



Common WordPress Vulnerabilities and Prevention Through Secure Coding Best Practices

Chloe Chamberland

Wordfence Threat Analyst

Masters of Science in Cybersecurity and Information Assurance

OSCP, OSWE, OSWP, Security+, CySA+, PenTest+, CASP+, C|EH, E|CSA, CHFI, eWPT, SSCP,
Associate of (ISC)2, AWS CCP, AWS SSA, AWS Security Specialty

Publication Date: July 13, 2021

© 2021 WORDFENCE ALL RIGHTS RESERVED

Table of Contents

I. Introduction	5
II. Missing Capability Checks	6
Introducing a Weakness	6
Identifying Missing Capability Checks	6
Adding Capability Checks	9
Additional Note	10
Example from WP Database Reset Plugin <= 3.1 - Unauthenticated Database Reset Vulnerability (CVE-2020-7048)	11
III. Missing Cross-Site Request Forgery Protection.	13
Opening a Door to Forged Requests	13
Identifying a Lack of Cross-Site Request Forgery Protection	13
Adding CSRF Protection	14
Additional Note	14
Example from Real-Time Find and Replace plugin <= 3.9 - Cross-Site Request Forgery to Cross-Site Scripting (CVE-2020-13641)	16
IV. Missing File Upload Validation	18
An Open Door to Remote Code Execution	18
Finding Unrestricted File Uploads	19
Properly Validating File Uploads	20
Example from Quiz and Survey Master Plugin < =7.0.0 - Unauthenticated Arbitrary File Upload (CVE-2020-35949)	22
V. Deserialization of User-Supplied Input	24
The Magic of PHP Object Injection Vulnerabilities	24
Discovering Deserialization Weaknesses	25
Preventing Deserialization Vulnerabilities	26
Example from Facebook for WordPress Plugin <=2.2.2 - Unauthenticated PHP Object Injection (CVE-2021-24217)	27
VI. Lack of Sanitization and Escaping on User-Supplied Inputs	29
Cross-Site Scripting Vulnerabilities Open Doors to Remote Code Execution	29
Discovering Cross-Site Scripting Vulnerabilities	30
The unfiltered_html capability	32
Sanitizing Inputs and Escaping Outputs to Prevent Cross-Site Scripting Vulnerabilities and Properly Format Data	32

Example from 301 Redirects – Easy Redirect Manager Plugin <= 2.4.0 - Stored Cross-Site Scripting Vulnerability (CVE-2019-19915)	35
VII. Unprepared SQL Queries	38
SQL Injection Can Lead to Sensitive Information Theft	39
Where are SQL Injection vulnerabilities found?	39
Securely Performing SQL Queries	41
Example from Tutor LMS: Authenticated <= 1.8.2 - SQL Injection (CVE-2021-24183)	42
VIII. Usage of Certain PHP Functions with User-Supplied Input	44
The Impact of Code & Command Injection	44
Finding Remote Command/Code Injection Weaknesses	45
How can this be corrected?	45
Example from Social Warfare Plugin <= 3.5.2 Unauthenticated Remote Command Execution	47
IX. Sensitive Information Disclosure	48
Uncovering Sensitive Information Disclosure Vulnerabilities	48
Preventing Sensitive Information Disclosure	49
Example from WPCentral Plugin <= 1.5.0 - Connection Key Disclosure (CVE-2020-9043)	50
X. File Access/Usage Weaknesses	52
Accessing, Including, and Modifying Files Can Do a Lot of Harm	52
Discovering File Access/Usage Weaknesses	53
Securely Performing File Operations	54
Example from Quiz and Survey Master Plugin <= 7.0.0 - Arbitrary File Deletion(CVE-2020-35951)	55
XI. Additional Security Flaws	57
Arbitrary Options and user_meta Update Vulnerabilities	57
Open Redirection Vulnerabilities	58
Conclusion	59

I. Introduction

WordPress plugins and themes are developed by many different third-party developers with diverse coding techniques and backgrounds. Any developer can publish a plugin or theme to the WordPress repository that any user can install on their WordPress site. This diversity is part of what makes WordPress so attractive and easy to use. However, this same diversity increases the probability of introducing vulnerabilities on WordPress sites. WordPress currently powers over 42% of the internet at the time of publication and as such, it is important that we maintain a secure ecosystem.

WordPress users often have no choice but to trust that plugin and theme developers have used secure coding practices and actively maintain their plugins. Unfortunately, the Wordfence Threat Intelligence Team among many other independent security researchers routinely find plugins and themes with vulnerabilities. Some of these vulnerabilities are simple, accidentally introduced celebrity bugs. Other vulnerabilities, however, are introduced by insecure coding practices and methodologies that could have easily been avoided.

Of course, we are all human and wonderfully imperfect, and no matter how experienced we are as developers, we can make mistakes. Security flaws will always be a part of any code. However, our goal in presenting this white paper is to encourage and inform so that secure code becomes the norm in the WordPress space.

This guide was created to analyze the most common and significant security-related coding flaws found in WordPress plugins and themes, in addition to providing recommendations as to how to correct these coding flaws. This information is based on years of experience with WordPress related vulnerabilities that our Threat Intelligence Team has.

II. Missing Capability Checks


One of the most common flaws that we see when auditing the security of WordPress plugins and themes is a lack of appropriate capability checks on functions and actions. In fact, missing capability checks lead to the vast majority of severe vulnerabilities in the WordPress ecosystem.

Most vulnerabilities will start with a missing capability check and then escalate into an additional security issue like Stored Cross-Site Scripting or arbitrary file upload. Therefore, the most critical security flaws in the WordPress ecosystem are often due to a missing capability check in conjunction with another flaw.

Introducing a Weakness

Capability checks verify that a user has the appropriate permission, or capability, to perform an action, often referred to as authorization. When a function does not have a capability check, then the function is not checking to see if the user performing the action has the appropriate authorization to execute the function. Therefore, anyone able to call that function or action, even those without the intended authorization, can abuse its functionality.

These functions can range from updating a plugin's settings to uploading files, actions that should only be allowed by a site administrator in most cases. It's important that plugins and themes have the appropriate capability checks throughout so that attackers without the appropriate authorization can't perform actions like updating a plugin's settings with malicious JavaScript to take over a WordPress site.



A missing capability check leads to a weakness classified by MITRE as "CWE-862: Missing Authorization"¹ which means that the software is missing the appropriate authorization checks and unauthorized parties have the ability to perform some action. This would also be identified by the OWASP's Top Ten as A5:2017-Broken Access Control².

Identifying Missing Capability Checks

Look for action hooks in WordPress and verify that they have the appropriate capability checks on their respective function. Hooks in WordPress are a way for code to trigger at

¹ <https://cwe.mitre.org/data/definitions/862.html>

² https://owasp.org/www-project-top-ten/2017/A5_2017-Broken_Access_Control

certain times during a WordPress site's code execution flow. Hooks are a native WordPress feature, and there are a significant number of different hooks that can be used by developers. For a complete guide on WordPress hooks, we recommend reviewing the WordPress Plugins handbook.³

Action hooks are used in the codebase of WordPress plugins, themes, and core like so:

```
add_action( 'admin_init', 'some_function_here' );
```

On the left you have the hook that is being registered and on the right you have the corresponding function that will execute when the requirements for the hook is met.

The following WordPress hooks are among the most common ones we see associated with insecure access control weaknesses, however, there are several additional WordPress hooks that can be tied to plugin or theme functions containing authorization weaknesses.

`wp_ajax`⁴

This hook is used to create AJAX actions for authenticated users and allows code to be run without requiring any pages to be reloaded. This hook requires the user to be authenticated to a WordPress site even though it does not perform any capability checks. This means that any user logged in, even those with just a subscriber or customer role, can trigger an AJAX action if access is not properly secured. As such, every `wp_ajax` action needs to have its own capability checks so that only users with the appropriate permission can perform the action. For example, an AJAX action used to update settings for a plugin should be restricted to those with the `manage_options` capability. The only time setting updates would not be restricted would be in rare instances where a site's subscriber might need to edit those settings.

The Wordfence Threat Intelligence team has found various instances where `wp_ajax` was used without appropriate capability checks. In fact, this accounted for a vast majority of vulnerabilities found by the Wordfence Threat Intelligence team in 2020.

`admin_init`⁵

This hook is used to run functions upon loading the WordPress administrative dashboard. Despite having "admin" in the name, it does not perform any checks to verify that the function is being triggered by an administrator, meaning that capability checks need to be added any time that `admin_init` is used. This hook will initialize the corresponding

³ <https://developer.wordpress.org/plugins/hooks/>

⁴ https://developer.wordpress.org/reference/hooks/wp_ajax_action/

⁵ https://developer.wordpress.org/reference/hooks/admin_init/

function while a user is accessing the `admin-ajax.php` and `admin-post.php` endpoints which can be accessed by unauthenticated users. This means that unauthenticated users can trigger any functions hooked to an `admin_init` action, so these hooks require appropriate capability checks.

`wp_ajax_nopriv`⁶

This hook is an extension of the `wp_ajax` hook and is used to create AJAX actions for users that are not authenticated. Corresponding functions should require authentication and certain capabilities to execute the AJAX action, however, often there is no capability check present. It is important to ensure that there are no `wp_ajax_nopriv` actions for functions that should require authentication or capability checks.

`admin_post`⁷

This hook is used to fire actions on usage of the `/wp-admin/admin-post.php` endpoint. It can be used to update settings and upload files, amongst other things. Just like with the `admin_init` hook, it does not perform any checks on its own to verify that the function is being triggered by an administrator. Rather, it only detects being triggered from the `admin-post.php` endpoint from an authenticated session. This means that any authenticated user, including subscribers, can trigger functions hooked to an `admin_post` action. Therefore, all of these endpoints require an appropriate capability check.

`admin_post_nopriv`⁸

This hook is an extension of the previous hook and is used to create an `admin-post` action for users that are not authenticated. Just like with the `nopriv` AJAX actions, the issue frequently seen here is that functions are hooked to `nopriv` actions when they should require authentication and certain capabilities to execute. It is important to ensure that there are no `admin_post_nopriv` actions for functions that should require authentication and certain capabilities.

`admin_action`⁹

This hook is used to fire actions in the admin dashboard, and again does not perform any checks to validate if a user is an administrative user, however, it does validate that the

⁶ https://developer.wordpress.org/reference/hooks/wp_ajax_nopriv_action/

⁷ https://developer.wordpress.org/reference/hooks/admin_post_action/

⁸ https://developer.wordpress.org/reference/hooks/admin_post_nopriv_action/

⁹ https://developer.wordpress.org/reference/hooks/admin_action_action/

request is coming from an authenticated session. It's important to make sure that any uses of `admin_action` have a corresponding capability check on the function.

`profile_update`¹⁰ & `personal_options_update`¹¹

These two hooks are used during WordPress user profile updates. Validate that these hooks do not perform any role or user meta updates without the appropriate capability checks so that lower-privileged users cannot perform privilege escalation while updating their profiles.

In addition to hooks with missing capability checks on their corresponding functions, all REST-API endpoints should be validated for appropriate protections. When looking for improperly secured REST-API endpoints, look for any instance of `register_rest_route` and verify that they have a `permissions_callback`¹² on the route.

As of WordPress version 5.5, all REST routes require the use of `permissions_callback`, otherwise a `_doing_it_wrong` notice will be returned. However, developers have the option to set the `permissions_callback` to `__return_true` in cases where they would like the REST endpoint to be publicly accessible. For that reason, we recommend looking for the use of `permissions_callback` when validating if a rest route has the appropriate capability check, though the capability check can exist in other places like the REST-API function itself.

Adding Capability Checks

Resolving flaws created by inadequate capability checks is simple by using the `current_user_can()`¹³ function. This function checks for the appropriate capability to perform an action on user-triggered functions.

If the developer would like a site's administrator to be the only user capable of performing an action, then they should make the capability check look for the `manage_options` capability. The `manage_options` check is reserved for administrators. For a complete guide on WordPress roles and user capabilities, please review the WordPress support

¹⁰ https://developer.wordpress.org/reference/hooks/profile_update/

¹¹ https://developer.wordpress.org/reference/hooks/personal_options_update/

¹² <https://developer.wordpress.org/rest-api/extending-the-rest-api/adding-custom-endpoints/#permissions-callback>

¹³ https://developer.wordpress.org/reference/functions/current_user_can/

article¹⁴. This guide can be used as a resource when determining which capability is the most appropriate to check for on individual WordPress functions.

When it comes to adding the appropriate capability checks to REST-API routes, use the `permissions_callback` function to validate that the user can access the specified REST-API endpoint. Just like with the `current_user_can()` function, you will need to specify an appropriate capability to check.

Additional Note

Just as important as it is to validate that all functions have capability checks, it is equally important to verify that the checks in use are appropriate for the intended use.

For example, there have been numerous cases where functions like `is_admin` have been erroneously used for capability checks, leading to access control security flaws¹⁵. The `is_admin` function only validates that a request is coming from within the WordPress admin dashboard, it does not verify that the request is coming from an administrator. Therefore, any logged in user can trigger any functions or actions that only have `is_admin` in use for access control. Therefore, it is highly recommended not to use `is_admin` in an attempt to add access control.

In addition, there have been cases where the capability checks used by developers have made it possible for unauthorized users to perform restricted actions when the appropriate capability has not been validated.¹⁶

This highlights the importance of not only adding a capability check, but to continuously be mindful of whether or not the capability check being used is appropriate for the action provided to users.

¹⁴ <https://wordpress.org/support/article/roles-and-capabilities/>

¹⁵ <https://www.wordfence.com/blog/2019/12/critical-vulnerability-patched-in-301-redirects-easy-redirect-manager/>

¹⁶ <https://www.wordfence.com/blog/2021/03/medium-severity-vulnerability-patched-in-user-profile-picture-plugin/>

Example from WP Database Reset Plugin <= 3.1 - Unauthenticated Database Reset Vulnerability ([CVE-2020-7048](#))¹⁷

The following is an example of an insecure access control vulnerability discovered in the WP Database Reset plugin. Below is the `admin_init` action that hooked to the `reset` function that triggered the reset of a WordPress database. This action also included resetting all users.

```
public function run() {  
    add_action( 'admin_init', array( $this, 'reset' ) );  
}
```

Insecure Code Prior to Patch

The `reset()` function hooked from the `admin_init` action was vulnerable because it had no capability check to validate that a database reset was being triggered by an administrator. Therefore, it was possible for unauthenticated users to trigger the `reset` function.

```
public function reset( array $tables ) {  
    if ( in_array( 'users', $tables ) ) {  
        $this->reset_users = true;  
    }  
  
    $this->validate_selected( $tables );  
    $this->set_backup();  
    $this->reinstall();  
    $this->restore_backup();  
}  
  
private function validate_selected( array $tables ) {  
    if ( ! empty( $tables ) && is_array( $tables ) ) {  
        $this->selected = array_flip( $tables );  
        return;  
    }  
  
    throw new Exception( __( 'You did not select any database tables', 'wordpress-database-reset' ) );  
}
```

¹⁷ <https://www.wordfence.com/blog/2020/01/easily-exploitable-vulnerabilities-patched-in-wp-database-reset-plugin/>

Corrected and Secured Code

This vulnerability was corrected when the `reset()` function hooked from the `admin_init` action implemented access control using `current_user_can('administrator')` to validate that the user making the request is an administrator, in addition to a nonce check for CSRF protection.

```
public function reset(array $tables)
{
    if (wp_verify_nonce(@$_REQUEST['submit_reset_form'], action: 'reset_nonce') && current_user_can('administrator')) {
        // Check if current user is Admin and check the nonce
    }

    if (in_array('users', $tables)) {
        $this->reset_users = true;
    }

    $this->validate_selected($tables);
    $this->set_backup();
    $this->reinstall();
    $this->restore_backup();
} else {
    throw new Exception(__('Please reload the page and try again. Double check your security code.', 'wordpress-database-reset'));
}
}
```

III. Missing Cross-Site Request Forgery Protection.

A Cross-Site Request Forgery (CSRF) attack starts with a malicious actor crafting a legitimate request to a target site. If they can successfully trick a victim into performing an action, such as clicking on a link while the victim is authenticated to the target site, then the attacker can perform actions on behalf of that authenticated user. This means that an attacker can perform an action that appears to be coming from a legitimate authenticated user session, however, the request sent to the website is completely forged by the attacker.

Cross-Site Request Forgery protection requires the use of nonces, which means “number used once.” In WordPress, nonces¹⁸ work slightly differently. Instead of only being used once, they are valid for a short period of time or until a user's session has ended; whichever comes first.

The Wordfence Threat Intelligence team frequently finds that WordPress functions miss the appropriate nonce protection on functions. These flaws are especially common in conjunction with flaws that are a result of inadequate authorization.

Opening a Door to Forged Requests

Verifying that WordPress plugins and themes have the appropriate Cross-Site Request Forgery protection ensures attackers cannot trick WordPress site owners into performing unwanted actions.

Because attacks using Cross-Site Request Forgery vulnerabilities appear to be coming from legitimate administrator sessions, it is often not possible for web application firewalls (WAFs) to protect websites from these types of attacks. As such, it is crucially important that plugin and theme developers provide adequate CSRF protection.

Missing nonce protections leads to a weakness classified by MITRE as “CWE-352: Cross-Site Request Forgery (CSRF)”¹⁹ which means that the software is missing the appropriate checks to verify the integrity of who submitted a request.

Identifying a Lack of Cross-Site Request Forgery Protection

As with inadequate access control vulnerabilities, look for WordPress hooks and validate that each corresponding function has the appropriate nonce validation functions in use.

¹⁸ https://codex.wordpress.org/WordPress_Nonces

¹⁹ <https://cwe.mitre.org/data/definitions/352.html>

The most common WordPress action hooks we recommend looking for are the same as previously discussed in the capability check section.

- `admin_init`
- `wp_ajax`
- `wp_ajax_nopriv`
- `admin_post`
- `admin_post_nopriv`
- `admin_action`
- `admin_menu`

If the hooked function uses `check_ajax_referrer`²⁰, `check_admin_referrer`²¹, or `wp_verify_nonce`²² then it does have a nonce check and it should be protected against CSRF vulnerabilities.

However, it is possible that nonce validation can be improperly implemented allowing for CSRF protection bypasses²³, so it is important to validate that the nonce check is not merely conditional on the nonce being present. The nonce should always be checked whether present in the request or not.

Adding CSRF Protection

Adding CSRF protection is relatively simple. To add CSRF protection, developers will need to create the nonce as part of the page or form using `wp_create_nonce`. Then, for the nonce validation, one of the following functions, `check_ajax_referrer`, `check_admin_referrer`, or `wp_verify_nonce`, needs to be added to the function that is being protected. There should be a conditional that terminates the execution of the rest of the function if a nonce check fails or if the nonce is not present in order to provide CSRF protection.

Additional Note

An additional flaw we often see is the use of WordPress nonces as a means of authorization instead of using the appropriate capability check. This is not advised. We recommend using WordPress nonces in conjunction with the appropriate capability

²⁰ https://developer.wordpress.org/reference/functions/check_ajax_referer/

²¹ https://developer.wordpress.org/reference/functions/check_admin_referer/

²² https://developer.wordpress.org/reference/functions/wp_verify_nonce/

²³ <https://blog.nintech.net/25-wordpress-plugins-vulnerable-to-csrf-attacks/>

check. Nonces can be compromised, and therefore, should always be assumed as such and never relied upon for authorization.

For example, the Wordfence Threat Intelligence Team discovered an unauthenticated arbitrary nonce generation present in Redirection for Contact Form 7²⁴ that made it possible for any unauthenticated attacker to generate an arbitrary valid nonce. That valid nonce could then be used to bypass any nonce checks that were being used as capability checks in other plugins. This would have made it possible for an attacker to trigger any functions that only used a nonce check as a means of access control.

²⁴ <https://www.wordfence.com/blog/2021/04/severe-vulnerabilities-patched-in-redirection-for-contact-form-7-plugin/>

Example from Real-Time Find and Replace plugin <= 3.9 - Cross-Site Request Forgery to Cross-Site Scripting ([CVE-2020-13641](#))²⁵

This example of a Cross-Site Request Forgery vulnerability escalated to a more severe Stored Cross-Site Scripting vulnerability within the Real-Time Find and Replace plugin. The `far_add_pages` function shown below added the page used for the plugin and hooked to the `far_options_page` function used to control the plugin's find and replace functionality.

```
function far_add_pages(){ // Add a submenu under Tools
    $page = add_submenu_page('tools.php', 'Real-Time Find and Replace', 'Real-Time Find and Replace', 'activate_plugins',
    'real-time-find-and-replace', 'far_options_page');
    add_action("admin_print_scripts-$page", 'far_admin_scripts');
```

Insecure Code Prior to Patch

The `far_options_page` function hooked from the menu page action has a nonce check in place, while the submenu page validates that a user has the `activate_plugins` capability in order to trigger the function. However, the function is missing a nonce check to provide CSRF protection.

```
function far_options_page(){
    if (isset($_POST['setup-update'])) {
        $_POST = stripslashes_deep($_POST);
        if (is_array($_POST['farfind'])){ // If atleast one find has been submitted
            foreach ($_POST['farfind'] as $key => $find){
                if (empty($find)){ // if empty ones have been submitted we get rid of the extra data submitted if any.
                    unset($_POST['farfind'][$key]);
                    unset($_POST['farregex'][$key]);
                    unset($_POST['farreplace'][$key]);
                }
            }
        }
    }
}
```

²⁵ <https://www.wordfence.com/blog/2020/04/high-severity-vulnerability-patched-in-real-time-find-and-replace-plugin/>

Corrected and Secured Code

The following is the same `far_options_page()` function hooked from menu page action that now has `check_admin_referrer('far_rules_form')` in place to validate that a request is coming from a legitimate authenticated user session providing adequate CSRF protection.

```
function far_options_page() {
    if ( isset( $_POST['setup-update'] ) ) {
        check_admin_referrer( action: 'far_rules_form' );
        $_POST = stripslashes_deep( $_POST );

        // If atleast one find has been submitted
        if ( isset ( $_POST['farfind'] ) && is_array( $_POST['farfind'] ) ) {
            foreach ( $_POST['farfind'] as $key => $find ){
```


IV. Missing File Upload Validation

File uploading, whether it be from a setting import functionality or from a user profile picture upload, has the potential to be disastrous for a WordPress site if the file isn't properly validated before being uploaded to the site's server.

While malicious file upload vulnerabilities may be less commonly found in WordPress plugins, themes and core, they are among the most critical vulnerabilities you can find in WordPress due to the fact that they open the door to remote code execution and complete site takeover.

An Open Door to Remote Code Execution

WordPress runs on PHP and, therefore, any PHP file installed in any directory of a WordPress site will become executable, unless specific security hardening has been implemented. When a user has the ability to upload a PHP file to a WordPress site, they can add functions to the file that can allow them to achieve remote code execution, which allows that user to run commands on the target server. This can allow an attacker to perform complex and dangerous attacks on the compromised server.

If there are vulnerabilities on the server, then the attacker can escalate their control over an affected server, potentially gaining root access to the entire server and taking over any other account hosted there. Even if they fail to gain root access, the attacker will have access to the WordPress site's hosting environment and be able to delete, modify and further infect any files and any sites hosted in the same account.

In addition to obtaining remote code execution through PHP file uploads, unrestricted file uploads can also make it possible for attackers to upload other files like HTML files, JavaScript files, or SVG files that can introduce scripts that, when accessed by administrators, perform a variety of actions. Furthermore, unrestricted file uploads can make it possible to upload executable files that could have a significant impact on a server, as we've seen in cases of cryptominers, spam emailers and other malware compromising server resources.

Insufficient file validation leads to a weakness classified by MITRE as "CWE-434: Unrestricted Upload of File with Dangerous Type"²⁶ which means that the software is missing the appropriate checks to validate a files type, and allows uploads of dangerous files, which in the case of WordPress would be PHP.

²⁶ <https://cwe.mitre.org/data/definitions/434.html>

Finding Unrestricted File Uploads

There are several things to look for when validating file upload functionalities.

`$_FILES`²⁷

This super global is used to obtain file data from a request, so it is critical to look for any instances of it and review the corresponding functionality. Verify that, if a file is being uploaded to the server from an external source, that there are appropriate file validation and sanitization mechanisms in place.

One mistake we have seen is the use of `$_FILES['type']` global variable to validate a file's type. This variable determines the file's type based on the Content-Type header sent from a browser, however this can easily be spoofed by simply intercepting a request and setting the Content-Type header. Attackers can easily set this to make the file appear safe while the content being sent is not. Content-Type headers should be treated as an unsafe method for file type checking.

In addition to looking for any instances of `$_FILES`, any PHP-specific file function used for uploading file content should also be examined. The following PHP functions can be used to upload files in PHP. If any of these functions are being used without any file type validation like `wp_check_filetype` or `wp_check_filetype_and_ext` then they may be vulnerable to arbitrary file uploads.

- `file_put_contents`²⁸
- `fopen`²⁹
- `fwrite`³⁰
- `move_uploaded_file`³¹

Further analysis may be required if there is a custom file type mechanism in place. If a custom file type checking mechanism is in place, then it must perform file type checking based on the content of the file, so that malicious content cannot be hidden in seemingly innocent files like image files, and it must also perform file type checking on the extension of the file so that files with dangerous extensions cannot be uploaded.

²⁷ <https://www.php.net/manual/en/reserved.variables.files.php>

²⁸ <https://www.php.net/manual/en/function.file-put-contents.php>

²⁹ <https://www.php.net/manual/en/function.fopen.php>

³⁰ <https://www.php.net/manual/en/function.fwrite>

³¹ <https://www.php.net/manual/en/function.move-uploaded-file.php>

When auditing for arbitrary file upload vulnerabilities in WordPress plugins and themes, examine any use of `exif_imagetype`³² to determine a file type. This function will use the first few bytes of a file, also known as magic bytes, to determine its type.

Unfortunately, files also can easily be spoofed to bypass this check by simply adding the required magic bytes to the top of a file. As such, `exif_imagetype` is considered an insecure way to perform file upload validation.

Important note

When looking for arbitrary file upload vulnerabilities, note that the `sanitize_file_name`³³ function can sometimes lead to arbitrary file upload vulnerabilities if not properly implemented.

For example, if the filename sanitization function occurs after a filetype extension check that only checks extension against a blacklist of known bad extensions, then an attacker could supply a filename to bypass this check. For example, an attacker could name a file `maliciousfile(.php)` which would pass the file extension filtering since `(.php)` is not a known malicious file extension. Once the file extension check is completed and the filename is passed to `sanitize_file_name`, the parentheses will be stripped from the filename converting it to `maliciousfile.php`, allowing for a malicious file upload bypassing the original file extension checking.

Further, some web servers will process the first extension found in a file, therefore, it is important to detect and prevent the uploading of files with double extensions. If a file upload is not using `wp_handle_upload` or `sanitize_file_name`, then it may be susceptible to double extension uploading and may introduce a vulnerability on some servers.

Properly Validating File Uploads

WordPress makes it easy to handle file uploads securely with the `wp_handle_upload()`³⁴ function. This function checks the extension of the file being uploaded with a predefined list of acceptable extensions. Using this native function when building file upload functionality for a WordPress plugin or theme will natively block potentially malicious files from being uploaded to a WordPress site.

³² <https://www.php.net/manual/en/function.exif-imagetype.php>

³³ https://developer.wordpress.org/reference/functions/sanitize_file_name/

³⁴ https://developer.wordpress.org/reference/functions/wp_handle_upload/

If it is preferred to use custom file upload features, it is recommended to use the `wp_check_filetype_and_ext()`³⁵ or `wp_check_filetype()`³⁶ functions at the very minimum to validate the uploaded file's type and its extension to verify that it should be allowed to be uploaded to a WordPress site.

It is also possible to use custom file filtering functions, however, they must be securely implemented. In order to do that, verify that the contents within the file are not malicious. As well, validate that the file's extension is a safe one, this can be done with a regular expression and blocklisting. The file extension checking should prohibit the upload of all various PHP extensions like `.php`, `.phtml`, `.phar`, `php6`, etc along with other potentially harmful file extensions like `.svg`, `.html`, `.js`, etc.

Alternatively, a developer can implement file extension allowlisting instead of blocklisting which will more easily restrict file upload ability to files that the application intends to allow. For example, if an application only needs to upload image files, then only allow files with the `.jpeg`, `.png`, `.gif`, and other desired image file extensions along with other checks.

Always make sure that `sanitize_file_name()` happens at least once before the file's extension is checked so that double extensions cannot be supplied and users will not be able to bypass extension checking.

³⁵ https://developer.wordpress.org/reference/functions/wp_check_filetype_and_ext/

³⁶ https://developer.wordpress.org/reference/functions/wp_check_filetype/

Example from Quiz and Survey Master Plugin <=7.0.0 - Unauthenticated Arbitrary File Upload ([CVE-2020-35949](https://www.wordfence.com/blog/2020/08/critical-vulnerabilities-patched-in-quiz-and-survey-master-plugin))³⁷

The following is an example of an arbitrary file upload vulnerability discovered in the Quiz and Survey Master plugin. Below are the `wp_ajax` and `wp_ajax_nopriv` actions that hooked to the `qsm_upload_image_fd_question` function that would upload a file upon submission of a quiz.

```
//Upload file of file upload question type
add_action('wp_ajax_qsm_upload_image_fd_question', array($this, 'qsm_upload_image_fd_question'));
add_action('wp_ajax_nopriv_qsm_upload_image_fd_question', array($this, 'qsm_upload_image_fd_question'));
```

Insecure Code Prior to Patch

This function was vulnerable to arbitrary file uploads due to the fact that it used the files Content-Type header to determine the files type by using `$_FILES['file']['type']`. This could be spoofed to anything that would pass the mime check like `image/jpeg` even when the file was using a PHP file extension.

```
public function qsm_upload_image_fd_question(){
    global $mlwQuizMasterNext;
    $question_id = isset($_POST['question_id']) ? sanitize_text_field($_POST['question_id']) : 0;
    $file_upload_type = $mlwQuizMasterNext->pluginHelper->get_question_setting($question_id, 'file_upload_type');
    $file_upload_limit = $mlwQuizMasterNext->pluginHelper->get_question_setting($question_id, 'file_upload_limit');
    $mimes = array();
    if($file_upload_type){
        $file_type_exp = explode(' ', $file_upload_type);
        foreach ($file_type_exp as $value) {
            if($value == 'image'){
                $mimes[] = 'image/jpeg';
                $mimes[] = 'image/png';
                $mimes[] = 'image/x-icon';
                $mimes[] = 'image/gif';
            }else if($value == 'doc'){
                $mimes[] = 'application/msword';
                $mimes[] = 'application/vnd.openxmlformats-officedocument.wordprocessingml.document';
            }else if($value == 'excel'){
                $mimes[] = 'application/excel, application/vnd.ms-excel, application/x-excel, application/x-msexcel';
                $mimes[] = 'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet';
            }else{
                $mimes[] = $value;
            }
        }
    }
    $json = array();
    if (in_array($_FILES["file"]["type"], $mimes)) {
        if($_FILES["file"]["size"] >= $file_upload_limit * 1024 * 1024){
            $json['type'] = 'error';
            $json['message'] = 'File is too large. File must be less than ' . $file_upload_limit . ' MB';
            echo json_encode($json);
            exit;
        }
    }
}
```

³⁷ <https://www.wordfence.com/blog/2020/08/critical-vulnerabilities-patched-in-quiz-and-survey-master-plugin>

Secured and Corrected Code Sample

The developers corrected this flaw by implementing the use of `wp_check_filetype` which retrieves the files type by checking the files extension and will only return it if it is in the list of acceptable extensions created by WordPress.

```
public function qsm_upload_image_fd_question(){
    global $mlwQuizMasterNext;
    $question_id = isset($_POST['question_id']) ? sanitize_text_field($_POST['question_id']) : 0;
    $file_upload_type = $mlwQuizMasterNext->pluginHelper->get_question_setting($question_id, 'file_upload_type');
    $file_upload_limit = $mlwQuizMasterNext->pluginHelper->get_question_setting($question_id, 'file_upload_limit');
    $mimes = array();
    if($file_upload_type){
        $file_type_exp = explode( 'delimiter:', ',', $file_upload_type);
        foreach ($file_type_exp as $value) {
            if($value == 'image'){
                $mimes[] = 'image/jpeg';
                $mimes[] = 'image/png';
                $mimes[] = 'image/x-icon';
                $mimes[] = 'image/gif';
            }else if($value == 'doc'){
                $mimes[] = 'application/msword';
                $mimes[] = 'application/vnd.openxmlformats-officedocument.wordprocessingml.document';
            }else if($value == 'excel'){
                $mimes[] = 'application/excel, application/vnd.ms-excel, application/x-excel, application/x-msexcel';
                $mimes[] = 'application/vnd.openxmlformats-officedocument.spreadsheetml.sheet';
            }else{
                $mimes[] = $value;
            }
        }
    }
    $json = array();
    $file_name = sanitize_file_name( $_FILES["file"]["name"] );
    $validate_file = wp_check_filetype( $file_name );
    if ( isset( $validate_file['type'] ) && in_array($validate_file['type'], $mimes) ) {
        if($_FILES["file"]["size"] >= $file_upload_limit * 1024 * 1024){
            $json['type'] = 'error';
            $json['message'] = __('File is too large. File must be less than ', 'quiz-master-next') . $file_upload_limit . ' MB';
            echo json_encode($json);
            exit;
        }
    }
}
```

V. Deserialization of User-Supplied Input

PHP is an object oriented programming (OOP)³⁸ language, which allows the code to call objects from classes. Due to the nature of Object Oriented Programming and the fact that PHP allows storing and transmitting objects via serialization, deserialization of user-supplied input can lead to potentially disastrous events. If the `unserialize()`³⁹ or `maybe_unserialize()`⁴⁰ functions are used on user-supplied input, then an attacker can modify the execution of loaded classes via magic methods.

In addition to the standard deserialization vulnerabilities as a result of the `unserialize` and `maybe_unserialize` functions, there are also phar deserialization weaknesses. A phar⁴¹ file is a PHP archive, in other words, it is a single file containing several PHP files. These phar files have serialized meta-data stored in the manifest of the file. When called through certain file functions, the meta-data will deserialize the same way the `unserialize()` function deserializes data. This can lead to a deserialization weakness. Though these are less commonly found and require unique conditions to exploit, they still require attention when developing code for WordPress plugins and themes or reviewing them for vulnerabilities.

These deserialization flaws can lead to a vulnerability which is identified as PHP Object Injection.

The Magic of PHP Object Injection Vulnerabilities

PHP Object Injection vulnerabilities on their own are not critical, however, when they are paired with a full POP chain they can become disastrous leading to full site takeover.

A POP chain is a way to chain magic methods⁴² together, or use a single magic method, to execute objects in classes. These actions perform actions that typically involve deleting, updating, or reading files and triggering commands remotely.

The most common magic methods that can be triggered as part of a POP chain are `__wakeup` and `__destruct`, which are automatically called during the deserialization of objects in PHP.

³⁸ https://www.w3schools.com/php/php_oop_what_is.asp

³⁹ <https://www.php.net/manual/en/function.unserialize.php>

⁴⁰ https://developer.wordpress.org/reference/functions/maybe_unserialize/

⁴¹ <https://www.php.net/manual/en/phar.using.intro.php>

⁴² <https://www.php.net/manual/en/language.oop5.magic.php>

Once an attacker has access to update, delete, or read files they can steal sensitive data, inject backdoors and malware into sites, and cause denial of service attacks. Further, when they have the ability to execute commands remotely they can completely take over the vulnerable server. As such, it is critical to ensure that deserialization of unvalidated user-supplied input does not occur.

Deserialization flaws lead to a weakness classified by MITRE as “CWE-502: Deserialization of Untrusted Data”⁴³ which means that the software has a flaw that allows the deserialization of data that has not been validated or sanitized, such as user-supplied input.

Discovering Deserialization Weaknesses

To find deserialization weaknesses, the presence of `unserialize()` or `maybe_unserialize()` functions anywhere throughout the codebase should be analyzed. If they are found, validate that no user input can be supplied to those functions. If user input can be supplied, then ensure that the input is sanitized or validated. This sanitization or validation should strip out any use of an object that is not the intended data. If there is no validation on user-supplied input passed to `unserialize()`, then it is likely that there is a PHP Object Injection vulnerability in the code and further analysis should be done.

In addition to looking for the presence of `unserialize()` or `maybe_unserialize()`, check for the use of any file system function with unfiltered user-supplied input. These functions include `file_exists`, `fopen`, `copy`, `filesize`, `mkdir`, and `file_get_content`, to name a few. Use of these functions in conjunction with unfiltered input can indicate the presence of a phar deserialization weakness. Further investigation should determine whether the `phar://` wrapper along with a complete path to a file can be supplied.

Note that the file does not need a .phar extension in order to be deserialized. During an exploit, a .phar files extension can simply be switched to .jpg due to the phar wrapper. This means that the upload of a phar file that needs to be used in a phar deserialization vulnerability will be relatively simple and can bypass traditional extension filtering protections.

⁴³ <https://cwe.mitre.org/data/definitions/502.html>

A recent example of a phar deserialization weakness resulting in a PHP Object Injection vulnerability was recently patched in WordPress⁴⁴ from the PHPMailer software. The PHPMailer software used the `file_exists()` function on the file path and allowed the use of wrappers in an attempt to support UNC filenames. This made it possible to deserialize phar files if user-supplied paths could be used.

If a PHP Object Injection vulnerability is discovered, then the next step is to look for any magic methods that would escalate the severity of the vulnerability. The most common magic methods that should be looked for are `__wakeup`, `__destruct`, and `__toString`, though it is important to remember that there are several additional magic methods that can potentially be used for a POP chain.

Due to the unique nature of WordPress core, plugin and theme ecosystem, it is entirely possible for a site to have a single plugin with a PHP Object Injection vulnerability installed, while another distinct plugin contains a magic method that could be used to complete a POP chain for the PHP Object injection vulnerability.

Preventing Deserialization Vulnerabilities

Use JSON encoding rather than PHP serialization for any user-supplied structured data so that unauthorized users cannot alter the flow of code execution to perform malicious actions.

As for phar deserialization vulnerabilities, validate that PHP wrappers like `phar://` cannot be supplied when users are able to supply the full path to files. In addition, validate that PHP wrappers cannot be supplied to functions like `file_exists`, `fopen`, `copy`, `filesize`, `mkdir`, `file_get_content` and others that may deserialize phar file metadata when supplied. Do so by sanitizing the user-supplied input to strip out known PHP wrappers⁴⁵. Alternatively, plugin functionality can restrict acceptance of user-supplied paths so that the user can only supply a filename, while strict functionality supplies the path to the file for usage in a function.

⁴⁴ <https://www.wordfence.com/blog/2021/05/wordpress-5-7-2-security-release-what-you-need-to-know/>

⁴⁵ <https://www.php.net/manual/en/wrappers.php>

Example from Facebook for WordPress Plugin <=2.2.2 - Unauthenticated PHP Object Injection ([CVE-2021-24217](https://www.wordfence.com/blog/2021/03/two-vulnerabilities-patched-in-facebook-for-wordpress-plugin))⁴⁶

The following is an example of an unauthenticated PHP Object Injection vulnerability that was present in the Facebook for WordPress plugin. Below are the `admin_post` and `admin_post_nopriv` actions that are hooked to the `handle_postback` function that would ultimately trigger the `run_action` function.

```
public function __construct( $auth_level = self::BOTH ) {
    if ( empty( $this->action ) ) {
        throw new Exception( message: 'Action not defined for class ' . __CLASS__ );
    }
    add_action( $this->action, array( $this, 'launch' ), (int) $this->priority, (int) $this->argument_count );
    if ( $auth_level & self::LOGGED_IN ) {
        add_action( "admin_post_wp_async_{$this->action}", array( $this, 'handle_postback' ) );
    }
    if ( $auth_level & self::LOGGED_OUT ) {
        add_action( "admin_post_nopriv_wp_async_{$this->action}", array( $this, 'handle_postback' ) );
    }
}
```

Insecure Code Prior to Patch

The `run_action` function was vulnerable to PHP Object injection due to the use of `unserialize()` on the user-supplied input obtained from the `event_data` variable. This plugin did have a complete POP chain with `__destruct` present in the GuzzleHTTP package included in the plugin.

```
protected function run_action() {
    try {
        $num_events = $_POST['num_events'];
        if( $num_events == 0 ){
            return;
        }
        $events = unserialize(base64_decode($_POST['event_data']));
        // When an array has just one object, the deserialization process
        // returns just the object
        // and we want an array
        if( $num_events == 1 ){
            $events = array( $events );
        }

        FacebookServerSideEvent::send($events);
    }
    catch (\Exception $ex) {
        error_log($ex);
    }
}
```

⁴⁶ <https://www.wordfence.com/blog/2021/03/two-vulnerabilities-patched-in-facebook-for-wordpress-plugin>

Secured and Corrected Code Example

The developers fixed the vulnerability by using `json_decode()` on the user-supplied data instead of using the `unserialize()` function. They also removed the GuzzleHTTP package from the plugin.

```
protected function run_action() {
    try {
        $num_events = $_POST['num_events'];
        if( $num_events == 0 ){
            return;
        }
        // $_POST['event_data'] is decoded from base 64, returning a JSON string
        // and decoded as a php array
        $events_as_array = json_decode(base64_decode($_POST['event_data']), assoc: true);
        // If the passed json string is invalid, no processing is done
        if(!$events_as_array){
            return;
        }
        $events = array();
        // Every event is a php array and casted to an Event object
        foreach( $events_as_array as $event_as_array ){
            $event = $this->convert_array_to_event($event_as_array);
            $events[] = $event;
        }
        FacebookServerSideEvent::send($events);
    }
    catch (\Exception $ex) {
        error_log($ex);
    }
}
```

VI. Lack of Sanitization and Escaping on User-Supplied Inputs

WordPress plugins and themes almost always use some form of user-supplied input, whether it be to change settings via the WordPress administrative dashboard or register a custom form that takes user-supplied information from the front end of the WordPress site. No matter how user data is handled, it must be properly secured. When user-supplied inputs are not properly secured through sanitizing and escaping, they can lead to a vulnerability called Cross-Site Scripting (XSS), which is the most common vulnerability we've seen in both WordPress plugins and themes.

This vulnerability occurs when users can supply values that contain HTML/scripting tags (e.g. <,>) or unescaped single or double quotes that can be used to add attributes to HTML elements in values that make it possible for users to inject malicious code like JavaScript into web pages. If these values are later echoed to or displayed on a web page then the browser will execute the supplied JavaScript which can be used to perform a wide variety of malicious actions.

There are three core forms of Cross-Site Scripting: reflected, stored, and DOM-based. Reflected XSS occurs when the malicious code is immediately echoed or displayed back in the browser and is not stored anywhere on the site's server or in the database. These are typically one-time exploits and require some form of social engineering to be successful. Alternatively, Stored XSS occurs when the malicious code is stored in the site database or on the server and can be executed multiple times, for instance, every time someone accesses a web page. These typically do not require any form of social engineering to be successful, though it is possible in some cases to, for instance, use a CSRF vulnerability to trick a user with high privileges into injecting Stored XSS, which would require social engineering. The final type is DOM-based XSS which is far less common in WordPress and occurs when user-supplied input can be supplied to a sink within the DOM (Document Object Model). These are usually harder to successfully exploit and will require some level of social engineering.

Cross-Site Scripting Vulnerabilities Open Doors to Remote Code Execution

Protecting against vulnerabilities created by unsanitized inputs and unescaped outputs is critical on WordPress sites. Cross-Site Scripting vulnerabilities can be used to perform a plethora of actions on a WordPress site if the JavaScript is triggered while an administrator is authenticated and visits a page containing the malicious JavaScript payload. An attacker could write a script to inject a backdoor in theme or plugin files.

Alternatively they could write a script to create a new administrative user account that could be used to log into the vulnerable target site and further infect it. These are just a couple of ways an attacker can use XSS vulnerabilities to impact a site.

In addition to WordPress site modification, Cross-Site Scripting vulnerabilities can be used to redirect visitors browsing a site to any external site. These are often other infected sites containing spam. Worse yet, these vulnerabilities can be used to exploit vulnerabilities in site visitors' browsers, or hook to a browser using tools like BeEF⁴⁷, allowing attackers to perform actions to further infect victims' computers or steal sensitive data.

Considering the criticality of Cross-Site Scripting vulnerabilities and how easily they can be introduced into software, it is especially important to make the best effort to avoid introducing them into WordPress core, themes, and plugins not only to protect the site owners but site visitors as well.

Unsanitized and unescaped inputs leads to a weakness classified by MITRE as "CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')"⁴⁸ which means that the software has a flaw that allows unsanitized or unescaped user-supplied input which can be used to execute malicious code in a user's browser. This could also be identified by the OWASP's Top Ten as A7:2017-Cross-Site Scripting (XSS)⁴⁹.

Discovering Cross-Site Scripting Vulnerabilities

When looking for the presence of Cross-Site Scripting vulnerabilities, look for any instance where user-supplied data is being accepted. This includes all of the following super global variables:

`$_POST`⁵⁰ : Used to obtain parameters from a HTTP POST request.

`$_GET`⁵¹ : Used to obtain parameters from a HTTP GET request.

`$_REQUEST`⁵²: Used to obtain parameters from any HTTP request type.

⁴⁷ <https://beefproject.com/>

⁴⁸ <https://cwe.mitre.org/data/definitions/79.html>

⁴⁹ [https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_\(XSS\)](https://owasp.org/www-project-top-ten/2017/A7_2017-Cross-Site_Scripting_(XSS))

⁵⁰ <https://www.php.net/manual/en/reserved.variables.post.php>

⁵¹ <https://www.php.net/manual/en/reserved.variables.get.php>

⁵² <https://www.php.net/manual/en/reserved.variables.request.php>

`$_SERVER`⁵³: Used to obtain various different header related values from an HTTP request

`$_COOKIE`⁵⁴ : Used to obtain HTTP cookie values from the cookie header.

`$_FILES`⁵⁵ : Used to obtain information like file content, title, and more about a file from an HTTP request.

Additionally, using a function such as `filter_input()`⁵⁶ on any of these superglobals *without specifying an adequate filter* may also indicate the presence of a vulnerability.

The use of these variables with user-supplied inputs must be validated to ensure that the supplied data is sanitized to guard against malicious activity. Sanitization can be done either using a WordPress sanitization function or a custom sanitization function.

Any user-supplied input should also be traced to see if the data is ever echoed or displayed to a webpage. If it is, then it should also be verified that the data is being escaped during output with a WordPress escaping function or custom escaping function.

A vulnerable example may look something like: `echo $uservalue` where the value isn't immediately retrieved from user-supplied inputs, however, was supplied by a user at some point.

In addition, the following WordPress specific functions can potentially introduce a XSS vulnerability if improperly implemented.

`add_query_arg`⁵⁷/`remove_query_arg`⁵⁸ : These functions are used to add and remove query arguments (aka parameters) from a string. The parameters supplied to these functions are echoed to a page. Without proper sanitizing and escaping on any user-supplied values passed to these functions, a malicious actor can easily add scripting tags to the values then echoed and executed in the browser.

⁵³ <https://www.php.net/manual/en/reserved.variables.server.php>

⁵⁴ <https://www.php.net/manual/en/reserved.variables.cookies.php>

⁵⁵ <https://www.php.net/manual/en/reserved.variables.files.php>

⁵⁶ <https://www.php.net/manual/en/function.filter-input.php>

⁵⁷ https://developer.wordpress.org/reference/functions/add_query_arg/

⁵⁸ https://developer.wordpress.org/reference/functions/remove_query_arg/

The `unfiltered_html`⁵⁹ capability

When looking for Cross-Site Scripting vulnerabilities, be mindful of the `unfiltered_html` capability. This allows site users with the administrator and editor roles to add `unfiltered_html` to pages, posts, widgets and other places throughout a WordPress site. There are several instances where administrative and editor users should be able to add scripting tags to areas, and not all instances of unsanitized input allowed by these two roles should be considered Cross-Site Scripting vulnerabilities.

Further, when investigating plugins or themes that allow editing of pages and posts, such as page builder plugins, it is important to verify that the `unfiltered_html` capability is being upheld. Verify that lower-level users without the `unfiltered_html` capability that have access to edit pages and posts, like contributors and authors, do not have any introduced capabilities that allow them to use HTML tags in pages and posts so that they cannot inject malicious scripts and exploit Cross-Site Scripting vulnerabilities. A perfect example of how this can easily become an issue is the Cross-Site Scripting vulnerabilities found throughout the Elementor ecosystem that allowed contributors and authors to inject malicious scripts in pages due to the lack of protection on tags that could be used by users in the Elementor editor.⁶⁰ If a plugin adds additional lower-level user roles, those should also be examined for these types of protections.

Sanitizing Inputs and Escaping Outputs to Prevent Cross-Site Scripting Vulnerabilities and Properly Format Data

WordPress provides an array of functions that can be used to sanitize user-supplied inputs.⁶¹ Sanitization will strip out disallowed characters to prevent Cross-Site Scripting vulnerabilities, and ensure data is formatted in the way that it should be. The following is a list of sanitization functions that can be used in WordPress plugins and themes. This list also contains sanitization functions that provide protection against other types of vulnerabilities that are related to unsanitized user inputs.

Please bear in mind that many of these functions are specific to their intended use case and can potentially cause issues or even introduce new vulnerabilities if used incorrectly or outside of appropriate contexts (for instance, running `sanitize_file_name` after validating the extension of an uploaded file).

⁵⁹ https://wordpress.org/support/article/roles-and-capabilities/#unfiltered_html

⁶⁰ <https://www.wordfence.com/blog/2021/04/recent-patches-rock-the-elementor-ecosystem/>

⁶¹ <https://developer.wordpress.org/plugins/security/securing-input/>

- `sanitize_text_field()`
- `sanitize_email()`
- `sanitize_file_name()`
- `sanitize_html_class()`
- `sanitize_key()`
- `sanitize_meta()`
- `sanitize_mime_type()`
- `sanitize_option()`
- `sanitize_title()`
- `sanitize_title_for_query()`
- `sanitize_title_with_dashes()`
- `sanitize_user()`
- `esc_url_raw()`

When it comes to escaping outputs, there are several additional WordPress functions that can be used.⁶² Use these whenever user-supplied data is output on pages as they will strip out any bad characters.

- `esc_attr()`
- `esc_html()`
- `esc_js()`
- `esc_textarea()`
- `esc_url()`
- `esc_url_raw()`

In addition to those escaping functions, WordPress also offers some additional escaping functions based on `wp_kses()`⁶³ that can be used for page and post editing that will strip out unsafe HTML attributes and tags for users without the `unfiltered_html` capability.

- `wp_kses()`
- `wp_kses_post()`
- `wp_kses_data()`
- `wp_filter_post_kses()`
- `wp_filter_nohtml_kses()`

As always, custom sanitizing and escaping functions can be implemented. However, the native functions provided by WordPress make the process much simpler. If using custom

⁶² <https://developer.wordpress.org/plugins/security/securing-output/>

⁶³ https://developer.wordpress.org/reference/functions/wp_kses/

sanitizing and escaping, ensure that common scripting tags cannot be included in user-supplied inputs, unless absolutely necessary for the functionality of the plugin or theme. Also, ensure that disallowed HTML entities cannot be echoed to a page with being properly escaped.

Ensuring that data is properly sanitized and escaped will not only ensure that Cross-Site Scripting vulnerabilities are not introduced, but it will also ensure data is properly formatted for the functionality of plugins and themes.

Example from 301 Redirects – Easy Redirect Manager Plugin <= 2.4.0 - Stored Cross-Site Scripting Vulnerability ([CVE-2019-19915](https://www.cve.org/CVERetail?cveId=CVE-2019-19915))⁶⁴

The following is an example from the 301 Redirects – Easy Redirect Manager plugin where there was an access control vulnerability due to the use of `is_admin` used in conjunction with `wp_ajax` actions without any capability checks that results in both an arbitrary redirect vulnerability and stored Cross-Site Scripting vulnerability due to the use of unsanitized user-supplied input.

```
public function __construct()
{
    global $EPS_Redirects_Plugin;

    if (is_admin()) {

        if (isset($_GET['page']) && $_GET['page'] == $EPS_Redirects_Plugin->config( name: 'page_slug')) {
            // actions
            add_action('activated_plugin', array($this, 'activation_error'));
            add_action('admin_footer_text', array($this, 'set_ajax_url'));

            // Other
            add_action('admin_init', array($this, 'clear_cache'));
        }

        // Ajax funcs
        add_action('wp_ajax_eps_redirect_get_new_entry', array($this, 'ajax_get_entry'));
        add_action('wp_ajax_eps_redirect_delete_entry', array($this, 'ajax_eps_delete_entry'));
        add_action('wp_ajax_eps_redirect_get_inline_edit_entry', array($this, 'ajax_get_inline_edit_entry'));
        add_action('wp_ajax_eps_redirect_save', array($this, 'ajax_save_redirect'));
        add_filter('plugin_action_links_' . plugin_basename( file: __FILE__ ), array($this, 'plugin_action_links'));
```

⁶⁴<https://www.wordfence.com/blog/2019/12/critical-vulnerability-patched-in-301-redirects-easy-redirect-manager/>

Insecure Code Prior to Patch

The `ajax_save_redirect()` function was vulnerable to Stored Cross-Site Scripting due to the acceptance of user-supplied input with no sanitization from the `id` parameter that is later echoed to the page as the `redirect_id`.

```
public function ajax_save_redirect()
{
    $update = array(
        'id' => ($_POST['id']) ? $_POST['id'] : false,
        'url_from' => $_POST['url_from'], // remove the $root from the url if supplied, and a leading /
        'url_to' => $_POST['url_to'],
        'type' => (is_numeric($_POST['url_to']) ? 'post' : 'url'),
        'status' => $_POST['status']
    );

    $ids = self::_save_redirects(array($update));

    $updated_id = $ids[0]; // we expect only one returned id.

    // now get the new entry...
    $redirect = self::get_redirect($updated_id);
    $html = '';

    ob_start();
    $dfrom = urldecode($redirect->url_from);
    $dto = urldecode($redirect->url_to);
    include(EPS_REDIRECT_PATH . 'templates/template.redirect-entry.php');
    $html = ob_get_contents();
    ob_end_clean();
    echo json_encode(array(
        'html' => $html,
        'redirect_id' => $updated_id
    ));
}
```

Secured and Corrected Code Sample

The following shows the addition of the CSRF protection and capability check, along with a check to verify that the user-supplied `id` value is an integer with the `intval()` function, that validates if the value is an integer. This provides adequate protection against the XSS vulnerability.

```
public function ajax_save_redirect()
{
    check_ajax_referer( action: 'eps_301_save_redirect');

    if (!current_user_can( capability: 'manage_options')) {
        wp_die( message: 'You are not allowed to run this action.');
```

```
    }

    $update = array(
        'id' => ($_POST['id'] ? intval($_POST['id']) : false,
        'url_from' => esc_attr($_POST['url_from']), // remove the $root from the url if supplied, and a leading /
        'url_to' => esc_attr($_POST['url_to']),
        'type' => (is_numeric($_POST['url_to']) ? 'post' : 'url'),
        'status' => $_POST['status']
    );

    $ids = self::_save_redirects(array($update));

    $updated_id = $ids[0]; // we expect only one returned id.
```

VII. Unprepared SQL Queries

Nearly all of a WordPress site's data, including but not limited to site options, user passwords, post and page content, and plugin settings, are stored in a database, most commonly a MySQL database. All of the data stored in the database can be accessed using Structured Query Language, also known as SQL. In order to retrieve and update data from the database, SQL queries must be used. Therefore, it's common for developers to use SQL queries in plugins and themes to select, insert, update or delete data from the database as part of plugin or theme functionality.

SQL queries can be chained together using the UNION operator, and additional data can be retrieved from databases by using boolean statements, sleep functions, and even error-inducing statements. If user-supplied input to a SQL query is not properly sanitized, escaped, or prepared, then it can allow an attacker to either obtain additional data from the database beyond the originally anticipated data, or even manipulate the data stored within the database. This is referred to as a SQL Injection vulnerability. There are 4 core types of SQL injection vulnerabilities that are frequently found: boolean/blind-based SQLi, error-based SQLi, time-based SQLi, and UNION-based SQLi.

Boolean-Based SQLi, also commonly referred to as blind-based SQLi, vulnerabilities occur when an immediate observable response of data cannot be obtained while injecting additional SQL commands or queries, however, a response indicating that the results are true or false occurs. This type of SQLi vulnerability requires diligence and time as the injected SQL query will pull data from the database on character at a time based on whether or not the injected query/statement returns true or false.

Error-Based SQLi vulnerabilities occur again when there is no immediate observable response of data from an injected SQL query, however, an error message is returned that can allow a user to determine if information is present in a database. This is very similar to Boolean-based SQLi vulnerabilities, however, it relies on error messages instead of true or false type responses.

Time-Based SQLi vulnerabilities occur when there is no observable difference in the response of a SQL query, however, injected queries or statements can return a response within a specified delayed response that allows the user to obtain data from the database. As with blind SQLi vulnerabilities, this type of vulnerability requires patience and can only retrieve values one character at a time, unless a value is easily guessed.

UNION-based and Classic SQLi vulnerabilities fall within the same category and occur when SQL queries can be chained together to immediately retrieve data from a database. These are the easiest forms of SQL injection vulnerabilities to exploit and they make it the easiest to obtain data from a database. These can often be used to obtain complete dumps of information like all rows in the wp_users table for example.

Note that an additional form of SQL injection, known as a stacked query, which consists of ending a SQL statement with a semicolon (;) and beginning a new statement, is common in other web applications but is not a viable attack on the PHP/MySQL stack used by WordPress.

SQL Injection Can Lead to Sensitive Information Theft

Not properly preparing or escaping data supplied to SQL queries results in a vulnerability called SQL Injection. This vulnerability refers to the ability to inject SQL queries into already existing SQL queries. If a plugin or theme is vulnerable to SQL Injection then an attacker can obtain data like WordPress usernames and passwords, along with any other sensitive data stored in the database like a site's salts and keys. In some incredibly *rare* cases, where a plugin is extremely poorly coded or the MySQL server is improperly configured, an attacker could modify information in the database as well.

For that reason, it is important to ensure that WordPress plugins and themes have adequate protection to ensure a site doesn't become vulnerable to SQL Injection attacks that can result in sensitive information disclosure.

SQL Injection flaws are classified by MITRE as "CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')"⁶⁵ which means that the software has a flaw that allows this as the result of improper sanitization and validation on SQL queries that allow users to manipulate the existing SQL query to obtain/modify additional data in the database. This could also be identified by the OWASP's Top Ten as A1:2017-Injection⁶⁶.

Where are SQL Injection vulnerabilities found?

When validating that there are no SQL Injection vulnerabilities present in a plugin or theme, the following WordPress functions that do not inherently perform any preparation or sanitization should be looked for:

⁶⁵ <https://cwe.mitre.org/data/definitions/89.html>

⁶⁶ https://owasp.org/www-project-top-ten/2017/A1_2017-Injection

`$wpdb->get_results()`⁶⁷: This function is used to retrieve a set of results via a SQL query and is not limited to just one row of results.

`$wpdb->query()`⁶⁸: This function is similar to `get_results`, however, any SQL query can be run and it is not restricted to SELECT queries.

`$wpdb->get_row()`⁶⁹: This function will run a SQL query and will return the results of a database row.

`$wpdb->get_col()`⁷⁰: This function will run a SQL query and will return the results of a database column.

If user-supplied input is used to run the query, validate that the function is used alongside `$wpdb->prepare()`⁷¹. If it is, then it can be considered to be properly validated and secured, however, if the function isn't being prepared, then it is likely vulnerable to SQL injection.

In addition to looking for WordPress specific functions that can be vulnerable to SQL injection vulnerabilities, it is important to look for any other instances of SQL queries that may be implemented with custom functionality. This includes looking for various SQL statements like:

- DELETE
- UPDATE
- INSERT
- ORDER BY

If user-supplied input is used to run the query, then validate that the use of the SQL query is performed with the appropriate preparation function like `$wpdb->prepare()`⁷², `$wpdb->update()`⁷³, `$wpdb->insert()`⁷⁴, or `$wpdb->delete()`⁷⁵. If it is not, then it should be further tested to see if it is vulnerable to SQL Injection.

⁶⁷ https://developer.wordpress.org/reference/classes/wpdb/get_results/

⁶⁸ <https://developer.wordpress.org/reference/classes/wpdb/query/>

⁶⁹ https://developer.wordpress.org/reference/classes/wpdb/get_row/

⁷⁰ https://developer.wordpress.org/reference/classes/wpdb/get_col/

⁷¹ The WordPress `$wpdb->prepare` function will prepare a SQL query by sanitizing any user-supplied inputs prior to running the SQL query, preventing attackers from being able to inject additional SQL queries or statements.

⁷² <https://developer.wordpress.org/reference/classes/wpdb/prepare/>

⁷³ <https://developer.wordpress.org/reference/classes/wpdb/update/>

⁷⁴ <https://developer.wordpress.org/reference/classes/wpdb/insert/>

⁷⁵ <https://developer.wordpress.org/reference/classes/wpdb/delete/>

Securely Performing SQL Queries

There is a fairly simple solution to ensure SQL queries are performed securely and that is through the use of prepared statements. WordPress makes this simple with the `$wpdb->prepare()` function. This can be used on its own prior to running SQL queries or be used in conjunction with the other SQL query functions like `$wpdb->get_row()` and `$wpdb->get_col()`.

In addition to using prepared statements, developers can also implement escaping on any user-supplied values that may get passed to the database with WordPress native SQL escaping functions like `esc_sql()`⁷⁶, for generic escaping, and `sanitize_sql_orderby()`⁷⁷, for escaping when passing user supplied input to an ORDER BY statement in a query. When using SQL escaping, we still recommend using the `$wpdb->prepare()` function alongside as special methods can make it possible for attackers to bypass the escaping and still achieve a SQL Injection.

⁷⁶ https://developer.wordpress.org/reference/functions/esc_sql/

⁷⁷ https://developer.wordpress.org/reference/functions/sanitize_sql_orderby/

Example from Tutor LMS: Authenticated <= 1.8.2 - SQL Injection (CVE-2021-24183)⁷⁸

The following is an example of an authenticated SQL injection vulnerability that could be exploited using UNION-based methods from the Tutor LMS plugin. The AJAX action `wp_ajax_tutor_quiz_builder` hooked to the `tutor_quiz_builder_get_question_form` could be triggered by authenticated users, such as students, due to the fact that the function did not perform any nonce or capability checks.

```
add_action('wp_ajax_tutor_quiz_builder_get_question_form', array($this, 'tutor_quiz_builder_get_question_form'));
```

Insecure Code Prior to Patch

The `tutor_quiz_builder_get_question_form` function was vulnerable to union-based SQL injection attacks due to the use of `$wpdb->get_row` without any preparation or sanitization on the `$question_id` value that was obtained via user-supplied input.

```
public function tutor_quiz_builder_get_question_form(){
    global $wpdb;
    $quiz_id = sanitize_text_field($_POST['quiz_id']);
    $question_id = sanitize_text_field(tutor_utils()->avalue_dot( key: 'question_id', $_POST));

    if ( ! $question_id){
        $next_question_id = tutor_utils()->quiz_next_question_id();
        $next_question_order = tutor_utils()->quiz_next_question_order_id($quiz_id);

        $new_question_data = array(
            'quiz_id' => $quiz_id,
            'question_title' => __('Question', 'tutor')." ".$next_question_id,
            'question_description' => '',
            'question_type' => 'true_false',
            'question_mark' => 1,
            'question_settings' => maybe_serialize(array()),
            'question_order' => $next_question_order,
        );

        $wpdb->insert( table: $wpdb->prefix.'tutor_quiz_questions', $new_question_data);
        $question_id = $wpdb->insert_id;
    }

    $question = $wpdb->get_row( query: "SELECT * FROM {$wpdb->prefix}tutor_quiz_questions where question_id = {$question_id} " );

    ob_start();
    include tutor()->path.'views/modal/question_form.php';
    $output = ob_get_clean();

    wp_send_json_success(array('output' => $output));
}
```

⁷⁸ <https://www.wordfence.com/blog/2021/03/several-vulnerabilities-patched-in-tutor-lms-plugin/>

Secured and Corrected Code Sample

This vulnerability was corrected when the developer added the use of `$wpdb->prepare()` to the `$wpdb->get_row()` function which performs preparation of the SQL query and sanitizes the user-supplied values. Doing so protected against arbitrary SQL queries and commands. This was in conjunction with the addition of a nonce and capability check for unauthorized use protection.

```
public function tutor_quiz_builder_get_question_form(){
    tutor_utils()->checking_nonce();

    global $wpdb;
    $quiz_id = sanitize_text_field($_POST['quiz_id']);
    $question_id = sanitize_text_field(tutor_utils()->avalue_dot( key: 'question_id', $_POST));

    if(!tutor_utils()->can_user_manage('quiz', $quiz_id)) {
        wp_send_json_error( array('message'=>__('Access Denied', 'tutor')) );
    }

    if ( ! $question_id){
        $next_question_id = tutor_utils()->quiz_next_question_id();
        $next_question_order = tutor_utils()->quiz_next_question_order_id($quiz_id);

        $new_question_data = array(
            'quiz_id'           => $quiz_id,
            'question_title'    => __('Question', 'tutor'). ' '.$next_question_id,
            'question_description' => '',
            'question_type'     => 'true_false',
            'question_mark'     => 1,
            'question_settings' => maybe_serialize(array()),
            'question_order'    => esc_sql( $next_question_order ) ,
        );

        $wpdb->insert( table: $wpdb->prefix.'tutor_quiz_questions', $new_question_data);
        $question_id = $wpdb->insert_id;
    }

    $question = $wpdb->get_row($wpdb->prepare( query: "SELECT * FROM {$wpdb->prefix}tutor_quiz_questions where question_id = %d ", $question_id));

    ob_start();
    include tutor()->path.'views/modal/question_form.php';
    $output = ob_get_clean();

    wp_send_json_success(array('output' => $output));
}
```

VIII. Usage of Certain PHP Functions with User-Supplied Input

There are several PHP functions that can be used for remote command and code execution when using PHP. A PHP command function makes it possible to run commands like `ls`, to list a directory, `touch`, to make a new file, `mkdir`, to make a new directory, and many other commands.

Any command that could be run via a command line while connected to a server can be run by a PHP command function. On the other hand, there are other PHP functions that can make it possible to run arbitrary PHP code like `<?php phpinfo();?>` and these are commonly referred to as PHP code execution functions.

Sometimes WordPress plugins and themes will use these functions to run commands and code in order to perform tasks like making new directories, uploading files, adding custom PHP code snippets to a site, and more. It is actually fairly uncommon to find PHP command/code execution functions as the sole cause of a vulnerability in WordPress, however, when it does occur it can cause significant damage to a WordPress website. The biggest concern occurs when users can control the input that gets run through the command or code function without any filtering or sanitization performed on what is being supplied.

The Impact of Code & Command Injection

Arbitrary command and code execution results in the vulnerabilities identified as Remote Code Execution and Remote Command Execution. Though these types of vulnerabilities are significantly less common in WordPress than in other platforms, they are still critical issues.

Just like with arbitrary file upload vulnerabilities, PHP functions that can run commands and execute code leads to the ability to run arbitrary scripts. These can be used to perform endless actions on a WordPress site's server. If an attacker has access to run commands directly on a server then they can do things like create new files to add malicious code to a site, traverse to other directories and potentially infect other sites hosted in the same account, escalate privileges on the server if other operating system level vulnerabilities exist, and more.

Command Injection flaws are classified by MITRE as “CWE-78: Improper Neutralization of Special Elements used in an OS Command (‘OS Command Injection’)”⁷⁹ which means that the software has a flaw that allows command injection as a result of improperly sanitizing inputs.

Finding Remote Command/Code Injection Weaknesses

In order to prevent the introduction of remote command execution vulnerabilities, avoid the use of the following functions when user-supplied input will be passed to the function:

- `exec()`⁸⁰: A command execution function that has no output.
- `shell_exec()`⁸¹: A command execution function that has output.
- `proc_open()`⁸²: A command execution function that opens file pointers for input and output.
- `system()`⁸³: A command execution function that will return output.
- `passthru()`⁸⁴: A command execution function that will return raw output.
- `eval()`⁸⁵: A code execution function that will run strings of PHP as code.

When conducting secure code reviews, look for the above functions and validate that there is no user-supplied input that could be used to execute remote commands on the server. If user input can be supplied, conduct further analysis to determine the extent of which commands or code can be executed. The presence of any of the above functions along with user-supplied input may mean that a remote code execution or command execution vulnerability is present.

How can this be corrected?

The solution for this is simple. Don’t use PHP command functions with user-supplied inputs unless absolutely necessary. If any user-supplied input needs to be passed to the function, then validate that the user-supplied input is completely sanitized and filtered so that arbitrary commands and code cannot be supplied.

⁷⁹ <https://cwe.mitre.org/data/definitions/78.html>

⁸⁰ <https://www.php.net/manual/en/function.exec.php>

⁸¹ <https://www.php.net/manual/en/function.shell-exec>

⁸² <https://www.php.net/manual/en/function.proc-open>

⁸³ <https://www.php.net/manual/en/function.system>

⁸⁴ <https://www.php.net/manual/en/function.passthru>

⁸⁵ <https://www.php.net/manual/en/function.eval>

In some cases plugins and themes add the ability to implement custom PHP code on a site. If the functionality of the plugin or theme's sole purpose is to add custom PHP code to a WordPress site and no filtering can be applied, validate that the appropriate access controls and Cross-Site Request Forgery protections are in place to prevent unauthorized access from lower-privileged users.

Example from Social Warfare Plugin <= 3.5.2 Unauthenticated Remote Command Execution⁸⁶

The following is an example of how a PHP code execution function, `eval()`, was improperly used by allowing user-supplied input from a remote hosted file. This plugin had an `admin_post` action hook, tied to a function that would retrieve the options for a site. If the `swp_url` parameter was supplied with a path to a file, then this function would retrieve the contents of that file and anything in the `<pre></pre>` tags would be run through the `eval()` function making it possible for users to supplied arbitrary PHP code like `phpinfo()` creating a remote code execution vulnerability.

Insecure Code Prior to Patch

```
if ( true == SWP_Utility::debug( key: 'load_options' ) ) {
    if (!is_admin()) {
        wp_die( message: 'You do not have authorization to view this page.' );
    }

    $options = file_get_contents( filename: $_GET['swp_url'] . '?swp_debug=get_user_options' );

    /* Bad url.
    if (!$options) {
        wp_die( message: 'nothing found' );
    }

    $pre = strpos($options, needle: '<pre>');
    if ($pre != 0) {
        wp_die( message: 'No Social Warfare found.' );
    }

    $options = str_replace( search: '<pre>', replace: '', $options );
    $cutoff = strpos($options, needle: '</pre>');
    $options = substr($options, start: 0, $cutoff);

    $array = 'return ' . $options . ';';

    try {
        $fetched_options = eval( $array );
    }
}
```

This vulnerability was corrected by completely removing this functionality from the plugin, so we have not provided an example of the secured and corrected code.

⁸⁶<https://www.wordfence.com/blog/2019/03/recent-social-warfare-vulnerability-allowed-remote-code-execution/>

IX. Sensitive Information Disclosure

Sensitive information can range from personally identifiable information to nonces that lower-privileged individuals should not have access to. In many cases, access to sensitive information can lead to improper access control vulnerabilities. However, it is important that these types of weaknesses have their own category of discussion.

In the past, our team has found several instances where nonces intended for only administrative or higher privileged access have been disclosed to lower-level users like subscribers or those even without authentication. A nonce should be considered sensitive information as the inadvertent disclosure of such can make it possible for lower-level users to perform unauthorized actions when proper access controls are missing.

In addition to nonce disclosure, there are more classic examples of sensitive information disclosure that should be looked for and protected against, including user information disclosure, sensitive access keys disclosure, log disclosure, and more.

Sensitive information disclosure flaws are classified by MITRE as “CWE-200: Exposure of Sensitive Information to an Unauthorized Actor.”⁸⁷This could also be identified by the OWASP’s Top Ten as A3:2017-Sensitive Data Exposure⁸⁸.

Uncovering Sensitive Information Disclosure Vulnerabilities

When looking for sensitive information disclosure vulnerabilities, we highly recommend looking for any functions that can add information to the source code of a page, especially in the /wp-admin area, which can be accessed by any authenticated user. This includes looking for the following hooks:

- `admin_footer`⁸⁹: A hook that will add data or scripts to the footer of an administrative page.
- `admin_head`⁹⁰: A hook that will add data or scripts to the header of an administrative page.
- `wp_footer`⁹¹: A hook that will add data or scripts to the footer of any page.
- `wp_head`⁹²: A hook that will add data or scripts to the header of any page.

⁸⁷ <https://cwe.mitre.org/data/definitions/200.html>

⁸⁸ https://owasp.org/www-project-top-ten/2017/A3_2017-Sensitive_Data_Exposure

⁸⁹ https://developer.wordpress.org/reference/hooks/admin_footer/

⁹⁰ https://developer.wordpress.org/reference/hooks/admin_head/

⁹¹ https://developer.wordpress.org/reference/functions/wp_footer/

⁹² https://developer.wordpress.org/reference/functions/wp_head/

- `wp_enqueue_script`⁹³: A hook that will add scripts to any page and enqueue it.
- `admin_enqueue_script`⁹⁴: A hook that will add scripts to any administrative page and enqueue it.
- `login_enqueue_script`⁹⁵: A hook that will add scripts to the login page and enqueue it.

When looking through these hooks, verify that the associated function does not contain any sensitive information like full paths to a site, nonces not intended to be seen by low-privilege users, personally identifiable information of other users, data from logs, sensitive access keys, or any other information that may be deemed sensitive. If this data is present, then the function should be validated to ensure that there is an appropriate capability check in place so that the sensitive information is only disclosed to users with the appropriate level of access.

In addition to looking for any hooks that can disclose sensitive information in the source code of a page, we recommend looking for export functionalities typically hooked to actions like `wp_ajax`, `admin_init`, and `admin_action`. When plugins or themes export data they frequently contain some sort of information from the target site and this information has the potential to be sensitive. Some examples include log file information, which can be used to take over a site if verbose email logs can be exported, email subscriber information, which can contain personally identifiable information about individuals, and site information like a site's full path, which can be used to exploit other vulnerabilities. When export functionality is present in a plugin or theme that exports any form of sensitive data, then it should be validated that the appropriate access controls are in place to prevent unauthorized users from exporting any information.

Preventing Sensitive Information Disclosure

Due to the fact that sensitive information disclosure vulnerabilities are typically the result of improper access control, these vulnerabilities can usually be remediated fairly easily by just implementing a capability check on the function that contains sensitive information. This can be done with the use of the `current_user_can()` function that checks for the appropriate capability.

⁹³ https://developer.wordpress.org/reference/functions/wp_enqueue_script/

⁹⁴ https://developer.wordpress.org/reference/hooks/admin_enqueue_scripts/

⁹⁵ https://developer.wordpress.org/reference/hooks/login_enqueue_scripts/

Example from WPCentral Plugin <= 1.5.0 - Connection Key Disclosure (CVE-2020-9043)⁹⁶

The following is an example of sensitive information disclosure from the WPCentral plugin. This plugin had a flaw that displayed a site's connection key in the source code of the /wp-admin dashboard to any user that was authenticated to the site. The plugin registered the `admin_footer` hook tied to the `wpc_modal_dialog` function which added information to the administrative footer, including the site's connection key. This connection key could then be used to log in as the site's administrator.

```
//Add link to show the Connection key once the plugin is activated
if(is_plugin_active($wpc_slug)){
    add_action('admin_init', 'wpc_enqueue_script');
    add_action('admin_init', 'wpc_enqueue_styles');
    add_filter('plugin_row_meta', 'wpc_add_connection_link', 10, 2);
    add_action('admin_head', 'wpc_modal_open_script');
    add_action('admin_footer', 'wpc_modal_dialog');

    add_action('plugins_loaded', 'wpc_load_plugin');

    include_once(ABSPATH.'wp-includes/pluggable.php');
```

Insecure Code Prior to Patch

The following is the `wpc_modal_dialog` function where the connection key is retrieved from the database and added to the HTML that is displayed in the wp-admin footer. There is no capability check on this function to validate that the user is an administrator who should have access to the site's connection key.

```
function wpc_modal_dialog(){
    $dialog = '
<div id="wpc_connection_key_dialog" style="...">
<p>Follow the steps here to connect your website to wpcentral dashboard:</p>
<ol>
<li>Copy the connection key below</li>
<li>Log into your <a href="https://panel.wpcentral.co/" target="_blank">wpcentral</a> account</li>
<li>Click on Add website to add your website to wpcentral.</li>
<li>Enter this website's URL and paste the Connection key given below.</li>
<li>You can also follow our guide for the same <a href="https://wpcentral.co/docs/getting-started/adding-website-in-wpcentral/" target="_blank">here</a>.</li>
</ol>

<p style="...">Note: Contact wpCentral Team at support@wpcentral.co for any issues</p>

<div style="..."><p style="...">wpCentral Connection Key</p></div>
<div style="...">_wpc_get_connection_key().</div>
</div>';

    print($dialog);
}
```

⁹⁶ <https://www.wordfence.com/blog/2020/02/vulnerability-in-wpcentral-plugin-leads-to-privilege-escalation/>

Secured and Corrected Code Sample

The following is how the developer corrected the flaw. The `wpc_modal_dialog` function was completely removed and replaced with the `wpc_add_connection_link` function that performed an authorization check to validate that a user had the `activate_plugins` capability, which is provisioned to administrators only, prior to displaying the scripts and data to the administrative dashboard.

```
function wpc_add_connection_link($links, $slug) {  
  
    if(is_multisite() && is_network_admin()){  
        return $links;  
    }  
  
    if ($slug !== WPCENTRAL_BASE) {  
        return $links;  
    }  
  
    if(!current_user_can( capability: 'activate_plugins')){  
        return $links;  
    }  
  
    wp_enqueue_script( handle: 'jquery');  
    wp_enqueue_script( handle: 'jquery-ui-core');  
    wp_enqueue_script( handle: 'jquery-ui-dialog');  
    wp_enqueue_style( handle: 'wp-jquery-ui');  
    wp_enqueue_style( handle: 'wp-jquery-ui-dialog');
```

X. File Access/Usage Weaknesses

File access/usage weaknesses can make it possible for attackers to read or delete arbitrary files on a server in addition to performing local or remote file inclusion in order to trigger code execution. WordPress plugins and themes may include files for a variety of use cases, such as using templates or loading settings pages in the dashboard. In addition, several plugins and themes make it possible to read files for the purpose of editing files or creating backups.

The insecure use of these capabilities can lead to remote file inclusion or local file inclusion vulnerabilities in addition to directory traversal and arbitrary file modification, reading, or deletion vulnerabilities. As such, it is incredibly important to ensure that these vulnerabilities are not introduced into plugins and themes and that when they are, they are detected rapidly to be remediated.

Accessing, Including, and Modifying Files Can Do a Lot of Harm

File access and inclusion vulnerabilities can be just as dangerous as many of the vulnerabilities previously discussed. Including local and remote files can result in vulnerabilities identified as local or remote file include if the function is not properly secured. When a file containing PHP content can be included via PHP, then that file becomes executable and an attacker can trigger code and include a webshell that will allow them to remotely run commands on a server.

Local file inclusion vulnerabilities may require the attacker to upload a file to the vulnerable site, however, the file is not required to have a .php extension to be executable and can be any file type like a .jpg or .txt file that will easily pass any extension checking. In addition, an attacker can send a request that updates a site's log file with PHP, and then include that log file to achieve remote code execution without ever requiring a file to be uploaded.

Remote file include vulnerabilities are a little more severe as they don't require any file to be uploaded to the server, but rather a remotely hosted file can be included and the code will still execute, making it incredibly easy for an attacker to infect a victim.

When it comes to arbitrary file deletion vulnerabilities, an attacker can delete a wp-config.php file on a target site which effectively transforms the site into a brand new installation. Once that occurs an attacker can set up the site as if it was a new installation and connect the site to their own database, setting new credentials for a new user account. After the attacker has completed the setup process, they can then access the

site's dashboard and upload malicious files that can be used to run commands on the server and further infect the victim or any other sites hosted in the victim's account.

For arbitrary file-read vulnerabilities, which typically occur as a result of directory traversal vulnerabilities, an attacker can potentially read sensitive files to steal information. Of greatest concern would be inadvertent read access of the wp-config.php file that contains a site's unique salts and keys along with database credentials. If the database of a vulnerable site is publicly accessible, an attacker could use the credentials to login and perform a wide variety of malicious actions such as adding a new administrative user account to further infect the site or stealing sensitive information from users' accounts, like their password hashes.

All of the vulnerabilities caused by accessing, including, or modifying files can cause serious harm to a WordPress site..

Remote File Include flaws are classified by MITRE as "CWE-98: Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')."⁹⁷

Directory traversal flaws are classified by MITRE as "CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')."⁹⁸

File Deletion and Modification flaws are classified by MITRE as "CWE-73: External Control of File Name or Path."⁹⁹

Discovering File Access/Usage Weaknesses

When looking for these types of vulnerabilities, examine any instances where PHP functions are used to include files, retrieve file contents, or delete files. Also look for PHP functions that are being used to accept user-supplied inputs. If any of the following functions accept user-supplied values, then there is the possibility that a file usage or access weakness is present.

Local and Remote File Includes:

- `include_once()`¹⁰⁰
- `include()`¹⁰¹

⁹⁷ <https://cwe.mitre.org/data/definitions/98.html>

⁹⁸ <https://cwe.mitre.org/data/definitions/22.html>

⁹⁹ <https://cwe.mitre.org/data/definitions/73.html>

¹⁰⁰ <https://www.php.net/manual/en/function.include-once.php>

¹⁰¹ <https://www.php.net/manual/en/function.include.php>

- `require_once()`¹⁰²
- `require()`¹⁰³

Arbitrary File Reading:

- `file_get_contents()`¹⁰⁴
- `fread()`¹⁰⁵
- `fopen()`¹⁰⁶

Arbitrary File Deletion:

- `wp_delete_file()`¹⁰⁷
- `unlink()`¹⁰⁸

Securely Performing File Operations

To securely perform file operations, user-supplied inputs should be limited and hard coded paths to files should be used. The path to the WordPress uploads directory can be obtained using the `wp_upload_dir()`¹⁰⁹ function, and it is recommended to use this as the prefix when performing file operations on any file that is uploaded to the `/wp-content/uploads/` directory so that users cannot supply arbitrary paths to files.

In addition, the use of characters that can make it possible to perform a directory traversal attack to read, modify, or include files should be sanitized out. This can be done with the `sanitize_file_name()` function which strips out a variety of special characters that could aid in exploiting file access/include attacks.

¹⁰² <https://www.php.net/manual/en/function.require-once.php>

¹⁰³ <https://www.php.net/manual/en/function.require.php>

¹⁰⁴ <https://www.php.net/manual/en/function.file-get-contents>

¹⁰⁵ <https://www.php.net/manual/en/function.fread>

¹⁰⁶ <https://www.php.net/manual/en/function.fopen>

¹⁰⁷ https://developer.wordpress.org/reference/functions/wp_delete_file/

¹⁰⁸ <https://www.php.net/manual/en/function.unlink>

¹⁰⁹ https://developer.wordpress.org/reference/functions/wp_upload_dir/

Example from Quiz and Survey Master Plugin <= 7.0.0 - Arbitrary File Deletion([CVE-2020-35951](https://www.wordfence.com/blog/2020/08/critical-vulnerabilities-patched-in-quiz-and-survey-master-plugin/))¹¹⁰

The following is an example of an arbitrary file deletion vulnerability from the Quiz and Survey Master Plugin. The plugin registered an AJAX action `wp_ajax_qsm_remove_file_fd_question` hooked to the `qsm_remove_file_fd_question` function intended to allow the deletion of any files uploaded through a quiz or form from the plugin. This functionality was intended for both authenticated and unauthenticated users which is why the plugin also registered a `nopriv` AJAX hook.

```
//remove file of file upload question type
add_action('wp_ajax_qsm_remove_file_fd_question', array($this, 'qsm_remove_file_fd_question'));
add_action('wp_ajax_nopriv_qsm_remove_file_fd_question', array($this, 'qsm_remove_file_fd_question'));
```

Insecure Code Prior to Patch

Unfortunately, the `qsm_remove_file_fd_question` function was insecurely implemented and allowed a user to supply a full path to a file as the `file_url` and performed no validation that the supplied file was one that was uploaded by the plugin's upload functionality. This meant that any user could supply a full path to any file, including the `wp-config.php` file, and delete the file through the `wp_delete_file` function.

```
public function qsm_remove_file_fd_question(){
    $file_url = isset($_POST['file_url']) ? sanitize_text_field($_POST['file_url']) : '';
    if($file_url){
        wp_delete_file($file_url);
        $json['type'] = 'success';
        $json['message'] = 'File removed successfully';
        echo json_encode($json);
        exit;
    }
    $json['type'] = 'error';
    $json['message'] = 'File not removed';
    echo json_encode($json);
    exit;
}
```

¹¹⁰<https://www.wordfence.com/blog/2020/08/critical-vulnerabilities-patched-in-quiz-and-survey-master-plugin/>

Secured and Corrected Code Sample

This was corrected by implementing custom file names on all uploads to ensure they contained the string `qsmfileupload_` and required that this string be at the beginning of the filename to ensure that only files previously uploaded from a quiz form could be deleted.

```
public function qsm_remove_file_fd_question(){
    $file_url = isset($_POST['file_url']) ? sanitize_text_field($_POST['file_url']) : '';
    $upload_dir = wp_upload_dir();
    $uploaded_path = $upload_dir['path'];
    if($file_url && stripos( $file_url, 'qsmfileupload_' ) && file_exists( filename: $uploaded_path . '/' . $file_url ) ){
        $attachment_url = $upload_dir['url'] . '/' . $file_url;
        $attachment_id = $this->qsm_get_attachment_id_from_url($attachment_url);
        wp_delete_file( file: $uploaded_path . '/' . $file_url );
        wp_delete_attachment( $attachment_id );
        $json['type'] = 'success';
        $json['message'] = __( 'File removed successfully', 'quiz-master-next' );
        echo json_encode($json);
        exit;
    }
    $json['type'] = 'error';
    $json['message'] = __( 'File not removed', 'quiz-master-next' );
    echo json_encode($json);
    exit;
}
```

XI. Additional Security Flaws

In addition to the most common coding flaws we see, there are several other vulnerabilities that are less commonly encountered. However, these flaws still pose a significant risk to WordPress sites. This section explores those vulnerabilities.

Arbitrary Options and user_meta Update Vulnerabilities

Arbitrary options update and arbitrary user_meta update vulnerabilities are typically the result of improper access control on functions, in combination with a lack of filtering on options being updated. These vulnerabilities are frequently used to escalate privileges on sites with vulnerable plugins and themes.

Arbitrary options update vulnerabilities make it possible for attackers to update the site option allowing user registration to be enabled and set the default role to administrator. This allows attackers to register on vulnerable sites as an administrator. On the other hand, arbitrary user_meta updates make it possible for an attacker with an account on a vulnerable site to escalate their privileges by updating the `wp_capabilities` meta key, which controls a user's role and capabilities, to the value of "administrator". In both cases, once an attacker has access to a site with administrative capabilities, they can completely take over the site and cause significant harm.

When looking for arbitrary options update vulnerabilities, any instance where the `update_option()` function is used where user input is supplied should be analyzed. If that is found then it should be verified that the function validates and sanitizes the input so that arbitrary options cannot be supplied. As a developer, ensure that options that can be updated via plugin or theme functionality are consistent with the options available within the code. Also ensure user input can not be supplied unless absolutely necessary. For example, if the plugin or theme intends to update an option that the plugin or theme introduces, then make sure that the `update_option()` function can only update that value.

When looking for arbitrary user_meta update vulnerabilities, any instance where `update_user_meta()` is used alongside user-supplied input should be further analyzed. If an instance of that function is found that accepts user-supplied input, then it should be validated that arbitrary metadata cannot be supplied, and if it can, further analysis should be done to see if there is an exploitable privilege escalation vulnerability. As a developer, it should be ensured that the user metadata that can be updated via functionality of plugins or themes is consistent with the user metadata in the code that should be updated. For example, if the plugin or theme intends to update a custom

user_meta field that the plugin or theme introduces, then make sure that the `update_user_meta()` function can only update that value.

Open Redirection Vulnerabilities

Open redirection vulnerabilities are the result of improper validation of a redirect location. This can result in site visitors and owners being redirected to external malicious sites that can steal sensitive information from a victim's browser or further infect a computer. This type of vulnerability is typically introduced in WordPress plugins and themes when the `wp_redirect()` function is used for redirection in conjunction with user-supplied values that determine the redirect location. This flaw can easily be corrected by switching to using the `wp_safe_redirect()` function which will validate that the redirect occurs locally. If the functionality of the plugin or theme requires a redirect to an externally hosted site, then it is recommended to use a hardcoded URL that can not be modified by a request.

Conclusion

This document has explored some of the most common coding security issues the Wordfence Threat Intelligence team finds when conducting secure code reviews on plugins and themes. There are numerous ways that vulnerabilities can be introduced, therefore, it is important to understand how they are introduced and what can be done to prevent these security issues.

As outlined, a vast majority of critical security issues introduced in WordPress are a result of insufficient access controls combined with additional weaknesses. An arbitrary options update vulnerability becomes significantly more harmful if an unauthenticated user with insufficient privileges is able to exploit it.

As a reminder, if you are a plugin or theme developer in the WordPress space, over 42% of the internet relies on you to make good security choices when creating your plugin or theme. As such, security is a critical part of your brand as well as the safety of the WordPress community. We hope this white paper helps you to avoid common mistakes when developing your plugin or theme, or with reviewing your plugin or theme once development has concluded. Better yet, have a third party security minded developer or security professional review the code to find any security flaws you might have missed.

We are all human, and you will likely introduce a security vulnerability in your software at some point. If you make your best effort to avoid it, then you will have better trust from your users and a more secure product in the end.

If you are a WordPress user, with little code development background, you can still get a good idea about a plugin or theme's security by just looking for a few things here and there, like sanitization on `$_POST` variables, for example. If you notice several missing capability checks or no CSRF protection on several endpoints, then you may want to reconsider and investigate a different plugin or theme for your WordPress site. Alternatively, you can reach out to a security researcher to conduct a more thorough review and coordinate fixes with the developer.

If you are a security researcher, we hope that this guide will assist you in finding some of the most common security related coding flaws in the WordPress ecosystem and allow you to coordinate with developers on recommended fixes for these flaws.

Please note that there are other security flaws in code that we may not be outlined in this document, this list just includes some of the most common and the most significant flaws our team frequently sees.