Chapter 7. Identifying Weak Points in Application Architecture

So far we have discussed a number of techniques for identifying components in a web application, determining the shape of APIs in a web application, and learning how a web application expects to interact with a user's web browser. Each technique is valuable by itself, but when the information gathered from them is combined in an organized fashion, even more value can be gained.

Ideally, you're keeping notes of some sort throughout the recon process, as suggested earlier. Proper documentation of your research is integral, as some web applications are so expansive that exploring all of their functionality could take months. The amount of documentation created during recon is ultimately up to you (the tester, hacker, hobbyist, engineer, etc.) and more isn't always more valuable if not prioritized correctly, although more data is still better than no data.

With each application you test, you would ideally end up with a well-organized set of notes. These notes should cover:

- Technology used in the web application
- List of API endpoints by HTTP verb
- List of API endpoint shapes (where available)
- Functionality included in the web application (e.g., comments, auth, notifications, etc.)
- Domains used by the web application
- Configurations found (e.g., Content Security Policy [CSP])
- Authentication/session management systems

Once you have finished compiling this list, you can use it to prioritize any attempts at hacking the application or finding vulnerabilities.

Contrary to popular belief, most vulnerabilities in a web application stem from improperly designed application architecture rather than from poorly written methods. Sure, a method that writes user-provided HTML directly to the DOM is definitely a risk and may allow a user to upload a script (if proper sanitization is not present) and execute that script on another user's machine (XSS).

But there are applications out there that have dozens of XSS vulnerabilities, while other similarly sized applications in the same industry have nearly zero. Ultimately, the architecture of an application and the architecture of the modules/dependencies within that application are fantastic markers of weak points from which vulnerabilities may arise.

Secure Versus Insecure Architecture Signals

As mentioned earlier, a single XSS vulnerability may be the result of a poorly written method. But multiple vulnerabilities are probably the sign of weak application architecture.

Let's imagine two simple applications that allow users to send direct messages (texts) to other users. One of these applications is vulnerable to XSS, while the other is not.

The insecure application might not reject a script when a request to store a comment is made to an API endpoint; its database might not reject the script, and it might not perform proper filtration and sanitization against the string representing the message. Ultimately, it is loaded into the DOM and evaluated as DOM `message<script>alert('hacked');</script>`, thus resulting in script execution.

The secure application, on the other hand, likely has one or many of the preceding protections. However, implementing multiples of these protections on a per-case basis would be expensive in terms of developer time and could be easily overlooked.

Even an application written by engineers skilled in application security would likely have security holes eventually if its application architecture was inherently insecure. This is because a secure application implements security prior to *and during* feature development, whereas an application with mediocre security implements security at feature development, and an insecure application might not implement any.

If a developer has to write 10 variations on the instant messaging (IM) system in the preceding example across a timespan of 5 years, it is likely that each implementation will be structurally different. However, the security risks between each implementation will be mostly the same.

Each of these IM systems includes the following functionality:

- UI to write a message
- API endpoint to receive a message just written and submitted
- A database table to store a message
- An API endpoint to retrieve one or more messages
- UI code to display one or more messages

At a bare minimum, the application code looks like this:

client/write.html

```
<!-- Basic UI for Message Input -->
<h2>Write a Message to <span id="target">TestUser</span></h2>
<input type="text" class="input" id="message"></input>
<button class="button" id="send" onclick="send()">send message</button>
```

client/send.js

```
const session = require('./session');
const messageUtils = require('./messageUtils');

/*
 * Traverses DOM and collects two values, the content of the message
 * to be sent and the username or other unique identifier (id) of
 * the target message recipient.
```

```
 *
 * Calls messgeUtils to generate an authenticated HTTP request to send
 * the provided data (message, user) to the API on the server.
 */
const send = function() {
  const message = document.querySelector('#send').value;
  const target = document.querySelector('#target').value;

  messageUtils.sendMessageToServer(session.token, target, message);
};
```

server/postMessage.js

```
const saveMessage = require('./saveMessage');

/*
 * Receives the data from send.js on the client, validating the user's
 * permissions and saving the provided message in the database if all
 * validation checks complete.
 *
 * Returns HTTP status code 200 if successful.
 */
const postMessage = function(req, res) {
  if (!req.body.token || !req.body.target || !req.body.message) {
    return res.sendStatus(400);
  }

  saveMessage(req.body.token, req.body.target, req.body.message)
  .then(() => {
    return res.sendStatus(200);
   })
   .catch((err) => {
    return res.sendStatus(400);
   });
};
```

server/messageModel.js

```
const session = require('./session');

/*
 * Represents a message object. Acts as a schema so all message objects
```

```
 * contain the same fields.
 */
const Message = function(params) {
  user_from: session.getUser(params.token),
  user_to: params.target,
  message: params.message
};

module.exports = Message;
```

server/getMessage.js

```
const session = require('./session');

/*
 * Requests a message from the server, validates permissions, and if
 * successful pulls the message from the database and then returns the
 * message to the user requesting it via the client.
 */
const getMessage = function(req, res) {
 if (!req.body.token) { return res.sendStatus(401); }
 if (!req.body.messageId) { return res.sendStatus(400); }

 session.requestMessage(req.body.token, req.body.messageId)
 .then((msg) => {
   return res.send(msg);
  })
  .catch((err) => {
   return res.sendStatus(400);
  });
};
```

client/displayMessage.html

```
<!-- displays a single message requested from the server -->
<h2>Displaying Message from <span id="message-author"></span></h2>
<p class="message" id="message"></p>
```

client/displayMessage.js

```
const session = require('./session');
const messageUtils = require('./messageUtils');

/*
 * Makes use of a util to request a single message via HTTP GET and then
 * appends it to the #message element with the author appended to the
 * #message-author element.
 *
 * If the HTTP request fails to retrieve a message, an error is logged to
 * the console.
 */
const displayMessage = function(msgId) {
 messageUtils.getMessageById(session.token, msgId)
 .then((msg) => {
  messageUtils.appendToDOM('#message', msg);
  messageUtils.appendToDOM('#message-author', msg.author);
 })
 .catch(() => console.log('an error occured')););
};
```

Many of the security mechanisms needed to secure this simple application could, and likely should, be abstracted into the application architecture rather than implemented on a case-by-case basis.

Take, for example, the DOM injection. A simple method built into the UI like the following would eliminate most XSS risk:

```
import { DOMPurify } from '../utils/DOMPurify';

// makes use of: https://github.com/cure53/DOMPurify
const appendToDOM = function(data, selector, unsafe = false) {
  const element = document.querySelector(selector);

  // for cases where DOM injection is required (not default)
  if (unsafe) {
   element.innerHTML = DOMPurify.sanitize(data);
  } else { // standard cases (default)
   element.innerText = data;
  }
};
```

Simply building your application around a function like this would dramatically reduce the risk of XSS vulnerabilities arising in your codebase.

However, the implementation of such methods is important—note that the DOM injection flag in the preceding code sample is specifically labeled `unsafe`. Not only is it off by default, but it also is the final param in the function signature, which means it is unlikely to be flipped by accident.

Mechanisms like the preceding `appendToDOM` method are indicators of a secure application architecture. Applications that lack these security mechanisms are more likely to include vulnerabilities. This is why identifying insecure application architecture is important for both finding vulnerabilities and prioritizing improvements to a codebase.

Multiple Layers of Security

In the previous example where we considered the architecture of a messaging service, we isolated and identified multiple layers where XSS risk could occur. The layers were:

- API POST
- Database Write
- Database Read
- API GET
- Client Read

The same can be said for other types of vulnerabilities, such as XXE or Cross-Site Request Forgery (CSRF)—each vulnerability can occur as a result of insufficient security mechanisms at more than one layer. For example, let's imagine that a hypothetical application (like the messaging app) added mechanisms at the API POST layer in order to eliminate XSS risk by sanitizing payloads (messages) sent by users. It may now be impossible to deploy an XSS via the API POST layer.

However, at a later point in time, another method of sending messages may be developed and deployed. An example of this would be a new API

POST endpoint that accepted a list of messages in order to support bulk messaging. If the new API endpoint does not offer sanitization as powerful as the original, it may be used to upload payloads containing script to the database, bypassing the original intentions of the developer in the single-message API.

I am bringing this up as a simple example to point out that an application is only as secure as the weakest link in its architecture. Had the developers of this service implemented mechanisms in multiple locations, such as API POST and Database Write stages, then the new attack could have been mitigated.

Sometimes, different layers of security can support different mechanisms for defending against a particular type of attack. For example, the API POST could invoke a headless browser and attempt to simulate the rendering of a message to the page, rejecting the message payload if any script execution is detected. A mitigation involving a headless browser would not be possible at the database layer or the client layer.

Different mechanisms can detect different attack payloads as well. The headless browser may detect script execution, but should a browser-specific API have a bug, it may be possible for the script to bypass this mechanism. This could occur because the payload would not execute in the headless browser but only in the browser of a user with a vulnerable browser version (which is different than the browser or version tested on the server).

All of these examples suggest that the most secure web applications introduce security mechanisms at many layers, whereas insecure web applications introduce security mechanisms at only one or two layers. When testing web applications, you want to look for functionality in an application that makes use of a few security mechanisms or requires a significant number of layers (hence likely to have a lower ratio of security mechanisms to layers). If you can isolate and determine what functionality meets this criteria, it should be prioritized over the rest when looking for vulnerabilities; it is more likely to be exploitable.

Adoption and Reinvention

A final risk factor to pay attention to is the desire for developers to reinvent existing technology. Generally, this does not start as an architecture problem. Instead it is usually an organizational problem, which is reflected and visible in the application architecture.

This is commonplace in many software companies, as reinventing tools or features comes with a number of benefits from a development perspective including:

- Avoiding complicated licenses
- Adding additional functionality to the feature
- Creating publicity via marketing the new tool or feature

Beyond that, creating a feature from scratch is usually much more fun and challenging than repurposing an existing open source or paid tool. But it is not always bad to reinvent, so each case must be evaluated individually.

There are scenarios where reinvention of existing software may bring more benefits than pitfalls to a company. An example of this would be if the best tool had a licensing agreement that required a significant commission leading to negative margins, or prohibited alteration so that the application would forgo essential functionality.

On the other hand, reinvention is risky from a security point of view. The risk waxes and wanes based on the particular functionality being reinvented but can span anywhere from a moderate security risk all the way to an extreme security risk.

In particular, well-versed security engineers suggest never rolling your own cryptography. Talented software engineers and mathematicians may be able to develop their own hashing algorithms to avoid using open algorithms—but at what cost?

Consider the hashing algorithm SHA-3. SHA-3 is an open source hashing algorithm that is part of the SHA family of algorithms, which has been in development for nearly 20 years, and has received robust testing from the National Institute of Standards and Technology (NIST), as well as contributions from the largest security firms in the United States.

Hashes generated from hashing algorithms are attacked regularly from a multitude of attack vectors (e.g., combinator attacks, Markov attacks, etc.). A developer-written hashing algorithm would have to hold up to the same robustness as the best open algorithms.

Rolling out an algorithm with the same extensive level of testing that NIST and other organizations provided for the development of SHA-3 would cost an organization tens of millions of dollars. But for zero dollars, the organization could adopt an implementation of SHA-3 from a source like OpenJDK and still gain all of the benefits that come from NIST and community testing.

It is likely that the lone software developer who decides to roll out their own hashing algorithm will not be able to meet the same standards and conduct robust testing. As a result, the organization's critical data will be an easy target for hackers.

So how can we determine which features or tools to adopt and which to reinvent? In general, a securely architected application will only reinvent features that are purely functional, such as reinventing a schema for storing comments or a notification system.

Features that require deep expertise in mathematics, operating systems, or hardware should probably be left alone by web application developers. This includes databases, process isolation, and most memory management.

It's impossible to be an expert at everything. A good web application developer understands this and will focus their energy on developing where their expertise lies and request assistance when operating outside of their primary domain. On the flip side, bad developers often do attempt to reinvent mission-critical functionality—this is not uncommon!

Applications full of custom databases, custom cryptography, and special hardware-level optimization often are the easiest to break into. Rare exceptions to this rule may exist, but they are the outliers and not the norm.

Summary

When talking about vulnerabilities in web applications, we are usually talking about issues that occur at the code level, or as a result of improperly written code. However, issues that appear at the code level can be easily spotted earlier in the application architecture. Often, the architectural design of an application leads to either a plethora of security bugs or a relatively low number of security bugs based on how the application's defenses are designed and distributed throughout the codebase.

Because of this, the ability to identify weak points in an application's architecture is a useful recon technique. Poorly architected features should be focused on first when looking for vulnerabilities, as often features with good security architecture will remain more consistent when jumping from endpoint to endpoint or attempting to bypass filtration systems.

Application architecture is often discussed at a very high level rather than the low level at which most security work takes place. This can make it a confusing topic to tackle if you aren't used to considering applications from a design perspective.

When investigating a web application as part of your recon efforts, make sure to consider the overall security architecture of the application as you make your map of it. Mastering architectural analysis not only will help you focus your efforts when looking for vulnerabilities, but might also help you identify weak architecture in future features by spotting patterns that caused bugs to appear in prior features.