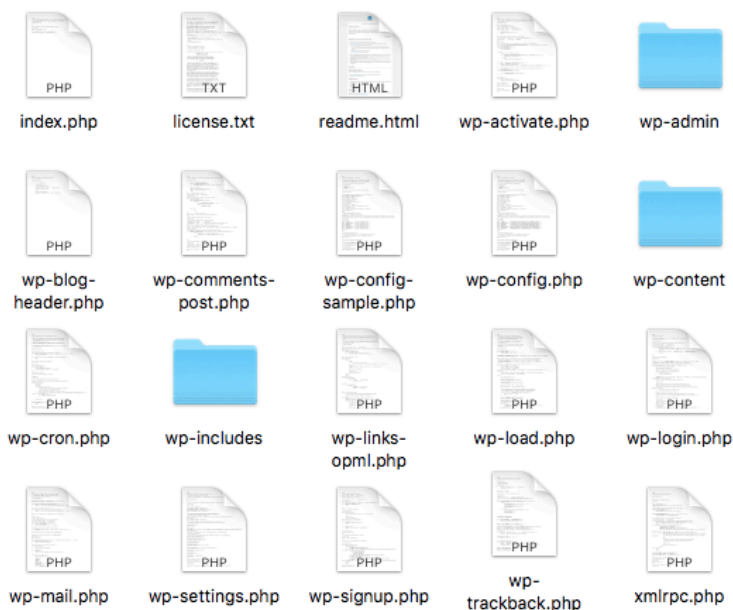


Allan Porter

I. WordPress.org Introduction

WordPress is a content management system that is written using PHP and uses MySQL, a relational database management system for its working with databases. It is currently one of the most popular content management systems out there and accounts for an astounding 30% of total websites. It provides users with a dashboard and user-friendly interface that allows them to handle their websites structure and data without prior knowledge of programming. It also allows for flexibility as users can load their own “themes” as templates into it without interfering with the core programming, as well as their own “plugins” for applications.

II. WordPress Code Structure



(The above image shows the set of files inside the WordPress directory.)

WordPress Configuration files:

(.htaccess: This file is used to alter the configuration of Apache Web servers (For those using Apache servers). For example it is here that one can add the functionality of leveraging browser caching if necessary by adding code to this file.)

wp-config.php: This is perhaps one of the most important files and is generated during the software's installation process. It contains database information (For example- DB Name, DB Username, DB password) and also the databases connection settings. It also contains security keys, which are variables that help encrypt the info that is stored inside cookies and help security in general. The keys are AUTH_KEY, SECURE_AUTH_KEY, NONCE_KEY and LOGGED_IN_KEY. These keys are also accompanied by salts (data added to passwords pre hashing), which help improve security and make these hashes harder to crack. It also contains the 'WP-debug' setting which allows for notifications to be displayed during development.

```

define( 'AUTH_KEY',          'zEan><4@5N+Gm$ W4#tT}$ZuUD]Hhv;|UT!Dus 9A`z,qD#|DWB`>r70/]LSKfJY' );
define( 'SECURE_AUTH_KEY',   'U>YnXDvsbRmy})yy{~$3y>cmtL;s}[W1!GPmX(6sjFWx5aiza>wV2MN=v,ucAtZf' );
define( 'LOGGED_IN_KEY',     'cF]e:&YB%g,qe%?(T}ReV@,x9HRgCK>Qes8CM8`9b-#+h2P9CB:$2-CyNrf] |&7' );
define( 'NONCE_KEY',        '`}D4xpU>2C=<@|oap^(i&u,r(f 79Yyx>DYjuk}JjZM$0j]KebTX88!G:vj+=giy' );
define( 'AUTH_SALT',         'kD}lf7~^Xt^c(Qp.Klv}-.bd),,ivkf%C2( 3T}bk:B:>BKhIxuH*ax+KT42l1=' );
define( 'SECURE_AUTH_SALT',  'y,)D?eg;Av(UZaOU7/302~TR5(>D82Tn#wamF2>lzsk}Y:q{({{SmHf,*,h0%|lC' );
define( 'LOGGED_IN_SALT',    '94VMQT1WLSKZPn4J]]zKc:Ga^.vt_+DbAJRB;dA5r3W-gU~+mmak%DW$6jH=0975' );
define( 'NONCE_SALT',       'qBy}v66KG;bXf2H`lXgf:lt~UJx0-mtUa7`.&4Gq:~pF_L />` i!^IpX<7UJ]@0' );

```

wp-config-sample.php: This file is used by the wp-config.php file and helps configure MySQL settings, secret keys, the database table prefix and absolute path.

Important Directories:

wp-admin Directory: This provides the functionality for the site administrator's capabilities and tools.

wp-content Directory: This directory stores the themes (templates), plugins (applications) and uploads (images, textfiles, audio files, videos) necessary for the site.

wp-includes Directory: This directory contains about 140 files each serving different roles. Some of these include links.php which manages links, cache.php which deals with managing the cache, widgets.php which allows for the functionality of default widgets, certificates, fonts, js and more. One of the most important of these files is the functions.php file which contains all of the functions and classes that can be used in developing themes.

Other files:

index.php: This loads the wp-blog-header.php which helps load the theme.

wp-blog-header.php: This loads the WordPress library, theme template and sets up the WordPress query.

wp-comment-post.php: This helps handle comment posting functionality and prevents duplicate comment postings.

wp-cron.php: This handles scheduled events.

wp-mail.php: This handles email functionality.

wp-settings.php: This is used to set or change common variables.

wp-signup.php: This handles a site's signup functionality.

wp-login.php: This handles a site's login functionality.

wp-load.php: This loads crucial files such as function.php and wp-settings.php.

wp-trackback.php: This contains the code that handles trackbacks and ping backs.

wp-activate.php: This code confirms the user activation key, after the registration of a user.

xmlrpc.php: This handles feature of WordPress that allows transmission of data using HTTP for transport and XML for encoding.

license.txt: This contains copyright and licensing information.

readme.html: Contains the ReadMe documentation.

III. How WordPress handles Cookies

When a user logs in to WordPress, 2 cookies are initially created as means to store information: wordpress[hash] and wordpress_logged_in[hash]. For login, wordpress[hash] is used to store the authentication details of the user including username and password. The function wordpress_logged_in[hash] determines whether or not the user is logged in and contains information about the user (username, password). Both of these hash all the users information, which makes them more difficult to decode. This helps security and makes it more difficult for malicious users to attain user information from cookies.

WordPress also uses the cookies WordPress_test_cookie and WP-settings-time-{\$uid}.

IV. How WordPress handles Password Generation and Hashing

The code for generating and hashing passwords in WordPress is contained in “pluggable.php”. WordPress has its own random function “wp_rand” that it uses to generate random numbers within a range. Their reason for using their own function is that it’s less predictable than PHP’s built in random functions. WordPress uses this “wp_rand” function in its password generation function to generate passwords.

For hashing passwords, WordPress uses two different methods. Their old way of hashing passwords was simply with a MD5 hash. In this method, the password hash is just the md5 hash of the password which is now widely considered insecure. Their newer method of hashing is by the utilization of the PHPass library. By default, PHPass will run the password through 8 iterations of MD5 hashing to generate the hash. However, it can also be changed so that PHPass instead uses CRYPT_BLOWFISH bcrypt to generate the hash by changing the constructor, which would be a positive security implementation. It’s also possible to replace the default password hashing solutions using plugins which adds flexibility and could help users customize their own hashing preferences.

The following is a little more information regarding PHPass:

- If the PHP version is ≥ 5.3 , then portable mode can be turned off because PHP has its own version of bcrypt, so hashes will be portable across servers.

See <https://stackoverflow.com/a/5343655>.

If the PHP version is ≥ 5.5 , the creators of PHPass recommend to use <https://php.net/manual/en/function.password-hash.php> which is the PHP built in password hashing function.

- PHP Website located here: <https://www.openwall.com/phpass/>

WordPress Hashing Conclusions:

MD5 and PHPass are the default solutions for password hashing in WordPress, but PHPass can be replaced by the built-in password hashing algorithm in newer versions of PHP. WordPress most likely uses both these methods of hashing in order to allow for backwards compatibility (just like how it uses md5 has a fallback). Overall, despite its fairly dangerous use of MD5,

WordPress does allow its users to customize the hashing method via multiple plugins which does allow for added flexibility.

V. Fuzzing Tests

The goal of using fuzzing here was to check if WordPress defends against well-known fuzzing attack vectors.

Project Goals

We used the search, comment, reply to comment, and login features in WordPress as potential fuzzing targets. A successful attack would consist of the server going down after a vector was used on a target. We also conducted a manual check for successful XSS attack by checking if a JavaScript alert appears when looking at a page with a comment or reply). As a convenience, we approved all comments and replies that we sent. However, in real life usage, we imagine that messages that look like spam or an attack would likely not be approved.

Results

None of the attack vectors managed to take down the server. Additionally, none of the cross-site scripting attacks had any noticeable effects (No alerts appeared). Attempts to utilize buffer overflow vectors also failed as the server send us 414 URI Too Long response status code.

Possible Areas of Further Research


The usage and results of the fuzzers here show that there are no long term effects of these common fuzzer attack vectors. The site didn't go down, and XSS didn't have any effects. Further investigation here could include if there were any short-term effects. Additionally, did the SQL injection attempts leak any information? What about the XML injection attacks or the LDAP injection? These can be manually looked into if not automated.

All fuzzing information including attack vectors and results for this project can be found here: <https://github.com/PotatoHashing/WordPress-Fuzzer>.

V. Security Testing the Software:

One of the first things I did in testing the WordPress core software was to run the software through the OWASP ZAP scanner which detects security vulnerabilities in applications. I ran ZAP on the WordPress implemented software locally and found a total of 5 alerts indicating potential security vulnerabilities with WordPress's core software. They were as follows:

ZAP Alerts:

X-Frame-Options Header Not Set
URL: http://localhost:8888/wordpress
Risk:  Medium
Confidence: Medium
Parameter: X-Frame-Options
Attack:
Evidence:
CWE ID: 16
WASC ID: 15
Source: Passive (10020 - X-Frame-Options Header Scanner)

Description:
X-Frame-Options header is not included in the HTTP response to protect against 'ClickJacking' attacks.

Other Info:

Solution:
Most modern Web browsers support the X-Frame-Options HTTP header. Ensure it's set on all web pages returned by your site (if you expect the page to be framed only by pages on your server (e.g. it's part of a FRAMESET) then you'll want to use SAMEORIGIN, otherwise if you never expect the page to be framed, you should use DENY. ALLOW-FROM allows specific websites to frame the web page in supported web browsers).

Reference:
<http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx>

1. **X-Frame-Options Header Not Set (11 instances)**

The Potential Danger:

Having X-Frame-Options headers not set, can make the application vulnerable to a clickjacking attack as attackers can place layered iframes from other sites beneath clickable components of a webpage to trick the user into unknowingly clicking on them.

Is this concerning?

WordPress likely doesn't include the frame headers always set by default as the developers want to allow as much flexibility as possible to users in later building upon the WordPress de-

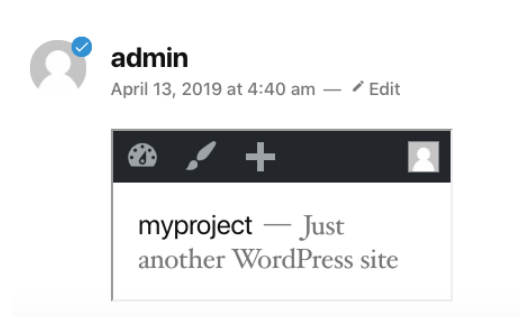
fault software. For example, if users would like to include iframes from other sites on some of their webpages, they would be confused when that functionality wouldn't be allowed by default. (This would violate the design principle of Least Astonishment). Though this does represent an added security vulnerability, it seems that the WordPress developers are choosing flexibility over total security and counting on the users to install themes or plugins on top of the core software to make their sites totally secure.

A Possible Solution:


A possible solution for users of WordPress would be to install a secure plugin that automatically sets this option and other security headers to secure defaults such as the plugin- Security Headers by Simon Waters. They could also edit the configuration htaccess file, and set the X-Frame-Options headers to either DENY (if they expect frames to never be used, SAME-ORIGIN (if they expect frames to be used from same origin) or ALLOW FROM (if they expect frames from only specific websites).

Trying this out:

I myself went and tried to place an iframe inside the comment field on a WordPress software post. As an admin, I submitted the comment- `<iframe src="yahoo.com" name="iframe_a"></iframe>` and it posted successfully and appeared as this:



That being said, when I tried doing this as a mere user of the site (not an admin), the iframe tags were properly escaped.

Cookie No HttpOnly Flag	
URL:	http://localhost:8888/wordpress/wp-login.php?reauth=1&redirect_to=http%3A%2F%2Flocalhost%3A8888%2Fwordpress%2Fwp-admin%2F
Risk:	 Low
Confidence:	Medium
Parameter:	wordpresspass_2b7738476b2cfaf3b5454b1e89821e63
Attack:	
Evidence:	Set-Cookie: wordpresspass_2b7738476b2cfaf3b5454b1e89821e63
CWE ID:	16
WASC ID:	13
Source:	Passive (10010 – Cookie No HttpOnly Flag)
Description: A cookie has been set without the HttpOnly flag, which means that the cookie can be accessed by JavaScript. If a malicious script can be run on this page then the cookie will be accessible and can be transmitted to another site. If this is a session cookie then session hijacking may be possible.	
Other Info:	
Solution: Ensure that the HttpOnly flag is set for all cookies.	
Reference: http://www.owasp.org/index.php/HttpOnly	

2. Cookie No HttpOnly Flag (13 instances)

The Potential Danger:

If Cookies are set without the HttpOnly flag, then cookies could possibly be accessible by JavaScript (as well as HTML) and therefore stolen by an attacker upon the running of a malicious script. This can possibly lead to session hijacking if there was session information inside the cookie and also other vulnerabilities.

Is this concerning?

I found a pretty formidable explanation as to why WordPress by default doesn't set certain HttpOnly flags on certain cookies (wp_test_cookie and wp-setting-time-{\$uid}), as written by a wordpress.org verified admin on the wordpress.org support forums. The wordpress.org admin explains that on test cookies (just like the ones that came up on the scan) HttpOnly flags aren't set as they are not needed since they don't contain any actual user data. Additionally, when cookies are cleared such as when logging out, the cookies also contain no data as they are set by just having a space. Also, some cookies aren't set because they may need to be accessed by JavaScript such as cookies saved for name, email and url when leaving comments so they can future be acted upon by JavaScript in potential themes. These contain no private information anyway so shouldn't pose much of a security risk. The admin also mentioned how the ac-

tual authentication cookies used in WordPress are by default HttpOnly implying there shouldn't be extensive security risk.

Source:

<https://wordpress.org/support/topic/why-wp-is-not-creating-eachcookie-with-httponly-option-set-as-true/>

A possible solution:

If one is still concerned about the cookies lacking HttpOnly flags they can set them as PHP options in the wp-config.php file. Though it seems like in most cases this is not necessary.

Password Autocomplete in Browser	
URL:	http://localhost:8888/wordpress/wp-login.php
Risk:	Low
Confidence:	Medium
Parameter:	user_pass
Attack:	
Evidence:	<code><input type="password" name="pwd" id="user_pass" class="input" value="" size="20" /></code>
CWE ID:	525
WASC ID:	15
Source:	Passive (10012 – Password Autocomplete in Browser)
Description:	The AUTOCOMPLETE attribute is not disabled on an HTML FORM/INPUT element containing password type input. Passwords may be stored in browsers and retrieved.
Other Info:	
Solution:	Turn off the AUTOCOMPLETE attribute in forms or individual input elements containing password inputs by using AUTOCOMPLETE='OFF'.
Reference:	http://www.w3schools.com/tags/att_input_autocomplete.asp https://msdn.microsoft.com/en-us/library/ms533486%28v=vs.85%29.aspx

3. Password Autocomplete in Browser (3 Instances)

The Potential Danger:

Because the AUTOCOMPLETE attribute isn't disabled on user password type inputs, passwords could possibly be stored on browsers and later retrieved.

Is this concerning?

On the Zap GitHub issues page, I found a user who pointed out that since major browsers ignore the autocomplete=OFF setting for password inputs anyways, it doesn't make sense to have this type of warning to try and disable these autocomplete features. Members then mostly agreed with her stance and took note of it.

Source: <https://github.com/zaproxy/zaproxy/issues/4215>

Possible Solutions?

This doesn't seem like much of an issue since it has come to my attention that most browsers would ignore the recommended solutions anyhow.

Web Browser XSS Protection Not Enabled
URL: http://localhost:8888/wordpress/wp-login.php?reauth=1&redirect_to=http%3A%2F%2Flocalhost%3A8888%2Fwordpress%2Fwp-admin%2F
Risk:  Low
Confidence: Medium
Parameter: X-XSS-Protection
Attack:
Evidence:
CWE ID: 933
WASC ID: 14
Source: Passive (10016 – Web Browser XSS Protection Not Enabled)

Description:
Web Browser XSS Protection is not enabled, or is disabled by the configuration of the 'X-XSS-Protection' HTTP response header on the web server

Other Info:
The X-XSS-Protection HTTP response header allows the web server to enable or disable the web browser's XSS protection mechanism. The following values would attempt to enable it:
X-XSS-Protection: 1; mode=block
X-XSS-Protection: 1; report=http://www.example.com/xss

Solution:
Ensure that the web browser's XSS filter is enabled, by setting the X-XSS-Protection HTTP response header to '1'.

Reference:
[https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)
<https://blog.veracode.com/2014/03/guidelines-for-setting-security-headers/>

4. Web Browser XSS Protection Not Enabled (19 instances)

The Potential Danger:

The X-XSS security header is by default built into most modern browsers. With XSS protection not enabled, the browser could possibly allow pages to load when Cross site scripting attacks are detected. XSS attacks could result in stolen user information and cookies.

Is this concerning?


Since this is usually enabled by the browser by default, it is not very concerning. That being said, setting the X-XSS header will force the browser to use XSS protection and add to total security.

A Possible Solution:

The X-XSS Protection HTTP response header should be set to 1, and this could be done by editing the .htaccess file or by the use of various WordPress plugins.

X-Content-Type-Options Header Missing

URL: <http://localhost:8888/wordpress/2019/03/>

Risk:  Low

Confidence: Medium

Parameter: X-Content-Type-Options

Attack:

Evidence:

CWE ID: 16

WASC ID: 15

Source: Passive (10021 - X-Content-Type-Options Header Missing)

Description:

The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'. This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type. Current (early 2014) and legacy versions of Firefox will use the declared content type (if one is set) rather than performing MIME sniffing.

Other Info:

This issue still applies to error type pages (401, 403, 500, etc) as those pages are often still affected by injection issues, in which case there is still concern for browsers sniffing pages away from their actual content type.

At "High" threshold this scanner will not alert on client or server error responses.

Solution:

Ensure that the application/web server sets the Content-Type header appropriately, and that it sets the X-Content-Type-Options header to 'nosniff' for all web pages.

If possible, ensure that the end user uses a standards-compliant and modern web browser that does not perform MIME-sniffing at all, or that can be directed by the web application/web server to not perform MIME sniffing.

Reference:

<http://msdn.microsoft.com/en-us/library/ie/gg622941%28v=vs.85%29.aspx>

https://www.owasp.org/index.php/List_of_useful_HTTP_headers

5. X-Content-Type-Options Header Missing (37 Instances).

The Potential Danger: As the X-Content-Type-Options header was not set to no-sniff, this means that older browsers could potentially misinterpret the declared response body type as a type other than the type declared based on the content present in the file. This could potentially lead to an attack.

Is this concerning?

Back in July 2018 Google was actually encouraging publishers to use this mechanism of protection (setting the Content type headers) due to reported attacks. Google Chrome now actually has built in protection against Spectre and Meltdown attacks which are attacks that result from this vulnerability. That being said, developers are still encouraged by Google to set this header to no-sniff to make sure they are not vulnerable to these attacks. This probably isn't set by default as the no-sniff header causes some browsers to do strict content type checking which could limit flexibility.

A Possible Solution:

The X-Content-Type-Options header should be set to no-sniff, and this could be done by editing the .htaccess file or by using various WordPress plugins such as "Security Headers".

VI. Checking for Common Security Vulnerabilities

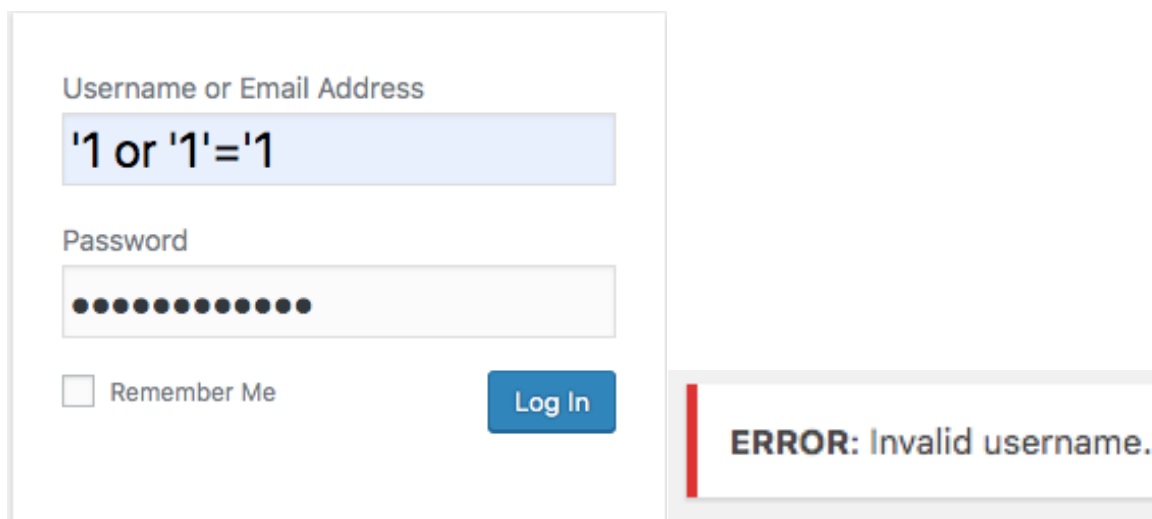
1. SQL Injection

SQL injection can occur when user input isn't properly checked by the application causing an attacker to attain unverified access to contents within the Database and in some cases even the ability to change them.

WordPress properly provides functions for validating, sanitizing and escaping user input. Validation checks that the user input conforms to proper formatting. Sanitization makes sure there are not any invalid characters present in the user input and escaping ensures that frontend input isn't harmful and isn't treated as JavaScript or procedural backend logic before going to the end user. WordPress provides a vast amount of sanitization and escape functions which can be found in the formatting.php file. These help to vastly reduce the potential for a SQL Injection attack.

Security in Action:

I tried to stake a SQL injection into the WordPress login form by inputting ' OR '1' = '1 indicating that I wanted to log in as the first WordPress user that's found in the database as it would interpret me as authenticated since it is set to a true statement. If the input isn't properly validated, sanitized, or escaped it would be successful and present a vulnerability. However, in this case it sent me a message that my input was invalid indicating that it properly checked my input and found that this was an invalid operation.



The image shows a screenshot of the WordPress login interface. The 'Username or Email Address' field contains the SQL injection payload: '1 or '1'='1. The 'Password' field is filled with dots. Below the fields is a 'Remember Me' checkbox and a 'Log In' button. To the right of the login form, a red error message is displayed: 'ERROR: Invalid username.'

Below is an example of how WordPress sanitizes the users entered username in the formatting.php file.

```

function sanitize_user( $username, $strict = false ) {
    $raw_username = $username;
    $username     = wp_strip_all_tags( $username );
    $username     = remove_accents( $username );
    // Kill octets
    $username = preg_replace( '|%([a-fA-F0-9][a-fA-F0-9])|', '', $username );
    $username = preg_replace( '/&.+?;/', '', $username ); // Kill entities

    // If strict, reduce to ASCII for max portability.
    if ( $strict ) {
        $username = preg_replace( '|^[^a-z0-9 _.\-@]|i', '', $username );
    }

    $username = trim( $username );
    // Consolidate contiguous whitespace
    $username = preg_replace( '|s+|', ' ', $username );

    /**
     * Filters a sanitized username string.
     *
     * @since 2.0.1
     *
     * @param string $username    Sanitized username.
     * @param string $raw_username The username prior to sanitization.
     * @param bool   $strict      Whether to limit the sanitization to specific characters. Default false.
     */
    return apply_filters( 'sanitize_user', $username, $raw_username, $strict );
}

```

A Caveat:

Plugins and themes built on top of the WordPress core software could potentially allow custom form input to be entered into the database without properly validation and sanitization. Though WordPress by default provides functions to escape, sanitize and validate input, if they are not properly utilized by developers, forms could still be vulnerable to being exploited by attackers which could end up causing SQL Injections and compromising the database. For this reason, many SQL Injections in WordPress sites can still occur.

2. CSRF Attacks

CSRF attacks occur when an attacker causes a user's browser to make requests that the user did not intend to make. CSRF attacks are often aided through the use of iframes which may cause users to not notice that they are executing some sort of request. This could be potentially dangerous particularly in WordPress as WordPress does not prevent iframes by default.

WordPress deals with CSRF attacks by the use of nonces which generally mean numbers that could be used only once as a means to prevent their misuse. However, in the case of WordPress, nonces are not just numbers but hashes composed of numbers and letters. This likens them much more to CSRF tokens and they are mainly used to ensure that the user sent request was actually completed by the user who was logged in. Nonces can be used in forms and are always used just once (hence the name). They are checked, before any request could be properly completed to ensure the authentication of the user sending the request. Nonces are sent in WordPress forms using the `wp_nonce_field()` function which adds two hidden fields

to the request made. The fields include one field with the name “_wpnonce” that includes a hash composed of numbers and letters and another field with the name “_wp_http_referer” that is used to verify the request originated from the users Wordpress source. The wp_verify_nonce() function is then used to check the validity of the nonce and the string denoting the action. If invalid, the request will not be processed.

A Caveat: Despite the use of nonces which is for the most part sufficient in preventing CSRF attacks, WordPress isn't perfect in preventing CSRF attacks. For example, plugins and themes built upon the functionality of WordPress still could potentially open the application up to many CSRF vulnerabilities. This is especially true considering the permitted use of iframes which aid many such attacks. This being said, the use of nonces properly can help prevent most CSRF attacks on WordPress.

Nonces Source: https://codex.wordpress.org/WordPress_Nonces


3. XSS Attacks

XSS attacks typically occur when an attacker injects malicious code into a website that potentially leads to a user sending their cookie information to the attacker and possible session hijacking

WordPress provides many escape functions that could stop XSS attacks from occurring. For example, the esc_html function ensures that no HTML special characters from the input are present in the user output. esc_js is a similar function that ensures that JavaScript code from the user input is completely broken before the output is then sent. The wp_kses function is a little more lenient and escapes HTML tags that aren't allowed but may in certain cases allow HTML tags for purposes of permitted actions such as italicizing the font.

An interesting thing I noticed when trying to commit XSS attacks in the WordPress frontend, was that users are required to input their name and email when commenting which is a good security measure in and of itself.

Additionally, when I tried to leave a comment on a post to test if `<script>alert(5)</script>` would work and display a JavaScript alert upon the output, the input was properly escaped and the output was simply `alert(5)`, as the `<script>` tags had been properly removed.



Allan
April 14, 2019 at 8:45 am

Leave a comment
Your email address will not be published. Required fields:


Comment

`<script>alert(5)</script>`

[Reply](#)

I then created a new account with the role of a contributor to test if post data input by a user with a contributor role can include HTML and JavaScript. As shown below, script tags were properly handled before the post data was output to the front end of the site.

This image shows the post form page:



The image shows a web browser window with a post form. The form contains the text `<script>alert(document.cookie)</script>` and `<script>alert(5)</script>`, which have been properly escaped. The browser's address bar shows the URL `http://localhost:3000/post/new` and the page title is "New Post".

This image shows the post output result:

alert(document.cookie)

UserOne April 14, 2019 Leave a comment Edit

<script>alert(5)</script>

UserOne April 14, 2019 Uncategorized Edit

These examples show how WordPress by default for the most part properly escapes user input before it is outputted which helps prevent XSS vulnerabilities.

A Caveat: All this being said and despite the presence of all of WordPress's escape functions in WordPress's core functionality, escaping isn't always used in forms in third party themes and plugins and that can potentially render many websites using those themes and plugins vulnerable to XSS attacks.

(It should be noted that due to WordPress hashing their cookie information, cookie theft, which is a common goal of XSS attacks, is less likely to cause harm since the authentication information is hashed. Additionally, WordPress uses HttpOnly authentication on authentication cookies which also helps security.)

4. Brute Force Attacks

Brute Force attacks occur when an attacker tries to input as many passwords as possible until the desired password is successfully cracked.

The Bad:

WordPress is often vulnerable to brute force attacks for a number of reasons:

1. WordPress by default allows users to use short passwords (they do have to agree to check the "confirm use of weak password option"). Nevertheless, this could still make it easy for attackers to brute force these weak passwords.

A Potential Fix: WordPress users can install security plugins or use themes that require passwords to be of a certain minimum character size. This will in turn help prevent brute force attacks from happening.

2. WordPress by default doesn't place a limit on the number of attempts a user can try and enter a password. This renders applications using WordPress vulnerable to brute force attacks as the attacker can try and input numerous passwords in the hopes of cracking one.

Analysis: This is actually a very concerning vulnerability. Brute Force attacks were responsible for approximately 16% of all WordPress run websites being compromised in 2016 per [wordfence.com](https://www.wordfence.com/blog/2016/03/attackers-gain-access-wordpress-) (<https://www.wordfence.com/blog/2016/03/attackers-gain-access-wordpress->

sites/) and continue to be one of the most common security vulnerabilities in the WordPress core software.

My speculation here is that WordPress most likely wants to provide maximum flexibility to their users and therefore does not want to inconvenience them by placing a hard limit on the number of times they can attempt to enter their passwords. They most likely hope that the user selects a theme or plugin on top of the core software that will implement such a login attempt limit feature and protect their site. That being said this does present a major and very concerning vulnerability.

A potential fix:

Users can download plugins such as “Login LockDown” that can implement the functionality of limiting permitted login attempts.

The Good:

WordPress uses 4 types of salts (‘AUTH_SALT’, ‘SECURE_AUTH_SALT’, ‘LOGGED_IN_SALT’ and ‘NONCE_SALT’). Salts are characters appended to passwords before hashing, and therefore increase security and make brute force attacks less likely.

5. Other:

Uses a Maximum Password Length:

By default, the DB allows a maximum of 60 characters in a username. This type of check is helpful in preventing certain vulnerabilities such as overflows.

Table: wp_users

Field	Type	Null
ID	bigint(20) unsigned	
user_login	varchar(60)	

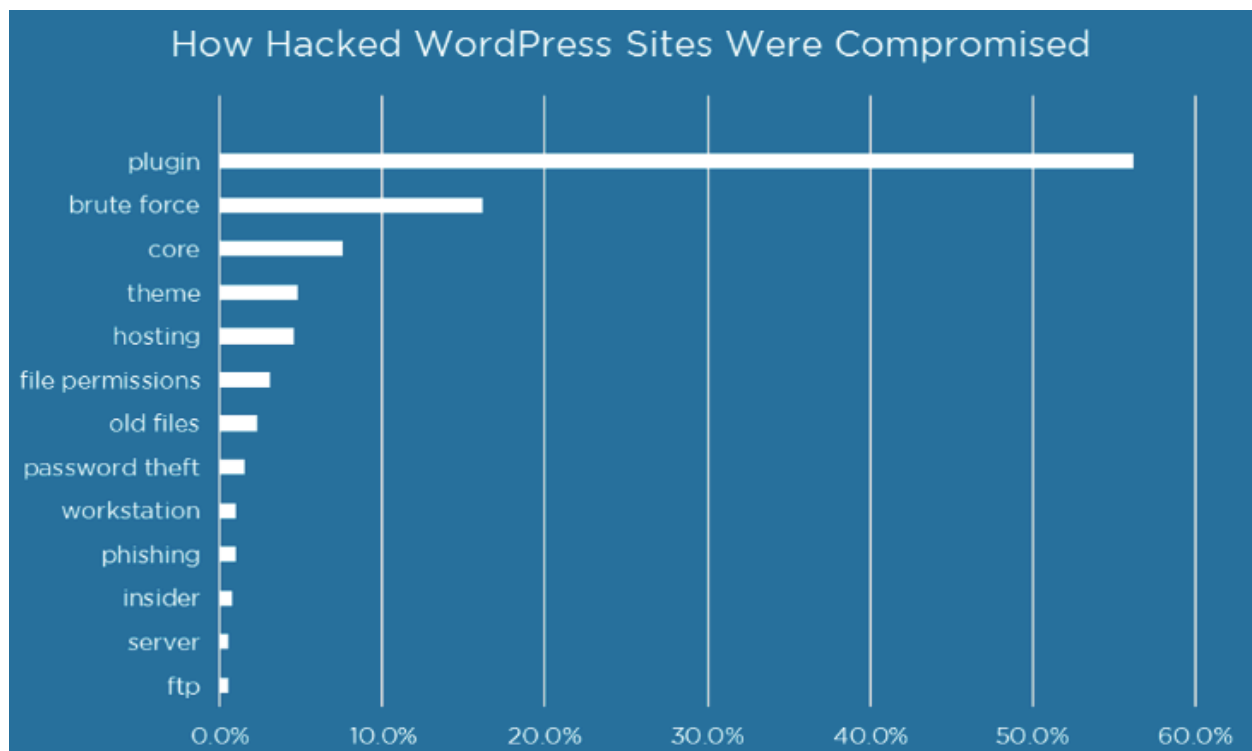
Doesn't Check if User Emails actually exist

By default, WordPress doesn't check if user provided emails actually exist, and instead just makes sure that they are formatted correctly. This can potentially lead to a user using another person's email and thereby subjecting that person to unwanted emails.

VII. Conclusion:

Since WordPress is by its nature customizable, it is no wonder that its developers place plenty of value on flexibility even if sometimes that flexibility allows security vulnerabilities. For example, I previously discussed how WordPress fails to disallow the use of iframes, which presents a vulnerability even if it does allow for added flexibility. Also, I explained how WordPress fails to by default limit user login attempts, which puts websites using WordPress at risk of brute force attacks. Brute force attacks are especially concerning as they could compromise both user confidentiality and integrity (which is especially dangerous considering there are many E-Commerce sites built upon WordPress). They could also in some cases overload the server which could lead to possible availability concerns.

That being said, the real dangers of security with regard to WordPress come in relation to its extensions, plugins and themes. Research has shown that nearly 60% of WordPress sites being compromised comes from security vulnerabilities in faulty themes or plugins as shown in the image below via [wordfence.com](https://www.wordfence.com/blog/2016/03/attackers-gain-access-wordpress-sites/) (<https://www.wordfence.com/blog/2016/03/attackers-gain-access-wordpress-sites/>) .



Therefore, users of WordPress must always be weary when using third party plugins/themes as these have not always been adequately checked for vulnerabilities. Also, to prevent Clickjacking, XSS, CSRF, SQL injections and brute force attacks totally, it's probably a good idea that users implement reliable security plugins that could safely account for these concerns.

Additionally, to improve security, users should definitely update the password hashing mechanism to extended DES or blowfish as that is key in protecting user information. Also making sure the security keys and salts are properly implemented in WordPress themes is very important as well.

Finally, since the WordPress software is routinely updated by its developers to account for exposed security vulnerabilities, it is very important for users to make sure their version of WordPress is always updated. Deprecated versions of WordPress can result in users' websites being vulnerable to previously exposed vulnerabilities and therefore attacks.

User Privacy Assessment

As for user privacy, as mentioned previously WordPress does a good job sanitizing and escaping input data, which makes SQL injections, XSS attacks and CSRF attacks less likely. Additionally, WordPress hashing their authentication cookies helps as well. That being said, the fact that WordPress by default has no limit on password attempts puts websites at risk of brute force attacks which can result in passwords being compromised and therefore user privacy as well. Furthermore, Wordpress also allows users to use short passwords which also presents such a vulnerability. These Brute Force vulnerabilities are probably the biggest privacy concern on WordPress.

Overall Assessment:

The Good:

- WordPress is fairly secure considering its necessary emphasis on flexibility
- It provides plenty of escape, sanitization and validation functions to prevent SQL injections and protect the database
- It properly escapes user input before it is outputted to prevent XSS and CSRF vulnerabilities
- WordPress uses security keys
- It uses nonces for CSRF Protection and also authentication of user intent
- It places a 60-character maximum limit on usernames
- Allows for a bevy of security plugins on top of the core software that help prevent vulnerabilities
- It has developers who routinely update the core software to help prevent vulnerabilities after they are exposed

The Bad:

- WordPress allows for Iframes
- It does not limit user login attempts which presents a Brute Force Vulnerability
- It allows for short passwords which presents a Brute Force Vulnerability
- It still allows the use of MD5 password hashing in some cases which is widely considered a broken hashing mechanism
- Plugins built on top of the core software are not always checked by the WordPress developers and could therefore present security vulnerabilities or even be malicious to users who download them.
- Users don't always update their core WordPress software, which can cause attackers to exploit known security vulnerabilities that would have been fixed by updated versions
- WordPress fails to by default set X-Content-Type headers to no-sniff