Top Business Logic Vulnerability

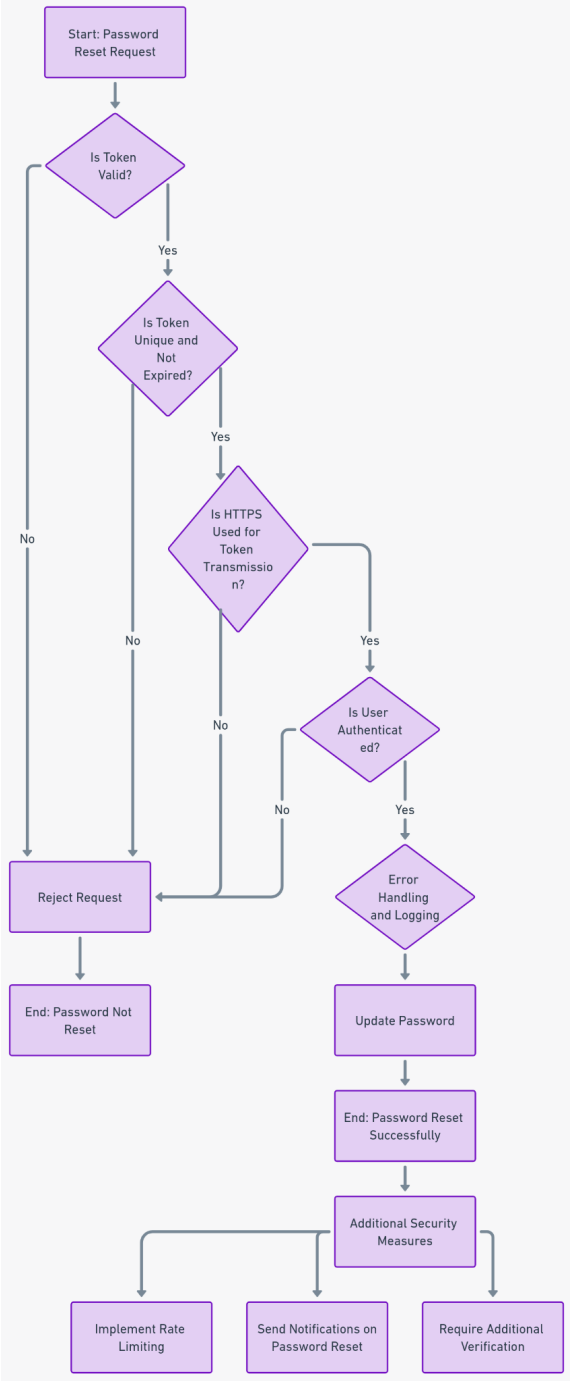
Web Edition

WWW.DEVSECOPSGUIDES.COM

DevSecOpsGuides.com

Top Business Logic Vulnerability in Web

Password reset broken logic



In the scenario , the password reset functionality of a web application is vulnerable due to broken logic. This vulnerability allows an attacker to reset the password of any user, including other users like 'Carlos', without needing the correct token. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Token Validation: Ensure that the password reset token is validated on the server side before allowing the password reset process to proceed.

Token Uniqueness and Expiry: Generate a unique token for each password reset request and set an expiry time for the token.

Secure Token Transmission: Use HTTPS to transmit the token securely.

User Authentication: Verify that the token is being used by the intended user.

Error Handling: Provide generic error messages to avoid enumeration attacks.

Logging and Monitoring: Log password reset attempts and monitor for suspicious activities.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.post('/forgot-password', (req, res) => {
    const { username, newPassword } = req.body;
    // Token is not validated
    updateUserPassword(username, newPassword);
    res.send('Password updated successfully');
```

```
});
```

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.post('/forgot-password', (req, res) => {
    const { username, newPassword, token } = req.body;

    if (!validateToken(username, token)) {
        res.status(400).send('Invalid or expired token');
        return;
    }

    updateUserPassword(username, newPassword);
    res.send('Password updated successfully');
});

function validateToken(username, token) {
    // Logic to validate the token for the user
    // Check if the token is valid, belongs to the user, and has not
expired
    // Return true if valid, false otherwise
}
```

In the compliant example, the server validates the token before allowing the password to be reset. This ensures that only the intended user can reset their password using a valid, unexpired token.

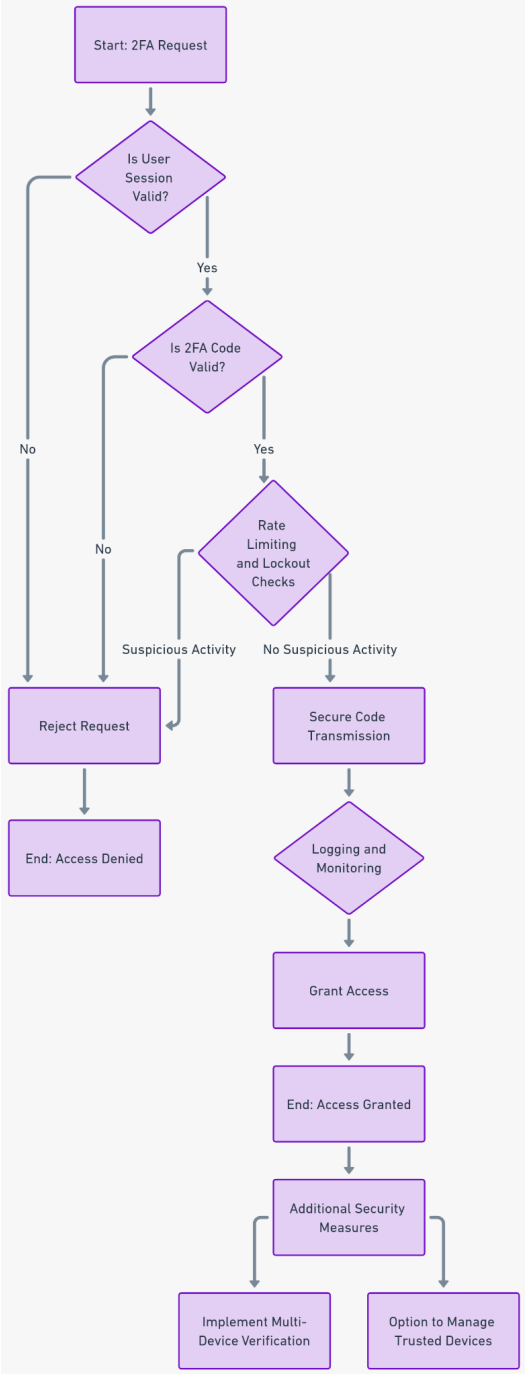
Additional Security Measures:

Implement rate limiting to prevent brute force attacks.

Send notifications to users when their password is reset.

Require additional verification (like security questions or email confirmation) during the password reset process.

2FA broken logic



In the scenario , the two-factor authentication (2FA) system of a web application is vulnerable due to flawed logic. This vulnerability allows an attacker to bypass the 2FA mechanism and gain unauthorized access to another user's account. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach for 2FA:

User Session Validation: Ensure that the 2FA process is tied to the user's current session and cannot be manipulated to target another user's account.

2FA Code Validation: Implement robust validation of the 2FA code, ensuring it matches the user's expected code and is within its validity period.

Rate Limiting and Lockout: Implement rate limiting and account lockout mechanisms to prevent brute-force attacks.

Secure Code Transmission: Use secure methods (like SMS, email, or an authenticator app) to transmit the 2FA code.

Logging and Monitoring: Log 2FA attempts and monitor for suspicious activities.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.post('/login2', (req, res) => {
    const { username, mfaCode } = req.body;
    // The 'verify' parameter is not tied to the user's session
    if (isValid2FACode(username, mfaCode)) {
        // User is logged in without validating the session
        res.redirect('/account');
```

```
    } else {
        res.status(401).send('Invalid 2FA code');
    }
});
```

In this non-compliant example, the 2FA process can be manipulated to target any user's account by changing the username parameter.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.post('/login2', (req, res) => {
  const { mfaCode } = req.body;
  const username = req.session.username; // Use username from the authenticated session

  if (!username || !isValid2FACode(username, mfaCode)) {
    res.status(401).send('Invalid 2FA code');
    return;
  }

  // User is logged in only after validating the 2FA code for the correct session
  res.redirect('/account');
});

function isValid2FACode(username, code) {
  // Validate the 2FA code for the user
  // Check if the code is correct and within its validity period
  // Return true if valid, false otherwise
}
```

In the compliant example, the 2FA process is tied to the user's session, ensuring that the 2FA code validation is specific to the logged-in user and cannot be manipulated to target another user's account.

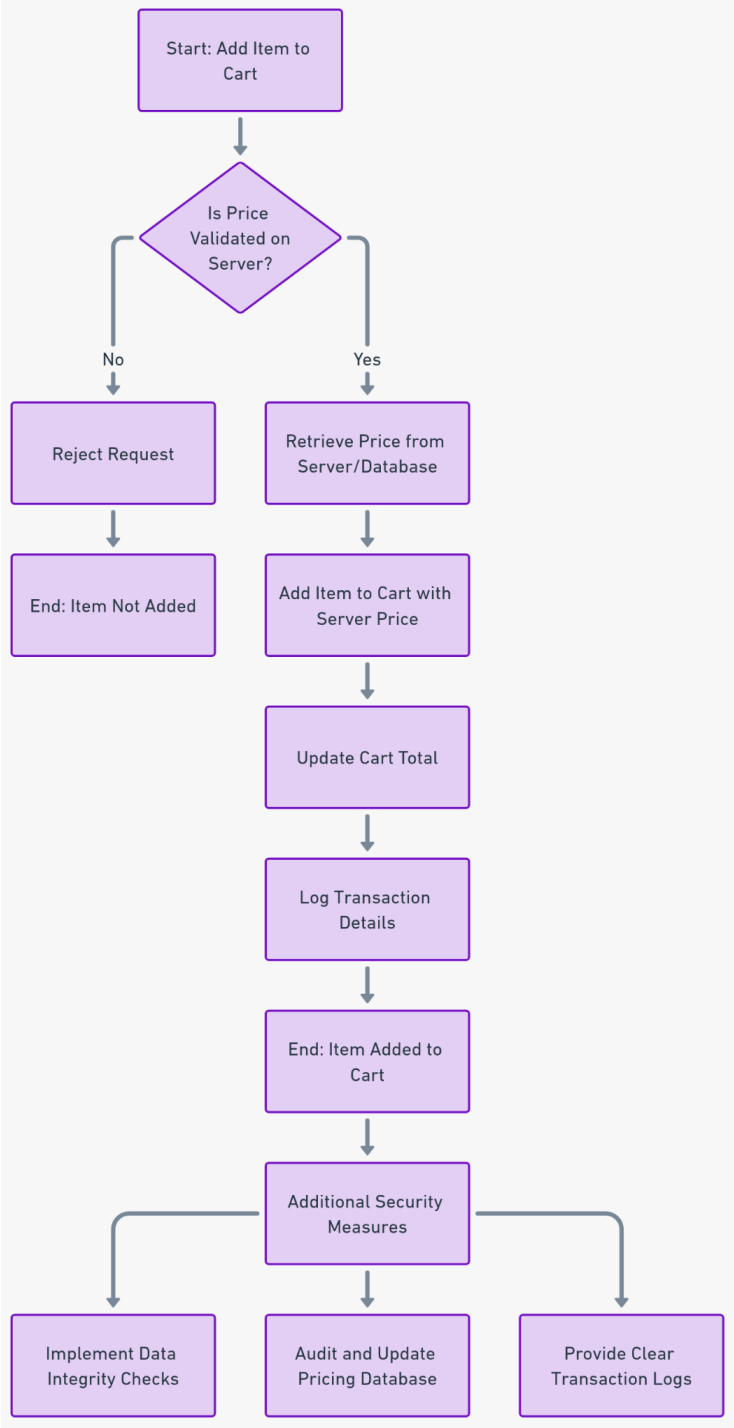
Additional Security Measures:

Implement multi-device verification where the user is notified of login attempts on new devices.

Use time-based one-time passwords (TOTP) which are valid only for a short period.

Provide the option for users to review and manage their trusted devices.

Excessive trust in client-side controls



In the scenario , the web application places excessive trust in client-side controls, particularly in the pricing mechanism of its purchasing workflow. This vulnerability allows an attacker to manipulate the price of items, potentially purchasing them for less than the intended price. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Server-Side Validation: Always validate critical data like pricing on the server side. Never trust client-side input as it can be easily manipulated.

Price Management: Manage prices securely on the server. Prices sent from the client should be used only for display purposes and not for transaction logic.

Use of Session or Database: Store critical transaction details like pricing in the server session or a secure database, not in client-side controls.

Logging and Monitoring: Log transaction details and monitor for unusual activities, such as sudden changes in the price of items in the cart.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.post('/cart', (req, res) => {
    const { itemId, price } = req.body;
    // Price is directly taken from client-side input
    addItemToCart(req.session.userId, itemId, price);
    res.send('Item added to cart');
});
```

In this non-compliant example, the price is directly taken from the client-side input, making it vulnerable to manipulation.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.post('/cart', (req, res) => {
    const { itemId } = req.body;
    const price = getItemPriceFromDatabase(itemId); // Get the actual
price from the server/database

    addItemToCart(req.session.userId, itemId, price);
    res.send('Item added to cart');
});

function getItemPriceFromDatabase(itemId) {
    // Logic to retrieve the actual price of the item from the database
    // Return the price
}
```

In the compliant example, the price is retrieved from the server or database, ensuring that it cannot be manipulated by the client.

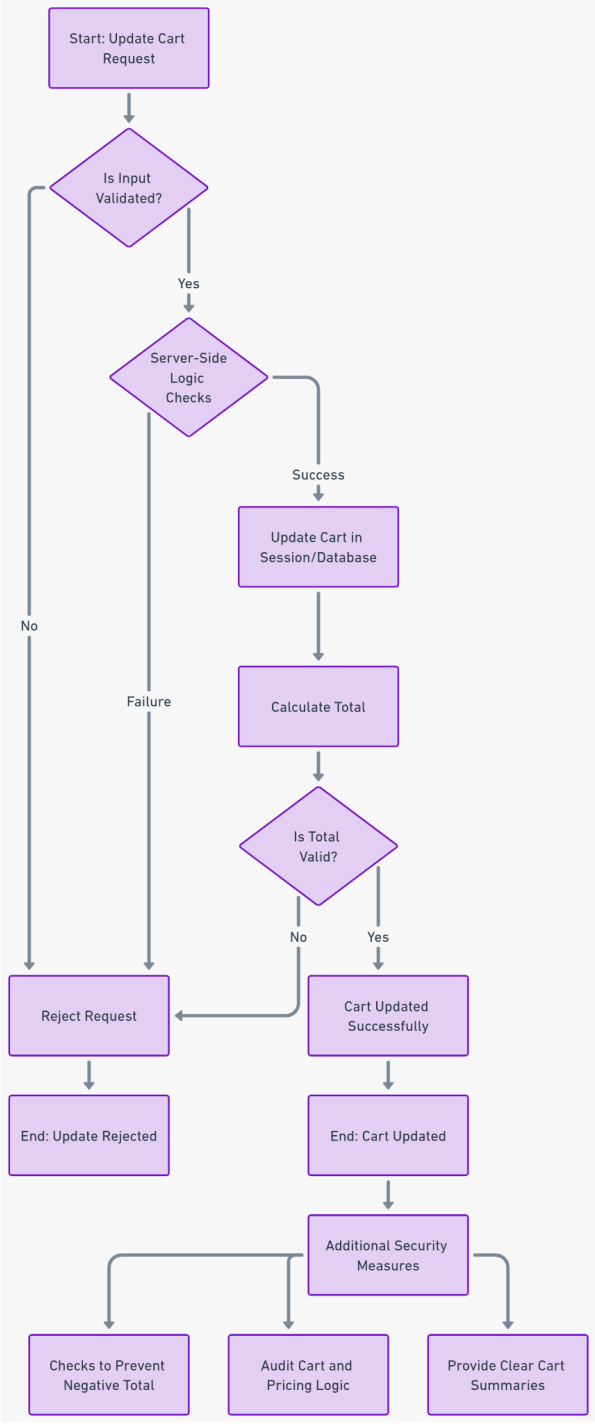
Additional Security Measures:

Implement checksums or hashes to verify the integrity of data sent from the client.

Regularly audit and update the pricing database to ensure accuracy.

Provide clear error messages and transaction logs for users to review their cart and purchases.

High-level logic vulnerability



In the scenario , the web application has a high-level logic vulnerability in its purchasing workflow, specifically in handling the quantity of items in the cart. This vulnerability allows an attacker to manipulate the quantity of items, potentially leading to negative pricing and unauthorized purchases. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Input Validation: Validate all user inputs on the server side, especially for critical parameters like quantity. Ensure that the quantity is a positive integer and within reasonable limits.

Server-Side Logic Checks: Implement logic checks on the server to prevent negative totals in the cart.

Use of Session or Database: Store cart details in the server session or a secure database, and always calculate totals based on server-side data.

Error Handling: Provide appropriate error messages for invalid inputs and handle such cases gracefully.

Logging and Monitoring: Log cart modifications and monitor for unusual activities, such as large changes in quantity.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.post('/cart', (req, res) => {
    const { itemId, quantity } = req.body;
    // Quantity is directly taken from client-side input without
validation
    updateCart(req.session.userId, itemId, quantity);
    res.send('Cart updated');
```

```
});
```

In this non-compliant example, the quantity is directly taken from the client-side input without any validation, allowing negative or unrealistic quantities.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.post('/cart', (req, res) => {
    const { itemId, quantity } = req.body;

    if (!isValidQuantity(quantity)) {
        res.status(400).send('Invalid quantity');
        return;
    }

    updateCart(req.session.userId, itemId, quantity);
    res.send('Cart updated');
});

function isValidQuantity(quantity) {
    // Check if the quantity is a positive integer and within reasonable
limits
    return Number.isInteger(quantity) && quantity > 0 && quantity <=
MAX_QUANTITY;
}
```

In the compliant example, the quantity is validated to ensure it is a positive integer and within reasonable limits, preventing negative totals in the cart.

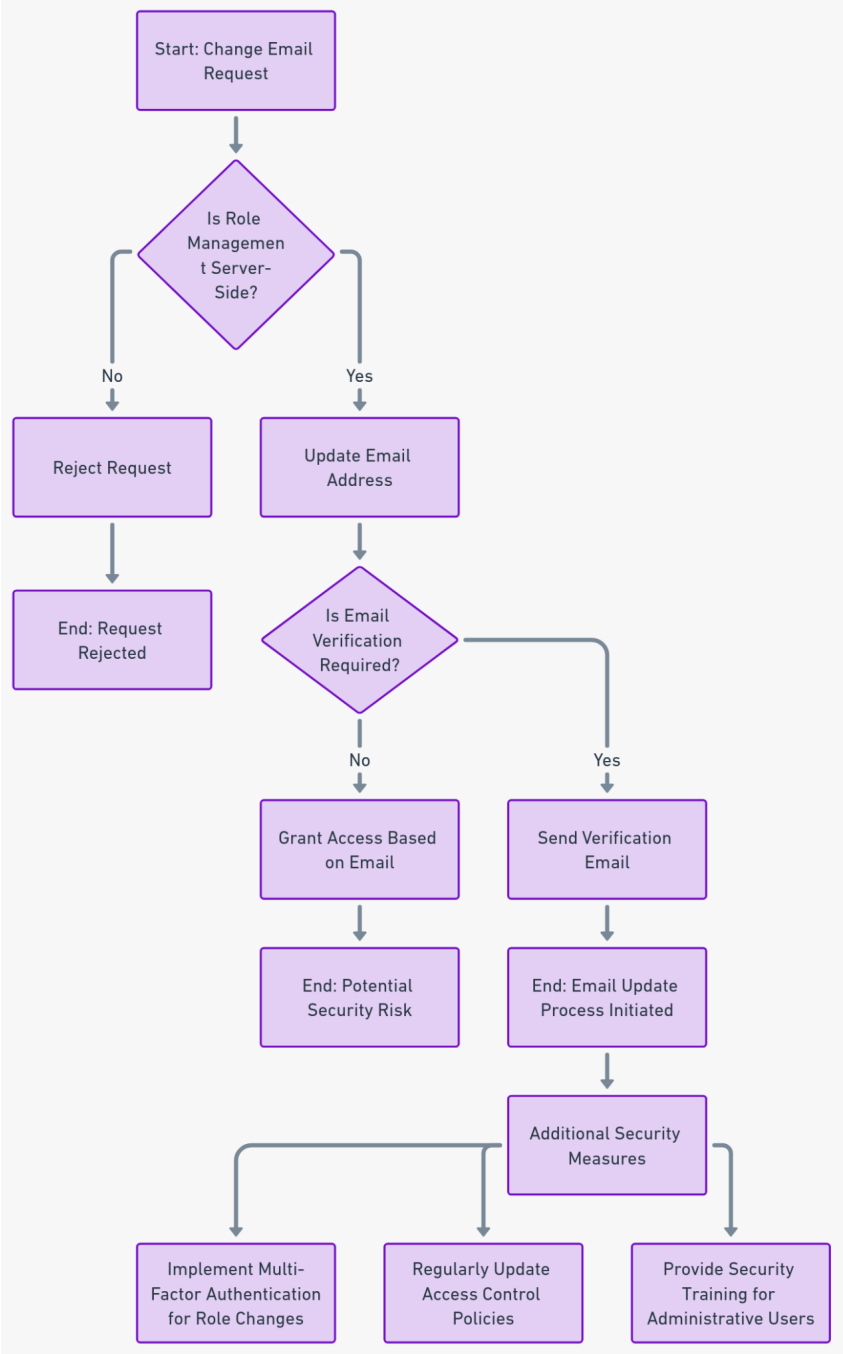
Additional Security Measures:

Implement checks to prevent the cart total from becoming negative.

Regularly audit and update the logic for cart and pricing calculations.

Provide clear summaries and breakdowns of cart contents and totals for user review.

Inconsistent security controls



In the scenario , the web application has inconsistent security controls, particularly in its access control logic for administrative functionalities. This vulnerability allows arbitrary users to gain access to administrative functions by manipulating their email address. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Robust Access Control: Implement robust access control mechanisms that verify a user's role and permissions before granting access to sensitive functionalities.

Role Management: Manage user roles and permissions securely on the server side. Role changes should be tightly controlled and not solely based on user input like email addresses.

Email Verification: Implement strict verification for email changes, especially when email domains are tied to specific privileges.

Audit and Monitoring: Regularly audit user roles and monitor for unusual activities, such as sudden role changes or access to privileged functionalities.

Error Handling: Provide generic error messages to prevent information leakage about sensitive functionalities.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.post('/change-email', (req, res) => {
    const { newEmail } = req.body;
    updateUserEmail(req.session.userId, newEmail);
```

```
    if (newEmail.endsWith('@dontwannacry.com')) {
        // Automatically granting admin privileges based on email domain
        grantAdminAccess(req.session.userId);
    }

    res.send('Email updated');
});
```

In this non-compliant example, the application automatically grants admin privileges based on the user's email domain, which can be easily manipulated.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.post('/change-email', (req, res) => {
    const { newEmail } = req.body;
    updateUserEmail(req.session.userId, newEmail);

    // Email verification process
    sendVerificationEmail(newEmail);

    // Admin privileges are not automatically granted based on email
domain
    res.send('Email update request received. Please verify your new
email');
});

function sendVerificationEmail(email) {
    // Send a verification email to the new address
    // The user must verify the email to complete the change
}
```

In the compliant example, the application requires email verification and does not automatically grant admin privileges based on the email domain.

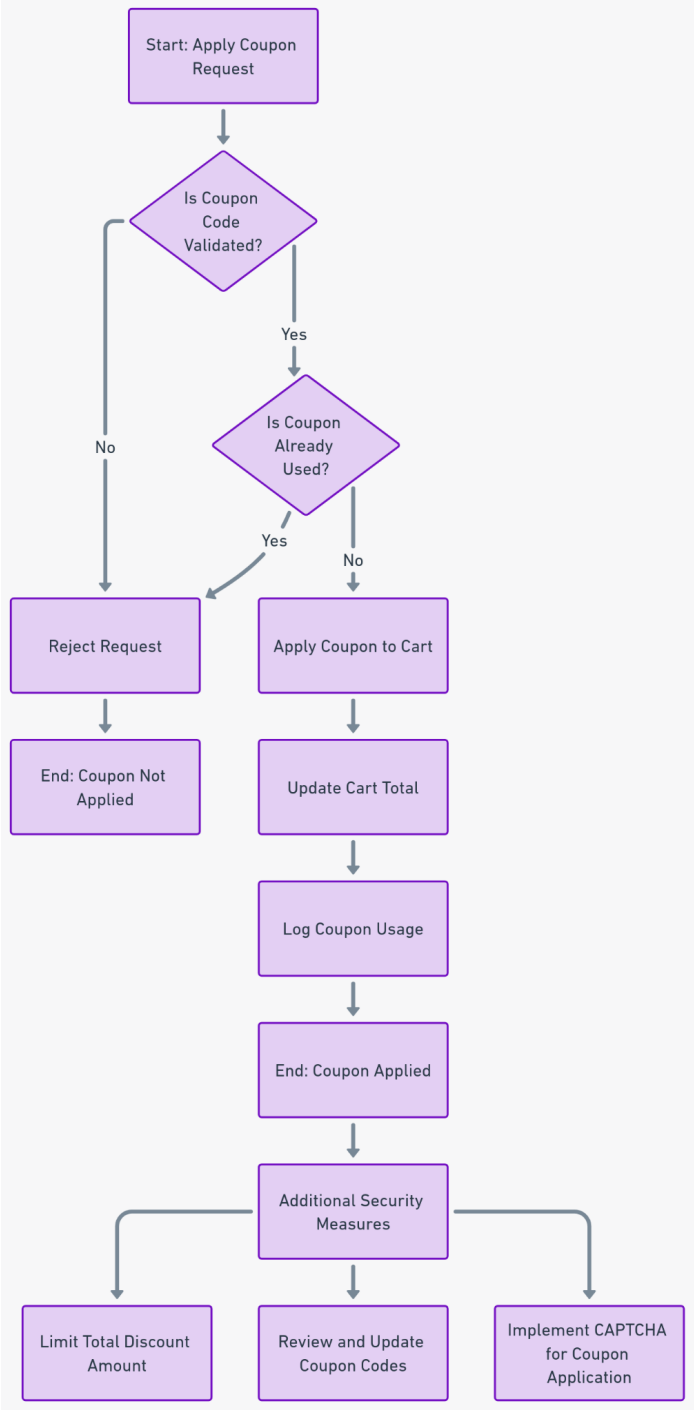
Additional Security Measures:

Implement multi-factor authentication for sensitive role changes.

Regularly review and update access control policies.

Provide training for users on security best practices, especially for those with administrative access.

Flawed enforcement of business rules



In the scenario , the web application has a flawed enforcement of business rules in its purchasing workflow, specifically in the handling of coupon codes. This vulnerability allows an attacker to exploit the coupon code system to receive unintended discounts. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Coupon Code Validation: Implement robust validation for coupon codes. Ensure each coupon can only be applied once per order.

Business Rule Enforcement: Enforce business rules on the server side, not just through client-side checks.

Coupon Usage Tracking: Keep track of used coupons in the user's session or in the database to prevent re-use in the same order.

Error Handling: Provide clear messages for invalid or already used coupon codes.

Audit and Monitoring: Regularly audit the coupon system and monitor for unusual activities, such as repeated use of the same coupon code.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.post('/apply-coupon', (req, res) => {
  const { couponCode } = req.body;
  if (isCouponValid(couponCode) && !isCouponAlreadyApplied(req.session.cart, couponCode)) {
    applyCouponToCart(req.session.cart, couponCode);
    res.send('Coupon applied');
  } else {
```

```
    res.send('Coupon cannot be applied');
  }
});

function isCouponAlreadyApplied(cart, couponCode) {
  // Checks if the coupon was applied in the last action only
  return cart.lastAppliedCoupon === couponCode;
}
```

In this non-compliant example, the application only checks if the coupon was applied in the last action, allowing users to alternate coupons and bypass the control.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.post('/apply-coupon', (req, res) => {
    const { couponCode } = req.body;
    if (isCouponValid(couponCode) &&
!hasCouponBeenUsed(req.session.cart, couponCode)) {
        applyCouponToCart(req.session.cart, couponCode);
        res.send('Coupon applied');
    } else {
        res.status(400).send('Coupon cannot be applied');
    }
});

function hasCouponBeenUsed(cart, couponCode) {
    // Checks if the coupon has been used at any point in the current
order
    return cart.usedCoupons.includes(couponCode);
}
```

In the compliant example, the application keeps track of all used coupons in the current order, preventing the same coupon from being applied more than once.

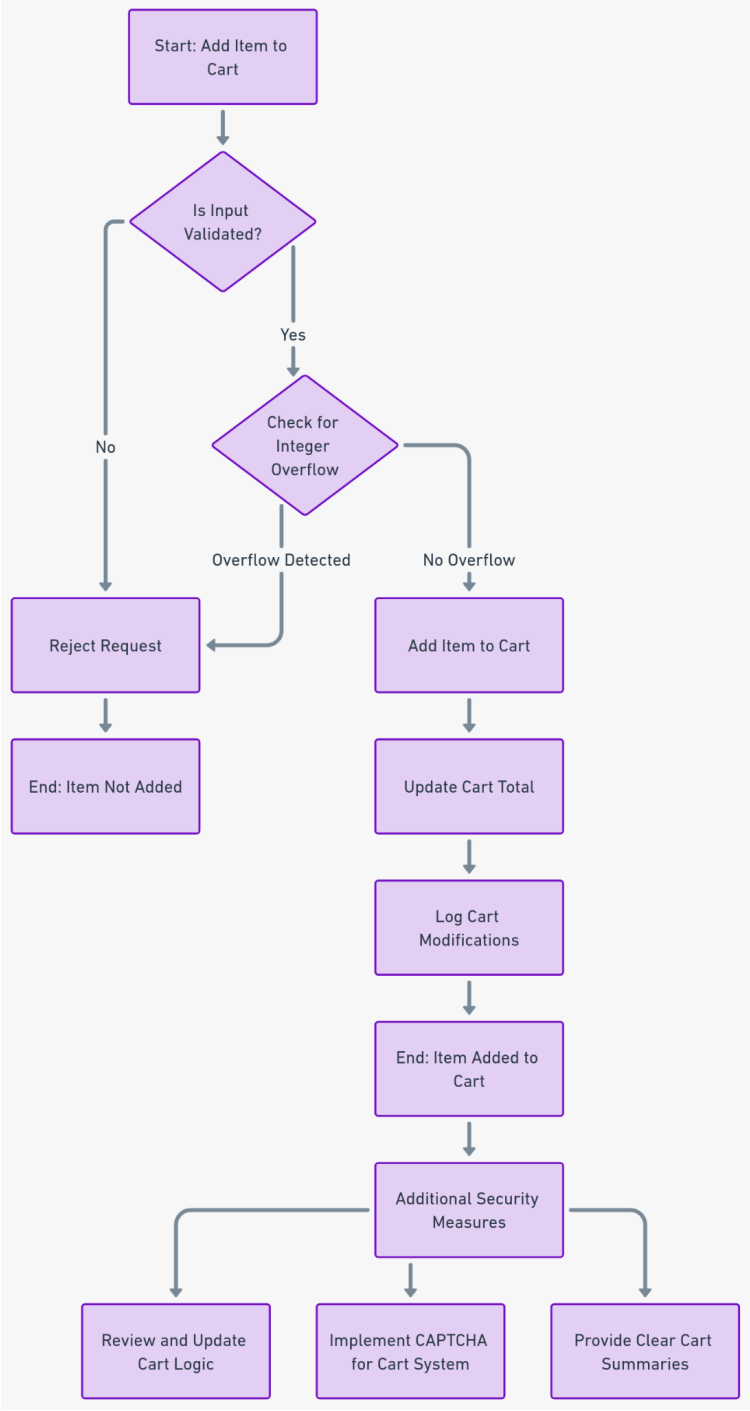
Additional Security Measures:

Limit the total discount amount that can be applied to an order.

Regularly review and update coupon codes and their usage rules.

Implement CAPTCHA to prevent automated abuse of the coupon system.

Low-level logic flaw



In the scenario , the web application has a low-level logic flaw in its purchasing workflow, specifically in handling the quantity and pricing of items in the cart. This vulnerability allows an attacker to manipulate the total price to an unintended negative value by exploiting the integer overflow issue. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Input Validation: Validate all user inputs on the server side, especially for critical parameters like quantity and price. Ensure that the quantity is a positive integer and within reasonable limits.

Integer Overflow Protection: Implement checks to prevent integer overflow. This can be done by setting maximum limits for quantities and total price.

Accurate Data Types: Use appropriate data types that can handle the range of expected values without overflow.

Error Handling: Provide appropriate error messages for invalid inputs and handle overflow cases gracefully.

Logging and Monitoring: Log cart modifications and monitor for unusual activities, such as large changes in quantity or total price.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.post('/cart', (req, res) => {
  const { itemId, quantity } = req.body;
  // No check for integer overflow
```

```
  addToCart(req.session.userId, itemId, quantity);
  res.send('Item added to cart');
});
```

In this non-compliant example, there's no check for integer overflow, allowing users to add large quantities that can cause the total price to overflow.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.post('/cart', (req, res) => {
    const { itemId, quantity } = req.body;

    if (!isValidQuantity(quantity)) {
        res.status(400).send('Invalid quantity');
        return;
    }

    try {
        addToCart(req.session.userId, itemId, quantity);
        res.send('Item added to cart');
    } catch (e) {
        res.status(500).send('Error updating cart');
    }
});

function isValidQuantity(quantity) {
    // Check if the quantity is a positive integer and within reasonable
limits
    // Also check for potential overflow
    return Number.isInteger(quantity) && quantity > 0 && quantity <=
MAX_QUANTITY;
}
```

In the compliant example, the quantity is validated to ensure it is a positive integer and within reasonable limits. Additionally, potential overflow is checked to prevent the total price from exceeding the maximum value.

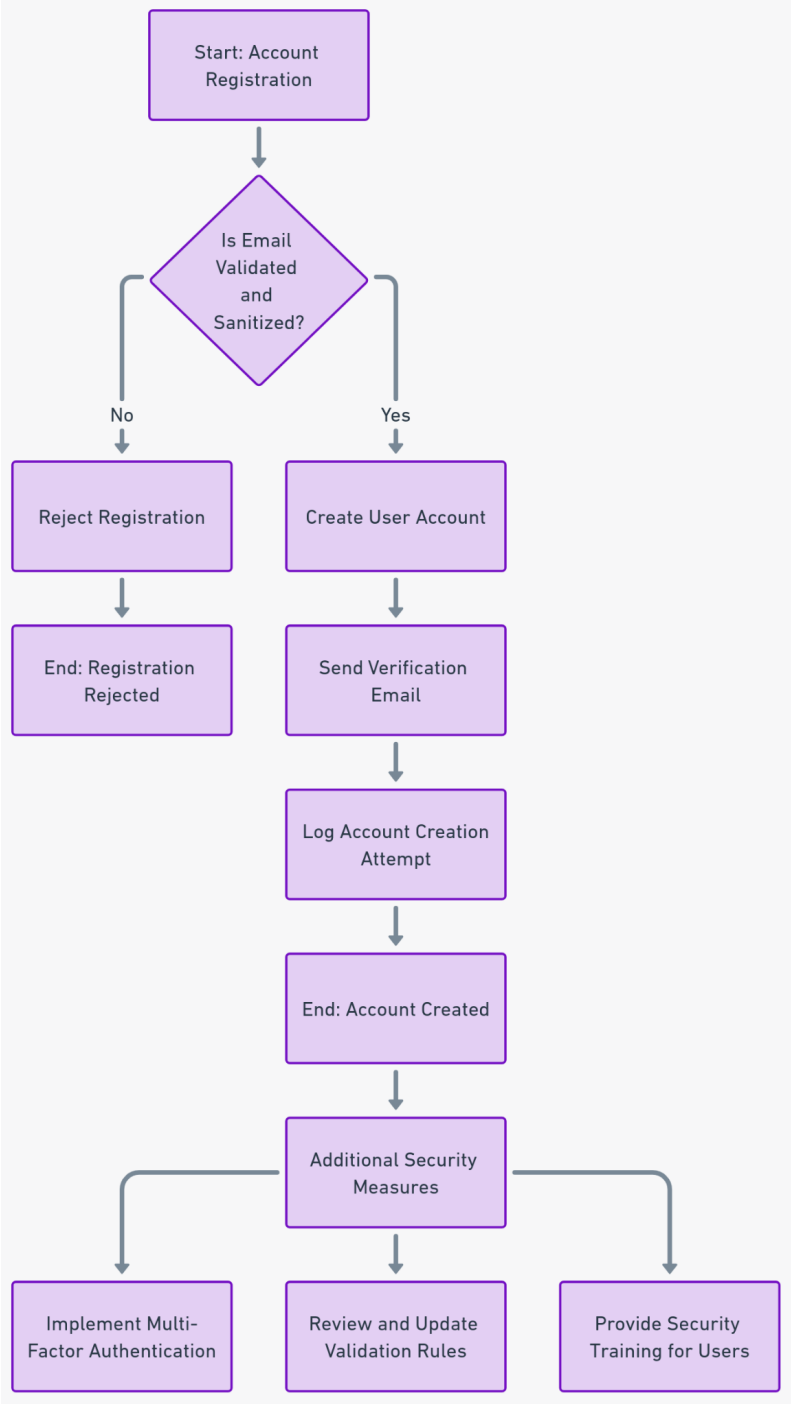
Additional Security Measures:

Regularly review and update the logic for cart and pricing calculations.

Implement CAPTCHA to prevent automated abuse of the cart system.

Provide clear summaries and breakdowns of cart contents and totals for user review.

Inconsistent handling of exceptional input



In the scenario , the web application has inconsistent handling of exceptional input, particularly in its account registration process. This vulnerability allows an attacker to exploit the system's handling of long email addresses to gain access to administrative functionality. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Input Validation and Sanitization: Implement robust validation and sanitization for all user inputs, especially for critical data like email addresses. Ensure that inputs conform to expected formats and lengths.

Consistent Input Handling: Ensure that the application handles inputs consistently across different components. The same validation logic should apply everywhere an input is processed.

Database Field Lengths: Align the lengths of input fields in the application with the corresponding database column sizes to prevent truncation issues.

Error Handling: Provide appropriate error messages for invalid inputs and handle edge cases gracefully.

Logging and Monitoring: Log account creation attempts and monitor for unusual patterns, such as registration with exceptionally long email addresses.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.post('/register', (req, res) => {
  const { email } = req.body;
  // No check for email length or format
  createUserAccount(email);
```

```
    res.send('Account created. Please verify your email');
});
```

In this non-compliant example, there's no check for the length or format of the email address, allowing users to register with excessively long emails.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.post('/register', (req, res) => {
    const { email } = req.body;

    if (!isValidEmail(email)) {
        res.status(400).send('Invalid email address');
        return;
    }

    createUserAccount(email);
    res.send('Account created. Please verify your email');
});

function isValidEmail(email) {
    // Check if the email is in a valid format and within the acceptable
length
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    return emailRegex.test(email) && email.length <= 255;
}
```

In the compliant example, the application validates the email address to ensure it is in a valid format and within the acceptable length, preventing truncation issues.

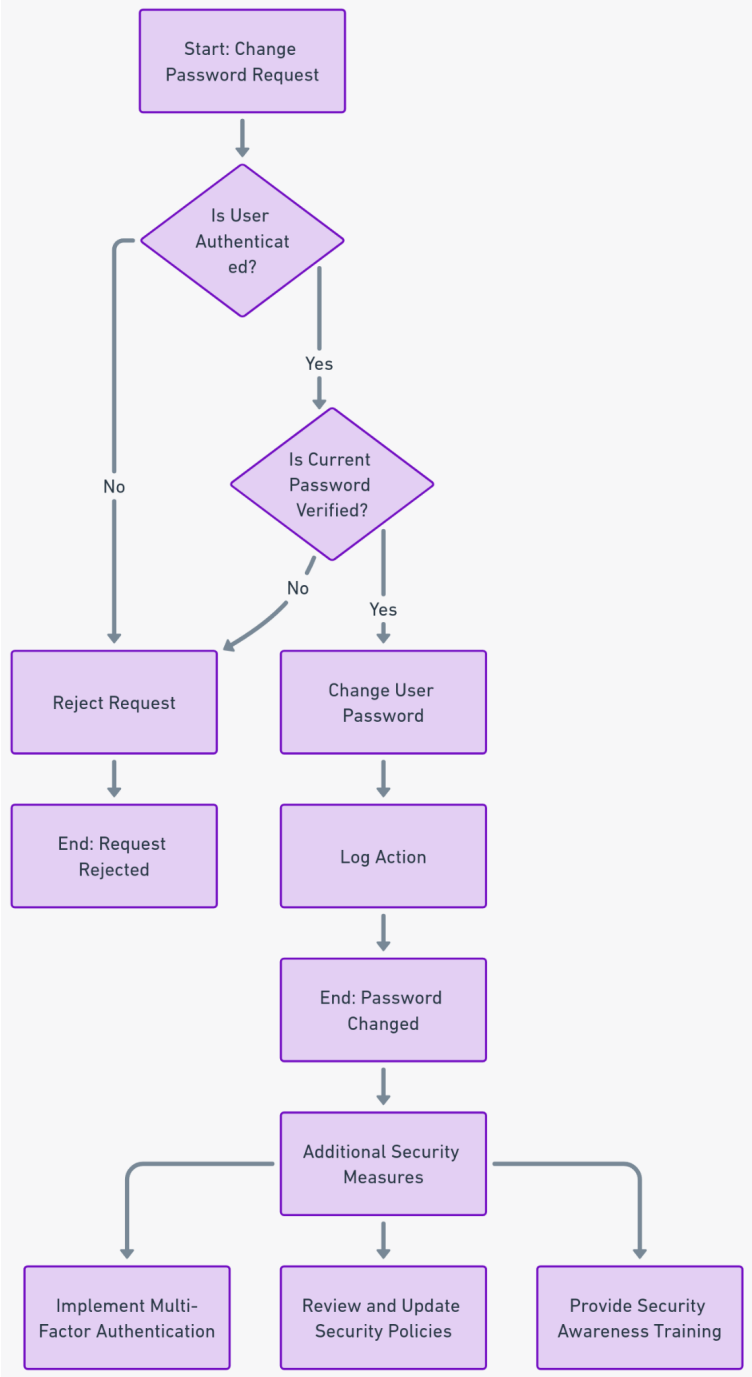
Additional Security Measures:

Implement multi-factor authentication for account registration and changes.

Regularly review and update input validation rules.

Provide training for users on security best practices.

Weak isolation on dual-use endpoint



In the scenario , the web application has weak isolation on a dual-use endpoint, particularly in its account management features. This vulnerability arises from flawed assumptions about a user's privilege level based on their input, allowing an attacker to exploit the logic to gain access to arbitrary users' accounts. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Strong Authentication Checks: Implement robust authentication checks for all sensitive actions, such as changing passwords. Always verify the current password before allowing changes.

Role and Privilege Verification: Verify the user's role and privileges before processing requests, especially when the request can affect other users' accounts.

Input Validation: Validate all inputs and reject requests with missing or unexpected parameters.

Error Handling: Provide appropriate error messages for invalid requests and handle edge cases securely.

Logging and Monitoring: Log all sensitive actions like password changes and monitor for unusual activities, such as attempts to modify other users' accounts.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.post('/my-account/change-password', (req, res) => {
  const { username, newPassword } = req.body;
  // No check for the current password or user's privilege
  changeUserPassword(username, newPassword);
  res.send('Password changed successfully');
});
```

In this non-compliant example, the application changes the password without verifying the current password or the user's privilege level.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.post('/my-account/change-password', (req, res) => {
    const { currentPassword, newPassword } = req.body;
    const username = req.session.username; // Get the username from the
session

    if (!currentPassword || !isPasswordCorrect(username,
currentPassword)) {
        res.status(400).send('Invalid current password');
        return;
    }

    changeUserPassword(username, newPassword);
    res.send('Password changed successfully');
});

function isPasswordCorrect(username, password) {
    // Check if the provided password is correct for the given username
    // Return true if correct, false otherwise
}
```

In the compliant example, the application verifies the current password before allowing the password change. It also uses the username from the session, preventing users from changing other users' passwords.

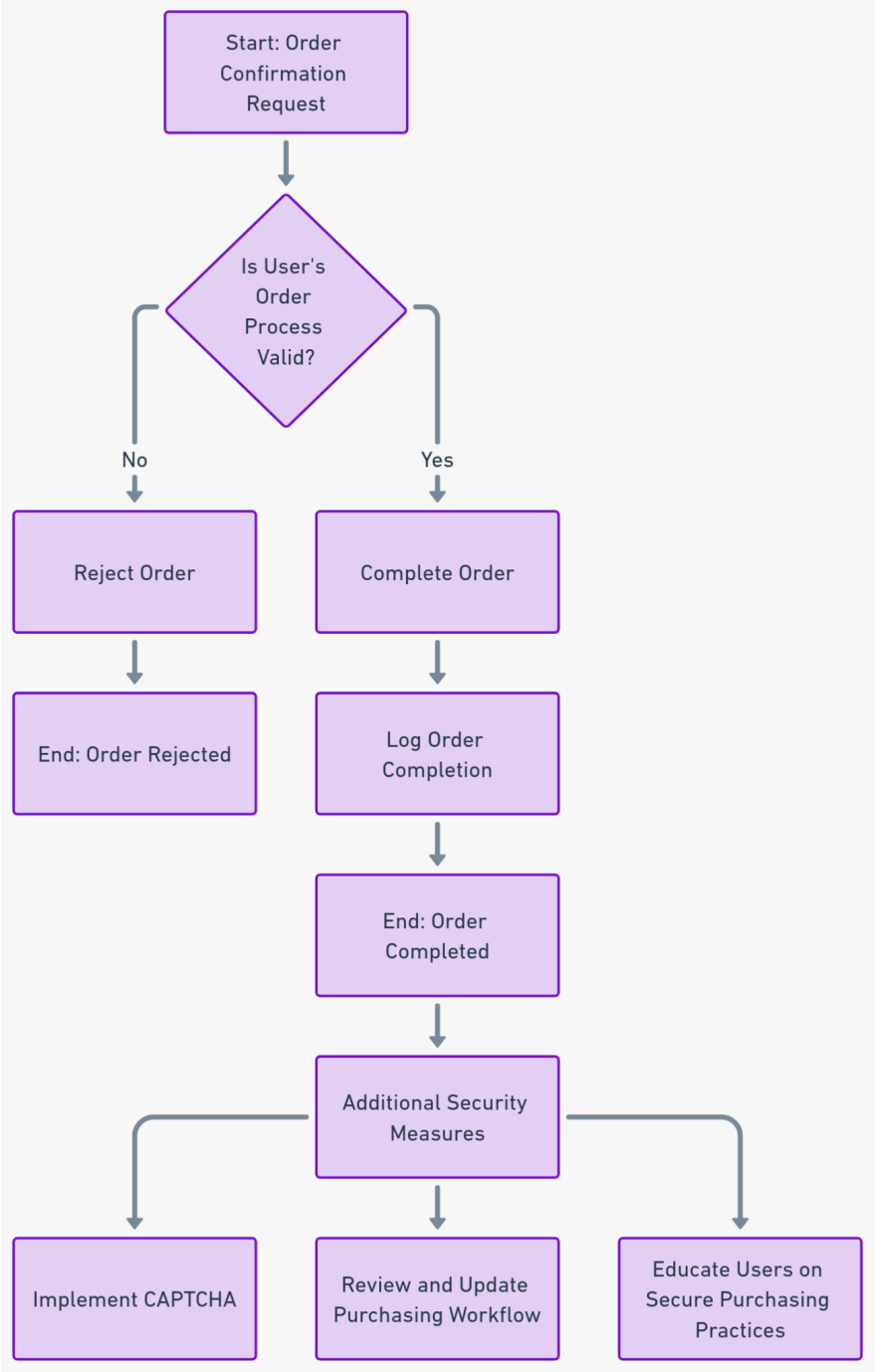
Additional Security Measures:

Implement multi-factor authentication for sensitive actions like password changes.

Regularly review and update security policies and access controls.

Provide security awareness training to users, emphasizing the importance of secure password practices.

Insufficient workflow validation



In the scenario , the web application has insufficient workflow validation in its purchasing process. This vulnerability arises from flawed assumptions about the sequence of events in the purchasing workflow, allowing an attacker to exploit this flaw to complete purchases without the cost being deducted. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Sequential Workflow Validation: Implement strict validation to ensure that each step of the purchasing workflow is completed in the correct order.

Server-Side State Management: Manage the state of the purchasing process on the server side. Do not rely solely on client-side inputs to determine the state of the workflow.

Transaction Integrity: Ensure that each transaction is atomic, meaning it either completes fully or not at all, to maintain data integrity.

Error Handling: Provide appropriate error messages for invalid workflow steps and handle edge cases securely.

Logging and Monitoring: Log all steps in the purchasing process and monitor for unusual patterns, such as skipping steps or repeating steps out of order.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.get('/cart/order-confirmation', (req, res) => {
  const { orderConfirmation } = req.query;
  if (orderConfirmation === 'true') {
    completeOrder(req.session.userId);
```

```
    res.send('Order completed');
  } else {
    res.send('Invalid order confirmation');
  }
});
```

In this non-compliant example, the application completes the order based solely on a query parameter, without validating the sequence of events in the purchasing process.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.get('/cart/order-confirmation', (req, res) => {
  if (!isOrderProcessValid(req.session.userId)) {
    res.status(400).send('Invalid order process');
    return;
  }

  completeOrder(req.session.userId);
  res.send('Order completed');
});

function isOrderProcessValid(userId) {
  // Check if the user has completed all necessary steps in the correct order
  // Return true if valid, false otherwise
}
```

In the compliant example, the application checks if the user has completed all necessary steps in the correct order before completing the order.

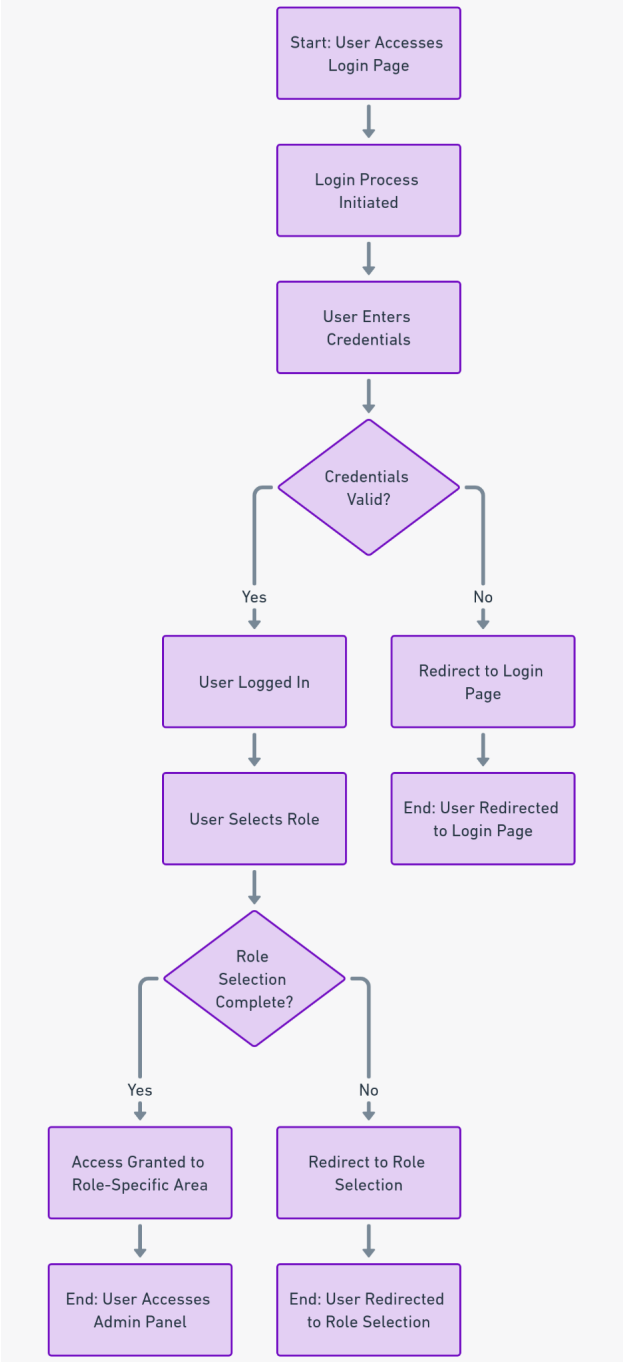
Additional Security Measures:

Implement CAPTCHA to prevent automated abuse of the purchasing process.

Regularly review and update the logic and security of the purchasing workflow.

Educate users about secure purchasing practices.

Authentication bypass via flawed state machine



In the scenario , the web application has an authentication bypass vulnerability due to a flawed state machine in its login process. This vulnerability arises from incorrect assumptions about the sequence of events during login, allowing an attacker to manipulate the process and gain unauthorized access. Let's discuss a secure coding approach for this scenario and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Strict State Management: Implement a strict state machine for the login process. Ensure that each step must be completed in the correct order before proceeding to the next.

Role Verification: Verify the user's role after login and before granting access to sensitive areas like the admin panel.

Session Management: Use secure session management practices. Invalidate the session if the expected sequence is violated.

Error Handling: Provide appropriate error messages and redirection for invalid access attempts.

Logging and Monitoring: Log all steps in the login process and monitor for unusual patterns, such as skipping steps or accessing unauthorized areas.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.get('/role-selector', (req, res) => {
  // User selects their role
  // No verification of the login process sequence
});
```

```
app.get('/admin', (req, res) => {
  if (req.session.role === 'administrator') {
    res.render('admin');
  } else {
    res.redirect('/home');
  }
});
```

In this non-compliant example, the application allows access to the admin panel based solely on the session role, without verifying the completion of the login process.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.get('/role-selector', (req, res) => {
    if (!req.session.isLoggedIn || !req.session.isLoginProcessComplete)
{
        res.redirect('/login');
        return;
    }
    // User selects their role
    req.session.isRoleSelected = true;
});

app.get('/admin', (req, res) => {
    if (req.session.role === 'administrator' &&
req.session.isRoleSelected) {
        res.render('admin');
    } else {
        res.redirect('/login');
    }
});
```

In the compliant example, the application checks if the user is logged in and if the login process (including role selection) is complete before allowing access to the admin panel.

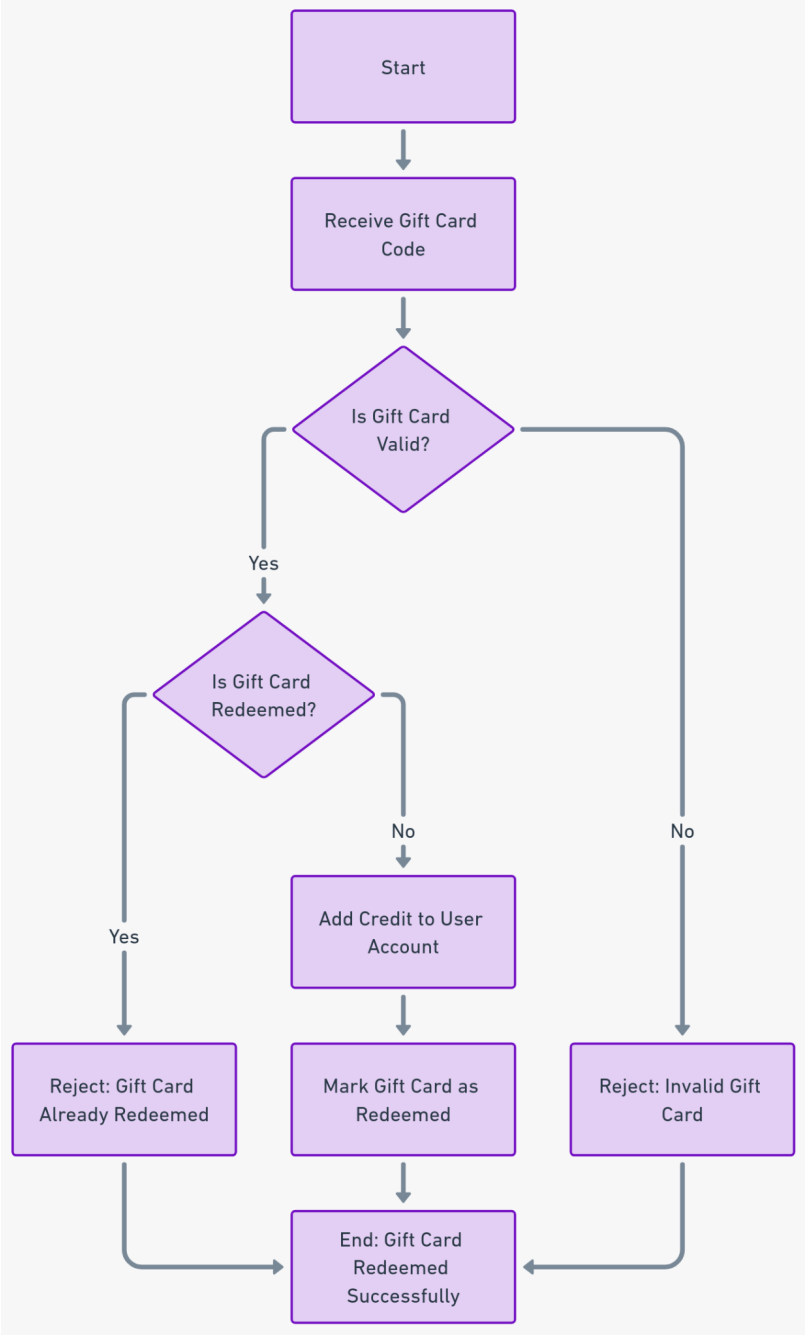
Additional Security Measures:

Implement multi-factor authentication for added security.

Regularly review and update the login process and state management logic.

Educate users about secure login practices and the importance of completing all steps.

Infinite money logic flaw



In the scenario, the web application has a logic flaw in its purchasing workflow, specifically in the handling of gift card purchases and redemptions. This flaw can be exploited to generate infinite store

credit. Let's discuss a secure coding approach to prevent such a flaw and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Validate Coupon and Gift Card Usage: Ensure that coupons and gift cards are used as intended. For instance, applying a discount coupon should not result in an increase in store credit.

One-Time Use for Gift Cards: Make sure that each gift card can only be redeemed once.

Transaction Integrity: Ensure that the financial transactions are atomic and consistent. The total store credit should reflect actual purchases and redemptions without discrepancies.

Audit and Monitoring: Regularly audit the system for any inconsistencies in financial transactions and monitor for patterns that indicate exploitation of logic flaws.

Limitations on Purchases and Redemptions: Implement limits on the number of gift cards that can be purchased or redeemed within a certain timeframe.

Noncompliant Code(Javascript):

```
// JavaScript/Node.js example of non-compliant code

app.post('/gift-card', (req, res) => {
  const { giftCardCode } = req.body;
  const creditToAdd = getGiftCardValue(giftCardCode);
  addUserCredit(req.session.userId, creditToAdd);
  res.send('Gift card redeemed');
});
```

In this non-compliant example, there's no check to ensure that the gift card is only redeemed once, leading to potential exploitation.

Compliant Code(Javascript):

```
// JavaScript/Node.js example of compliant code

app.post('/gift-card', (req, res) => {
    const { giftCardCode } = req.body;

    if (!isValidGiftCard(giftCardCode) ||
isGiftCardRedeemed(giftCardCode)) {
        res.status(400).send('Invalid or already redeemed gift card');
        return;
    }

    const creditToAdd = getGiftCardValue(giftCardCode);
    addUserCredit(req.session.userId, creditToAdd);
    markGiftCardAsRedeemed(giftCardCode);
    res.send('Gift card redeemed');
});

function isGiftCardRedeemed(giftCardCode) {
    // Check if the gift card has already been redeemed
    // Return true if redeemed, false otherwise
}
```

In the compliant example, the application checks if the gift card is valid and whether it has already been redeemed before adding credit to the user's account.

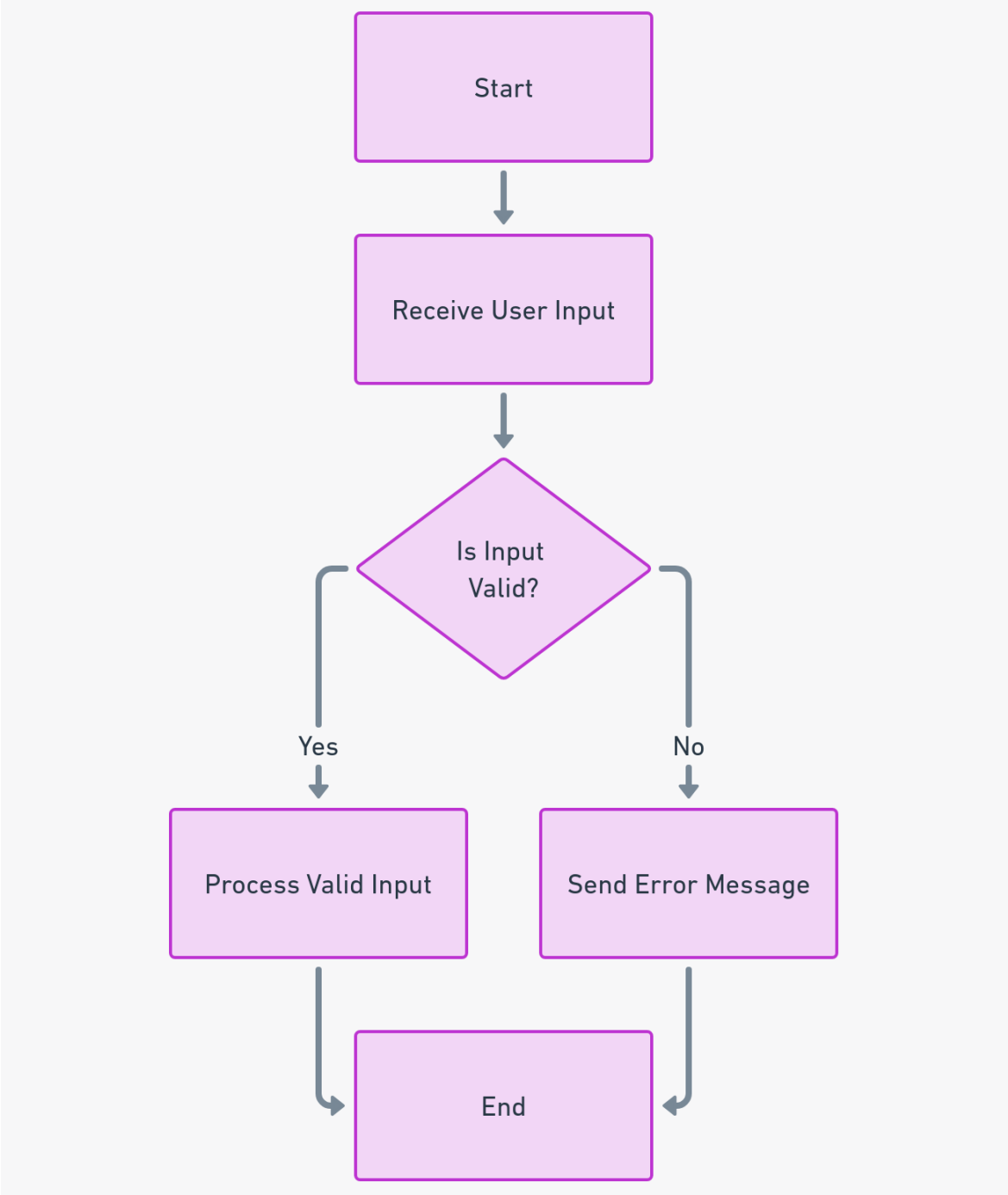
Additional Security Measures:

Implement CAPTCHA to prevent automated abuse of the gift card system.

Regularly review and update the logic for financial transactions.

Educate users about secure purchasing practices.

Authentication bypass via encryption oracle



In the scenario , the web application has an authentication bypass vulnerability due to the exposure of an encryption oracle. This vulnerability allows an attacker to manipulate encrypted data (like cookies) to gain unauthorized access. Let's discuss a secure coding approach to prevent such a flaw and provide examples of non-compliant (vulnerable) and compliant (secure) code.

Secure Coding Approach:

Avoid Reflective Encryption Oracles: Do not create scenarios where user input can be encrypted and then decrypted by the application in a way that is observable by the user.

Proper Encryption Key Management: Use strong, unique keys for different purposes and ensure they are securely managed.

Use of Secure Encryption Schemes: Implement encryption schemes that are resistant to manipulation, such as authenticated encryption with associated data (AEAD).

Input Validation: Validate user inputs to prevent them from being used to manipulate encryption or decryption processes.

Error Handling: Avoid providing detailed error messages that could give clues to attackers about the encryption process or data format.

Noncompliant Code(Javascript):

```
// Example of a non-compliant code in a Node.js/Express application

app.post('/post/comment', (req, res) => {
  const email = req.body.email;
  // Encrypt the email and set it in a cookie
  const encryptedEmail = encrypt(email);
  res.cookie('notification', encryptedEmail);
```

```
  // ...
});

app.get('/post', (req, res) => {
  const notification = req.cookies.notification;
  const decryptedNotification = decrypt(notification);
  // Reflect decrypted data in the response
  res.send(`Invalid email address: ${decryptedNotification}`);
});
```

In this non-compliant example, the application encrypts user input and reflects the decrypted data in the response, creating an encryption oracle.

Compliant Code(Javascript):

```
// Example of a compliant code in a Node.js/Express application

app.post('/post/comment', (req, res) => {
    const email = req.body.email;
    if (!isValidEmail(email)) {
        res.send('Invalid email address');
        return;
    }
    // Process the valid email address
    // ...
});

function isValidEmail(email) {
    // Implement email validation logic
    // Return true if valid, false otherwise
}
```

In the compliant example, the application validates the email address and does not use encryption or decryption based on user input.

Additional Security Measures:

Implement rate limiting to prevent brute-force attacks.

Regularly audit and update encryption algorithms and keys.

Educate developers about the risks of creating encryption oracles.

Reference

- https://portswigger.net/web-security/logic-flaws/