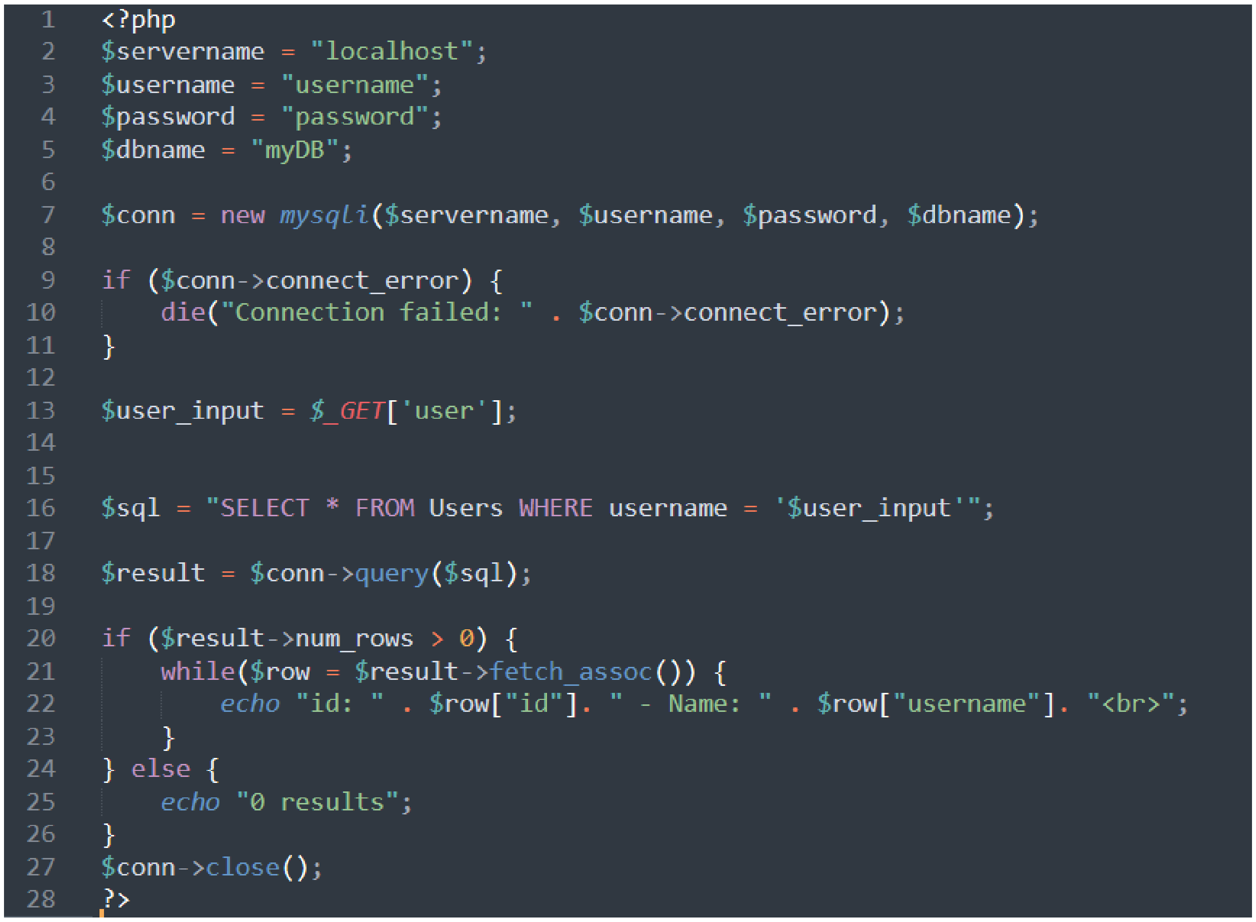
VULNERBILITY ASSESMENTS

This document explores common security vulnerabilities mainly in PHP web applications. Vulnerable code snippets are presented alongside secure alternatives

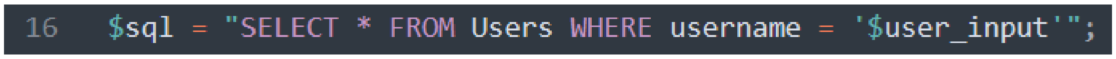
Secure coding reviews

❖ Code Vulnerable to SQL injection:



```php
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

$conn = new mysqli($servername, $username, $password, $dbname);

if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$user_input = $_GET['user'];


$sql = "SELECT * FROM Users WHERE username = '$user_input'";

$result = $conn->query($sql);

if ($result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        echo "id: " . $row["id"]. " - Name: " . $row["username"]. "<br>";
    }
} else {
    echo "0 results";
}
$conn->close();
?>
```

Vulnerable line:

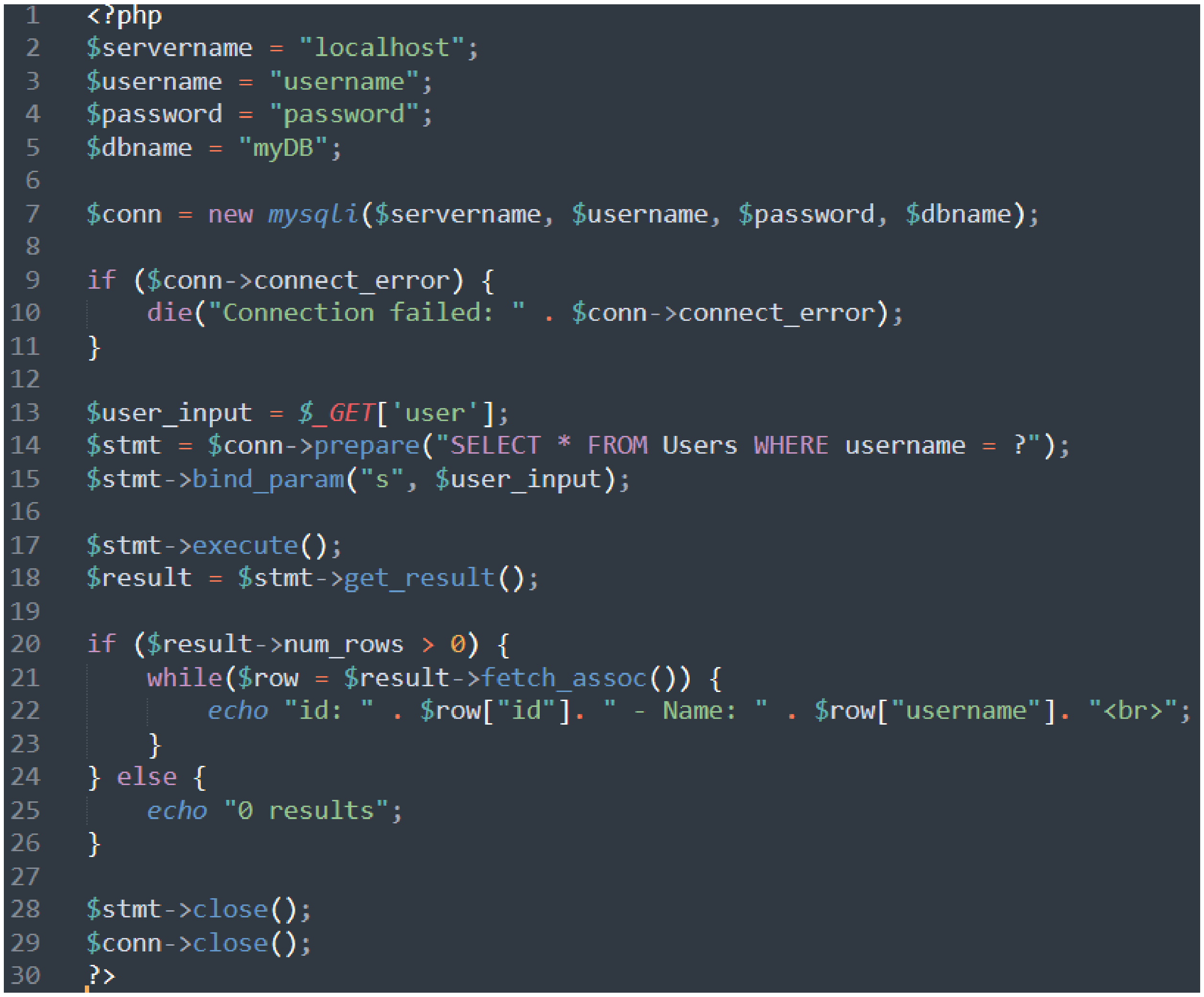


```php
$sql = "SELECT * FROM Users WHERE username = '$user_input'";
```

This line directly incorporates user input into the SQL query without any sanitization or validation. This means that an attacker can manipulate the input to inject malicious SQL code, potentially compromising the security and integrity of the database.

The **$user_input** variable, which comes from a user-provided GET parameter, is directly included in the SQL query string. If this input is not properly sanitized, an attacker can manipulate it.

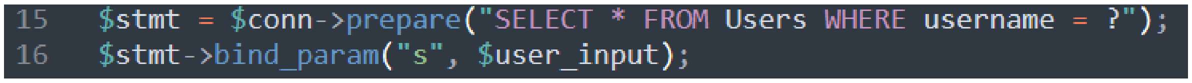
Resolved Code:



```php
<?php
$servername = "localhost";
$username = "username";
$password = "password";
$dbname = "myDB";

$conn = new mysqli($servername, $username, $password, $dbname);

if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$user_input = $_GET['user'];
$stmt = $conn->prepare("SELECT * FROM Users WHERE username = ?");
$stmt->bind_param("s", $user_input);

$stmt->execute();
$result = $stmt->get_result();

if ($result->num_rows > 0) {
    while($row = $result->fetch_assoc()) {
        echo "id: " . $row["id"]. " - Name: " . $row["username"]. "<br>";
    }
} else {
    echo "0 results";
}

$stmt->close();
$conn->close();
?>
```

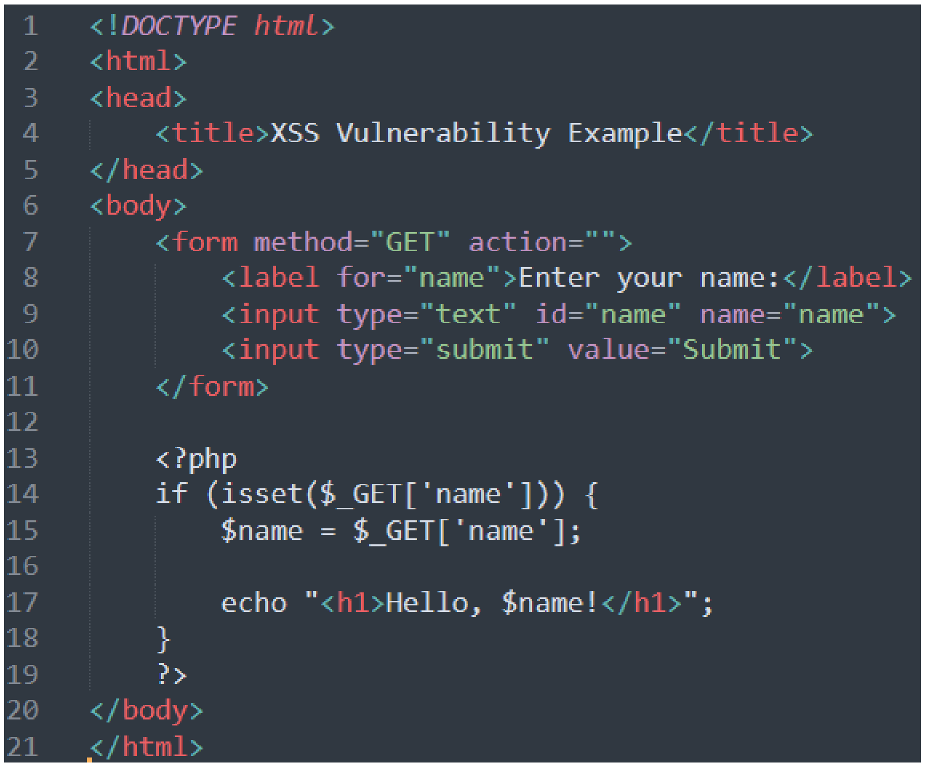
Prepared Statements separate the SQL query structure from the data, ensuring that the database treats user input strictly as data. This prevents attackers from injecting malicious SQL code into the query.

Parametrized Statements involve using placeholders (**?**) for user inputs in SQL statements. These placeholders are later replaced by the actual data provided by the user. This approach ensures that the user input cannot alter the SQL query structure.



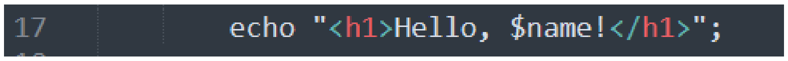
```php
$stmt = $conn->prepare("SELECT * FROM Users WHERE username = ?");
$stmt->bind_param("s", $user_input);
```

❖ Code Vulnerable to XSS injection:



```
<!DOCTYPE html>
<html>
<head>
    <title>XSS Vulnerability Example</title>
</head>
<body>
    <form method="GET" action="">
        <label for="name">Enter your name:</label>
        <input type="text" id="name" name="name">
        <input type="submit" value="Submit">
    </form>

    <?php
    if (isset($_GET['name'])) {
        $name = $_GET['name'];

        echo "<h1>Hello, $name!</h1>";
    }
    ?>
</body>
</html>
```

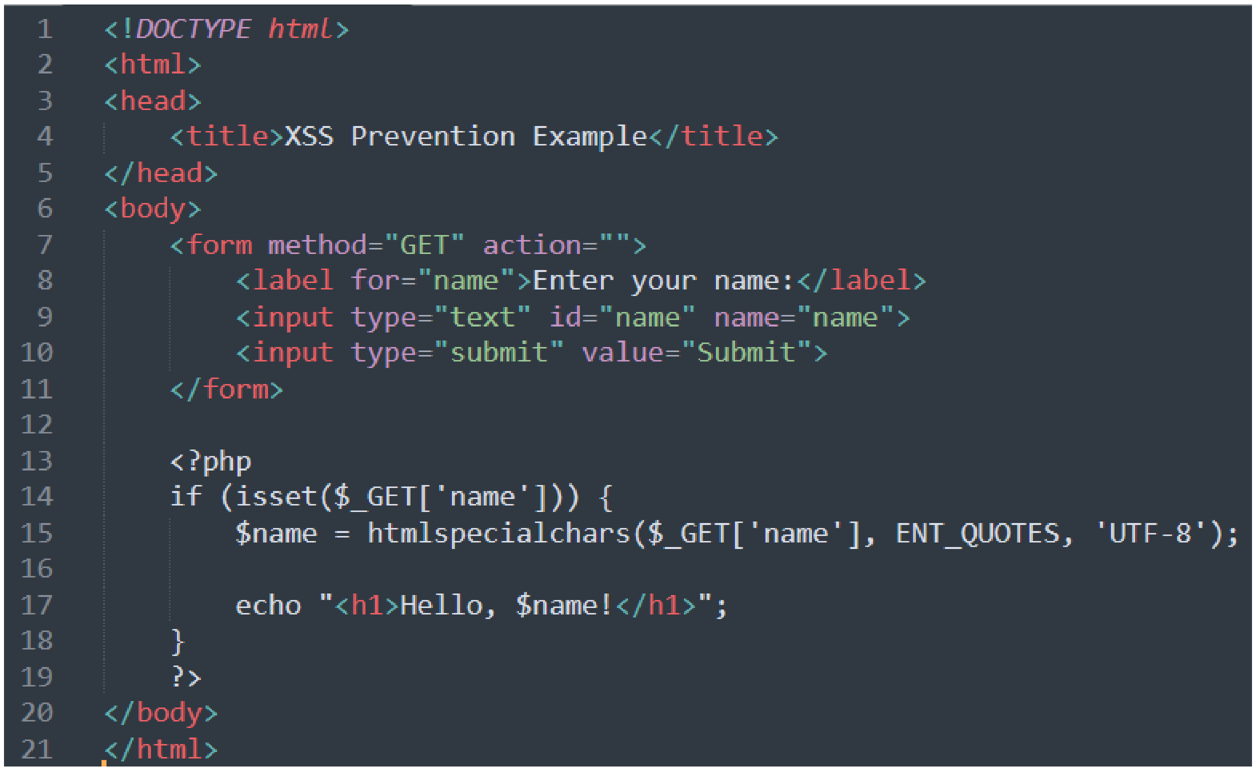
Vulnerable line:



```
        echo "<h1>Hello, $name!</h1>";
```

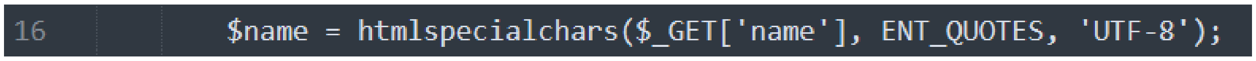
The variable **`$name`** is directly embedded into the HTML output. If an attacker provides a specially crafted input, it can include HTML or JavaScript code that will be executed by the browser when the page is rendered.

Resolved Code:



```
<!DOCTYPE html>
<html>
<head>
    <title>XSS Prevention Example</title>
</head>
<body>
    <form method="GET" action="">
        <label for="name">Enter your name:</label>
        <input type="text" id="name" name="name">
        <input type="submit" value="Submit">
    </form>

    <?php
    if (isset($_GET['name'])) {
        $name = htmlspecialchars($_GET['name'], ENT_QUOTES, 'UTF-8');

        echo "<h1>Hello, $name!</h1>";
    }
    ?>
</body>
</html>
```

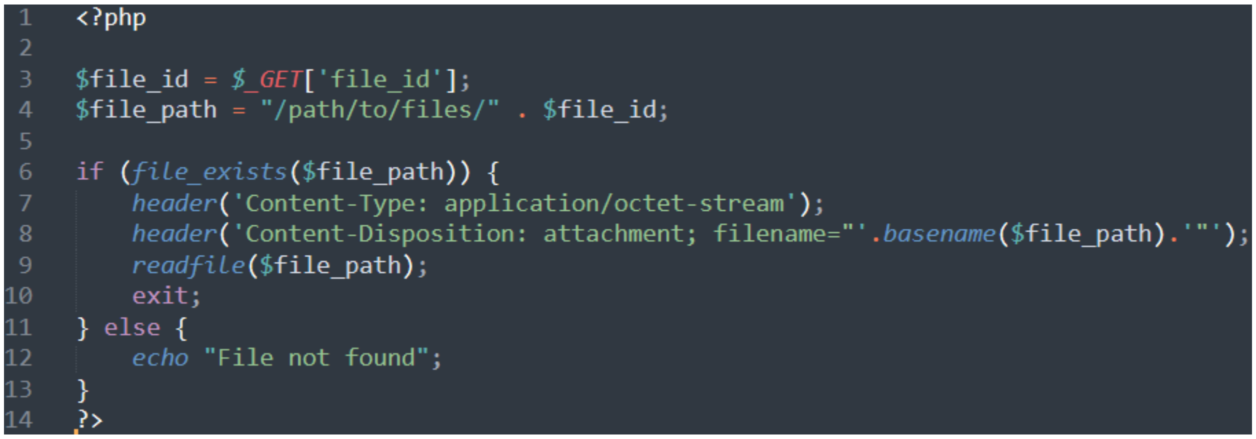
The **`htmlspecialchars`** function is used to convert special characters to HTML entities. This prevents the browser from interpreting any user input as HTML or JavaScript code. By using this revised code safely escapes user input, preventing the browser from interpreting it as executable code and thereby mitigating XSS attacks.



```
        $name = htmlspecialchars($_GET['name'], ENT_QUOTES, 'UTF-8');
```

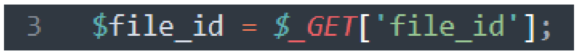
- The first parameter is the string to be escaped.
- The second parameter, **`ENT_QUOTES`**, ensures both double and single quotes are converted to HTML entities.
- The third parameter specifies the character encoding, ensuring the function handles the input correctly.

❖ Code Vulnerable to IDOR:



```php
<?php

$file_id = $_GET['file_id'];
$file_path = "/path/to/files/" . $file_id;

if (file_exists($file_path)) {
    header('Content-Type: application/octet-stream');
    header('Content-Disposition: attachment; filename="'.basename($file_path).'"');
    readfile($file_path);
    exit;
} else {
    echo "File not found";
}
?>
```

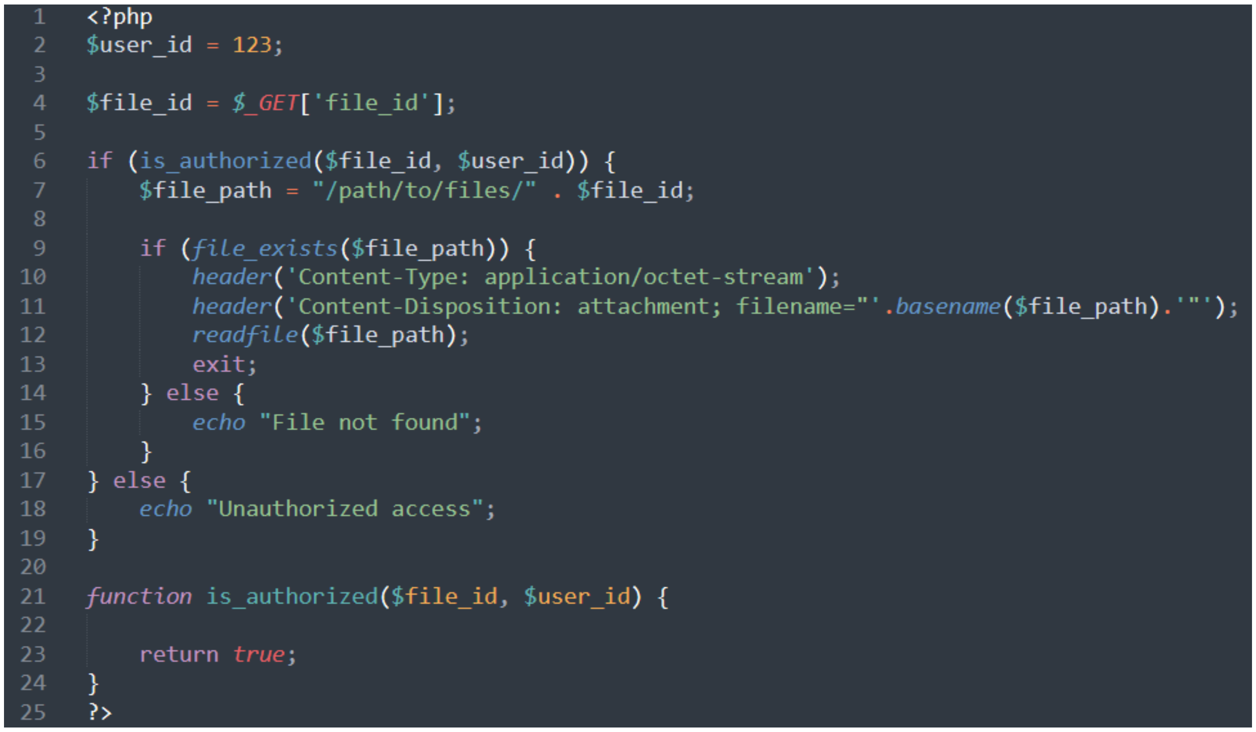
Vulnerable line:



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

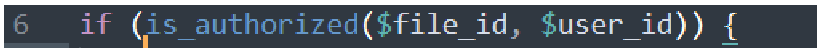
In this vulnerable line, the code directly retrieves the **`file_id`** parameter from the user's input without any validation or authorization checks. This allows an attacker to manipulate the **`file_id`** parameter in the URL to access files they are not authorized to access.

Resolved Code:



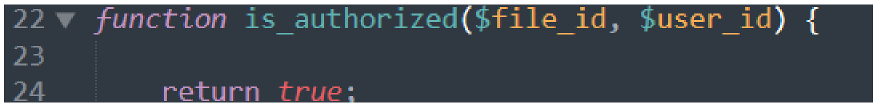
```php
<?php
$user_id = 123;

$file_id = $_GET['file_id'];

if (is_authorized($file_id, $user_id)) {
    $file_path = "/path/to/files/" . $file_id;

    if (file_exists($file_path)) {
        header('Content-Type: application/octet-stream');
        header('Content-Disposition: attachment; filename="'.basename($file_path).'"');
        readfile($file_path);
        exit;
    } else {
        echo "File not found";
    }
} else {
    echo "Unauthorized access";
}

function is_authorized($file_id, $user_id) {

    return true;
}
?>
```

Authorization Check: Before accessing the file, the code performs an authorization check (**`is_authorized`**) to ensure that the user is authorized to access the requested file. This could involve checking if the file belongs to the user or if the user has the necessary permissions.



```
if (is_authorized($file_id, $user_id)) {
```

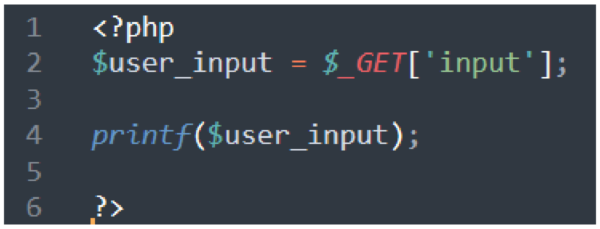
Authorization Logic: You need to implement the **`is_authorized`** function with appropriate authorization logic tailored to your application's requirements. This function should return `true` if the user is authorized to access the file and `false` otherwise.



```
function is_authorized($file_id, $user_id) {

    return true;
```

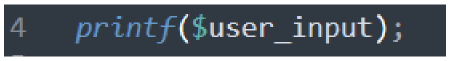
By implementing proper authorization checks, you prevent unauthorized access to sensitive files, thereby securing your application against IDOR attacks.

❖ Code Vulnerable to Format String Attacks:



```
<?php
$user_input = $_GET['input'];

printf($user_input);

?>
```

Format string vulnerabilities typically occur when user-controlled input is passed as the format string argument to functions like **`printf`**, **`sprintf`**, or **`fprintf`** without proper sanitization.

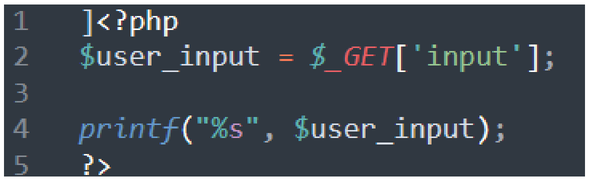


```
printf($user_input);
```

In this line, the **`printf`** function is called with the user-controlled input **`$user_input`** directly as the format string. If an attacker supplies a format string containing format specifiers like `%s`, `%x`, etc., it can lead to various issues, including:

- **Leaking Stack Data**: An attacker can use format specifiers to read and leak sensitive data from the stack.
- **Arbitrary Memory Read/Write**: With certain format specifiers, an attacker can read or write arbitrary memory locations, leading to information disclosure or code execution vulnerabilities.

Resolved Code:

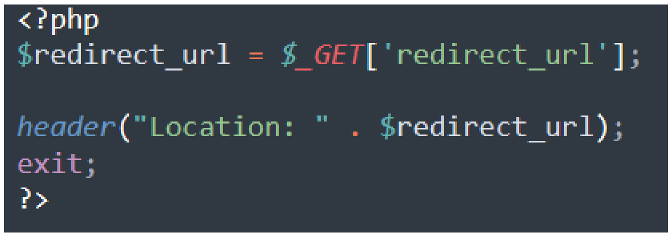


```php
]<?php
$user_input = $_GET['input'];

printf("%s", $user_input);
?>
```

Avoid passing user input directly as the format string. Instead, use placeholders and pass user input as arguments to the formatting functions. Here's the secured code snippet:

- The **`printf`** function is called with a fixed format string `"%s"`.
- The user input **`$user_input`** is passed as a separate argument to **`printf.`**
- Using **`"%s"`** as the format string ensures that **`$user_input`** is treated as a string and does not contain any format specifiers that could be exploited by an attacker.

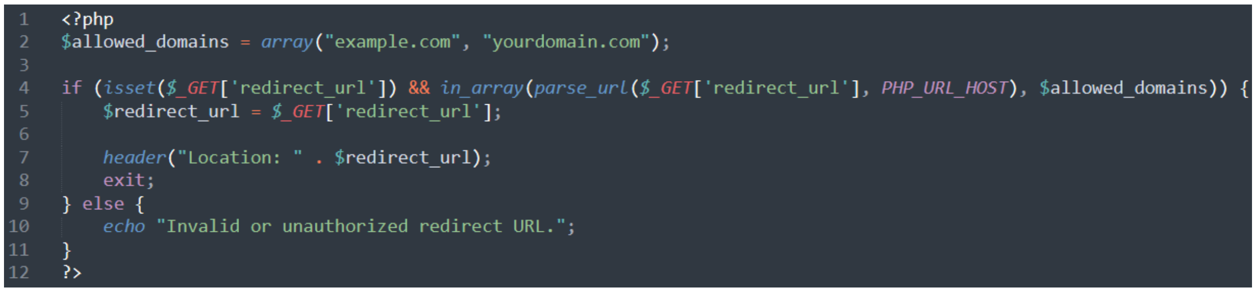
❖ Code Vulnerable to Open Redirect Attacks:



```php
<?php
$redirect_url = $_GET['redirect_url'];

header("Location: " . $redirect_url);
exit;
?>
```

In this vulnerable code, the script blindly trusts the **`redirect_url`** parameter passed via the GET request without proper validation or sanitization. An attacker can exploit this by providing a malicious URL that redirects users to a phishing site or any other malicious destination.

Resolved Code:



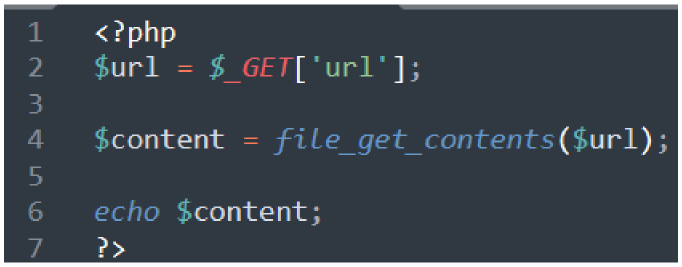
```php
<?php
$allowed_domains = array("example.com", "yourdomain.com");

if (isset($_GET['redirect_url']) && in_array(parse_url($_GET['redirect_url'], PHP_URL_HOST), $allowed_domains)) {
    $redirect_url = $_GET['redirect_url'];

    header("Location: " . $redirect_url);
    exit;
} else {
    echo "Invalid or unauthorized redirect URL.";
}
?>
```

Allowed Domains List: Maintain a list of trusted domains that the script is allowed to redirect to. This list should include only domains that you control or explicitly trust.

Validation and Sanitization: Use **`parse_url`** to extract the host/domain from the provided URL. Then, check if the extracted domain is present in the list of allowed domains.

Secure Redirect: Only redirect if the provided URL passes validation and is included in the list of allowed domains. Otherwise, display an error message or take appropriate action.

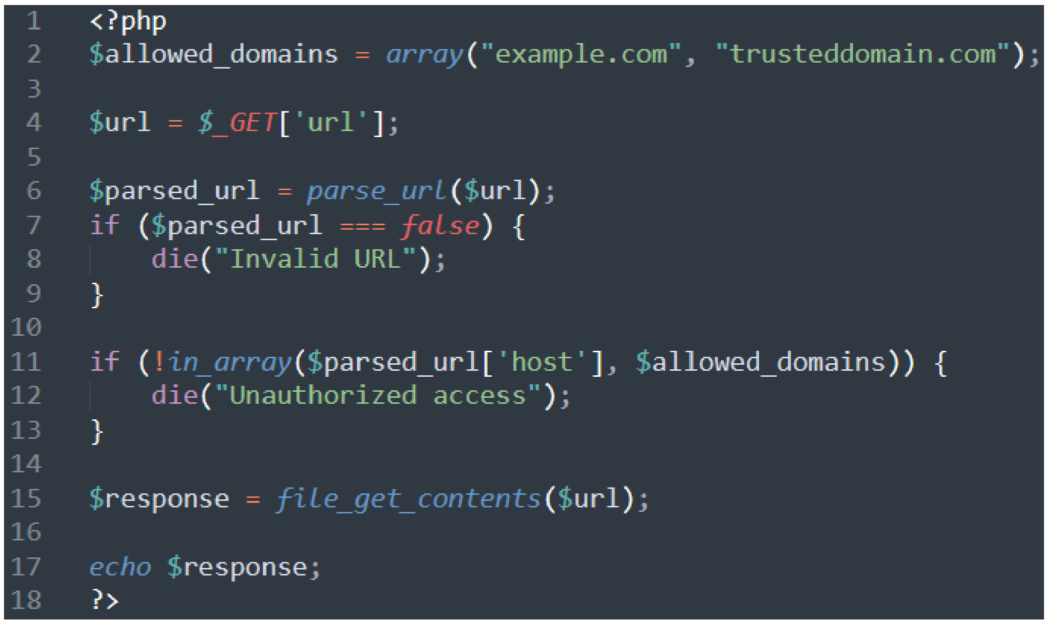
❖ Code Vulnerable to SSRF:



```php
<?php
$url = $_GET['url'];

$content = file_get_contents($url);

echo $content;
?>
```

In this vulnerable code snippet, the script accepts a URL from the user via the **`$_GET`** superglobal and fetches the content of that URL using **`file_get_contents()`**. An attacker can exploit this by providing a malicious URL that leads to internal services or sensitive resources, potentially leading to unauthorized access or information disclosure.

Resolved Code:



```php
<?php
$allowed_domains = array("example.com", "trusteddomain.com");

$url = $_GET['url'];

$parsed_url = parse_url($url);
if ($parsed_url === false) {
    die("Invalid URL");
}

if (!in_array($parsed_url['host'], $allowed_domains)) {
    die("Unauthorized access");
}

$response = file_get_contents($url);

echo $response;
?>
```

By implementing proper input validation and using a whitelist approach, you can effectively mitigate SSRF vulnerabilities in PHP applications. This approach restricts the URLs that the script can access, preventing unauthorized requests to internal services or sensitive resources.

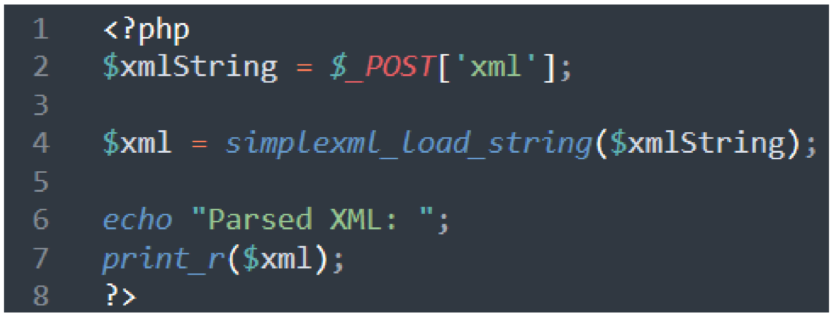
Whitelist Approach: Maintain a whitelist of allowed domains or IP addresses that the script is permitted to access. This restricts the URLs that the script can fetch content from.

Parse URL: Use **`parse_url()`** to extract the hostname from the provided URL. This ensures that the URL is properly formatted and helps prevent SSRF attacks with malformed or non-URL input.

Check Allowed Domains: Verify that the hostname extracted from the URL is included in the whitelist of allowed domains. If the hostname is not in the whitelist, terminate the script execution to prevent unauthorized access.

Secure Content Retrieval: If the URL passes validation, securely fetch the content using **`file_get_contents().`** By enforcing the whitelist, the script ensures that only authorized URLs are accessed, mitigating the risk of SSRF attacks.

❖ <u>Code Vulnerable to XXE Attacks:</u>

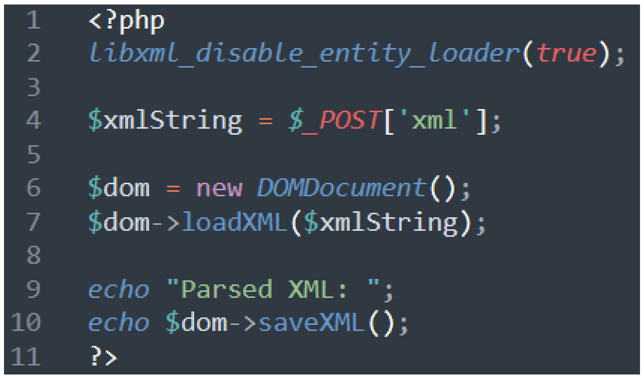


```
<?php
$xmlString = $_POST['xml'];

$xml = simplexml_load_string($xmlString);

echo "Parsed XML: ";
print_r($xml);
?>
```

In this vulnerable code snippet, the script accepts an XML string from the user via the **`$_POST`** superglobal and parses it using **`simplexml_load_string().`** An attacker can exploit this by providing a malicious XML payload containing external entity references that could access sensitive files, perform server-side request forgery (SSRF), or consume excessive server resources.

XXE attacks can have severe consequences, including data breaches, service disruptions, and compromise of server integrity. To prevent XXE attacks, web applications should implement proper input validation, disable external entity loading in XML parsers, and use safe XML processing libraries that mitigate XXE vulnerabilities.

Resolved Code:



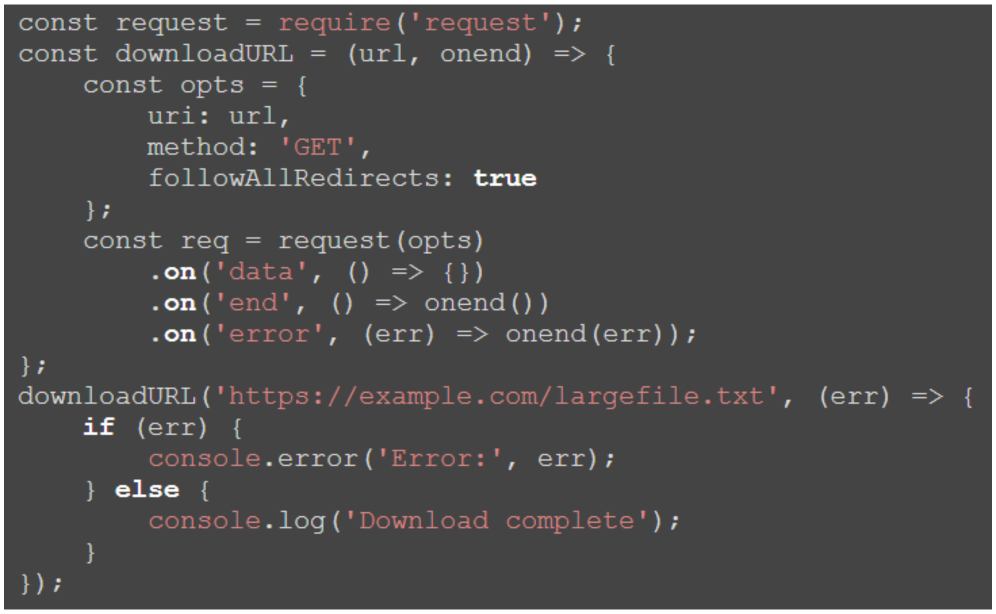
```php
<?php
libxml_disable_entity_loader(true);

$xmlString = $_POST['xml'];

$dom = new DOMDocument();
$dom->loadXML($xmlString);

echo "Parsed XML: ";
echo $dom->saveXML();
?>
```

Disable External Entity Loading: Use `libxml_disable_entity_loader(true)` to disable the loading of external entities in the XML parser. This prevents XXE attacks by preventing the parser from resolving external entities.

Secure XML Parsing: Instead of using `simplexml_load_string()`, which is vulnerable to XXE attacks, use `DOMDocument` to parse the XML string securely. `DOMDocument` does not resolve external entities by default, providing a safer alternative.

Output Parsed XML: Use `saveXML()` to output the parsed XML. This method securely returns the parsed XML without risking XXE vulnerabilities.

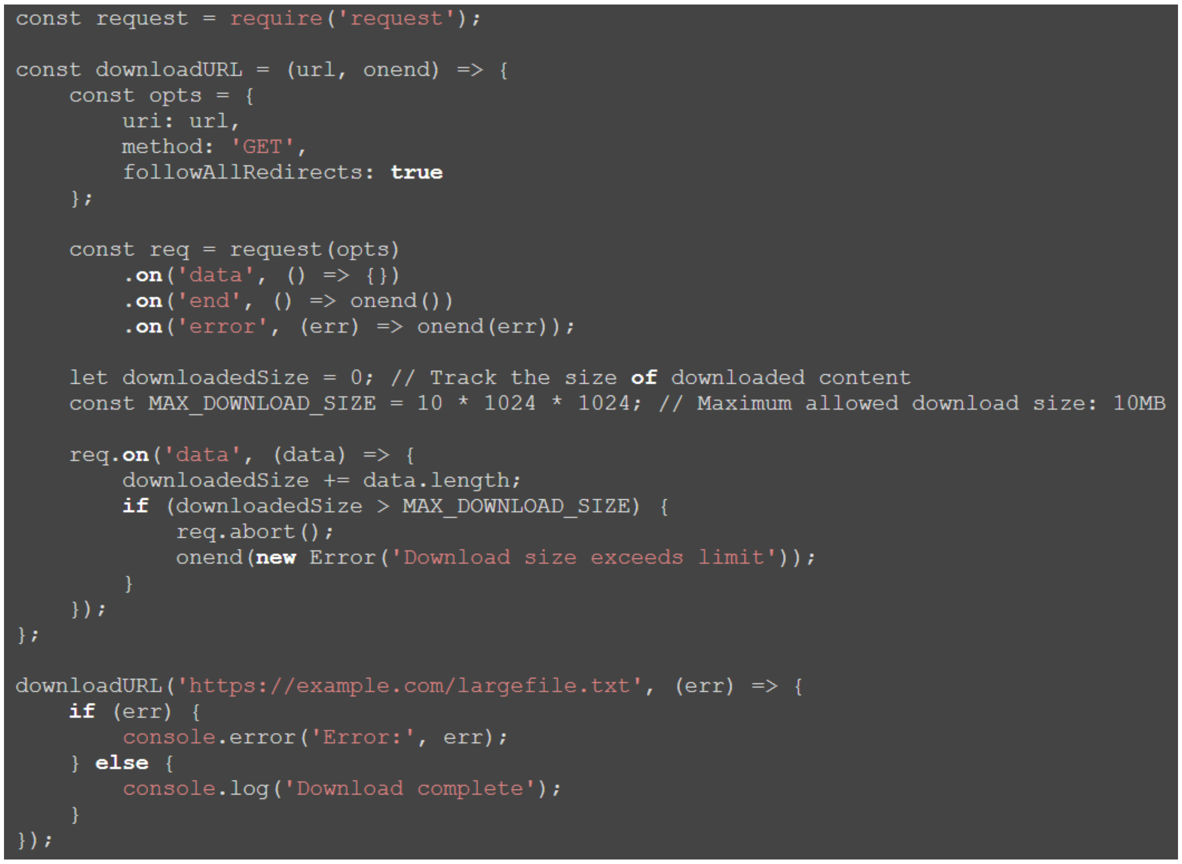
❖ Code Vulnerable to DOS attack:



```javascript
const request = require('request');
const downloadURL = (url, onend) => {
    const opts = {
        uri: url,
        method: 'GET',
        followAllRedirects: true
    };
    const req = request(opts)
        .on('data', () => {})
        .on('end', () => onend())
        .on('error', (err) => onend(err));
};
downloadURL('https://example.com/largefile.txt', (err) => {
    if (err) {
        console.error('Error:', err);
    } else {
        console.log('Download complete');
    }
});
```

In this code, the **`downloadURL`** function makes an HTTP request to a specified URL without any limitation on the download size. This can potentially lead to a DoS vulnerability if an attacker sends requests to download extremely large files, causing the server to consume excessive resources and become unresponsive.

Resolved Code:



```javascript
const request = require('request');

const downloadURL = (url, onend) => {
    const opts = {
        uri: url,
        method: 'GET',
        followAllRedirects: true
    };

    const req = request(opts)
        .on('data', () => {})
        .on('end', () => onend())
        .on('error', (err) => onend(err));

    let downloadedSize = 0; // Track the size of downloaded content
    const MAX_DOWNLOAD_SIZE = 10 * 1024 * 1024; // Maximum allowed download size: 10MB

    req.on('data', (data) => {
        downloadedSize += data.length;
        if (downloadedSize > MAX_DOWNLOAD_SIZE) {
            req.abort();
            onend(new Error('Download size exceeds limit'));
        }
    });
};

downloadURL('https://example.com/largefile.txt', (err) => {
    if (err) {
        console.error('Error:', err);
    } else {
        console.log('Download complete');
    }
});
```

To address the denial-of-service (DoS) vulnerability, we need to implement a mechanism to limit the size of the downloaded content.

- We've introduced a **`downloadedSize`** variable to track the size of the downloaded content.
- We've defined a constant **`MAX_DOWNLOAD_SIZE`** to specify the maximum allowed download size, which is set to 10MB.
- Within the **`req.on('data', ...)`** callback, we increment **`downloadedSize`** by the length of each received chunk of data. If the total downloaded size exceeds **`MAX_DOWNLOAD_SIZE`**, we abort the request and invoke the **`onend`** callback with an error indicating that the download size exceeds the limit.