Chapter 36. Mitigating Business Logic Vulnerabilities

In Chapter 18, we discussed the elusive business logic vulnerability. This is an advanced form of vulnerability that is not easily detectable via automation—and not easily found via penetration testing.

Business logic vulnerabilities usually require deep knowledge of an application's *business logic*, and as such, are more difficult to attack. Fortunately, because deep engineering knowledge is often required to understand such vulnerabilities, defending against these vulnerabilities is quite a bit easier than attacking them.

Given the presumption that your security team is working closely with your engineering teams, you will actually have an advantage when it comes to mitigating business logic vulnerabilities and protecting your application. This chapter discusses methods of preventing and mitigating business logic vulnerabilities.

Architecture-Level Mitigations

The most important step toward mitigating business logic vulnerabilities occurs in the architecture phase, prior to any application code being written. In traditional web application architecture designs, the *intended user* is considered alongside the *intended use case*.

It is unfortunate that this is the case, as many other technical domains have already identified the value in *worst-case scenario* design. Let's evaluate the following example demonstrating the benefits of worst-case design before discussing how we can use these principles to benefit our application's security.

Consider the programming case of an algorithm (A1) that runs with a median Big-O time of (n). Given five elements as an input to the algorithm A1, we can deduce this algorithm will have a median runtime of C = 5.

However, if 1 out of 10 times the algorithm has a run time of n^2, then with five elements, we are left with a runtime of C = 25 in the worst-case scenario. If all other runs end up at the median run time of C = 5, then we have 9 runs that take a time of 5 and 1 run that takes a time of 25. This is represented in Table 36-1.

Table 36-1. Algorithm A1 runtimes

Run number	**Runtime**	**Median runtime**	**Average runtime**
1	5	5	5
2	5	5	5
3	5	5	5
4	5	5	5
5	5	5	5
6	5	5	5
7	5	5	5
8	5	5	5
9	5	5	5
10	25	5	7

This leaves us with an average runtime over 10 tries of (9 × 5 + 25)/10 = 7 for the A1 algorithm. The median runtime is still 5.

Let’s consider that we had two algorithms available initially, A1 (which we just looked at) and A2. Here are the specs for each using the median evaluation:

A1 median time: 5

A2 median time: 6

With the approach of evaluating based on the median, we assume A1 is the better option. But if we drill down and realize that A2 is always running at a constant speed of 6, we then realize that over time A1 trends toward 7, which makes it 16% less efficient than A1.

This is an example of *best-case design*. Good security architects never use best-case design and always use worst-case design.

Had an architect chosen the A1 algorithm, it would perform faster than A2 in this application until an edge case occurred. Over many iterations, those edge cases would slow down A1 relative to A2 to the point at which A2 is actually faster than A1.

Simply considering from the start the malicious use case for every functional component in an application gives you the advantage of avoiding the majority of business logic vulnerabilities. By implementing worst-case scenario design in every application architecture, you will catch business logic vulnerabilities prior to any code ever being written.

In previous chapters, we discussed how most business logic vulnerabilities arise from a user making use of applications in unintended or unexpected ways. Similarly to the preceding example, accounting for the worst-case scenario for every architecture produced will allow you to find these unexpected scenarios prior to writing any code or exposing your users to a vulnerable production best-case design application.

This is likely the most effective method of stopping business logic vulnerabilities in terms of time allocation. Post-architecture, these vulnerabilities take much more time and are more expensive to resolve.

Statistical Modeling

On the more mathematical end of the detection spectrum, we can make use of *statistical modeling*. This technique combines aspects of *fuzzing* (the process of testing an application with random data inputs) with data science and browser automation in order to allow you to prune, select, and iterate in a much more rapid fashion in regard to detecting business logic vulnerabilities than with randomized fuzzing alone.

In the first step of this process, a developer integrates analytics into the web application or produces a hypothesis regarding the ways in which a user will make use of the application. This model should include both *inputs* and *directional pageflows (actions)* that a user is assumed to make use of.

Modeling Inputs

For each input in the application, consider the most likely values a user would make use of. For a dropdown list, rank the dropdown items in terms of frequency chosen. For a free text input, rank the most common inputs. For a radio button, consider which is selected most often.

Uncommon and unexpected inputs should also be considered. In other words, if the most common value input into to a free text field labeled “name” would be a name from the list of the top one thousand most popular names, we must also include the fact that a smaller percentage of users will choose a name that is not on the list.

In practice, these names will eventually begin to include uncommon characters, lengths, and so on. These uncommon edge cases are important because they are more likely to highlight logic vulnerabilities where code did not appropriately account for said input value.

Continue modeling fields in the application until all user-interactive fields have been modeled. Modeled inputs (and actions) should be stored in a format that is easy to ingest programmatically, for example JSON, YAML, CSV, or XML.

Modeling Actions

In a legacy web application, *actions* are typically directional pageflows. In a modern application, these actions often include clicks on buttons, links, and forms that create AJAX requests in the background—in addition to directional pageflows.

Sometimes, web applications will also make use of interactive elements that spawn modals, initiate websocket or real-time communication (RTC) network calls, or call JavaScript functions directly. We need to model all of these actions prior to moving forward.

Luckily, action paths are easy to obtain and model with popular analytics tools that most companies already make use of. So it's likely your business already has this data despite not having acted on it for security insights.

For easier automation down the line, record all of the potential actions your users may take in the same format as you did for the inputs.

Model Development

The next step after building a model of inputs and actions is to automate user flows programmatically. This will enable you to begin identifying business logic vulnerabilities that occur, but in a safe local testing environment.

From a technical perspective, a user is just a set of bits in a database, so populating a database with model users should not be difficult. Simulating the web app's traversal and use of actions with those users, on the other hand, can be a bit more technically challenging.

Luckily, tools called *headless browsers* exist to allow web app traversal automation in a streamlined and easily adoptable fashion. These browsers implement the browser DOM, run JavaScript code, and can perform network queries like a normal browser—but are controlled programmatically rather than via a UI.

The most popular of these tools is Google's Headless Chrome, which adopted a similar API to Phantom.js and Slimer.js, which were the most frequently used browser-control tools from the previous decade. You will need to make use of one of these tools to test your model programmatically.

Consider the following code snippet, which makes use of Google's Headless Chrome API under the hood:

```
import puppeteer from 'puppeteer';
import data from 'model';
import tools from 'tools';

(async () => {
  const browser = await puppeteer.launch();
  const page = await browser.newPage();

  await page.goto(data.startURL);

  // Configure headless browser
  await page.setViewport({width: 1080, height: 1024});

  // Create user
  await page.type('.sign-up-username', data[0].username);
  await page.type('.sign-up-username', data[0].password);
  await page.click('.sign-up');
  await tools.logSignUpStatus()

  // Add comment when signed up
  const commentBox = '.comment-box';
  await page.waitForSelector(commentBox);
  await page.type('.comment-box', data[0].messages[0]);
  await page.click('.submit-comment');
  await tools.logCommentStatus()

  // Close browser and end automation
  await browser.close();
})();
```

This simple example makes use of the Puppeteer wrapper library for Google's Headless Chrome, which is typically only accessible via the command line or terminal. It draws in data from the model we developed and programmatically executes a set of inputs, clicks, and page transitions against the target website.

The primary difference between this example and your implementation is that the specific automations required for this form of vulnerability detection will depend on the specific business logic within your web application. However, the programming model is likely to be very similar to the Headless Chrome Puppeteer example; browsers like Chrome now implement standard APIs for automating the browser in such a fashion.

Model Analysis

Throughout the process of running a model against a headless web browser making use of your web application, you should be logging every network request in terms of payload, response, and HTTP status code. In particular, you want to make sure that any unexpected errors or faults in the server are easy to find after running through a significant sample size.

These errors will often lead you to business logic vulnerabilities, which can then be remediated by your engineering teams.

In the case that an error is found that cannot be exploited—do not worry. This exercise still provides valuable insights into nonsecurity bugs, and resolving them will improve your user experience.

Summary

Business logic vulnerabilities arise from an application's architects either not considering a specific edge case or forgetting to implement proper checks and balances inside of application logic. These vulnerabilities present a number of issues, primarily due to the fact that they differ from one application to another.

While business logic vulnerabilities may share some similarities between two applications in the same industry, the methods by which they are attacked from both a logical standpoint and a technical standpoint will likely differ. As such, these vulnerabilities are both difficult to attack and difficult to detect, mitigate, and remediate.

The most important component of properly preparing to detect, mitigate, and remediate these vulnerabilities is to begin thinking of how your programmed functionality can be used in unintended ways. Once this pattern of thought has been established, application architecture can be more securely designed, automated defense systems may become feasible to develop, and the security posture of your application against these attacks may finally begin to improve. Do not neglect these vulnerabilities, even if they sometimes initially appear as programming bugs rather than security vulnerabilities.

The business logic that is described in the source code of your web application is often the most critical digital asset your business has. The more these vulnerabilities are neglected because of difficulty or required expertise, the more and more dangerous they become due to the fact that more functionality and power is often added to an application over time. Remember the old programming adage, “1 minute of fixing architecture bugs saves 10 minutes of resolving implementation bugs.”