CHAPTER

8

Implementation Issues (I)

Now that we have come this far, we would like to talk a bit about implementation issues. Implementing cryptographic systems is sufficiently different from implementing normal programs to deserve its own treatment.

The big problem is, as always, the weakest-link property (see Section 1.2). It is very easy to screw up the security at the implementation level. In fact, implementation errors such as buffer overflows are one of the biggest security problems in real-world systems. With few exceptions, you don't hear about cryptography systems that are broken in practice. This is not because the cryptography in most systems is any good; we've reviewed enough of them to know this is not the case. It is just easier in most cases to find an implementation-related hole than it is to find a cryptographic vulnerability, and attackers are smart enough not to bother with the cryptography when there is this much easier route.

So far in this book we have restricted our discussion to cryptography, but in this chapter we will focus more on the environment in which the cryptography operates. Every part of the system affects security, and to do a really good job, the entire system must be designed from the ground up not just with security in mind, but with security as one of the primary goals. The ''system'' we're talking about is very big. It includes everything that could damage the security properties if it were to misbehave.

One major part is, as always, the operating system. But historically, none of the operating systems in widespread use was designed with security as a primary goal. And the diversity of operating systems is enormous—from the operating systems we interact with on our desktop computers to operating systems on embedded devices and phones. The logical conclusion to draw

from this is that it is impossible to implement a secure system. We don't know how to do it, and we don't know anyone else who knows how to do it, either.

Real-life systems include many components that were never designed for security, and that makes it impossible to achieve the level of security that we really need. So should we just give up? Of course not. When we design a cryptographic system, we do our very best to make sure that at least our part is secure. This might sound like an odd mentality: all we care about is our little domain. But we *do* care about the other parts of the system; we just can't do anything about them in the context of this book. That is one of the reasons for writing this book: to get other people to understand the insidious nature of security, and how important it is to do it right.

Another important reason to get at least the cryptography right is one we mentioned before: attacks on the cryptography are especially damaging because they can be invisible. If the attacker succeeds in breaking your cryptography, you are unlikely to notice. This can be compared to a burglar who has a set of keys to your house. If the burglar exercises reasonable caution, how would you ever find out?

Our long-term goal is to make secure computer systems. To achieve that goal, everybody will have to do their part. This book is about making the cryptography secure. Other parts of the system will have to be made secure, too. The overall security of the system is going to be limited by the weakest link, and we will do our utmost to ensure that the weakest link will never be the cryptography.

Another important reason to do the cryptography right is that it is very difficult to switch cryptographic systems once they've been implemented. An operating system runs on a single computer. Cryptographic systems are often used in communication protocols to let many computers communicate with each other. Upgrading the operating system of a single computer is feasible, and in practice it is done relatively often. Modifying the communication protocols in a network is a nightmare, and as a result many networks still use the designs of the 1970s and 1980s. We must keep in mind that any new cryptographic system we design today, if adopted widely, is quite likely to still be in use 30 or 50 years from now. We hope that by that time the other parts of the system will have achieved a much higher level of security, and we certainly don't want cryptography to be the weakest link.

8.1 Creating Correct Programs

The core of the implementation problem is that we in the IT industry don't know how to write a correct program or module. (A ''correct'' program is one that behaves exactly according to its specifications.) There are several reasons for the difficulty we seem to have in writing correct programs.

8.1.1 Specifications

The first problem is that for most programs, there is no clear description of what they are supposed to do. If there are no specifications, then you cannot even check whether a program is correct or not. For such programs, the whole concept of correctness is undefined.

Many software projects have a document called the functional specification. In theory, this should be the specification of the program. But in practice, this document often does not exist, is incomplete, or specifies things that are irrelevant for the behavior of the program. Without clear specifications, there is no hope of getting a correct program.

There are really three stages in the specification process:

Requirements Requirements are an informal description of what the program is supposed to achieve. It is really a ''*what* can I do with it'' document, rather than a ''how exactly do I do something with it'' document. Requirements are often a bit vague and leave details out in order to concentrate on the larger picture.

Functional specification The functional specification gives a detailed and exhaustive definition of the behavior of the program. The functional specification can only specify things that you can measure on the outside of the program.

For each item in the functional specification, ask yourself whether you could create a test on the finished program that would determine whether that item was adhered to or not. The test can only use the external behavior of the program, not anything from the inside. If you can't create a test for an item, it does not belong in the functional specification.

The functional specification should be complete. That is, every piece of functionality should be specified. Anything not in the functional specification does not have to be implemented.

Another way to think of the functional specification is as the basis for testing the finished program. Any item can, and should, be tested.

Implementation design This document has many names, but it specifies how the program works internally. It contains all of the things that cannot be tested from the outside. A good implementation design will often split the program into several modules, and describe the functionality of each. In turn, these module descriptions can be seen as the requirements for the module, and the whole cycle starts all over again, this time by splitting the module itself into multiple sub-modules.

Of these three documents, the functional specification is without a doubt the most important one. This is the document against which the program

will be tested when it is finished. You can sometimes get by with informal requirements, or an implementation design that is nothing but a few sketches on a whiteboard. But without functional specifications, there is no way to even describe what you have achieved in the end when the program is finished.

8.1.2 Test and Fix

The second problem in writing correct programs is the test-and-fix development method that is in almost universal use. Programmers write a program, and then test whether it behaves correctly. If it doesn't, they fix the bugs and test again. As we all know, this does not lead to a correct program. It results in a program that kind of works in the most common situations.

Back in 1972, Edsger Dijkstra commented in his Turing Award lecture that testing can only show the presence of bugs, never the absence of bugs [35]. This is very true, and ideally we would like to write programs that we can demonstrate to be correct. Unfortunately, current techniques in proving the correctness of programs are not good enough to handle day-to-day programming tasks, let alone a whole project.

Computer scientists do not know how to solve this. Maybe it will be possible in the future to prove that a program is correct. Maybe we just need a far more extensive and thorough testing infrastructure and methodology. But even without having a full solution, we can certainly do our very best with the tools we do have.

There are some simple rules about bugs that any good software engineering book includes:

- If you find a bug, first implement a test that detects the bug. Check that the bug is detected. Then fix the bug, and check that the test no longer finds the bug. And then keep running that test on every future version to make sure the bug does not reappear.
- Whenever you find a bug, think about what caused it. Are there any other places in the program where a similar bug might reside? Go check them all.
- Keep track of every bug you find. Simple statistical analysis of the bugs you have found can show you which part of the program is especially buggy, or what type of error is made most frequently, etc. Such feedback is necessary for a quality control system.

This is not even a bare minimum, but there is not a lot of methodology to draw from. There are quite a few books that discuss software quality. They don't all agree with each other. Many of them present a particular software development methodology as *the* solution, and we are always suspicious of such one-cure-does-it-all schemes. The truth is almost always somewhere in the middle.

8.1.3 Lax Attitude

The third problem is the incredibly lax attitude of many in the computer industry. Errors in programs are frequently accepted as a matter of course. If your word processor crashes and destroys a day's worth of work, everybody seems to think this is quite normal and acceptable. Often they blame the user: ''You should have saved your work more often.'' Software companies routinely ship products with known bugs in them. This wouldn't be so bad if they only sold computer games, but nowadays our work, our economy, and—more and more—our lives depend on software. If a car manufacturer finds a defect (bug) in a car after it was sold, they will recall the car and fix it. Software companies get away with disclaiming any and all liability in their software license, something they wouldn't be allowed to do if they produced any other product. This lax attitude means there are still not enough serious efforts being made at producing correct software.

8.1.4 So How Do We Proceed?

Don't ever think that all you need is a good programmer or code reviews or an ISO 9001–certified development process or extensive testing or even a combination of all of them. Reality is much more difficult. Software is too complex to be tamed by a few rules and procedures. We find it instructive to look at the best engineering quality control system in the world: the airline industry. Everybody in that industry is involved in the safety system. There are very strict rules and procedures for almost every operation. There are multiple backups in case of failures. Every nut and bolt of the airplane has to be flight-qualified before it can ever be used. Anytime a mechanic takes a screwdriver to the plane, his work is checked and signed off by a supervisor. Every modification is carefully recorded. Any accident is meticulously investigated to find all the underlying causes, which are then fixed. This fanatical pursuit of quality has a very high cost. An airplane is probably an order of magnitude more expensive than it would be if you just sent the drawings to an ordinary engineering firm. But the pursuit of quality has also been amazingly effective. Flying is an entirely routine operation today, in a machine where every failure is potentially fatal—a machine where you cannot just hit the brakes and stop when something goes wrong. One where the only safe way back to the ground is the quite delicate operation of landing on one of the rare specially prepared spots in the world. The airline industry has been amazingly effective at making flying secure. We would do well to learn all we can from them. Maybe writing correct software *would* cost an order of magnitude more than what we are used to now. But given the cost to society of the bugs in software that we see today, we are sure it would be cost-effective in the long run.

8.2 Creating Secure Software

So far, we have only talked about correct software. Just writing correct software is not good enough for a security system. The software must be secure as well.

What is the difference? Correct software has a specified functionality. If you hit button *A*, then *B* will happen. Secure software has an additional requirement: a *lack* of functionality. No matter what the attacker does, she cannot do *X*. This is a very fundamental difference; you can test for functionality, but not for lack of functionality. The security aspects of the software cannot be tested in any effective way, which makes writing secure software much more difficult than writing correct software. The inevitable conclusion is:

Standard implementation techniques
are entirely inadequate to create secure code.

We actually don't know how to create secure code. Software quality is a vast area that would take several books to cover. We don't know enough about it to write those books, but we *do* know the cryptography-specific issues and the problems that we see most frequently, and that is what we will discuss in the rest of this chapter.

Before we start, let us make our point of view clear: unless you are willing to put real effort into developing a secure implementation, there is little point in bothering with the cryptography. Designing cryptographic systems might be fun, but cryptography is generally only a small part of a larger system.

8.3 Keeping Secrets

Anytime you work with cryptography, you are dealing with secrets. And secrets have to be kept. This means that the software that deals with the secrets has to ensure that they don't leak out.

For the secure channel we have two types of secrets: the keys and the data. Both of these secrets are transient secrets; we don't have to store them for a long time. The data is only stored while we process each message. The keys are only stored for the duration of the secure channel. Here we will only discuss keeping transient secrets. For a discussion on storing secrets long-term, see Chapter 21.

Transient secrets are kept in memory. Unfortunately, the memory on most computers is not very secure. We will discuss each of the typical problems in turn.

8.3.1 Wiping State

A basic rule of writing security software: wipe any information as soon as you no longer need it. The longer you keep it, the higher the chance that someone will be able to access it. What's more, you should definitely wipe the data before you lose control over the underlying storage medium. For transient secrets, this involves wiping the memory locations.

This sounds easy to do, but it leads to a surprising number of problems. If you write the entire program in C, you can take care of the wiping yourself. If you write a library for others to use, you have to depend on the main program to inform you that the state is no longer needed. For example, when the communication connection is closed, the crypto library should be informed so that it can wipe the secure channel session state. The library can contain a function for this, but there's a reasonable chance that the programmer of the application won't call this function. After all, the program works perfectly well without calling this function.

In some object-oriented languages, things are a bit easier. In C++, there is a destructor function for each object, and the destructor can wipe the state. This is certainly standard practice for security-relevant code in C++. As long as the main program behaves properly and destroys all objects it no longer needs, the memory state will be wiped. The C++ language ensures that all stack-allocated objects are properly destroyed when the stack is unwound during exception handling, but the program has to ensure that all heap-allocated objects are destroyed. Calling an operating system function to exit the program might not even unwind the call stack. And you have to ensure that all sensitive data is wiped even if the program is about to exit. After all, the operating system gives no guarantees that it will wipe the data soon, and some operating systems don't even bother wiping the memory before they give it to the next application.

Even if you do all this, the computer might still frustrate your attempts. Some compilers try too hard to optimize. A typical security-relevant function performs some computations in local variables, and then tries to wipe them. You can do this in C with a call to the `memset` function. Good compilers will optimize the `memset` function to in-line code, which is more efficient. But some of them are too clever by half. They detect that the variable or array that is being wiped will never be used again, and ''optimize'' the `memset` away. It's faster, but suddenly the program does not behave the same way anymore. It is not uncommon to see code that reveals data that it happens to find in memory. If the memory is given to some library without having been wiped first, the library might leak the data to an attacker. So check the code that your compiler produces, and make sure the secrets are actually being wiped.

In a language like Java, the situation is even more complicated. All objects live on the heap, and the heap is garbage-collected. This means that the finalization function (similar to the C++ destructor) is not called until the garbage collector figures out that the object is no longer in use. There are no specifications about how often the garbage collector is run, and it is quite conceivable that secret data remains in memory for a very long time. The use of exception handling makes it hard to do the wiping by hand. If an exception is thrown, then the call-stack unwinds without any way for the programmer to insert his own code, except by writing *every* function as a big try clause. The latter solution is so ugly that it is impractical. It also has to be applied throughout the program, making it impossible to create a security library for Java that behaves properly. During exception handling, Java happily unwinds the stack, throwing away the references to the objects without cleaning up the objects themselves. Java is really bad in this respect. The best solution we've been able to come up with is to at least ensure that the finalization routines are run at program exit. The `main` method of the program uses a `try-finally` statement. The `finally` block contains some code to force a garbage collect, and to instruct the garbage collector to attempt to complete all the finalization methods. (See the functions `System.gc()` and `System.runFinalization()` for more details.) There is still no guarantee that the finalization methods will be run, but it is the best we've been able to find.

What we really need is support from the programming language itself. In C++ it is at least theoretically possible to write a program that wipes all states as soon as they are no longer needed, but many other features of the language make it a poor choice for security software. Java makes it very difficult to wipe the state. One improvement would be to declare variables as ''sensitive,'' and have the implementation guarantee that they will be wiped. Even better would be a language that always wipes all data that is no longer needed. That would avoid a lot of errors without significantly affecting efficiency.

There are other places where secret data can end up. All data is eventually loaded into a CPU register. Wiping registers is not possible in most programming languages, but on register-starved CPUs like the x86, it is very unlikely that any data will survive for any reasonable amount of time.

During a context-switch (when the operating system switches from running one program to running the next program), the values in the registers of the CPU are stored in memory where their values might linger for a long time. As far as we know, there is nothing you can do about this, apart from fixing the operating system to ensure the confidentiality of that data.

8.3.2 Swap File

Most operating systems (including all current Windows versions and all UNIX versions) use a virtual memory system to increase the number of programs that can be run in parallel. While a program is running, not all of its data is kept

in memory. Some is stored in a swap file. When the program tries to access data that is not in memory, the program is interrupted. The virtual memory system reads the required data from the swap file into a piece of memory, and the program is allowed to continue. What's more, when the virtual memory system decides that it needs more free memory, it will take an arbitrary piece of memory from a program and write it to the swap file.

Of course, not all virtual memory systems or configurations keep the data secret, or encrypt it before it is written to the disk. Most software is designed for a cooperative environment, not the adversarial environment that cryptographers work in. So our problem is the following: the virtual memory system could just take some of the memory of our program and write it to the swap file on disk. The program never gets told, and does not notice. Suppose this happens to the memory in which the keys are stored. If the computer crashes—or is switched off—the data remains on the disk. Most operating systems leave the data on disk even when you shut them down properly. There may be no mechanism to wipe the swap file, so the data could linger indefinitely on disk. Who knows who will have access to this swap file in future? We really cannot afford the risk of having our secrets written to the swap file.[1]

So how do we stop the virtual memory system from writing our data to disk? On some operating systems there are system calls that you can use to inform the virtual memory system that specified parts of memory are not to be swapped out. Some operating systems support a secure swap system where the swapped-out data is cryptographically protected, but these systems might require the user to toggle the appropriate system configuration flags. If neither of these options is available on all the systems that you wish to run your application, there might not be much you can do to protect against this particular avenue of attack.

Assuming you can lock the memory and prevent it from being swapped out, which memory should be locked? All the memory that can ever hold secrets, of course. This brings up a secondary problem. Many programming environments make it very hard to know where exactly your data is being stored. Objects are often allocated on a heap, data can be statically allocated, and many local variables end up on the stack. Figuring out the details is complicated and very error-prone. Probably the best solution is to simply lock all the memory of your application. Even that is not quite as easy as it sounds, because you could lose a number of operating system services such as the automatically allocated stack. And locking all the memory makes the virtual memory system ineffective.

It shouldn't be this difficult. The proper solution is, of course, to make a virtual memory system that protects the confidentiality of the data. This is an

[1]In fact, we should never write secrets to any permanent media without encrypting them, but that is an issue we will discuss later.

operating system change, and beyond our control. Even if the next version of your operating system were to have this feature, you should carefully check that the virtual memory system does a good job of keeping secrets. And, depending on your application, you may still have to deal with the fact that your application needs to run on older systems or systems in insecure configurations.

8.3.3 Caches

Modern computers don't just have a single type of memory. They have a hierarchy of memories. At the bottom is the main memory—often gigabytes in size. But because the main memory is relatively slow, there is also a cache. This is a smaller but faster memory. The cache keeps a copy of the most recently used data from the main memory. If the CPU wants to access the data, it first checks the cache. If the data is in the cache, the CPU gets the data relatively quickly. If the data is not in the cache, it is read (relatively slowly) from main memory, and a copy is stored in the cache for future use. To make room in the cache, a copy of some other piece of data is thrown away.

This is important because caches keep copies of data, including copies of our secret data. The problem is that when we try to wipe our secrets, this wiping might not take place properly. In some systems, the modifications are only written to the cache and not to the main memory. The data will eventually be written to main memory, but only when the cache needs more room to store other data. We don't know all the details of these systems, and they change with every CPU. There is no way to know if there is some interaction between the memory allocation unit and the cache system that might result in some wipe operations escaping the write-to-main-memory part when the memory is deallocated before the cache is flushed. Manufacturers never specify how to wipe data in a guaranteed manner. At least, we have never seen any specifications like that, and as long as it is not specified, we can't trust it.

A secondary danger of caches is that under some circumstances a cache learns that a particular memory location has been modified, perhaps by the other CPU in a multi-CPU system. The cache then marks the data it has for that location as ''invalid,'' but typically the actual data is not wiped. Again, there might exist a copy of our secrets that has not been wiped.

There is very little you can do about this. It is not a great danger, because in most systems, minus physical attacks, only the OS code can access the cache mechanisms directly. And we have to trust the operating system anyway, so we could trust it with this as well. We are nevertheless concerned about these designs, because they clearly do not provide the functionality that is required to implement security systems properly.

8.3.4 Data Retention by Memory

Something that surprises many people is that simply overwriting data in memory does not delete the data. The details depend to some extent on the exact type of memory involved, but basically, if you store data in a memory location, that location slowly starts to ''learn'' the data. When you overwrite or switch off the computer, the old value is not completely lost. Depending on the circumstances, just powering the memory off and back on again can recover some or all of the old data. Other memories can ''remember'' old data if you access them using (often undocumented) test modes [57].

Several mechanisms cause this phenomenon. If the same data is stored for a time in the same location in SRAM (Static RAM), then this data becomes the preferred power-up state of that memory. A friend of ours encountered this problem with his home-built computer long ago [17]. He wrote a BIOS that used a magic value in a particular memory location to determine whether a reset was a cold reboot or a warm reboot.[2] After a while the machine refused to boot after power-up because the memory had learned the magic value, and the boot process therefore treated every reset as a warm reboot. As this did not initialize the proper variables, the boot process failed. The solution in his case was to swap some memory chips around, scrambling the magic value that the SRAM had learned. For us, it was a lesson to remember: memory retains more data than you think.

Similar processes happen in DRAM (Dynamic RAM), although they are somewhat more complicated. DRAM works by storing a small charge on a very small capacitor. The insulating material around the capacitor is stressed by the resulting field. The stress results in changes to the material, specifically causing the migration of impurities [57]. An attacker with physical control over the memory can potentially recover this data. Additionally, because of how DRAM capacitors discharge, their values may remain for seconds at room temperature if power is removed or even longer if the memory is cooled.

These are important problems. The latter class of issues was recently demonstrated in the context of cold boot attacks [59]. These researchers were able to recover secret cryptographic keys from the memories of computers after they were rebooted. These researchers were also able to physically extract the memory from one computer, put that memory in another computer, and recover the cryptographic keys. If your computer is ever compromised (e.g., stolen), you do not want the data that you had in memory to be

[2]In those days home-built machines were programmed by entering the binary form of machine language directly. This led to many errors, and the one sure way to recover from a program that crashed was to reset the machine. A cold reboot is one after power-up. A warm reboot is the sort performed when the user presses the reset button. A warm reboot does not reinitialize all the state, and therefore does not wipe the settings the user made.

compromised as well. To achieve this goal, we have to make the computer forget information.

We can only give a partial solution, which works if we make some reasonable assumptions about the memory. This solution, which we call a Boojum,[3] works for relatively small amounts of data, such as keys. Our description of Boojum has been updated slightly since the first edition of this book, and includes a defense against the cold boot attack from [59]. For Boojum, let m be the data we want to store. Instead of storing m, we generate a random string R and store both R and $h(R) \oplus m$ where h is a hash function. These two values are stored in different memory locations, preferably not too close together. One trick is to change R regularly. At regular intervals, say every 1 second, we generate a new random R', and update the memory to store $R \oplus R'$ and $h(R \oplus R') \oplus m$. This ensures that each bit of the memory is written with a sequence of random bits. To wipe the memory, you simply write a new m with the value zero.

To read information from this storage, you read both parts, hash the first, and XOR them together to get m. Writing is done by XORing the new data with $h(R)$ and storing it in the second location.

Care should be taken that the bits of R and $h(R) \oplus m$ are not adjacent on the RAM chip. Without information about how the RAM chip works, this can be difficult, but most memories store bits in a rectangular matrix of bits, with some address bits selecting the row and other address bits selecting the column. If the two pieces are stored at addresses that differ by `0x5555`, it is highly unlikely that the two will be stored adjacent on the chip. (This assumes that the memory does not use the even-indexed address bits as row number and the odd-indexed address bits as column number, but we have never seen a design like that.) An even better solution might be to choose two random addresses in a very large address space. This makes the probability that the two locations are adjacent very small, independent of the actual chip layouts of the memory.

This is only a partial solution, and a rather cumbersome one at that. It is limited to small amounts of data. But using this solution ensures that there is no physical point on the memory chip that is continually stressed or unstressed depending on the secret data. Further, as long as k bits of R are not recoverable, the attacker would have to exhaustively search for those k bits before being able to recover $h(R) \oplus m$.

There is still no guarantee that the memory will be wiped. If you read the documentation of a memory chip, there are no specifications that prevent the chip from retaining all data ever stored in it. No chip does that, of course, but it shows that we can at most achieve a heuristic security.

We have concentrated on the main memory here. The same solution will work for the cache memory, except that you cannot control the position on the chip where the data will be stored. This solution does not work for the

[3] After Lewis Carroll's *The Hunting of the Snark* [24].

CPU registers, but they are used so often for so much different data that we doubt they will pose a data retention problem. On the other hand, extension registers, such as floating point registers or MMX-style registers, are used far less frequently, so they could pose a problem.

If you have large amounts of data that need to be kept secret, then the solution of storing two copies and XORing new random strings into both copies regularly becomes too expensive. A better solution is to encrypt a large block of data and store the ciphertext in memory that potentially retains information. Only the key needs to be stored in a way that avoids data retention, for example, using a Boojum. For details, see [32].

8.3.5 Access by Others

There's yet another problem with keeping secrets on a computer: other programs on the same machine might access the data. Some operating systems allow different programs to share memory. If the other program can read your secret keys, you have a serious problem. Often the shared memory has to be set up by both programs, which reduces the risk. In other situations, the shared memory might be set up automatically as a result of loading a shared library.

Debuggers are especially dangerous. Modern operating systems often contain features designed to be used by debuggers. Various Windows versions allow you to attach a debugger to an already running process. The debugger can do many things, including reading the memory. Under UNIX, it is sometimes possible to force a core-dump of a program. The core-dump is a file that contains a memory image of the program data, including all of your secrets.

Another danger comes from especially powerful users. Called *superusers*, or *administrators*, these users can access things on the machine that normal users cannot. Under UNIX, for example, the superuser can read any part of the memory.

In general, your program cannot effectively defend itself against these types of attacks. If you are careful, you may be able to eliminate some of these problems, but often you'll find yourself limited in what can be achieved. Still, you should consider these issues on the particular platform you are working on.

8.3.6 Data Integrity

In addition to keeping secrets, we should protect the integrity of the data we are storing. We use the MAC to protect the integrity of the data during transit, but if the data can be modified in memory, we still have problems.

In this discussion, we will assume that the hardware is reliable. If the hardware is unreliable, there is very little you can do. If you are unsure about the hardware reliability, perhaps you should spend part of your time and memory simply to verify it, although that is really the operating system's job.

One thing we try to do is make sure the main memory on our machines is ECC (error-correcting code) memory.[4] If there is a single bit failure, then the error-correcting code will detect and correct the error. Without ECC memory, any bit error leads to the CPU reading the wrong data.

Why is this important? There is an enormous number of bits in a modern computer. Suppose the engineering is done really well, and each bit has only a 10^{-15} chance of failing in each second. If you have 128 MB of memory, then you have about 10^9 bits of memory, and you can expect one bit failure every 11 days. The error rate increases with the amount of memory in the machine, so it is even worse if you have 1 GB of memory, with one failure every 32 hours. Servers typically use ECC memory because they have more memory and run for longer periods of time. We like to have the same stability in all machines.

Of course, this is a hardware issue, and you typically don't get to specify the type of memory on the machine that will run the final application.

Some of the dangers that threaten data confidentiality also endanger the data integrity. Debuggers can sometimes modify your program's memory. Superusers can directly modify memory, too. Again, there is nothing you can do about it, but it is useful to be aware of the situation.

8.3.7 What to Do

Keeping a secret on a modern computer is not as easy as it sounds. There are many ways in which the secret can leak out. To be fully effective, you have to stop all of them. Unfortunately, current operating systems and programming languages do not provide the required support to stop the leakage completely. You have to do the best you can. This involves a lot of work, all of it specific to the environment you work in.

These problems also make it very difficult to create a library with the cryptographic functions in it. Keeping the secrets safe often involves modifications to the main program. And of course, the main program also handles data that should be kept confidential; otherwise, it wouldn't need the cryptography library in the first place. This is the familiar issue of security considerations affecting every part of the system.

8.4 Quality of Code

If you create an implementation for a cryptographic system, you will have to spend a great deal of time on the quality of the code. This book is not about programming, but we will say a few words about code quality here.

[4]You have to make sure that all components of the computer support ECC memory. Beware of slightly cheaper memory modules that do not store the extra information but instead recompute it on the fly. This defeats the whole purpose of ECC memory.

8.4.1 Simplicity

Complexity is the main enemy of security. Therefore, any security design should strive for simplicity. We are quite ruthless about this, even though this does not make us popular. Eliminate all the options that you can. Get rid of all those baroque features that few people use. Stay away from committee designs, because the committee process always leads to extra features or options in order to achieve compromise. In security, simplicity is king.

A typical example is our secure channel. It has no options. It doesn't allow you to encrypt the data without authenticating it, or to authenticate the data without encrypting it. People always ask for these features, but in many cases they do not realize the consequences of using partial security features. Most users are not informed enough about security to be able to select the correct security options. The best solution is to have no options and make the system secure by default. If you absolutely have to, provide a single option: secure or insecure.

Many systems also have multiple cipher suites, where the user (or someone else) can choose which cipher and which authentication function to use. If at all possible, eliminate this complexity. Choose a single mode that is secure enough for all possible applications. The computational difference between the various encryption modes is not that large, and cryptography is rarely the bottleneck for modern computers. Apart from getting rid of the complexity, it also gets rid of the danger that users might configure their application to use weak cipher suites. After all, if choosing an encryption and authentication mode is so difficult that the designer can't do it, it will be even more challenging for a user to make an informed decision.

8.4.2 Modularization

Even after you have eliminated a lot of options and features, the resulting system will still be quite complex. There is one main technique for making the complexity manageable: modularization. You divide the system into separate modules, and design, analyze, and implement each module separately.

You should already be familiar with modularization; in cryptography it becomes even more important to do it right. Earlier we talked about cryptographic primitives as modules. The module interface should be simple and straightforward. It should behave according to the reasonable expectations of a user of the module. Look closely at the interface of your modules. Often there are features or options that exist to solve some other module's problems. If possible, rip them out. Each module should solve its own problems. We have found that when module interfaces start to develop weird features, it is time to redesign the software because they are almost always a result of design deficiencies.

Modularization is so important because it is the only efficient way we have of dealing with complexity. If a particular option is restricted to a single module, it can be analyzed within the context of this module. However, if the option changes the external behavior of one module, it can affect other modules as well. If you have 20 modules, each with a single binary option that changes the module behavior, there are over a million possible configurations. You would have to analyze each of these configurations for security—an impossible task.

We have found that many options are created in the quest for efficiency. This is a well-known problem in software engineering. Many systems contain so-called optimizations that are useless, counterproductive, or insignificant because they do not optimize those parts of the system that form the bottleneck. We have become quite conservative about optimizations. Usually we don't bother with them. We create a careful design, and try to ensure that work can be done in large ''chunks.'' A typical example is the old IBM PC BIOS. The routine to print a character on the screen took a single character as an argument. This routine spent almost all of its time on overhead, and only a very small fraction on actually putting the character on the screen. If the interface of the routine had allowed a string as argument, then the entire string could have been printed in only slightly more time than it took to print a single character. The result of this bad design was that all DOS machines had a terribly slow display. This same principle applies to cryptographic designs. Make sure that work can be done in large enough chunks. Then only optimize those parts of your program that you can *measure* as having a significant effect on the performance.

8.4.3 Assertions

Assertions are a good tool to help improve the quality of your code.

When implementing cryptographic code, adopt an attitude of professional paranoia. Each module distrusts the other modules, and always checks parameter validity, enforces calling sequence restrictions, and refuses unsafe operations. Most of the times these are straightforward assertions. If the module specifications state that you have to initialize the object before you use it, then using an object before initialization will result in an assertion error. Assertion failures should always lead to an abort of the program with ample documentation of which assertion failed, and for what reason.

The general rule is: any time you can make a meaningful check on the internal consistency of the system, you should add an assertion. Catch as many errors as you can, both your own and those of other programmers. An error caught by an assertion will not lead to a security breach.

There are some programmers who implement assertion checking in development, but switch it off when they ship the product. This is not the security perspective. What would you think of a nuclear power station where the operators train with all the safety systems in place, but switch them off when

they go to work on the real reactor? Or a parachutist who wears his emergency parachute while training on the ground, but leaves it off when he jumps out of the airplane? Why would anyone ever switch off the assertion checking on production code? That is the only place where you really need it! If an assertion fails in production code, then you have just encountered a programming error. Ignoring the error will most likely result in some kind of wrong answer, because at least one assumption the code makes is wrong. Generating wrong answers is probably the worst thing a program can do. It is much better to at least inform the user that a programming error has occurred, so he does not trust the erroneous results of the program. Our recommendation is to leave all your error checking on.

8.4.4 Buffer Overflows

Buffer overflow problems have been known for decades. Perfectly good solutions to avoid them have been available for the same amount of time. Some of the earliest higher-level programming languages, such as Algol 60, completely solved the problem by introducing mandatory array bounds checking. Even so, buffer overflows cause a huge number of security problems on the Internet. There also exist a larger number of software attacks beyond buffer overflows, such as format string attacks and integer overflow attacks.

But these are things we cannot change. We can give you advice on how to write good cryptographic code. Avoid any programming language that allows buffer overflows. Specifically: don't use C or C++. And don't ever switch off the array bounds checking of whichever language you use instead. It is such a simple rule, and will probably solve half of all your security bugs.

8.4.5 Testing

Extensive testing is always part of any good development process. Testing can help find bugs in programs, but it is useless to find security holes. Never confuse testing with security analysis. The two are complementary, but different.

There are two types of tests that should be implemented. The first is a generic set of tests developed from the module's functional specifications. Ideally, one programmer implements the module and a second programmer implements the tests. Both work from the functional specification. Any misunderstanding between the two is a clear indication that the specifications have to be clarified. The generic tests should attempt to cover the entire operational spectrum of the module. For some modules, this is simple; for others, the test program will have to simulate an entire environment. In much of our own code, the test code is about as big as the operational code, and we have not found a way of significantly improving that.

A second set of tests is developed by the programmer of the module itself. These are designed to test any implementation limits. For example, if a module

uses a 4 KB buffer internally, then extra tests of the boundary conditions at the start and end of the buffer will help to catch any buffer-management errors. Sometimes it requires knowledge of the internals of a module to devise specific tests.

We frequently write test sequences that are driven by a random generator. We will discuss pseudorandom number generators (PRNGs) extensively in Chapter 9. Using a PRNG makes it very easy to run a very large number of tests. If we save the seed we used for the PRNG, we can repeat the same test sequence, which is very useful for testing and debugging. Details depend on the module in question.

Finally, we have found it useful to have some ''quick test'' code that can run every time the program starts up. In one of Niels's projects, he had to implement AES. The initialization code runs AES on a few test cases and checks the output against the known correct answers. If the AES code is ever destabilized during the further development of the application, this quick test is very likely to detect the problem.

8.5 Side-Channel Attacks

There is a whole class of attacks that we call side-channel attacks [72]. These are possible when an attacker has an additional channel of information about the system. For example, an attacker could make detailed measurements of the time it takes to encrypt a message. Depending on how the system is implemented, this timing information could allow an attacker to infer private information about the message itself or the underlying encryption key. If the cryptography is embedded in a smart card, then the attacker can measure how much current the card draws over time. Magnetic fields, RF emissions, power consumption, timing, and interference on other data channels can all be used for side-channel attacks.

Not surprisingly, side-channel attacks can be remarkably successful against systems that are not designed with these attacks in mind. Power analysis of smart cards is extremely successful [77].

It is very difficult, if not impossible, to protect against all forms of side-channel attacks, but there are some simple precautions you can take. Years ago, when Niels worked on implementing cryptographic systems in smart cards, one of the design rules was that the sequence of instructions that the CPU executed could only depend on information already available to the attacker. This stops timing attacks, and makes power analysis attacks more complicated because the sequence of instructions being executed can no longer leak any information. It is not a full solution, and modern power analysis techniques would have no problem breaking the smart cards that

were fielded in those days. Still, that fix was about the best that could be done with the smart cards of the day. Resistance against side-channel attacks will always come from a combination of countermeasures—some of them in the software that implements the cryptographic system, and some of them in the actual hardware.

Preventing side-channel attacks is an arms race. You try to protect yourself against the known side channels, and then a smart person somewhere discovers a new side channel, so then you have to go back and take that one into account as well. In real life, the situation is not that bad, because most side-channel attacks are difficult to perform. Side channels are a real danger to smart cards because the card is under full control of the adversary, but only a few types of side channels are practical against most other computers. In practice, the most important side channels are timing and RF emissions. (Smart cards are particularly vulnerable to measuring the power consumption.)

8.6 Beyond this Chapter

We hope this chapter has made it clear that security does not start or stop with the cryptographic design. All aspects of the system have to do their part to achieve security.

Implementing cryptographic systems is an art in itself. The most important aspect is the quality of the code. Low-quality code is the most common cause of real-world attacks, and it is rather easy to avoid. In our experience, writing high-quality code takes about as long as writing low-quality code, if you count the time from start to finished product, rather than from start to first buggy version. Be fanatical about the quality of your code. It can be done, and it needs to be done, so go do it!

There are a number of great books for further reading. Among these are *Software Security: Building Security In* by McGraw [88], *The Security Development Lifecycle* by Howard and Lipner [62], and *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities* by Dowd, McDonald, and Schuh [37].

8.7 Exercises

Exercise 8.1 Describe how each of the issues in Section 8.3 applies to your personal computer's hardware and software configuration.

Exercise 8.2 Find a new product or system that manipulates transient secrets. This might be the same product or system you analyzed for Exercise 1.8. Conduct a security review of that product or system as described in Section 1.12,

this time focusing on issues surrounding how the system might store these secrets (Section 8.3).

Exercise 8.3 Find a new product or system that manipulates secret data. This might be the same product or system you analyzed for Exercise 1.8. Conduct a security review of that product or system as described in Section 1.12, this time focusing on issues surrounding code quality (Section 8.4).

Exercise 8.4 Monitor the bugtraq mailing list for one week. Create a table listing all the different types of vulnerabilities that are announced or fixed during that week, as well as the number of such vulnerabilities for each type. What sorts of larger inferences can you draw from this table? See `http://www.schneier.com/ce.html` for additional information about the bugtraq mailing list.