Too Big to Fail
Breaking WordPress Core

Netanel Rubin
What is WordPress?

- A CMS/blogging platform
- The **most popular** in the world
  - ~60% market share
- One of the **most secure** Web Apps in the world
  - No SQLI/LFI/RCE for the past **4 years**
    - Plenty of plugin vulnerabilities, though
What Did We Find In WordPress?

- A **Privilege Escalation** attack
  - Any subscriber can become an author
- An **SQL Injection**
  - Compromising the database
- A **Persistent XSS**
  - Executing arbitrary JS on all privileged users
- Basically, complete **compromise** of both the **server** and the **clients**

*For the full, detailed, white paper, please see:*
How WordPress Works

- Any **user** can access the admin panel
  - But using a **capabilities** system, not every admin page

<table>
<thead>
<tr>
<th></th>
<th>Subscriber</th>
<th>Administrator</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_page</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>read_post</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>edit_posts</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>install_themes</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>edit_plugins</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>
• We assume we are subscribers at the site
  • The lowest role possible
  • We can only read public posts and pages
    • Can’t even comment

• We need more capabilities!
Exploiting The Un-Exploitable

- How does WordPress check our capabilities?

```php
if (current_user_can('edit_posts')) // Can we edit posts?
```

```php
if (current_user_can('edit_post', 1)) // Can we edit post ID 1?
```

- Each role has specific permissions
- `current_user_can()` maps a requested capability into the appropriate role permission
  - And returns true/false based on our permissions

- But how?
Exploiting The Un-Exploitable

- Let’s look on the “edit_post” capability check
  - Responsible for checking if the user can edit a specific post

```php
case 'edit_post': // Edit Post/Page
case 'edit_page':
    $post = get_post( $args[0] ); // Get the post

    // If the post doesn't exist, no capabilities needed
    if ( empty( $post ) )
        break;
```

- If the post ID doesn’t exist => no permissions needed!
Exploiting The Un-Exploitable

- We can access code that checks capabilities for a post ID, but doesn’t check it exists
- But we want to be able to edit a post that does exist!
- How can we do that?
The Need For Speed

- Using the capabilities bug, we could access the post editing code

```php
function edit_post( $post_data = null ) {
    if ( empty($post_data) )
        $post_data = &$_POST;

    $post_ID = (int) $post_data['post_ID']; // Get the post ID
    $post = get_post( $post_ID ); // Get the post
    ...
    $success = wp_update_post( $post_data ); // Update the post in the DB
}
```
function wp_update_post($postarr = array(), $wp_error = false) {
    // First, get all of the original fields.
    $post = get_post($postarr['ID'], ARRAY_A);
    
    if ( is_null($post) ) {
        if ( $wp_error )
            return new WP_Error('invalid_post', 'Invalid post');
        return 0;
    }
    ...
}
The Need For Speed

• We’re stuck :(  

• We need an **INVALID** post ID for `'edit_post()'`  
• But a **VALID** post ID for `'wp_update_post()'`

• Wait…

• What if we could create the post between these function calls?
The Need For Speed

- WordPress doesn’t allow subscribers to **create a post**

- In fact, when we try to do so it **blocks** our access by calling `wp_dashboard_quick_press()`:

```
switch($action) {
    case 'post-quickdraft-save':
        if ( ! current_user_can( 'edit_posts' ) )
            $error_msg = "You don’t have access to add new posts.";

        // If there’s an error (no token, no capabilities)
        if ( $error_msg )
            return wp_dashboard_quick_press( $error_msg );
```
The Need For Speed

• But what does ‘wp_dashboard_quick_press()’ do?

• It creates a post.

```php
function wp_dashboard_quick_press( $error_msg = false ) {
    ...
    $post = get_default_post_to_edit( 'post' , true);
    ...
}
```
The Need For Speed

- Now we can create a post
  - But how do we create it exactly at the right time?
- We will delay the script
  - By executing a lot of DB queries

```php
foreach ((array) $post_data['tax_input'] as $taxonomy => $terms) {
    // Make sure the terms variable is an array
    $terms = explode(',', trim($terms, '\n\t\r\0\x0B,'));

    // Fetch the required terms from the DB
    foreach ($terms as $term) {
        $_term = get_terms($term);
    }
}
```
The Need For Speed

• Using the race condition, we were able to **edit a real post**
  1. We send an “edit post” request for an **invalid post ID**
     • With our large taxonomy array
  2. While the script executes, we send a “create post” request, which **creates that post**
  3. When the taxonomy queries are done, the post already exists in the DB
     • Allowing us to **update it** as we wish
I’m an Author!

- We can now edit the post data
  - We change its status to “trash”
I’m an Author!

• What now?
  • Editing posts *doesn’t* compromise anyone

• We need to leverage this new attack surface
All Your Shortcodes Are Belong To Us

• WordPress validates the post content for XSS
  1. It uses KSES for HTML validating
  2. Then, it expands shortcodes and validates them too
  3. The resulting HTML is displayed as is

• Wait…
  • WordPress FIRST validates the HTML
  • THEN it expands shortcodes, which adds more HTML

• Let’s dig into that behavior
All Your Shortcodes Are Belong To Us

- Regular link HTML:
  
  ```html
  <a href="http://4chan.org/b/" title='OK'></a>
  ```

- Regular shortcode:
  
  ```html
  [gallery ids="729,732,731,720" order='DESC']
  ```

- **KSES only** validates the link HTML
- **Shortcodes only** validate the shortcode HTML
- **2 different mechanisms**
  - Validating the same thing
  - In different context!
All Your Shortcodes Are Belong To Us

- Let’s combine the two mechanisms!

- This shortcode text:

```
[caption width='1' caption='TEST']
```

- Will result in this HTML:

```
<figcaption class="wp-caption-text">TEST</figcaption>
```
All Your Shortcodes Are Belong To Us

- Let’s combine the two mechanisms!

- This shortcode text:

\[
\text{[caption width='1' caption='<a href=""'>}
\]

- Will result in this HTML:

\[
\text{<figcaption class="wp-caption-text"} <a href=""</a> \\
\text{</figcaption>}
\]
All Your Shortcodes Are Belong To Us

- Let’s combine the two mechanisms!

- This shortcode + HTML text:

```
[caption width='1' caption='<a href=""'>]\</a><a href="" onClick='alert(1)'></a>
```

- Will result in this HTML:

```
<figcaption class="wp-caption-text"><a href=""</a><a href="" onClick='alert(1)'></a></figcaption></a><a href="" onClick='alert(1)'></a>
```

- Bingo!
- Persistent XSS on the site’s front page
Now we can **compromise** the clients
But we want to **break** the **server** too

**We need a server side vulnerability!**
Delete Me If You Can

• We can **add** comments to our post
  • We can **edit** them
  • We can **delete** them
  • We can **restore** them
  • We can **approve** them

• Approving a comment means **changing** the “**comment_approve**” DB field
  • We can **set** that field to **whatever we want**
Delete Me If You Can

- When we delete a post, its comments are deleted too
  - Actually, their “comment_approve” value is changed

- When we restore a post, its comments are restored too
  - Actually, their “comment_approve” value is restored

- But how does WordPress know which values to restore?
Delete Me If You Can

- When we delete a post, its comments approve value is stored in the post metadata.
- When we restore a post, its comments approve value is assigned using that metadata.
function wp_untrash_post_comments( $post_id ) {
    // Get the previous comments status from the post meta
    $statuses = get_post_meta($post_id, 'trash_meta_comments_status');

    // Set the comments status to what it was prior to the trashing
    foreach ( $statuses as $status => $comments ) {
        // Update the comments status
        $wpdb->query( "UPDATE $wpdb->comments SET comment_approved = '$status' WHERE comment_ID IN (" . $comments . ")" );
    }
}

• Than, this code happens:

• We control the status
• We control the SQL ;)

Delete Me If You Can
PWNGE Sum Up

• We used a race condition to cause a privilege escalation
• We used 2 faulty HTML validators to cause an XSS
• We used a broken restore mechanism to cause an SQL Injection
• So long WordPress security
  • You will be missed <3
Who uses WordPress?

- Sony / Best Buy / Ford
- BBC / The NYT / The Wall Street Journal
- US Army / NASA / Sweden

- 70 million more websites!
WordPress’s Significance

- Huge client reach
  - 30% more visitors than Amazon!
- It stores sensitive data
  - Passwords, emails, address
  - Some plugins support credit card storage!

- All in all, WordPress handles 126 Million unique visitors per month
Have you reported it?

- Yes.
  - We reported to WordPress’s security contact
    - Provided a full technical description including suggested fixes
  - The vulnerabilities were assigned 4 CVEs
    - CVE-2015-5623 – Subscriber Privilege Escalation
    - CVE-2015-2213 – SQL Injection
    - CVE-2015-5714 – Shortcode XSS
    - CVE-2015-5715 – Post Publish Privilege Escalation
Have they fixed it?

- Yes.
- WordPress fixed the issues using 3 patches
  - Approximately 2 months from disclosure to final fix

Initial Report  Patch 4.2.3  Patch 4.2.4  Patch 4.3.1 + Final Publication
Jul 14  Jul 23  Aug 4  Sep 15

63 days
Summary

- Even if it’s responsible for 126M users a month
- Even if governments use it
- Even if it’s **THE** Web Platform

- **It seems no code is secure**
Thanks!