ADVANCED ATTACK DETECTION IN BUSINESS MANUFACTURING

Master in Data Analytics, Cybersecurity and Cloud Computing

POPBL 1 – January 2020

Authors: Mikele Zurutuza, Ivan Valdes, Alex Ander Tesouro and Xabier Etxezarreta

Supervisor: Alain Perez

The core of this project is the deployment of limited function microservices and honeypots in the cloud infrastructure of AWS. By means of the built network and some data collecting tools, the analysis of host and network data has been possible. Besides, intelligence extracted from the data gathered has been displayed in a dashboard.

Proiektu honen muina funtzio murriztuko mikrozerbitzuen eta honeypoten altxatzea izan da AWS azpiegitura hodeian. Eraikitako sarea eta datuak jasotzeko zenbait tresnen laguntzaz, makinen eta sarearen beraren datuen analisia egin ahal izan da. Horrez gain, interfaze baten erabileraren ondorioz, bildutako datuen adimenean sakontzea posible izan da.

El núcleo de este proyecto es el despliegue de unos microservicios de función limitada y honeypots en la nube de AWS. Por medio de la red creada y algunas herramientas de recolección de datos, ha sido posible realizar el análisis de datos de máquinas y red. Además de una interfaz que facilite la obtención de conocimiento, entre otras cosas.

Contents

1. Introduction

The purpose of this document is to give a detailed description of the project developed in the first semester of the *Master in Data Analytics, Cybersecurity and Cloud Computing*.

The project has consisted of merging the knowledge obtained during the semester by putting in practice the intelligence acquired in each lecture coursed. Therefore, the work that has been carried out has included many Computer Science fields.

The main core has been a specific network built in Amazon Web Services. The use of AWS resources has been essential in order to develop the elements fitting best the needs of the project. The elements included have been different micro services, each of them having a limited function.

Machine learning has also played an important role as selected information has been extracted from the network in order to analyse data. It is worth mentioning the security that has been established in the entire architecture.

2. Main objectives

The project is based on a manufacturing company that makes use of micro services to manage its production. It requires the deployment of a platform to identify advanced security attacks. To do so, several bait machines, known as honeypots, have been deployed in the cloud. This has permit to obtain intelligence not only from attackers attempting to access those machines, but also the intelligence offered.

To this end, the following objectives have been defined:

- Automated creation and deployment of cloud-based infrastructure.
- Honeypots deployment for intelligence gathering.
- Micro services based application deployment and new features development.
- Ensure the high availability and scalability of the deployed services.
- Motorization of all machines to supervise their behaviour.
- Analysis of the collected data with unsupervised machine learning for anomaly detection.
- Real-time monitoring of the data collected on the different machines.
- Ensure the confidentiality, integrity and availability of information.

3. Project development

1.1. Architecture

At this point, we will briefly describe the general architecture of the project. From here, we will describe each part in depth.

The goal of this project is to implement a monitoring system over a cloud-based application and a group of honeypots with data visualization and use the collected data to train an unsupervised model for anomaly detection.

Next, we are going to show a scheme of the different services we have deployed and how they communicate between them to reach the objective.

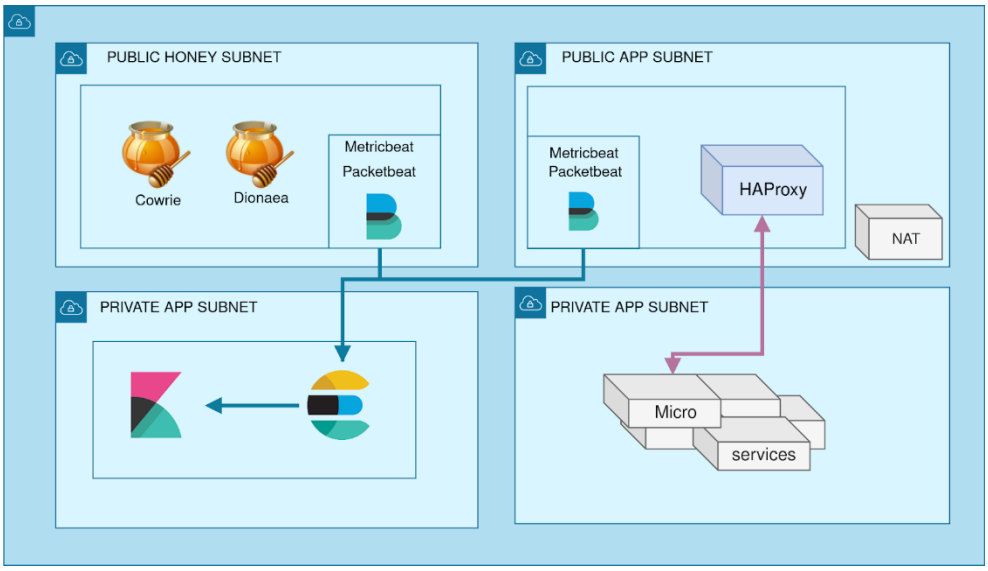

Figure 1: Infraestructure schema.

We have two types of honeypots publicly deployed. Cowrie and Dionaea. In these machines, we have metric beat and Packeteat that collect different data.

On the other public network, we have a NAT server and the api gateway of our application. The api gateway communicates with the private subnet that has a machine with the rest of the microservices. In the machine of the api gateway data is also collected in the same way.

The data collected on both machines is sent to the ElasticSearch service which is on a private subnet. Elasticsearch stores the data and communicates with Kibana to display the data on a real-time dashboard.

1.2. Deployment

The Automatic deployment is a feature that allows an automatic and reliable deployment of an entire application architecture, in order to simplify the deployment process, reduce a possible human error and make the whole deployment process as fast as possible.

It requires a deep knowledge of the result to be obtained and especially of the interaction that each service performs when deployed.

It should be noted that automatic deployment is not a whole process in itself, but rather the orchestration of various processes and functionalities that result in the correct deployment of an infrastructure.

The deployment consist is various differenced phases, in our case we have two differenced ones. The first one would be the phase of installation and activation, this phase consists of creating all the elements of the infrastructure and its implementation. The other phase consists in the removal of an existing deployed infrastructure; this phase can be done automatically or by human request.

The objective of our project includes the creation of an infrastructure that supports the different functions that we have to develop, this infrastructure will be created and developed in the AWS environment and one of the tasks will be to automatically deploy the infrastructure from scratch getting it fully operational without the need for human interaction or any assistance during deployment.

We also want to be able to obtain notifications during the different steps that have been taken and possible errors occurred during the deployment process, as information to simplify the development and be able to response to possible errors.

1.2.1. Selected tool

To carry out this process there are several tools widely known, documented and used in the professional world to provide solutions of the type of automatic provisioning and deployment. The best known are AWS CloudFormation, Terraform and Ansible. All three solutions are capable of meeting our needs.

In our case, the decision was to use AWS CloudFormation, this service is one of the many services offered by AWS which is ultimately the provider of the rest of our infrastructure and therefore the provider of the architecture to be deployed in an automated way.

CloudFormation allows us to schedule the entire deployment of the elements necessary to create the infrastructure by programming a JSON or YAML file, as we prefer. In the file we must specify each element by means of a name specified by AWS, in addition we will be able to specify the different necessary configuration values.

1.2.2. How it works

In order to use AWS CloudFormation the first step is to develop a file from which the service will read the different components and their configuration for deployment, this document has several different vital parts.

The first part consists of specifying the version of the template, this is necessary because over the years the structure of the file have change due to updates from the AWS development team, so this parameter is used by AWS CloudFormation to identify the syntax and carry out the correct reading and use.

The second part is a description of what will make the template itself, we can put anything in this section, and in order to identify the function of the CloudFormation developed we will set a proper simple description.

The third part is to establish the necessary parameters for the proper functioning as the key that will be used to identify and ensure access to the owner of the resources created or general roles. The key must be in possession of the developer team in order to be able to access the deployed resources.

Finally the section describing the different AWS resources that will be created as well as their configuration and how they are referenced to each other. This part is the largest one because it will have every resource needed.

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Despliegue de red para el PBL v1
Parameters:
  KeyName:
    Description: Key Pair name
    Type: AWS::EC2::KeyPair::KeyName
    Default: popbl_key_ivaldes_saccount
  IAMRoleCW:
    Description: EC2 attached IAM role for CWLogs
    Type: String
    Default: CloudWatchLogRole
    ConstraintDescription: must be an existing IAM role which

Mappings:
  EC2RegionMap:
    us-east-1:
      pblbasicami: ami-062f7200baf2fa504
      AmazonLinuxNATAMIHVMEBSBacked64bit: ami-303b1458
Resources:
  SecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: PBL security group
```

Figure 2: AWS CloudFormation template example.

Once the template for the deployment is created, it remains to interact with AWS for its use. AWS provides the opportunity to use the AWS CloudFront service, which integrates an automatic deployment system based on Stacks.

A stack is a collection of AWS resources that you can manage as a single unit. Therefore, the user can create and delete a collection of resources by using the stacks. All the resources in a stack are defined by the stack's AWS CloudFormation template.

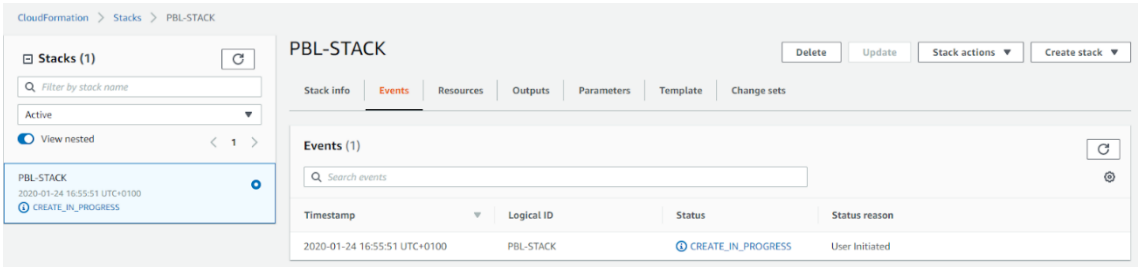

Figure 3: Stack creation.

AWS CloudFormation allows for rollback in case of error. If it is decided to enable this option when launching a stack, in case of an error, everything created by the previous steps of the deployment will be removed. This is possible because AWS CloudFormation ensures that all the stack resources are created or deleted as intended. This happens because the stacks are treated as a single unit so they all mas be created or deleted successfully for the stack to be created or removed.

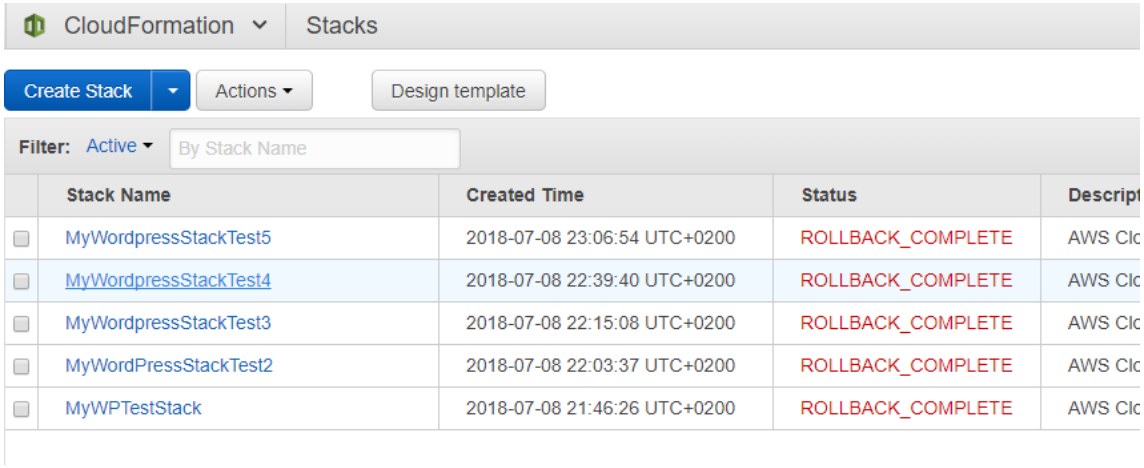

Figure 4: Stack rollback.

In this project, we use CloudFormation for the deployment of the application and we make use of the user-data script of the instances for provisioning. As cloud formation works, when the creation of an instance is complete a signal is sent to CloudFormation indicating that the resource has been created. The problem is that it does not take into account that the execution of the user-data has been executed correctly.

To fix this, we included a creation policy to add a timeout for a signal. If a signal is not received in the specified time the creation of the stack will be marked as failed.

We have created a function in our deployment script that sends a success or error signal depending on the exit code of the previously executed function.

```
function send_cfn_signal {
    echo send_cfn_signal $1 $2
    if [ "$1" = "error" ]; then
        if (($2 != 0)); then
            echo sending error signal
            /opt/aws/bin/cfn-signal --stack $AWS_STACKNAME --resource $AWS_RESOURCE --region $AWS_REGION -e 2
            exit 1
        fi
    fi

    if [ "$1" = "success" ]; then
        echo send success signal
        /opt/aws/bin/cfn-signal --stack $AWS_STACKNAME --resource $AWS_RESOURCE --region $AWS_REGION
    fi
}
```

Figure 5: Signal functionality.

This way when we want an error to be indicated we call this function along with the exit_code. At the end of the script, we send a "success" signal.

1.2.3. Description of the deployed network

The following section will describe the infrastructure with the components defined in the AWS CloudFormation template.

The first thing that is created is a VPC that contains four subnets, in which are the different EC2 that contain the deployed applications.

The subnets are divided into two private and two public to keep more secure the machines deployed in the private and do not need or should not have traffic from outside the internal network of the VPC.

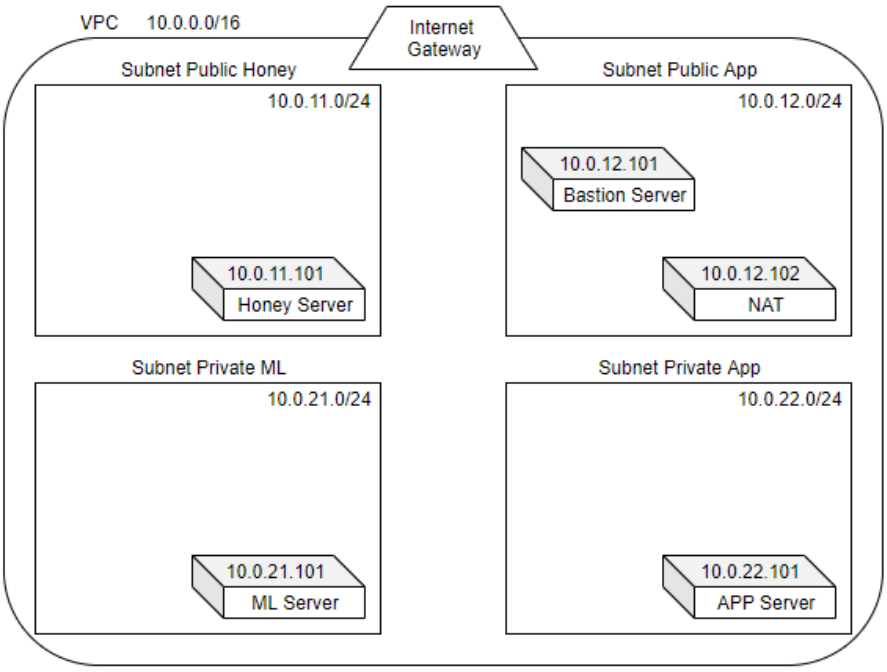

Figure 6: Cloud infrastructure schema.

The different EC2 machines will be described below:

Through the user data installation, the servers will install Docker, Git and Docker Compose. From Git, it will download a parameterized script that can be specified with a word that you want to deploy, this is possible because the script knows what to deploy on each machine.

deploy.sh

```
42
43  # Read Arg
44
45  TARGET="$1"
46
47  case "$TARGET" in
48
49      haproxy)
50          pushd /home/ec2-user
51          git clone $APP_GIT
52          send_cfn_signal error $?
53          cd servicesapp_popbl
54          cp dot_env_example .env
55          docker-compose up -d haproxy
56          send_cfn_signal error $?
57          popd
58          send_cfn_signal success
59
60      ;;
61
62      app)
63
64      ........
```

Figure 7: Parametrized deployment script sample.

Bastion server:

It consists of an instance of the t2.micro type that runs a completely clean aws-linux image. In this case the server will deploy the services that collect data (in this case the sniffer) and the haproxy.

Application server:

It consists of an instance of the t2.medium type that runs a completely clean aws-linux image. In this case, the services installed will be the VAULT secret manager and all the microservices that make up the general application.

Honey server:

It consists of an instance of the t2.micro type that runs a completely clean aws-linux image. In this case the service that will be installed and start up the HoneyPot services.

Machine Learning Server:

It consists of an instance of the t2.micro type that runs a completely clean aws-linux image. In this case, the services that make up the machine learning section will be installed.

All the machines that run the user data will redirect all the output generated by the next commands to a log file as a first step, this file will be stored in /var/log/user-data.log.

In addition to the resources created and configured in the process of creating the Stack resources must be obtained from the Internet, such as updates or applications developed to be deployed on the machines created, to achive this, we use the content of the User Data defined in the template, this content can be found in the configuration section of each of the EC2 equipment.

The User Data is a section that is responsible for executing the commands specified in its content as soon as the machine starts, it's used as a script that is launched in order to perform a task before anything else. It is integrated with the creation of the machine so the possible errors that give the User Data will trigger the rollback and will mark the creation of the Stack as failed.

In this case, we use the userdata for different basic functions like, updating and downloading resources, and launching the different scripts that contain the application launch requirements and procedures.

This process is followed on all our EC2 server machines with the exception of the NAT server, which uses the User Data to add some rules to the iptables in order to display the kibana dashboard from the public part, acting as a reverse proxy.

As mentioned before, the deployment of applications is carried out with GIT, is through GIT where we get a parameterized script that is responsible for the creation, configuration and launch of applications that must be deployed within the machine in question. These applications are also hosted in GIT.

The script developed also contains the ability to issue signals that are interpreted by AWS CloudFormation and can indicate the failure or success of a procedure, this is done because the User Data verifies whether it has been successful execution of the processes specified within it but not if these processes have been completed successfully.

Therefore AWS CloudFormation will mark as successful the creation of an EC2 machine before it completes the script that is ordered to run from the User Data, as this is not the desired behavior, a creation policy that allows us to specify a time during which the launch of the stack will wait to validate or not the creation of an EC2 machine has been configured in the temple of CloudFormation. In our case, a time of 10 minutes has been specified, since it is the maximun time that the script should take to do all

the work. During these ten minutes, the creation of the stack will still be able to send signals to the stack creation, so in case of error the rollback functionality could be activated.

```
HoneyServer:
  Type: AWS::EC2::Instance
  CreationPolicy:
    ResourceSignal:
      Timeout: PT10M
```

Figure 8: Creation policy with 10 minutes timeout.

1.2.4. CloudWatchLogs integration

One of the most common problems during the deployment of the infrastructure is the need to connect via ssh to the machines in order to read the logs generated by the userdata. This process slows down the deployment process considerably. For this reason, we decided to look for a solution to this problem.

By using the CloudwatchLogs [1] service, we can centralize all the logs in AWS and see how the execution of the provisioning script has gone in each of the machines.

To do this it is necessary to install a package on each machine and add a configuration file indicating which file will be read and in which group of awslogs we want it to be saved.

```
[/var/log/user-data.log]
datetime_format = %b %d %H:%M:%S
file = /var/log/user-data.log
buffer_duration = 5000
log_stream_name = /var/log/user-data.log
initial_position = start_of_file
log_group_name = {hostname}
```

Figure 9: awslogs.conf configuration file.

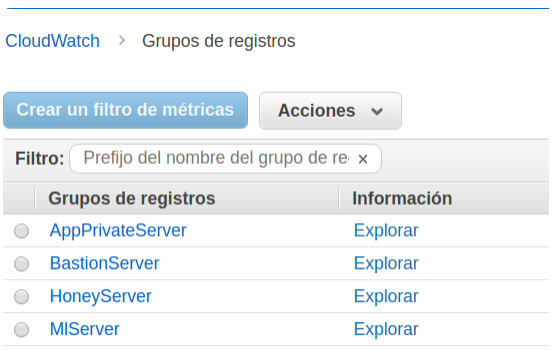

Figure 10: Collected log groups.

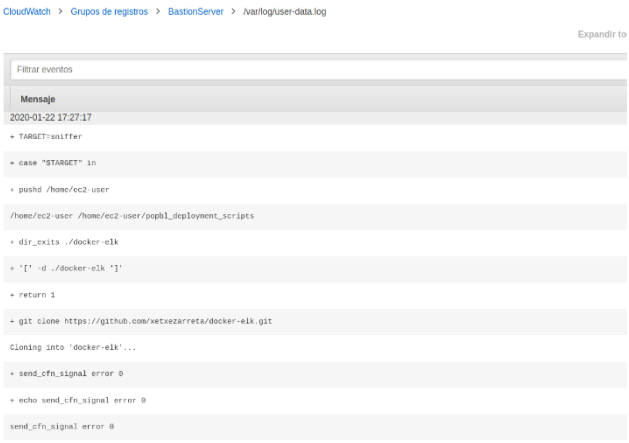

Figure 11: Collected user data example.

1.3. Micro services

In order to build the architecture of the project, we have implemented an application that follows a microservices oriented architecture. This architecture has been worked on throughout the semester in the subject of Advanced Software Architectures. In this point, we will describe what the application consists of, what its functionality is, and how the different patterns we have worked on have been implemented.

1.3.1. Proposed solution

The application simulates a machine that is capable of manufacturing a unique type of piece. Customers can register and enter money in the application to be able to order pieces from the machine. These orders will then be sent to a specific address. This application has been part of an exercise carried out during the semester. For this project, certain aspects have been improved and new functionalities have been added.

La aplicación está compuesta por los siguientes servicios.

- Auth: This service allows the user to register into the application and authenticate later to use the rest of the app.
- Delivery: This service is in charge of creating the deliveries.
- Logger: This service stores and shows the logs of the application.
- Machine: This service simulates a machine manufacturing pieces.
- Order: This service allows the user to create orders.
- Payment: This service allow the user to add money to their account.

To make easier the implementation of the services we will use several third party services:

- **Haproxy** [2]: This service will be running between the microservices and the final client. It is the responsible of creating just one endpoint (this is the endpoint that the user has to know) and redirect all requests to the certain microservice depending on the URL path. This solves multi-endpoint problem. With HAproxy, services only need to know the haproxy endpoint to communicate with the rest of services.
- **RabbitMQ** [3]: This service will be responsible for creating queues with the publisher and subscriber logic in order to create a communication bridge between different services. The publisher will place a specific message in the desired queue, the subscriber will read the message with the corresponding topic and will execute a callback with the procedure needed to perform with that information.
- **Consul** [4]: This service will be responsible of keeping the ip and the port of all the services. Every service registers in consul and sets the url to perform healthchecks.

1.3.2. Saga implementation

In this type of architecture, the transactions that act on resources of several services have to be managed by using sagas. In our application, we have two sagas: CreateOrder and Cancel Order. When the service is asked through api rest to cancel or create an order, the service creates the orchestrator that is formed by a state machine that guides the necessary calls of the transaction in a synchronous way. These orchestrators are stored in an indexed list to keep track of the transactions.

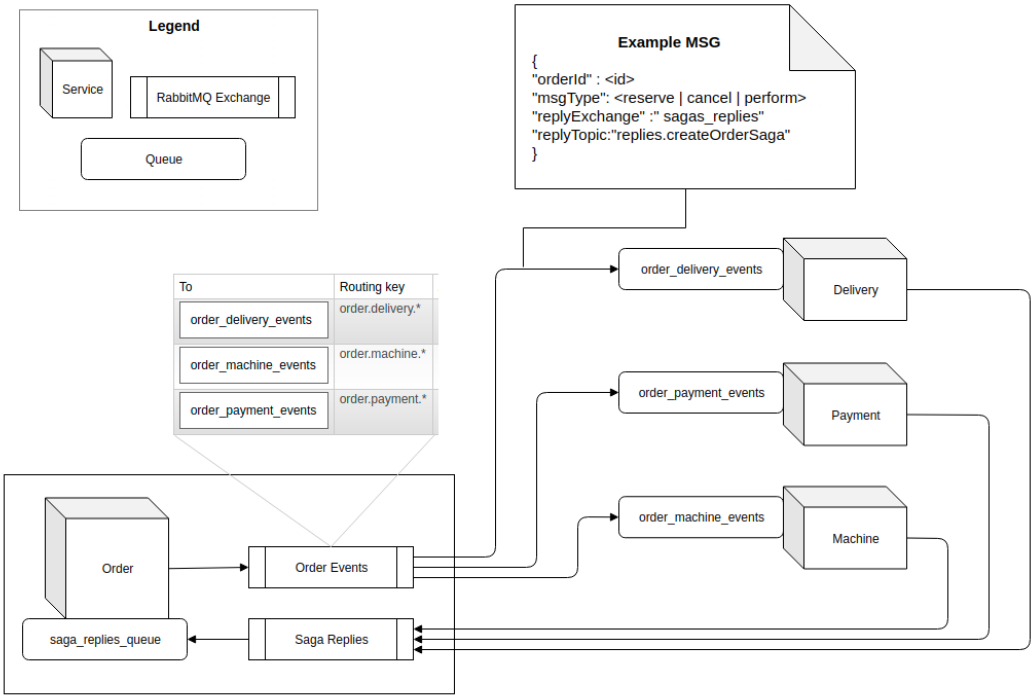

Figure 12: Saga messaging schema.

This diagram shows the architecture of the sagas. The orchestrator publishes the events in the exchange of his service: *order_events*. The topic will indicate which service should receive the message. The services that are part of the saga only have to know the topic that corresponds to them to create a queue and subscribe. In the message is integrated all the necessary information of what the service should do. The action that "msgType" should do and in which exchange and topic they have to reply. All the replies of the sagas CreateOrder and CancelOrder are sent to the exchange saga_replies but with a different topic.

CREATE ORDER SAGA

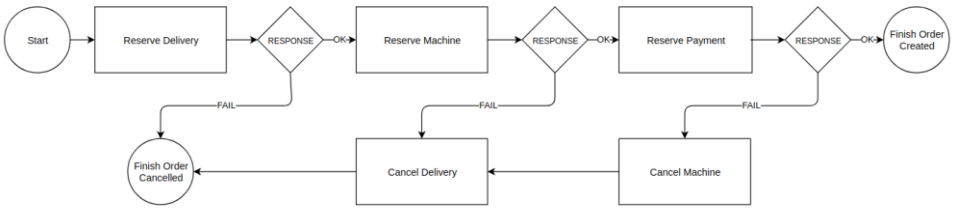

Figure 13: Create order saga state machine.

This saga need to check three services:

- Delivery checks if the country value of the order is in the accepted region.
- The machine simulated queue has a fixed size of 5 pieces. If the number of reserved pieces by other orders and the number of pieces of the current order is greater than the number of free spaces in the queue, we consider the machine queue is full and we send a Fail response in the positive case an ok response is sent.
- Payment checks if the client has enough money.

The order of the stats is not arbitrary. We have set the payment as the last state to ensure that the payment is never cancelled (Pessimistic view).

CANCEL ORDER NEW SAGA

Before starting the saga, we use a semantic lock to avoid collisions between sagas. If the order is pending, we send a “Order processing” message to the client.

```
if order.status == Order.STATUS_PENDING:
    return "Your order is still on our servers please try later"

if order.status == Order.STATUS_CANCELLED:
    return "Your order creation process failed and the order was cancelled"

if order.status == Order.STATUS_FINISHED:
    return "Sorry your order was completed and delivered"

if order.status == Order.STATUS_ACCEPTED:
    # Start Saga
```

Figure 14: Cancel order Saga semantic lock.

Then, if the order is accepted, the CancelOrder Saga can start. As with the other saga, the requests are sent synchronously.

- Machine: The order pieces are removed from the queue and from the machine.
- Delivery: The delivery is removed for the database.
- Payment: The money is returned back to the client.

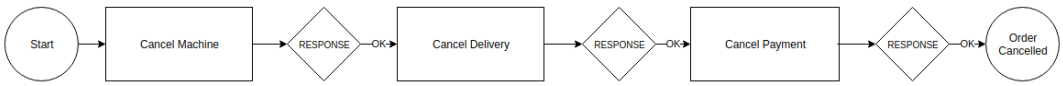

Figure 15: Cancel order saga state machine

1.3.3. Messaging

In addition to the sagas, we use messaging to keep the track of an order and update the order entry of the order service when the machine finishes the pieces.

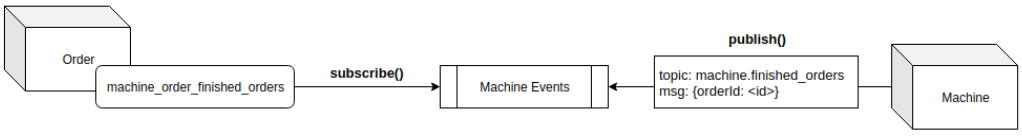

Figure 16: Order finished messaging schema.

1.3.4. Service discovery

When there are using lots of microservices with high availability and automatic scalability, IP address configuration used to be difficult to manage.

Each microservice have to know the others IP addresses. If one microservice goes down and another one is up to replace the previous one, the IP address of the old one is mandatory to be updated by the new one. But not just in one microservices, all other microservices have to know that new IP address. And the way to accomplish this manually is really hard to maintain.

Because of that, a service discovery pattern has be implemented.

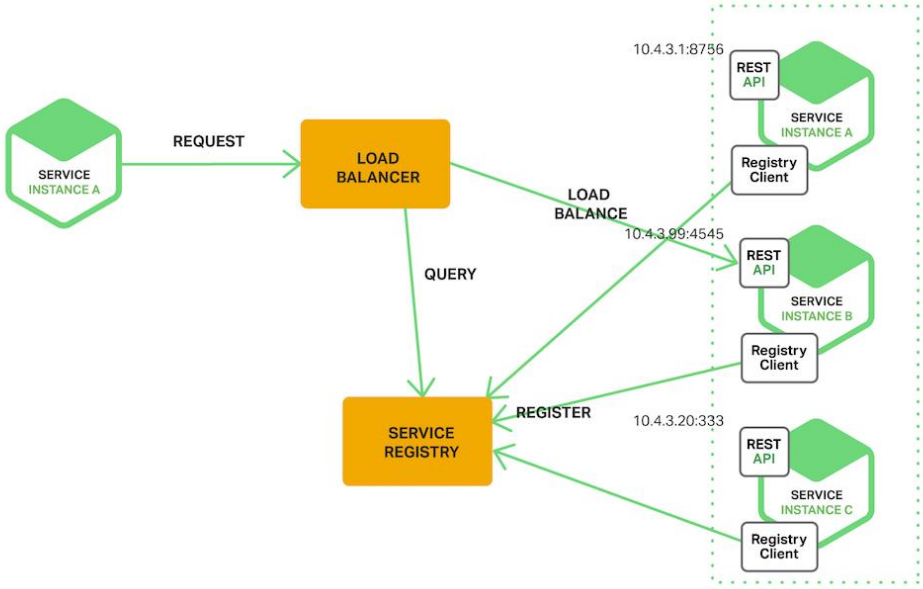

Figure 17: Service discovery schema.

Here we have a new agent in our infrastructure. Service Registry. This is going to be the owner to know which microservices are available and what kind of microservice is (in case of the replication). Consul is the implementation that we have used about Service Registry.

To understand how this architecture works, we are going to divide in some steps. Imagine that there is no service available in this moment.

Step 1. Service instance A is going up.

Instance A is registered in Consul, assigning its IP address and "**payment**" microservice type. Now Consul knows that Instance A is the payment microservice with one IP address. Also knows the health check URL to be used by Consul.

Step 2. Load Balancer information gathering.

HAProxy ask to Consul about "**payment**" microservice. Consul response the IP address and the health check URL to be used by Consul and now by HAProxy too.

Step 3. A client's payment request arrives.

A client start using the API. A payment request is arrived to HAProxy and HAProxy uses the IP that have been gathered before by Consul, to forward the client request.

Some things that have to be in account.

- If Consul goes down and HAProxy cannot request more information about, is going to use previous cached information.
- HAProxy and Consul are going to use microservices health checks for their own purposes.

- When a new microservice is up, it has to be registered in Consul to be able to be used by the HAProxy.
- If any microservice has to request something to another microservice, it is going to use Consul in order to know the IP address.

In other words. Consul works as a DNS . Microservices just know the name of other microservices, it is not needed to know more about that. With the microservice name, Consul is going to provide the rest of the information.

Figure 18: Consul dashboard screenshot.

1.3.5. Logging

The objective is to make a service that provides an exchange where all other services could publish with their specific binding key any kind of logging messages. These messages will be stored by the logger service and they will be accessible via API REST. This enable us to centralize the logs and debug faster.

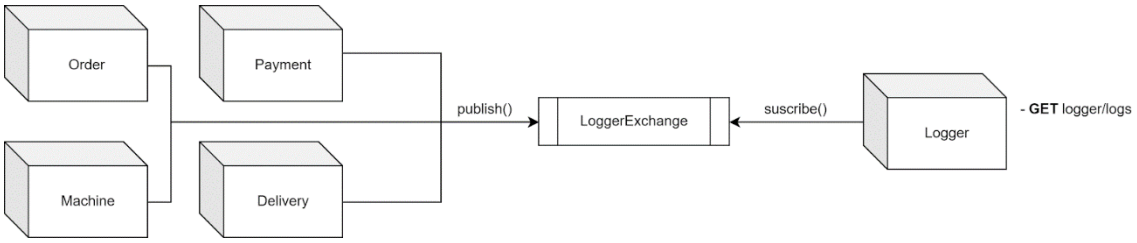

Figure 19: Logger service structure.

We have defined this data structure to store the logging messages.


```
{
    "creation_date": "Sat, 14 Dec 2019 14:17:50 GMT",
    "id": 2,
    "level": "INFO",
    "msg": "Payment Reserve: {'clientId': 2, 'quantity': 10, 'orderId': 14, 'msgType': 'reserve'}",
    "service_name": "Order",
    "timestamp": "2019-12-14 14:17:50.355943",
    "update_date": "Sat, 14 Dec 2019 14:17:50 GMT"
},
```


Figure 20: Example of logger stored message.

1.3.6. Security

Auth Service

Any user who wishes to use the service that our system offers must log in using the authentication service. This service is a microservice that receives a user and a password from an existing user and will validate it to generate an access. A new user can register its credentials using this microservice as well.

Taking this into account, a user with correct credentials could log in through the authentication service, however, in order to use the rest of the microservices, it should be validated as well.

To be able to abstract the validation process in our microservice architecture, the authentication service is not used to validate the user every time he needs to use a microservice, instead, after logging in, he offers a JWT token that both, the client, as access key, and the microservices, will store. In this way, the user can send that password along with each message and all microservices will be able to validate it if the JWT match.

The registration process:

This process consists of sending one to the registration section of the authentication service a user and a password, if the user does not already exist, the service will proceed with adding it to the database, if it exists, it will issue an error and the process will end. Before adding the user to the database, a basic rol is set and the password gets hashed, is after this data attunement when the user is added to the database.

The authentication process:

This process consists of receiving the username and password of a possible user and validating it. If the user does not exist in the database an error will be issued and the process will be terminated, if it exists, the password that the user has sent in the message will be hashed and then it will be contrasted with the one stored in the database, taking into account that the stored one was hashed in the registration step.

If the hashed passwords matches, the login process can be interpreted as correct, in this moment a JWT token is generated and sent to the user as a response to the login process.

The characteristics of the JWT token:

A part of the user data is used to create the JWT token, such as the username and its permissions. In addition, the private key is used together with the rsa256 algorithm to create a token key that can only be interpreted by the server and the user will have to use every time he wants to use a service as a validation method.

Securing RabbitMQ

RabbitMQ message broker does not use cryptography by default but could be implemented over TLS. In the previous stage, all messages between clients and RabbitMQ server were in plain text. Now, those messages are ciphered by public/private key using port 5671.

```
# Service RabbitMQ
rabbitmq:
  image: "rabbitmq:3.8.1-management-alpine"
  hostname: "rabbit1"
  environment:
    RABBITMQ_DEFAULT_USER: "${RABBITMQ_USER}"
    RABBITMQ_DEFAULT_PASS: "${RABBITMQ_PASS}"
    RABBITMQ_DEFAULT_VHOST: "/"
    RABBITMQ_SSL_CERTFILE: "${RABBITMQ_SERVER_CERT}"
    RABBITMQ_SSL_KEYFILE: "${RABBITMQ_SERVER_KEY}"
    RABBITMQ_SSL_CACERTFILE: "${RABBITMQ_CA_CERT}"
    RABBITMQ_MANAGEMENT_SSL_CERTFILE: "${RABBITMQ_SERVER_CERT}"
    RABBITMQ_MANAGEMENT_SSL_KEYFILE: "${RABBITMQ_SERVER_KEY}"
    RABBITMQ_MANAGEMENT_SSL_CACERTFILE: "${RABBITMQ_CA_CERT}"
  volumes:
    - './cert_rabbitmq:/cert_rabbitmq'
  networks:
    lb4_network:
      ipv4_address: '${RABBITMQ_IP}'
  expose:
    - '${RABBITMQ_PORT}'
    - "15671"
  ports:
    - "15671:15671"
    - '${RABBITMQ_PORT}:${RABBITMQ_PORT}'
  restart: unless-stopped
```

Figure 21: rabbitmq microservice docker-compose configuration.

As we have used Docker to deploy this microservice, just setting certificates in the docker-compose file rabbitmq change its own configuration to use TLS.

In order to use SSL/TLS in the web interface, we have to set public/private certs also in the docker-compose. Now, available port for rabbitmq management-gui over SSL/TLS is 15671

All clients need those certs to accomplish successfully the communication with the message broker. As we can see in the image below.

As we would explain later all this secrets are kept encrypted in the vault service.

```
context = ssl.create_default_context(
    cafile=ca_certs)
context.load_cert_chain(certfile,
                        keyfile)
ssl_options = pika.SSLOptions(context, rabbitmq_ip)

credentials = pika.PlainCredentials(rabbitmq_user, rabbitmq_pass)
parameters = pika.ConnectionParameters(
    ssl_options=ssl_options,
    host=rabbitmq_ip, port=rabbitmq_port, virtual_host='/', credentials=credentials)
```

Figure 22: rabbitmq client connection configuration.

A TLS context is defined in the connection process using CA public key and the client private key. From this moment, the message that is going to be sent to the broker it will be cyphered to ensure that nobody can read those messages using any type of sniffer like Wireshark.

Securing Haproxy

HAProxy has been configured to implement SSL/TLS connection with user api client.

In the configuration, frontend api_gateway section, we have defined to use a public/private key cert in the 8443 port.

```
frontend api_gateway
  bind *:80
  bind *:${HAPROXY_PORT} ssl crt /cert_haproxy/haproxy.pem
  redirect scheme https if !{ ssl_fc }
```

Figure 23: HAProxy frontend configuration to use SSL/TLS.

If somebody uses http port 80, the request has been redirected to the 8443 port to ensure that all requests are under SSL/TLS.

Reliability

All microservices have its own health_check URL to ensure all time what is the status of the microservice. This URL is used by the Consul and the HAProxy API gateway.

```
@app.route('/order/health', methods=['HEAD', 'GET'])
def health_check():
    # print("Health check on order")
    return "OK:200"
```

Figure 24: Order microservice health_check URL method.

Reliability

All microservices have its own health_check URL to ensure all time what is the status of the microservice. This URL is used by the Consul and the HAProxy API gateway.

```
@app.route('/order/health', methods=['HEAD', 'GET'])
def health_check():
    # print("Health check on order")
    return "OK:200"
```

Figure 25: Order microservice health_check URL method.

If the microservice is overloaded, this method cannot be accomplished by the microservice, the response status is not going to have an HTTP status code 200 and Consul or the HAProxy are going to know that the microservice is not available for a time period.

auth_service

	Queue			Session rate			Sessions						Bytes		Denied		Errors		Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
authapp1	0	0	-	0	0		0	0	-	0	0	?	0	0		0		0	0	0	0	2h32m UP	L4OK in 0ms	1	Y	-	1	1	0s	-

Figure 26: auth microservice health check by HAProxy.

ID	Node	Address	Node Checks	Service Checks
auth1	0ec1c6840678	192.168.4.2:8000	1	1

Figure 27: auth microservice health check by Consul.

As it can be seen above, Consul and HAProxy are getting a good health of auth microservice.

1.4. Honeypots

In this section, the design and installation of the selected honeypots will be explained.

A honeypot is a computer security system, which is used to detect or study cybersecurity threats, or to track rare, or new, attack methods. They are designed to catch attackers' attention so that they can lead them to execute their malicious routines. By the use of honeypots, not only new means of attack can be discovered, but also unknown vulnerabilities in our own real network can be found, and therefore, it can be useful for designing more effective protection strategies.

In general, a honeypot consists of a system formed by a computer, applications and data that simulate a real system behavior, and it is shown as part of a network; nevertheless, a honeypot is an isolated system which is separately monitored. There is no point in legitimate users accessing them, so any interaction with the honeypots is considered unfriendly.

Depending on deployment, honeypots are classified as these two main types: production honeypots and research honeypots.

- **Production honeypots** are easy to use, gather limited information and are used mainly by companies. These are located inside the production network with other servers by an organization to ameliorate the company's security.
- **Research honeypots** are run to collect information about the techniques attackers used in order to gain access to the system. The difference with the previously mentioned honeypots is that these are used to make a deeper study on the threats received.

Depending on design criteria, there are three types of honeypots: low-interaction, medium-interaction and high-interaction honeypots.

- **Low-interaction** honeypots mimic the services that are frequently requested by attackers. This kind of honeypots make use of considerably few resources. Due to that, multiple machines can be hosted in a single physical system, their response time is relatively small and less code is required, lowering the complexity of the system security.
- **Medium-interaction** honeypots are considered similar to low-interaction with regards to their implementation. They only provide partial implementation of services and they do not permit usual and full interaction with the system.
- **High-interaction** honeypots simulate real system activities. It allows the attackers interact with the system as if they were in a regular operating system. The aim of these honeypots is to collect the maximum amount of information related to the methods attackers are making use of.

In this case, the following two honeypots have been installed and configured:

1.4.1. Cowrie

Cowrie is a medium-interaction SSH and Telnet honeypot designed to log forced attacks and the shell interaction performed by the attacker. This honeypot can simulate a fake file system providing the option for creating and removing files. It also gives the possibility to the owner to add content to files or create fake files, so that the attacker can use the "cat" command in the shell and see whatever is written inside.

Session logs of anyone interacting with the honeypot are stored in a UML Compatible format, in order to study them in a precise way with original timings. It is also worth mentioning that Cowrie is capable of storing the files that have been downloaded from the Internet by means of "wget" or "curl", or the ones that have been uploaded via SFTP or SCP. Cowrie is a system based on Kippo honeypot, but it contains

additional features, which make it more interesting. For instance, it supports SSH execution commands, or it creates a log of all the TCP connection attempts, among others.

1.4.2. Dionaea

Dionaea is a low-interaction honeypot that aims to trap malware-exploiting vulnerabilities. It embeds Python as its coding language in order to simulate protocols. It is able to rise different type of services and wait until attackers take control of those by malicious requests and payloads. It makes use of a library called LibEmu, which detects and evaluates payloads sent by attackers, having as a principal goal to obtain a copy of the malware. LibEmu is used to detect measure and if necessary, execute the shellcode.

These are some of the protocols Dionaea can trap malwares from:

- Server Message Block (SMB) is the main protocol offered by Dionaea and it is a broadly popular target for worms.
- Hypertext Transfer Protocol (HTTP) is supported on port 80 as well as HTTPS.
- File Transfer Protocol (FTP) is provided on port 21. It enables creating directories, and uploading and downloading files.
- Trivial File Transfer Protocol (TFTP) is open on port 60, which is used to serve files.

The honeypots described above have been dockerized and put together in a single machine located in a separate public subnet of the network. A docker-compose.yml file has been created with the appropriate configuration of each honeypot. Each Dockerfile has been adjusted to the honeypots' needs for their correct deployment on the machine. Therefore, Cowrie and Dionaea have been containerized separately.

1.5. Data collection and storage

As mentioned in previous sections, one of the goals to accomplish in this project, has been to collect data from the microservices and honeypots deployed in the cloud. The information obtained it is considered valuable. However, the practice of assembling them can be challenging. That is why an appropriate way of managing the data is essential.

1.5.1. Elastic Stack

In this case, we invoke Elastic Stack (ELK Stack) tool. This is a set of open source resources from Elastic software designed to help users take any data and in any format and search, analyse and visualize it in real time. It consists of three services offered by the family of Elastic: *Elasticsearch*, *Logstash* and *Kibana*. The means by which raw data is collected are called *Beats* and there is a wide variety of types based on the kind of data. In this case, Metricbeat and Packetbeat have been the selected ones. Moreover, the services used in this project, and as a consequence, the ones that will be explained more deeply, are only Elasticsearch and Kibana. The image below summarises the services that have been used with the purpose of accomplishing this task.

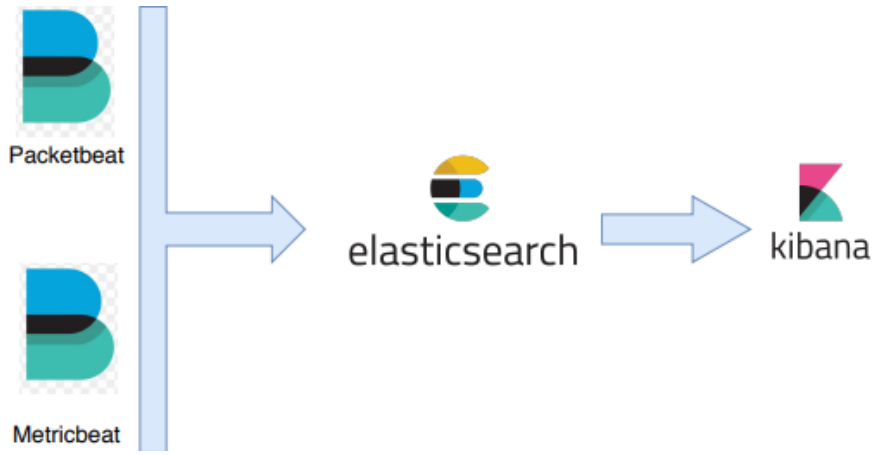

Figure 28: Elastic stack.

Elasticsearch

Elasticsearch is a distributed and open source product focusing on search and analytics engine for any type of data, including textual, numerical, geospatial, structured and unstructured. It is the central component of the Elastic Stack.

Its speed, scalability and ability to index many elements of any type can be highlighted. An index refers to the group of JSON documents that are related to each other. Raw data is collected from a wide range of sources, such as logs, system metrics and web applications. Before indexing the data gathered, there is a process responsible for parsing, normalizing and enriching it. Once it is indexed in Elasticsearch, users can make queries against it and use aggregations to obtain specific summaries.

Logstash

Logstash is the service whose main function is to process data and send it to Elasticsearch. It enables ingesting data from multiple sources simultaneously and modifying or transforming before it is indexed into Elasticsearch.

Kibana

Kibana is used to visualize and manage data for Elasticsearch. It provides real-time graphs, histograms, charts and maps, among others. The data to be displayed is selected from the indexes that have been previously stored in Elasticsearch.

As it has already been mentioned, from the above three services, two of them have been used. The reason why Logstash has been disregarded is the unnecessariness of processing the data before storing it in Elasticsearch.

Beats

In general, beats are lightweight data shippers that are installed on the machines that data is being collected from. This data is then sent to Logstash or Elasticsearch. There are seven types of beats; filebeat, metricbeat, packetbeat, winlogbeat, auditbeat, heartbeat and functionbeat. Each of them gathering different data. In this project, the focus has been set on host and network data. Therefore, the beats used have been metricbeat and packetbeat.

- **Metricbeat:** Metricbeat is usually installed on a server from which operating system metrics or running services metrics are collected. It mainly refers to host data. It gathers information related to the cpu and memory usage of the system.

- **Packetbeat:** Packetbeat refers to network data, instead. This beat collects data from each flow perceived. It offers the possibility to define the interval of a network flow. For example, if in 30 seconds no other packets are received, it will be considered the flow has reached its end. It collects data describing the network flow, including packets and bytes received and sent, the IP from which the flow has begun. In addition, there is the option for adjusting its configuration to assemble further information, such as geospatial data. By its use, information related to the source country, continent or region can be obtained.

  Putting things in context, the beats described have been installed in the honeypots machines and in the bastion machine of the micro services. This way it will be possible to bring together not only host information, including attackers' interaction with honeypots, but also the common traffic going through our network.

1.5.2. Security

It is the security of the means of forwarding data what has been considered a point worth implementing. For that purpose, our own Certificate Authority (CA) has been created and SSL certificates have been used. Therefore, the data collected from the beats installed has been sent to the Elastic Stack in a secure way. The data forwarding process has been divided in two steps; sending data from the metricbeat and packetbeat to Elasticsearch, and directing data from Elasticsearch to Kibana. Both have been secured.

1.5.3. Dockerization

Regarding dockerization, the services mentioned above have been merged and dockerized divided in two groups. On one hand, Elastic Stack tool has been deployed in a machine, where the data management and visualization will be set. On the other, Metricbeat and Packetbeat have been dockerized in selected machines; in the systems from which data is going to be collected.

1.6. Data analytics

1.6.1. Data recollection

In order to detect anomalies in network flows, data has been collected with the Packetbeat tool. Packetbeat automatically detects the packets belonging to the same flow, extracting common features from each of them (e.g. number of packets sent during the whole flow). For Packetbeat, if no packet belonging to an active flow arrives for 30 seconds, it considers the flow closed or terminated. Even so, every 10 seconds it saves information generated by the active flows until that moment.

In order for the data to be representative, a lot of normally behaving network traffic has to be collected from the system. Once normality has been defined, any data outside of that normality will be treated as an anomaly. To define this normality, we first deployed the microservices architecture and placed Packetbeat on the public machine where users access the services offered by the microservices. With a python script, we have simulated the legitimate use of the microservices in order to capture this traffic. In this way, all the traffic coming from the outside to interact with the microservices has been collected. On the other hand, on another public machine, honeypots have been deployed and flow information has been collected with Packetbeat as well. The data collected in the microservices and in the honeypots has been stored separately. These data have been exported in csv format from Kibana for analysis and model generation.

Focusing on the types of data collected, as mentioned above, flow information and its characteristics are collected. In the Packetbeat configuration file, the following protocols have been defined for collection, with the aim of covering as much as possible.

PROTOCOL	DESCRIPTION
icmp	ICMP4 and ICMP6
dhcpv4	DHCP for IPv4 ports
dns	DNS traffic
http	HTTP traffic
tls	TLS traffic

1.6.2. Data processing

Feature selection

Packetbeat collects lot of flow data such as source and destination IP, flow source geolocation information, packet flow information, etc. An attempt has been made to select data that is generic in all flows so that the anomaly detection model does not depend on data such as IPs, ports, geolocation, etc.

For this reason, characteristics related to the number of packets, number of bytes and duration of the flow have been selected, discarding all others. Below are all the selected features of the data collected with Packetbeat.

- **source.bytes:** Number of bytes sent from source.
- **source.packets:** Number of packets sent from source.
- **destination.bytes:** Number of bytes sent from destination.
- **destination.packets:** Number of packets sent from destination.
- **network.packets:** Total packets sended on the flow (source.packets + destination.packets)
- **network.bytes:** Total bytes sended on the flow (source.bytes + destination.bytes)
- **event.duration:** Flow duration.

Data cleaning

There are two options for dealing with Nan and null values. The first method is to remove the rows or columns where these types of values exist. The main problem with this method is that we can lose many data, which we are not interested in. The second method consists on filling the missing values, for example assigning the average of the other values. If we assign values, we have the problem that the outliers can affect this value a lot and we can be introducing a lot of noise in the data. In our case, since we are trying to detect anomalies, we are not interested in deleting the outliers, since we are treating them as an unusual behaviour, as an anomaly.

As the second option was discarded, this problem was solved by eliminating the missing values. This was done in two different steps. First, we removed the columns in which all the values are null or Nan. In our case, the selected features, there was no column with all the missing values. Once this is checked, the rows where there are missing values were delete.

Once removed, we checked the dimension of the new dataset. 26,000 rows have been maintained, with a minimum loss with respect to the dataset with the missing values. Knowing this, we proceed to standardize the data before the generation of the model.

Data standardization

Data Standardization is a data processing workflow that converts the structure of disparate datasets into a common data format. Standardizing data is recommended because otherwise the range of values in each feature will act as a weight when determining how to cluster data, which is typically undesired. Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data.

Considering that the standard metric for DBSCAN is Euclidean distance, known as L2 norm, if one feature feature has a range of values much larger than others, clustering will be completely dominated by that feature.

For the standardization process, the Sklearn function called StandardScaler has been used. This method standardizes features by removing the mean and scaling to unit variance.

$$z = (x - u) / s$$

Where u is the mean of the samples and s is the standard deviation of the samples.

1.6.3. Model generation and selection

As said before, the main idea behind using clustering for anomaly detection is to learn the normal behaviour in the data and using this information to point out if one point is anomalous or not. Centroid based clustering algorithms, for example, K-Means, all points are fitted into the clusters, so if you have anomalies these points will belong to the clusters and probably affect their centroids. This can cause you to not detect anomalies. Because of this, in this project the use of this type of algorithms has been discarded, focusing the work on density-based algorithms.

Parameter tuning

There are two of the most important parameters of DBSCAN: min_samples and eps. Selecting the best values in these two parameters is not an easy task. In our case, we have followed the indications of the paper called 'DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN' [5]. On page 11, section 4.1 Heuristics for Choosing DBSCAN Parameters defines a guideline for choosing these two parameters. Along with this, the results have been drawn with the Plotly tool to contrast the results obtained visually.

Min_samples parameter defines the minimum number of neighbors a given point should have in order to be classified as a core point. It's important to note that the point itself is included in the minimum number

of samples. The paper defines that the min_samples parameter has to be greater or equal to the dataset dimension. In this case, it recommends that the min_samples parameter should be double to the dataset dimension.

$$min\_samples = 2*dim$$

As we have a dataset of dimension 7, the value used as min_samples has been 14.

On the other hand, epsilon or eps defines the maximum distance between two points. Is the value that two points are considered neighbours if the distance between the two points is below the threshold epsilon. We performed visual tests to see how this parameter affected our data, achieving the best result with an epsilon value of 2. Apart from this, we have tried some alternatives to find the best eps value. One of them has been with the OPTICS algorithm. OPTICS is an algorithm for finding density-based clusters in spatial data, its basic idea is similar to DBSCAN. The big difference from DBSCAN is that you do not have to define the epsilon parameter. This parameter is defined only to set an upper limit and control the performance. No successful results have been obtained with OPTICS.

1.6.4. Model validation

In terms of results, they have not been as satisfactory as we had hoped. The main problem has been the generation of data that represent the normal behavior of our network. Although we have used a python script that makes use of the micro-services to generate 'normal' data, we have only been able to capture a limited behavior of our system that does not represent the real normality of it. Even so, below are the results obtained from the analysis carried out.

Clustering visual representation

First, we visually analyze the clustering results obtained with the DBSCAN algorithm. The DBSCAN algorithm has been trained with 26000 instances or network flows. The X-axis represents the duration of the event (flow), and the Y-axis the number of packets transmitted during that flow.

Our data have been grouped in 4 different clusters and some instances, 9 in particular, have not been grouped in any cluster and will be treated as outliers.

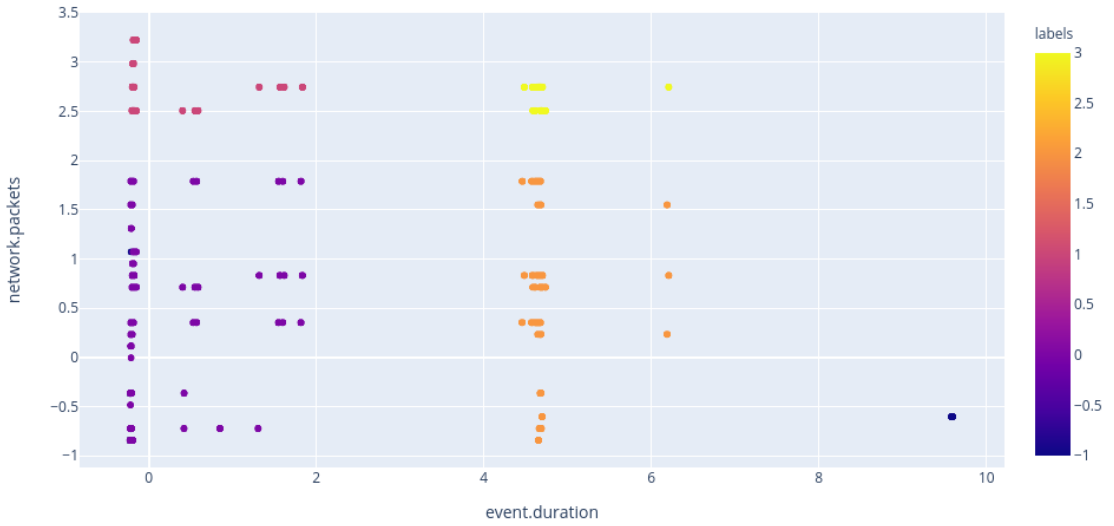

Figure 29: DBSCAN features correlation.

CLUSTERS	NUMBER OF INSTANCES
cluster 0 (purple)	23176
cluster 1 (red)	1588
cluster 2 (orange)	811
cluster 3 (yellow)	270
outlier -1 (blue)	9

Clustering internal analysis

To understand the grouping of each cluster and to know the type of traffic that is grouped in each cluster, we have looked at the source IP and the destination IP of each instance and we have reached the following conclusions.

Cluster number 0 has grouped most of the captured traffic. Analysing the IPs where the flows have originated, the first three IPs are internal traffic of the microservices (10.0.12.101 and 192.168.4.10) and traffic generated with the python script (52.205.255.12). In addition, it has grouped traffic from some IPs that we do not know. This unknown traffic probably is similar to the traffic we have generated. We can conclude that most of the traffic is of this type and this cluster defines a normal use.

```
CLUSTER: 0
10.0.12.101        10844
192.168.4.10       10596
52.205.255.12       1583
54.191.248.14         18
83.97.20.46           11
185.156.73.52         10
92.118.37.61          10
45.77.163.135          8
```

Figure 30: Cluster 0 instances source ip.

On the other hand, cluster number 1, has also grouped legitimate flows generated by us. Most of the flows are those generated by the script. The main difference with respect to the flows of cluster 0 is that the number of packets transmitted in the flows is greater.

```
CLUSTER: 1
52.205.255.12    1582
10.0.12.101         6
Name: source.ip, dtype: int64
192.168.4.10     1582
52.94.233.129       6
Name: destination.ip, dtype: int64
```

Figure 31: Cluster 1 instances source/destination IP.

Thirdly, cluster number 2 and 3 also groups traffic generated by us, the source and destination IPs are known. The difference with the previous clusters is that the durations of the flows are longer.

```
CLUSTER: 2
52.205.255.12      270
10.0.12.101        260
192.168.4.10       254
185.156.73.52       14
125.16.135.50        5
92.118.37.61         4
185.153.198.211      4
Name: source.ip, dtype: int64
10.0.22.101       508
10.0.12.101       297
46.165.221.137      6
Name: destination.ip, dtype: int64

CLUSTER: 3
52.205.255.12    270
Name: source.ip, dtype: int64
192.168.4.10    270
Name: destination.ip, dtype: int64
```

Figure 32: Cluster 2 instances source/destination ip.

Finally, in reference to the outliers, we have IPs with an unknown origin. The characteristics of these flows have not been found in flow generated by us so they would be directly treated as anomalies.

```
CLUSTER: -1
31.135.214.214    4
197.156.71.245    4
52.205.255.12     1
Name: source.ip, dtype: int64
10.0.12.101     8
192.168.4.10    1
Name: destination.ip, dtype: int64
```

Figure 33: Outliers instances source/destination ip.

SHAP for global explainability

To understand why some instances have been grouped in a cluster and others as outliers, first, a library of Explainable Artificial Intelligence called SHAP has been used. This library calculates the SHAP values of each instance, allowing to create global and local explanations in a visual way. To do this, the instances with the results obtained with DBSCAN have been classified, assigning to each instance the value of the cluster (-1 in case of outlier) to which it belongs. According to SHAP, only event.duration, source.packets and source.bytes affect the output of the model, event.duration being the most important of all.

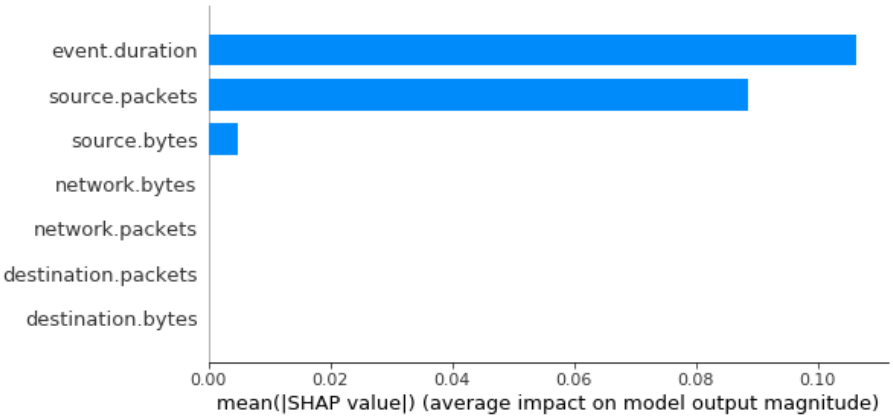

Figure 34: Features average impact on model output.

Silhouette coefficient

Another method to validate the quality of the model has been the Silhouette Coefficient. Silhouette refers to a method of interpretation and validation of consistency within clusters of data. The

silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges from −1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighbouring clusters. We have obtained a value of 0.721.

1.7. Visual analytics

In the aforementioned sections, the practice of collecting, managing and analysing the obtained data has been the point to focus on. However, it is crucial to visualize the results concluded from previous research. The visualization has been split into two main fields; display of host and network data obtained from honeypots and micro services in Kibana, and visual graphs of the representation of the intelligence gained based on machine learning techniques.

1.7.1. Kibana dashboard

The purpose of the dashboard created in Kibana, the visualization tool belonging to Elastic Stack, has been to give a visual representation of a detailed and selected evidence of the data collected from our network. In the following lines, each element of the dashboard will be explained and justified.

Memory usage

With the aim of showing the memory used in the system in real time, creating a time-series graph has been considered an appropriate option. In a single graph have been displayed both, the memory used in the machine where micro services are deployed and the one where honeypots are deployed. The colour has played an important role in making the difference between the data type; the orange refers to the metrics obtained from honeypots, whereas the blue refers to the ones obtained from the micro services. The difference can be seen when hovering on each line of the graph.

Regarding the scale, it is represented based on percentages, and two scales are shown on both sides, the same. However, the numerical values on the right cannot be seen entirely, even if we make a bigger graph.

Figure 35: RAM usage plot.

The memory being used in real time has also been represented by means of percentages. The colours have been kept the same as in the graph combining both values. The scale in this case is the same, and the higher the value, the more filled is the half-circle.

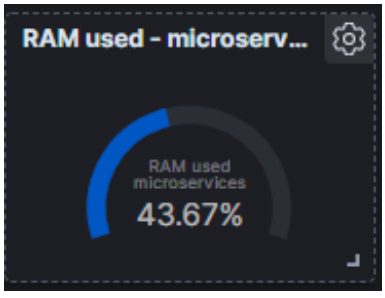

Figure 36: Microservices RAM usage gauge plot.

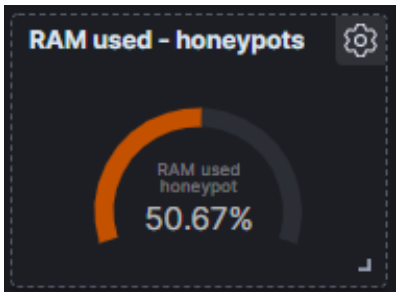

Figure 37: Honeypot RAM usage gauge plot.

CPU usage

The amount of CPU that is being used is also displayed the same way as the RAM memory. The orange and blue colours have represented the same as in the graphs above, with the aim of not mixing concepts.

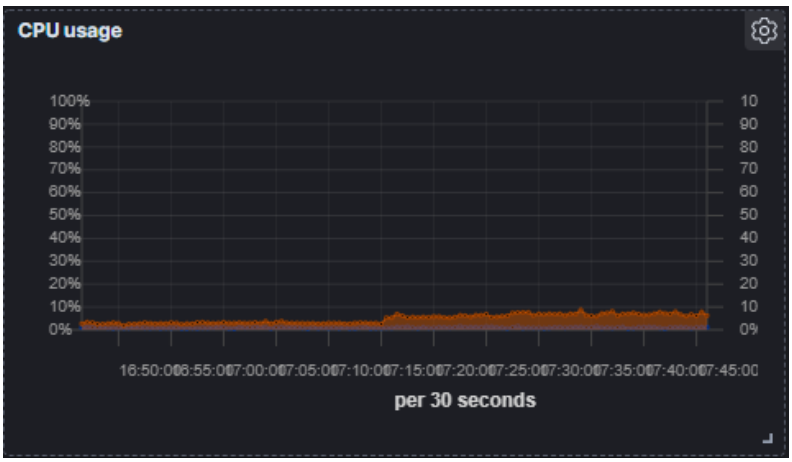

Figure 38: CPU usage plot.

Apart from the time-series visualization, the actual value of the percentage referring to the CPU usage has also been shown, the same way as the RAM memory.

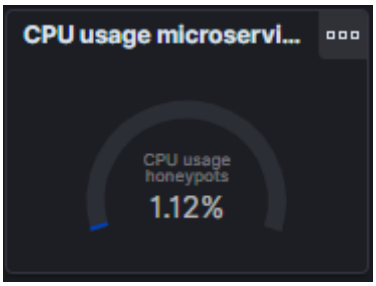

Figure 39: Microservices CPU usage gauge plot.

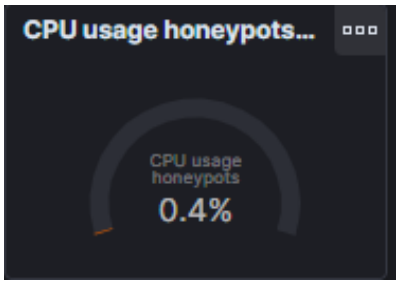

Figure 40: Honeypots CPU usage gauge plot.

Source and destination IPs

In order to have notice of what is the IP of source and destination, a table showing each machine's source and destination IPs are shown, respectively. There is a third column related to the sum of packets received in the honeypots and micro services, which has been considered a relevant, when describing the network flow.

table - microservices

Source IP	Destination IP	Sum of packets
192.168.4.10	10.0.22.101	3,752,626
10.0.12.101	10.0.22.101	1,876,313
10.0.12.101	10.0.21.101	1,814,414
10.0.12.101	169.254.169.123	279
10.0.12.101	52.46.157.70	90
10.0.12.101	106.13.237.44	34

table honeypots

Source IP	Destination IP	Sum of packets
10.0.11.101	10.0.21.101	358,115
10.0.11.101	169.254.169.123	273
10.0.11.101	52.94.225.93	80
10.0.11.101	106.13.45.212	62
10.0.11.101	10.0.0.2	54
89.248.172.105	172.18.0.3	912

Figure 41: Packet flows on microservices and honeypots.

Source of attacks

Bearing in mind that the beats gathered include geospatial information, by the use of a map, the countries from which attacks are originated have been specified. Colours depend on the sum of source packets that have been received as shown in the ranges at the bottom right of the map. The more packets the honeypots get, the darker the colour of the country.

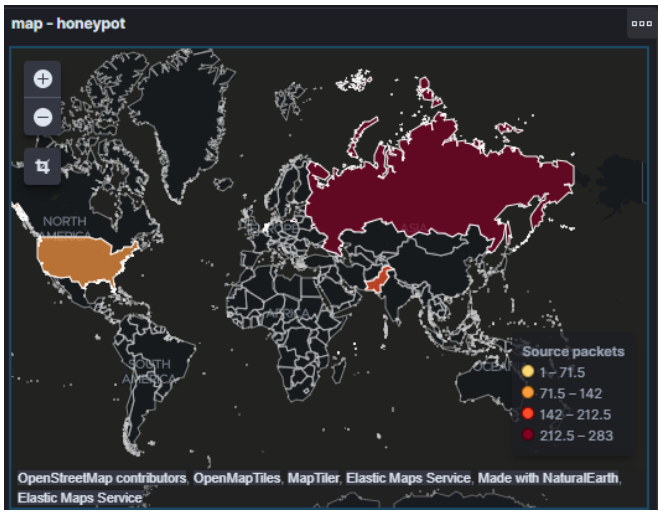

Figure 42: Honeypots map attack.

Inbound and outbound traffic

To finish, it has also been worth considering the representation of the traffic coming in and out of the machines. To do so, we have opted for simple metrics in bytes. Regarding visual variables, the size of the values and texts displayed have played an important role. The actual inbound and outbound traffic are shown in a bigger size, as they refer to the relevant value in time. Below the actual value, it is shown the total traffic that has come in or out, so that is considered to be like secondary content in terms of visualization.

Referring to the dashboard itself, these metrics are located exactly below the tables that have been previously talked about.

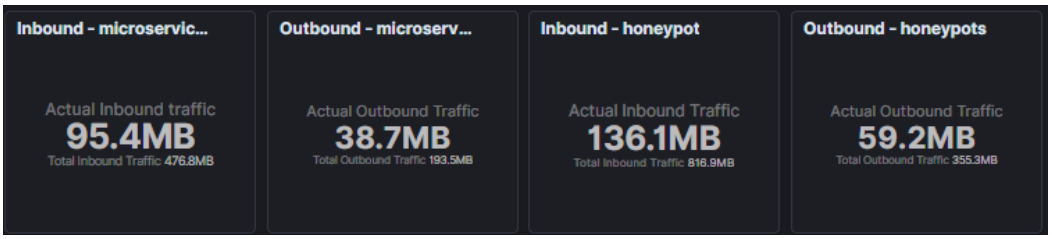

Figure 43: Inbound and outbound traffic.

1.7.2. Plotly

Displaying data can provide a visual interpretation of the evidence that we are working on, as well as it can contribute to acquiring knowledge of the analysis that it is being conducted. For this purpose, we invoke the *Visual Analytics Process*.

Visual Analytics refers to the science of analytical reasoning by interactive visual interfaces. It consists of the interaction between data, visualizations, machine learning models, and user knowledge, with the aim of getting knowledge. As it is shown in the image below, the loop in which the collaboration of the rest of the elements is crucial stores the intelligence got from the analysis and contributes to reaching faster and probably better conclusions in the future.

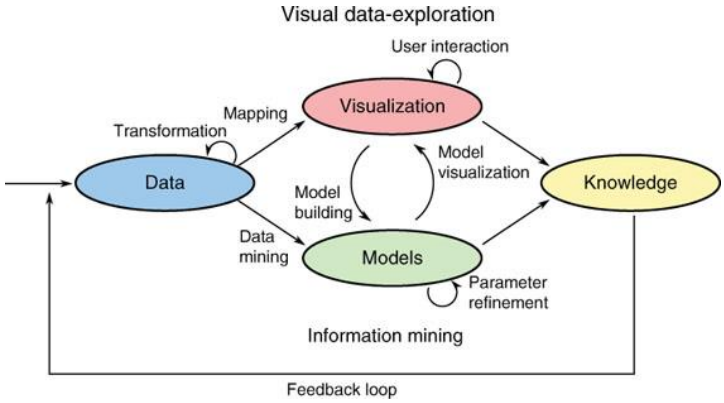

Figure 44: Visual data exploration process.

1. In real life visual analysis scenarios, it is common to make use of data that might not derive from the same source, thus the first step is making an exploratory analysis and preprocessing data, by transforming the data with the purpose of extracting significant information for the next steps. Commonly, the preprocessing tasks are considered to be; cleaning, normalization and grouping, among others.

As it has been pointed out through the description of the current project, the source of data collection has not been unique. In this case, data has been collected from different machines and systems.

2. Once most relevant data is ready, the user or the analyst, can take the following step by visualizing or data mining. It is probable that a single visualization is not enough to obtain the expected kind of conclusions. That is why the interaction between the user and the visualization is essential. The user should have the possibility to zoom on what is displayed or to consider different visual sights. By visually representing the data, the knowledge extracted from those graphs can take part in the creation of machine learning models. Even though, those models can also be created as the first step after data transformation.

3. On the model creation, the user has the possibility to interact with the automatic methods adjusting or modifying previously used parameters, or changing the algorithms being used. Visualization plays an important role when showing the results that are being acquired.

4. The process of alternating between visual representations and machine learning models is the main characteristic of *Visual Analytics*, and it leads to a persistent improvement, perfection and verification of the results.

Following this process, the dashboard generated has as main idea to show the results obtained from the analysis of the data traffic of a server, to extract information of these characteristics it has been decided to visualize the following ideas:

- The correlation between the different types of data captured and treated by a model.
- The values and scores resulting from the generation of the model.
- The grouping of the properties of each cluster obtained from the model.

Each of these ideas is represented by a graph or section of its own and in the case of the graph will be interactive, giving the possibility to visualize the information of a particular cluster.

The dashboard will also allow the parameterization of the algorithm used to train the model and generate the results.

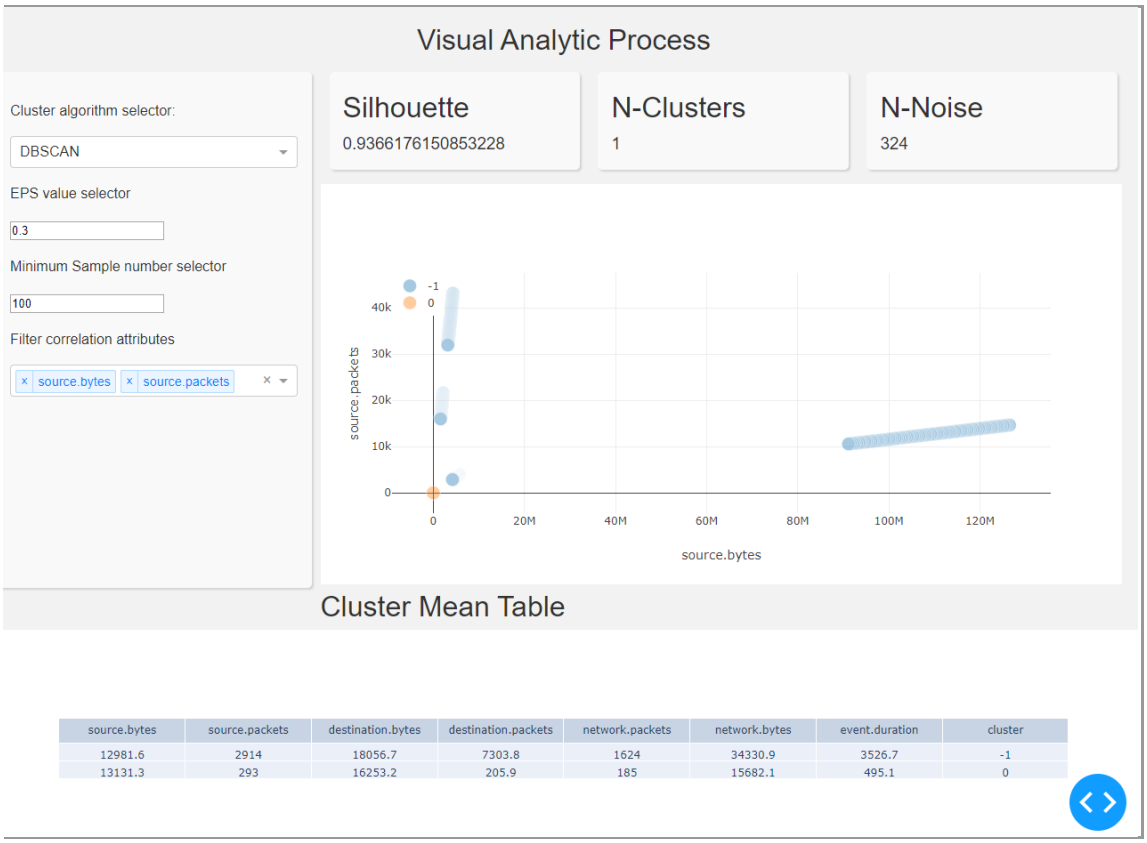

Figure 45: Plotly dash dashboard.

User interaction

In order to enable the parameterization of key variables in the dashboard, a section has been created that allows the user to make a series of inputs to change these variables.

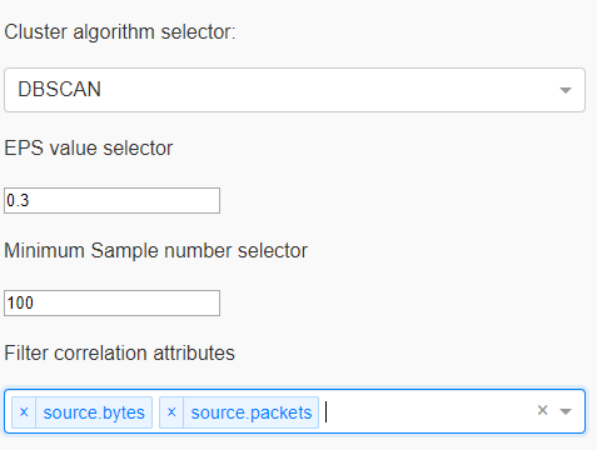

Figure 46: User interaction options.

The first variable that can be altered is the clustering algorithm used to create the model. To be able to edit this value it has been decided to use a dropdown menu since the options are few and specific, in this

way the user will find in a single glance all the possibilities and will only have to click in the desired one, avoiding at the same time possible typographical errors.

The next two customizable variables are the two parameters with which the clustering algorithm selected in the previous phase is initialized, the EPS values and the minimum number of samples. These values are numerical. It has been decided to use a text section as input since the values can vary in great amount and therefore it loses the sense to establish a button to modify in x fixed amount the value.

Finally, we have a selector that offers each of the data analysed by the algorithm to select two. As the objective is to select more than one, the dropdown admits more than one selection. We have opted for the multiple dropdown since the options are few and the user can see all at once. By selecting one, that option will disappear to prevent the user from selecting the same variable twice by mistake.

The chosen value can also be deleted just clicking in the cross of the selection field, putting it back as a selection option automatically.

Data charts

The following data chart has been created in order to show three different values, these values are informative of the results obtained with the configuration inserted in the parameterization section.

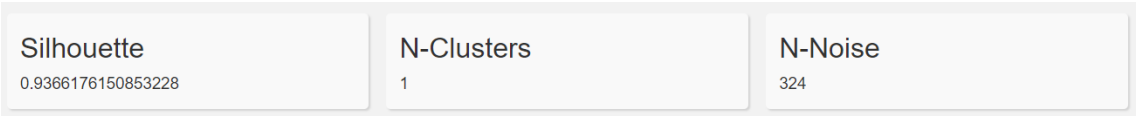

Figure 47: Data chart section.

It consists, as can be seen, of some rectangles that contain the name of the variable and its value below. It is a simple, clear and direct way to specify the values without giving rise to confusion.

The values shown are numerical and are related to the results of the Silhouette score, the number of noise and the number of clusters obtained by the trained model with the given parameters.

Correlation chart

To represent the correlation between the variables selected above, two values must be specified in the multiple dropdown. The graph will show the correlation in a scatterplot by means of colored circles, the colors will be represented in a legend next to the cluster to which they belong.

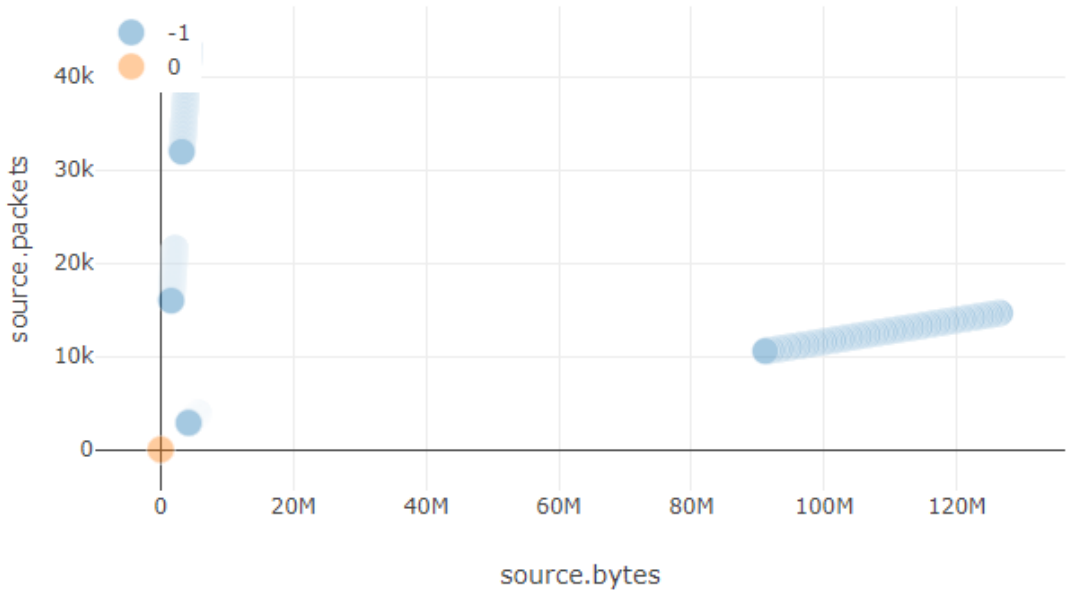

Figure 48: Correlation chart.

The user will have the possibility to see the precise values by hovering on the desired point with the mouse.

The user can also click on the legend to hide the points belonging to a specific cluster and alter the display in real time as a filtering measure.

The graph also allows the user to make a cut of the data so that the graph focuses its display on the cut made, allowing greater ease when working in spaces with large separations between data of interest.

Mean table

This table has been created in order to display the averages of the values belonging to each cluster. In this way and specially when the number of cluster is low, the user will be able to have a statistical idea of the average of types of values in particular.

source.bytes	source.packets	destination.bytes	destination.packets	network.packets	network.bytes	event.duration	cluster
12981.6	2914	18056.7	7303.8	1624	34330.9	3526.7	-1
13131.3	293	16253.2	205.9	185	15682.1	495.1	0

Figure 49: Mean table.

The different columns form the table can be moved with the mouse, in order to alter their order in case of a more personalized display is wanted to be generated.

1.8. Secret management

The applications presented make use of different types of security measures in order to maintain the privacy, availability and integrity of the data at all times. These methods guarantee the authentication and encryption of the information used during the communications. However, to obtain this, users, passwords and certificates must be accessible for automatic deployment. This means that certificates and keys must be shared on the Internet to allow automatic deployment. This compartment generates an attack vector when the keys are exposed; this is where the control of secrets is applied in order to alleviate the problem.

These values to manage can be database credentials, passwords, third party API keys and even arbitrary text, any data that could put at risk the security of our infrastructure will be called secret.

Secrets must be kept hidden, in this case the development team has made use of a public repository in order to develop in group all the necessary code for the different parts of the project as well as to simplify the deployment of the necessary data automatically through CloudFormation. Since CloudFormation is making the installation from scratch, it must also obtain the necessary certificates and credentials to apply the necessary security to each application. Therefore, the certificates must also be in the repository. This, together with the fact that the repository is public, puts the secrets of our project in the hands of any attacker.

To solve this problem, and to be able to keep secrets accessible but hidden, there are a number of software solutions classified as secret managers. The team has evaluated the different solutions that best fit the problem to be solved, finding two possible solutions, AWS Secret Manager and Vault, which will be further evaluated to determine which one to implement in the project.

1.8.1. AWS Secret Manager

AWS Secret Manager is one of the leading candidates, considering that it is a tool offered by Amazon, the same company we are using for the provisioning of our entire infrastructure. This is why it is specially integrated with their other services and may be the most logical choice at first.

However, despite its integration with the rest of their services and the simplicity of its deployment, the type of AWS account being used did not allow its unrestricted use so it was decided to find another alternative.

1.8.2. Vault

VAULT is an application that allows the creation of different secret storage engines; this gives the ability to keep different types of secrets in environments accommodated for future use and easier implementation.

We chose it because of its great adaptability and potential; it is a well-known option used in the professional sector and therefore provides a great value of knowledge. One of the problems we face is to learn from this tool, as it can be hard at first because it is so large and complete.

Once we decided that the solution to use would be VAULT we began the investigation of how to use it to meet our needs, in the process we discovered certain keys of the routine functionalities of the application, functionalities that we will expose next.

As mentioned above, the VAULT uses a system called engine to provide an environment and a solution suitable for the type of secret you want to store.

VAULT consists of a server that is responsible for not only keeping the secrets safe but also allows other computers to request these secrets safely through https. It also offers a web interface that will allow us

to make the first configuration in a more visual way and will allow us in the future to add and remove secrets in a more visual and graphic way.

Initially when the VAULT server is started it will be in sealed state, in this state will not respond to any request asking for secrets or let us add any new ones. To be able to add secrets and more importantly ask for the ones we already have stored, we must unseal it. To do this we will have to use a key generated automatically during the installation of the VAULT. This key along with others must be stored safely, as they will be the password to use and access the VAULT, which is ultimately a place that centralizes our secrets.

Another key that auto generates is the so called root token, this token must be used to verify every secret request that is made, the process of adding a new one and also it will be necessary to log in the web interface that offers VAULT. So this token must be known to every machine how wants to get secrets from VAULT.

To carry out the installation of the VAULT it was decided to dock the service in one of the machines of our infrastructure to which the rest of the machines had access that required the use of some secret. In this way we can lift the service when we are interested and keep it next to their premises in a container of our own.

For its installation we must have an installation of Docker and Docker-compose on the computer that will act as a VAULT server. In this case, the VAULT installation has been uploaded to a git repository so that the machine that will act as the server can download everything necessary to start the service when it is created using the User Data programmed in the CloudFormation script. The secrets are stored in the volume of the docker in question so they will always be present with each installation when downloaded from the repository. It should be noted that the secrets are stored in unreadable text and unusable by an attacker so it is safe to upload them to a repository even if it is public, provided that the keys mentioned above are not also uploaded.

In order for each machine that needs a secret to get what it needs it must have a connection to the machine that is hosting the VAULT server as well as a vault client installation that will allow it to make secret requests to the server, in order to make the request the machine must know the root token. This vault cli service will therefore be installed on every deployment on every machine that needs to interact with the VAULT; the installation will be done by using the user data programmed in the CloudFormation script so that it will be automatically installed.

By the next steps we will download and set the installation of the VAULT client in the desired machine:

sudo mkdir -p /opt/vault/directory

sudo wget https://releases.hashicorp.com/vault/1.2.3/vault_1.2.3_linux_amd64.zip

unzip vault_1.2.3_linux_amd64.zip

sudo chown root:root vault

sudo mv vault /usr/local/bin/

After doing this commands we will be able to use vault calls, this calls include the unseal process of the VAULT server, this can be done with the following command:

vault operator unseal -tls-skip-verify generatedKey

As can been seen the used command uses the “vault” word at the start, this is the way to call every function given by the previously installed VAULT client.

In our case, we want to keep three types of secrets: users, passwords and certificates. These three types of secrets are necessary for the proper functioning of the different services that are deployed in the project. To store them there are different ways, as mentioned above there are specialized engines for the storage of different types of secrets, for certificates, for credentials of a specific application or more generic as value keys.

In this case, we decided to use an engine with the ability to store values based on a key. This way by using a key such as rabbitmq_username vault would give us the username of the rabbitmq. For certificates, we also use the key value, being the key the name of the certificate in question and being the value the text that is stored inside the certificate. Obviously, the stored certificates were generated and tested before being stored in the VAULT.

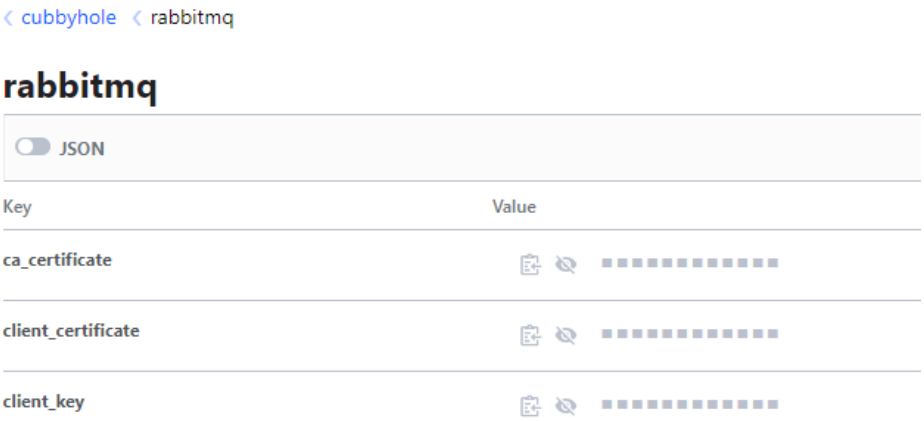

Figure 50: Secrets stored in the Vault.

In order to obtain a key value from the previously installed vault client we will have to specify a path, this path consists of the internal directory of vault where the key is located. The path specifies in which engine the secret is stored and in which set of secrets it is found. A set of secrets in a list with several sercrets that respond to a general name, something like a directory that serves to differentiate and better organize our secrets.

For example, the following one would be a valid route:

vault read -field username -tls-skip-verify cubbyhole/rabbitmq

Where we can see that the secret we are looking for is the username, a key that is found in the key engine value called cubbyhole and that responds to rabbitmq's secret directory.

To automate the obtaining of the necessary secrets a script has been created that automates the obtaining of the values and the creation of the certificates by writing the values stored in the certificate files. This script is executed when the objective computer is deployed and will ask for the necessary values for the correct functioning of the services it has.

The execution of the script is done in the user data and the values obtained are stored in local variables of the user data session, so when the user data process is finished it will be closed and the local variables of the session will disappear.

In our case we decided to store the necessary certificates for the correct and safe operation of rabbitmq, as well as the user and password for the administration, in VAULT. This same process can and should be replicated for all possible secrets in order to achieve a more secure infrastructure.

4. Conclusions

We have obtained several conclusions about the work done, below we will present those ideas and conclusions that we have obtained as a group for each section of the project.

In general, we have managed to secure the secrets as intended despite having to do extra work because the most suitable tool, AWS secret manager, was not available in our account version. With this work, we have been able to verify that VAULT is a powerful tool that integrates well with the cloud architecture and in other projects, we can reuse the tool.

We have managed to correctly deploy the application in the cloud in such a way that we have been allowed to collect data from the traffic generated, we have implemented a new saga that interacts and creates conflict with the saga created during the semester, and we have managed to solve the conflict so that the two sagas work in the application as expected. We would like to have been able to add more functionalities to the application and have used better the logs for the collection of intelligence.

We have managed to train a model in an unsupervised way with the data obtained, different metrics have been used to validate the models and we have been able to interpret the data to reach a final conclusion. We believe that the results obtained could be significantly improved if the quality of the data were better. The quality of the data and not the quantity has been the main limiting factor.

Along with the data analysis part, this is the section in which we have gone the furthest in terms of objectives, we have managed to deploy and implement the infrastructure and all the services in an automated manner. We have prepared a working environment in the cloud that has allowed us to deploy everything quickly and that would allow us to replicate the entire infrastructure and launch the entire project with a single file. With more time, we would have tried to implement high availability and to port part of our application to aws services, such as databases.

In conclusion, we believe that we have achieved the most important objectives of the project by obtaining a functional product that would allow it to be continued with relative ease.

5. Future lines

The project has room for improvement, so that many of the points in the rubric can be improved. Below are several improvement points described.

One of the key points of the project and therefore, more important, is to improve the quality of data and the process of data collection through other technologies, such as Osquery, or similar.

In the security aspect, due to lack of time, only some secrets were stored safely in the VAULT secret manager, so the next step should be to store absolutely all the secrets in VAULT.

In the field of architecture deployment there are also several possible improvements, such as the use of scalable databases RDS offered by AWS. Another point would be the creation and adaptation of the infrastructure to the auto scaling groups offered by AWS.

An interesting objective would be to achieve the use and total integration of a honeypot of our own, as proposed in the rubric.

Finally make the programmed sagas of our microservices to be able to recover the state in which they were if the microservices are turned off or falls for an error. So when the saga goes up again it will continue from the previous stage.

6. Annexes

Deployment Scripts (**https://gitlab.danz.eus/ivan.valdesi/popbl_deployment_scripts**)

- Parametrized deployment script
- Cloudformation Script
- AWS Logs
- Simule normal use of the app script

Deployment Scripts (**https://gitlab.danz.eus/ivan.valdesi/popbl_deployment_scripts**)
- Parametrized deployment script
- Cloudformation Script
- AWS Logs
- Simule normal use of the app script

Vault files (**https://gitlab.danz.eus/alexander.tesouro/popbl_vault**)

- Vault helper script
- Vault docker-compose

Microservices Application (**https://gitlab.danz.eus/ivan.valdesi/servicesapp_popbl**)

Data analysis (**https://github.com/xetxezarreta/beats-anomaly-detection**)

- Data analysis notebook

Honeypots (**https://github.com/xetxezarreta/docker-elk/tree/elastic/sb-honey**)
- Honeypots docker-compose
- Packetbeat config file
- Metricbeat config file

Elastic Stack (**https://github.com/xetxezarreta/docker-elk/tree/elastic/sb-ml**)

- Elastic Stack docker-compose
- Elasticsearch config file
- Kibana config file

Microservices beats (**https://github.com/xetxezarreta/docker-elk/tree/elastic/sb-ml**)
- Packetbeat config file
- Metricbeat config file

Plotly dashboard (**https://gitlab.danz.eus/alexander.tesouro/popbl_dash**)

7. Bibliography

1 A. C. Logs. [En línea]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/WhatIsCloudWatchLogs.html.

2 «Haproxy,» [En línea]. Available: https://www.haproxy.org/.

3 «RabbitMQ,» [En línea]. Available: https://www.rabbitmq.com/.

4 «Consul,» [En línea]. Available: https://www.consul.io/.

5 «DBSCAN,» [En línea]. Available: http://www.ccs.neu.edu/home/vip/teach/DMcourse/2_cluster_EM_mixt/notes_slides/revisitofrevisitDBSCAN.pdf.

6 «Cowrie docs,» [En línea]. Available: https://readthedocs.org/projects/cowrie/downloads/pdf/latest/.

7 «Dionaea docs,» [En línea]. Available: https://readthedocs.org/projects/dionaea/downloads/pdf/stable/.