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 `http://www.microsoft.com/security/sdl/adopt/eop.aspx`. 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).

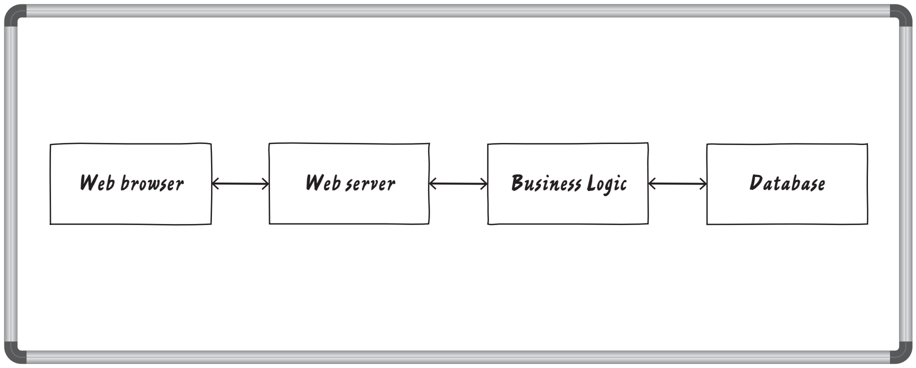


Figure 1-1: A whiteboard diagram

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.

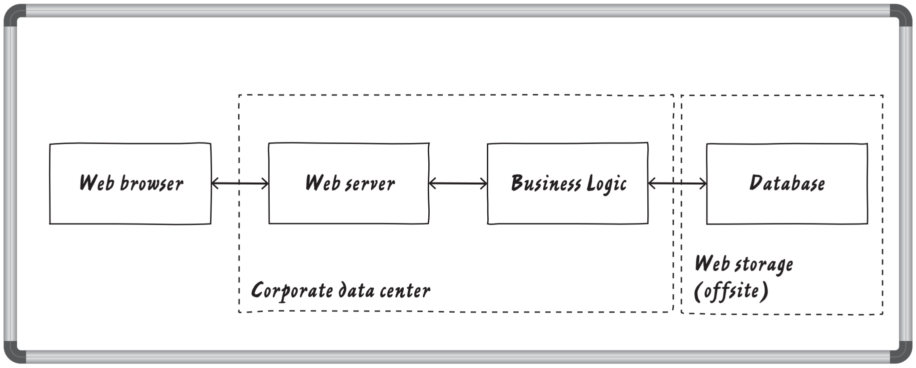


Figure 1-2: Trust boundaries added to a whiteboard diagram

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.)

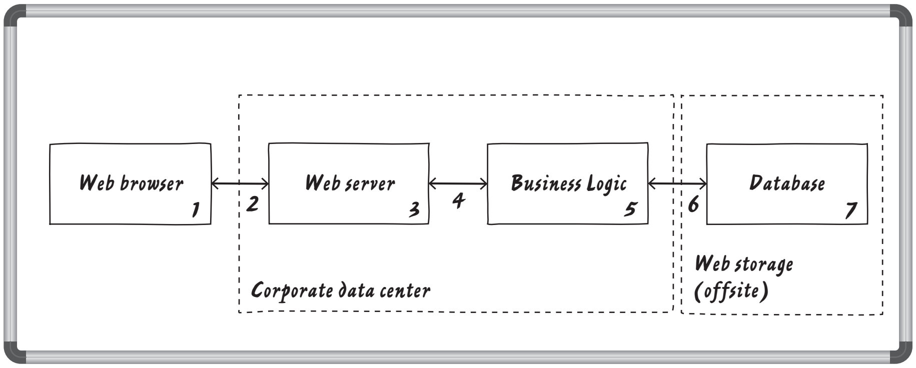


Figure 1-3: Numbers and trust boundaries added to a whiteboard diagram

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.

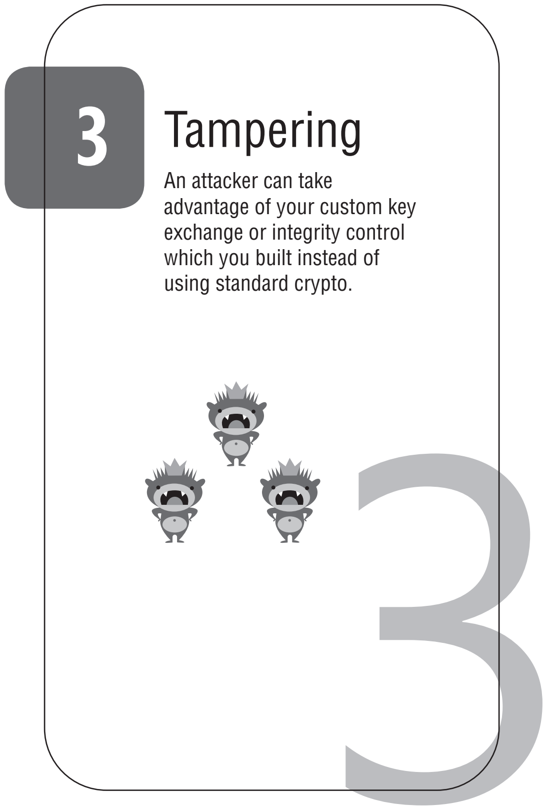

3
Tampering
An attacker can take advantage of your custom key exchange or integrity control which you built instead of using standard crypto.
3

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).
- **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.
- **Information Disclosure** is about exposing information to people who are not authorized to see it.
- **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 vertebrae.)
- **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.

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.

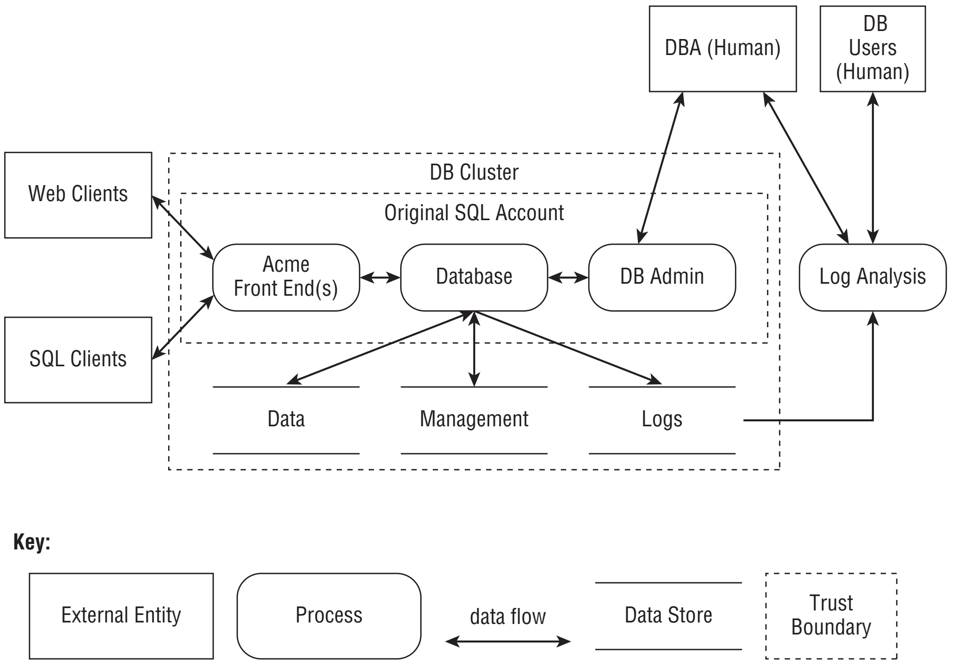


Figure 2-1: Data flow diagram of the Acme/SQL database

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."

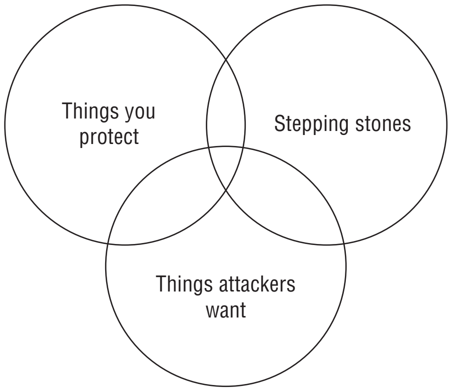


Figure 2-2: The overlapping definitions of assets

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.

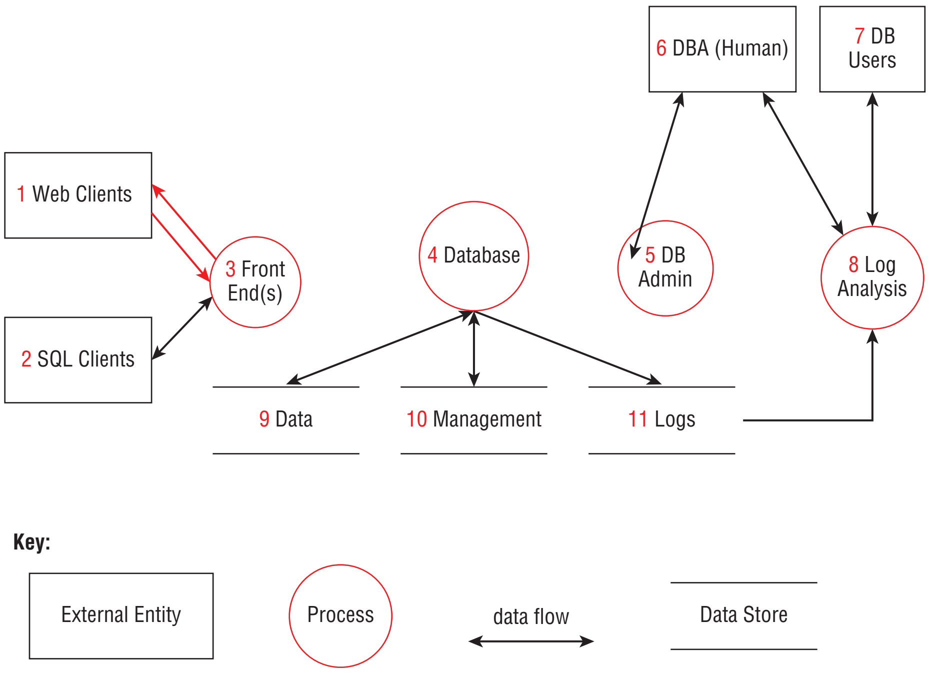


Figure 2-3: A classic DFD model

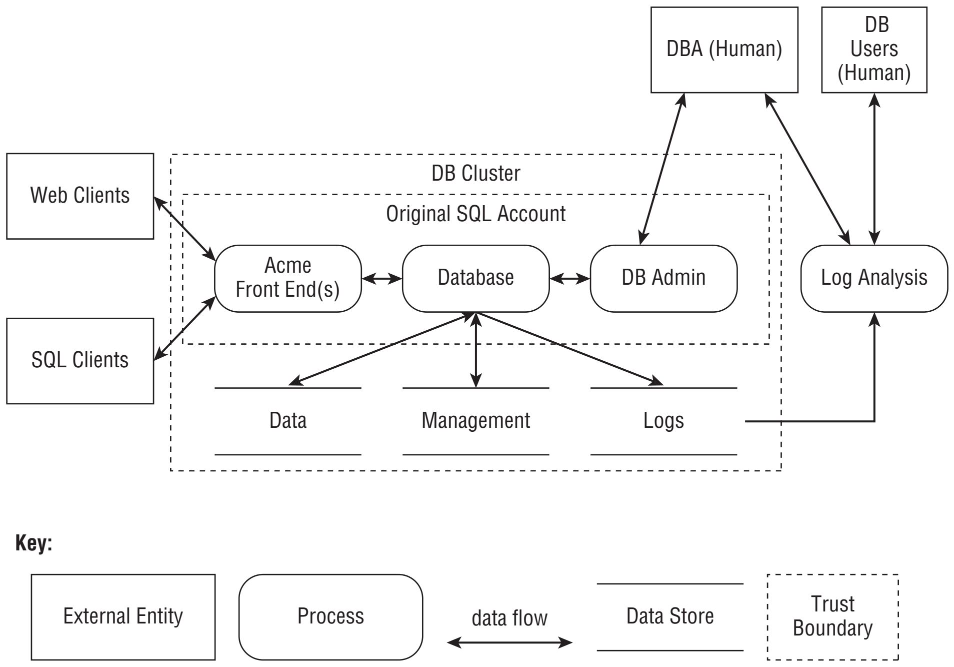


Figure 2-4: A modern DFD model (previously shown as Figure 2-1)

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."

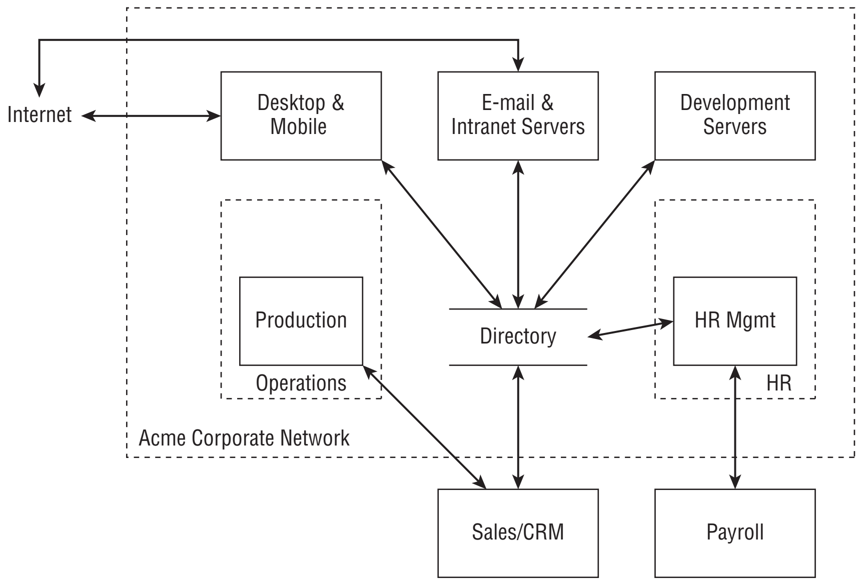


Figure 2-5: An operational network model

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.

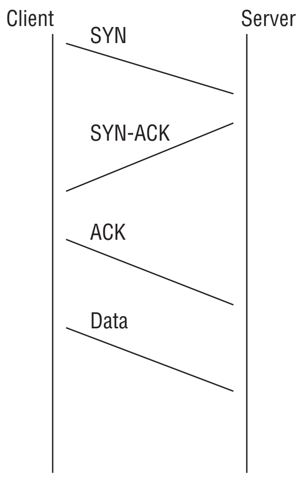


Figure 2-6: Swim lane diagram (showing the start of a TCP connection)

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.

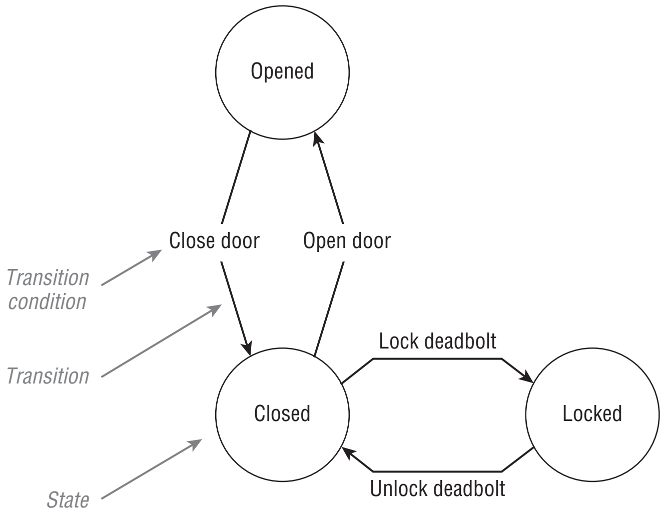


Figure 2-7: A state machine diagram

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.

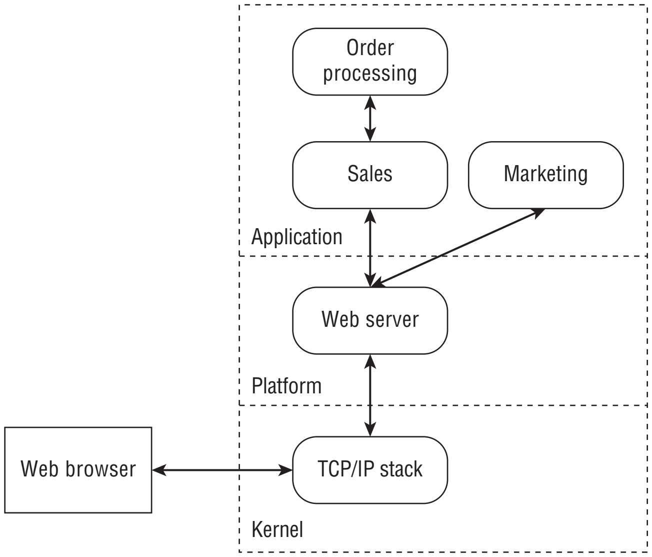


Figure 2-8: Trust boundaries in a web server

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.

CHAPTER

3

STRIDE

As you learned in Chapter 1, "Dive in and Threat Model!," STRIDE is an acronym that stands for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. The STRIDE approach to threat modeling was invented by Loren Kohnfelder and Praerit Garg (Kohnfelder, 1999). This framework and mnemonic was designed to help people developing software identify the types of attacks that software tends to experience.

The method or methods you use to think through threats have many different labels: finding threats, threat enumeration, threat analysis, threat elicitation, threat discovery. Each connotes a slightly different flavor of approach. Do the threats exist in the software or the diagram? Then you're finding them. Do they exist in the minds of the people doing the analysis? Then you're doing analysis or elicitation. No single description stands out as always or clearly preferable, but this book generally talks about finding threats as a superset of all these ideas. Using STRIDE is more like an elicitation technique, with an expectation that you or your team understand the framework and know how to use it. If you're not familiar with STRIDE, the extensive tables and examples are designed to teach you how to use it to discover threats.

This chapter explains what STRIDE is and why it's useful, including sections covering each component of the STRIDE mnemonic. Each threat-specific section provides a deeper explanation of the threat, a detailed table of examples for that threat, and then a discussion of the examples. The tables and examples are designed to teach you how to use STRIDE to discover threats. You'll also learn about approaches built on STRIDE: STRIDE-per-element, STRIDE-per-interaction, and DESIST. The other approach built on STRIDE, the *Elevation of Privilege* game, is covered in Chapters 1, "Dive In and Threat Model!" and 12, "Requirements Cookbook," and Appendix C, "Attacker Lists."

Understanding STRIDE and Why It's Useful

The STRIDE threats are the opposite of some of the properties you would like your system to have: authenticity, integrity, non-repudiation, confidentiality, availability, and authorization. Table 3-1 shows the STRIDE threats, the corresponding property that you'd like to maintain, a definition, the most typical victims, and examples.

Table 3-1: The STRIDE Threats

THREAT	PROPERTY VIOLATED	THREAT DEFINITION	TYPICAL VICTIMS	EXAMPLES
Spoofing	Authentication	Pretending to be something or someone other than yourself	Processes, external entities, people	Falsely claiming to be Acme.com, winsock .dll, Barack Obama, a police officer, or the Nigerian Anti-Fraud Group
Tampering	Integrity	Modifying something on disk, on a network, or in memory	Data stores, data flows, processes	Changing a spreadsheet, the binary of an important program, or the contents of a database on disk; modifying, adding, or removing packets over a network, either local or far across the Internet, wired or wireless; changing either the data a program is using or the running program itself

THREAT	PROPERTY VIOLATED	THREAT DEFINITION	TYPICAL VICTIMS	EXAMPLES
Repudiation	Non-Repudiation	Claiming that you didn't do something, or were not responsible. Repudiation can be honest or false, and the key question for system designers is, what evidence do you have?	Process	Process or system: "I didn't hit the big red button" or "I didn't order that Ferrari." Note that repudiation is somewhat the odd-threat-out here; it transcends the technical nature of the other threats to the business layer.
Information Disclosure	Confidentiality	Providing information to someone not authorized to see it	Processes, data stores, data flows	The most obvious example is allowing access to files, e-mail, or databases, but information disclosure can also involve filenames ("Termination for John Doe.docx"), packets on a network, or the contents of program memory.
Denial of Service	Availability	Absorbing resources needed to provide service	Processes, data stores, data flows	A program that can be tricked into using up all its memory, a file that fills up the disk, or so many network connections that real traffic can't get through
Elevation of Privilege	Authorization	Allowing someone to do something they're not authorized to do	Process	Allowing a normal user to execute code as admin; allowing a remote person without any privileges to run code

In Table 3-1, "typical victims" are those most likely to be victimized: For example, you can spoof a program by starting a program of the same name, or by putting a program with that name on disk. You can spoof an endpoint on the same machine by squatting or splicing. You can spoof users by capturing their authentication info by spoofing a site, by assuming they reuse credentials across sites, by brute forcing (online or off) or by elevating privilege on their machine. You can also tamper with the authentication database and then spoof with falsified credentials.

Note that as you're using STRIDE to look for threats, you're simply enumerating the things that might go wrong. The exact mechanisms for how it can go wrong are something you can develop later. (In practice, this can be easy or it can be very challenging. There might be defenses in place, and if you say, for example, "Someone could modify the management tables," someone else can say, "No, they can't because...") It can be useful to record those possible attacks, because even if there is a mitigation in place, that mitigation is a testable feature, and you should ensure that you have a test case.

You'll sometimes hear STRIDE referred to as "STRIDE categories" or "the STRIDE taxonomy." This framing is not helpful because STRIDE was not intended as, nor is it generally useful for, categorization. It is easy to find things that are hard to categorize with STRIDE. For example, earlier you learned about tampering with the authentication database and then spoofing. Should you record that as a tampering threat or a spoofing threat? The simple answer is that it doesn't matter. If you've already come up with the attack, why bother putting it in a category? The goal of STRIDE is to help you find attacks. Categorizing them might help you figure out the right defenses, or it may be a waste of effort. Trying to use STRIDE to categorize threats can be frustrating, and those efforts cause some people to dismiss STRIDE, but this is a bit like throwing out the baby with the bathwater.

Spoofing Threats

Spoofing is pretending to be something or someone other than yourself. Table 3-1 includes the examples of claiming to be Acme.com, winsock.dll, Barack Obama, or the Nigerian Anti-Fraud Office. Each of these is an example of a different subcategory of spoofing. The first example, pretending to be Acme.com (or Google.com, etc.) entails spoofing the identity of an entity across a network. There is no mediating authority that takes responsibility for telling you that Acme.com is the site I mean when I write these words. This differs from the second example, as Windows includes a `winsock.dll`. You should be able to ask the operating system to act as a mediating authority and get you to `winsock`. If you have your own DLLs, then you need to ensure that you're opening them with the appropriate path (`%installdir%\dll`); otherwise, someone might substitute one in a working directory, and get your code to do what they want. (Similar issues exist with unix and `LD_PATH`.) The third example, spoofing Barack Obama, is an instance of pretending to be a specific person. Contrast that with the fourth example, pretending to be the President of the United States or the Nigerian Anti-Fraud Office. In those cases, the attacker is pretending to be in a role. These spoofing threats are laid out in Table 3-2.

Table 3-2: Spoofing Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Spoofing a process on the same machine	Creates a file before the real process	
	Renaming/linking	Creating a Trojan "su" and altering the path
	Renaming	Naming your process "sshd"
Spoofing a file	Creates a file in the local directory	This can be a library, executable, or config file.
	Creates a link and changes it	From the attacker's perspective, the change should happen between the link being checked and the link being accessed.
	Creates many files in the expected directory	Automation makes it easy to create 10,000 files in `/tmp`, to fill the space of files called `/tmp /"pid.NNNN,` or similar.
Spoofing a machine	ARP spoofing	
	IP spoofing	
	DNS spoofing	Forward or reverse
	DNS Compromise	Compromise TLD, registrar or DNS operator
	IP redirection	At the switch or router level
Spoofing a person	Sets e-mail display name	
	Takes over a real account	
Spoofing a role	Declares themselves to be that role	Sometimes opening a special account with a relevant name

Spoofing a Process or File on the Same Machine

If an attacker creates a file before the real process, then if your code is not careful to create a new file, the attacker may supply data that your code interprets, thinking that your code (or a previous instantiation or thread) wrote that data, and it can be trusted. Similarly, if file permissions on a pipe, local procedure call, and so on, are not managed well, then an attacker can create that endpoint, confusing everything that attempts to use it later.

Spoofing a process or file on a remote machine can work either by creating spoofed files or processes on the expected machine (possibly having taken admin rights) or by pretending to be the expected machine, covered next.

Spoofing a Machine

Attackers can spoof remote machines at a variety of levels of the network stack. These spoofing attacks can influence your code's view of the world as a client, server, or peer. They can spoof ARP requests if they're local, they can spoof IP packets to make it appear that they're coming from somewhere they are not, and they can spoof DNS packets. DNS spoofing can happen when you do a forward or reverse lookup. An attacker can spoof a DNS reply to a forward query they expect you to make. They can also adjust DNS records for machines they control such that when your code does a reverse lookup (translating IP to FQDN) their DNS server returns a name in a domain that they do not control—for example, claiming that 10.1.2.3 is `update.microsoft.com`. Of course, once attackers have spoofed a machine, they can either spoof or act as a man-in-the-middle for the processes on that machine. Second-order variants of this threat involve stealing machine authenticators such as cryptographic keys and abusing them as part of a spoofing attack.

Attackers can also spoof at higher layers. For example, *phishing attacks* involve many acts of spoofing. There's usually spoofing of e-mail from "your" bank, and spoofing of that bank's website. When someone falls for that e-mail, clicks the link and visits the bank, they then enter their credentials, sending them to that spoofed website. The attacker then engages in one last act of spoofing: They log into your bank account and transfer your money to themselves or an accomplice. (It may be one attacker, or it may be a set of attackers, contracting with one another for services rendered.)

Spoofing a Person

Major categories of spoofing people include access to the person's account and pretending to be them through an alternate account. Phishing is a common way to get access to someone else's account. However, there's often little to prevent anyone from setting up an account and pretending to be you. For example, an attacker could set up accounts on sites like LinkedIn, Twitter, or Facebook and pretend to be you, the Adam Shostack who wrote this book, or a rich and deposed prince trying to get their money out of the country.

Tampering Threats

Tampering is modifying something, typically on disk, on a network, or in memory. This can include changing data in a spreadsheet (using either a program such as Excel or another editor), changing a binary or configuration file on disk, or modifying a more complex data structure, such as a database on disk. On a network, packets can be added, modified, or removed. It's sometimes easier to add packets than to edit them as they fly by, and programs are remarkably bad about handling extra copies of data securely. More examples of tampering are in Table 3-3.

Table 3-3: Tampering Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Tampering with a file	Modifies a file they own and on which you rely	
	Modifies a file you own	
	Modifies a file on a file server that you own	
	Modifies a file on their file server	Loads of fun when you include files from remote domains
	Modifies a file on their file server	Ever notice how much XML includes remote schemas?
	Modifies links or redirects	
Tampering with memory	Modifies your code	Hard to defend against once the attacker is running code as the same user
	Modifies data they've supplied to your API	Pass by value, not by reference when crossing a trust boundary
Tampering with a network	Redirects the flow of data to their machine	Often stage 1 of tampering
	Modifies data flowing over the network	Even easier and more fun when the network is wireless (WiFi, 3G, et cetera)
	Enhances spoofing attacks	

Tampering with a File

Attackers can modify files wherever they have write permission. When your code has to rely on files others can write, there's a possibility that the file was written maliciously. While the most obvious form of tampering is on a local disk, there are also plenty of ways to do this when the file is remotely included, like most of the JavaScript on the Internet. The attacker can breach your security by breaching someone else's site. They can also (because of poor privileges, spoofing, or elevation of privilege) modify files you own. Lastly, they can modify links or redirects of various sorts. Links are often left out of integrity checks. There's a somewhat subtle variant of this when there are caches between things you control (such as a server) and things you don't (such as a web browser on the other side of the Internet). For example, *cache poisoning attacks* insert data into web caches through poor security controls at caches (OWASP, 2009).

Tampering with Memory

Attackers can modify your code if they're running at the same privilege level. At that point, defense is tricky. If your API handles data by reference (a pattern often chosen for speed), then an attacker can modify it after you perform security checks.

Tampering with a Network

Network tampering often involves a variety of tricks to bring the data to the attacker's machine, where he forwards some data intact and some data modified. However, tricks to bring you the data are not always needed; with radio interfaces like WiFi and Bluetooth, more and more data flow through the air. Many network protocols were designed with the assumption you needed special hardware to create or read arbitrary packets. The requirement for special hardware was the defense against tampering (and often spoofing). The rise of software-defined radio (SDR) has silently invalidated the need for special hardware. It is now easy to buy an inexpensive SDR unit that can be programmed to tamper with wireless protocols.

Repudiation Threats

Repudiation is claiming you didn't do something, or were not responsible for what happened. People can repudiate honestly or deceptively. Given the increasing knowledge often needed to understand the complex world, those honestly repudiating may really be exposing issues in your user experiences or service architectures. Repudiation threats are a bit different from other security threats, as they often appear at the business layer. (That is, above the network layer such as TCP/IP, above the application layer such as HTTP/HTML, and where the business logic of buying products would be implemented.)

Repudiation threats are also associated with your logging system and process. If you don't have logs, don't retain logs, or can't analyze logs, repudiation threats are hard to dispute. There is also a class of attacks in which attackers will drop data in the logs to make log analysis tricky. For example, if you display your logs in HTML and the attacker sends `</tr>` or `</html>`, your log display needs to treat those as data, not code. More repudiation threats are shown in Table 3-4.

Table 3-4: Repudiation Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Repudiating an action	Claims to have not clicked	Maybe they really did
	Claims to have not received	Receipt can be strange; does mail being downloaded by your phone mean you've read it? Did a network proxy pre-fetch images? Did someone leave a package on the porch?
	Claims to have been a fraud victim	
	Uses someone else's account	
	Uses someone else's payment instrument without authorization	
Attacking the logs	Notices you have no logs	
	Puts attacks in the logs to confuse logs, log-reading code, or a person reading the logs	

Attacking the Logs

Again, if you don't have logs, don't retain logs, or can't analyze logs, repudiation actions are hard to dispute. So if you aren't logging, you probably need to start. If you have no log centralization or analysis capability, you probably need that as well. If you don't properly define what you will be logging, an attacker may be able to break your log analysis system. It can be challenging to work through the layers of log production and analysis to ensure reliability, but if you don't, it's easy to have attacks slip through the cracks or inconsistencies.

Repudiating an Action

When you're discussing repudiation, it's helpful to discuss "someone" rather than "an attacker." You want to do this because those who repudiate are often not actually attackers, but people who have been failed by technology or process. Maybe they really didn't click (or didn't perceive that they clicked). Maybe the spam filter really did eat that message. Maybe UPS didn't deliver, or maybe UPS delivered by leaving the package on a porch. Maybe someone claims to have been a victim of fraud when they really were not (or maybe someone else in a household used their credit card, with or without their knowledge). Good technological systems that both authenticate and log well can make it easier to handle repudiation issues.

Information Disclosure Threats

Information disclosure is about allowing people to see information they are not authorized to see. Some information disclosure threats are shown in Table 3-5.

Table 3-5: Information Disclosure Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Information disclosure against a process	Extracts secrets from error messages	
	Reads the error messages from username/passwords to entire database tables	
	Extracts machine secrets from error cases	Can make defense against memory corruption such as ASLR far less useful
	Extracts business/personal secrets from error cases	
Information disclosure against data stores	Takes advantage of inappropriate or missing ACLs	
	Takes advantage of bad database permissions	
	Finds files protected by obscurity	
	Finds crypto keys on disk (or in memory)	
	Sees interesting information in filenames	
	Reads files as they traverse the network	
	Gets data from logs or temp files	
	Gets data from swap or other temp storage	
	Extracts data by obtaining device, changing OS	

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Information disclosure against a data flow	Reads data on the network	
	Redirects traffic to enable reading data on the network	
	Learns secrets by analyzing traffic	
	Learns who's talking to whom by watching the DNS	
	Learns who's talking to whom by social network info disclosure	

Information Disclosure from a Process

Many instances in which a process will disclose information are those that inform further attacks. A process can do this by leaking memory addresses, extracting secrets from error messages, or extracting design details from error messages. Leaking memory addresses can help bypass ASLR and similar defenses. Leaking secrets might include database connection strings or passwords. Leaking design details might mean exposing anti-fraud rules like "your account is too new to order a diamond ring."

Information Disclosure from a Data Store

As data stores, well, store data, there's a profusion of ways they can leak it. The first set of causes are failures to properly use security mechanisms. Not setting permissions appropriately or hoping that no one will find an obscure file are common ways in which people fail to use security mechanisms. Cryptographic keys are a special case whereby information disclosure allows additional attacks. Files read from a data store over the network are often readable as they traverse the network.

An additional attack, often overlooked, is data in filenames. If you have a directory named "May 2013 layoffs," the filename itself, "Termination Letter for Alice.docx," reveals important information.

There's also a group of attacks whereby a program emits information into the operating environment. Logs, temp files, swap, or other places can contain data. Usually, the OS will protect data in swap, but for things like crypto keys, you should use OS facilities for preventing those from being swapped out.

Lastly, there is the class of attacks whereby data is extracted from the device using an operating system under the attacker's control. Most commonly (in 2013), these attacks affect USB keys, but they also apply to CDs, backup tapes, hard drives, or stolen laptops or servers. Hard drives are often decommissioned without full data deletion. (You can address the need to delete data from hard drives by buying a hard drive chipper or smashing machine, and since such machines are awesome, why on earth wouldn't you?)

Information Disclosure from a Data Flow

Data flows are particularly susceptible to information disclosure attacks when information is flowing over a network. However, data flows on a single machine can still be attacked, particularly when the machine is shared by cloud co-tenants or many mutually distrustful users of a compute server. Beyond the simple reading of data on the network, attackers might redirect traffic to themselves (often by spoofing some network control protocol) so they can see it when they're not on the normal path. It's also possible to obtain information even when the network traffic itself is encrypted. There are a variety of ways to learn secrets about who's talking to whom, including watching DNS, friend activity on a site such as LinkedIn, or other forms of social network analysis.

NOTE Security mavens may be wondering if side channel attacks and covert channels are going to be mentioned. These attacks can be fun to work on (and side channels are covered a bit in Chapter 16, "Threats to Cryptosystems"), but they are not relevant until you've mitigated the issues covered here.

Denial-of-Service Threats

Denial-of-service attacks absorb a resource that is needed to provide service. Examples are described in Table 3-6.

Table 3-6: Denial-of-Service Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Denial of service against a process	Absorbs memory (RAM or disk)	
	Absorbs CPU	
	Uses process as an amplifier	
Denial of service against a data store	Fills data store up	
	Makes enough requests to slow down the system	
Denial of service against a data flow	Consumes network resources	

Denial-of-service attacks can be split into those that work while the attacker is attacking (say, filling up bandwidth) and those that persist. Persistent attacks can remain in effect until a reboot (for example, `while(1){fork();}`), or even past a reboot (for example, filling up a disk). Denial-of-service attacks can also be divided into amplified and unamplified. Amplified attacks are those whereby small attacker effort results in a large impact. An example would take advantage of the old unix chargen service, whose purpose was to generate a semi-random character scheme for testing. An attacker could spoof a single packet from the chargen port on machine A to the chargen port on machine B. The hilarity continues until someone pulls a network cable.

Elevation of Privilege Threats

Elevation of privilege is allowing someone to do something they're not authorized to do—for example, allowing a normal user to execute code as admin, or allowing a remote person without any privileges to run code. Two important ways to elevate privileges involve corrupting a process and getting past authorization checks. Examples are shown in Table 3-7.

Table 3-7: Elevation of Privilege Threats

THREAT EXAMPLES	WHAT THE ATTACKER DOES	NOTES
Elevation of privilege against a process by corrupting the process	Send inputs that the code doesn't handle properly	These errors are very common, and are usually high impact.
	Gains access to read or write memory inappropriately	Writing memory is (hopefully obviously) bad, but reading memory can enable further attacks.
Elevation through missed authorization checks		
Elevation through buggy authorization checks		Centralizing such checks makes bugs easier to manage
Elevation through data tampering	Modifies bits on disk to do things other than what the authorized user intends	

Elevate Privileges by Corrupting a Process

Corrupting a process involves things like smashing the stack, exploiting data on the heap, and a whole variety of exploitation techniques. The impact of these techniques is that the attacker gains influence or control over a program's control flow. It's important to understand that these exploits are not limited to the attack surface. The first code that attacker data can reach is, of course, an important target. Generally, that code can only validate data against a limited subset of purposes. It's important to trace the data flows further to see where else elevation of privilege can take place. There's a somewhat unusual case whereby a program relies on and executes things from shared memory, which is a trivial path for elevation if everything with permissions to that shared memory is not running at the same privilege level.

Elevate Privileges through Authorization Failures

There is also a set of ways to elevate privileges through authorization failures. The simplest failure is to not check authorization on every path. More complex for an attacker is taking advantage of buggy authorization checks. Lastly, if a program relies on other programs, configuration files, or datasets being trustworthy, it's important to ensure that permissions are set so that each of those dependencies is properly secured.

Extended Example: STRIDE Threats against Acme-DB

This extended example discusses how STRIDE threats could manifest against the Acme/SQL database described in Chapter 1, "Dive In and Threat Model!" and 2, "Strategies for Threat Modeling," and shown in Figure 2-1. You'll first look at these threats by STRIDE category, and then examine the same set according to who can address them.

Spoofing

- A web client could attempt to log in with random credentials or stolen credentials, as could a SQL client.
- If you assume that the SQL client is the one you wrote and allow it to make security decisions, then a spoofed (or tampered with) client could bypass security checks.
- The web client could connect to a false (spoofed) front end, and end up disclosing credentials.
- A program could pretend to be the database or log analysis program, and try to read data from the various data stores.

Tampering

- Someone could also tamper with the data they're sending, or with any of the programs or data files.
- Someone could tamper with the web or SQL clients. (This is nominally out of scope, as you shouldn't be trusting external entities anyway.)

NOTE These threats, once you consider them, can easily be addressed with operating system permissions. More challenging is what can alter what data within the database. Operating system permissions will only help a little there; the database will need to implement an access control system of some sort.

Repudiation

- The customers using either SQL or web clients could claim not to have done things. These threats may already be mitigated by the presence of logs and log analysis. So why bother with these threats? They remind you that you need to configure logging to be on, and that you need to log the "right things," which probably include successes and failures of authentication attempts, access attempts, and in particular, the server needs to track attempts by clients to access or change logs.

Information Disclosure

- The most obvious information disclosure issues occur when confidential information in the database is exposed to the wrong client. This information may be either data (the contents of the salaries table) or metadata (the existence of the termination plans table). The information disclosure may be accidental (failure to set an ACL) or malicious (eavesdropping on the network). Information disclosure may also occur by the front end(s)—for example, an error message like "Can't connect to database foo with password bar!"
- The database files (partitions, SAN attached storage) need to be protected by the operating system and by ACLs for data within the files.
- Logs often store confidential information, and therefore need to be protected.

Denial of Service

- The front ends could be overwhelmed by random or crafted requests, especially if there are anonymous (or free) web accounts that can craft requests designed to be slow to execute.
- The network connections could be overwhelmed with data.
- The database or logs could be filled up.

- If the network between the main processes, or the processes and databases, is shared, it may become congested.

Elevation of Privilege

- Clients, either web or SQL, could attempt to run queries they're not authorized to run.
- If the client is enforcing security, then anyone who tampers with their client or its network stream will be able to run queries of their choice.
- If the database is capable of running arbitrary commands, then that capability is available to the clients.
- The log analysis program (or something pretending to be the log analysis program) may be able to run arbitrary commands or queries.

NOTE The log analysis program may be thought of as trusted, but it's drawn outside the trust boundaries. So either the thinking or the diagram (in Figure 2-1) is incorrect.

- If the DB cluster is connected to a corporate directory service and no action is taken to restrict who can log in to the database servers (or file servers), then anyone in the corporate directory, including perhaps employees, contractors, build labs, and partners can make changes on those systems.

NOTE The preceding lists in this extended example are intended to be illustrative; other threats may exist.

It is also possible to consider these threats according to the person or team that must address them, divided between Acme and its customers. As shown in Table 3-8, this illustrates the natural overlap of threat and mitigation, foreshadowing the Part III, "Managing and Addressing Threats" on how to mitigate threats. It also starts to enumerate things that are not requirements for Acme/SQL. These non-requirements should be documented and provided to customers, as covered in Chapter 12. In this table, you're seeing more and more actionable threats. As a developer or a systems administrator, you can start to see how to handle these sorts of issues. It's tempting to start to address threats in the table itself, and a natural extension to the table would be a set of ways for each actor to address the threats that apply.

Table 3-8: Addressing Threats According to Who Handles Them

THREAT	INSTANCES THAT ACME MUST HANDLE	INSTANCES THAT IT DEPARTMENTS MUST HANDLE
Spoofing	Web/SQL/other client brute forcing logins DBA (human) DB users	Web client SQL client DBA (human) DB users
Tampering	Data Management Logs	Front end(s) Database DB admin
Repudiation	Logs (Log analysis must be protected.) Certain actions from web and SQL clients will need careful logging. Certain actions from DBAs will need careful logging.	Logs (Log analysis must be protected.) If DBAs are not fully trusted, a system in another privilege domain to log all commands might be required.
Information disclosure	Data, management, and logs must be protected. Front ends must implement access control. Only the front ends should be able to access the data.	ACLs and security groups must be managed. Backups must be protected.
Denial of service	Front ends must be designed to minimize DoS risks.	The system must be deployed with sufficient resources.

Continues

Table 3-8 (*continued*)

THREAT	INSTANCES THAT ACME MUST HANDLE	INSTANCES THAT IT DEPARTMENTS MUST HANDLE
Elevation of privilege	Trusting client The DB should support prepared statements to make injection harder. No "run this command" tools should be in the default install. No default way to run commands on the server, and calls like `exec()` and `system()` must be permissioned and configurable if they exist.	Inappropriately trusting clients that are written locally Configure the DB appropriately.

STRIDE Variants

STRIDE can be a very useful mnemonic when looking for threats, but it's not perfect. In this section, you'll learn about variants of STRIDE that may help address some of its weaknesses.

STRIDE-per-Element

STRIDE-per-element makes STRIDE more prescriptive by observing that certain threats are more prevalent with certain elements of a diagram. For example, a data store is unlikely to spoof another data store (although running code can be confused as to which data store it's accessing.) By focusing on a set of threats against each element, this approach makes it easier to find threats. For example, Microsoft uses Table 3-9 as a core part of its Security Development Lifecycle threat modeling training.

Table 3-9: STRIDE-per-Element

	S	T	R	I	D	E
External Entity	x		x			
Process	x	x	x	x	x	x
Data Flow		x		x	x	
Data Store		x	?	x	x	

Applying this chart, you can focus threat analysis on how an attacker might tamper with, read data from, or prevent access to a data flow. For example, if data is flowing over a network such as Ethernet, it's trivial for someone attached to that same Ethernet to read all the content, modify it, or send a flood of packets to cause a TCP timeout. You might argue that you have some form of network segmentation, and that may mitigate the threats sufficiently for you. The question mark under repudiation indicates that logging data stores are involved in addressing repudiation, and sometimes logs will come under special attack to allow repudiation attacks.

The threat is to the element listed in Table 3-9. Each element is the victim, not the perpetrator. Therefore, if you're tampering with a data store, the threat is to the data store and the data within. If you're spoofing in a way that affects a process, then the process is the victim. So, spoofing by tampering with the network is really a spoof of the endpoint, regardless of the technical details. In other words, the other endpoint (or endpoints) are confused about what's at the other end of the connection. The chart focuses on spoofing of a process, not spoofing of the data flow. Of course, if you happen to find spoofing when looking at the data flow, obviously you should record the threat so you can address it, not worry about what sort of threat it is. STRIDE-per-element has the advantage of being prescriptive, helping you identify what to look for where without being a checklist of the form "web component: XSS, XSRF..." In skilled hands, it can be used to find new types of weaknesses in components. In less skilled hands, it can still find many common issues.

STRIDE-per-element does have two weaknesses. First, similar issues tend to crop up repeatedly in a given threat model; second, the chart may not represent your issues. In fact, Table 3-9 is somewhat specific to Microsoft. The easiest place to see this is "information disclosure by external entity," which is a good description of some privacy issues. (It is by no means a complete description of privacy.) However, the table doesn't indicate that this could be a problem. That's because Microsoft has a separate set of processes for analyzing privacy problems. Those privacy processes are outside the security threat modeling space. Therefore, if you're going to adopt this approach, it's worth analyzing whether the table covers the set of issues you care about, and if it doesn't, create a version that suits your scenario. Another place you might see the specificity is that many people want to discuss spoofing of data flows. Should that be part of STRIDE-per-element? The spoofing action is a spoofing of the endpoint, but that description may help some people to look for those threats. Also note that the more "x" marks you add, the closer you come to "consider STRIDE for each element of the diagram." The editors ask if that's a good or bad thing, and it's a fine question. If you want to be comprehensive, this is helpful; if you want to focus on the most likely issues, however, it will likely be a distraction.

So what are the exit criteria for STRIDE-per-element? When you have a threat per checkbox in the STRIDE-per-element table, you are doing reasonably well.

If you circle around and consider threats against your mitigations (or ways to bypass them) you'll be doing pretty well.

STRIDE-per-Interaction

STRIDE-per-element is a simplified approach to identifying threats, designed to be easily understood by the beginner. However, in reality, threats don't show up in a vacuum. They show up in the interactions of the system. STRIDE-per-interaction is an approach to threat enumeration that considers tuples of (*origin*, *destination*, *interaction*) and enumerates threats against them. Initially, another goal of this approach was to reduce the number of things that a modeler would have to consider, but that didn't work out as planned. STRIDE-per-interaction leads to the same number of threats as STRIDE-per-element, but the threats may be easier to understand with this approach. This approach was developed by Larry Osterman and Douglas MacIver, both of Microsoft. The STRIDE-per-interaction approach is shown in Tables 3-10 and 3-11. Both reference two processes, Contoso.exe and Fabrikam.dll. Table 3-10 shows which threats apply to each interaction, and Table 3-11 shows an example of STRIDE per interaction applied to Figure 3-1. The relationships and trust boundaries used for the named elements in both tables are shown in Figure 3-1.

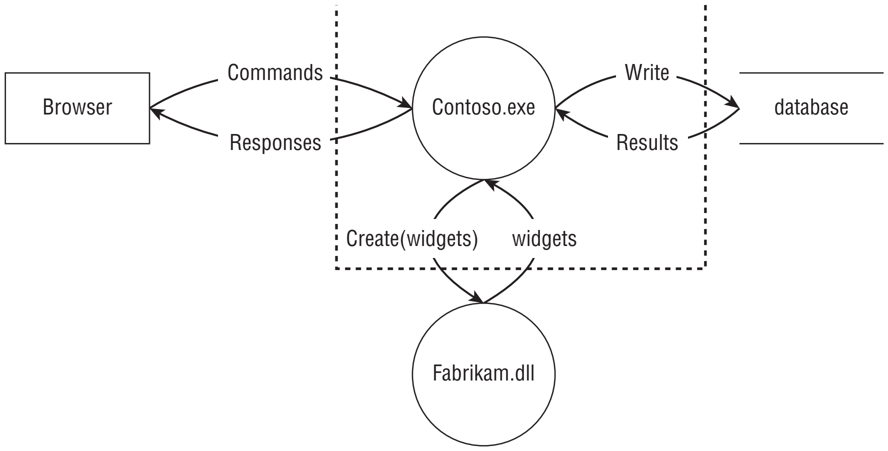


Figure 3-1: The system referenced in Table 3-10

In Table 3-10, the table columns are as follows:

- A number for referencing a line (For example, "Looking at line 2, let's look for spoofing and information disclosure threats.")

- The main element you're looking at
- The interactions that element has
- The STRIDE threats applicable to the interaction

Table 3-10: STRIDE-per-Interaction: Threat Applicability

#	ELEMENT	INTERACTION	S	T	R	I	D	E
1	Process (Contoso)	Process has outbound data flow to data store.	x			x		
2		Process sends output to another process.	x		x	x	x	x
3		Process sends output to external interactor (code).	x		x	x	x	
4		Process sends output to external interactor (human).			x			
5		Process has inbound data flow from data store.	x	x			x	x
6		Process has inbound data flow from a process.	x		x		x	x
7		Process has inbound data flow from external interactor.	x				x	x
8	Data Flow (com-mands/responses)	Crosses machine boundary		x		x	x	
9	Data Store (database)	Process has outbound data flow to data store.		x	x	x	x	
10		Process has inbound data flow from data store.			x	x	x	
11	External Interactor (browser)	External interactor passes input to process.	x		x	x		
12		External interactor gets input from process.	x					

Table 3-11: STRIDE-per-Interaction (Example)

	ELEMENT	INTERACTION	S	T	R	I	D	E
1	Process (Contoso)	Process has outbound data flow to data store.	"Database" is spoofed, and Contoso writes to the wrong place.			P2: Contoso writes information in "database which should not be in database" (e.g., passwords).		
2		Process sends output to other process.	Fabrikam is spoofed, and Contoso writes to the wrong place.		Fabrikam claims not to have been called by Contoso.	P2: Fabrikam is not authorized to receive data.	None unless calls are synchronous	Fabrikam can impersonate Contoso and use its privileges.
3		Process sends output to external interactor (with the interactor being code).	Contoso is confused about the identity of the browser.		Browser disclaims and doesn't acknowledge the output.	P2: Browser gets data it's not authorized to get.	None unless calls are synchronous	
4		Process sends output to external interactor (for a human interactor).			Human disclaims seeing the output.			

Table 3-11 (continued)

ELEMENT	INTERACTION	S	T	R	I	D	E
5	Process has inbound data flow from data store.	"Database" is spoofed, and Contoso reads the wrong data.	Contoso state is corrupted by data read from the data store.			Process state is corrupted by the data retrieved from the data store.	Process internal state is corrupted based on data read from the file, leading to code execution .
6	Process has inbound data flow from a process.	Contoso believes it's getting data from Fabrikam.		Contoso denies getting data from Fabrikam.		Contoso crashes/stops due to Fabrikam interaction.	Fabrikam passes data or args that allow it to change flow of execution of Contoso.
7	Process has inbound data flow from external interactor.	Contoso believes it's getting data from the browser, when in fact it's a random attacker.				Contoso crashes/stops due to browser interaction.	Browser passes data or args that allow it to change flow of execution of Contoso.

Continues

Table 3-11 (*continued*)

	ELEMENT	INTERACTION	S	T	R	I	D	E
8	Data Flow (commands/ responses)	Crosses machine boundary		Data flow is modified by MITM attack.		The contents of the data flow are sniffed on the wire.	The data flow is interrupted by an external entity (e.g., messing with TCP sequence numbers.)	
9	Data Store (database)	Process has outbound data flow to data store.		Database is corrupted.	Contoso claims not to have written to database.	Database reveals information.	Database cannot be written to.	
10		Process has inbound data flow from data store.			Contoso claims not to have read from database.	Database discloses information.	Database cannot be read from.	
11	External Interactor (browser)	External interactor passes input to process.	Contoso is confused about the identity of the browser.		Contoso claims not to have received the data.	~~P2: process not authorized to receive the data~~ (We can't stop it.)		
12		External interactor gets input from process.	Browser is confused about the identity of Contoso.		~~Contoso claims not to have sent the data~~ (Not our problem.)			

When you have a threat per checkbox in the STRIDE-per-interaction table, you are doing reasonably well. If you circle through and consider threats against your mitigations (or ways to bypass them) you'll be doing pretty well.

STRIDE-per-interaction is too complex to use without a reference chart handy. (In contrast, STRIDE is an easy mnemonic, and STRIDE-per-element is simple enough that the chart can be memorized or printed on a wallet card.)

DESIST

DESIST is a variant of STRIDE created by Gunnar Peterson. DESIST stands for Dispute, Elevation of privilege, Spoofing, Information disclosure, Service denial, and Tampering. (Dispute replaces repudiation with a less fancy word, and Service denial replaces Denial of Service to make the acronym work.) Starting from scratch, it might make sense to use DESIST over STRIDE, but after more than a decade of STRIDE, it would be expensive to displace at Microsoft. (CEO of Scorpion Software, Dana Epp, has pointed out that acronyms with repeated letters can be challenging, a point in STRIDE's favor.) Therefore, STRIDE-per-element, rather than DESIST-per-element, exists as the norm. Either way, it's always useful to have mnemonics for helping people look for threats.

Exit Criteria

There are three ways to judge whether you're done finding threats with STRIDE. The easiest way is to see if you have a threat of each type in STRIDE. Slightly harder is ensuring you have one threat per element of the diagram. However, both of these criterion will be reached before you've found all threats. For more comprehensiveness, use STRIDE-per-element, and ensure you have one threat per check.

Not having met these criteria will tell you that you're not done, but having met them is not a guarantee of completeness.

Summary

STRIDE is a useful mnemonic for finding threats against all sorts of technological systems. STRIDE is more useful with a repertoire of more detailed threats to draw on. The tables of threats can provide that for those who are new to security, or act as reference material for security experts (a function also served by Appendix B, "Threat Trees"). There are variants of STRIDE that attempt to add focus and attention. STRIDE-per-element is very useful for this purpose, and can be customized to your needs. STRIDE-per-interaction provides more focus, but requires a crib sheet (or perhaps software) to use. If threat modeling experts were to start over, perhaps DESIST would help us make better ... progress in finding threats.

CHAPTER

4

Attack Trees

As Bruce Schneier wrote in his introduction to the subject, "Attack trees provide a formal, methodical way of describing the security of systems, based on varying attacks. Basically, you represent attacks against a system in a tree structure, with the goal as the root node and different ways of achieving that goal as leaf nodes" (Schneier, 1999).

In this chapter you'll learn about the attack tree building block as an alternative to STRIDE. You can use attack trees as a way to find threats, as a way to organize threats found with other building blocks, or both. You'll start with how to use an attack tree that's provided to you, and from there learn various ways you can create trees. You'll also examine several example and real attack trees and see how they fit into finding threats. The chapter closes with some additional perspective on attack trees.

Working with Attack Trees

Attack trees work well as a building block for threat enumeration in the four-step framework. They have been presented as a full approach to threat modeling (Salter, 1998), but the threat modeling community has learned a lot since then.

There are three ways you can use attack trees to enumerate threats: You can use an attack tree someone else created to help you find threats. You can create a tree to help you think through threats for a project you're working on. Or you can create trees with the intent that others will use them. Creating new trees for general use is challenging, even for security experts.

Using Attack Trees to Find Threats

If you have an attack tree that is relevant to the system you're building, you can use it to find threats. Once you've modeled your system with a DFD or other diagram, you use an attack tree to analyze it. The attack elicitation task is to iterate over each node in the tree and consider if that issue (or a variant of that issue) impacts your system. You might choose to track either the threats that apply or each interaction. If your system or trees are complex, or if process documentation is important, each interaction may be helpful, but otherwise that tracking may be distracting or tedious. You can use the attack trees in this chapter or in Appendix B "Threat Trees" for this purpose.

If there's no tree that applies to your system, you can either create one, or use a different threat enumeration building block.

Creating New Attack Trees

If there are no attack trees that you can use for your system, you can create a project-specific tree. A project-specific tree is a way to organize your thinking about threats. You may end up with one or more trees, but this section assumes you're putting everything in one tree. The same approach enables you to create trees for a single project or trees for general use.

The basic steps to create an attack tree are as follows:

1. Decide on a representation.
2. Create a root node.
3. Create subnodes.
4. Consider completeness.
5. Prune the tree.
6. Check the presentation.

Decide on a Representation

There are AND trees, where the state of a node depends on all of the nodes below it being true, and OR trees, where a node is true if any of its subnodes are true. You need to decide, will your tree be an AND or an OR tree? (Most will be OR trees.) Your tree can be created or presented graphically or as an outline. See the section "Representing a Tree" later in this chapter for more on the various forms of representation.

Create a Root Node

To create an attack tree, start with a root node. The root node can be the component that prompts the analysis, or an adversary's goal. Some attack trees use the problematic state (rather than the goal) as the root. Which you should use is a matter of preference. If the root node is a component, the subnodes should be labeled with what can go wrong for the node. If the root node is an attacker goal, consider ways to achieve that goal. Each alternative way to achieve the goal should be drawn in as a subnode.

The guidance in "Toward a Secure System Engineering Methodology" (Salter, 1999) is helpful to security experts; however, it doesn't shed much light on how to actually generate the trees, comparative advice about what a root node should be (in other words, whether it's a goal or a system component and, most important, when one is better than the other), or how to evaluate trees in a structured fashion that would be suitable for those who are not security experts. To be prescriptive:

- Create a root node with an attacker goal or high-impact action.
- Use OR trees.
- Draw them into a grid that the eye can track linearly.

Create Subnodes

You can create subnodes by brainstorming, or you can look for a structured way to find more nodes. The relation between your nodes can be AND or OR, and you'll have to make a choice and communicate it to those who are using your tree. Some possible structures for first-level subnodes include:

- Attacking a system:
  - physical access
  - subvert software
  - subvert a person
- Attacking a system via:
  - People
  - Process
  - Technology
- Attacking a product during:
  - Design
  - Production
  - Distribution
  - Usage
  - Discard

You can use these as a starting point, and make them more specific to your system. Iterate on the trees, adding subnodes as appropriate.

NOTE **Here the term subnode is used to include leaf (end) nodes and nodes with children, because as you create something you may not always know whether it is a leaf or whether it has more branches.**

Consider Completeness

For this step, you want to determine whether your set of attack trees is complete enough. For example, if you are using components, you might need to add additional trees for additional components. You can also look at each node and ask "is there another way that could happen?" If you're using attacker motivations, consider additional attackers or motivations. The lists of attackers in Appendix C "Attacker Lists" can be used as a basis.

An attack tree can be checked for quality by iterating over the nodes, looking for additional ways to reach the goal. It may be helpful to use STRIDE, one of the attack libraries in the next chapter, or a literature review to help you check the quality.

Prune the Tree

In this step, go through each node in the tree and consider whether the action in each subnode is prevented or duplicative. (An attack that's worth putting in a tree will generally only be prevented in the context of a project.) If an attack is prevented, by some mitigation you can mark those nodes to indicate that they don't need to be analyzed. (For example, you can use the test case ID, an "I" for impossible, put a slash through the node, or shade it gray.) Marking the nodes (rather than deleting them) helps people see that the attacks were considered. You might choose to test the assumption that a given node is impossible. See the "Test Process Integration" section in Chapter 10 "Validating That Threats Are Addressed" for more details.

Check the Presentation

Regardless of graphical form, you should aim to present each tree or subtree in no more than a page. If your tree is hard to see on a page, it may be helpful to break it into smaller trees. Each top level subnode can be the root of a new tree, with a "context" tree that shows the overall relations. You may also be able to adjust presentation details such as font size, within the constraints of usability.

The node labels should be of the same form, focusing on active terms. Finally, draw the tree on a grid to make it easy to track. Ideally, the equivalent level subnodes will show on a single line. That becomes more challenging as you go deeper into a tree.

Representing a Tree

Trees can be represented in two ways: as a free-form (human-viewable) model without any technical structure, or as a structured representation with variable types and/or metadata to facilitate programmatic analysis.

Human-Viewable Representations

Attack trees can be drawn graphically or shown in outline form. Graphical representations are a bit more work to create but have more potential to focus attention. In either case, if your nodes are not all related by the same logic (AND/OR), you'll need to decide on a way to represent the relationship and communicate that decision. If your tree is being shown graphically, you'll also want to decide if you use a distinct shape for a terminal node: The labels in a node should be carefully chosen to be rich in information, especially if you're using a graphical tree. Words such as "attack" or "via" can distract from the key information. Choose "modify file" over "attack via modifying file." Words such as "weak" are more helpful when other nodes say "no." So "weak cryptography" is a good contrast to "no cryptography."

As always, care should be taken to ensure that the graphics are actually information-rich and communicative. For instance, consider the three representations of a tree shown in Figure 4–1.

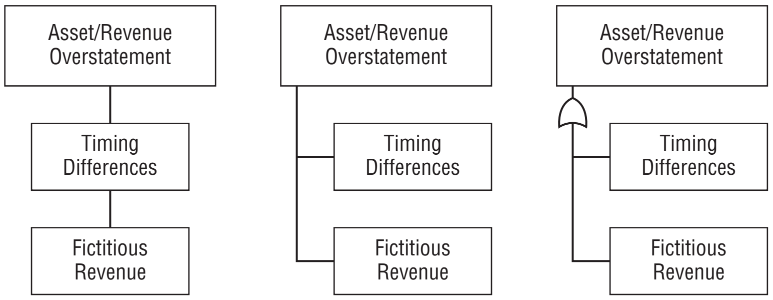


Figure 4–1: Three representations of a tree

The left tree shows an example of a real tree that simply uses boxes. This representation does not clearly distinguish hierarchy, making it hard to tell which nodes are at the same level of the tree. Compare that to the center tree, which uses a tree to show the equivalence of the leaf nodes. The rightmost tree adds the "OR gate" symbol from circuit design to show that any of the leaf nodes lead to the parent condition.

Additionally, tree layout should make considered use of space. In the very small tree in Figure 4–2, note the pleasant grid that helps your eye follow the layout. In contrast, consider the layout of Figure 4–3, which feels jumbled. To focus your attention on the layout, both are shown too small to read.

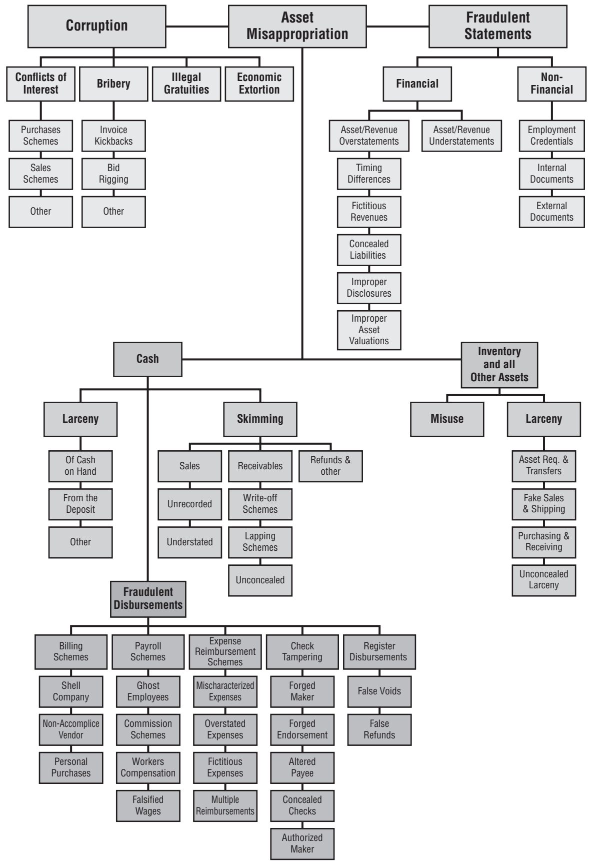
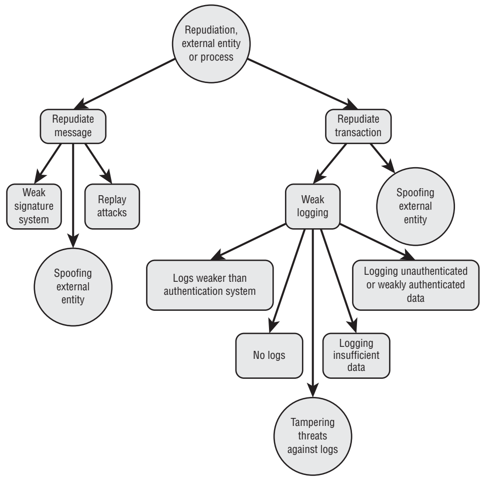


Figure 4-2: A tree drawn on a grid



Repudiation, external entity or process
Repudiate message
Repudiate transaction
Weak signature system
Replay attacks
Spoofing external entity
Weak logging
Spoofing external entity
Logs weaker than authentication system
Logging unauthenticated or weakly authenticated data
No logs
Logging insufficient data
Tampering threats against logs

Figure 4-3: A tree drawn without a grid

NOTE In *Writing Secure Code 2* (Microsoft Press, 2003), Michael Howard and David LeBlanc suggest the use of dotted lines for unlikely threats, solid lines for likely threats, and circles to show mitigations, although including mitigations may make the trees too complex.

Outline representations are easier to create than graphical representations, but they tend to be less attention-grabbing. Ideally, an outline tree is shown on a single page, not crossing pages. The question of how to effectively represent AND/OR is not simple. Some representations leave them out, others include an indicator either before or after a line. The next three samples are modeled after the trees in "Election Operations Assessment Threat Trees" later in this chapter. As you look at them, ask yourself precisely what is needed to achieve the goal in node 1, "Attack voting equipment."

1. Attack voting equipment
    - 1.1 Gather knowledge
        - 1.1.1 From insider
        - 1.1.2 From components
    - 1.2 Gain insider access
        - 1.2.1 At voting system vendor
        - 1.2.2 By illegal insider entry

The preceding excerpt isn't clear. Should the outline be read as a need to do each of these steps, or one or the other to achieve the goal of attacking voting equipment? Contrast that with the next tree, which is somewhat better:

1. Attack voting equipment
    - 1.1 Gather knowledge (and)
        - 1.1.1 From insider (or)
        - 1.1.2 From components
    - 1.2 Gain insider access (and)
        - 1.2.1 At voting system vendor (or)
        - 1.2.2 By illegal insider entry

This representation is useful at the end nodes: It is clearly 1.1.1 or 1.1.2. But what does the "and" on line 1.1 refer to? 1.1.1 or 1.1.2? The representation is not clear. Another possible form is shown next:

1. Attack voting equipment
   - O 1.1 Gather knowledge
     - T 1.1.1 From insider
     - O 1.1.2 From components
   - O 1.2 Gain insider access
     - T 1.2.1 At voting system vendor
     - T 1.2.2 By illegal insider entry

This is intended to be read as "AND Node: 1: Attack voting equipment, involves 1.1, gather knowledge either from insider or from components AND 1.2, gain insider access . . ." This can be confusing if read as the children of that node are to be ORed, rather than being ORed with its sibling nodes. This is much clearer in the graphical presentation. Also note that the steps are intended to be sequential. You must gather knowledge, then gain insider access, then attack the components to pull off the attack.

As you can see from the preceding examples, the question of how to use an outline representation of a tree is less simple than you might expect. If you are using someone else's tree, be sure you understand their intent. If you are creating a tree, be sure you are clear on your intent, and clear in your communication of your intent.

Structured Representations

Graphical and outline presentation of trees are useful for humans, but a tree is also a data structure, and a structured representation of a tree makes it possible to apply logic to the tree and in turn, the system you're modeling. Several software packages enable you to create and manage complex trees. One such package allows the modeler to add costs to each node, and then assess what attacks an attacker with a given budget can execute. As your trees become more complex, such software is more likely to be worthwhile. See Chapter 11 "Threat Modeling Tools" for a list of tree management software.

Example Attack Tree

The following simple example of an attack tree (and a useful component for other attack tree activity) models how an attacker might get into a building. The entire tree is an OR tree; any of the methods listed will achieve the goal. (This tree is derived from "An Attack Tree for the Border Gateway Protocol" [Convery, 2004].)

Goal: Access to the building

1. Go through a door
   a. When it's unlocked:
      i. Get lucky.
      ii. Obstruct the latch plate (the "Watergate Classic").
      iii. Distract the person who locks the door at night.
   b. Drill the lock.
   c. Pick the lock.
   d. Use the key.
      i. Find a key.
      ii. Steal a key.
      iii. Photograph and reproduce the key.
      iv. Social engineer a key from someone.
         1. Borrow the key.
         2. Convince someone to post a photo of their key ring.
   e. Social engineer your way in.
      i. Act like you're authorized and follow someone in.
      ii. Make friends with an authorized person.
      iii. Carry a box, a cup of coffee in each hand, etc.
2. Go through a window.
   a. Break a window.
   b. Lift the window.
3. Go through a wall.
   a. Use a sledgehammer or axe.
   b. Use a truck to go through the wall.
4. Gain access via other means.
   a. Use a fire escape.
   b. Use roof access from a helicopter (preferably black) or adjacent building.
   c. Enter another part of the building, using another tenant's access.

Real Attack Trees

A variety of real attack trees have been published. These trees may be helpful to you either directly, because they model systems like the one you're modeling, or as examples of how to build an attack tree. The three attack trees in this section show how insiders commit financial fraud, how to attack elections, and threats against SSL.

Each of these trees has the nice property of being available now, either as an extended example, as a model for you to build from, or (if you're working around fraud, elections, or SSL), to use directly in analyzing a system which matters to you.

The fraud tree is designed for you to use. In contrast, the election trees were developed to help the team think through their threats and organize the possibilities.

Fraud Attack Tree

An attack tree from the Association of Certified Fraud Examiners is shown with their gracious permission in Figure 4–4, and it has a number of good qualities. First, it's derived from actual experience in finding and exposing fraud. Second, it has a structure put together by subject matter experts, so it's not a random collection of threats. Finally, it has an associated set of mitigations, which are discussed at great length in Joseph Wells' *Corporate Fraud Handbook* (Wiley, 2011).

Election Operations Assessment Threat Trees

The largest publicly accessible set of threat trees was created for the Elections Assistance Commission by a team centered at the University of Southern Alabama. There are six high-level trees. They are useful both as an example and for you to use directly, and there are some process lessons you can learn.

NOTE **This model covers a wider scope of attacks than typical for software threat models, but is scoped like many operational threat models.**

1. Attack voting equipment.
2. Perform an insider attack.
3. Subvert the voting process.
4. Experience technical failure.
5. Attack audit.
6. Disrupt operations.

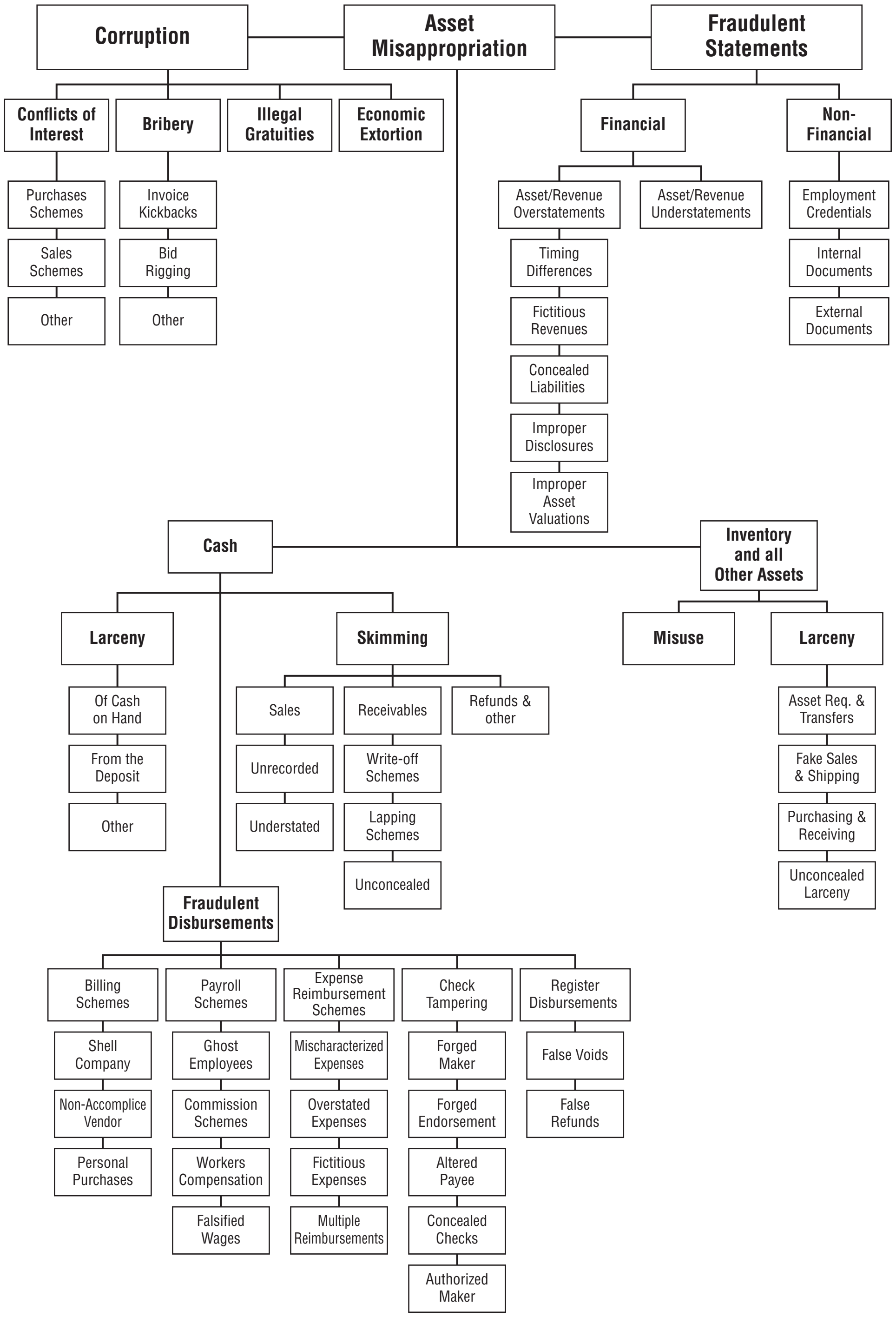


Figure 4–4: The ACFE fraud tree

If your system is vulnerable to threats such as equipment attack, insider attack, process subversion or disruption, these attack trees may work well to help you find threats against those systems.

The team created these trees to organize their thinking around what might go wrong. They described their process as having found a very large set of issues via literature review, brainstorming, and original research. They then broke the threats into high-level sets, and had individuals organize them into trees. An attempt to sort the sets into a tree in a facilitated group process did not work (Yanisac, 2012). The organization of trees may require a single person or a very close-knit team; you should be cautious about trying for consensus trees.

Mind Maps

Application security specialist Ivan Ristic (Ristić, 2009) conducted an interesting experiment using a mind map for what he calls an SSL threat model, as shown in Figure 4–5.

This is an interesting way to present a tree. There are very few mind-map trees out there. This tree, like the election trees, shows a set of editorial decisions and those who use mind maps may find the following perspective on this mind map helpful:

- The distinction between "Protocols/Implementation bugs" and "End points/Client side/secure implementation" is unclear.
- There's "End points/Client side/secure implementation" but no "server side" counterpart to that.
- Under "End points/server side/server config" there's a large subtree. Compare that to Client side where there's no subtree at all.
- Some items have an asterisk (*) but it's unclear what that means. After discussion with Ivan, it turns out that those "may not apply to everyone."
- There's an entire set of traffic analytic threats that allow you to see where on a secure site someone is. These issues are made worse by AJAX, but more important here, how should they fit into this mind map? Perhaps under "Protocols/specifications/scope limits"?
- It's hard to find elements of the map, as it draws the eye in various directions, some of which don't align with the direction of reading.

Perspective on Attack Trees

Attack trees can be a useful way to convey information about threats. They can be helpful even to security experts as a way to quickly consider possible attack types. However, despite their surface appeal, it is very hard to create attack trees.

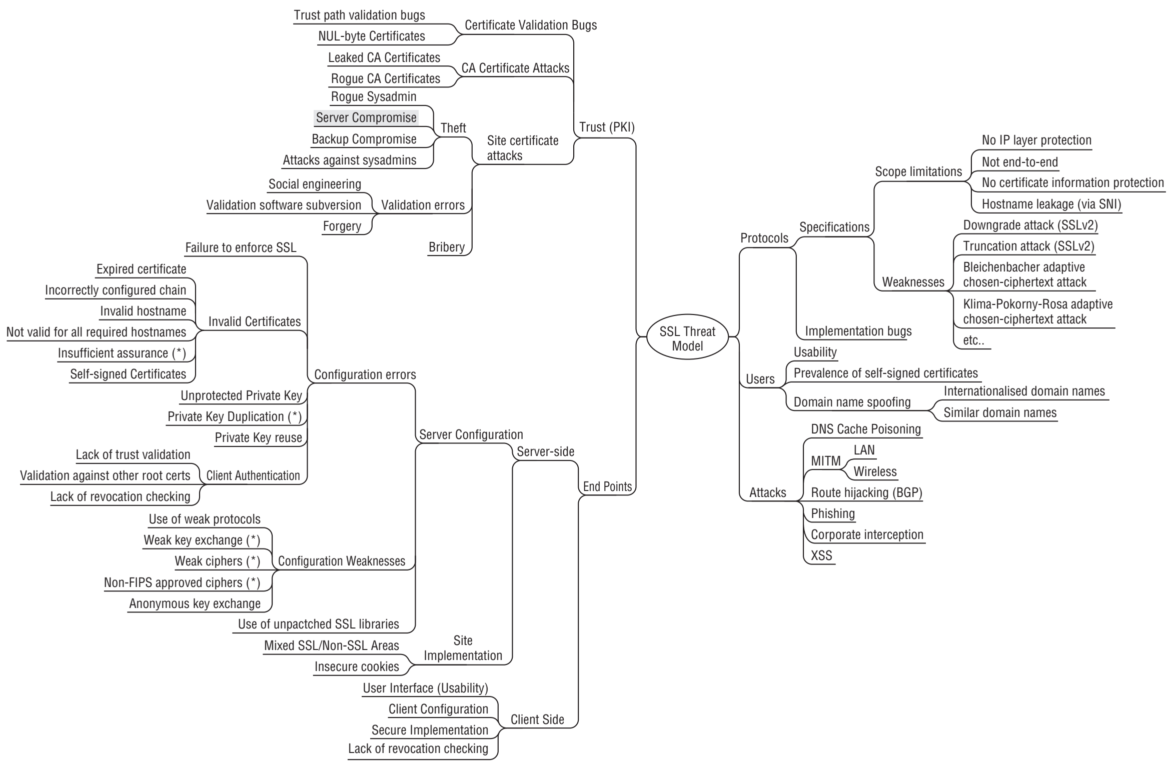


Figure 4–5: Ristic's SSL mind map

I hope that we'll see experimentation and perhaps progress in the quality of advice. There are also a set of issues that can make trees hard to use, including completeness, scoping, and meaning:

- **Completeness:** Without the right set of root nodes, you could miss entire attack groupings. For example, if your threat model for a safe doesn't include "pour liquid nitrogen on the metal, then hit with a hammer," then your safe is unlikely to resist this attack. Drawing a tree encourages specific questions, such as "how could I open the safe without the combination?" It may or may not bring you to the specific threat. Because there's no way of knowing how many nodes a branch should have, you may never reach that point. A close variant of the this is how do you know that you're done? (Schneier's attack tree article alludes to these problems.)
- **Scoping:** It may be unreasonable to consider what happens when the computer's main memory is rapidly cooled and removed from the motherboard. If you write commercial software for word processing, this may seem like an operating system issue. If you create commercial operating systems, it may seem like a hardware issue. The nature of attack trees means many of the issues discovered will fall under the category of "there's no way for us to fix that."
- **Meaning:** There is no consistency around AND/OR, or around sequence, which means that understanding a new tree takes longer.

Summary

Attack trees fit well into the four-step framework for threat modeling. They can be a useful tool for finding threats, or a way to organize thinking about threats (either for your project or more broadly).

To create a new attack tree to help you organize thinking, you need to decide on a representation, and then select a root node. With that root node, you can brainstorm, use STRIDE, or use a literature review to find threats to add to nodes. As you iterate over the nodes, consider if the tree is complete or overly-full, aiming to ensure the right threats are in the tree. When you're happy with the content of the tree, you should check the presentation so others can use it. Attack trees can be represented as graphical trees, as outlines, or in software.

You saw a sample tree for breaking into a building, and real trees for fraud, elections, and SSL. Each can be used as presented, or as an example for you to consider how to construct trees of your own.

CHAPTER

6

Privacy Tools

Threat modeling for privacy issues is an emergent and important area. Much like security threats violate a required security property, privacy threats are where a required privacy property is violated. Defining privacy requirements is a delicate balancing act, however, for a few reasons: First, the organization offering a service may want or even need a lot of information that the people using the service don't want to provide. Second, people have very different perceptions of what privacy is, and what data is private, and those perceptions can change with time. (For example, someone leaving an abusive relationship should be newly sensitive to the value of location privacy, and perhaps consider their address private for the first time.) Lastly, most people are "privacy pragmatists" and will make value tradeoffs for personal information.

Some people take all of this ambiguity to mean that engineering for privacy is a waste. They're wrong. Others assert that concern over privacy is a waste, as consumers don't behave in ways that expose privacy concerns. That's also wrong. People often pay for privacy when they understand the threat and the mitigation. That's why advertisements for curtains, mailboxes, and other privacy-enhancing technologies often lead with the word "privacy."

Unlike the previous three chapters, each of which focused on a single type of tool, this chapter is an assemblage of tools for finding privacy threats. The approaches described in this chapter are more developed than "worry about privacy," yet they are somewhat less developed than security attack libraries such as CAPEC (discussed in Chapter 5, "Attack Libraries"). In either event, they are important enough to include. Because this is an emergent area, appropriate exit criteria are less clear, so there are no exit criteria sections here.

In this chapter, you'll learn about the ways to threat model for privacy, including Solove's taxonomy of privacy harms, the IETF's "Privacy Considerations for Internet Protocols," privacy impact assessments (PIAs), the nymity slider, contextual integrity, and the LINDDUN approach, a mirror of STRIDE created to find privacy threats. It may be reasonable to treat one or more of contextual integrity, Solove's taxonomy or (a subset of) LINDDUN as a building block that can snap into the four-stage model, either replacing or complementing the security threat discovery.

NOTE **Many of these techniques are easier to execute when threat modeling operational systems, rather than boxed software. (Will your database be used to contain medical records? Hard to say!) The IETF process is more applicable than other processes to "boxed software" designs.**

Solove's Taxonomy of Privacy

In his book, *Understanding Privacy* (Harvard University Press, 2008), George Washington University law professor Daniel Solove puts forth a taxonomy of privacy *harms.* These harms are analogous to threats in many ways, but also include impact. Despite Solove's clear writing, the descriptions might be most helpful to those with some background in privacy, and challenging for technologists to apply to their systems. It may be possible to use the taxonomy as a tool, applying it to a system under development, considering whether each of the harms presented is enabled. The following list presents a version of this taxonomy derived from Solove, but with two changes. First, I have added "identifier creation," in parentheses. I believe that the creation of an identifier is a discrete harm because it enables so many of the other harms in the taxonomy. (Professor Solove and I have agreed to disagree on this issue.) Second, exposure is in brackets, because those using the other threat modeling techniques in this Part should already be handling such threats.

- (Identifier creation)
- Information collection: surveillance, interrogation
- Information processing: aggregation, identification, insecurity, secondary use, exclusion
- Information dissemination: breach of confidentiality, disclosure, increased accessibility, blackmail, appropriation, distortion, [exposure]
- Invasion: intrusion, decisional interference

Many of the elements of this list are self-explanatory, and all are explained in depth in Solove's book. A few may benefit from a brief discussion. The harm of surveillance is twofold: First is the uncomfortable feeling of being watched and second are the behavioral changes it may cause. *Identification* means the association of information with a flesh-and-blood person. *Insecurity* refers to the psychological state of a person made to feel insecure, rather than a technical state. The harm of secondary use of information relates to societal trust. Exclusion is the use of information provided to exclude the provider (or others) from some benefit.

Solove's taxonomy is most usable by privacy experts, in the same way that STRIDE as a mnemonic is most useful for security experts. To make use of it in threat modeling, the steps include creating a model of the data flows, paying particular attention to personal data.

Finding these harms may be possible in parallel to or replacing security threat modeling. Below is advice on where and how to focus looking for these.

- Identifier creation should be reasonably easy for a developer to identify.
- Surveillance is where data is collected about a broad swath of people or where that data is gathered in a way that's hard for a person to notice.
- Interrogation risks tend to correlate around data collection points, for example, the many "* required" fields on web forms. The tendency to lie on such forms may be seen as a response to the interrogation harm.
- Aggregation is most frequently associated with inbound data flows from external entities.
- Identification is likely to be found in conjunction with aggregation or where your system has in-person interaction.
- Insecurity may associate with where data is brought together for decision purposes.
- Secondary use may cross trust boundaries, possibly including boundaries that your customers expect to exist.
- Exclusion happens at decision points, and often fraud management decisions.
- Information dissemination threats (all of them) are likely to be associated with outbound data flows; you should look for them where data crosses trust boundaries.
- Intrusion is an in-person intrusion; if your system has no such features, you may not need to look at these.
- Decisional interference is largely focused on ways in which information collection and processing may influence decisions, and as such it most likely plays into a requirements discussion.

Privacy Considerations for Internet Protocols

The Internet Engineering Task Force (IETF) requires consideration of security threats, and has a process to threat model focused on their organizational needs, as discussed in Chapter 17, "Bringing Threat Modeling to Your Organization." As of 2013, they sometimes require consideration of privacy threats. An informational RFC "Privacy Considerations for Internet Protocols," outlines a set of security-privacy threats, a set of pure privacy threats, and offers a set of mitigations and some general guidelines for protocol designers (Cooper, 2013). The combined security-privacy threats are as follows:

- Surveillance
- Stored data compromise
- Mis-attribution or intrusion (in the sense of unsolicited messages and denial-of-service attacks, rather than break-ins)

The privacy-specific threats are as follows:

- Correlation
- Identification
- Secondary use
- Disclosure
- Exclusion (users are unaware of the data that others may be collecting)

Each is considered in detail in the RFC. The set of mitigations includes data minimization, anonymity, pseudonymity, identity confidentiality, user participation and security. While somewhat specific to the design of network protocols, the document is clear, free, and likely a useful tool for those attempting to threat model privacy. The model, in terms of the abstracted threats and methods to address them, is an interesting step forward, and is designed to be helpful to protocol engineers.

Privacy Impact Assessments (PIA)

As outlined by Australian privacy expert Roger Clarke in his "An Evaluation of Privacy Impact Assessment Guidance Documents," a PIA "is a systematic process that identifies and evaluates, from the perspectives of all stakeholders, the potential effects on privacy of a project, initiative, or proposed system or scheme, and includes a search for ways to avoid or mitigate negative privacy impacts." Thus, a PIA is, in several important respects, a privacy analog to security threat modeling. Those respects include the systematic tools for identification and evaluation of privacy issues, and the goal of not simply identifying issues, but also mitigating them. However, as usually presented, PIAs have too much integration between their steps to snap into the four-stage framework used in this book.

There are also important differences between PIAs and threat modeling. PIAs are often focused on a system as situated in a social context, and the evaluation is often of a less technical nature than security threat modeling. Clarke's evaluation criteria include things such as the status, discoverability, and applicability of the PIA guidance document; the identification of a responsible person; and the role of an oversight agency; all of which would often be considered out of scope for threat modeling. (This is not a critique, but simply a contrast.) One sample PIA guideline from the Office of the Victorian Privacy Commissioner states the following:

"Your PIA Report might have a Table of Contents that looks something like this:

1. Description of the project
2. Description of the data flows
3. Analysis against 'the' Information Privacy Principles
4. Analysis against the other dimensions to privacy
5. Analysis of the privacy control environment
6. Findings and recommendations"

Note that step 2, "description of the data flows," is highly reminiscent of "data flow diagrams," while steps 3 and 4 are very similar to the "threat finding" building blocks. Therefore, this approach might be highly complementary to the four-step model of threat modeling.

The appropriate privacy principles or other dimensions to consider are somewhat dependent on jurisdiction, but they can also focus on classes of intrusion, such as those offered by Solove, or a list of concerns such as informational, bodily, territorial, communications, and locational privacy. Some of these documents, such as those from the Office of the Victorian Privacy Commissioner (2009a), have extensive lists of common privacy threats that can be used to support a guided brainstorming approach, even if the documents are not legally required. Privacy impact assessments that are performed to comply with a law will often have a formal structure for assessing sufficiency.

The Nymity Slider and the Privacy Ratchet

University of Waterloo professor Ian Goldberg has defined a measurement he calls *nymity*, the "amount of information about the identity of the participants that is revealed [in a transaction]." Nymity is from the Latin for name, from which *anonymous* ("without a name") and *pseudonym* ("like a name") are derived.

Goldberg has pointed out that you can graph nymity on a continuum (Goldberg, 2000). Figure 6-1 shows the nymity slider. On the left-hand side, there is less privacy than on the right-hand side. As Goldberg points out, it is easy to move towards more nymity, and extremely difficult to move away from it. For example, there are protocols for electronic cash that have most of the privacy-preserving properties of physical cash, but if you deliver it over a TCP connection you lose many of those properties. As such, the nymity slider can be used to examine how privacy-threatening a protocol is, and to compare the amount of nymity a system uses. To the extent that it can be designed to use less identifying information, other privacy features will be easier to achieve.

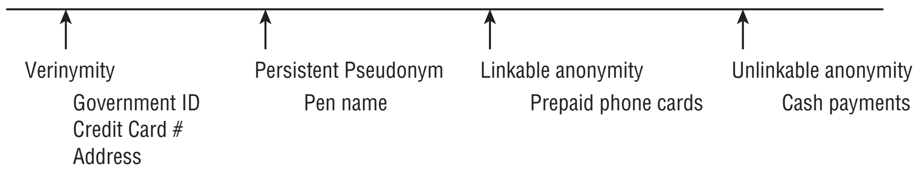


Figure 6-1: The nymity slider

When using nymity privacy in threat modeling, the goal is to measure how much information a protocol, system, or design exposes or gathers. This enables you to compare it to other possible protocols, systems, or designs. The nymity slider is thus an adjunct to other threat-finding building blocks, not a replacement for them.

Closely related to nymity is the idea of *linkability*. Linkability is the ability to bring two records together, combining the data in each into a single record or virtual record. Consider several databases, one containing movie preferences, another containing book purchases, and a third containing telephone records. If each contains an e-mail address, you can learn that `joe@example.org` likes religious movies, that he's bought books on poison, and that several of the people he talks with are known religious extremists. Such intersections might be of interest to the FBI, and it's a good thing you can link them all together! (Unfortunately, no one bothered to include the professional database showing he's a doctor, but that's beside the point!) The key is that you've engaged in linking several datasets based on an identifier. There is a set of identifiers, including e-mail addresses, phone numbers, and government-issued ID numbers, that are often used to link data, which can be considered strong evidence that multiple records refer to the same person. The presence of these strongly linkable data points increases linkability threats.

Linkability as a concept relates closely to Solove's concept of *identification* and *aggregation*. Linkability can be seen as a spectrum from strongly linkable with multiple validated identifiers to weakly linkable based on similarities in the data.

("John Doe and John E. Doe is probably the same person.") As data becomes richer, the threat of linkage increases, even if the strongly linkable data points are removed. For example, Harvard professor Latanya Sweeney has shown how data with only date of birth, gender, and zip code uniquely identifies 87 percent of the U.S. population (Sweeney, 2002). There is an emergent scientific research stream into "re-identification" or "de-anonymization," which discloses more such results on a regular basis. The release of anonymous datasets carries a real threat of re-identification, as AOL, Netflix, and others have discovered. (McCullagh, 2006; Narayanan, 2008; Buley, 2010).

Contextual Integrity

Contextual integrity is a framework put forward by New York University professor Helen Nissenbaum. It is based on the insight that many privacy issues occur when information is taken from one context and brought into another. A *context* is a term of art with a deep grounding in discussions of the spheres, or arenas, of our lives. A context has associated roles, activities, norms, and values. Nissenbaum's approach focuses on understanding contexts and changes to those contexts. This section draws very heavily from Chapter 7 of her book *Privacy in Context*, (Stanford Univ. Press, 2009) to explain how you might apply the framework to product development.

Start by considering what a context is. If you look at a hospital as a context, then the roles might include doctors, patients, and nurses, but also family members, administrators, and a host of other roles. Each has a reason for being in a hospital, and associated with that reason are activities that they tend to perform there, norms of behavior, and values associated with those norms and activities.

Contexts are places or social areas such as restaurants, hospitals, work, the Boy Scouts, and schools (or a type of school, or even a specific school). An event can be "in a work context" even if it takes place somewhere other than your normal office. Any instance in which there is a defined or expected set of "normal" behaviors can be treated as a context. Contexts nest and overlap. For example, normal behavior in a church in the United States is influenced by the norms within the United States, as well as the narrower context of the parishioners. Thus, what is normal at a Catholic Church in Boston or a Baptist Revival in Mississippi may be inappropriate at a Unitarian Congregation in San Francisco (or vice versa). Similarly, there are shared roles across all schools, those of student or teacher, and more specific roles as you specify an elementary school versus a university. There are specific contexts within a university or even the particular departments of a university.

Contextual integrity is violated when the informational norms of a context are breached. Norms, in Nissenbaum's sense, are "characterized by four key parameters: context, actors, attributes, and transmission principles." Context is roughly as just described. Actors are senders, recipients, and information subjects. Attributes refer to the nature of the information—for example, the nature or particulars of a disease from which someone is suffering. A transmission principle is "a constraint on the flow (distribution, dissemination, transmission) of information from party to party." Nussbaum first provides two presentations of contextual integrity, followed by an augmented contextual integrity heuristic. As the technique is new, and the "augmented" approach is not a strict superset of the initial presentation, it may help you to see both.

Contextual Integrity Decision Heuristic

Nissenbaum first presents contextual integrity as a post-incident analytic tool. The essence of this is to document the context as follows:

1. Establish the prevailing context.
2. Establish key actors.
3. Ascertain what attributes are affected.
4. Establish changes in principles of transmission.
5. Red flag

Step 5 means "if the new practice generates changes in actors, attributes, or transmission principles, the practice is flagged as violating entrenched informational norms and constitutes a prima facie violation of contextual integrity." You might have noticed a set of interesting potential overlaps with software development and threat modeling methodologies. In particular, actors overlap fairly strongly with personas, in Cooper's sense of personas (discussed in Appendix B, "Threat Trees"). A contextual integrity analysis probably does not require a set of personas for bad actors, as any data flow outside the intended participants (and perhaps some between them) is a violation. The information transmissions, and the associated attributes are likely visible in data flow or swim lane diagrams developed for normal security threat modeling.

Thus, to the extent that threat models are being enhanced from version to version, a set of change types could be used to trigger contextual integrity analysis. The extant diagram is the "prevailing context." The important change types would include the addition of new human entities or new data flows.

Nissenbaum takes pains to explore the question of whether a violation of contextual integrity is a worthwhile reason to avoid the change. From the perspective of threat elicitation, such discussions are out of scope. Of course, they are in scope as you decide what to do with the identified privacy threats.

Augmented Contextual Integrity Heuristic

Nissenbaum also presents a longer, 'augmented' heuristic, which is more prescriptive about steps, and may work better to predict privacy issues.

1. Describe the new practice in terms of information flows.
2. Identify the prevailing context.
3. Identify information subjects, senders, and recipients.
4. Identify transmission principles.
5. Locate applicable norms, identify significant changes.
6. Prima facie assessment
7. Evaluation
    a. Consider moral and political factors.
    b. Identify threats to autonomy and freedom.
    c. Identify effects on power structures.
    d. Identify implications for justice, fairness, equality, social hierarchy, democracy and so on.
8. Evaluation 2
    a. Ask how the system directly impinges on the values, goals, and ends of the context.
    b. Consider moral and ethical factors in light of the context.
9. Decide.

This is, perhaps obviously, not an afternoon's work. However, in considering how to tie this to a software engineering process, you should note that steps 1, 3, and 4 look very much like creating data flow diagrams. The context of most organizations is unlikely to change substantially, and thus descriptions of the context may be reusable, as may be the work products to support the evaluations of steps 7 and 8.

Perspective on Contextual Integrity

I very much like contextual integrity. It strikes me as providing deep insight into and explanations for a great number of privacy problems. That is, it may be possible to use it to predict privacy problems for products under design. However, that's an untested hypothesis. One area of concern is that the effort to spell out all the aspects of a context may be quite time consuming, but without spelling out all the aspects, the privacy threats many be missed. This sort of work is challenging when you're trying to ship software and Nissenbaum goes so far as to describe it as "tedious" (*Privacy In Context*, page 142). Additionally, the act of fixing a context in software or structured definitions may present risks that the fixed representation will deviate as social norms evolve.

This presents a somewhat complex challenge to the idea of using contextual integrity as a threat modeling methodology within a software engineering process. The process of creating taxonomies or categories is an essential step in structuring data in a database. Software engineers do it as a matter of course as they develop software, and even those who are deeply cognizant of taxonomies often treat it as an implicit step. These taxonomies can thus restrict the evolution of a context—or worse; generate dissonance between the software-engineered version of the context or the evolving social context. I encourage security and privacy experts to grapple with these issues.

LINDDUN

LUNDDUN is a mnemonic developed by Mina Deng for her PhD at the Katholieke Universiteit in Leuven, Belgium (Deng, 2010). LINDDUN is an explicit mirroring of STRIDE-per-element threat modeling. It stands for the following violations of privacy properties:

- Linkability
- Identifiability
- Non-Repudiation
- Detectability
- Disclosure of information
- Content Unawareness
- Policy and consent Noncompliance

LINDDUN is presented as a complete approach to threat modeling with a process, threats, and requirements discovery method. It may be reasonable to use the LINDDUN threats or a derivative as a tool for privacy threat enumeration in the four-stage framework, snapping it either in place of or next to STRIDE security threat enumeration. However, the threats in LINDDUN are somewhat unusual terminology; therefore, the training requirements may be higher, or the learning curve steeper than other privacy approaches.

NOTE **LINDDUN leaves your author deeply conflicted. The privacy terminology it relies on will be challenging for many readers. However, it is, in many ways, one of the most serious and thought-provoking approaches to privacy threat modeling, and those seriously interested in privacy threat modeling should take a look. As an aside, the tension between non-repudiation as a privacy threat and repudiation as a security threat is delicious.**

Summary

Privacy is no less important to society than security. People will usually act to protect their privacy given an understanding of the threats and how they can address them. As such, it may help you to look for privacy threats in addition to security threats. The ways to do so are less prescriptive than ways to look for security threats.

There are many tools you can use to find privacy issues, including Solove's taxonomy of privacy harms. (A harm is a threat with its impact.) Solove's taxonomy helps you understand the harm associated with a privacy violation, and thus, perhaps, how best to prioritize it. The IETF has an approach to privacy threats for new Internet protocols. That approach may complement or substitute Privacy Impact Assessments. PIAs and the IETF's processes are appropriate when a regulatory or protocol design context calls for their use. Both are more prescriptive than the nymity slider, a tool for assessing the amount of personal information in a system and measuring privacy invasion for comparative purposes. They are also more prescriptive than contextual integrity, an approach which attempts to tease out the social norms of privacy. If your goal is to identify when a design is likely to raise privacy concerns, however, then contextual integrity may be the most helpful. Far more closely related to STRIDE-style threat identification is LINDDUN, which considers privacy violations in the manner that STRIDE considers security violations.

CHAPTER

8

Defensive Tactics and Technologies

So far you've learned to model your software using diagrams and learned to find threats using STRIDE, attack trees, and attack libraries. The next step in the threat modeling process is to address every threat you've found.

When it works, the fastest and easiest way to address threats is through technology-level implementations of defensive patterns or features. This chapter covers the standard tactics and technologies that you will use to mitigate threats. These are often operating system or program features that you can configure, activate, apply or otherwise rapidly engage to defend against one or more threats. Sometimes, they involve additional code that is widely available and designed to quickly plug in. (For example, tunneling connections over SSH to add security is widely supported, and some unix packages even have options to make that easier.)

Because you likely found your threats via STRIDE, the bulk of this chapter is organized according to STRIDE. The main part of the chapter addresses STRIDE and privacy threats, because most pattern collections already include information about how to address the threats.

Tactics and Technologies for Mitigating Threats

The mitigation tactics and technologies in this chapter are organized by STRIDE because that's most likely how you found them. This section is therefore organized by ways to mitigate each of the STRIDE threats, each of which includes a brief recap of the threat, tactics that can be brought to bear against it, and the techniques for accomplishing that by people with various skills and responsibilities. For example, if you're a developer who wants to add cryptographic authentication to address spoofing, the techniques you use are different from those used by a systems administrator. Each subsection ends with a list of specific technologies.

Authentication: Mitigating Spoofing

Spoofing threats against code come in a number of forms: faking the program on disk, squatting a port (IP, RPC, etc.), splicing a port, spoofing a remote machine, or faking the program in memory (related problems with libraries and dependencies are covered under tampering); but in general, only programs running at the same or a lower level of trust are spoofable, and you should endeavor to trust only code running at a higher level of trust, such as in the OS.

There is also spoofing of people, of course, a big, complex subject covered in Chapter 14, "Accounts and Identity." Mitigating spoofing threats often requires unusually tight integration between layers of systems. For example, a maintenance engineer from Acme, Inc. might want remote (or even local) access to your database. Is it enough to know that the person is an employee of Acme? Is it enough to know that he or she can initiate a connection from Acme's domain? You might reasonably want to create an account on your database to allow Joe Engineer to log in to it, but how do you bind that to Acme's employee database? When Joe leaves Acme and gets a job at Evil Geniuses for a Better Tomorrow, what causes his access to Acme's database to go away?

NOTE **Authentication and authorization are related concepts, and sometimes confused. Knowing that someone really is Adam Shostack should not authorize a bank to take money from my account (there are several people of that name in the U.S.). Addressing authorization is covered in the Authorization: Mitigating Elevation of Privilege**

From here, let's dig into the specific ways in which you can ensure authentication is done well.

Tactics for Authentication

You can authenticate a remote machine either with or without cryptographic trust mechanisms. Without crypto involves verifying via IP or "classic" DNS entries. All the noncryptographic methods are unreliable. Before they existed, there were attempts to make hostnames reliable, such as the double-reverse DNS lookup. At the time, this was sometimes the best tactic for authentication.

Today, you can do better, and there's rarely an excuse for doing worse. (SNMP may be an excuse, and very small devices may be another). As mentioned earlier, authenticating a person is a complex subject, covered in Chapter 14. Authenticating on-system entities is somewhat operating system dependent.

Whatever the underlying technical mechanisms are, at some point cryptographic keys are being managed to ensure that there's a correspondence between technical names and names people use. That validation cannot be delegated entirely to machines. You can choose to delegate it to one of the many companies that assert they validate these things. These companies often do business as "PKI" or "public key infrastructure" companies, and are often referred to as "certification authorities" or "CAs". You should be careful about relying on that delegation for any transaction valued at more than what the company will accept for liability. (In most cases, certificate authorities limit their liability to nothing). Why you should assign it a higher value is a question their marketing departments hope will not be asked, but the answer roughly boils down to convenience, limited alternatives, and accepted business practice.

Developer Ways to Address Spoofing

Within an operating system, you should aim to use full and canonical path names for libraries, pipes, and so on to help mitigate spoofing. If you are relying on something being protected by the operating system, ensure that the permissions do what you expect. (In particular, unix files in `/tmp` are generally unreliable, and Windows historically has had similarly shared directories.) For networked systems in a single trust domain, using operating system mechanisms such as Active Directory or LDAP makes sense. If the system spans multiple trust domains, you might use persistence or a PKI. If the domains change only rarely, it may be appropriate to manually cross-validate keys, or to use a contract to specify who owns what risks.

You can also use cryptographic ways to address spoofing, and these are covered in Chapter 16, "Threats to Cryptosystems." Essentially, you tie a key to a person, and then work to authenticate that the key is correctly associated with the person who's connecting or authenticating.

Operational Ways to Address Spoofing

Once a system is built, a systems administrator has limited options for improving spoofing defenses. To the extent that the system is internal, pressure can be brought to bear on system developers to improve authentication. It may also be possible to use DNSSEC, SSH, or SSL tunneling to add or improve authentication. Some network providers will filter outbound traffic to make spoofing harder. That's helpful, but you cannot rely on it.

Authentication Technologies

Technologies for authenticating computers (or computer accounts) include the following:

- IPSec
- DNSSEC
- SSH host keys
- Kerberos authentication
- HTTP Digest or Basic authentication
- "Windows authentication" (NTLM)
- PKI systems, such as SSL or TLS with certificates

Technologies for authenticating bits (files, messages, etc.) include the following:

- Digital signatures
- Hashes

Methods for authenticating people can involve any of the following:

- Something you know, such as a password
- Something you have, such as an access card
- Something you are, such as a biometric, including photographs
- Someone you know who can authenticate you

Technologies for maintaining authentication across connections include the following:

- Cookies

Maintaining authentication across connections is a common issue as you integrate systems. The cookie pattern has flaws, but generally, it has fewer flaws than re-authenticating with passwords.

Integrity: Mitigating Tampering

Tampering threats come in several flavors, including tampering with bits on disk, bits on a network, and bits in memory. Of course, no one is limited to tampering with a single bit at a time.

Tactics for Integrity

There are three main ways to address tampering threats: relying on system defenses such as permissions, use of cryptographic mechanisms, and use of logging technology and audit activities as a deterrent.

Permission mechanisms can protect things that are within their scope of control, such as files on disk, data in a database, or paths within a web server. Examples of such permissions include ACLs on Windows, unix file permissions, or `.htaccess` files on a web server.

There are two main cryptographic primitives for integrity: hashes and signatures. A *hash* takes an input of some arbitrary length, and produces a fixed-length digest or hash of the input. Ideally, any change to the input completely transforms the output. If you store a protected hash of a digital object, you can later detect tampering. Actually, anyone with that hash can detect tampering, so, for example, many software projects list a hash of the software on their website. Anyone who gets the bits from any source can rely on them being the bits described on the project website, to a level of security based on the security of the hash and the operation of the web site. A signature is a cryptographic operation with a private key and a hash that does much the same thing. It has the advantage that once someone has obtained the right public key, they can validate a lot of hashes. Hashes can also be used in binary trees of various forms, where large sets of hashes are collected together and signed. This can enable, for example, inserting data into a tree and noting the time in a way that's hard to alter. There are also systems for using hashes and signatures to detect changes to a file system. The first was co-invented by Gene Kim, and later commercialized by Tripwire, Inc. (Kim, 1994).

Logging technology is a weak third in this list. If you log how files change, you may be able to recover from integrity failures.

Implementing Integrity

If you're implementing a permission system, you should ensure that there's a single permissions kernel, also called a *reference monitor*. That reference monitor should be the one place that checks all permissions for everything. This has two main advantages. First, you have a single monitor, so there are no bugs, synchronization failures, or other issues based on which code path called. Second, you only have to fix bugs in one place.

Creating a good reference monitor is a fairly intricate bit of work. It's hard to get right, and easy to get wrong. For example, it's easy to run checks on references (such as symlinks) that can change when the code finally opens the file. If you need to implement a reference monitor, perform a literature review first.

If you're implementing a cryptographic defense, see Chapter 16. If you're implementing an auditing system, you need to ensure it is sufficiently performant that people will leave it on, that security successes and failures are both logged, and that there's a usable way to access the logs. You also need to ensure that the data is protected from attackers. Ideally, this involves moving it off the generating system to an isolated logging system.

Operational Assurance of Integrity

The most important element of assuring integrity is about process, not technology. Mechanisms for ensuring integrity only work to the extent that integrity failures generate operational exceptions or interruptions that are addressed by a person. All the cryptographic signatures in the world only help if someone investigates the failure, or if the user cannot or does not override the message about a failure. You can devote all your disk access operations to running checksums, but if no one investigates the alarms, they won't do any good. Some systems use "whitelists" of applications so only code on the whitelist runs. That reduces risk, but carries an operational cost.

It may be possible to use SSH or SSL tunneling or IPSec to address network tampering issues. Systems like Tripwire, OSSEC, or L5 can help with system integrity.

Integrity Technologies

Technologies for protecting files include:

- ACLs or permissions
- Digital signatures
- Hashes
- Windows Mandatory Integrity Control (MIC) feature
- Unix immutable bits

Technologies for protecting network traffic:

- SSL
- SSH
- IPSec
- Digital signatures

Non-Repudiation: Mitigating Repudiation

Repudiation is a somewhat different threat because it bridges the business realm, in which there are four elements to addressing it: preventing fraudulent transactions, taking note of contested issues, investigating them, and responding to them. In an age when anyone can instantly be a publisher, assuming that you can ignore the possibility of a customer (or noncustomer) complaint or contested charge is foolish. Ensuring you can accept customer complaints and investigate them is outside the scope of this book, but the output from such a system provides a key validation that you have the right logs.

Note that repudiation is sometimes a feature. As Professor Ian Goldberg pointed out when introducing his *Off-the-Record* messaging protocol, signed conversations can be embarrassing, incriminating, or otherwise undesirable (Goldberg, 2008). Two features of the *Off-the-Record* (OTR) messaging system are that it's secure (encrypted and authenticated) and deniable. This duality of feature or threat also comes up in the LINDDUN approach to privacy threat modeling.

Tactics for Non-Repudiation

The technical elements of addressing repudiation are fraud prevention, logs, and cryptography. Fraud prevention is sometimes considered outside the scope of repudiation. It's included here because managing repudiation is easier if you have fewer contested transactions. Fraud prevention can be divided into fraud by internal actors (embezzlement and the like) and external fraud. Internal fraud prevention is a complex matter; for a full treatment see *The Corporate Fraud Handbook* (Wells, 2011). You should have good account management practices, including ensuring that your tools work well enough that people are not tempted or forced to share passwords as part of getting their jobs done. Be sure you log and audit the data in those logs.

Logs are the traditional technical core of addressing repudiation issues. What is logged depends on the transaction, but generally includes signatures or an IP address and all related information. There are also cryptographic ways to address repudiation, which are currently mostly used between larger businesses.

Tactics for Preventing Fraud by External Parties

External fraud prevention can be seen as a matter of payment fraud prevention, and ensuring that your customers remain in control of their account. In both cases, details about the state of the art changes quickly, so talk to your peers. Even the most tight-lipped companies have been willing to have very frank discussions with peers under NDA.

In essence, stability is good. For example, someone who has been buying two romance novels a month from you for a decade and is still living at the same address is likely the person who just ordered another one. If that person suddenly moves to the other side of the world, and orders technical books in Slovakian with a new credit card with a billing address in the Philippines, you might have a problem. (Then again, they might have finally found true love, and you don't want to upset your loyal customers.)

Tools for Preventing Fraud by External Parties

In their annual report on online fraud, CyberSource includes a survey of popular fraud detection tools and their perceived effectiveness (CyberSource, 2013). Their 2013 survey includes a set of automated tools:

- Validation services
- Proprietary data/customer history
- Multi-merchant data
- Purchase device tracing

Validation services include tracking verification numbers (aka CVN/CVV), address verification services, postal address verification, Verified by Visa/ MasterCard SecureCode, telephone number verification/reverse lookups, public records services, credit checks, and "out-of-wallet/in-wallet" verification services.

Proprietary data and customer history includes customer order history, in house "negative lists" of problematic customers, "positive lists" of VIP or reliable customers, order velocity monitoring, company-specific fraud models (these are usually built with manual, statistical, or machine learning analyses of past fraudulent orders), and customer website behavioral analysis.

Multi-merchant data focuses on shared negative lists or multi-merchant purchase velocity analyzed by the merchant. (This analysis is nominally also performed by the card processors and clearing houses, so the additional value may be transient.)

Finally, purchase device tracking includes device "fingerprinting" and IP address geolocation. The CyberSource report also discusses the importance of tools to help manual review, and how a varied list is both very helpful and time consuming. Because manual review is one of the most expensive components of an anti-fraud approach to repudiation threats, it may be worth investing in tools to gather all the data into one (or at least fewer) places to improve analyst productivity.

Implementing Non-Repudiation

The two key tools for non-repudiation are logging and digital signatures. Digital signatures are probably most useful for business-to-business systems.

Log as much as you can keep for as long as you need to keep it. As the price of storage continues to fall, this advice becomes easier and easier to follow. For example, with a web transaction, you might log IP address, current geolocation of that address, and browser details. You might also consider services that either provide information on fraud or allow you to request decision advice. To the extent that these companies specialize, and may have broader visibility into fraud, this may be a good area of security to outsource. Some of the information you log or transfer may interact with your privacy policies, and it's important to check.

There are also cryptographic digital signatures. Digital signature should be distinguished from electronic signature, which is a term of art under U.S. law referring to a variety of mechanisms with which to produce a signature, some as minimalistic as "press 1 to agree to these terms and conditions." In contrast, a digital signature is a mathematical transformation that demonstrates irrefutably that someone in possession of a mathematical key took an action to cause a signature to be made. The strength of "irrefutable" here depends on the strength of the math, and the tricky bits are possession of the key and what human intent (if any) may have lain behind the signature.

Operational Assurance of Non-Repudiation

When a customer or partner attempts to repudiate a transaction, someone needs to investigate it. If repudiation attempts are frequent, you may need dedicated people, and those people might require specialized tools.

Non-Repudiation Technologies

Technologies you can use to address repudiation include:

- Logging
- Log analysis tools
- Secured log storage
- Digital signatures
- Secure time stamps
- Trusted third parties
- Hash trees
- As mentioned in "tools for preventing fraud" above

Confidentiality: Mitigating Information Disclosure

Information disclosure can happen with information at rest (in storage) or in motion (over a network). The information disclosed can range from the content of communication to the existence of an entity with which someone is communicating.

Tactics for Confidentiality

Much like with integrity, there are two main ways to prevent information disclosure: Within the confines of a system, you can use ACLs, and outside of it you must use cryptography.

If what must be protected is the content of the communication, then traditional cryptography will be sufficient. If you need to hide who is communicating with whom and how often, you'll need a system that protects that data, such as a cryptographic mix or onion network. If you must hide the fact that communication is taking place at all, steganography will be required.

Implementing Confidentiality

If your system can act as a reference monitor and control all access to the data, you can use a permissions system. Otherwise, you'll need to encrypt either the data or its "container." The data might be a file on disk, a record in a database, or an e-mail message as it transits over the network. The container might be a file system, database, or network channel, such as all e-mail between two systems, or all packets between a web client and a web server.

In each cryptographic case, you have to consider who needs access to the keys for encrypting and the decrypting data. For file encryption, that might be as simple as asking the operating system to securely store the key for the user so that the user can get to it later. Also, note that encrypted data is not integrity controlled. The details can be complex and tricky, but consider a database of salaries, where the cells are encrypted. You don't need to know the CEO's salary to know that replacing your salary with it is likely a good thing (for you); and if there's no integrity control, replacing the encrypted value of your salary with the CEO's salary will do just fine.

An important subset of information disclosure cases related to the storage of passwords or backup authentication mechanisms is considered in depth in Chapter 14.

Operational Assurance of Confidentiality

It may be possible to add ACLs to an already developed system, or to use chroot or similar sandboxes to restrict what it can access. On Windows, the addition of a SID to a program and an inherited deny ACL for that SID may help (or it may break things). It is usually possible to add a disk or file encryption layer to protect information at rest from disclosure. Disk crypto will work "by default" with all the usual caveats about how keys are managed. It works for adversarial custody of the machine, but not if the password is written down or otherwise stored with the machine. With regard to a network, it may be possible to use SSH or SSL tunneling or IPSec to address network tampering issues.

Confidentiality Technologies

Technologies for confidentiality include:

- Protecting files:
  - ACLs/permissions
  - Encryption
  - Appropriate key management
- Protecting network data:
  - Encryption
  - Appropriate key management
- Protecting communication headers or the fact of communication:
  - Mix networks
  - Onion routing
  - Steganography

NOTE In the preceding lists, "appropriate key management" is not quite a technology, but is so important that it's included.

Availability: Mitigating Denial of Service

Denial-of-service attacks work by exhausting some resource. Traditionally, those resources are CPU, memory (both RAM and hard drive space can be exhausted), and bandwidth. Denial-of-service attacks can also exhaust human availability. Consider trying to call the reservations line of a very exclusive restaurant—the French Laundry in Napa Valley books all its tables within 5 minutes of the phone being open every day (for a day 30 days in the future). The resource under contention is the phone lines, and in particular the people answering them.

Tactics for Availability

There are two forms of denial of service attacks: brute force and clever. Using the restaurant example, brute force involves bringing 100 people to a restaurant that can seat only 25. Clever attacks bring 20 people, each of whom makes an ever-escalating list of requests and changes, and runs the staff ragged. In the online world, brute force attacks on networks are somewhat common under the name DDoS (Distributed Denial of Service). They can also be carried out against CPU (for example, `while(1) fork()`) or disk. It's simple to construct a small zip file that will expand to whatever limit might be in place: the maximum size of a file or space on the file system. Recall that a zip file is structured to describe the contents of the real file as simply as possible, such as 65,535 0s. That three-byte description will expand to 64K, for a magnification effect of over 21,000—which is awfully cool if you're an attacker.

Clever denial-of-service attacks involve a small amount of work by an attacker that causes you to do a lot of work. For example, connecting to an SSL v2 server, the client sends a client master key challenge, which is a random key encrypted such that the server does (relatively) expensive public key operations to decrypt it. The client does very little work compared to the server. This can be partially addressed in a variety of ways, most notably the Photuris key management protocol. The core of such protocols is proof that the client has done more work than the server, and the body of approaches is called *proof of work*. However, in a world of abundant bots and volunteers to run DDoS software for political causes, Ben Laurie and Richard Clayton have shown reasonably conclusively that "Proof-of-Work Proves Not to Work" (in a paper of that name [Laurie]).

A second important strategy for defending against denial-of-service attacks is to ensure your attacker can receive data from you. For example, defenses against SYN flooding attacks now take this form. In a SYN flood attack, a host receives a lot of connection attempts (TCP SYNchronize) and it needs to keep track of each one to set up new connections. By sending a slew of those, operating systems in the 1990s could be run out of memory in the fixed-size buffers allocated to track SYNs, and no new connections could be established. Modern TCP stacks calculate certain parts of their response to a SYN packet using some cryptography. They maintain no state for incoming packets, and use the cryptographic tools to validate that new connections are real (Rescorla, 2003).

Implementing Availability

If you're implementing a system, consider what resources an attacker might consume, and look for ways to limit those resources on a per-user basis. Understand that there are limits to what you can achieve when dealing with systems on the other side of a trust boundary, and some of the response needs to be operational. Ensure that the operators have such mechanisms.

Operational Assurance of Availability

Addressing brute force denial-of-service attacks is simple: Acquire more resources such that they don't run out, or apply limits so that one bad apple can't spoil things for others. For example, multi-user operating systems implement quota systems, and business ISPs may be able to filter traffic coming from certain sources.

Addressing clever attacks is generally in the realm of implementation, not operations.

Availability Technologies

Technologies for protecting files include:

- ACLs
- Filters
- Quotas (rate limiting, thresholding, throttling)
- High-availability design
- Extra bandwidth (rate limiting, throttling)
- Cloud services

Authorization: Mitigating Elevation of Privilege

Elevation of privilege threats are one category of unauthorized use, and the only one addressed in this section. The overall question of designing authorization systems fills other books.

Tactics for Authorization

As discussed in the section "Implementing Integrity," having a reference monitor that can control access between objects is a precursor to avoiding several forms of a problem, including elevation of privilege. Limiting the attack surface makes the problem more tractable. For example, limiting the number of `setuid` programs limits the opportunity for a local user to become root. (Technically, programs can be `setuid` to something other than root, but generally those other accounts are also privileged.) Each program should do a small number of things, and carefully manage their input, including user input, environment, and so on. Each should be sandboxed to the extent that the system supports it. Ensure that you have layers of defense, such that an anonymous Internet user can't elevate to administrator with a single bug. You can do this by having the code that listens on the network run as a limited user. An attacker who exploits a bug will not have complete run of the system. (If they're a normal user, they may well have easy access to many elevation paths, so lock down the account.)

The permission system needs to be comprehensible, both to administrators trying to check things and to people trying to set things. A permission system that's hard to use often results in people incorrectly setting permissions, (technically) enabling actions that policy and intent mean to forbid.

Implementing Authorization

Having limited the attack surface, you'll need to very carefully manage the input you accept at each point on the attack surface. Ensure that you know what you want to accept and how you're going to use that input. Reject anything that doesn't match, rather than trying to make a complete list of bad characters. Also, if you get a non-match, reject it, rather than try to clean it up.

Operational Assurance of Authorization

Operational details, such as "we need to expose this to the Internet" can often lead to those deploying technology wanting to improve their defensive stance. This usually involves adding what can be referred to as *defense in depth* or *layered defense*. There are several ways to do this.

First, run as a normal or limited user, not as administrator/root. While technically that's not a mitigation against an elevation-of-privilege threat, but a harbinger of such, it's inline with the "principle of least privilege." Each program should run as its own limited user. When unix made "nobody" the default account for services, the nobody account ended up with tremendous levels of authorization. Second, apply all the sandboxing you can.

Authorization Technologies

Technologies for improving authorization include:

- ACLs
- Group or role membership
- Role based access control
- Claims-based access control
- Windows privileges (runas)
- Unix sudo
- Chroot, AppArmor or other unix sandboxes
- The "MOICE" Windows sandbox pattern
- Input validation for a defined purpose

NOTE MOICE is the "Microsoft Office Isolated Conversion Environment." The name comes from the problem that led to the pattern being invented, but the approach can now be considered a pattern for sandboxing on Windows. For more on MOICE, see (LeBlanc, 2007).

NOTE **Many Windows privileges are functionally equivalent to administrator, and may not be as helpful as you desire. See (Margosis, 2006) for more details.**

Tactic and Technology Traps

There are two places where it's easy to get pulled into wasting time when working through these technologies and tactics. The first distraction is risk management. The tactics and technologies in this chapter aren't the only ways to address threats, but they are the best place to start. When you can use them, they will be easier to implement and work better than more complex or nuanced risk management approaches. For example, if you can address a threat by changing a network endpoint to a local endpoint, there's no point to engaging in the more time consuming risk management approaches covered in the next chapter. The second distraction is trying to categorize threats. If you found a threat via brainstorming or just the free flow of ideas, don't let the organization of this chapter fool you into thinking you should try to categorize that threat. Instead, focus on finding the best way to address it. (Teams can spend longer in debate around categorization than it would take to implement the fix they identified—changing permissions on a file.)

Addressing Threats with Patterns

In his book, *A Pattern Language*, architect Christopher Alexander and his colleagues introduced the concept of architectural patterns (Alexander, 1977). A *pattern* is a way of expressing how experts capture ways of solving recurring problems. Patterns have since been adapted to software. There are well-understood development patterns, such as the three-tier enterprise app.

Security patterns seem like a natural way to group and communicate about tactics and technologies to address security problems into something larger. You can create and distribute patterns in a variety of ways, and this section discusses some of them. However, in practice, these patterns have not been popular. The reasons for this are not clear, and those investing in using patterns to address security problems would likely benefit from studying the factors that have limited their popularity.

Some of those factors might include engineers not knowing when to reach for such a text, or the presentation of security patterns as a distinct subset, apart from other patterns. At least one web patterns book (Van Duyne, 2007) includes a chapter on security patterns. Embedding security patterns where non-specialists are likely to find them seems like a good pattern.

Standard Deployments

In many larger organizations, an operations group will have a standard way to deploy systems, or possibly several standard ways, depending on the data's sensitivity. In these cases, the operations group can document what sorts of threats their standard deployment mitigates, and provide that document as part of their "on-boarding" process. For example, a standard data center at an organization might include defenses against DDoS, or state that "network information disclosure is an accepted risk for risk categories 1–3."

Addressing CAPEC Threats

CAPEC (MITRE's Common Attack Pattern Enumeration and Classification) is primarily a collection of attack patterns, but most CAPEC threat patterns include defenses. This chapter has primarily organized threats according to STRIDE. If you are using CAPEC, each CAPEC pattern includes advice about how to address it in its "Solutions and Mitigations" section. The CAPEC website is the authoritative source for such data.

Mitigating Privacy Threats

There are essentially three ways to address privacy threats: Avoid collecting information (minimization), use crypto in various clever ways, and control how data is used (compliance and policy). Cryptography is a technology, while minimization and compliance are more tactics you can apply. Each requires effort to integrate into your design or implementation.

Minimization

Perhaps obviously, it is impossible to use information you don't have in a way that impacts someone's privacy. Therefore, minimizing your collection and retention of information reduces risk. Minimizing what you collect is by far more reliable than attempting to use policy controls on the data. Of course, it also eliminates any utility that you can get from that data. As such, minimization is generally a business call regarding risk and reward. Over the past decade, with breach disclosure laws, the balance of factors related to decisions about the collection and retention of information have changed dramatically. Some legal scholars have gone so far as to compare personal data to toxic waste. Holly

Towle, an attorney specializing in electronic commerce, offers 10 principles for handling toxic waste, or personally identifying information (PII). Each of these is addressed in depth in her article (Towle, 2009):

- Do not touch it unless you have to.
- If you have to touch it, learn how or whether to do so—mistakes can be fatal or at least seriously damaging.
- Do not use normal methods to transport (transfer) it.
- Attempt to crack the whip over contractor handling it.
- Do not store some of it at all.
- Store what you need but in a manner avoiding spills, and limit access.
- Be alert for suspicious odors and other red flags.
- Report spills to the relevant people and agencies.
- Dispose of it only by special means.
- Get ready to be sued or incur often unreasonable expenses no matter how much care you take.

Minimization is a conceptually simple way to address privacy. In practice, however, it can become complex and contentious. The value of collecting data is easy to see, and it's hard to know what you'll be unable to do if you don't collect it.

Cryptography

There are a variety of ways to use cryptographic techniques to address privacy concerns. The applicability of each is dependent on the threat model, in the sense of who you're worried about. Each of these techniques is the subject of a great deal of research, so rather than try to provide a full description of each technique and risk leaving out key details, the following sections explain where each is useful as a response to a threat.

Hashing or Encrypting Data

If your privacy concern is someone accidentally viewing data, or running simple database queries, it may help to encrypt the data. If you want a record that can only be accessed once someone has a specific string (such as an e-mail address or SSN), you can use a cryptographic hash of that data. For example, if you store `hash(adam.shostack@example.com)`, then only someone who knows that e-mail address can look it up.

WARNING **Simple hashing doesn't protect your data if the attacker is willing to build a "dictionary" and hash each term in the dictionary. For SSNs, that's only a billion hashes to run, which is cheap on modern hardware. Hashing is therefore not the right defense if your data is low entropy, either because the data is short strings, or because it's highly structured. In those cases, you'll likely want to encrypt, and use a unique key and initialization vector per plaintext.**

Split-Key Systems

Splitting keys is useful when you're concerned about the threat of someone decrypting the data without authorization. It's possible to encrypt data with multiple keys, such that all or some fraction of the keys is needed to decrypt it. For example, if you store $m = e_{k1}(e_{k2}(plaintext))$, then to decrypt `m`, you need the party that holds `k1` to decrypt the `m`, then send that to whomever holds `k2`.

If you're worried about availability threats, there are split-key cryptographic systems for which the keys are mathematically related, and you only need `k-of-n` keys to get the plaintext out of the system. Such systems encrypt the data with `n` keys. Those `n` keys are mathematically related in ways which allow any `k` of the `n` keys to decrypt the data. These are useful, for example, for backing up the master key to a system where that master key may be needed a decade later. Such a system is used to backup the root keys for DNSSEC.

Private Information Retrieval

If the threat is a database owner watching a client's queries and learning from them, then a set of techniques called *private information retrieval* may be useful. Private information retrieval techniques are generally fairly bandwidth intensive, as they often retrieve far more information than is desired to get at the data without revealing anything to the database owner.

Differential Privacy

When the threat is a database client running multiple queries to violate the database owner's privacy policies, *differential privacy* provides the database owner with a way to first measure how much information has been given out, and then stop answering queries that provide additional information. This does not mean that the database needs to stop answering queries. Many queries will not change, or differentiate, the amount of information that can be inferred, even after the database has reached a specified privacy limit.

NOTE **Differential privacy offers very strong protection for a very specific definition of privacy.**

Mixes and Mix-Like Systems

A *mix* is a system for preventing traffic analysis and providing untracability to message senders or recipients. That is, an observer should not be able to trace a message back to a person after it has been through a mix. Mixes work by maintaining a pool of messages, and now and then sending messages out. To avoid trusting a single mix, there may be a network of mixes operated by different parties.

There are two major modes in which mixes operate: interactive-time and batch. Interactive-time mixes can be used for scenarios like web browsing, but are less secure against traffic analysis. There are also interactive systems that do not mix traffic but aim to conceal its source and destination. Such interactive systems include Tor.

Blinding

Blinding helps defend against surveillance threats that use cryptographic keys as identifiers. For example, if Alice is worried that a certificate authority might track her vote, then she might want a voting registration system that can do deep checking to ensure that she is authorized to vote, providing her with an anonymous voting chit that can be used to prove her right to vote. Online, this can be done through the use of blinding.

Blinding is a cool math trick that can solve real problems. The math may look a little intimidating, but you can understand it with high school algebra. Think of signing as doing exponentiation modulo p. (Modulo is a remainder—1 mod 12 is 1, 14 mod 12 is 2, where 14 mod 10 is 4. The modulo math is needed for certain security properties.) Therefore if s is the signature, a is Alice's key, and c is the CA's key, then a signature is $s = a^c \mod p$. Normally, the CA calculates the signature, and sends s back to Alice. Now the CA knows s, and can use that knowledge. So how can the CA calculate s without knowing it? Because multiplication is commutative, the CA can calculate something related to s. Blinding works by Alice multiplying her key by some blinding factor (b) before sending the product of that multiplication (ab) to the CA. The CA then calculates $s = (ab)c \mod p$, and sends s to Alice. Alice then divides s by b, and $s/b = ac$. So Alice now knows s/b, which appears for all the world like a signature on a, but the CA doesn't know that a is associated with Alice. The math shown here is a subset of what's needed to do this securely, with the goal of giving you an idea of how it works. Proper blinding requires that you deal with a plethora of mathematical threats, which are covered in a book such as (Ferguson, 2012).

Compliance and Policy

To the extent that a business decision has been made to gather and store sensitive information about people, the organization needs to put controls around it. Those controls can either be policy or technical, and they can meet either your business need or regulatory needs, or both. These approaches are not as crisp as the tools and tactics you can apply to security problems, and to the extent that you can apply minimization or cryptography to privacy problems, it will be easier and more effective.

Policies

The first class of controls are organizational policies that specify who can do what with the information. From the technologist's perspective, these can be frustratingly vague statements, such as "only authorized people will be allowed access." However, they are an important first step in setting requirements. From a statement like that you can derive a technical approach, such as "only a security group can access the data." Then you'll need to ensure that that policy is enforced across a variety of information systems, and then you're at the level of tactics and technologies.

Regulatory Requirements

Personal data is subject to a long and complex list of privacy rules that differ from jurisdiction to jurisdiction. As with everything else covered in this book, this section on mitigating privacy threats is not intended to replace proper legal advice.

There's one other thing to be said about mitigating privacy threats to your organization. In many cases, organizations are required by law to collect and protect a set of information that they must treat as toxic waste, at great expense. It makes a great deal of privacy sense for organizations and their industry groups to argue against requirements to gather such data. That includes rolling back existing mandates and holding firm against new mandates to collect data that you'd prefer not to hold.

Summary

The best way to address threats is to use standard, well-tested features or products that add security against the threats you've identified. These tactics and technologies are available to address each of the STRIDE threats. There are tactics and technologies available to both developers and operations.

Authentication technologies mitigate spoofing threats. You can authenticate computers, bits, or people. Integrity technologies mitigate tampering threats. Generally, you want integrity protection for files and network connections. Non-repudiation technologies mitigate repudiation, which can include fraud and other repudiations. Anti-fraud technologies include validation services, or use of customer history that's either local or shared by others. There are also a variety of cryptographic and operational measures you can take to increase assurance around your logs. Information disclosure threats are addressed by confidentiality technologies. Those can be most easily applied to files or network connections; however, it can also be important to protect container data, such as filenames or the fact of communication. Preventing denial of service involves ensuring that code doesn't have arbitrary limits that prevent it from taking advantage of all the available resources. Preventing elevation of privilege generally works by first ensuring that the code is constrained by mechanisms such as ACLs, and then by more complex sandboxes.

Patterns are collections of tactics and technologies. They seem like a natural approach. For reasons which are unclear, they haven't really taken off, and those who are considering using them would be advised to understand why.

Mitigating privacy threats is best done by minimizing what you collect, and then applying cryptography; however, there are limits to the tactics and technologies available, and sometimes you must fall back to compliance tools or policy.

The issue of standard tactics and technologies not being applicable everywhere is not limited to privacy. In the next chapter, you'll learn about making structured tradeoffs between ways to address threats.

CHAPTER

11

Threat Modeling Tools

This chapter covers tools to help you threat model. Tooling can help threat modeling in a number of ways. It can help you create better models, or create models more fluidly. Tools can help you remember to engage in various steps, or provide assistance performing those steps. Tools can help create a more legible or even beautiful threat model document. Tools can help you check your threat model for completeness. Finally, tools can help you create actionable output from a threat model.

Tools can also act as a constraint. You may find yourself stymied by usability issues, such as fields you're unsure how to fill out. Or you might find that a tool cramps your style. Some trade-offs are unavoidable as tools are created, so the chapter starts with general tools that are useful in threat modeling, and then progresses to more specialized tools.

A few disclosures: I do not have personal experience with each tool described here, and some of the tools I created myself. (Those are treated at greater length, because there's less risk of me insulting the authors.)

This chapter starts by describing some generally useful tools and how to apply them to threat modeling. You'll then learn about the open-source tools that are available, followed by commercial tools. The chapter closes with a few words about tools that don't yet exist.

Generally Useful Tools

This section discusses tools that are not specialized for threat modeling but can be tremendously useful. It covers a few of the more useful tools to encourage you to think about the tools you already use and with which you are familiar.

Whiteboards

I can hardly imagine threat modeling without a whiteboard. No technology I've used has the immediacy, flexibility, and visibility to a group than a whiteboard when iteratively drawing system architecture. Whiteboards also have the advantage of transience—drawing on paper just isn't the same. On a whiteboard, no one tries to correct details such as a line not being connected properly, so the discussion can be focused on how the system actually works.

For distributed teams, a webcam focused on a whiteboard may work, or you may have "virtual whiteboarding" technologies that work for you.

Office Suites

Microsoft Office contains a number of tools that are very useful in threat modeling. Word is a great tool for recording threats in free-form. What to record is dependent on the approach you've chosen. Excel can be used for issue tracking and status. Visio is great for turning whiteboards into more precise documents. Of course, Office is one of several suites with word processing, spreadsheet, and drawing functionality. The only caveats would be the limitations of the tools. The document tool should be more than text—a feature such as embedded images is extremely useful. Similarly, use a vector drawing tool that enables you to move symbols as symbols. Automatic connector management is also super-useful, and of course this feature is not unique to Visio.

To state the obvious, Microsoft Word, Excel, and Visio are commercially licensed tools.

Bug-Tracking Systems

Whatever bug-tracking system you use should also be used to track threats. A good bug from threat modeling can take many forms. The form you use will influence how you title and discuss bugs, and there is no universally right way to approach it. (The right way is the way that works best for you and your organization.) The title could express any of the following:

- **The threat itself**: Here the bug title is of a form such as "an attacker can threaten the component" or "the component is vulnerable to threat."

For example, "the front end is vulnerable to spoofing because we use reusable passwords."

- **The mitigation:** Here, the bug title is of a form such as "the component needs mitigation." For example, "the front end needs to run only over SSH." In the text of the bug, you should also explain the threat.
- **The need to test a mitigation:** This is what you can title a bug if someone says, "Oh, the front end isn't vulnerable to that." Rather than absorb time in the meeting to discuss or check the threat, file a bug, "Test front end vulnerability to *threat*" and ensure that there are good tests for the bug.
- **The need to validate an assumption:** These bugs are filed to ensure that someone follows up on an assumption you discover while threat modeling, and on which you depend for a security property. The bug should have a title such as "security depends on assumption A" or "security property X of component Y depends on assumption Z." For example, "Security depends on the assumption that no one would ever find the key in the fake rock that looks exactly like the rocks at our last house."
- **Other tracking items:** You should treat the preceding items as suggestions, not a form into which all bugs need to fit. If you find something worth tracking, file a bug.

When tracking security bugs from threat modeling, there are a few fields that can make running queries and analysis more reliable. These include whether the bug is a security bug, whether it's "stop ship," and how the bug was found (for example, threat modeling, fuzzing, code review, customer report). You can also check the tables in Chapter 7, "Processing and Managing Threats," and Chapter 9, "Trade-Offs When Addressing Threats," for useful fields. For example, you might want a risk management approach field whose values could be avoid, address, accept, or transfer.

The right fields to use will depend in large part on the queries you want to run, which of course depend on the questions you want to ask. Some questions you might want to ask include the following:

- Do we have any open security bugs?
- Do we have any open threat modeling bugs?
- Do we have any high-severity threat modeling bugs left to fix?
- How much risk are we transferring to end-users in the security operations guide or via warning dialogs?
- What department head has signed off on the largest business risk? Which department head has signed off on the most risks?

Open-Source Tools

A variety of open-source tools for threat modeling are available. The open source tools illustrate some of the challenges in creating a high-quality threat modeling tool.

TRIKE

There are two tools named TRIKE. The first was a standalone desktop tool, written in Smalltalk. That tool is no longer being maintained, and TRIKE is now implemented in a spreadsheet. According to documentation, it works best in Excel 2011 for the Macintosh (Trike, 2013). TRIKE is sometimes referred to as "OctoTrike."

TRIKE does not fit cleanly into the four-stage framework defined in this book. The TRIKE spreadsheet contains 19 pages, which are grouped as follows: one overview, seven main threat pages (actors, data model, intended actions, connections, protocols, threats, and security objectives), four record-keeping pages (use case index, use case details, document index, and development team) and seven reference sheets (actor types, data types, action, network layers, meaningful threats, intended response, and guide words). As of this writing, the help spreadsheet appears to be a reference document, not an introduction of the system.

SeaMonster

SeaMonster is an Eclipse-based attack tree and misuse case tool that was developed by students at the Norwegian University of Science and Technology. It appears to be abandoned since 2010 (SeaMonster, 2013). The code is still available.

Elevation of Privilege

Elevation of Privilege (the game) is designed to be the easy way to get started threat modeling. It works by inviting individuals to participate in a game. The game consists of 74 physical playing cards in six suits, named for the STRIDE threats, with most suits having cards 2 through Ace. Two suits have fewer cards in order to avoid redundant threats, and it was challenging to find broadly applicable threat instances that were easily explained on a card. Each card has a specific instance of a STRIDE threat. For example, the 6 of Tampering reads "An attacker can write to a data store your code relies on." Another example card is shown in Figure 11-1.

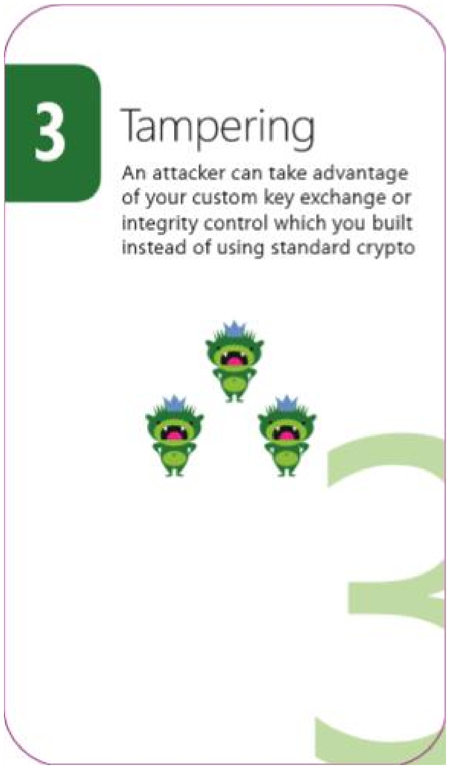

3
Tampering
An attacker can take advantage of your custom key exchange or integrity control which you built instead of using standard crypto

Figure 11-1: An *Elevation of Privilege* card

The *Elevation of Privilege* files can be downloaded from `http://www.microsoft.com/sdl/adopt/eop.aspx`. Before starting *Elevation of Privilege*, participants or the game organizer create a diagram of a system being modeled. People then come together for a game. The organizer explains the rules, and may ask people to "put their skepticism on hold." The game starts by dealing out the deck, and is then structured into turns. The first card played is always the 3 of Tampering. Play proceeds around the table in hands.

Each hand starts with a player selecting a suit to lead and playing in that suit. Each player plays by selecting a card and connecting it to the diagram. The player must play in the suit that was led if they have a card in that suit. If they don't, they may play any card. When play has gone once around the table, the hand ends. The player who played the highest card wins the hand. The highest card is either in the suit that was led, or, if a card in the Elevation of Privilege suit card was played, the highest card played from the EoP played wins the hand. (All Elevation of Privilege threat cards are higher ranked than the suit that was led, and only Elevation of Privilege cards can win when someone leads in another suit.) Players get a point for connecting the threat on their card to the diagram with a "buggable threat," and a point for winning the hand by playing the highest card either in the suit that was led or in EoP. Any EoP card trumps the suit that was led. To encourage creativity, each ace card says "You've invented a new threat," and the threats are enumerated on cards included in the pack. The game ends either when time allocated has elapsed or when all the cards have been played. The winner is the player with the most points.

A buggable threat is one a team identifies and is willing to file a bug for. It's a simple and implicit element of most software development. Some teams may find it more helpful to ask "would we add that to the backlog?" However you want to approach it (in the context of the game), you want an understandable and shared bar to test threats, so that you focus on finding the good ones.

Games are less threatening than "serious" work, and they provide structure and hints to the beginner, enabling new players to find a threat based on the cards in their hands. The game is also intended to help players find a *flow state*, a concept that is covered in depth in Chapter 19, "Architecting for Success." EoP is covered in greater depth in my paper *Elevation of Privilege*: *Drawing Developers into Threat Modeling* [Shostack, 2012].

Microsoft makes the files (source and PDF) available under a Creative Commons BY-3.0 license, allowing you to take it, modify it, make derivative works, and even sell them.

Commercial Tools

Here are a few commercially licensed threat modeling tools. I mention a few commercial tools as examples, but *caveat emptor*.

ThreatModeler

ThreatModeler from MyAppSecurity.com is a defense-oriented tool based on data elements, roles, and components. It uses a set of attack libraries, including the MITRE CAPEC (see Chapter 4, "Attack Trees"), the WASC threat classification, and others. The tool generates attack trees with the component as the root, requirements that can be violated as a first level of subnode, and then threats and attacks as the next layers. According to the documentation, ThreatModeler is intended to be used by architects, developers, security professionals, QA professionals, or senior executives. ThreatModeler requires Windows.

Corporate Threat Modeller

Corporate Threat Modeller from SensePost is a tool built to support a methodology designed after an analysis of the strengths and weaknesses of a number of threat modeling approaches. Those approaches included threat trees and OCTAVE, a US-CERT-originated system for threat modeling a business (White, 2010). The analysis also looked at Microsoft's SDL Threat Modeling Tool v3, and the Microsoft "IT Infrastructure Threat Modeling Guide," (McRee, 2009) which shows how to use STRIDE-per-element to threat model IT infrastructure.

The Corporate Threat Modeler was explicitly designed for consultants. Insofar as it was developed with an explicitly stated target user (not "everyone"), it is one of the most interesting tools on the market. The approach starts with an architectural overview, and then applies a somewhat complex risk equation. The tool is free to download.

Secur*IT*ree

Secur*IT*ree is threat risk software from Amenaza Technologies, which launched in 2007 to positive reviews (SC Magazine, 2007). The product seems like a well thought through tool for constructing, managing and interpreting threat trees. It contains not only the ability to manage trees, but a set of ways to filter those trees. Each node in the tree has behavioral/capability indicators: a cost to execute, noticability, and a technical ability. It also has impact/attacker benefit indicators of attacker gain and victim loss, along with stored notes for a node or subtree. You can filter the tree based on a given attacker ability. Secur*IT*ree comes with a library of threat trees, which is likely to help its customers get to the interesting part of the threat modeling work faster. Secur*IT*ree also includes some excellent screencast-delivered training (Ingoldsby, 2009). Secur*IT*ree runs on Windows, Mac, and Linux.

Little-JIL

If you're making use of threat trees at a research institution, the Little-JIL software may be helpful. "Little-JIL is a graphical language for defining processes that coordinate the activities of autonomous agents and their use of resources during the performance of a task." It has been used for creating an elections process model and a set of fault trees for that model (Simidchieva, 2010). The full fault trees are available as a graphML model. The software used to create and process the models may be freely used at research institutions (Laser, undated).

Microsoft's SDL Threat Modeling Tool

Microsoft has shipped at least four families of threat modeling tools. They are the *Elevation of Privilege* card game, the SDL Threat Modeling Tool v3, the Threat Analysis and Modeling Tool, and the Threat Modeling Tool v1 and 2. I was the project lead for *Elevation of Privilege* and the SDL Threat Modeling Tool v3 and 3.1. The currently available SDL Threat Modeling Tool is (or has been) available free from Microsoft.

The SDL Threat Modeling Tool v3 was designed in reaction to the complexities and usability issues encountered when engineers who were not threat modeling experts tried to use the older tools. It was the first tool designed around the four-stage framework. The tool has four major screens, designed around the tasks that naturally fit together: Draw Diagrams, Analyze Model, Describe Environment, and Generate Reports. The Draw Diagrams screen, shown in Figure 11-2, includes both the capability to draw diagrams with a constrained Visio stencil set and a diagram validation section with heuristics. The Analyze Model screen, shown in Figure 11-3, is automatically filled out with threats according to STRIDE-per-element.

Each threat instance contains a set of guiding questions to help engineers think through the threat, and an area to record the threat, mitigation, to track whether work on the threat is complete, and to file a bug. The Describe Environment screen is something of a catch-all to track assumptions, external notes and the context of the threat model. The Reports screen includes an all-up report, an open issues report, a list of bugs, and a diagrams-only report intended to facilitate printing. The tool also contains a manual, a sample threat model (for the tool itself), and a getting started guide, all accessible via the Help menu.

As shown in Figure 11-2, the main Draw Diagrams screen contains the following, clockwise from upper left (excluding the menu):

- The diagrams control, enabling you to create sub-diagrams
- The Visio shapes you can use as diagram elements
- The “default” diagram (discussed in the next paragraph)
- The numbered Screens control
- Diagram validation (feedback)
- A help pane

The default diagram is present because human factor testing has shown that less experienced threat modelers are sometimes stymied by a blank screen. Providing them with a starting diagram serves two purposes. One, it demonstrates what is expected in that space. Two, rather than needing to create a diagram, a novice can modify what’s there, which is an easier task.

One other feature worth mentioning from Figure 11-2 is the help field. Generally, help is a menu option that software engineers ignore, because they believe they’re too smart to need to read what they expect will be a badly written help file. Therefore, the tool has basic help onscreen.

The Analyze Model screen shown in Figure 11-3 has two panes. The left pane is a list of diagram elements and the threats associated with them, presented as a tree with a single level of branches. The right pane is titled with the element name (“Results”) and a description of the element. Under that is a reminder of the STRIDE-per-element threats to which it is subject, and an option to not generate threat placeholders, with a reason box. A large portion of the screen is devoted to onscreen guidance: "Some questions to ask about this threat type." The guidance is specific to threat and element category (process, data store, data flow, or external entity).

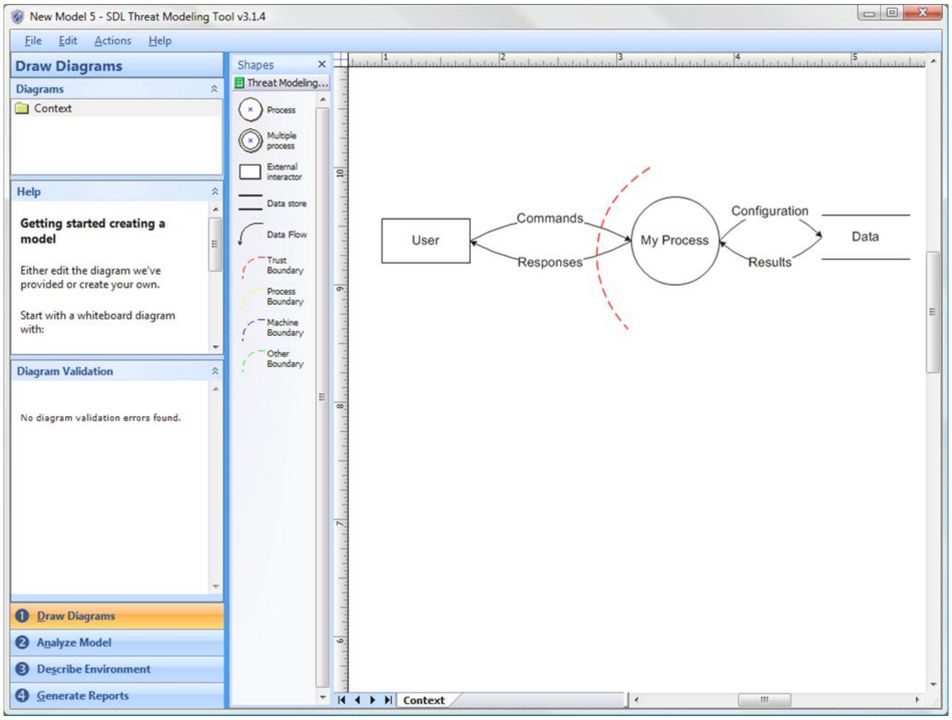
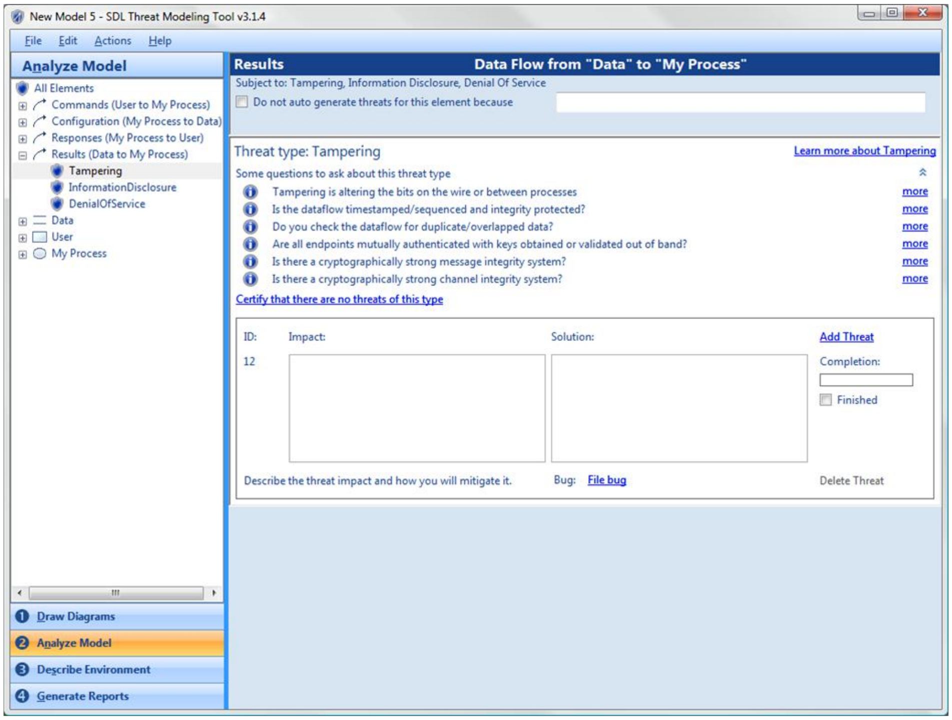


Figure 11-2: The SDL Threat Modeling Tool "Draw Diagrams" screen

There's also a command link to "Certify that there are no threats of this type." The word *certify* was carefully chosen to convey gravity. The last element on the screen is where the threat modeler describes the threat impact, and how it will be mitigated. Most of that is done in two large text entry boxes, but there is also a Finished check box, a "file bug" command link, and a completion bar. The completion bar (shown empty, under the word *completion*) fills out in four segments to encourage text entry in the Impact and Solution fields, as well as checking "finished" and filing a bug. There is also an Add Threat command link in case someone discovers an additional tampering threat against the results data flow.

The bug filing is intentionally abstracted into an API, and the tool ships with sample code to connect to a variety of bug tracking systems, or allow you to connect to whatever you use. When developing the tool, we intentionally spent time to create an API and to ship it under the Microsoft Public License (an Open

Source Initiative Approved License), because bugs are a critical part of ensuring that the threat model leads to something actionable.



Results Data Flow from "Data" to "My Process"

Subject to: Tampering, Information Disclosure, Denial Of Service

Do not auto generate threats for this element because

Threat type: Tampering Learn more about Tampering

Some questions to ask about this threat type

- Tampering is altering the bits on the wire or between processes more
- Is the dataflow timestamped/sequenced and integrity protected? more
- Do you check the dataflow for duplicate/overlapped data? more
- Are all endpoints mutually authenticated with keys obtained or validated out of band? more
- Is there a cryptographically strong message integrity system? more
- Is there a cryptographically strong channel integrity system? more

Certify that there are no threats of this type

ID: 12 Impact: Solution: Add Threat

Completion:

Finished

Describe the threat impact and how you will mitigate it. Bug: File bug Delete Threat

Figure 11-3: The SDL TM Tool Analyze Model screen

NOTE Within the Draw Diagrams screen, the term "validation" causes consternation when diagrams don't conform. For example, one heuristic is to show where all data comes from. That can be addressed by including three extra elements: an installer external entity, a data flow, and a trust boundary. Doing so is not obvious, and requires roughly 10 steps. In a future version, it would be great to see diagram feedback that includes advice on how to address each. Also, boxed trust boundaries would be one of several improvements that could be made to the shapes in the default set. Others include rounded rectangles for processes, less curvy lines, and better positioning of text.

The SDL TM Tool v3.1 series is a no-cost download from Microsoft (`www.microsoft.com/security/sdl/adopt/threatmodeling.aspx`) and it requires Visio 2007 or 2010 to use. The tool is compatible with the Visio 2010 evaluation version. Newer versions of the tool may become available (after this book goes to press) with different dependencies.

Tools That Don't Exist Yet

There are two categories of features that people often ask for that are worth a brief discussion: automated model creation and automated threat identification. A great many people want tools that can take a piece of software that's already been written and extract a data flow or other architectural diagram. This is an attractive goal, and one that might be feasible for programs written in strongly typed languages. Marwan Abi-Antoun has done some work showing how to extract data flow diagrams for Java (Abi-Antoun, 2009). (The system with open code and published DFDs he found to use for testing was only 3,000 lines of code.) If technology to do this is further developed, it will present great value to threat modeling, but also create a temptation to not perform any threat modeling or analysis until late in a project. Threat modeling after the code has been completed limits options for addressing issues. This is discussed further in Chapter 7, "Processing and Managing Threats" and Chapter 17, "Bringing Threat Modeling to Your Organization."

Similarly, tools that can take a diagram or other model and produce lists of threats would be lovely. A Spanish graduate student, Guifré Ruiz, and colleagues have created a first version of such a tool (Ruiz, 2012). However, these tools carry a risk that security analysis will focus only on known threats from an attack library. Such tools cannot (currently) analogize from closely related threats the way an experienced person can. Threat analysis that could reliably extend that knowledge to prevent new systems from making mistakes others have made would be a useful step forward. As more such tools are developed, it will be important to consider the balance between human and automated security design analysis. After all, to the extent that you need software engineers to create new functionality, that new functionality and the new combinations that result may expose new threats. It's not impossible to imagine a tool that would find threats against code not yet written, but it's hard to imagine one that would do so as comprehensively as an expert threat modeler.

Summary

A wide variety of tools are available for threat modeling. General-purpose tools such as whiteboards and bug-tracking systems can be very helpful, and tools such as word processors, spreadsheets, and diagramming tools can be used to help you threat model. Also available are a variety of specialized threat modeling tools. Microsoft has shipped several of these free, including *Elevation of Privilege* and the SDL Threat Modeling Tool, and you can find other commercial and open-source tools that may aid your efforts. There is also demand for tools that can automate model creation or threat identification, although such tools may come at a high price if they appear to find threats while missing new threats or are used too late in the development process.

CHAPTER

13

Web and Cloud Threats

In many ways, threat modeling for the web and cloud are very much like threat modeling for anything else, but these unique environments have some recurring threats, which are covered in this chapter.

This chapter is organized into web threats, cloud threats, cloud provider threats, and mobile threats. Web threats are broken into website threats, web browser, and plugin threats. Many of the cloud threats are expressed with respect to infrastructure as a service (IaaS) and platform as a service (PaaS). It closes with a section on mobile threats.

Web Threats

The web is composed of a simple and powerful set of protocols and languages. It has become a cliché to say that it has changed everything. It's easy to forget that the web is software like other software. Although you might assume that you need to threat model it in some new ways, the truth is that it's like most other software, so techniques such as STRIDE and attack trees work well for web technologies.

Website Threats

Public websites receive large amounts of scrutiny, and suffer from all that the world can throw at them. The classic STRIDE threats all apply, as do a slew of web-specific attacks that happen when you forget that there's a trust boundary between them and the apparently nice doggy that is wagging its tail and slobbering in remarkably cleverly formed SQL in your forms and JavaScript in your URLs in order to cause harm.

Usually, threats such as SQL injections and XSS are handled later in the software engineering process. You'll be developing using patterns, libraries, and frameworks that make each threat less likely, using appropriate testing tools to catch problems during testing, and watching the logs after deployment. So you're done, right? Unfortunately, no. You should be threat modeling to find the unique threats your site will be vulnerable to, such as your ad provider, your analytics code, and that authentication database you're using from some crazy start-up in San Francisco. A standard data flow diagram (DFD) showing where the data comes from for your server is essential, and a client-side DFD is also a pretty good idea, if only to ensure that you have a good test suite. A client-side DFD can also be a good way to create a list that helps you track when your dependencies issue security updates.

Web Browser and Plugin Threats

The web browser has become people's primary portal onto the Internet, and occasionally their last line of automated defense against attacks. Anyone considering building a web browser needs at least one expert with a deep knowledge of the history of browser security issues.

Browser companies could substantially help matters by being super-diligent about their security goals, and how those goals manifest between tabs, websites, and the OS hosting the browser. Clarity from browser creators on how to create a plugin that does not violate security would also be most welcome.

Web plugins or add-ons that extend browser security are becoming less and less common, in part because the two best-known and widely deployed plugins, Java and Flash, have both suffered serious and ongoing security problems. Another reason plug-ins are becoming less common is Apple's willingness to exclude Flash from the iPhone.

Any of the building blocks for finding threats can be applied to a web browser. For example, the STRIDE threats all apply to a web browser. There's spoofing of web pages for phishing or other goals, cross-tab tampering and tampering with files included from other servers. There's information disclosure about browser history vis CSS sniffing, and similar examples exist for each STRIDE threat. There are also very specific attack libraries available for web browsers and website designers.

If you're going to create a browser plugin, there are two unique elements to consider: the browser's security model and the browser's privacy model. You should also realize that auto-update is important and must be done securely.

Browser Security Model

You must deeply understand and respect the browser's security model, and not accidentally break it. This security model includes elements such as the same-origin policy, the boundaries between pages, and what can and can't open a new window, resize a window, and so on. You must also remember to treat the other sides of connections as malicious with respect to the browser. That is, from the browser's perspective, your component that sits on a web server could be under the control of an attacker, who can then send malicious content to your plugin and compromise additional systems. (Similarly, your plugin may be modified or run through a proxy that rewrites data to attack the server components.) However, there are times when breaking the browser security model is intentional and appropriate. For example, some security testing plugins do so.

Several experts have told me in all seriousness that browser security models are now so complex that I should not even write a section about this. I'm tempted to say that browser plugins, like crypto, is a domain for which you need expertise and penetration testing. It would be wonderful if browser manufacturers could fix this, and offer easier to understand plugin models so that plugins could be developed without putting the people who install them at risk. For a history of the three flaws in one popular plugin, see Mark Pilgrim's article "Avoid Common Pitfalls in Greasemonkey" (O'Reilly Network, 2005). (As an aside, Pilgrim's blog post is a good example of a "Note to API Callers," as discussed in Chapter 7, "Processing and Managing Threats.") For a book-length treatment of the full complexity of modern browsers, see Michal Zalewski's *The Tangled Web* (No Starch Press, 2011).

Browser Privacy Model

Similar to respecting the browser's security model, your plugin should respect the browser's privacy model. You should not allow tracking or surveillance in any way that the browser does not, and you should ensure that your controls are at least as accessible as those offered by the browser.

Auto-Update

It is very likely that you will have numerous security issues with your plugin, and you should therefore ensure that it's easy to report bugs to you, and that your updater works well with the browser's auto-update mechanism, and that you have a security update process that can deliver security updates without any tradeoffs such as new or changed user interfaces, new licenses, or similar impediments to upgrading.

Cloud Tenant Threats

You can use an attacker grouping approach to break out cloud threats. There are two main classes of new attackers (threats) when you move an IT system to the cloud: those from insiders at the cloud operator, and those from your fellow tenants of the cloud system. There are generally some new instances of availability threats, based on the increased complexity of connectivity. There are also two sets of legal threats: those that add complexity and/or effort or reduce the assurance of compliance, and the different legal standards around data given to third parties. Lastly, there's a hybrid of those legal threats, which are threats to forensic response. In this section, you'll see the term *cloud provider* used to refer to an organization that offers any combination of infrastructure, platform, or software as a service. Their customer is you, and like their other customers, you are a tenant of their service. Attackers might be tenants, or those who have broken in.

Insider Threats

When you move your data or operations to someone else's cloud, you add a trust boundary. That boundary has the employees and contractors of the cloud operator inside of it, with your data. As administrators, they have unavoidable technical access to the data you provide. They may intentionally attack you, fall victim to an attack themselves, accidentally misconfigure software, or fail to perform maintenance, such as wiping disks between re-allocations.

There are two ways to mitigate this threat: contractually and cryptographically. Contractual approaches dominate today because they're easier; and for most of the risks, it turns out that a contract is sufficient. However, contracts may not be subject to negotiation unless you're spending lots of money. Unfortunately, companies often use contracts to protect personal information, where much of the risk is external to the companies signing the contracts.

The cryptographic approach is to encrypt the data (and possibly obfuscate the code) before sending it. This is easier with a cloud storage system than with software as a service. Well-encrypted, integrity-protected, and authenticated data can be stored anywhere with a very low reduction in safety. (The encrypted state of the data will influence what processing can be done on the encrypted data. There is interesting cryptographic research on processing encrypted data, which as of this writing usually isn't part of the standard crypto libraries that you should use.) Of course, the keys need to be stored securely, or you're trading confidentiality and integrity for availability.

Co-Tenant Threats

These threats are to tenants of "infrastructure as a service" (IaaS) and to a lesser degree "platform as a service" (PaaS), whereby the provider/tenant trust boundary allows tenants to execute arbitrary code inside the data center. In IaaS, the code execution privileges are effectively unlimited, although they may formally be limited to what a non-administrative account can do. In PaaS, the code that is supposed to execute is far more limited. Historically, few platforms were constructed with a threat model that the most important attacker is already on the system. It is far more common, and probably even appropriate, for platform designers to worry about the trust boundary between the system and those who cannot execute any code. Unless the platform software has been carefully constructed to resist elevation of privilege attacks and to prioritize finding and fixing those, there are likely vulnerabilities that allow an attacker to violate the rules. That leads to second-order threats to the cloud tenants.

There is also a set of threats from other tenants, ranging from the trivial to the movie plot. At the trivial end, you may be behind the same single firewall as your competitor, or someone without an IT department. You might also be on the same domain as they are, such as cloudapp.net or s3.amazonaws.com. Some defenses, such as firewalls, need to be managed as part of the cloud deployment. Your systems may also come under attack as stepping-stones to other tenants of the cloud provider. Beyond that, another tenant might try to bust out of their virtual machine and take over a host, giving them access to your machine as well (if you're sharing machines). An attacker might also be able to access the network or storage (either local cache or storage specific to the cloud). Another tenant might be taken over to run a DoS attack against you, or an attacker might sign up to do so.

Threats to Compliance

There are three typical issues here. First, for many compliance regimes—but most notably PCI and HIPAA—the entire stack, including physical security, needs to be compliant. Therefore, the only way to have a PCI assessed app is for your cloud provider to be assessed PCI compliant. Second, there can be issues with auditing and logging. Not all cloud operators will provide logs of access to their APIs or web consoles. This can lead to technical issues, such as it may be impossible to see who added or changed accounts. That technical issue can lead to a compliance issue. The final issue is when you get into the PaaS and SaaS market, you often lose the ability to leverage cryptography at the filesystem or database level, as well as any concept of end-to-end encryption.

Legal Threats

The primary new legal threat when you move data to the cloud relates to laws that (at least in the U.S.) substantially reduce your ability to know about or challenge legal requests for your information, such as subpoenas or warrants. Data you store on your own systems is often more legally protected than data you store on someone else's systems. The legal demands served on your cloud provider may also contain provisions forbidding them from telling you about those demands, or you may be informed after the data has already been provided.

Of course, there's also the need to negotiate agreements related to privacy, security, and reliability. Contractual provisions for both privacy and security need to cover your business needs and your compliance needs. Privacy is covered by a hodge-podge of U.S. regulations. In the European Union, and those places with a safe harbor agreement for data from European countries, there's a set of requirements, most importantly for the organization holding the data to name a data custodian. That custodian has certain responsibilities, and you'll need to address those if you're moving data collected under EU or other similar privacy regimes.

It is, of course, important to consider these issues with your attorney; this section is intended only to outline some of the points you should discuss with them. Your attorney may have additional concerns.

Threats to Forensic Response

After an intrusion, your VM may be shut down by the cloud provider. You may have instantiated a system large enough that performing complete snapshot or a memory dump is time consuming; but most important, you may not have a defensible chain of custody.

Miscellaneous Threats

Some cloud providers offer easy to use virtual machine images, uploaded by kind strangers for your convenience, and out of the goodness of their hearts.

Sometimes, the people who uploaded them really did so out of the goodness of their hearts. Other times, the images are not so safe, and trusting them for anything serious is probably foolish.

Cloud Provider Threats

There is also a set of threats to the cloud provider that cloud providers must consider, and cloud customers should consider. These are all threats from or caused by the folks to whom you give access to your systems. These are split into threats caused by malicious behavior by the tenant that targets the provider, such as attempts by the tenant to hack the provider; and threats caused by tenant behavior, such as blacklisting.

Threats Directly from Tenants

The largest threat is that a tenant will find a way to break out of whatever sandbox you put them in, and be able to take actions you don't want them taking. Those actions might be on the order of running code on the raw hardware and tampering with either your billing or your other customers, or it might be connecting to networks that should be firewalled off. In particular, US-CERT has warned about threats to the IPMI (Intelligent Platform Management Interface) (US-CERT, 2013). These threats are sometimes managed by putting IPMI on an isolated network. If a tenant breaks out, they may be able to access such a management network. The sandbox escape threats are more likely when clients are held back by fewer security boundaries. Thus, a client in a Software as a Service (SaaS) environment has more barriers than one trying to escape from an Infrastructure as a Service offering.

There's also a fraud (repudiation) threat, which is that a new tenant might sign up with someone else's personal data and/or credit card, preparatory to committing fraud, running a botnet, or DDoSing a game server. These threats can be partially addressed by charging the card immediately for the first portion of service, or charging a sign-up fee, although that may be inhibitory to new business. If there are throttles for e-mail sent or similar things, it may be useful to tie them to length of tenancy.

Unfortunately, many of the ways to address these threats are at odds with the low-friction, get-started-quickly value proposition that cloud providers want to offer. There may be interesting ways to address these trade-offs that have yet to be invented, but for now appropriate monitoring for anomalies is important.

Such monitoring has to build on the unique elements of the business, so custom code will be needed.

Threats Caused by Tenant Behavior

There's a set of threats that tenants can cause, including spamming and piracy. These problems are magnified in those cases where accounts are free to anyone, and essentially anonymous. These threats are less clear-cut than issues such as spoofing, tampering, or information disclosure. If your requirements are clear, there's no question about whether the threat violates them. In contrast, perhaps the e-mails are all going to an authorized list? Perhaps the person who uploaded that song is the artist, or is authorized to do so? Or perhaps the use of an image is permissible under the law? This lack of clarity makes these threats harder to manage than the classical security violations.

The United States has a somewhat clear set of rules regarding *notice and take down*. These give cloud providers a legal defense against copyright claims until they receive a formalized notice of infringement, after which they must take certain actions, generally taking down the content. Following those processes is prudent, as jumping to judgment and action on a notice may threaten the protections you otherwise have.

These threats lead to an indirect threat to the provider and other tenants, which is backlash from the attacks. That backlash can include visits or calls from law enforcement, reports from other parties that require investigation or response, and blacklisting. Once a tenant has sent spam, been part of a botnet, and so on, there's a risk that the IP, subnet, or ASN may be blacklisted. The behaviors that lead to blacklisting may be violations of the terms of service. Responding to the behavior and getting IP addresses de-listed is likely a manual and time-consuming task.

Mobile Threats

Threats to mobile devices are generally similar to threats to other computers. For example, someone who can run code on the device may read your files, use your authentication data, etc. There are a few additional threats for mobile devices, including increased likelihood of device loss, difficulty managing the devices, and business models conflicting with security updates.

The threat of device loss is clearly higher for mobile devices than for servers. The two main ways to address device loss are device wipe and data wipe. Device wipe can cause conflict, as many devices are owned by your employees or contractors, so in wiping their device, you may be deleting data that is not yours to delete. Data wipe can be accomplished by sending encrypted data to the device, and only sending the key needed to access the data when the data is accessed.

Many mobile devices are locked to specific software providers, and constrain which software can be loaded onto the device. This may threaten the ability of a compliance team to load them up with compliance-ware. When a mobile device is locked, then you define the person holding the device as a threat. The device and its physical shell become a trust boundary that you must carefully consider, as all communication is subject to tampering, denial of service, and information disclosure.

Lastly, many wireless carriers threaten the ability of the software makers to deliver patches to devices in the field, claiming that arbitrary patching threatens them with bricked devices and support costs. This means that fielded mobile devices can be vulnerable to security problems that are fixed in newer versions of software.

Summary

The web is comprised of software which can be threat modeled like any other software. There are a variety of recurring threat classes that are best managed by using safer languages and test frameworks that focus on those classes. These threats are things like XSS and SQL injection. There are also recurring patterns within the web and cloud that you should consider when threat modeling.

The browser and its plugins have threat models that should be documented by the browser maker. Those threat models should cover both security and privacy, and you need to understand them to program the browser.

Cloud tenants come under threat from insiders at the cloud provider and from co-tenants. Limits on code execution make attacking your co-tenants harder, so as you move up the stack from IaaS to PaaS to SaaS, co-tenant attacks become less likely. There are also a variety of threats to compliance and legal threats that cloud tenants must account for. In addition, forensic response may be threatened if you are a cloud tenant.

Cloud providers have to worry about threats of their tenants attacking them, and the side effects that can occur when their tenants attack others. In particular, if tenants can access management networks that are normally isolated, the impact on security can be exceptionally large.

Mobile computers, including laptops, tablets, and phones are much like other computers, except the frequency of device loss is far higher. Some mobile devices are locked in ways that restrict the loading of software, making it hard to add compliance-ware, and possibly changing the threat model to include the person who uses the device. Such locking may also threaten patching, leaving devices vulnerable to known issues, and effectively removing the "need to find a vulnerability" barrier to attack.