Chapter 24. Threat Modeling Applications

Threat modeling is a valuable method of improving your application's security posture—if implemented correctly. With an improper or hasty implementation, threat modeling exercises do nothing but check boxes and waste valuable time.

When properly implemented, threat modeling allows software and security engineers to work together in a cohesive manner. The shared engineering and security workflow resulting from a well-implemented threat model allows companies to identify threats (e.g., potential vulnerabilities), threat actors (e.g., potential hackers), existing mitigations (e.g., security controls), and the delta between threats and mitigations.

A well-implemented threat modeling process, in addition to the preceding benefits, serves as a living document for rapidly and effectively handing off security knowledge from one engineer to another. It implements robustness in your organization's security knowledge in a way not possible through most other methods.

While you don't want to make use of a hastily implemented threat modeling process, learning how to design and implement a robust threat modeling workflow will dramatically improve your organization's security posture in the long run.

Designing an Effective Threat Model

A proper threat model should be able to accomplish a number of goals. If all of these goals aren't accomplished, the impact of the threat model is limited:

- Document knowledge
- Identify threat actors
- Identify risks (attack vectors)
- Identify mitigations
- Identify delta (between risks and mitigation)

Each of these major goals builds on top of the others. For example, a threat model as a living *forwards knowledge repository* is most valuable when it contains not only data selected regarding the architecture of the application but also its threat actors, potential threats, etc.

Likewise, a threat model that only contains risk identification and does not document mitigations is less than desirable because risks are not actionable. Despite being valuable in some capacity, *risks are not actionable until existing mitigations are known*. Implementing a mitigation for a risk that already has mitigations is a recipe for introducing more bugs, technical debt, and even potentially adding more vulnerabilities to your codebase.

Threat Modeling by Example

Imagine a fictional corporation, MegaBank, is implementing a new feature with which users can review existing features on MegaBank. This will help MegaBank in determining which features have been well received as well as promote those features to other users.

Knowing this feature will be public facing, and likely to be abused if not implemented properly, MegaBank asks its application security team to work with its developers to develop a threat model. Holly Hacker, a senior member of the MegaBank security team, sees the value in this initiative, and knowing the goals of a threat model, she begins working with the development team to collect data to produce said threat model.

Logic Design

The first data that Holly collects is simple: she requests a description of the new feature from a logic design perspective. In the case of a software

product, *logic design* refers to a description of the feature or product from a *functionality perspective*. This perspective is higher level than an engineering architecture or design document, but lower level than a marketing or sales description. Typically the level of detail for this description is the same as would be used by a UX designer. A UX designer must understand the application logic prior to being able to develop an effective UI.

The description that Holly comes up with is as follows:

> *The MegaBank "user review" feature is a public-facing feature in which an authenticated user will be able to fill out a web form and post a review regarding any existing MegaBank features. The review form will contain a score between 0 and 5, as well as a text-based review body. When a review is submitted, a MegaBank server will store the review content and score in a database, which can later be queried by users that click "show reviews" on any "activate feature" page. Examples of these pages include the "activate cryptocurrency wallet" or "activate bank loans" features. The design of MegaBank's current user experience is that optional features are "off" by default, so this "user review" functionality will help promote the enablement of new features if the functionality is successful.*
>
> —Logic Design, MegaBank User Reviews Threat Model

The logic design overview is important to a threat model, but it's one of the most often disregarded components. It is important because it can later be used to spot *logic vulnerabilities*, which are vulnerabilities derived from the application not following the application logic for which it is intended to function. These types of vulnerabilities are different from traditional application-level vulnerabilities, as they are unique to each application and its specific business requirements.

An example of logic vulnerability in this case would be a user being able to post a score with a value greater than 5 by bypassing the web form and sending a custom HTTP request. Simply storing an integer value in a database by itself is not a vulnerability. However, in the provided edge case, it is indeed a vulnerability due to the unique application logic of this fea-

ture. This is because any sort of aggregation or average would be skewed by an out-of-range number, misleading users and causing them to lose trust in the authenticity of MegaBank user reviews. This would be damaging to MegaBank's brand reputation.

Technical Design

Technical design documentation and collection is a much more common step in the threat modeling process than logic design documentation and collection. Unfortunately, while technical designs are effective for finding vulnerabilities and describing the implementation details of a product or feature, they often don't explain the business goals and logic. This is why the aforementioned logic design data collection is required.

That being said, technical design data collection and documentation has some particularities required in order to be effective, so it's not as easy as simply grabbing some data flow diagrams and throwing them in a document.

The goal of collecting technical design documents and aggregating them into a threat model is to allow engineers and researchers to later identify threats that are generated as part of the application architecture.

Technical design should be comprehensive enough to identify the following key variables in an application:

- What tools and technologies are used to implement this feature?
    - Programming languages
    - Databases
    - Build tools
    - Package managers
    - Frameworks
- What third-party services are used in the implementation of this feature?
    - In-network third-party services (e.g., AWS subresources like S3)
    - Out-of-network third-party services (e.g., Datadog, CircleCI)

- In which ways does data flow from one module to the next (data flow diagrams [DFD] are beneficial here)?
    - Data formats in transit
    - Encryption in transit
    - Network protocols
- In which ways are the networks configured for this feature (internal/VPN, external, etc.)?
    - Firewalls
    - VPNs
    - Physical networks
- What authentication and authorization controls are used in this feature?
    - Are multiple types of authentication tokens permitted?
    - Do all API endpoints perform the same type of authorization checks?
- What is the database schema? (Is any of the data regulated or controlled?)
    - NoSQL databases that are schemaless should list all potential data shapes.

In the case of MegaBank's user review feature, our fictional security engineer came up with the following (shortened) technical design:

First, our existing React user interface will have two new components added. The first component is `getReviews` *and the second is* `createReview`*. Both of these components will have an associated database entry and REST API endpoints.*

The `getReviews` *component will fire off an HTTP GET against our AWS EC2-hosted HTTP REST server, which will then query the PostgreSQL database looking for reviews associated with the feature noted in the query param* `featureID`*. The JSON associated with the result of the query will be returned unless no reviews are found, in which case an error will be returned with the text "no reviews found for chosen feature."*

The `createReview` *component will fire off an HTTP post with the user-submitted form information, which will first verify the user is logged in by checking their session cookie against the existing session management service. If this check succeeds, the associated AWS EC2 REST endpoint will validate the type of the score is integer and the type of the review is text and then store it in the PostgreSQL database.*

The programming language used will be JavaScript on both the client and the HTTP REST server. The HTTP REST server will be running ExpressJS and making use of the open source `body-parser` *and* `cookie-parser` *modules via the npm package manager. All languages, frameworks, and dependencies are running on the latest stable version. The hardware on which this HTTP server will reside will be an AWS EC2 instance inside of the company AWS cloud.*

Data sent over the network will be encrypted via TLS 1.3 using a certificate generated by Let's Encrypt.

All APIs are also optionally queryable via GraphQL.

—Technical Design, MegaBank User Reviews Threat Model

At this level of granularity, a security engineer can begin to evaluate the technical design against the logic design in a way that will enable them to start finding both traditional web application vulnerabilities and logic vulnerabilities.

In this case, the method by which the text is rendered is not noted, meaning this could potentially become an XSS attack vector. The logic by which the SQL query is generated is also not noted, which is dangerous given the fact that a user-submitted string is being stored. Beyond that, there may be information disclosure in the case of an error message stating no reviews are found for the chosen feature. If a feature is not yet publicly available, iterating through this API could disclose its existence.

Typically, it's best to wait until all of the relevant data for the threat model has been collected before brainstorming potential vulnerabilities, but with proper data collection, these sorts of vulnerabilities will become visible rapidly. It's good to take note of them along the way.

Threat Identification (Threat Actors)

With logic design and technical design out of the way, it's time for Holly Hacker to start considering threat actors. Threat actors are archetypical attackers—each of whom have unique routes for attacking a system (in this case, the MegaBank user reviews feature).

When considering threat actors, we should look at the previous design documents and start brainstorming what tiers of permission each type of user may have access to. These aren't just programmatic authorization tiers, but also insider permissions that occur naturally in any organization. For example, consider the potential attackers for this hypothetical application and permissions shown in Table 24-1.

Table 24-1. Potential threat actors

Threat actor	Powers and permissions, risk
User admin	Can use admin UI to read/update database. May use powers to steal PII. Could modify existing ratings. Currently, usage of the admin tool is not properly logged. No accountability.
Customer support user	Can read database data but cannot update fields. Could steal PII.
Review aggregator script	Periodically runs in order to average the review scores across multiple reviews. Has direct database access as database admin, cannot access other utilities or execute Bash. If compromised, could run any query against database. Significant damage potential.
Authenticated user	Can post reviews and review scores. Can read public reviews by other users. Could attempt to POST malicious payloads such as SQL injection or overflow reviews via bypassing web form and directly creating HTTP POST.
Unauthenticated (guest) user	Can read review scores. No write or update access. Low threat as long as DoS or circular graph queries are accounted for.

As you can see, after compiling a list of all potential threat actors, we reach a few unexpected conclusions. First, the threat actors list should be a comprehensive list of application users—not only the humans, but the machine-powered users as well. In this case, a Linux user "review aggregator script" is added to the threat actors list because it runs autonomously and has privileged access to the SQL databases that power this new feature.

It's a common mistake to only consider direct human attacks, and an even worse mistake to only consider external human users. When compiling your list of threat actors, ensure you include internal, external, and machine users. Furthermore, each of these should be split into authenticated and unauthenticated permissions sets where possible. For example,

it's likely your application handles unauthenticated public user permissions differently than authenticated public users. As a result, each of these should be individually considered during your threat modeling exercises.

Note the following obvious but important detail: each of these threat actors has a varying attack surface area, aka the types and methods of attack they could take advantage of. While this may seem like an obvious detail, it's important to keep in mind throughout not only threat modeling but development as well because it can assist you and your organization to appropriately rank risks in such a way that you identify and resolve the most significant attacks from the most dangerous threat actors prior to dealing with the less dangerous ones.

Threat Identification (Attack Vectors)

With knowledge of the potential threat actors documented, it's now time to consider the threats that are poised against your application. We can categorize these threats as attack vectors—they are not yet vulnerabilities since the system hasn't been developed. Instead, they are potential (future) pathways by which a threat actor could attack your system.

Typically, the best way to brainstorm attack vectors as part of a threat modeling exercise is through cross-analysis of three aforementioned components: logic design, technical design, and threat actors.

Logic design exposes components of the business logic that are relevant when looking at the technical design. For example, permissions systems may be defined in the technical design in terms of implementation, but without the logic design, it is difficult to determine if a permissions system is implemented in a way that could be exploited. Edge cases in these types of systems are very common in production applications.

Next, we want to consider the threat actors when evaluating attack vectors. Not all attack vectors will be feasible for all threat actors to attack. Some attack vectors will require privileged access of sorts; for example, internal VPN access or credentials to log in to an admin-only server appli-

cation. By constantly considering the relationships between these categories, we can identify who would and would not be able to take advantage of an attack vector.

As such, our table of attack vectors incorporates knowledge from all of the prior headings included in the threat model plus a risk ranking column (Severity) that will help us prioritize timing and resources for fixes. Our final table would appear something akin to the information summarized in Table 24-2.

Table 24-2. Attack vectors

Threat name	Severity[a]	Threat actor	Description
Improper validation—score	P1	All users except guest user	Due to lack of defined validations, the POST to create a new review can include a score that is out of the bounds of 0 and 5, leading to skewed ratings after aggregation.
Information disclosure— `FeatureID`	P3	All users	The error messages provided on the `getReviews` endpoint allow a user to scan for potentially unreleased features or features they should not be aware exist (gated).
SQL injection	P0	All users except guest user	Improper implementation on POST payload could lead to SQL injection against `postReview` endpoint.
GraphQL circular and large queries	P1	All users	Circular and large queries become possible that are not possible with traditional REST. These are not prevented by rate limits and require max query times to prevent.
GraphQL introspection and errors	P1	All users	GraphQL's introspection engine should be disabled or we will leak server configuration details. GraphQL internal errors also leak server information if not suppressed and replaced with custom errors.
High privilege user attacks	P0	Admin and review ag-	Privileged tokens have permissions to read/update the database but do not have any

Threat name	Severity[a]	Threat actor	Description
		gregator script	monitoring or accountability. One compromised token could compromise the entire system.

[a] *P* stands for *priority*. P0–P4 are common annotations used to designate severity in the security industry, with P0 being highest and P4 being lowest.

As you can see, by cross-referencing the previous sections of the threat model, we were able to come up with several attack vectors and tie each of them to a unique threat actor (or set of threat actors). The attack vectors table is only partial for the purpose of example, but it demonstrates several key points.

First and foremost, there are many attack vectors that originate from internal threat actors. From admin users to internal scripts running on a server, you can't ever assume that only the functionality intended for use will be invoked. Internal users get hacked, go rogue, or in the case of machine users, malfunction or encounter bugs. Each of these cases needs to be addressed during threat modeling, otherwise the system is running on a "best-case scenario" model rather than a "worst-case scenario" model. When dealing with matters of security, the worst-case scenario is always to be considered first.

Next, we can see that each of these attack vectors presents a different level of severity—that is, risk to the company or product. The way in which severity can be calculated will be discussed in depth in a future chapter, but for now it's important to simply note that not all attack vectors are of equal risk. And this is a good thing because it's a valuable method of prioritizing work when team size or work capacity is limited.

Finally, we note a wide variety of attacks from SQL injection to logic vulnerabilities. Threat modeling helps us to identify what archetypal risks our product or feature is up against, which can help when purchasing tools or investing engineering hours. Without such data, hours or dollars

could be invested poorly, securing the product or feature against attacks that are unlikely to occur.

Identifying Mitigations

In the first part of our threat model, we collected and organized all of the data required to understand this new feature. We evaluated the logic design and, afterward, dug into the technical design. From that data gathering phase, we identified threat actors—users that could attack our systems.

After identifying threat actors, we used the information gathered to identify risks in the form of attack vectors. These attack vectors represent methods by which a threat actor would attack our systems.

The next step in building a good threat model is to identify mitigations for each attack vector found. This step is quite simple; it's solely a data collection phase.

The existing mitigations for our example feature are shown in Table 24-3.

Table 24-3. Existing mitigations

Mitigation name	Attack vectors mitigated	Description
Validation logic	Improper validation—score	Our validation engine rejects any scores that are not integers or that scale beyond the bounds of 0 and 5. This prevents aggregation errors.
SQL Injection Mitigations	SQL Injection	All calls that invoke SQL code make use of our domain-specific language (DSL), which forces the use of prepared statements to prevent SQL injection attacks.

As you can see from the table, this application has existing mitigations for improper validation and SQL injection-style attacks. The remaining at-

tacks we noted during attack vector identification do not have known mitigations.

Delta Identification

Now that the attack vectors and existing mitigations have been documented, we come to one of the most important phases of threat modeling: delta identification.

Here we cross reference the attack vectors with the mitigation list, removing all attack vectors with sufficient mitigations already in place. This leaves us with the following list shown in Table 24-4.

Table 24-4. Delta identification

Threat name	Severity	Threat actor	Description
Information disclosure— `FeatureID`	P3	All users	The error messages provided on the `getReviews` endpoint allow a user to scan for potentially unreleased features or features they should not be aware exist (gated).
GraphQL circular and large queries	P1	All users	Circular and large queries become possible that are not possible with traditional REST. These are not prevented by rate limits and require max query times to prevent.
GraphQL introspection and errors	P1	All users	GraphQL's introspection engine should be disabled or we will leak server configuration details. GraphQL internal errors also leak server information if not suppressed and replaced with custom errors.
High privilege user attacks	P0	Admin and review aggregator script	Privileged tokens have permissions to read/update the database but do not have any monitoring or accountability. One compromised token could compromise the entire system.

For each of these unmitigated attack vectors, we must brainstorm a sufficient mitigation. Without sufficient mitigations for each of the unmitigated attack vectors, the feature should not be permitted to launch. Table 24-5 lists the mitigations for the attack vectors that were identified.

Table 24-5. Mitigations for identified attack vectors

Mitigation name	Attack vector mitigated	Description
Better error messages	Information disclosure—`FeatureID`	Error messages should be generic and not provide and information in regard to our application or data.
GraphQL compute limits	GraphQL circular and large queries	As GraphQL queries can become expensive and self-referential, in addition to rate limits, we must implement a maximum compute time for each query. At the end of that compute time, the request will fail.
GraphQL configuration	GraphQL introspection and errors	GraphQL must not surface any of its own error messages, and introspection queries must be turned off.
Privileged permissions rework	High privilege user attacks	To prevent damage caused by compromised users or auth tokens, privileged user tokens will be scoped only to the functionality that user requires. Furthermore, each API call will be logged off-platform using a mechanism by which the privileged user cannot revoke or delete logs. Additionally, the review aggregator script will only have write permissions in one column in the database (aggregate score) and read permissions in another column (score).

With each of these mitigations documented and in place, the next step is marking off each mitigation once it is implemented and pointing the threat model toward the implementation. Once all of the mitigations are implemented, the threat model's immediate purpose is complete and the development of the feature can continue forward.

However, should the scope of the feature increase in the future, then it is time to come back to the threat model. At this point we would update the

forward documented knowledge, update the threat actors and threat vectors, and reevaluate the mitigations and delta.

Summary

As you can see, a threat model is a fantastic all-around security tool. A properly implemented threat model is capable of documenting both technical knowledge and security knowledge in one centralized repository, allowing it to become a long-lived and highly effective source of information.

Furthermore, the threat modeling exercise in and of itself is very capable of identifying threat actors (attackers) and attack vectors (potential vulnerabilities) prior to the application even being developed.

And finally, by identifying attack vectors and listing them alongside existing mitigations, it is possible to begin refining the security posture of a system early on in the development phase when such issues are cheapest to fix. It has been said that 10 hours of fixes in the architecture phase is worth 100 hours of fixes postlaunch. If this is true, a good threat model may be one of the most cost-effective security tools in any security engineer's toolbox.