Introduction to Software Security

Chapter 2.1:
Secure Design Principles

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Loren Kohnfelder
loren.kohnfelder@gmail.com

DRAFT — Revision 1.3, February 2019.

1 Objectives

- Understand key principles that underlie the design of secure software.
- Learn how to apply security principles to software design.

2 Overview

Before learning how to design and build secure software, or evaluate the security of existing software, we will start from its underlying principles. These principles motivate the techniques we will introduce in later modules, and guide our thinking about software security. These principles have been culled from years of experience in the software security community. An experienced designer or programmer might incorporate these principles into their work without consciously thinking about them.

The first part of our discussion covers a collection of principles that guide how we think about incorporating security into our software. The second part describes how we put checks into our code to protect it. The third part describes principles for making the code more difficult to attack.

3 The Design Process

When you start on the design of a new piece of software, it is vital to include security in the earliest discussions and planning. The early inclusion of security can set the stage for a project that maintains a high standard of security. This is important because going back later and "adding security" is always more work and more likely to be flawed since it was not designed in from the beginning.

3.1 Transparent Design

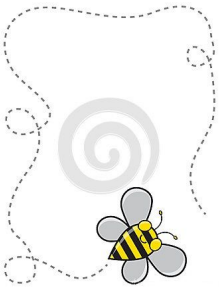
There is a well established principle in cryptography that the security of your communications should not depend on hiding the encryption algorithm, but instead be based on the communicating parties sharing a secret, a *key*. If the algorithm is effective, then knowing it does not give the attacker an advantage in decrypting a message for which they do not have the key. All software should be based on this principle: the security of your software should be based on its ability to prevent unauthorized accesses and not on some secret about its structure.

And there is even an advantage to making your code public: there are more eyes on your code, so more chances for someone to find (and we hope report) flaws in your design or implementation. This philosophy of transparency is a cornerstone of *open source* software, where the source code is freely available. More eyes on the code is considered an advantage to increase its security. The earliest work on fuzz random testing found that the open source variants of Unix were more robust than the commercial (closed source) versions[1,2].

The opposite of transparent design is unflatteringly called *security through obscurity*. While keeping your code design secret – and even making it intentionally complicated, unobvious, or messy – can make the job of the attacker more difficult, it does not guarantee that a well-trained and well-equipped (and patient) attacker cannot ultimately exploit your system. Such code is also more fragile. Once it is broken, the attacker is free to share the secret with anyone.

3.2 Avoid Predictability

Extending the discussion of encryption, let's say that you are using a strong encryption algorithm that is well known. If your choice of keys is predictable, then an attacker might be able to easily guess your key and read your communication or send fraudulent communication. This kind of predictability was used heavily in the code breaking efforts during World War II.



In software, we often generate secrets, such as when we generate sessions ID after logging on to an online service or website. These secrets are used as proof of identity for short intervals, so that we do not need to go through a full authentication protocol for each access to the server. If these secrets follow an obvious pattern, then they might be guessed, allowing unauthorized access to the server. Randomness is the key (sic) addressing this issue.

Whether it is generating session keys, random elements in a URL, or passwords, you want to use randomness to avoid providing hints (or "tells", as the World War II codebreakers called them) that the attacker can use to narrow down the space of possibilities. In Section 3 on secure programming, we will see examples of how to use this principle to avoid certain web attacks. And in Section 4 on defensive techniques, we will see how operating system designers have used this principle to avoid making the location of the code and data in a program predictable.

It is worth noting that randomness is a complex concept with a strong mathematical foundation. The use of weak random number generators (RNG) or cryptographic hash generators can weaken even the best design. For example, the SHA-1 hash function has been considered insecure since 2005, being replaced with the SHA-2 family, such as SHA-256 and SHA-512. Random number generators have specifically been vulnerable to numerous notable attacks, including one against Netscape (one of the earliest web

[1] B.P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities", *Communications of the ACM* **33**, 12 (December 1990). Also appears (in German translation) as "Fatale Fehlertractigkeit: Eine Empirische Studie zur Zuverlassigkeit von UNIX-Utilities", *iX*, March 1991.

[2] B.P. Miller, D. Koski, C.P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services", *Computer Sciences Technical Report #1268*, University of Wisconsin-Madison, April 1995. Appears (in German translation) as "Empirische Studie zur Zuverlasskeit von UNIX-Utilities: Nichts dazu Gerlernt", *iX*, September 1995.

browsers) in the mid 1990s and against Windows 2000 in the mid-2000s. In general, pick the best hash or RNG that you can find, specifically one meant to be "cryptographically secure" leaving the details to the mathematicians and cryptographers.

3.3 Economy of Design or Least Common Mechanism

In a famous aphorism, variously attributed to the physicists Herbert Spencer[3] or Albert Einstein,

"Everything should be as simple as possible but no simpler". The essence of this idea is to strip away unnecessary complexity, leaving only what you need to get the job done. In software, complexity makes it more difficult to find bugs in the code, make the code run fast, and find security flaws.

Note that this aphorism includes the warning "...but no simpler." This warning is necessary because it is possible to reduce a design or piece of code past the point of good sense. For example, putting checks on the return values of every system call and external library call is essential to correct and secure operation of software. However, such checks can clutter the code and make it more difficult to read, so there is a temptation to leave off at least some of these checks.

This principle can be stated as: do things once, in a common place. If you are going to check authorization for access to a resource, have a single function/method that does this job. If you are going to run user code in a sandbox, have a single function/method that does privilege de-escalation (capturing all the subtle and many aspects of privilege). Doing things once allows you to concentrate your energy and attention on the problem, and then reap the benefits of your design when you need to do the same task again.

A function that implements a particular security operation should be named in an obvious way and documented clearly. This will help when another programmer who needs to implement the same functionality and reduce the chances that they will reinvent what is already done (perhaps in an inferior way).

As software projects grow over time, we often discover common functionality that is embedded in multiple places in the design or code. The best response to such a discovery is to refactor the code such that you extract the common functionality and put it into a single function (or functions), and place calls to that function where the code used to be embedded. Refactoring is time consuming, so there is a great temptation to just cut-and-paste the original code. The cut-and-paste strategy likely will produce short term gain for long term pain. Each application of this strategy makes the code more complex and more difficult to refactor in the future.

[3] "It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience." From "On the Method of Theoretical Physics," the Herbert Spencer Lecture, Oxford University, June 10, 1933. There is some belief that Spencer based this idea on discussions that he had with Albert Einstein.

3.4 Accept Security Responsibility

Security starts with the decision to include it as a priority element of your software design and implementation. Including security means that you must acquire the skills and spend the time necessary to ensure that security is an intrinsic part of your design. Behind such a decision is willingness to accept the upfront costs for including security.

An immediate cost of accepting responsibility for security is a delay in writing the first lines of code until you have thought about the structure of your systems, the threats that may affect it, and the defenses that you will need to provide the necessary level of security. Threat Modeling, described in the next chapter, is a good way to bring security into your design in a structured (and documented) way.

Later costs include training your teams to understand how to code securely. The material in Section 3 offers a good resource to acquire such skills. Along with coding securely, you will need code reviews that explicitly target security issues. Companies with mature software security programs will often have two code reviews related to a commit, one for functionality and the other for security. The testing or quality assurance (QA) phase of your project will also need to verify the integrity of security features. Most testing teams do not have security training, so acquiring such training is an additional cost.

The last area of responsibility is that of communicating with your user community. Users should have a clear understanding of what steps you are taking to produce secure software. And, equally important, when a security flaw is discovered, users should get timely reports as to the nature and severity of the flaw, the scope of its impact, and the fix.

Of course, there is great benefit to all these costs. The earlier that a design or coding flaw is detected, the cheaper that it is to fix. And such a robust security program can reduce the number of security events you will face, providing further cost benefits.

4 Protect the Target

4.1 Complete Mediation

Protecting your software means protecting all paths into that software, and protecting access to the resources controlled by your software. The battle between the attacker and defender is an asymmetric one: the attacker only has to find one path into your system, while you have to ensure that every path is defended. It is the covering of “every path” that forms complete mediation.

The starting point for complete mediation is understanding the attack surface (presented in Section 1) of your software. For a given resource, you need to understand all the paths from the attack surface to accesses to that resource.

Once you understand all the paths, you need to control them. Well designed software will have a single point that controls access to a resource, for all interactions. Such control is an application of the Economy of Design principle presented earlier.

4.2 Defense in Depth

It is commonly said that security should be like an onion, coming in multiple layers. If you somehow break through one layer, there are more layers left to protect you.[4] Experienced security practitioners like to brag how they wear both belts and suspenders. The opposite of layered security is what we like to call "deep fried security", which is crispy on the outside and sweet and tasty on the inside. If you break through the brittle outside layer, all the insides are exposed.

Suppose that you are writing a new class `C2` in a software system, and this class includes method `mm`. Of course, this new class is going to call methods in some existing classes, for example you need to call method `m` in class `C1`. In the lines of code before the call to `C1.m`, you check the parameters that you are passing to `m` to make sure that they are valid, so you know that `m` will always compute something sensible. Nevertheless, as a careful programmer, you still check the return value from `C1.m` just to make sure that you didn't miss something.

Now, suppose you are writing yet another new class, `C3`, that includes method `mmm`. And `mmm` needs to call method `C2.mm`. Before this call, you will carefully check the parameters you are passing to mm, and after the call, you will check the return value even though you are sure that `mm` should not return an error since you checked the input parameters.

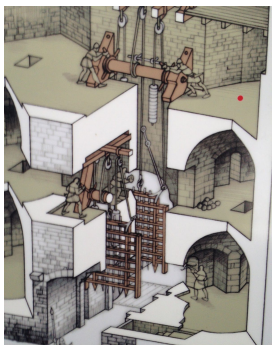
At each layer, we do explicit checking to prevent unnoticed errors, which can prevent unnoticed exploits. Such checking is crucial for several reasons, including:

- You might have missed a corner case with your parameter checking, so a future call might now pass a valid parameter.
- Someone (including you, in the future) might change the method in a way that you did not expect, so that the range of valid parameter values has changes. Since the person making the change might not know every place in the code that calls the changed method, they might not be able to update all the places that call it.
- You might run the code in a new environment, perhaps on a new release of the operating system, so that the notion of what is a valid parameter could change.

[4] Security is also said to be like an onion in that it makes you cry, but that is less helpful here!

4.3 Separation of Privilege

The goal of separation of privilege is to require more than one entity to grant permission before an action can be taken. It is used when high levels of security are needed. A classic example of such a separation is used with the launch of a nuclear missile. This is an event of such severity that we do not want any individual with the power to initiate such a launch. As a result, there typically are several people involved, each with physical and digital keys that **all must be present and authenticated** to make this happen.



In the real world, we can see many less dramatic examples of separation of privilege. For example, to open a safe deposit box, you typically require two keys, one kept by the bank and one kept by the owner of the box. In addition, the boxes are in a vault with its own independent security. Also in the financial world, commercial checking accounts will have a requirement for multiple signatures on a check when the amount is above a certain threshold.

5 Making the Target Harder to Hit

5.1 Least Privilege

A program ideally should run at the lowest reasonable privilege level necessary to do its job. The most obvious example is that a program should not run as "root" or "administrator" unless it needs that all-encompassing level of privilege. When we want to protect software from a regular user, we are often tempted to install and run it as root. However, using root privilege means that any flaw in the software will have more global and perhaps more catastrophic results.

Suppose that you need to create a new service that answers queries based on a database. You want the data in this database protected from direct access by normal users, so your first inclination is to install the software as root, and make all the files associated with the database accessible only by root. Such a strategy satisfies the goal of preventing normal users from accessing the database, but is fragile and dangerous if compromised.

A more effective strategy would be to create a new user ID for the new service. This new ID allows the service to protect its processes and files from other users, but does not require running as root.

In practice, least privilege should not be carried too far with numerous privilege adjustments to always literally be at minimum. Good designs look for the sweet spots and use appropriate levels of privilege in balance with the needs of the system and attendant risks.

5.2 Least Information

In the intelligence world, they call the principle of Least Information "need to know". The idea is that you should only access the information that you need to do your job. If you don't have

access to other information, you will not accidentally leak it or inappropriately modify it.

In a later module, we will talk about TMI (too much information) errors that inadvertently expose information advantageous to attackers. For example, suppose that client code running on a user's machine is making a query to a server to find out some other user's phone number. This phone number is stored in a database, where each record contains all the information for a single user. If the query returned the entire record to the client, this might expose other private information needlessly. The phone number query should only return those values (attributes) that are relevant to the query. We have seen exactly this kind of error made in real world systems. Fortunately, we caught it before anyone else did.

Such an error also happened in December 2015, during the U.S. presidential primary election campaign at a company that stores voter data. A misconfiguration in the database allowed data private to Hillary Clinton's campaign to be accessed by members of Bernie Sanders' campaign. This time, the error ended up on the front page of the world's major newspapers!

5.3 Secure by Default

We should always think about the failure cases when we write code. All programs have flaws, so we want to write software that minimizes the effect of such flaws. An example that we have repeatedly seen is in a function that validates a user name and password on the server. We have often seen this function written with first line of code something like:

```
login = TRUE;
```

where the rest of the function checks the user name and password against the values in the database, setting the variable to `FALSE` if they do not match.

This code is inherently fragile because if there are any unforeseen errors (and there usually will be), then such an error will inadvertently cause the function to return `TRUE` when it is not appropriate to do so. Simply reversing the logic – starting with a value initialized to `FALSE` and only setting it to `TRUE` if all login conditions are satisfied – is much less error prone. And, if there is an error, it is likely to cause less serious outcomes.

6 Summary

This chapter was a natural extension of Thinking Like an Attacker, providing some specific principles on which to base our technical approaches to designing and building secure software, by anticipating the attacker and taking countermeasures. How you think about software is crucial and is as important as learning a collection of specific techniques. Taken together, having basic principles for secure software design and a list of techniques for writing more secure code, you will be well positioned to more effectively fend off the bad guys.

7 Exercises

1. TBD
2. TBD
3. TBD
4. TBD

5. TBD