Exploiting business logic error vulnerabilities

BY AYOUB · DECEMBER 9, 2025 · LAST UPDATED ON JANUARY 7, 2026

It's no secret that complexity is the biggest rival of safe applications. As web apps become more sophisticated, they create countless opportunities for logic flaws to arise. Unlike technical vulnerabilities that can be easily automated, business logic errors emerge from the gap between how developers expect systems to behave and how attackers can manipulate them.

In this article, we explore how to identify and exploit business logic flaws to bypass restrictions, escalate privileges, and introduce exploitable behavior.

Let's dive in!

What are business logic error vulnerabilities

Business logic vulnerabilities occur when an application's workflow can be exploited in ways that violate intended business rules, even though the code functions exactly as written. They arise from the gap between implementation and actual behavior, when developers make assumptions about how users will behave but fail to account for edge cases, or don't consistently validate state transitions across complex workflows.

However, it is essential to note that not all undocumented behavior necessarily results in a security vulnerability. Therefore, it is crucial to understand when the identified logic error poses a security threat to the organization, and when it doesn't, especially when you're participating in bug bounty programs.

For that, we must have a look at the 3 severity scoring metrics and ask ourselves if the identified behavior has an impact on any of them:

1. **Confidentiality:** Does the identified logic flaw allow for viewing confidential information that is otherwise non-accessible (e.g. being able to change the numerical order ID during checkout to gain access to sensitive information of other orders)?
2. **Integrity:** Does the identified logic flaw allow altering data that is normally protected (e.g. being able to assign your account to another tenant by exploiting a quirky logic flaw within the invitation functionality)?
3. **Availability:** Can you partly or completely deny access to a single component, or even the entire application, rendering it unusable (e.g. creating an invalid record within a tenant that would continuously crash the application and make the platform unavailable)?

Answering these questions will help determine whether you're dealing with an actual security weakness or a simple, functional bug. Have a look at a few more practical examples:

- **A complex access matrix.** Permission profiles are essential in enterprise applications. The necessity for granular permission controls per component and user severely impacts complexity, usually

resulting in access control vulnerabilities when complex permission profiles are in use.

- **Incomplete validation of business rules.** A refund system might check if an order exists, but not whether it's already been refunded, allowing for duplicate refunds. Although human involvement is essential for such sensitive transactional operations, organizations are sometimes forced to rely entirely on automated systems, which opens the possibility for financial fraud at scale.
- **Trusting client-side data.** Applications that rely on user-provided prices, currency, quantities, or user IDs without server-side verification open themselves to potential manipulation. The lack of server-side validation can also allow for injection attacks such as (No)SQLi, XXE, SSRF and XSS.
- **Race conditions in critical operations.** Two simultaneous withdrawal requests might both succeed if the account balance check isn't properly locked, allowing for overdrafts.
- **Flawed state management.** Multi-step processes such as checkout flows may allow users to skip steps, replay steps, or access states they shouldn't reach, potentially enabling the completion of a purchase without payment, for example.

Examples of non-exploitable behavior

As mentioned previously, not all logic flaws are considered security weaknesses. Some merely exhibit odd behavior that cannot be further escalated and are often functional bugs that have not been identified during internal QA tests.

Let's have a look at a few practical examples which cannot be considered to pose a security threat to the organization:

- **Modifying data that is solely yours, despite the application's front-end preventing it from being altered.** This behavior cannot always constitute a security weakness. Even when you manage to modify the restricted input fields via a proxy intercepting tool, the impact is still limited. However, it's an opportunity for us to check for inconsistent input validation that could allow for injection attacks, such as XSS or SQLi.
- **Incorrect calculations that don't affect authorization or money.** A fitness app that helps you to provide arbitrary values, such as calories burned or an e-learning platform with a personal scoring system whose calculations can be arbitrarily altered.
- **Performance or efficiency issues.** The ability to create an unlimited number of data objects with its sole impact of (slightly) degrading performance on the client-side.

The previously mentioned examples are all instances of logical flaws that do not have an actual security impact. It is essential to note that some organizations may want to be aware of such flaws, while others will acknowledge them as accepted risks.

Exploitable logic flaws vs Non-exploitable behavior

To accurately distinguish exploitable from non-exploitable logic flaws, you must at all times ask in what way an actual attacker can exploit the identified behavior.

For instance, can you incur more privileges than before? Does the identified behavior allow for injection attacks (e.g. due to inconsistent input validation)? Is it possible to perform actions that are sensitive by nature and prohibited by default (e.g. requesting multiple refunds for a single item)? Can you deny access to a certain resource or application (e.g. via an in-app denial of service)?

Exploiting business logic errors

Now that we have a basic understanding of what logic flaws are (and what cannot be considered as such), we proceed to the exploitation of these weaknesses.

Logic flaws that allow for broken access controls

Permission profiles are not that uncommon in web applications. They provide for an administrator account to set permissions for users or groups within a single tenant. However, the more granular permissions are defined, the more complex it gets. This can lead to broken access control issues arising, especially when multiple permission rules are enforced on a single user or user group.

Another example is the failure to enforce access controls when unintended behavior is introduced. Take the following code snippet into consideration. Can you seem to spot the issue?

```
app.get('/api/checkout/confirm', (req, res) => {
    const orderId = req.session.orderId;

    // Check if payment was finalized
    if (!finalizePayment(orderId)) {
        return res.status(400).json({ error: 'Payment not finalized' });
    }

    // Redirect customer to order confirmation page
    db.query('SELECT * FROM orders WHERE id = ?', [orderId], (err, results) => {
        if (err) return res.status(500).json({ error: 'Database error' });

        const orderData = results[0];

        res.json({
            orderId: orderData.id,
            customerName: orderData.customer_name,
            customerEmail: orderData.customer_email,
            items: orderData.items,
            totalAmount: orderData.total_amount,
            shippingAddress: orderData.shipping_address
        });
    });
});

app.post('/api/checkout/initiate', (req, res) => {
    const { cart, order_initiation_id } = req.body;

    if (order_initiation_id) {
        req.session.orderId = order_initiation_id;
    } else {
        req.session.orderId = requestNewOrderId();
    }

    // Initiate checkout session
    initiateCheckout(req.session.orderId, cart);

    res.json({
        status: 'success',
        message: 'Order initiated',
        orderId: order_initiation_id
    });
});
```

Logic flaw vulnerability that allows for broken access controls

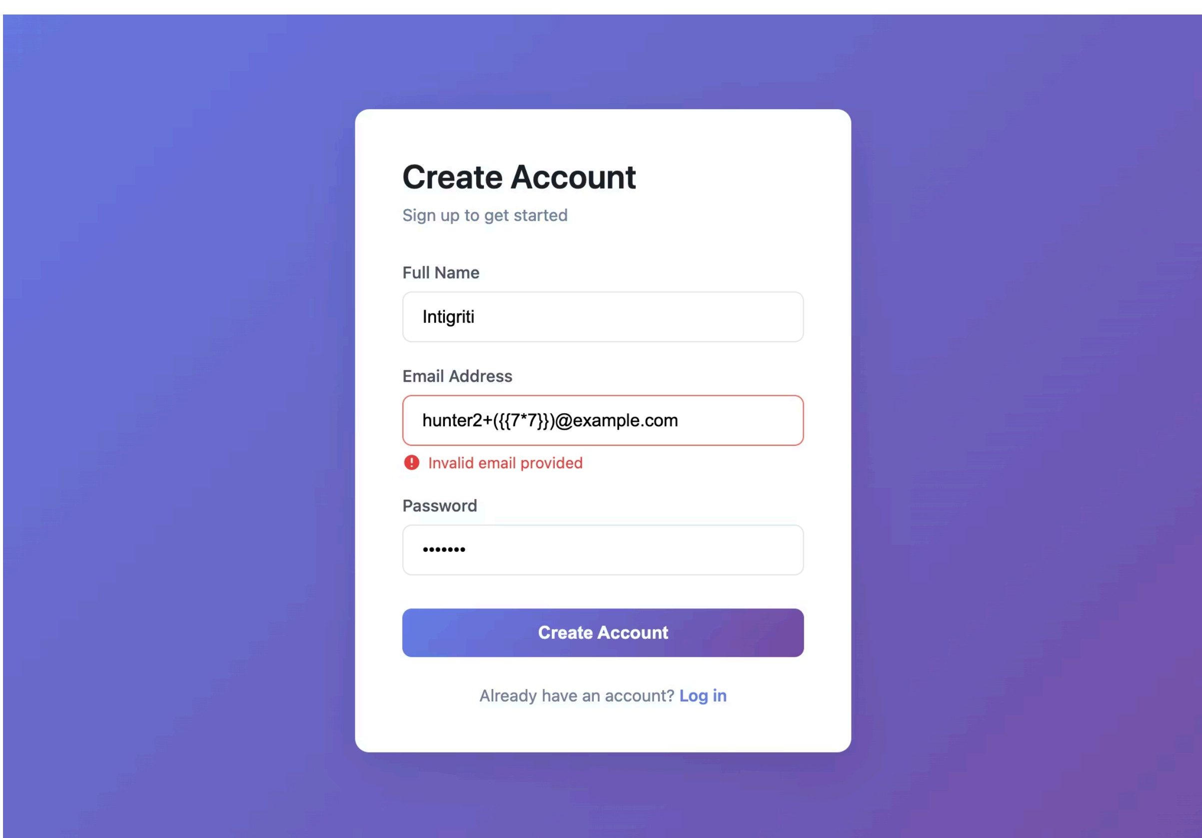
In the code snippet above, we can see that the checkout confirmation route retrieves order information based on the order ID within the session object. If we have a closer look at the previous step in the checkout process, we can notice that the order ID is read from the order_initiation_id body parameter. The issue here is that the developers never considered validating this order ID to ensure it is unique, non-existent, and most importantly, belongs to the current customer.

With this in mind, we can set the `order_initiation_id` to any value we like, finalize the checkout, and possibly access other customers' confidential data.

Logic flaws that allow for injection attacks

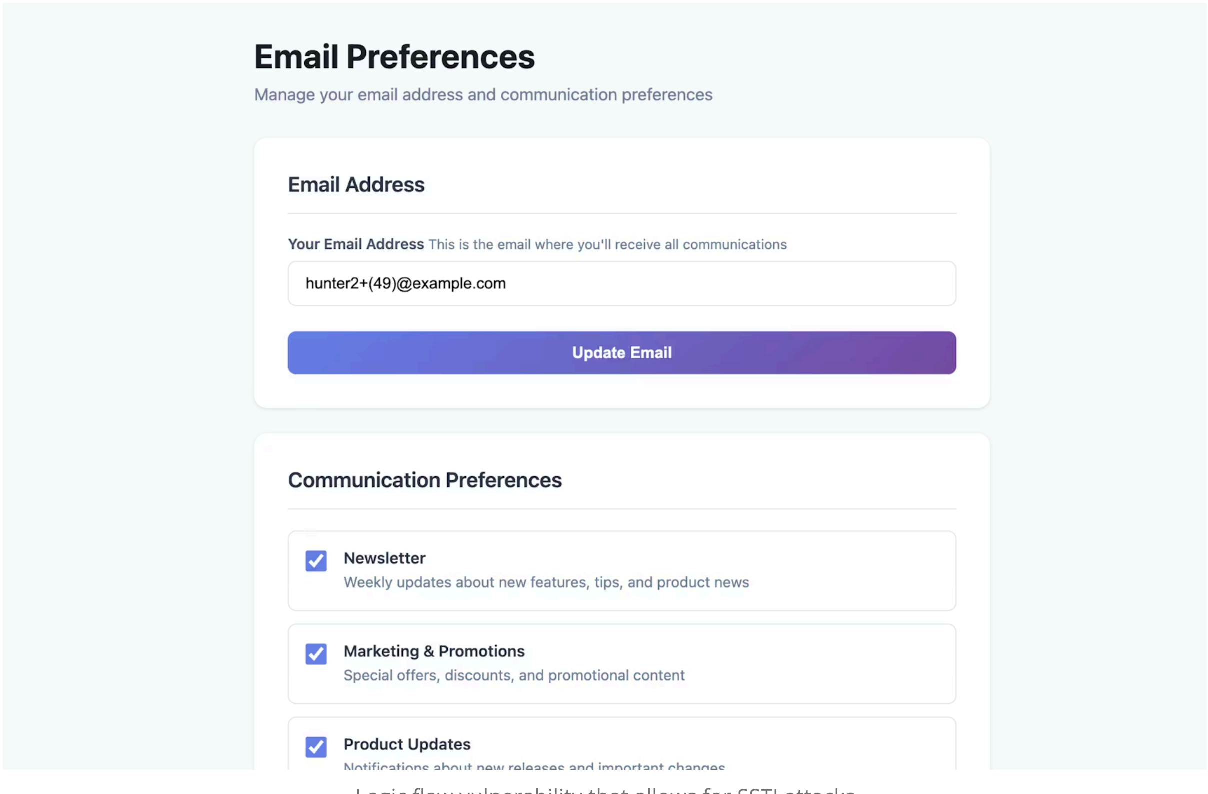
Inconsistent input validation often leads to injection attacks. With this in mind, you should be on the lookout for places where input validation is limited or non-existent. Take the following example into consideration.

The application below was designed to disallow malformed email addresses that contain XSS, SSTI or SQLi payloads.



Logic flaw vulnerability that allows for injection attacks

However, after signing up, you decide to unsubscribe from all marketing emails, where you're presented with a new portal that allows you to configure all your email preferences, including changing your email. If this same portal syncs changes with the main application, it may not validate your input at all and allow you to change your email to include a payload, similar to our previous attempt:



Logic flaw vulnerability that allows for SSTI attacks

This lack of consistent input validation can allow for injection attacks, as we've observed in the example above.

Logic flaws that allow for price manipulation vulnerabilities

Similar to injection attacks, logic flaws can also result in price manipulation vulnerabilities, primarily due to incorrect handling of user parameters. Have a look at the example below. Can you identify the weakness?

```
1  app.post('/api/checkout', async (req, res) => {
2      const { items, currency } = req.body;
3      const userId = req.session.userId;
4
5      // Calculate total price from items
6      let totalPrice = 0;
7      for (const item of items) {
8          const product = await db.query(
9              'SELECT price FROM products WHERE id = ?',
10             [item.productId]
11         );
12         totalPrice += product[0].price * item.quantity;
13     }
14
15     const currencyCode = currency || 'USD';
16
17     // Create order record
18     const order = await db.query(
19         'INSERT INTO orders (user_id, total_amount, currency) VALUES (?, ?, ?)',
20         [userId, totalPrice, currencyCode]
21     );
22
23     // Initiate new transaction request with payment gateway
24     const paymentRequest = {
25         orderId: order.insertId,
26         amount: totalPrice,
27         currency: currencyCode,
28         merchantId: process.env.MERCHANT_ID
29     };
30
31     const paymentResponse = await paymentGateway.processPayment(paymentRequest);
32     if (paymentResponse.status != 'finalized') {
33         return res.status(400).json({ error: 'Something went wrong during payment.' });
34     }
35
36     res.json({
37         success: true,
38         orderId: order.insertId,
39         paymentStatus: paymentResponse.status,
40         amount: totalPrice,
41         currency: currencyCode
42     });
43 });
```

Logic flaw vulnerability that allows for price manipulation

In the code snippet above, we can notice that on **line 15**, the currency code is read from a user-controllable source. However, upon closer examination of the code, we can observe that the code responsible for calculating the checkout price does not account for this. Instead, it forwards this value, without further validation, to the third-party payment gateway to finalize the checkout.

Practically, this means that we can adjust the currency code from `USD` to `JPY` and lower our checkout price.

```
POST /api/checkout HTTP/2
Host: checkout.example.com
Content-Length: 122
Content-Type: application/json
Cookie: session=eyJ...
Connection: keep-alive

{
    "items": [
        {
            "productId": 42,
            "quantity": 1
        }
    ],
    "currency": "JPY"
}
```

```
HTTP/1.1 200 OK
Date: Tue, 09 Dec 2025 13:37:00 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 124
Connection: keep-alive
Location: /account
Vary: Cookie
Set-Cookie: session=eyJ...; HttpOnly; Path=/
Strict-Transport-Security: max-age=31536000; includeSubDomains

{
    "success": true,
    "orderId": 78291,
    "paymentStatus": "completed",
    "amount": 299.99,
    "currency": "JPY"
}
```

Currency confusion

The aforementioned flaw stems from a simple oversight that could have enabled malicious buyers to purchase sold goods at highly discounted prices.

Logic errors that allow for cryptographic flaws

Logic errors can also enable cryptographic flaws, for instance, due to weak random generation or unsafe storage of sensitive tokens. Incorrect handling can also open up security gaps, allowing attackers to predict or generate entirely new tokens.

One example of such flaws is support for the none algorithm in JSON Web Token implementations. Let's have a closer look at an example.

```python
import os
import jwt
from datetime import datetime, timedelta

SECRET_KEY = os.environ.get('SECRET_KEY')

def create_token(user_data, algorithm='HS256'):
    """Create JWT token for user"""
    payload = {
        'user_id': user_data['id'],
        'username': user_data['username'],
        'role': user_data['role'],
        'exp': datetime.utcnow() + timedelta(hours=24)
    }

    token = jwt.encode(payload, SECRET_KEY, algorithm=algorithm)
    return token

def verify_token(token):
    """Verify and decode JWT token"""
    try:
        header = jwt.get_unverified_header(token)

        # @TODO: Add validation for session tokens! We currently
        # need this for a new feature to allow sharing cart items (still WIP)
        if header.get('alg') == 'none':
            decoded = jwt.decode(
                token,
                options={"verify_signature": False} # Insensitive operation so we don't need to verify the signature
            )
            return decoded
        else:
            decoded = jwt.decode(
                token,
                SECRET_KEY,
                algorithms=['HS256']
            )
            return decoded
    except jwt.ExpiredSignatureError:
        return None
    except jwt.InvalidTokenError:
        return None
    except Exception:
        return None
```

JWT None algorithm attack

On **line 27**, you can closely observe how the developer fails to account for the scenario whereby an attacker sends an unsigned JWT token to an authenticated endpoint. The lack of proper handling stems from an oversight, which in this instance, resulted in a full authentication bypass.

To test for such cases, you must attempt to understand common implementation routes developers take to develop certain functionalities within applications.

Conclusion

Business logic flaws are often undermined as they demand a certain level of experience and knowledge about the target from the researcher, which is rarely provided. In this article, we've explored the importance of testing for logic flaws and how this weakness can result in impactful vulnerabilities.

So, you've just learned something new about business logic flaws… Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs and CTFs or… browse through our [70+ public bug bounty programs on Intigriti](), and who knows, maybe earn a bounty on your next submission!

AUTHOR

Ayoub

Senior security content developer

REQUEST A DEMO	VISIT THE WEBSITE	GET IN TOUCH
intigriti.com/demo	intigriti.com	hello@intigriti.com