CHAPTER

1

Dive In and Threat Model!

Anyone can learn to threat model, and what's more, everyone should. Threat modeling is about using models to find security problems. Using a model means abstracting away a lot of details to provide a look at a bigger picture, rather than the code itself. You model because it enables you to find issues in things you haven't built yet, and because it enables you to catch a problem before it starts. Lastly, you threat model as a way to anticipate the threats that could affect you.

Threat modeling is first and foremost a practical discipline, and this chapter is structured to reflect that practicality. Even though this book will provide you with many valuable definitions, theories, philosophies, effective approaches, and well-tested techniques, you'll want those to be grounded in experience. Therefore, this chapter avoids focusing on theory and ignores variations for now and instead gives you a chance to learn by experience.

To use an analogy, when you start playing an instrument, you need to develop muscles and awareness by playing the instrument. It won't sound great at the start, and it will be frustrating at times, but as you do it, you'll find it gets easier. You'll start to hit the notes and the timing. Similarly, if you use the simple four-step breakdown of how to threat model that's exercised in Parts I-III of this book, you'll start to develop your muscles. You probably know the old joke about the person who stops a musician on the streets of New York and asks "How do I get to Carnegie Hall?" The answer, of course, is "practice, practice, practice." Some of that includes following along, doing the exercises, and developing an

understanding of the steps involved. As you do so, you'll start to understand how the various tasks and techniques that make up threat modeling come together.

In this chapter you're going to find security flaws that might exist in a design, so you can address them. You'll learn how to do this by examining a simple web application with a database back end. This will give you an idea of what can go wrong, how to address it, and how to check your work. Along the way, you'll learn to play *Elevation of Privilege*, a serious game designed to help you start threat modeling. Finally you'll get some hands-on experience building your own threat model, and the chapter closes with a set of checklists that help you get started threat modeling.

Learning to Threat Model

You begin threat modeling by focusing on four key questions:

1. What are you building?
2. What can go wrong?
3. What should you do about those things that can go wrong?
4. Did you do a decent job of analysis?

In addressing these questions, you start and end with tasks that all technologists should be familiar with: drawing on a whiteboard and managing bugs. In between, this chapter will introduce a variety of new techniques you can use to think about threats. If you get confused, just come back to these four questions.

Everything in this chapter is designed to help you answer one of these questions. You're going to first walk through these questions using a three-tier web app as an example, and after you've read that, you should walk through the steps again with something of your own to threat model. It could be software you're building or deploying, or software you're considering acquiring. If you're feeling uncertain about what to model, you can use one of the sample systems in this chapter or an exercise found in Appendix E, "Case Studies."

The second time you work through this chapter, you'll need a copy of the *Elevation of Privilege* threat-modeling game. The game uses a deck of cards that you can download free from `threatmodelingbook.com/resources`. You should get two–four friends or colleagues together for the game part.

You start with building a diagram, which is the first of four major activities involved in threat modeling and is explained in the next section. The other three include finding threats, addressing them, and then checking your work.

What Are You Building?

Diagrams are a good way to communicate what you are building. There are lots of ways to diagram software, and you can start with a whiteboard diagram of how data flows through the system. In this example, you're working with a simple web app with a web browser, web server, some business logic and a database (see Figure 1-1).

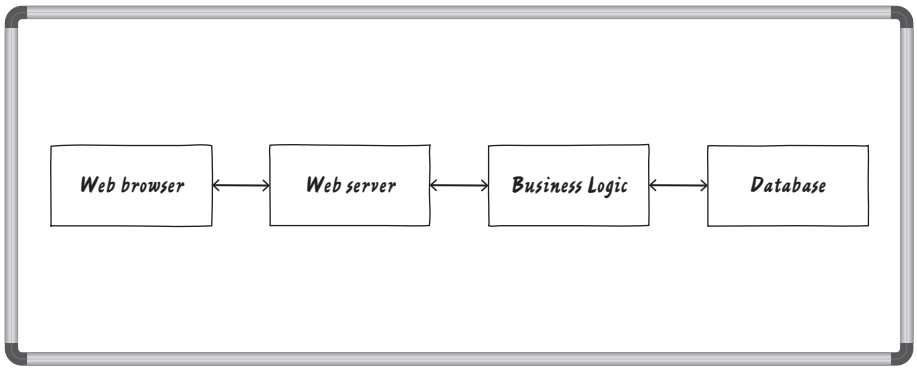

Figure 1-1: A whiteboard diagram

Some people will actually start thinking about what goes wrong right here. For example, how do you know that the web browser is being used by the person you expect? What happens if someone modifies data in the database? Is it OK for information to move from one box to the next without being encrypted? You might want to take a minute to think about some things that could go wrong here because these sorts of questions may lead you to ask "is that allowed?" You can create an even better model of what you're building if you think about "who controls what" a little. Is this a website for the whole Internet, or is it an intranet site? Is the database on site, or at a web provider?

For this example, let's say that you're building an Internet site, and you're using the fictitious Acme storage-system. (I'd put a specific product here, but then I'd get some little detail wrong and someone, certainly not you, would get all wrapped around the axle about it and miss the threat modeling lesson. Therefore, let's just call it Acme, and pretend it just works the way I'm saying. Thanks! I knew you'd understand.)

Adding boundaries to show who controls what is a simple way to improve the diagram. You can pretty easily see that the threats that cross those boundaries are likely important ones, and may be a good place to start identifying threats. These boundaries are called *trust boundaries*, and you should draw

them wherever different people control different things. Good examples of this include the following:

- Accounts (UIDs on unix systems, or SIDS on Windows)
- Network interfaces
- Different physical computers
- Virtual machines
- Organizational boundaries
- Almost anywhere you can argue for different privileges

TRUST BOUNDARY VERSUS ATTACK SURFACE

A closely related concept that you may have encountered is *attack surface*. For example, the hull of a ship is an attack surface for a torpedo. The side of a ship presents a larger attack surface to a submarine than the bow of the same ship. The ship may have internal "trust" boundaries, such as waterproof bulkheads or a Captain's safe. A system that exposes lots of interfaces presents a larger attack surface than one that presents few APIs or other interfaces. Network firewalls are useful boundaries because they reduce the attack surface relative to an external attacker. However, much like the Captain's safe, there are still trust boundaries inside the firewall. A trust boundary and an attack surface are very similar views of the same thing. An attack surface is a trust boundary and a direction from which an attacker could launch an attack. Many people will treat the terms are interchangeable. In this book, you'll generally see "trust boundary" used.

In your diagram, draw the trust boundaries as boxes (see Figure 1-2), showing what's inside each with a label (such as "corporate data center") near the edge of the box.

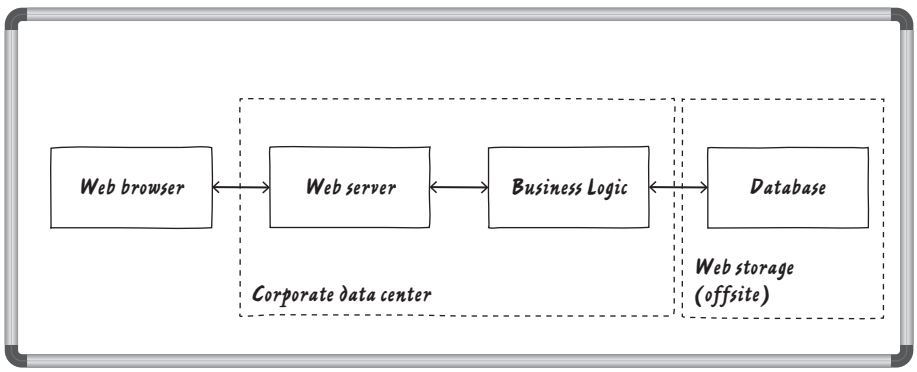

Figure 1-2: Trust boundaries added to a whiteboard diagram

As your diagram gets larger and more complex, it becomes easy to miss a part of it, or to become confused by labels on the data flows. Therefore, it can be very helpful to number each process, data flow, and data store in the diagram, as shown in Figure 1-3. (Because each trust boundary should have a unique name, representing the unique trust inside of it, there's limited value to numbering those.)

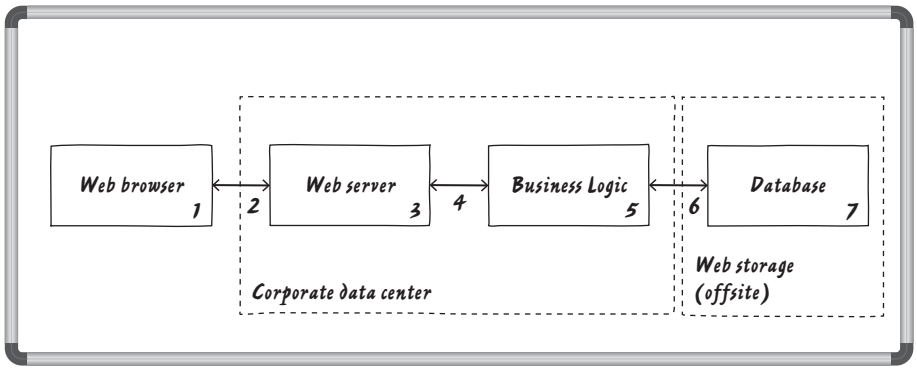

Figure 1-3: Numbers and trust boundaries added to a whiteboard diagram

Regarding the physical form of the diagram: Use whatever works for you. If that's a whiteboard diagram and a camera phone picture, great. If it's Visio, or OmniGraffle, or some other drawing program, great. You should think of threat model diagrams as part of the development process, so try to keep it in source control with everything else.

Now that you have a diagram, it's natural to ask, is it the right diagram? For now, there's a simple answer: Let's assume it is. Later in this chapter there are some tips and checklists as well as a section on updating the diagram, but at this stage you have a good enough diagram to get started on identifying threats, which is really why you bought this book. So let's identify.

What Can Go Wrong?

Now that you have a diagram, you can really start looking for what can go wrong with its security. This is so much fun that I turned it into a game called, *Elevation of Privilege*. There's more on the game in Appendix D, "Elevation of Privilege: The Cards," which discusses each card, and in Chapter 11, "Threat Modeling Tools," which covers the history and philosophy of the game, but you can get started playing now with a few simple instructions. If you haven't already done so, download a deck of cards from `http://www.microsoft.com/security/sdl/adopt/eop.aspx`. Print the pages in color, and cut them into individual cards. Then shuffle the deck and deal it out to those friends you've invited to play.

NOTE Some people aren't used to playing games at work. Others approach new games with trepidation, especially when those games involve long, complicated instructions. *Elevation of Privilege* takes just a few lines to explain. You should give it a try.

How To Play Elevation of Privilege

Elevation of Privilege is a serious game designed to help you threat model. A sample card is shown in Figure 1-4. You'll notice that like playing cards, it has a number and suit in the upper left, and an example of a threat as the main text on the card. To play the game, simply follow the instructions in the upcoming list.

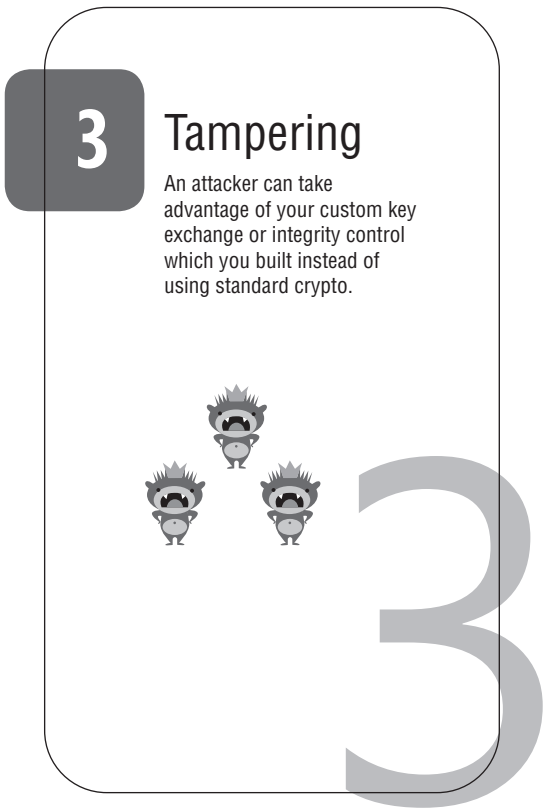

Figure 1-4: An Elevation of Privilege card

1. Deal the deck. (Shuffling is optional.)
2. The person with the 3 of Tampering leads the first round. (In card games like this, rounds are also called "tricks" or "hands.")

3. Each round works like so:
   A. Each player plays one card, starting with the person leading the round, and then moving clockwise.
   B. To play a card, read it aloud, and try to determine if it affects the system you have diagrammed. If you can link it, write it down, and score yourself a point. Play continues clockwise with the next player.
   C. When each player has played a card, the player who has played the highest card wins the round. That player leads the next round.
4. When all the cards have been played, the game ends and the person with the most points wins.
5. If you're threat modeling a system you're building, then you go file any bugs you find.

There are some folks who threat model like this in their sleep, or even have trouble switching it off. Not everyone is like that. That's OK. Threat modeling is not rocket science. It's stuff that anyone who participates in software development can learn. Not everyone wants to dedicate the time to learn to do it in their sleep.

Identifying threats can seem intimidating to a lot of people. If you're one of them, don't worry. This section is designed to gently walk you through threat identification. Remember to have fun as you do this. As one reviewer said: "Playing *Elevation of Privilege* should be *fun*. Don't downplay that. We play it every Friday. It's enjoyable, relaxing, and still has business value."

Outside of the context of the game, you can take the next step in threat modeling by thinking of things that might go wrong. For instance, how do you know that the web browser is being used by the person you expect? What happens if someone modifies data in the database? Is it OK for information to move from one box to the next without being encrypted? You don't need to come up with these questions by just staring at the diagram and scratching your chin. (I didn't!) You can identify threats like these using the simple mnemonic STRIDE, described in detail in the next section.

Using the STRIDE Mnemonic to Find Threats

STRIDE is a mnemonic for things that go wrong in security. It stands for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege:

- **Spoofing** is pretending to be something or someone you're not.

- **Tampering** is modifying something you're not supposed to modify. It can include packets on the wire (or wireless), bits on disk, or the bits in memory.
- **Repudiation** means claiming you didn't do something (regardless of whether you did or not).
- **Information Disclosure** is about exposing information to people who are not authorized to see it.
- **Denial of Service** are attacks designed to prevent a system from providing service, including by crashing it, making it unusably slow, or filling all its storage.
- **Elevation of Privilege** is when a program or user is technically able to do things that they're not supposed to do.

NOTE This is where *Elevation of Privilege*, the game, gets its name. This book uses *Elevation of Privilege*, italicized, or abbreviated to EoP, for the game—to avoid confusion with the threat.

Recall the three example threats mentioned in the preceding section:

- How do you know that the web browser is being used by the person you expect?
- What happens if someone modifies data in the database?
- Is it ok for information to go from one box to the next without being encrypted?

These are examples of spoofing, tampering, and information disclosure. Using STRIDE as a mnemonic can help you walk through a diagram and select example threats. Pair that with a little knowledge of security and the right techniques, and you'll find the important threats faster and more reliably. If you have a process in place for ensuring that you develop a threat model, document it, and you can increase confidence in your software.

Now that you have STRIDE in your tool belt, walk through your diagram again and look for more threats, this time using the mnemonic. Make a list as you go with the threat and what element of the diagram it affects. (Generally, the software, data flow, or storage is affected, rather than the trust boundary.) The following list provides some examples of each threat.

- **Spoofing:** Someone might pretend to be another customer, so you'll need a way to authenticate users. Someone might also pretend to be your website, so you should ensure that you have an SSL certificate and that you use a single domain for all your pages (to help that subset of customers who read URLs to see if they're in the right place). Someone might also place a deep link to one of your pages, such as `logout.html` or `placeorder.aspx`. You should be checking the Referrer field before taking action. That's not a complete solution to what are called CSRF (Cross Site Request Forgery) attacks, but it's a start.

- **Tampering:** Someone might tamper with the data in your back end at Acme. Someone might tamper with the data as it flows back and forth between their data center and yours. A programmer might replace the operational code on the web front end without testing it, thinking they're uploading it to staging. An angry programmer might add a coupon code "PayBobMore" that offers a 20 percent discount on all goods sold.
- **Repudiation:** Any of the preceding actions might require digging into what happened. Are there system logs? Is the right information being logged effectively? Are the logs protected against tampering?
- **Information Disclosure:** What happens if Acme reads your database? Can anyone connect to the database and read or write information?
- **Denial of Service:** What happens if a thousand customers show up at once at the website? What if Acme goes down?
- **Elevation of Privilege:** Perhaps the web front end is the only place customers should access, but what enforces that? What prevents them from connecting directly to the business logic server, or uploading new code? If there's a firewall in place, is it correctly configured? What controls access to your database at Acme, or what happens if an employee at Acme makes a mistake, or even wants to edit your files?

The preceding possibilities aren't intended to be a complete list of how each threat might manifest against every model. You can find a more complete list in Chapter 3, "STRIDE." This shorter version will get you started though, and it is focused on what you might need to investigate based on the very simple diagram shown in Figure 1-2. Remember the musical instrument analogy. If you try to start playing the piano with Ravel's Gaspard (regarded as one of the most complex piano pieces ever written), you're going to be frustrated.

Tips for Identifying Threats

Whether you are identifying threats using *Elevation of Privilege*, STRIDE, or both, here are a few tips to keep in mind that can help you stay on the right track to determine what could go wrong:

- **Start with external entities:** If you're not sure where to start, start with the external entities or events which drive activity. There are many other valid approaches though: You might start with the web browser, looking for spoofing, then tampering, and so on. You could also start with the business logic if perhaps your lead developer for that component is in the room. Wherever you choose to begin, you want to aspire to some level of organization. You could also go in "STRIDE order" through the diagram. Without some organization, it's hard to tell when you're done, but be careful not to add so much structure that you stifle creativity.

- **Never ignore a threat because it's not what you're looking for right now.** You might come up with some threats while looking at other categories. Write them down and come back to them. For example, you might have thought about "can anyone connect to our database," which is listed under information disclosure, while you were looking for spoofing threats. If so, that's awesome! Good job! Redundancy in what you find can be tedious, but it helps you avoid missing things. If you find yourself asking whether "someone not authorized to connect to the database who reads information" constitutes spoofing or information disclosure, the answer is, who cares? Record the issue and move along to the next one. STRIDE is a tool to guide you to threats, not to ask you to categorize what you've found; it makes a lousy taxonomy, anyway. (That is to say, there are plenty of security issues for which you can make an argument for various different categorizations. Compare and contrast it with a good taxonomy, such as the taxonomy of life. Does it have a backbone? If so, it's a vertebrate.)
- **Focus on feasible threats:** Along the way, you might come up with threats like "someone might insert a back door at the chip factory," or "someone might hire our janitorial staff to plug in a hardware key logger and steal all our passwords." These are real possibilities but not very likely compared to using an exploit to attack a vulnerability for which you haven't applied the patch, or tricking someone into installing software. There's also the question of what you can do about either, which brings us to the next section.

Addressing Each Threat

You should now have a decent-sized list or lists of threats. The next step in the threat modeling process is to go through the lists and address each threat. There are four types of action you can take against each threat: Mitigate it, eliminate it, transfer it, or accept it. The following list looks briefly at each of these ways to address threats, and then in the subsequent sections you will learn how to address each specific threat identified with the STRIDE list in the "What Can Go Wrong" section. For more details about each of the strategies and techniques to address these threats, see Chapters 8 and 9, "Defensive Building Blocks" and "Tradeoffs When Addressing Threats."

- **Mitigating threats** is about doing things to make it harder to take advantage of a threat. Requiring passwords to control who can log in mitigates the threat of spoofing. Adding password controls that enforce complexity or expiration makes it less likely that a password will be guessed or usable if stolen.
- **Eliminating threats** is almost always achieved by eliminating features. If you have a threat that someone will access the administrative function of

a website by visiting the `/admin/URL`, you can mitigate it with passwords or other authentication techniques, but the threat is still present. You can make it less likely to be found by using a URL like `/j8e8vg21euwq/`, but the threat is still present. You can eliminate it by removing the interface, handling administration through the command line. (There are still threats associated with how people log in on a command line. Moving away from HTTP makes the threat easier to mitigate by controlling the attack surface. Both threats would be found in a complete threat model.) Incidentally, there are other ways to eliminate threats if you're a mob boss or you run a police state, but I don't advocate their use.

- **Transferring threats** is about letting someone or something else handle the risk. For example, you could pass authentication threats to the operating system, or trust boundary enforcement to a firewall product. You can also transfer risk to customers, for example, by asking them to click through lots of hard-to-understand dialogs before they can do the work they need to do. That's obviously not a great solution, but sometimes people have knowledge that they can contribute to making a security tradeoff. For example, they might know that they just connected to a coffee shop wireless network. If you believe the person has essential knowledge to contribute, you should work to help her bring it to the decision. There's more on doing that in Chapter 15, "Human Factors and Usability."
- **Accepting the risk** is the final approach to addressing threats. For most organizations most of the time, searching everyone on the way in and out of the building is not worth the expense or the cost to the dignity and job satisfaction of those workers. (However, diamond mines and sometimes government agencies take a different approach.) Similarly, the cost of preventing someone from inserting a back door in the motherboard is expensive, so for each of these examples you might choose to accept the risk. And once you've accepted the risk, you shouldn't worry over it. Sometimes worry is a sign that the risk hasn't been fully accepted, or that the risk acceptance was inappropriate.

The strategies listed in the following tables are intended to serve as examples to illustrate ways to address threats. Your "go-to" approach should be to mitigate threats. Mitigation is generally the easiest and the best for your customers. (It might look like accepting risk is easier, but over time, mitigation is easier.) Mitigating threats can be hard work, and you shouldn't take these examples as complete. There are often other valid ways to address each of these threats, and sometimes trade-offs must be made in the way the threats are addressed.

Addressing Spoofing

Table 1-1 and the list that follows show targets of spoofing, mitigation strategies that address spoofing, and techniques to implement those mitigations.

Table 1-1: Addressing Spoofing Threats

THREAT TARGET	MITIGATION STRATEGY	MITIGATION TECHNIQUE
Spoofing a person	Identification and authentication (usernames and something you know/have/are)	Usernames, real names, or other identifiers: ❖ Passwords ❖ Tokens ❖ Biometrics Enrollment/maintenance/expiry
Spoofing a "file" on disk	Leverage the OS	❖ Full paths ❖ Checking ACLs ❖ Ensuring that pipes are created properly
	Cryptographic authenticators	Digital signatures or authenticators
Spoofing a network address	Cryptographic	❖ DNSSEC ❖ HTTPS/SSL ❖ IPsec
Spoofing a program in memory	Leverage the OS	Many modern operating systems have some form of application identifier that the OS will enforce.

- When you're concerned about a person being spoofed, ensure that each person has a unique username and some way of authenticating. The traditional way to do this is with passwords, which have all sorts of problems as well as all sorts of advantages that are hard to replicate. See Chapter 14, "Accounts and Identity" for more on passwords.
- When accessing a file on disk, don't ask for the file with `open(file)`. Use `open(/path/to/file)`. If the file is sensitive, after opening, check various security elements of the file descriptor (such as fully resolved name, permissions, and owner). You want to check with the file descriptor to avoid *race conditions*. This applies doubly when the file is an executable, although checking after opening can be tricky. Therefore, it may help to ensure that the permissions on the executable can't be changed by an attacker. In any case, you almost never want to call `exec()` with `./file`.
- When you're concerned about a system or computer being spoofed when it connects over a network, you'll want to use DNSSEC, SSL, IPsec, or a combination of those to ensure you're connecting to the right place.

Addressing Tampering

Table 1-2 and the list that follows show targets of tampering, mitigation strategies that address tampering, and techniques to implement those mitigations.

Table 1-2: Addressing Tampering Threats

THREAT TARGET	MITIGATION STRATEGY	MITIGATION TECHNIQUE
Tampering with a file	Operating system	ACLs
	Cryptographic	❖ Digital Signatures ❖ Keyed MAC
Racing to create a file (tampering with the file system)	Using a directory that's protected from arbitrary user tampering	ACLs Using private directory structures (Randomizing your file names just makes it annoying to execute the attack.)
Tampering with a network packet	Cryptographic	❖ HTTPS/SSL ❖ IPsec
	Anti-pattern	Network isolation (See note on network isolation anti-pattern.)

- **Tampering with a file:** Tampering with files can be easy if the attacker has an account on the same machine, or by tampering with the network when the files are obtained from a server.
- **Tampering with memory:** The threats you want to worry about are those that can occur when a process with less privileges than you, or that you don't trust, can alter memory. For example, if you're getting data from a shared memory segment, is it ACLed so only the other process can see it? For a web app that has data coming in via AJAX, make sure you validate that the data is what you expect after you pull in the right amount.
- **Tampering with network data:** Preventing tampering with network data requires dealing with both spoofing and tampering. Otherwise, someone who wants to tamper can simply pretend to be the other end, using what's called a *man-in-the-middle attack*. The most common solution to these problems is SSL, with IP Security (IPsec) emerging as another possibility. SSL and IPsec both address confidentiality and tampering, and can help address spoofing.

- **Tampering with networks anti-pattern:** It's somewhat common for people to hope that they can isolate their network, and so not worry about tampering threats. It's also very hard to maintain isolation over time. Isolation doesn't work as well as you would hope. For example, the isolated United States SIPRNet was thoroughly infested with malware, and the operation to clean it up took 14 months (Shachtman, 2010).

NOTE A program can't check whether it's authentic after it loads. It may be possible for something to rely on "trusted bootloaders" to provide a chain of signatures, but the security decisions are being made external to that code. (If you're not familiar with the technology, don't worry, the key lesson is that a program cannot check its own authenticity.)

Addressing Repudiation

Addressing repudiation is generally a matter of ensuring that your system is designed to log and ensuring that those logs are preserved and protected. Some of that can be handled with simple steps such as using a reliable transport for logs. In this sense, syslog over UDP was almost always silly from a security perspective; syslog over TCP/SSL is now available and is vastly better.

Table 1-3 and the list that follows show targets of repudiation, mitigation strategies that address repudiation, and techniques to implement those mitigations.

Table 1-3: Addressing Repudiation Threats

THREAT TARGET	MITIGATION STRATEGY	MITIGATION TECHNIQUE
No logs means you can't prove anything.	Log	Be sure to log all the security-relevant information.
Logs come under attack	Protect your logs.	❖ Send over the network. ❖ ACL
Logs as a channel for attack	Tightly specified logs	Documenting log design early in the development process

- **No logs means you can't prove anything:** This is self-explanatory. For example, when a customer calls to complain that they never got their order, how will this be resolved? Maintain logs so that you can investigate what happens when someone attempts to repudiate something.

- **Logs come under attack:** Attackers will do things to prevent your logs from being useful, including filling up the log to make it hard to find the attack or forcing logs to "roll over." They may also do things to set off so many alarms that the real attack is lost in a sea of troubles. Perhaps obviously, sending logs over a network exposes them to other threats that you'll need to handle.
- **Logs as a channel for attack:** By design, you're collecting data from sources outside your control, and delivering that data to people and systems with security privileges. An example of such an attack might be sending mail addressed to `"</html> haha@example.com"`, causing trouble for web-based tools that don't expect inline HTML.

You can make it easier to write secure code to process your logs by clearly communicating what your logs can't contain, such as "Our logs are all plaintext, and attackers can insert all sorts of things," or "Fields 1–5 of our logs are tightly controlled by our software, fields 6–9 are easy to inject data into. Field 1 is time in GMT. Fields 2 and 3 are IP addresses (v4 or 6)..." Unless you have incredibly strict control, documenting what your logs can contain will likely miss things. (For example, can your logs contain Unicode double-wide characters?)

Addressing Information Disclosure

Table 1-4 and the list which follows show targets of information disclosure, mitigation strategies that address information disclosure, and techniques to implement those mitigations.

Table 1-4: Addressing Information Disclosure Threats

THREAT TARGET	MITIGATION STRATEGY	MITIGATION TECHNIQUE
Network monitoring	Encryption	❖ HTTPS/SSL ❖ IPsec
Directory or filename (for example `layoff-letters/adamshostack.docx`)	Leverage the OS.	ACLs
File contents	Leverage the OS.	ACLS
	Cryptography	File encryption such as PGP, disk encryption (FileVault, BitLocker)
API information disclosure	Design	Careful design control Consider pass by reference or value.

- **Network monitoring:** Network monitoring takes advantage of the architecture of most networks to monitor traffic. (In particular, most networks now broadcast packets, and each listener is expected to decide if the packet matters to them.) When networks are architected differently, there are a variety of techniques to draw traffic to or through the monitoring station.

  If you don't address spoofing, much like tampering, an attacker can just sit in the middle and spoof each end. Mitigating network information disclosure threats requires handling both spoofing and tampering threats. If you don't address tampering, then there are all sorts of clever ways to get information out. Here again, SSL and IP Security options are your simplest choices.

- **Names reveal information:** When the name of a directory or a filename itself will reveal information, then the best way to protect it is to create a parent directory with an innocuous name and use operating system ACLs or permissions.

- **File content is sensitive:** When the contents of the file need protection, use ACLs or cryptography. If you want to protect all the data should the machine fall into unauthorized hands, you'll need to use cryptography. The forms of cryptography that require the person to manually enter a key or passphrase are more secure and less convenient. There's file, filesystem, and database cryptography, depending on what you need to protect.

- **APIs reveal information:** When designing an API, or otherwise passing information over a trust boundary, select carefully what information you disclose. You should assume that the information you provide will be passed on to others, so be selective about what you provide. For example, website errors that reveal the username and password to a database are a common form of this flaw, others are discussed in Chapter 3.

Addressing Denial of Service

Table 1-5 and the list that follows show targets of denial of service, mitigation strategies that address denial of service, and techniques to implement those mitigations.

Table 1-5: Addressing Denial of Service Threats

THREAT TARGET	MITIGATION STRATEGY	MITIGATION TECHNIQUE
Network flooding	Look for exhaustible resources.	❖ Elastic resources ❖ Work to ensure attacker resource consumption is as high as or higher than yours.
		Network ACLS
Program resources	Careful design	Elastic resource management, proof of work
	Avoid multipliers.	Look for places where attackers can multiply CPU consumption on your end with minimal effort on their end: Do something to require work or enable distinguishing attackers, such as client does crypto first or login before large work factors (of course, that can't mean that logins are unencrypted).
System resources	Leverage the OS.	Use OS settings.

- **Network flooding:** If you have static structures for the number of connections, what happens if those fill up? Similarly, to the extent that it's under your control, don't accept a small amount of network data from a possibly spoofed address and return a lot of data. Lastly, firewalls can provide a layer of network ACLs to control where you'll accept (or send) traffic, and can be useful in mitigating network denial-of-service attacks.
- **Look for exhaustible resources:** The first set of exhaustible resources are network related, the second set are those your code manages, and the third are those the OS manages. In each case, elastic resourcing is a valuable technique. For example, in the 1990s some TCP stacks had a hardcoded limit of five half-open TCP connections. (A half-open connection is one in the process of being opened. Don't worry if that doesn't make sense, but rather ask yourself why the code would be limited to five of them.) Today, you can often obtain elastic resourcing of various types from cloud providers.

- **System resources:** Operating systems tend to have limits or quotas to control the resource consumption of user-level code. Consider those resources that the operating system manages, such as memory or disk usage. If your code runs on dedicated servers, it may be sensible to allow it to chew up the entire machine. Be careful if you unlimit your code, and be sure to document what you're doing.
- **Program resources:** Consider resources that your program manages itself. Also, consider whether the attacker can make you do more work than they're doing. For example, if he sends you a packet full of random data and you do expensive cryptographic operations on it, then your vulnerability to denial of service will be higher than if you make him do the cryptography first. Of course, in an age of botnets, there are limits to how well one can reassign this work. There's an excellent paper by Ben Laurie and Richard Clayton, "Proof of work proves not to work," which argues against proof of work schemes (Laurie, 2004).

Addressing Elevation of Privilege

Table 1-6 and the list that follows show targets of elevation of privilege, mitigation strategies that address elevation of privilege, and techniques to implement those mitigations.

Table 1-6: Addressing Elevation of Privilege Threats

THREAT TARGET	MITIGATION STRATEGY	MITIGATION TECHNIQUE
Data/code confusion	Use tools and architectures that separate data and code.	❖ Prepared statements or stored procedures in SQL ❖ Clear separators with canonical forms ❖ Late validation that data is what the next function expects
Control flow/ memory corruption attacks	Use a type-safe language.	Writing code in a type-safe language protects against entire classes of attack.
	Leverage the OS for memory protection.	Most modern operating systems have memory-protection facilities.

THREAT TARGET	MITIGATION STRATEGY	MITIGATION TECHNIQUE
	Use the sandbox.	❖ Modern operating systems support sandboxing in various ways (AppArmor on Linux, AppContainer or the MOICE pattern on Windows, Sandboxlib on Mac OS). ❖ Don't run as the "nobody" account, create a new one for each app. Postfix and QMail are examples of the good pattern of one account per function.
Command injection attacks	Be careful.	❖ Validate that your input is the size and form you expect. ❖ Don't sanitize. Log and then throw it away if it's weird.

- **Data/code confusion:** Problems where data is treated as code are common. As information crosses layers, what's tainted and what's pure can be lost. Attacks such as XSS take advantage of HTML's freely interweaving code and data. (That is, an .html file contains both code, such as Javascript, and data, such as text, to be displayed and sometimes formatting instructions for that text.) There are a few strategies for dealing with this. The first is to look for ways in which frameworks help you keep code and data separate. For example, prepared statements in SQL tell the database what statements to expect, and where the data will be.

  You can also look at the data you're passing right before you pass it, so you know what validation you might be expected to perform for the function you're calling. For example, if you're sending data to a web page, you might ensure that it contains no <, >, #, or & characters, or whatever.

  In fact, the value of "whatever" is highly dependent on exactly what exists between "you" and the rendition of the web page, and what security checks it may be performing. If "you" means a web server, it may be very important to have a few < and > symbols in what you produce. If "you" is something taking data from a database and sending it to, say PHP, then the story is quite different. Ideally, the nature of "you" and the additional steps are clear in your diagrams.
- **Control flow/memory corruption attacks:** This set of attacks generally takes advantage of weak typing and static structures in C-like languages to enable an attacker to provide code and then jump to that code. If you

use a type-safe language, such as Java or C#, many of these attacks are harder to execute.

Modern operating systems tend to contain memory protection and randomization features, such as Address Space Layout Randomization (ASLR). Sometimes the features are optional, and require a compiler or linker switch. In many cases, such features are almost free to use, and you should at least try all such features your OS supports. (It's not completely effortless, you may need to recompile, test, or make other such small investments.)

The last set of controls to address memory corruption are sandboxes. Sandboxes are OS features that are designed to protect the OS or the rest of the programs running as the user from a corrupted program.

NOTE Details about each of these features are outside the scope of this book, but searching on terms such as type safety, ASLR, and sandbox should provide a plethora of details.

- **Command injection attacks:** Command injection attacks are a form of code/data confusion where an attacker supplies a control character, followed by commands. For example, in SQL injection, a single quote will often close a dynamic SQL statement; and when dealing with unix shell scripts, the shell can interpret a semicolon as the end of input, taking anything after that as a command.

In addition to working through each STRIDE threat you encounter, a few other recurring themes will come up as you address your threats; these are covered in the following two sections.

Validate, Don't Sanitize

Know what you expect to see, how much you expect to see, and validate that that's what you're receiving. If you get something else, throw it away and return an error message. Unless your code is perfect, errors in sanitization will hurt a lot, because after you write that sanitize input function you're going to rely on it. There have been fascinating attacks that rely on a sanitize function to get their code into shape to execute.

Trust the Operating System

One of the themes that recurs in the preceding tables is "trust the operating system." Of course, you may want to discount that because I did much of this

work while working for Microsoft, a purveyor of a variety of fine operating system software, so there might be some bias here. It's a valid point, and good for you for being skeptical. See, you're threat modeling already!

More seriously, trusting the operating system is a good idea for a number of reasons:

- The operating system provides you with security features so you can focus on your unique value proposition.
- The operating system runs with privileges that are probably not available to your program or your attacker.
- If your attacker controls the operating system, you're likely in a world of hurt regardless of what your code tries to do.

With all of that "trust the operating system" advice, you might be tempted to ask why you need this book. Why not just rely on the operating system?

Well, many of the building blocks just discussed are discretionary. You can use them well or you can use them poorly. It's up to you to ensure that you don't set the permissions on a file to 777, or the ACLs to allow Guest accounts to write. It's up to you to write code that runs well as a normal or even sandboxed user, and it's certainly up to you in these early days of client/server, web, distributed systems, web 2.0, cloud, or whatever comes next to ensure that you're building the right security mechanisms that these newfangled widgets don't yet offer.

File Bugs

Now that you have a list of threats and ways you would like to mitigate them, you're through the complex, security-centered parts of the process. There are just a few more things to do, the first of which is to treat each line of the preceding tables as a bug. You want to treat these as bugs because if you ship software, you've learned to handle bugs in some way. You presumably have a way to track them, prioritize them, and ensure that you're closing them with an appropriate degree of consistency. This will mean something very different to a three-person start-up versus a medical device manufacturer, but both organizations will have a way to handle bugs. You want to tap into that procedure to ensure that threat modeling isn't just a paper exercise.

You can write the text of the bugs in a variety of ways, based on what your organization does. Examples of filing a bug might include the following:

- Someone might use the `/admin/` interface without proper authorization.
- The admin interface lacks proper authorization controls,
- There's no automated security testing for the `/admin/` interface.

Whichever way you go, it's great if you can include the entire threat in the bug, and mark it as a security bug if your bug-tracking tool supports that. (If you're a super-agile scrum shop, use a uniquely colored Post-it for security bugs.)

You'll also have to prioritize the bugs. Elevation-of-privilege bugs are almost always going to fall into the highest priority category, because when they're exploited they lead to so much damage. Denial of service often falls toward the bottom of the stack, but you'll have to consider each bug to determine how to rank it.

Checking Your Work

Validation of your threat model is the last thing you do as part of threat modeling. There are a few tasks to be done here, and it is best to keep them aligned with the order in which you did the previous work. Therefore, the validation tasks include checking the model, checking that you've looked for each threat, and checking your tests. You probably also want to validate the model a second time as you get close to shipping or deploying.

Checking the model

You should ensure that the final model matched what you built. If it doesn't, how can you know that you found the right, relevant threats? To do so, try to arrange a meeting during which everyone looks at the diagram, and answer the following questions:

- Is this complete?
- Is it accurate?
- Does it cover all the security decisions we made?
- Can I start the next version with this diagram without any changes?

If everyone says yes, your diagram is sufficiently up to date for the next step. If not, you'll need to update it.

Updating the Diagram

As you went through the diagram, you might have noticed that it's missing key data. If it were a real system, there might be extra interfaces that were not drawn in, or there might be additional databases. There might be details that you jumped to the whiteboard to draw in. If so, you need to update the diagram with those details. A few rules of thumb are useful as you create or update diagrams:

- Focus on data flow, not control flow.
- Anytime you need to qualify your answer with "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 find yourself needing more detail to explain security-relevant behavior, draw it in.
- Any place you argued over the design or construction of the system, draw in the agreed-on facts. This is an important way to ensure that everyone ended that discussion on the same page. It's especially important for larger teams when not everyone is in the room for the threat model discussions. If they see 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.
- The diagram should tell a story, and support you telling stories while pointing at it.
- Don't draw an eye chart (a diagram with so much detail that you need to squint to read the tiny print).

Diagram Details

If you're wondering how to reconcile that last rule of thumb, don't draw an eye chart, with all the details that a real software project can entail, one technique is to use a sub diagram that shows the details of one particular area. You should look for ways to break things out that make sense for your project. For example, if you have one hyper-complex process, maybe everything in that process should be covered in one diagram, and everything outside it in another. If you have a dispatcher or queuing system, that's a good place to break things up. Your databases or the fail-over system is also a good split. Maybe there's a set of a few elements that really need more detail. All of these are good ways to break things out.

The key thing to remember is that the diagram is intended to help ensure that you understand and can discuss the system. Recall 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 me think about what might go wrong?"

Checking Each Threat

There are two main types of validation activities you should do. The first is checking that you did the right thing with each threat you found. The other is asking if you found all the threats you should find.

In terms of checking that you did the right thing with each threat you did find, the first and foremost question here is "Did I do something with each unique threat I found?" You really don't want to drop stuff on the floor. This is "turning the crank" sort of work. It's rarely glamorous or exciting until you find the thing you overlooked. You can save a lot of time by taking meeting minutes and writing a bug number next to each one, checking that you've addressed each when you do your bug triage.

The next question is "Did I do the right something with each threat?" If you've filed bugs with some sort of security tag, run a query for all the security bugs, and give each one a going-over. This can be as lightweight as reading each bug and asking yourself, "Did I do the right thing?" or you could use a short checklist, an example of which ("Validating threats") is included at the end of this chapter in the "Checklists for Diving in and Threat Modeling" section.

Checking Your Tests

For each threat that you address, ensure you've built a good test to detect the problem. Your test can be a manual testing process or an automated test. Some of these tests will be easy, and others very tricky. For example, if you want to ensure that no static web page under `/beta` can be accessed without the beta cookie, you can build a quick script that retrieves all the pages from your source repository, constructs a URL for it, and tries to collect the page. You could extend the script to send a cookie with each request, and then re-request with an admin cookie. Ideally, that's easy to do in your existing web testing framework. It gets a little more complex with dynamic pages, and a lot more complex when the security risk is something such as SQL injection or secure parsing of user input. There are entire books written on those subjects, not to mention entire books on the subject of testing. The key question you should ask is something like "Are my security tests in line with the other software tests and the sorts of risks that failures expose?"

Threat Modeling on Your Own

You have now walked through your first threat model. Congratulations! Remember though: You're not going to get to Carnegie Hall if you don't practice, practice, practice. That means it is time to do it again, this time on your own, because doing it again is the only way to get better. Pick a system you're working on and threat model it. Follow this simplified, five-step process as you go:

1. Draw a diagram.
2. Use the EoP game to find threats.

3. Address each threat in some way.
4. Check your work with the checklists at the end of this chapter.
5. Celebrate and share your work.

Right now, if you're new to threat modeling, your best bet is to do it often, applying it to the software and systems that matters to you. After threat modeling a few systems, you'll find yourself getting more comfortable with the tools and techniques. For now, the thing to do is practice. Build your first muscles to threat model with.

This brings up the question, what should you threat model next?

What you're working on now is the first place to look for the next system to threat model. If it has a trust boundary of some sort, it may be a good candidate. If it's too simple to have trust boundaries, threat modeling it probably won't be very satisfying. If it has too many boundaries, it may be too big a project to chew on all at once. If you're collaborating closely on it with a few other people who you trust, that may be a good opportunity to play *EoP* with them. If you're working on a large team, or across organizational boundaries, or things are tense, then those people may not be good first collaborators on threat modeling. Start with what you're working on now, unless there are tangible reasons to wait.

Checklists for Diving In and Threat Modeling

There's a lot in this chapter. As you sit down to really do the work yourself, it can be tricky to assess how you're doing. Here are some checklists that are designed to help you avoid the most common problems. Each question is designed to be read aloud and to have an affirmative answer from everyone present. After reading each question out loud, encourage questions or clarification from everyone else involved.

Diagramming

1. Can we tell a story without changing the diagram?
2. Can we tell that story without using words such as "sometimes" or "also"?
3. Can we look at the diagram and see exactly where the software will make a security decision?
4. Does the diagram show all the trust boundaries, such as where different accounts interact? Do you cover all UIDs, all application roles, and all network interfaces?
5. Does the diagram reflect the current or planned reality of the software?

6. Can we see where all the data goes and who uses it?
7. Do we see the processes that move data from one data store to another?

Threats

1. Have we looked for each of the STRIDE threats?
2. Have we looked at each element of the diagram?
3. Have we looked at each data flow in the diagram?

NOTE Data flows are a type of element, but they are sometimes overlooked as people get started, so question 3 is a belt-and-suspenders question to add redundancy. (A belt-and-suspenders approach ensures that a gentleman's pants stay up.)

Validating Threats

1. Have we written down or filed a bug for each threat?
2. Is there a proposed/planned/implemented way to address each threat?
3. Do we have a test case per threat?
4. Has the software passed the test?

Summary

Any technical professional can learn to threat model. Threat modeling involves the intersection of two models: a model of what can go wrong (threats), applied to a model of the software you're building or deploying, which is encoded in a diagram. One model of threats is STRIDE: spoofing, tampering, repudiation, information disclosure, denial of service, and elevation of privilege. This model of threats has been made into the *Elevation of Privilege* game, which adds structure and hints to the model.

With a whiteboard diagram and a copy of *Elevation of Privilege*, developers can threat model software that they're building, systems administrators can threat model software they're deploying or a system they're constructing, and security professionals can introduce threat modeling to those with skillsets outside of security.

It's important to address threats, and the STRIDE threats are the inverse of properties you want. There are mitigation strategies and techniques for developers and for systems administrators.

Once you've created a threat model, it's important to check your work by making sure you have a good model of the software in an up-to-date diagram, and that you've checked each threat you've found.