CHAPTER

2

Strategies for Threat Modeling

The earlier you find problems, the easier it is to fix them. Threat modeling is all about finding problems, and therefore it should be done early in your development or design process, or in preparing to roll out an operational system. There are many ways to threat model. Some ways are very specific, like a model airplane kit that can only be used to build an F-14 fighter jet. Other methods are more versatile, like Lego building blocks that can be used to make a variety of things. Some threat modeling methods don't combine easily, in the same way that Erector set pieces and Lego set blocks don't fit together. This chapter covers the various strategies and methods that have been brought to bear on threat modeling, presents each one in depth, and sets the stage for effectively finding threats.

You'll start with very simple methods such as asking "what's your threat model?" and brainstorming about threats. Those can work for a security expert, and they may work for you. From there, you'll learn about three strategies for threat modeling: focusing on assets, focusing on attackers, and focusing on software. These strategies are more structured, and can work for people with different skillsets. A focus on software is usually the most appropriate strategy. The desire to focus on assets or attackers is natural, and often presented as an unavoidable or essential aspect of threat modeling. It would be wrong not to present each in its best light before discussing issues with those strategies. From there, you'll learn about different types of diagrams you can use to model your system or software.

NOTE **This chapter doesn't include the specific threat building blocks that discover threats, which are the subject of the next few chapters.**

"What's Your Threat Model?"

The question "what's your threat model?" is a great one because in just four words, it can slice through many conundrums to determine what you are worried about. Answers are often of the form "an attacker with the laptop" or " insiders," or (unfortunately, often) "huh?" The "huh?" answer is useful because it reveals how much work would be needed to find a consistent and structured approach to defense. Consistency and structure are important because they help you invest in defenses that will stymie attackers. There's a compendium of standard answers to "what's your threat model?" in Appendix A, "Helpful Tools," but a few examples are listed here as well:

- A thief who could steal your money
- The company stakeholders (employees, consultants, shareholders, etc.) who access sensitive documents and are not trusted
- An untrusted network
- An attacker who could steal your cookie (web or otherwise)

NOTE **Throughout this book, you'll visit and revisit the same example for each of these approaches. Your main targets are the fictitious Acme Corporation's "Acme/SQL," which is a commercial database server, and Acme's operational network. Using Acme examples, you can see how the different approaches play out against the same systems.**

Applying the question "what's your threat model?" to the Acme Corporation example, you might get the following answers:

- For the Acme SQL database, the threat model would be an attacker who wants to read or change data in the database. A more subtle model might also include people who want to read the data without showing up in the logs.
- For Acme's financial system, the answers might include someone getting a check they didn't deserve, customers who don't make a payment they owe, and/or someone reading or altering financial results before reporting.

If you don't have a clear answer to the question, "what's your threat model?" it can lead to inconsistency and wasted effort. For example, start-up Zero-Knowledge

Systems didn't have a clear answer to the question "what's your threat model?" Because there was no clear answer, there wasn't consistency in what security features were built. A great deal of energy went into building defenses against the most complex attacks, and these choices to defend against such attackers had performance impacts on the whole system. While preventing governments from spying on customers was a fun technical challenge and an emotionally resonant goal, both the challenge and the emotional impact made it hard to make technical decisions that could have made the business more successful. Eventually, a clearer answer to "what's your threat model?" let Zero-Knowledge Systems invest in mitigations that all addressed the same subset of possible threats.

So how do you ensure you have a clear answer to this question? Often, the answers are not obvious, even to those who think regularly about security, and the question itself offers little structure for figuring out the answers. One approach, often recommended is to brainstorm. In the next section, you'll learn about a variety of approaches to brainstorming and the tradeoffs associated with those approaches.

Brainstorming Your Threats

Brainstorming is the most traditional way to enumerate threats. You get a set of experienced experts in a room, give them a way to take notes (whiteboards or cocktail napkins are traditional) and let them go. The quality of the brainstorm is bounded by the experience of the brainstormers and the amount of time spent brainstorming.

Brainstorming involves a period of idea-generation, followed by a period of analyzing and selecting the ideas. Brainstorming for threat modeling involves coming up with possible attacks of all sorts. During the idea generation phase, you should forbid criticism. You want to explore the space of possible threats, and an atmosphere of criticism will inhibit such idea generation. A moderator can help keep brainstorming moving.

During brainstorming, it is key to have an expert on the technology being modeled in the room. Otherwise, it's easy to make bad assumptions about how it works. However, when you have an expert who's proud of their technology, you need to ensure that you don't end up with a "proud parent" offended that their software baby is being called ugly. A helpful rule is that it's the software being attacked, not the software architects. That doesn't always suffice, but it's a good start. There's also a benefit to bringing together a diverse grouping of experts with a broader set of experience.

Brainstorming can also devolve into out-of-scope attacks. For example, if you're designing a chat program, attacks by the memory management unit against the CPU are probably out of scope, but if you're designing a motherboard,

these attacks may be the focus of your threat modeling. One way to handle this issue is to list a set of attacks that are out of scope, such as "the administrator is malicious" or "an attacker edits the hard drive on another system," as well as a set of attack equivalencies, like, "an attacker can smash the stack and execute code," so that those issues can be acknowledged and handled in a consistent way. A variant of brainstorming is the exhortation to "think like an attacker," which is discussed in more detail in Chapter 18, "Experimental Approaches."

Some attacks you might brainstorm in threat modeling the Acme's financial statements include breaking in over the Internet, getting the CFO drunk, bribing a janitor, or predicting the URL where the financials will be published. These can be a bit of a grab bag, so the next section provides somewhat more focused approaches.

Brainstorming Variants

Free-form or "normal" brainstorming, as discussed in the preceding section, can be used as a method for threat modeling, but there are more specific methods you can use to help focus your brainstorming. The following sections describe variations on classic brainstorming: scenario analyses, pre-mortems, and movie-plotting.

Scenario Analysis

It may help to focus your brainstorming with scenarios. If you're using written scenarios in your overall engineering, you might start from those and ask what might go wrong, or you could use a variant of Chandler's law ("When in doubt, have a man come through a door with a gun in his hand.") You don't need to restrict yourself to a man with a gun, of course; you can use any of the attackers listed in Appendix C, "Attacker Lists."

For an example of scenario-specific brainstorming, try to threat model for handing your phone to a cute person in a bar. It's an interesting exercise. The recipient could perhaps text donations to the Red Cross, text an important person to "stop bothering me," or post to Facebook that "I don't take hints well" or "I'm skeevy," not to mention possibilities of running away with the phone or dropping it in a beer.

Less frivolously, your sample scenarios might be based on the product scenarios or use cases for the current development cycle, and therefore cover failover and replication, and how those services could be exploited when not properly authenticated and authorized.

Pre-Mortem

Decision-sciences expert Gary Klein has suggested another brainstorming technique he calls the *pre-mortem* (Klein, 1999). The idea is to gather those involved

in a decision and ask them to assume that it's shortly after a project deadline, or after a key milestone, and that things have gone totally off the rails. With an "assumption of failure" in mind, the idea is to explore why the participants believe it will go off the rails. The value to calling this a pre-mortem is the framing it brings. The natural optimism that accompanies a project is replaced with an explicit assumption that it has failed, giving you and other participants a chance to express doubts. In threat modeling, the assumption is that the product is being successfully attacked, and you now have permission to express doubts or concerns.

Movie Plotting

Another variant of brainstorming is movie plotting. The key difference between "normal brainstorming" and "movie plotting" is that the attack ideas are intended to be outrageous and provocative to encourage the free flow of ideas. Defending against these threats likely involves science-fiction-type devices that impinge on human dignity, liberty, and privacy without actually defending anyone. Examples of great movies for movie plot threats include *Ocean's Eleven*, *The Italian Job*, and every Bond movie that doesn't stink. If you'd like to engage in more structured movie plotting, create three lists: flawed protagonists, brilliant antagonists, and whiz-bang gadgetry. You can then combine them as you see fit.

Examples of movie plot threats include a foreign spy writing code for Acme SQL so that a fourth connection attempt lets someone in as admin, a scheming CFO stealing from the firm, and someone rappelling from the ceiling to avoid the pressure mats in the floor while hacking into the database from the console. Note that these movie plots are equally applicable to Acme and its customers.

The term movie plotting was coined by Bruce Schneier, a respected security expert. Announcing his contest to elicit movie plot threats, he said: "The purpose of this contest is absurd humor, but I hope it also makes a point. Terrorism is a real threat, but we're not any safer through security measures that require us to correctly guess what the terrorists are going to do next" (Schneier, 2006). The point doesn't apply only to terrorism; convoluted but vividly described threats can be a threat to your threat modeling methodology.

Literature Review

As a precursor to brainstorming (or any other approach to finding threats), reviewing threats to systems similar to yours is a helpful starting point in threat modeling. You can do this using search engines, or by checking the academic literature and following citations. It can be incredibly helpful to search on competitors or related products. To start, search on a competitor, appending terms such as "security," "security bug," "penetration test," "pwning," or "Black Hat," and use your creativity. You can also review common threats in this book,

especially Part III, "Managing and Addressing Threats" and the appendixes. Additionally, Ross Anderson's *Security Engineering* is a great collection of real world attacks and engineering lessons you can draw on, especially if what you're building is similar to what he covers (Wiley, 2008).

A *literature review* of threats against databases might lead to an understanding of SQL injection attacks, backup failures, and insider attacks, suggesting the need for logs. Doing a review is especially helpful for those developing their skills in threat modeling. Be aware that a lot of the threats that may come up can be super-specific. Treat them as examples of more general cases, and look for variations and related problems as you brainstorm.

Perspective on Brainstorming

Brainstorming and its variants suffer from a variety of problems. Brainstorming often produces threats that are hard or impossible to address. Brainstorming intentionally requires the removal of scoping or boundaries, and the threats are very dependent on the participants and how the session happens to progress. When experts get together, unstructured discussion often ensues. This can be fun for the experts and it usually produces interesting results, but oftentimes, experts are in short supply. Other times, engineers get frustrated with the inconsistency of "ask two experts, get three answers."

There's one other issue to consider, and that relates to exit criteria. It's difficult to know when you're done brainstorming, and whether you're done because you have done a good job or if everyone is just tired. Engineering management may demand a time estimate that they can insert into their schedule, and these are difficult to predict. The best approach to avoid this timing issue is simply to set a meeting of defined length. Unfortunately, this option doesn't provide a high degree of confidence that all interesting threats have been found.

Because of the difficulty of addressing threats illuminated with a limitless brainstorming technique and the poorly defined exit criteria to a brainstorming session, it is important to consider other approaches to threat modeling that are more prescriptive, formal, repeatable, or less dependent on the aptitudes and knowledge of the participants. Such approaches are the subject of the rest of this chapter and also discussed in the rest of Part II, "Finding Threats."

Structured Approaches to Threat Modeling

When it's hard to answer "What's your threat model?" people often use an approach centered on models of their assets, models of attackers, or models of their software. Centering on one of those is preferable to using approaches that attempt to combine them because these combinations tend to be confusing.

Assets are the valuable things you have. The people who might go after your assets are attackers, and the most common way for them to attack is via the software you're building or deploying.

Each of these is a natural place to start thinking about threats, and each has advantages and disadvantages, which are covered in this section. There are people with very strong opinions that one of these is right (or wrong). Don't worry about "right" or "wrong," but rather "usefulness." That is, does your approach help you find problems? If it doesn't, it's wrong for you, however forcefully someone might argue its merits.

These three approaches can be thought of as analogous to Lincoln Log sets, Erector sets, and Lego sets. Each has a variety of pieces, and each enables you to build things, but they may not combine in ways as arbitrary as you'd like. That is, you can't snap Lego blocks to an Erector set model. Similarly, you can't always snap attackers onto a software model and have something that works as a coherent whole.

To understand these three approaches, it can be useful to apply them to something concrete. Figure 2-1 shows a data flow diagram of the Acme/SQL system.

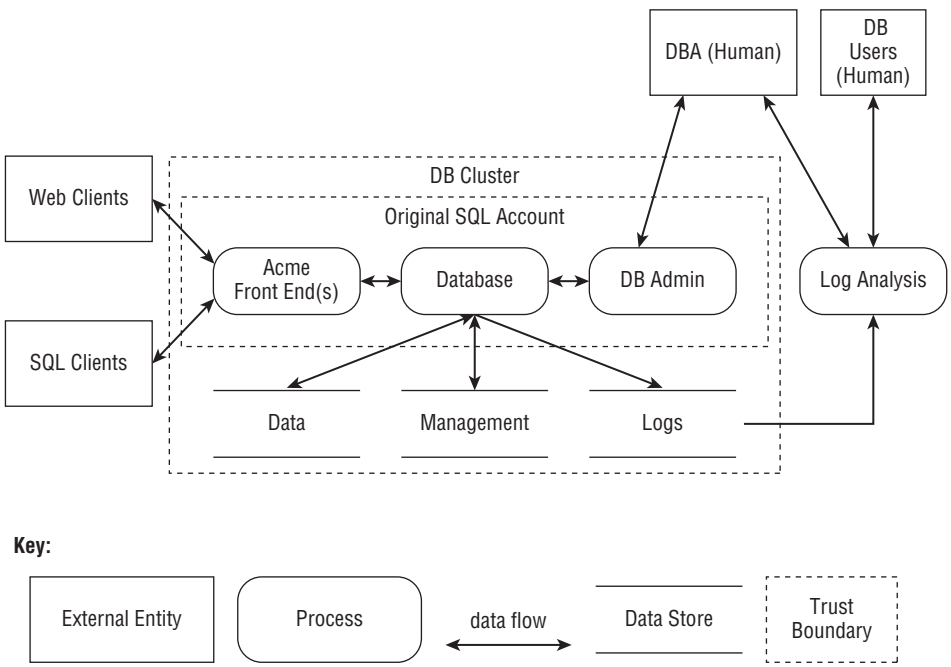

Figure 2-1: Data flow diagram of the Acme/SQL database

Looking at the diagram, and reading from left to right, you can see two types of clients accessing the front ends and the core database, which manages

transactions, access control, atomicity, and so on. Here, assume that the Acme/SQL system comes with an integrated web server, and that authorized clients are given nearly raw access to the data. There could simultaneously be web servers offering deeper business logic, access to multiple back ends, integration with payment systems, and so on. Those web servers would access Acme/SQL via the SQL protocol over a network connection.

Back to the diagram, Figure 2-1 also shows there is also a set of DB Admin tools that the DBA (the human database administrator) uses to manage the system. As shown in the diagram, there are three conceptual data stores: Data, Management (including metadata such as locks, policies, and indices), and Logs. These might be implemented in memory, as files, as custom stores on raw disk, or delegated to network storage. As you dig in, the details matter greatly, but you usually start modeling from the conceptual level, as shown.

Finally, there's a log analysis package. Note that only the database core has direct access to the data and management information in this design. You should also note that most of the arrows are two-way, except Database ➪ Logs and Logs ➪ Log Analysis. Of course, the Log Analysis process will be querying the logs, but because it's intended as a read-only interface, it is represented as a one-way arrow. Very occasionally, you might have strictly one-way flows, such as those implemented by SNMP traps or syslog. Some threat modelers prefer two one-way arrows, which can help you see threats in each direction, but also lead to busy diagrams that are hard to label or read. If your diagrams are simple, the pair of one way arrows helps you find threats, and is therefore better than two-way. If your diagram is complex, either approach can be used.

The data flow diagram is discussed in more detail later in this chapter, in the section "Data Flow Diagrams." In the next few sections, you'll see how to apply asset, attacker, and software-centric models to find threats against Acme/SQL.

Focusing on Assets

It seems very natural to center your approach on assets, or things of value. After all, if a thing has no value, why worry about how someone might attack it? It turns out that focusing on assets is less useful than you may hope, and is therefore not the best approach to threat modeling. However, there are a small number of people who will benefit from asset-centered threat modeling. The most likely to benefit are a team of security experts with experience structuring their thinking around assets. (Having found a way that works for them, there may be no reason to change.) Less technical people may be able to contribute to threat modeling by saying "focus on this asset." If you are in either of these groups, or work with them, congratulations! This section is for you. If you aren't one of those people, however, don't be too quick to skip ahead. It is still important

to have a good understanding of the role assets can play in threat modeling, even if they're not in a starring role). It can also help for you to understand why the approach is not as useful as it may appear, so you can have an informed discussion with those advocating it.

The term asset usually refers to something of value. When people bring up assets as part of a threat modeling activity, they often mean something an attacker wants to access, control, or destroy. If everyone who touches the threat modeling process doesn't have a working agreement about what an asset is, your process will either get bogged down, or participants will just talk past each other.

There are three ways the term asset is commonly used in threat modeling:

- Things attackers want
- Things you want to protect
- Stepping stones to either of these

You should think of these three types of assets as families rather than categories because just as people can belong to more than one family at a time, assets can take on more than one meaning at a time. In other words, the tags that apply to assets can overlap, as shown in Figure 2-2. The most common usage of asset in discussing threat models seems to be a marriage of "things attackers want" and "things you want to protect."

Figure 2-2: The overlapping definitions of assets

NOTE **There are a few other ways in which the term asset is used by those who are threat modeling—such as a synonym for computer, or a type of computer (for example, "Targeted assets: mail server, database"). For the sake of clarity, this book only uses asset with explicit reference to one or more of the three families previously defined, and you should try to do the same.**

Things Attackers Want

Usually assets that attackers want are relatively tangible things such as "this database of customer medical data." Good examples of things attackers want include the following:

- User passwords or keys
- Social security numbers or other identifiers
- Credit card numbers
- Your confidential business data

Things You Want to Protect

There's also a family of assets you want to protect. Unlike the tangible things attackers want, many of these assets are intangibles. For example, your company's reputation or goodwill is something you want to protect. Competitors or activists may well attack your reputation by engaging in smear tactics. From a threat modeling perspective, your reputation is usually too diffuse to be able to technologically mitigate threats against it. Therefore, you want to protect it by protecting the things that matter to your customers.

As an example of something you want to protect, if you had an empty safe, you intuitively don't want someone to come along and stick their stethoscope to it. But there's nothing in it, so what's the damage? Changing the combination and letting the right folks (but only the right folks) know the new combination requires work. Therefore you want to protect this empty safe, but it would be an unlikely target for a thief. If that same safe has one million dollars in it, it would be much more likely to pique a thief's interest. The million dollars is part of the family of things you want that attackers want, too.

Stepping Stones

The final family of assets is stepping stones to other assets. For example, everything drawn in a threat model diagram is something you want to protect because it may be a stepping stone to the targets that attackers want. In some ways, the set of stepping stone assets is an attractive nuisance. For example, every computer has CPU and storage that an attacker can use. Most also have Internet connectivity, and if you're in systems management or operational security, many of the computers you worry most about will have special access to your organization's network. They're behind a firewall or have VPN access. These are stepping stones. If they are uniquely valuable stepping stones in some way, note that. In practice, it's rarely helpful to include "all our PCs" in an asset list.

NOTE **Referring back to the safe example in the previous section, the safe combination is a member of the stepping stone family. It may well be that stepping-stones and things you protect are, in practice, very similar. The list of technical elements you protect that are not members of the stepping-stone family appears fairly short.**

Implementing Asset-Centric Modeling

If you were to threat model with an asset-focused approach, you would make a list of your assets and then consider how an attacker could threaten each. From there, you'd consider how to address each threat.

After an asset list is created, you should connect each item on the list to particular computer systems or sets of systems. (If an asset includes something like "Amazon Web Services" or "Microsoft Azure," then you don't need to be able to point to the computer systems in question, you just need to understand where they are—eventually you'll need to identify the threats to those systems and determine how to mitigate them.)

The next step is to draw the systems in question, showing the assets and other components as well as interconnections, until you can tell a story about them. You can use this model to apply either an attack set like STRIDE or an attacker-centered brainstorm to understand how those assets could be attacked.

Perspective on Asset-Centric Threat Modeling

Focusing on assets appears to be a common-sense approach to threat modeling, to the point where it seems hard to argue with. Unfortunately, much of the time, a discussion of assets does *not* improve threat modeling. However, the misconception is so common that it's important to examine why it doesn't help.

There's no direct line from assets to threats, and no prescriptive set of steps. Essentially, effort put into enumerating assets is effort you're not spending finding or fixing threats. Sometimes, that involves a discussion of what's an asset, or which type of asset you're discussing. That discussion, at best, results in a list of things to look for in your software or operational model, so why not start by creating such a model? Once you have a list of assets, that list is not (ahem) a stepping stone to finding threats; you still need to apply some methodology or approach. Finally, assets may help you prioritize threats, but if that's your goal, it doesn't mean you should start with or focus on assets. Generally, such information comes out naturally when discussing impacts as you prioritize and address threats. Those topics are covered in Part III, "Managing and Addressing Threats."

How you answer the question "what are our assets?" should help focus your threat modeling. If it doesn't help, there is no point to asking the question or spending time answering it.

Focusing on Attackers

Focusing on attackers seems like a natural way to threat model. After all, if no one is going to attack your system, why would you bother defending it? And if you're worried because people will attack your systems, shouldn't you understand them? Unfortunately, like asset-centered threat modeling, attacker-centered threat modeling is less useful than you might anticipate. But there are also a small number of scenarios in which focusing on attackers can come in handy, and they're the same scenarios as assets: experts, less-technical input to your process, and prioritization. And similar to the "Focusing on Assets" section, you can also learn for yourself why this approach isn't optimal, so you can discuss the possibility with those advocating this approach.

Implementing Attacker-Centric Modeling

Security experts may be able to use various types of attacker lists to find threats against a system. When doing so, it's easy to find yourself arguing about the resources or capabilities of such an archetype, and needing to flesh them out. For example, what if your terrorist is state-sponsored, and has access to government labs? These questions make the attacker-centric approach start to resemble "*personas*," which are often used to help think about human interface issues. There's a spectrum of detail in attacker models, from simple lists to data-derived personas, and examples of each are given in Appendix C, "Attacker Lists" That appendix may help security experts and will help anyone who wants to try attacker-centric modeling and learn faster than if they have to start by creating a list.

Given a list of attackers, it's possible to use the list to provide some structure to a brainstorming approach. Some security experts use attacker lists as a way to help elicit the knowledge they've accumulated as they've become experts. Attacker-driven approaches are also likely to bring up possibilities that are human-centered. For example, when thinking about what a spy would do, it may be more natural (and fun) to think about them seducing your sysadmin or corrupting a janitor, rather than think about technical attacks. Worse, it will probably be challenging to think about what those human attacks mean for your system's security.

Where Attackers Can Help You

Talking about human threat agents can help make the threats real. That is, it's sometimes tough to understand how someone could tamper with a configuration file, or replace client software to get around security checks. Especially when dealing with management or product teams who "just want to ship,"

it's helpful to be able to explain who might attack them and why. There's real value in this, but it's not a sufficient argument for centering your approach on those threat agents; you can add that information at a later stage. (The risk associated with talking about attackers is the claim that "no one would ever do that." Attempting to humanize a risk by adding an actor can exacerbate this, especially if you add a type of actor who someone thinks "wouldn't be interested in us.").

You were promised an example, and the spies stole it. More seriously, carefully walking through the attacker lists and personas in Appendix C likely doesn't help you (or the author) figure out what they might want to do to Acme/SQL, and so the example is left empty to avoid false hope.

Perspective on Attacker-Centric Modeling

Helping security experts structure and recall information is nice, but doesn't lead to reproducible results. More importantly, attacker lists or even personas are not enough structure for most people to figure out what those people will do. Engineers may subconsciously project their own biases or approaches into what an attacker might do. Given that the attacker has his own motivations, skills, background, and perspective (and possibly organizational priorities), avoiding such projection is tricky.

In my experience, this combination of issues makes attacker-centric approaches less effective than other approaches. Therefore, I recommend against using attackers as the center of your threat modeling process.

Focusing on Software

Good news! You've officially reached the "best" structured threat modeling approach. Congrats! Read on to learn about software-centered threat modeling, why it's the most helpful and effective approach, and how to do it.

Software-centric models are models that focus on the software being built or a system being deployed. Some software projects have documented models of various sorts, such as architecture, UML diagrams, or APIs. Other projects don't bother with such documentation and instead rely on implicit models.

Having large project teams draw on a whiteboard to explain how the software fits together can be a surprisingly useful and contentious activity. Understandings differ, especially on large projects that have been running for a while, but finding where those understandings differ can be helpful in and of itself because it offers a focal point where threats are unlikely to be blocked. ("I thought *you* were validating that for a SQL call!")

The same complexity applies to any project that is larger than a few people or has been running longer than a few years. Projects accumulate complexity,

which makes many aspects of development harder, including security. Software-centric threat modeling can have a useful side effect of exposing this accumulated complexity.

The security value of this common understanding can also be substantial, even before you get to looking for threats. In one project it turned out that a library on a trust boundary had a really good threat model, but unrealistic assumptions had been made about what the components behind it were doing. The work to create a comprehensive model led to an explicit list of common assumptions that could and could not be made. The comprehensive model and resultant understanding led to a substantial improvement in the security of those components.

NOTE **As complexity grows, so will the assumptions that are made, and such lists are never complete. They grow as experience requires and feedback loops allow.**

Threat Modeling Different Types of Software

The threat discovery approaches covered in Part II, can be applied to models of all sorts of software. They can be applied to software you're building for others to download and install, as well as to software you're building into a larger operational system. The software they can be applied to is nearly endless, and is not dependent on the business model or deployment model associated with the software.

Even though software no longer comes in boxes sold on store shelves, the term *boxed software* is a convenient label for a category. That category is all the software whose architecture is definable, because there's a clear edge to what is the software: It's everything in the box (installer, application download, or open source repository). This edge can be contrasted with the deployed systems that organizations develop and change over time.

NOTE **You may be concerned that the techniques in this book focus on either boxed software or deployed systems, and that the one you're concerned about isn't covered. In the interests of space, the examples and discussion only cover both when there's a clear difference and reason. That's because the recommended ways to model software will work for both with only a few exceptions.**

The boundary between boxed software models and network models gets blurrier every year. One important difference is that the network models tend to include more of the infrastructural components such as routers, switches, and data circuits. Trust boundaries are often operationalized by these components, or by whatever group operates the network, the platforms, or the applications.

Data flow models (which you met in Chapter 1, "Dive In and Threat Model!" and which you'll learn more about in the next section) are usually a good choice for both boxed software and operational models. Some large data center operators have provided threat models to teams, showing how the data center is laid out. The product group can then overlay its models "on top" of that to align with the appropriate security controls that they'll get from operations. When you're using someone else's data center, you may have discussions about their infrastructure choices that make it easy to derive a model, or you might have to assume the worst.

Perspective on Software-Centric Modeling

I am fond of software-centric approaches because you should expect software developers to understand the software they're developing. Indeed, there is nothing else you should expect them to understand better. That makes software an ideal place to start the threat-modeling tasks in which you ask developers to participate. Almost all software development is done with software models that are good enough for the team's purposes. Sometimes they require work to make them good enough for effective threat modeling.

In contrast, you can merely hope that developers understand the business or its assets. You may aspire to them understanding the people who will attack their product or system. But these are hopes and aspirations, rather than reasonable expectations. To the extent that your threat modeling strategy depends on these hopes and aspirations, you're adding places where it can fail. The remainder of this chapter is about modeling your software in ways that help you find threats, and as such enabling software centric-modeling. (The methods for finding these threats are covered in the rest of Part II.)

Models of Software

Making an explicit model of your software helps you look for threats without getting bogged down in the many details that are required to make the software function properly. Diagrams are a natural way to model software.

As you learned in Chapter 1, whiteboard diagrams are an extremely effective way to start threat modeling, and they may be sufficient for you. However, as a system hits a certain level of complexity, drawing and redrawing on whiteboards becomes infeasible. At that point, you need to either simplify the system or bring in a computerized approach.

In this section, you'll learn about the various types of diagrams, how they can be adapted for use in threat modeling, and how to handle the complexities of larger systems. You'll also learn more detail about trust boundaries, effective labeling, and how to validate your diagrams.

Types of Diagrams

There are many ways to diagram, and different diagrams will help in different circumstances. The types of diagrams you'll encounter most frequently are probably data flow diagrams (DFDs). However, you may also see UML, swim lane diagrams, and state diagrams. You can think of these diagrams as Lego blocks, looking them over to see which best fits whatever you're building. Each diagram type here can be used with the models of threats in Part II.

The goal of all these diagrams is to communicate how the system works, so that everyone involved in threat modeling has the same understanding. If you can't agree on how to draw how the software works, then in the process of getting to agreement, you're highly likely to discover misunderstandings about the security of the system. Therefore, use the diagram type that helps you have a good conversation and develop a shared understanding.

Data Flow Diagrams

Data flow models are often ideal for threat modeling; problems tend to follow the data flow, not the control flow. Data flow models more commonly exist for network or architected systems than software products, but they can be created for either.

Data flow diagrams are used so frequently they are sometimes called "threat model diagrams." As laid out by Larry Constantine in 1967, DFDs consist of numbered elements (data stores and processes) connected by data flows, interacting with external entities (those outside the developer's or the organization's control).

The data flows that give DFDs their name almost always flow two ways, with exceptions such as radio broadcasts or UDP data sent off into the Ethernet. Despite that, flows are usually represented using one-way arrows, as the threats and their impact are generally not symmetric. That is, if data flowing to a web server is read, it might reveal passwords or other data, whereas a data flow from the web server might reveal your bank balance. This diagramming convention doesn't help clarify channel security versus message security. (The channel might be something like SMTP, with messages being e-mail messages.) Swim lane diagrams may be more appropriate as a model if this channel/message distinction is important. (Swim lane diagrams are described in the eponymous subsection later in this chapter.)

The main elements of a data flow diagram are shown in Table 2-1.

Table 2-1: Elements of a Data Flow Diagram

ELEMENT	APPEARANCE	MEANING	EXAMPLES
Process	Rounded rectangle, circle, or concentric circles	Any running code	Code written in C, C#, Python, or PHP
Data flow	Arrow	Communication between processes, or between processes and data stores	Network connections, HTTP, RPC, LPC
Data store	Two parallel lines with a label between them	Things that store data	Files, databases, the Windows Registry, shared memory segments
External entity	Rectangle with sharp corners	People, or code outside your control	Your customer, Microsoft.com

Figure 2-3 shows a classic DFD based on the elements from Table 2-1; however, it's possible to make these models more usable. Figure 2-4 shows this same model with a few changes, which you can use as an example for improving your own models.

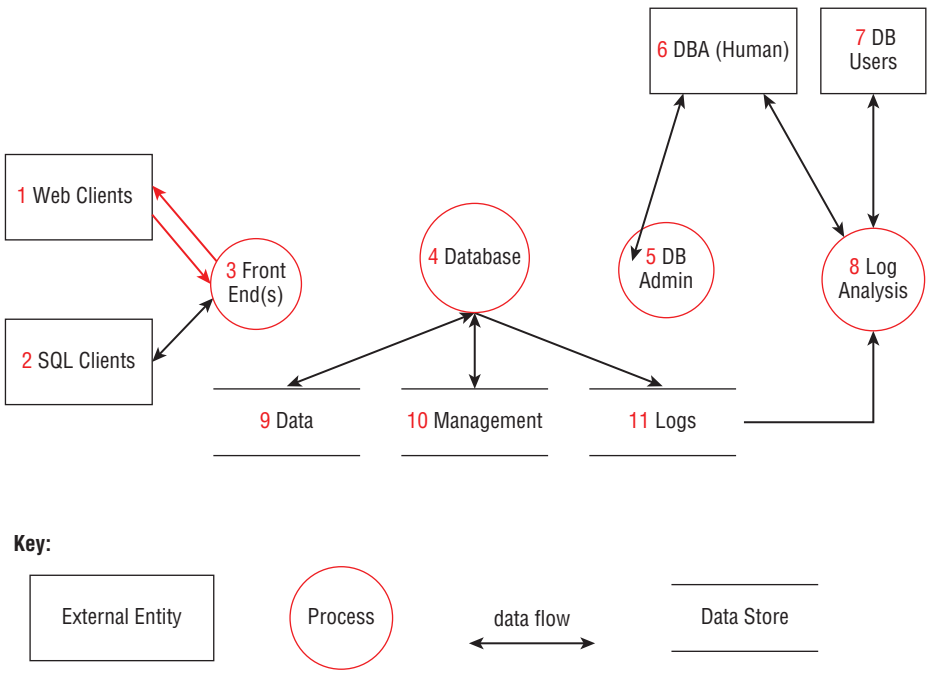

Figure 2-3: A classic DFD model

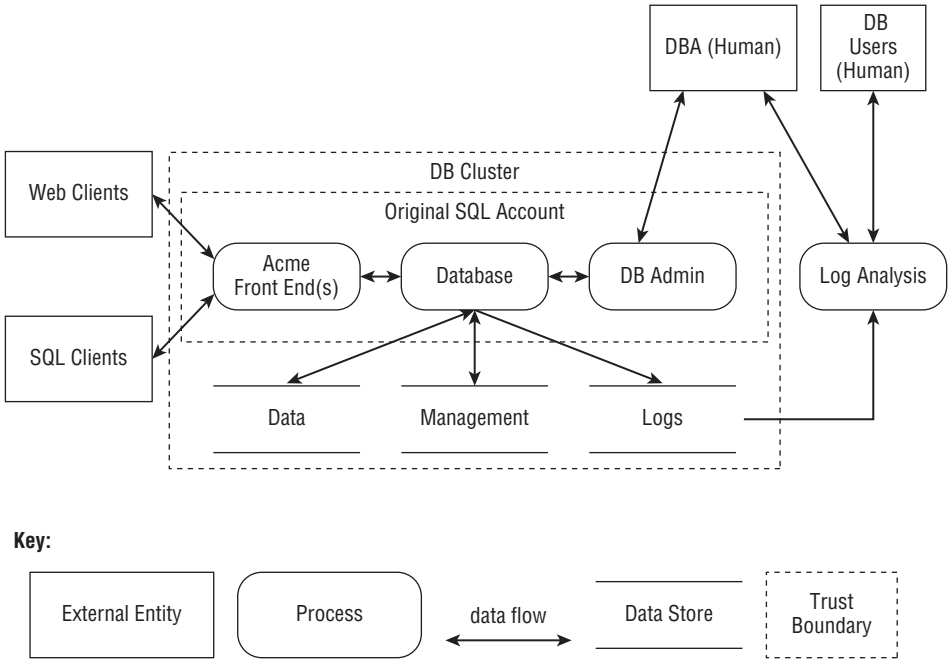

Figure 2-4: A modern DFD model (previously shown as Figure 2-1)

The following list explains the changes made from classic DFDs to more modern ones:

- The processes are rounded rectangles, which contain text more efficiently than circles.
- Straight lines are used, rather than curved, because straight lines are easier to follow, and you can fit more in larger diagrams.

Historically, many descriptions of data flow diagrams contained both "process" elements and "complex process" elements. A process was depicted as a circle, a complex process as two concentric circles. It isn't entirely clear, however, when to use a normal process versus a complex one. One possible rule is that anything that has a subdiagram should be a complex process. That seems like a decent rule, if (ahem) a bit circular.

DFDs can be used for things other than software products. For example, Figure 2-5 shows a sample operational network in a DFD. This is a typical model for a small to mid-sized corporate network, with a representative sampling

of systems and departments shown. It is discussed in depth in Appendix E, "Case Studies."

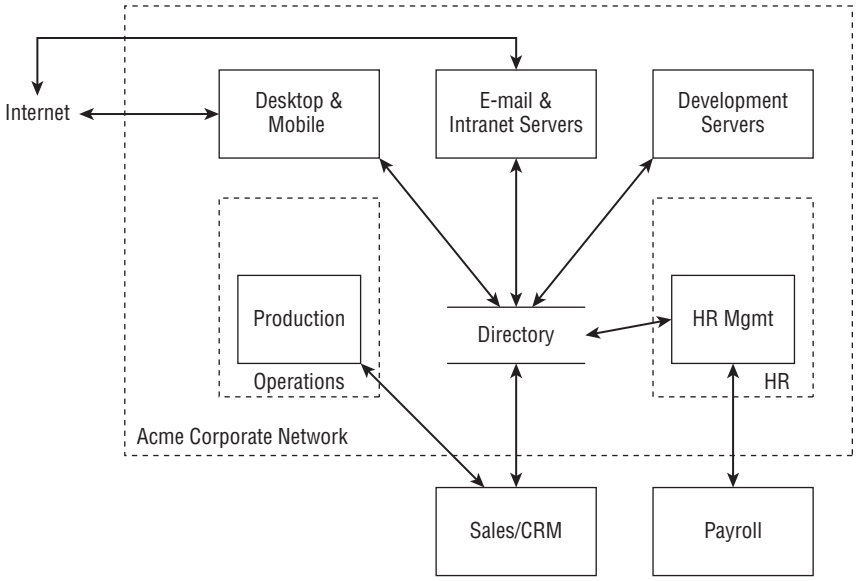

Figure 2-5: An operational network model

UML

UML is an abbreviation for Unified Modeling Language. If you use UML in your software development process, it's likely that you can adapt UML diagrams for threat modeling, rather than redrawing them. The most important way to adapt UML for threat modeling diagrams is the addition of trust boundaries.

UML is fairly complex. For example, the Visio stencils for UML offer roughly 80 symbols, compared to six for DFDs. This complexity brings a good deal of nuance and expressiveness as people draw structure diagrams, behavior diagrams, and interaction diagrams. If anyone involved in the threat modeling isn't up on all the UML symbols, or if there's misunderstanding about what those symbols mean, then the diagram's effectiveness as a tool is greatly diminished. In theory, anyone who's confused can just ask, but that requires them to know they're confused (they might assume that the symbol for fish

excludes sharks). It also requires a willingness to expose one's ignorance by asking a "simple" question. It's probably easier for a team that's invested in UML to add trust boundaries to those diagrams than to create new diagrams just for threat modeling.

Swim Lane Diagrams

Swim lane diagrams are a common way to represent flows between various participants. They're drawn using long lines, each representing participants in a protocol, with each participant getting a line. Each lane edge is labeled to identify the participant; each message is represented by a line between participants; and time is represented by flow down the diagram lanes. The diagrams end up looking a bit like swim lanes, thus the name. Messages should be labeled with their contents; or if the contents are complex, it may make more sense to have a diagram key that abstracts out some details. Computation done by the parties or state should be noted along that participant's line. Generally, participants in such protocols are entities like computers; and as such, swim lane diagrams usually have implicit trust boundaries between each participant. Cryptographer and protocol designer Carl Ellison has extended swim lanes to include the human participants as a way to structure discussion of what people are expected to know and do. He calls this extension *ceremonies*, which is discussed in more detail in Chapter 15, "Human Factors and Usability."

A sample swim lane diagram is shown in Figure 2-6.

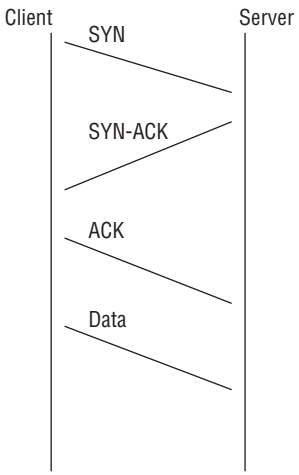

Figure 2-6: Swim lane diagram (showing the start of a TCP connection)

State Diagrams

State diagrams represent the various states a system can be in, and the transitions between those states. A computer system is modeled as a machine with state, memory, and rules for moving from one state to another, based on the valid messages it receives, and the data in its memory. (The computer should course test the messages it receives for validity according to some rules.) Each box is labeled with a state, and the lines between them are labeled with the conditions that cause the state transition. You can use state diagrams in threat modeling by checking whether each transition is managed in accordance with the appropriate security validations.

A very simple state machine for a door is shown in Figure 2-7 (derived from Wikipedia). The door has three states: opened, closed, and locked. Each state is entered by a transition. The "deadbolt" system is much easier to draw than locks on the knob, which can be locked from either state, creating a more complex diagram and user experience. Obviously, state diagrams can become complex quickly. You could imagine a more complex state diagram that includes "ajar," a state that can result from either open or closed. (I started drawing that but had trouble deciding on labels. Obviously, doors that can be ajar are poorly specified and should not be deployed.) You don't want to make architectural decisions just to make modeling easier, but often simple models are easier to work with, and reflect better engineering.

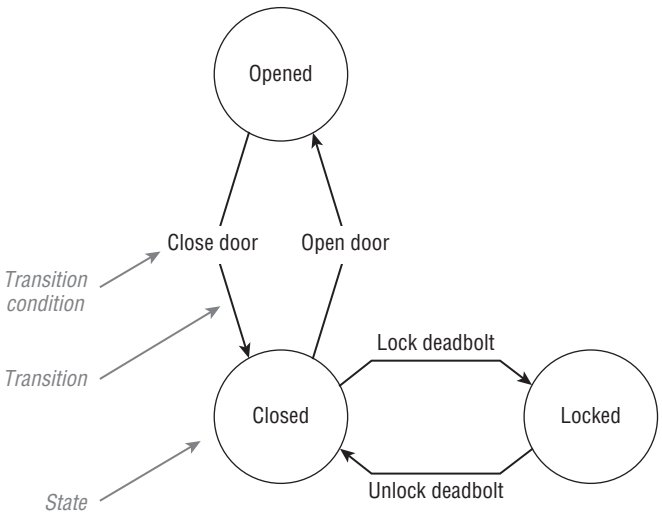

Figure 2-7: A state machine diagram

Trust Boundaries

As you saw in Chapter 1, a trust boundary is anyplace where various principals come together—that is, where entities with different privileges interact.

Drawing Boundaries

After a software model has been drawn, there are two ways to add boundaries: You can add the boundaries you know and look for more, or you can enumerate principals and look for boundaries. To start from boundaries, add any sorts of enforced trust boundary you can. Boundaries between unix UIDs, Windows sessions, machines, network segments, and so on should be drawn in as boxes, and the principal inside each box should be shown with a label.

To start from principals, begin from one end or the other of the privilege spectrum (often that's root/admin or anonymous Internet users), and then add boundaries each time they talk to "someone else."

You can always add at least one boundary, as all computation takes place in some context. (So you might criticize Figure 2-1 for showing Web Clients and SQL Clients without an identified context.)

If you don't see where to draw trust boundaries of any sort, your diagram may be detailed as everything is inside a single trust boundary, or you may be missing boundaries. Ask yourself two questions. First, does everything in the system have the same level of privilege and access to everything else on the system? Second, is everything your software communicates with inside that same boundary? If either of these answers are a no, then you should now have clarified either a missing boundary or a missing element in the diagram, or both. If both are yes, then you should draw a single trust boundary around everything, and move on to other development activities. (This state is unlikely except when every part of a development team has to create a software model. That "bottom up" approach is discussed in more detail in Chapter 7, "Processing and Managing Threats.")

A lot of writing on threat modeling claims that trust boundaries should only cross data flows. This is useful advice for the most detailed level of your model. If a trust boundary crosses over a data store (that is, a database), that might indicate that there are different tables or stored procedures with different trust levels. If a boundary crosses over a given host, it may reflect that members of, for example, the group "software installers," have different rights from the "web content updaters." If you find a trust boundary crossing an element of a diagram other than a data flow, either break that element into two (in the model, in reality, or both), or draw a subdiagram to show them separated into multiple entities. What enables good threat models is clarity about what boundaries exist and how those boundaries are to be protected. Contrariwise, a lack of clarity will inhibit the creation of good models.

Using Boundaries

Threats tend to cluster around trust boundaries. This may seem obvious: The trust boundaries delineate the attack surface between principals. This leads some to expect that threats appear *only* between the principals on the boundary, or only matter on the trust boundaries. That expectation is sometimes incorrect. To see why, consider a web server performing some complex order processing. For example, imagine assembling a computer at Dell's online store where thousands of parts might be added, but only a subset of those have been tested and are on offer. A model of that website might be constructed as shown in Figure 2-8.

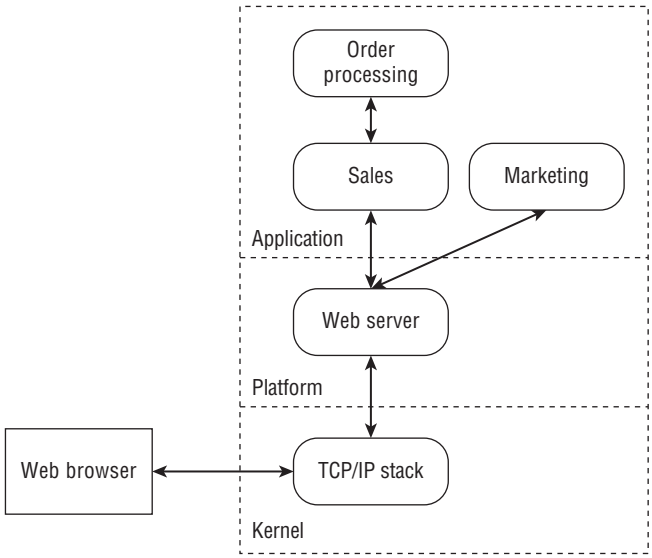

Figure 2-8: Trust boundaries in a web server

The web server in Figure 2-8 is clearly at risk of attack from the web browser, even though it talks through a TCP/IP stack that it presumably trusts. Similarly, the sales module is at risk; plus an attacker might be able to insert random part numbers into the HTML post in which the data is checked in an order processing module. Even though there's no trust boundary between the sales module and the order processing module, and even though data might be checked at three boundaries, the threats still follow the data flows. The client is shown simply as a web browser because the client is an external entity. Of course, there are many other components around that web browser, but you can't do anything about threats to them, so why model them?

Therefore, it is more accurate to say that threats tend to cluster around trust boundaries and complex parsing, but may appear anywhere that information is under the control of an attacker.

What to Include in a Diagram

So what should be in your diagram? Some rules of thumb include the following:

- Show the events that drive the system.
- Show the processes that are driven.
- Determine what responses each process will generate and send.
- Identify data sources for each request and response.
- Identify the recipient of each response.
- Ignore the inner workings, focus on scope.
- Ask if something will help you think about what goes wrong, or what will help you find threats.

This list is derived from Howard and LeBlanc's *Writing Secure Code, Second Edition* (Microsoft Press, 2009).

Complex Diagrams

When you're building complex systems, you may end up with complex diagrams. Systems do become complex, and that complexity can make using the diagrams (or understanding the full system) difficult.

One rule of thumb is "don't draw an eye chart." It is important to balance all the details that a real software project can entail with what you include in your actual model. As mentioned in Chapter 1, one technique you can use to help you do this is a subdiagram showing the details of one particular area. You should look for ways to break out highly-detailed areas that make sense for your project. For example, if you have one very complex process, maybe everything inside it is one diagram, and everything outside it is another. If you have a dispatcher or queuing system, that might be a good place to break things up. Maybe your databases or the fail over system is a good place to split. Maybe there are a few elements that really need more detail. All of these are good ways to break things out.

One helpful approach to subdiagrams is to ensure that there are not more subdiagrams than there are processes. Another approach is to use different diagrams to show different scenarios.

Sometimes it's also useful to simplify diagrams. When two elements of the diagram are equivalent from a security perspective, you can combine them. Equivalent means inside the same trust boundary, relying on the same technology, and handling the same sort of data.

The key thing to remember is that the diagram is intended to help ensure that you understand and can discuss the system. Remember the quote that opens this book: "All models are wrong, some models are useful." Therefore, when

you're adding additional diagrams, don't ask "is this the right way to do it?" Instead, ask "does this help us think about what might go wrong?"

Labels in Diagrams

Labels in diagrams should be short, descriptive, and meaningful. Because you want to use these names to tell stories, start with the outsiders who are driving the system; those are nouns, such as "customer" or "vibration sensor." They communicate information via data flows, which are nouns or noun phrases, such as "books to buy" or "vibration frequency." Data flows should almost never be labeled using verbs. Even though it can be hard, you should work to find more descriptive labels than "read" or "write," which are implied by the direction of the arrows. In other words, data flows communicate their information (nouns) to processes, which are active: verbs, verb phrases, or verb/noun chains.

Many people find it helpful to label data flows with sequence numbers to help keep track of what happens in what order. It can also be helpful to number elements within a diagram to help with completeness or communication. You can number each thing (data flow 1, a process 1, et cetera) or you can have a single count across the diagram, with external entity 1 talking over data flows 2 and 3 to process 4. Generally, using a single counter for everything is less confusing. You can say "number 1" rather than "data flow 1, not process 1."

Color in Diagrams

Color can add substantial amounts of information without appearing overwhelming. For example, Microsoft's Peter Torr uses green for trusted, red for untrusted and blue for what's being modeled (Torr, 2005). Relying on color alone can be problematic. Roughly one in twelve people suffer from color blindness, the most common being red/green confusion (Heitgerd, 2008). The result is that even with a color printer, a substantial number of people are unable to easily access this critical information. Box boundaries with text labels address both problems. With box trust boundaries, there is no reason not to use color.

Entry Points

One early approach to threat modeling was the "asset/entry point" approach, which can be effective at modeling operational systems. This approach can be partially broken down into the following steps:

1. Draw a DFD.
2. Find the points where data flows cross trust boundaries.
3. Label those intersections as "entry points."

NOTE **There were other steps and variations in the approaches, but we as a community have learned a lot since then, and a full explanation would be tedious and distracting.**

In the Acme/SQL example (as shown in Figure 2-1) the entry points are the "front end(s)" and the "database admin" console process. "Database" would also be an entry point, because nominally, other software could alter data in the databases and use failures in the parsers to gain control of the system. For the financials, the entry points shown are "external reporting," "financial planning and analysis," "core finance software," "sales" and "accounts receivable."

Validating Diagrams

Validating that a diagram is a good model of your software has two main goals: ensuring accuracy and aspiring to goodness. The first is easier, as you can ask whether it reflects reality. If important components are missing, or the diagram shows things that are not being built, then you can see that it doesn't reflect reality. If important data flows are missing, or nonexistent flows are shown, then it doesn't reflect reality. If you can't tell a story about the software without editing the diagram, then it's not accurate.

Of course, there's that word "important" in there, which leads to the second criterion: aspiring to goodness. What's important is what helps you find issues. Finding issues is a matter of asking questions like "does this element have any security impact?" and "are there things that happen sometimes or in special circumstances?" Knowing the answers to these questions is a matter of experience, just like many aspects of building software. A good and experienced architect can quickly assess requirements and address them, and a good threat modeler can quickly see which elements will be important. A big part of gaining that experience is practice. The structured approaches to finding threats in Part II, are designed to help you identify which elements are important.

How To Validate Diagrams

To best validate your diagrams, bring together the people who understand the system best. Someone should stand in front of the diagram and walk through the important use cases, ensuring the following:

- They can talk through stories about the diagram.
- They don't need to make changes to the diagram in its current form.
- They don't need to refer to things not present in the diagram.

The following rules of thumb will be useful as you update your diagram and gain experience:

- Anytime you find someone saying "sometimes" or "also" you should consider adding more detail to break out the various cases. For example, if you say, "Sometimes we connect to this web service via SSL, and sometimes we fall back to HTTP," you should draw both of those data flows (and consider whether an attacker can make you fall back like that).
- Anytime you need more detail to explain security-relevant behavior, draw it in.
- Each trust boundary box should have a label inside it.
- Anywhere you disagreed over the design or construction of the system, draw in those details. This is an important step toward ensuring that everyone ended that discussion on the same page. It's especially important for larger teams where not everyone is in the room for the threat model discussions. If anyone sees a diagram that contradicts their thinking, they can either accept it or challenge the assumptions; but either way, a good clear diagram can help get everyone on the same page.
- Don't have data sinks: You write the data for a reason. Show who uses it.
- Data can't move itself from one data store to another: Show the process that moves it.
- All ways data can arrive should be shown.
- If there are mechanisms for controlling data flows (such as firewalls or permissions) they should be shown.
- All processes must have at least one entry data flow and one exit data flow.
- As discussed earlier in the chapter, don't draw an eye chart.
- Diagrams should be visible on a printable page.

NOTE ***Writing Secure Code*** **author David LeBlanc notes that "A process without input is a miracle, while one without output is a black hole. Either you're missing something, or have mistaken a process for people, who are allowed to be black holes or miracles."**

When to Validate Diagrams

For software products, there are two main times to validate diagrams: when you create them and when you're getting ready to ship a beta. There's also a third triggering event (which is less frequent), which is if you add a security boundary.

For operational software diagrams, you also validate when you create them, and then again using a sensible balance between effort and up-to-dateness. That sensible balance will vary according to the maturity of a system, its scale, how tightly the components are coupled, the cadence of rollouts, and the nature of new rollouts. Here are a few guidelines:

- Newer systems will experience more diagram changes than mature ones.
- Larger systems will experience more diagram changes than smaller ones.
- Tightly coupled systems will experience more diagram changes than loosely coupled systems.
- Systems that roll out changes quickly will likely experience fewer diagram changes per rollout.
- Rollouts or sprints focused on refactoring or paying down technical debt will likely see more diagram changes. In either case, create an appropriate tracking item to ensure that you recheck your diagrams at a good time. The appropriate tracking item is whatever you use to gate releases or rollouts, such as bugs, task management software, or checklists. If you have no formal way to gate releases, then you might focus on a clearly defined release process before worrying about rechecking threat models. Describing such a process is beyond the scope of this book.

Summary

There's more than one way to threat model, and some of the strategies you can employ include modeling assets, modeling attackers, or modeling software. "What's your threat model" and brainstorming are good for security experts, but they lack structure that less experienced threat modelers need. There are more structured approaches to brainstorming, including scenario analysis, pre-mortems, movie plotting, and literature reviews, which can help bring a little structure, but they're still not great.

If your threat modeling starts from assets, the multiple overlapping definitions of the term, including things attackers want, things you're protecting, and stepping stones, can trip you up. An asset-centered approach offers no route to figure out what will go wrong with the assets.

Attacker modeling is also attractive, but trying to predict how another person will attack is hard, and the approach can invite arguments that "no one would do that." Additionally, human-centered approaches may lead you to human-centered threats that can be hard to address.

Software models are focused on that what software people understand. The best models are diagrams that help participants understand the software and find threats against it. There are a variety of ways you can diagram your software, and DFDs are the most frequently useful.

Once you have a model of the software, you'll need a way to find threats against it, and that is the subject of Part II.