Security Vulnerabilities Documentation

Vulnerability 1: SQL Injection in Authentication

Type: SQL Injection (Authentication Bypass) **Location**: includes/functions.php - login() function **Severity**: Critical

Description

The login function directly concatenates user input into SQL queries without proper sanitization:

```
$query = "SELECT * FROM users WHERE username = '$username' AND password =
'$base64'";
```

How to Fix

Replace the vulnerable query with prepared statements:

```
// FIXED: Use prepared statements to prevent SQL injection
$query = "SELECT * FROM users WHERE username = ? AND password = ?";
$stmt = mysqli_prepare($conn, $query);
mysqli_stmt_bind_param($stmt, "ss", $username, $base64);
mysqli_stmt_execute($stmt);
$result = mysqli_stmt_get_result($stmt);
```

Flag Association: This vulnerability enables access to Flag 1 (Admin Password Discovery)

Vulnerability 2: SQL Injection in Review Update

Type: SQL Injection (Data Extraction) **Location**: includes/functions.php - updateReview() function **Severity**: Critical

Description

The review update function is vulnerable to SQL injection through the $review parameter:

```
$query
= "UPDATE coffee_reviews SET review_text = '$review' WHERE id =
$existing_review_id";
```

While there are some basic protections (blocking semicolons and certain keywords), the function is still vulnerable to UPDATE-based injection.

How to fix

Implement proper prepared statements:

```
// FIXED: Use prepared statements for secure parameter binding
$query = "UPDATE coffee_reviews SET review_text = ? WHERE id = ?";
$stmt = mysqli_prepare($conn, $query);
mysqli_stmt_bind_param($stmt, "si", $review, $existing_review_id);
return mysqli_stmt_execute($stmt);
```

Flag Association: This vulnerability enables access to Flag 1 (Admin Password Discovery)

Vulnerability 3: Unrestricted File Upload

Type: Unrestricted File Upload **Location**: includes/functions.php - handleFileUpload() function **Severity**: Critical

Description

The file upload function for admin users has no file type validation:

```
function handleFileUpload($file, $target_dir = './uploads/') {
    // Simple admin check - only admins can upload profile pictures
    if (!isAdmin()) {
        return false;
    }
    // ... no file type validation for admins
    $target_file = $target_dir . basename($file['name']);
    if (move_uploaded_file($file['tmp_name'], $target_file)) {
        return basename($file['name']);
    }
}
```

How to Fix

Implement strict file type validation and secure file handling:

```
function handleFileUpload($file, $target_dir = './uploads/') {
    if (!isAdmin()) {
        return false;
    }

    if (!isset($file['tmp_name']) || !is_uploaded_file($file['tmp_name']))
{
        return false;
    }

    // FIXED: Validate file extension
    $allowed_extensions = ['jpg', 'jpeg', 'png', 'gif'];
    $file_extension = strtolower(pathinfo($file['name'],
PATHINFO_EXTENSION));

    if (!in_array($file_extension, $allowed_extensions)) {
        error_log("Blocked upload of disallowed file type: " .
$file_extension);
        return false;
    }

    // FIXED: Validate MIME type
    $allowed_mime_types = ['image/jpeg', 'image/png', 'image/gif'];
    $finfo = finfo_open(FILEINFO_MIME_TYPE);
    $mime_type = finfo_file($finfo, $file['tmp_name']);
    finfo_close($finfo);

    if (!in_array($mime_type, $allowed_mime_types)) {
        error_log("Blocked upload of disallowed MIME type: " .
$mime_type);
        return false;
    }

    // FIXED: Generate safe filename to prevent directory traversal
    $safe_filename = uniqid() . '_' . preg_replace('/[^a-zA-Z0-9._-]/',
'', basename($file['name']));
    $target_file = $target_dir . $safe_filename;

    // FIXED: Ensure upload directory is outside web root or not
executable
    if (move_uploaded_file($file['tmp_name'], $target_file)) {
        return $safe_filename;
    }

    return false;
}
```

Security Recommendations

- Input Validation: Implement comprehensive input validation and sanitization
- Prepared Statements: Use prepared statements for all database queries
- File Upload Security: Implement strict file type validation and secure file storage
- Access Controls: Implement proper role-based access controls
- Error Handling: Implement secure error handling that doesn't leak sensitive information
- Security Headers: Add appropriate security headers to prevent various attacks
- Regular Security Audits: Conduct regular code reviews and security assessments
- Or dont use php ;D

Flag Association: This vulnerability enables access to Flag 2 (File System Access)

Vulnerability 4: Buffer Overflow in Coffee Blend Input

Type: Buffer Overflow **Location**: barista_academy.c - ctf_challenge_one() **Severity**: High

Description

The original implementation of the barista challenge used an unsafe input function to read the user's favorite coffee blend into a fixed-size buffer:

```
char coffee_order[16];
gets(coffee_order); // Vulnerable: no bounds checking
```

This allowed an attacker to input more than 15 characters, overflowing the buffer and overwriting adjacent variables in memory, such as `coffee_strength`. By carefully crafting the input, an attacker could set `coffee_strength` to a target value and unlock the flag.

How to Fix

Use a safe input function that limits the number of characters read:

```
// FIXED: Use fgets() to prevent buffer overflow
fgets(coffee_order, 16, stdin);
coffee_order[strcspn(coffee_order, "\n")] = '\0';
```

Flag Association: This vulnerability enables access to Flag 3 (Coffee Strength Flag)

Vulnerability 5: Integer Overflow in Coffee Purchase

Type: Integer Overflow **Location**: barista_academy.c - ctf_challenge_two() **Severity**: High

Description

The original code used 32-bit integers for balance and cost calculations:

```
int account_balance = 1100;
int total_cost = 900 * number_coffees;
```

If a very large number of coffees was entered, the multiplication could overflow, resulting in a negative or very large positive value. This could allow an attacker to bypass balance checks and increase their balance to an unintended value, unlocking the flag.

How to Fix

Use 64-bit types for calculations and validate input:

```
// FIXED: Use long for calculations
long account_balance = 1100;
long total_cost = (long)900 * number_coffees;
```

Flag Association: This vulnerability enables access to Flag 4 (Coffee Shop Balance Flag)

Vulnerability 6: Format String Vulnerability in Secret Order Printer

Type: Format String Vulnerability **Location**: barista_academy.c - ctf_final_challenge() **Severity**: High

Description

The original implementation of the secret order printer used `printf(buffer)` directly, allowing user-controlled format strings:

```
char buffer[1024];
scanf("%1024s", buffer);
printf(buffer); // Vulnerable: user-controlled format string
```

This allowed an attacker to leak memory content from the stack, including sensitive variables and flags, by supplying format specifiers such as `%llx`.

How to Fix

Always use format specifiers when printing user input:

```
// FIXED: Use format string with user input as an argument
printf("%s", buffer);
```

Flag Association: This vulnerability enables access to Flag 5 (Secret Order Printer Flag)