

LAB_2

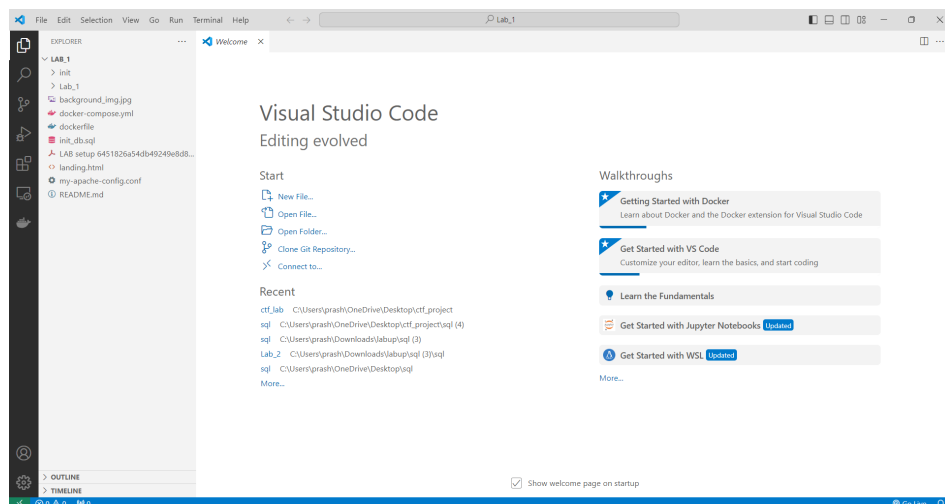
The goal of this Lab is to Identify the Vulnerability in the code **and patch Lab 1**. There are many ways to do code analysis. Here we are focusing on static code analysis.

When you unzip the Lab1. It Look like this this contain docker image and the code of the Lab

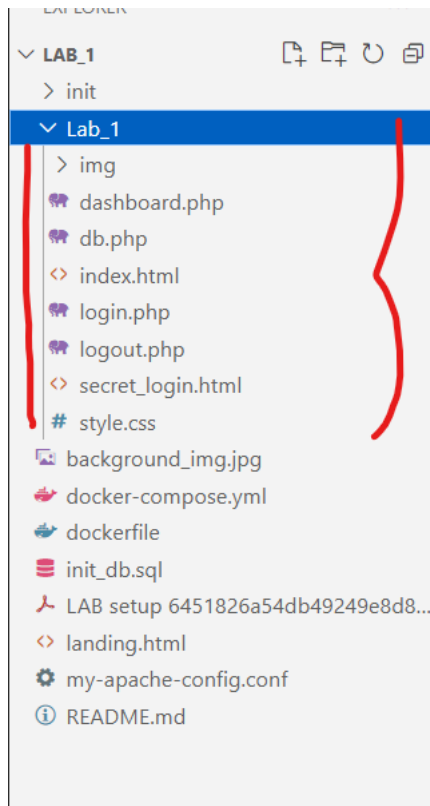
Name	Date modified	Type	Size
init	23/03/2024 11:54	File folder	
Lab_1	23/03/2024 11:54	File folder	
background_img.jpg	23/03/2024 11:45	JPG File	51 KB
docker-compose.yml	23/03/2024 12:59	Yaml Source File	1 KB
dockerfile	23/03/2024 12:56	File	2 KB
init_db.sql	23/03/2024 11:45	SQL Source File	1 KB
LAB setup 6451826a54db49249e8d855fe...	23/03/2024 11:45	Microsoft Edge PD...	11,246 KB
landing.html	23/03/2024 11:52	Chrome HTML Do...	2 KB
my-apache-config.conf	23/03/2024 11:45	CONF File	1 KB
README.md	23/03/2024 21:14	Markdown Source ...	1 KB

Your Goal is to do Code Analysis find the vulnerability and patch it.

You can use any Ide and Code editor to look at the code. I am using VScode. Open the directory in VScode.



The Lab one directory contains all the code. You have to do code analysis here.



How to do Code Analysis or Static Code Analysis (SAST)

Static Application Security Testing (SAST) involves examining the source code without executing it. It detects security flaws, adherence to coding standards, and other potential issues by analyzing code at rest.

1. Automated Static Code Analysis

1. **Select the Right Tools:** Choose static analysis tools that are well-suited for your project's language and framework. Popular options include SonarQube, ESLint (for JavaScript), and Checkmarx.
2. **Integrate with Your Development Environment:** Most tools can be integrated into IDEs (Integrated Development Environments) and CI/CD pipelines. This integration allows for continuous analysis and immediate feedback.
3. **Configure the Analysis Rules:** Customize the tool's ruleset based on your project's requirements. Disable irrelevant rules to reduce noise and focus on high-impact vulnerabilities.
4. **Run the Analysis:** Execute the tool against your codebase. This can be triggered manually or automatically as part of your development process.

5. **Review and Interpret Results:** Examine the findings for potential false positives. Prioritize issues based on severity, exploitability, and impact.
6. **Fix Identified Issues:** Address the vulnerabilities and code quality issues. For complex issues, further manual review might be necessary.
7. **Iterate:** Regularly update your tool's configuration and ruleset based on the evolving project needs and newly discovered vulnerabilities.

Example:

Consider a Java application where the SonarQube analysis detects a potential SQL injection vulnerability:

```
javaCopy code
String query = "SELECT * FROM users WHERE username = '" + username + "'";
```

SonarQube would flag this line as critical and suggest using prepared statements or ORM frameworks to mitigate the risk.

2. Manual Static Code Analysis:

Manual static code analysis involves a systematic examination of the source code by developers or security analysts to identify vulnerabilities that automated tools might miss.

1. **Code Review Setup:** Establish a code review process where developers submit their code for review before merging. Use tools like GitHub pull requests for tracking.
2. **Review for Common Vulnerabilities:** Focus on identifying common security vulnerabilities outlined by OWASP, such as SQL injection, XSS, and improper error handling.
3. **Check Source and Sink:** Pay special attention to how data flows through the application. Identify untrusted inputs (sources) and critical operations (sinks), such as database queries and file operations. Ensure that data passing from source to sink is properly validated and sanitized.
4. **Validate Conditions and Loops:** Ensure that logical conditions are correctly implemented and that loops have proper termination conditions to prevent issues like infinite loops or logic errors.
5. **Assess Error Handling and Logging:** Check that the application handles errors gracefully without exposing sensitive information and logs errors appropriately for future analysis.
6. **Look for Code Smells:** Identify bad practices that could indicate deeper problems, such as duplicate code, overly complex methods, or improper resource management.
7. **Documentation and Comments:** Ensure the code is well-documented and comments accurately reflect the current logic and purpose of code sections.
8. **Feedback and Resolution:** Provide constructive feedback on identified issues and work collaboratively to address them. Document the review findings for future reference.

Example:

During a manual review, a developer might notice that user input is directly included in a file path construction:

```
javaCopy code
File file = new File("/var/data/" + userInput);
```

This code should be flagged for potential directory traversal. The reviewer would suggest validating and sanitizing `userInput` or using a safer method to handle file paths

Key Points to Always Check

- **Source and Sink Analysis:** Always track where data comes from and where it is used. Ensure all data flowing from sources to sinks is validated and sanitized.
 - **Validation of External Inputs:** Ensure that all external inputs are validated against a strict specification (type, length, format).
 - **Proper Implementation of Conditions:** Review logical conditions for correctness and ensure they accurately reflect the intended logic.
 - **Error Handling:** Check for robust error handling that does not leak sensitive information.
 - **Use of Cryptography:** Ensure that cryptography, if used, follows best practices (e.g., using strong algorithms and proper key management).
 - **Dependency Security:** Regularly update dependencies and check for known vulnerabilities using tools like OWASP Dependency-Check.
 - **Secure Configuration:** Verify that the application is configured securely, with features like unnecessary services disabled and default credentials changed.
-

Solution of the Lab

This is Vulnerable code Snippet

This is the Vulnerable code Snippet from the previous lab (It is present in **Login.php**). You have to do code analysis and Patch it.

```
<?php
session_start();
include 'db.php';

$username = $_POST['username'];
$password = $_POST['password'];

$sql = "SELECT * FROM users WHERE username = '$username' AND password = '$password'";
$result = $conn->query($sql);

if ($result !== FALSE && $result->num_rows > 0) {
    // Successful login
    $_SESSION['loggedin'] = true;
    $_SESSION['username'] = $username;

    header("Location: dashboard.php");
    exit();
} else {
```

```

        // Login failed
        echo "Login failed!";
    }

    $conn->close();
?>

```

Patch Code (Solution)

How to Patch the Vulnerability

To patch this vulnerability, you should use **prepared** statements with parameterized queries. This approach ensures that user input is treated as data, not as part of the SQL code. Here's how you can patch the code:

Edit the the **login.php** of Lab 1 and put the below code. It will path the SQL injection Vulnerability in the lab.

'ss' specifies that both parameters are strings.

```

<?php

session_start();
include 'db.php';

$username = $_POST['username'];
$password = $_POST['password'];

// Using prepared statements to prevent SQL Injection
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
$stmt->bind_param("ss", $username, $password); // 'ss' specifies that both parameters are string
$stmt->execute();
$result = $stmt->get_result();

if ($result !== FALSE && $result->num_rows > 0) {
    // Successful login
    $_SESSION['loggedin'] = true;
    $_SESSION['username'] = $username;

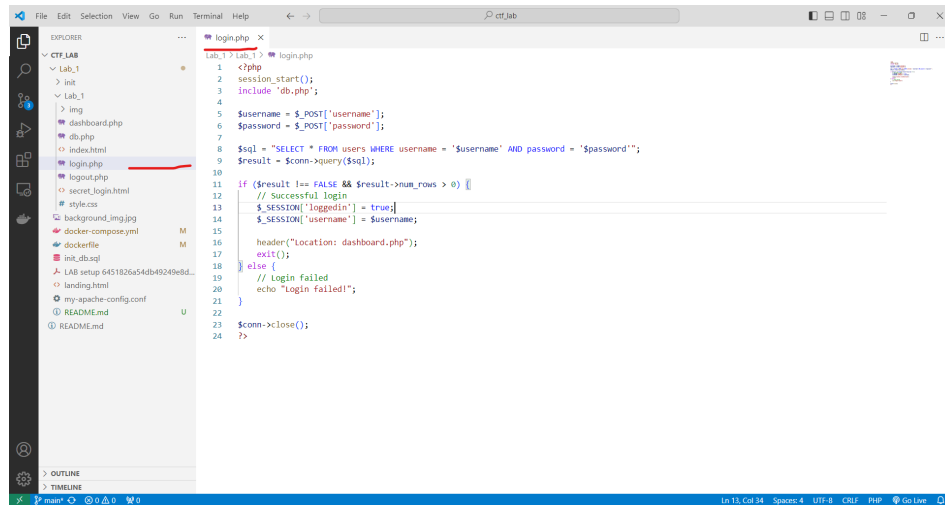
    header("Location: dashboard.php");
    exit();
} else {
    // Login failed
    echo "Login failed!";
}

$conn->close();

```

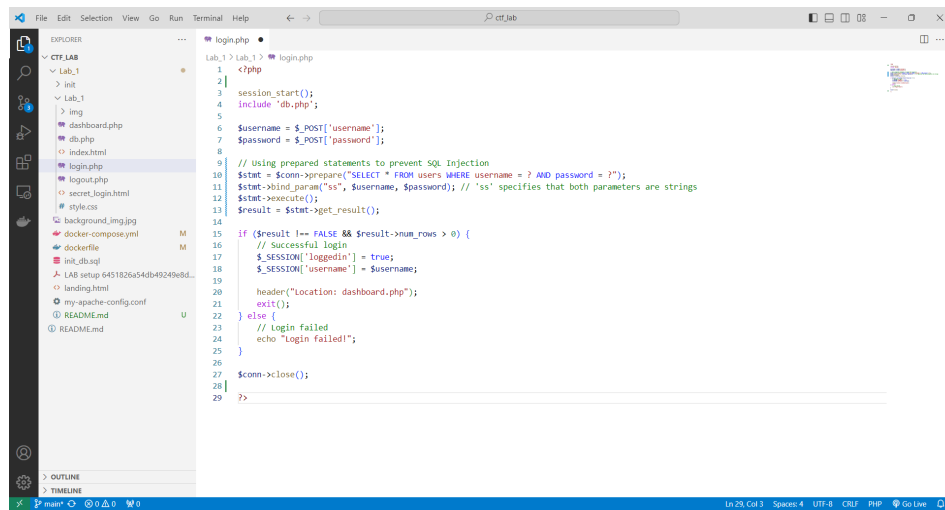
?>

This is the Vulnerable part of the code base.



```
1 <?php
2 session_start();
3 include 'db.php';
4
5 $username = $_POST['username'];
6 $password = $_POST['password'];
7
8 $sql = "SELECT * FROM users WHERE username = '$username' AND password = '$password'";
9 $result = $conn->query($sql);
10
11 if ($result != FALSE && $result->num_rows > 0) {
12     // Successful login
13     $_SESSION['loggedin'] = true;
14     $_SESSION['username'] = $username;
15     header("Location: dashboard.php");
16     exit();
17 }
18 else {
19     // Login failed
20     echo "Login failed!";
21 }
22
23 $conn->close();
24 ?>
```

Put the Patch code in it.



```
1 <?php
2
3 session_start();
4 include 'db.php';
5
6 $username = $_POST['username'];
7 $password = $_POST['password'];
8
9 // using prepared statements to prevent SQL Injection
10 $stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");
11 $stmt->bind_param("ss", $username, $password); // 'ss' specifies that both parameters are strings
12 $stmt->execute();
13 $result = $stmt->get_result();
14
15 if ($result != FALSE && $result->num_rows > 0) {
16     // successful login
17     $_SESSION['loggedin'] = true;
18     $_SESSION['username'] = $username;
19     header("Location: dashboard.php");
20     exit();
21 }
22 else {
23     // Login failed
24     echo "Login failed!";
25 }
26
27 $conn->close();
28
29 ?>
```

Now run this command in **docker compose up** where this file is available docker-compose.yml.

It will patch the SQL injection Vulnerability.

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

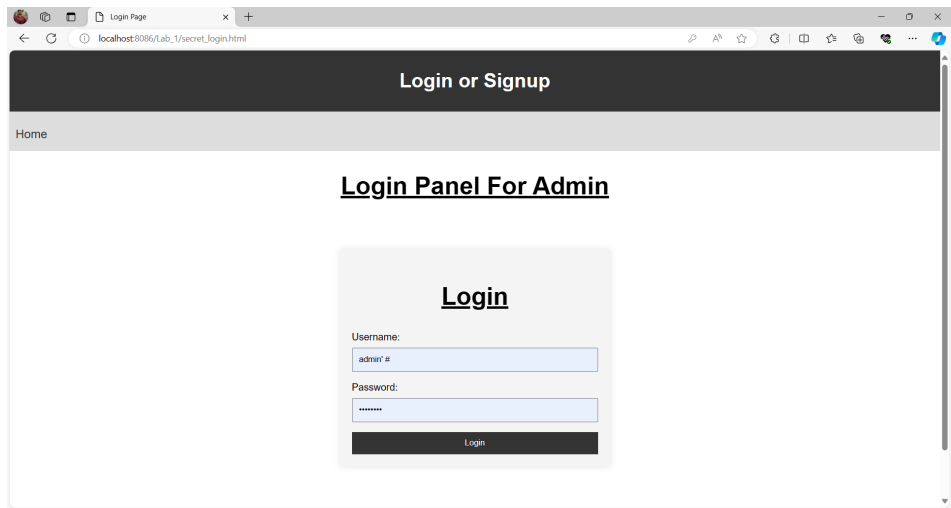
Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\prash\OneDrive\Desktop\ctf_project\ctf_lab\lab_1> docker compose up
[*] Running 2/0
  Container lab_1-mysql-1 Created 0.0s
  Container lab_1-php-apache-1 Created 0.0s
Attaching to mysql-1, php-apache-1
mysql-1 | 2024-03-24 17:31:16+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 5.7.44-1.el7 started.
mysql-1 | 2024-03-24 17:31:17+00:00 [Note] [Entrypoint]: Switching to dedicated user 'mysql'
mysql-1 | 2024-03-24 17:31:17+00:00 [Note] [Entrypoint]: Entrypoint script for MySQL Server 5.7.44-1.el7 started.
mysql-1 | 'var/lib/mysql/mysql.sock' -> '/var/run/mysqld/mysqld.sock'
mysql-1 | 2024-03-24T17:31:18.524828Z 0 [Warning] TIMESTAMP with implicit DEFAULT value is deprecated. Please use --explicit_defaults_for_timestamp se
mysql-1 | rver option (see documentation for more details).
mysql-1 | 2024-03-24T17:31:18.530419Z 0 [Note] mysqld (mysqld 5.7.44) starting as process 1 ...
mysql-1 | 2024-03-24T17:31:18.570740Z 0 [Note] InnoDB: PUNCH HOLE support available
mysql-1 | 2024-03-24T17:31:18.570856Z 0 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
mysql-1 | 2024-03-24T17:31:18.570868Z 0 [Note] InnoDB: Uses event mutexes
mysql-1 | 2024-03-24T17:31:18.570872Z 0 [Note] InnoDB: GCC builtin __atomic_thread_fence() is used for memory barrier
mysql-1 | 2024-03-24T17:31:18.570878Z 0 [Note] InnoDB: Compressed tables use zlib 1.2.13
mysql-1 | 2024-03-24T17:31:18.570887Z 0 [Note] InnoDB: Using Linux native AIO
mysql-1 | 2024-03-24T17:31:18.579145Z 0 [Note] InnoDB: Number of pools: 1
mysql-1 | 2024-03-24T17:31:18.590816Z 0 [Note] InnoDB: Using CPU crc32 instructions
mysql-1 | 2024-03-24T17:31:18.643084Z 0 [Note] InnoDB: Initializing buffer pool, total size = 128M, instances = 1, chunk size = 128M
mysql-1 | 2024-03-24T17:31:18.761544Z 0 [Note] InnoDB: Completed initialization of buffer pool
php-apache-1 | AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.20.0.3. Set the 'ServerName' directive gl
mysql-1 | obally to suppress this message
mysql-1 | 2024-03-24T17:31:18.826841Z 0 [Note] InnoDB: If the mysqld execution user is authorized, page cleaner thread priority can be changed. See th
php-apache-1 | AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 172.20.0.3. Set the 'ServerName' directive gl
mysql-1 | obally to suppress this message
mysql-1 | 2024-03-24T17:31:18.930936Z 0 [Note] InnoDB: Highest supported file format is Barracuda.
mysql-1 | 2024-03-24T17:31:19.010927Z 0 [Note] InnoDB: Creating shared tablespace for temporary tables
mysql-1 | 2024-03-24T17:31:19.019104Z 0 [Note] InnoDB: Setting file './ibtmp1' size to 12 MB. Physically writing the file full; Please wait ...
mysql-1 | 2024-03-24T17:31:19.176030Z 0 [Note] InnoDB: File './ibtmp1' size is now 12 MB.
mysql-1 | 2024-03-24T17:31:19.177506Z 0 [Note] InnoDB: 96 redo rollback segment(s) found, 96 redo rollback segment(s) are active.
mysql-1 | 2024-03-24T17:31:19.177528Z 0 [Note] InnoDB: 32 non-redo rollback segment(s) are active.
mysql-1 | 2024-03-24T17:31:19.180491Z 0 [Note] InnoDB: Waiting for purge to start
mysql-1 | 2024-03-24T17:31:19.230714Z 0 [Note] InnoDB: 5.7.44 started; Log sequence number 12223679
```

Let's try to check whether the SQL Injection payload in the web app is still Vulnerable or not.

From the previous Lab, this is the Payload that works.

6	admin' #	admin' #	302	false	false	431	
8	admin' or '1'='1	admin' or '1'='1	302	false	false	431	
10	admin' or '1'='1'#	admin' or '1'='1'#	302	false	false	430	
12	admin'or 1=1 or '='	admin'or 1=1 or '='	302	false	false	430	
15	admin' or 1=1#	admin' or 1=1#	302	false	false	430	



As you can see we successfully patch the vulnerability.



Let's try other payloads. we try all of them no one is working. we sucessfully patch this