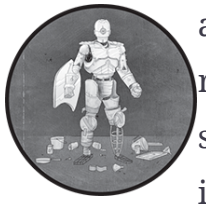
1

FOUNDATIONS

> *Honesty is a foundation, and it's usually a solid foundation. Even if I do get in trouble for what I said, it's something that I can stand on.*
>
> *—Charlamagne tha God*



Software security is at once a logical practice and an art, one based on intuitive decision making. It requires an understanding of modern digital systems, but also a sensitivity to the humans interacting with, and affected by, those systems. If that sounds daunting, then you have a good sense of the fundamental challenge this book endeavors to explain. This perspective also sheds light on why software security has continued to challenge the field for so long, and why the solid progress made so far has taken so much effort, even if it has only chipped away at some of the problems. Yet

there is very good news in this state of affairs, because it means that all of us can make a real difference by increasing our awareness of, and participation in, better security at every stage of the process.

We begin by considering what security exactly is. Given security's subjective nature, it's critical to think clearly about its foundations. This book represents my understanding of the best thinking out there, based on my own experience. Trust undergirds all of security, because nobody works in a vacuum, and modern digital systems are far too complicated to be built single-handedly from the silicon up; you have to trust others to provide everything (starting with the hardware, firmware, operating system, and compilers) that you don't create yourself. Building on this base, next I present the six classic principles of security: the three components of classic information security and the three-part "Gold Standard" used to enforce it. Finally, the section on information privacy adds important human and societal factors necessary to consider as digital products and services become increasingly integrated into the most sensitive realms of modern life.

Though readers doubtlessly have good intuitions about what words such as *security*, *trust*, or *confidentiality* mean, in this book these words take on specific technical meanings worth teasing out carefully, so I suggest reading this chapter closely.

As a challenge to more advanced readers, I invite you to attempt to write better descriptions yourself—no doubt it will be an educational exercise for everyone.

Understanding Security

All organisms have natural instincts to chart a course away from danger, defend against attacks, and aim toward whatever sanctuary they can find.

It is important to appreciate just how remarkable our innate sense of physical security is, when it works. By contrast, we have few genuine signals to work with in the virtual world—and fake signals are easily fabricated. Before we approach security from a technical perspective, let's consider a real-world story as an illustration of what humans are capable of. (As we'll see later, in the digital domain we need a whole new set of skills.)

The following is a true story from an auto salesman. After conducting a customer test drive, the salesman and customer returned to the lot. The salesman got out of the car and continued to chat with the customer while walking around to the front of the car. "When I looked him in the eyes," the salesman recounted, "That's when I said, 'Oh no. This guy's gonna try and steal this car.'" Events accelerated: the

customer-turned-thief put the car in gear and sped away while the salesman hung on for the ride of his life *on the hood of the car*. The perpetrator drove violently in an unsuccessful attempt to throw him from the vehicle. (Fortunately, the salesman sustained no major injuries and the criminal was soon arrested, convicted, and ordered to pay restitution.)

A subtle risk calculation took place when those men locked eyes. Within fractions of a second, the salesman had processed complex visual signals, derived from the customer's facial expression and body language, distilling into a clear intention of a hostile action. Now imagine that the same salesman was the target of a *spear phishing* attack (a fraudulent email designed to fool a specific target, as opposed to a mass audience). In the digital realm, without the signals he detected when face-to-face with his attacker, he'd be much more easily tricked.

When it comes to information security, computers, networks, and software, we need to think analytically to assess the risks we face if we want to have any hope of securing digital systems. And we must do this despite being unable to directly see, smell, or hear bits or code. Whenever you're examining data online, you're using software to display information in human-readable fonts, and typically, there's a lot of code between you and the actual bits; in fact, it's potentially a hall of

mirrors. So you must trust your tools and trust that you really are examining the data you think you are.

Software security centers on the protection of digital assets against an array of threats, an effort largely driven by a basic set of security principles that the rest of this chapter will discuss. By analyzing a system from these first principles, we can learn how vulnerabilities slip into software, as well as how to proactively avoid and mitigate problems. These foundational principles, along with other design techniques covered in subsequent chapters, apply not only to software but also to designing and operating bicycle locks, bank vaults, or prisons.

The term *information security* refers specifically to the protection of data and how access is granted. *Software security* is a broader term that focuses on the design, implementation, and operation of software systems that are trustworthy, including the reliable enforcement of information security.

Trust

Trust is equally critical in the digital realm, yet too often taken for granted. Software security ultimately depends on trust, because you cannot control every part of a system, write all of your own software, or vet all suppliers of dependencies.

Modern digital systems are so complex that not even the major tech giants can build a complete technology stack from scratch. From the silicon to the operating systems, networking, peripherals, and the numerous software layers that make it all work, the systems we rely on routinely are remarkable technical accomplishments of immense size and complexity. Since nobody can build these systems all by themselves, organizations rely on hardware and software products often chosen based on features or pricing—but it's important to remember that each dependency also involves a *trust decision*.

Security demands that we examine these trust relationships closely, even though nobody has the time or resources to investigate and verify everything. Failing to trust enough means doing a lot of needless work to protect a system when no real threat is likely. On the other hand, trusting too freely could mean getting blindsided later. Put bluntly, when you fully trust an entity, they are free to fail without consequences. Trust can be violated in two fundamentally different ways: by malice (cheating, lying, subterfuge) and by incompetence (mistakes, misunderstandings, negligence).

The need to make critical decisions in the face of incomplete information is precisely what trust is best suited for. But our innate sense of trust relies on subtle sensory inputs wholly unsuited to the digital realm. The following discussion begins

with the concept of trust itself, dissects what trust as we experience it is, and then shifts to trust as it relates to software. As you read along, try to find the common threads and connect how you think about software to your intuitions about trust. Tapping into your existing trust skills is a powerful technique that over time gives you a gut feel for software security that is more effective than any amount of technical analysis.

Feeling Trust

The best way to understand trust is to pay attention while experiencing what relying on trust actually feels like. Here's a thought experiment—or an exercise to try for real, with someone you *really trust*—that brings home exactly what trust means. Imagine walking along a busy thoroughfare with a friend, with traffic streaming by only a few feet away. Sighting a crosswalk up ahead, you explain that you would like them to guide you across the road, that you are relying on them to cross safely, and that you are closing your eyes and will obediently follow them. Holding hands, you and your friend proceed to the crosswalk, where they gently turn you to face the road, gesturing by touch that you should wait. Listening to the sounds of speeding cars, you know well that your friend (and now, guardian) is waiting until it is safe to cross, but your heartbeat has most likely also increased noticeably, and you

may find yourself listening attentively for any sound of impending danger.

Now your friend unmistakably leads you forward, guiding you to step down from the curb. If you decide to step into the road with your eyes closed, what you are feeling is pure trust—or perhaps some degree of the lack thereof. Your mind keenly senses palpable risk, your senses strain to confirm safety directly, and something deep down is warning you not to do it. Your own internal security monitoring system has insufficient evidence and wants you to open your eyes before moving; what if your friend somehow misjudges the situation, or worse, is playing a deadly evil trick on you? Ultimately, it's the trust you have invested in your friend that allows you to override those instincts and cross the road.

Raise your own awareness of digital trust decisions, and help others see how important their impact is on security. Ideally, when you select a component or choose a vendor for a critical service, you'll be able to tap into the very same intuitions that guide trust decisions like in the exercise just described.

You Cannot See Bits

All of this discussion is to emphasize the fact that when you think you are "looking directly at the data," you are actually looking at a distant representation. In fact, you are looking at

pixels on a screen that you believe represent the contents of certain bytes whose physical location you don't know with any precision, and many millions of instructions were likely executed in order to map the data into the human-legible form on your display. Digital technology makes trust especially tricky, because it's so abstract, lightning fast, and hidden from direct view. Whenever you examine data, remember that there is a lot of software and hardware between the actual data in memory and the pixels that form characters that we interpret as the data value. If something in there were maliciously misrepresenting the actual data, how would you possibly know? Ground truth about digital information is extremely difficult to observe directly.

Consider the lock icon in the address bar of a web browser indicating a secure connection to the website. The appearance or absence of these distinctive pixels communicates a single bit to the user: safe or unsafe. Behind the scenes, there is a lot of data and considerable computation, as will be detailed in Chapter 11, all rolling up into a binary yes/no security indication. Even an expert developer would face a Herculean task attempting to manually confirm the validity of just one instance. So all we can do is trust the software—and there is every reason that we should trust it. The point here is to recognize how deep and pervasive that trust is, not just take it for granted.

Competence and Imperfection

Most attacks begin by exploiting a software flaw or misconfiguration that resulted from the honest, good faith efforts of programmers and IT staff, who happen to be human, and hence imperfect. Since licenses routinely disavow essentially all liability, all software is used on a *caveat emptor* basis. If, as is routinely claimed, "all software has bugs," then a subset of those bugs will be exploitable, and eventually the attackers will find a few of those bugs and have an opportunity to use them maliciously. It's relatively rare for software professionals to fall victim to an attack due to misplaced trust in malicious software, enabling a direct attack.

Fortunately, making big trust decisions about operating systems and programming languages is usually easy. Many large corporations have extensive track records of providing and supporting quality hardware and software products, and it's quite reasonable to trust them. Trusting others with less of a track record might be riskier. While they likely have many skilled and motivated people working diligently, the industry's lack of transparency makes the security of their products difficult to judge. Open source provides transparency, but depends on the degree of supervision the project owners provide as a hedge against contributors slipping in code that is buggy or even outright malicious. Remarkably, no software

company even attempts to distinguish itself by promising higher levels of security or indemnification in the event of an attack, so as customers we have no such options. Legal, regulatory, and business agreements all provide additional ways of mitigating the uncertainty around trust decisions.

Take trust decisions seriously, but recognize that nobody gets it right 100 percent of the time. The bad news is that these decisions will always be imperfect, because, as the US Securities and Exchange Commission warns us, “past performance does not guarantee future results.” The good news is that people are highly evolved to gauge trust—though it works best face-to-face, decidedly not via digital media—and in the vast majority of cases we do make the right trust decisions, provided we have accurate information and act with intention.

Trust Is a Spectrum

Trust is always granted in degrees, and trust assessments always have some uncertainty. At the far end of the spectrum, such as when undergoing major surgery, we may literally entrust our lives to medical professionals, willingly ceding not just control over our bodies but our very consciousness and ability to monitor the operation. In the worst case scenario, if they should fail us and we do not survive, we literally have no

recourse whatsoever (legal rights of our estate aside). Everyday trust is much more limited: credit cards have limits to cap the bank's potential loss on nonpayment, while cars have valet keys so we can limit access to the trunk.

Since trust is a spectrum, a "trust but verify" policy is a useful tool that bridges the gap between full trust and complete distrust. In software, you can achieve this through the combination of authorization and diligent auditing. Typically, this involves a combination of *automated auditing* (to accurately check a large volume of mostly repetitive activity logs) and *manual auditing* (spot checking, handling exceptional cases, and having a human in the loop to make final decisions). We'll cover auditing in more detail later in this chapter.

Trust Decisions

In software, you have a binary choice: to trust, or not to trust? While some systems do enforce a variety of permissions on applications, you still need to either allow or disallow each given permission. When in doubt, you can safely err on the side of distrusting, so long as at least one candidate solution reasonably gains your trust. If you are too demanding in your assessments, and no product can gain your trust, then you are stuck with the prospect of building the component yourself.

Think of making trust decisions as cutting branches off a decision tree that otherwise would be effectively infinite. When you can trust a service or computer to be secure, that saves you the effort of doing deeper analysis. On the other hand, if you are reluctant to trust, then you need to build and secure more parts of the system, including all subcomponents. *Figure 1-1* illustrates an example of making a trust decision. If there is no available cloud storage service you would fully trust to store your data, then you must operate the service yourself, and this entails further trust decisions: to use a trusted hosting service or do it yourself, and to use existing database software that you trust or write it yourself. Note that when you don't trust a provider then more trust decisions are sure to follow since you cannot do everything.

For explicitly distrusted inputs—which should include virtually all inputs, especially anything from the public internet or any client—treat that data with suspicion and the highest levels of care (for more on this, see "Reluctance to Trust" on page 68 in Chapter 4). Even for trusted inputs, it can be risky to assume they are perfectly reliable. Consider opportunistically adding safety checks when it's easy to do so, if only to reduce the fragility of the overall system and to prevent the propagation of errors in the event of an innocent bug.

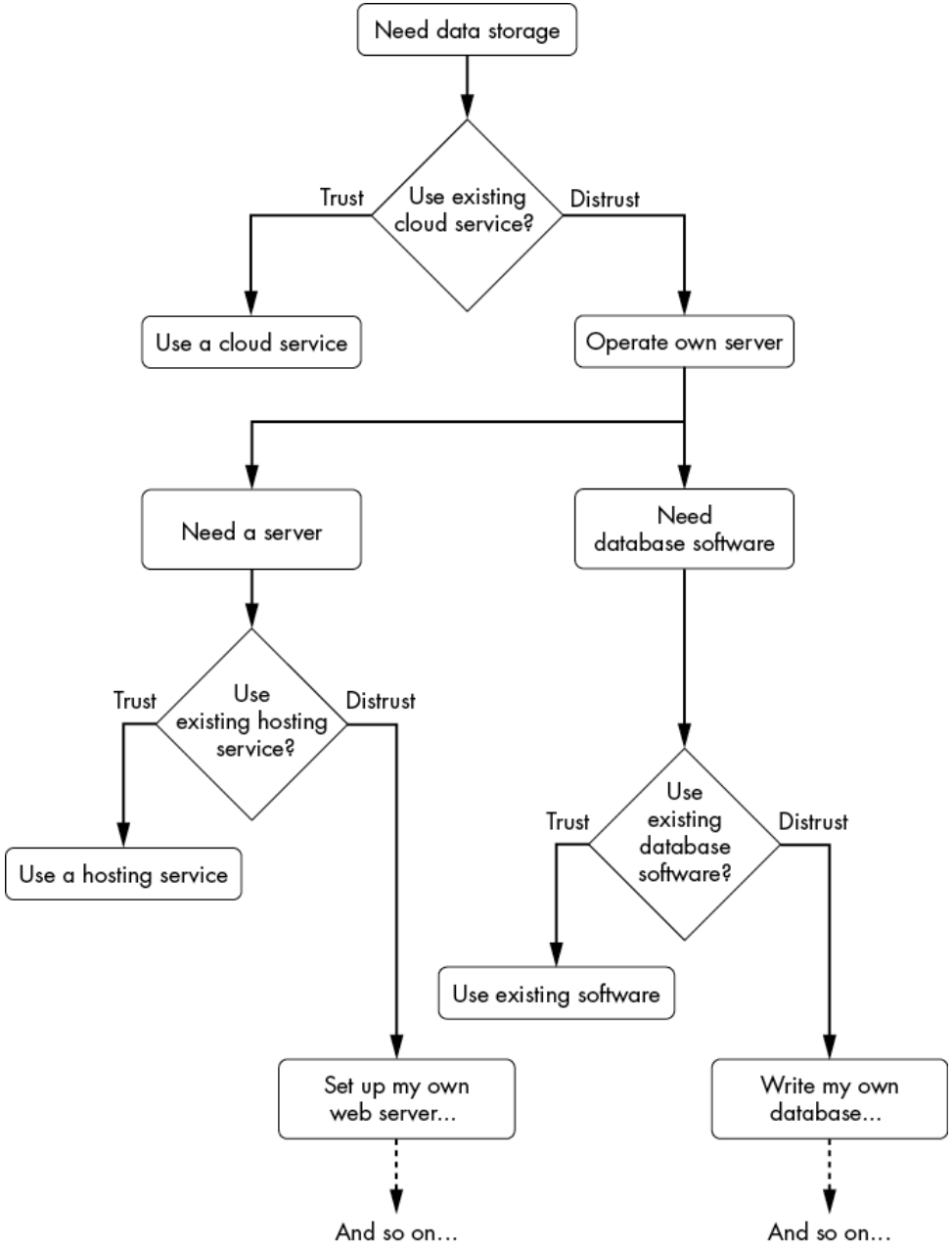


Figure 1-1: An example of a decision tree with trust decisions

Implicitly Trusted Components

Every software project relies on an extensive stack of technology that is *implicitly trusted*, including hardware, operating systems, development tools, libraries, and other dependencies that are impractical to vet, so we trust them based on the reputation of the vendor. Nonetheless, you should maintain some sense of what is implicitly trusted, and give these decisions due consideration, especially before greatly expanding the scope of implicit trust.

There are no simple techniques for managing implicit trust, but here is an idea that can help: minimize the number of parties you trust. For example, if you are already committed to using Microsoft (or Apple, and so forth) operating systems, lean toward using their compilers, libraries, applications, and other products and services, so as to minimize your exposure. The reasoning is roughly that trusting additional companies increases the opportunities for any of these companies to let you down. Additionally, there is the practical aspect that one company's line of products tend to be more compatible and better tested when used together.

Being Trustworthy

Finally, don't forget the flip side of making trust decisions, which is to *promote* trust when you offer products and services. Every software product must convince end users that

it's trustworthy. Often, just presenting a solid professional image is all it takes, but if the product is fulfilling critical functions, it's crucial to give customers a solid basis for that trust.

Here are some suggestions of basic ways to enhance trust in your work:

- Transparency engenders trust. Working openly allows customers to assess the product.
- Involving a third party builds trust through their independence (for example, using hired auditors).
- Sometimes your product is the third party that integrates with other products. Trust grows because it's difficult for two parties with an arm's-length relationship to collude.
- When problems do arise, be open to feedback, act decisively, and publicly disclose the results of any investigation and steps taken to prevent recurrences.
- Specific features or design elements can make trust visible—for example, an archive solution that shows in real time how many backups have been saved and verified at distributed locations.

Actions beget trust, while empty claims, if anything, erode trust for savvy customers. Provide tangible evidence of being trustworthy, ideally in a way that customers can potentially

verify for themselves. Even though few will actually vet the quality of open source code, knowing that they could (and assuming others likely are doing so) is nearly as convincing.

Classic Principles

The guiding principles of information security originated in the early days of computing, when computers were emerging from special locked, air-conditioned, raised-floor rooms and starting to be connected in networks. These traditional models are the “Newtonian physics” of modern information security: a good and simple guide for many applications, but not the be-all and end-all. For example, information privacy is one of the more nuanced considerations for modern data protection and stewardship that traditional information security principles do not cover.

The foundational principles group nicely into two sets of three. The first three principles, which I will call *C-I-A*, define data access requirements; the other three, in turn, concern how access is controlled and monitored. We call these the *Gold Standard*. The two sets of principles are interdependent, and only as a whole do they protect data assets.

Beyond the prevention of unauthorized data access lies the question of who or what components and systems should be

entrusted with access. This is a harder question of trust, and ultimately beyond the scope of information security, even though confronting it is unavoidable in order to secure any digital system.

Information Security's C-I-A

We traditionally build software security on three basic principles of information security: *confidentiality*, *integrity*, and *availability*. Formulated around the fundamentals of data protection, the individual meanings of the three pillars are intuitive:

Confidentiality

Allow only authorized data access—don't leak information.

Integrity

Maintain data accurately—don't allow unauthorized modification or deletion.

Availability

Preserve the availability of data—don't allow significant delays or unauthorized shutdowns.

Each of these brief definitions describes the goal and defenses against its subversion. In reviewing designs, it's often helpful to think of ways one might undermine security, and work back to defensive measures.

All three components of C-I-A represent ideals, and it's crucial to avoid insisting on perfection. For example, an analysis of even solidly encrypted network traffic could allow a determined eavesdropper to deduce something about the communications between two endpoints, like the volume of data exchanged. Technically, this exchange of data weakens the confidentiality of interaction between the endpoints; but for practical purposes, we can't fix it without taking extreme measures, and usually the risk is minor enough to be safely ignored. (One way to conceal the fact of communication is for endpoints to always exchange a constant volume of data, adding dummy packets as needed when actual traffic is lower.) What activity corresponds to the traffic, and how might an adversary use that knowledge? The next chapter explains similar threat assessments in detail.

Notice that authorization is inherent in each component of C-I-A, which mandates only the right disclosures, modifications of data, or controls of availability. What constitutes "right" is an important detail, and an authorization policy needs to specify that, but it isn't part of these fundamental data protection

primitive concepts. That part of the story will be discussed in “The Gold Standard” starting on page 14.

Confidentiality

Maintaining confidentiality means disclosing private information in only an authorized manner. This sounds simple, but in practice it involves a number of complexities.

First, it’s important to carefully identify what information to consider private. Design documents should make this distinction clear. While what counts as sensitive might sometimes seem obvious, it’s actually surprising how people’s opinions vary, and without an explicit specification, we risk misunderstanding. The safest assumption is to treat all externally collected information as private by default, until declared otherwise by an explicit policy that explains how and why the designation can be relaxed.

Here are some oft-overlooked reasons to treat data as private:

- An end user might naturally expect their data to be private, unless informed otherwise, even if revealing it isn’t harmful.
- People might enter sensitive information into a text field intended for a different use.

- Information collection, handling, and storage might be subject to laws and regulations that many are unaware of. (For example, if Europeans browse your website, it may be subject to EU law, such as the General Data Protection Regulation.)

When handling private information, determine what constitutes proper access. Deciding when and how to disclose information is ultimately a trust decision, and it's worth not only spelling out the rules, but also explaining the subjective choices behind those rules.

Compromises of confidentiality happen on a spectrum. In a complete disclosure, attackers acquire an entire dataset, including metadata. At the lower end of the spectrum might be a minor disclosure of information, such as an internal error message or similar leak of no real consequence. As an example of a partial disclosure, consider the practice of assigning sequential numbers to new customers: a wily competitor can sign up as a new customer and get a new customer number from time to time, then compute the successive differences to learn the numbers of customers acquired during each interval. Any leakage of details about protected data is to some degree a confidentiality compromise.

It's so easy to underestimate the potential value of minor disclosures. Attackers might put data to use in a completely different way than the developers originally intended, and combining tiny bits of information can provide more powerful insights than any of the individual parts on their own. Learning someone's ZIP code might not tell you much, but if you also know their approximate age and that they're an MD, you could perhaps combine this information to identify the individual in a sparsely populated area—a process known as *deanonymization* or *reidentification*. By analyzing a supposedly anonymized dataset published by Netflix, researchers were able to match numerous user accounts to IMDb accounts: it turns out that your favorite movies are an effective means of unique personal identification.

Integrity

Integrity, used in an information security context, is simply the authenticity and accuracy of data, kept safe from unauthorized tampering or removal. In addition to protecting against unauthorized modification, an accurate record of the *provenance* of data—the original source, and any authorized changes made—can be an important, and stronger, assurance of integrity.

One classic defense against many tampering attacks is to preserve versions of critical data and record their provenance.

Simply put, keep good backups. Incremental backups can be excellent mitigations because they're simple and efficient to put in place and provide a series of snapshots that detail exactly what data changed, and when. However, the need for integrity goes far beyond the protection of data, and often includes ensuring the integrity of components, server logs, software source code and versions, and other forensic information necessary to determine the original source of tampering when problems occur. In addition to limited administrative access controls, secure digests (similar to checksums) and digital signatures are also strong integrity checks, as explained in Chapter 5.

Bear in mind that tampering can happen in many different ways, not necessarily by modifying data in storage. For instance, in a web application, tampering might happen on the client side, on the wire between the client and server, by tricking an authorized party into making a change, by modifying a script on the page, or in many other ways.

Availability

Attacks on availability are a sad reality of the internet-connected world and can be among the most difficult to defend against. In the simplest cases, the attacker may just send an exceptionally heavy load of traffic to the server, overwhelming it with what looks like valid uses of the service.

This principle implies that information is *temporarily* unavailable; while data that is permanently lost is also unavailable, this is generally considered to be fundamentally a compromise of integrity.

Anonymous denial-of-service (DoS) attacks, often for ransom, threaten any internet service, posing a difficult challenge. To best defend against these attacks, host on large-scale services with infrastructure that stands up to heavy loads, and maintain the flexibility to move infrastructure quickly in the event of problems. Nobody knows how common or costly DoS attacks really are, since many victims resolve these incidents privately. But without a doubt, you should create detailed plans in advance to prepare for such incidents.

Many other kinds of availability threats are possible as well. For a web server, a malformed request that triggers a bug, causing a crash or infinite loop, can devastate its service. Other attacks can also overload the storage, computation, or communication capacity of an application, or perhaps use patterns that break the effectiveness of caching, all of which pose serious issues. Unauthorized destruction of software, configuration, or data (even with backup, delays can result) also can adversely impact availability.

The Gold Standard

If C-I-A is the goal of secure systems, the Gold Standard describes the means to that end. *Aurum* is Latin for gold, hence the chemical symbol “Au,” and it just so happens that the three important principles of security enforcement start with those same two letters:

Authentication

High-assurance determination of the identity of a principal

Authorization

Reliably only allowing an action by an authenticated principal

Auditing

Maintaining a reliable record of actions by principals for inspection

NOTE

Jargon alert: because the words are so long and similar, you may encounter the handy abbreviations authN (for authentication) and authZ (for authorization) as short forms that plainly distinguish them.

A *principal* is any reliably authenticated entity: a person, business or organization, government entity, application, service, device, or any other agent with the power to act.

Authentication is the process of reliably establishing the validity of the principal's credentials. Systems commonly allow registered users to authenticate by proving that they know the password associated with their user account, but authentication can be much broader. Credentials may be something the principal knows (a password) or possesses (a smart card), or something they are (biometric data); we'll talk more about credentials in the next section.

Data access for authenticated principals is subject to *authorization* decisions, either allowing or denying their actions according to prescribed rules. For example, filesystems

with access control settings may make certain files read-only for specific users. In a banking system, clerks may record transactions up to a certain amount, but might require a manager to approve larger transactions.

If a service keeps a secure log that accurately records what principals do, including any failed attempts at performing some action, the administrators can perform a subsequent *audit* to inspect how the system performed and ensure that all actions are proper. Accurate audit logs are an important component of strong security, because they provide a reliable report of actual events. Detailed logs provide a record of what happened, shedding light on exactly what transpired when an unusual or suspicious event takes place. For example, if you discover that an important file is gone, the log should ideally provide details of who deleted it and when, providing a starting point for further investigation.

The Gold Standard acts as the enforcement mechanism that protects C-I-A. We defined confidentiality and integrity as protection against *unauthorized* disclosure or tampering, and availability is also subject to control by an authorized administrator. The only way to truly enforce authorization decisions is if the principals using the system are properly authenticated. Auditing completes the picture by providing a reliable log of who did what and when, subject to regular

review for irregularities, and holding the acting parties responsible.

Secure designs should always explicitly separate authentication from authorization, because combining them leads to confusion, and audit trails are clearer when these stages are cleanly divided. These two real-world examples illustrate why the separation is important:

- “Why did you let that guy into the vault?” “I have no idea, but he looked legit!”
- “Why did you let that guy into the vault?” “His ID was valid for ‘Sam Smith’ and he had a written note from the branch manager.”

The second response is much more complete than the first, which is of no help at all, other than proving that the guard is a nitwit. If the vault was compromised, the second response would give clear details to investigate: Did the branch manager have authority to grant vault access and write the note? If the guard retained a copy of the ID, then that information helps identify and find Sam Smith. By contrast, if the branch manager’s note had just said, “let the bearer into the vault”—authorization without authentication—investigators would have had little idea what happened or who the intruder was after security was breached.

Authentication

An authentication process tests a principal's claims of identity based on credentials that demonstrate they really are who they claim to be. Or the service might use a stronger form of credentials, such as a digital signature or a challenge, which proves that the principal possesses a private key associated with the identity, which is how browsers authenticate web servers via HTTPS. The digital signature is a better form of authentication because the principal can prove they know the secret without divulging it.

Evidence suitable for authentication falls into the following categories:

- *Something you know*, like a password
- *Something you have*, like a secure token, or in the analog world some kind of certificate, passport, or signed document that is unforgeable
- *Something you are*—that is, biometrics (fingerprint, iris pattern, and such)
- *Somewhere you are*—your verified location, such as a connection to a private network in a secure facility

Many of these methods are quite fallible. Something you know can be revealed, something you have can be stolen or copied, your location can be manipulated in various ways, and even

something you are can potentially be faked (and if it's compromised, you can't later change what you are). On top of those concerns, in today's networked world, authentication almost always happens across a network, making the task more difficult than in-person authentication. On the web, for instance, the browser serves as a trust intermediary, locally authenticating and, only if successful, then passing along cryptographic credentials to the server. Systems commonly use multiple authentication factors to mitigate these concerns, and auditing these frequently is another important backstop. Two weak authentication factors are better than one (but not a lot better).

Before an organization can assign someone credentials, however, it has to address the gnarly question of how to determine a person's true identity when they join a company, sign up for an account, or call the helpdesk to reinstate access after forgetting their password.

For example, when I joined Google, all of us new employees gathered on a Monday morning opposite several IT admin folks, who checked our passports or other ID against a new employee roster. Only then did they give us our badges and company-issued laptops and have us establish our login passwords.

By checking whether the credentials we provided (our IDs) correctly identified us as the people we purported to be, the IT team confirmed our identities. The security of this identification depended on the integrity of the government-issued IDs and supporting documents (for example, birth certificates) we provided. How accurately were those issued? How difficult would they be to forge, or obtain fraudulently? Ideally, a chain of association from registration at birth would remain intact throughout our lifetimes to uniquely identify each of us authentically. Securely identifying people is challenging largely because the most effective techniques reek of authoritarianism and are socially unacceptable, so to preserve some privacy and freedom, we opt for weaker methods in daily life. The issue of how to determine a person's true identity is out of scope for this book, which will focus on the Gold Standard, not this harder problem of *identity management*.

Whenever feasible, rely on existing trustworthy authentication services, and do not reinvent the wheel unnecessarily. Even simple password authentication is quite difficult to do securely, and dealing securely with forgotten passwords is even harder. Generally speaking, the authentication process should examine credentials and provide either a pass or fail response. Avoid indicating partial success, since this could aid an attacker zeroing in on the credentials by trial and error. To

mitigate the threat of brute-force guessing, a common strategy is to make authentication inherently computationally heavyweight, or to introduce increasing delay into the process (also see “Avoid Predictability” on page 61 in Chapter 4).

After authenticating the user, the system must find a way to securely bind the identity to the principal. Typically, an authentication module issues a token to the principal that they can use in lieu of full authentication for subsequent requests. The idea is that the principal, via an agent such as a web browser, presents the authentication token as shorthand assurance of who they claim to be, creating a *secure context* for future requests. This context binds the stored token for presentation with future requests on behalf of the authenticated principal. Websites often do this with a secure cookie associated with the browsing session, but there are many different techniques for other kinds of principals and interfaces.

The secure binding of an authenticated identity can be compromised in two fundamentally different ways. The obvious one is where an attacker usurps the victim’s identity. Alternatively, the authenticated principal may collude and try to give away their identity or even foist it off on someone else. An example of the latter case is the sharing of a paid streaming subscription. The web does not afford very good ways of

defending against this because the binding is loose and depends on the cooperation of the principal.

Authorization

A decision to allow or deny critical actions should be based on the identity of the principal as established by authentication. Systems implement authorization in business logic, an access control list, or some other formal access policy.

Anonymous authorization (that is, authorization without authentication) can be useful in rare circumstances; a real-world example might be possession of the key to a public locker in a busy station. Access restrictions based on time (for example, database access restricted to business hours) are another common example.

A single guard should enforce authorization on a given resource. Authorization code scattered throughout a codebase is a nightmare to maintain and audit. Instead, authorization should rely on a common framework that grants access uniformly. A well-structured design can help the developers get it right. Use one of the many standard authorization models rather than confusing ad hoc logic wherever possible.

Role-based access control (RBAC) bridges the connection between authentication and authorization. RBAC grants access

based on roles assigned to authenticated principals, simplifying access control with a uniform framework. For example, roles in a bank might include a clerk, manager, loan officer, security guard, financial auditor, and IT administrator. Instead of choosing access privileges for each person individually, RBAC designates one or more roles based on each person's responsibilities to automatically and uniformly assign them associated privileges. In more advanced models, one person might have multiple roles and explicitly select which role they choose to apply for a given access.

Authorization mechanisms can be much more granular than the simple read/write access control that operating systems traditionally provide. By designing more robust authorization mechanisms, you can strengthen security by limiting access without losing useful functionality. These more advanced authorization models include *attribute-based access control (ABAC)*, *policy-based access control (PBAC)*, and many more.

Consider a simple bank teller example to see how fine-grained authorization might tighten up policy:

Rate-limited

> Tellers may do up to 20 transactions per hour, but more would be considered suspicious.

Time of day

Teller transactions must occur during business hours, when clocked in.

No self-service

Tellers are forbidden to do transactions with their personal accounts.

Multiple principals

Teller transactions over $10,000 require separate manager approval (eliminating the risk of one bad actor moving a lot of money at once).

Finally, even read-only access may be too high a level for certain data, like passwords. Systems usually check login passwords by comparing digests, which avoids any possibility of leaking the actual plaintext password. The username and password go to a frontend server that computes the digest of the password and passes it to an authentication service, quickly destroying any trace of the plaintext password. The authentication service cannot read the plaintext password from the credentials database, but it can read the digest, which it compares to what the frontend server provided. In this way, it checks the credentials, but the authentication service never

has access to any passwords, so even if compromised, the service cannot leak them. Unless the design of interfaces affords these alternatives, they will miss these opportunities to mitigate the possibility of data leakage. We'll explore this further when we discuss the pattern of "Least Information" on page 57 in Chapter 4.

Auditing

In order for an organization to audit system activity, the system must produce a reliable log of all events that are critical to maintaining security. These include authentication and authorization events, system startup and shutdown, software updates, administrative accesses, and so forth. Audit logs must also be tamper-resistant, and ideally even difficult for administrators to meddle with, to be considered fully reliable records. Auditing is a critical leg of the Gold Standard, because incidents do happen, and authentication and authorization policies can be flawed. Auditing can also provide necessary oversight to mitigate the risk of inside jobs in which authorized principals betray their trust.

If done properly, audit logs are essential for routine monitoring, measuring system activity level, detecting errors and suspicious activity, and, after an incident, determining when and how an attack actually happened and gauging the extent of the damage. Remember that completely protecting a

digital system is not simply a matter of correctly enforcing policies; it's about being a responsible steward of information assets. Auditing ensures that trusted principals acted properly within the broad range of their authority.

In May 2018, Twitter disclosed an embarrassing bug: they had discovered that a code change had inadvertently caused raw login passwords to appear in internal logs. It's unlikely that this resulted in any abuse, but it certainly hurt customer confidence and should never have happened. Logs should record operational details but not store any actual private information so as to minimize the risk of disclosure, since many members of the technical staff may routinely view the logs. For a detailed treatment of this requirement, see the sample design document in Appendix A detailing a logging tool that addresses just this problem.

The system must also prevent anyone from tampering with the logs to conceal bad acts. If the attacker can modify logs, they'll just clean out all traces of their activity. For especially sensitive logs at high risk, an independent system under different administrative and operational controls should manage audit logs in order to prevent the perpetrators of inside jobs from covering their own tracks. This is difficult to do completely, but the mere presence of independent oversight often serves as a powerful disincentive to any funny business, just as a modest

fence and conspicuous video surveillance camera can be an effective deterrent to trespassing.

Furthermore, any attempt to circumvent the system would seem highly suspicious, and any false move would result in serious repercussions for the offender. Once caught, they would have a hard time repudiating their guilt.

Non-repudiability is an important property of audit logs; if the log shows that a named administrator ran a certain command at a certain time and the system crashed immediately, it's hard to point fingers at others. By contrast, if an organization allowed multiple administrators to share the same account (a terrible idea), it would have no way of definitively knowing who actually did anything, providing plausible deniability to all.

Ultimately, audit logs are useful only if you monitor them, analyze unusual events carefully, and follow up, taking appropriate actions when necessary. To this end, it's important to log the right amount of detail by following the *Goldilocks principle*. Too much logging bloats the volume of data to oversee, and excessively noisy or disorganized logs make it difficult to glean useful information. On the other hand, sparse logging with insufficient detail might omit critical information, so finding the right balance is an ongoing challenge.

Privacy

In addition to the foundations of information security—C-I-A and the Gold Standard—another fundamental topic I want to introduce is the related field of information privacy. The boundaries between security and privacy are difficult to clearly define, and they are at once closely related and quite different. In this book I would like to focus on the common points of intersection, not to attempt to unify them, but to incorporate both security and privacy into the process of building software.

To respect people's digital information privacy, we must extend the principle of confidentiality by taking into account additional human factors, including:

- Customer expectations regarding information collection and use
- Clear policies regarding appropriate information use and disclosure
- Legal and regulatory issues relating to the collection and use of various classes of information
- Political, cultural, and psychological aspects of processing personal information

As software becomes more pervasive in modern life, people use it in more intimate ways involving sensitive areas of their

lives, resulting in many complex issues. Past accidents and abuses have raised the visibility of the risks, and as society grapples with new challenges through political and legal means, handling private information properly has become challenging.

In the context of software security, this means:

- Considering the customer and stakeholder consequences of all data collection and sharing
- Flagging all potential issues, and getting expert advice where necessary
- Establishing and following clear policies and guidelines regarding private information use
- Translating policy and guidance into software-enforced checks and balances
- Maintaining accurate records of data acquisition, use, sharing, and deletion
- Auditing data access authorizations and extraordinary access for compliance

Privacy work tends to be less well-defined than the relatively cut-and-dried security work of maintaining proper control of systems and providing appropriate access. Also, we're still working out privacy expectations and norms as society ventures deeper into a future with more data collection. Given

these challenges, you would be wise to consider maximal transparency about data use, including keeping your policies simple enough to be understood by all, and to collect minimal data, especially personally identifiable information.

Collect information for a specific purpose only, and retain it only as long as it's useful. Unless the design envisions an authorized use, avoid collection in the first place. Frivolously collecting data for use "someday" is risky, and almost never a good idea. When the last authorized use of some data becomes unnecessary, the best protection is secure deletion. For especially sensitive data, or for maximal privacy protection, make that even stronger: delete data when the potential risk of disclosure exceeds the potential value of retaining it. Retaining many years' worth of emails might occasionally be handy for something, but probably not for any clear business need. Yet internal emails could represent a liability if leaked or disclosed, such as by power of subpoena. Rather than hang onto all that data indefinitely "just in case," the best policy is usually to delete it.

A complete treatment of information privacy is outside the scope of this book, but privacy and security are tightly bound facets of the design of any system that collects data about people—and people interact with almost all digital systems, in one way or another. Strong privacy protection is only possible

when security is solid, so these words are an appeal for awareness to consider and incorporate privacy considerations into software by design.

For all its complexity, one best practice for privacy is well known: the necessity of clearly communicating privacy expectations. In contrast to security, a privacy policy potentially affords a lot of leeway as to how much an information service does or does not want to leverage the use of customer data. “We will reuse and sell your data” is one extreme of the privacy spectrum, but “some days we may not protect your data” is not a viable stance on security. Privacy failures arise when user expectations are out of joint with actual privacy policy, or when there is a clear policy and it is somehow violated. The former problem stems from not proactively explaining data handling to the user. The latter happens when the policy is unclear, or ignored by responsible staff, or subverted in a security breakdown.

NOTE

See Appendix D for a cheat sheet summarizing the C-I-A and Gold Standard principles.
