ALMA MATER STUDIORUM
UNIVERSITY OF BOLOGNA

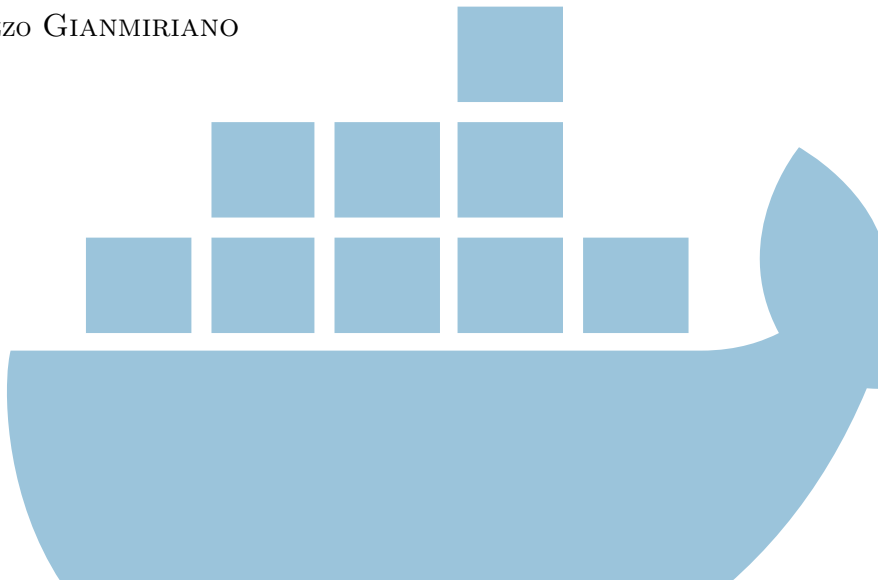
Master in Computer Engineering

Deception Component Generator

LDAP server

A.Y. 2023-2024

Porrazzo Gianmiriano



Index

Introduction

The main aim is to create a deception component generator for a ldap server. Defensive deception is one of the methods that cybersecurity experts have started to use. This technique consists in creating fake services and components that appear as valuable targets to attackers. In that way defenders can divert the attacker's attention and resources away from critical assets.

This approach is known as **Cyber denial and deception** and has as main focus to delay the attack operation, understand the tools and techniques used by the attackers and push the threats into a safe area.
By the definition given, deception's goals are two-fold:

- provide false or misleading information to the attacker, but capable to give him the belief and confidence that the attack is being performed on real data. This is called *Deception*.
- create uncertainty about the reality of the environment that the attacker is facing, to slow down the attacking operations or lead it to waste its time and resources. That is *Denial*.

Thaks to that attackers spend time and effort trying to compromise these fake elements, leaving less capacity to target the actual valuable asset.

The goal here is to create a deception component for a ldap server. A ldap server could be used to access information about an organization, such as who the employees are and what role they have or also password an personal information if present in the directory.

To reach the goal i used docker to have an easy way to deploy the service and at the same time allow personal configuration and create, update and populate the directory managed by the LDAP protocol.

Now the cybersecurity expert can quickly generate the data, create the docker container, run it and have a distraction for a potential attacker.

In the following i'll describe the whole process in detail.

CHAPTER 1

What is LDAP

Before starting to create the deception component generator, it's usefull to understand what LDAP is.

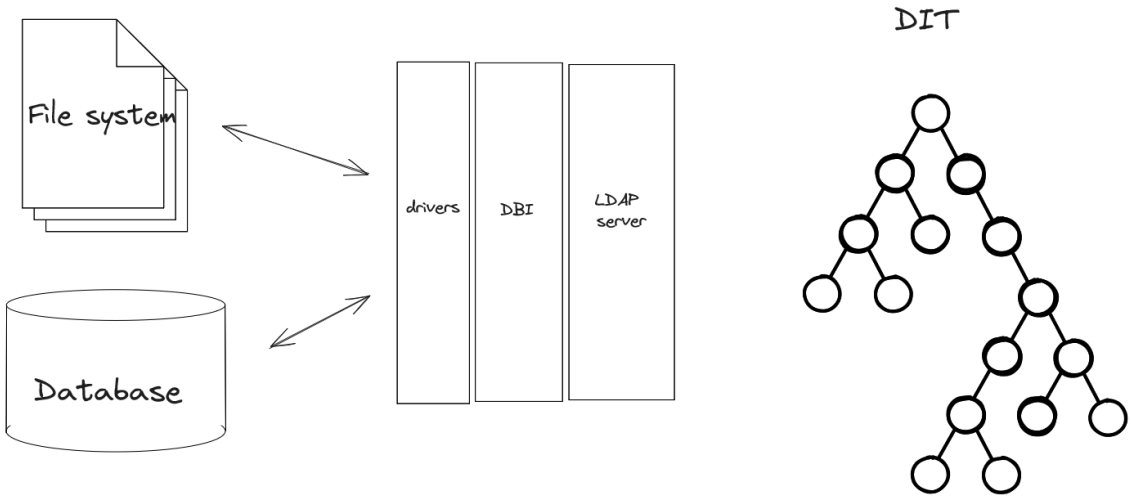
LDAP [1] stands for Lightweight Directory Access Protocol, so it's a protocol to make queries and modify a directory service.
In fact, the protocol does not include a directory service per se, but some implementations of it also have an integrated directory service.
At the beginning, the LDAP protocol was conceived to create an alternative for the DAP protocol, in particular a less resource intensive and lighter one. LDAP was also conceived to use the TCP/IP protocol to make operation related to the X.500 directory easier.

The LDAP protocol has been updated over the years and now it's at the third version.
In the third version of the protocol, the simple authentication and security layer (SASL) was added, which is a framework for authentication and security in the internet protocols.

1.1 How it works

As previously mentioned, the directory could be a generic one, that is possible thanks to the interface, called DataBase Interface easily implementable in a module to guide the beckend.
The DBI manages a collection of entry or objects. The entries are a composed by attributes, which have a type and one or more values. Every entry has a Distinguish name (DN), that also indicate the position from the root.
The entries are also hierarchically ordered, and the final structure is called DIT (Directory Information Tree) and is a way to store the representation of the DB managed by the DBI.

Figure 1.1: How LDAP works



It is used a hierarchy because LDAP was born as a system to organize data about people and resources

inside a company. The hierarchy also allows us to make direct links to the data, an easy partitioning system to administrate, control the access and locate easily data.

An entry is a collection of the attributes, in which the value of it is stored directly, and it's labeled by the type of itself.
Between the various possible attributes, two must be indicated: dn and one or mode objectClass. This format is also used to exchange entries between client and server, and is called LDIF (LDAP interchange format).
For inserting an entry it's necessary to have well formatted objects and a uniform view of the data, commonly used between all the users. This is the function of a SCHEMA. A schema is a set of rules that describes the data and have two types of definitions:

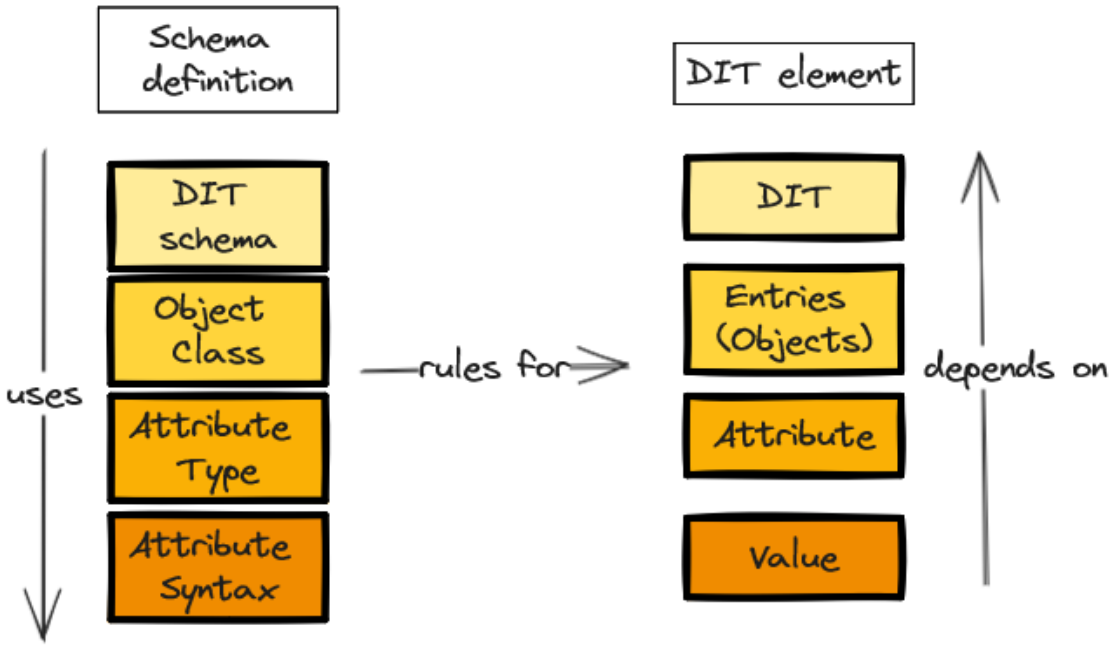
- ObjectClass
- attributeType

Every entry is based on one or more objectClass, which describes the types of attributes that must or should be in the entry. Every attribute has its own type, with also the set of rules necessary to compare them.
The attribute type can be one of the commonly used or can also be described by the user, but in that case, it must be defined in a file and added to the server configuration. It can be done using the ldif format or a schema file.
This is valid also for objectClass.

```
olcAttributeTypes: (...)
    DESC 'name'
    EQUALITY caseMatch
    SUBSTR caseMatchSubs
    SYNTAX ....
```

Listing 1.1: Example of definition of an attributeType (for objectClasses is similar)

Figure 1.2: Correspondance between schema definition and DIT elements



One famous implementation of LDAP is OpenLDAP [2], which includes the Access protocol and a directory. This is the one choosen to create the docker image to configure.

CHAPTER 2 | Docker

Docker [3] is a set of PaaS products that uses OS virtualization to deliver software in packeges called containers. A container is a unit of software that packages up software and its dependencies, so the application runs and could also be shared.

To develop a docker container there are several options: one is to create a Docekerfile in which we can detail all the specifications and the features that we want to run in that container. Usually that is to run a single application in a sandbox environment.
In the study case the application is an OpenLDAP server. To run that it's needed an operating system: the choice i made is to use a debian image.

2.1 | Creation of the container

```
# Image
FROM debian:buster-backports
USER root

ENV DEBIAN_FRONTEND=noninteractive
ENV LDAP_DEBAUG_LEVEL=256

# Configuration variables

ENV DATA_DIR="/init/data"
ENV CONFIG_DIR="/init/config"

ENV LDAP_DOMAIN=example.com
ENV LDAP_ORGANISATION="Example, Inc"
ENV LDAP_BINDDN="cn=admin,dc=example,dc=com"
ENV LDAP_SECRET=admin

# Install and updates
RUN apt-get update && apt-get upgrade -y && apt-get install --no-install-
    recommends -y \
    slapd \
    ldap-utils \
    ldapscripts \
    schema2ldif \
    rm -rf /var/lib/apt/lists/*
RUN apt-get purge python* -y

# update python
RUN wget https://www.python.org/ftp/python/3.11.1/Python-3.11.1.tgz
RUN tar -xvf Python-3.11.1.tgz
```

```
RUN cd Python-3.11.1 && \
    ./configure --enable-optimizations && \
    make altinstall

# Ollama
RUN pip3.11 install --upgrade pip && pip3.11 install langchain
RUN curl https://ollama.ai/install.sh | sh
# Copy generated files to the container
COPY ./init /init

# Expose the LDAP port
EXPOSE 10389 10636

# Command to start ldap server
CMD ["/bin/bash", "/init/init.sh"]
```

Once the base image is chosen, it is necessary to install the package that will be used. The list of packages is shown in the figure. In particular slapd is the daemon that runs the ldap server, ldap-utils contains all the tools to search, add, modify and delete entries from the server from the outside. The others contains usefull tools that will be used in the next steps.
After that it is made a copy of the deception and configuration data inside the conteiner, and are exposed two ports for reach the container from the outside. Two ports because one is for the base TCP connection (10389) and one for the connection using TLS (10636).
At the end of the Dockerfile it's launched a script that configures the server.

2.2 | Configuration scripts

The script is writen in bash and call starts and configure a daemon used in the data generation and calls two other scripts:

- the first is used to reconfigure the server
- the second is used to run the server using the specified ports

In the first scripts are present four functions:

- *reconfigure_slapd()*
- *configure_admin_conig_pw()*
- *load_initial_data()*
- *convert_schema()*

The first one configures the server using the info submitted in the Dockerfile. This step is needed because when slapd is installed it creates a default config, so to modify the dafault paramenters we use that function.
The *configure_admin_config_pw()* insert the info of a specific ldif file that contains data about the adimn.
load_initial_data() finds all the ldif files inserted into the copied folder and add them following a specified order (in our case the order is taken from the file names).
The last function converts the .schema files into .ldif definitions, using the tool **schema2ldif** [4] and add them into the directory.
After that configuration steps the server is created. To allow it to run on the exposed ports, it's stopped and reactivated using the following script.

```
#!/bin/sh

set -eux

reconfigure_slapd(){
echo "Reconfigure slapd..."
    cat <<EOL | debconf-set-selections
slapd slapd/internal/generated_adminpw password ${LDAP_SECRET}
slapd slapd/internal/adminpw password ${LDAP_SECRET}
slapd slapd/password2 password ${LDAP_SECRET}
slapd slapd/password1 password ${LDAP_SECRET}
slapd slapd/dump_database_destdir string /var/backups/slapd-VERSION
slapd slapd/domain string ${LDAP_DOMAIN}
slapd shared/organization string ${LDAP_ORGANISATION}
slapd slapd/purge_database boolean true
slapd slapd/move_old_database boolean true
slapd slapd/allow_ldap_v2 boolean false
slapd slapd/no_configuration boolean false
slapd slapd/dump_database select when needed
EOL

    DEBIAN_FRONTEND=noninteractive dpkg-reconfigure slapd
}

configure_admin_config_pw(){
  echo "Configure admin config password..."
  adminpw=$(slappasswd -h {SSHA} -s "${LDAP_SECRET}")
  adminpw=$(printf '%s\n' "$adminpw" | sed -e 's/[\/&]/\\&/g')
  sed -i s/ADMINPW/${adminpw}/g ${CONFIG_DIR}/configadminpw.ldif
  ldapmodify -Y EXTERNAL -H ldapi:/// -f ${CONFIG_DIR}/configadminpw.ldif -Q
}

load_initial_data() {
    echo "Load data..."
    dataf=$(find ${DATA_DIR} -maxdepth 1 -name \*_\*.ldif -type f | sort)
    for ldif in ${dataf}; do
        echo "Processing file ${ldif}..."

        base_dn=${LDAP_BASEDN:-}
        if  [ ! -z "${base_dn}" ]; then
            echo "updating base dn dc=example,dc=com -> ${base_dn}"
            sed -i "s/dc=example,dc=com/${base_dn}/g" "${ldif}"
        fi

        domain=${LDAP_DOMAIN:-}
        if  [  "${domain}" != "example.com" ]; then
            echo "updating emails @example.com -> @${domain}"
            sed -i "s/@example.com/@${domain}/g" "${ldif}"
        fi

        ldapadd -x -H ldapi:/// \
          -D ${LDAP_BINDDN} \
          -w ${LDAP_SECRET} \
          -f ${ldif}
    done
}
convert_schema(){
  echo "converting schemas..."
  suffix=".schema"
  local data=$(find ${DATA_DIR} -maxdepth 1 -name \*_*\.schema -type f | sort)
  for schema in ${data}; do
    echo "converting file ${schema} ..."
    schema2ldif ${schema} > ${schema%"$suffix"}.ldif
    ldapadd -Y EXTERNAL -H ldapi:/// -f ${schema%"$suffix"}.ldif -Q
  done
```

```
}
# Init
reconfigure_slapd
chown -R openldap:openldap /etc/ldap
slapd -h "ldapi:///" -u openldap -g openldap

configure_admin_config_pw
load_initial_data
convert_schema
PID=$( cat /run/slapd/slapd.pid)
kill -INT $PID && sleep 1
```

Once generated the Dockerfile and the relative configuration we can run the container and see that it works.

Figure 2.1: Example of working container

```
t14@t14: /mnt/c/Users/$ docker exec test ldapsearch -H ldap://localhost:10389 -x -b "ou=people,dc=exa
mple,dc=com" -D "cn=admin,dc=example,dc=com" -w admin "(objectClass=inetOrgPerson)"
# extended LDIF
#
# LDAPv3
# base <ou=people,dc=example,dc=com> with scope subtree
# filter: (objectClass=inetOrgPerson)
# requesting: ALL
#

# David Lee, people, example.com
dn: cn=David Lee,ou=people,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
givenName: David
sn: Lee
mail: david.lee@example.com
cn: David Lee

# Ethan Kim, people, example.com
dn: cn=Ethan Kim,ou=people,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
givenName: Ethan
sn: Kim
mail: ethan.kim@example.com
cn: Ethan Kim

# Grace Lee, people, example.com
dn: cn=Grace Lee,ou=people,dc=example,dc=com
objectClass: top
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
givenName: Grace
sn: Lee
mail: grace.lee@example.com
cn: Grace Lee
```

CHAPTER 3 | Data generation

To make the deception component generator a real one, it's necessary to have some data to insert into it. For generate this datas i've used one of the most talked technologies of our time: Large Language Models. In fact this is a task that they do well.
As first step i've had to decide which LLM use: the model choosen is LLAMA2, principally because it's an open source project and it is possible to run it locally. Done that i've used langchain [5] to write a simple python script that generates the data. But before that it was necessary to find a way for use a LLM locally.

3.1 | Problem

The benefit of running it locally have a cost: the precision of the model depends on the machine in which it's running. Eveni if the model is pretrained, depending on some parameters, the model could run better or worse.

To fix this problem i had to choose in which way run the model. So two were the possible approaches:

- use llama-cpp-python package
- use ollama

Figure 3.1: Llama-cpp data sample

```
Here are ten different LDIF files that can be used to create or modify an LDAP server:

1. Creating a new user with the given username, email address, and password:
```
dn: cn=John Doe,ou=Users,dc=example,dc=com
cn: John Doe
email: johndoe@example.com
uid: 1001
sn: Doe
chpass: 1QWxhbG9uIHBhZ2UgY2FtcGFyYW0KMjAyNzAwNDAwMS8=
```
2. Updating the user's email address for John Doe:
```
dn: cn=John Doe,ou=Users,dc=example,dc=com
cn: John Doe
email: johndoe@newdomain.com
uid: 1001
sn: Doe
chpass: 1QWxhbG9uIHBhZ2UgY2FtcGFyYW0KMjAyNzAwNDAwMS8=
```
3. Creating a new group with the given name and members:
```
dn: cn=Sales,ou=Groups,dc=example,dc=com
cn: Sales
mail: sales@example.com
member: cn=John Doe,ou=Users,dc=example,dc=com
member: cn=Jane Smith,ou=Users,dc=example,dc=com
```
4. Adding a member to an existing group for John Doe:
```
dn: cn=Sales,ou=Groups,dc=example,dc=com
cn: Sales
mail: sales@example.com
member: cn=John Doe,ou=Users,dc=example,dc=com

```
5. Deleting a user with the given username:
```
dn: cn=John Doe,ou=Users,dc=example,dc=com
delete
```
6. Creating a new organizational unit (OU) with the given name and location:
```
dn: ou
```
```

3.2 | llama-cpp-python

This is a project born to create an interface to use llama-cpp in python [6]. After setting up the environment, and create a python script to generate data (even inside the docker itself), what we can see is that the data generated are not so heterogeneous and it needs a lot of time to generate them. Another negative aspect is that i've to download the model and insert it in the docker to make it run, so the docker grows in its weight.

3.3 | ollama

To avoid all this problems i have decided to use Ollama [7]. Ollama is a recent created project that-makes users run LLMs locally in a docker-like way. So even in this case you have to download your model, but now we do not need to insert in our docker file.
The idea is to generate data before, save it and then upload them in the docker. Here we can see a more heterogeneous data creation. Even if there are changes to be made to make it works, it's bettere than the previous output.

Figure 3.2: Ollama data sample


```
{
 "dn": "cn=Jane Smith,ou=people,dc=example,dc=com",
  "objectClass":
    [
      "top",
      "person",
      "organizationalPerson"
    ],
  "cn": "Jane Smith",
  "sn": "Smith",
  "givenName": "Jane",
  "mail": "jsmith@example.com",
  "userPassword": "{SSHA}yh3GQ8D+V21P9+KM4X5Ll5B0"
}
{
 "dn": "cn=Bob Johnson,ou=people,dc=example,dc=com",
 "objectClass":
   [
     "top",
     "person",
     "organizationalPerson"
   ],
 "cn": "Bob Johnson",
 "sn": "Johnson",
 "givenName": "Bob",
 "mail": "bjohnson@example.com",
 "userPassword": "{SSHA}yMuK7t3Pw+X8N9+G5p0B42L4"
}
{
 "dn": "cn=Alice Green,ou=people,dc=example,dc=com",
 "objectClass":
   [
    "top",
    "person",
    "organizationalPerson"
   ],
 "cn": "Alice Green",
 "sn": "Green",
 "givenName": "Alice",
 "mail": "agreen@example.com",
 "userPassword": "{SSHA}yN6B3Lj8+C0V7+K6p1Lb9R48"
}
```


After finding ollama, i've started to think on how i could implement it in a python script to make it generate random data when the docker is started.
After reading the documentation online i've found a way to do so:

```
"""
Use ollama in a scipt
"""
import json,os,sys
```

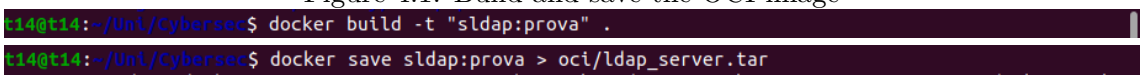
```
from langchain.llms import Ollama

ltemp = {
 "dn": "cn=John Doe,ou=people,dc=example,dc=com",
 "objectClass": ["top", "person", "organizationalPerson"],
 "cn": "John Doe",
 "sn": "Doe",
 "mail": "jdoe@example.com",
 "userPassword": "{SSHA}y6fKt9JVyP1SX2+5UkGX8X+Yq4"
}
olama= Ollama(
    model="llama2",
)
out=olama.predict(f"generate 5 example of an employee of Example, Inc. \nUse
    the following template: {json.dumps(ltemp)}.")

print(out)
lines=out.splitlines()
data = lines[2:-1]
data = [line.lstrip('0123456789. ') for line in data]

with open("out.txt", "a") as f:
    for d in data:
        f.write(d)
        f.write("\n")

with open("out.txt", "r") as f:
    text=f.read()
    lines=text.split('\n')

os.remove("out.txt")
print(lines)
with open("/init/data/23_example.ldif", "a") as f:
    for line in lines:
        if len(line)>1:
            dic = json.loads(line)
            print(dic)
            for key, value in dic.items():
                if isinstance(value, list):
                    for item in value:
                        f.write(f'{key}:{item}\n')
                else:
                    f.write(f'{key}:{value}\n')
            f.write('\n')
```

Analyzing that i've observed that it needs some time to run, but at least it creates a valid number of random data and in a more structured way (as we can see in the pictures reported). In that way it's easier to do some data cleaning to obtain a high quality output.
So now i've created the docker for the service, with the possibility to configure it and generate deception data: the last think to do is try to deploy it.

CHAPTER 4 Deploy

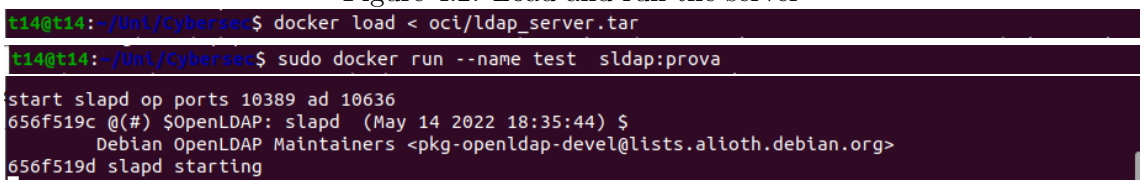
Once understood how create the server and generate the data to insert into it, the last phase is deploy the product.
Before that what i've had to do is create a OCI image, from the Dockerfile, using the docker build, command.

Figure 4.1: Build and save the OCI image



After that i can run the OCI image, as follow:

Figure 4.2: Load and run the server



Now everithing is ready and setted up, so a cybersecurity expert can download the image and run it with randomly generated data, that use a Ollama to generate them.

CHAPTER 5 Final considerations

The creation of a deception component was possible going through different phases.
At the beginning i've had to understand the service that was chosen to be duplicated with random data, after i've had to understand how create a Dockerfile to create it and make configuration possible and, at the end, the main point of all is how to generate credible data to insert into it.

This was the main task, because some LLM runned locally performed not so good for what i was aimning to do. So to solve that i gave a quick look at the possibility to define a grammar in llama-cpp-python, to get a precise output. Unfortunately even doing so the generated data were not so credible.
After some research i've found the Ollama project and i've tried to use it for that purpose. Even if in that case there's no possibility for use a specific grammar for the output, i've obtained a more credible output and in a more ldif compatiple syntax.

At the end of all i've created the OCI image for the LDAP server with all the needed features.

One consideration that i can made about this is that using LLM in a docker make it grow in size a lot, because it has to have the model installed to run it and generate the data. All that can make the installation slower and the container pretty heavy. One solution that could be applied is to have a different machine or a different container, that act like a server, in which run the choosen model and make requests to it to make it generate data. In that way we could have lighter and faster containers.

Except for this detail, now the container is finished and everyone can, using some commands, create a fake LDAP server with data in it generated automatically. So we can divert an attacker to target that service instead a real one and make it to loose time and resources doing that.

Bibliography

[1] IETF. “Ldap protocol.” (1993), [Online]. Available: https://ldap.com.

[2] K. Z. et al. “Openldap server.” (1998), [Online]. Available: https://www.openldap.org/.

[3] S. H. et al. “Docker.” (2013), [Online]. Available: https://www.docker.com/.

[4] “Schema2ldif.” (2020), [Online]. Available: https://github.com/fusiondirectory/schema2ldif.

[5] H. Chase. “Langchain.” (2022), [Online]. Available: https://python.langchain.com/docs/get_started/introduction.

[6] “Llama-cpp-python.” (2023), [Online]. Available: https://llama-cpp-python.readthedocs.io/en/latest/.

[7] “Ollama.” (2023), [Online]. Available: https://ollama.ai/.