...     utils.Prehashed(chosen_hash)
... )
```

Elliptic Curves: An Alternative to RSA

It's time we told you the truth about asymmetric cryptography. Everything we've told you so far has been RSA-specific and quite a bit of what RSA does is actually unique.

When we talk about asymmetric, or public key, cryptography, we are referring to any cryptographic operations that involve a public and private key pair. In Chapter 4 we looked almost exclusively at RSA encryption, and in this chapter, we explored RSA signatures. Conveniently, RSA signatures are also based on RSA encryption (i.e., encrypting a hash of the data to be signed). But most other asymmetric algorithms do not even support encryption as a mode of operation at all and do not use encryption for generating a signature. Other asymmetric algorithms, for example, generate a signature or tag that does not involve any encryption and verify the signature without any kind of reversible operation such as decryption.

This is one reason why we have tried to qualify our conversations about asymmetric cryptography through the book by referring specifically to "RSA public keys," "RSA encryption," and "RSA asymmetric operations." You should not assume that other asymmetric algorithms provide the same operations or do them in the same way.

Why focus so much on RSA encryption? We do this here because RSA has been one of the most popular algorithms for asymmetric operations for decades. It is still found absolutely everywhere, and you will be hard-pressed not to run into it somewhere. DSA (digital signing algorithm) is another asymmetric algorithm, but it is only usable for signatures, not for encryption. For educational and practical purposes, then, RSA is a great place to start.

With that said, RSA is slowly getting phased out. It has been found to have a lot of weaknesses, some of which we have explored already. Cryptography based on "elliptic curves"[4] has been used both to sign data and to exchange keys. In this chapter we will look at ECDSA's signing capabilities. In Chapter 6 we will look at something called Elliptic-Curve Diffie-Hellman (ECDH) that is used to create and agree on session keys. ECDH's key agreement provides an alternative (arguably a better alternative) to the key transport functionality enabled by RSA encryption.

To sign data with elliptic curves, you make use of the ECDSA algorithm. Just as you must choose parameters for RSA (such as *e*, the public exponent), you must also choose parameters in EC-based operations. The most obvious of these is the underlying curve.

[4] The math upon which elliptic-curve cryptography is based is beyond the scope of the book. The goal of this section is simply to make you aware of the algorithms and show you how to use them.

Again, the actual mathematics are not discussed in this book, so we will satisfy ourselves by saying that different elliptic curves can be used in these algorithms.

For ECDSA, the cryptography library provides a number of NIST-approved curves. It should be noted that some cryptographers are wary of these curves because it is possible that the US government recommends curves that it knows can be broken. With that said, these are the only curves currently provided by the library. If you use these in production, you should keep an eye out for additional information about security vulnerabilities and potential replacements.

For this test, we will use NIST's P-384 curve, which is referred to as SECP384r1 in the library. From the cryptography documentation

```
>>> from cryptography.hazmat.backends import default_backend
>>> from cryptography.hazmat.primitives import hashes
>>> from cryptography.hazmat.primitives.asymmetric import ec
>>> private_key = ec.generate_private_key(
...     ec.SECP384R1(), default_backend()
... )
>>> data = b"this is some data I'd like to sign"
>>> signature = private_key.sign(
...     data,
...     ec.ECDSA(hashes.SHA256())
... )
>>> public_key = private_key.public_key()
>>> public_key.verify(signature, data, ec.ECDSA(hashes.SHA256()))
```

As with RSA signing, you do have to pick a hash function. Again, we have chosen SHA-256. You will notice that, although it might seem daunting to pick a curve function, once that's done, the rest of the operation is very straightforward.

ECDSA also has the same pre-hashed API as RSA for processing large amounts of data.

Certificates: Proving Ownership of Public Keys

In our example with Alice and Bob and public keys, we assumed that every interested party had the public key of every other interested party. In our scenario, this *might* be possible. The HQ could gather all the spies together and have everyone exchange public keys.[5]

This might not be feasible over time, however.

What if Noel, a new spy, enters the field after everyone else? Assume agent Charlie has been captured and Noel has been sent to take his place. Alice and Bob already had Charlie's key, but they don't yet have Noel's key.

Of course, Noel can't just show up and hand out a public key. Otherwise, Eve could send in fake agents handing out public keys claiming to be real EA agents. She can create certificates just as easily as HQ. How can Alice and Bob recognize that Noel is a true EATSA agent and is not working for Eve?

One possibility is to have HQ send Alice and Bob a message with the name and public key of the new agent. Alice and Bob already trust HQ and already have HQ's public key. HQ can act as a *trusted third party* between them and Noel. In the early days of PKI, this was exactly what was proposed to establish trust. This model was called a "registry." A registry would be a central repository of identity-to-public-key mappings. The registry's own public key would be disseminated everywhere: newspapers, magazines, textbooks, physical mailings, and so forth. So long as everyone had a true copy of the registry's key, they could look up the public key of anyone registered within it.

The problem at the time, although it is less of a problem now, was scale. Although contemporary computing envisions the Googles, Amazons, and Microsofts of the world handling billions of connections from all over the world all the time, such was not the case in the 1990s. It was believed that an online registry was simply not scalable.

In the case of our spies, they have to assume they may be cut off from HQ. They may have to go into deep cover, or they may be on the run from Eve, or maybe the EA wants to disavow any of their activities for a time. For any or all of these reasons, they may not be able to get a timely message from HQ. If they're on the run from Eve, it would be great if they could tell whether the spy who meets them at the safe house is on their side.

This brings us to certificates. A public key certificate is just data; it generally includes a public key, the metadata related to ownership of the key, and a signature over all of the

[5] An analog of this happens in the real world: PGP signing parties. You might want to look for more information about this using your favorite web search engine.

contents by a known "issuer." The metadata includes information such as the identity of the owner, the identity of the issuer, an expiration date, a serial number, and so forth. The concept is to bind the metadata, especially for identity, to the public key. The identity can be a name, an email address, a URL, or any other agreed-upon identifier.

Instead of simply handing out public keys to their agents, HQ can now hand out *certificates.*[6] First, the agent generates their own key pair (nobody, not even HQ, should ever have the agent's private key). Next, HQ takes the agent's public key and starts creating a certificate by including the identifying information about the agent, such as their code name.[7] To complete the certificate, HQ *signs* it with the HQ private key and becomes the issuer.

To repeat, the public key in the certificate belongs to the agent. The agent keeps their own private key private.[8] As illustrated in Figure 5-3, the signature *on* the certificate was generated by the issuer's private key (in this case, HQ's private key).

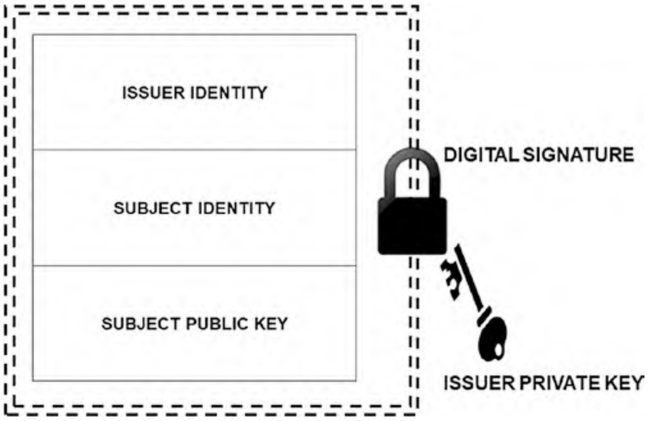

Figure 5-3. *The primary purpose of a certificate is to bind an identity and a public key together. An issuer can sign the certificate data preventing modification and providing trust.*

[6] Remember, these *contain* public keys, but are also signed, etc.

[7] Although, remember that certificates are *public*! Don't put information in a certificate that you don't want other people to see. Maybe that's not the right place for a code name?

[8] Apparently, some web servers ask for a "certificate" to be installed but require both a certificate and a private key. This is an unfortunate misuse of words that have clear meanings. Certificates are public and only contain public keys. Private keys are private and are not part of a certificate.

Let's go back to our scenario where Alice is on the run in West Antarctica with Eve's agents hot on her trail. She gets to a safe house and sees an agent she's never met before: Charlie. To prove that he is who he says he is, Charlie presents his certificate. Alice checks that the identity data matches his claim (e.g., that the identity in the certificate is "charlie"). Next, Alice checks that the issuer of the certificate is HQ and then verifies the signature included in the certificate. Remember, the signature in the certificate is signed by the *issuer* (HQ). Using HQ's public key issued to her before she left on the mission, Alice's signature check is successful. Thus, Alice knows that the certificate must have been issued by HQ because nobody else could have generated a valid signature. The certificate is authentic, and Alice now has (and trusts) Charlie's public key for future communications.

Of course, there is one more wrinkle. Charlie's certificate is *public*! There's nothing to stop Eve from having a copy and present it to Alice herself. How does Alice know that the person at the door that claims to be Charlie, with certificate in hand, really is Charlie?

Charlie must now prove his identity by signing some data for Alice. Alice gives him some kind of test message, and Charlie signs it with his private key. Alice verifies the signature on this data using the public key from his certificate. The signature check passes, so Alice knows that Charlie must be the owner of the certificate. Only the owner has (or should have!) the private key associated with the public key necessary to sign data. Of course, if Charlie were captured and his private key compromised, all bets would be off!

In summary, Charlie signs with his private key to prove it is his certificate, but Alice checks the signature in the certificate to ensure that the certificate itself was issued by someone she trusts. Alice's point of view for this process is shown in Figure 5-4.

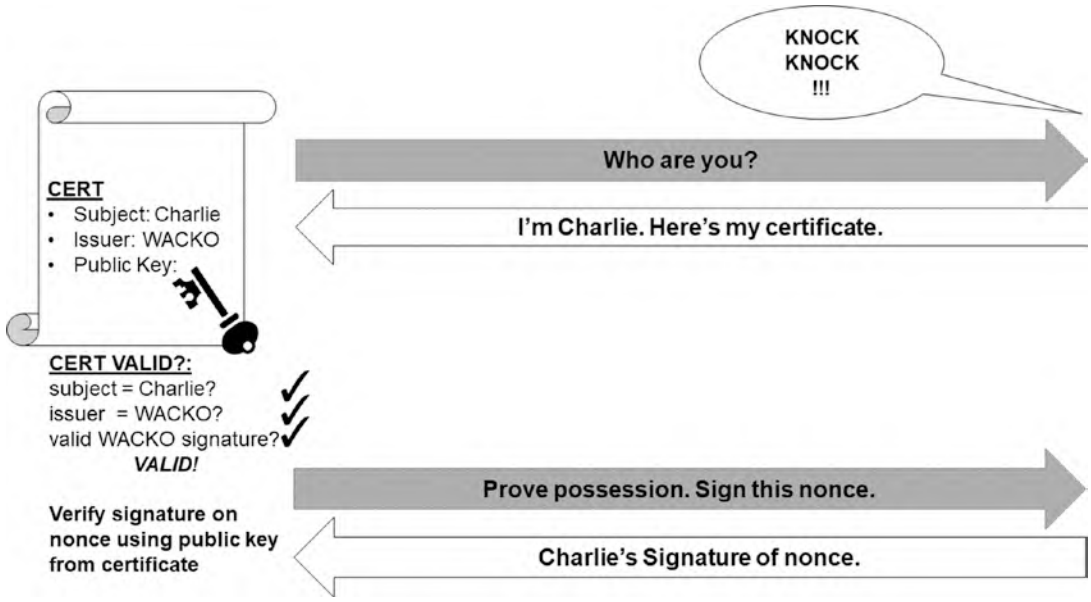
KNOCK KNOCK !!!
Who are you?
CERT
• Subject: Charlie
• Issuer: WACKO
• Public Key:
I'm Charlie. Here's my certificate.
CERT VALID?:
subject = Charlie? ✓
issuer = WACKO? ✓
valid WACKO signature? ✓
VALID!
Prove possession. Sign this nonce.
Verify signature on nonce using public key from certificate
Charlie's Signature of nonce.

Figure 5-4. *Who's knocking at the door? Alice would like to know before she lets in whoever it is!*

Let's go through some examples to see how this works. For the first exercise, we are not going to use real certificates, at least not yet. For now, we're going to use a simple dictionary as our certificate data structure, and we're going to use the Python json module to convert it to bytes.

Warning: Not for Production Use

My, we do say that "not for production use" thing a lot, don't we? We kind of have to. Cryptography is uniquely and simultaneously subtle and alluring: the concepts are relatively simple to describe, but tiny details can make the difference between good security and no security. Those details are sometimes hard to discover, and proving that they are correct is *hard*.

Don't use *any* of the non-library implementations from this book in production and do not assume that even our use of libraries is an appropriate solution. Don't assume that an example has taught you enough to roll your own crypto, and don't assume that you have mastered the correct use of the libraries. Don't even assume that our list of things that go wrong is complete!

Remember, YANAC (You Are Not A Cryptographer... yet!). We'll be saying this again. It's what we do.

The example we're going to work has three parties: the party claiming an identity (Charlie), also known as the *subject*, the party verifying the claim (Alice), and the trusted third party that issued the certificate (HQ). Two of these parties, Charlie and HQ, will need RSA key pairs. You can generate RSA key pairs and save them to disk using the `rsa_simple.py` script from Chapter 4. For the rest of this exercise, we will assume that HQ's keys are saved in `hq_public.key` and `hq_private.key` and Charlie's keys are saved in `charlie_public.key` and `charlie_private.key`.

Also, for clarity, we have created three separate scripts for each one of these parties. The first script is used *by the issuer* (HQ) to generate a certificate from an existing public key.

Listing 5-6. Fake Certificate Issuer

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization

import sys, json

ISSUER_NAME = "fake_cert_authority1"

SUBJECT_KEY = "subject"
ISSUER_KEY = "issuer"
PUBLICKEY_KEY = "public_key"

def create_fake_certificate(pem_public_key, subject, issuer_private_key):
    certificate_data = {}
    certificate_data[SUBJECT_KEY] = subject
    certificate_data[ISSUER_KEY] = ISSUER_NAME
    certificate_data[PUBLICKEY_KEY] = pem_public_key.decode('utf-8')
    raw_bytes = json.dumps(certificate_data).encode('utf-8')
    signature = issuer_private_key.sign(
        raw_bytes,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return raw_bytes + signature

if __name__=="__main__":
    issuer_private_key_file = sys.argv[1]
    certificate_subject = sys.argv[2]
    certificate_subject_public_key_file = sys.argv[3]
    certificate_output_file = sys.argv[4]

    with open(issuer_private_key_file, "rb") as private_key_file_object:
        issuer_private_key = serialization.load_pem_private_key(
                         private_key_file_object.read(),
                         backend=default_backend(),
                         password=None)

    with open(certificate_subject_public_key_file, "rb") as public_key_file_object:
        certificate_subject_public_key_bytes = public_key_file_object.read()

    certificate_bytes = create_fake_certificate(certificate_subject_public_key_bytes,
                                                certificate_subject,
                                                issuer_private_key)

    with open(certificate_output_file, "wb") as certificate_file_object:
        certificate_file_object.write(certificate_bytes)
```

Let's walk through Listing 5-6. There is only one function: `create_fake_certificate`. We are using the name "fake" not to indicate fraud, but rather that this is not a real certificate. Again, please don't ever use this in production.[9]

The function creates a dictionary and loads three fields: a subject name (identity), an issuer name, and a public key. Note that there are (parts of) two key pairs being used in this file. There is an issuer private key and the subject public key. It is the subject's private key that is being stored in the certificate. This public key in many ways *represents* the subject as it will be used to prove his or her identity. That is why it is so important that the certificate be signed.[10] Otherwise, anyone could create a certificate to claim any identity they like.

Once the dictionary is loaded, we use `json` to serialize the dictionary to a string. JSON is a common and standard format, but in Python 3.x, it cannot encode bytes directly and it outputs a text string. For compatibility with the Python `cryptography` library, we load the PEM-encoded keys as binary bytes rather than as text. The public key to be stored in this JSON certificate has to be converted to a string first, but because it is PEM-encoded (i.e., it is already plaintext), we can convert it to UTF-8 safely. Similarly, the entire output of the `json.dumps()` operation is converted to bytes with a safe UTF-8 conversion.

The bytes are then signed using the *issuer's* private key. Only the issuer should have access to this private key because it is the issuer's way of proving to the world that it (the issuer) has created the certificate. Our final certificate is the raw bytes from json concatenated with the bytes from the signature.

In our hypothetical example, Charlie wants to claim the identity "charlie." Charlie starts out by generating a key pair. The public key (*not the private key*) is sent to the HQ certificate-issuing department and a request to make a certificate. The human beings within the issuing department should verify that Charlie has the right to claim the identity "charlie." For example, the officer in charge might ask to see Charlie's agency ID, review paperwork from a superior officer, check fingerprints, and so forth to ensure that the real Charlie will be given the certificate.

[9] Told you.

[10] By a trusted authority.

The issuer script takes four parameters: the issuer private key file, the claimed identity that will be put into the certificate, the public key associated with the identity, and an output filename for the certificate. Using the keys you generated for this exercise, run the script as shown in the following:

```
python fake_certs_issuer.py \
  hq_private.key \
  charlie \
  charlie_public.key \
  charlie.cert
```

This will generate a (fake) certificate for Charlie with the claimed identity and associated public key, all signed by HQ.

Now Charlie can prove that he has the identity "charlie" to Alice. He starts by giving her the claimed identity ("charlie") and providing the certificate.

The second script here is for Alice to verify Charlie's claimed identity.

Listing 5-7. Verify Identity in a Fake Certificate

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization

import sys, json, os

ISSUER_NAME = "fake_cert_authority1"

SUBJECT_KEY = "subject"
ISSUER_KEY = "issuer"
PUBLICKEY_KEY = "public_key"

def validate_certificate(certificate_bytes, issuer_public_key):
    raw_cert_bytes, signature = certificate_bytes[:-256], certificate_
    bytes [-256:]

    issuer_public_key.verify(
        signature,
        raw_cert_bytes,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256())
    cert_data = json.loads(raw_cert_bytes.decode('utf-8'))
    cert_data[PUBLICKEY_KEY] = cert_data[PUBLICKEY_KEY].encode('utf-8')
    return cert_data

def verify_identity(identity, certificate_data, challenge, response):
    if certificate_data[ISSUER_KEY] != ISSUER_NAME:
        raise Exception("Invalid (untrusted) Issuer!")

    if certificate_data[SUBJECT_KEY] != identity:
        raise Exception("Claimed identity does not match")

    certificate_public_key = serialization.load_pem_public_key(
        certificate_data[PUBLICKEY_KEY],
        backend=default_backend())

    certificate_public_key.verify(
        response,
        challenge,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256())

if __name__ == "__main__":
    claimed_identity = sys.argv[1]
    cert_file = sys.argv[2]
```

```
issuer_public_key_file = sys.argv[3]

with open(issuer_public_key_file, "rb") as public_key_file_object:
    issuer_public_key = serialization.load_pem_public_key(
                     public_key_file_object.read(),
                        backend=default_backend())

with open(cert_file, "rb") as cert_file_object:
    certificate_bytes = cert_file_object.read()

cert_data = validate_certificate(certificate_bytes, issuer_public_key)

print("Certificate has a valid signature from {}".format(ISSUER_NAME))

challenge_file = input("Enter a name for a challenge file: ")
print("Generating challenge to file {}".format(challenge_file))

challenge_bytes = os.urandom(32)
with open(challenge_file, "wb+") as challenge_file_object:
    challenge_file_object.write(challenge_bytes)

response_file = input("Enter the name of the response file: ")

with open (response_file, "rb") as response_object:
    response_bytes = response_object.read()

verify_identity(
    claimed_identity,
    cert_data,
    challenge_bytes,
    response_bytes)
print("Identity validated")
```

Listing 5-7 requires three arguments: the claimed identity of the party, the certificate presented, and the issuer's public key.

The verification of the claimed identity has to run in two parts. First, it loads the certificate to see if it is signed by HQ's public key. This is performed by the `verify_certificate` function. Remember that the signature verification function raises an exception if the signature check fails. You will notice that to get the signature, the script just takes the last 256 bytes of the certificate. Because the signature is concatenated at the end, and because we always use an RSA signature from a 2048-bit key, the signature is always 256 bytes.

If the signature verifies, we take the other bytes and load them into a dictionary using the `json` module (again converting bytes to string for the JSON operation and then from string to bytes for the public key data).

Alice runs the script:

```
python fake_certs_verify_identity.py \
  charlie \
  charlie.cert \
  hq_public.key
```

At this point Alice's script has given her some information, but it is waiting for more input. What does Alice know right now, at this phase of the process? She knows that she has been presented with a true certificate that was actually signed by HQ. What happens next? She doesn't yet know if the party presenting the certificate is really Charlie. To do that, she needs to test him or her to see if they have the private key.

She generates a random message and saves it to the file `charlie.challenge`, which she will ask the person claiming to be Charlie to sign with his private key. The script is waiting for that random message, so Alice provides the name of the file she just created, `charlie.challenge`.

Although Alice isn't finished, we now need to switch over to Charlie's operations. Leave Alice's script running until we get back. Charlie will use another script, and his private key, to answer Alice's challenge.

Listing 5-8. Prove Identity on a Fake Certificate

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives import serialization
```

```

import sys

def prove_identity(private_key, challenge):
    signature = private_key.sign(
        challenge,
        padding.PSS(
            mgf = padding.MGF1(hashes.SHA256()),
            salt_length = padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    return signature

if __name__ == "__main__":
    private_key_file = sys.argv[1]
    challenge_file = sys.argv[2]
    response_file = sys.argv[3]

    with open(private_key_file, "rb") as private_key_file_object:
        private_key = serialization.load_pem_private_key(
                         private_key_file_object.read(),
                         backend=default_backend(),
                         password=None)

    with open(challenge_file, "rb") as challenge_file_object:
        challenge_bytes = challenge_file_object.read()

    signed_challenge_bytes = prove_identity(
        private_key,
        challenge_bytes)

    with open(response_file, "wb") as response_object:
        response_object.write(signed_challenge_bytes)
```

Charlie's script in Listing 5-8 is straightforward. It takes in three arguments: the certificate subject's private key, the challenge filename, and the response filename that will be used to store the response. The response is generated simply by taking the challenge bytes and signing them with the private key. Run this script (in a separate terminal from Alice's) as shown:

```
python fake_certs_prove_identity.py \
  charlie_private.key \
  charlie.challenge \
  charlie.response
```

Charlie has thus answered Alice's challenge and put the response into the file `charlie.response`. Now we can finally finish Alice's script, which is waiting for the response filename. Enter the filename generated by Charlie (`charlie.response`) to proceed.

Alice's script loads the response and verifies it. To do this, Alice's script now moves to the `verify_identity` function. It starts by checking that the name in the certificate matches the identity claimed (e.g., "charlie") and that the issuer is HQ. Next, it loads the public key from the certificate and verifies that the signature on the challenge bytes is valid.

This proves to Alice that not only is the certificate Charlie presented valid, but Charlie is the subject (owner). The person claiming to be Charlie must have the associated private key or he would not have been able to answer her challenge.

EXERCISE 5.10. DETECT FAKE CHARLIES

Experiment with the preceding scripts to check out the various errors from trying to deceive Alice. Create a false issuer and sign the certificate with this private key. Have someone with the wrong private key present Charlie's certificate. Make sure to understand all the different checks being performed in the code.

Although our certificates are "fake," they are designed to teach the basic principles behind the certificate concept. Real certificates typically use a format called X.509. We will discuss X.509 in detail in Chapter 8.

Certificates and Trust

One question you might have asked yourself is, why did we name the issuer? After all, if Alice, Bob, and all the other agents are always going to trust HQ, why require the issuer to be named in the certificate?

In our hypothetical world in which Antarctica is locked in its civil cold war, there may be many issuers of certificates. For example, other agencies besides the espionage unit may want to issue certificates. What if the EA military starts to issue certificates? What if the EA Department of Education starts to issue certificates? Should Alice and Bob trust those as well? Maybe they will want to trust military certificates but not education certificates?

In certificate parlance, we also call an issuer a "certificate authority" (CA), and certificate validators have to decide which certificate authorities they will trust. In fact, CAs also have their own certificates with their identity name and their public key. Thus, the *Issuer* field of a certificate should be the same identity as the *Subject* in the CA's certificate.

If the CA has a certificate, who signs *that*? There is a concept called an "intermediate" CA. An intermediate CA has its certificate signed by a "higher" CA. In the EA government, perhaps, there might be a top-level CA that signs all other CAs for defense, education, espionage, and so forth. This creates a hierarchical chain of certificates with the highest certificate called a "root" certificate.

Who signs this ultimate root CA?

The answer is: itself. This CA's certificate is known as a *self-signed* certificate. Note that anyone can generate a self-signed certificate, so great care must be taken in deciding which self-signed, root certificates to trust. Basically, they become axiomatically trusted *along with all of the certificates that they sign*!

While this can be a little complicated to visualize, it does make things a little bit easier to manage. The entire EA government could have a single top-level CA. All employees, agents, or even citizens need only have the very top-most, root CA certificate. All other identities can be verified in a chain. For example, Charlie might keep three certificates: his personal certificate, the intermediate certificate for the espionage CA that signed his certificate, and the root EA certificate itself. Charlie can present these three certs to any other EA employee and have him or her verify the chain back to the root.

Things become slightly more complicated (and introduce potential security risks) when there are multiple roots. For example, perhaps the EA government doesn't have a single, top-level root. After all, do you really want your espionage orders to be signed by a CA that can be traced back to the government? Suppose then that the EA government keeps two roots: one for departments and organizations that operate "visibly" and one for groups and individuals that operate covertly.

Should Charlie and the other agents trust both roots?

EXERCISE 5.11. THE CHAINS WE FORGED IN LIFE

Modify the identity validation programs to support a chain of trust. First, create some self-signed certificates for the EA government (at least two as described previously). The existing issuer script can already do this. Just make the issuer private key for the self-signed certificate to be the organization's own private key. Thus, the organization is signing its own cert, and the private key used to sign the certificate matches the public key *in* the certificate.

Next, create certificates for intermediate CAs such as "Department of Education," "Department of Defense," "Espionage Agency," and so forth. These certificates should be signed by the self-signed certificates in the previous step.

Finally, sign certificates for Alice, Bob, and Charlie by the espionage CA. Perhaps create some certificates for employees of the defense department and the education department. These certificates should be signed by the appropriate intermediate CA.

Now modify the verification program to take a chain of certificates instead of just a single certificate. Get rid of the command-line parameter for the issuer's public key and instead hard-code which of the root certificate filenames are trusted. To specify the chain of certificates, have the program take the claimed identity as the first input (as it already does) and then an arbitrary number of certificates to follow. Each certificate's issuer field should indicate the next certificate in the chain. For example, to validate Charlie, there may be three certificates: `charlie.cert`, `espionage.cert`, `covert_root.cert`. The issuer of `charlie.cert` should have the same subject name as `espionage.cert` and so forth. The verify program should only accept an identity if the last certificate in the chain is already trusted.

Certificates are very important to modern cryptography and computer security. In Chapter 8, we will introduce real X.509 certificates and discuss how real CAs operate and additional issues and solutions as part of learning about TLS.

Revocation and Private Key Protection

Certificates and the public keys they contain are very powerful. At the same time, they come with a very dangerous Achilles heel. How do you disable them if the associated private key is compromised?

What we are talking about here is a concept called "revocation." To revoke a certificate is to reverse the endorsement of the issuer. HQ might have issued a certificate to Charlie, but if Charlie is captured and his private key lost, HQ needs to figure out a way to tell all of the other agents not to trust that certificate anymore.

Unfortunately, this is not easily done. If you recall, one of the reasons why CAs came into existence instead of online registries was the desire for offline verification. How can an offline verification process provide real-time revocation data?

The simple answer is, "It can't." There are only two options. Either the verification process must have a real-time component or the revocation cannot be updated in real time. Both options are available for certificates today in the form of the Online Certificate Status Protocol (OCSP), which checks a certificate's status on the fly, and Certificate Revocation Lists (CRLs), which are lists published from time to time with revoked certs. We will review both of these in more detail in Chapter 8.

Because of the difficulty of revoking a certificate, private keys **must** be protected with the utmost care. When real-time signing is not needed, private keys should be kept offline and in a secure environment. If they must be used in real time, and must be stored on a server, certificates should be stored with the minimum permissions necessary and readable on a strictly need-to-know basis. For end-user keys, such as those used for email and other applications, private keys stored on disk should be adequately protected by symmetric encryption with a strong password. Ideally, avoid storing private keys on desktops and servers altogether (especially in the modern era of continuous backups) and, instead, store private keys in a hardware security module.

It might not be a bad idea to keep certificates with a relatively short expiration date and rotate them as necessary.

Replay Attacks

There is one last security issue to address before moving on from message integrity. It applies equally to both MACs and signatures. The issue is *replay attacks*.

A replay attack occurs when a legitimate message from a previous communication is used by an attacker at a later time when it should no longer be valid.

Let's consider the following message: "We attack at dawn!"

We can secure this message from modification and authenticate the sender with either MACs or signatures. But what would prevent Eve from intercepting that message and sending it on *a different day*? Perhaps she would choose to send it on a day when the EA is *not* planning an attack? Eve may not be able to change the message contents; perhaps she cannot even read them, but that does not stop her from resending (replaying) the message whenever she wants.

For this reason, almost all cryptographically secured messages typically need some kind of unique component that distinguishes them from all other messages. This piece of data is often referred to as a *nonce*. In many circumstances a nonce can be a random number. If you take a quick peek back at Chapter 3, you will see that the IV value passed to AES counter mode was called a nonce. Nonces, especially random number nonces, are also used to keep messages from being the same when doing so would introduce security vulnerabilities.

However, to prevent replay attacks, simply using a random number won't do. In order to detect a replay, the receiver must *keep track* of the nonces that have been used and reject them when seen a second time.

This can be terribly problematic. How big of a list of nonces should be kept? A hundred? A thousand? Do you remove a nonce from the list after a certain period of time? If you do and the attacker knows it, the attacker can now use it in a replay. For example, if the attacker knows you only keep track of nonces received in the last 5 minutes, the attacker can replay something from *6* minutes ago with a reasonable amount of success.

Some systems use timestamps instead of random nonces. Using a timestamp, a receiver can reject data that is too old. The problem with this approach is that all of the computers have to have synchronized clocks for it to work reliably. Plus, data with an "old" timestamp must be accepted within some window. After all, the message won't arrive instantaneously. How large a window do you permit? However big it is, the bad guy will figure out a way to use it against you.

It's possible to combine both approaches together. You can send data with a timestamp *and* a random number. The timestamp is used to get rid of data that is *really* old and the nonce is used to prevent replays within the time window permitted. This means that the clocks only need to be relatively close (perhaps even within 24 hours) and that the list of nonces to be stored is bounded.

You have now seen *two* pieces of metadata that you need to consider sending in a message: the nonce and/or timestamp to prevent replays and the sender/receiver names. In general, you should put all relevant context into the message so that it cannot be used outside of that context.

EXERCISE 5.12. REPLAY IT AGAIN SAM!

Use either MAC or signatures to send a message from Alice to Bob or vice versa. Include a nonce in the message to prevent replays using all three mechanisms described in this section. Send some replays from Eve and try to get around Alice and Bob's defenses.

Summarize-Then-MAC

Another chapter, another firehose of information! In this chapter we covered message authentication codes, which are keyed codes computed over a series of data. Without the key, it is impossible to change the data undetectably. Moreover, when two parties share a MAC key, they can be sure that (unless the shared key has been compromised) if one of them received a correctly MACed message, it came from the other party.

Using asymmetric operations, one can use a private key to create a signature over a piece of data (typically over the hash of the data). Unlike MAC operations, which can only ensure correctness and authenticity to those individuals sharing the key, a widely distributed public key can be used theoretically by anyone (that trusts it) to validate the signature over the data.

And we also provided a quick overview of basic certificate operations.

And now that our summary is complete, here is the HMAC-SHA256 (in hex) over the preceding three paragraphs (i.e., from “Another chapter…” through “… certificate operations.”) using our twice-cited XKCD password:

```
c4d60c7336911cd0a23132f11ae1ca8ba392a05ae357c81bc995876693886b9e
```

Now you have a way of telling if any corrections or changes were made to this summary by our editors after we submitted it to them!

CHAPTER 6

Combining Asymmetric and Symmetric Algorithms

In this chapter, we'll spend time getting familiar with how asymmetric encryption is typically used, where it is a critical *part* of communication privacy, but not responsible for *all* of it. Typically, asymmetric encryption, also known as "public key cryptography", is used to establish a trusted *session* between two parties, and the communication while within that session is protected with much faster symmetric methods.

Let's dive in with a short example and some code!

Exchange AES Keys with RSA

Armed with their newer cryptography technologies, Alice and Bob have become more brazen in their covert operations. Alice has managed to infiltrate the Snow Den Records Center in West Antarctica and is attempting to steal a document related to genetic experiments to turn penguins completely white, thus creating a perfectly camouflaged Antarctic soldier. WA soldiers are quickly moving on her position, and she decides to risk transmitting the document over a short-wave radio to Bob who is monitoring from outside the building. Eve is certainly listening and Alice does not want her to know which document has been stolen.

The document is nearly ten megabytes. RSA encryption of the entire document will take *forever*. Fortunately, she and Bob agreed beforehand to use RSA encryption to send AES session keys and then transmit the document using AES-CTR with HMAC. Let's create the code they will use to make this alphabet soup work.

S. J. Nielson and C. K. Monson, *Practical Cryptography in Python*,
https://doi.org/10.1007/978-1-4842-4900-0_6

First, let's assume that Alice and Bob already have each other's certificates and public keys. Bob cannot risk giving away his position by transmitting; he will be limited to monitoring the channel, and Alice will just have to hope that the message is received. The agreed-upon transmission protocol is to transmit a single stream of bytes with all data concatenated together. The transmission stream includes

- An AES encryption key, IV, and a MAC key encrypted under Bob's public key
- Alice's signature over the hash of the AES key, IV, and MAC key
- The stolen document bytes, encrypted under the encryption key
- An HMAC over the entire transmission under the MAC key

As we have done before, let's create a class to manage this transmission process. The code snippet in Listing 6-1 shows key pieces of the operations.

Listing 6-1. RSA Key Exchange

```
import os
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives import hashes, hmac
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import padding, rsa

# WARNING: This code is NOT secure. DO NOT USE!
class TransmissionManager:
    def __init__(self, send_private_key, recv_public_key):
        self.send_private_key = send_private_key
        self.recv_public_key = recv_public_key
        self.ekey = os.urandom(32)
        self.mkey = os.urandom(32)
        self.iv = os.urandom(16)

        self.encryptor = Cipher(
             algorithms.AES(self.ekey),
             modes.CTR(self.iv),
             backend=default_backend()).encryptor()
        self.mac = hmac.HMAC(
             self.mkey,
             hashes.SHA256(),
             backend=default_backend())

    def initialize(self):
        data = self.ekey + self.iv + self.mkey
        h = hashes.Hash(hashes.SHA256(), backend=default_backend())
        h.update(data)
        data_digest = h.finalize()
        signature = self.send_private_key.sign(
            data_digest,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH),
            hashes.SHA256())
        ciphertext = self.recv_public_key.encrypt(
            data,
            padding.OAEP(
                mgf=padding.MGF1(algorithm=hashes.SHA256()),
                algorithm=hashes.SHA256(),
                label=None)) # rarely used.Just leave it 'None'
        ciphertext = data+signature
        self.mac.update(ciphertext)
        return ciphertext

    def update(self, plaintext):
        ciphertext = self.encryptor.update(plaintext)
        self.mac.update(ciphertext)
        return ciphertext

    def finalize(self):
        return self.mac.finalize()
```

Hopefully, all of the pieces here are familiar, and if you follow the code paths, it should also be pretty easy to see how things come together. Perhaps you've noticed that we are drawing on concepts from Chapter 3, Chapter 4, and Chapter 5! All of these pieces are coming together to shape a more advanced whole.

A few points are worth noting. First, we chose to use AES-CTR, so there is no need for padding. Earlier in the book, we used the term "nonce" to describe the initialization value to the algorithm, as this is what the `cryptography` library calls it. In other literature it is still called an IV, however, so we use that term here. Either way, the IV (or nonce) is the starting value for the counter.

Note that we are not using Sign-Then-Encrypt as we discussed in Chapter 5. As always, this is an example program not meant to be used for real security. You might want to review the issues we discussed in relation to Sign-Then-Encrypt to see how Eve could strip out the signature, change the keys, and re-sign.

Nevertheless, that's not the major vulnerability we will discuss. After all, in our scenario, Bob is probably only going to accept data from Alice. The issues of swapping out a signature are more applicable when more than one signature can be accepted.

Like most of the APIs you've seen so far, we use `update` and `finalize`, but we added a new method called `initialize`. For transmission, Alice would call `initialize` first to get the signed and encrypted header with the session keys. Next, she would call `update` as many times as needed to feed the entire document through. When everything is finished, she would call `finalize` to get the HMAC trailer over all of the transmitted contents.

EXERCISE 6.1. BOB'S RECEIVER

Implement the reverse of this transmitter by creating a `ReceiverManager`. The exact API might vary a little, but you will probably need at least an `update` and `finalize` method. You will need to unpack the keys and IV using Bob's private key and verify the signature using Alice's public key. Then, you will decrypt data until it's exhausted, finally verifying the HMAC over all received data.

Remember, the last bytes of the transmission are the HMAC trailer and are not data to be decrypted by AES. But when `update` is called, you may not yet know whether these are the last bytes or not! Think through it carefully!

Asymmetric and Symmetric: Like Chocolate and Peanut Butter

Hopefully, Alice's transmission to Bob in the exercise at the beginning of this chapter gave you a small taste for how asymmetric and symmetric cryptography work together. The protocol we outlined in code works, but lacks some important subtlety, as is often the case with our first attempts. As you might expect by now, the preceding code is *not secure* and we will demonstrate at least one problem with it shortly. It does illustrate the ideas behind putting the two systems together, though.

Let's see what we can learn from what we have. We'll start with session keys.

We first introduced the term *session key* in Chapter 4 but did not discuss it much. A session key is by nature a temporary thing; it is used for a single communication session and then discarded permanently, never to be reused. In the preceding code, notice that the AES and MAC keys are generated at the beginning of a session by the communications manager. Every time a new communications manager is created, a new set of keys is created. The keys are not stored or recorded anywhere. Once all the data is encrypted, they are thrown away.[1]

On the receiving end, the session keys are decrypted using the recipient's private key. Once these keys are decrypted, they are then used to decrypt the rest of the data and process the MAC. Again, after the transmitted data is processed, the keys can—and should—be destroyed.

Symmetric keys make good session keys for multiple reasons. First of all, symmetric keys are easy to create; in our example, we simply generated random bytes. We could also *derive* symmetric keys from a base secret by using key derivation functions. This is a common approach, and we will see later that you almost always need to derive *multiple* keys for typical secure communication. Regardless of how they are created, symmetric keys (and IVs) are plain old ordinary bytes, unlike most asymmetric keys that require some additional structure (e.g., public exponents, chosen elliptic curves, etc.).

Second, symmetric keys are good session keys because symmetric algorithms are *fast*. We have already mentioned this a time or two, but it is worth repeating. AES is typically on the order of hundreds of times faster than RSA, so the more data that can be

[1] Well, they *should* be thrown away. In real applications, this might mean securely overwriting memory with zeros and ensuring that all copies are accounted for. It's not paranoia when they are actually out to get you.

encrypted by AES, the better. This is another reason that symmetric keys are sometimes called "bulk data transfer" keys.

Finally, let's also recognize that symmetric keys are good session keys because they are *not* always good long-term keys! Remember, symmetric keys cannot be *private* keys because they must always be shared between at least two parties. The longer a shared key is in use, the higher the risk that trust breaks down between the parties and the key should no longer be shared. In the case of Alice's break-in to the Snow Den archive, she risks capture and the compromise of any keys she has with her. The loss of her asymmetric private key is serious, as we discussed when we talked about certificate revocation, but if Alice and Bob were using the same shared symmetric key for all of their communications, the loss of that key would be even worse as any intercepted communications between them that were encrypted using that key could now be decrypted.

On the other hand, asymmetric keys are very useful for long-term identification. Using certificates, asymmetric keys can establish a sort of proof of identity; once this is done, the shorter-term keys do the work of actually transmitting data between the authenticated parties. That said, sometimes *ephemeral* (quickly discarded) asymmetric keys are incredibly valuable. We will see this both with key exchanges that have the "forward secrecy" attribute and with how ransomware attackers lock up their victims' files.

Measuring RSA's Relative Performance

Even though we've hammered home just how much slower RSA is than AES, let's have some fun and run a few experiments. We're going to write a tester that will generate random test vectors for encryption and decryption. Then we can compare the performance of RSA and AES for ourselves.

For this walk-through, we are going to create a more complex file from smaller bits. Listing 6-2 shows the imports for the overall script. You can start with this as a skeleton and build/copy the other pieces in.

Listing 6-2. Imports for Encryption Speed Test

```
# Partial Listing: Some Assembly Required

# Encrypt ion Speed Test Component
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes
import time, os
```

Let's start by creating some algorithms to test. We will define one class for each algorithm, and the instances of the class will build encryption and decryption objects. Builders will be self-contained, providing all keys and necessary configuration. Each will have a name attribute with a human-readable label and a get_cipher_pair() method to create a new encryptor and decryptor. This method must generate new encryption and decryption objects each time it is called.

AES is very straightforward because the cryptography library already provides most of the machinery, as shown in Listing 6-3.

Listing 6-3. AES Library Use

```
# Partial Listing: Some Assembly Required

# Encryption Speed Test Component
class AESCTRAlgorithm:
    def __init__(self):
        self.name = "AES-CTR"

    def get_cipher_pair(self):
        key = os.urandom(32)
        nonce = os.urandom(16)
        aes_context = Cipher(
            algorithms.AES(key),
            modes.CTR(nonce),
            backend=default_backend())
        return aes_context.encryptor(), aes_context.decryptor()
```

The get_cipher_pair() operation creates new keys and nonces each time it is invoked. We could put this in the constructor because we don't really care if we reuse keys for these speed tests, but the re-generation of a few bytes for a key and a nonce is probably not really a limiting factor for speed.

RSA encryption is a little more complicated. It really wasn't meant to encrypt arbitrary amounts of data. Unlike AES, which has counter and CBC modes to tie blocks together, RSA must encrypt its data all at once, and the size of data it can manipulate is limited by various factors. An RSA key with a 2048-bit modulus cannot encrypt more than 256 bytes at a time. In fact, once you add in the OAEP (with SHA-256 hash) padding, it's significantly less: only 190 bytes![2]

If we were actually concerned about the security of the encryption, we could thus not use RSA for more than 190 bytes of data. However, what we are really testing here is a *hypothetical* RSA encryptor that does not exist in the real world. What we want to explore is this: *if* RSA could encrypt arbitrary amounts of data, how long would it take? For this test, we will encrypt each chunk of 190 bytes one at a time and concatenate the results together. Note that when we encrypt with the OAEP padding, the 190 bytes of plaintext becomes 256 bytes of ciphertext. When we are decrypting, we need to decrypt 256-byte chunks.

Although a truly secure RSA encryption algorithm would have to bind the bytes of the different individual encryption operations together, this version is the *fastest* it could ever be, so it gives us an upper bound on speed, making for an interesting comparison.

With this in mind, we can construct our RSA encryption and decryption algorithm like Listing 6-4.

Listing 6-4. RSA Implementation

```
# Partial Listing: Some Assembly Required

# Encryption Speed Test Component
class RSAEncryptor:
    def __init__(self, public_key, max_encrypt_size):
        self._public_key = public_key
        self._max_encrypt_size = max_encrypt_size

    def update(self, plaintext):
```

[2] Don't confuse bytes and bits here! Even an AES-256 key is 256 *bits*, or just 32 bytes. So RSA can safely hold even a "large" AES key.

```
        ciphertext = b""
        for offset in range(0, len(plaintext), self._max_encrypt_size):
            ciphertext += self._public_key.encrypt(
                plaintext[offset:offset+self._max_encrypt_size],
                padding.OAEP(
                    mgf=padding.MGF1(algorithm=hashes.SHA256()),
                    algorithm=hashes.SHA256(),
                    label=None))
        return ciphertext

    def finalize(self):
        return b""

class RSADecryptor:
    def __init__(self, private_key, max_decrypt_size):
        self._private_key = private_key
        self._max_decrypt_size = max_decrypt_size

    def update(self, ciphertext):
        plaintext = b""
        for offset in range(0, len(ciphertext), self._max_decrypt_size):
            plaintext += self._private_key.decrypt(
                ciphertext[offset:offset+self._max_decrypt_size],
                padding.OAEP(
                    mgf=padding.MGF1(algorithm=hashes.SHA256()),
                    algorithm=hashes.SHA256(),
                    label=None))
        return plaintext

    def finalize(self):
        return b""

class RSAAlgorithm:
    def __init__(self):
        self.name = "RSA Encryption"

    def get_cipher_pair(self):
        rsa_private_key = rsa.generate_private_key(
          public_exponent=65537,
          key_size=2048,
          backend=default_backend())
        max_plaintext_size = 190 # largest for 2048 key and OAEP
        max_ciphertext_size = 256
        rsa_public_key = rsa_private_key.public_key()
        return (RSAEncryptor(rsa_public_key, max_plaintext_size),
                RSADecryptor(rsa_private_key, max_ciphertext_size))
```

Note that we created our encryptor and decryptor to have the same API as the AES encryptor and decryptor. Namely, we provide `update` and `finalize` methods. The finalize methods don't do anything as RSA encryption (with padding) processes each chunk exactly the same way. The chunk-by-chunk encryption takes each 190-byte piece of the input, encrypts it to the 256-byte ciphertext, and returns the concatenation of all of these pieces. The decryptor reverses the processes, taking each 256-byte chunk in for decryption. Our `RSAAlgorithm` class constructs the appropriate encryptor and decryptor using these classes.

Now that we have a couple of algorithms to test, we need to create a mechanism for generating plaintext and keeping track of the encryption and decryption times. To this end, we created a class in Listing 6-5 that generates plaintexts randomly and receives notification of each individual ciphertext chunk produced. When the test calls for ciphertexts for the subsequent decryption test stage, it replays those ciphertext chunks exactly how it received them. Based on the notification of encrypted ciphertexts and decrypted plaintexts, it can also keep track of how long the overall operation takes.

Listing 6-5. Random Text Generation

```
# Partial Listing: Some Assembly Required

# Encryption Speed Test Component
class random_data_generator:
    def __init__(self, max_size, chunk_size):
        self._max_size = max_size
        self._chunk_size = chunk_size

        # plaintexts will be generated,
        # ciphertexts recorded
        self._ciphertexts = []

        self._encryption_times = [0, 0]
        self._decryption_times = [0,0]

    def plaintexts(self):
        self._encryption_times[0] = time.time()
        for i in range(0, self._max_size, self._chunk_size):
            yield os.urandom(self._chunk_size)

    def ciphertexts(self):
        self._decryption_times[0] = time.time()
        for ciphertext in self._ciphertexts:
            yield ciphertext

    def record_ciphertext(self, c):
        self._ciphertexts.append(c)
        self._encryption_times [1] = time.time()

    def record_recovertext(self, r):
        # don't store, just record time
        self._decryption_times[1] = time.time()

    def encryption_time(self):
        return self._encryption_times [1] - self._encryption_times [0]

    def decryption_time(self):
        return self._decryption_times [1] - self._decryption_times [0]
```

Notice that a new random_data_generator contains timing and data specific to each individual test run. So a new object needs to be created for each test.

Now, armed with an algorithm and a data generator, we can, as in Listing 6-6, write a fairly generic test function.

Listing 6-6. Encryption Tester

```
# Partial Listing: Some Assembly Required

# Encryption Speed Test Component
def test_encryption(algorithm, test_data):
    encryptor, decryptor = algorithm.get_cipher_pair()

    # run encryption tests
    # might be slower than decryption because generates data
    for plaintext in test_data.plaintexts():
        ciphertext = encryptor.update(plaintext)
        test_data.record_ciphertext(ciphertext)
    last_ciphertext = encryptor.finalize()
    test_data.record_ciphertext(last_ciphertext)

    # run decryption tests
    # decrypt the data already encrypted
    for ciphertext in test_data.ciphertexts():
        recovertext = decryptor.update(ciphertext)
        test_data.record_recovertext(recovertext)
    last_recovertext = decryptor.finalize()
    test_data.record_recovertext(last_recovertext)
```

Using these building blocks, we can test these encryption algorithms over various chunk sizes to see if speed increases or decreases based on the amount of data they're handling. For example, Listing 6-7 is a test of AES-CTR and RSA on 100MB of data with chunk sizes ranging from 1 KiB to 1 MiB.

Listing 6-7. Algorithm Tester

```
# Encryption Speed Test Component
test_algorithms = [RSAAlgorithm(), AESCTRAlgorithm()]

data_size = 100 * 1024 * 1024 # 100 MiB
chunk_sizes = [1*1024, 4*1024, 16*1024, 1024*1024]
stats = { algorithm.name : {} for algorithm in test_algorithms }
```

```
for chunk_size in chunk_sizes:
    for algorithm in test_algorithms:
        test_data = random_data_generator(data_size, chunk_size)
        test_encryption(algorithm, test_data)
        stats[algorithm.name][chunk_size] = (
            test_data.encryption_time(),
            test_data.decryption_time())
```

The `stats` dictionary is used for holding encryption and decryption times for the various algorithms in the various tests. These can be used to generate some fun graphs. For example, Figure 6-1 and Figure 6-2 are the encryption and decryption graphs for the tests we ran.

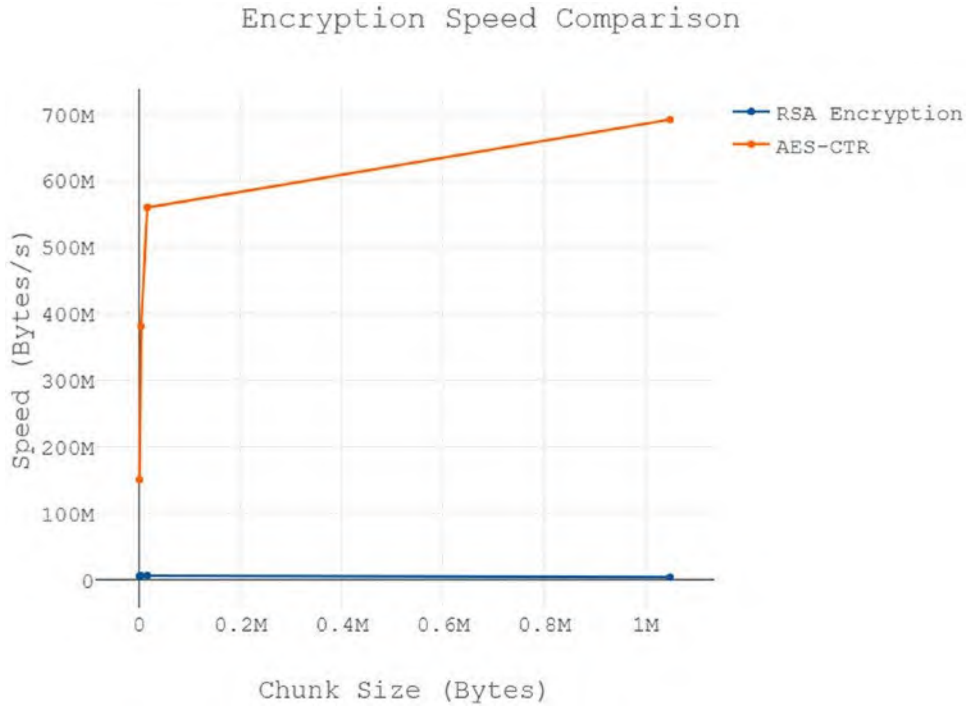

***Figure 6-1.** A Comparison of RSA encryption speeds vs. AES-CTR*

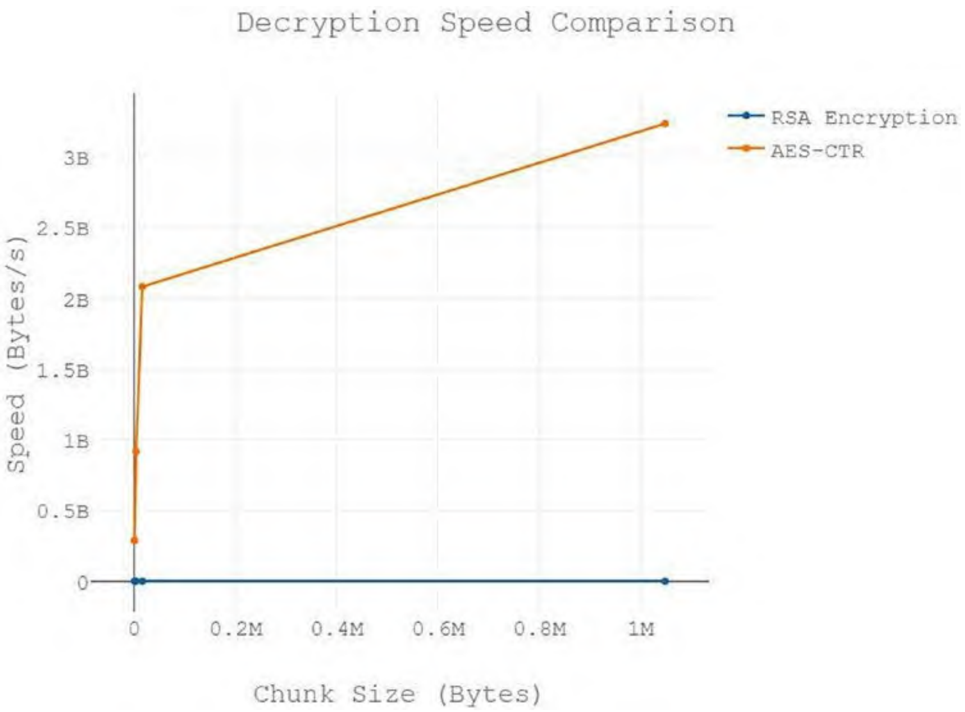

Figure 6-2. *A Comparison of RSA decryption speeds vs. AES-CTR*

As you can see, RSA operations are so much slower, it's not even really a comparison. By the way, if you run the tests we did, RSA encryption over 100 MiB can be slow (about 20 seconds on our computer), but decryption is so bad, it's just off the charts (about *400 seconds* for our tests!). RSA decryption is slower than RSA encryption, so this isn't a surprise. When you have tests that run this long, make sure to save the statistics in raw, numerical format and then generate graphs from this data. That way you can regenerate graphs quickly and easily without running the entire test again.

EXERCISE 6.2. RSA RACING!

Use your preceding tester to compare the performance of RSA with a 1024-bit modulus, a 2048-bit modulus, and a 4096-bit modulus. Please note, you will need to change your chunk size to 62 bytes for 1024-bit RSA keys with OAEP (and SHA-256 hashing) and 446 bytes for 4096-bit RSA keys with OAEP (and SHA-256 hashing).

EXERCISE 6.3. COUNTERS VS. CHAINS!

Use your tester to compare the performance of AES-CTR against AES-CBC.

EXERCISE 6.4. MACS VS. SIGNATURES

Modify your algorithms to sign or apply a MAC to the data in the `finalize` methods. Try disabling encryption (just have the update methods return the unmodified plaintext) so that you can compare only the speed of the MAC and signature. Is the difference as extreme? Can you think why this is so?

EXERCISE 6.5. ECDSA VS. RSA SIGNING

In addition to testing the speed of MAC vs. RSA signing, also compare the speed of RSA signing with ECDSA signing. It's hard to get a fair comparison because it isn't always obvious what your key size is with ECDSA, but look at the list of supported curves in the `cryptography` library documentation and try them out to see which ones are faster in general, as well as how they compare to RSA signing using different modulus sizes.

Hopefully, these timed tests have helped to reinforce why, security reasons aside, symmetric ciphers are preferred over asymmetric ciphers for bulk data transfer.

Diffie-Hellman and Key Agreement

For the last couple of sections in this chapter, we will look at another type of asymmetric cryptography known as Diffie-Hellman (or DH) and a more recent variant called Elliptic-Curve Diffie-Hellman (or ECDH).

DH is a little different than RSA. Where RSA can be used to encrypt and decrypt messages, DH is only used for exchanging keys. In fact, it is technically called the Diffie-Hellman key exchange. As we have already explored in this chapter, outside of signatures, RSA encryption is largely used only for transmitting keys, also called "key transport." This means that if Alice has Bob's RSA public key, Alice can send Bob an encrypted key that only Bob can decrypt.

Figure 6-3 shows key transport in a TLS 1.2 handshake. We will discuss the TLS 1.2 handshake in more detail in Chapter 8, where this figure will also make an appearance.

But notice that the client in this figure can generate a random session key, encrypt it under the server's public key, and "transport" it back. This process also proves the server is in possession of the certificate because only the server could decrypt the session key and use it to communicate. No signatures are required for the server.[3]

On the other hand, DH and ECDH actually create a key seemingly out of thin air. No secrets are transmitted between the parties, encrypted or otherwise. Instead, they exchange public parameters that allow them to simultaneously compute the same key on both sides. This process is called *key agreement*.

To get started, Diffie-Hellman creates a pair of mathematical numbers, one private, one public, for each participant. The DH and ECDH key agreement protocol requires that *both* Alice and Bob have key pairs. In over-simplistic terms, Alice and Bob share their public keys with each other. The foreign public key and the local private key—when combined—create a shared secret on both sides.

A non-mathematical explanation in A. J. Han Vinck's course "Introduction to Public Key Cryptography" [14] is depicted in Figure 6-4.

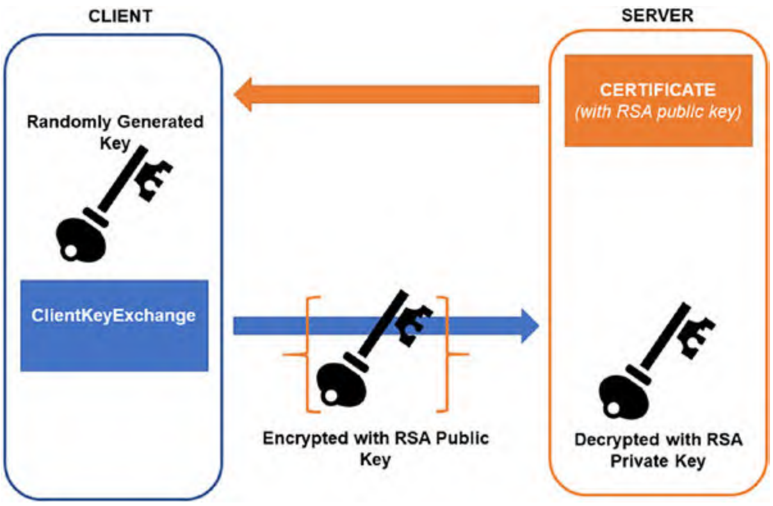

Figure 6-3. *An illustration of key transport using TLS*

Please note that, unlike RSA, DH and ECDH do not allow for the transmission of arbitrary data. Alice can send Bob any message she chooses, encrypted under Bob's RSA public key. Using DH or ECDH, however, all that the two can do is agree upon some

[3] TLS does not typically authenticate the client. But if client authentication is requested, it will have to independently prove it is the owner of a certificate by signing a challenge nonce from the server.

random data; they don't get to choose the message contents. The random data can be, and typically is, used as a symmetric key or for deriving symmetric keys.

In addition to not being able to exchange arbitrary contents, key exchange is also limited in that it requires *bidirectional* information exchange. In our scenario at the beginning of this chapter, Bob could not transmit for fear of discovery. Were that actually the case, DH and ECDH key exchanges would be impossible and RSA encryption would be the only option. This really isn't an issue in almost all real-world scenarios. In real Internet applications, we typically assume that both sides are free to communicate with each other.

Coding a DH key exchange in Python is straightforward. The example in Listing 6-8 is taken, with a few simplifications, directly from the cryptography module's online documentation.

Listing 6-8. Diffie-Hellman Key Exchange

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import dh
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.backends import default_backend

# Generate some parameters. These can be reused.
parameters = dh.generate_parameters(generator=2, key_size=1024,
                                      backend=default_backend())

# Generate a private key for use in the exchange.
private_key = parameters.generate_private_key()

# In a real handshake the peer_public_key will be received from the
# other party. For this example we'll generate another private key and
# get a public key from that. Note that in a DH handshake both peers
# must agree on a common set of parameters.
peer_public_key = parameters.generate_private_key().public_key()
shared_key = private_key.exchange(peer_public_key)

# Perform key derivation.
derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data',
        backend=default_backend()
    ).derive(shared_key)
```

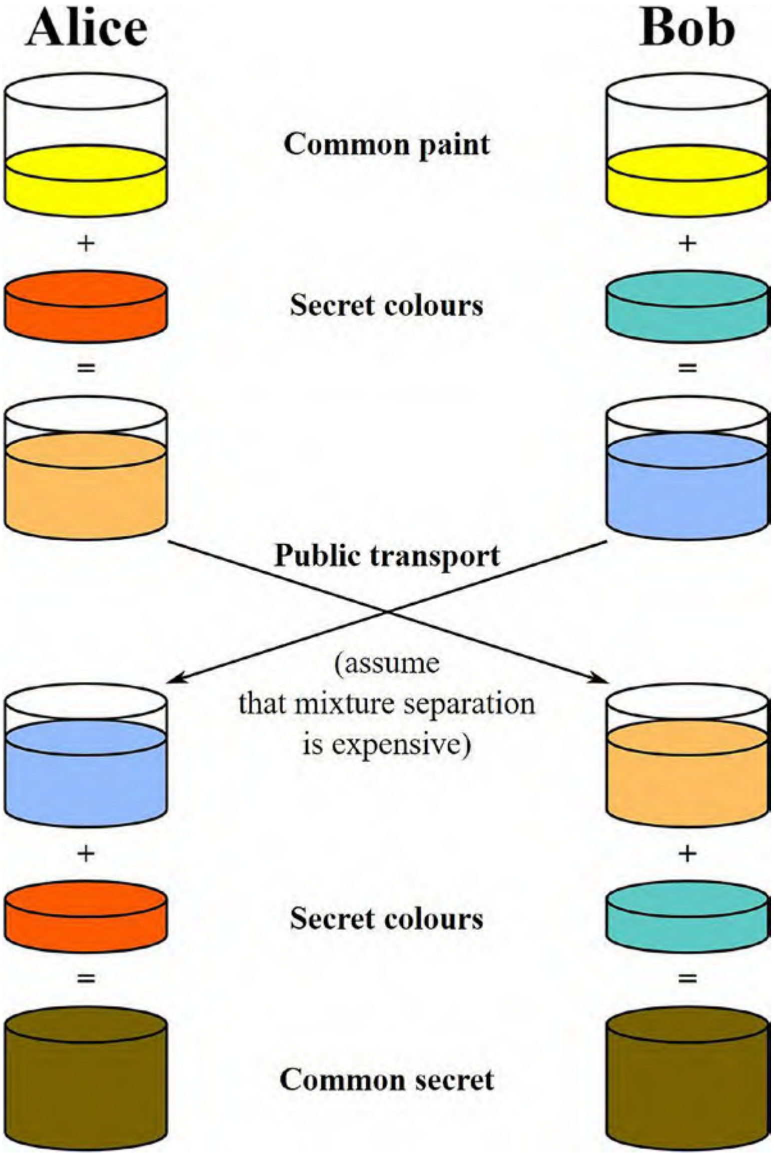

Figure 6-4. *Intuition behind Diffie-Hellman*

Unlike RSA there are a lot fewer pitfalls and gotchas.

There are only two parameters to the exchange: the generator and the key size. There are only two legal values for the generator, 2 and 5. Strangely enough, for a cryptographic protocol, the choice of generator doesn't matter for security reasons but must be the same for both sides of the exchange.

The key size, however, is important and should be at least 2048 bits. Key lengths between 512 and 1024 bits are vulnerable to known attack methods.

Warning: Slow Parameter Generation

Diffie-Hellman is touted as being pretty fast for generating keys on the fly. However, generating the parameters that can generate keys can be pretty slow. We warned you against using key sizes smaller than 2048 and then used 1024 in our own code example. We wanted to give you code that wouldn't take forever to run to illustrate the basic operations.

So if parameter generation is so slow, why do we say DH is fast? The same parameters can generate many keys so the cost is amortized. So make sure not to regenerate the parameters with every key generation, or DH will run unacceptably slow. Alternatively, use ECDH which is much faster.

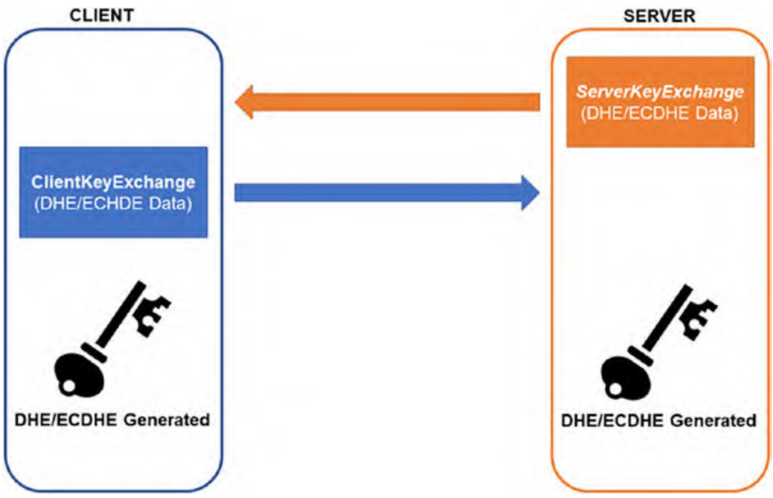

Figure 6-5. *An illustration of key agreement using TLS*

The other recommended setting is to derive another key from the shared key rather than using the shared key directly. The key derivation function is similar to the ones we looked at in Chapter 3.

The TLS 1.2 handshake can either do the key transport using RSA encryption or key agreement using DH/ECDH. Again, this will be discussed in Chapter 8 in detail, but Figure 6-5 shows that both sides exchange the public data, derive a key, and can communicate using the agreed-upon keys. Unlike key transport, however, there is no authentication. Either or both sides will have to sign the public data to prove that possession of a public key.

Elliptic-Curve Diffie-Hellman (or ECDH) is a variant of DH that is becoming popular in modern use. It works in the same way but uses elliptic curves for some of the internal mathematical computations. The code for using ECDH is almost identical to DH in the cryptography module as shown in Listing 6-9.

Listing 6-9. Elliptic-Curve DH

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

# Generate a private key for use in the exchange.
private_key = ec.generate_private_key(
    ec.SECP384R1(), default_backend()
)
# In a real handshake the peer_public_key will be received from the
# other party. For this example we'll generate another private key
# and get a public key from that.
peer_public_key = ec.generate_private_key(
    ec.SECP384R1(), default_backend()
).public_key()
shared_key = private_key.exchange(ec.ECDH(), peer_public_key)

# Perform key derivation.
derived_key = HKDF(
    algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data ',
        backend=default_backend()
    ).derive(shared_key)
```

In most circumstances, creating keys with DH or ECDH key agreement is preferred over RSA key exchange. There are a number of reasons but perhaps the biggest one is *forward secrecy*.

Diffie-Hellman and Forward Secrecy

Using RSA encryption, we can generate a symmetric key, encrypt it under someone's public key, and send it to them. This allows both parties to share a session key securely, provided that the exchange protocol follows certain rules. There are even ways to have both sides contribute to a key. Each party could send the other some random data, and the concatenation of both could be fed to a hash function to produce the session key.

Unfortunately, RSA key transport does *not* provide a really fantastic property called *forward secrecy*. Forward secrecy means that even if a key is eventually compromised, it *does not reveal anything about* ***previous*** *communications*.

Let's go back to Alice, Bob, and Eve. Alice and Bob already assume that Eve is recording everything that is transmitted. That's why they are encrypting the transmission in the first place. So, after the transmission is complete, Eve has a recording of the ciphertext that she cannot yet decrypt. But, rather than toss it aside, Eve files it away in storage.

But also recall that, in our scenario, Alice actually believed she was on the verge of being captured. If the guards capture her and her keys are compromised, what is lost? Fortunately, nothing. Remember that Alice encrypted the session keys under *Bob's* public key. Capturing Alice won't make it any easier to decrypt that data (do you see the advantage of this over shared keys?).

But suppose that Eve finds Bob, perhaps even a long time in the future. Even if it's years later, if Eve manages to get Bob's private key, she can go back to her recording of Alice's previous transmission and decrypt it! Bob's private key will still decrypt the session keys, and Eve will then be able to decrypt the entire transmission.

Forward secrecy is much stronger than this. If a protocol has forward secrecy, Eve can *never* recover data from a terminated session no matter what long-term keys she manages to obtain. Forward secrecy is not possible when session keys are sent directly via RSA encryption (in the way we've just described) because once the RSA private key is compromised, any recorded data from previous sessions is now vulnerable.

With Diffie-Hellman (DH) and Elliptic-Curve Diffie-Hellman (ECDH), forward secrecy is achieved by the use of *ephemeral keys*. RSA produces an ephemeral *symmetric* session key, but DH and ECDH actually produce ephemeral *asymmetric* keys as well! A new ephemeral key pair is (or should be!) generated with every single key agreement operation and then *thrown away*. The symmetric key is also ephemeral and thrown away after each session. Because DH and ECDH are typically used in this fashion, an "E" is often tacked onto the end of the acronym (DHE or ECDHE).[4]

Now that a new key pair is used for every exchange, compromising a single asymmetric key only reveals a single symmetric key and, accordingly, a single communications session. And when the ephemeral DH and ECDH private keys are properly disposed of, there is no key left for Eve to compromise and no way that she could ever decrypt these sessions. In some ways, it is akin to the old spy trope of swallowing the key so the spy's nemesis can't unlock the movie's McGuffin.

Observe that, in theory, Alice and Bob could also do an *ephemeral* RSA key exchange. They *could* generate new RSA key pairs for *every single key transport*, sending each other their new public keys before transmitting the session key, then destroying the key pair after transmission.

The problem is that generating RSA keys is slow in computer terms. You might not have thought it took very long to generate your RSA keys for the examples in this book, but for computers involved in rapid communications (such as setting up a secure connection from your browser to a web site), RSA is mind-numbingly slow. DH and ECDH are much, much faster. Because of the key generation speed, DH and ECDH are the common choices for forward-secrecy-style communications.

This ephemeral mode of operation is the preferred mode for DH and ECDH under almost all circumstances, which is why DH and ECDH often mean DHE and ECDHE.

[4] The TLS protocol, which we'll talk about at the end of this book, is very strict. When TLS says "DH," it does *not* mean DHE and vice versa. This distinction is not always so clear in other contexts.

EXERCISE 6.6. OFF TO THE RACES!

Write a python program to generate a thousand or so 2048-bit RSA private keys and a program to generate a thousand or so DH and ECDH keys. How does the performance compare?

There's only one other limitation to Diffie-Hellman methods: they have no *authentication*. Because the keys are completely ephemeral, there is no way to tie them to an identity; you don't know with whom you are speaking. Remember that beyond communicating confidentially, we need to know with whom we're communicating confidentially. By itself, DH and ECDH do not provide any such assurances.

For this reason, many DH and ECDH key exchanges also require a long-term public key, such as RSA or ECDSA, and that key is typically protected within a signed certificate. These long-term keys, however, are never used for encryption or key transport and are not used in the actual exchange of key data in any way. Their sole purpose is to establish the identity of the other party, usually by signing some of the ephemeral DH/ECDH data being exchanged, and to ensure freshness via some kind of challenge or nonce.

Remember, to ensure forward secrecy, the Diffie-Hellman parameters must be regenerated for *every key exchange*. If you take a look through the `cryptography` library documentation, you'll notice that they include sample code that, as written, does *not* provide forward secrecy. This code sample saves what should be a single-use key for later. Make sure that your keys are destroyed after use (never logged).

Now that we've walked through the very high-level concepts, let's help Alice and Bob with authenticated ECDH key exchange code. First we will create some code for the key exchange (Listing 6-10), and then we'll modify it to be authenticated.

Listing 6-10. Unauthenticated ECDH

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF

class ECDHExchange:
    def __init__(self, curve):
        self._curve = curve

        # Generate an ephemeral private key for use in the exchange.
        self._private_key = ec.generate_private_key(
             curve, default_backend())

        self.enc_key = None
        self.mac_key = None

    def get_public_bytes(self):
        public_key = self._private_key.public_key()
        raw_bytes = public_key.public_bytes(
            encoding=serialization.Encoding.PEM,
            format=serialization.PublicFormat.SubjectPublicKeyInfo)
        return raw_bytes

    def generate_session_key(self, peer_bytes):
        peer_public_key = serialization.load_pem_public_key(
            peer_bytes,
            backend=default_backend())
        shared_key = self._private_key.exchange(
            ec. ECDH(),
            peer_public_key)

        # derive 64 bytes of key material for 2 32-byte keys
        key_material = HKDF(
            algorithm=hashes.SHA256(),
            length=64,
            salt=None,
            info=None,
            backend=default_backend()).derive(shared_key)

        # get the encryption key
        self.enc_key = key_material[:32]

        # derive an MAC key
        self.mac_key = key_material[32:64]
```

To use the `ECDHExchange`, both parties instantiate the class and call the `get_public_bytes` method to get the data that needs to be sent to the other party. When those bytes are received, they are passed to `generate_session_key` where they are de-serialized into a public key and used to create a shared key.

So, what's with `HKDF`? This is a key derivation function that is useful for real-time network communications, but should not be used for data storage. It takes the shared key as input and derives a key (or key material) from it. Notice that in our example, we derive both an encryption key and a MAC key. This is done by using HKDF to derive 64 bytes of key material and then splitting it into two 32-byte keys. In reality, we need to derive a lot more data, and we'll discuss this in the next section. But for now, it demonstrates the basics of the ECDH exchange.

To repeat one last time, notice that ECDH is *generating* its private key on the fly. This key must be destroyed after every key exchange, along with any session keys created.

EXERCISE 6.7. RUDIMENTARY ECDH EXCHANGE

Use the `ECDHExchange` class to create shared keys between two parties. You will need to have two instances of the program running. Each program should write their public key bytes to disk for the other program to load. When they're finished, have them print out the bytes of the shared key so that you can verify that they both come up with the same key.

EXERCISE 6.8. NETWORK ECDH EXCHANGE

In the upcoming chapters, we will start using the network to exchange data between two peers. If you already know how to do some client-server programming, modify the previous ECDH exchange program to send the public data over the network instead of saving it to disk.

Our ECDH code so far just does the ECDH ephemeral key exchange. Both sides have a key, but since we aren't yet doing any authentication, neither side can be certain about whom they're talking to! Remember, the ephemeral nature of the ECDH keys means that they cannot be used to establish identity.

To remedy this, we are going to modify our ECDHExchange program to *also* be authenticated. In addition to an ephemeral asymmetric key, it will also use a long-term asymmetric key to sign the data.

Let's modify our ECDHExchange class and rename it AuthenticatedECDHExchange, which we do in Listing 6-11. First, we need to modify the constructor to take a long-term (*persistent*) private key as a parameter. This will be used for signing.

Listing 6-11. Authenticated ECHD

```
# Partial Listing: Some Assembly Required

from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
import struct # needed for get_signed_public_pytes

class AuthenticatedECDHExchange:
    def __init__(self, curve, auth_private_key):
        self._curve = curve
        self._private_key = ec.generate_private_key(
             self._curve,
             default_backend())
        self.enc_key = None
        self.mac_key = None

        self._auth_private_key = auth_private_key
```

Please note the difference between _private_key, which is generated and is ephemeral, and _auth_private_key. The latter is passed in as a parameter. This persistent key will be used to establish identity. We could use an RSA key here and it would work just fine, but in keeping with the elliptic-curve theme, we will assume this is an ECDSA key.

Instead of just generating public bytes to send to the other side, we will use Listing 6-12 to generate *signed* public bytes.

Listing 6-12. Authenticated ECDH Signed Public Bytes

```
# Partial Listing: Some Assembly Required

# Part of AuthenticatedECDHExchange class
def get_signed_public_bytes(self):
    public_key = self._private_key.public_key()

    # Here are the raw bytes.
    raw_bytes = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo)

    # This is a signature to prove who we are.
    signature = self._auth_private_key.sign(
        raw_bytes,
        ec.ECDSA(hashes.SHA256()))

    # Signature size is not fixed.Include a length field first.
    return struct.pack("I", len(signature)) + raw_bytes + signature
```

When the other side receives our data, they will need to unpack the first four bytes to get the length of the signature before they do anything else. The signature can be verified using the other party's long-term public key (just like we did with RSA). If the signature works out, we have some confidence that the ECDH parameters we received came from the expected party.

EXERCISE 6.9. ECDH LEFT TO THE READER

We did not show code for verifying the public parameters received in the `AuthenticatedECDHExchange` class. Luckily for you, we've left it as an exercise to the reader! Update the `generate_session_key` method to be `generate_authenticated_session_key`. This method should implement the algorithm previously described for getting the signature length, verifying the signature using a public key, and then deriving the session keys.

The principles in this section are important. You might consider working through this section a couple of times until you are comfortable with both sending a key encrypted under RSA and generating an ephemeral key on the fly using DH or ECDH. Make sure you also understand why the DH/ECDH approach has forward secrecy and the RSA version does not.

EXERCISE 6.10. BECAUSE YOU LOVE TORTURE

To emphasize that RSA technically could be used as an ephemeral exchange mechanism, modify your preceding ECDH program to generate an ephemeral set of RSA keys. Exchange the associate public keys and use each public key to send 32 bytes of random data to the other party. Combine both 32-byte transmissions with XOR to create a "shared key" and run it through HKDF just as the ECDH example does. Once you've proved to yourself that this works, review your results from Exercise 6.2 to see why this is too slow to be practical.

Also, creating a shared key with RSA encryption requires a round trip to create the key (transmission of certificate and reception of encrypted key), whereas DH and ECDH only require one transmission from each party to the other. When we learn about TLS 1.3, for example, you'll see how this can greatly impact performance.

Challenge-Response Protocols

We have briefly introduced challenge-response protocols in Chapter 5. In particular, Alice used challenge-response to validate that the man claiming to be Charlie was the owner of the certificate with the identity "charlie." At its core, a challenge-response protocol is about one party proving to another that they *currently* control either a shared secret or a private key. Let's look at both examples.

First, suppose that Alice and Bob share some key $K_{A,B}$. If Alice is communicating over a network with Bob, a simple authentication protocol is to send Bob a nonce N (potentially unencrypted) and ask him to encrypt it. For security reasons, it's a good idea for the response to include the identity of the communicating parties. Accordingly, Bob should reply with $\{A, B, N\}_{K_{A,B}}$. If only Alice and Bob share the key $K_{A,B}$, then only Bob could have responded to Alice's challenge correctly. Even if Eve overhears the challenge and knows N, she should not be able to encrypt it without the key.

For the asymmetric example, it is more or less the same, but uses signatures generated by private keys. This time, Bob is communicating with Alice over the network and wishes to be sure that he is talking to the real Alice. So he sends a nonce N and asks her to sign it with her public key. As with Bob's challenge, Alice should also send her name and Bob's. Her transmission should therefore look like $\{H(B, A, N)\}_{K^{-1}A}$ (for RSA signatures anyway). Bob verifies with the Alice's public key that the signature is correct. Only the possessor of the private key could have signed that challenge.

Challenge-response algorithms are relatively simple, but they can go wrong in many ways. For one thing, the nonce must be sufficiently large and sufficiently random to be unguessable, even with knowledge of previous transmissions. In the early days of remote keys for cars, for example, the transmitter used a 16-bit nonce. Thieves only had to record a transmission once and then interrogate the system over and over until it cycled through all the possible nonces and returned to the one they recorded. At that point, they could replay the nonce and gain access to the car.

Another way this can go wrong is via a "(hu)man-in-the-middle" (MITM) attack. Suppose that Eve wants to convince Alice that she is Bob. Eve waits until Bob wants to talk to Alice and then intercepts all of their communications. Then, she initiates communication with Alice pretending to be Bob. Alice responds with a challenge N to prove that the person she is talking to (Eve) is Bob. Eve immediately turns around and sends the challenge to Bob who, wanting to talk to Alice, was already expecting it. Bob happily signs the challenge and sends it back to Eve who forwards it directly on to Alice. (For a fascinating, but probably fictional example, Ross Anderson describes a "MIG-in-the-middle" scenario of this attack [1, Chap. 3].)

One way to defeat this MITM problem is to transmit information that only the true party can use. For example, even if Eve forwards along Bob's response to the challenge it won't help her if Alice's response is to send Bob a session key encrypted under his public key. Eve won't be able to decrypt it. If all subsequent communication takes place using that session key, Eve is still locked out. Alternatively, Alice and Bob could use ECDH plus signatures to generate a session key. Even if Eve can intercept every transmission between the two of them, Alice and Bob can create a session key that only they can use. The most Eve can do is block the communications.

The point here is to illustrate all the different kinds of considerations that need to go into authenticating the party you're talking to.

Once the identity of a party has been established, all subsequent communications of the session *must be tied to that authentication*. For example, Alice and Bob might authenticate one another using challenge-response, but unless they establish a session MAC key and use it to digest all of their subsequent communications, they cannot be sure who is sending the message.

Sometimes, initialization data must be sent in the clear before encrypted communications can be established. All of this data must also be tied together at some point. After session keys have been established, one option is to send a hash of all the unauthenticated data sent so far using the newly established secure channel. If the hash doesn't match what is expected, then the communicating parties can assume that an attacker, like Eve, has modified some of the initialization data.

In summary, when combining asymmetric and symmetric cryptography, don't just think about the confidentiality part (encryption). Remember that knowing to whom you are speaking is just as important as, if not more important than, knowing that the communications between the two of you are unreadable to anyone else. You might not want the whole world to read your love poetry, but you definitely don't want your amorous expressions to be received by the wrong person! Keep in mind that after establishing the other party's identity, you must ensure that there is a chain of authenticity for all the remaining communications for the rest of the session. If the initial identity is proved with signatures and the remaining data is authenticated by MACs, ensure that there is no break in the chain as you switch from one to the other.

Common Problems

After seeing a bit about how asymmetric and symmetric keys work together, you might be tempted to create your own protocol. Definitely resist that urge. The goal of these exercises is to teach you the *principles* and to illuminate your *understanding*, but that alone is not sufficient to prepare you to develop cryptographic protocols. The history of cryptography is littered with protocols that were later found to be exploitable even though they were written by cryptographers with more experience than you or I have.

Let's take the example we used with Alice sending the encrypted document to Bob. Did you notice that we broke one of our recommendations from previous chapters? Our data has no nonce! This means that Bob has no idea if the message from Alice is "fresh." What if that data was recorded by Eve from a year ago and is just being replayed now?

Here's another example. In our derivation of encryption keys, we only generated a single encryption key between both parties. This is only secure for *one-way communication!* If you want full-duplex communication (the ability to send data in both directions), you will need an encryption key for each direction!

But wait. Why can't we use the same key to send data from Alice to Bob as we use to send data from Bob to Alice?

Do you remember what you learned about in Chapter 3? You do not reuse the same key and IV to encrypt two different messages! In full-duplex communication, that is exactly what you would be doing. Suppose that AEC-CTR mode is being used for the bulk data transport. If Alice uses a key to encrypt messages she sends to Bob, and Bob uses the same key to encrypt messages to Alice, both data streams could be XORed together to get the XOR of the plaintext messages! As we have seen, that is catastrophic. In fact, if Eve can trick Alice or Bob into encrypting data on her behalf (e.g., by planting "honeypot" data that will certainly be picked up and transmitted), she can XOR that data out leaving the other data as plaintext.

A naive key exchange using RSA encryption could be exploited using this very same principle. Suppose that Alice sends Bob an initial secret K encrypted under Bob's public key. Alice and Bob correctly derive session keys and IVs for full-duplex communication from K. As an example, Bob has a key $K_{B,A}$ that he uses to send encrypted messages to Alice, and Alice has a key $K_{A,B}$ that she uses to send encrypted messages to Bob. (Bob uses $K_{A,B}$ to decrypt Alice's messages and Alice uses $K_{B,A}$ to encrypt Bob's messages.)

But suppose that Eve records all of these transmissions. Then, at a much later date, she replays the initial transmission of K to Bob. Bob doesn't know that it's a replay and he uses K to derive $K_{B,A}$. Now he starts sending data to Eve encrypted under this key.

While it's true that Eve does not have $K_{B,A}$ and cannot decrypt Bob's messages directly, she does have the messages Bob sent to Alice under the same key from the earlier transmissions. Again, assuming that Alice and Bob use AES-CTR, the two transmission streams can be XORed together to potentially extract sensitive information. There are ways to solve this (e.g., by reintroducing challenge-response) but there are many ways it can go wrong even still.

It is very difficult to get all the parts of a cryptographic protocol right, even for the experts. In general, do not design your own protocols. Use existing protocols as much as possible and existing implementations whenever feasible. Above all, we want to remind you one more time, YANAC (You Are Not A Cryptographer… yet!).

EXERCISE 6.11. EXPLOITING FULL-DUPLEX KEY REUSE

In previous exercises, you XORed some data together to see if you could still find patterns, but you didn't actually XOR the two cipher streams together. Imagine if Alice and Bob used your ECDH exchange and derived the same key for full-duplex communication. Use the same key to encrypt some documents together for Alice to send to Bob and for Bob to send to Alice. XOR the cipher streams together and validate that the result is the XOR of the plaintext. See if you can figure out any patterns from the XORed data.

EXERCISE 6.12. DERIVING ALL THE PIECES

Modify the ECDH exchange program to derive six pieces of information: a write encryption key, a write IV, a write MAC key, a read decryption key, a read IV, and a read MAC key. The hard part will be getting both sides to derive the same keys. Remember, the keys will be derived in the same order. So how does Alice determine that the first key derived is her write key and not Bob's write key? One way to do this is to take the first *n* bytes of each side's public key bytes as an integer and whoever has the lowest number goes "first."

An Unfortunate Example of Asymmetric and Symmetric Harmony

Most of our examples of cryptography are beneficial in some way, or are at least not *inherently* evil. Unfortunately, bad guys can use cryptography just as well as the good guys. And given that they can make a lot of money from evil, they can be highly motivated to produce creative, efficient uses of the technology.

One area where bad guys are incredibly good at using cryptography is *ransomware*. If you've been living in a cave in West Antarctica for the past decade and haven't heard about ransomware, it's basically software that encrypts your files and refuses to unlock them until you pay the extortionists behind it.

The cryptography behind early ransomware was simplistic and naive. The ransomware encrypted every file with a different AES key, but all of the AES keys were stored *on the system* in a file. The ransomware's decryptor could easily find the keys and decrypt the file, but so could security researchers. If you don't want somebody to unlock a file, it's bad idea to leave the keys just lying around (under the doormat, so to speak).

Ransomware authors logically turned to asymmetric encryption as a solution. The immediately obvious advantage of asymmetric cryptography is that a public key could be on the victim's system and a private key could be somewhere else. For all of the reasons you've seen in this chapter, the files themselves cannot be encrypted with RSA directly. RSA doesn't even have the capacity to encrypt data larger than between 190 and 256 bytes, and if it did, it would be too slow. The user might notice their system getting locked down long before the encryption was complete.

Instead, the ransomware could encrypt all of the AES keys individually. After all, an AES key is just 16 bytes for AES-128 and 32 bytes for AES-256. Each key can be easily RSA encrypted before being stored on the victim's system. RSA encrypts with the public key, so as long as the private key isn't available to the victim, they won't be able to decrypt the AES keys.

There are two naive variants of this approach, both of which are problematic. The first approach is to generate the key pair ahead of time and hard-code the public key into the malware itself. After the malware encrypts all the AES keys with the public key, a victim has to pay the ransom to have the private key sent to them for decryption. The obvious flaw in this design is that the *same* private key would unlock all of the systems attacked by the ransomware, as each copy of the malicious attack file had the same public key baked into it.

The second approach is for the ransomware to generate the RSA key pair on the victim's system and transmit the private key to the command and control server. Now there is a unique public key encrypting the AES keys, and when the attacker releases the private key for decryption, it only unlocks the specific victim's files. The problem here is that the system has to be *online* to get rid of the private key, and many network monitoring systems will detect transmissions to risky IPs where command and control servers often operate. Transmitting the private key might give away the ransomware before it has even started encrypting files on the system. It is stealthier to do everything locally until the system is fully locked.

Modern ransomware solves all of these problems with a pretty clever approach. First, the attacker generates a long-term asymmetric key pair. For our purposes, let's just assume it is an RSA key pair, and we will call these keys the "permanent" asymmetric keys.

Next the attacker creates some malware and hard-codes the permanent public key into the malware. When the malware activates on a victim's machine, the first thing that it does is generate a *new* asymmetric key pair. Again, for simplicity, let's assume that it is an RSA key pair. We will call it the "local" key pair. It immediately encrypts the newly generated local private key by the attacker's permanent public key embedded in the malware. The unencrypted local private key is *deleted*.

Now the malware begins to encrypt the files on the disk using AES-CTR or AES-CBC. Each file is encrypted under a different key, and then each key is encrypted by the *local* public key. The unencrypted version of the key is destroyed as soon as the file is finished being encrypted.

When the whole process is done, the victim's files are encrypted by AES keys that are themselves encrypted by the local RSA public key. These AES keys could be decrypted by the local RSA private key, but that key is encrypted under the attacker's permanent public key, and the private key is not on the computer.

Now the attacker contacts the victim and demands the ransom. If the victim agrees and pays the ransom (usually by way of Bitcoin), the attacker provides some kind of authentication code to the malware. The malware transmits the encrypted local private key to the attacker. Using his or her permanent private key, he or she decrypts the local private key and sends it back to the victim. Now all of the AES keys can be decrypted and the files subsequently decrypted.

What's clever about this algorithm is that the attacker does not disclose his or her permanent private key. It remains private. A secondary private key is decrypted by the attacker for the victim to use in unlocking the rest of the system.

Warning: Risky Exercise

The upcoming exercise is somewhat risky. You should not do this exercise unless you have a virtual machine that can be restored to a snapshot or a jail (e.g., a chroot jail) with files that can be permanently lost.

Furthermore, this exercise has you create a simplified version of ransomware. We do not condone nor encourage any actual use of ransomware in any form. Don't be stupid, don't be evil.

EXERCISE 6.13. PLAYING THE VILLAIN

Help Alice and Bob create some ransomware to infect WA servers. Start by creating a function that will encrypt a file on disk using an algorithm of your choice (e.g., AES-CTR or AES-CBC). The encrypted data should be saved to a new file with some kind of random name. Before moving on, test encrypting and decrypting the file.

Next, create the fake malware. This malware should be configured with a target directory and the permanent public key. The public key can be hard-coded directly into the code if you wish. Once up and running, it needs to generate a new RSA key pair, encrypt the local private key with the permanent public key, and then delete any unencrypted copies of the local private key. If the private key is too big (e.g., more than 190 bytes), encrypt it in chunks.

Once the local key pair is generated, begin encrypting the files in the target directory. As an extra precaution, you can ask for manual approval before encrypting each file to make sure you don't accidentally encrypt the wrong thing. For each file, encrypt it under a new random name and store a plaintext metadata file with the original name of the file, the encrypted key, and IV. Delete the original file if you feel that you can do so safely (we will **not** be held responsible for any mistakes on your part! Use a VM, only operate in a target directory on copies of unimportant files, and manually confirm each deletion!).

The rest should be straightforward. Your "malware" utility needs to save the encrypted private key to disk. This should be decrypted by a separate command and control utility that has access to the permanent private key. Once decrypted, it should be loaded by the malware and used to decrypt/release the files.

While you are hopefully not a malware/ransomware author, this section should also be helpful to you in thinking about how to encrypt "data at rest." Much of what we talk about in this book is to protect "data in motion," which is data traveling across a network or in some other way between two parties. The ransomware example illustrates protecting data that stays largely in place, typically to be encrypted and decrypted by the same party.

Utilities that encrypt files on disk have to deal with the wretched key management problem just like they do with data in motion. In general, there will have to be one key per file just like there has to be one key per network communications session; this prevents key reuse. The keys (and IVs) must be stored, or it must be possible to regenerate them. If they are stored, they must be encrypted by some kind of master key and stored along with additional metadata about the algorithm used and so forth. This information can be prepended to the beginning of the encrypted file, or it can be stored somewhere in a manifest.

If the keys are regenerated later, this is typically done by deriving the keys from a password, as we have already discussed in Chapter 2. As there needs to be a different key for each file, a random per-file salt is used in the derivation process to ensure that key's uniqueness. The salt must be stored with the file, and the loss of the salt would result in a lost file that could never be decrypted.

This is the basic cryptographic concept behind securing data at rest, but production systems are usually far more complicated. NIST, for example, requires that compliant systems have a defined cryptographic key *life cycle*. This includes a pre-operational, operational, post-operational, and deletion stages as well as an operational "crypto" period for each key. This period is further broken down into an "originator usage period" (OUP) for when sensitive data can be generated and encrypted and a "recipient usage period" (RUP) for when this data can be decrypted and read. Key management systems are expected to handle key rollover (migrating encrypted data from one key to another), key revocation, and many other such functions.

We won't bother you with another reference to YANAC… but by this point in the book, we hope your own subconscious is starting to do it for us!

That's a Wrap

The main thrust of this chapter is that you can wrap up a temporary symmetric communication session within an initial asymmetric session establishment protocol. A lot of the world's asymmetric infrastructure is focused on long-term identification of parties and that infrastructure is useful for establishing identities in some way and based on some model of trust. But once that trust is established, it's more secure and more efficient to create a temporary symmetric key (well, actually several of them) to handle encrypting and MACing the data going forward.

We reviewed, for example, that you can transport a key from one party to another using RSA encryption. This approach was the primary approach used for a long time. Although still present in many systems, it is being retired for many reasons. More favored these days is using an ephemeral key agreement protocol, such as DH and ECDH (actually, DHE and ECDHE to be precise) to create a session key with perfect forward secrecy.

Either way, whether by key transport or key agreement, the parties can then derive the suite of keys necessary for communications. Or, a single party can derive the keys necessary for encrypting data on a hard drive. In both cases, the asymmetric operations are primarily used to establish identity and get an initial key, while the symmetric operations are used for the actual encryption of data.

If you can understand these principles, you can be conversant about most cryptography systems you'll find.