About Data Flow Diagrams (DFDs)

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.

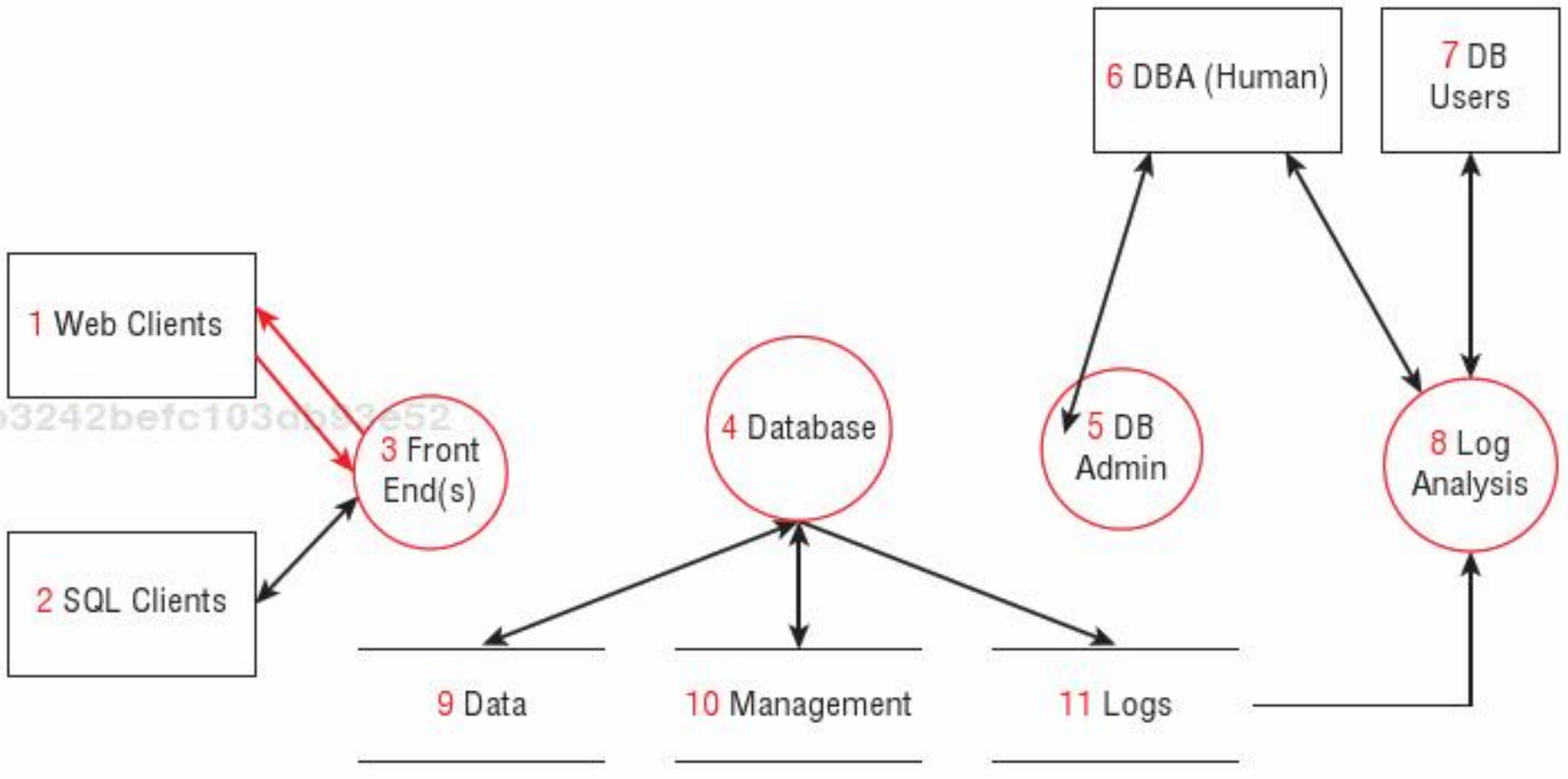
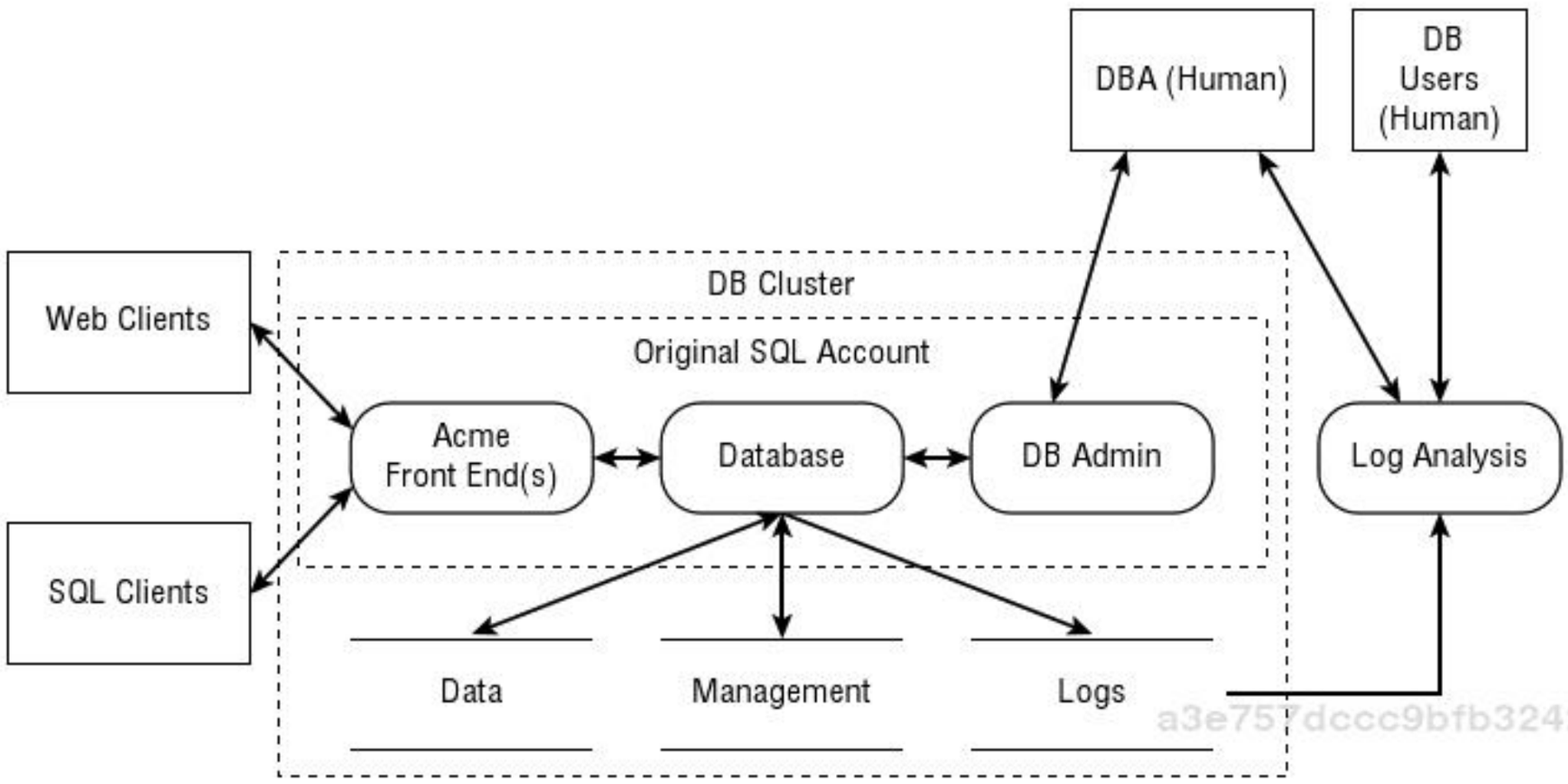


Figure 2-3: A classic DFD model



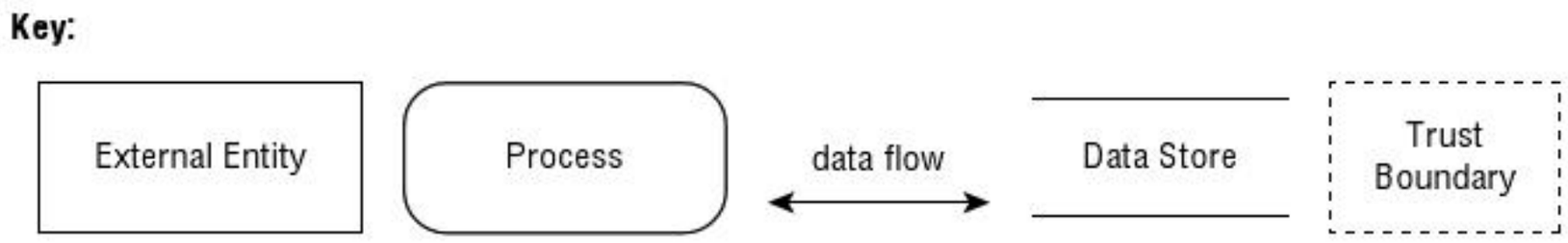


Figure 2-4: A modern DFD model (previously shown as Figure 2-1)

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."

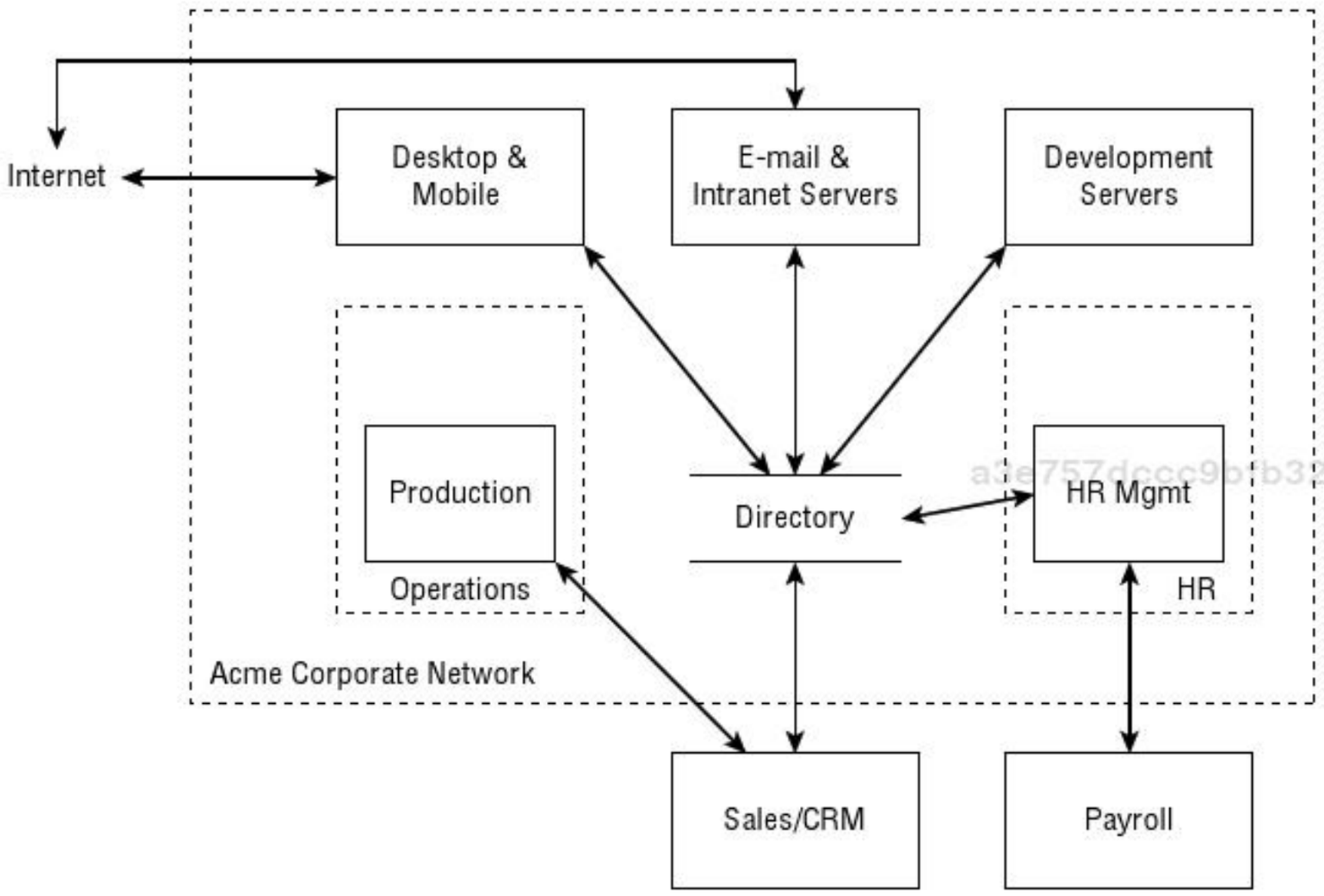


Figure 2-5: An operational network model

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.

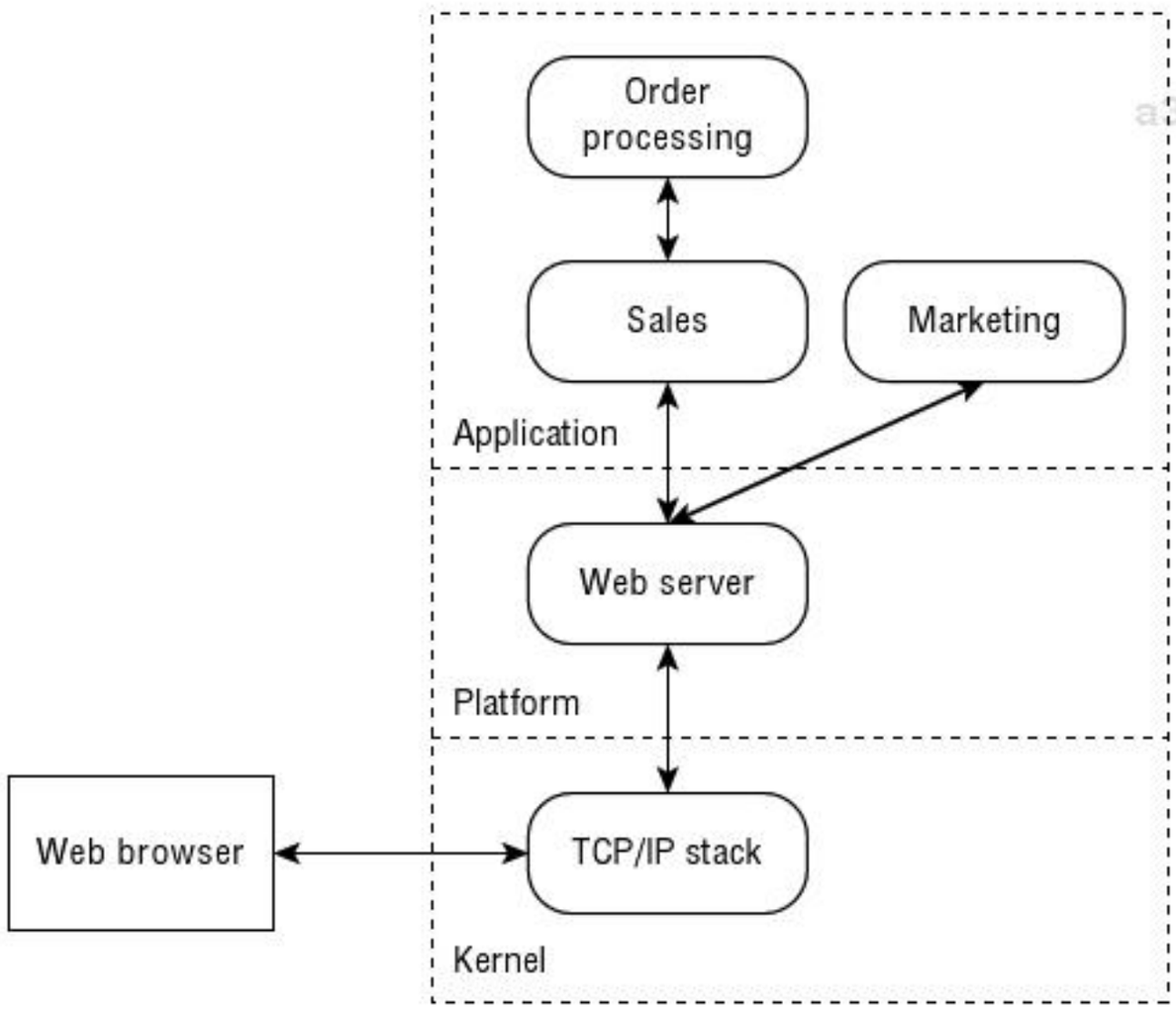


Figure 2-8: Trust boundaries in a web server

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.