Chapter 25. Reviewing Code for Security

The code review stage must always occur after the architecture stage in a security-conscious organization—never before.

Some technology companies today uphold a "move fast and break things" mantra, but such a philosophy often is abused and used as a method of ignoring proper security processes. Even in a fast-moving company, it is imperative that application architecture is reviewed prior to shipping code. Although, from a security perspective, it would be ideal to review the entire feature architecture up front, that may not be feasible in uncertain conditions. At a minimum, the major and well-known features should be architected and reviewed. When new features come up, they should be both architected and reviewed for security prior to development as well.

The proper time to review code for security gaps is once the architecture behind the code commit has been properly reviewed. This means code reviews should be the second step in an organization that follows secure development best practices.

This has two benefits. The first and most obvious benefit is that of security, but having an additional reviewer who typically is viewing the code from outside the immediate development team has its own merits as well. This provides the developer with an unbiased pair of eyes that may catch otherwise unknown bugs and architecture flaws.

As such, the code security review phase is vital for both application functionality as well as application security. Code security reviews should be implemented as an additional step in organizations that only have functional reviews. Doing so will dramatically reduce the number of high-im-

pact security bugs that would otherwise be released into a production environment.

Generally speaking, code security reviews make the most sense when they take place on merge requests (also traditionally called “pull requests,” which is less of an accurate term in most cases). It makes sense to perform code security reviews at merging, as the full feature set has been developed and all systems that require connection should have been integrated. This is one point in time where the full scope of the code can be reviewed in a single sitting.

It may be possible to intertwine the code security review with the development process in a more granular method, such as per commit or even with a pair-programming approach. Either method would require consistent, ongoing work, as both would see the code from a point in time that does not cover the full scope of the code. However, for mission-critical security features, this may be a wise approach. With one mind focused on the feature and another on security, it may be possible to write an extremely security-conscious feature that would be otherwise impossible with reviews at merge-request time.

The timing your organization chooses for reviewing its code for security holes is up to the organization and must fit in with its existing processes. However, the preceding methods likely will be the most practical and effective for integrating security code reviews into your development process.

How to Start a Code Review

A code security review should operate very similarly to a code functionality review. Functionality reviews are standard in almost every development organization, which makes the learning curve for code security reviews much shorter.

A first step in reviewing code for security is to pull the branch in question down to a local development machine. Some organizations allow reviews in a web-based editor (provided by GitHub or GitLab; see Figure 25-1), but

these online tools are not as comprehensive as the tools you can take advantage of locally.

Here is a common local review flow that can be done from the terminal:

1. Check out main with `git checkout main`.
2. Fetch and merge the latest master with `git pull origin main`.
3. Check out the feature branch with `git checkout <username>/feature`.
4. Run a `diff` against the main with `git diff origin/main...`

The `git diff` command should return two things:

- A list of files that differ on the main branch and the current branch
- A list of changes in those files between the main branch and the current branch

This is the starting point for any code functionality review and any code security review. The differences between the two start after this point.

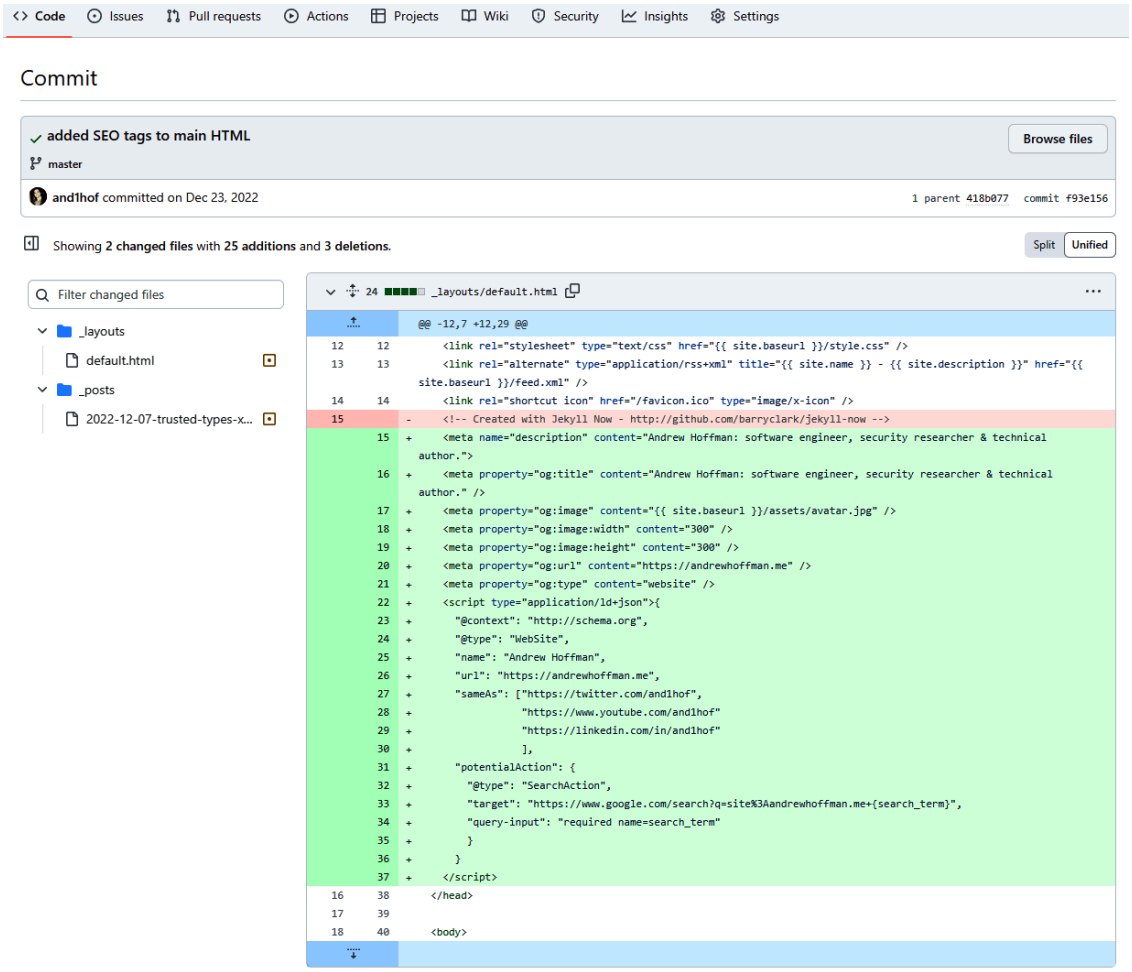

Figure 25-1. GitHub and its competitors (GitLab, Bitbucket, etc.) all offer web-based collaboration tools for making code reviews easier

Archetypical Vulnerabilities Versus Business Logic Vulnerabilities

A code functionality review checks code to ensure it meets a feature spec and does not contain usability bugs. A code security review checks for common vulnerabilities such as XSS, CSRF, injection, and so on, but more importantly checks for logic-level vulnerabilities that require deep context into the purpose of the code and cannot be easily found by automated tools or scanners.

In order to find vulnerabilities that arise from logic bugs, we need to first have context in regard to the goal of the feature. This means we need to understand *the users of the feature*, *the functionality of the feature*, and *the business impact of the feature*.

Here we run into some differences in what we have primarily discussed throughout the book when we talk about vulnerabilities. Most of the vulnerabilities we have investigated are common archetypes of well-known vulnerabilities. But it is just as possible that an application with a very specific use case has vulnerabilities that cannot be listed in a book designed for general education on software security.

Consider the following context regarding a new social media feature to be integrated into MegaBank—MegaChat:

- We are building a social media portal that allows registered users to apply for membership.
- Membership is approved by moderators based on a review of the user's activity prior to membership.
- Users have limited functionality, but when upgraded to members, they have increased functionality.
- Moderators are automatically given member functionality plus additional moderation capabilities.
- Unlike users, who can only post text media, members can upload games, videos, and artwork.
- We gate the membership because hosting this type of media is expensive, and we wish to reduce the amount of low-quality content as well as protect ourselves from bot accounts and freeloaders who are only looking to host their content.

From this we can gather the following:

Users and roles

- The users are MegaBank customers.
- The users are split into three roles: user (default), member, and moderator.
- Each user role has different permissions and functionality.

Feature functionality

- Users, members, and moderators can post text.
- Members and moderators can post video, games, and images.

- Moderators can use moderation features, including upgrading users to members.

Business impact

- The cost of hosting videos, games, and images is high.
- Membership comes at the risk of freeloading (storage/bandwidth cost) and bots (storage/bandwidth cost).

An archetypical vulnerability would be an XSS in a post made by a user. A business logic vulnerability would be a specific API endpoint that is coded improperly and allows a user to send up a payload with `isMember: true` in order to post videos, even though the user has not been granted the member functionality by a moderator.

The code review is where we will look for archetypal vulnerabilities and try to find business logic vulnerabilities that require deep application context.

Where to Start a Security Review

Ideally, you should begin your code review with the highest risk components of an application. However, you may not always be aware of what those components are if you have been asked to perform a security review against an application you did not have a say in designing. This is frequent in consulting or when working on existing products.

As a result, I propose a framework to simplify the security code review process and help you get started with a security review. This framework can be used until you are familiar enough with the given application to begin evaluating features of the application based on risk.

Imagine a basic web application with two components: a client in the browser and a server that talks to that client. Sure, we could begin by reviewing the server-side code. In fact, there is nothing wrong with that. But there may be functionality on the server that is not exposed to the client. This means that without having a good understanding of the functionality intended for users (versus internal methods and such), your ef-

fort may be accidentally focused on lower-risk code when high-risk code should be prioritized.

This is a confusing concept to grasp, but just like in Chapter 21 on secure application architecture, we need to realize that in an ideal world *every* piece of application code would be equally reviewed. Unfortunately, that reflects an ideal world. In the real world there are often deadlines, timelines, and alternate projects that require attention.

As a result, a good place to start in the actual source code is anywhere that a client (browser) makes a request to the server. Starting on the client is great because it will begin to give you a good idea of the surface area you are dealing with. From there you can learn what type of data is exchanged between the client and server and if multiple servers are being utilized rather than one. Furthermore, you can learn about the payloads being exchanged and how these payloads are being interpreted on the server.

After evaluating the client itself, follow the client's API calls back to the server. Begin evaluating calls that connect the client and the server in the web application.

Once this is complete, you should probably consider tracing the helper methods, dependencies, and functionality those APIs rely on. This means evaluating databases, logs, uploaded files, conversion libraries, and anything else that the API endpoints call directly or via a helper library.

Next, cover the bases by looking over every bit of functionality that *could* be exposed to the client but isn't directly called. This could be APIs built to support upcoming functionality, or perhaps just functionality that was accidentally exposed and should be internal.

Finally, after those major points in the codebase have been covered, dedicate your time to the rest of the codebase. Determine the route taken via analysis of the business logic and prioritization based on the risks you envision such an application encountering.

To summarize, an effective way of determining what code to review in a security review of a web application is as follows:

1. Evaluate the client-side code to gain understanding of the business logic and understand what functionality users will be capable of using.
2. Using knowledge gained from the client review, begin evaluating the API layer, in particular, the APIs you found via the client review. In doing this, you should be able to get a good understanding of what dependencies the API layer relies on to function.
3. Trace the dependencies in the API layer, carefully reviewing databases, helper libraries, logging functions, etc. In doing this, you will get close to having covered the majority of user-facing functionality.
4. Using the knowledge of the structure of the client-linked APIs, attempt to find any public-facing APIs that may be unintentionally exposed or intended for future feature releases. Review these as you find them.
5. Continue on throughout the remainder of the codebase. This should actually be pretty easy because you will already be familiar with the codebase having read through it in an organic method versus trying to brute force an understanding of the application architecture.

This is not the only method of working your way through a security review, and certain applications with niche security requirements may require a different review path. However, I suggest this path because it will grant you familiarity with the application at an organic pace and allow you to prioritize user-facing functionality while leaving potentially low-risk functionality toward the end.

As you become more familiar with the secure code review process, and the particular applications you find yourself reviewing, you should be able to modify this set of guidelines to better suit your application and the risks your application faces.

Secure-Coding Anti-Patterns

Security reviews at the code level share some similarities with architecture reviews that occur prior to code being written. Code reviews differ from architecture reviews because they are the ideal point in time to actually find vulnerabilities, whereas such vulnerabilities are only hypothetical if brought up during the architecture stage.

There are a number of anti-patterns to be on the lookout for as you go through any security review. Many times, an anti-pattern is just a hastily implemented solution or a solution that was implemented without the appropriate prior knowledge. Regardless of the cause, understanding how to spot anti-patterns will really help speed up your review process.

The following anti-patterns are all quite common, but each of them can wreak havoc on a system if they make it into a production build.

Blocklists

In the world of security, mitigations that are temporary should often be ignored and instead a permanent solution should be found, even if it takes longer. The only time a temporary or incomplete solution should be implemented is if there is a preplanned timeline from which a true complete solution will be designed and implemented. Blocklists are an example of temporary or incomplete security solutions.

Imagine you are building a server-side filtering mechanism for a list of acceptable domains that your application can integrate with:

```
const blocklist = ['http://www.evil.com', 'http://www.badguys.net'];

/*
 * Determine if the domain is allowed for integration.
 */
const isDomainAccepted = function(domain) {
 return !blocklist.includes(domain);
};
```

This is a common mistake because it looks like a solution. But even if it currently acts as a solution, it can be considered both incomplete (unless perfect knowledge of all domains is considered, which is unlikely) and temporary (even with perfect knowledge of all current domains, more evil domains could be introduced in the future).

In other words, a blocklist only protects your application if you have perfect knowledge of all possible current and future inputs. If either of those cannot be obtained, the blocklist will not offer sufficient protection and usually can be bypassed with a little bit of effort (in this case, the hacker could just buy another domain).

Allowlists are always preferable in the security world. This process could be much more secure by just flipping the way integrations are permitted:

```
const allowlist = ['https://happy-site.com', 'https://www.my-friends.com'];

/*
 * Determine if the domain is allowed for integration.
 */
const isDomainAccepted = function(domain) {
  return allowlist.includes(domain);
};
```

Occasionally, engineers will argue that allowlists create difficult product development environments, as allowlists require continual manual or automated maintenance as the list grows. With manual effort, this can indeed be a burden, but a combination of manual and automated effort could make the maintenance much easier while maintaining most of the security benefit.

In this example, requiring integrating partners to submit their website, business license, etc., for review prior to being allowlisted would make it extremely difficult for a malicious integration to slip through. Even if they did, it would be difficult for them to get through again once removed from the allowlist (they would need a new domain and business license).

Boilerplate Code

Another security anti-pattern to look for is the use of *boilerplate* or default framework code. This is a big one, and it's easy to miss because frameworks and libraries often require effort to tighten security, when they really should come with heightened security right out of the box and require loosening.

A classic example of this is a configuration mistake in MongoDB that caused older versions of the MongoDB database to be accessible over the internet by default when installed on a web server. Combined with no mandatory authentication requirements on the databases, this resulted in tens of thousands of MongoDB databases on the web being hijacked by scripts demanding Bitcoin in exchange for their return. A couple of lines in a configuration file could have resolved this by preventing MongoDB from being internet accessible (locally accessible only).

Similar issues are found in most major frameworks used around the world. Take Ruby on Rails, for example. Using boilerplate 404 page code can easily give away the version of Ruby on Rails you are using. The same goes for EmberJS, which has a default landing page designed to be removed in production applications.

Frameworks abstract away annoyingly difficult and routine work for developers, but if the developers do not understand the abstraction occurring in the framework, it is very possible the abstraction could be performed incorrectly and without proper security mechanisms in place. Hence, avoid launching any boilerplate code into production environments unless that boilerplate code has been properly evaluated and configured.

Trust-by-Default

When building an application with multiple levels of functionality, all of which request resources from the host operating system, it is crucial to implement a proper permissions model for your own code.

Imagine an application capable of generating server-side logs, writing files to disk, and performing updates against a SQL database. In many implementations, a user account will be generated on the server with permissions for logging, database access, and disk access. The application will run under this user account for all functionality. However, this means that if a vulnerability is found that permits code execution or alters the intended execution of the script, all three of these valuable server-side resources could be compromised.

Instead, a secure application would generate permissions for logging, writing to disk, and performing database operations independently of one another. Each module in a secure application would run under its own user, with specifically configured permissions that only allow what the specific function requires to operate. By doing so, a critical failure in one module would not leak over to the others, and a vulnerability in the SQL module should not give a hacker access to files or logs on the server.

Client/Server Separation

A final anti-pattern to look out for is the client/server coupling anti-pattern. This anti-pattern occurs when the client and server application code are so tightly bound that one cannot function without the other. This anti-pattern is mostly found in older web applications, but it still can be found in monolithic applications today. A secure application consisting of a client and a server should have the client and the server developed independently, and the two should communicate over a network using a predefined data format and network protocol.

Applications that consist of deep coupling between the client and server code, for example, PHP templating code with authentication logic, become much easier to exploit due to lack of separation. Rather than reading the results of a network request, a module sends back its HTML code, including any form data (for example, when dealing with authentication). Then the server must be responsible for parsing that HTML code and ensuring no script execution or parameter tampering occurs inside both the HTML code and the authentication logic.

In a totally separated client/server application, the server is not responsible for the structure and content of the HTML data. Instead, the server rejects any HTML sent and only accepts authentication payloads using a predefined data transit format.

In a distributed application, each module is responsible for less unique security mechanisms. On the other hand, a monolithic application that couples client and server code must consider security mechanisms against many languages, and consider that the data received could be formatted a large number of ways rather than a single, predefined way.

In conclusion, separation of concerns is always important from an engineering perspective as well as from a security perspective. Properly separated modules result in easier-to-manage security mechanisms, which do not need to overlap or consider rare edge cases that would occur as a result of complex interactions between multiple data/script types.

Summary

When reviewing code for security, we need to consider more than just looking for common vulnerabilities (which we will discuss in upcoming chapters). We also need to consider anti-patterns in the application that may look like solutions but become problems later down the line. Code security reviews should also be comprehensive—covering all of the potential areas for vulnerabilities to be found.

During code review, we need to consider the specific usage requirements of the application so that we can understand what logical vulnerabilities could be introduced that would not easily fit into a common, predefined vulnerability archetype. When starting a code review, we should take a logical path that allows us to gain understanding of the use cases for the application so that we can begin assessing and evaluating risk in the application. In more established applications where high-risk areas are well known, most of the reviewing effort should be focused on those areas, with the remaining areas reviewed in descending order of risk.

Ultimately, integrating security reviews into your code review pipeline will help you mitigate the odds of introducing vulnerabilities into your codebase if done correctly. The code security review process should be part of any modern software development pipeline, and it should be performed by security-knowledgeable engineers alongside the product or feature developer, when possible.