



**PHPSecure**  
vulnerabilities scanner

# VULNERABILITY REPORT

**X**

Generated on October 30, 2023 at 1:28:52 AM UTC

PHPSecure.net

# INTRODUCTION

## What kind of vulnerabilities does PHP Secure scan for?

PHP Secure has checked your project on the **most common and dangerous types of vulnerability**:

1. SQL injection vulnerabilities
2. Command Injection (*also known as Shell Injection*)
3. Cross-Site Scripting (XSS) vulnerabilities
4. PHP Serialize Injections
5. Remote Code Executions
6. Double Escaping
7. Directory Traversal (*also known as File Path Traversal*)
8. Regular Expression Denial of Service (*ReDoS*)

Many of these vulnerabilities are a top risk in the [OWASP Top 10](#). They can provide attackers unauthorized access to an application and its data, potentially leading to breaches, data theft, and the compromise of your website. PHP Secure's advanced scanning capabilities can help proactively secure your code and safeguard against such attacks.

# SCAN RESULTS

## 1. Scan Coverage

 **100%** Scanned  
code

















**100% of your code was successfully scanned,** as no undefined classes were detected.

When there are no undefined classes or functions in your code, PHP Secure's analysis depth of finding vulnerabilities is very high.

PHP Secure is powered by a powerful and growing database of vulnerabilities. Multiple tests have shown that the scanner detects issues with very high accuracy and a minimal rate of false positives.

## 2. Detections

	Scan Time	Oct 30, 2023, 1:28 AM 4s
	Scanned Files	9
	Total Lines of Code	591
	Scanned Code %	100%
	Analysis Depth	Very High
	Analyzed Issue Types	8

	SQL Injections Found	1		Command Injections Found	0
	PHP Serialize Injections Found	0		Remote Code Executions Found	0
	Cross-Site Scripting (XSS) Found	0		Double Escaping Found	0
	Directory Traversal Found	0		Regular Expression Denial of Service Found	0

### Warning

**We detected 1 vulnerability in your code.** The vulnerability is listed below. Address it as soon as possible to prevent a potential breach.

Overall risk

Critical



● Low ● Medium ● High ● Critical

## 3. Vulnerabilities

### 3.1. Vulnerable files

#### Vulnerability Summary

Vulnerability Found: 1

login\_process.php

1

## 3.2. Vulnerable code segments

### login\_process.php

```
11
12      // TODO: Implement proper validation and sanitization for the input
13      fields
14
15      // Retrieve the admin credentials from the database
16      $sql = "SELECT * FROM admins WHERE username = '$username'";
17      $result = mysqli_query($conn, $sql);
18
19      SQL Injection has been found. Change this code to no longer construct SQL
20      queries directly from user-controlled data.
21
22      if (mysqli_num_rows($result) === 1) {
23          $admin = mysqli_fetch_assoc($result);
24          $hashedPassword = $admin['password'];
25      }
```

# WHAT DO I DO NOW?

The vulnerabilities detected in your code can potentially allow an attacker to manipulate or extract data from your database. It's crucial that you address this vulnerability as soon as possible to prevent any data breaches or unauthorized access.

## SQL Injections found

### ■ What is an SQL injection?

An **SQL injection** is a type of cyber attack where an attacker exploits vulnerabilities in a website or application to execute unintended database queries. The attacker manipulates user input fields (such as a search bar or login form) to inject malicious SQL code into the backend database. This can allow them to gain unauthorized access to sensitive information, modify or delete data, and even take control of the entire system.

SQL injections are unfortunately very common because of the prevalence of SQL injection vulnerabilities and the value of the target, as databases often contain important information that an attacker would find valuable.

**27.8%** of applications are vulnerable to SQL injections on initial scan.\*

\* Per PHP Secure statistics

### ■ How does an SQL injection occur?

Attackers commonly scout the application firewall for vulnerable user input points, like a search field or a contact form. After identifying a vulnerability, the attacker creates input content known as a **malicious payload** that executes unauthorized SQL injection commands at the backend database.



### Case Study: 2016 U.S. Election Breach

Ahead of the 2016 U.S. presidential election, nation state-sponsored attackers used SQL injection to compromise voter records databases in at least two states, potentially allowing the attackers to download or delete voter registration data and disrupt voting.

## ■ What risks do SQL injections pose?

Attackers commonly develop SQLi commands to perform a wide variety of malicious acts, such as:

- Stealing sensitive data, such as usernames, passwords, credit card information, and other personally identifiable information (PII)
- Modifying or deleting data in the database, leading to data loss or corruption
- Executing unauthorized commands on the server, allowing the attacker to take control of the system
- Denial of Service (DoS) attacks, which can cause system downtime and impact business operations
- Using the compromised application as a launching point for further attacks against other systems or networks

These attacks can lead to legal and financial consequences for businesses and organizations that experience a breach, in addition to compromising the security and integrity of the application.

## ■ SQLi - How can I fix it?

Fortunately, there are several steps that developers and security professionals can take to prevent SQL injections, such as using an intelligent code scanner like PHP Secure, input validation and sanitization, prepared statements, and stored procedures. These best practices and many others are included at the end of this report.

To address SQL injection vulnerabilities, it's important to sanitize user input before sending it to the database. Sanitizing involves escaping all potentially harmful characters to prevent them from affecting the generated SQL query. There are various methods to



achieve this, but two effective approaches are Object-Relational Mapping (ORM) libraries and prepared statements.

- ORM libraries provide a layer of abstraction between your application code and the database, which makes it easier to write secure SQL queries.
- Prepared statements are another approach that use placeholders for user input, allowing the data to be escaped during binding. All user input fields must be properly sanitized.

We recommend that you take immediate action to sanitize all user input fields and review your code to ensure that no additional SQL injection vulnerabilities are present. Conducting regular security testing is also essential to identify any new vulnerabilities that may arise. By addressing these vulnerabilities, you can help to protect your system and data from potential attacks.

# PRACTICAL PREVENTION TIPS

To help you ensure your code is secure, we have compiled a list of 10 best practices based on the OWASP Top 10 Proactive Controls, which are widely considered the gold standard for application security. Our list is written in concise, easy-to-understand language, and includes brief overviews of each control, coding examples, actionable advice, and further resources to help you create secure software.

## ■ Best Practice 1: Verify for security early and often

It used to be standard practice to security test an application near the end of the development cycle and then hand a list of bugs over to developers for resolution. However, tackling a laundry list of fixes days before release is no longer acceptable: scrimping on security puts your application at risk and can delay its launch.

Instead of backloading your security testing, you can detect vulnerabilities and protect against malicious actors when you scan early and often during the entire development lifecycle. Use a code scanner like PHP Secure that can provide remediation guidance within your development workflow.

### SECURITY TIPS:

- Incorporate data protection from the very start of your project, and make security a part of your project's definition of "done".
- Use the OWASP Application Security Verification Standard (ASVS) as a reference when defining security requirements and creating test cases.
- Collaborate with your security team or specialist to address vulnerabilities through testing methods.
- Incorporate proactive controls into your stubs and drivers.
- Make security testing a part of your continuous integration process for faster, automated feedback.

## SOLUTION

Use a code scanner like PHP Secure that can provide remediation guidance within your development workflow.

## ■ Best Practice 2: Validate all user inputs

To prevent injection attacks, it's important to [validate all user input](#) by limiting it to the required characters and ensuring that the data is both syntactically and semantically valid.

**Syntactic validity** refers to whether the input follows the expected format or syntax for the data type or data structure. For example, if a user is asked to input their email address, syntactically valid input would follow the format of "example@email.com". If the user input something like "example[at]email.com", it would be syntactically invalid.

**Semantic validity**, on the other hand, refers to whether the input makes sense in the context of the application or system. For example, if a user is asked to input their age, semantically valid input would be a positive integer. If the user input a negative integer or a non-numeric value like "twenty", it would be semantically invalid.

### SECURITY TIPS:

- Assume that all incoming data is untrusted.
- Develop allowlists for checking syntax. For example, regular expressions are a great way to implement allowlist validation, as they offer a way to check whether data matches a specific pattern.
- Input validation must take place on the server side. This extends across multiple components, including HTTP headers, cookies, GET and POST parameters (including hidden fields), and file uploads. It also encompasses user devices and back-end web services.
- Use client-side controls only as a convenience.

### EXAMPLE 1

This code checks whether a username, password, or email contains any invalid characters using regular expressions. If invalid characters are found, the code terminates with an error message, and if the input is valid, a database command is executed.

```
if (preg_match("/^[A-Za-z0-9]/", $username) ||
    (preg_match("/^[A-Za-z0-9\\!_]/", $password) ||
    (preg_match("/^[A-Za-z0-9_-@]/", $email)) {
    die("Invalid Characters!");
} else {
    # Run Database Command
}
```

## EXAMPLE 2

This code checks the validity of an email address after removing any illegal characters from it using the `FILTER_SANITIZE_EMAIL` and `FILTER_VALIDATE_EMAIL` filters. If the email address is considered valid, a message is printed to the screen.

```
<?php
$sanitized_email = filter_var($email, FILTER_SANITIZE_EMAIL);
if (filter_var($sanitized_email, FILTER_VALIDATE_EMAIL)) {
    echo "This sanitized email address is considered valid.\n";
}
```

## ■ Best Practice 3: Utilize PDO and placeholders to send data and queries separately

PDO and placeholders provide a way to separate data and the query itself, preventing injection attacks.

PDO is a good choice for connecting to various databases. However, if you are using a database other than MySQL, you may need to use a different driver-specific option such as `pg_prepare()` and `pg_execute()` for PostgreSQL.

Placeholders provide a secure method of binding input parameters to a prepared statement, which is then executed. This allows the database to distinguish between SQL code and the user's data, effectively neutralizing the threat of SQL injection.

## EXAMPLE 1

In the example, a PDO object is created with the database host, name, username, and password. Then, a prepared statement is created using a parameterized query.

Parameterized queries separate the SQL code from the data being inserted into it, so that any user input is sanitized and doesn't affect the SQL code. This effectively prevents SQL injections.

The parameterized query in the example includes the placeholder `:id` instead of directly including the user input. The `bindParam()` method is then used to bind the `:id` placeholder to the actual user input variable `$id`. Finally, the `execute()` method is called to run the query and retrieve the results.

```
$db = new PDO('mysql:host=localhost;dbname=test', $user, $pass);  
$stmt = $db->prepare("SELECT * FROM articles WHERE id=:id");  
$stmt->bindParam(':id', $id);  
$stmt->execute();  
$result = $stmt->fetchAll();
```

## EXAMPLE 2

This example uses the MySQLi extension, which is a driver specifically designed for use with MySQL databases. It demonstrates the use of prepared statements with parameter binding, which is similar to PDO.

By using a prepared statement with a placeholder, we can ensure that user input is properly sanitized and not interpreted as part of the SQL query, which can prevent SQL injection attacks. This is because the user input is treated as a parameter and is bound to the prepared statement separately, rather than being concatenated directly into the SQL query string.

```
$stmt = $dbConnection->prepare('SELECT * FROM employees WHERE name = ?');  
$stmt->bind_param('s', $name);  
$stmt->execute();  
$result = $stmt->get_result();  
while ($row = $result->fetch_assoc()) {  
    // do something with $row  
}
```

## ■ Best Practice 4: Use whitelists of allowed keys and values

Using whitelists is an effective way to limit the possible values that can be submitted in data, and can help prevent injection attacks. In this example, the use of a whitelist ensures that only the allowed keys are present in the data, and any other keys are rejected. This can prevent malicious users from submitting data with unexpected keys, which can potentially cause security vulnerabilities.

### EXAMPLE

```
$allowedKeys = ['username', 'password', 'email', 'password_confirmation'];  
$availableKeys = array_keys($_POST);  
$keysDiff = array_diff($allowedKeys, $availableKeys);  
if (!empty($keysDiff)) {  
    die('Invalid keys in request');  
}
```

## ■ Best Practice 5: Avoid using the GET method for forms

Using the GET method in forms can lead to security vulnerabilities, as sensitive data can be easily intercepted by malicious users or third-party applications. In the following example, the use of the GET method to retrieve the 'city\_id' parameter can be problematic, as it exposes the value of the parameter in the URL. This can allow malicious users to modify the parameter value and potentially perform unauthorized actions in the application.

The use of the POST method is recommended for forms that handle sensitive data or perform actions that modify the state of the application. Additionally, it is important to validate and sanitize all input data to prevent security vulnerabilities.

## EXAMPLE

```
$city_id = $_GET['city_id'];  
settype($city_id, 'integer');
```

## ■ Best Practice 6: Validate data inputs

One of the most common security threats to web applications is Cross-Site Request Forgery (CSRF) attacks. In a CSRF attack, a malicious user can exploit the trust between a user and a website by submitting a form that performs a malicious action on behalf of the user, without their knowledge or consent.

To prevent CSRF attacks, it's important to verify that the data submitted in a form has originated from a trusted source. The example below achieves this by checking the validity of the CSRF token submitted with the form. The CSRF token is a unique identifier that is generated by the server and is included in the form. When the form is submitted, the server checks whether the token is valid, which confirms that the form has originated from the expected source.

## EXAMPLE

```
$csrfToken = $_POST['csrf_token'] ??;  
if (!validateToken($csrfToken)) {  
    die('Invalid CSRF token');  
}
```

## ■ Best Practice 7: Restrict direct access to system files

Restricting direct access to system files is a key security measure that can help prevent attacks. Attackers can use injection to gain access to sensitive system files by manipulating input to force the server to execute arbitrary commands.

This best practice involves checking the requested file's directory against a pre-approved list of allowed directories to ensure that the file being accessed is within the approved locations. By restricting access to system files, you can help prevent attackers

from exploiting vulnerabilities in your system and protect sensitive data from unauthorized access or modification.

The code example demonstrates how to implement this best practice by retrieving the requested file name from the URL parameter and verifying that it is within one of the pre-approved directories before allowing access to the file.

## EXAMPLE

```
$filename = $_GET['file_name'];
$allowedDirectories = [
    '/var/www/avatars/',
    '/var/www/public/',
]
$directory = dirname($filename);
if (!in_array($filename, $allowedDirectories)) {
    die('Invalid file name')
}
fopen($filename);
```

## ■ Best Practice 8: Practice the principle of least privilege

To minimize the potential damage of a successful attack, you should minimize the privileges assigned to every database account in your environment. This security practice is known as the principle of least privilege.

The principle of least privilege states that a user or process should only be given the minimum level of access or permissions required to perform their function. By limiting privileges, you can reduce the risk of accidental or intentional data breaches, as well as limit the damage that can be caused by a compromised account.

Practically, this means that you should not assign DBA or admin type access rights to your application accounts. We understand that this is easy, and everything just "works" when you do it this way, but it is very dangerous.



## SECURITY TIPS:

- Use strong passwords and implement multi factor authentication to prevent attacks by making it more difficult for an attacker to gain unauthorized access to systems and resources.
- Start from the ground up to determine what access rights your application accounts require, rather than trying to figure out what access rights you need to take away. Make sure that accounts that only need read access are only granted read access to the tables they need access to.
- In general, each separate web application accessing a database should have a designated database user account that the application will use to connect to the DB. That way, the designer of the application can have good granularity in the access control, thus reducing the privileges as much as possible.

As an example, a login page requires read access to the username and password fields of a table, but no write access of any form (no insert, update, or delete). However, the sign-up page certainly requires insert privilege to that table; this restriction can only be enforced if these web apps use different DB users to connect to the database.

- If an account only needs access to portions of a table, consider creating a view that limits access to that portion of the data and assigning the account access to the view instead, rather than the underlying table. Rarely, if ever, grant create or delete access to database accounts.
- If you adopt a policy where you use stored procedures everywhere, and don't allow application accounts to directly execute their own queries, then restrict those accounts to only be able to execute the stored procedures they need. Don't grant them any rights directly to the tables in the database.

## Beyond SQL Injection: Parameter tampering

SQL injection is not the only threat to your database data. Attackers can simply change the parameter values from one of the legal values they are presented with, to a value that is unauthorized for them, but the application itself might be authorized to access. As such, minimizing the privileges granted to your application will reduce the likelihood of such unauthorized access attempts, even when an attacker is not trying to use SQL injection as part of their exploit.

## EXAMPLE

Suppose you have a web application that requires access to a database to retrieve user data. You can implement the principle of least privilege by creating a separate database user with restricted privileges that only has access to the necessary tables and columns. Below is an example of how this can be implemented in PHP using the PDO library.

In this example, a separate database user named `readonly_user` with restricted privileges is created. This user is only granted `SELECT` privileges on the `users` table, and only for the `name` and `email` columns. The application then uses this user to retrieve user data from the database, minimizing the risk of accidental or intentional modification of the data. This is an example of how the principle of least privilege can be applied in practice to reduce the attack surface of an application.

```
// Create a database connection using the least privileged database user
$dsn = 'mysql:host=localhost;dbname=mydb';
$username = 'readonly_user';
$password = 'password';
$options = array(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8');
$dbh = new PDO($dsn, $username, $password, $options);

// Retrieve user data using the least privileged database user
$stmt = $dbh->prepare('SELECT name, email FROM users WHERE id = :id');
$stmt->bindParam(':id', $user_id, PDO::PARAM_INT);
$stmt->execute();
$user_data = $stmt->fetch(PDO::FETCH_ASSOC);

// Close the database connection
$dbh = null;
```

## ■ Best Practice 9: Control access with views

You can use SQL views to further increase the granularity of access by limiting read access to specific fields of a table or joins of tables.

As an example, suppose that a system is required (perhaps due to some specific legal requirements) to store the passwords of users, instead of salted-hashed passwords.

A database designer could use views to compensate for this limitation by revoking all access to the table (except the owner/admin) and create a view that outputs the hash of the password field instead of the field itself. Any attack that succeeds in stealing database information will be restricted to stealing the hash of the passwords, since no user for any of the web applications has access to the table itself.

## ■ Best Practice 10: Always Use SSL Certificates

- To get end-to-end secured data transmission over the internet, always use SSL certificates in your applications. It is a globally recognized standard protocol known as Hypertext Transfer Protocol (HTTPS) to transmit data between the servers securely. Using an SSL certificate, your application gets the secure data transfer pathway, which almost makes it impossible for hackers to intrude on your servers.
- Regularly update software, web servers, databases, and operating systems.

## Conclusion

PHP applications are always vulnerable to external attacks, but using the tips mentioned above, you can easily secure the cores of your application from any malicious attack.

Besides these tips, many techniques can help you secure your web application from external attacks, like using the best cloud hosting solution that ensures you have optimum security features, cloud WAF, document root setup, whitelisting IP addresses, and more.

# ADDITIONAL RESOURCES

- For **background information on SQL injections**, see the [OWASP article on SQL injections](#) and [Blind SQL injections](#).
- For **developers and other technical professionals**, the [OWASP Developer Guide](#) contains valuable information on mitigating vulnerabilities.
- For a **wiki of code review resources**, visit the [OWASP Code Review Guide](#), especially the [Reviewing Code for SQL Injection](#) article.
- For **practical testing advice**, see [Testing for SQL Injection](#) under the [OWASP Testing Guide](#).