Extracted from:

Practical Security

Simple Practices for Defending Your Systems

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Practical Security

Simple Practices for Defending Your Systems

Roman Zabicki
edited by Adaobi Obi Tulton

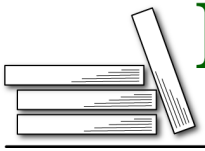
Practical Security

Simple Practices for Defending Your Systems

Roman Zabicki

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Adaobi Obi Tulton
Copy Editor: Molly McBeath
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-634-1
Book version: P1.0—February 2019

To Marnie

Thanks for all the geek time

Don't Roll Your Own Crypto

Writing cryptography software isn't like writing regular software. When writing regular software, little bugs tend to have little impacts. If you have an off-by-one bug, you could expect a small bug, for example, omitting one result on a search page. If you forget to check for null references, maybe a program crashes. But with cryptography, a small mistake may leave you with a system that encrypts and decrypts correctly for well-intentioned inputs but fails entirely when faced with malicious input. The developer needs to either rediscover the entire field from scratch or subject the code to the scrutiny of others with a deep understanding of the field. Just as you can't tickle yourself, you can't find the mistakes you've made that involve flaws you haven't learned about yet. Bruce Schneier has a nice essay on Schneier's Law that expands on this.[9]

There are many different attack models to consider. A common, though misguided, mental model of a secure crypto system is one where the attacker gets to see a single message of modest length. If the attacker can decrypt it, then the system is insecure; otherwise it's suitable for any and all purposes. That certainly is an attack that a crypto system should be able to defend against. But there are many other models to consider, such as the following:

- An attacker who can listen to many encrypted messages between two parties
- An attacker who can replay previously transmitted encrypted messages
- An attacker who can replay modified variants of previously transmitted encrypted messages
- An attacker who can replay modified variants of previously transmitted encrypted messages to a recipient who's expecting only well-behaved communication and who therefore displays helpful error messages if anything goes wrong during decryption
- An attacker who can listen to the encrypted communication between two parties where some of the plaintexts are known to the attacker
- An attacker who can influence the contents of encrypted communication between two other parties

Defenses that protect against more limited adversaries may fail against more advanced adversaries.

9. https://www.schneier.com/blog/archives/2011/04/schneiers_law.html

It's easy to see all of the security breaches in the news and decide to protect ourselves by building a new kind of encryption. That's a laudable goal, but it's misguided. Without a deep understanding of how systems have been compromised, it's unlikely that someone would be able to design a safer system from first principles. Better to reign in that desire to build a new crypto system until you have broken a couple yourself. If you haven't broken anything yet, you are likely to just repeat other people's mistakes from the past. As the saying goes,

> Those who do not learn history are destined to have George Santayana quoted at them.

Fields, Kerckhoffs, and Shannon

It always catches my attention when I see similar advice from multiple traditions. It feels like triangulating in on the truth.

Linguist and cryptographer Auguste Kerckhoffs is best known for a pair of essays written in 1883. The key piece of advice from these essays is known these days as Kerckhoffs's Law, one translation of which is, "A cryptosystem should be secure even if everything about the system, except the key, is public knowledge."

Electrical engineer and cryptographer Claude Elwood Shannon made fundamental advances to circuit design and information encoding. He played an important role in American cryptography during World War II and worked with Alan Turing. Of particular interest to us is Shannon's Maxim, "The enemy knows the system."

It's not really a surprise that two early pioneers in cryptology would have such similar advice for us. What catches my eye, however, is how well this fits in with a lesson from W.C. Fields, a famous comedian, entertainer, and perhaps security researcher. Fields coined a famous saying that I like to call Fields's Imperative: "Never give a sucker an even break." (NGASAEB)

Fields's Imperative reminds us that when we're building a system, our design determines what the adversary has to achieve in order to defeat it. If we build a system that relies on the secrecy of the implementation for its security, we're giving the adversary an even break. Kerckhoffs and Shannon told us that we should expect our adversaries to understand our implementation. Consider how hard you'd have to work to make sure an adversary could never do any of the following:

- Find your backups

- Find your source control
- Find a disgruntled current or former developer
- Threaten or bribe a gruntled current or former developer
- Compromise a single computer that runs your software and then decompile the software
- Watch network traffic

Why bring this up? Because people who roll their own crypto commonly arrive at designs that assume a secret implementation and don't provide security if the adversary understands the implementation.

For further evidence of Kerckhoffs's Law and Shannon's Maxim, look at NaCl and Tink. They show that it's possible to be secure while also disclosing the full implementation.

Another way of looking at this is to ask yourself what the adversary would need to do in order to win. Are you content to let the adversary win if all they need to do is decompile your program? Or if all they need to do is see your source code? No! Don't set the bar that low. Never give a sucker an even break!

So what is modern cryptography built on? How does it provide security even while letting the implementation be known to the adversary?

Modern cryptography is built out of mathematical problems that appear to have no efficient solutions. To take one example, the security of RSA encryption is based on the difficulty of factoring large numbers. That is, given a large number x, find two numbers y and z such that $x = y * z$. In grammar school we learn how to take y and z and multiply them together to get x. But going the other way and splitting x into $y * z$ appears to be difficult. We can do it, but not always efficiently. It's easy to factor 35, for instance. By the time you finish reading this sentence, you'll probably have figured out that 35 can be represented as 5 * 7. You probably did this by either remembering your multiplication tables or trying to divide 35 by each integer up through 5. Try that approach on a number with hundreds of digits (as is the case in RSA), and you'll see quickly that this approach works slowly. Mathematicians have been working on this problem for centuries but haven't come up with anything terribly efficient. Mathematicians' tears are the best basis for cryptographic systems.

What does the adversary need to do to win? If your answer isn't as good as "Make a fundamental advance to mathematics that has eluded mathematicians for centuries," then you're better off not rolling your own crypto.

Security When the Enemy Knows the System

So if the entire implementation of our crypto systems is known to the adversary, how can we be secure? The adversary can just run our code after all.

The key is key. Encryption algorithms don't just take plain text as input, they take a key as well. A well-written encryption algorithm will produce wildly different outputs when encrypting a given plaintext with keys that differ only slightly. The key is the only part that needs to be kept secret. Rather than keeping an entire algorithm secret, we just need to keep our key secret.

An encryption algorithm should be so strong that even if an attacker had full access to the source code, the attacker would have no better option than to brute force all possible passwords. We won't cover how the encryption libraries recommended in this chapter achieve this goal. We'll merely note that they've been found to do so. If we've built a system like this, all the implementer has to do is to pick a suitably large random encryption key. Encryption keys are commonly 256 bits. That means that there are 2^256 possible values for an encryption key. That's 2 * 2 * … * 2, or 256 2s all multiplied together. 2^256 is an awfully big number. How big? Well, it's even bigger than the number of different ways you can order a standard deck of playing cards. There are about 8.06 x 10^67 (or 8 followed by 67 zeroes) different ways to order a deck of playing cards. The number 2^256 is about 1.15 x 10^77 (or 1 followed by 77 zeroes). So there are about 1.4 billion times more 256-bit encryption keys than there are ways to order a standard deck of playing cards.

Joe asks:

What's So Great About a Deck of Playing Cards?

I have to turn your attention to one of the most fun bits of math I've ever read.[a] It's a great way to help visualize just how many different ways you can order a standard deck of playing cards. When you order a deck, there are 52 possibilities for the first card. You pick the first one, and that leaves 51 possibilities for the second card, 50 for the third card, and so on. That makes for 52 * 51 * 50 * … * 4 * 3 * 2 * 1 different possibilities. The shorthand for this is 52! (pronounced "52 factorial") and it's so big, that, well, just read the linked story. It's fun.

a. czep.net/weblog/52cards.html

Kerckhoffs's and Shannon's advice comes primarily from witnessing firsthand the futility of keeping implementations secret. An additional benefit to building systems where the only secret is the key is that it's really hard to keep secrets. If a password is compromised, suspected of being compromised, or has just

been in use for too long, a secure system should be able to replace it with a new one and recover. (The problem of discovering what, if anything, an adversary did with compromised passwords and recovering from that is outside the scope of this book.)

Consider the relative difficulty of changing out all the passwords in a system versus the difficulty of changing out the entire system itself.