Chapter 18. Business Logic Vulnerabilities

In the previous chapters, we discussed a number of common vulnerabilities that affect most web applications. These vulnerabilities were easily categorized using terms like *injection* or *denial of service*. The aforementioned vulnerabilities almost always took on a consistent shape, which made them easy to define categorically. This also means that both offensive and defensive strategies for attacking or mitigating common vulnerabilities are relatively consistent across all affected applications.

Until now, we have studied archetypal web application vulnerabilities. But what happens when we encounter a vulnerability that is unique to a single application? Unique vulnerabilities most frequently occur as a result of an application implementing specific *business logic* rules. A hacker then learns ways to make use of those programmed rules and obtain unintended outcomes. In other words, the vulnerabilities we have studied up until now are vulnerabilities that may occur as a result of *application logic* but not because of *business rules*.

Application logic combines information and instructions in order to perform tasks common to web applications, like rendering an image or performing a network call. Business rules, on the other hand, are specific to the implementing business. An example of a business rule would be only allowing a passenger to cancel a reservation if the current time is greater than 24 hours prior to the booking time.

A vulnerability in the implementation of a business rule is often denoted as a *business logic vulnerability*. This form of vulnerability is much more difficult to find and exploit because, per the definition, they require un-

derstanding of a specific set of business rules that are unique to a particular application.

This is in contrast to common vulnerabilities that make use of standard application logic implemented in relatively consistent ways across many applications. As a result of this added complexity, business logic vulnerabilities are often the most difficult vulnerabilities to find and exploit—but also often have the greatest impact and net the greatest rewards if exploited.

On a final note, business logic vulnerabilities, while being difficult to find manually, are also almost impossible to find using standard automated tooling. This is because almost all tools designed to find vulnerabilities are not designed for a specific application and set of business rules, but instead configured to attack a wide array of applications that make use of shared application frameworks, languages, and system designs.

Without further ado, let's evaluate some common business logic vulnerabilities—and discuss methods of exploiting them.

Custom Math Vulnerabilities

Math-related vulnerabilities are one of the most common forms of business logic vulnerability. Almost all applications make use of mathematics to perform calculations on behalf of their users, but the logic behind the mathematical operations often differs.

Consider the example application MegaBank. In previous chapters we noted that the web application hosted by MegaBank for its customers permits an authenticated user to complete a transaction by sending currency to another user.

Let's consider what a transfer looks like on behalf of an application like MegaBank:

1. The currently authenticated user (User A) designates in a user interface that they wish to send $500 to User B.

2. The server validates that User B exists.
3. The server sends a confirmation request to User A .
4. User A sends a confirmation “accept” request to the server .
5. The server subtracts from user A by reducing the `balance` field associated with User A in the database by 500.
6. The server adds 500 to the `balance` field associated with User B in the database.
7. The server sends a confirmation of transaction completion to User A.

Now, there are a number of issues with this transfer method in terms of redundancy, fail-safe, and concurrency. For the sake of this chapter, let’s just evaluate what could go wrong with the math performed on the server in order to finalize this transaction.

First and foremost, while the server checked to ensure that User B existed, it did not check to see if User A actually had the funds required to transfer to User B. A simple missing validation on User A’s balance (checking for sufficient amount to transfer) could have a number of implications.

In the best-case scenario, validations in the database schema would cancel the transaction and error out—despite the application missing the appropriate business logic checks in its code. In the worst-case scenario, User A’s funds stop at $0 while User B’s funds continue to increment by $500 per transaction. This allows User B and User A to collaborate, withdrawing thousands of dollars prior to the bank taking notice.

The preceding example is actually a very simple business logic vulnerability.

An application becomes a fantastic target for a hacker to exploit if it does not implement both of the following requirements:

- The appropriate math to meet its business logic expectations
- The appropriate validations against the inputs into a mathematical function

Such vulnerabilities are most likely not going to be found by any static analysis tooling due to their reliance on the bank's specific business model regarding transfers.

You could describe an attack against this vulnerability as a hacker finding a way to bypass intended functionality while staying in line with the programmed rules.

Programmed Side Effects

Most applications are programmed with a primary use case in mind, which is used for developing the business rules that the application will operate upon. Oftentimes, as the complexity of a web application increases, the application will begin to develop *side effects*, or unintended changes, that occur as a result of using programmed functionality in a way that was not foreseen by the developer.

Consider another web application: MegaCrypto. In this hypothetical web application, MegaBank users are able to buy, sell, and convert cryptocurrencies between each other using the new MegaCrypto web portal. MegaBank predicts this will improve its popularity with younger, more tech-savvy customers.

Like any other exchange, MegaBank knows it needs liquidity to power user purchases, sales, and conversions of cryptocurrencies. It decides to initially offer five major cryptocurrency coins. The prices for which MegaCrypto sells these currencies are to be determined by the market demand. When MegaCrypto was launched, MegaBank made the assumption that local market rates would reflect global market rates for the cryptocurrency coins MegaCrypto decided to offer on its platform. The coins sold and exchanged on MegaCrypto are as follows:

- MegaCoin
- ByteCoin
- DoggoCoin
- GhostCoin
- HardDriveCoin

Because the intended functionality of this application is that market demand sets the rates at which a cryptocurrency can be purchased or exchanged, the current rates are as follows:

- MegaCoin—$1
- ByteCoin—$50,000
- DoggoCoin—$5
- GhostCoin—$2,000
- HardDriveCoin—$10

Each night at midnight, MegaCrypto refills its liquidity pools by purchasing first from its users (if sale orders have been filed) and second from another exchange. MegaCrypto's business logic operates in this way, first repurchasing crypto from its users that are selling because this reduces the fees MegaCrypto would otherwise pay to purchase liquidity from another exchange.

A MegaCrypto customer, Henry Hacker, is able to view public records and notes that MegaCrypto is first purchasing coins back from its users when liquidity is low. In other words, he has reverse engineered the business logic that MegaCrypto uses to maintain liquidity pools. He notes this in his recon journal.

Henry Hacker observes that the total amount of coins being held at any given time by the new exchange is quite low. In other words, at full capacity the liquidity of this exchange is a small percentage of the global cryptocurrency market—in fact, it's a small percentage of what a group of investors could pool together. Just around $1 million total coins are held in liquidity reserves at any given time.

After performing several transactions and watching the new exchange refill its liquidity pools by purchasing from its existing users at market rates, Henry Hacker decides to ask his wealthy friends to pool some money for him to invest. He obtains $1 million in initial capital.

Henry sees that over time the demand for ByteCoin at the MegaCrypto exchange is an average of 20 purchases per day ($1 million at a price of

$50,000/ea). He notes an annual average outflow of 20 ByteCoins from MegaCrypto's liquidity pools to its users' MegaCrypto wallets.

Similarly, each afternoon MegaCrypto creates buy orders for 20 ByteCoins in its local user pool at the local market rate. By buying back coins at night from local users, MegaCrypto ends up with significant earnings from transaction fees, and its users are happy.

Henry decides to write an algorithm that will automatically purchase 20 ByteCoins each night at a rate of $50,000, which matches the global market rate. Immediately afterward, his script will list these ByteCoins for sale at a rate of $100,000 each. After deploying this algorithm to production, he wakes up with $2 million in his account the day after investing only $1 million the day prior.

What Henry did might actually not be illegal, but it would definitely be considered a vulnerability according to MegaCrypto. Without rapid remediation, it could cost MegaCrypto millions of dollars in losses per day.

The issue here is that MegaCrypto's application logic didn't account for the fact that a limited, local liquidity pool might not be sufficiently large enough to mimic global market prices. Furthermore, such a small total liquidity pool allows for very wealthy individuals (or in this case, a group) to manipulate the market up and down.

MegaCrypto is an exchange of digital coins, which means all it had to do was include logic in its application to prevent the purchase of local coins (and go elsewhere), provided no local coins were being offered at global market rates.

What appears on the outside to be an economic issue is actually also a business logic vulnerability. These substantial losses for MegaCrypto could have been avoided with proper edge-case detection that allowed MegaCrypto to only repurchase local coins if the price was within a percentage threshold of the average global market price.

Vulnerabilities like this require deep understanding of complex systems, but once that knowledge is acquired, it may become possible to attack a

web application or set of digital systems in ways that are highly profitable and difficult to detect.

Quasi-Cash Attacks

So far we have looked at business logic vulnerabilities that arise as a result of improper math, as well as business logic vulnerabilities that arise as a result of side effects of intended functionality that were not appropriately factored into the initial application architecture.

However, sometimes primary functionality can be vulnerable to business logic vulnerabilities, especially when the primary functionality deals with multiple interworking systems. In the credit card industry, one of the most common (and expensive) forms of business logic vulnerability is the quasi-cash transaction vulnerability.

Consider the following credit card offered by MegaBank, our example company. This card is called MegaCard, and it has the following attributes:

- 25% APR
- $0 Annual Fee
- 5% cash back on all purchases (for the first three years)

This is the first card being offered by MegaBank to its customers. It is being developed in-house rather than MegaBank selling a product produced by a major established card vendor like Visa or Mastercard.

The assumption is that MegaBank makes so much money from its other business units that the 5% annual cash back will be appropriately funded for the first three years after this card's launch, in order to bring in new customers. It is to be assumed that all competitors offer lower cashback rates, so this rate will drive rapid mass-market adoption of the MegaCard.

Henry Hacker sees the high cashback rate and decides to pick up a MegaCard. Initially he is happy with the very high cashback rate, but after being laid off from EvilCorp, he gets desperate for cash.

While at EvilCorp, he spent quite a bit of time working with APIs for integrating alternative methods of digital payment into EvilCorp's business units. One of these tools was known as PayBuddy. It offered an online portal where vendors could sign up, create invoices, take payments, and direct deposit earnings into a business bank account.

PayBuddy charges only 1% of each transaction, which made it appealing to Henry's previous employer, EvilCorp. While learning how to use the API for PayBuddy, Henry Hacker set up his own merchant account, connected it to his bank account, and made use of it for testing after work.

Shortly after his recent layoff from EvilCorp, Henry came up with an idea:

1. Using his merchant account on PayBuddy, Henry would create an invoice for $1,000 for one minute of consulting.
2. He would than pay the $1,000 invoice with his MegaCard.
3. Now he would have a credit of $1,000 on his MegaCard, but a debit of $990 in his merchant account ($1,000, net 1% fee).
4. Next he would be reimbursed 5% (card reward rate) of the $1,000 purchase, or $50, to his personal bank account.
5. After step 4, Henry would transfer the $990 from PayBuddy to his personal bank account.
6. Finally, Henry's personal bank account would contain $1,040 ($990 from PayBuddy and $50 in rewards from MegaCard), which is $40 more than he started with.

By repeating this sequence of transactions over and over, Henry would continually add $40 dollars in net worth per cycle to his personal bank account. Manually, this would be a slow and tedious process. However, Henry Hacker creates a script that is capable of repeating all of the preceding steps once per minute. With this automation, he is now earning $2,400 in MegaCard rewards *per hour*. In just 24 hours, Henry can earn $57,600 in rewards via this exploit—enough to pay his rent for an entire year.

As a final step, Henry cashes out his balance before MegaBank notices. MegaBank is stuck with the bill because technically Henry didn't violate

any rules in his service agreement.

In this case, the MegaCard worked by design. The card was "secure," enabled 5% cash back on purchases, and was easy to use. What the design of MegaCard did not account for was a use case in which a MegaCard customer would also be a vendor and thus be able to perform rapid, automated purchases against themselves.

This is known as a *quasi-cash transaction vulnerability* because PayBuddy wasn't really a bank account. Instead, it was a financial intermediary that allowed programmatic business transactions that would eventually be settled with real money but, for the time being, were really just fields in a database.

If MegaCard existed by itself, in a more traditional (non-digital) economy, Henry Hacker would likely have performed transactions in an easily predictable way. Buying groceries, going to the movies, or eating out at a restaurant are the activities that MegaBank assumed Henry Hacker would partake in for the purpose of utilizing this new credit card.

Unfortunately, because MegaBank did not include checks and balances in the application logic running on the servers for MegaCard, Henry was able to make tens of thousands of dollars exploiting this business logic vulnerability.

Vulnerable Standards and Conventions

Depending on the application you are targeting, business logic vulnerabilities can range from easy to spot to requiring huge amounts of domain-specific knowledge in order to identify. There are, however, some tricks you can apply to make finding these vulnerabilities a bit easier. These tricks often appear as edge cases arising from common anti-patterns that programmers make use of without understanding the implications. Let's discuss numeric precision loss, which is one such trick.

As previously mentioned, business logic vulnerabilities often involve some form of mathematical operation that is either not handling inputs

correctly or is incorrectly structured and resulting in an unexpected output. If code is executing on a server somewhere that you do not have access to—it may be difficult to correctly guess how the mathematical functions are structured and which edge cases are accounted for.

Luckily, because mathematics is a standardized system, some common errors may lead you down a path towards finding a logic vulnerability. One example of this stems from understanding how most programming languages handle numbers.

Almost every major programming language makes use of the IEEE754 floating-point numeric format (JavaScript, Java, Ruby, Python, etc). This numeric format is used because memory in computers is (or at least was) scarce, leading to the desire for more efficient methods of storing extremely long numbers rather than simply storing every bit in memory.

IEEE754 offers an efficient mechanism of number storage when compared to other formats by using a floating point (variable-location decimal versus fixed-location decimal) and relying on scientific notation to represent numbers. So what's the downside to the use of IEEE754?

You see, in order to enable more memory-efficient number storage, IEEE754 has a trade-off in which it compromises *precision* in exchange for *space*. This means, that in any calculation involving a decimal point, it is possible that the resulting value is *approximated* but not *identical* to the true mathematical result.

Try the following steps:

1. Open up the Chrome web browser.
2. Navigate to the Developer tools (typically the F12 shortcut).
3. Click the Console tab.
4. Execute JavaScript code against Chrome's v8 JavaScript interpreter.
5. Type the following code: `0.1 + 0.2`.
6. Click Enter.

You would expect the JavaScript interpreter in this case to return the correct mathematical result of `0.3`, but instead it returns

`0.30000000000000004`. This occurred because the IEEE754 numeric format can't represent the sum of `0.1 + 0.2`, so instead it approximates the result with *precision loss*.

In this case, the precision loss is `0.0000000000000004`, which is such an extremely small number that in most cases it won't result in any improper computation. However, when dealing with financial calculations—especially rapidly automated calculations—this miscalculation could add up to a significant sum in a short time.

Imagine a financial services application that recalculates user account balances when reloaded. If the `0.0000000000000004` error margin is simply added to the balance, it would take millions of recalculations in order to cause the true balance and recorded balance to differ significantly.

On the other hand, if involved in a more complex calculation where sums are multiplied together or worse, exponentiated, this could get out of hand very quickly, leading to extreme differences in recorded and actual balances. As such, many business logic vulnerabilities arise from improper mathematical calculation. Understanding how mathematical operations are commonly implemented will give you an advantage when looking for these vulnerabilities in a web application.

To summarize, while business logic vulnerabilities are unique to an application's business logic, most applications share similarities in *implementation* that can be used to give insight into where business logic vulnerabilities may live.

In this case, by knowing how programming languages handle floating-point numbers by default, you are given some surface area from which to begin manual testing.

Exploiting Business Logic Vulnerabilities

Now that we have seen a number of business logic vulnerabilities, it should be apparent that logic vulnerabilities are hard to categorize with any specificity, as they are tightly bound to the specific intended use case of a particular application.

When attempting to attack a web application using business logic vulnerabilities, the very first thing you need to do is become intimately familiar with the intended use case for the application. Using the MegaCard example—frame your mind as if you are MegaBank's archetypical MegaCard customer. You want to get groceries, go to the movies, and eat out using the card.

Write down every intended use case you can think of for the application or technology. Then, for each intended use case, make sure to detail how you imagine that works on the backend (in the application logic).

For example, MegaCard notes may look like this:

1. I purchase groceries at FoodCo using my MegaCard.
2. Once I swipe my card, some type of network handshake is performed between MegaBank and FoodCo.
3. MegaBank's databases add a credit equal to the value of the food I purchased against my card, which will later be settled against my bank account when I pay off the balance.
4. MegaBank's server calculates an amount equal to 5% of the food purchase and in a separate transaction, debits it against my personal bank account.

More complex application functionality (like the market manipulation example) will take significantly more time and effort to understand and map out. Once you have mapped out all of the intended functionality and your best guess at how the application works internally on its servers, now you are ready to begin planning attacks.

For each functionality you have recorded, consider edge cases that might not be appropriately addressed. These could look like the following:

- If I purchase $1,000 of groceries in CAD, will I get 5% of that ($50) in USD or will currency conversion take place?
- If I set up an online merchant store with PayBuddy, given the fee difference with the reward (1% versus 5%), will I be able to net 4% on zero-cost transactions?
- If I set up an online merchant account, can I perform a negative invoice of –$1,000, adding money to my credit card balance rather than subtracting it?
- If I purchase and then immediately refund, will I still retain the 5% reward offered?

These edge cases are your initial attack vectors when considering the exploitation of MegaCard via business logic vulnerabilities.

As shown in the prior examples, the most common method of exploiting business logic vulnerabilities is to find a business logic edge case that is not appropriately handled in the application code.

Summary

Business logic vulnerabilities are among the most advanced form of vulnerability to find in a web application, but they are often showcased most frequently by pen testers and bug bounty hunters. These vulnerabilities often have significant impact, and they are difficult or impossible to detect in an automated fashion.

Previously in this book, you learned a number of ways to attack web applications using common and standard forms of attack. But to truly develop mastery and become an expert penetration tester or bug bounty hunter, you need to begin venturing into the realm of discovering business logic vulnerabilities.

Finding business logic vulnerabilities often takes true ingenuity, creativity, and the capacity to think outside of the box. If you can master finding

these vulnerabilities, I believe you will find that your skill set is in extremely high demand.