UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

José Amândio Ferreira dos Santos

MUTUALISTICALLY INTEGRATING SERVICE MESH WITH EXTERNAL CONFIDENTIAL APPLICATIONS

CAMPINA GRANDE - PB

2022

José Amândio Ferreira dos Santos

MUTUALISTICALLY INTEGRATING SERVICE MESH WITH EXTERNAL CONFIDENTIAL APPLICATIONS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientadora: Andrey Elísio Monteiro Brito.

CAMPINA GRANDE - PB

2022

José Amândio Ferreira dos Santos

MUTUALISTICALLY INTEGRATING SERVICE MESH WITH EXTERNAL CONFIDENTIAL APPLICATIONS

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Andrey Elísio Monteiro Brito

Orientador – UASC/CEEI/UFCG

Professora Reinaldo Gomes

Examinador – UASC/CEEI/UFCG

Professor Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 06 de abril de 2022.

CAMPINA GRANDE - PB

RESUMO

As malhas de serviço se tornaram populares, pois ajudam a monitorar e gerenciar aplicações baseadas em microserviços: armazenando e processando aplicações que frequentemente incluem conteúdo sensível em seus dados, e permitindo a adição transparente de funcionalidades através de um proxy, sem incluí-las no próprio código. Em paralelo, houve um crescimento na demanda por aplicações sensíveis que isolam dados sensíveis em um enclave de CPU protegido durante o processamento. O uso de aplicações confidenciais em malhas de serviço é uma união incompatível em seu estado atual. Um simples proxy acaba expondo dados que, até então, eram protegidos pela aplicação confidencial. Embora as malhas de serviços atuais não suportem bem, as malhas de serviços e as aplicações confidenciais podem ser combinadas de fato. Testamos vários proxy's que poderiam atender a esta demanda e avaliamos duas opções que podem ajudar a atingir este objetivo: um GHOSTUNNEL confidencial e o SCONE Network Shield.

.

Mutualistically integrating service mesh with external confidential applications

José Amândio Ferreira dos Santos[1]*, Andrey Elísio Monteiro Brito[2]

Abstract
Service Meshes became popular as they help monitor and manage microservice-based applications: storing and processing applications that often include sensitive content in their data, and enabling the transparent addition of functionalities through a proxy, without including them in the code itself. In parallel, there has been a growth in demand for sensitive applications that isolate sensitive data in a protected CPU enclave during processing. The use of confidential applications in service meshes is an incompatible union in its current state. A simple proxy ends up exposing data that, until then, were protected by the confidential application. Although current services meshes do not support it well, service meshes and confidential applications can be indeed combined. We tested several proxies that could meet this demand and evaluated two options that can help achieving this goal: a confidential *GHOSTUNNEL* and *SCONE's network shield*.

Keywords
ISTIO — GHOSTUNNEL PROXY — SPIRE — SCONE – ENVOY — CONFIDENTIAL APPLICATION

[1] *Universidade Federal de Campina Grande, Campina Grande, Paraíba*
[2] *Universidade Federal de Campina Grande, Campina Grande, Paraíba*
***Corresponding author**: jose.amandio.santos@ccc.ufcg.edu.br

Introduction

A Service Mesh is a tool that helps in sharing data between different components of an application. It enable the transparent addition of features, such as the ability to observe how the environment communication works and traffic and security management, without immediate changes to the source code [1]. In addition, it assists in monitoring and managing microservices-based applications. The emergence and popularization of the service mesh resulted in the removal of the logic that governs communication between services, which was previously present in the application itself, transferring it to an infrastructure layer that, as the complexity of communication increased, made its use increasingly necessary.

In parallel with the service meshes popularity, there was the growth in the usage of Intel SGX [2], which enabled operating system and user processes, or applications, to define remote regions of memory, called enclaves, in which code and data can be protected, thus making access difficult for potential intruders. These enclaves gave rise to a new application paradigm known as *confidential applications*.

The integration of sensitive applications into service meshes would enable their use with microservices along with a computing model that has become popular in recent years. However, using a proxy to stay ahead of applications, when they are in the service mesh, makes it difficult to use, since another non-confidential element begins to exist in the environment. When trying to introduce a confidential application into the mesh, during all requests the data, hitherto protected, would be exposed to potential intruders whenever it arrives at the proxy next to the application. As, by nature, the application handles sensitive data and the distribution of certificates made by such services issues weak identities, the environment becomes unsuitable for applications that need an environment (and threat model) that may include someone with access to the infrastructure.

Current service meshes are either a completely confidential environment or a completely non-confidential environment, in which the attestations of your applications are made by the mesh's own control plane.

In this state, applications that are already running and applications that make an attestation of their environment in another entity will not be able to use any of the current types of mesh, as they need a safe environment that does not require so many rules for its execution. in the mesh.

Thus, a possible solution to these problems starts with introducing a new identity provider to the service mesh that establishes a previous verification of the application to issue a certificate with a strong identity. It is still necessary to look for techniques that make it feasible to provide the issuance of identities for external applications. Finally, there is also the need to make sure the proxy runs within a trusted environment (i.e., an SGX enclave).

While building a trustworthy proxy. There are two basic approach. First, we can transfer the communication logic to communicate securely through headers and certificates using a proxy. We did it from the interception of the communication

made by other applications, adding the necessary information for the request to be carried out successfully and safely with other applications present on the mesh. We prove this secure communication through the issuance of certificates by the fabric controller.

Alternatively, the proxy can be integrated into the application itself with the help of a runtime environment. It receives an identity from the sender, established by the mesh, thus achieving communication in the same way as the applications present in the mesh. Such modifications should make the service fabric capable of securely communicating with external sensitive applications and maintaining the main benefits offered by both techniques.

With this proxy working properly, it is possible that applications already running, such as a confidential server, can communicate with a fabric after both receive a certificate. In addition, confidential applications that are attested by a specific attestation service can be attested and if you receive a certificate with the service mesh applications, you can communicate in a simple and practical way.

1. Background

To begin the explanation of confidential proxies it is necessary to explain some basic concepts and technologies:

- ISTIO, a popular service mesh;
- The SPIFFE standard and one implementation, SPIRE;
- The process of attesting a confidential application with SCONE.

1.1 ISTIO

ISTIO is an open-source service mesh that layers transparently onto existing distributed applications providing a uniform and more efficient way to secure, connect, and monitor services.

It is designed for extensibility and can handle diverse deployment needs. ISTIO's control plane runs on Kubernetes. We can add applications deployed in that cluster to the mesh, extend the mesh to other clusters, or even connect VMs or other endpoints running outside of Kubernetes.

A large ecosystem of contributors, partners, integrations, and distributors extends and leverages ISTIO for various scenarios. Several vendors have products that integrate ISTIO and manage it. [3]

ISTIO's architecture is divided into the data plane and the control plane. In the data plane, ISTIO support is added to a service by deploying a sidecar proxy within your environment. This sidecar proxy sits alongside a microservice and routes requests to and from other proxies. Together, these proxies form a mesh network that intercepts network communication between microservices. The control plane manages and configures proxies to route traffic. The control plane also configures components to enforce policies and collect telemetry. [4]

As one of the most popular service meshes, ISTIO is an appropriated choice for this process to effectively communicate a confidential application outside the mesh with the applications present in it.

1.2 SPIRE

SPIRE [5] works as a trusted certificate issuer for our applications, which will generate certificates with IDs following the SPIFFE standard [6]. A SPIFFE ID is a structured string (represented as a URI) that serves as the "name" of an entity that is defined in the SPIFFE Identity and Verifiable Identity Document (SVID) specification.

A SPIFFE Verifiable Identity Document (SVID) is a document that carries the SPIFFE ID itself. It is the functional equivalent of a passport – a presented document that carries the presenter's identity. Of course, similar to passports, they must be resistant to forgery, and it must be evident that the document belongs to the presenter. An SVID includes cryptographic properties that enable it to be (*i*) proven as authentic, and (*ii*) proven to belong to the presenter. In addition, it has an attestation process to provide identities for our application. [7]

Attestation in the context of SPIRE is asserting the identity of a workload. This procedure is done in two phases: first, node attestation (the node's identity, which the workload is running on, is verified) and then workload attestation (the workload on the node is verified). In both cases, this is achieved by comparing attributes gathered from the workload process itself and from the SPIRE Agent's node to a set of selectors defined when the workload was registered. These trusted third parties are platform-specific.

SPIRE has a flexible architecture that enable it to use multiple trusted third parties for node and workload attestations, depending on the workload environment. Thus, the attestation needs to know which trusted third parties and which types of information to use. It obtains the first from entries in agent and server configuration files and the second from selector values specified when workloads were registered.

This form of attestation enable the creation of third-party plugins that check the integrity of nodes and workloads from information obtained with the TPM and SGX drivers, thus verifying confidential applications and environments. [8]

1.3 SCONE

SCONE is a Secure CONtainer Environment for Docker that uses SGX to run Linux applications in secure containers. [9]

SCONE (Secure CONtainer Environment) supports the execution of confidential applications inside containers running inside a Kubernetes cluster. SCONE also supports the execution of confidential applications inside VMs with emphasis on containers. SCONE supports all common programming languages. [10]

Our intention is to use SCONE applications so that it is possible to perform their communication with applications in the mesh.

1.3.1 Attestation

The attestation of SCONE applications requires the use of a CAS (Configuration and Attestation Service), which can be public or executed locally, and a LAS (Local Attestation Service) previously deployed on each machine. With these services active, we must create a session for our application to store policies that describe how the service is executed amd how it gets and share access to secrets and data.

1.3.2 Network Shield

The network shield works as a proxy coupled to the SCONE environment, helping to encrypt non-TLS requests without changing the code, transforming a previously HTTP request into an HTTPS request.

The network shield, upon establishing a new connection checking, performs a TLS handshake and encrypts/decrypts any data transmitted through the socket. This approach does not require client- or service-side changes. The private key and certificate are read from the container's file system. Thus, they are protected by the file system shield. [9]

Analyzing the use of the network shield, we can see its usefulness as a potential confidential sidecar for our communication. Since it is present in the same environment as the client applications.

2. Methods

This section describes how we used the previously presented tools to create a enabler for security application development, by providing a way to apply as security applications, including a trusted way to apply to an issuer of security identities, including a trusted application to the service fabric. The starting point was to modify a service mesh by adding an identity issuer that enable issuing and attesting identities to external applications.

We used ISTIO as our service mesh provider and SPIRE as the identity issuer. Subsequent to that, the creation of sidecar proxies will be started that help applications – running in a SCONE environment – to communicate with the mesh.

2.1 ISTIO with SPIRE

In ISTIO, the proxy sidecars receive their identities through a UNIX Domain Socket (UDS) that they share with an ISTIO agent running in the same container. To make ISTIO use SPIRE to provide identity it is necessary to make some modifications. When replacing the ISTIO identity-issuing mechanism with that of SPIRE, the sidecars can communicate with the UDS of the SPIRE node agent instead of the ISTIO agent UDS,this was done changing the proxy configuration. Once we achieved the objective, all left was to change the SPIRE code to support the proxy sidecar configuration generated by ISTIO.

For the sidecar testing process, it was necessary to upload an ISTIO that had the certificate rotation done by SPIRE. Because of this ISTIO was automatically generated, it was impossible to share certificates for non-mesh applications. For this modification, we need to modify the ISTIO configuration file, mapping the previous certificate rotation socket, the *Citadel* (default of ISTIO), to be SPIRE.

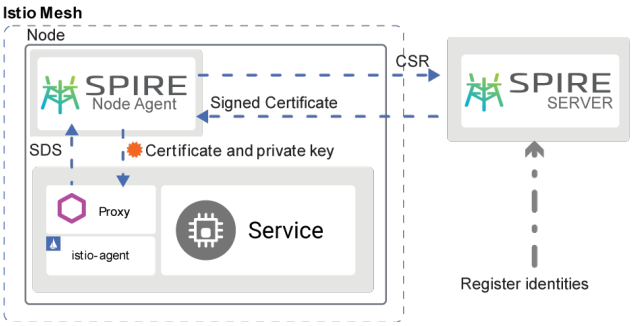

Figure 1. ISTIO SPIRE workflow [11]

The subsequent step is developing a helper application that facilitates communication between the confidential application and the mesh. The way in which the communication process works and how ISTIO and SPIRE work underneath is illustrated in the Figure 1.

2.2 Testing ISTIO-SPIRE execution

With the development of ISTIO-SPIRE (ISTIO with SPIRE as the identity issuer), the use of node's identity to perform requests using test requests is viable by running a simple server using httpbin, inside the mesh. It will receive the support of the ISTIO proxy with the certificates provided by SPIRE. Then, with the server up, it is only necessary to register its identity in SPIRE to enable certificate rotation for the application's sidecar.

It is still necessary to run an application that serves as a client and sends requests to our server to perform this communication. This client application is deployed outside the mesh.

When both applications are up, we send a request from the client to the server, which will return a STATUS 200.

After this test, to guarantee that applications can only communicate with TLS, the STRICT mode was activated as follows:

```
kubectl apply -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
    name: "default"
    namespace: "istio-system"
spec:
    mtls:
      mode: STRICT
EOF
```

In order to maintain the security of the mesh environment, since we want SCONE applications to share their data to it, we enabled ISTIO's STRICT mode so that external applications can only communicate with the presence of a certificate. With these certificates, only applications that we trust can

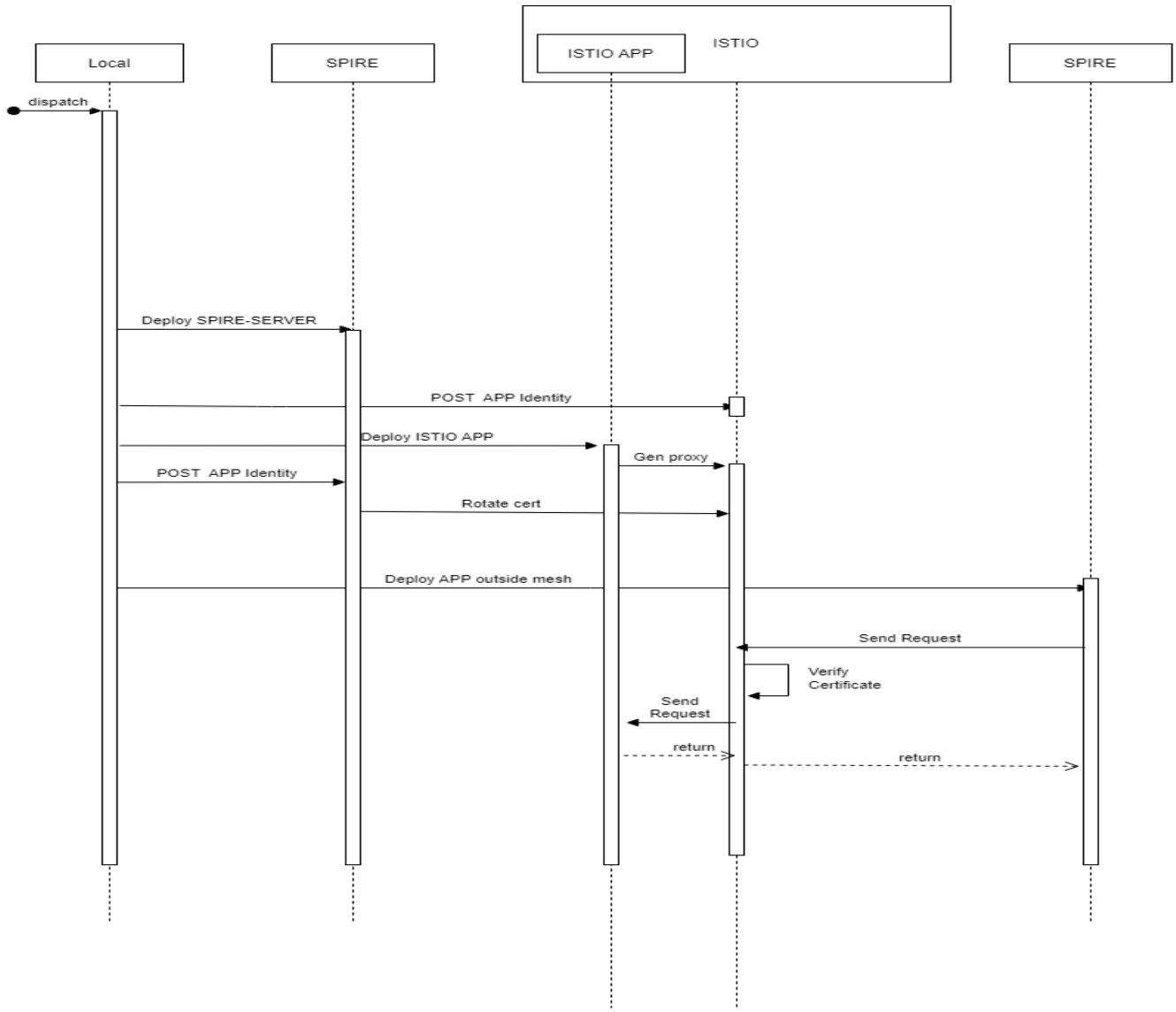

Figure 2. Sequencial simple ISTIO-SPIRE communication

have access to data received or created in the mesh environment. However, with SPIRE in this mesh, the application only needs to be certified. It will receive a certificate that will communicate with everyone inside the ISTIO since everyone in the mesh trusts who receives credentials from the SPIRE-SERVER provider.

With straightforward communication explained, we can move on to adding SCONE applications and ways to make it easier for these applications to communicate with the mesh automatically and without changing the initial code.

Initially, the viable sidecars were mapped and then reduced to just three, because they can communicate with SPIFFE ID:

- GHOSTUNNEL PROXY
- ENVOY PROXY
- Network shield

Thus, the subsequent steps are to make the sidecar confidential (if necessary), performing communication from a sensitive application to an application present in a popular mesh. With one of these sidecars performing the execution that helps communication with the mesh, we will already have the expected result.The process can be observed by observing the Figure 2.

2.3 Confidential GHOSTUNNEL

GHOSTUNNEL is a simple TLS proxy with mutual authentication support for securing non-TLS backend applications. GHOSTUNNEL supports two modes, client mode and server mode. When in the server mode, it runs in front of a back-end server, accepts TLS-secured connections, and then proxies to the (insecure) backend. A backend can be a TCP domain/port or a UNIX domain socket. And when in the client mode, it accepts (insecure) connections through a TCP or UNIX domain

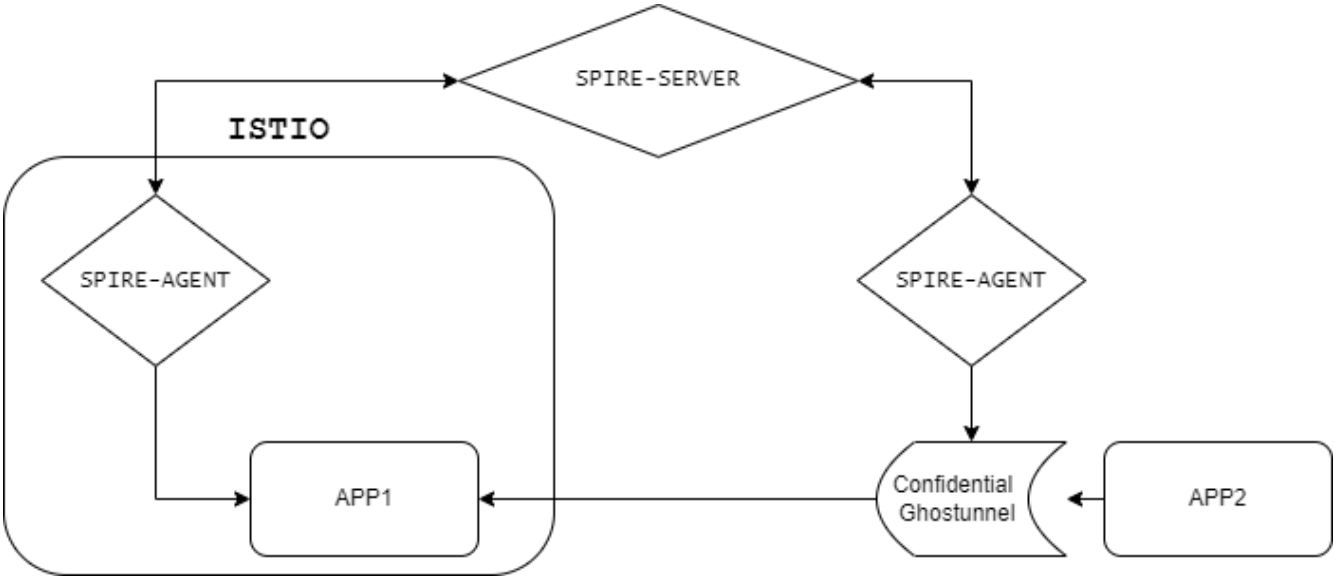

Figure 3. Illustrate communication with confidential GHOSTUNNEL

socket and proxies them to a TLS-secured service.

The confidential GHOSTUNNEL build process is performed using a SCONE image that presents the build configuration tools in the GO language. Then was used the source code in the latest version of SCONE image,which runs in Go 1.15. Finally, we use the application build command to make it confidential and add the necessary environment variables for the execution and The entire process is described at a higher level in Figure 3. For this execution, it was necessary to follow these steps:

- Using a Dockerfile, we perform the build of GHOSTUNNEL in an alpine image to make it confidential and solve errors that may occur in this version and thus minimize errors in a build on the SCONE image.
- As a next step, was started a build on the SCONE image. Finally, was added information about the environment variables needed to run SCONE.
- After the build and with the image present locally, was inserted a session into an existent CAS and had a LAS present on the machine so that it is possible perform the execution of the confidential GHOSTUNNEL, for, in the end, use GHOSTUNNEL command to communicate with the application present in the mesh:

```
/usr/bin/ghostunnel client --unsafe-
    listen --use-workload-api-addr {
    SOCKET-ADDR} --listen={GHOSTUNNEL-
    ADDR}--target={ISTIO-APP} --verify-
    uri {SPIFFE-VERIFY}
```

With this, the workflow is illustrated in more detail using a sequence diagram referenced to Figure 4.

Even reaching our goal of communicating with the mesh, confidential GHOSTUNNEL still has limitations. It is unable to read certificates that do not come from SPIRE – which is possible in the original GHOSTUNNEL – and it cannot securely get certificates to use in the application.

2.4 Confidential ENVOY

ENVOY is a self-contained process signed to run alongside every application server. All of the ENVOYs form a transparent communication mesh in which each application sends and receives messages to and from localhost and is unaware of the network topology. [12]

For these sidecars, it was necessary to transform the application into a confidential application. Because the ENVOY proxy builds are based on BAZEL, a free software tool for the automation of building and testing of software, it adds a greater complexity than CMAKE and makes it challenging to use sconify. The general workflow to generate a confidential image is described below.

An extra stage that sconifies, i.e., converts a native image into a confidential container image to an existing CI/CD pipeline for a node-based application/service. The transformation is controlled via the command line arguments of the sconify image. Typically, the appropriate arguments would be selected when configuring the CI/CD pipeline. The image sconification should, in most cases, be wholly automated and executed as part of the CI/CD pipeline. One might run the test stage after the sconification of the image. We develop a script that uses an image provided by ENVOY and uses it in the sconify process. The script uses the latest version of SCONE image and ubuntu 18.04 as the base image of the application. See below:

```
#!/bin/bash

set -x
```

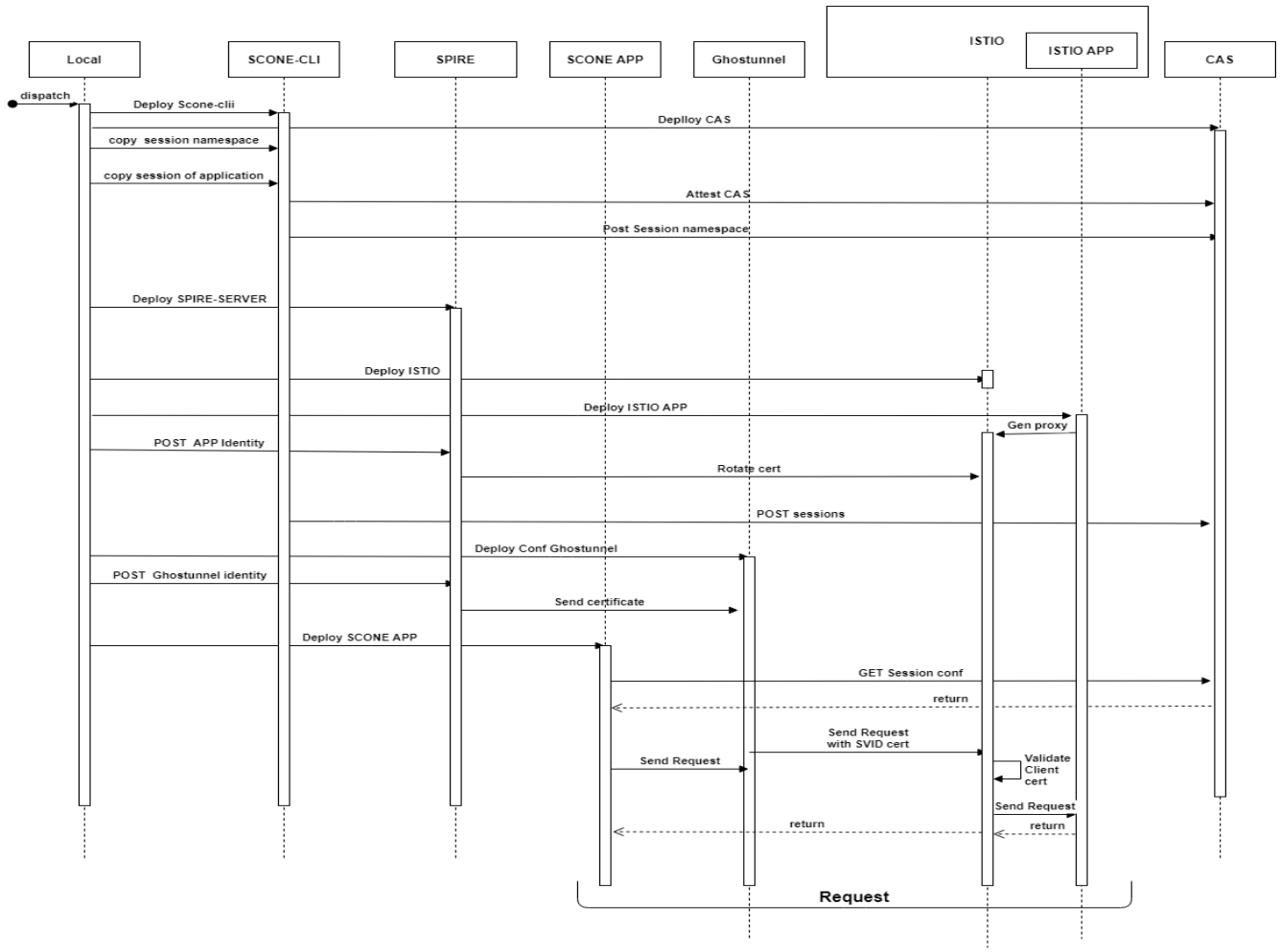

Figure 4. Sequencial GHOSTUNNEL PROXY workflow

```
echo "Sconify: envoy"

# Base Image
BASE_IMAGE="ubuntu:18.04"

# Image of target application.
SOURCE_IMAGE="envoyproxy/envoy-debug:v1
    .21-latest"

# Binary path
BINARY="/usr/local/bin/envoy"

# Docker image name
TARGET_IMAGE=${TARGET_IMAGE:="envoy:
    sconify"}

# Sconify native image.
SCONIFY_IMAGE=${SCONIFY_IMAGE_GLIBC:="
    registry.scontain.com:5050/clenimar/
    sconify-image-dev:5.6.0-18012022-5
    eaf8d6d"}
SCONE_CAS_ADDR=${SCONE_CAS_ADDR:="5-6-0.
    scone-cas.cf"}
CAS_NAMESPACE=${CAS_NAMESPACE:="envoy-
    $RANDOM-$RANDOM-$RANDOM"}

SESSION_NAME="envoy-session"
SERVICE_NAME="service"

ENV_VAR="envoy --version"

# Docker run performing image
    sconification with all variables
    created previously
# ...
```

- In the sconification process, firstly, indicate the base image of the code and the image with the application present, along with the path to the application's binary.
- As subsequent steps, the name of the image that the script should create with the sconified application and the image in which the sconification process will occur.
- The conclusion adds variables to fill the session to be created for the application, which occurs automatically

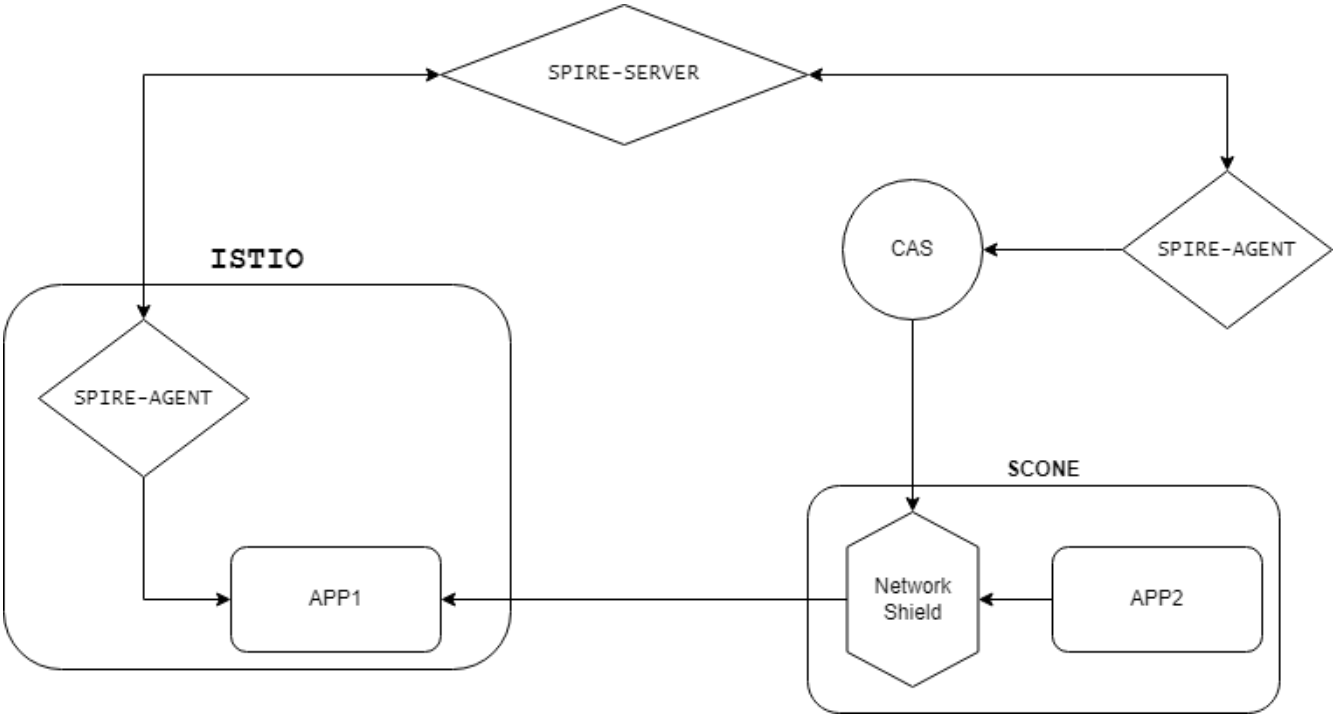

Figure 5. Illustrate communication with network shield

in sconify.

- With all the variables ready, just execute a ”Docker run” and analyze the automatic build process done by sconify.

When running the script, we create an image with the confidential application of ENVOY as a result. However, when running, the application loops, and no commands are executed, preventing ENVOY as a confidential sidecar.

2.5 Network Shield

Some container services, such as Apache and NGINX, always encrypt network traffic; others, such as Redis and Memcached, assume that the traffic is protected by orthogonal means, such as TLS proxies, which terminate the encrypted connection and forward the traffic to the service in plaintext. Such a setup is appropriate only for data centers. The proxy and service communication is assumed to be trusted, which is incompatible with our threat model: an attacker could control the unprotected channel between the proxy and the service and modify the data. Therefore, a TLS network connection must be terminated inside the enclave for secure containers. SCONE permits clients to establish secure tunnels to container services using TLS. It wraps all socket operations and redirects them to a network shield. The network shield performs a TLS handshake and encrypts/decrypts any data transmitted through the socket. This approach does not require the client or service-side changes. The private key and certificate are read from the container’s file system. Thus, they are protected by the file system shield. [9]

Using Network Shield, we intend to use it as a proxy sidecar and execute the entire process that is described at a higher level in the figure 5.

One of the positive points of the network shield is that it is a feature present in SCONE, so we do not create the need to make it a confidential application. Based on this, it is necessary to configure the network shield with the SPIRE server to receive updated certificates.

Unlike the other applications presented, this functionality does not support sockets, so to achieve certificate rotation, it is necessary to use the SVIDStore plugin present in SPIRE.

The plugin posts sessions with the constraints specified in the selectors to ensure that only attested workloads will get access to the right SVIDs. The plugin depends on the CA Trusted Anchor certificate for this CAS instance plugin to work normally. This certificate is extracted from the file /cas/config.json created by the SCONE CLI after executing the SCONE cas attest command. The trust anchor is the first certificate of the certificate chain written in the file. The plugin uses session templates to give operators more flexibility. Each template has placeholders used by the plugin to inject information and constraints for access control in the secret store.

To use the network shield before the SCONE application, it is needed to post a session in CAS. Based on this premise, was prepared the following session for its activation:

```
name: ${MY_NAMESPACE}/netshield-session
version: "0.3"

services:
```

```
  - name: secret-service
    mrenclaves: [{ENCLAVE_HASH}]
    command: python3 /app/secret_service
        .py
    pwd: /app
    environment:
      SCONE_MODE: hw
      SCONE_LOG: 7
      # network shield variables

      SCONE_NETWORK_SHIELD: unprotected

      SCONE_NETWORK_SHIELD_CLIENT_1: "
          protected"
      SCONE_NETWORK_SHIELD_CLIENT_1_
          DESTINATION: "TCP:DNS:PORT"
      SCONE_NETWORK_SHIELD_CLIENT_1_
          DESTINATION_IP: "*"
      SCONE_NETWORK_SHIELD_CLIENT_1_
          SERVER_AUTH: disabled
      SCONE_NETWORK_SHIELD_CLIENT_1_
          IDENTITY: |
        $$SCONE::svid:privatekey:pkcs8:
            pem$$
        $$SCONE::svid:crt$$

secrets:
  - name: svid
    import:
      session: ${MY_NAMESPACE}/netshield
          -session-injector
      secret: svid

security:
  attestation:
    tolerate: [debug-mode,
        hyperthreading, outdated-tcb,
        insecure-igpu]
    ignore_advisories: "*"
```

This YAML will focus on the variables with the prefix "SCONE NETWORK SHIELD". The first variable present is "SCONE NETWORK SHIELD," that have the options "protected" and "unprotected". To perform it, we used the unprotected option, because of using the opposite option the network shield would inspect all requests leaving the application, including DNS resolutions (Domain Name System) made using UDP (User Datagram Protocol), which is not supported by the network shield, as it will cause an error when detecting the UDP request made by DNS. Based on this, we can analyze and explain the YAML variables used in the process of using the network shield:

SCONE NETWORK SHIELD CLIENT 1 determines the mode of operation for an outbound connection (determined by 'SCONE NETWORK SHIELD CLIENT name DESTINATION' and 'SCONE NETWORK SHIELD CLIENT name DESTINATION IP'). using the variable as 'unprotected', the network traffic can pass through unfiltered.

SCONE NETWORK SHIELD CLIENT 1 DESTINATION is presented in the format of 'protocol DNS name port', which is the application that will receive the request from our application.

SCONE NETWORK SHIELD CLIENT 1 DESTINATION IP is displayed in IPv4 or IPv6 address format. How was it already determined the address by its DNS name, it is not necessary to specify its address in this variable and put a wild card.

SCONE NETWORK SHIELD CLIENT 1 SERVER AUTH is Only applicable if the mode is 'protected'. We used the 'unprotected' mode to perform this request. where was it placed the value 'disable' in this variable.

SCONE NETWORK SHIELD CLIENT 1 IDENTITY presents X.509 client identity (concatenated PEM-encoded PKCS8 private key, X.509 end-entity certificate, and optionally chain CA certificates). Only applicable if the mode is 'protected'.

After creating the session and posting it to CAS, starting the confidential SCONE application is viable. With this, the application requests an application present in the mesh, and, with the use of the network shield, it can perform a TLS communication without knowing the certificate. With this, the workflow can be seen in more detail using a sequence diagram referenced to Figure 6.

3. Results and Discussion

At the end, it was viable to communicate confidential requests with a service mesh. It was necessary to create a service mesh that could securely share and rotate certificates to applications outside the mesh to achieve this goal. Thus ISTIO SPIRE was born, which mixed the most popular service mesh with an application that can provide the certificates of the desired shape.

For creating the confidential ISTIO SPIRE, we chose proxy applications that could use SPIRE to generate the TLS certificates, and thus we chose ENVOY, GHOSTUNNEL, and Network Shield. Also, to stay ahead of SCONE applications, it was needed confidential proxies, so it was necessary to turn GHOSTUNNEL and ENVOY into confidential applications. During this process, ENVOY presented several complications that made its conversion a SGX-version not feasible. Meanwhile, we managed to achieve this goal with GHOSTUNNEL and, consequently, the first proxy that can help our application was built.

To conclude, the execution of the network shield, an application coupled to SCONE, as a proxy for this communication presented the expected results. We obtained two proxies that work differently and performed the desired communication.

3.1 Comparison

When comparing the sidecars that were successful in their execution, the network shield and the confidential GHOS-

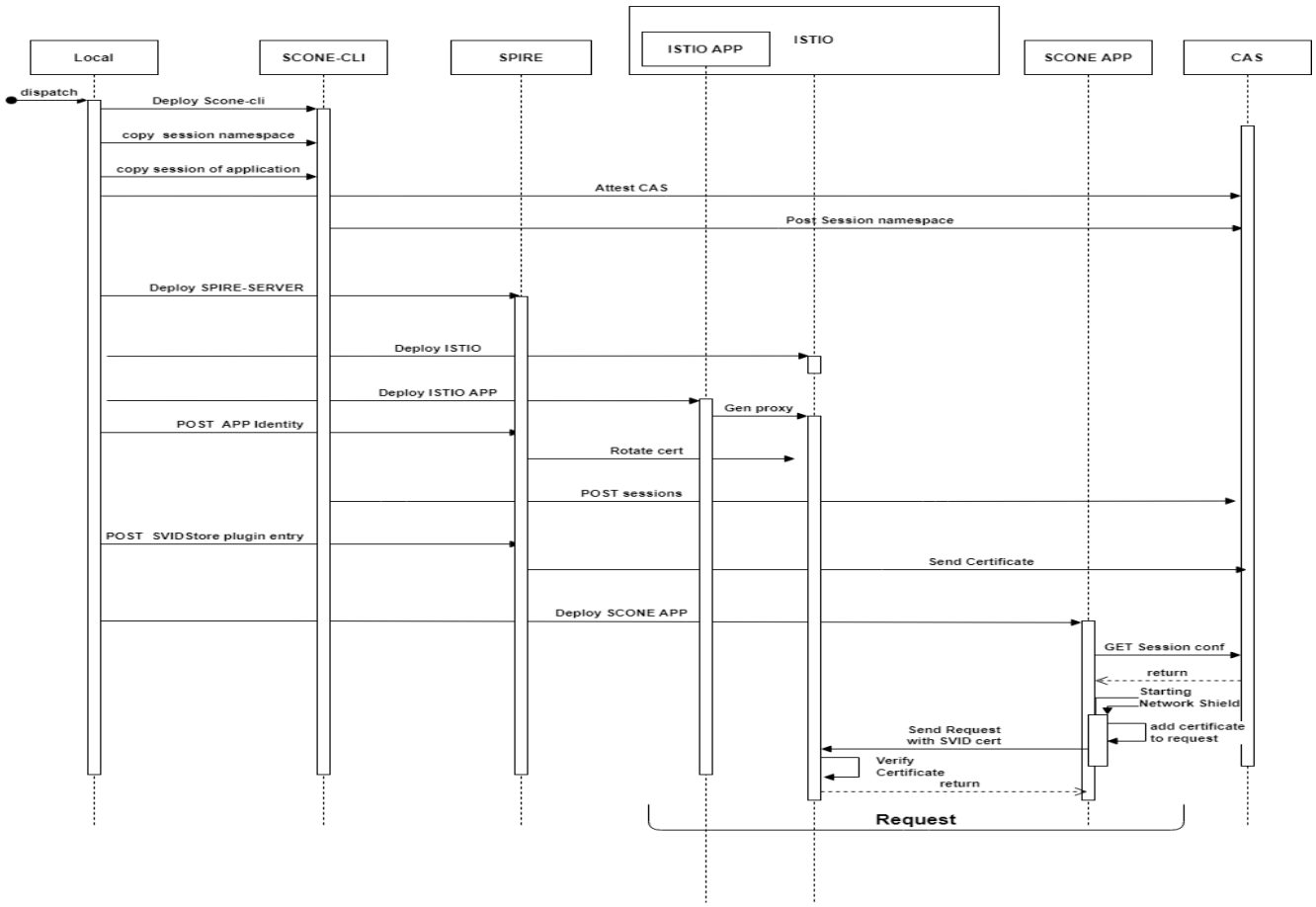

Figure 6. Sequencial network shield workflow

TUNNEL, some tradeoffs are exposed.

3.1.1 Network Shield

The network shield can be presented as a simpler proxy since its entire process, as referenced to Figure 6 is smaller than that of GHOSTUNNEL. Therefore, we determine objective applications for this proxy, which have fast executions and simple code.

In addition, with the session posted once, there is no need to repeat this process, making its execution faster and more practical.

Still presenting its practicality, the session created with a network shield can be in front of several requests for the applications in the service mesh. In turn, the confidential GHOSTUNNEL only supports requests for only one application in the mesh.

3.1.2 Confidential GHOSTUNNEL

As an advantage for CONFIDENTIAL GHOSTUNNEL, we have to application decoupled from SCONE, making SCONE applications already running can opt for the use of a sidecar.

In addition, basic communication metrics exist that are not present in the network shield, also in GHOSTUNNEL has a built-in status feature that can be used to collect metrics and monitor a running instance. Metrics can be fed into Graphite (or other systems) to see number of open connections, rate of new connections, connection lifetimes, timeouts, and other info.

Another determining factor for using GHOSTUNNEL is that the network shield needs to restart the application to renew the application's certificate since the session is only loaded when the application is started. Thus, we can see that the network shield approach is not ideal for a server, but it can be something that does not worry in client applications. From this negative factor of the network shield, the GHOSTUNNEL stands out about the rival sidecar. As it is decoupled from SCONE, its certificate updates do not affect the application.

Finally, we can observe from the points presented the advantages and disadvantages of both proxies through the table 1.

3.2 Results

In this way, is observed the usage scenarios of both sidecars. Where we can use network shield in applications that are getting ready to go up, and GHOSTUNNEL confidential is used in applications that are already active to start communicating with the mesh.

Table 1. Comparison table between GHOSTUNNEL PROXY and Network Shield

Property	Network Shield	GHOSTUNNEL
Handle requests from multiple applications	✓	X
Decoupled from SCONE	X	✓
Rotate the certificate without stopping the application	X	✓
Metrics	X	✓

Conclusion

In the end, we come to the conclusion of the 3 possible proxies:

- GHOSTUNNEL: was converted to a confidentiality process with SCONE; it helped SCONE applications to communicate with a mesh in a secure way;
- ENVOY: even though it is the most complete sidecar of the three, had problems during the process of confidential compilation; with our current tools it was not possible to proceed with its use as a confidential sidecar;
- SCONE's Network Shield: after a configuration process and using a SCONE plugin, was able to help the communication of confidential applications to the mesh.

With the successful execution of the proxies with confidential GHOSTUNNEL and network shield, and creating a mesh that supports the application of SPIRE, we can stablish communications between SCONE applications and applications present in a mesh. The application uses a secure certificate issued that can provide certificates for applications outside and inside the mesh. That way, confidential applications can make requests with applications present in the mesh. It is also viable to have basic communication metrics that go through one of the proxies. In addition, we showed the advantages and disadvantages of using each proxy, as well delimited the ideal scenarios for their usage.

4. Future Work

We are currently planning modifications to the ENVOY installation method to ease the process of making timely, confidential changes to your code so that it is feasible to make it a confidential proxy. In addition, we intend to create a form of attestation for the applications in the mesh and those in the SCONE environment so that another layer of security is added when performing the request with TLS. As another objective, we have improved the confidential GHOSTUNNEL so that it is possible to receive certificates from sources other than SPIRE and make the process of reading certificates safer.

References

[1] What is a service mesh? **ISTIO**, 2022. Avaliable in: https://istio.io/latest/about/service-mesh/, visited on: 03-14-2022.

[2] Intel® software guard extensions programming reference **Intel SGX**. Avaliable in: https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf, year = 2014,visited on: 03-15-2022.

[3] What is istio **ISTIO**, 2022. Avaliable in: https://istio.io/latest/about/service-mesh/#what-is-istio, visited on: 03-14-2022.

[4] What is istio **REDHAT**. Avaliable in: https://www.redhat.com/en/topics/microservices/what-is-istio, visited on: 03-15-2022.

[5] Spire concepts **SPIFFE**, 2014. Avaliable in:https://spiffe.io/docs/latest/spire-about/spire-concepts/, visited on 03-15-2022.

[6] Spiffe-id **SPIRE**, 2022. Avaliable in: https://github.com/spiffe/spiffe/blob/main/standards/SPIFFE.md#2-the-spiffe-id, visited on:, 03-14-2022.

[7] The-spiffe-verifiable-identity-document **SPIRE**, 2022. Avaliable in: https://github.com/spiffe/spiffe/blob/main/standards/SPIFFE.md#3-the-spiffe-verifiable-identity-document, 03-14-2022.

[8] Spire concepts - attestation **SPIRE**, 2022. Avaliable in: https://spiffe.io/docs/latest/spire-about/spire-concepts/#attestation, visited on: 03-15-2022.

[9] Fetzer; et al SERGEI, Arnautov; Christof. **SCONE: Secure Linux Containers with Intel SGX**. pages 694–695, 2009.

[10] News and overview - scone executive summary **scontain**, 2022. Avaliable in: https://sconedocs.github.io, , visited on: 03-14-2022.

[11] Roee Shlomo Doron Chen and Tomer Solomon. IBM attesting istio workload identities with spiffe and spire, 2020. Avaliable in: https://developer.ibm.com/articles/istio-identity-spiffe-spire/, visited on: 03-15-2022.

[12] What is envoy **ENVOY PROXY**, 2022. Avaliable in: https://www.envoyproxy.io/docs/envoy/latest/intro/what_is_envoy, visited on: 03-14-2022.