THE MOST FORGOTTEN WEB VULNERABILITIES

Written by 0KaL @ WhiteCollarGroup

Reviewed by WCG147, 3du and NeoInvasor

2016

“Now you know

who you’re fighting.”

(Quote from Troy movie)

The Most Forgotten Web Vulnerabilities
Written by 0KaL (@0KaL_H4) for WhiteCollarGroup (@WCollarGroup)
Reviewed by WCG147, 3du and NeoInvasor (thanks to Brazilian Cyber Army)
For SystemHacker, #0rch1d and #M0rph
Version: 1.0 published in February, 10th 2016

Introduction

PHP is adorable. It’s learning curve is short, yet its features allow you to create virtually any kind of web application.

But PHP, as some use to say, allows the programmer to do a lot of weird things. While I really think it’s a good aspect for a programming language, for beginners it might bring a lot of headaches and angry clients.

The vulnerabilities that we are going to see here really do deserve some attention from us (as any other vulnerability). They’re not vulnerabilities that will provide, to some attacker, permissions to hack the entire server (maybe this is the reason why it’s so forgotten), but this does not mean that they can’t create a path, sometimes with some basic social engineering[1], for the attacker to reach this target.

[1] Act of using communication for fooling and getting privileged information about the victim.

Table of contents

Attribute-based Cross-site Scripting

You may be already aware about XSS (Cross-site scripting, the "X" is to avoid confusion with Cascading Stylesheet[2]). Let's say that you must receive some HTML code from the client and save it to display later. As you really must receive HTML tags, `htmlspecialchars()`[3] won't help us. **It's good to remember that WYSIWYG editors can be easily disabled by some attacker, and also need HTML filters on the server side.**

You're smart enough to avoid some attacker to include iframes or scripts on your page by whitelisting some HTML tags:

```
$content = strip_tags($_POST['content'], '<b><i><u><p><div><img>');
```

But what about the HTML events?

For example, can you imagine what happens if the user sends the code below? (Let's remember an attacker could easily, through Developer Tools, for example, bypass a WYSIWYG editor with no HTML button and have a simple textarea, instead).

```
<img onLoad="javascript:alert('xss');" src="http://placehold.it/350x150" />
```

Let's see: you've blocked `<script>` tag, but that's not the only way to execute Javascript code in HTML. The code above still runs and the alert is being executed (and if an alert can be performed, a cookie or session stealing can also happen - we will see why it is dangerous in this article).

The "onLoad" event is available for the following HTML tags, according to W3schools:

```
<body>, <frame>, <iframe>, <img>, <input type="image">, <link>, <script>,
<style>
```

[2] A language for setting and creating styles for web pages.
[3] Htmlspecialchars() converts HTML brackets to entities, so it will show up as text in the browser, and not evaluated.

But keep in mind that there are more DOM events available, with support for almost all HTML tags. See a complete list on W3schools.

Also, not just events may cause problems. If the user just wants to mess everything up, he could do it easily by inserting CSS code into style attribute. He could, for example, create a fixed div written "Hacked by Haxor" that occupies the whole window.

How can I protect?

As you've seen, strip_tags() isn't enough. It's needed to whitelist tags and attributes. Andstrip_tags() only filters tags. Unfortunately PHP does not have a function that filters attributes. Luckly, there are many options out there that will help us. One of them is HTML Purifier.

After downloading, it's very easy to use and it will remove the dangerous HTML tags and attributes. For example:

```
<?php

require_once './htmlpurifier/library/HTMLPurifier.auto.php';

$dirty_html = '<img onLoad="javascript:alert(\'xss\');"
src="http://placehold.it/350x150" /><script> alert(1); </script>';

$config = HTMLPurifier_Config::createDefault();
$purifier = new HTMLPurifier($config);
$clean_html = $purifier->purify($dirty_html);

echo $clean_html;
```

Will return:

```
<img src="http://placehold.it/350x150" alt="350x150" />
```

Another way of injecting attributes may be made possible when you insert some user input on another attribute. Let’s take a look on this example:

```
<?php

$link = htmlspecialchars($_GET['link']);

?>

<a href='<?php echo $link; ?>'>Click here</a> to continue.
```

If the user accesses `yoursite.com/script.php?link=http://google.com' onClick='alert(1);`, this will be the result:

```
<a href='http://google.com' onClick='alert(1);'>Click here</a> to continue.
```

Why did it happen? By default, `htmlspecialchars()` will only replace double-quotes (`"`) to `&quot;`. As the example above uses single quotes (`'`), it’s needed to tell`htmlspecialchars` to convert it as well. It can be done simply by adding `ENT_QUOTES` constant as second argument. It will make both the quotes to be converted to entities.

```
$link = htmlspecialchars($_GET['link'], ENT_QUOTES);
```

Now, this is what we get:

```
<a href='http://google.com&#039; onClick=&#039;alert(1);'>Click here</a> to
continue.
```

Of course, some validation with `filter_var()` is also highly recommended.

Multi-level SQL Injection

We know that it’s needed to filter user inputs, do some casting or use prepared statements with placeholders to protect against SQL Injection (which is the addition of more SQL queries from a variable controlled by the attacker). But SQL Injection can also be multi-level (or “multi-order”).

Let’s say that you have a forum with the following database tables:

Table: “topics”

id	author	title	text
1	2	Hi	Hey!
2	2	Hey	Hi!

Table: “users”

id	username	password
1	admin	(pwd)
2	hacker	(pwd)

And let’s say that, when showing a topic, you query:

```
$id = (int)$_GET['id']; // protecting!

$qry = your_query_function("

    SELECT u.username, t.title. t.text

    FROM topics t

    LEFT JOIN users u

        ON t.author=u.id

    WHERE t.id=$id");
```

But you have to show links to other topics by this user below the user’s post. In order to produce it, you do:

```
$qry = your_query_function("
    SELECT u.username, t.title. t.text
    FROM topics t
    LEFT JOIN users u
        ON u.username='$row[author]'");
```

As `$row[author]` comes directly from the database, and not from some user input, and you’ve protected it with some `real_escape_string()`-like function, you’re safe, right?**Wrong.**

Let’s say that the attacker, while registering, have set his username as `' UNION ALL SELECT version(), version() --`. As when registering you’ve protected the string before inserting it to the database, it was just stored on it, and not injected. But once the attacker opens one of his topics and our “More topics by this user” query runs, let’s see what we will have:

```
SELECT u.username, t.title. t.text
FROM topics t
LEFT JOIN users u
    ON u.username='' UNION ALL SELECT version(), version() -- '
```

Here we see that his SQL Injection was successful: in a second level. Of course this is a harder attack to do, but when a hacker wants, a hacker does.

How can I protect?

- Always protect/cast values before inserting it to your SQL query, even when it comes from the own database.
- Avoid inserting variables inside your SQL queries. Never do it if your database engine supports prepared statements.

Cross-site Request Forgery

Cross-site Request Forgery (CSRF) means making the user to do something he wouldn’t want to.

Let’s say you have a page for posting a message:

```
<form action="post.php" method="post">

<input name="message" type="text" />

<input type="submit" value="Post it now" />

</form>
```

Too simple: the user writes something, click the submit button and post.php will save the content of the message input into your database.

But what if a naughty hacker creates a simple HTML page with the following content:

```
<iframe id="haxor" style="display: none;" name="haxor" width="300"
height="150"></iframe>



<form action="http://your_beautiful_site.com/post.php" method="post"
target="haxor" id="haxorfrm">

<input name="message" type="hidden" value="HAXORED BY HAXOR" />

</form>

<script type="text/javascript">

document.getElementById('haxorfrm').submit();

</script>
```

Let’s analyze it:

- A hidden iframe called “haxor”. The user won’t see it.

- A form whose only input is a hidden one (the user won't see it), which targets haxor iframe[4] (the user won't see it being submitted) and that submits itself once it's loaded.
- The message input, hidden, with the value "HAXORED BY HAXOR".

Let's imagine the sheet: the hacker saves it to PasteHTML or any free hosting site, then send it to the victim while he's logged on (there are many ways to make sure that the victim is logged on, like sending it from a comment, guestbook or contact form) - note that, if the user isn't logged on, nothing happens (if you've checked for sessions everywhere), but if someday he logs in and opens that link, you'll be dammed.

Once this page is opened, the form is submitted through the hidden iframe (so the user will not see anything) and its action will be performed: the "HAXORED BY HAXOR" will be posted.

How can I protect?

Generate a token and verify if on every single action that modifies anything on the data.

Before showing the form, just generate a random token, put it on a session and inside a hidden input and verify it when the form is submitted. Like this:

```
<?php

session_start();

$token = md5(uniqid(rand(), true));

$_SESSION['csrftoken'] = $token;

?>

<form action="post.php" method="post">

<input type="hidden" name="csrftoken" value="<?php echo $token; ?>" />
```

[4] Iframe is a HTML tag to include a page inside another, as it was another browser window "included" in the page. Not the same as popup as it's part of the page elements and has no close button.

```
<input name="message" type="text" />

<input type="submit" value="Post it now" />

</form>
```

On post.php:

```
<?php

session_start();

if($_POST['csrftoken']!=$_SESSION['csrftoken']) {

  echo "CSRF attack detected!";

  exit;

}

// save in db, etc
```

Of course if you have several forms on your page (which is pretty usual) it will be a pain. For this case, I've made a small class that will help you. Download it here, include it on your script and just do:

```
<?php

require 'csrf.class.php';

$token = csrf::token('post_form'); // create an identifier for your form or action, so you can have multiple tokens

?>

<form action="post.php" method="post">

<input type="hidden" name="csrftoken" value="<?php echo $token; ?>" />

<input name="message" type="text" />

<input type="submit" value="Post it now" />

</form>
```

And in post.php:

```
<?php
```

```
require 'csrf.class.php';

if(!csrf::verify('post_form', $_POST['csrftoken'])) { // once you verify, the existing token will be removed after being validated

   echo "CSRF attack detected";

  exit;

}

// save to db, etc
```

If you don’t want to download the class from the link, just copy it here:

```
<?php

@session_start();



/*

 * @author @0KaL_H4

 */



class csrf {

    public function token($action) {

        if (!isset($_SESSION[md5(dirname(__FILE__))]['csrf_tokens'])) {

            $_SESSION[md5(dirname(__FILE__))]['csrf_tokens'] = array();

        }

        $gen_token = md5(uniqid(rand(), true));

        $_SESSION[md5(dirname(__FILE__))]['csrf_tokens'][md5($action)] = $gen_token;

        return $gen_token;

    }
    public function verify($action, $check) {
        if (!isset($_SESSION[md5(dirname(__FILE__))]['csrf_tokens'])) {
            $_SESSION[md5(dirname(__FILE__))]['csrf_tokens'] = array();
            return false;
        }
        if (isset($_SESSION[md5(dirname(__FILE__))]['csrf_tokens']) AND

isset($_SESSION[md5(dirname(__FILE__))]['csrf_tokens'][md5($action)]) AND
                $_SESSION[md5(dirname(__FILE__))]['csrf_tokens'][md5($action)]
== $check) {

unset($_SESSION[md5(dirname(__FILE__))]['csrf_tokens'][md5($action)]);
            return true;
        } else {
            return false;
        }
    }
}
```

How about non-forms?

If you have any GET request that can perform some action on the site, and you don’t check for a token, you’re still vulnerable. For example:

```
<img src="http://yoursite.com/logout.php" />
```

Will logout the user, and:

```
<img src="http://yoursite.com/admin/delete_post?id=123" />
```

You can see that possibilities are unlimited.

So what must you do? Just check for a token too!

```
<?php

require 'csrf.class.php';

$token = csrf::token('logout');

?>

<a href="logout.php?token=<?php echo $token; ?>">Logout</a>

<?php

require 'csrf.class.php';

if (!csrf::verify('logout', $_GET['token'])) {

    echo "CSRF attack detected!";

    exit;

}



// do logout, etc
```

Local File Inclusion

PHP inclusion[5] consists in two vulnerabilities: local and remote file inclusion (LFI and RFI). Remote file inclusion receives a good attention from developers, since it would allow any attacker to set a remote (from another server) PHP file to be included, and it could easily be a webshell.

But Local File Inclusion may also be dangerous. And this is not about stealing the server's /etc/passwd or /etc/shadow[6]. It's still possible to create PHP codes inside the server, then execute it.

Here's how: Apache saves every request into your Apache Access Logs. Let's say the client (attacker) has sent the following request, using NetCat (as a web browser would convert some tags to entities and make it hard):

```
$ nc yoursite.com 80

GET /<?print`id`;?> HTTP/1.0



(twice enter)
```

Your webserver would throw some 404 error page to the attacker's command line, but his request would have been saved to the access logs. Once he can discover where it's located (and hosting companies tend to standardize it), and include the Apache Access Logs through some LFI vulnerability, that code would be executed. It was just an id command in the terminal, but could be easily a wget[8] downloading some webshell.

[5] PHPi is any kind of vulnerability that allows the inclusion of dangerous PHP code.
[6] /etc/passwd is where the users data (username, group...) is stored on Unix, while /etc/shadow stores the encrypted passwords.
[8] Wget is a command line tool for downloading files.

How can I protect?

If you really can’t predefine what files can be included, at least put enough protection in the variable that will be used as include, like so:

```
<?php

$include = basename($_GET['include']);

if(preg_match("^([a-z]+)$") AND file_exists('includes/'.$include.'.php')) {

    include 'includes/'.$include.'.php';

} else {

    // 404 error

}
```

Session fixation

Sessions are used for storing temporary information about the user. On every running script where sessions are used, you must call `session_start()` function, so a cookie named, by default, `PHPSESSID` is sent to the user (this name can be changed). This cookie stores an unique ID to identify the user every time he accesses the server again, and get only the sessions that he owns.

However, it’s easy to change PHPSESSID through some reflected or stored/permanent XSS vulnerability on the site, using Javascript. Also, it’s possible to do the same by setting it on the URL:

```
http://your_site.com/login.php?PHPSESSID=<a session id here>
```

Once the attacker sends this URL to the victim and the victim opens it, will be using the SESSID created by the attacker. This means that, once the attacker changes his SESSID to the same value using any cookie editor (most of the browsers already have it natively through Developer Tools), or uses his SESSID, all the sessions will be shared between the victim and the attacker. Could you imagine it? Once the victim logs in, the attacker will also be logged in on the same account, as the sessions are the same.

There will also be problems if the user copies the URL and shares it (those who access that link will have the same sessions).

How can I defend myself?

You must make sure that you `session.use_trans_sid` and `session.use_only_cookies` on your php.ini file are set in a way that denies this attacks:

```
session.use_trans_sid=Off

session.use_only_cookies=On
```

As nowadays any browser will support cookies, it’s not needed to have a session ID on the URL anymore.

It does not protect, however, if your application has some XSS vulnerability somewhere and the attacker could post Javascript code to be executed for the victim.

Of course, your application shalln't be vulnerable to XSS, as it could lead to several security problems. But to make things a little better, it's recommended to call `session_regenerate_id()` when the user logs in. It will generate a new SESSID and send it to the browser, and as it's unpredictable by the attacker, you will finally be safe.

Also, fix up the XSS vulnerability, if you have some. Or you will still be vulnerable. Let's see why? Read on the next topic:

Session Hijacking

Even if you regenerate the user session once the user logs in, if the attacker can get the session ID again through some XSS vulnerability on your site, he could just set his PHPSESSID cookie on his browser to the new victim's PHPSESSID value, then just refresh the page and, here we go, all the sessions are being shared again!

Unfortunately, there is no way: you must fix your XSS vulnerability. Of course the attacker still can access the user cookies if he obtains physical access to the victim's machine (and *physical access is root access*), or send him some trojan horse that will steal the cookies and send him back. But if the attacker could do it, he could easily also send a keylogger and get the password, instead of just cookies. So it's not a problem of mine, since it's the user's lack of knowledge about basic security, right? Wrong. There are ways to protect against it too. One of them is allowing the user to use OTP (one-time password) and two-steps verification. It's not as hard as it seems to be, and there are many ready-to-use PHP classes out there. A basic search about TOTP could put you in the way. But it's a matter for another article.

We've heard that security is never enough, so let's just make it more difficult to the hacker. We can store a key based on user's browser ("User Agent") to the sessions, and if this key is not valid, we log him out.

We can do it just by putting this code on every page (a global include, of course):

```
<?php

@session_start();

if (!isset($_SESSION['HTTP_USER_AGENT']) OR
$_SESSION['HTTP_USER_AGENT']!=$_SERVER['HTTP_USER_AGENT']) {

    session_destroy();
    header("Location: login.php");

    exit;

}
```

And when the user logs in, we must save his user agent to a session, of course:

```
$_SESSION['HTTP_USER_AGENT'] = $_SERVER['HTTP_USER_AGENT'];
```

Of course the attacker can get, through some XSS vulnerability, the victim’s user agent too, but we’re just making things hard. The answer for session hijacking is: fix up your XSS vulnerability!

Internal session stealing

When using shared hosting plans, unfortunately we will face many risks. A badly maintained server will have some old kernel and some website hosted on the same machine will be hacked, this hacker will use a local root exploit or some symlink bypass exploit and get access to your site as well. Unfortunately there are things that we are limited just to asking for protection from the hosting company, and keeping backups (or migrating to some VPS/dedicated).

But there are things that we can do by ourselves, instead of giving it to the server software. Sessions, for example. The main problem with sessions is that they're all, by default, stored on the same folder (usually the temporary folder, which is tmp for Unix servers), regardless of the site, and a webshell, as it runs in the same user than the webserver, can read these files.

Fortunately, PHP gives us options to store these sessions in another folder, as we want. We could just make a folder, put it outside our `public_html` or deny access to it through .htaccess, make sure our script isn't vulnerable to Local File Inclusion and put it in our php.ini:

```
session.save_path = "/home/your_username/sessions_folder/"
```

Note that the entire path (from system root) is required. You can also define it using the `session_save_path()` function before `session_start()`. If, for some reason, your server does not allow it, then you could use `session_set_save_handler()`. It allows you to create your own sessions handler and save them as you want. In the documentation for this function, there are examples of how to create custom handlers for sessions, showing even how to use the database to store sessions.

And now?

Well, *now you know who you’re fighting.* Know your enemy, think like him, hack yourself. Stay safe.