# Nebula security domains

Luca Lombardi

February 13, 2024

**Abstract**

Application to implement the concept of **security domains** in a Nebula network. The project had been developed for the course of Cybersecurity M of the University of Bologna. Full documentation can be found here
A **security domain** is a set of logically related resources that can communicate and are subject to the following constraints:

- A resource can have multiple **security domains**
- The information related to the **security domain** must be present in the resource's digital certificate
- A resource can open a connection only toward another resource that share at least one **security domain** otherwise it is blocked by default

The purpose of the application is to automatically generate all the configuration files needed to implement the provided configuration. A single file should contain all the necessary information.

## 1 Introduction

Nebula is an overlay networking tool designed to be fast, secure, and scalable. Connect any number of hosts with on-demand, encrypted tunnels that work across any IP network and without opening firewall ports.
The goal of the project is to implement the logic of **Security Domain** using the tool that Nebula gives us and automate the process of converting high-level logic into configuration files and rules for the nodes of the virtual network. Although partially implemented for ease of use and testing purposes, the project will not focus on the automatic generation of configurations that are not involved in the concept of **Security Domain**, remote deployment, and host security.

All tests have been run on virtual machines generated through Vagrant running Debian 12 (Bookworm64) operating system and virtualized with VirtualBox. The machines have been configured to run on different subnets. An additional machine is defined as acting as a router and allowing all others to connect to the VM lighthouse simulating the public internet. All VagrantFiles are available on the official documentation on github.

The project was divided into three parts, in the first a basic working prototype was developed, in the second part the problem was completely reanalyzed from which a second version was developed and in the third starting from an edge case the prototype was extended with new features to handle it.

## 2 The first version

### 2.1 Implementing security domains

For firewall configuration Nebula uses a logic of default deny on top of which "allow" rules are added. To implement Security Domains I relied on Nebula's "group" feature. With the group feature, I can define a group for each security domain, assign the hosts to those groups, and set them to allow connections only from nodes that share at least one group. Rules are defined with the following syntax:

```
firewall:
  outbound_action: drop
  inbound_action: drop

  ...

  outbound:    #allow any connection generated by this host
    - port: any
      proto: any
      host: any

  inbound:     #allow only connections that are from a member of SecDom 1 o 2
    - port: any
      proto: any
      group:
        - SecDom1

   - port: any
      proto: any
      group:
        - SecDom2
```

## 2.2 Configuration file

For simplicity, in this first version, I will use the JSON format to define the Security Domain configuration, each host will be defined as an object in a list with all the parameters needed to generate the config file and the key certificate pair.

```
[
   {
      "name": "laptop1",
      "nebula_ip": "192.168.100.11/24",
      "machine_ip": "192.168.2.11",
      "security_domains": ["LaptopSD", "ServerSD"]
   },
   ...
]
```

Complete file here.

## 2.3 Software

In this first version, working on Linux, I developed a script that would generate the configuration files with the appropriate rules. Given its many limitations, I won't focus on it in this document and instead move immediately to the second version. More can be found in the official documentation. The full script can be found here.

# 3 The second version

In the prototype, we saw that using the group feature of nebula allows us to implement all the nebula security domains as needed by requirements. The solution implemented though, as with all the first ideas, is not the best. Here I've reanalyzed the project from a top-down perspective and implemented a new solution. Following are the main aspects I've focused on with the respective conclusions.

## 3.1 Single Responsibility principle

In the previous configuration file, we had to define on top of all the hosts and the Security Domains other data like lighthouse and hosts IP, the reason being that we asked the script to generate all

certificates and keys on top of the configuration files that had to then be modified. From a project point of view, we asked for a single script to do everything. Let's reanalyze the process and see how to handle it better.
The workflow of the system can be defined with the following points:
1) The Security Domain config file is created.
2) Certification and Keys are created for every node of the network.
3) Configuration files are created for every host.
4) Configuration files are modified to implement Security Domain logic.
5) All generated files are distributed to the hosts.
By separating the creation of keys, certificates and configuration files from the modification to implement the firewall rules we could simplify the nebula Security Domain file.

## 3.2 Configuration file format

In the previous version we used JSON as the technology to define the Security Domains configuration file but we have other options to choose from. Of the existing technologies the most common are **JSON**, **YAML** (used by nebula itself) and **TOML**:

- **JSON** is perhaps the simplest and most intuitive.
- **YAML** seems the most suitable solution being used by nebula itself.
- **TOML** is newer than the previous ones and represents a simpler version of YAML.

After some research my choice still falls on **JSON** for the following reasons:

- **YAML** is much more complex and error-prone than one might think (see the yaml document from hell for reference).
- **TOML** despite being more secure than YAML is not particularly intuitive.
- **JSON** is intuitive, much less error-prone (for now the file has to be generated by hand so this is a big plus) and it is always possible to convert from **JSON** to **YAML**, the opposite is not necessarily true.

Regarding the arrangement of data in the file, two paths can be followed:

1. I write a list of security domains, in each domain I define the list of hosts that belong to it.
   ADVANTAGES:
   - More intuitive for the user when defining the network.
   - Easily answers questions such as "who belongs to this Security Domain?"
2. I write a list of hosts, and in each host, I define the list of security domains to which it belongs (as it was in the prototype).
   ADVANTAGES:
   - Easier to implement at the code level.
   - Easier to answer questions such as "which Security Domain does this host belong to?"

Obviously **User Friendliness** takes precedence over everything else, so the first point is also the best.
The configuration file is then defined as follows:

```
[
   {
      "name": "serverSD",
      "hosts": ["server1", "server2"]
   },
  ...
]
```

A couple of notes:
- As defined above the only data that should appear in the file are the IDs of the Hosts and the SecDoms to which they belong, data such as IPs and special parameters aren't needed and should not be part of it.
- Host IDs and Security Domain names don't need to be unique for this project however making them so is not only a good practice but would facilitate everything and benefit the end user.

## 3.3 Can a Lighthouse be part of a security domain?

A lighthouse still needs the same files as any other host, steps 2, 3, and 5 of the workflow are therefore necessary for it as well, the question remains whether a lighthouse can be part of a Security Domain or not.

Given the nature of Nebula, every host (except in extremely special cases) must be able to connect at least once to a lighthouse to function properly. It is therefore clear that every host in the network should be able to connect to a lighthouse. In situations where there is only one lighthouse, limiting its connection would be a serious mistake. However, in Nebula, it is possible to implement several lighthouses and in that case it might be possible. I still believe that it wouldn't be a good practice because the only reason I can think for denying access to a lighthouse would be to redistribute the workload in the network but to do so nebula already has better tools and it's not the job of Security Domains to take care of it.

(Little extra: nebula communicates with peer-to-peer encrypting, it is not possible to sniff the traffic passing through the lighthouse, also in case of an attack, one could consider blocking certain hosts from communicating with the lighthouse but it is not the task of this project to handle such a case.)

Final answer then is NO, but practically speaking I can't block a user from putting a node called "lighthouse" in the config, I will at least print some warning telling them it's not a good practice.

## 3.4 What about scalability?

As the number of nodes increases, the complexity in the JSON file increases linearly as does the file creation time.

## 3.5 Software

Using a Linux script has several limitations: - It is not easily portable to other platforms and/or architectures (if not impossible).

- It is not easily portable to other platforms and/or architectures (if not impossible).
- It exploits "jq" a tool not necessarily present, to ensure execution it would require automatic installation and consequently sudo permission which normally isn't needed.
- It's monolithic.

For the new version, we will use a programming language that is deployable on different platforms and organized as much as possible in components. My choice falls on Python for ease of use and familiarity.

### 3.5.1 Main

The main class mirrors the 5-point structure defined earlier to which several initial checks on parameters, files, and data are appended. Almost all functions are placed in a try-catch construct (not shown here) to handle errors in the provided files. generateSD.py

```
def main():
    #1 checks if files are correct
    checkParam()

   #checks if lighthouse is in a SecDom, if it is print a warning
   if SD.hasLightouse(securityDomainsData):
      print("...")

    #2 key and crt generation for all hosts
   Gen.generateCrt(hostsSetupData)

    #3 generation of all config files
   Gen.generateConf(hostsSetupData)

   #4 editing of all config files
   SD.addFirewallRules(securityDomainsData)

    #5 distribution
    Dist.sendFiles(sys.argv[1])
```

NOTE: At this point, I end up with two config files, one with the data containing the IPs and data to create the files and one with the security domains, in this case, I've also implemented a function to merge the Security Domain data into both files and have both options defined earlier to view the structure of my network. The perfect solution would be to completely abandon the manual generation of these files and switch to a software tool to define the entire network and hide the saved file formats from the end user while allowing them to access, view, and modify the information related to the network structure through the tool.

### 3.5.2 Data generation

A library was defined to generate node data. Here I'll present only the function responsible for generating the keys and certificates, the rest was not part of the project and can be found here.

```
scriptDir = "../scripts/"
outputDir = "../TmpFileGenerated/"
configFilePath = "../config-default.yaml"

def generateCrt(hostsSetupData):
    try:
        for host in hostsSetupData:
            if len(host["groups"]) == 0:
                os.system("./nebula-cert ...")
            else:
                os.system("./nebula-cert ...")
    except:
        raise Exception("Could not parse Data correctly")
```

This function takes an array of dictionaries in which a "group" parameter contains all groups a host is part of including all Security Domains are defined. This could be generated from the data of the Security Domains or, like in my case, from another source. In my case, having already a second configuration file defining all hosts of the network, I've preferred to use it after updating it with the data of the Security Domains.

### 3.5.3 Security Domain logic

A second library was defined to implement all Security Domain related functions: SecurityDomain.py

```
def addFirewallRules(securityDomainsData, hostsList = None):
    if hostsList == None:
        hostsList = getHostsList(securityDomainsData)

    for host in hostsList:
      ...
      configFile = open(outputDir + "config_" + host + ".yaml")
      configData = yaml.load(configFile)

        #save old inbound config and empty it
        oldConfig = configData["firewall"]["inbound"]
        configData["firewall"]["inbound"] = []

        #add rules
        for SecDom in securityDomainsData:
            if host in SecDom["hosts"]:
                configData["firewall"]["inbound"].append(
                {'port': 'any', 'proto': 'any', 'group': SecDom["name"]}
                )

        ...
      #save previous data if needed
        if not configData["firewall"]["inbound"]:
            configData["firewall"]["inbound"] = oldConfig

        #save data in new file
        with open(outputDir + "config_" + host + ".yaml", 'w') as f:
            yaml.dump(configData, f)
```

```
def getHostsList(securityDomainsData):
    try:
        hostsList = []
        for SecDom in securityDomainsData:
            for host in SecDom["hosts"]:
                if not host in hostsList:
                    hostsList.append(host)
        return hostsList
    except:
        raise Exception("...")
```

The optional hostList parameter is given because getHostList is not particularly efficient, there could be a better way of creating that list, the option is left to the programmer. The list does not need to be limited to only hosts that are part of a Security Domain but only host that are part of a Security Domain as defined in the config file will be modified.

If a host is not part of any Security Domain its current configuration should not be overwritten, to achieve this we first save the current inbound configuration and in the end, we check if the config is still empty, if it is it means the host is not part of any Security Domain and we save the previously saved config.

## 3.6 Test

In this second version I've run a test composed of 6 machines including 3 laptops in one Security Domain, 2 servers in a different Security Domain, and the lighthouse. Two of the three laptops must be able to connect with only one of the servers. It simulates a distributed server in a cluster of machines with one of them acting as a gate for some of the laptops. The VagrantFile can be found here

```
[
   {
      "name": "serverSD",
      "hosts": ["server1", "server2"]
   },

   {
      "name": "serverAccessSD",
      "hosts": ["server1", "laptop1", "laptop2"]
   },

   {
      "name": "laptopSD",
      "hosts": ["laptop1", "laptop2", "laptop3"]
   }
]
```

## 4 Edge case - Mistrustful Colleagues

If we slightly modify the previous case by adding the requirement that laptops cannot connect we run into some problems. In this case removing “laptopSD” would not be sufficient, laptop1 and laptop2 would still be able to communicate by both being part of “serverAccessSD”.

With the way we have defined Security Domain rules so far we would be forced to define a Security Domain for each laptop that wants to connect with the server in which only the server and the laptop itself are present. The solution is not only inconvenient to define but would be extremely unscalable and unmaintainable. Therefore, we need to extend the system by introducing a new concept: **Roles**.

I define three possible roles into which a host can fall: **SenderOnly**, **ReceiverOnly**, **Both**.

### 4.1 How are roles implemented

Roles can be implemented in Nebula in the following ways:

- **SenderOnly**: needs to block the receiving from the Security Domain even though it’s part of it, we simply don’t add the inbound allow rule.
- **ReceiverOnly**: should allow to send everything excluded to a specific group, to do that i would need to manually define a rule to let out the connection for every group that this host is part of and it’s at least a sender plus the lighthouse (NOTE: this node will not talk to anyone that is not in a Security Domain unless manually configured to do so. . . )
- **Both**: as defined in the previous sprint

From the **ReceiverOnly** we start to see some of the limitations of the Nebula software. In a large network with hundreds of subdomains, this quickly turns unmanageable by hand. If no role is defined in the config file it is supposed to be a **Both** role.

### 4.2 Redefining outbound rules

Being a **ReceiverOnly** host in a Security Domain means changing completely the way outbound rules are defined, changing from a default allow to a default deny + other rules. To implement this there are two options:

1. Identify ReceiverOnly in advance and treat them differently.
2. Change the way outbound rules are defined for all the hosts.

In this test, we will follow the second option. Implementing roles is just an idea and could very well be discarded in the future so there is no reason to make development too complicated.

### 4.3 Test

In this third case, we will have 6 machines including 3 laptops, 2 servers in a SecDom, and the lighthouse. The three laptops must be able to connect with one of the servers but should not be able to connect. For a more complete example, we will define server1 as a “ReceiverOnly” node and all the laptops as “SenderOnly” nodes. The VagrantFile can be found here

### 4.4 Software

The configuration file changes as follows:

```
[
    {
        "name": "serverSD",
        "hosts": [
            {
                "name": "server1",
                "role": ""
            },
            {
                "name": "server2",
                "role": ""
            }
        ]
    },

    {
        "name": "serverAccessSD",
        "hosts": [
            {
                "name": "server1",
                "role": "receiver"
            },
            {
                "name": "laptop1",
                "role": "sender"
            },
            ...
        ]
    }
]
```

Compared to the previous version the main difference is in how the rules are generated: Given a host, for each Security Domain we check if it’s a **SenderOnly**, **ReceiverOnly**, or **Both**

– For **SenderOnly** we allow only the ping in inbound and any sending for outbound rules.

– For **ReceiverOnly** no sendings are allowed in outbound, inbound is the same.

– For **Both** inbound configuration is the same and outbound is allowed.

On top of those rules, one extra outbound allow is needed to connect to the lighthouse, this is implemented by allowing outbound to a specific host identified with his ca_name. A new parameter is then required in the “lighthouse name” for allowing outbound connection to the lighthouse, by default is defined as “lighthouse”. Full script here.

## 5 Conclusion

During development, even though the application covers more functionality than initially requested, I paid special attention so that the required functionality would be independent of the rest. The end result not only shows how the Security Domain concept can be implemented in Nebula but also how to structure an easily portable application to automate the entire process.

Among possible future developments, a must would be a tool to manage a virtual network and automatically generate Security Domain files removing the need to define well-formed files by hand and hiding unnecessary implementation details from the user.

It also remains to be seen whether a multi-lighthouse configuration can be easily managed through Security Domains or whether it poses additional challenges.

## 6 Useful links

- Nebula github
- Medium: introducing nebula, the open source global overlay network
- Nebula doc
- Nebula quick start
- Nebula config reference
- Nebula official slack (Big thanks to the people on the server for their assistance and availability)
- Project github