Business Logic Vulnerabilities Notes

Eyad Islam El-Taher

February 8, 2026

Introduction

Business logic vulnerabilities are flaws in the design and implementation of an application that allow attackers to elicit unintended behavior by manipulating legitimate functionality. These vulnerabilities result from failing to anticipate unusual application states and subsequently failing to handle them safely.

Note: The term "business logic" refers to the set of rules defining how an application operates. Since these rules aren't always directly business-related, these vulnerabilities are also known as "application logic vulnerabilities" or "logic flaws."

Logic flaws are often invisible during normal application use but can be exposed when attackers interact with applications in ways developers never intended. Business logic's primary purpose is to enforce rules and constraints that prevent users from performing actions with negative business impact or that don't make sense.

Business Logic Vulnerabilities Happen When

1. **Flawed Assumptions:** Development teams make incorrect assumptions about how users will interact with the application
   - Assuming users only interact via web browsers
   - Relying on weak client-side controls for validation
   - Failing to anticipate edge cases and unusual input
2. **Inadequate Input Validation:** Insufficient or improper validation of user-supplied data
   - Missing server-side validation
   - Over-reliance on client-side controls
   - Failure to validate transaction-critical values
3. **Complex System Oversight:** In overly complicated systems that even developers don't fully understand
   - Unclear interactions between different components
   - Undocumented assumptions about system behavior
4. **Unusual Workflow Manipulation:** When users deviate from expected behavior patterns
   - Circumventing intended workflows
   - Combining functions in unexpected ways
   - Exploiting gaps in state management

Business Logic Vulnerabilities Happen Where

Logic vulnerabilities can manifest in various parts of an application:

- **Authentication Mechanisms:**
  - Registration processes
  - Login workflows
  - Password reset functionality
  - Multi-factor authentication

- **Authorization Controls:**
  - Access control checks
  - Privilege escalation paths
  - Role-based access control (RBAC) implementations
- **Transaction Processing:**
  - Payment workflows
  - Shopping cart functionality
  - Discount and coupon applications
  - Inventory management
- **User Input Processing:**
  - Form submissions
  - File uploads
  - API parameter handling
  - Configuration changes
- **State Management:**
  - Session handling
  - Workflow state transitions
  - Multi-step processes
  - Concurrent operation handling
- **Business Rules Enforcement:**
  - Business policy implementations
  - Compliance rule enforcement
  - Validation rule sets
  - Constraint checking

Impact of Business Logic Vulnerabilities

The impact of business logic vulnerabilities varies significantly based on the affected functionality:

- **Authentication Bypass:**
- **Financial Impact:**
- **Data Integrity:**
- **Business Process Disruption:**
- **Unauthorized Actions:**
- **Resource Abuse:**
- **Information Disclosure:**

Detection Challenges

Business logic vulnerabilities present unique detection challenges:

- **Human Knowledge Requirement:** Understanding of business domain and attacker motivations
- **Automation Resistance:** Difficult to detect with automated vulnerability scanners
- **Context Dependence:** Highly specific to application functionality and business rules
- **Subtle Manifestations:** Often appear as minor behavioral anomalies rather than clear errors

These characteristics make business logic vulnerabilities prime targets for manual security testing, bug bounty programs, and thorough code reviews.

Examples of Business Logic Vulnerabilities

Excessive Trust in Client-Side Controls

A fundamental vulnerability occurs when developers assume users only interact through web interfaces and rely on client-side validation for security.

Vulnerability Pattern

- **Assumption:** Users only use browser interface
- **Flaw:** Relying on client-side validation as security
- **Reality:** Attackers can intercept/modify requests (Burp Proxy)
- **Result:** Client-side controls become useless

Practical Example: Price Manipulation

Scenario

E-commerce site with store credit system. Client-side JavaScript prevents purchases exceeding credit balance.

Exploitation Steps

1. Attempt purchase → Rejected (insufficient credit)
2. Intercept `POST /cart` request in Burp
3. Modify `price` parameter to arbitrary value
4. Set price below available credit
5. Complete purchase with manipulated price

```
# Original:
POST /cart HTTP/1.1
item_id=123&price=100&quantity=1

# Modified:
POST /cart HTTP/1.1
item_id=123&price=1&quantity=1
```

Failing to Handle Unconventional Input

Overview

Applications often fail to properly handle unexpected or unconventional user input, leading to business logic vulnerabilities. Developers frequently anticipate only "normal" user behavior without considering edge cases or malicious input scenarios.

Common Vulnerable Scenarios

- **Numeric Input Issues:**
  - Negative values where only positives expected
  - Zero values causing division errors
  - Extremely large numbers causing overflow
  - Decimal values where integers expected
- **String Input Issues:**
  - Exceptionally long strings
  - Special characters and scripts

  - Whitespace variations
  - Unicode and encoding issues
- **Type Confusion:**
  - Strings where numbers expected
  - Arrays where strings expected
  - Boolean values misinterpreted
  - Null values not handled

Practical Example: Negative Amount Transfer

Vulnerability Description

A banking application transfers funds between accounts. The logic checks if sender has sufficient funds but fails to validate that the transfer amount is positive.

Attack Scenario

- Attacker sends $-1000 to victim
- Logic: $-1000 <= $balance (always true)
- Result: Attacker receives $1000 from victim
- Direction: Funds transfer in "wrong" direction

Testing Methodology

Numeric Field Testing

- **Extreme Values:**
  - Very large positive numbers (999999999)
  - Very large negative numbers (-999999999)
  - Zero (0)
  - Decimal values (0.01, 100.99)

String Field Testing

1. **Length Testing:**
   - Empty strings
   - Very long strings (10,000+ characters)
2. **Character Testing:**
   - Special characters (@, #, $, %, etc.)
   - Unicode characters
   - Control characters
   - Script tags

Key Questions During Testing

- Are input limits properly enforced?
- What happens when limits are reached?
- Is input normalized/transformed?
- Are business rules validated server-side?
- How does the application handle invalid types?

Example 1: Negative Quantity Price Manipulation

Vulnerability

Cart system fails to validate that item quantities must be positive numbers, allowing negative quantities that create negative total prices.

Exploitation Steps

1. Add cheap item to cart normally
2. Intercept POST /cart request via Burp
3. Change quantity parameter to negative value
4. Negative quantity deducts from cart total
5. Add expensive item
6. Use negative quantity of cheap item to offset total price
7. Purchase expensive item for reduced/negative total

Key Finding

- Negative quantities accepted without validation
- Total price can become negative
- Business logic assumes quantities ≥ 1
- No server-side validation of quantity sign

Example 2: Integer Overflow Price Bypass

Vulnerability

Price calculation uses 32-bit integers that overflow when total exceeds maximum value, causing price to wrap to negative values.

Exploitation Steps

1. Leather jacket price: 133700 cents
2. Maximum 32-bit integer: 2,147,483,647
3. Calculate units needed: 2,147,483,647 $\div$ 133,700 $\simeq$ 16,057 units
4. Use Burp Intruder to add 99 units repeatedly
5. Total price exceeds max $\rightarrow$ wraps to negative (Integer overflow)
6. Adjust quantities to settle total between \$0-\$100
7. Complete purchase with manipulated price

Key Finding

- Integer overflow in price calculation
- Negative totals from overflow wrap-around
- No bounds checking on cumulative totals
- Business logic assumes reasonable quantities

Example 3: Email Truncation Admin Bypass

Vulnerability

Email addresses truncated to 255 characters during storage, allowing domain spoofing via crafted long addresses.

Exploitation Steps

1. Discover /admin endpoint requires @dontwannacry.com email
2. Register with 255+ character email:
   ```
   very-long-string@dontwannacry.com.mail.net
   ```
3. Position the "m" that is last character of "@dontwannacry.com" at character 255
4. Email client receives full address (no truncation)
5. Application stores truncated version (255 chars) –> "very-long-string@dontwannacry.com" only
6. Truncated address appears as @dontwannacry.com so Gain admin access

Key Finding

- Email truncation without validation
- Different truncation points in system components
- Business logic relies on email domain for authorization
- No verification of complete email address integrity

Making Flawed Assumptions About User Behavior

Overview

A common root cause of business logic vulnerabilities is developers making incorrect assumptions about how users will behave. These flawed assumptions lead to security gaps when users act in unexpected or malicious ways.

Assumption: Trusted Users Remain Trustworthy

Applications may implement strict initial controls but then assume users remain trustworthy after passing them. This leads to lax enforcement later in the user journey.

Example: Email Change Admin Bypass

Scenario

Admin panel requires @dontwannacry.com email addresses. System validates email during registration but not during subsequent changes.

Vulnerability

- Initial registration validates email domain
- Email change feature lacks same validation
- Once authenticated, users are "trusted"
- No re-validation of admin privileges

Exploitation Steps

1. Discover /admin requires @dontwannacry.com email
2. Register with normal email address and complete confirmation process
3. Log into account and navigate to "My account" → Change email
4. Change to @dontwannacry.com address
5. Gain admin access without validation

Users Won't Always Supply Mandatory Input

Overview

Developers often assume users will always supply values for mandatory input fields. While browsers prevent normal form submission without required fields, attackers can manipulate parameters in transit, including removing them entirely.

Vulnerability Pattern

When multiple functions share the same server-side script, the presence/absence of parameters determines which code path executes. Removing parameters can expose restricted functionality.

Example: Password Change Without Verification

Vulnerability

Password change functionality requires current password, but server-side logic fails when parameter is missing.

Exploitation Steps

1. Intercept POST /my-account/change-password request
2. Remove current-password parameter entirely
3. Change username parameter to administrator
4. Submit request without current password
5. Password successfully changed for admin account
6. Log in as administrator with new password

Users Won't Always Follow the Intended Sequence

Overview

Applications often assume users will follow predefined step-by-step workflows, but attackers can skip steps or access endpoints out of order, leading to security bypasses.

Example: 2FA Step Bypass

Vulnerability

Two-factor authentication requires login → verification code entry sequence, but doesn't verify completion before granting access.

Exploitation Steps

1. Log into own account, note /my-account URL
2. Log out
3. Log into victim's account with credentials
4. When prompted for verification code, manually navigate to /my-account
5. Access granted without completing 2FA step

Key Finding

- Application doesn't track authentication state between steps
- Step completion not verified before granting access
- Assumes users will complete all steps in order

Common Vulnerable Workflows

- **Multi-Step Authentication:** Login → 2FA → Access
- **Purchase Flows:** Cart → Shipping → Payment → Confirmation
- **Registration Processes:** Signup → Verification → Profile setup
- **Administrative Actions:** Approval → Review → Execution
- **File Uploads:** Selection → Review → Upload → Processing

Assumptions About Sequence of Events

Overview

Developers often assume users will follow applications through predictable state sequences. Attackers can use forced browsing to interact with endpoints in any order, accessing application states that were never intended.

Tool Capabilities

- Burp Proxy: Intercept and analyze requests
- Burp Repeater: Replay requests arbitrarily
- Forced Browsing: Access endpoints in any sequence
- State Manipulation: Interact with application in unexpected states

Example 1: Order Confirmation Bypass

Vulnerability

Purchase workflow: Add items → Checkout → Payment → Confirmation. System doesn't verify payment completion before showing confirmation.

Exploitation Steps

1. Buy affordable item normally
2. Capture order confirmation URL: GET /cart/order-confirmation?order-confirmation=true
3. Add expensive leather jacket to cart
4. Directly access order confirmation URL
5. Order completes without payment deduction

Example 2: Role Selection Bypass

Vulnerability

Login workflow: Credentials → Role selection → Home page. System defaults to admin role if selection skipped.

Exploitation Steps

1. Intercept login process
2. Forward POST /login request
3. Drop GET /role-selector request
4. Directly browse to home page
5. Role defaults to administrator
6. Access admin panel without selection

Domain-Specific Flaws

Overview

Business logic vulnerabilities often involve domain-specific rules and workflows. Understanding the business domain is crucial for identifying how attackers could manipulate intended functionality for malicious purposes.

E-commerce Discount Abuse Example

Classic Vulnerability: Discount Timing

- Shop offers 10% discount on orders over $1000
- Vulnerability: Discount applied before final cart validation
- Attack: Add items to reach threshold → Apply discount → Remove unwanted items
- Result: Discount on order no longer meeting criteria

Example: Gift Card Coupon Exploit

Scenario

Shop offers 30% coupon (SIGNUP30) on $10 gift cards. Users can purchase discounted gift cards and redeem them for store credit.

Vulnerability

Profit generation through automated gift card purchase and redemption cycle.

Exploitation Steps

1. Obtain 30% discount coupon via newsletter signup
2. Purchase $10 gift card → $7 after discount
3. Redeem gift card → Receive $10 store credit
4. Profit: $3 per cycle
5. Automate process using Burp Suite macros

Automation Setup

1. Click Settings in the top toolbar and click Sessions and then in the Session handling rules panel, click Add
2. go to the Scope tab. Under URL scope, select Include all URLs.
3. Go back to the Details tab. Under Rule actions, click Add > Run a macro. Under Select macro, click Add again to open the Macro Recorder.
4. Record macro sequence:

   ```
   1. POST /cart              (Add gift card)
   2. POST /cart/coupon       (Apply coupon)
   3. POST /cart/checkout     (Complete purchase)
   4. GET /order-confirmation (Get gift card code)
   5. POST /gift-card         (Redeem gift card)
   ```

   Then, click OK. The Macro Editor opens.
5. In the list of requests, select GET /cart/order-confirmation?order-confirmed=true. Click Configure item. In the dialog that opens, click Add to create a custom parameter. Name the parameter gift-card and highlight the gift card code at the bottom of the response. Click OK
6. Select the POST /gift-card request and click Configure item again. In the Parameter handling section, use the drop-down menus to specify that the gift-card parameter should be derived from the prior response (response 4). Click OK.
7. In the Macro Editor, click Test macro. Look at the response to GET /cart/order-confirmation?order-confirmation=true and note the gift card code that was generated. Look at the POST /gift-card request. Make sure that the gift-card parameter matches and confirm that it received a 302 response. Keep clicking OK until you get back to the main Burp window.
8. Send the GET /my-account request to Burp Intruder. Make sure that Sniper attack is selected.
9. In the Payloads side panel, under Payload configuration, select the payload type Null payloads and then click on Resource pool to open the Resource pool side panel. Add the attack to a resource pool with the Maximum concurrent requests set to 1.
10. Start the attack.

Domain Understanding

1. Study business rules and workflows
2. Identify financial incentives and rewards
3. Map transaction cycles and loops
4. Understand business objectives vs technical implementation

Attack Surface Identification

- Discount and coupon systems
- Loyalty and reward programs
- Referral systems
- Bulk pricing and tiered discounts
- Return and refund policies

Providing an Encryption Oracle

Overview

An encryption oracle occurs when an application allows users to encrypt arbitrary data and obtain the ciphertext. This becomes dangerous when other parts of the application use the same encryption algorithm.

Example: Cookie Decryption/Encryption Oracle

Vulnerability

Application provides two functions:

- Email validation error → Encrypts input → Sets cookie
- Notification cookie → Decrypts → Shows error message

Exploitation Steps

1. **Identify Encryption Oracle:**
   - Invalid email → Encrypted notification cookie
   - Error message shows decrypted email
   - Two-way oracle: Encrypt via POST, decrypt via GET
2. **Decrypt Stay-Logged-In Cookie:**
   - Use decrypt endpoint with stay-logged-in cookie
   - Reveals format: `username:timestamp`
   - Copy administrator's timestamp from own cookie
3. **Encrypt Administrator Cookie:**
   - Input: `administrator:timestamp`
   - Encrypt via email parameter
   - Get ciphertext from notification cookie
4. **Remove Prefix:**
   - Ciphertext includes "Invalid email address: " prefix
   - Block cipher requires 16-byte multiples
   - Add padding, encrypt, remove prefix blocks
   - Result: Clean administrator cookie ciphertext
5. **Authentication Bypass:**
   - Replace stay-logged-in cookie with crafted ciphertext
   - Access as administrator

Technical Analysis

Oracle Endpoints

```
# Encryption Oracle
POST /post/comment
email=arbitrary-data --> Sets encrypted notification cookie

# Decryption Oracle
GET /post?postId=x
notification-cookie=ciphertext --> Shows decrypted data
```

Key Findings

- **Two-way Oracle:** Both encrypt and decrypt available
- **Algorithm Reuse:** Same encryption for cookies and notifications
- **Prefix Vulnerability:** Predictable plaintext structure
- **Authentication Bypass:** Forge authentication cookies

Email Address Parser Discrepancies

Overview

Email address parsing varies across application components, allowing attackers to bypass domain restrictions using encoding techniques.

Vulnerability

Different components parse the same email address differently:

- Registration validation: Checks domain restrictions
- Email server: Parses and sends confirmation emails
- Database: Stores and retrieves email addresses
- Authentication: Verifies email domains for access control

Example: Domain Restriction Bypass via UTF-7

Scenario

Application restricts registration to `@ginandjuice.shop` domain.

Testing Steps

1. **Identify Restriction:**
   - Attempt registration with `foo@exploit-server.net`
   - Error: "email domain must be ginandjuice.shop"
2. **Test Encoding Bypasses:**
   - ISO-8859-1 encoded: `=?iso-8859-1?q?=61=62=63?=foo@ginandjuice.shop`
   - UTF-8 encoded: `=?utf-8?q?=61=62=63?=foo@ginandjuice.shop`
   - Both blocked: "Registration blocked for security reasons"
3. **Find Working Encoding:**
   - UTF-7 encoded: `=?utf-7?q?&AGEAYgBj-?=foo@ginandjuice.shop`
   - Accepted - UTF-7 bypasses validation
4. **Exploit:**

   ```
   =?utf-7?q?attacker&AEA-[EXPLOIT-SERVER-ID]&ACA-?=@ginandjuice.shop
   ```

   - Validation sees: `@ginandjuice.shop` (passes)
   - Mailer interprets: `attacker@[EXPLOIT-SERVER-ID]` (sends confirmation)
   - Registration successful with attacker's email

"'latex

How to Prevent Business Logic Vulnerabilities

Core Prevention Principles

To effectively prevent business logic vulnerabilities, focus on two key areas:

1. **Domain Understanding:** Ensure developers and testers fully comprehend the business domain the application serves
2. **Assumption Management:** Avoid making implicit assumptions about user behavior or application component interactions

Critical Prevention Measures

Assumption Validation

- Identify all assumptions about server-side state
- Implement explicit validation for each assumption
- Verify input sensibility before processing
- Continuously re-evaluate assumptions during development

Team Knowledge Management

- Ensure developers understand business rules thoroughly
- Train testers on expected application behavior
- Document edge cases and unusual scenarios
- Foster communication between technical and business teams

Development Best Practices

Documentation Standards

1. **Design Documentation:**
    - Maintain clear design documents for all workflows
    - Document data flows and state transitions
    - Note all assumptions made at each stage
    - Include business rule explanations
2. **Code Clarity:**
    - Write self-documenting, readable code
    - Avoid unnecessary complexity
    - Use descriptive variable and function names
    - Well-written code should be understandable without excessive documentation
3. **Complexity Management:**
    - For complex logic, produce detailed documentation
    - Document all assumptions and expected behaviors
    - Include examples of valid and invalid scenarios
    - Note dependencies and side-effects

Code Quality Practices

- **Dependency Tracking:**
    - Document all code component dependencies
    - Analyze side-effects of dependency manipulation
    - Consider how malicious users could exploit dependencies
- **Testing Integration:**
    - Include business logic tests in test suites
    - Test edge cases and unusual user behavior
    - Validate assumptions through automated testing
- **Code Review Focus:**
    - Review for implicit assumptions
    - Check business rule implementation correctness
    - Validate error handling and edge cases