Bussiness Logic

: 24/08/2022

Aug 23, 2022

9 min read
BL Web-Notes

Business Logic

References

AllAboutBugBounty/Business Logic Errors.md at master · daffainfo/AllAboutBugBounty

Application Logic Bypasses

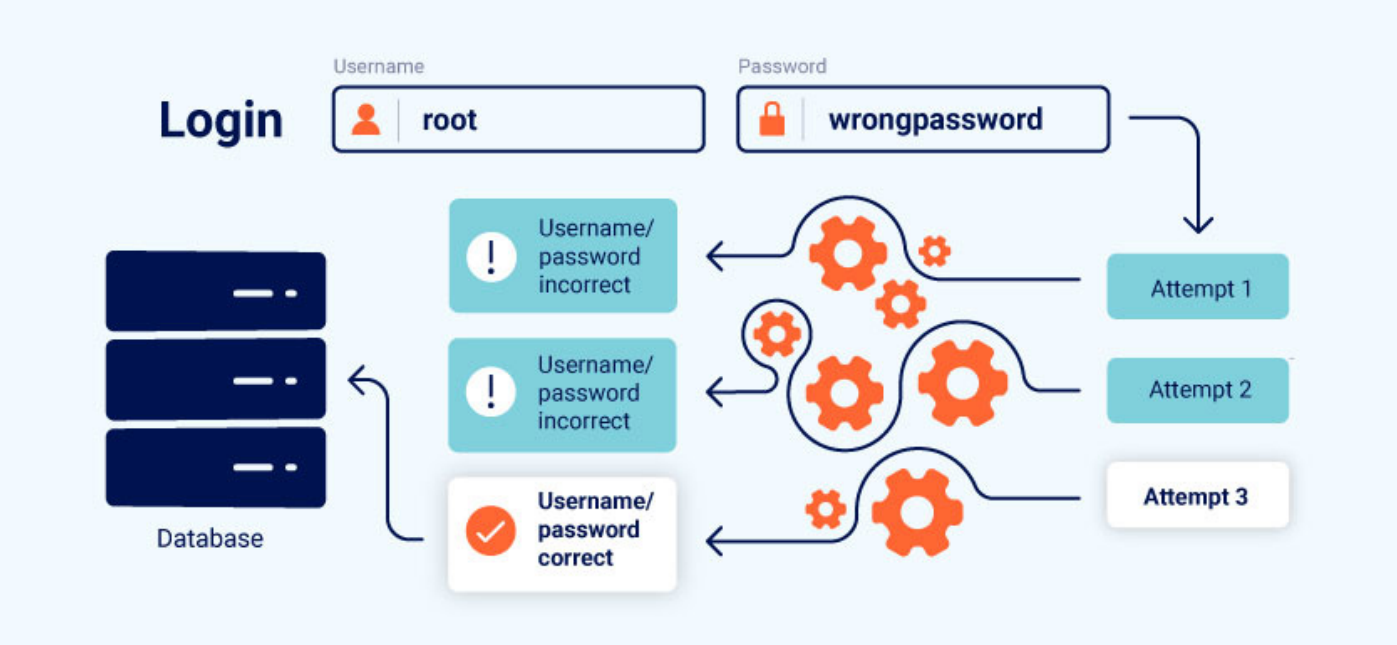
Bug-bounty/bugbounty_checklist.md at master · sehno/Bug-bounty

Business Logic Flaws

GitHub - ngalongc/bug-bounty-reference: Inspired by https://github.com/djadmin/awesome-bug-bounty, a list of bug bounty write-up that is categorized by the bug nature

Business Logic Flaws - Pastebin.com

In this section, we'll introduce the concept of business logic vulnerabilities and explain how they can arise due to flawed assumptions about user behavior.



What are business logic vulnerabilities?

Business logic vulnerabilities are **flaws in the design and implementation** of an application that allow an attacker to elicit **unintended behavior**.

This enable the attacker to make legel activity to achieve malicious goal.

> In this context, the term “business logic” simply refers to the set of rules that define **how the application operates**. As these rules aren’t always directly related to a business, the associated vulnerabilities are also known as “**application logic vulnerabilities**” or simply “**logic flaws**”.

Logic Flaws are not visiable to the one who didn’t look for, However the attacker can exploit the application by a way the developer didn’t intened.

The business rules dictate how the application should **react when a given scenario occurs**. This includes preventing users from doing things that will have a negative impact on the business or that simply don’t make sense.

By passing unexpected values into server-side logic, an attacker can potentially induce the application to **do something that it isn’t supposed to**.

How do business logic vulnerabilities arise?

Business logic vulnerabilities often arise because the design and development teams make flawed assumptions about **how users will interact with the application**.

For example, if the developers assume that users will pass data exclusively **via a web browser**, the application may rely entirely on weak client-side controls to validate input.

These are easily **bypassed** by an attacker using an **intercepting proxy**.

Ultimately, this means that when an attacker deviates from the **expected user behavior**, the application fails to take appropriate steps to prevent this and, subsequently, **fails to handle the situation safely**.

To avoid these vulnerabilities, developers must be aware about every function and functions that combined with unexpected way.

What is the impact of business logic vulnerabilities?

Fundamentally, the impact of any logic flaw depends on **what functionality it is related to**. If the flaw is in the authentication mechanism, for example, this could have a serious impact on your overall security.

Attackers could potentially exploit this for privilege escalation, or to **bypass authentication entirely**, gaining access to sensitive data and functionality. This also exposes an increased attack surface for other exploits.

What are some examples of business logic vulnerabilities?

Excessive trust in client-side controls

A fundamentally flawed assumption is that users will only interact with the application via the provided **web interface**.

However, an attacker can simply use tools such as **Burp Proxy** to tamper with the data after it has been sent by the browser but before it is passed into the **server-side logic**.

Accepting data at face value, without performing proper **integrity checks and server-side validation**, can allow an attacker to do all kinds of damage with relatively minimal effort.

Lab: Excessive trust in client-side controls Web Security Academy

Lab: 2FA broken logic Web Security Academy

Failing to handle unconventional input

For example, the application may be designed to accept arbitrary values of a certain data type, but the **logic determines whether or not this value is acceptable** from the perspective of the business. Many applications incorporate numeric limits into their logic. This might include limits designed to manage inventory, apply budgetary restrictions, trigger phases of the supply chain, and so on.

Let's take the simple example of an online shop. When ordering products, users typically specify the **quantity that they want to order**. Although any integer is theoretically a valid input, the business logic might prevent users from ordering more units than are currently in stock.

Consider a funds transfer between two bank accounts. This functionality will almost certainly check whether the sender has **sufficient funds** before completing the transfer:

```
$transferAmount = $_POST['amount'];
$currentBalance = $user->getBalance();

if ($transferAmount <= $currentBalance) {
    // Complete the transfer
} else {
    // Block the transfer: insufficient funds
}
```

But if the logic doesn't sufficiently prevent users from supplying a **negative value** in the `amount` parameter, this could be exploited by an attacker to both bypass the balance check and transfer funds in the "wrong" direction.

When auditing an application, you should use tools such as **Burp Proxy** and **Repeater** to try submitting unconventional values. In particular, try input in ranges that legitimate users are unlikely to ever enter. This includes exceptionally high or exceptionally low numeric inputs and abnormally long strings for text-

based fields. You can even try **unexpected data types**. By observing the application's response, you should try and answer the following questions:

- Are there any limits that are imposed on the data?
- What happens when you reach those limits?
- Is any transformation or normalization being performed on your input?

Lab: High-level logic vulnerability Web Security Academy

Lab: Low-level logic flaw Web Security Academy

Lab: Inconsistent handling of exceptional input Web Security Academy

---

Application Logic Errors

For example, let's say an application implements a `three-step login process`. First, the application checks the `user's password`. Then, it sends an `MFA code` to the user and verifies it. Finally, the application asks a `security question` before logging in the user:

A normal authentication flow would look like this:

1. The user visits https://example.com/login/. The application prompts the user for their `password`, and the user enters it.
2. If the password is correctly entered, the application `sends an MFA code` to the user's email address and redirects the user to https://example.com/mfa/. Here, the user enters the MFA code.
3. The application checks the MFA code, and if it is correct, redirects the user to https://example.com/security_questions/. There, the application asks the user several `security questions` and logs in the user if the answers they provided are correct.

While the `vulnerable application` redirects users to step 3 after the completion of step 2, it doesn't verify that `step 2 is completed before` users are allowed to advance to step 3.

If attackers can directly access https://example.com/security_questions/, they could `bypass the multifactor authentication` entirely.

Another example :

Let's say an online shop allows users to pay via a `saved payment method`.

When users `save a new payment method`, the site will verify whether the credit card is valid and current. That way, when the user submits an order via a saved payment method, `the application won't have to verify it again`.

Say that the POST request to submit the order with a saved payment method looks like this :

```
POST /new_order
Host: shop.example.com
```

```
(POST request body)
item_id=123
&quantity=1
&saved_card=1
&payment_id=1 //payment_id parameter refers to the ID of  the user's saved
credit card
```

If users pay with a `new credit card`, the card will be `verified at the time of checkout`. Say the POST request to submit the order with a new payment method looks like this:

```
POST /new_order
Host: shop.example.com
(POST request body)
item_id=123
&quantity=1
&card_number=1234-1234-1234-1234
```

So a malicious user can submit a request with a `saved_card parameter` and a `fake credit card number`. Because of this error in payment verification, they could order unlimited items for free with the unverified card:

```
POST /new_order
Host: shop.example.com
(POST request body)
item_id=123
&quantity=1
&saved_card=1
&card_number=0000-0000-0000-0000
```

Hunting for Application Logic Errors

Step 1: Learn About Your Target

Browse the application as a regular user to uncover functionalities and interesting features.

You can also read the application's engineering blogs and documentation.

For example, if you find out that the application just added a `new payment option` for its online store, you can test that payment option first since new features are often the least tested by other hackers.

And if you find out that the application uses WordPress, you should try to access `/wp-admin/admin.php`, the default path for WordPress admin portals.

Step 2: Intercept Requests While Browsing

`Intercept requests` while browsing the site and pay attention to sensitive functionalities.

Take note of how sensitive functionalities and access control are `implemented`, and how they interact with `client requests`.

For the new payment option you found, what are the requests needed to `complete the payment`? Do any request parameters indicate the payment type or how much will be charged? When accessing the admin portal at /wp-admin/ admin.php, are any special HTTP headers or parameters sent?

Step 3: Think Outside the Box

Finally, use your creativity to think of ways to bypass access control or otherwise interfere with application logic.

Play with the requests that you have intercepted and craft requests that should not be granted.

Finding Your First Application Logic Error or Broken Access Control!

1. Learn about your target application. The more you understand about the `architecture and development process` of the web application, the better you'll be at spotting these vulnerabilities.
2. `Intercept requests` while browsing the site and pay attention to sensitive functionalities. Keep track of every request sent during these actions.
3. Use your creativity to think of ways to `bypass access control` or otherwise interfere with application logic.
4. Think of ways to combine the vulnerability you've found with other vulnerabilities to `maximize the potential impact` of the flaw.
5. Draft your report! Be sure to communicate to the receiver of the report how the issue could be exploited by malicious users.

`Prevention`

- You'll need a detailed understanding of how your application works, how users interact with each other, how functionalities are carried out, and how complex processes work.
- Carefully review each process for any logical flaws that might lead to a security issue. Conduct rigorous and routine testing against each functionality that is critical to the application's security.

How to prevent business logic vulnerabilities

In short, the keys to preventing business logic vulnerabilities are to:

- Make sure developers and testers understand the domain that the application serves

- Avoid making implicit assumptions about user behavior or the behavior of other parts of the application
- Maintain clear design documents and data flows for all transactions and workflows, noting any assumptions that are made at each stage.
- Write code as clearly as possible. If it’s difficult to understand what is supposed to happen, it will be difficult to spot any logic flaws. Ideally, well-written code shouldn’t need documentation to understand it. In unavoidably complex cases, producing clear documentation is crucial to ensure that other developers and testers know what assumptions are being made and exactly what the expected behavior is.
- Note any references to other code that uses each component. Think about any side-effects of these dependencies if a malicious party were to manipulate them in an unusual way.****