Demystifying the Threat-Modeling Process

Peter Torr
Microsoft

In today's hostile online environment, software must be designed to withstand malicious attacks of all kinds. Unfortunately, even security-conscious products can fall prey when designers fail to understand the threats their software faces or the ways in which adversaries might try to attack it.

To better understand a product's threat environment and defend against potential attacks, Microsoft uses *threat modeling*, which should be treated like any other part of the design and specification process. In fact, singling it out as a special activity performed outside the normal design process actually detracts from its importance to the overall development life cycle. We must consider security needs throughout the design process, just as we do with performance, usability, localizability, serviceability, or any other facet.

Starting this process early in the development life cycle is important because it can reveal architectural weaknesses that might require significant changes to the product. Making design changes early in the product life cycle is much cheaper than trying to make them at the end.

Scoping the process

Threat modeling an entire product is typically too complex, whereas doing so for individual features is too simplistic. Instead, we focus on logical groups of functionality termed *components*. There are two basic approaches to determining the individual components to threat model in a product or system. You can start with the entire product, enumerate all its dependencies and entry points (as outlined below), and determine whether you have a manageable number of these (a dozen or fewer). If not, you can split the component into several smaller (logical) pieces and try again. The second approach is to perform the same high-level analysis on each individual feature and combine logically related features until you arrive at the same manageable degree of complexity. These are just rules of thumb, and experience will show the best approach for your products.

Gathering background information

Threat modeling begins with gathering various types of information from the component's specifications and compiling it into a *threat-model document*. You will use this living document both to describe the component's security-relevant design points and record the findings of the threat-modeling process.

Use scenarios

Use scenarios are high-level descriptions of how a component will be deployed and used. Avoid descriptions intended to justify the feature or market it to users; you simply need to list how the product will be installed and used. Consider the following example:

- This is a client application installed on Windows XP Service Pack 2 (and later).
- The application is designed for use by corporations inside firewalls (not across the Internet).
- It exposes a COM automation interface.
- It can call Web Service APIs using Windows Integrated Authentication.
- It can be run by low-privileged users, but some features require administrator privileges.

You might also choose to list "anti-scenarios"—configurations or usage patterns that are explicitly unsupported or known to be insecure.

Dependencies

Dependencies are other features or technologies on which your component relies to perform its functions. It's important to document these because you will be making assumptions about them, which you'll need to verify later. An example set of dependencies might include

- the Microsoft XML Parser Library v3.0,
- any HTTP 1.1-compliant Web server, and
- an authentication subsystem delivered by Bob's team.

You can typically exclude generic operating-system dependencies such as file-system or memory-management APIs, although you should note richer subsystems, such as HTML renderers or email servers.

Implementation assumptions

It's important to enumerate the assumptions you make about the component during design and development because you must verify them later. They can also help scope the threat-modeling process. Example assumptions might include

- the component will be called only on a single thread;
- all requests have been authenticated by the Web server before reaching our component; and
- users won't attempt to download files that are larger than 5 Gbytes.

Note that the assumptions you make might be invalid or unreasonable, which is why you must enumerate and validate them.

Internal security notes

Internal security notes provide additional information to those reading the threat model who might not be intimately familiar with the technologies or acronyms used therein. For example, does AES stand for "Advanced Encryption Standard" or "Attachment Execution Services"? This is also a good place to provide information, such as deployment configurations or descriptions of specific flows, about the data-flow diagrams (described later).

External security notes

A threat model should also contain a section for external security notes, but you can typically fill this in toward the end of the process. I list it here for completeness.

Describing the component

In addition to the background information, the threat-model document should include a security-focused description of the component's design. Details about how features are implemented or internal code structure are irrelevant; we're interested only in how the component behaves with respect to its inputs and outputs.

Entry points

Entry points represent interfaces with other software, hardware, and users. Any part of your component that sends or receives data to or from a file or external entity is an entry point. For example, these might include open network connections listening on ports, data files opened from the user's home directory, or an interactive user.

If your component exposes an API, you should break the API into a set of logically related operations to model as entry points rather than modeling individual methods or parameters. If your API supports a partial-trust environment (such as the Microsoft Common Language Runtime), you can also segment it according to the permissions needed to access it.

Trust levels

Once you've established your component's entry points, tag each with a trust level that represents the degree to which the entry point can be trusted to send or receive data. In many cases, there are only three trust levels:

- *administrator*—local administrators who have complete control over the system;
- *user*—interactive users and their settings (assumed not to have administrative rights); and
- *untrusted*—external data files, network connections, or other potentially malicious input.

More complicated systems might have additional trust levels, although they might not have a defined hierarchy like the three main trust levels do.

Protected assets

Protected assets are the resources with which your component interacts and which must be kept safe from harm—the things an attacker will be looking to steal, modify, or disrupt. Common examples of protected assets include the system's computing resources (such as CPU, memory, or storage), core parts of the system (encryption keys, for example), and users' or customers' personal information, documents, emails, and so on. You should make a list of the trust levels required to gain access to each asset.

Data-flow diagrams

The threat-model document's heart, and the most useful tool for generating threats against the component, is the data-flow diagram. A DFD is a graphical representation of the component, showing all the inputs and outputs as well as all logical internal processes.

Building good DFDs is both an art and a science, and guidance on doing so could fill an article on its own. In the interest of brevity, let's just look at an example DFD and point out its most salient properties.

Figure 1 shows a sample DFD for safely opening email attachments. The numbered arrows describe the data flow through the system, and should be self-explanatory. Additionally, you'll note the use of different colors and shapes for the objects

If your component exposes an API, break the API into a set of logically related operations rather than modeling individual methods or parameters.

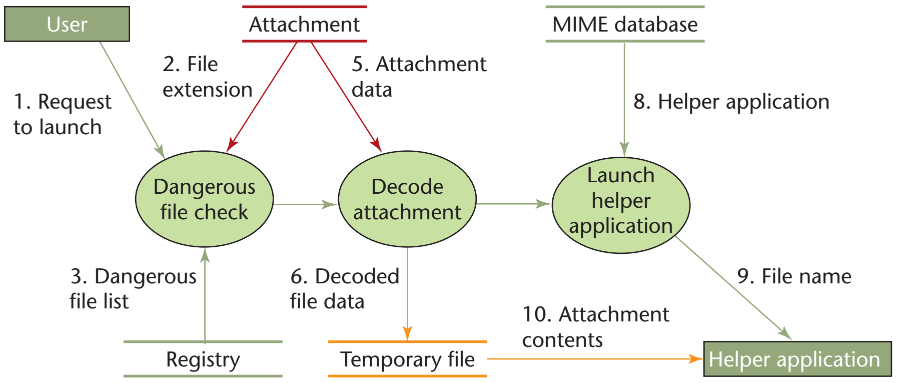

Figure 1. Sample data-flow diagram (DFD) for opening email attachments. Green items represent trusted objects; red items are untrusted; and orange items are "somewhat" trusted. Circles represent logical processes within the component being modeled; rectangles represent entities that you have limited (or no) control over; double horizontal lines represent passive data stores; and arrows represent data flows.

on the diagram, which helps increase the diagram's information density. The three colors map directly to trust levels—green items represent trusted objects; red items are untrusted; and orange items are "somewhat" trusted. You don't usually need to separate the administrator and user trust levels at this level of granularity (they can both be green), but you can add more colors if you require additional trust levels.

The different shapes used in the diagram also convey information: circles represent *logical processes* within the modeled component; rectangles represent *external entities* that you have limited (or no) control over; and double horizontal lines represent *passive data stores*.

Processes represent the logical steps the component takes to perform its task; each should be easily described in a single (short) sentence. They don't necessarily map directly to objects or methods in your code, and you shouldn't draw an inheritance or call graph as your DFD. Processes are always trusted (they are your code), and they must both send and receive data. You can break complex processes down into more detailed subdiagrams to avoid "spaghetti" diagrams.

External entities represent entry points or dependences over which you don't have direct control. This includes libraries, other programs, remote machines, devices, and people. Every external entity must match up to one or more entry points or dependencies, as enumerated earlier. They can have any trust level and can send or receive data (or both).

Data stores represent data at rest and typically include items such as files, registry keys, or in-memory structures. Like external entities, data stores must match up to one or more entry points, can be of any trust level, and can send or receive data (or both).

Data flows represent how information moves from one object to another. They are always unidirectional; if similar data flows back and forth between two objects on the diagram, you need two separate flows. Data flows can be trusted or untrusted, and they're always labeled with logical sequence numbers and a concise description of the information being sent. Avoid labeling flows "read" or "write"; instead, they must contain the data being read or written. If numerous discreet data items are represented by a single flow (for example, username, password, server name, and authentication scheme), you might want to consolidate them into a single label (such as "log-on information") and use the internal security notes to enumerate what constitutes that information.

What if I find a problem?

The simple act of diagramming your component will sometimes make certain security or logic flaws apparent. This is a great side-effect of methodically performing the threat-modeling process, but you must avoid the temptation to fix the design on-the-fly and build a diagram that represents the "correct" implementation. This stage of the process is about documenting how the system currently works; if you document how you think the system should work, you'll end up missing threats later on in the process.

Nevertheless, it's important to record weaknesses as you identify them. However, including them in the threat-model document at this point makes it harder to focus your team in the next phase. Instead, I recommend recording the threats separately and bringing them up in the brainstorming meeting later on. You should use some form of reminder such as an Outlook task or a "work item" bug assigned to yourself in case you forget to bring the threat up at the meeting.

Obtaining threats

Once you've gathered all the necessary documentation about the component, you're ready to schedule a brainstorming meeting to enumerate all potential threats. This meeting should include all the product-team members responsible for the component (design, development, and test) and any subject-matter experts (SMEs) for the dependencies your component consumes. If possible, you should also include a security expert or a seasoned threat modeler from another team to help drive the process. I suggest distributing the threat-model document at least two days in advance so that participants are familiar with the background information and design points before the meeting starts.

To ensure that everyone is focused on the task at hand, the meeting must have a clear purpose. It isn't a "threat model review," which would imply that you were examining a completed document. Rather, this is where everyone provides creative input into the process. Set aside a small amount of time at the beginning for an overview of the threat-model document and to cover any high-level concerns. Then, dedicate the majority of the meeting to analyzing the DFD to obtain threats.

To help everyone remember the types of threats to which your component might be exposed, consider writing the acronym STRIDE on the whiteboard:

- Spoofing—attackers pretend to be someone (or something) else.
- Tampering—attackers change data in transit or at rest.
- Repudiation—attackers perform actions that can't be traced back to them.
- Information disclosure—attackers steal data in transit or at rest.
- Denial of service—attackers interrupt a system's legitimate operation.
- Elevation of privilege—attackers perform actions they aren't authorized to perform.

The technique for brainstorming threats is actually fairly straightforward. Starting with the first object on the DFD (the object from which data flow 1 originates), consider each of the STRIDE categories and how they might apply to the object and accompanying data flow (a detailed description of STRIDE is out of scope for this article). When thinking about threats, try to express them in terms of "entity X performs action Y, resulting in negative outcome Z." Although not always possible, phrasing your threats in this way makes it much easier to investigate and mitigate them later on. Remember that threats always exist—bad guys are always trying to do bad things—so it doesn't matter if the system already protects against a given threat; write it down anyway.

For the example DFD in Figure 1, we would first look at the first data flow, labeled `request to launch`, and the entity from which it comes (user). We might come up with the following (incomplete set of) threats:

- An attacker programmatically sends the `request to launch` message, resulting in an unwanted attachment execution (spoofing).
- An attacker instruments the attachment with a "Web beacon" such that merely opening it sends the user's email address and various other information back to the attacker (information disclosure).
- An attacker maliciously crafts the attachment metadata in the email message to exploit coding errors in the parsing logic, leading to arbitrary code execution (elevation of privilege).

At this stage of the process, don't get bogged down in implementation details or argue about threats. Don't allow excuses such as "that's a silly threat," "we already check for that in the code," "nobody will ever try that," or "it's too hard to do that." The threat-generation stage is where threats are documented, not argued about. Deciding whether the threats are (or should be) mitigated comes later. Nobody should hold back their ideas for fear of getting shot down by another participant. Write down every threat that's identified, and include enough detail in the description to allow developers or testers to look at it a week later and understand what they need to do to implement or test a mitigation to the threat.

Being systematic and investigating every data flow through the system is important in the brainstorming process, but you also need to let your creativity flow. Are there entry points in the component that aren't represented in the diagram? Can an attacker force new data flows or bypass existing ones by calling a different API or taking actions in an unexpected order? For example, how does the component choose the name and location of the temporary file it writes to disk in flow 6 in Figure 1? Is this file name under the attackers' control, and if so, could they maliciously form and use it to harm the user? Another potential loophole to investigate is whether the attachment filename could be crafted such that the extension appears to be safe (such as .jpg) in data flow 2, but ends up being written as a "dangerous" file type (such as a .exe) as the file is decoded and saved to disk.

Resolving threats

Once you've exhausted all the data flows in the DFD and can think of no other creative ways to attack the system, revisit each threat and decide what action to take to resolve it. Typically, this happens at a follow-up meeting at which the key stakeholders agree on the resolutions and assign owners for follow-up work. Each threat will fall into one of the following broad classes:

- already mitigated by the component;
- already mitigated by a dependency or other component;
- not mitigated and requires mitigation;
- not mitigated, but a dependency or other component is responsible for it; or
- not mitigated, but doesn't require mitigation.

Each class of threat has a different process for resolution, but you must track them all down and follow through to closure, including updating the threat-model document with the chosen resolution.

If you believe the threat is mitigated, a tester generates a set of attack scenarios to verify that it's in place and working as expected. These tests must become part of your standard test suite; they're very important for checking regressions. Additionally, if developers are at all uncertain about the mitigation's quality or scope, they should also schedule a security-focused code review of the area to ensure that it's behaving as expected.

If you believe a threat is mitigated by a dependency or some other system component, you must add this assumption to your threat model and then verify it either by directly communicating with the owners, reading the component's security documentation, or manually verifying.

If the threat isn't currently mitigated but is serious enough to warrant it, an architect or developer must design and implement a suitable solution. Any nontrivial mitigation should go through its own design review and threat-model review. Once you've implemented the mitigation, you must test it by following the process detailed earlier.

If you believe the threat isn't mitigated but is out of scope for your component, document it and communicate it to the dependency or other entity that you believe is best equipped to deal with the threat. This might include documenting the threats for your system's clients, especially if you're building a generic platform and exposing an API.

Finally, you might have identified a threat that you don't believe is serious enough to warrant mitigation—as when the threat requires unrealistic preconditions or results in no material gain for the attacker. Nevertheless, capturing the threat in the threat-model document is important for alerting future team members that it exists. Today's "unrealistic preconditions" might be tomorrow's common practice.

Following up

Just two tasks remain in the threat-modeling process after you've generated the threats and determined their resolutions.

The first is to be vigilant about tracking down your dependencies, verifying your assumptions, and communicating your security requirements to others. This effort can take some time to complete, but it's vital to the security of the larger product or system. Ensuring that every component inside a product is secure is a great start, but it's insufficient for building secure systems if the interfaces between those components (and the assumptions they make about each other) are insecure.

The second task is to be vigilant about tracking any changes that are made to your design (and those of your dependencies) through the rest of the product development cycle and to ensure that the changes don't invalidate your assumptions or inadvertently introduce holes in your security mitigations. All nontrivial code changes require quick reviews of the threat model to ensure that no existing assumptions or mitigations are being invalidated. At the end of each major milestone (or before shipping) you should also undertake a complete review of the threat model to ensure that it still accurately reflects the component's design.

When performed as a structured process, threat modeling is an efficient and effective means of identifying and mitigating risks to your software components. Being well-prepared for the threat brainstorming meeting is a relatively simple undertaking, yet it's crucial to holding a successful session. The brainstorming session itself should follow a methodical approach to identifying threats, while still letting participants think about the problem creatively. Finally, following up with external dependencies and verifying assumptions is vital, and revisiting the threat model is necessary any time design changes are made to the component after the fact. □

Acknowledgments

The threat-modeling process has evolved over the years at Microsoft with the input of many fine people. Having participated in numerous threat-model reviews over the past few years, I've learned a lot of real-life lessons for improving its efficiency and effectiveness (which I have attempted to share with you in this article), but I couldn't have done it without the prior work of many others.

***Peter Torr** is a security program manager at Microsoft. He studied computer science and philosophy at the University of Melbourne and now specializes in Internet client security and refining the art of threat modeling as part of the Secure Windows Initiative team. Contact him via his blog at http://blogs.msdn.com/ptorr/.*