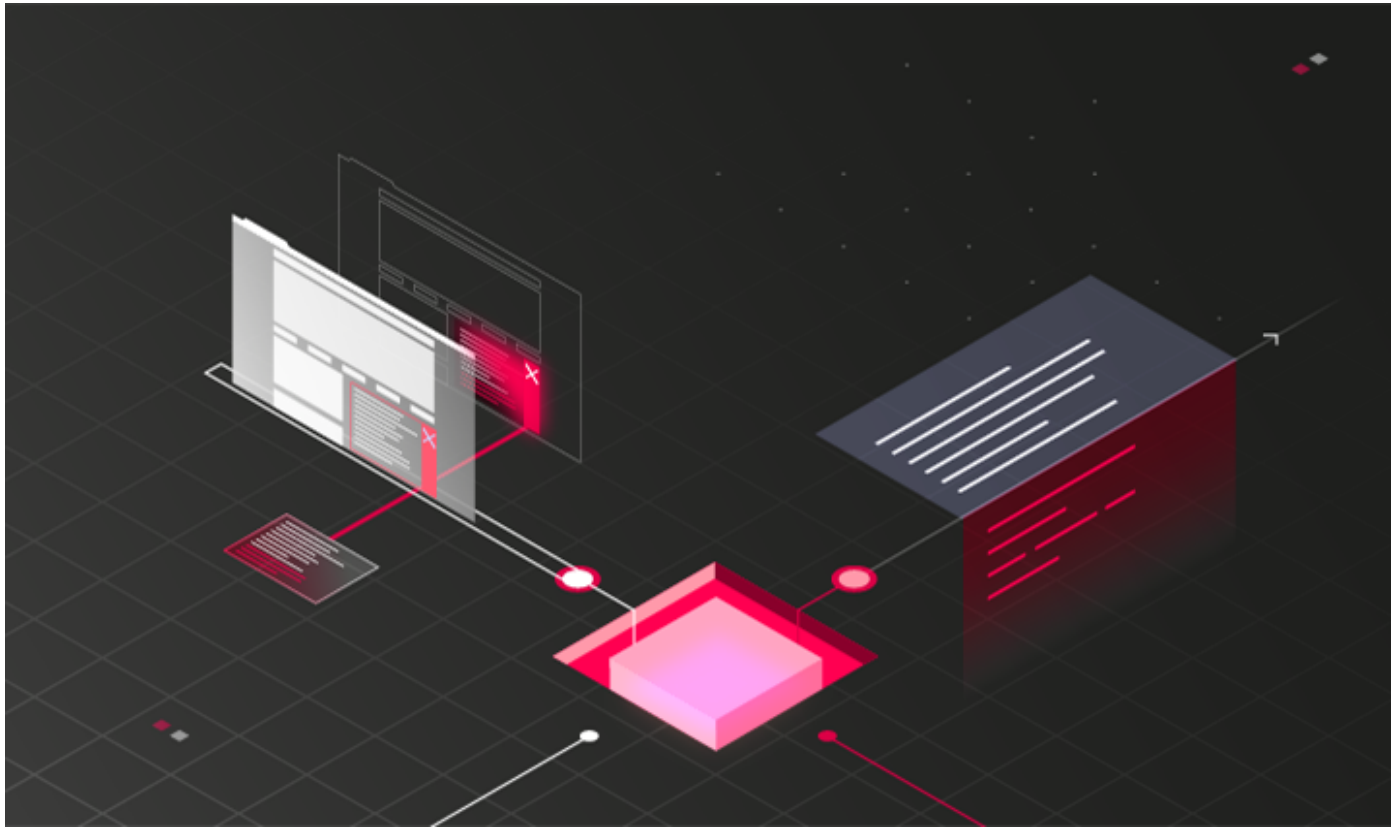
WordPress 5.1 CSRF to Remote Code Execution

9 min read — 13 Mar 2019 by Simon Scannell

Last month we released an authenticated remote code execution (RCE) vulnerability in WordPress 5.0. This blog post reveals another critical exploit chain for WordPress 5.1 that enables an **unauthenticated** attacker to gain remote code execution on any WordPress installation prior to version **5.1.1** (CVE-2019-9787).



PHP WORDPRESS REMOTE CODE EXECUTION CROSS SITE REQUEST FORGERY CROSS SITE SCRIPTING VULNERABILITY CMS TOP5

Impact

An attacker can take over any WordPress site that has comments enabled by tricking an administrator of a target blog to visit a website set up by the attacker. As soon as the victim administrator visits the malicious website, a cross-site request forgery (CSRF) exploit is run against the target WordPress blog in the background, without the victim noticing. The CSRF

exploit abuses multiple logic flaws and sanitization errors that when combined lead to Remote Code Execution and a full site takeover.

The vulnerabilities exist in WordPress versions prior to **5.1.1** and is exploitable with default settings.

WordPress is used by over 33% of all websites on the internet, according to its own download page. Considering that comments are a core feature of blogs and are enabled by default, the vulnerability affected millions of sites.

Technical Analysis

0:00

CSRF in comment form leads to HTML injection

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 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'], 'unfilt
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**[1].

However, we discovered a logic flaw in the sanitization process for administrators. As you can see 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. However, 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.

Escalating the additional HTML injection to a Stored XSS

The fact that we can inject additional HTML tags and attributes still leads to a stored XSS vulnerability in the WordPress core. This is because some attributes that usually can't be set in comments are parsed and manipulated in a faulty way that leads to an arbitrary attribute injection.

After WordPress is done sanitizing the 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), where 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. This attribute can only be set if the comment is filtered via `wp_filter_post_kses()`. If it is, it processes the `rel` attribute and then puts the `<a>` tag back together.

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.

For 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 allows attackers to inject a stored XSS payload into the target website by chaining this sanitization flaw with the CSRF vulnerability.

Directly executing the XSS via an iframe

The next step for an attacker to gain Remote Code Execution after creating the malicious comment is to get the injected JavaScript executed by the administrator. The comment is displayed in the frontend of the targeted WordPress blog. The frontend is not protected by the `X-Frame-Options` header by WordPress itself. This means the comment can be displayed in a hidden `<iframe>` on the website of the attacker. Since the injected attribute is an

`onmouseover` event handler, the attacker can make the iframe follow the mouse of the victim to instantly trigger the XSS payload.

This allows an attacker to execute arbitrary JavaScript code with the session of the administrator who triggered the CSRF vulnerability on the target website. All of the JavaScript execution happens in the background without the victim administrator noticing.

Escalating the JavaScript execution to Remote Code Execution

Now that is possible to execute arbitrary JavaScript code with the session of the administrator, Remote Code Execution can be achieved easily. By default, WordPress allows administrators of a blog to directly edit the `.php` files of themes and plugins from within the admin dashboard. By simply inserting a PHP backdoor, the attacker can gain arbitrary PHP code execution on the remote server.

Patch

By default, WordPress automatically installs security updates and you should already run the latest version 5.1.1. In case you or your hoster disabled the auto-update functionality for some reason, you can also disable comments until the security patch is installed. Most importantly, make sure to logout of your administrator session before visiting other websites.

Timeline

Date	What
2018/10/24	Reported that it is possible to inject more HTML tags than should be allowed via CSRF to WordPress.
2018/10/25	WordPress triages the report on Hackerone.
2019/02/05	WordPress proposes a patch, we provide feedback.
2019/03/01	Informed WordPress that we managed to escalate the additional HTML injection to a Stored XSS vulnerability.
2019/03/01	WordPress informs us that a member of the WordPress security team already found the issue and a patch is ready.
2019/03/13	WordPress 5.1.1 Security and Maintenance Release

Summary

This blog detailed an exploit chain that starts with a CSRF vulnerability. The chain allows for any WordPress site with default settings to be taken over by an attacker, simply by luring an administrator of that website onto a malicious website. The victim administrator does not

notice anything on the website of the attacker and does not have to engange in any other form of interaction, other than visiting the website set up by the attacker.

We would like to thank the volunteers of the WordPress security team which have been very friendly and acted professionally when working with us on this issue.

---

1. https://core.trac.wordpress.org/ticket/10931 [return]

Related Posts

- WordPress 5.0.0 Remote Code Execution
- Wormable Stored XSS on WordPress.org
- WordPress Privilege Escalation through Post Types
- WordPress Design Flaw Leads to WooCommerce RCE
- WARNING: WordPress File Delete to Code Execution

Simon Scannell

Security Researcher

Simon is a self taught security researcher at RIPS Technologies and is passionate about web application security and coming up with new ways to find and exploit vulnerabilities. He currently focuses on the analysis of popular content management systems and their security architecture.