Practical Data Security

Kyle Isom

Table of contents

Acknowledgements

This book almost certainly would not have been written were it not for the Pragmatic Programmers, LLC, and their PragProWriMo event (http://pragprog.com/magazines/2011-10/up-front) . Although I’ve had the idea to do this for a while, the PragProWriMo was the spark that ignited the fuel.

I'd also like to thank the people who reviewed this book: Beau and Essie Holton, Matthew Sowers, and Clark Lester.

CHAPTER

Introduction

This is a book about how to use data security practically. It's written assuming that you are a coder who's already writing cool and audacious projects, but you don't have a background in writing secure systems. I also assume that you have a passing familiarity with UNIX-based operating systems; while I don't expect you to be a UNIX wizard, I also expect you to not be terribly daunted when tinkering about the command line. I try not to get in your way and merely to facilitate a discussion in most cases about ways we as coders can write better applications for our users.

The news headlines are full of stories about popular online services being com-promised. It's becoming something that we are more used to living with. I think that we can do better. My particular interest in this problem started with two events that coincided: the first involved writing software for work to build encrypted updates for a project I was working on. I was doing this in Python, and so became fairly familiar with the PyCrypto library (https://www.dlitz.net/software/pycrypto/) . At the same time, a friend of mine was working on writing authentication code, also in Python. He was trying to use the PyCrypto library as well, but really didn't understand how cryptography worked and how to apply it in this case. That inspired me to write a ten page introduction to cryptography (also illustrated with Python). I've been mulling over the idea of expanding that into a book on data security, and now the time has come to write it.

From my perspective, a lot of the problem stems from a fundamental misunderstanding of what security is, and how to integrate it. I will try to address that issue here, and hopefully help other developers to understand not only why it's important, but to see that it's not just snake oil to be thrown into the mix so that it "just works."

The book is subtitled Illustrated With Python, which I am slightly worried will discourage those not familiar with the Python language. I chose to use Python because it is both a language I am well familiar with, and because it is a very readable language. Readers should note that the emphasis in the book is not on the Python way to do things; readers who are familiar with Python will note that I shy away from idiomatic Python that will be difficult for those unfamiliar with the language to understand.

CHAPTER 0

Introduction to Data Security

Data security is one of those things you hear a lot about, but most of us are busy getting things done; unfortunately for our users, data security tends to be put on the back-burner, if we remember it at all. Data security doesn't have to be a burden on the developer, and most importantly, it's in the best interest of users. One principle I particularly like for software engineering is the Principle of Least Astonishment.

THE PRINCIPLE OF LEAST ASTONISHMENT:
A program's behaviour should not surprise the end user.

How does this relate to data security? A user never expects their data to be leaked out of the application or their information to be shared with people they dont intend for it to. This might be an explicit understanding or an implicit one, especially when users don't really understand the problem domain. If your users are from a wide range of backgrounds, they may not have a solid understanding of the basics of how to keep their information safe. In fact, they may not even realise what they don't know. So, it becomes even more important for developers to ensure that, as much as possible, their data is safe. What I will discuss in this book are ways to do that, while keeping in mind the misconception (not always true) that the more secure a system is, the less accessible it is. I'd like to address that misconception, and help you to make your applications as transparently secure as possible.

The first thing we'll take a look at are some of the objectives of information security that are most generally applicable to software, i.e. those that we can address. Then we'll take a high-level walk through some basic UNIX security that's applicable (and perhaps on the Windows platform as well). Even if your application is a web application, it's probably running on a UNIX machine of some sort, and that represents an area that you should pay attention to as well. If you're a Windows developer, there will still be some information applicable, but you can probably skim through this part. Then we'll take a high level look at cryptography; I've chosen to stay away from deep mathematical discussions and only discuss those aspects you need to understand as a coder to effectively make use of it. Following that, we'll see a few scenarios where your code is vulnerable. The rest of this chapter will be a gentle introduction to each of those areas, and the rest of the book will flesh it out with code examples.
There are four information security principles you should keep in mind when writing software:

1. Privacy
2. Anonymity
3. Repudiation
4. Integrity

Let's take a look at these in a little more depth, and see how they apply to the software we're writing.

First, we visit the concepts of **authentication** and **authorisation**. Authentication is confirming the identity of a user--most often by way of a username (the user's identity) and password (the secret used to authenticate). Other ways of authentication include using smart cards, where the identity is given by the cryptographic key on the card and the authentication is done by entering a PIN to unlock the key, or perhaps using something like a one-time password token tied to an OpenID. Once the user's identity has been confirmed (the user is now **authenticated**), their access to resources is controlled by **authorisation**. Authorisation is the check to see "is this user allowed to do this?" If we look at this from the perspective of a user's ability to access files on a computer, they are typically authenticated by their login and password (or, on some laptops, via a login and fingerprint). When a user tries to read or write to a file, the system checks the access control list (an authorisation scheme) to see if the user is authorised to do this. Let's look at the Twitter application on a mobile as another example: when you try to use the app, you have to sign into twitter and authorise access on Twitter's website for the application. Your username and password authenticate you to Twitter, and then you choose to authorise the application's access to the site. Authentication and authorisation may be done by other means as well for entities other than users. We'll call any entity a user in this book, but that user might be a person, a computer, or something else entirely. With that dead horse beaten, we can move on.

The principle of **privacy** states that information may only be viewed by those authorised to view it. In terms of UNIX files, a user might be able to log into a computer, but does not have access to read another user's files. On twitter, you can protect your timeline, so that only your friends can read your tweet. It's not unreasonable for a user to expect they can control who else has access to their resources; obviously a user with a public blog expects anyone to be able to read their posts, but they may not expect just anyone to see other details about that user. When you're designing an application, it helps to think about which pieces of information a user should have control over. Some applications will require fine-grained control--say it's designed so user A can choose to share one post only with group B and another post only with group C. Others will be course-grained--users in group B will always have access to posts by user A.

Anonymity is a hot topic these days with the so-called "nym wars" on Google+ as but one example. This boils down to the idea that, depending on your stance, a user should be able to separate their different identities from each other. In the Google+ debate, it's the idea that users want to be able to separate their real-world identity from their electronic identity. That is, my identity on Google+ may not be the same identity I want to use on my blog. This brings out another point about security: most security engineering isn't just a technical question; it's a social question. Many of our applications aren't just technical applications, but have social value as well; many security decisions are made by weighing social considerations. Your decision in regards to anonymity is going to be a social decision, but it should be decided while designing the application. Bolt-on security (added after the design phase) may be somewhat effective, but it tends to cause a lot of problems.

Repudiation is a fancy word that means the ability to deny ownership of information. You'll hear this term a lot in discussions about cryptography, particularly with digital signatures. If a user's data is **repudiable**, the user can plausibly deny owning that data (for example, user A might want to be able to deny having posted something unpopular about a sensitive topic). Conversely, if a user's information is **non-repudiable**, she cannot plausibly deny owning that information. I'll have more to say on this in a bit, but for now you know what I'm talking about when I bandy about the term like "non-repudiable."

Users also want some degree of assurance that their data won't be changed or tampered with. This is the principle of **data integrity**--information may only be changed by people authorised to do so. This isn't just a security issue; natural disasters and junior sysadmins (some would say naming both is a redundancy) have a clever way of accidentally modifying your data without the user's authorisation. Of course, we expect (and probably constantly tell) users to backup their data, but your application may not be easily backed up by the user. There's also a case to be made for **transparency** in both showing users what data you have of theirs on your systems or in your application and allowing them to download or backup that information easily. Even data like photos, which are easy to back up, may come from a disparate number of sources, should be considered. For example, a user might upload photos from his phone, his laptop, and his tablet. All these different sources may make backing up more difficult, although far from impossible, and having one place to back them all up from may be useful.

These basic principles are all properties to consider when designing your system; they fit alongside all the other software engineering principles like modularity that should be considered. It is far easier to design these in at the start than to try to bolt them on later.

Another thing to think about is the use case for your software; namely, you might not be able to think of all the ways users will use the system. Of course, we'd like to think of our users as living in an ideal world where using real names isn't an issue and no one has anything to hide, but this is not the world we live in. You should also consider some of the challenges faced by users living in non-ideal places. If you're allowing people to upload data or users must download information to synchronise, consider that many users don't have high-speed internet or a large bandwidth cap. In many countries, power is at a premium, and users shut down their computers when they aren't using them out of economic necessity, instead of being able to leave them on. Users might also be living in restrictive regions where posting information critical of other people might be dangerous for them. If you have any desire for your software to have a global impact, you should consider them. If your application isn't region-specific, you should most certainly keep this in mind when designing your software.

There are a couple areas you can start to actually design security into your code. First, there are the fundamental security mechanisms available on the platform you are coding on. There are coding errors that can lead to flaws, such as the well-known SQL injections and buffer overflows that are some of the mainstays of many successful attacks. There are mechanisms to secure data in transit (when the data is being accessed and used) and at rest (when the data is not being used). As coding errors have been dealt with quite effectively in many other books (see the bibliography for more), we'll focus on the fundamental security mechanisms and other data security mechanisms, and in particular, how we can use basic UNIX security mechanisms and cryptography effectively. We'll also consider some real world ways in which failing to secure your code can lead to an exploit.

CHAPTER 1

Practical Threat Modeling

Threat modeling is the art of thinking about what threats your application will face. By understanding the threats, you can begin thinking about how to mitigate them.

One useful tool when threat modeling is an **attack tree**. In an attack tree, you identify what threats you might face, and what those attacks lead to. During the initial stages, you don't focus on mitigation (or what you have done so far). It's important to consider what sorts of attacks can be made: while you might think an attack is mitigated or you may not see it as practical, changes to the code can easily change the situation. It's important to regularly review the attack tree, updating it as necessary. If you add a feature, consider the potential security impact. The point of the attack tree is just to visualise what could happen; later parts of the security engineering process transform this into a practical outlook.

Let's consider a simple web site that allows users to update their information; it's backed by a fairly simple database. Some potential attack vectors that come to mind in a quick overview are:

1. SQL injection on the login form -> privilege escalation
2. password interception on the wire -> stealing user's account information
3. password brute forcing -> stealing user's account information
4. unauthorised access to resources, i.e. user B can read or update user A's information without their consent.

It's also helpful to identify vulnerability points with these:

1. login form (SQL injection, password brute forcing)
2. the wire, i.e. the connection between the user and the server (password interception)
3. whereever user information can be looked up

From here, we can identify mitigation steps fairly easily:

1. Sanitize all user inputs.
2. Secure the connection using SSL.
3. Use bcrypt (https://en.wikipedia.org/wiki/Bcrypt) to store passwords.
4. Force all sensitive information to be routed through authorisation middleware.

Generally, it's good practise to keep these in separate documents; i.e. an attack tree (though often annotated with the vulnerability points), and a security plan. This allows you to consider the two separately when needed. The security plan should reference the attack tree in considering what attacks are being defended against. These are also living documents; as mentioned before, they will need to be reviewed and updated regularly.

## Identifying Attack Vectors A good place to check for attack vectors is to consider the generic attack points:

1. Where does data enter the system? Unsanitised data can pose a risk, as can data that is not normalised to expected values.
2. Where does data leave the system? **Information leakage** is a security issue where information that should not leave the system does, or is revealed by the system unintentionally (typically without effort on the part of the attacker).
3. What sensitive data does the system use? Passwords an obvious answer that come to find first; also, consider email addresses and other personal details. Resources that should not be widely accessible to the public fall under this category as well; for example, if you are publishing a paid magazine online, you want to consider restricting access to it.
4. Where and how is sensitive information stored? For example, when passwords aren't being used, how are they being stored? Are you using bcrypt or are they being stored in plaintext?

This can be a difficult task. You have to be creative, and you will likely still not consider everything an attacker will. Having several people involved in developing the attack tree is very useful, as they will all have different mindsets and backgrounds to contribute. One technique I find useful is to visualise your program as a system, diagramming subsystems and information flow. Just as physical security engineers will study the physical layout of the area they are working to protect and bank robbers will often observe their target beforehand, it is another tool useful for visualising where security systems fit into the overall architecture.

## Example Let's consider an example: I have a website that provides a front end for my small team's ebook library. Here is the systems diagram I have for the site:

systems diagram

And here is a sample attack tree I've come up with:

attack tree

What are the attack vectors? If we organise them by the four generic attack points, we could come up with:

1. Where does data enter the system?
    - login form
    - user book uploads
    - user metadata fields
    - book cover images
    - user bio
2. Where does data leave the system?
    - book download
    - user bio display
    - displaying catalog information
    - book cover display

3. What sensitive data is present in the system?

    - passwords
    - user catalog

4. Where and how is sensitive information stored?

    - passwords: stored in the database as bcrypted hashes
    - user catalog: stored in the database and accessed through an access control list (ACL)

APPENDIX A
Bibliography

Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Introduction to Differential Power Analysis and Related Attacks." Cryptography Research, San Francisco, CA 1998. http://www.cryptography.com/resources/whitepapers/DPATechInfo.pdf

Paul Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis." Cryptography Research, San Francisco, CA 2000. http://www.cryptography.com/resources/whitepapers/DPA.pdf