

WHITEPAPER

# How we found 5 0days in WordPress

Simon Scannell, **RIPS Technologies**

[www.ripstech.com](http://www.ripstech.com)

18.11.2019

<b>1. Introduction</b>	<b>2</b>
<b>2. Methodology</b>	<b>3</b>
2.1 Traditional Code Audit Approaches	3
2.2 Drawbacks of Traditional Approaches	3
2.3 Case Study	4
2.3.1 Limited Local File Inclusion	5
2.3.2 Breaking the Limitation	5
2.4 Finding an Efficient Methodology	7
2.4.1 Step #1 - Component Identification	7
2.4.2 Step #2 - Feature Breakdown	8
2.4.3 Step #3 - Feature Vulnerabilities	9
2.4.4 Step #4 - Vulnerability Chains	10
<b>3. Vulnerability Analysis</b>	<b>12</b>
3.1 CVE-2018-12895: Authenticated File Deletion	12
3.1.1 Abstracting the Media File Functionality	12
3.1.2 Background - Understanding Post Meta Entries	13
3.1.3 Insufficient Validation in Post Meta Component	14
3.1.4 Impact and Limitations	15
3.2 CVE-2019-8943: Authenticated Path Traversal and LFI	16
3.2.1 Abstracting the Image Cropping Functionality	16
3.2.2 Path Traversal in Image Editing	17
3.2.3 Impact and Limitations	19
3.3 CVE-2019-9787: Unauthenticated CSRF to XSS	20
3.3.1 Abstracting the Comment Functionality	20
3.3.2 Sanitization Bypass in SEO Optimization	21
3.3.3 Limitations and Bug Chaining	22
3.3.4 CSRF Vulnerability in Comments	22
3.3.5 Impact and Limitations	24
<b>4. Exploitation Chain</b>	<b>25</b>
4.1 Step #1 - Plugin Vulnerabilities	25
4.2 Step #2 - Attacking WordPress Core via CSRF	25
4.3 Step #3 - Exploiting Authenticated Vulnerabilities	26
4.4 Bonus: Wormable Stored XSS on WordPress.org	27
4.5 Putting it all together	28
<b>Summary</b>	<b>30</b>
<b>References</b>	<b>31</b>

# 1. Introduction

WordPress is a highly popular content management system used by over 34% of all websites on the internet. It's ease of use, great compatibility with a variety of servers and its huge list of free and powerful plugins (over 50.000) make WordPress the first choice for quickly and easily setting up a website without any technical knowledge or excessive budget. WordPress can be customized and optimized to a point that even governments and billion-dollar corporations use this blogging CMS to manage their websites. From america.gov to the swedish government, and from Microsoft to Facebook: they all use WordPress.

The popularity of this CMS makes it an attractive target for cybercriminals seeking to find previously unknown, and hence unpatched, vulnerabilities (*0days*) in order to take over as many websites as possible. Also nation states and other sophisticated hacking groups are interested in backdooring high value targets. We observed that the high interest in WordPress' security by different groups lead to many vulnerabilities being discovered and patched in the past. Additionally, bug bounty programs and 0day acquisition platforms attract a vast amount of bounty hunters that slowly but surely squeezed easy to find vulnerabilities out of the WordPress core. Hence, the well-reviewed code of the most popular web application is a great challenge but also a good candidate to experiment with different approaches of code auditing.

When we started our vulnerability research on the WordPress core code, we quickly realized that in order to find critical vulnerabilities one must move away from the traditional paradigm of how to find simple vulnerabilities in web applications and come up with more effective approaches and methodologies to source code auditing. This paper documents our approach of separating source code into components and combining several low-impact bugs into powerful Privilege Escalation and Remote Code Execution exploits. As a result, we found and combined five vulnerabilities into a powerful exploit chain that in the end allowed unauthenticated attackers to take over any high value target running WordPress. All issues have been responsibly disclosed to the WordPress security team and a patch is available. We believe that our documentation of vulnerability discovery does not only help other researchers to manifest their audit methodology but also helps developers to better understand the mindset of attackers and to sharpen their mindset for secure coding.

## 2. Methodology

This chapter introduces the methodology that we have used to audit the WordPress core. We will first look back on the most common approach to audit source code and point out drawbacks when analyzing well-reviewed applications such as WordPress. We will then present a case study and introduce a step-by-step guide to follow our methodology used.

### 2.1 Traditional Code Audit Approaches

The easiest way of thinking about vulnerabilities in web applications is in terms of an *user input* → *sink relationship*. A *sink* is simply a dangerous feature that may allow to execute security sensitive operations when untrusted user input is passed to it without being properly sanitized beforehand. For example, a database, a file write or an executed system command poses such a sensitive feature that could be abused by attackers if they manage to influence the feature with user-supplied data.

A very simple example of such a vulnerability could look like the following:

```
1 $dir = $_GET['dir'];  
2 ...  
3 system("ls " . $dir);
```

To find such bugs, researchers traditionally follow the trace of every input that can be controlled linearly through the code and verify if the controlled data is passed insecurely to a sensitive sink. Alternatively, a list of sensitive sinks is collected and the trace of data flow is followed backwards-directed. Since the data flow must be followed across function and file boundaries, **static code analysis** tools can help to automate this time-intense process.

### 2.2 Drawbacks of Traditional Approaches

Due to the high value of 0days in wide-spread web applications such as WordPress though, many skilled hackers and researchers already took on the challenge of tracing every single controllable user input through hundreds of thousands of lines of code to find vulnerabilities that follow the traditional *input* → *sink* pattern. This slowly but surely squeezed easy to find vulnerabilities out of highly popular applications over time.

As a consequence, the vulnerabilities that are still being discovered today are increasingly complex. New vulnerabilities that are found in highly popular source codes changed from being first-order vulnerabilities to complex exploit chains that are second, third - or even more - order vulnerabilities. This means that more and more steps are involved before an *input* actually hits a *sink*. More importantly, between each step there might be a condition that must be fulfilled before you can move on to the next step.

When we scanned the WordPress core with our static code analysis solution RIPS we mainly discovered seemingly low-impact vulnerabilities that follow the *input* → *sink* pattern. Often, these vulnerabilities were considered to be non-exploitable by a limitation imposed by the application. Many bugs look like “almost” vulnerabilities but cyber criminals, bug bounty and 0day acquisition programs are only interested in fully exploitable issues. So these low-impact bugs were probably left behind unreported by other researchers.

We learned that in order to come up with new critical security issues in well-reviewed code, the most promising approach is to combine several low-impact flaws into high-impact exploit chains. The difficulty in combining multiple low-impact vulnerabilities lies in establishing a connection between those issues. The different issues that could be chained together might be located in entirely different functionalities of a web application and have no obvious connection to each other within the source code itself.

This lack of this obvious connection is the reason why the traditional approach overlooks a new exploitable vulnerability in the core. When a researcher only tries to trace user-controlled data through the source code of an application to see if it is directly passed to a critical sensitive sink, he will face limitations imposed by the application that he audits. Since these limitations often cannot be bypassed directly, researchers give up and deem the traced path they were following as non-exploitable. However, the limitations can often be broken by abusing a bug at an entirely different location in the source code.

## 2.3 Case Study

Let's have a look at a classical real-world example of such a vulnerability that we discovered in the WordPress core. We chained two bugs/features that have no connection to each other within the source code. But each of those bugs would have been considered as not exploitable when looking at them separately.

Please note that the following example requires administrator privileges for exploitation and poses as a technical example. Although it is possible by default for administrators

to execute arbitrary PHP code by uploading new plugins and themes, these features could also be disabled on a properly hardened WordPress installation that followed the official [hardening guide](#) of WordPress. We will look at other *unauthenticated* vulnerabilities in [Section 3](#).

### 2.3.1 Limited Local File Inclusion

Our static code analysis tool reported a second-order Local File Inclusion (LFI) vulnerability in the theme component of the WordPress core ([RIPS vulnerability report](#)). It allows an attacker to `include()` and execute any file, independent of its extension, but only from within the currently active theme directory. A theme in WordPress is supposedly an existing directory that contains template files, as well as stylesheet / JavaScript files. Hence, this LFI vulnerability is limited and cannot be exploited directly.

WordPress securely ensures that the second-order user input that is passed to the `include()` call is the name of a file that actually exists within the currently active theme directory. It is impossible to deploy a Path Traversal attack to bypass this limitation. It is also not possible, at least on a properly hardened WordPress installation, to upload files to this theme directory or to modify the content of files within the theme directory (e.g. change the contents of `.css` files). This means that although it is possible to include any file from within the active theme directory, it is not possible to include malicious PHP code. As a result, this issue might be considered as a valid feature rather than a security bug and is not patched by WordPress at the time of writing.

But what if we can find another vulnerability in an independent component of WordPress that would allow us to sneak PHP code into that directory?

### 2.3.2 Breaking the Limitation

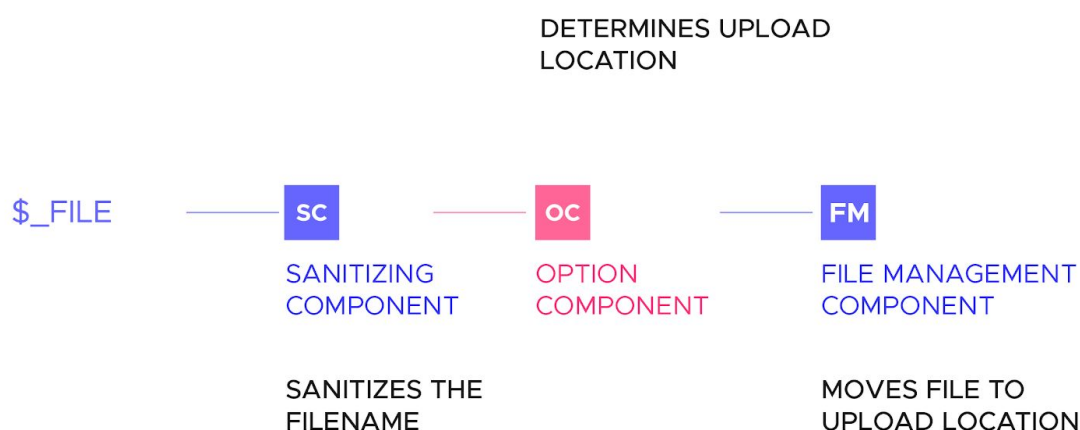
Since WordPress ensures that an included page template must be actually located in the currently active theme directory, we knew that the only way to exploit the vulnerability was to somehow modify an existing file within the theme directory, or to upload or move a file that contains our malicious PHP code to that directory.

In order to effectively find such weakness in the WordPress code that would allow us to modify files, we decided to only focus on a few components in the WordPress core that would likely contain such a weakness. We cut down our research effort to the *plugin update* component and the *media file upload* component because some file processing needs to happen there.

However, we quickly realized that looking at the file upload functionality of WordPress without a structured approach would not yield any results. This is because uploaded files and their filenames are sanitized and accessed by various functions and various different components of WordPress play a role in the file upload process. Capabilities play a role in which files can be uploaded, certain file types are handled differently, internal references to the file are stored in the database and so on.

For this reason we broke down the upload functionality for each file type into a simple, linear process. For example, when a `.txt` file is uploaded to a WordPress site, WordPress will 1) sanitize the file name, then 2) fetch the `upload_path` setting from the database which determines the directory that the `.txt` file will be moved to, and finally, 3) move the file to the destination directory.

## FILE UPLOAD PROCESS



With this abstraction of the file upload process for a `.txt` file, we quickly discovered a way to exploit the limited LFI vulnerability. If an attacker could overwrite the `upload_path` setting, he could point it to the directory of the currently active theme. An uploaded `.txt` file would then simply be moved to the theme directory. It is indeed possible to make the `upload_path` setting point to any directory.

<code>uninstall_plugins</code>	SERIALIZED DATA
<code>upload_path</code>	/point/to/any/path
<code>upload_url_path</code>	



This allowed us to change the `upload_path` to make it point to the same directory where the currently active theme is located. For example if WordPress' default *twenty nineteen* theme was used, one would have to simply set the `upload_path` to `wp-content/themes/twenty nineteen`. By abusing this bug, it is possible to break the limitation of not being able to upload files to the currently active theme directory. By combining both vulnerabilities, an attacker can execute arbitrary PHP code.

## 2.4 Finding an Efficient Methodology

The case described above was a simple example of a vulnerability that can only be exploited when a researcher establishes a connection between two bugs (or *features*) that are located in different locations of the source code.

Here, the traditional approach of simply following user input through the source code of WordPress until it reaches a sink was not sufficient. A researcher who would have only followed the user input that is passed to `include()` in the theme component would have found a non-exploitable File Inclusion vulnerability but not notice the connection to the option component of WordPress, as there is no reference to it in the code. While this works well for developers who only need to patch potentially vulnerable code, a researcher needs to find a way of exploitation.

Our goal was to audit the source code in a more structured approach and with a methodology in mind that helps us to connect multiple low-impact vulnerabilities on a logical level. We could then chain separate bugs into more powerful high-impact vulnerabilities. For this purpose, we structured our audit in four steps.

### 2.4.1 Step #1 - Component Identification

Since the connections between low-impact bugs often exist on a logical and architectural level of the web application, it makes sense to break down the web application into functional components that each have a unique purpose contributing to the application instead of simply looking at a web application as a heap of functions and classes.

A component could, for example, be the *theme* component of WordPress. The theme component holds all functionality and logic that is required by a WordPress theme.



## OVERVIEW



Another component might be the *file management* component of WordPress which handles all file operations.

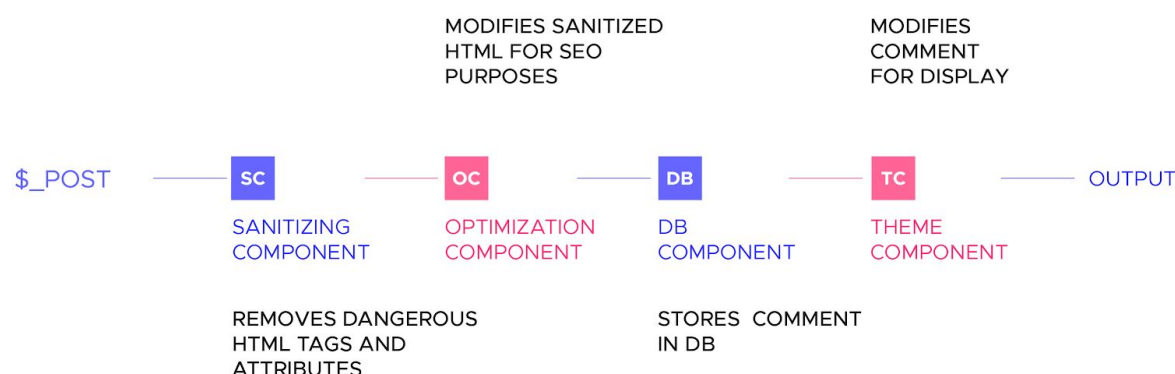
Another way of thinking about these components are black boxes. A component receives data, processes it and passes it on. For example, the theme component receives data, such as the type of a blog post that should be displayed and the data that should be inserted into a template.

What exactly a component is and what the purpose of the abstraction is will become clear when a practical example is given in the next step.

## 2.4.2 Step #2 - Feature Breakdown

When researchers audit a specific functionality for weaknesses, for example how a comment is created on a site, it makes sense to break down the functionality into a series of the involved components.

## COMMENT FUNCTIONALITY



For example, when a comment is added to a blog post of a WordPress site, the user input is passed through multiple components. The comment is 1) first sanitized against XSS attacks (XSS-sanitization component), 2) then optimized for SEO purposes (optimization component), 3) then stored in the database (database component), and eventually when a user wants to view the comment, 4) fetched from the database and modified again (optimization component), before it is finally 5) embedded into the resulting HTML page (theme component). As a results, we can map the comment feature to a series of *five* different components of the WordPress core.

### 2.4.3 Step #3 - Feature Vulnerabilities

Once we have broken down a functionality into such a series of components we can ask ourselves for each involved component: “What is the purpose of this component in the functionality and what could go wrong for that functionality?”.

For example, in step #2 when we analyzed the example of how a comment is created in WordPress, there is a SEO component that modifies the HTML markup of the comment string after it has been sanitized. The answer to the question “What is the purpose of this component in the functionality and what could go wrong?” is that the component is designed to parse, modify and optimize the HTML markup of the comment string for SEO purposes. A Cross-Site Scripting vulnerability could occur if the parsing and modification process can be broken. We will show more practical examples in [Section 3](#) that demonstrate how we approached finding vulnerabilities in single components.

A researcher can effectively search or scan for flaws in single components when he knows exactly what to look for. Searching for flaws in single components with a lot of context in mind (the context depends on the purpose of the component within the functionality), a researcher can notice low-impact issues that would otherwise go unnoticed or even get ignored.

#### 2.4.4 Step #4 - Vulnerability Chains

Following the previous steps, a researcher first has to

1. break the web application down into (security relevant) components,
2. abstract functionalities into a series of the involved components and understand how these components interact with each other,
3. and identify one or more context-dependent weaknesses in one or more components within a series of components in a functionality.

He can then find out how the discovered weaknesses relate to the rest of the functionality and how they can be combined to achieve an exploitable security issue with high impact. In this last step he will likely encounter limitations and challenges that prevent exploitation. Often the reason why these weaknesses are still present in the source code is that some limitations prevent exploitation.

However, now that the researcher understands the limitations and what the solution would be to break the limitations, he can begin auditing the source code with the goal of finding another bug to break it. Since limitations are often times very specific, the likelihood of such a bug still existing in the application is very high.

To pick up the example with the exploitation of the limited LFI again, all a researcher had to do was find a bug that allows a user to upload any file to a certain directory. Since such a bug does not lead to Remote Code Execution directly, we assumed that other researchers would have left it out.

In order to effectively find a bug that breaks his limitations he can cut the research effort down to a few components that could likely contain such an issue. He can then look at the functionalities present within that component and audit them with the same approach that lead to the first bug being discovered.

When we looked at our example of an authenticated Remote Code Execution vulnerability earlier ([Section 2.3](#)), we did exactly that. Recall that we discovered a LFI vulnerability that allowed to include arbitrary files from a certain directory, however that

directory was not accessible and no file that contained PHP code could be placed into it. The only way to break the limitation was to upload or move a user controlled file to this certain directory with another feature. Knowing this, we cut the research effort down to the *plugin update* and *media file* component of WordPress. We then used the same steps used to find the original bug, meaning breaking each functionality down into the series of involved components, in order to find a bug that allowed us to upload a file to a certain directory and break the limitation for exploitation.

We will look at more complex examples in the next section.

## 3. Vulnerability Analysis

In this section we demonstrate our approach to auditing the WordPress core by breaking the source code down into components and building connections between those components. We then analyze the real-world security vulnerabilities that we detected by using our approach. Obviously not all parts of our audit process can be mapped to a specific concept or fixed guideline that guarantees success for other code bases but our presented methodology becomes evident. We further cover technical details about WordPress internals. No prior WordPress knowledge is required. Each breakdown will focus on specific bugs that we explored step-by-step in order to find a connection and build a chain.

We first detected small bugs within the administration area of WordPress that we managed to chain to an authenticated code execution vulnerability. However, the vulnerability required administrator privileges. As a next step, we tried lowering the privilege bar and looked for vulnerabilities requiring less privileges on a target site. Finally, we managed to find an unauthenticated vulnerability that allowed us to take over the session of high privileged users and exploit any of the discovered authenticated Remote Code Execution vulnerabilities.

### 3.1 CVE-2018-12895: Authenticated File Deletion

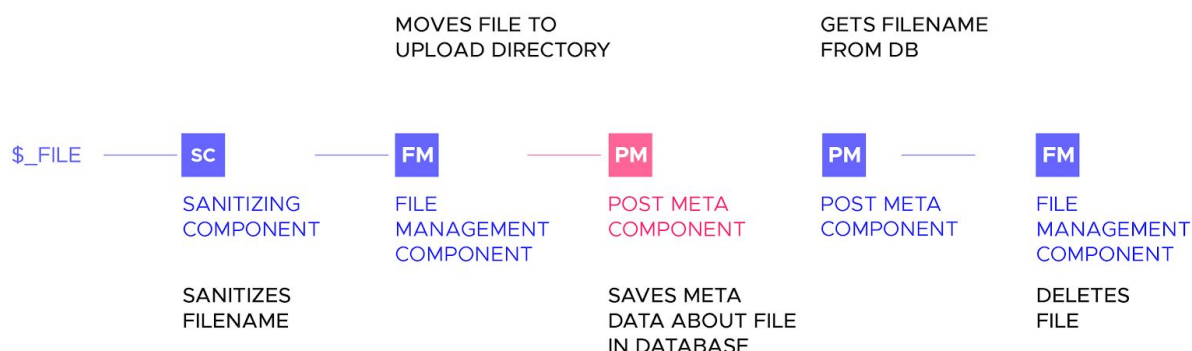
We discovered our first critical vulnerability in the WordPress component that handles uploaded media files. Everything that has to do with file uploads and file management is prone to a series of vulnerability classes: File Writes, File Deletions and File Manipulations to name a few. In this case we found a File Deletion vulnerability that allows to delete arbitrary files from the file system.

The vulnerability can be triggered by an authenticated user with limited *author* privileges. These users can only edit new blog posts but can not use security sensitive features, for example, installing WordPress plugins. By using this vulnerability, an author could gain administrator privileges or execute arbitrary code on the server.

#### 3.1.1 Abstracting the Media File Functionality

When we audited the way media files are handled by WordPress, we abstracted the functionality into a series of involved components.

## FILE UPLOAD AND FILE DELETION PROCESS



We started with the file upload. An uploaded file and its filename is first 1) sanitized by the file sanitization component. It is then 2) moved to the upload directory by the file management component. When this step succeeded, so called 3) *Post Meta* entries are generated that store information about the just uploaded file in the database. This information includes for example the filename of the file in the filesystem and the time of the upload. The Post Meta component is later used again to determine the filename in the filesystem for further file operations in the file management component.

We decided that we would have the highest chance of finding a vulnerability by looking at the so called *Post Meta* component. Since Path Traversal attacks and all kinds of tricks when it comes to File Upload vulnerabilities have long been known and are prevented in WordPress, we assumed that the sanitization and file management component is mostly secure. We decided that since the Post Meta component is unique to WordPress and probably requires deep architectural knowledge about the CMS in order to be exploited, auditing this component would most likely result in at least some limited vulnerability that we could chain with another bug.

### 3.1.2 Background - Understanding Post Meta Entries

In order to understand the logical bug that we discovered within the Post Meta component of WordPress, we will first need a bit of background knowledge on how Post Meta entries are created and used by the WordPress core. When for example an image is uploaded to a WordPress installation, it is first moved to the uploads directory (`wp-content/uploads`). WordPress will also create an internal reference to this image in the database in order to keep track of meta information such as the owner of the image or the time of the upload. This meta information is stored as *Post Meta* entries in the database. Each of these entries are key / value pairs that are assigned to a certain ID.

## Example Post Meta reference to an uploaded image 'evil.jpg'

```

1  MariaDB [wordpress]> SELECT * FROM wp_postmeta WHERE post_ID = 50;
2
3  +-----+-----+-----+
4  | post_id | meta_key          | meta_value          |
5  +-----+-----+-----+
6  | 50      | _wp_attached_file | evil.jpg             |
7  | 50      | _wp_attachment_metadata | a:5:{s:5:"width";i:450 ... |
8  | ...     | ...               | ...                  |
9  +-----+-----+-----+

```

A quick check of the WordPress database table `wp_postmeta` provides an overview of what data is stored here. Apparently, sensitive strings such as file names and serialized data can be found.

### 3.1.3 Insufficient Validation in Post Meta Component

After we abstracted the file upload into a series of components, we learned that when a file is uploaded, its filename is stored in a Post Meta database entry. The issue with the *Post Meta* entries prior to WordPress 4.9.9 and 5.0.1 is that it was possible to modify any entries and set them to arbitrary values. When an image is updated (e.g. its description is changed), the `edit_post()` function is called. This function directly operates on the `$_POST` array.

## Arbitrary Post Meta values can be updated.

```

1  function edit_post( $post_data = null ) {
2
3      if ( empty($postarr) )
4          $postarr = &$_POST;
5      :
6      if ( ! empty( $postarr['meta_input'] ) ) {
7          foreach ( $postarr['meta_input'] as $field => $value ) {
8              update_post_meta( $post_ID, $field, $value );
9          }
10     }

```

As can be seen in the code above, it is possible to inject arbitrary *Post Meta* entries. Since no check is made on which entries are modified, an attacker can, for example, update the `_wp_attached_file` meta entry and set it to **any** value. This does not rename the actual file on the filesystem in any way, it just changes the value in the database. As a next step, we configured our static code analysis tool to automate the process of finding security sensitive functions that operate with Post Meta data.



```

/wp-includes/post.php

function wp_delete_attachment( $post_id, $force_delete = false ) {
    :
    $meta = wp_get_attachment_metadata( $post_id );
    :
    if ( ! empty($meta['thumb']) ) {
        // Don't delete the thumb if another attachment uses it.
        if ( ! $wpdb->get_row( $wpdb->prepare( "SELECT meta_id FROM $wpdb->postmeta
            $thumbfile = str_replace(basename($file), $meta['thumb'], $file);
            /** This filter is documented in wp-includes/functions.php */
            $thumbfile = apply_filters( 'wp_delete_file', $thumbfile );
            @ unlink( path_join($uploadpath['basedir'], $thumbfile) );
        }
    }
    :
}

```

As a result, we detected that the meta data is used when a file is deleted again. Since WordPress trusts the value it receives from the Post Meta component, it will blindly delete the file with the received name. This means an attacker can delete any file by first updating the Post Meta entry of the `_wp_attached_file` entry and setting it to the name of the file that he wants to delete. The attacker can then instruct WordPress to delete any file on the file system ([RIPS vulnerability report](#)).

### 3.1.4 Impact and Limitations

An attacker can use this second-order file deletion vulnerability to delete the main configuration file of WordPress: `wp-config.php`. Once this file is missing, WordPress is tricked to believe that it has not yet been installed. As a result it will display the WordPress installation routine to the website user. The attacker can then re-install WordPress and is able to control the database connection settings which he can point to his own remote database server. This way, he can load a new administrator account into WordPress which enables to use administrator features. When this vulnerability is exploited by a low-privileged author user the privileges are thus escalated. Further, the attacker can control critical WordPress settings in the database that allow to execute arbitrary PHP code.

The limitation of this vulnerability is that although the attacker gains full control over the WordPress installation, he loses access to data stored in the database by deleting the configuration file which contains the database credentials. This way, user accounts, blog posts, pages, files and design configurations are lost and this will immediately alarm administrators about the attack.

We decided to look for a more critical vulnerability that bases on the Post Meta issue which would enable an attacker to install a backdoor without setting of any alarms.

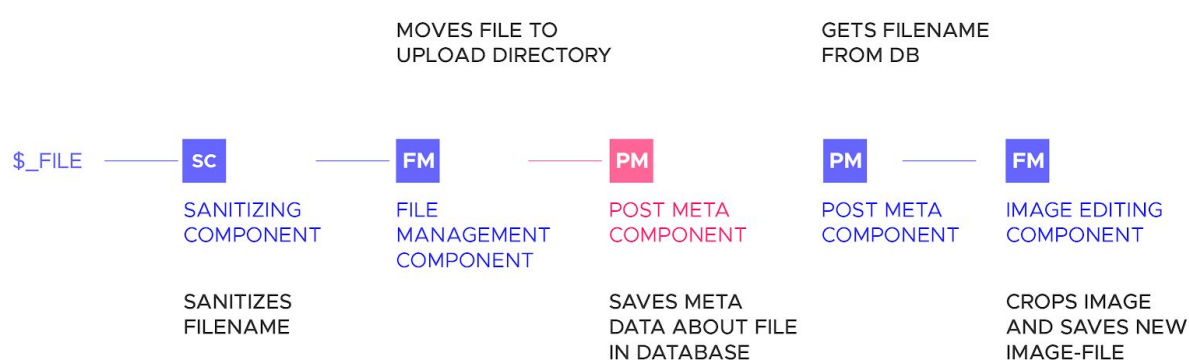
## 3.2 CVE-2019-8943: Authenticated Path Traversal and LFI

After we identified the previously described flaw we investigated further reports of our static analysis tool that based on the usage of Post Meta data. This lead to the discovery of another vulnerability. A Path Traversal vulnerability was detected within the image editing component of WordPress. More precisely, it was hidden in the functionality responsible for cropping an image that can be uploaded by a low-privileged user. In order to exploit this vulnerability, we had to chain it with a Local File Inclusion vulnerability. In the following, we will investigate both vulnerabilities in detail.

### 3.2.1 Abstracting the Image Cropping Functionality

RIPS discovered that the functionality which enables lower privileged users to crop uploaded images also relies on Post Meta entries to actually load the images to crop ([RIPS vulnerability report](#)). Hence, we abstracted the image cropping functionality to gain an overview of what we had to do in order to exploit the Post Meta weakness.

#### IMAGE UPLOAD AND CROPPING FUNCTIONALITY



We documented the following process. When an image is uploaded to a WordPress site, it will be 1) moved to the uploads directory `wp-content/uploads` by the file management component. WordPress then 2) creates an internal reference to the image in the database in order to keep track of meta information such as the owner of the image or the time of the upload (Post Meta information). When a lower privileged user then wants to crop the image, he submits the ID of the image that he wants to edit. WordPress then 3) pulls the corresponding `_wp_attached_file` Post Meta entry from the database (Post Meta component) and passes it to the 4) image editing component. This component finally 5) passes the resulting image to the file management component so the new image is saved back to the filesystem.

Since we could control which file WordPress would pass to the image editing component, we decided to take a closer look at this component. We investigated if we could somehow leverage our detected security flaw that allows us to pass any image to the image editing component.

### 3.2.2 Path Traversal in Image Editing

The Path Traversal vulnerability is in the `wp_crop_image()` function which gets called when a user crops an image. The function takes the ID of an image to crop (`$attachment_id`) and fetches the corresponding `_wp_attached_file` *Post Meta* entry from the database. Note that due to the flaw in `edit_post()`, the return value `$src_file` of `get_post_meta()` can be controlled by an attacker.

Simplified `wp_crop_image()` function. The actual code is located in `wp-admin/includes/image.php`

```
1 function wp_crop_image( $attachment_id, $src_x, ... ) {
2
3     $src_file = $file = get_post_meta( $attachment_id, '_wp_attached_file' );
4     :
```

As a next step, WordPress ensures that the image actually exists and loads it. WordPress has two ways of loading the given image.

- 1) The first is to simply look for the filename provided by the `_wp_attached_file` *Post Meta* entry in the `wp-content/uploads` directory (line 2 of the next code snippet).
- 2) If that method fails, WordPress will try to download the image from its own server as a fallback. To do so it will generate a download URL consisting of the URL of the `wp-content/uploads` directory and the filename stored in the `_wp_attached_file` *Post Meta* entry (line 6).

Let's have a look at a concrete example: If the value stored in the `_wp_attached_file` *Post Meta* entry was `evil.jpg`, then WordPress would first try to check if the file `wp-content/uploads/evil.jpg` exists. If not, then it would try to download the file from the following URL:

<https://targetserver.com/wp-content/uploads/evil.jpg>

The reason why WordPress is trying to download the image instead of looking for it locally is that a plugin could generate the image on the fly when the URL is visited.

In the following the affected code is listed. Note that no sanitization whatsoever is performed when building the filename and / or URL of the image. WordPress simply concatenates the upload directory and the URL with the `$src_file` which can be controlled by an attacker.

```

1      :
2      if ( ! file_exists( "wp-content/uploads/" . $src_file ) ) {
3          // If the file doesn't exist, attempt a URL fopen on the src link.
4          // This can occur with certain file replication plugins.
5          $uploads = wp_get_upload_dir();
6          $src = $uploads['baseurl'] . "/" . $src_file;
7      } else {
8          $src = "wp-content/uploads/" . $src_file;
9      }
10
11     $editor = wp_get_image_editor( $src );
12     :

```

Once WordPress has successfully loaded a valid image via `wp_get_image_editor()`, it will crop the image. The cropped image is then saved back to the filesystem (regardless of whether it was downloaded or not). The resulting filename is going to be the `$src_file` returned by `get_post_meta()` which is under control of the attacker. The only modification made to the resulting file name string is that the *basename* of the file is prepended by the string “cropped-” (line 4 of the next code snippet.) To follow the example of our *evil.jpg*, the resulting filename would be *cropped-evil.jpg*.

WordPress then creates any directories in the resulting path that do not exist yet via `wp_mkdir_p()` (line 6). The image is then finally written to the filesystem using the `save()` method of the image editor object. The `save()` method also performs no Path Traversal checks on the given file name.

```

1      :
2      $src = $editor->crop( $src_x, $src_y, $src_w, $src_h, $dst_w, $dst_h, $src_abs );
3
4      $dst_file = str_replace( basename( $src_file ), 'cropped-' . basename( $src_file ), $src_file );
5
6      wp_mkdir_p( dirname( $dst_file ) );
7
8      $result = $editor->save( $dst_file );

```

So far, we have discussed that it is possible to determine which file gets loaded into the image editor, since no sanitization checks are performed. However, the image editor will throw an exception if the file is not a valid image. The first assumption might be that it is only possible to crop images outside the uploads directory then. However, the circumstance that WordPress tries to download the image if it is not found leads to a Path Traversal vulnerability.

The idea is to set `_wp_attached_file` to `evil.jpg?shell.php`, which would lead to a HTTP request being made to the following URL:

`https://targetserver.com/wp-content/uploads/evil.jpg?shell.php`.

This request would return a valid image file, since everything after the `?` character is ignored in the HTTP context. The resulting filename would be `evil.jpg?cropped-shell.php`.

However, although the `save()` method of the image editor does not check against Path Traversal attacks, it will append the extension of the mime type of the image being loaded to the resulting filename. In this case, the resulting filename would be `evil.jpg?cropped-shell.php.jpg`. This renders the newly created file harmless again. But if we are injecting path traversal sequences it is still possible to plant the resulting image into any directory by using a payload such as `evil.jpg?/../../evil.jpg`.

### 3.2.3 Impact and Limitations

The very first example discussed in this paper (Section 2.3.1) showed how a limited LFI vulnerability exists within the theme component of WordPress. The LFI was limited in the sense that only files from a certain directory could be included and that it was not possible to write files to that directory that an attacker could actually include.

The fact that we can now plant an image file into any directory enables an even lower privileged attacker to exploit the otherwise limited LFI. The attacker can inject PHP executable code within the EXIF meta data section of the image that will be planted into the directory he can include from. By doing so, he can execute arbitrary PHP code on the server. This vulnerability was nominated for a **Pwnie Award** for best server-side bug.

Contrary to the previously analyzed file delete exploitation, this vulnerability has the advantage of being stealthy and does not corrupt any data on the site. However, it still requires the same user privileges as the first vulnerability. Hence our next step was to

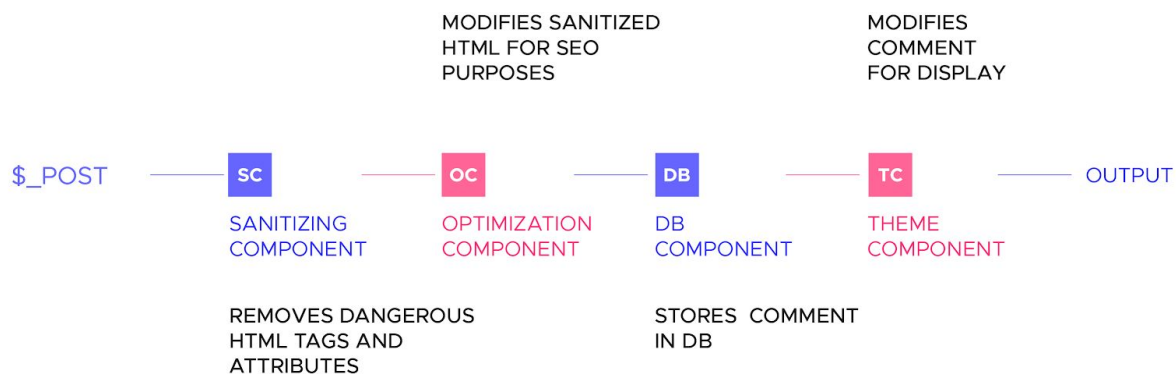


look for a vulnerability that was unauthenticated or would let us exploit our so far discovered vulnerabilities without authentication.

### 3.3 CVE-2019-9787: Unauthenticated CSRF to XSS

Since bugs that require no authentication in web applications have the highest impact, we placed our focus on those functionalities that unauthenticated users have access to. The most obvious feature was the comment functionality of WordPress. Unauthenticated users can add comments to a blog post and even include some very basic HTML tags and attributes within the comment string.

#### COMMENT FUNCTIONALITY



#### 3.3.1 Abstracting the Comment Functionality

We began by breaking the comment functionality down into a series of components. When a comment is created on a WordPress site, the user input is passed through multiple components. It is 1) first sanitized against XSS attacks (XSS sanitization component), 2) then optimized for SEO purposes (optimization component), 3) then stored in the database (database component), and eventually when a user wants to view the comment 4) fetched from the database and modified again (optimization component), before it is finally 5) embedded into the resulting HTML page (theme component).

By abstracting the comment functionality into such a series of components, we were able to quickly determine which component we should spend our time trying to find weaknesses in. Since we assumed that the sanitization component after all these years

of being audited was secure, we assumed that the easiest way to find a vulnerability was to find a bug in one of the components that alter the comment string after it has been sanitized. Since the comment is optimized for SEO purposes in the second step of the comment creation functionality and is modified before being embedded into the HTML markup in the fourth step, we decided to look at both these components. We quickly realized that the fourth component, the one that modifies the comment before it is being embedded into the HTML markup, did not modify the comment string strongly enough for XSS flaws to occur. For this reason we decided to look for a weakness within the SEO optimization component.

### 3.3.2 Sanitization Bypass in SEO Optimization

After WordPress sanitized a new comment it will modify `<a>` tags within the comment string to optimize them for SEO purposes. This is done by parsing the attribute string (e.g. `href="#" title="some link" rel="nofollow"`) of the `<a>` tags into an associative array (line 3004 of the following snippet). Here, the key is the name of an attribute and the value the attribute value.

```
wp-includes/formatting.php

3002     function wp_rel_nofollow_callback( $matches ) {
3003         $text = $matches[1];
3004         $atts = shortcode_parse_atts($matches[1]);
3005         :
```

WordPress then checks if the `rel` attribute is set. If so, it is processed and the `<a>` tag is put back together with the new `rel` attribute.

```
wp-includes/formatting.php

3013     if (!empty($atts['rel'])) {
3014         // the processing of the 'rel' attribute happens here
3015         :
3016         $text = '';
3017         foreach ($atts as $name => $value) {
3018             $text .= $name . '=' . $value . ' ';
3019         }
3020     }
3021     return '<a ' . $text . ' rel="' . $rel . '">';
3022 }
```



The flaw occurs in the lines 3017 and 3018 of the above snippet, where the attribute values are concatenated back together without being escaped.

An attacker can create a comment containing a crafted `<a>` tag and set for example the `title` attribute of the anchor to `title='XSS " onmouseover=alert(1) id=' '`. This attribute is valid HTML and would pass the sanitization step.

However, this only works because the crafted title tag uses *single quotes*. When the attributes are put back together, the value of the title attribute is wrapped around in double quotes (line 3018). This means an attacker can inject additional HTML attributes by injecting an additional double quote that closes the `title` attribute.

Let's have a look at the following example:

`<a title='XSS " onmouseover=evilCode() id=' '>` would turn into

`<a title='XSS " onmouseover=evilCode() id=' '>` after processing.

Since the comment has already been sanitized at this point, the injected `onmouseover` event handler is stored in the database and does not get removed.

This bug allows us to inject arbitrary HTML attributes into the comment string. We only needed to figure out how to trigger the bug.

### 3.3.3 Limitations and Bug Chaining

We had discovered a weakness in the SEO optimization component of WordPress that was a part of the comment creation functionality. However, we realized that the parsing error would only trigger if the `rel` attribute could be set in the HTML markup of the comment. This was an issue, since the sanitization component of WordPress does not actually allow the `rel` attribute to be set and removes it from the user supplied comment string, meaning the bug discovered in the SEO optimization component can never be abused. This was probably the reason it had never been reported in the first place.

The only way to exploit this weakness was to find another bug that would allow us to set the `rel` tag. Our assumption was that we probably would need to find a bug in the same functionality, which limited the components that we could audit for bugs to only a few.

We decided that the highest chance of success was to find a weakness in the rather complex XSS sanitization component of WordPress. Since we did not need a full bypass for the XSS filters of WordPress, just one that allowed us to inject a usually harmless `rel` attribute we assumed that it was worth a try. Additionally, we knew exactly what we were looking for. Researchers who did not specifically look for a bug that allows to inject a `rel` attribute would probably not have noticed such a bug in the past.

### 3.3.4 CSRF Vulnerability in Comments

When looking for a bypass in the comment sanitization process that allows to set the `rel` attribute, we analyzed when and how the XSS filters for specific attributes are triggered. We discovered that for specific comments, no filter for the `rel` attribute was invoked. We could create a comment without invoking a filter that would remove the `rel` attribute from a comment string via a CSRF vulnerability that abuses the trackback feature of WordPress.

Moreover, WordPress performs no CSRF validation when a user posts a new comment. This is because some WordPress features such as trackbacks and pingbacks would break if there was any input validation. This means an attacker can create comments in the name of administrative users of a WordPress blog via CSRF attacks. This can become a security issue since administrators of a WordPress blog are allowed to use arbitrary HTML tags in comments, even `<script>` tags. In theory, an attacker could simply abuse the CSRF vulnerability to create a comment containing malicious JavaScript code.

WordPress tries to solve this problem by generating an extra nonce for administrators in the comment form. When the administrator submits a comment and supplies a valid nonce, the comment is created without any sanitization. If the nonce is invalid, the comment is still created but is sanitized.

The following code snippet shows how this is handled in the WordPress core:

```
/wp-includes/comment.php (Simplified code)

3239  :
3240  if ( current_user_can( 'unfiltered_html' ) ) {
3241      if ( ! wp_verify_nonce( $_POST['wp_unfiltered_html_comment'], 'unfiltered-html-comment' ) ) {
3242          $_POST['comment'] = wp_filter_post_kses($_POST['comment']);
3243      }
3244  } else {
3245      $_POST['comment'] = wp_filter_kses($_POST['comment']);
3246  }
3247  :
```

The fact that no CSRF protection is implemented for the comment form has been known since 2009. However, we discovered a logical flaw in the sanitization process for administrators. As shown in the above code snippet, the comment is always sanitized with `wp_filter_kses()`, unless the user creating the comment is an administrator with the `unfiltered_html` capability. If that is the case *and* no valid nonce is supplied, the

comment is sanitized with `wp_filter_post_kses()` instead (line 3242 of the above code snippet).

The difference between `wp_filter_post_kses()` and `wp_filter_kses()` lies in their strictness. Both functions take in the unsanitized comment and leave only a selected list of HTML tags and attributes in the string. Usually, comments are sanitized with `wp_filter_kses()` which only allows very basic HTML tags and attributes, such as the `<a>` tag in combination with the href attribute.

This allows an attacker to create comments that can contain much more HTML tags and attributes than comments should usually be allowed to contain. Although `wp_filter_post_kses()` is much more permissive, it still removes any HTML tags and attributes that could lead to Cross-Site-Scripting vulnerabilities. However, the important difference is that `wp_filter_post_kses()` allows `rel` tags to be set. As a result, we can inject a `rel` attribute via a CSRF flaw.

### 3.3.5 Impact and Limitations

By combining the CSRF flaw that allows an attacker to set the `rel` attribute in a comment string with the parsing flaw in the SEO optimization component that leads to an arbitrary HTML attribute injection, an attacker can create and store a comment with a XSS payload (persistent Cross-Site Scripting). The vulnerability requires that comments are enabled on a target WordPress sites but these are enabled by default.

In order to escalate this to Remote Code Execution impact, he can abuse another flaw. The page that displays the newly created comment is not protected by the `X-Frame-Options` header. This means when an attacker can trick an administrator into visiting a website that triggers the CSRF exploit, he can create a hidden `iframe` in the background of the page to display the comment and immediately execute the JavaScript code contained in it. The attacker can now execute arbitrary JavaScript code on the target site with the session of the admin user.

## 4. Exploitation Chain

So far we have analyzed four vulnerabilities that can lead to Remote Code Execution in WordPress with default settings. We reported all vulnerabilities to the WordPress security team responsibly and ensured a patch is available before disclosing the technical details. A sophisticated attacker who would not have reported these issues could have chained and use these bugs to target and take over any high value target running WordPress though. In this section we will explore the steps that a motivated attacker would take to attack a specific WordPress instance.

### 4.1 Step #1 - Plugin Vulnerabilities

By far the easiest way to take over a WordPress site would be to create a list of plugins which the targeted site uses and then to look for critical vulnerabilities in each of these plugins. There are over 50.000 free plugins WordPress administrators can install, many of which are vulnerable to some sort of attack. For example, during our [Advent Calendar 2018](#), we demonstrated how even the most popular plugins with over 5 million active installations contained severe issues.

This clearly demonstrated the danger that third-party plugins can pose to WordPress sites. Considering that tools such as *wpscan* can effectively enumerate a list of all plugins a target site uses, the first step of an attacker is to simply go through this list and attempt to find such an unauthenticated vulnerability. In the context of this paper we assume that all plugins on a targeted site are secure or the vulnerabilities require some sort of user authentication.

### 4.2 Step #2 - Attacking WordPress Core via CSRF

When no vulnerable plugin is found an attacker would audit the WordPress core as detailed in this paper. For example, he could exploit the CSRF vulnerability that leads to persistent XSS (CVE-2019-9787) as discussed in [Section 3.1](#). All the attacker needs to do is to leave a harmless comment below a blog post that contains a link to an attacker controlled website.

*I liked your blog post and copied your content here: [www.my-similar-blog.com](#)*

By default, all comments have to be moderated by an administrator. Since the administrator needs to be authenticated for moderation, the likelihood of the administrator being authenticated when he clicks on the malicious website link in the moderated comment is very high. Once the administrator opens this website, the attacker can then leverage the CSRF vulnerability and add a new comment to the admin panel that triggers a Cross-Site Scripting payload. This JavaScript payload rendered in the browser of the authenticated admin could, for example, trigger the authenticated vulnerabilities described in the following to execute arbitrary code.

### 4.3 Step #3 - Exploiting Authenticated Vulnerabilities

Once an attacker succeeds in compromising an account on a targeted website, his next steps depend on the privileges of the hijacked account. Per default there are five user roles in WordPress: The *subscriber*, *contributor*, *author*, *editor* and *administrator* role. Depending on their role, users can perform different actions.

- *Subscribers* can only read content.
- *Contributors* can create new blog posts, but can't publish them by themselves. A user with a higher user role must do it for them.
- *Authors* can publish their own posts and can upload media files.
- *Editors* can do the same as Authors, but can use arbitrary HTML tags, even `<script>` tags within blog posts and comments.
- *Administrators* can do all of the above and can install new plugins and even edit `.php` files of themes and plugins directly. This feature can be disabled for hardening purposes.

The *subscriber* role is not likely to be used on WordPress sites as it is only a guest account. The *contributor* role can't upload files which would prevent an attacker who compromised an account with this user role to exploit any of the discussed RCE vulnerabilities since these depend on access to the file management system. However, we published details about a 5th vulnerability in WordPress. This vulnerability is a privilege escalation (CVE-2018-20152) that allows contributors to still execute arbitrary code on most WordPress sites ([technical details](#)).

If the attacker can compromise an account with *author* or *editor* privileges, he can exploit either one of the two previously described Remote Code Execution vulnerabilities that depend on file upload access.

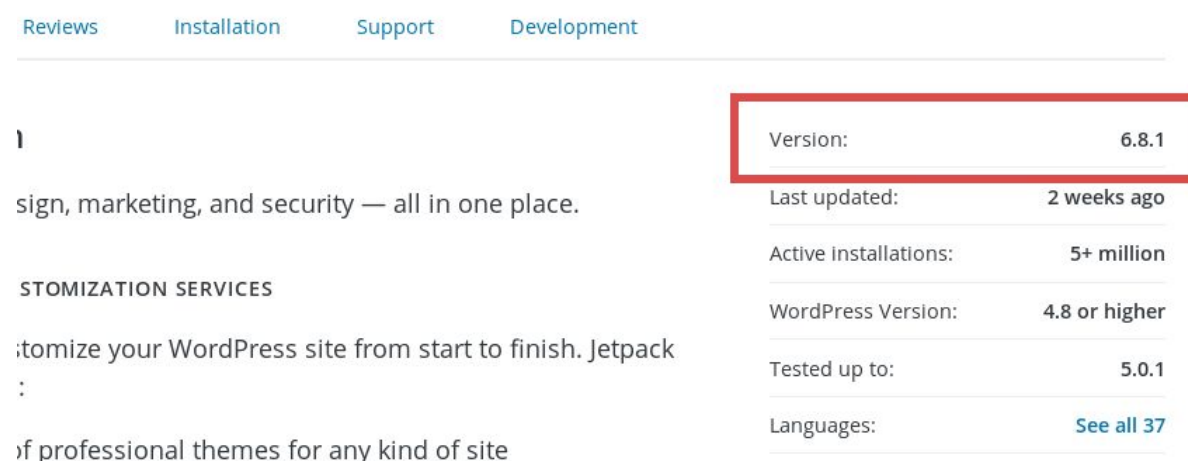
Finally, if the attacker can compromise an *administrator* account directly, he can abuse a WordPress feature that enables administrators to directly edit the contents of `.php`

files of installed plugins. This feature can be disabled and is expected to be a high value target. The attacker can exploit the Remote Code Execution vulnerability from [Section 2](#) or any of the other Remote Code Execution vulnerabilities in order to still gain full access to the underlying web server.

However, although we layed out how an attacker can escalate from almost any user role to Code Execution he needs to compromise an account on the target site first. Although previous attacks of sophisticated groups have shown their effectiveness in compromising accounts, it is still not always possible to do so. WordPress is a platform that does not need a lot of administrative accounts. Typically, companies will create only one or two administrator accounts and the employees managing those accounts can be security-aware IT professionals that won't get easily phished. What now?

## 4.4 Bonus: Wormable Stored XSS on WordPress.org

The WordPress.org website holds the plugin and theme repositories used by all WordPress sites. Furthermore, it manages the accounts developers use to edit the code of their themes and plugins. In May 2019, we have notified the WordPress security team about a critical Stored XSS vulnerability on this website found by our static code analysis solution. The vulnerability occurred when the plugin version numbers from the repository are displayed.



The screenshot shows the WordPress.org plugin page for Jetpack. At the top, there are tabs for Reviews, Installation, Support, and Development. Below the tabs, the plugin name 'Jetpack' is displayed. To the right of the plugin name, the version number '6.8.1' is highlighted in a red box. Below the version number, there is a table with the following information:

Version:	6.8.1
Last updated:	2 weeks ago
Active Installations:	5+ million
WordPress Version:	4.8 or higher
Tested up to:	5.0.1
Languages:	<a href="#">See all 37</a>

The WordPress.org website is built using the WordPress CMS. The plugins as presented in the plugin repository are merely posts of a dedicated post type that are displayed with a special template. The listing below shows the code responsible for displaying the version number.



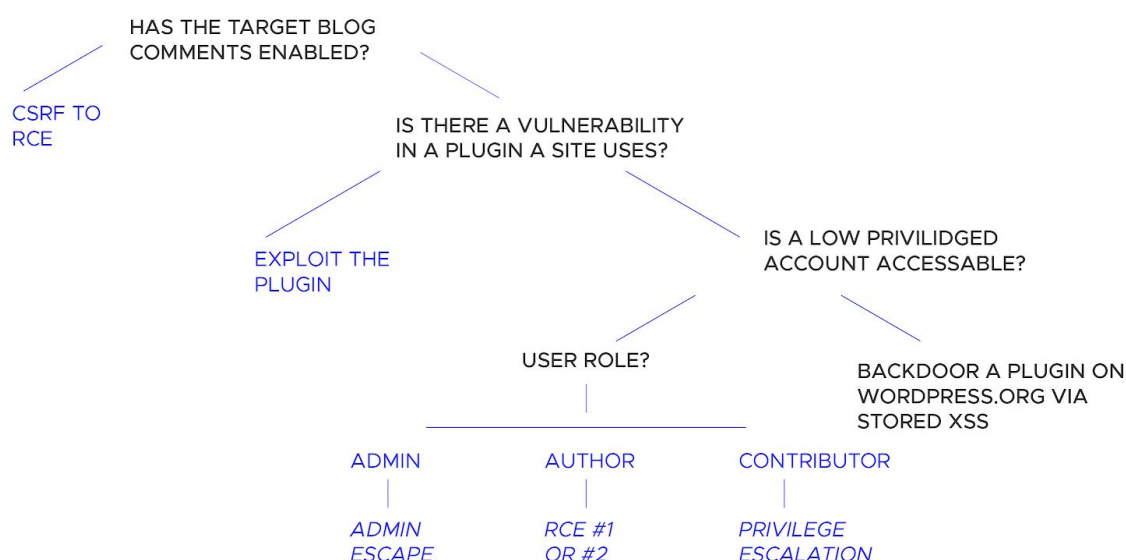
```
wordpress.org/public_html/wp-content/plugins/plugin-directory/widgets/class-
meta.php
```

```
43 <li><?php printf( __( 'Version: %s', 'wporg-plugins' ),
44 ' <strong>' . get_post_meta( $post->ID, 'version', true ) . '</strong>' ); ?></li>
```

Here, `get_post_meta()` is the same function that interacts with the Post Meta component of WordPress which as we have shown earlier can't be trusted. Any user could have created a new plugin and injected arbitrary JavaScript code into the plugin version of his plugin. This JavaScript code would then be executed in the browser of other plugin developers browsing the plugin repository. This would allow to add the attacker as a *commiter* to other plugins in order to add a backdoor and to infect the version number with the JavaScript payload as well, such that the payload would spread like a worm amongst plugins.

## 4.5 Putting it all together

### EXPLOIT CHAIN



We now have demonstrated the ability for an attacker to 1) exploit a WordPress site without any credentials if it has comments enabled, how he can 2) escalate from almost any user role despite all hardening mechanisms being enabled to Remote Code Execution and 3) how an attacker can as a last resort, if all previous vectors fail, can hijack the accounts of plugin maintainers of plugins the site uses and insert backdoors into the plugins.



## Summary

WordPress is the most popular application in the Web that runs on 34% of all websites. Its code base is not more or less secure than any other web application. Contrarily, it is arguably one of the most reviewed code bases by security experts, bug bounty hunters, and community developers that search for security issues with different motives every day.

Our research goal was to find out how difficult it is for malicious attackers to uncover critical security vulnerabilities in a well-reviewed code base. As a result, we uncovered four vulnerabilities with Remote Code Execution impact and documented our methodology behind finding and exploiting them. We also discovered a Privilege Escalation vulnerability that allows low privileged users to execute arbitrary PHP code on many WordPress sites using a similar approach. All security issues were reported to the vendor responsibly.

Although we could automate the task of detecting the core issues with the help of static code analysis, finding a way for exploitation involved manual auditing to understand the code functionalities. We introduced our audit methodology and believe that it can be applied to all kinds of code assessments. In fact, we recently audited Adobe's ecommerce solution Magento with 2.2 million lines of code using the same approach illustrated in this paper. As a result, we found and reported 6 critical security vulnerabilities that lead to remote command execution and that are currently fixed by the vendor.

We hope that this whitepaper also helps developers to look at their own source code from an attackers perspective and to raise security awareness. As demonstrated, even low-severe looking weaknesses and harmless features can be combined to critical-impact vulnerabilities that should always be addressed.

## References

<https://blog.ripstech.com/2018/wordpress-file-delete-to-code-execution/>

<https://blog.ripstech.com/2018/wordpress-configuration-cheat-sheet/>

<https://blog.ripstech.com/2018/php-security-advent-calendar/>

<https://blog.ripstech.com/2018/wordpress-post-type-privilege-escalation/>

<https://blog.ripstech.com/2018/wordpress-org-stored-xss/>

<https://blog.ripstech.com/2019/wordpress-security-month/>

<https://blog.ripstech.com/2019/wordpress-image-remote-code-execution/>

<https://blog.ripstech.com/2019/wordpress-csrf-to-rce/>