Chapter 26. Vulnerability Discovery

After securely architected code has been designed, written, and reviewed, a pipeline should be put in place to ensure that no vulnerabilities slip through the cracks. Typically, applications with the best architecture experience the least amount of vulnerabilities and the lowest risk vulnerabilities. After that, applications with sufficiently secure code review processes in place experience fewer vulnerabilities than those without such processes (but more than those with a secure-by-default architecture).

Even securely architected and sufficiently reviewed applications still fall prey to the occasional vulnerability. Vulnerabilities can slip through reviews, or come as a result of an unexpected behavior when the application is run in a different environment or its intended environment is upgraded. As a result, you need vulnerability discovery processes in place that target production code rather than preproduction code.

Security Automation

The initial step in discovering vulnerabilities past the architecture and review phases is the automation phase. Automating vulnerability discovery is essential, but not because it will catch all vulnerabilities. Instead, automation is (usually) cheap, effective, and long-lasting.

Automated discovery techniques are fantastic at finding routine security flaws in code that may have slipped past architects and code reviewers. Automated discovery techniques are not good at finding logical vulnerabilities specific to how your application functions, or finding vulnerabilities that require “chaining” to be effective (multiple weak vulnerabilities that produce a strong vulnerability when used together).

Security automation comes in a few forms; the most common are:

- Static analysis
- Dynamic analysis
- Vulnerability regression testing

Each of these forms of automation has a separate purpose and position in the application development life cycle, but each is essential as it picks up types of vulnerabilities the others would not.

Static Analysis

The first type of automation you should write, and possibly the most common, is static analysis. Static analyzers are scripts that look at source code and evaluate the code for syntax errors and common mistakes. Static analysis can take place locally during development (a linter) and on-demand against a source code repository or on each commit/push to the main branch.

Many robust and powerful static analysis tools exist, such as the following:

- Checkmarx (most major languages—paid)
- PMD (Java—free)
- Bandit (Python—free)
- Brakeman (Ruby—free)

Each of these tools can be configured to analyze the syntax of a document containing text and representing a file of code. None of these tools actually execute code, as that would move them into the next category called *dynamic analysis* or sometimes *runtime analysis*. Static analysis tools should be configured to look for common OWASP top 10 vulnerabilities.

Many of these tools exist for major languages in both free and paid form. Static analysis tools can also be written from scratch—but tools built in-house often do not perform well on codebases at scale.

For example, the following exploits are often detectable via static analysis:

General XSS

Look for DOM manipulation with innerHTML.

Reflected XSS

Look for variables pulled from a URL param.

DOM XSS

Look for specific DOM sinks like `setInterval()`.

SQL injection

Look for user-provided strings being used in queries.

CSRF

Look for state-changing GET requests.

DoS

Look for improperly written regular expressions.

Further configuration of static analysis tooling can also help you enforce best secure coding practices. For example, your static analysis tools should reject API endpoints that do not have the proper authorization functions imported, or functions consuming user input that do not draw from a single source of truth validations library.

Static analysis is powerful for general-purpose vulnerability discovery, but it may also be a source of frustration because it will report many false positives. Additionally, static analysis suffers when dealing with dynamic languages (like JavaScript). Statically typed languages like Java or C# are much easier to perform static analysis on, as the tooling understands the expected data type, and that data cannot change type as it traverses through functions and classes.

Dynamically typed languages, on the other hand, are much more difficult to perform accurate static analysis on. JavaScript is a fine example of this because JavaScript variables (including functions, classes, etc.) are mutable objects—they can change at any point in time. Furthermore, with no typecasting, it is difficult to understand the state of a JavaScript application at any time without evaluating it at runtime.

To conclude, static analysis tooling is great for finding common vulnerabilities and misconfigurations, particularly with regard to statically typed programming languages. Static analysis tooling is not effective at finding advanced vulnerabilities involving deep application knowledge, chaining of vulnerabilities, or vulnerabilities in dynamically typed languages.

Dynamic Analysis

Static analysis looks at code, typically prior to execution. On the other hand, dynamic analysis looks at code post-execution. Because dynamic analysis requires code execution, it is much more costly and significantly slower. In a large application, dynamic analysis requires a production-like environment (servers, licenses, etc.) prior to having any utility.

Dynamic analysis is fantastic at picking up actual vulnerabilities, whereas static analysis picks up many potential vulnerabilities but has limited ways of confirming them.

Dynamic analysis executes code prior to analyzing the outputs and comparing them against a model that describes vulnerabilities and misconfigurations. This makes it great for testing dynamic languages, as it can see the output of the code rather than the (vague) inputs and flow. It is also great for finding vulnerabilities that occur as a side effect of proper application operation—for example, sensitive data improperly stored in memory or side-channel attacks.

Dynamic analysis tools exist for many languages and frameworks. Some examples of these are:

- IBM AppScan (paid)

- Veracode (paid)
- Iroh (free)

Due to the increased complexity of functioning in a production-like environment, the better tools are often paid or require significant upfront configuration. Simple applications can build their own dynamic analysis tools, but for complete automation at the CI/CD level, they will require significant effort and a bit of upfront cost.

Unlike static analysis tools, dynamic analysis tooling that is properly configured should have fewer false positives and give deeper introspection with regard to your application. The trade-off is in maintenance, cost, and performance when compared to static analysis tooling.

Vulnerability Regression Testing

The final form of automation that is essential for a secure web application is vulnerability regression testing nets. Static analysis and dynamic analysis tools are cool, but they are difficult to set up, configure, and maintain compared to regression tests.

A vulnerability regression testing suite is simple. It works similarly to a functional or performance testing suite, but a vulnerability regression test tests to see whether a known vulnerability still exists. This prevents the flaw from being reintroduced to the codebase.

You don't need a special framework for security vulnerability tests. Any testing framework capable of reproducing the vulnerability should do. Figure 26-1 shows Jest, a fast, clean, and powerful testing library for JavaScript applications. Jest can be easily modified to test for security regressions.

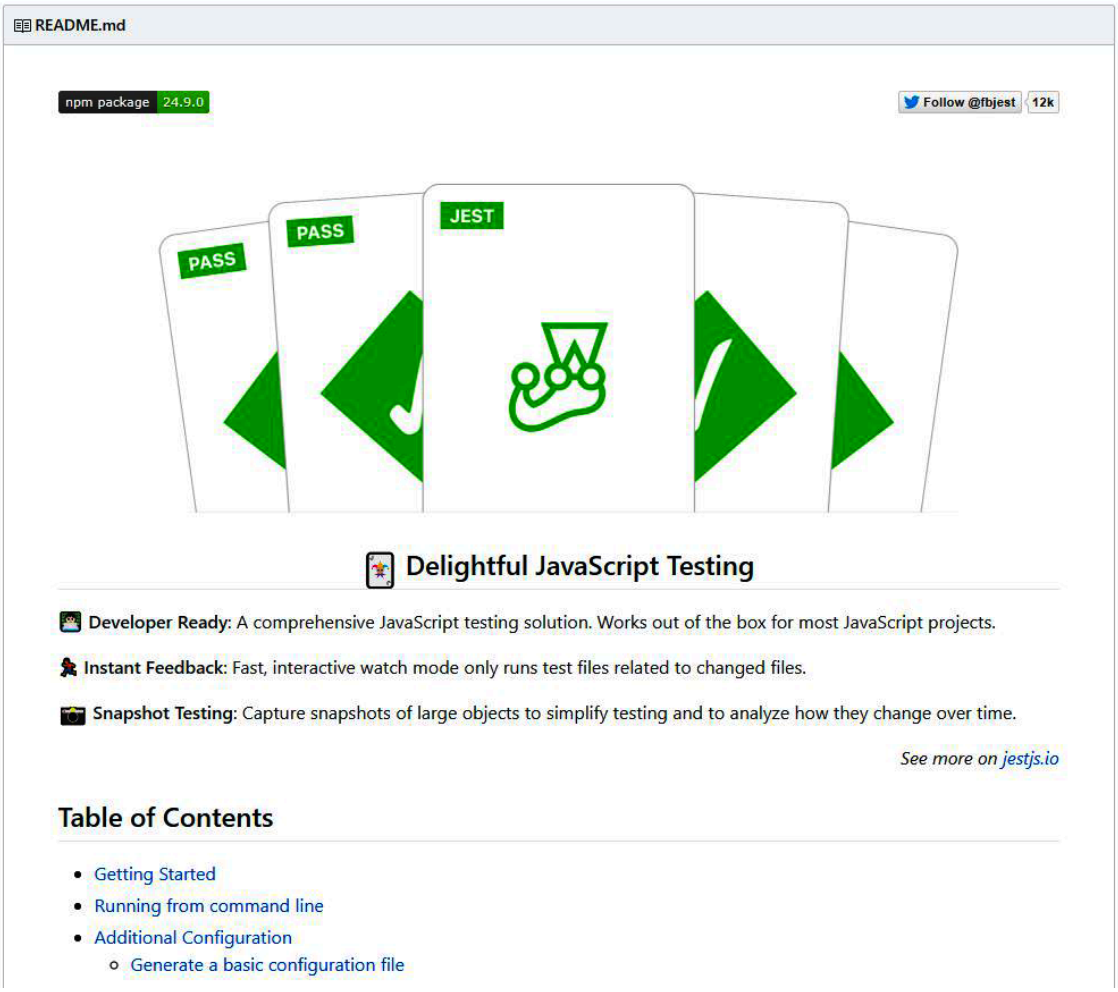

Figure 26-1. The Jest testing library

Imagine the following vulnerability. Software engineer Steve introduced a new API endpoint in an application that allows a user to upgrade or downgrade their membership on-demand from a UI component in their dashboard:

```
const currentUser = require('../currentUser');
const modifySubscription = require('../../modifySubscription');

const tiers = ['individual', 'business', 'corporation'];

/*
 * Takes an HTTP GET on behalf of the currently authenticated user.
 *
 * Takes a param `newTier` and attempts to update the authenticated
 * user's subscription to that tier.
 */
app.get('/changeSubscriptionTier', function(req, res) {
 if (!currentUser.isAuthenticated) { return res.sendStatus(401); }
 if (!req.params.newTier) { return res.sendStatus(400); }
```

```
  if (!tiers.includes(req.params.newTier)) { return res.sendStatus(400); }

  modifySubscription(currentUser, req.params.newTier)
  .then(() => {
    return res.sendStatus(200);
  })
  .catch(() => {
    return res.sendStatus(400);
  });
});
```

Steve's old friend Jed, who is constantly critiquing Steve's code, realizes that he can make a request like `GET /api/changeSubscriptionTier` with any tier as the `newTier` param and sends it via hyperlink to Steve. When Steve clicks this link, a request is made on behalf of his account, changing the state of his subscription in his company's application portal.

Jed has discovered a CSRF vulnerability in the application. Luckily, although Steve is annoyed by Jed's constant critiquing, he realizes the danger of this exploit and reports it back to his organization for triaging. Once triaged, the solution is to switch the request from an `HTTP GET` to an `HTTP POST` instead.

Not wanting to look bad in front of his friend Jed again, Steve writes a vulnerability regression test:

```
const tester = require('tester');
const requester = require('requester');

/*
 * Checks the HTTP Options of the `changeSubscriptionTier` endpoint.
 *
 * Fails if more than one verb is accepted, or the verb is not equal
 * to 'POST'.
 * Fails on timeout or unsuccessful options request.
 */
const testTierChange = function() {
 requester.options('http://app.com/api/changeSubscriptionTier')
  .on('response', function(res) {
   if (!res.headers) {
```

```
      return tester.fail();
    } else {
      const verbs = res.headers['Allow'].split(',');
      if (verbs.length > 1) { return tester.fail(); }
      if (verbs[0] !== 'POST') { return tester.fail(); }
    }
  })
  .on('error', function(err) {
    console.error(err);
    return tester.fail();
  })
};
```

This regression test looks similar to a functional test, and it is!

The difference between a functional test and a vulnerability test is not the framework but the purpose for which the test was written. In this case, the resolution to the CSRF bug was that the endpoint should only accept `HTTP POST` requests. The regression test ensures that the endpoint `changeSubscriptionTier` only takes a single HTTP verb, and that verb is equal to `POST`. If a change in the future introduces a non-POST version of that endpoint, or the fix is overwritten, then this test will fail, indicating that the vulnerability has regressed.

Vulnerability regression tests are simple. Sometimes they are so simple, they can be written prior to a vulnerability being introduced. This can be useful for code in which minor insignificant-looking changes could have a big impact. Ultimately, vulnerability regression testing is a simple and effective way of preventing vulnerabilities that have already been closed from re-entering your codebase.

The tests themselves should be run on commit or push hooks when possible (reject the commit or push if the tests fail). Regularly scheduled runs (daily) are the second-best choice for more complex version control environments.

Responsible Disclosure Programs

In addition to having the appropriate automation in place to catch vulnerabilities, your organization should also have a well-defined and publicized way of disclosing vulnerabilities in your application. It's possible your internal testing doesn't cover all potential use cases of your customers. Because of this, it's very possible your customers will find vulnerabilities that would otherwise go unreported.

Unfortunately, several large organizations have taken vulnerability reports from their users and turned them into lawsuits and hush orders against the reporter. Because the law doesn't define the difference between ethical research and malicious exploitation well, it's very possible that your application's most tech-savvy users will not report accidentally found vulnerabilities unless you explicitly define a path for responsible disclosure.

A good responsible disclosure program will include a list of ways that your users can test your application's security without incurring any legal risk. Beyond this, your disclosure program should define a clear method of submission and a template for a good submission.

To reduce the risk of public exposure prior to the vulnerability being patched in your application, you can include a clause in the responsible disclosure program that prevents a researcher from publicizing a recently found vulnerability. Often a responsible disclosure program will list a period of time (weeks or months) during which the reporter cannot discuss the vulnerability externally while it is fixed. A properly implemented vulnerability disclosure program will further reduce the risk of exploitable vulnerabilities being left open and improve public reception of your development team's commitment to security.

Bug Bounty Programs

Although responsible disclosure allows researchers and end users to report vulnerabilities found in your web application, it does not offer incen-

tives for actually testing your application and finding vulnerabilities. Bug bounty programs have been employed by software companies for the past decade, offering cash prizes in exchange for properly submitted and documented vulnerability reports from end users, ethical hackers, and security researchers.

Starting a bug bounty program used to be a difficult process that required extensive legal documentation, a triage team, and specially configured sprint or kanban processes for detecting duplicates and resolving vulnerabilities. Today, intermediate companies exist to facilitate the development and growth of a bug bounty program.

Companies like HackerOne and Bugcrowd provide easily customizable legal templates and a web interface for submission and triaging. HackerOne is one of the most popular bug bounty platforms on the web and helps small companies set up bug bounty programs and connect with security researchers and ethical hackers (see Figure 26-2).

Figure 26-2. HackerOne, a bug bounty platform

Making use of a bug bounty program in addition to issuing a formal responsible disclosure policy will allow freelance penetration testers (bug bounty hunters) and end users to not only find vulnerabilities, but also be incentivized to report them.

Third-Party Penetration Testing

In addition to creating a responsible disclosure system and incentivizing disclosure via bug bounty programs, third-party penetration testing can give you deeper insight into the security of your codebase that you could not otherwise get via your own development team. Third-party penetration testers are similar to bug bounty hunters as they are not directly affiliated with your organization, but provide insight into the security of your web application. Bug bounty hunters are (mostly, minus the top 1%) freelance penetration testers. They work when they feel like it and don't have a particular agenda to stick to.

Penetration testing firms, on the other hand, can be assigned particular parts of an application to test—and often through legal agreements can be safely provided with company source code (for more accurate testing results). Ideally, contracted tests should target high-risk and newly written areas of your application's codebase prior to release into production. Post-release tests are also valuable for high-risk areas of the codebase and for testing to ensure security mechanisms remain constant across platforms.

Summary

There are many ways to find vulnerabilities in your web application's codebase, each with its own pros, cons, and position in the application's life cycle. Ideally, several of these techniques should be employed to ensure that your organization has the best possible chance of catching and resolving serious security vulnerabilities before they are found or exploited by a hacker outside of your organization.

By combining vulnerability discovery techniques like the ones described in this chapter, with proper automation and feedback into your SSDL, you will be able to confidently release production web applications without significant fear of serious security holes being discovered in production.