Mishandling of Exceptional Conditions Vulnerability: A Detailed Study

Dinesh Kumar Mohanty, Dragan Peraković
Research Scholar, Full-time professor,
Swiss School of Business Management,
Geneva, Switzerland
kiitkp03@gmail.com, dragan.perakovic@fpz.hr

***Abstract*— Mishandling of exceptional conditions is ranked as 10 in the OWASP Top 10 list in 2025, consisting of 24 sub-vulnerabilities. Web applications fail to predict, identify, and react to an unusual behavior of an application, revealing sensitive information in the form of a stack trace or error handling. Which is prone to security vulnerabilities like server crashes, outages, and sometimes a DDOS attack. Since these vulnerabilities expose sensitive information for attackers in gaining attacked system information, when exploited, this may cause serious reputational damage to the organization. In this research paper we will be talking about real-time test cases and possible mitigation methodologies. We will discuss possible remediation techniques that can be adopted by the developers when developing the product.**

***Index Terms*—OWASP, error handling, exception handling, web application vulnerability, vulnerability exception**

I. INTRODUCTION

Mishandling of Exceptional Conditions—It is a broader category in which an application fails to predict, identify, and react to an unusual behavior of an application, revealing sensitive information in the form of a stack trace or error handling. Which is prone to security vulnerabilities like server crashes, outages, and sometimes a DDOS attack [1]. This category consists of 24 Common Weakness Enumerations (CWE), meaning 24 vulnerabilities in this category that focuses on improper error handling, logical errors, failing open, and other related scenarios stemming from abnormal conditions and systems that may be encountered [1]. Security exceptions can occur when an application does not handle user-supplied input when it tries to execute, or sometimes developers unknowingly enable echoing or printing sensitive information, which then gets rendered within the user interface (UI), which is a fruitful thing for attackers to gather basic information about the application, its version, and the path on which the application is installed, which then can be used for a further chain of attacks.

II. LIST OF MAPPED VULNERABILITIES

Below are the CWE IDs that are clubbed into this category of OWASP Top 10 vulnerabilities for the year 2025.

Table 1: CWE IDs under Mishandling of Exceptional Conditions

CWE Number	CWE Name
CWE-209	Generation of Error Message Containing Sensitive Information
CWE-215	Insertion of Sensitive Information Into Debugging Code
CWE-234	Failure to Handle Missing Parameter
CWE-235	Improper Handling of Extra Parameters
CWE-248	Uncaught Exception
CWE-252	Unchecked Return Value
CWE-274	Improper Handling of Insufficient Privileges
CWE-280	Improper Handling of Insufficient Permissions or Privileges
CWE-369	Divide By Zero
CWE-390	Detection of Error Condition Without Action

CWE-391	Unchecked Error Condition
CWE-394	Unexpected Status Code or Return Value
CWE-396	Declaration of Catch for Generic Exception
CWE-397	Declaration of Throws for Generic Exception
CWE-460	Improper Cleanup on Thrown Exception
CWE-476	NULL Pointer Dereference
CWE-478	Missing Default Case in Multiple Condition Expression
CWE-484	Omitted Break Statement in Switch
CWE-550	Server-generated Error Message Containing Sensitive Information
CWE-636	Not Failing Securely ('Failing Open')
CWE-703	Improper Check or Handling of Exceptional Conditions
CWE-754	Improper Check for Unusual or Exceptional Conditions
CWE-755	Improper Handling of Exceptional Conditions
CWE-756	Missing Custom Error Page

III. METHODOLOGY

This is an internal security testing method that is followed for each and every CWE number.

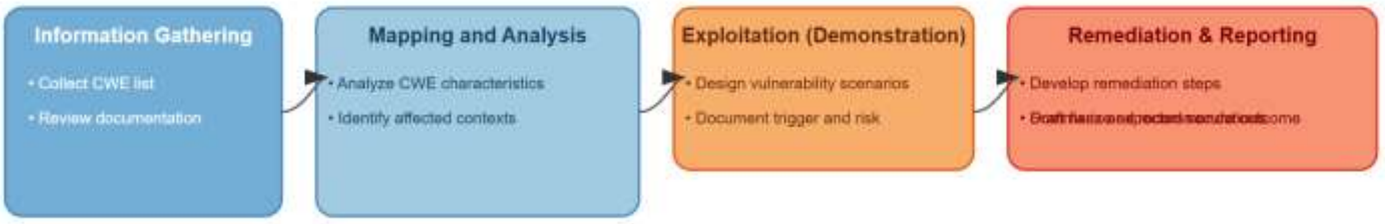

Information Gathering
• Collect CWE list
• Review documentation
Mapping and Analysis
• Analyze CWE characteristics
• Identify affected contexts
Exploitation (Demonstration)
• Design vulnerability scenarios
• Document trigger and risk
Remediation & Reporting
• Develop remediation steps

Figure 1: Methodology

1. Information gathering: This is the step where we collected a detailed list of relevant CWEs for the category with a review of specific CWE IDs.

2. Mapping and Analysis: Analysis of each CWE and why it is being exploited.

3. Exploitation (Demonstration): Design vulnerable code for each CWE that developers are left with.

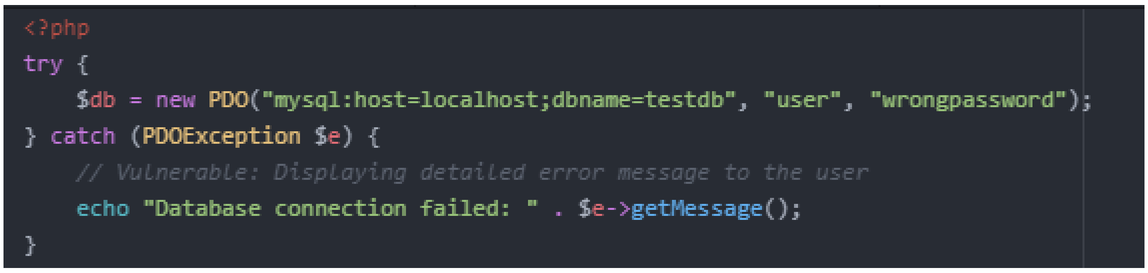
4. Remediation & Reporting: Develop remediation strategies for each vulnerability scenario with fixed code a developer should adopt while developing and its best practices.

IV. RESULT DISCUSSION

1. CWE-209 Generation of Error Message Containing Sensitive Information

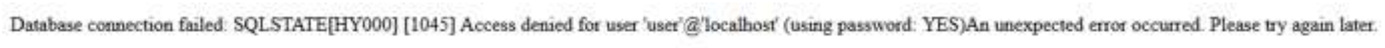
Information:
An application that generates an error message that includes sensitive information about its environment, user information, database connection, internal SQL errors, or associated data [2]. Attackers can use this information to perform other high-severity vulnerabilities like SQL injection or credential stuffing.

Exploitation:



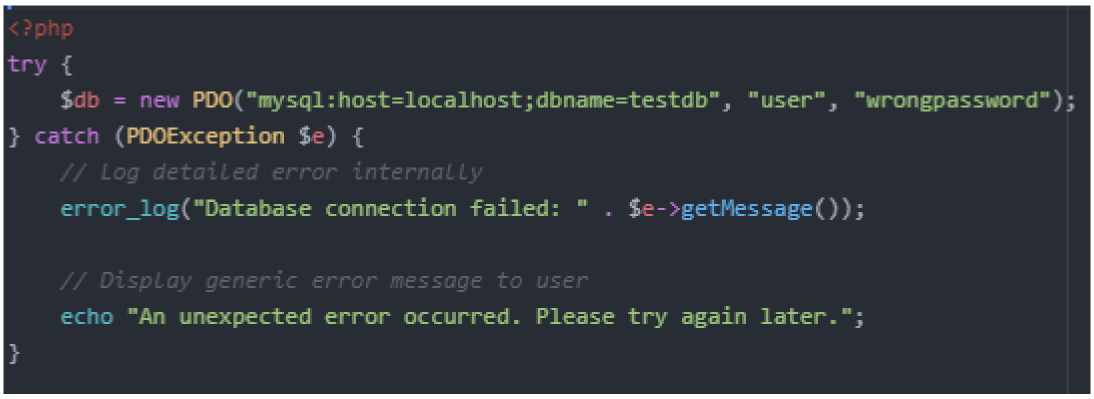
```php
<?php
try {
    $db = new PDO("mysql:host=localhost;dbname=testdb", "user", "wrongpassword");
} catch (PDOException $e) {
    // Vulnerable: Displaying detailed error message to the user
    echo "Database connection failed: " . $e->getMessage();
}
```

Figure 2: CWE-209 Vulnerable code



```
Database connection failed: SQLSTATE[HY000] [1045] Access denied for user 'user'@'localhost' (using password: YES)An unexpected error occurred. Please try again later.
```

Figure 3: CWE-209 Vulnerable code output



```php
<?php
try {
    $db = new PDO("mysql:host=localhost;dbname=testdb", "user", "wrongpassword");
} catch (PDOException $e) {
    // Log detailed error internally
    error_log("Database connection failed: " . $e->getMessage());

    // Display generic error message to user
    echo "An unexpected error occurred. Please try again later.";
}
```

Figure 4: CWE-209 Fixed code

Remediation: Do not expose detailed error information to the end user; rather, log it internally to a log server or log files for lateral analysis of the issue by showing a generic error message that will make it difficult for an attacker to guess how the application is behaving.

2. CWE-215 Insertion of Sensitive Information Into Debugging Code

Information:
Inserting sensitive information (like passwords, tokens, and user data) into debugging code creates the risk that such info will be exposed if debugging is enabled or not properly disabled before going live [3]. This can result in information leakage, credential theft, or compliance violations [3].

Exploitation:

Table 2: Vulnerable Code

Location	Vulnerability
ini_set('display_errors',1);	Enables full debug output to the user
Debug section in login failure	Prints user password, DB host, DSN, file paths
Exception handler	Prints raw PDO exception, possibly revealing credentials or schema
Uses real-looking secrets	Demonstrates leakage of sensitive info

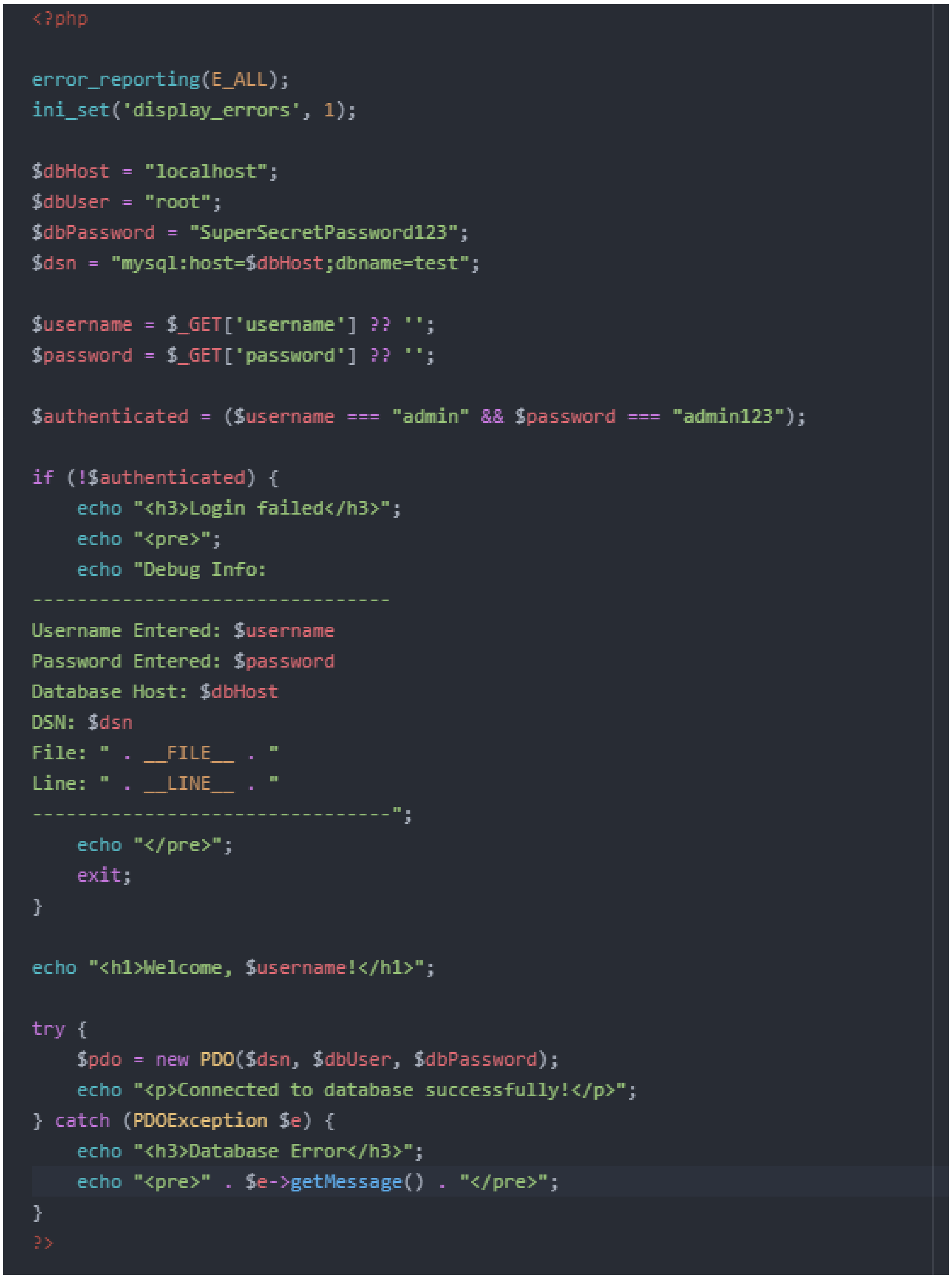

```php
<?php

error_reporting(E_ALL);
ini_set('display_errors', 1);

$dbHost = "localhost";
$dbUser = "root";
$dbPassword = "SuperSecretPassword123";
$dsn = "mysql:host=$dbHost;dbname=test";

$username = $_GET['username'] ?? '';
$password = $_GET['password'] ?? '';

$authenticated = ($username === "admin" && $password === "admin123");

if (!$authenticated) {
    echo "<h3>Login failed</h3>";
    echo "<pre>";
    echo "Debug Info:
--------------------------------
Username Entered: $username
Password Entered: $password
Database Host: $dbHost
DSN: $dsn
File: " . __FILE__ . "
Line: " . __LINE__ . "
--------------------------------";
    echo "</pre>";
    exit;
}

echo "<h1>Welcome, $username!</h1>";

try {
    $pdo = new PDO($dsn, $dbUser, $dbPassword);
    echo "<p>Connected to database successfully!</p>";
} catch (PDOException $e) {
    echo "<h3>Database Error</h3>";
    echo "<pre>" . $e->getMessage() . "</pre>";
}
?>
```

Figure 5: CWE-215 Vulnerable code

Remediation

- Disable Debugging in Production: Always ensure debug code or verbose error reporting is disabled in the production environment. This can be controlled using environment variables or configuration flags (APP_DEBUG=false) [3].
- Remove Sensitive Data from Debug Output: Avoid printing any sensitive data such as user passwords, user tokens, or personally identifiable information in debugging output [3].
- Use Logging Securely: Send debug information to secure logs accessible only to developers or administrators, not to client browsers [3].
- Code Review & Automation: Regularly review code to remove debug statements before deployment and use automated scans to detect sensitive data exposure in debugging code [3].
- Error Handling Practices: Implement generic user-facing error messages that do not leak details about the system internals or sensitive data [3].

3. CWE-234 Failure to Handle Missing Parameter

Information:
CWE-234 occurs when code assumes that required parameters are present and valid but fails to verify them. If a parameter is missing, empty, or malformed, the system may:

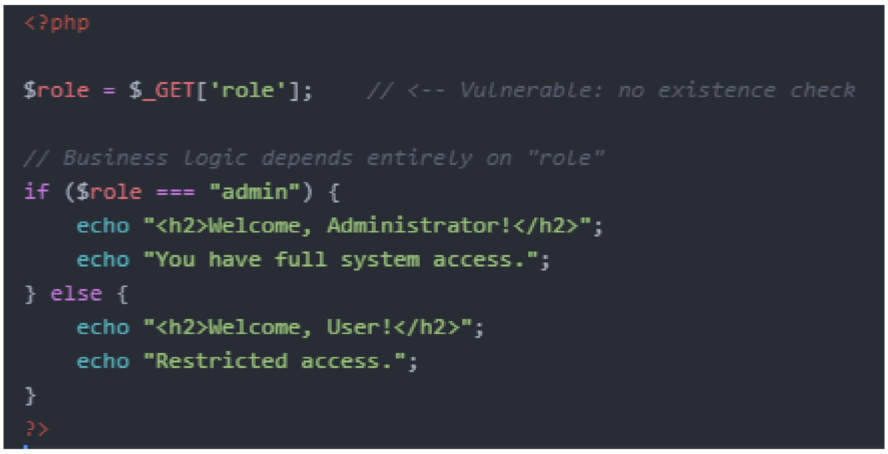
- Use default or unintended values
- Trigger warnings or exceptions
- Skip security checks
- Lead to authentication/authorization bypass
- Cause logic errors or data corruption

Exploitation:

Table 3: Vulnerable Code details

Location	Vulnerability
Parameter is used without checking	$role = $_GET['role']; assumes "role" exists
Missing parameter triggers fallback logic	NULL → user-level access
No error, no logging	Failure is silent and can lead to security bypass
Logic decision is based on missing input	A missing parameter is treated as valid input

This is dangerous because an attacker can simply remove a parameter to alter system behavior.



```php
<?php

$role = $_GET['role'];    // <-- Vulnerable: no existence check

// Business logic depends entirely on "role"
if ($role === "admin") {
    echo "<h2>Welcome, Administrator!</h2>";
    echo "You have full system access.";
} else {
    echo "<h2>Welcome, User!</h2>";
    echo "Restricted access.";
}
?>
```

Figure 6: CWE-234 Vulnerable code

Remediation:

1. Validate required parameters that ensure parameters exist before use
2. Use isset() and !empty(), that prevent accidental NULL/empty inputs
3. Enforce type validation—convert or reject unexpected types
4. Apply server-side validation—never rely on client-side checks
5. Fail securely—Missing parameters should cause controlled error responses
6. Log missing/invalid parameters—helps detect abuse patterns

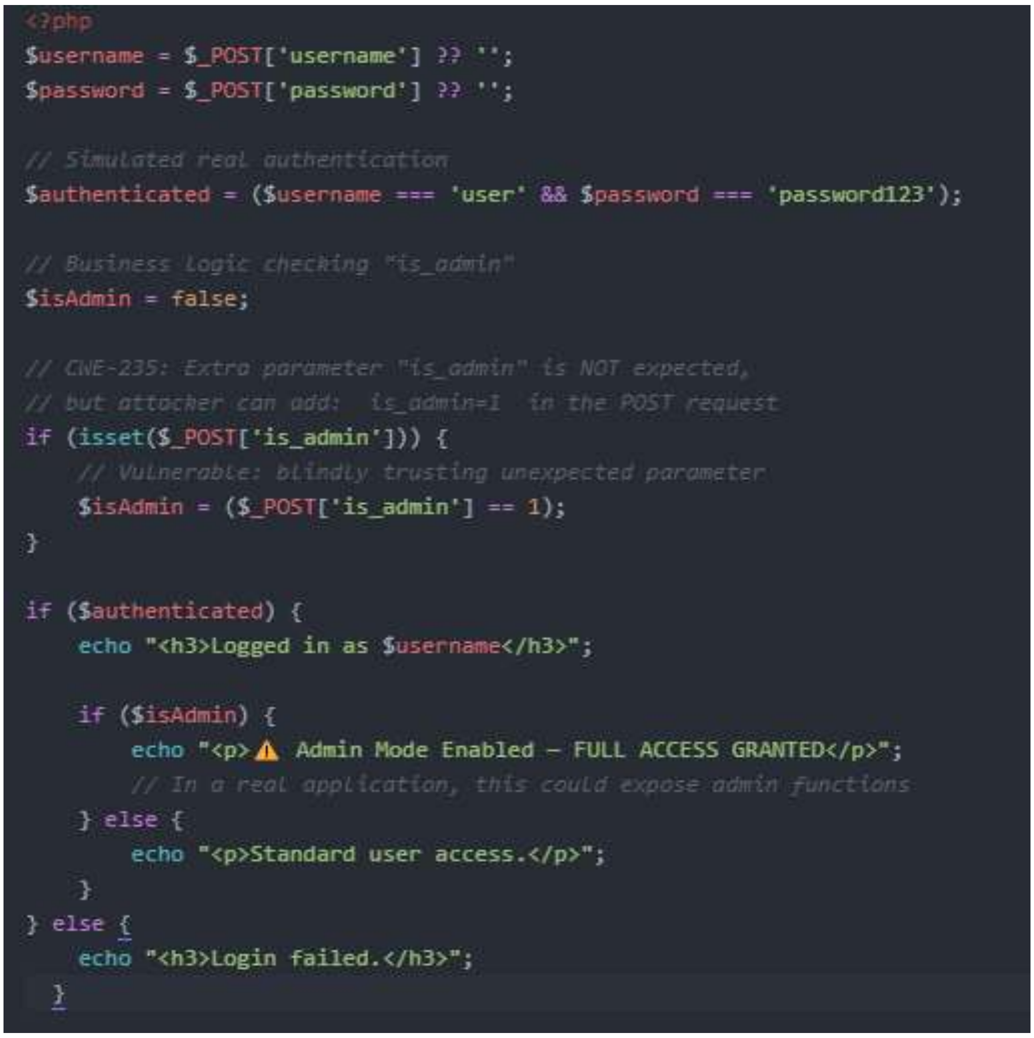
4. CWE-235 Improper Handling of Extra Parameters

Information:
The product does not handle or incorrectly handles when the number of parameters, fields, or arguments with the same name exceeds the expected amount [4]. When software accepts **extra, unexpected, or unvalidated parameters** and silently ignores them, unintentionally **acts on them**, or treats them as valid input [4]. This can lead to security bypass, privilege escalation, logic manipulation, data tampering, and incorrect business behavior. Attackers often add extra parameters to requests to manipulate server logic [4].

Exploitation:

Table 4: Vulnerable Code details

Vulnerability	Description
App expects only username & password	But does not enforce allowed parameter list
Extra parameter is_admin is accepted	Allows attacker to add unauthorized input
Trusting unvalidated parameters	Lets attacker set admin privileges
No input whitelist	Server has no parameter contract or schema



```
<?php
$username = $_POST['username'] ?? '';
$password = $_POST['password'] ?? '';

// Simulated real authentication
$authenticated = ($username === 'user' && $password === 'password123');

// Business logic checking "is_admin"
$isAdmin = false;

// CWE-235: Extra parameter "is_admin" is NOT expected,
// but attacker can add:  is_admin=1  in the POST request
if (isset($_POST['is_admin'])) {
    // Vulnerable: blindly trusting unexpected parameter
    $isAdmin = ($_POST['is_admin'] == 1);
}

if ($authenticated) {
    echo "<h3>Logged in as $username</h3>";

    if ($isAdmin) {
        echo "<p>⚠ Admin Mode Enabled — FULL ACCESS GRANTED</p>";
        // In a real application, this could expose admin functions
    } else {
        echo "<p>Standard user access.</p>";
    }
} else {
    echo "<h3>Login failed.</h3>";
  }
```

Figure 8: CWE-235 Vulnerable code

Remediation:

1. Parameter whitelisting—Accept only expected parameters
2. Input schema validation—Enforce types, names, and constraints
3. Reject unknown parameters—Fail fast with error codes
4. Server-side checking only—Never rely on client-side UI forms
5. Logging unexpected parameters—Helps identify probing/attacks
6. Use frameworks with strong request validation—e.g., Laravel Form Requests, Symfony Validators

5. CWE-248 Uncaught Exception

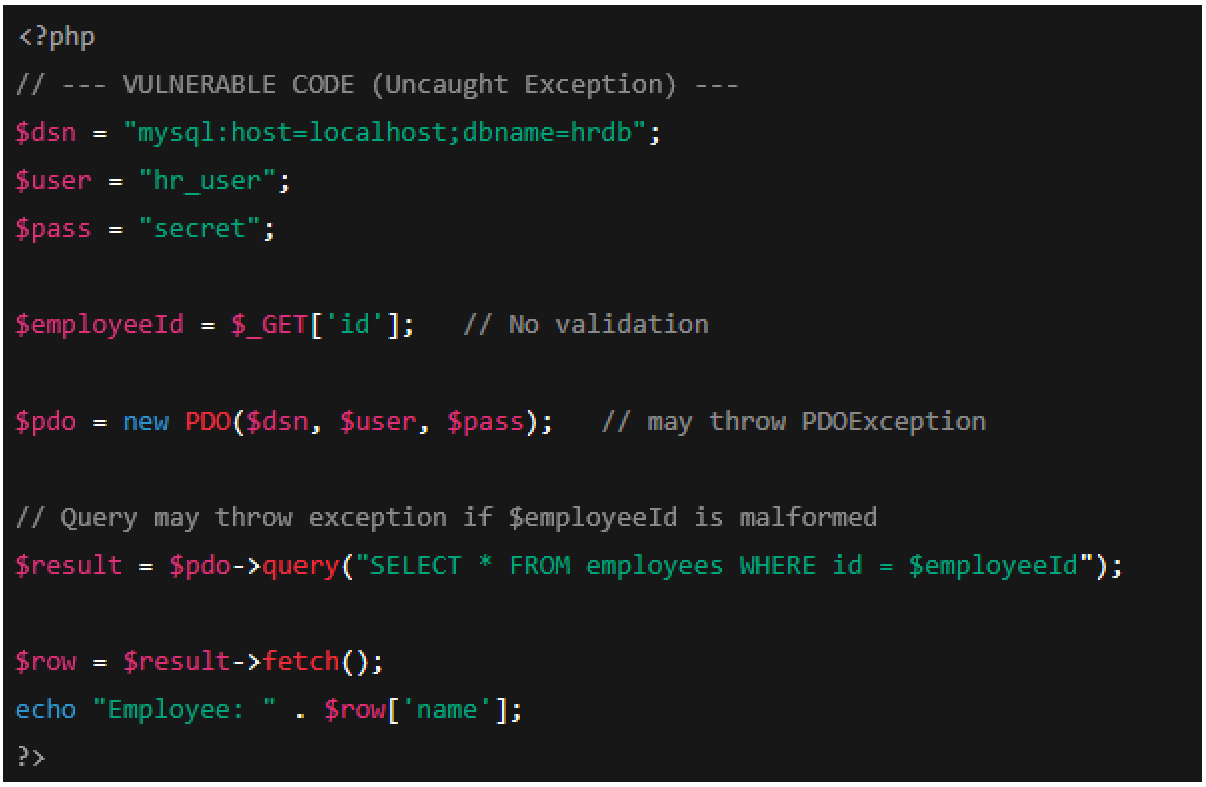
Information:
CWE-248 occurs when an application throws an exception that is not caught or handled, causing a crash, application instability, or information leakage [5]. Uncaught exceptions often expose stack traces, file paths, or sensitive internal configuration [5]. In enterprise environments, this may lead to denial of service, security bypass, or data integrity issues [5].

Exploitation:
Attackers intentionally trigger exceptions by providing:

- Malformed or unexpected input
- Missing or oversized parameters
- Forced database or file I/O failures
- Invalid data types
- Invalid authentication tokens

Uncaught exceptions can cause:

1. Denial of Service (DoS)
2. Information leakage via stack traces
3. Authentication or logic bypass
4. Resource leakage or data corruption



```
<?php
// --- VULNERABLE CODE (Uncaught Exception) ---
$dsn = "mysql:host=localhost;dbname=hrdb";
$user = "hr_user";
$pass = "secret";

$employeeId = $_GET['id'];   // No validation

$pdo = new PDO($dsn, $user, $pass);   // may throw PDOException

// Query may throw exception if $employeeId is malformed
$result = $pdo->query("SELECT * FROM employees WHERE id = $employeeId");

$row = $result->fetch();
echo "Employee: " . $row['name'];
?>
```

Figure 8: CWE-248 Vulnerable code

Problems

- No exception handling
- No input validation
- Query executed directly with potentially malicious input
- Uncaught exceptions → crash + info leak

Remediation:

1. Uncaught exceptions—Wrap code in try/catch, add global exception handler
2. Stack traces exposed—Disable display_errors, enable secure logging
3. DoS via malformed input—Validate and sanitize all input
4. Unhandled PDO exceptions—Use try/catch (PDOException) blocks
5. Resource leakage—Use finally to close connections/files
6. Fail-open behaviors—Fail secure (deny access on exception)

6. CWE-252 Unchecked Return Value

Information:

CWE-252 occurs when software calls a function but does not check the return value, assuming it succeeded [6]. If the function fails and the error is ignored:

- the program continues with invalid or uninitialized data
- security checks may be bypassed
- corrupted state may propagate
- authentication, file operations, or database calls may behave unexpectedly

PHP functions frequently return false, null, or 0 when they fail. Failure to check these returns can lead to dangerous behavior.
Common sources of unchecked return values in PHP:

- file_get_contents()
- fopen(), fwrite(), fclose()
- json_decode()
- password_verify()
- mkdir()
- move_uploaded_file()
- PDO::query()

Exploitation:

Even though an unchecked return value is not a direct vulnerability by itself, attackers exploit it to:

1. Bypass Authentication or Authorization

Example: Developers forget to check if password_verify() returned false.
Attacker supplies any password → authentication flow continues because the result wasn't validated.

2. Trigger Unexpected Application State

Unchecked json_decode() failures allow attackers to send malformed JSON and cause:

- undefined variable usage
- bypass of validation logic
- crashes and inconsistent application behavior

3. Cause File Overwrite or File Upload Issues

Unchecked return values from move_uploaded_file() may result in:

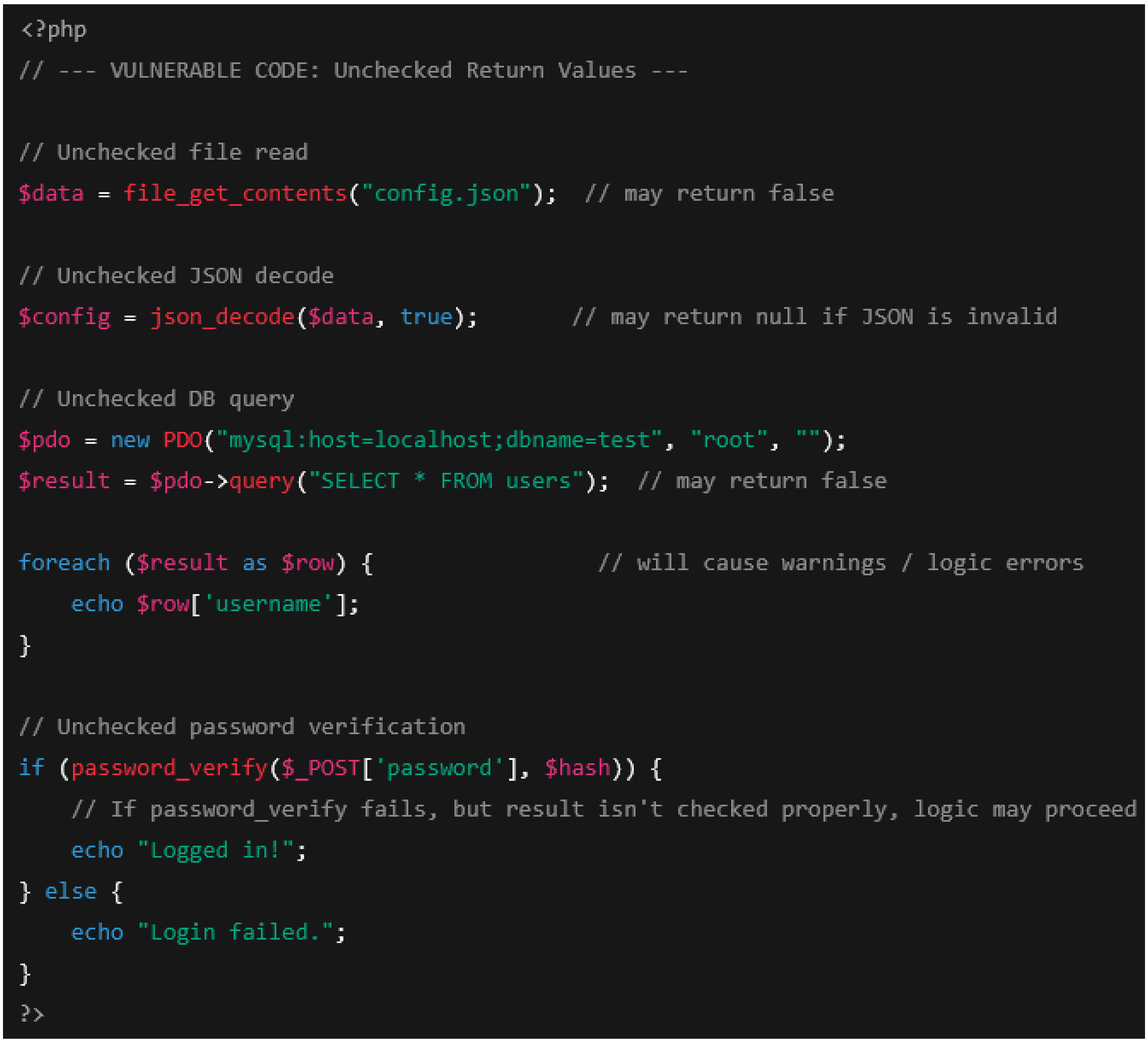
- assumptions that a file was uploaded successfully
- processing of non-existent files
- bypass of upload integrity checks

4. Denial of Service (DoS) Through Error Cascading

When failures are ignored:

- DB connection fails → application still executes logic → fatal errors
- fopen() fails → application attempts to write anyway → crash or loop

Attackers intentionally trigger these failures.



```php
<?php
// --- VULNERABLE CODE: Unchecked Return Values ---

// Unchecked file read
$data = file_get_contents("config.json");  // may return false

// Unchecked JSON decode
$config = json_decode($data, true);       // may return null if JSON is invalid

// Unchecked DB query
$pdo = new PDO("mysql:host=localhost;dbname=test", "root", "");
$result = $pdo->query("SELECT * FROM users");  // may return false

foreach ($result as $row) {                    // will cause warnings / logic errors
    echo $row['username'];
}

// Unchecked password verification
if (password_verify($_POST['password'], $hash)) {
    // If password_verify fails, but result isn't checked properly, logic may proceed
    echo "Logged in!";
} else {
    echo "Login failed.";
}
?>
```

Figure 9: CWE-252 Vulnerable code

Remediation:

- Assuming operations succeed—Always check false, null, empty results
- Missing input validation—Validate before calling sensitive functions
- Not checking json_decode() errors—Verify using json_last_error()
- Not verifying DB query returns—Check for false, use try/catch
- Password verification mishandled—Validate result explicitly
- File operations unchecked—Validate fopen/move_uploaded_file return

7. CWE-274 Improper Handling of Insufficient Privileges

Information:
CWE-274 occurs when an application detects that a user lacks privileges but fails to respond correctly [7].

Instead of blocking the request securely, the application may:

- continue execution
- partially perform the action
- provide sensitive information
- perform a fallback action that unintentionally grants access
- return ambiguous messages that leak information

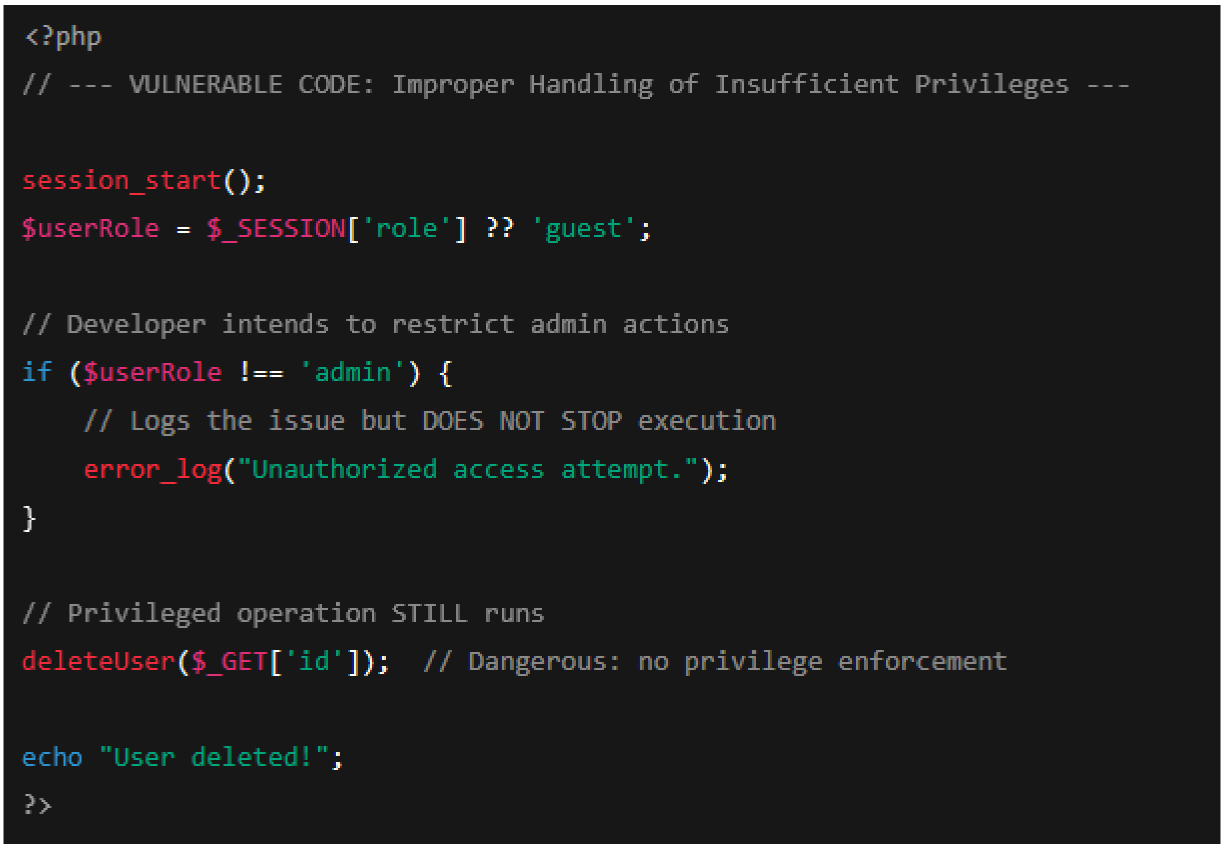
This often happens when:

- privilege checks are optional
- developers "fail open" instead of "fail closed"
- authorization logic is placed after actions
- missing/invalid permission checks do not stop execution

This flaw is commonly found in:

- admin panels
- API endpoints
- file access operations
- payment and billing systems
- RBAC/ACL frameworks
- multi-tenant applications

Exploitation:



```php
<?php
// --- VULNERABLE CODE: Improper Handling of Insufficient Privileges ---

session_start();
$userRole = $_SESSION['role'] ?? 'guest';

// Developer intends to restrict admin actions
if ($userRole !== 'admin') {
    // Logs the issue but DOES NOT STOP execution
    error_log("Unauthorized access attempt.");
}

// Privileged operation STILL runs
deleteUser($_GET['id']);  // Dangerous: no privilege enforcement

echo "User deleted!";
?>
```

Figure 10: CWE-274 Vulnerable code

Application failed to:

- Privilege check logs the problem but **does not return/exit**.
- Unauthorized users can delete accounts.
- No safe failure behavior.

Remediation:

- Privilege failure does not stop action—Unauthorized access, hence use exit() after rejecting
- Logging only, no enforcement—Attack continues, combine log + forbidden response
- Too much information in errors—Leaks RBAC structure, use generic 403 messages
- Privilege checking after action—Partial actions occur, always check before performing
- Missing validation—Input manipulation attacks, Sanitize/validate all input
- Fail-open behaviors—Unintended access, default to deny on errors

8. CWE-280 Improper Handling of Insufficient Permissions or Privileges

Information:
CWE 280 occurs when an application performs an action even though the requester does not have sufficient permissions or when it fails to verify permissions properly [8]. It covers improper handling of both OS permissions and application/service-level privileges, while CWE-274 focuses more on logical authorization flaws inside the application [8].

This CWE is broader than CWE 274 because it includes:
- file system permissions
- OS-level permissions
- Service-level permissions
- database permissions
- application-level authorization

Improper permission handling can lead to unauthorized:
- file access
- file modification or deletion
- gaining shell-level access
- reading sensitive configuration files
- performing restricted business operations
- database manipulation
- privilege escalation

Exploitation:
1. Accessing Files with Weak Permission Checks

If the application assumes the file system permission succeeded or does not validate the user's permissions, an attacker may:
- read logs
- read configuration files
- read uploaded documents belonging to other users
- overwrite or delete files

Example: fopen() succeeds for any user because the app never checks OS-level permissions.

2. Bypassing Application Authorization

If permission checks are weak, misordered, or missing, attackers may access:
- admin panels
- API endpoints
- sensitive business data
- restricted resources

This leads to privilege escalation.

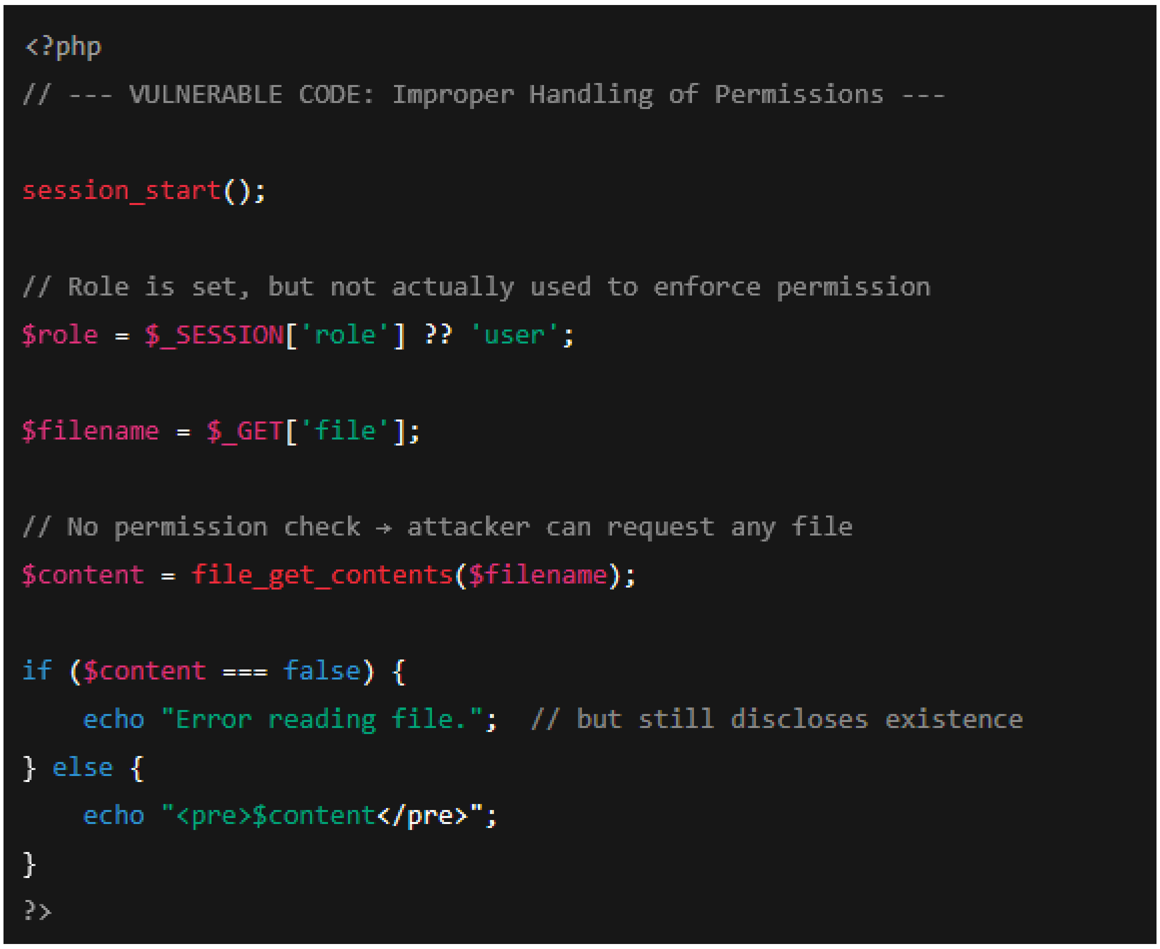
3. Exploiting Fallback or Partial Checks

If a permission check fails but the system:
- uses a default value
- continues execution
- assumes permissive access ("fail open")

Attackers can intentionally cause errors to trigger insecure fallback behavior.

4. Exploiting OS-level Misconfigurations

If the PHP process runs with excessive permissions (e.g., as www-data with write access everywhere), attackers can:
- replace application files
- modify log files
- plant backdoors
- change configuration files



```
<?php
// --- VULNERABLE CODE: Improper Handling of Permissions ---

session_start();

// Role is set, but not actually used to enforce permission
$role = $_SESSION['role'] ?? 'user';

$filename = $_GET['file'];

// No permission check → attacker can request any file
$content = file_get_contents($filename);

if ($content === false) {
    echo "Error reading file.";  // but still discloses existence
} else {
    echo "<pre>$content</pre>";
}
?>
```

Figure 11: CWE-280 Vulnerable code

Remediation:

- No permission enforcement—Implement RBAC/ACL and validate before actions
- Directory traversal—Read sensitive OS/application files; hence, sanitize filename and restrict directory
- Overprivileged PHP account: The attacker gains OS-level access and hence run PHP with minimal filesystem privileges
- Missing checks for file operations—Processing sensitive files blindly is a risk; hence, validate file existence, location, and permissions
- Fail-open behavior—Unauthorized success if permission check fails, always fail securely (deny-by-default)

9. CWE-369 Divide By Zero

Information:

CWE 369 occurs when an application performs a division or modulo operation where the divisor may be zero [9].

Consequences include:

- runtime errors / crashes
- application termination (DoS)
- undefined or inconsistent program behavior
- logic errors, leading to incorrect results

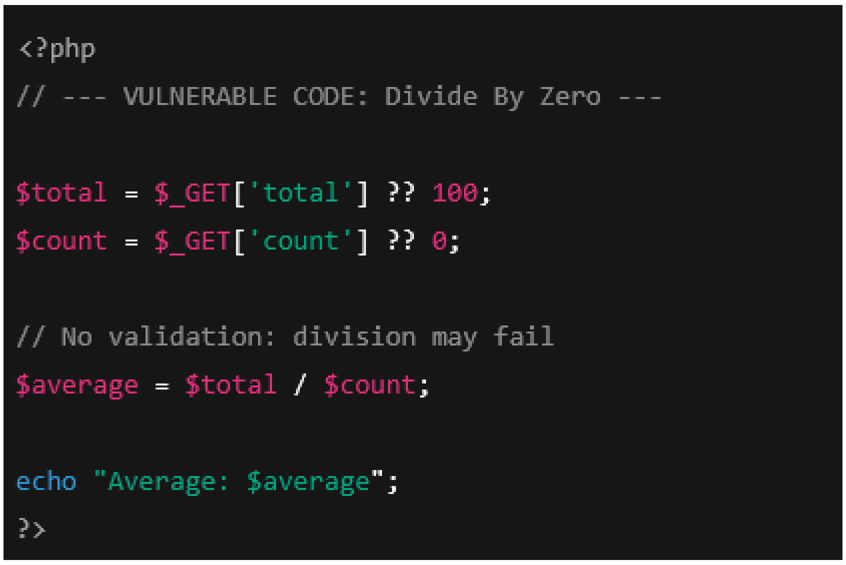
In PHP, division by zero generates a warning (or a fatal error depending on context) and may cause unexpected behavior in calculations, reports, or downstream logic [9].
Common sources:

- user-supplied input used as divisor
- calculations derived from database or file input
- iterative calculations where a counter can reach zero
- modulo operations

Exploitation:

Attackers exploit divide-by-zero errors by providing zero as input for a divisor, potentially triggering:

1. Denial of Service (DoS)
   - Application crashes or stops responding
   - Could be part of an automated attack to disrupt services
2. Incorrect Calculations / Logic Flaws
   - Business logic relying on numeric computation may produce NaN, Infinity, or invalid values
   - Reports, financial calculations, or quotas may be corrupted
3. Chained Failures
   - Downstream functions may rely on the result, causing multiple errors



```php
<?php
// --- VULNERABLE CODE: Divide By Zero ---

$total = $_GET['total'] ?? 100;
$count = $_GET['count'] ?? 0;

// No validation: division may fail
$average = $total / $count;

echo "Average: $average";
?>
```

Figure 12: CWE-369 Vulnerable code

- If count is 0 → PHP generates a warning: Division by zero
- Application may terminate depending on error settings
- Malicious users can intentionally crash the endpoint

Remediation:

if ($average == 0) { throw new InvalidArgumentException("Zero divisor"); }.

10. CWE-390 Detection of Error Condition Without Action

Information:

CWE 390 occurs when an application detects an error condition but takes no meaningful action, effectively ignoring it [9]. Although the application may log the error or silently detect it, failing to handle the error can lead to:

- incorrect program state
- security bypass or logic errors
- resource leaks
- data corruption
- system instability

This flaw is often caused by:

- empty catch blocks
- checking a return value but not responding
- logging without corrective measures
- failing to halt execution or notify the user

Exploitation:
Attackers can exploit CWE-390 indirectly by:

1. Triggering logic errors
   - Application continues despite a failed operation
   - Example: authentication failure is detected but ignored → unauthorized access
2. Resource exhaustion or data corruption
   - Ignored I/O errors or DB failures may corrupt files or database tables
3. Security bypass
   - If privilege checks fail but are ignored, attackers gain access

Remediation:
To remediate CWE-390, apply the following principles:

1. Take Meaningful Action
   - Handle errors explicitly
   - Options include: retry, fail securely, alert, or halt execution
2. Log Errors Securely
   - Always log errors for auditing
   - Avoid revealing sensitive info to users
3. Fail Securely
   - Stop execution when critical errors occur
   - Use safe defaults if needed
4. Provide User Feedback (Optional)
   - Inform users in a generic, safe way
   - Avoid exposing internal state
5. Use Exception Handling
   - Wrap error-prone operations in try/catch blocks
   - Act on exceptions rather than ignoring them

11. CWE-391 Unchecked Error Condition

Information:
CWE 391 occurs when an application performs operations that may fail but does not check for failure, whereas in CWE 390 (where the error is detected but ignored), CWE 391 involves not even detecting the error [9].
Consequences include:

- application continues with invalid or unexpected data
- security controls may be bypassed
- resource leaks (files, DB connections, memory)
- corrupted output or database state
- potential Denial of Service (DoS)

Common causes in PHP:

- calling file, database, or network functions without verifying return values
- using functions that can return false or null without checking
- assuming operations always succeed (fail-open assumption)

Examples:

- file_get_contents($file) → may return false if file missing
- fopen($filename, "r") → may return false
- PDO::query($sql) → may return false on SQL error

Exploitation:
Attackers can exploit unchecked error conditions by:

1. Supplying invalid input
   - e.g., malformed file names or SQL queries → application continues with invalid results
2. Triggering resource failures
   - e.g., full disk, unavailable DB → application continues unaware of failures
3. Bypassing security logic
   - If a function that verifies authorization fails but the return is unchecked, the attacker may gain unauthorized access
4. Causing corruption or crash
   - Unchecked errors propagate → corrupted output or fatal errors

Example Code:

```
<?php
$fp = fopen($_GET['file'], 'r');
// No check if $fp is false
$data = fread($fp, filesize($_GET['file']));
echo $data;
?>
```

In the above example, if fopen() fails → $fp is false → fread() fails → application may expose warnings or crash.

Remediation

1. Always check return values
   - Verify whether functions succeed or fail
   - Example: check false, null, or empty return
2. Use exceptions where possible
   - Wrap error-prone operations in try/catch blocks
   - Handle exceptions explicitly
3. Fail securely
   - Stop execution when a critical operation fails
   - Provide safe default values if appropriate
4. Log and alert
   - Log failures for auditing
   - Optionally alert administrators on critical failures
5. Validate inputs before use
   - Reduce likelihood of triggering errors

12. CWE-394 Unexpected Status Code or Return Value

Information:

CWE-394 occurs when an application receives a status code, return value, or error code from a function, API, or service but fails to handle it properly, assuming the operation succeeded or ignoring unexpected values [10].

Consequences include:

- logic errors or incorrect processing
- security bypass (e.g., unauthorized access)
- resource leaks or incomplete transactions
- corrupted data
- system instability

Common sources in PHP applications:

- HTTP API calls (e.g., curl_exec()) returning HTTP error codes
- Database operations returning error codes (e.g., PDO::errorCode())
- System or OS-level function return values (mkdir(), unlink(), file_put_contents())
- Library functions returning status codes or boolean success/failure values

Exploitation:

Attackers can exploit unexpected status codes or return values by:

1. Sending inputs that trigger failures
   - For example, malformed requests, SQL injections, or missing parameters
2. Bypassing security checks
   - If the application assumes a function succeeds when it actually returns an error, attackers may gain unauthorized access
3. Causing data corruption
   - Ignored failures during DB insert/update/delete operations can result in inconsistent or incomplete data
4. Triggering DoS or crashes
   - Unhandled negative return codes from critical functions can terminate scripts unexpectedly

Example Code:

```
<?php
        $data = $_POST['data'];
$result = file_put_contents("/var/data/file.txt", $data);
// Return value not checked
echo "File saved successfully!";
?>
```

In the above example, If file_put_contents() fails (e.g., permissions issue, disk full) → application incorrectly reports success and users or other functions may rely on the incorrect assumption.

Remediation:

1. Always check return values and status codes
   - Example: verify HTTP status, boolean success/failure, or numeric return codes
2. Fail securely
   - If an operation fails, halt dependent operations, provide safe defaults, or return an error
3. Log all unexpected status codes
   - Maintain auditing and traceability
4. Use exceptions where possible
   - Wrap critical operations in try/catch and handle exceptions explicitly
5. Validate dependent operations
   - Only proceed if the previous step returned the expected success value

13. CWE-396 Declaration of Catch for Generic Exception

Information:
Exploitation:
Remediation:

14. CWE-460 Improper Cleanup on Thrown Exception

Information:
CWE-460 occurs when an application throws an exception but fails to clean up resources or restore application state properly [11]. When an exception occurs:

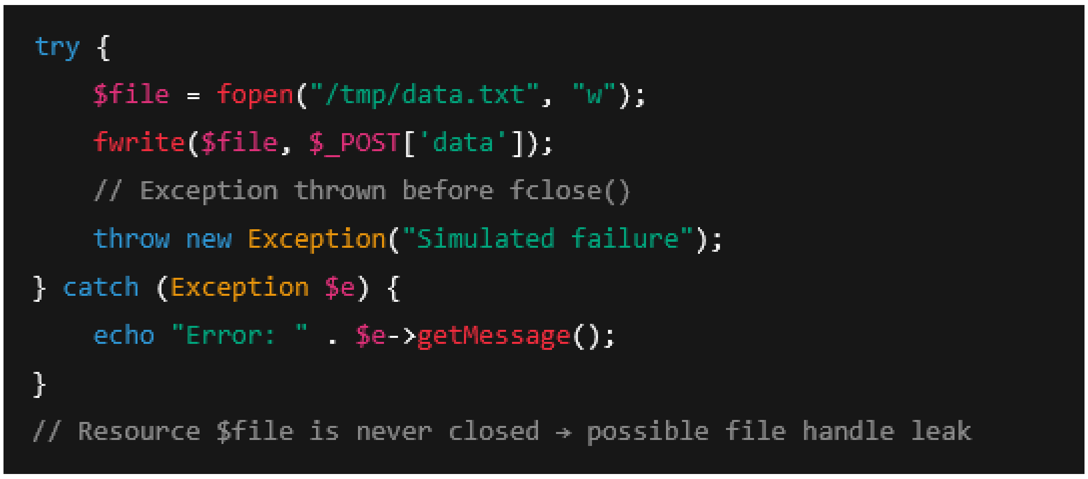
- open files, sockets, or database connections may remain open
- temporary data may remain unremoved
- locks may not be released
- memory leaks may occur
- partial or inconsistent operations can leave the system in an invalid state

In PHP applications, this often happens when:

- try/catch blocks fail to properly close resources
- temporary files or sessions are not cleaned
- database transactions are not rolled back
- code relies on side effects outside the exception handling

Exploitation:
Attackers may exploit improper cleanup by:

1. Resource exhaustion
   - Leaving file handles, sockets, or DB connections open → DoS
2. Data leakage
   - Temporary or sensitive files not removed
3. Logic bypass
   - Partial operations left unrolled can be manipulated to bypass business logic
4. State corruption
   - Transactions or in-memory structures remain inconsistent, allowing subsequent operations to fail or behave incorrectly



```php
try {
    $file = fopen("/tmp/data.txt", "w");
    fwrite($file, $_POST['data']);
    // Exception thrown before fclose()
    throw new Exception("Simulated failure");
} catch (Exception $e) {
    echo "Error: " . $e->getMessage();
}
// Resource $file is never closed → possible file handle leak
```

Figure 13: CWE-460 Vulnerable code

Remediation:

1. Always clean up resources in a finally block
   - Ensure files, connections, and locks are released regardless of exceptions
2. Use context managers or automatic resource handlers
   - PHP 8+ supports objects with destructors that clean up
3. Rollback transactions on failure
   - Always ensure database or transactional resources revert to a consistent state
4. Delete temporary files or sensitive data
   - Ensure temporary storage is removed even on exceptions
5. Fail securely
   - Prevent partial operations from being committed if an exception occurs

15. CWE-476 NULL Pointer Dereference

Information:
CWE-476 occurs when an application dereferences a pointer or variable that may be NULL (or undefined in PHP terms) [12].

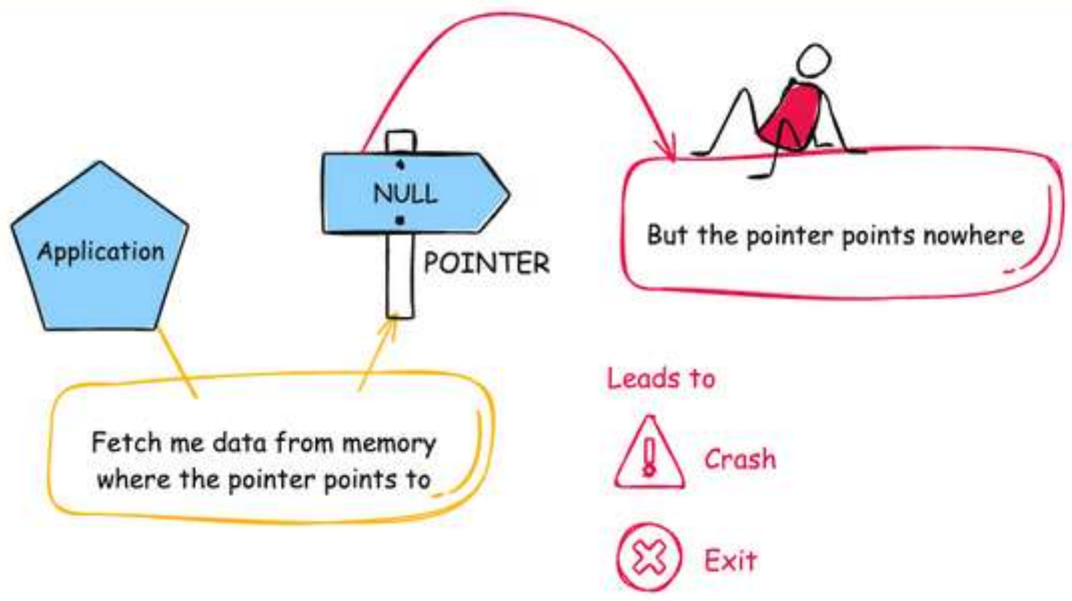

Application
NULL
POINTER
But the pointer points nowhere
Fetch me data from memory where the pointer points to
Leads to
Crash
Exit

Figure 14: vulnerable scenario

Dereferencing NULL can lead to:

- application crashes
- runtime errors or warnings

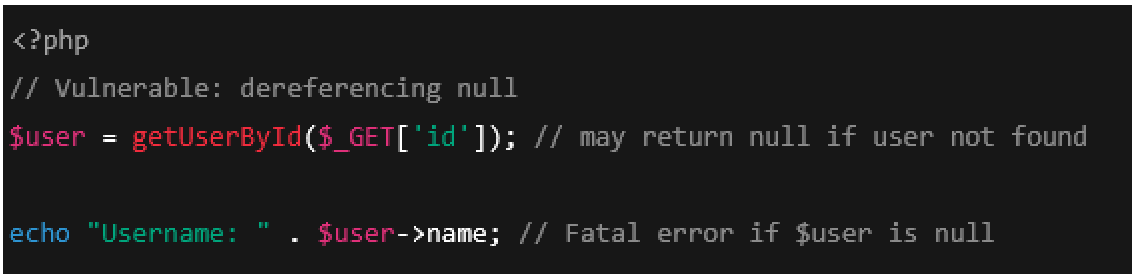
- logic failures
- Denial of Service (DoS)

In PHP, NULL pointer dereferences typically appear as:

- calling a method on a variable that is null
- accessing properties of an object that has not been instantiated
- using [] array operations on null

Exploitation:
Attackers can exploit NULL pointer dereference by:

1. Providing unexpected input
   - If user input is assigned to a variable, attackers can supply null or omit the input entirely
2. Triggering application crash or warnings
   - Example: Call to a member function on null → may reveal stack trace or sensitive info
3. Bypassing logic checks
   - Certain code paths may skip authorization or validation when a NULL occurs



```
<?php
// Vulnerable: dereferencing null
$user = getUserById($_GET['id']); // may return null if user not found

echo "Username: " . $user->name; // Fatal error if $user is null
```

Figure 15: CWE-476 Vulnerable code

Remediation:

- Null check: if ($user) { echo $user->name ?? 'Unknown'; }.
- Validate input early—Ensure functions returning objects or arrays never return null unless explicitly allowed
- Fail securely - Stop execution or return safe defaults when NULL is detected

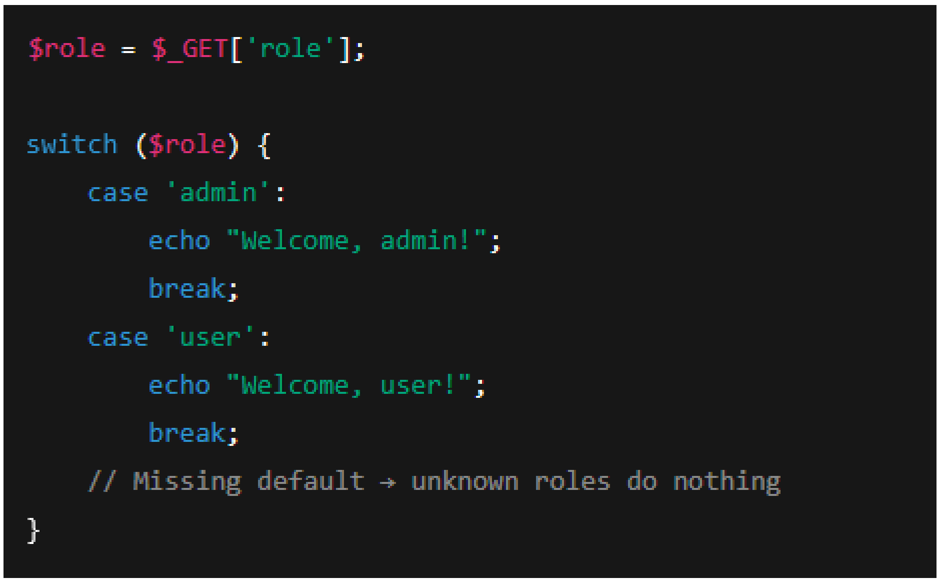
16. CWE-478 Missing Default Case in Multiple Condition Expression

Information:
CWE-478 occurs when a switch or multiple-condition expression does not include a default case [13].
Without a default case:

- unexpected input may be ignored
- application may behave unpredictably
- logic errors or security bypass may occur
- unhandled cases can leave the system in an invalid state

Exploitation:
Attackers can exploit a missing default case by:

1. Providing unexpected input
   - Example: passing $_GET['role'] as "superadmin" → system may bypass access checks
2. Triggering logic bypass
   - Unhandled cases may skip security enforcement
3. Causing incorrect or inconsistent state
   - Application assumes only known cases exist, leading to undefined behavior
4. Information leakage
   - In some scenarios, unhandled cases may reveal hints about internal logic



```php
$role = $_GET['role'];

switch ($role) {
    case 'admin':
        echo "Welcome, admin!";
        break;
    case 'user':
        echo "Welcome, user!";
        break;
    // Missing default → unknown roles do nothing
}
```

Figure 16: CWE-478 Vulnerable code

If attacker sends /script.php?role=guest → no output, may bypass intended handling or cause silent failure

Remediation:

- Add default: throw new Exception("Invalid action");.
- Fail securely—Treat unknown or unexpected values as unauthorized or invalid
- Validate input before switch—ensure $role contains only expected values
- Log unexpected cases—helps auditing and detection of malicious activity

17. CWE-484 Omitted Break Statement in Switch

Information:
CWE-484 occurs when a switch statement in code omits a break or return statement at the end of a case, causing fall-through to subsequent cases [14].

Consequences include:

- unintended execution of multiple cases
- logic errors or security bypass
- incorrect application behavior
- potential data corruption

Exploitation:
Attackers can exploit omitted break statements by:

1. Triggering fall-through logic
   - Unexpected operations execute for inputs not intended for that case
2. Bypassing authorization checks
   - Elevated privileges may be inadvertently granted
3. Causing unintended side effects
   - Database updates, file operations, or function calls execute multiple times

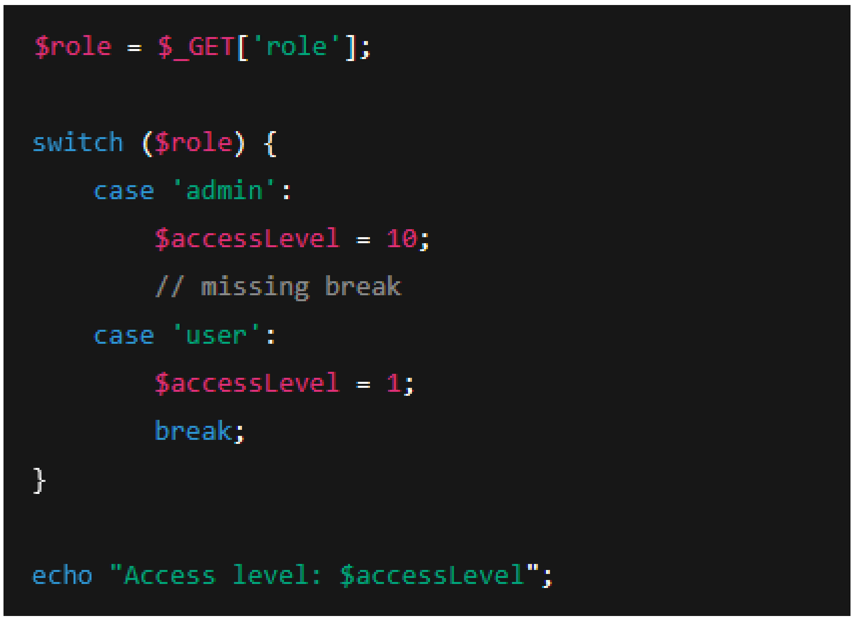

```
$role = $_GET['role'];

switch ($role) {
    case 'admin':
        $accessLevel = 10;
        // missing break
    case 'user':
        $accessLevel = 1;
        break;
}

echo "Access level: $accessLevel";
```

Figure 17: CWE-484 Vulnerable code

If attacker sends role=admin → $accessLevel is overwritten to 1 instead of 10 → logic error

Remediation:

- Add break; after each case unless intentional.
- Use default for unexpected cases—ensures unknown values are handled safely
- Consider using if/else when fall-through is unintended—makes logic explicit and reduces human error
- Enable static analysis / code linters—Tools can warn about missing break statements

18. CWE-550 Server-generated Error Message Containing Sensitive Information

Information:
CWE-550 occurs when a server returns error messages that include sensitive information [15].
Such as:

- stack traces
- database errors
- file paths
- API keys or credentials
- configuration details

Exposing this information can:

- reveal internal architecture
- provide attackers with clues for exploitation
- compromise sensitive data
- facilitate further attacks like SQL injection, path traversal, or privilege escalation

In PHP applications, this often occurs when error reporting is enabled in production:

- ini_set('display_errors', 1);
- error_reporting(E_ALL);

Exploitation:
Attackers can exploit sensitive error messages to:

1. Gain insights into internal implementation
   - Stack traces reveal function names, file paths, or line numbers
2. Facilitate SQL Injection or XSS
   - Database errors disclose table names, column names, or queries
3. Target specific vulnerabilities
   - Knowing PHP version, libraries, or framework details can help in crafting attacks

Remediation:

- Use generic messages for end users - Avoid exposing stack traces, database messages, or file paths
- Do not display detailed errors in production - ini_set('display_errors', 0); & error_reporting(E_ALL);

- Use error_log() or logging libraries to record errors for developers
- Sanitize error output - Never concatenate sensitive internal information into user-facing messages
- Implement global exception handling - Catch all unhandled exceptions and provide safe responses

19. CWE-636 Not Failing Securely ('Failing Open')

Information:
CWE-636 occurs when an application encounters an error or unexpected condition—but instead of failing safely, it allows the operation to continue, often granting access or allowing improper behavior [16]. This is called failing open, the opposite of failing closed (which defaults to denial) [16].

Fail-open behavior can occur when:

- authentication or authorization systems return unexpected values
- input validation fails
- configuration files fail to load
- security checks encounter errors
- external dependencies (API, DB, LDAP, OAuth) return failures

When the system cannot determine if access *should* be allowed, it defaults to allow instead of deny, creating a severe security risk.

Exploitation:

1. Authorization Failures

If the permission/role system fails (DB error, missing value), and the system defaults to *grant access*, attackers can escalate privileges.

Example:
```
$role = getUserRole($userId); // returns null due to DB error
if (!$role) {
  $role = 'admin'; // developer fallback (fail-open)
}
```

2. Authentication Bypass

If external authentication (OAuth, SSO, LDAP) fails and the system treats the user as authenticated:

- attacker gets access without valid credentials
- system assumes user is logged in

3. Validation Failures

If input validation fails and the system proceeds anyway:

- SQL injection
- path traversal
- XSS
- unauthorized file access

4. Configuration Errors

If the application cannot load:

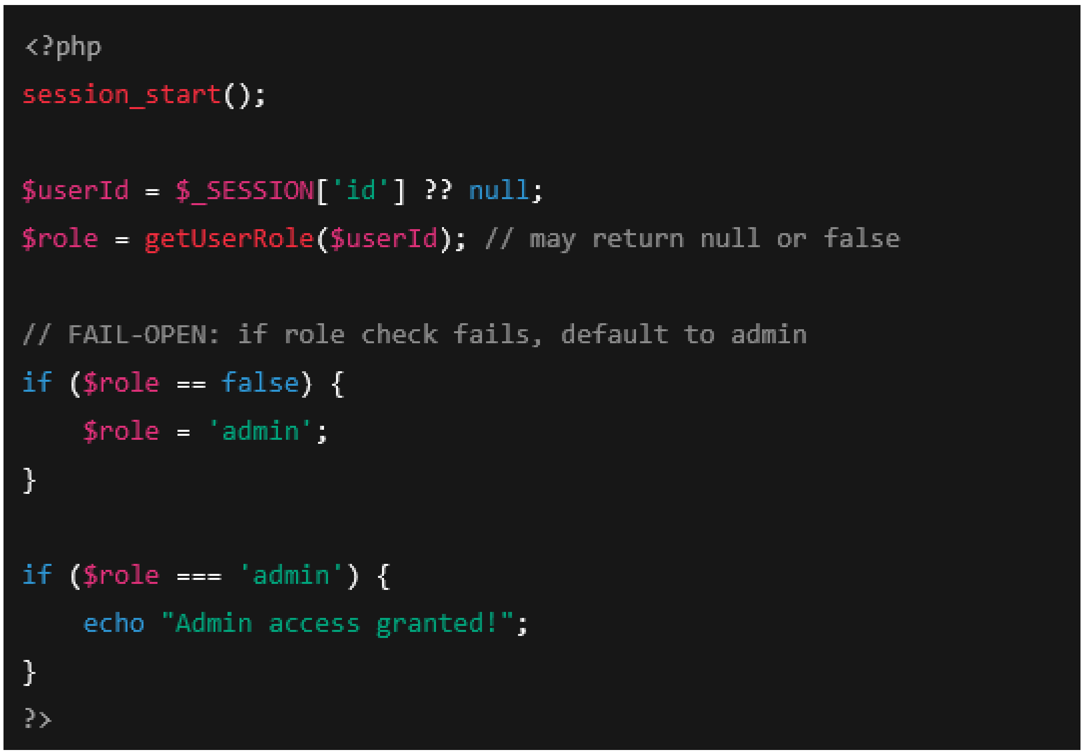
- ACL rules
- API keys
- security settings

5. Denial-of-Service–Based Bypass

Attackers force failures intentionally:

- flood authentication servers
- send malformed JSON / headers
- tamper with cookies

If the system fails open → attacker gains access.



```php
<?php
session_start();

$userId = $_SESSION['id'] ?? null;
$role = getUserRole($userId); // may return null or false

// FAIL-OPEN: if role check fails, default to admin
if ($role == false) {
    $role = 'admin';
}

if ($role === 'admin') {
    echo "Admin access granted!";
}
?>
```

Figure 18: CWE-636 Vulnerable code

- If getUserRole() fails (DB down / missing user), attacker gets *admin* access.
- This is a classic fail-open privilege escalation.

Remediation:

1. Fail Closed (Deny by Default), If a role or credential cannot be determined → deny access.
2. Validate All External Data: Authorization data must be trusted (never default to permissive).
3. Handle Authentication/Authorization Errors Explicitly
4. Implement Default Deny Rules in Access Control, ACL, or RBAC systems should explicitly deny unknown states.
5. Log All Failures—Administrators must be aware of authorization or validation failures.

20. CWE-754 Improper Check for Unusual or Exceptional Conditions

Information:
CWE-754 occurs when an application fails to correctly detect or handle unusual, unexpected, or exceptional conditions [17]. This often results in:

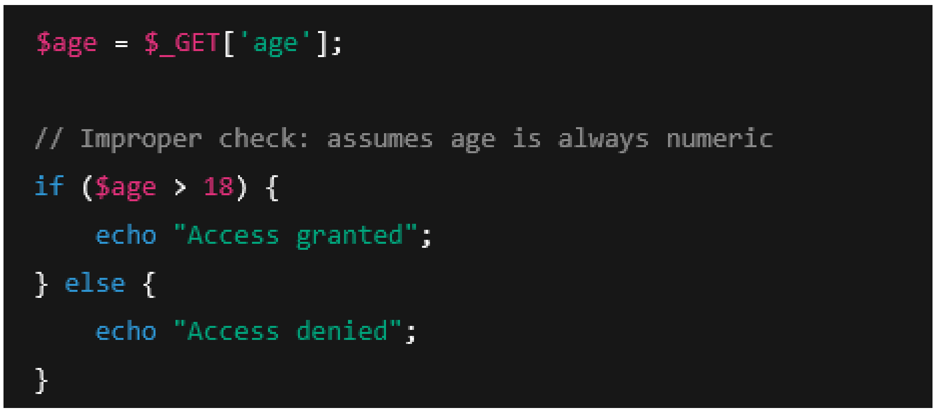
- the system continuing under unsafe assumptions
- security bypass
- incorrect logic or invalid state
- crashes or Denial of Service (DoS)

In PHP applications, common scenarios include:

- improperly checking for error codes or return values
- assuming external inputs are always valid
- failing to validate file operations, network connections, or user-supplied data
- ignoring or misinterpreting exception conditions

Exploitation:
Attackers can exploit improper checks for unusual conditions by:

1. Providing malformed input
    - Example: empty strings, nulls, or unexpected data types
    - Application logic may fail silently or behave incorrectly
2. Triggering edge cases
    - Overflow, divide-by-zero, negative array indices, invalid dates
3. Bypassing security logic
    - If an authorization function fails to handle exceptional conditions, attackers may gain unauthorized access
4. Causing application instability
    - Crashes, unhandled exceptions, or inconsistent application state



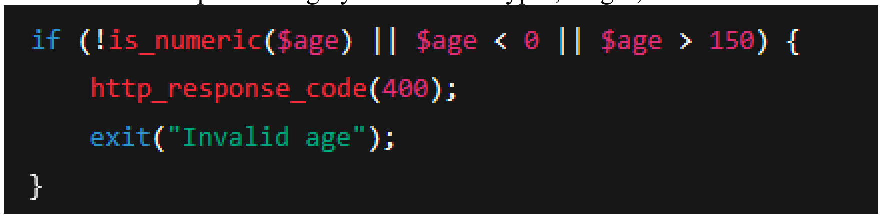
```php
$age = $_GET['age'];

// Improper check: assumes age is always numeric
if ($age > 18) {
    echo "Access granted";
} else {
    echo "Access denied";
}
```

Figure 19: CWE-754 Vulnerable code

- If attacker sends /script.php?age=abc → PHP may coerce to 0 → access denied
- But other logic relying on $age as numeric may fail unpredictably

Remediation:

1. Validate input thoroughly—check data types, ranges, and formats



```php
if (!is_numeric($age) || $age < 0 || $age > 150) {
    http_response_code(400);
    exit("Invalid age");
}
```

Figure 20: CWE-754 Fixed code

2. Check for all possible exceptional return values—database queries, file operations, API calls, or function results
3. Handle unexpected states securely—fail closed (deny access) or return safe defaults if unusual conditions are detected
4. Use exception handling—wrap error-prone operations in try/catch and act on exceptions
5. Log unusual conditions - Maintain audit trail for exceptional or abnormal inputs

V. CONCLUSION

The analysis of CWEs such as CWE-248 (Uncaught Exception), CWE-252 (Unchecked Return Value), CWE-369 (Divide by Zero), CWE-550 (Server-generated Error Messages Containing Sensitive Information), CWE-636 (Failing Open), and CWE-754 (Improper Check for Unusual or Exceptional Conditions) highlights a persistent challenge in secure software development: robust error handling and validation are often overlooked, even in enterprise-grade applications.

The patterns observed across these vulnerabilities indicate that:

1. Unchecked assumptions—about return values, inputs, or external conditions—frequently lead to security weaknesses.
2. Improper exception and resource handling is a primary vector for both Denial of Service (DoS) and information disclosure.
3. Fail-open behaviors and missing edge case handling can unintentionally grant unauthorized access or escalate privileges.
4. Lack of secure coding discipline, such as missing default cases in control structures or absent logging, exacerbates the risk of exploitation.
5. PHP-specific patterns (e.g., null dereferences, unchecked function results, and unhandled database errors) demonstrate that even dynamic, loosely typed languages are prone to subtle yet critical vulnerabilities when defensive coding practices are ignored.

These observations reinforce the importance of systematic validation, fail-secure defaults, and comprehensive exception handling. They also suggest that static analysis, automated testing, and secure coding education are essential tools in mitigating such vulnerabilities.

Future research could focus on:

- Automated detection of fail-open logic and unchecked error conditions in PHP and similar languages
- Best practices for resource management and exception safety in web applications
- Empirical studies of vulnerability prevalence in real-world enterprise systems to inform targeted mitigation strategies
- Detailed analysis for other programming languages such as Java, Python, .Net, etc.

Overall, the collective insights from these CWEs underscore that robust, secure error handling is not just a best practice but a fundamental requirement for resilient and trustworthy software systems.

REFERENCES

[1] "A10 Mishandling of Exceptional Conditions - OWASP Top 10:2025 RC1." Accessed: Nov. 22, 2025. [Online]. Available: https://owasp.org/Top10/2025/A10_2025-Mishandling_of_Exceptional_Conditions/
[2] "CWE - CWE-209: Generation of Error Message Containing Sensitive Information (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/209.html
[3] "CWE - CWE-215: Insertion of Sensitive Information Into Debugging Code (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/215.html
[4] "CWE - CWE-235: Improper Handling of Extra Parameters (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/235.html
[5] "CWE - CWE-248: Uncaught Exception (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/248.html
[6] "CWE - CWE-252: Unchecked Return Value (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/252.html
[7] "CWE - CWE-274: Improper Handling of Insufficient Privileges (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/274.html
[8] "CWE - CWE-280: Improper Handling of Insufficient Permissions or Privileges (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/280.html
[9] "CWE - CWE-391: Unchecked Error Condition (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/391.html
[10]"CWE - CWE-394: Unexpected Status Code or Return Value (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/394.html
[11]"CWE - CWE-460: Improper Cleanup on Thrown Exception (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/460.html
[12]"CWE - CWE-476: NULL Pointer Dereference (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/476.html
[13]"CWE - CWE-478: Missing Default Case in Multiple Condition Expression (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/478.html
[14]"CWE - CWE-484: Omitted Break Statement in Switch (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/484.html
[15]"CWE - CWE-550: Server-generated Error Message Containing Sensitive Information (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/550.html
[16]"CWE - CWE-636: Not Failing Securely ('Failing Open') (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/636.html
[17]"CWE - CWE-754: Improper Check for Unusual or Exceptional Conditions (4.18)." Accessed: Nov. 23, 2025. [Online]. Available: https://cwe.mitre.org/data/definitions/754.html