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

MATHEUS STHEFANO LEITE DA SILVA

SQUAD: A SECURE, SIMPLE STORAGE SERVICE FOR SGX-BASED MICROSERVICES

CAMPINA GRANDE - PB

2019

MATHEUS STHEFANO LEITE DA SILVA

SQUAD: A SECURE, SIMPLE STORAGE SERVICE FOR SGX-BASED MICROSERVICES

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.

Orientador: Professor Dr. Andrey Elísio Monteiro Brito.

CAMPINA GRANDE - PB

2019

S586s Silva, Matheus Sthefano Leite da.
Squad : a secure, simple storage service for SGX-based microservices. / Matheus Sthefano Leite da Silva. – 2019.

12 f.

Orientador: Prof. Dr. Andrey Elísio Monteiro Brito.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Microservices architecture. 2. Intel SXG. 3. Security. 4. Storage service. 5. SXG-based microservices. I. Brito, Andrey Elísio Monteiro. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

MATHEUS STHEFANO LEITE DA SILVA

SQUAD: A SECURE, SIMPLE STORAGE SERVICE FOR SGX-BASED MICROSERVICES

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 Dr. Andrey Elísio Monteiro Brito
Orientador – UASC/CEEI/UFCG

Professor Dr. Herman Martins Gomes
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Examinador – UASC/CEEI/UFCG

Trabalho aprovado em: 02 de julho 2019.

CAMPINA GRANDE - PB

Squad: A Secure, Simple Storage Service for SGX-based Microservices

Matteus Sthefano Leite da Silva
Universidade Federal de Campina Grande - UFCG
Campina Grande, Brasil
matteus.sthefano.ls@gmail.com

ABSTRACT

Intel SGX has been the subject of numerous research and development projects. Moreover, this technology has been considered a robust option to secure the data being processed in cloud environments. Despite this, configuring SGX-based applications in complex and dynamic scenarios such as microservice architectures is still a challenge. The process of configuring such applications must guarantee the trustworthiness of the services, and must be simple and efficient. We then propose a solution for configuring and provisioning secrets to SGX-based applications made with help of the Intel SGX SDK. We present a simple solution that can be easily validated and hardened. Also, the solution is pluggable and can be extended to fit specific requirements or leverage other tools (e.g., for data persistence). In addition to describing our proposal, we also provide an evaluation that shows low overhead to the initialization and configuration time of SGX microservices deployed on Kubernetes. This work contributes to the state-of-the-art of research on using trusted execution environments in cloud computing.

KEYWORDS

security, microservices, configuration, Intel SGX

1 INTRODUCTION

The microservices architecture pattern has gained importance in the software industry and it has been adopted in a huge number of new projects. Using this pattern, an application is split in a set of microservices, each one concerned in a single aspect or task. The growing adoption of this architecture model is supported by the rise of technologies to manage containers and its communications, like Kubernetes and Docker Swarm.

To secure the data being processed by microservices Intel SGX has been considered as a robust option [9], but there are still challenges when it comes to how to configure and deliver confidential data to these applications. In spite of Intel SGX providing security to the data being processed, providing initial configuration to SGX-enabled microservices built with Intel SGX Software Development Kit (SDK) is not a trivial task. Remotely attesting these services is the most common and secure way to deliver secrets and configuration after the startup [6, 10].

When considering configuring and delivering secrets to a single application in the cloud, the process is straightforward, however, this is not a realistic case. In a cloud scenario, commonly we have a huge number of components running at the same time, working together to achieve a goal. So a more realistic scenario is to have more than one SGX application running, stopping, starting, restarting, scaling horizontally, scaling vertically, within a dynamic workflow. This dynamic feature of cloud computing makes the configuration a difficult process, once every single service that starts up must be attested before receives sensitive information.

The process of configuration must guarantee the trustworthiness of the services, but also must be simple and efficient, without adding extra time to the startup process and minimizing the cost of the currently extremely scarce protected memory.

In this paper, we present Squad: a **S**ecure, **S**imple **S**torage **S**ervice for cloud-based microservices built with the Intel SGX SDK. Squad enables microservices administrators to provide initial configuration and secrets to services running within Intel SGX enabled platforms. The Squad could be instantiated once and is capable to store configuration for an entire microservices ecosystem, attesting multiple services concurrently and providing configuration.

Besides been a multi-tenant service, Squad enables three types of authentication: For early development stages, certificates could be used for controlling the access. For pre-release applications, the secrets can be provisioned only to applications signed by a specific developer. Finally, for production applications, the secrets will be delivered only to a specific version of the application, as attested by Squad with help of Intel SGX, blocking adulteration even by the original developer himself.

This paper is organized as follows: in Section 2 we present a background, explaining TLS protocol, Intel SGX, and Kubernetes. In Section 3 we introduce the related works. An application example that motivates our research is presented in Section 4. Section 5 describes the threat model considered in our work. Next, the proposed solution is presented in Section 6. We then evaluate our solution in Section 7 and discussion results in Section 8. Finally, we finished this paper with Section 9, presenting the conclusions.

2 BACKGROUND

In this section, we provide background information on concepts and technologies to aid the understanding of the context of our work.

2.1 Transport Layer Security

Transport Layer Security (TLS) is a network protocol designed to provide security to communications over computer networks. The first version of the TLS protocol was defined in RFC 2246 [5] but the TLS is constantly evolving. This protocol is the industry standard for securing connections over networks and provides:

(1) Confidentiality: because the data transmitted in a TLS channel is encrypted with a symmetric shared key, derived during the connection handshake.
(2) Integrity: because each message is checked using a message authentication code (MAC) to prevent data loss or a malicious alteration during the transmission.

(3) Authentication: the communicating parties can be authenticated using asymmetric cryptography.

2.2 Kubernetes

Kubernetes is an extensible open-source platform for managing containerized workloads and services [1]. It can orchestrate computing, networking, and storage infrastructure on behalf of user workloads. These features make Kubernetes suitable to be used as a microservices platform. Also, Kubernetes is widely supported. Among the Kubernetes supporters we can highlight Google (Kubernetes was open-sourced by Google in 2014), Amazon Web Services, Microsoft Azure and the OpenStack cloud platform. It provides a declarative application programming interface (API).

Kubernetes presents a certain number of abstractions that represents the state of the systems. In this paper, an important abstraction of Kubernetes to understand is the Pod. A Kubernetes Pod is the basic execution unit of a Kubernetes application, representing processes running on the clusters [2].

2.3 Intel SGX

In this Section, we present a resume about what is Intel SGX and some important features of this technology that our proposal relies on. Most of the information here is taken from Intel SGX developer zone and its documentation and complemented by works from the research community.

Intel Software Guard Extensions (SGX) is a new set of instructions and changes in memory access mechanisms added to Intel Architecture, that is available on recent off-the-shelf processors based on the Skylake microarchitecture or newer. It is a hardware-based technology that allows an application to instantiate a protected area in the application address space. In this paper, we will refer to that protected area as an enclave. The access to memory pages inside enclaves is protected by the SGX-enabled processor [6, 11].

All of the enclaves instantiated within an Intel SGX enabled machine resides into a protected memory area, called Processor Reserved Memory (PRM), which is allocated by the BIOS at boot time. When a processor executes enclave code, it begins to run in enclave mode. This mode changes the memory access process. The CPU memory protection blocks access to the PRM from all external agents, referencing such address to a non-existent memory physical address. Before allowing access to PRM, the hardware checks if the process is running in enclave mode, also if the memory page belongs to the enclave in execution and verifies the correctness of the virtual address.

2.3.1 Programming Model. According to Silva [13], an Intel SGX application is divided into two parts: the unprotected area, that works like a common C/C++ application, and a protected area with one or more enclaves. It's important to note that each enclave has its memory regions isolated from the other enclaves.

The unprotected part of the application runs within the machine main memory. This part is an interface between the enclaves and other applications. It also could be used to process non-sensitive data.

The protected part is composed of one or more enclaves and must be used to process sensitive data. This part should, ideally, manage only the sensitive data, as the operations inside enclaves have the additional cost of the cryptographic processes over the protected memory pages. Another important characteristic is the fact that code running in protected part is not able to make system calls.

2.3.2 Remote Attestation. Anati et al. [6] defines attestation as to the process of demonstrating that a piece of software has been properly instantiated on the platform. As a part of Intel SGX, architecture is the ability to perform remote attestation, allowing thirty party software to gain confidence that an enclave has been properly instantiated in an SGX enabled machine.

The Intel SGX architecture establishes two measurements that can identify an enclave: the enclave identity (1) and the sealing identity (2).

(1) **The Enclave Identity**, also named MRENCLAVE, which is an SHA-256 digest of an internal log that records all the activity while the enclave is built: the relative position of the memory pages in the enclave, security flags associated with these pages as well as the contents of the pages such as code, data, stack, and heap, make up the log.
(2) **The sealing identity** includes Sealing Authority, a product ID and a version number. Typically the Sealing Authority is the enclave builder. The enclave builder signs an RSA enclave certificate (SIGSTRUCT) that contains the expected value of the Enclave Identity (MRENCLAVE), and the public key to verify the signature. The SGX hardware verifies the signature and, if the check pass, a hash of the public key of the Sealing Authority is stored. The value stored is MRSIGNER.

When a **challenger** software wants to gain confidence that an **attester** is running without modifications and in a proper SGX enabled platform, it can remotely attest this application and check the measurements obtained with the expected reference values.

The Intel SGX SDK provides some tools that enable the creation of a structure called QUOTE. The QUOTE keeps information about the enclave identities and about the platform which the enclave is running. The Intel also provides the Intel Attestation Service (IAS): a service that can verify the QUOTE authenticity using a group signature scheme [6].

The remote attestation process, as used in this work, follows the protocol established by Intel in its developer guide reference. The communication involves three agents: the challenger, the attester and the **IAS**. The last one uses a mutual transport layer security as an authentication mechanism. In other words, both IAS and the challenger must present a valid X.509 certificate in order to stablish a communication channel. The simplified message flow [1] is as follows:

- **Message 0:** the **challenger** sends its public key to the **attester**, indicating the intent to start a new attestation process. This public key will be used by the **attester** to create a context for a Diffie-Hellman key exchange (DHKE), that will occur during the remote attestation.
- **Message 1:** after receive the message 0, the **attester** sends its public key to the **challenger**.

[1] Detailed flow, with code examples, is available at https://software.intel.com/en-us/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example.

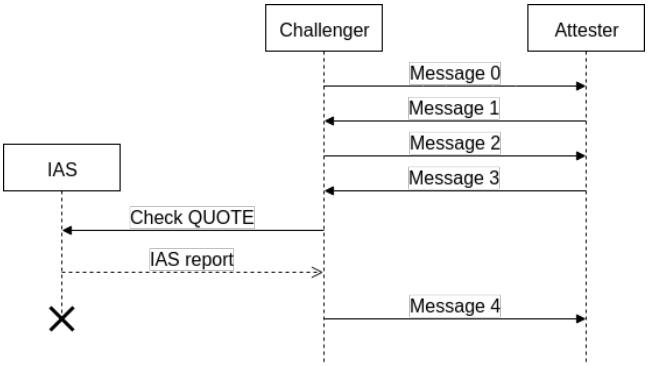

Figure 1: Remote attestation flow. High-level figure based on the remote attestation end to end example provided by Intel.

- **Message 2:** at this point, the **challenger** is able to derive a symmetric shared key (SMK) using the DHKE algorithm. The message 2 is generated containing the **challenger**'s public key, a signature of the concatanation of both public keys (challenger and attester), and a message authentication code (MAC) generated with the shared key.
- **Message 3:** The **attester** checks the message 2 and creates a QUOTE with help of the Quoting Enclave. The QUOTE goes within the message 3 and a MAC is also generated with the SMK.
- **QUOTE Checking:** the **challenger** verifies the MAC and sends the QUOTE to **IAS**. The IAS verifies the QUOTE signature and structure then answering with the Attestation Verification Report (AVR). If the AVR status is "OK", the **challenger** can trust in the content of the QUOTE and then compare against the reference value (MRENCLAVE or MRSIGNER).
- **Message 4:** this message is a confirmation that the attestation process was completed successfully. Within the confirmation, the **challenger** can send secrets to the **attester**.

2.3.3 Sealing. When an enclave is instantiated, Intel SGX ensures integrity and confidentiality to the data being processed, if the data is always within the boundaries of the enclave. However, when the enclave exits, any data within the enclave will be lost. If the data should be used in future executions of the same enclave or other versions this enclave, enclaves should have a method to persist the data for future use.

To achieve the goal of persisting data to future use, in the seminal paper describing the attestation and sealing [6], Anati et al. explains that Intel SGX SDK provides access to persistent Sealing Keys that the developer can use to encrypt and protect the integrity of data saved to machine storage. Protection against replay attacks is also possible using monotonic counters.

There are two policies to the access for the sealing keys. They based in the two identities explained in the Section 2.3.2:

(1) **Sealing to the Enclave Identity** - Used when only the same enclave code, with the same MRENCLAVE, must be able to retrieve the data. It means that not even the new version of the application that sealed the data can retrieve the data successfully.
(2) **Sealing to the Sealing Identity** - Used when the goal is to be able to transfer data to other enclaves or new versions of the same enclave. In opposite to the first policy, this policy uses the MRSIGNER to sealing the data. Thus, the data is accessible to any enclave that has been signed by the same developer.

2.3.4 Protected file system library. The protected file system library is a new feature added to Intel SGX in version 1.7. This library can be used to operate over files like in the regular C file API. The files are encrypted and saved to disk during a write operation. Then, during the read operation, the file is decrypted and verified for confidentiality and integrity. To encrypt a file, the user should provide a file encryption key. This encryption key could be provisioned after the enclave startup to share a file with various enclaves.

2.4 Life Cycle of an Intel SGX Application

Also, according to Silva [13], Intel SGX based applications should be instantiated without sensitive data in memory. Some configurations, such as cryptography keys, are critical and should not be in the code. Moreover, secrets and configurations cannot be delivered to an enclave over a network without checking its integrity and the platform which it is running on. Thus, we can describe the generic life cycle of an SGX application as follows:

(1) Enclave initialization - The insecure part of the application is responsible for creating the enclave. In the initialization process, the MRENCLAVE is built.
(2) Attestation - An attestation is needed for other applications to gain trust in the enclave. The result of the attestation process a secure communication channel between the enclave and the challenger.
(3) Configuration provisioning - the application receives its configuration and sensitive data through the secure communication channel established during the attestation process.
(4) Software evolution - Eventually, any software could need some update. The configuration should be provided to the new version.

3 RELATED WORK

There are some works focused on running Linux applications on Intel SGX enclaves. Among these works we can highlight:

- **SGX-LKL,** which is a library OS designed to run unmodified Linux binaries inside SGX enclaves [4] SGX-LKL is an open-source project[2].
- **Graphene-SGX** is a port of Graphene library OS [14] presented by Tsai et al. [15]. Graphene-SGX was open-source in June 2016[3].
- **Open Enclave SDK**, according to its documentation[4], is an SDK for building enclave applications in C and C++, open-sourced by Microsoft. These enclaves, for now, are Intel SGX

[2] Available at https://github.com/lsds/sgx-lkl
[3] Available at https://github.com/oscarlab/graphene
[4] Available at https://github.com/microsoft/openenclave

enclaves (like explained in Section 2.3), but the documentation cites that other options of trusted execution environments (ARM Trustzone [12]) will be added to the SDK.

- **SCONE** is a platform that facilitates always encrypted execution providing secure containers on top of an untrusted OS [7]. SCONE enables users to run applications inside SGX enclaves without code modifications, transparently to developers and service operators. Fetzer also published work on building critical applications using microservices [9] with the help of the SCONE environment.

Most of the tools above focus on running legacy applications, but migrating complete applications to run inside enclaves can bring complications, as these applications were not intended to run within enclaves. At the very least, porting all the application to an enclave uses additional protected memory. The last one, SCONE, has a built-in system to attest and configuring applications running inside the secure containers. To achieve this goal, the SCONE environment has a dedicated component: the SCONE Configuration and Attestation Service (CAS). The CAS is responsible to attest applications running on secure containers and deliver secrets. In opposition to other solutions to run applications securely, SCONE is not an open-source solution. Moreover, the process to run the applications inside SGX enclaves is transparent and not controlled by the developer. In other words, the developer does not have control about the partitioning of the application in both untrusted and trusted parties (as explained in Section 2.3.1). This lack of control could result in more code inside an enclave than the necessary, increasing the overhead associated with the computation inside enclaves.

4 APPLICATION EXAMPLE

Let us consider a scenario where a data provider generates sensitive location data for various applications interested in consuming these data (e.g., for marketing purposes). Each consumer application has different rights over the data, with different levels of anonymization. Some may have access to the detailed location data, with ZIP codes, street names and so on. Some may have access to the location with low precision, for example, with a precision of a half mile. And some may have access to very low precision, for example, able to determine only the cities that a person have visited. Let us suppose that the data is disseminated via a distributed streaming platform like the Apache Kafka [5]. The applications that consume this data run on a Kubernetes platform.

Each consumer application has its own secure application running on an SGX enabled environment to process the data received. The vendor must deliver the location data to each consumer without revealing the encryption key used to encrypt the data on the vendor's databases.

In such a scenario, the data provider must check the compliance of the consumer's applications regarding the usage of the data. Furthermore, the applications need to get information on how to retrieve the data (e.g., address of the message bus, encryption keys). Of course, these data have to be protected while in transit. One approach is to encrypt the data in the source and then re-encrypt before handling to each consumer so that there is not a single key shared by all applications. Using a single key shared by all applications represents a potential security issue. In possession of the key that encrypts the source, an attack can be facilitated.

Considering this example, each consumer's application must receive a different initial configuration to read the data and know the right encryption key to retrieve data from that location. In addition, the data provider needs to attest each SGX enclave deployed in the cluster to be a consumer application.

However, providing the initial configuration to applications based on Intel SGX is difficult. According to Anati et al. [6] and Knauth et al. [10], the remote attestation process is the best option to assess the endpoint's trustworthiness. Moreover, message 4, described in Section 2.3.2, can deliver secrets to enclaves securely.

Looking for other options to load configuration after the enclave initialization, we can cite sealing and protected file system. If a developer decides to use the sealing functionalities of the SGX SDK, it means that the own application has to previously seal and store the configuration which is infeasible for a scenario with Kubernetes, for instance. In the case of using the protected file system library, the application must have a behavior similar to that of using the sealing option in addition to initially need a key for the encryption of the files.

Therefore, using remote attestation is the most viable choice to provide initial configuration. However, creating an application that attests other applications and provisions them the secrets is a tedious, repetitive work that is prone to failures (e.g., bugs in the application could compromise security). We then propose Squad: **s**ecure, **s**imple, **s**torage **s**ervice.

The Squad can be instantiated once per application owner and is able to provision secrets and initial configuration to multiple applications. Because of its simplicity, it is easily validated and hardened, and therefore can be reused to increase the security of the early bootstrap steps of applications based on Intel SGX.

5 THREAT MODEL

The attack surface for this scenario considers that an attacker can gain privileged access to the machines where the Kubernetes pods are running, trying to compromise the configuration that the SGX applications will receive. It can be done reading the disks associated with these pods or even analyzing the main memory of the processes in execution. Our solution must deal with these threats, avoiding unauthorized from reading the sensitive configuration.

We also consider that an attacker can intercept the messages exchanged between the configuration keeper and either the operator of the cluster or the SGX applications being configured. Nevertheless, we assume that an attacker cannot break state of the art cryptography, used in symmetric and asymmetric encryption schemes.

We assume that the implementation of our approaches, as well of Intel SDKs and processors, are free of bugs. We nevertheless, do not assume that the operating system or hypervisors, which compose most of the code in the software stack, are free of bugs.

Our proposal was designed to run in an adversarial (i.e., untrusted) environment, considering that even the owner of the infrastructure where the Kubernetes is running may be interested in compromising the applications integrity or the data confidentiality.

[5] Information about Apache Kafka is available at https://kafka.apache.org/.

6 SQUAD

In this section, we describe Squad, its architecture and plugins, as well as the communication flow between the services and the applications that request initial configuration and secrets.

The Squad is an SGX-enabled configuration and storage service. It offers a simple and extensible interface which can easily provide configuration for other SGX enabled applications running in a cloud environment. For instance, the SGX-based applications in a Kubernetes cluster can easily get configured at initialization time. The Squad will attest and provide the secrets automatically to every Kubernetes Pod that reaches the *Running* state. The Squad itself can be deployed with the help of Kubernetes.

The remote attestation process was implemented to attest various applications at the same time. Each application being attested keeps its attestation context (four bytes that uniquely identify an attester) and send it in every message. In this way, Squad can handle the attestation process with various replicas of the same application into a Pod.

6.1 Architecture

Squad architecture and implementation are quite simple. And all of the messages that leave the enclave are protected by secure communication channels. This simplicity helps to maintain the ease of update and the small attack surface.

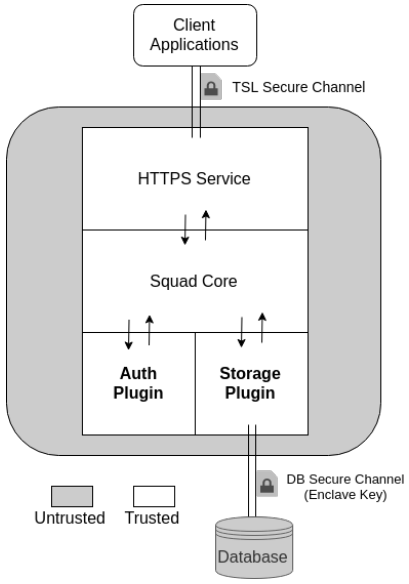

Figure 2: Squad Architecture.

As shown in Figure 2, Squad has the following components:

(1) **The HTTPS service**, which is responsible to expose an HTTP interface over the standard Transport Layer Security protocol to client applications. This component is built on top of a version of the mbedTLS library [3] ported to Intel SGX enclaves, ensuring that every TLS connection is terminated inside the trusted part of Squad [6].

(2) **The Squad core** handles the requests, processes these requests and uses the plugins to obtain a proper response for each request. This component is also responsible for the initial setup of the plugins.

(3) **The *auth* plugins** are the pieces of software responsible for the attestation of the clients, ensuring the authenticity and the access level of these clients. A plugin should receive information about the connected clients and decide on how the Squad core should respond to the requests. This decision is made based on the rules loaded to the storage at the initial configuration time. The default Squad's auth plugin is described in Section 6.3.

(4) **the storage plugins** are the pieces of software responsible for the storage of the data used by the *auth* plugins and the storage of the secrets that will be provisioned. These plugins basically provide an interface between Squad and another entity specialized on storage. By delegating this responsibility, Squad gains simplicity and can take advantage of well-accepted solutions. The default plugin that comes with Squad is also described in section 6.3.

(5) **The insecure portion** is the piece of software that runs outside enclaves. As in any Intel SGX based application, Squad also has an insecure portion. But the only concern of this part of Squad is to forward encrypted, opaque messages, thanks to the TLS connection terminated inside the enclaves.

6.2 Communication Flow

In this section, we will describe how is the communication flow during the configuration of applications in a Kubernetes cluster with Squad. The flow described here considers the default implementation of the authentication and authorization plugin. More about that plugin in the Section 6.3.1.

Figure 3 depicts Squad in the context of a Kubernetes cluster among other applications to be configured (**App A** in focus). Just one instance of Squad is needed to configure the other enclaves in the cluster.

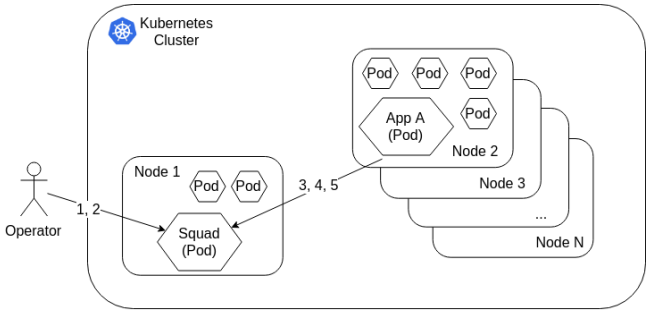

Figure 3: Communication flow.

We divided the communication flow into five steps. Two steps are needed before Squad becomes ready to deliver configuration and secrets to the rest of the cluster. After the two initial steps, three steps of communication with the applications are needed to have

[6] This component has been development jointly with part of the Zero Touch Provision (ZTP) project team (Fábio Fernando de Oliveira Silva, Ionésio Lima da Costa Júnior, Ricardo Araujo Santos, Rodolfo Andrade Marinho Silva), at Laboratório de Sistemas Distribuídos

the configuration provisioned to the applications. The directions of the arrows in the figure indicate the direction of HTTPS requests performed. The first two steps are the following:

- **(1) Initial attestation.** With the Squad property deployed and already running, the operator can use a Python helper application (**squad-cli-helper**, executing in a trusted machine) to attest Squad. This attestation ensures that Squad is running as expected.
- **(2) Loading of the configuration.** The previous step created a secure communication channel, resulting a cryptography key shared in the attestation process. This process is explained in Section 2.3.2. Actually, this step is equivalent to the message 4 shown in Figure 1.

After these steps Squad is ready to feed other applications based on the configuration set loaded. The steps 3, 4 and 5 are requests that the **App A** and the other applications perform to Squad in order to retrieve the desired configuration/secrets.

- **(3) Challenge request.** After the deployment phase, the **App A** sends a POST HTTPS request to Squad, indicating which secrets it wants to receive. As a response to that request, Squad answers with message 0 of the SGX attestation process, sending its public key.
- **(4) Context initialization.** The **App A** will send its public key to Squad (message 1). With this public key, Squad will create a context to keep the data about the attestation that is in progress. The Squad response to this request with the attestation message 2.
- **(5) QUOTE verification and secret's delivery** After check the message 2, the attester will generate a QUOTE structure and it will send a new request with this QUOTE as the payload. In possession of the QUOTE, Squad will make all the checks needed and respond with the secrets pertaining to the **App A**.

6.3 Plugins

As described in the section about its architecture, the Squad can be easily extended with plugins. This feature makes the Squad suitable for various deployment scenarios. The architecture, as previously described, makes it easy to add new functionality to the software and customize the way the software performs authentication, authorization, and storage.

For the experiments of this paper, we have implemented two plugins. The first one is an implementation of an authentication and authorization plugin that can remotely attest Intel SGX enclaves, and then decide if they should have access to some secret. The second one is an implementation of a storage plugin capable of interacting with an SQLite database and store the secrets that Squad is working on.

6.3.1 Auth Plugin. The default authentication (and authorization) plugin enables three types of access-levels. For early development stages, certificates could be used for controlling the access. These certificated could be provisioned using regular tools such as Kubernetes Secrets. For prerelease applications, the secrets can be provisioned only to applications signed by a specific developer. Finally, for production applications, the secrets will only be handed to the specific version of the running application as attested by Intel SGX, blocking adulteration of the applications even by the original developer himself.

For the applications in early development stages, the *auth* plugin is able to verify the certificate, checking for some known certification authority (CA). The plugin also can be used with self-signed certificates, favoring the development process.

For prerelease applications, the plugin leverages **sealing identity** features, explained in the item 2 of Section 2.3.2, to identify an application and the different versions while that application is not stable yet.

Ultimately, the thirty access-level provided is used to production-ready applications. Using the properties of the **enclave identity**, also explained in Section 2.3.2, the plugin is able to verify the identity and integrity of the application's enclave, and then decides when secrets are delivered.

6.3.2 Storage Plugin. The Squad comes with a built-in storage plugin that can communicate with an SQLite database directly from its trusted part. The storage plugin encrypts the data before sending it to the database files and the cryptographic key used in this process can be obtained with help of Intel SGX SDK in two ways: using the **sealing identity** or the **enclave identity**. The first way allows the data to be used by different versions of Squad (e.g. two versions with different auth plugins). The second way will allow only a specific version of Squad to use the data. This plugin was developed porting the SQLite C library to be loaded inside SGX enclaves. The SQlite was chosen for being lightweight adding almost no overhead to Squad.

7 EVALUATION

In this section, we present a set of experiments we considered to understand how much time a secure application needs to start up and configure itself in a microservices scenario considering Kubernetes as the container orchestrator.

The experiments were designed to compare our proposal with an approach using SCONE (discussed in the related work, Section 3), and another approach using a traditional application, not using Intel SGX, which retrieves configuration from environment variables without additional security mechanisms.

7.1 Experiments Setup

To set up the experiments we used two virtual machines. The first one is a virtual machine provisioned by an OpenStack cloud[7]. The virtual machine is equipped with two virtual CPUs, 4 GB RAM, 20 GB of storage and 8 MB of memory dedicated to run SGX enclaves. The second one is a virtual machine with almost the same specification, but without protected memory available.

We deployed the components to run the experiments as follows: in the machine with protected memory available, we deployed a Kubernetes cluster to run the configuration provider (Squad or SCONE CAS) and a sample application written to these experiments. The other machine has the scripts that manage the cluster and loads the set of configuration into the configuration keeper.

[7]Documentation about OpenStack is available at https://www.openstack.org/

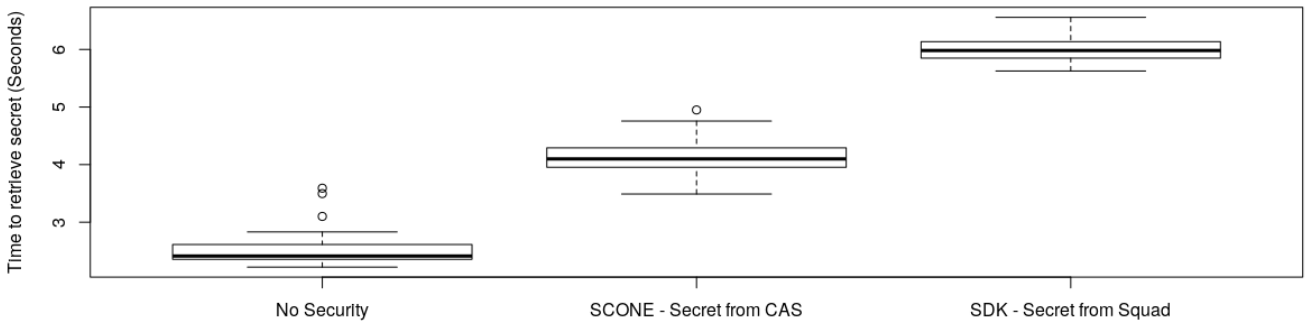

Figure 4: Time needed to retrieve a secret from Squad and SCONE CAS.

In addition, in the machine without protected memory, we deployed an application we named *checkpoint*. This checkpoint is responsible only to receive a request from the applications deployed into the cluster, saving time when this event occurs.

7.2 Scenarios, Metrics, and Parameters

To evaluate the approach used by Squad in relation to other approaches, we planned an experiment where we have a simple application to be deployed with the help of Kubernetes. The application is as simple as possible and has the following lifecycle:

(1) The application starts;
(2) Retrieve the initial configuration from the configuration provider;
(3) With the secret in hand, the application performs an HTTPS *GET* request to the checkpoint ensuring that the process of loading the initial configuration finished and some communication was made.

To understand the cost of using Squad in the Kubernetes deployment, we consider three scenarios:

(1) Scenario with Squad as the secure configuration provider;
(2) Scenario with SCONE CAS as the secure configuration provider;
(3) Scenario without a secure configuration provider. The configuration is provisioned via environment variables without additional security mechanisms.

We have decided to measure the time needed for the application to get configured with each configuration provider. The metric was obtained as the time elapsed from the execution of the Kubernetes's deployment command (`kubectl apply`) to the moment in which the application request arrives in the checkpoint. We also measure the time elapsed from the triggering of the configuration provider until one secret is successfully loaded into it. For both scenarios, we have executed 30 cycles of deployment, to obtain a reliable sample.

7.3 Results

According to the achieved results, seen in Figure 4, the configuration process in scenario where an application retrieves secrets from Squad takes more time than the scenarios with SCONE and without security mechanisms. The insecure application takes 2.53 seconds (on average) to complete the deployment, including the configuration process, until the request to the checkpoint is made. The secure application takes 4.14 seconds, on average, when getting configuration from SCONE CAS and 5.99 seconds, on average, when getting configuration from Squad.

In Figure 5 we can see a comparison between the time needed to deploy and load the configuration into Squad and SCONE. The time needed by Squad to get ready to deliver configuration is, on average, 8.39 seconds. This time is almost double of the needed by SCONE CAS: 4.37 seconds.

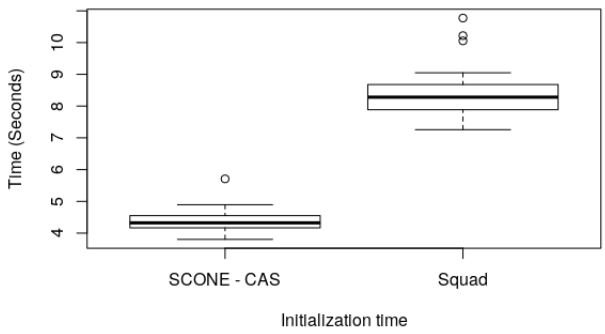

Figure 5: Initialization time.

The higher values for Squad and SDK applications in the experiments were expected because as an application based on the SGX SDK depends on the SGX Platform SoftWare (PSW). In Linux, the PSW is implemented as a service, and this service must be running in order to SGX applications to run on a particular machine. Thus, the Pod in which the application or Squad are running on has to

load the PSW before loading the application, introducing additional delay.

8 DISCUSSION

According to the measured results described in the previous section, the solution proposed to configure Intel SGX SDK applications running in a microservices architecture presents a low overhead to the startup of the services provisioned by Kubernetes.

With the help of Squad, other SGX enabled applications can be easily deployed and configured into Kubernetes. The only additional work is to store the secrets and the authenticated and authorization rules into Squad through the first attestation process, used to gain confidence that Squad is running property.

In spite of the fact of solutions like the SCONE platform making easy to run applications on SGX machines and orchestrators like Docker Swarm and Kubernetes, these platforms take from the developer the control over the applications (which parts will run inside enclaves, as explained earlier in Section 2.3.1) and are proprietary solutions. Our solution can be used directly with applications built with the Intel SGX SDK, in which the developer has complete control over the decision about how the application runs.

Our proposed configuration process does not depends on environment variables or command line arguments as with the SCONE environment. In this way, the size of the data shared is not limited either by the size of an environment variable or a command line argument. In addition, specific needs of various challenges can be achieved using the extensible aspect of Squad, what is difficult with proprietary solutions.

The Squad also deals with the threats mentioned in Section 5. It is robust against the owner of the infrastructure where the Kubernetes runs thanks to Intel SGX technology. The sensitive configuration and secrets are always being processed inside an SGX enclave, inheriting all security properties. Moreover, all the communication with other applications and with the `squad-cli-helper` is made over the TLS protocol, ensuring integrity and confidentiality to the data passing by. Finally, Squad is also attestable. It means that someone using Squad is protected even against the Squad developers, once any adulteration in the expected code will reflect in the Squad's MRENCLAVE.

9 CONCLUSION

Intel SGX has been considered a robust option to build secure applications that process sensitive data. Despite this, there is a lack of solutions to provide initial configuration to enclaves built with the Intel SGX SDK. We then proposed Squad: a secure, simple storage service for SGX-based microservices.

The squad is capable of attesting other SGX applications and provide configuration and secrets over TLS. It can be easily deployed with the help of Kubernetes and in 8.39 seconds, on average, it is ready to attest and deliver secrets to a cluster.

The default authentication plugin that comes with Squad takes advantage of the two identities provided by the Intel SGX architecture and can use the enclave identity or the sealing identity to decide when authorizing the access for some configuration. Another important feature of Squad comes with the default storage plugin. Our solution only maintains in the enclave memory a few pieces of information: the data being processed, the context needed to make the TLS handshakes, and the contexts to the attestation processes. The rest of the data is stored in the encrypted database. In addition, one instance of Squad is needed per cluster and it has fault tolerance following a checkpoint-based rollback-restart approach [8]. If a Squad instance stops for any reason, the new instance can retrieve all the data stored.

In a summary, Squad is intended to be a lightweight and easy-to-use solution to the attestation and secret provisioning of secure application built with Intel SGX SDK, being specially useful when multiple applications are needed, such as in a microservice environment.

ACKNOWLEDGMENTS

To my parents, Socorro and Paulo, for unlimited support. I thank my partner, Jéssica, for her patience in the face of long hours of study.

To my fellow team members along these three years for being part of my personal and professional growth: Adbys, Beatriz, Cleide, Dalton, Elayne, Erickson, Fábio Jorge, Fábio Silva, Gabriel, Igor, Ionésio, Iury, Jefferson, Kaio, Larissa, Luiz Eduardo, Léo, Marcus, Pedro, Ricardo, Roldolfo, Viviane, Ícaro.

To Fatinha and Klébia, for all the support since the first day at UFCG.

To all my professors. In special the educators: Andrey, Franklin, Fubica, Gustavo, Joseana, João, Leandro, Manel, Marcelo, Massoni, Matheus, Neto, Pammella, Peter, Reinaldo.

Last but not least, thanks to my advisor, Andrey Brito, for the three years of advising, and all the walks from the lab to the CAA building.

REFERENCES

[1] [n. d.]. Kubernetes Documentation. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Last access in 2019-06-09.

[2] [n. d.]. Kubernetes Documentation - Pods. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/. Last access in 2019-06-10.

[3] [n. d.]. Mbed TLS. https://tls.mbed.org/. Last access in 2019-06-09.

[4] [n. d.]. SGX-LKL. https://github.com/lsds/sgx-lkl. Last access in 2019-06-11.

[5] [n. d.]. TLS RFC 2246. https://tools.ietf.org/html/rfc2246. Last access in 2019-06-09.

[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13. ACM New York, NY, USA.

[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 689–703. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[8] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. 2002. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.* 34, 3 (Sept. 2002), 375–408. https://doi.org/10.1145/568522.568525

[9] C. Fetzer. 2016. Building Critical Applications Using Microservices. *IEEE Security Privacy* 14, 6 (Nov 2016), 86–89. https://doi.org/10.1109/MSP.2016.129

[10] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. 2018. Integrating Remote Attestation with Transport Layer Security. arXiv:cs.CR/1801.05863

[11] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *Hasp@ isca* 10, 1 (2013).

[12] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. 2016. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*.

IEEE, 445–451.

[13] Rodolfo de Andrade Marinho SILVA et al. 2018. Desafios no desenvolvimento de aplicações seguras usando Intel SGX. (2018).

[14] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 9.

[15] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library {OS} for Unmodified Applications on {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 645–658.