NebulaRAT (Remote Access Tool)

Luca Lombardi, Stefano Paparella, Pasquale Basta

July 2024

Abstract

Application to implement an enterprise remote access tool leveraging Nebula technology. The project had been developed for the course of Scalable and Reliable Services M of the University of Bologna. Full documentation can be found here.
The application must meet the following requirements:

- Support a **database** of remote machines by importing the **Nebula certificates** and the current **Nebula firewall rules** for reaching them.
- Present a Web interface protected by **auth** to access a portal showing all the available machines to the user.
- An Admin interface to define **security roles** for users and configure what machines can be available.
- The ability to generate a **short-lived certificate** on the fly allows the user to connect to the desired machine.

The purpose of the application is to temporarily allow system admins to connect to nodes on the Nebula network bypassing firewall restrictions. This must be obtained through the **short-lived certificate** automatically generated.

Chapter 1

NebulaRAT overview

Nebula is a zero-trust 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 project's goal is to implement a **web application to allow one or more admins** of a Nebula network **to access assigned machines** through Nebula itself. The project's main focus is to use **Nebula-generated certificates** to bypass firewall configurations of the nodes. For this reason, **security will be a top priority** in managing permissions, roles, and certificate emissions. We will leverage the power of **Azure cloud services** to host the web app and the database.

The project does not focus on Nebula host configuration or certificate management. We will assume to work with a correctly pre-configured network.

CI/CD has been achieved using **Azure** combined with **Github Actions** while **Terraform** has been used to implement **infrastructure as a code**.

Chapter 2

Design choices

In this chapter, we'll discuss the design choices we took, based on requirements, budget and utility of Azure resources, functions and data.

2.1 Short-lived certificate generation

Generating certificates with a specific duration in Nebula is rather simple. However, **making sure that a certificate allows access only to a specific machine** is not. To solve this we looked at several ideas. The best would be to rely on an external system to request the generation and/or modification of rules dynamically on the machines for which I require a certificate. Having no such system, **the proposed solution is** as follows: we exploit the **CIDR** parameter in the firewall rules to **define specific IPs** on each machine that we **expect an admin to have** when trying to connect with a temporary certificate.

```
inbound:
    - port: any
      proto: any
      cidr: 192.168.100.121/32
      ca_name: Myorg, Inc
```

This requires that when deploying the configuration some IPs be reserved and assigned to each machine, which is not difficult to do automatically. Any short-lived certificate will need to be generated with the correct IP. As for the **validity time**, we let the user choose between **2, 4, or 8 hours**.

2.2 User and permission management

The initial idea was to define permissions in **two distinct dimensions**: First, **which machines a user can access**, and second, **what actions a user can perform**. This second point was to include both special machine accesses and app-related permissions. In the end, we did not define special machine access permissions because of Nebula's limitations. This would be possible only by modifying the configuration files on the end machines, which we ruled out earlier.

The authentication phase also consists of a **simple username/password pair and eventually a two-factor authentication code** sent by email.

Nebula was born as a virtual network for **Zero-Trust architecture**, makes sense that this tool would follow the same principles. **We could use the same Nebula** to implement it hosting the web app on a node of the Nebula network. Unfortunately, Azure webapp does not give the freedom necessary to achieve that, and **changing to an IaaS would remove all the advantages** PaaS gives and implementing zero-trust in Azure can only be achieved via EntraID tool, which we had no access to.

2.3 Web Interface

2.3.1 Framework Selection

Python has various web frameworks, such as Flask, and Django, which enable the rapid development of web features. Among the various frameworks available, we chose to use **Flask**, for various reasons:

- **Simplicity and Lightness:** simple to use and easy to learn
- **Flexibility:** Flask is highly flexible, as various extensions can be added to manage different functionalities, in fact, we used Flask-Login, and Flask-SqlAlchemy.
- **Azure and Flask:** Azure offers support for Flask applications, through Azure App Service which simplifies the deployment and management of the Flask web application on Azure
- Django has a steeper learning curve than Flask

Within our app, we decided to create **two main routes** regarding the assignment and revocation of the machine, because one route will take care

of **showing the possible choices** to the authorized user, while the other route will manage the actual **action and data processing**.
Another design choice was to define one route for the standard user dashboard and another route for the admin dashboard. This separation is important to **clearly separate responsibilities** since the standard user will be able to simply view the dashboard, with the table of available machines, while the admin will be able to perform additional functions, i.e. adding a new user, management operations, etc... Therefore, differentiating these two routes allows you to efficiently and securely **manage the access** of different types of users. A future development would be to use a single route for both dashboards, dynamically managing content based on the user's role.

2.4 Database

2.4.1 What DB to use?

We had different possibilities, from the simpler **Azure DB for SQL** to **PostgreSQL** and **MongoDB**. We've opted for PostgreSQL because it's **always compliant to *ACID properties***, while MongoDB only in limited scenarios. Azure DB for SQL was simpler to use and manage, but it wasn't always *ACID-compliant* because it depends on the storage engine used: we aim to have data saved **consistently and durably** in a DB where multiple accesses do not interfere with each other.

2.4.2 Database implementation

First version

The embryonic version of the DB provided only three tables, to carry out tests on assignment/revocation of access to certain machines: **MACCHINA**, **UTENTE**, and **USA**. The latter satisfies the N-to-N relationship between the other two tables, with a **unique** `(utente_id, macchina_id)` **torque within the table**. This table is also **resilient** to the elimination of entries from the associated tables.

Second version

In the second version of the DB, we added two tables, **CERT** and **CONF**, which contained the information regarding the machines certificates and configuration files. Regarding the first one, it's worth to clarify one point: **certificates come with their related keys**, so one could think about storing them too. However, we didn't do that, because we intended

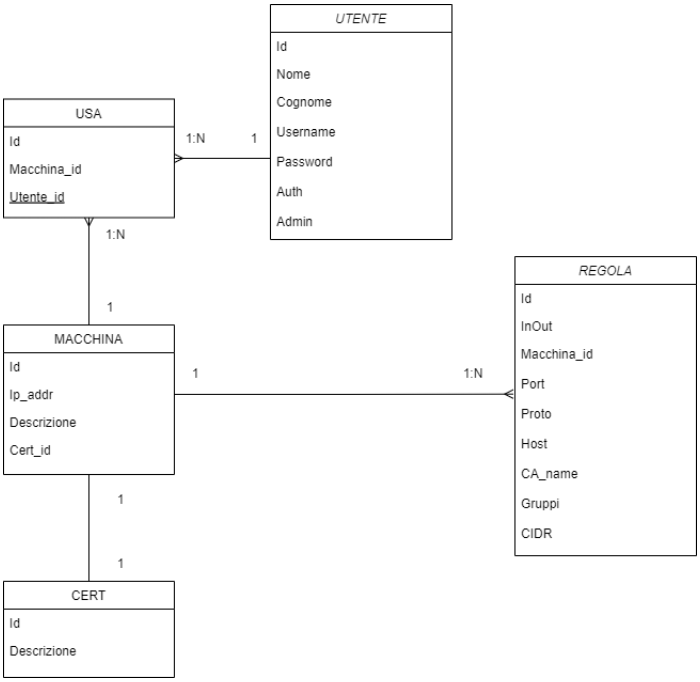


Figure 2.1: Final version for the database.

keys as **users private data**, thus if one loses his key, he **must generate a new *(certificate, key)* couple** for his Nebula machine.

Another table we introduced was **REGOLA**, which contains the firewall inbound and outbound rules, plus a reference to the CONF table, to realize the 1-to-N relationship between machines and rules. More precisely, CONF was thought to contain the IP address of the machine and some general rules, such as ***TCP***, ***UDP***, and ***default timeouts***, but then we realized that those weren't necessary for our purpose, so we removed them. With this removal, the CONF table had to carry only the information about the machine's IP address, which could be easily moved to the MACCHINA table, thus deleting the CONF table itself. The result was a **lighter DB** and s**impler design and development** of complex queries.

2.4.3 Database connection with Python

Once we defined what DB to use and which data had to contain, the next design choice was about **what Python framework to use for the connection to the database**. We had two possibilities: the first one was **psycopg2**, a library designed specifically for the connection to the Postgres

DB and based on launching queries by executing SQL raw code; the second one was to use **SQLAlchemy**, an ORM (*Object-Relational Mapping*), which can be used to connect to **different kinds of databases** and allows devs to handle tables and data through an **OOP approach**. We choose the latter framework, because it allows us to use both raw SQL code and specific class methods to launch queries, thus reducing the risk of a SQL Injection attack by a malicious user.

2.4.4 PopulateDB

We decided to create a script that reads initial configuration files, thus simulating the config of the company using our web app. These files must be in JSON format, and through the data it recovers from these files, it populates the tables present within the database. Choosing to use this script which automatically populates the tables allows you to have:

- **Automation and Efficiency:** This is very advantageous when you have a lot of data. The script also reduces the risk of human errors and saves time and effort.
- **Scalability:** With a script, it is easier to manage large amounts of data. You can simply modify the script, to handle more data or to add new fields without having to manually repeat the data entry process.
- **Data Consistency:** The use of the script guarantees the consistency and validity of the data inserted into the tables.

So we said that the configuration files must be in JSON format, this is because:

- **Data Structure:** JSON is suitable for structured data like ours and allows easy understanding and manipulation of data for both developers and machines.
- **Interoperability:** Different programming languages and frameworks support JSON.

We hypothesized that there are two initial configuration files, one that contains information relating to the network configuration, and one that contains data relating to the user. Initially, we had foreseen that in the configuration files relating to the user, the password would be the default one and that it would be shown in plain text, assuming that the user changed it at the first login. However, for security reasons we decided to insert the hash of the password into the JSON file, avoiding showing it in clear text.

2.4.5 Download generated certificates and keys

Another important feature is to allow users to download the *short-lived certificates* they generated, along with the correlated keys. Our first idea was to allow the user to **decide where to store the generated files**, via a graphic window built with *Tkinter*, but this library installation required the use of the *apt-get install* command, which in turn couldn't be run on Azure due to the absence of super admin privileges. The solution was to use the Flask `send_file` function, which allows to download **only one file at a time** directly in the Downloads folder. For this reason, and given that we needed to download more files, we have decided to put certificates and keys in a single *.zip* file, which is downloaded by the user.

Chapter 3

Security

3.1 Secrets management

A key point is the management of **secrets**. The application presents different elements and variables that should not be made public:

1. **Google credential**s for two-factor authentication emails
2. **Database credential**
3. **Flask-BCrypt secret** for hashing passwords
4. **Certificate and key of CA** signing the short-lived certificates

The best thing would be to use the **Azure Key Vault**, which can not only handle the passwords and secrets but also the certificate with which we create the short-lived. Due to the limitations of the university Azure account, this is not possible for us (it necessarily relies on **EntraID** to which we do not have access). Secrets are therefore defined as automatically generated **environment variables** via **Terraform**.

```
...
#Flask-BCrypt secret
#export TF_VAR_flask_secret=(the secret)

variable "flask_secret" {
  sensitive = true
}
```

We also made sure that the files **terraform.tfstate** and **terraform.tfstate.backup** are not uploaded on Git Hub.

3.2 Credentials management

Think about the users' credentials stored in the database. We decided to store encrypted passwords, instead of plaintexts. At first, we had to decide how to encrypt passwords: we opted for **hashing** instead of a **key-based encryption**, because it's *one-way*, safer than the latter. At this point the question is: *What do we use to encrypt passwords?* We could have used one of the most famous libraries, such as ***Cryptography***, but our decision fell on ***Flask-Bcrypt***, a Flask sub-library based on the ***Blowfish*** encryption algorithm, which is easier to use than the first one, because it has **only two functions**, which have been used for handling the *login*, *user add*, and *change password* features.

3.3 Security Issues

To locate security issues in our project code, we used Snyk and Git Guardian, which dynamically analyze the project repository for security issues like:

1. **Old versions of Python libraries**, which were update to the latest version.

2. **SQL Injection**, due to the use of **raw SQL code**. We fixed this exploit by **parametrizing** the inputs of the SQLAlchemy ***execute()*** function, through the use of the *:var_name* notation inside the queries.

   Example: the query command is defined as

   ```
   macchine = db.session.execute(text(sel_macchine),
       {'username':session["username"]})
   ```

   while the query related to this command was formatted as

   ```
   sel_macchine = SELECT [...] WHERE username = :username;
   ```

   This kind of operation leads to the **escaping of the input parameters**, so that they are **not treated** as SQL statements.
   Please note that this kind of exploit is possible only when there's some raw SQL code, while when we run the ***query()*** function, via

class instances representing db tables, the SQL injection isn't possible anymore, because there's no SQL code to manipulate.

3. **Path traversal**. This kind of attack was possible when the Flask *send_file* function was executed, because of the presence of a path to a resource in the hosting system. To solve this critical flaw, we created a Python function that:

    1) generates the **base path**, which goes from the current folder to the folder that contains all the files that are generated;
    2) builds the **full path** to the resource and verifies that it is actually a file;
    3) generates the **true absolute path** to the file and calculates the **common path** between it and the complete path to the file;
    4) checks the **equality** between the file specified in the real path and the one specified by the user;
    5) redirects the user to an error page if it has detected a path traversal attempt, allows the download otherwise.

Chapter 4

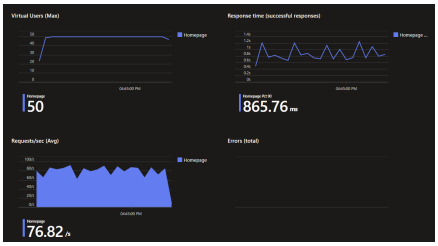
Conclusions

4.1 Load test and cost analisys

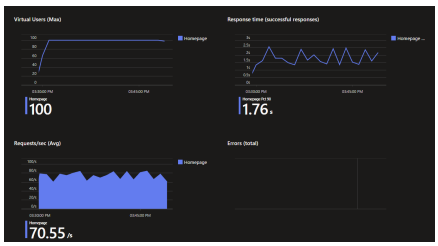
For load analysis, we relied on the **Azure Load Testing** tool.

1. **First basic test**
   - WebApp Pricing plan: B1 (Most basic plan, cost per month 12.30 Euro)
   - Database Pricing Plan: Standard_B1ms (1 vCore) (Most basic plan, cost per month 13.60 Euro)

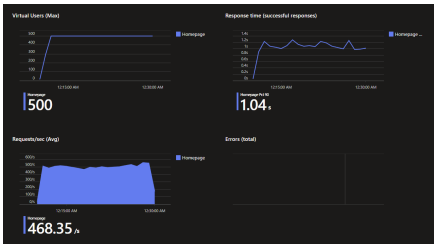
Even with the most basic services that Azure provides, the web app can handle a load of **50 users simultaneously** while maintaining a response time of **less than a second**. Already with 100 users, the average response time increases above one second. In most cases, however, handling 50 users simultaneously can be considered sufficient.



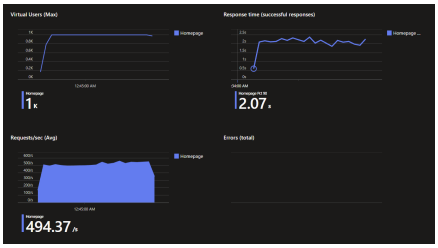
(a) test with 50 users.



(b) test with 100 users.



(a) test with 500 users.



(b) test with 1000 users.

1. **Test with Premium Services**
   - WebApp Pricing plan: B1 (Minimum premium plan, cost per month 79.266 Euro)
   - Database Plan: Premium v3 P0V3 (1 vCore) (Minimum premium plan, cost per month 108. Euro)

By improving the hardware we see a marked **performance improvement** by handling smoothly 500 users at once. This achievement is important to **rule out an immediate bottleneck in the code**. Azure also allows scaling to services with even better performance, with obviously a proportional increase in cost, but we do not see this as necessary.

4.2 Useful links

- Project github
- WebApp page

- Nebula github
- Medium: introducing nebula, the open source global overlay network
- Nebula doc
- Nebula quick start
- Nebula config reference