PDF Report for CS-437/SEC537 Term project

Members:

Kerem Acildi (31132)

Caglar Uysal (29112)

Drive folder with videos:
https://drive.google.com/drive/folders/1MKisiTVCzkmjDdQY9b18LKwL41NOlOgl?usp=sharing

Name of The Vulnerability: Blind OS Command Injection

Category of The Vulnerability According to OWASP Top 10 2021: Injection (A03:2021)

Technical Explanation of The Vulnerability: Blind OS Command Injection occurs when an application allows user-controlled input to be passed directly to an operating system command without proper sanitization or validation. The "blind" aspect means that while the attacker cannot see the output of the command, they can infer its success through behavioral changes in the application (e.g., response delays or side effects).

Exploit of The Vulnerability: An attacker could inject additional commands via an input field. For example:

localhost && sleep 5

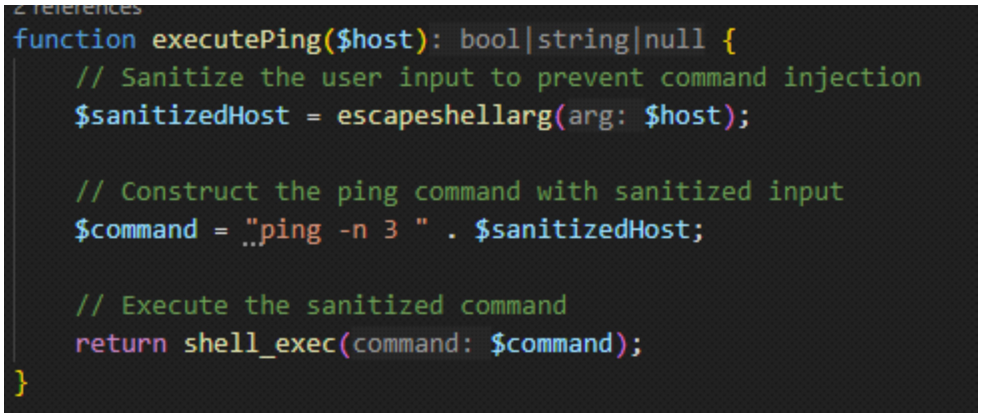
This would cause a 5-second delay, indicating that the command was successfully executed.

Unpatched code snippet:

```
2 references
function executePing($host): bool|string|null {
    // Directly use the user input in the shell command (intentionally vulnerable)
    $command = "ping -n 3 " . $host;
    return shell_exec(command: $command); // Executes user input directly which makes this page vulnerable

}
```

Patched Code Snippet:



```
2 references
function executePing($host): bool|string|null {
    // Sanitize the user input to prevent command injection
    $sanitizedHost = escapeshellarg(arg: $host);

    // Construct the ping command with sanitized input
    $command = "ping -n 3 " . $sanitizedHost;

    // Execute the sanitized command
    return shell_exec(command: $command);
}
```

Name of The Vulnerability: Reflected XSS

Category of The Vulnerability According to OWASP Top 10 2021: Cross-Site Scripting (A07:2021)

Technical Explanation of The Vulnerability:

Reflected XSS occurs when an application includes unvalidated and unsanitized user input directly into its HTML output. This allows attackers to inject malicious scripts, which execute in the context of a victim's browser.

Exploit of The Vulnerability:

An attacker could inject a script via a URL parameter, such as:

ert('XSS')</script>

The injected script would execute in the user's browser, potentially stealing sensitive information like session cookies.

Unpatched code snippet:

```php
// Process search query
$searchResults = [];
if (isset($_GET['query']) && $rss) {
    $query = strtolower(string: trim(string: $_GET['query'])); // user input is taken directly without sanitizing it
    foreach ($newsItems as $item) {                            // which makes this page vulnerable
        $title = strtolower(string: (string) $item->title);
        $description = strtolower(string: (string) $item->description);

        if (strpos(haystack: $title, needle: $query) !== false || strpos(haystack: $description, needle: $query) !== false) {
            $searchResults[] = $item;
        }
    }
}
?>
```

Patched Code Snippet:

```php
<div class="container">
    <div class="search-box">
        <form method="GET" action="">
            <input type="text" name="query" placeholder="Search news..." value="<?= htmlspecialchars(string: $_GET['query'] ?? '') ?>"> <!--using html special chars for patching-
            <button type="submit">Search</button>
        </form>
    </div>
    <?php if (isset($errorMessage)): ?>
        <p style="color: red; text-align: center;"> <?= htmlspecialchars(string: $errorMessage); ?></p> <!--using html special chars for patching-->
    <?php elseif (!empty($searchResults)): ?>
        <h2>Search Results:</h2>
        <?php foreach ($searchResults as $item): ?>
            <div class="news-item">
                <h3 class="news-title">
                    <a href="<?= htmlspecialchars(string: $item->link); ?>" target="_blank"> <!--using html special chars for patching-->
                        <?= htmlspecialchars(string: $item->title); ?> <!--using html special chars for patching-->
                    </a>
                </h3>
                <p class="news-description"> <?= htmlspecialchars(string: $item->description); ?></p> <!--using html special chars for patching-->
                <small>Published on: <?= date(format: "Y-m-d H:i", timestamp: strtotime(datetime: $item->pubDate)); ?></small>
            </div>
        <?php endforeach; ?>
    <?php elseif (isset($_GET['query'])): ?>
        <p>No results found for "<?= htmlspecialchars(string: $_GET['query']); ?>".</p> <!--using html special chars for patching-->
    <?php else: ?>
        <p>Enter a search term to find news articles.</p>
    <?php endif; ?>
</div>
<div class="footer">
    &copy; <?= date(format: 'Y'); ?> Search News | Powered by PHP
</div>
body>
html>
```

Name of The Vulnerability: CWE -434 Unrestricted File Upload

Category of The Vulnerability According to OWASP Top 10 2021: Insecure Design (A04:2021)

Technical Explanation of The Vulnerability: Unrestricted file upload occurs when an application fails to validate or restrict uploaded files, allowing attackers to upload malicious files such as executable scripts (e.g., `.php` files). These files could be executed on the server, leading to a full compromise.

Exploit of The Vulnerability: An attacker could upload a malicious PHP script disguised with a double extension, such as:

malicious.php.txt

If executed, this file could allow remote code execution.

Unpatched Code snippet:

```php
$allowedExtensions = ['jpg', 'png', 'txt']; // Does not properly block files like .php.txt or .exe.jpg so vulnerable to obfuscated extensions if file contents are not checked

// Check if the form is submitted
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if (!empty($_FILES['file'])) {
        $file = $_FILES['file'];

        // Get the original file name
        $fileName = $file['name'];
        $fileExtension = pathinfo(path: $fileName, flags: PATHINFO_EXTENSION);

        // Flawed validation: checks extension but allows disguised dangerous files
        if (in_array(needle: $fileExtension, haystack: $allowedExtensions)) {
            // Create the target file path
            $targetFilePath = $uploadDir . $fileName;

            // Move the uploaded file to the target directory
            if (move_uploaded_file(from: $file['tmp_name'], to: $targetFilePath)) {
                $message = "File uploaded successfully: " . htmlspecialchars(string: $fileName);
            } else {
                $message = "Failed to upload file.";
            }
        } else {
            $message = "File type not allowed.";
        }
    } else {
        $message = "No file selected for upload.";
    }
}

// Create the upload directory if it does not exist
if (!is_dir(filename: $uploadDir)) {
    mkdir(directory: $uploadDir, permissions: 0777, recursive: true);
}
```

Patched Code Snippet:

```php
// Allowed MIME types and file extensions
$allowedMimeTypes = ["image/jpeg", "image/png", "text/plain"];     // using whitelist approach not blacklist approach
$allowedExtensions = ["jpg", "jpeg", "png", "txt"];

// Create the upload directory if it does not exist
if (!is_dir(filename: $uploadDir)) {
    mkdir(directory: $uploadDir, permissions: 0777, recursive: true);
}

// Function to validate the file upload
1 reference
function validateFileUpload($file, $allowedMimeTypes, $allowedExtensions): bool {     //validating the files content type and extension
    // Use finfo to determine the MIME type of the file
    $finfo = finfo_open(flags: FILEINFO_MIME_TYPE);
    $fileType = finfo_file(finfo: $finfo, filename: $file['tmp_name']);
    finfo_close(finfo: $finfo);

    $fileExtension = strtolower(string: pathinfo(path: $file['name'], flags: PATHINFO_EXTENSION));

    // Check MIME type and file extension
    if (!in_array(needle: $fileType, haystack: $allowedMimeTypes) || !in_array(needle: $fileExtension, haystack: $allowedExtensions)) {
        return false;
    }

    return true;
}
```

Name of The Vulnerability: SSRF Blacklist based input filter

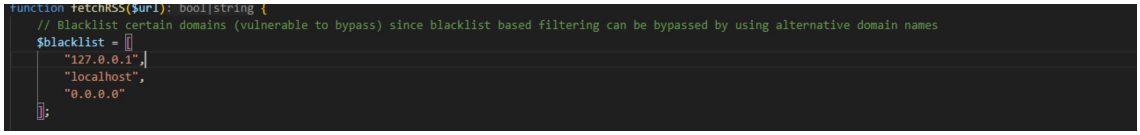
Category of The Vulnerability According to OWASP Top 10 2021: Security Misconfiguration (A05:2021)

Technical Explanation of The Vulnerability: Improper handling of external RSS feeds may lead to vulnerabilities such as injection attacks, loading malicious scripts, or exposing sensitive information. Fetching unvalidated feeds can allow attackers to introduce crafted XML payloads.

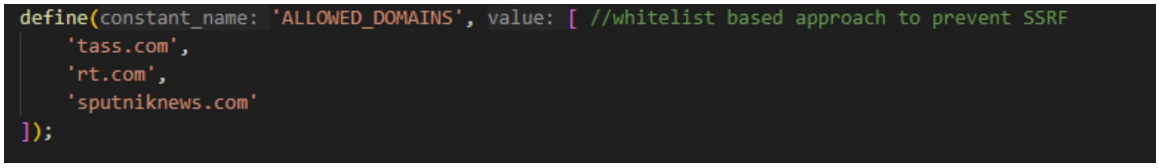
Exploit of The Vulnerability:

An attacker could provide a malicious RSS URL that executes unintended actions or exposes server data.

Unpatched Code Snippet:



Patched Code Snippet:



Name of The Vulnerability: CWE-35 Path Traversal

Category of The Vulnerability According to OWASP Top 10 2021: Security Misconfiguration (A05:2021)

Technical Explanation of The Vulnerability:

Path Traversal occurs when an application improperly validates user input used in file paths, allowing attackers to access files outside the intended directory. By using special characters such as `../`, attackers can traverse directories and access restricted files.

Exploit of The Vulnerability:

An attacker could exploit the vulnerability by submitting a crafted file path, such as:

../../../../etc/passwd

This would allow access to the server's password file if proper validation is not in place.

Unpatched Code Snippet:

```php
// Function to read a file based on user input (intentionally vulnerable)
2 references
function readFileContents($filePath): bool|string {
    // Dynamically set the base directory relative to the script location
    $baseDir = __DIR__ . DIRECTORY_SEPARATOR; // Base directory is the script's directory
    $fullPath = $baseDir . $filePath; // Concatenate base directory and user input

    // Debugging: Print the constructed path
    echo "Attempting to access: " . $fullPath . "<br>";

    return file_get_contents(filename: $fullPath); // Vulnerable to path traversal
}
```

Patched Code Snippet:

```php
// Function to safely read a file based on user input
2 references
function readFileContents($filePath): bool|string {
    // Dynamically set the base directory relative to the script location
    $baseDir = __DIR__ . DIRECTORY_SEPARATOR . 'files';

    // Ensure the base directory exists
    if (!is_dir(filename: $baseDir)) {
        die("Base directory does not exist.");
    }

    // Get the real path of the base directory
    $realBaseDir = realpath(path: $baseDir);
    $realFilePath = realpath(path: $realBaseDir . DIRECTORY_SEPARATOR . $filePath);

    // Validate the file path to ensure it stays within the base directory
    if ($realFilePath === false || strpos(haystack: $realFilePath, needle: $realBaseDir) !== 0) {
        die("Access denied. Invalid file path.");
    }

    // Check if the file exists
    if (!file_exists(filename: $realFilePath)) {
        die("File does not exist.");
    }

    // Read and return the file contents
    return file_get_contents(filename: $realFilePath);
}

$fileContent = "";
if (isset($_GET['file'])) {
    $file = basename(path: $_GET['file']); // Restrict input to file name only
    $fileContent = readFileContents($file);
}
?>
```

Name of The Vulnerability: SQL Injection(1) Blind SQL injection

Category of The Vulnerability According to OWASP Top 10 2021: A03:2021 – Injection

Technical Explanation of The Vulnerability: User-supplied sql data is not validated, filtered, or sanitized by the application. Blind Sql injection attacks does not rely on seeing the data. In our case we used blind sql injection to bypass admin credentials and logn as an admin.

Exploit of The Vulnerability:

An attacker could provide a <admin username>' # as username and any password.

Unpatched Code Snippet:

```
// Directly embedding user input into the query allows attackers to inject malicious SQL code.
// Lack of parameterized queries means special characters in input can alter the SQL logic.
    $query = "SELECT * FROM users WHERE username = '$user' AND password = '$pass'";
    $stmt = $conn->query($query);

    if ($stmt->rowCount() > 0) {
        echo "<h2>Login successful! Welcome, " . htmlspecialchars(string: $user) . "</h2>";
    } else {
        echo "<h2>Invalid username or password.</h2>";
    }
```

Patched Code Snippet:

```
    // Using prepared statements ensures the query structure is precompiled, preventing SQL injection.
    // Parameters are bound securely with bindParam(), treating user input as data, not executable code.
    $stmt = $conn->prepare(query: "SELECT * FROM users WHERE username = :username AND password = :password");
    $stmt->bindParam(param: ':username', var: &$user);
    $stmt->bindParam(param: ':password', var: &$pass);
    $stmt->execute();

    if ($stmt->rowCount() > 0) {
        echo "<h2>Login successful! Welcome, " . htmlspecialchars(string: $user) . "</h2>";
    } else {
        echo "<h2>Invalid username or password.</h2>";
    }
}
```

Name of The Vulnerability: SQL Injection(2) Union SQL injection

Category of The Vulnerability According to OWASP Top 10 2021: A03:2021 – Injection

Technical Explanation of The Vulnerability: User-supplied sql data is not validated, filtered, or sanitized by the application. use the UNION keyword to retrieve data from other tables within the database. This is commonly known as a SQL injection UNION attack. The UNION keyword enables you to execute one or more additional SELECT queries and append the results to the

original query. In our case we used Union attack to gather information from "dummy_data" sql table which should not be accessible from anywhere in the application.

Exploit of The Vulnerability:

An attacker could exploit the vulnerability by submitting a crafted sql query, such as:

' UNION SELECT id, name, email FROM dummy_data#

This would allow access to the private data tables that server has.

Unpatched Code Snippet:

```
// Directly embedding user input in the query allows attackers to inject malicious SQL code.
// Lack of prepared statements or parameterized queries leaves the database exposed to manipulation.
$query = "SELECT id, username, role FROM users WHERE role = '$role'";
$stmt = $conn->query($query);

if ($stmt->rowCount() > 0) {
    while ($row = $stmt->fetch(mode: PDO::FETCH_ASSOC)) {
        echo "<tr>";
        echo "<td>" . htmlspecialchars(string: $row['id']) . "</td>";
        echo "<td>" . htmlspecialchars(string: $row['username']) . "</td>";
        echo "<td>" . htmlspecialchars(string: $row['role']) . "</td>";
        echo "</tr>";
    }
```

Patched Code Snippet:

```
if (isset($_GET['role']) && !empty($_GET['role'])) {
    $role = $_GET['role'];
    // The use of a prepared statement ensures the query structure is fixed, preventing SQL injection.
    // Binding the :role parameter securely sanitizes user input, treating it as data, not executable code.
    $query = "SELECT id, username, role FROM users WHERE role = :role";
    $stmt = $conn->prepare(query: $query);
    $stmt->bindParam(param: ':role', var: &$role, type: PDO::PARAM_STR);
    $stmt->execute();

    if ($stmt->rowCount() > 0) {
        while ($row = $stmt->fetch(mode: PDO::FETCH_ASSOC)) {
            echo "<tr>";
            echo "<td>" . htmlspecialchars(string: $row['id']) . "</td>";
            echo "<td>" . htmlspecialchars(string: $row['username']) . "</td>";
            echo "<td>" . htmlspecialchars(string: $row['role']) . "</td>";
            echo "</tr>";
        }
```

Table for who did what part:

- txt report → **Caglar Uysal**
- pdf report → **Caglar Uysal**
- video recording → **Caglar Uysal**

- cwe-434 implementation and patching → **Caglar Uysal**
- cwe-35 implementation and patching → **Caglar Uysal**
- SSRF with blacklist based input filter implementation and patching → **Kerem Acildi**
- Blind OS Command Injection implementation and patching → **Kerem Acildi**
- Reflected XSS implementation and patching → **Caglar Uysal**
- SQL injection (1) implementation and patching → **Caglar Uysal**
- SQL injection (2) implementation and patching → **Caglar Uysal**
- Connecting php with database → **Caglar Uysal**
- XML parsing → **Kerem Acildi**