Capítulo 3

Autenticando aplicações nativas da nuvem com identidades SPIFFE[1]

Eduardo Falcão (UFRN), Matteus Silva (UFCG), Clenimar Souza (Scontain), Andrey Brito (UFCG)

Abstract

The purpose of this tutorial is to present how cloud-native applications are authenticated with the zero-trust model and the SPIFFE specification. A distributed application is used as study case and its microservices are attested to receive SPIFFE identities that allow mutual authentication according to the zero-trust principles. To illustrate the advantages of zero-trust model with SPIFFE identities in cloud-native computing environments, we leverage both attestation mechanisms based in Kubernetes as well as mechanisms based in Intel SGX (using the SCONE framework).

Resumo

A proposta deste minicurso é apresentar como aplicações nativas da nuvem são autenticadas utilizando o modelo confiança-zero e o padrão SPIFFE. Uma aplicação distribuída é utilizada como estudo de caso e seus microsserviços são atestados de modo a receber identidades SPIFFE que permitirão autenticação mútua de acordo com os princípios de confiança zero. Para ilustrar as vantagens do modelo de confiança zero com identidades SPIFFE em ambientes de computação nativa da nuvem, usaremos tanto mecanismos de atestação baseados em Kubernetes, como mecanismos baseados em Intel SGX (usando o arcabouço SCONE).

3.1. Introdução

Computação na nuvem é um paradigma bastante conhecido por pessoas da área de Tecnologia da Informação (TI), e até por pessoas que não têm afinidade com a área mas usam

[1]Os códigos e scripts utilizados neste documento, assim como erratas, guias mais detalhados e atualizações podem ser encontrados no repositório do minicurso: `https://github.com/ufcg-lsd/minicurso-sbseg-2021`.

aplicativos e serviços que popularizaram o termo. Surgiu no fim do século XX, e ganhou força no início do século XXI graças à evolução de tecnologias de virtualização e ampla disponibilidade através dos provedores públicos de computação na nuvem. Do ponto de vista da operacionalização de sistemas, sua proposta geral é facilitar e otimizar a alocação de recursos gerando economia financeira e eficiência gerencial de recursos computacionais e humanos [Mell and Grance 2011].

O surgimento da computação na nuvem também impactou a Engenharia e Arquitetura de Software. A princípio, muitos sistemas de propósito geral eram desenvolvidos com arquitetura monolítica e implantados de modo centralizado em servidor único. Ainda antes do surgimento da computação na nuvem já eram construídos sistemas descentralizados, mas em uma proporção menor, pois as tecnologias e técnicas eram complexas e pouco difundidas quando comparadas às disponíveis atualmente. Entretanto, a popularização da computação na nuvem e a evolução de tecnologias de desenvolvimento e implantação facilitaram a criação de sistemas completamente distribuídos. Dentre as arquiteturas de sistemas distribuídos, o padrão arquitetural de microsserviços tem se destacado, sobretudo quando se trata de desenvolvimento de sistemas a serem hospedados na nuvem, pois compartilham alguns princípios semelhantes, como por exemplo, a facilidade e rapidez na alocação e desalocação de recursos e serviços.

Com a adoção desse padrão arquitetural surgiram alguns desafios decorrentes do gerenciamento de diferentes perspectivas de uma aplicação baseada em microsserviços. É evidente que gerenciar a implantação, comunicação, alocação de recursos, e segurança de uma grande quantidade de microsserviços torna-se mais complexo em comparação com aplicações monolíticas. Essa dificuldade levou a comunidade a juntar esforços para a criação de um conjunto de técnicas e ferramentas que viabilizassem a operação de microsserviços: a **computação nativa da nuvem**.

Uma vez que a computação nativa da nuvem facilita a distribuição dos serviços, do ponto de vista de segurança, abordagens tradicionais de proteção baseadas em perímetro têm sido depreciadas pois com vários microsserviços implantados em diferentes plataformas de nuvem, privadas e públicas, não existe mais um perímetro claramente estabelecido como havia antes. A falta de um perímetro claro aumenta a superfície de ataque, problemática esta que aumentou a relevância do **modelo confiança-zero**.

Finalmente, ataques cibernéticos cada vez mais sofisticados têm causado continuamente grandes vazamento de dados em organizações de todo o porte. Por esta razão, além das conhecidas estratégias para proteção de dados em repouso (criptografia) e em trânsito (protocolos de comunicação seguros), a comunidade está discutindo formas de proteger os dados durante seu processamento. Esta última técnica é chamada de **computação confidencial**.

Este trabalho descreve a importância desses três pilares para a criação de aplicações distribuídas que podem ser consideradas seguras diante de um modelo de ameaça rigoroso, no qual até mesmo o provedor de nuvem não é considerado confiável. Em outro trabalho [Brito et al. 2020], nós já descrevemos de forma aprofundada conceitos e práticas relacionados à computação nativa da nuvem e à computação confidencial. Neste trabalho nós revisamos esses conceitos essenciais e focamos em como aplicações nativas da nuvem executando sobre diferentes plataformas e tecnologias podem ser autenticadas

com identidades interoperáveis baseadas no padrão SPIFFE.

Este trabalho está estruturado da seguinte forma. A próxima seção apresenta os conceitos de microsserviços e computação nativa da nuvem, provendo informações básicas sobre a ferramenta mais popular para orquestração de contêineres, o Kubernetes. A seção 3.3 explica o que é o modelo confiança-zero e introduz conceitos básicos sobre identidades de software. Além disso, são apresentados o padrão SPIFFE, a ferramenta SPIRE, e uma demonstração onde serviços são autenticados com identidades SPIFFE. Os conceitos de computação confidencial e o arcabouço SCONE são apresentados na seção 3.4. A seção 3.5 apresenta como estudo de caso uma aplicação distribuída que autentica serviços não-confidenciais e confidenciais com identidades SPIFFE. A seção 3.6 encerra este trabalho apontando os principais desafios e direções de pesquisa.

3.2. Computação nativa da nuvem

A computação nativa da nuvem[2] (ou CNC, do inglês, *cloud-native computing*) é um conjunto de técnicas que permitem a implementação e a implantação de aplicações escaláveis em ambientes dinâmicos, como nuvens privadas, públicas ou híbridas. Aplicações nativas da nuvem apresentam baixo acoplamento, são resilientes, gerenciáveis e observáveis, tendo os estágios do seu ciclo de vida automatizados e integrados sempre que possível. Por estes motivos, estão bem alinhadas à arquitetura de microsserviços, cuja principal forma de distribuição é através de contêineres.

O conceito de computação nativa da nuvem ganhou força com a fundação da *Cloud Native Computing Foundation* (CNCF), em 2015. A CNCF tem como objetivo divulgar e dar suporte à adoção do paradigma de computação nativa da nuvem, e o faz através da organização de eventos e da manutenção de um ecossistema de projetos de código aberto e neutros em relação a fornecedores (*vendor-neutral*). Os projetos apoiados pela CNCF[3] podem ter os seguintes níveis de maturidade: iniciantes, em incubação, ou graduados. Alguns arcabouços bastante conhecidos são projetos considerados graduados pelo CNCF, ou seja, projetos que possuem ampla adoção pela comunidade e em ambientes de produção. Dentre os projetos graduados podem ser mencionados o containerd (execução de contêineres), o Kubernetes (orquestração de contêineres), o Helm (gerenciador de pacotes para Kubernetes), e o Prometheus (monitoramento de recursos). Alguns projetos que estão em incubação pelo CNCF são o gRPC (comunicação via chamadas de procedimento remoto), o SPIFFE (especificação para criação de identidades) e o SPIRE (implementação de referência do SPIFFE).

Os projetos acompanhados pela CNCF são excelentes ferramentas para a construção e operação de aplicações nativas da nuvem, por geralmente promoverem uma utilização eficiente dos recursos da nuvem. Além disso, a utilização de interfaces declarativas e genéricas, que podem ser manipuladas ou integradas por meio de processos automatizados e monitoráveis, faz com que as soluções construídas a partir destes projetos sejam portáveis para diversas infraestruturas e provedores. Neste minicurso construiremos uma

[2]A definição oficial de computação nativa da nuvem pode ser encontrada em `https://github.com/cncf/toc/blob/main/DEFINITION.md`.

[3]A lista dos projetos acompanhados pela CNCF pode ser encontrada em `https://www.cncf.io/projects/`.

aplicação nativa da nuvem que será implantada e executada em contêineres. A aplicação usará Kubernetes para orquestração dos contêineres, e Kafka para comunicação. Finalmente, a aplicação usará identidades SPIFFE para autenticação, e tecnologias de computação confidencial para processamento de dados sensíveis.

3.2.1. Microsserviços

Usar provedores de nuvem para hospedar aplicações e serviços não é novidade. Contudo, nem todas as aplicações conseguem aproveitar a dinamicidade e a elasticidade que provedores de nuvem proporcionam. Aplicações de arquitetura monolítica, por exemplo, bastante comuns antes do surgimento e popularização dos provedores de nuvem, geralmente não permitem uma alocação de recursos mais eficiente e alinhada com as necessidades reais da aplicação. Isto acontece porque a lógica de negócio implementada pela aplicação geralmente tem pontos críticos específicos e que requerem mais recursos do que outros, mas o monólito não oferece a granularidade necessária para alocar mais recursos somente aos pontos críticos, o que resulta em mais custos.

A arquitetura de microsserviços, cuja ideia principal é separar cada ponto específico da lógica de negócio em serviços distintos e com interfaces bem definidas, permite explorar melhor a elasticidade oferecida por provedores de nuvem, já que a aplicação é agora composta por uma série de aplicações menores que se comunicam entre si. Deste modo, a necessidade de alocar mais recursos à aplicação pode ser suprida de forma mais eficiente, já que naturalmente alguns serviços precisam de mais recursos que outros. Além disso, a utilização de microsserviços tende a acelerar o processo de desenvolvimento, onde times menores e mais especializados podem entregar valor mais rapidamente.

Por definição, um microsserviço é um processo coeso, independente, que interage com outros serviços através de mensagens [Dragoni et al. 2017]. Aplicações baseadas em microsserviços, quando bem projetadas, oferecem uma série de benefícios relacionados. Cada microsserviço deve ser configurado com suas dependências de forma autônoma. Isto promove isolamento, fraco acoplamento, e facilita o processo de construção e lançamento de novas versões em produção. Outro benefício do fraco acoplamento é a possibilidade de evolução de um microsserviço, isoladamente, para incorporar novas tecnologias, o que seria impossível para aplicações de arquitetura monolítica.

Idealmente, iniciar e terminar microsserviços deveria ser tão simples e fácil de modo que eles sejam considerados descartáveis. Um fator que contribui para esta característica é o não armazenamento de estado (*stateless*), pois facilita a adequação dos recursos à carga experimentada através da criação e término de instâncias de processamento (escalonamento horizontal). Isto também ajuda na recuperação contra falhas de *software* e *hardware*, pois basta reiniciar os microsserviços com falhas de sistema, ou iniciar uma nova instância do microsserviço em outro servidor que não apresente problemas. Estas e outras recomendações podem ser encontradas em artigos científicos [Khan 2017, Hoffman 2016] ou de forma resumida no manifesto "The Twelve-Factor App" [Wiggins 2017], criado pelos desenvolvedores do Heroku[4].

Aplicações baseadas em microsserviços incorporam os benefícios supramencio-

[4]`https://www.heroku.com/`

nados a um custo de complexidade decorrente da heterogeneidade de tecnologia e plataforma empregadas. Microsserviços coesos são mais fáceis de serem implementados, mas a operação e manutenção de centenas ou até milhares deles não é uma tarefa trivial. A comunicação também pode se tornar difícil de gerenciar. Microsserviços heterogêneos podem se comunicar por diferentes protocolos de troca de mensagens, de forma síncrona ou assíncrona. APIs REST[5] é um exemplo de abordagem síncrona que mitiga a heterogeneidade de comunicação entre microsserviços. O aspecto negativo é que serviços produtores e consumidores ficam acoplados, dificultando a implementação de estratégias de balanceamento de carga, escalonamento de recursos, e recuperação de falhas.

Abordagens assíncronas como, por exemplo, os sistemas publicar-assinar e sistemas de filas de mensagens [Eugster et al. 2003, Dobbelaere and Esmaili 2017], desacoplam produtor de consumidor. Nessas abordagens, mensagens enviadas pelos serviços são agrupadas em *brokers* de mensagens tais como barramentos ou filas. Os serviços interessados (assinantes) podem receber as mensagens seguindo dois modelos de notificação: *pull* e *push*. No modelo *push*, sempre que o *broker* receber uma mensagem ele a encaminhará para os assinantes, enquanto no modelo *pull* os assinantes são notificados de atualizações mas decidem o momento de verificar se o conteúdo novo lhes interessam. Dois exemplos populares de ferramentas de publicar-assinar são o Apache Kafka[6] e o RabbitMQ[7].

Com as mensagens armazenadas nos *brokers*, diferentes políticas de balanceamento de carga podem ser aplicadas, a depender da ferramenta utilizada. No Kafka, por exemplo, a carga pode ser balanceada configurando apropriadamente tópicos e partições. Uma forma simples de lidar com a sobrecarga em um *broker* é monitorar o acúmulo de mensagens e adicionar novas instâncias do microsserviço em momentos de pico. Já para recuperação de falhas, um *broker* assíncrono armazena mensagens por períodos configuráveis (guardando dos últimos minutos ou até últimos dias) e pode reenviá-las para ajudar na restauração de um componente.

Escalabilidade e tolerância a falhas são propriedades que podem ser asseguradas através de diferentes abordagens complementares. No contexto de aplicações CNC, o Kubernetes é uma ferramenta que monitora os estados de contêineres (microsserviços) para decidir se deveria criar ou terminar contêineres para fins de escalabilidade ou recuperação de falhas. A próxima seção apresenta essa ferramenta e seus principais conceitos.

3.2.2. Kubernetes

Kubernetes é uma ferramenta de orquestração de contêineres originalmente desenvolvida pela Google, e fortemente influenciada por sistemas internos de gerenciamento de contêineres da empresa, como o Borg [Verma et al. 2015]. Seu lançamento coincide com um período de crescimento na adoção de contêineres como forma de empacotar, distribuir e implantar aplicações, o que fez surgir a necessidade de soluções para gerenciamento de contêineres em larga escala.

---

[5] API é abreviação para *Application Programming Interface*, e REST é abreviação para *Representational State Transfer*.

[6] `https://kafka.apache.org`

[7] `https://www.rabbitmq.com`

Kubernetes adota um modelo declarativo, que é muito comum no paradigma de computação nativa da nuvem, onde se definem fatos, ou o estado esperado de partes do sistema, em vez de ações, como é no paradigma imperativo. Os nós de controle de um *cluster* Kubernetes possuem um conjunto de controladores que constantemente observam o estado atual do sistema e, se necessário, atuam sobre ele para atingir o estado esperado. Assim, é possível delegar aos controladores o gerenciamento de aspectos como escalonamento, tolerância a falhas, replicação, comunicação em rede, descoberta de serviços (*service discovery*) e gerenciamento de infraestrutura e recursos, permitindo um foco maior na aplicação em si e em sua evolução. Outro ponto que dá grande flexibilidade às APIs de Kubernetes é o sistema de etiquetas e seletores, que permite atribuir etiquetas a praticamente qualquer tipo de recurso, e implementar filtros, ou seletores, para que um conjunto de fatos seja aplicado somente a um subconjunto dos recursos existentes.

Um *cluster* Kubernetes é composto por dois tipos de nós: os nós de controle, também conhecidos por nós mestres, que abrigam o servidor de API do *cluster* e diversos controladores; e os nós trabalhadores, que de fato executam as cargas e aplicações submetidas pelos usuários. Nós trabalhadores também executam o agente de nó do Kubernetes, ou *kubelet*, que é responsável por reportar o estado do nó para os controladores dos nós de controle. O estado do *cluster* é armazenado no banco de dados chave-valor etcd, também localizado nos nós de controle. Para configurações de alta disponibilidade, é recomendado que se tenha ao menos 3 nós de controle. No presente momento, Kubernetes suporta oficialmente até *5000* nós [8] por *cluster*, o que ilustra a capacidade de gerenciar infraestruturas muito grandes.

3.2.2.1. Tipos de Recurso

Qualquer aplicação submetida a um *cluster* Kubernetes é definida em um arquivo, ou manifesto, escrito na linguagem YAML [9]. Por padrão, a API do *cluster* oferece tipos ou recursos primitivos (*resource types*) que oferecem garantias e comportamentos diferentes. Recursos são sempre criados no contexto de um *namespace*. O *namespace* padrão é chamado de *default*, e serviços de controle do *cluster* rodam no *namespace* "kube-system". Destes tipos oferecidos, o mais básico é o *pod*, que representa um conjunto de um ou mais contêineres que funcionam como uma unidade lógica. Por exemplo, um servidor de banco de dados e um *proxy* reverso: embora sejam dois contêineres distintos, eles podem ser vistos como uma unidade lógica de aplicação. Contêineres no mesmo *pod* compartilham o espaço de rede local e podem compartilhar pedaços de sistema de arquivo definidos em volumes. É importante notar que *pods* são considerados efêmeros e *stateless*, podendo ser criados, movidos ou removidos livremente pelos controladores do *cluster* a fim de atingir o estado esperado do sistema. *Pods* não oferecem qualquer garantia de replicação ou tolerância a falha e, por este motivo, devem estar sempre associados a algum controlador de aplicação. Existem quatro tipos básicos de controladores de aplicação, e cada um possui um comportamento específico, como descrito na Tabela 3.1.

Kubernetes também oferece tipos primitivos para possibilitar a comunicação entre

[8] https://kubernetes.io/docs/setup/best-practices/cluster-large/
[9] https://yaml.org

Nome do recurso	Descrição
Deployment	Um controlador de aplicação que assegura uma quantidade determinada de réplicas para um *pod*. Caso o número de réplicas atual não seja o especificado, o controlador cria ou remove réplicas para atingir o estado esperado. É possível modificar o número desejado de réplicas a qualquer momento. Indicado para aplicações *stateless* que executam indefinidamente.
DaemonSet	Um controlador de aplicação que assegura que uma e somente uma réplica do *pod* será executada em cada nó do *cluster*. É possível filtrar nós através de etiquetas e seletores. Indicado para implementar aplicações como agentes de armazenamento, monitoramento e atestação.
StatefulSet	Um controlador de aplicação que assegura uma quantidade determinada de réplicas para um *pod*, mas também lhes atribui uma identidade permanente, o que habilita aplicações *stateful*. O estado em si é guardado em volumes persistentes (*PersistentVolumes*) que sempre é associado à identidade permanente. Volumes persistentes suportam diversos provedores de armazenamento (como NFS, Ceph, ou serviços específicos de provedores de nuvem, como o Azure File). *StatefulSets* são indicados para aplicações que executam indefinidamente e que possuem estado, como um servidor de banco de dados, por exemplo.
Job	Um controlador de aplicação para *pods* que executam até a completude (*run-to-completion*). Indicado para aplicações em lotes (*batch*), como tarefas de análise de dados ou de aprendizado de máquina. É possível definir execuções paralelas e políticas de tolerância a falha. Apesar de simples, este recurso pode ser complementado por outros sistemas para a construção de *batch schedulers* mais complexos [Sampaio et al. 2019].

Tabela 3.1. Controladores de aplicação básicos de um *cluster* Kubernetes.

pods. Cada *pod* possui um endereço IP único na rede interna do *cluster*. Entretanto, por serem tratados como recursos efêmeros, não se deve utilizar os endereços IP dos *pods* para comunicação entre serviços. A solução está no tipo primitivo *Service*, que expõe um determinado conjunto de *pods* e, com isso, oferece um endereço estável para comunicação via rede. Na prática, *Services* funcionam como balanceadores de carga para o conjunto de *pods* que eles expõem, e as requisições são alternadas entre os *pods* disponíveis. Além disso, cada *Service* é registrado no servidor de DNS interno do *cluster*, o que permite que

outros *pods* consigam acessá-los através de seu nome.

Services têm tipos distintos. O tipo padrão, *ClusterIP*, expõe o conjunto de *pods* apenas na rede interna do *cluster*, isto é, apenas para outros *pods* no mesmo *cluster*. Os tipos *LoadBalancer* e *NodePort*, por sua vez, além de possibilitarem a comunicação intra-*cluster*, também expõem o conjunto de *pods* para tráfego externo. A diferença entre esses tipos vive na forma como o conjunto de *pods* é exposto: no tipo *LoadBalancer*, o controlador do *cluster* especializado em APIs de provedores de nuvem (*Cloud Controller Manager*, ou CCM) provisiona um balanceador de carga gerenciado pelo provedor para expor o conjunto de *pods*; no tipo *NodePort*, por sua vez, os controladores de nó abrem uma porta específica, chamada de *nodePort*, em todos os nós do *cluster*. A porta é escolhida aleatoriamente dentro de um intervalo predefinido, chamado de intervalo de *nodePort*, que compreende as portas de 30000 até 32767. O tráfego pode ser direcionado para o endereço IP do nó na *nodePort* escolhida. Não é necessário utilizar o endereço IP de nós que hospedem os *pods*-alvo, pois o tráfego é redirecionado automaticamente para o *pod* adequado.

Outra forma de expor um conjunto de *pods* para o mundo exterior é através do tipo *Ingress*, que também funciona como um balanceador de carga, mas a nível de aplicação, que expõe *Services* (que atuam na camada de transporte) para o mundo exterior. A vantagem do *Ingress* é a sua flexibilidade, já que é possível implementar comportamentos complexos, como *traffic splitting*, *circuit breakers*, *rate limiters*, *canary deployments*, *virtual host routing*, entre outros. O tipo *Ingress*, contudo, não possui um controlador específico por padrão. Cabe ao usuário escolher e instalar um controlador de *Ingress*. Das diversas opções disponíveis na comunidade, destacam-se os controladores NGINX Ingress Controller (baseado em NGINX), *Voyager* e *HAProxy Ingress Controller* (baseados em HAProxy), *Ambassador* e *Contour* (baseados em Envoy Proxy).

Kubernetes também oferece tipos de recursos específicos para a configuração de aplicações. O tipo *ConfigMap* define um conjunto de arquivos que podem ser referenciados dinamicamente por um ou mais *pods* via sistema de arquivos (volumes) ou variáveis de ambiente. O tipo *Secret* também define um conjunto de arquivos, porém com seu conteúdo codificado pelo método Base64, destinado para a arquivos ou dados sensíveis, como credenciais ou senhas. Vale salientar que, embora seu conteúdo não seja de imediato decifrável, a decodificação de Base64 para texto plano é trivial, logo o tipo *Secret* não fornece nenhuma real garantia de confidencialidade ou integridade.

Por padrão, todos os *pods* do *cluster* possuem uma conta de serviço padrão (descrita pelo tipo primitivo *ServiceAccount*), com permissões limitadas. Ao utilizar uma conta de serviço, cada *pod* recebe um *token* de autorização através de um *Secret* montado automaticamente pelo *pod*, chamado de *Service Account Token* (SAT), que pode ser utilizado para se autenticar com outros serviços do *cluster*. Também é possível utilizar *Projected Service Accounts*, num fluxo bastante similar, mas que permite associar parâmetros adicionais ao *Projected Service Account Token* (PSAT), como tempo de vida e metadados. Desta forma, componentes do *cluster* que recebem estes *tokens* podem verificar a sua validade através de uma API especial do *cluster* chamada de *TokenReview API*. Caso os *pods* precisem de mais permissões, como por exemplo para criar recursos no próprio *cluster* através da API do Kubernetes, é necessário utilizar uma conta de serviço

com as permissões adequadas atribuídas no sistema de autorização baseada em papéis (do inglês, *Role-Based Access Control*, ou RBAC).

Uma vez compreendidas a importância do padrão arquitetural de microsserviços, e da ferramenta de orquestração de contêineres, Kubernetes, na construção de aplicações CNC, o próximo passo é entender o modelo confiança-zero e como o padrão SPIFFE facilita a construção de aplicações CNC em conformidade com esse modelo.

3.3. Confiança Zero

Uma consequência de se segmentar uma grande aplicação em vários microsserviços é que a superfície de ataque aumenta substancialmente. Uma aplicação monolítica é comumente hospedada em um único servidor, talvez suportado por serviços como um banco de dados em separado, mas ainda assim a gestão é simples. Desses serviços, apenas uma pequena fração é exposta ao mundo e, portanto, configurar políticas de segurança para gerenciar a exposição de um número reduzido de serviços é mais simples. Esta abordagem de proteção baseada no estabelecimento de um perímetro seguro para a rede cria a sensação de que as entidades dentro do perímetro estão protegidas e são confiáveis. Entretanto, quando algum atacante consegue penetrar o perímetro de segurança, movimentos laterais dentro da rede são difíceis de detectar e proteger.

Deste problema surgiu a ideia do modelo confiança-zero (do inglês *Zero-Trust*). Nesse modelo, por mais que um serviço esteja dentro do perímetro de segurança, onde teoricamente estaria protegido, o serviço não deveria confiar em nada, nem mesmo em outros serviços em tese confiáveis que estejam dentro do perímetro [Rose et al. 2020]. Nesse caso, espera-se o pior: que o perímetro tenha sido penetrado. Portanto, o modelo confiança-zero propõe que camadas extras de segurança sejam aplicadas, além do estabelecimento do perímetro de segurança (quando aplicável). Desse modo, os mecanismos de autenticação e autorização de todos os microsserviços, incluindo os que não são expostos à Internet, deveriam funcionar rigorosamente a todo momento.

É evidente que o modelo confiança-zero tem alinhamento com aplicações baseadas em microsserviços, e consequentemente aplicações CNC. Por outro lado, esse nível de rigor naturalmente torna a implementação das políticas de segurança mais complexas, principalmente para aplicações de microsserviços, dada a sua heterogeneidade. Pensando nisso, especialistas em segurança criaram o SPIFFE [Feldman et al. 2020] (abreviação do inglês *Secure Production Identity Framework for Everyone*): uma especificação que detalha a criação de identidades padronizadas para sistemas dinâmicos em ambientes heterogêneos, como é o caso de aplicações CNC.

As duas principais vantagens de identidades SPIFFE são indiscutivelmente a segurança e padronização. Identidades padronizadas permitem que microsserviços implementados com diferentes tecnologias sejam interoperáveis do ponto de vista de comunicação. A segurança reside na atestação da aplicação solicitante da identidade e no uso de identidades verificável e com robustez criptográfica. Identidades SPIFFE só são emitidas após atestação da integridade do ambiente no qual a aplicação está executando, além da atestação da integridade da própria aplicação. Em posse de suas identidades SPIFFE, as aplicações podem estabelecer comunicação segura usando conexões TLS (abreviação do inglês *Transport Layer Security*) mutuamente autenticadas. Assim, a aplicação do padrão

SPIFFE força as aplicações ou serviços a usar comunicações mutualmente autenticadas, provendo garantias de confidencialidade e integridade.

Sob a perspectiva de projeto e desenvolvimento de aplicações CNC usando identidades SPIFFE, pouco ou nada precisa ser alterado. Aspectos técnicos de emissão e autenticação de identidades podem ser terceirizados para tecnologias como SPIRE (abreviação do inglês *SPIFFE Runtime Environment*[10]) e *proxies* como o Envoy[11] e Ghostunnel[12]. Os aspectos de segurança podem ser mais facilmente configurados com essas aplicações e isso permite que times de desenvolvimento foquem nos aspectos lógicos da aplicação.

A próxima seção apresenta conceitos gerais sobre identidades de *software* e os motivos que levaram à criação do SPIFFE. Detalhes do padrão SPIFFE são apresentados em seguida. Por fim, são apresentados a arquitetura do arcabouço SPIRE e o fluxo básico das principais operações.

3.3.1. Conceitos Básicos sobre Identidades de Software

Identidades de *software* servem para identificar unicamente serviços e aplicações, assim como pessoas também são identificadas no mundo real. Na posse de um documento de identidade, um *software* ou pessoa pode comprovar para um terceiro quem realmente é. Para isso, o documento de identidade, seja ele de um *software* ou de um humano, é submetido a um processo de verificação de integridade.

Documentos de identidade carregam consigo algum pedaço de informação que permita comprovar sua veracidade e integridade. Um passaporte, por exemplo, possui algum tipo de carimbo ou marca d'água extremamente difícil de ser falsificado. Identidades de *software* possuem informações criptográficas que são impossíveis de serem forjadas. Só as Autoridades Certificadoras (ACs) são capazes de criar novas identidades com as informações necessárias para verificação de integridade. No Brasil temos o Instituto Nacional de Tecnologia da Informação (ITI) como AC raiz, que é encarregada de credenciar e descredenciar outras ACs intermediárias. As ACs intermediárias podem emitir identidades/certificados para serem usados por quaisquer outras organizações. Portanto, no processo de verificação de identidade, é fundamental verificar se a identidade foi emitida por uma AC proveniente de uma cadeia de confiança idônea e de reputação.

A técnica mais conhecida e adotada para realizar este processo de emissão e verificação de identidades emprega uma Infraestrutura de Chave Pública (ICP). A Figura 3.1 ilustra o funcionamento básico de uma ICP para emitir e verificar identidades. Em uma ICP, serviços podem requisitar identidades a uma AC na forma de certificados. Para isto, um serviço precisa enviar uma solicitação de assinatura de certificado (ou CSR, abreviação do inglês *Certificate Signing Request*) à AC. Após verificar que as informações prestadas no CSR são verdadeiras, a AC emitirá o certificado assinado digitalmente utilizando sua chave privada. Sempre que quiser comprovar sua identidade para uma aplicação terceira, o serviço simplesmente enviará seu certificado. Para verificar a integridade do certificado, a aplicação verificará se o certificado foi emitido por alguma AC confiável. Isto é feito usando o pacote de confiança disponibilizado publicamente pelas ACs. Esse

[10] `https://spiffe.io/docs/latest/spire-about/`
[11] `https://www.envoyproxy.io`
[12] `https://github.com/ghostunnel/ghostunnel`

pacote inclui a parte pública do par de chaves da AC, e essa chave é usada para verificar a integridade da assinatura digital no certificado.

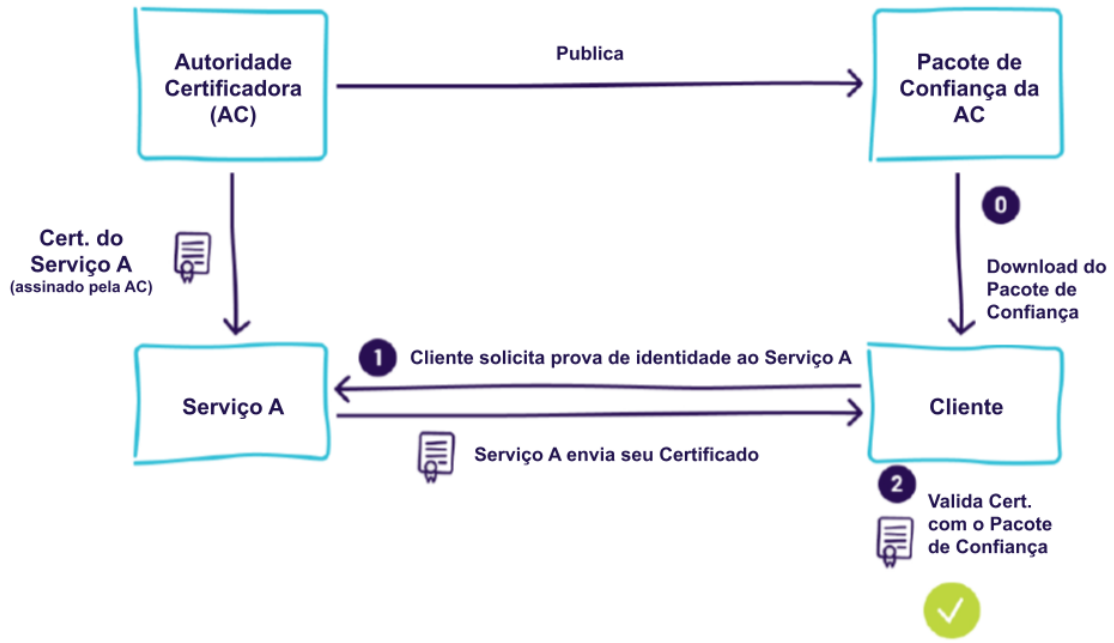

Figura 3.1. Emissão e verificação de certificados em uma ICP. Fonte: traduzido de [Feldman et al. 2020].

Identidades de *software* podem ser usadas para uma série de objetivos. O emprego mais comum de identidades é a autenticação, isto é, identificar-se para um terceiro. Os protocolos mais utilizados para tal são os certificados X.509 e JSON Web Tokens. Uma vez autenticado, um serviço pode utilizar sua identidade para obter autorização para acessar outros serviços com algum nível de restrição. Uma forma simples de implementar autorização é criando uma lista de permissão dos serviços que são autorizados a enviar requisições.

Outra aplicação extremamente importante é a proteção de dados em trânsito. Dois serviços com suas respectivas identidades podem usar o protocolo TLS para criar conexões seguras que permitem a troca de mensagens com confidencialidade e integridade. Cada certificado carrega consigo uma chave pública do serviço que o certificado identifica. Dois serviços que permutam seus certificados conseguem, a partir das chaves públicas, estabelecer uma nova chave secreta e compartilhada entre ambos que será usada para criptografar as mensagens enviadas e descriptografar as mensagens recebidas usando algum algoritmo de criptografia simétrica. No TLS 1.3, o algoritmo para criação dessa chave compartilhada é chamado Diffie-Hellman [Diffie and Hellman 1976]. Para assegurar a integridade das mensagens, cada serviço adiciona ao fim da mensagem um resumo (*digest*) do conteúdo combinado com a chave compartilhada, empregando alguma função *hash*. O algoritmo para criação desse valor de referência para integridade é chamado HMAC [Bellare et al. 1996] (abreviação do inglês *keyed-Hash Message Authentication Code*).

Identidades de *software* têm um ciclo de vida semelhante a identidades de huma-

nos. Depois de criada, toda identidade possui uma data de expiração, e isso leva à necessidade de sua renovação. Além disso, identidades podem ser revogadas. Identidades roubadas, por exemplo, precisam ser revogadas para que outros serviços maliciosos não possam usá-las para se passar pelo serviço dono da identidade. Porém, para que a revogação tome efeito é preciso que a parte verificadora valide se o certificado foi revogado, atualizando a lista de certificados revogados, e às vezes esse processo é negligenciado. Logo, é recomendável que certificados possuam tempo de expiração curto, pois a expiração breve de uma identidade protege contra situações de vazamento/roubo e remove a criticidade de verificadores negligentes que não atualizam com frequência a lista de certificados revogados.

Uma aplicação baseada em microsserviços tipicamente considera uma identidade por microsserviço. No processo de emissão de cada identidade, deveria também haver uma etapa para atestar a integridade do serviço, de modo que as informações declaradas na identidade emitida correspondam de fato ao serviço que as recebem. Considerando que os microsserviços podem ser implementados com diferentes tecnologias, diferentes estratégias de atestação podem ser executadas a depender da plataforma e tecnologia do microsserviço. Outro aspecto relacionado às identidades do microsserviço é o gerenciamento de risco relacionado ao vazamento de identidades. Uma forma de mitigar este problema é ajustando o tempo de validade de uma identidade de acordo com o nível de sensibilidade das informações que cada microsserviço processa. Naturalmente, serviços que processam dados sensíveis precisam renovar suas identidades mais frequentemente. Todo esse contexto apresentado tende a tornar o gerenciamento de identidades mais complexo quando se trata de aplicações baseadas em microsserviços, o que deu origem a uma especificação para produção segura de identidades: o SPIFFE.

3.3.2. Identidades SPIFFE

O SPIFFE é uma especificação [Feldman et al. 2020] que determina como operacionalizar identidades de *software*. Considerando a natureza heterogênea de uma aplicação baseada em microsserviços, o objetivo do SPIFFE é prover interoperabilidade para identidades de *software* de forma agnóstica quanto às plataformas e tecnologias. Para tanto, o SPIFFE define interfaces e documentos necessários para emitir e verificar identidades criptográficas de forma automatizada.

Além dos problemas relacionados à heterogeneidade dos microsserviços, é objetivo do SPIFFE também solucionar problemas relacionados à raiz de confiança. O termo "raiz de confiança" é usado para referir-se a alguma entidade ou componente de *software* no qual pode-se sempre confiar. Portanto, a raiz de confiança é a base de um sistema de autenticação, pois sem ela não seria possível assegurar a veracidade de informações sobre identidades.

A forma mais simples de autenticar um serviço é utilizando algum identificador e senha. No entanto, isso cria a necessidade de gerenciar as senhas dos microsserviços, o que poderia ser realizado através de ferramentas e repositórios de segredos como o *HashiCorp Vault*[13]. Mas para acessar o *HashiCorp Vault* também é necessário que exista alguma autenticação, o que poderia ser realizado por identificador e senha. Logo, é

[13] `https://www.vaultproject.io/`

evidente que a solução de usar senhas para autenticar microsserviços e ferramentas de gerenciamento dessas senhas criará um problema da mesma natureza – sempre existirá uma última senha a ser protegida. O SPIFFE soluciona este problema combinando procedimentos de atestação com a criação de uma cadeia de confiança determinada pelo operador da infra-estrutura.

Para melhor compreender como o SPIFFE soluciona os problemas mencionados é importante conhecer os conceitos básicos da especificação. No vocabulário SPIFFE, cada aplicação ou microsserviço é chamada de carga de trabalho (*workload*), e por esta razão, neste documento utilizaremos esse termo. Note que o conceito de aplicação inclui sistemas de tamanho e complexidade variados, e consequentemente estendemos essa abstração para o conceito de carga de trabalho. Uma carga de trabalho pode ser um servidor web, um banco de dados MySQL, uma aplicação processando itens numa fila, e até mesmo um conjunto de aplicações com propósito único empacotadas em um contêiner ou *pod*. A seguir são apresentados os demais conceitos básicos relacionados ao SPIFFE:

1. SPIFFE ID: forma de representação do nome (ou identidade) da carga de trabalho;
2. Documento de Identidade (ou SVID, abreviação do inglês *SPIFFE Verifiable Identity Document*): um documento que é passível de verificação criptográfica para provar a identidade de uma carga de trabalho a um terceiro;
3. API de Carga de Trabalho: uma API simples, localizada na mesma máquina em que a carga de trabalho executa, usada para emitir identidades sem a necessidade de autenticação;
4. Pacote de Confiança SPIFFE: um formato para representar o conjunto de chaves públicas pertencentes à cadeia de confiança da CA emissora de identidades SPIFFE;
5. Federação SPIFFE: um mecanismo para compartilhar pacotes de confiança SPIFFE e permitir verificação de identidades independente dos limites organizacionais.

Um SPIFFE ID é uma cadeia de caracteres que serve como identificação única para uma carga de trabalho. Ele é formatado como um identificador de recursos uniforme (ou URI, abreviado do inglês *Uniform Resource Identifier*) e possui três partes: o URI, o nome do domínio de confiança, e o nome ou identidade da carga de trabalho. Um exemplo simples de SPIFFE ID poderia ser `spiffe://sbc.org.br/ecos`, onde `spiffe://` é o esquema URI, `sbc.org.br` é o domínio de confiança, e `ecos` seria o nome ou identidade para o serviço de registro em eventos da SBC.

Para qualquer identidade SPIFFE o esquema URI sempre será o mesmo (`spiffe://`). O domínio de confiança corresponde à raiz de confiança do sistema, e poderia representar uma organização ou departamento executando sua própria infraestrutura SPIFFE. Cargas de trabalho que executam no mesmo domínio de confiança recebem documentos de identidade que são verificados com as mesmas chaves raiz do domínio de confiança. No entanto, também é possível usar mais de um domínio de confiança em uma mesma organização, para implementar políticas de segurança mais específicas para emitir identidades em diferentes setores de uma organização. Embora o domínio de confiança de uma

instalação SPIFFE não precise ser ancorada a uma raiz de confiança Web (*e.g.*, o domínio daquela organização na Internet), este vínculo pode existir. Por fim, não existe regra específica para formar nome ou identidade (última parte do SPIFFE ID). As organizações são livres para escolher o formato de nome que faça mais sentido para elas (por exemplo, usando tanto nomes legíveis e descritivos, como códigos opacos).

Uma carga de trabalho é uma aplicação (ou parte de uma aplicação, *e.g.*, microsserviço) responsável por realizar alguma tarefa. No contexto SPIFFE, uma carga de trabalho é tipicamente mais granular do que uma máquina física ou virtual, possuindo, às vezes, a granularidade de um processo. É comum que cargas de trabalho sejam implantadas em contêineres, pois isso ajuda no gerenciamento de dependências e alocação de recursos. Além disso, a conteinerização de cargas de trabalho promove isolamento, o que evita o roubo de identidades por cargas de trabalho maliciosas.

Similarmente como acontece em procedimentos de emissão de documentos de identidades para humanos, a emissão de documentos de identidade SPIFFE (SVID) envolve um procedimento de coleta de informações da carga de trabalho para emitir a identidade com informações íntegras. Este processo soluciona o problema de precisar armazenar senhas quando se emprega alguma autenticação, pois neste caso, a autenticação é substituída pela coleta de informações da carga de trabalho, informações estas suficientes para a emissão do SVID. Cada SVID emitido contém o SPIFFE ID referente à carga de trabalho, e é assinado pela CA que representa o domínio de confiança no qual o serviço está inserido. Por uma questão de facilidade de integração, em vez de criar um novo formato de documento, o SPIFFE adotou formatos já disseminados na comunidade, como certificados X.509 e JWTs. Um SPIFFE ID é embutido em um X509-SVID no campo de extensão "*Subject Alternative Name*". De modo geral, X509-SVIDs são preferíveis a JWT-SVIDs, dado que este último poderia ser interceptado por intermediários (em canais de comunicação inseguros) e usado em ataques de repetição, onde o SVID seria reutilizado por um terceiro para se passar pelo dono da identidade.

Todo domínio de confiança possui um pacote de confiança associado. Esse pacote de confiança é usado por um serviço para verificar a integridade de SVIDs cujo SPIFFE ID pertença ao domínio de confiança relacionado. O pacote de confiança é uma coleção de certificados que contém as chaves públicas das CAs. Como os pacotes de confiança não contém segredos, mas apenas chaves públicas, então eles podem ser compartilhados publicamente. Por outro lado, é necessário distribuí-los de forma segura para evitar modificações não autorizadas, ou seja, é importante assegurar a integridade das chaves.

Existirão situações nas quais cargas de trabalho em diferentes domínios de confiança precisarão se comunicar de forma segura. Isso acontece pois nem sempre é possível agrupar todas elas em um único domínio de confiança. Para ser capaz de estabelecer confiança nas cargas de trabalho é preciso identificá-las, e para identificá-las de modo seguro cada domínio de confiança precisa possuir os pacotes de confiança dos demais domínios. No entanto, por uma questão de segurança, as chaves das próprias CAs são rotacionadas de tempos em tempos, e por isso é importante que haja um mecanismo de compartilhamento automatizado e seguro. O SPIFFE preconiza que os pacotes de confiança sejam permutados através de serviços HTTP protegidos por TLS, chamados de *endpoints* de pacotes. Portanto, operadores que desejem federar sua infraestrutura com outros domínios

de confiança precisam configurar os domínios de segurança terceiros com seus respectivos *endpoints* de pacote, permitindo que os pacotes de confiança sejam recuperados periodicamente.

A API de Carga de Trabalho é uma API local, exposta através de um servidor gRPC, na qual a carga de trabalho usa para obter seu SVID, pacotes de confiança, e a chave privada relacionada ao SVID para assinar dados em nome da carga de trabalho. Para evitar a necessidade de que cargas de trabalho tenham credenciais, essa API não requer autenticação. A API coletará informações do sistema operacional e carga de trabalho em execução para emitir SVIDs com informações de identidade corretas. A API de Carga de Trabalho SPIFFE foi pensada de tal forma que novas soluções de verificação de informações possam ser criadas, provendo interoperabilidade e suporte para novas tecnologias. A comunicação via gRPC é bidirecional. Isto é importante pois pacotes de confiança e SVIDs são rotacionados periodicamente, e quando isto acontece, a carga de trabalho é notificada para atualização. A Figura 3.2 ilustra o funcionamento básico de uma API de Carga de Trabalho.

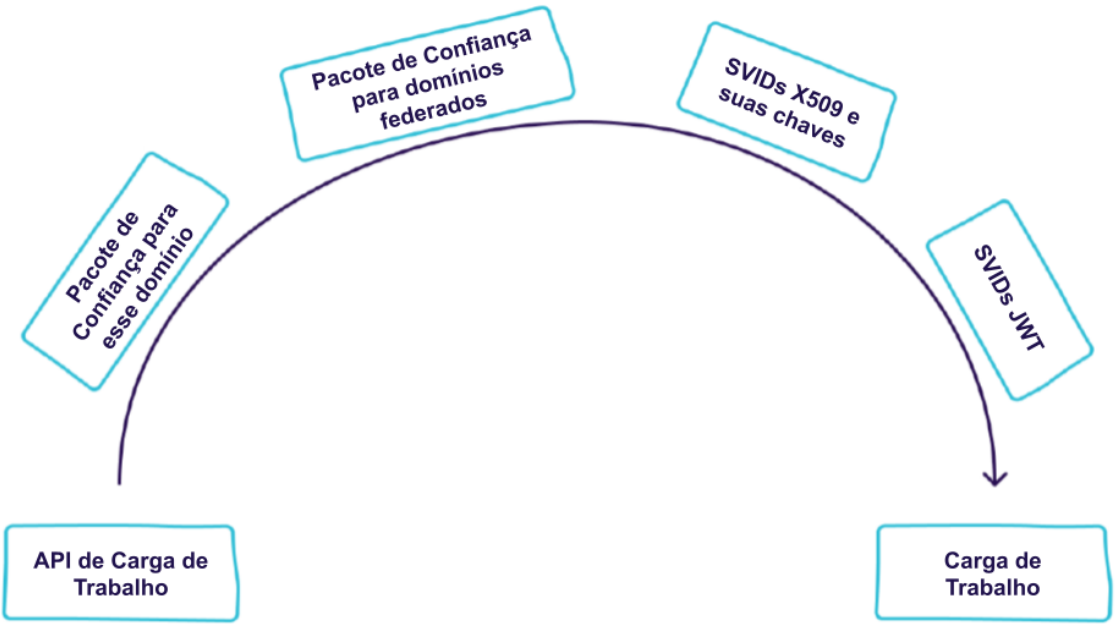

Figura 3.2. API de Carga de Trabalho fornece SVIDs e informações relacionadas. Fonte: traduzido de [Feldman et al. 2020]

3.3.3. SPIRE: *SPIFFE Runtime Environment*

O SPIRE é a implementação de referência do padrão SPIFFE para emissão de SVIDs. Embora existam outras ferramentas que também emitem identidades SPIFFE, tais como Istio, HashiCorp Consul, e Kuma, neste minicurso nós decidimos usar o arcabouço SPIRE pois ele é capaz de emitir SVIDs para serviços executando em variadas tecnologias e plataformas.

O SPIRE tem dois componentes principais: o servidor e o agente. O servidor é responsável por atestar os agentes e entregar-lhes seus respectivos SVIDs. O principal papel do agente é atestar as cargas de trabalho e emitir suas identidades SPIFFE. O SPIRE

é um projeto de código aberto, e ambos os componentes seguem uma arquitetura orientada à *plugins*, o que permite que novos mecanismos de atestação baseados em quaisquer tecnologias possam ser implementados e incorporados ao arcabouço.

Antes de aprofundar-se no funcionamento e características de servidores e agentes é importante compreender a configuração básica de implantação. Uma infraestrutura SPIRE é tipicamente composta por um servidor e múltiplos agentes. Cada agente é implantado em um nó, que geralmente é uma máquina física ou virtual. E cada agente pode servir múltiplas cargas de trabalho, com a restrição de que todas estejam executando no mesmo nó do agente. Um aspecto comum entre servidor e agente é que ambos expõem uma API. A API do servidor, também chamada de API de nó, serve principalmente para que os agentes solicitem suas identidades. Similarmente, a API de Carga de Trabalho serve para que as cargas de trabalho solicitem suas identidades. Uma ilustração dessa configuração básica de uma infraestrutura SPIRE é apresentada na Figura 3.3.

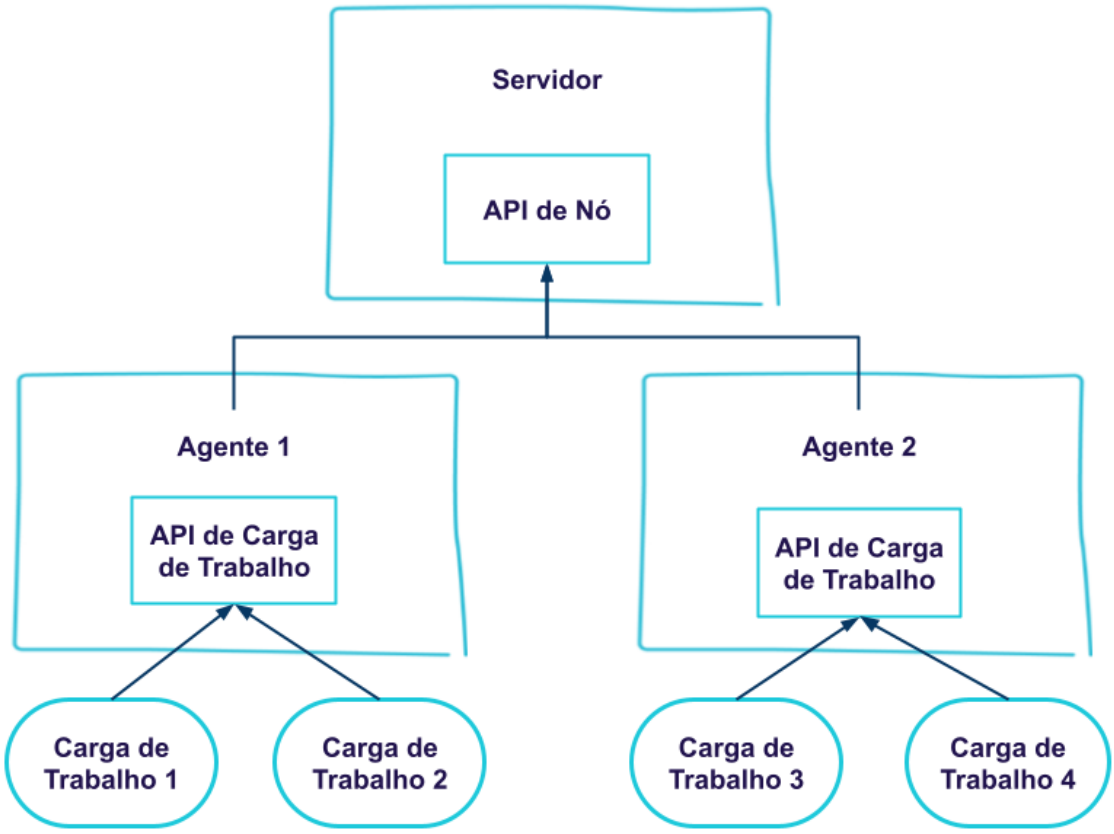

Figura 3.3. Configuração básica de uma infraestrutura SPIRE.

Uma vez compreendidas as noções básicas de uma implantação SPIRE, a seguir são apresentados detalhes de funcionamento do servidor e, posteriormente, são apresentados detalhes de funcionamento dos agentes.

3.3.3.1. Servidor SPIRE

Um servidor SPIRE é responsável por gerenciar e emitir todas as identidades do domínio de confiança no qual foi configurado. Cada SVID emitido é, portanto, assinado pelo servidor SPIRE. Na maioria dos cenários o servidor SPIRE produzirá um certificado auto-assinado para assinar os SVIDs. Esse tipo de configuração é suficiente quando as identidades são usadas dentro da própria organização do domínio de confiança. Em instalações mais amplas e complexas pode ser desejável usar a natureza hierárquica de certificados X.509 para emitir SVIDs assinados por alguma AC publicamente reconhecida. Isso permitiria vários servidores SPIRE serem capazes de emitir SVIDs para um mesmo domínio de confiança. Para esse fim, o SPIRE contém o *plugin* de autoridade *upstream*, que permite obter o certificado de assinatura a partir de outra AC.

O servidor não decide de forma autônoma as identidades a serem emitidas, e para quais nós ou cargas de trabalho elas poderiam ser emitidas. Essas informações precisam ser cadastradas no servidor através de uma interface de linha de comando (CLI, do inglês *Command Line Interface*) ou via API, e são armazenadas em um banco de dados próprio. O conjunto de informações relacionadas a uma identidade é chamado de registro de entrada (Figura 3.4). Como uma identidade pode ser atribuída a um nó ou a uma carga de trabalho, registros de entradas podem ser associados a nós ou cargas de trabalho. Um registro de entrada é composto por: um SPIFFE ID, um conjunto de um ou mais seletores, e o ID do nó (*parent id*). Os seletores são informações coletadas para identificar a carga de trabalho ou nó. O SPIFFE ID é a exata identidade que deve ser emitida para a carga de trabalho ou nó cujo seletor coletado seja igual ao seletor informado no registro. O ID do nó especifica o SPIFFE ID do nó no qual a carga de trabalho precisa estar executando para receber o SVID relacionado àquele registro de entrada. Logo, o ID do nó é aplicável somente para cargas de trabalho.

Figura 3.4. Informações do registro de entrada. Fonte: traduzido de [Feldman et al. 2020].

Documentos de identidade SPIFFE contém informações extremamente importantes acerca da identidade de algum *software*. Logo, é fundamental assegurar a veracidade das informações a serem emitidas no SVID. O SPIFFE preconiza que as informações disponíveis no ambiente de execução (nós e cargas de trabalho) sejam usadas como evidência para comprovação de identidade, e emissão dos SVIDs. Esse processo de coleta

de informações no nó ou carga de trabalho é chamado de atestação.

O servidor é o componente mais crítico da infraestrutura SPIRE, pois ele detém as chaves de assinatura. Um atacante com acesso às chaves de assinatura poderia criar quaisquer novos SVIDs com o SPIFFE ID que desejar. Desse modo o atacante poderia ter acesso a quaisquer serviços e informações dentro do domínio de confiança relacionado àquela chave de assinatura. Por esta razão, o servidor deve ser implantado em uma infraestrutura segura, que comumente é alguma máquina dentro da própria organização, mas podendo ser também uma máquina na nuvem feita confiável com algum mecanismo de proteção de integridade. Esse requisito remove a necessidade de atestação do servidor, pois ele é a raiz de confiança de uma infraestrutura SPIRE.

Pelo fato de ser a raiz de confiança, é o servidor SPIRE que atesta os nós. Ao inicializar, o agente solicita sua identidade ao servidor, que por sua vez inicia a atestação de nó. A atestação envolve algumas trocas de mensagens entre servidor e agente, de modo que o servidor possa obter os seletores do nó para criação do SVID. Essa comunicação é intermediada por um *plugin* do lado do servidor e outro *plugin* no lado do agente, ilustrados na Figura 3.5. Após receber os seletores, o servidor verifica se existe algum registro de entrada que associe algum seletor recebido com algum SPIFFE ID, e em caso positivo o servidor emite um SVID para o agente, juntamente com os registros de entrada associados àquele agente.

Figura 3.5. Arquitetura do *plugin* atestador de nó. Fonte: adaptado e traduzido de [Feldman et al. 2020].

Há três estratégias simples para atestação de nós com sistema operacional Linux e que sejam configurados manualmente: i) *token* de entrada, ii) certificado X.509 e iii) certificado SSH. A primeira estratégia consiste em gerar um *token* no servidor SPIRE e configurar o agente para apresentar o *token* no momento da atestação. Após verificar que o *token* foi de fato gerado pelo servidor, o servidor emite um SVID para o agente. A segunda abordagem emprega certificados X.509. Nesse tipo de atestação os nós são pré-configurados com certificados gerados a partir do certificado raiz ou intermediário, presente no servidor. Sempre que o servidor receber certificados que ele verifique como sendo gerados a partir do certificado raiz, então ele emitirá um SVID para o agente. A terceira opção de atestação de nó consiste em utilizar certificados SSH, que são provisionados automaticamente em alguns nós. O agente pode usar este certificado para receber um SVID com o SPIFFE ID com identidade `spire/agent/sshpop/<hash>`, onde

o *hash* é calculado a partir do próprio certificado. As três estratégias de atestação são ilustradas na Figura 3.6.

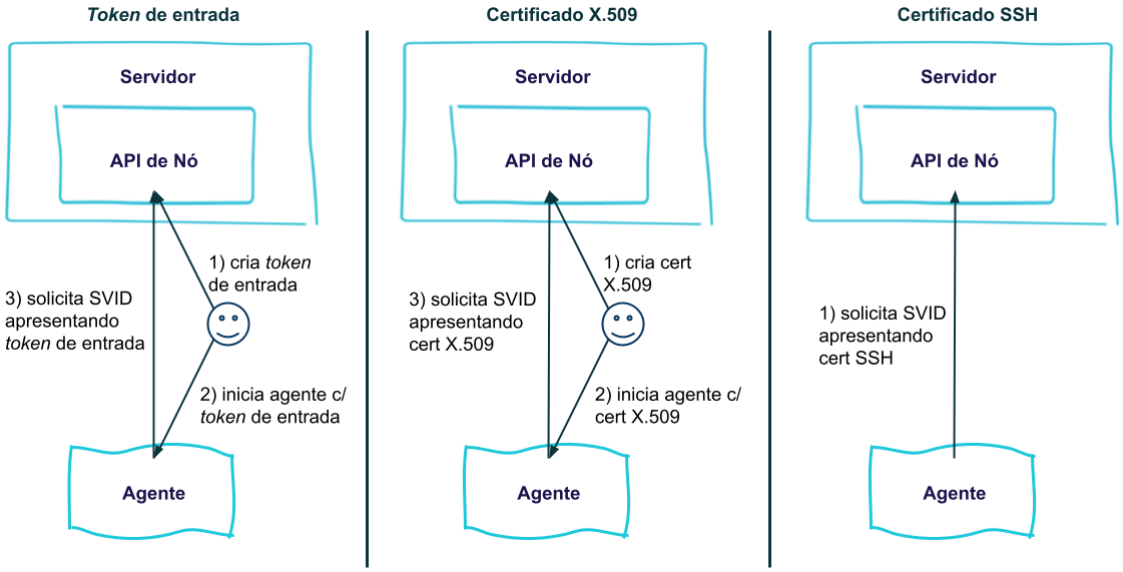

Figura 3.6. Atestação de nós com *token* de entrada, certificado X.509, e certificado SSH.

É interessante perceber que os *plugins* de atestação de nó apresentados na Figura 3.6 não requerem a criação de registros de entrada. O *plugin* que gera um *token* de entrada, por exemplo, usa o próprio *token* gerado como seletor, e o *plugin* de certificados X.509 utilizará o próprio certificado do pacote de confiança para verificar o certificado X.509 recebido pelo nó. Por fim, o *plugin* de certificados SSH sempre irá emitir um SVID para o nó baseado no *hash* do certificado SSH.

É bastante comum que os agentes e suas cargas de trabalho sejam hospedados em nuvens públicas. Muitos provedores de nuvem oferecem APIs que permitem que o nó, que tipicamente é uma VM, possa provar sua identidade. Agentes SPIRE podem usar essas APIs para atestação de nó de modo automático, sem que seja necessário configurar manualmente *tokens* de entrada ou certificados.

Instâncias computacionais EC2 (*Elastic Cloud Compute*) da *Amazon Web Services* (AWS) podem usar o *plugin* de atestação de nó chamado AWS Instance ID (AWS_IID). Com esse *plugin*, um agente é capaz de recuperar um documento de identificação da instância, assinado digitalmente pela AWS, e enviar para o servidor[14], que usará a API da AWS para verificar a integridade do documento. Além disso, o *plugin* AWS_IID é capaz de prover os seguintes seletores: etiqueta (*tag*) da instância, id do grupo de segurança, nome do grupo de segurança, e o papel no sistema de controle de acesso da AWS (*IAM role*). Em posse dos seletores, o servidor verificará se há algum registro de entrada que associe alguma identidade para algum dos seletores recebidos e, em caso positivo, emite um SVID para o agente com SPIFFE ID `agent/aws_iid/<account_id>/<regiao>/<instance_id>`. O fluxo para atestação de instâncias EC2 da AWS

[14]Essa identificação é enviada para o servidor sobre uma conexão TLS autenticada através de um pacote de inicialização no qual o agente é manualmente configurado.

é ilustrado na Figura 3.7. Também existem *plugins* de atestação de nós para instâncias computacionais de outros provedores como o *Google Cloud Platform* e *Microsoft Azure*, todos com a mesma ideia de obter e validar um documento de identificação da instância com o provedor de nuvem.

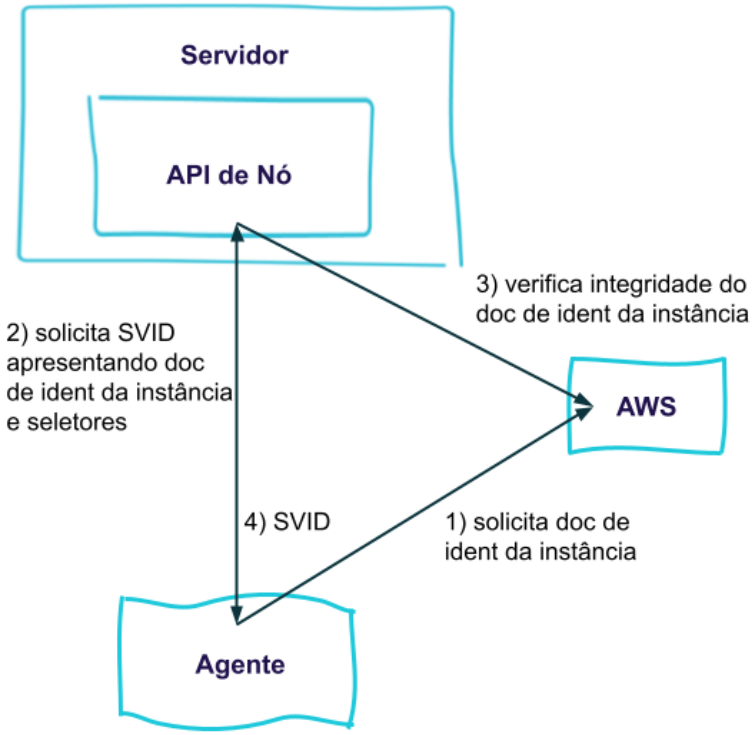

Figura 3.7. Atestação de instâncias EC2 da AWS.

Além da emissão de SVIDs para instâncias EC2 AWS no formato padrão, é possível criar registros de entrada que associem um seletor com um SPIFFE ID específico. Por exemplo, o SPIFFE ID `spiffe://<domínio-de-confiança>/broker` poderia estar em um SVID emitido para um agente SPIRE executando em uma instância EC2 AWS que possuísse uma etiqueta de instância específica, *e.g.*, `tag:app:broker`. Para tal, seria necessário criar um registro de entrada no servidor, utilizando o seguinte comando:

```
$ spire-server entry create
  -node
  -spiffeID spiffe://<domínio-de-confiança>/broker
  -selector tag:app:broker
```

Além dos *plugins* de atestação de nó e autoridade *upstream*, o servidor SPIRE contém *plugins* de banco de dados e de gerenciamento de chaves. O servidor SPIRE necessita de um banco de dados para armazenar, recuperar e atualizar informações como registros de entrada, nós previamente atestados e seus seletores. Esse *plugin* pode ser configurado para utilizar MySQL, SQLite 3 (opção padrão), ou PostgresSQL. O *plugin*

gerenciador de chaves controla como o servidor armazenará as chaves privadas utilizadas na assinatura de SVIDs.

Servidores SPIRE usam um arquivo de configuração para decidir quais *plugins* serão instanciados, além de definir detalhes da API de nó, domínio de confiança e AC. Um exemplo de arquivo de configuração é apresentado na Tabela 3.2. Mais detalhes sobre como configurar o servidor SPIRE e seus *plugins* podem ser encontrados na documentação do SPIRE[15].

```
server {
    # API
    bind_address = "0.0.0.1"
    bind_port = "8081"
    socket_path = "/tmp/spire-server/private/api.sock"
    # Autoridade certificadora e domínio de confiança
    ca_subject {
        country = ["US"]
        organization = ["SPIFFE"]
        common_name = ""
    }
    trust_domain = "example.org"
    data_dir = "./.data"
    log_level = "DEBUG"
}
plugins {
    DataStore "sql" {
        plugin_data {
            database_type = "sqlite3"
            connection_string = "./.data/datastore.sqlite3"
        }
    }
    NodeAttestor "join_token" { plugin_data {} }
    KeyManager "memory" { plugin_data = {} }
    UpstreamAuthority "disk" {
        plugin_data {
            key_file_path = "./conf/server/dummy_upstream_ca.key"
            cert_file_path = "./conf/server/dummy_upstream_ca.crt"
        }
    }
}
```

Tabela 3.2. Arquivo de configuração de um servidor SPIRE.

Por fim, é importante mencionar que a CLI do servidor SPIRE tem uma série de funcionalidades que são muito úteis. Com a CLI é possível: i) criar um *token* de entrada; ii) criar, atualizar, listar, exibir e remover registros de entrada; iii) configurar, listar, exibir e remover pacotes de confiança; iv) banir, listar, exibir e remover agentes atestados;

[15] https://spiffe.io/docs/latest/deploying/spire_server/

v) criar SVIDs no formato X.509 ou JWT; e vi) validar o arquivo de configuração do servidor.

A seguir são detalhados o funcionamento e responsabilidade de agentes SPIRE e como configurá-los.

3.3.3.2. Agente SPIRE

Um agente SPIRE deve executar em cada nó no qual as cargas de trabalho executam. Para receber seu SVID, um agente é submetido a uma atestação de nó, conforme discutido na seção anterior. Depois de atestado, um agente usa seu SVID para se comunicar com o servidor utilizando uma conexão TLS com o objetivo de obter os registros de entradas de carga de trabalho que possuam ID do nó igual ao SPIFFE ID do agente. Em seguida, o agente envia CSRs para o servidor que por sua vez retorna SVIDs para cada registro de entrada de carga de trabalho. Em algum momento as cargas de trabalho solicitarão seus SVIDs através da API de Carga de Trabalho servida pelo agente. A arquitetura do agente é ilustrada na Figura 3.8.

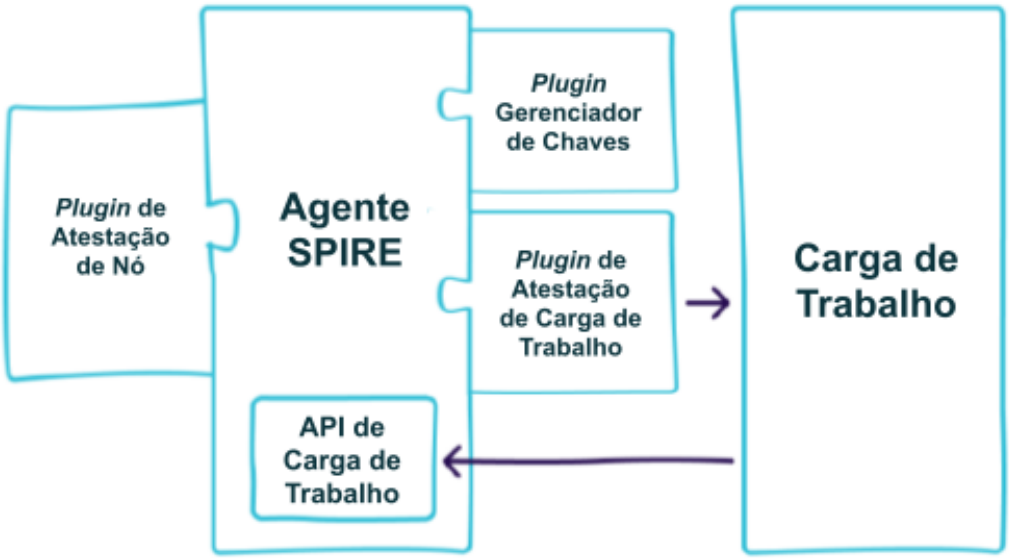

Figura 3.8. Arquitetura do agente SPIRE. Fonte: adaptado e traduzido de [Feldman et al. 2020]

Os principais componentes do agente são o *plugin* de atestação de nó, *plugin* de atestação de carga de trabalho, e *plugin* gerenciador de chaves. O *plugin* de atestação de nó que reside no agente é responsável por coletar as informações que identificam o agente e enviá-las para o servidor. O *plugin* de atestação de carga de trabalho é responsável por coletar as informações que identificam a carga de trabalho e entregá-las para o agente na forma de seletores. Por fim, o *plugin* gerenciador de chaves é encarregado de gerar e usar chaves privadas para os SVIDs X.509 emitidos para as cargas de trabalho.

No momento que um agente recebe uma solicitação de uma carga de trabalho, o agente usará as capacidades do sistema operacional para determinar exatamente qual

processo abriu aquela conexão. No caso de sistemas Linux, o agente executará chamadas de sistema para recuperar o ID do processo, o ID do usuário, e o identificador único global do sistema remoto que está se comunicando via aquele soquete específico. Os metadados do *kernel* são diferentes em sistemas BSD e Windows. Depois de identificar o processo, o agente comunica aos *plugins* de atestação o ID do processo, e a atestação segue coletando informações adicionais daquele processo de acordo com o *plugin* utilizado.

A arquitetura de *plugins* permite que diferentes estratégias e tecnologias possam ser usadas na atestação de carga de trabalho. O exemplo mais simples de *plugin* gera seletores baseados nas informações de *kernel* como usuário e grupo nos quais o processo está executando. Outro exemplo simples é o *plugin* que se comunica com o *daemon* do *Docker* para obter seletores como ID da imagem, rótulos, e variáveis de ambiente do contêiner. Há também outros *plugins* mais complexos como o do *Kubernetes* e *SCONE*, cujos funcionamentos são detalhados na seção seguinte.

Independentemente de qual seja o *plugin* de atestação de carga de trabalho, todos eles seguem um fluxo padrão, ilustrado na Figura 3.9. Primeiramente, a carga de trabalho fará uma chamada à API solicitando um SVID (seta 1). Em seguida, o agente se comunica com o *kernel* do nó para obter os seletores do processo da carga de trabalho (setas 2 e 3). Por fim, o agente confrontará os seletores coletados pelo agente com os registros de entrada providos pelo servidor para decidir se emitirá um SVID para a carga de trabalho e entregará zero ou mais SVIDs à carga (seta 4).

O SVID de cada carga de trabalho é associado a um SPIFFE ID de nó. Quando um registro de entrada de carga de trabalho é criado no servidor SPIRE, além dos seletores e SPIFFE ID da carga de trabalho, também é necessário especificar o SPIFFE ID do nó em que a carga de trabalho deve executar para ser elegível para receber seu SVID. Por exemplo, para o nó com o SPIFFE ID `spiffe://<domínio-de-confiança>/broker` ser capaz de emitir um SVID com `spiffe://<domínio-de-confiança>/queue` para uma carga de trabalho executando neste nó e disparada pelo usuário com ID 1000, seria necessário adicionar um registro de entrada no servidor com o seguinte comando:

```
$ spire-server entry create
  -parentID spiffe://<domínio-de-confiança>/broker
  -spiffeID spiffe://<domínio-de-confiança>/queue
  -selector unix:uid:1000
```

Diferentemente de atestação de nós, o agente SPIRE suporta carregar simultaneamente múltiplos *plugins* de atestação de carga de trabalho. Isso permite combinar seletores para entradas de carga de trabalho. O registro de entrada de carga de trabalho usado como exemplo poderia ser mais robusto se, além do ID do usuário, utilizasse como seletores o caminho para o binário da carga de trabalho e resumo SHA256 do binário da carga de trabalho:

```
$ spire-server entry create
  -parentID spiffe://<domínio-de-confiança>/broker
  -spiffeID spiffe://<domínio-de-confiança>/queue
  -selector unix:uid:1000
```

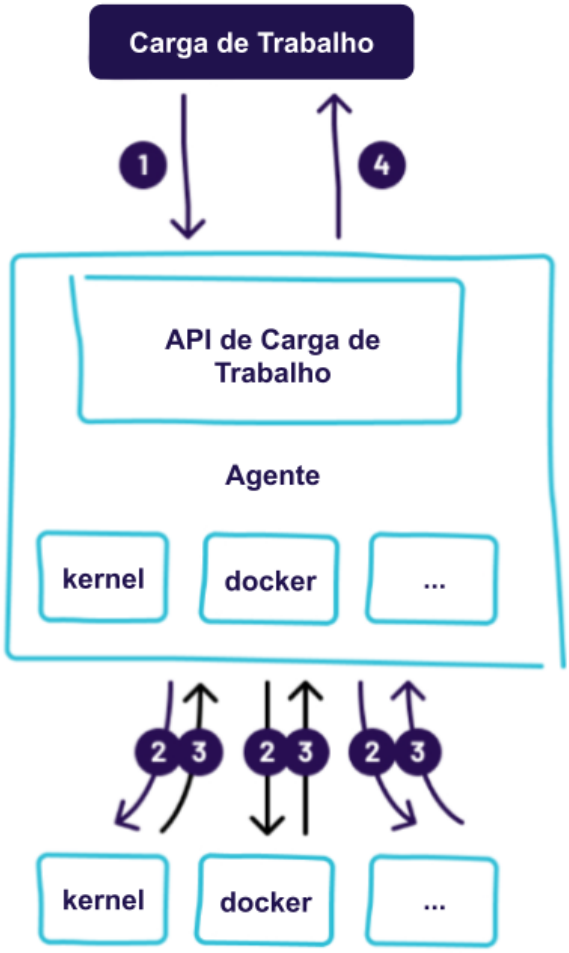

Figura 3.9. Fluxo da atestação de carga de trabalho. Fonte: traduzido de [Feldman et al. 2020]

```
-selector unix:path:/usr/bin/my_daemon
-selector unix:sha256:<SHA256-of-my-daemon>
```

A Tabela 3.3 apresenta os seletores dos *plugins* de atestação de cargas de trabalho *unix* e *docker*. A Tabela 3.4 apresenta o arquivo de configuração para um agente executando com os *plugins* de atestação de cargas de trabalho *unix* e *docker*. Mais detalhes sobre como configurar um agente SPIRE e seus *plugins* podem ser encontrados no guia de configuração dos agentes[16].

Um dos benefícios proporcionados pela infraestrutura SPIFFE é que as identidades possuem tempo de expiração e são renovadas com frequência, com o objetivo de mitigar possíveis roubos de identidades. Agentes SPIRE são responsáveis por gerar novos SVIDs e comunicar essas atualizações às cargas de trabalho interessadas. Os pacotes de confiança também são rotacionados, e os agentes monitoram essas atualizações e notificam as cargas de trabalho. O agente mantém todas essas informações em memória *cache*, de modo que os SVIDs possam ser entregues mesmo que o servidor esteja indisponível. Além disso, essa estratégia diminui os tempos de resposta pois evitam que o

[16] https://spiffe.io/docs/latest/deploying/spire_agent/

Seletor	Descrição
unix:uid	ID do usuário que disparou a carga de trabalho (*e.g.*, **unix:uid:1000**)
unix:user	Nome do usuário que disparou a carga de trabalho (*e.g.*, **unix:user:nginx**)
unix:gid	ID do grupo do usuário que disparou a carga de trabalho (*e.g.*, **unix:gid:1000**)
unix:group	Nome do grupo do usuário que disparou a carga de trabalho (*e.g.*, **unix:group:www-data**)
unix:path	Caminho para binário da carga de trabalho (*e.g.*, **unix:path:/usr/bin/my_daemon**)
unix:sha256	Resumo SHA256 do binário da carga de trabalho (*e.g.*, **unix:sha256:3a6eb0790f39...ac87c94f3856b2d**)
docker:label	Par chave:valor de cada rótulo do contêiner (*e.g.*, **docker:label:com.example.name:foo**)
docker:env	*String* de cada variável de ambiente do contêiner (*e.g.*, **docker:env:DEPLOY_MODE=production**)
docker:image_id	ID da imagem do contêiner (*e.g.*, **docker:image_id:77af4d6b9913**)

Tabela 3.3. Seletores dos *plugins* de atestação de carga de trabalho *unix* e *docker*.

agente precise se comunicar com o servidor para responder a cada solicitação das cargas de trabalho.

3.3.4. Ghostunnel Proxy

Infraestruturas SPIRE são responsáveis por emitir identidades para cargas de trabalho e gerenciar as identidades e pacotes de confiança associados. Uma vez que as cargas de trabalho estejam em posse de suas identidades elas precisam ser capazes de usá-las para comunicação segura via TLS e para autenticação. A maneira mais simples de autenticar serviços e permitir que eles se comuniquem de forma segura é utilizando *proxies* como, por exemplo, o Ghostunnel.

O Ghostunnel é um *proxy* TLS com suporte para autenticação mútua para aplicações que foram concebidas sem suporte à comunicação via TLS. O Ghostunnel pode ser executado em dois modos: cliente e servidor. Um Ghostunnel no modo servidor é implantado na frente de um servidor e aceita conexões TLS. A conexão TLS é então terminada no *proxy* que encaminha no formato plano esperado pelo serviço original (mas visível apenas na máquina local). Esse servidor pode ser uma API aceitando conexões TCP em um domínio e porta específicos, ou um soquete de domínio Unix. No modo cliente, o Ghostunnel aceita uma comunicação insegura de um serviço cliente implantado no mesmo nó e a roteia para o destino final através de uma conexão TLS.

A abordagem mais comum é implantar o Ghostunnel como um *sidecar*[17] na carga

[17]Um *sidecar* é uma espécie de módulo acoplado à carga de trabalho para adicionar funcionalidades.

```
agent {
    data_dir = "./.data"
    log_level = "DEBUG"
    server_address = "127.0.0.1"
    server_port = "8081"
    socket_path ="/tmp/spire-agent/public/api.sock"
    trust_bundle_path = "./conf/agent/root_ca.crt"
    trust_domain = "example.org"
}

plugins {
    NodeAttestor "join_token" { plugin_data {} }
    KeyManager "disk" {
        plugin_data {
            directory = "./.data"
        }
    }
    WorkloadAttestor "unix" {
        plugin_data {
         # Se verdadeiro, o caminho da carga de trabalho será descoberto
         # pelo plugin para prover seletores adicionais.
         discover_workload_path = false

         # Limite do tamanho do binário da carga de trabalho quando for
         # calcular alguns seletores (e.g., sha256).
         # O valor 0 não impõe limite.
         workload_size_limit = 0
        }
    }
    WorkloadAttestor "docker" {
        plugin_data {
         # The location of the docker daemon socket.
         docker_socket_path = ""

         # The API version of the docker daemon.
         docker_version = ""
        }
    }
}
```

Tabela 3.4. Arquivo de configuração de um agente SPIRE.

de trabalho. A ideia de executar o Ghostunnel como um *sidecar* tem como objetivo permitir que ele seja atestado e obtenha uma identidade SPIFFE. Em posse do SVID o Ghostunnel no modo cliente ou no modo servidor será capaz de estabelecer uma conexão TLS com o serviço final. A Figura 3.10 ilustra a implantação do Ghostunnel como *sidecar* para cargas de trabalho em nós diferentes.

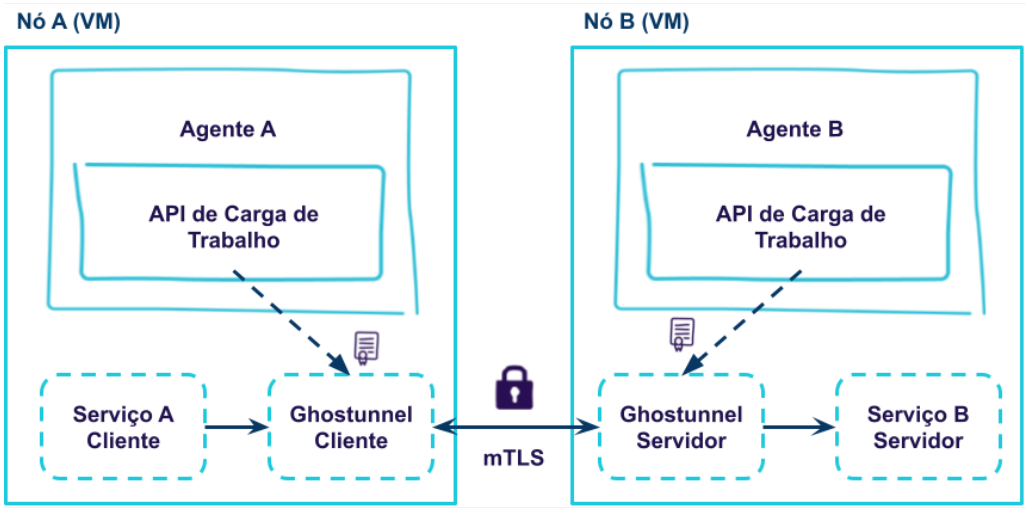

Figura 3.10. *Proxy* Ghostunnel para comunicação segura entre dois serviços via mTLS.

O Ghostunnel no modo servidor vai receber um SVID do agente local, permitindo que clientes remotos confirmem que se comunicam com o servidor correto, e também podem filtrar conexões com origem em cargas com SVIDs com SPIFFE IDs específicos, o que permite limitar a conexão a cargas autorizadas. De modo semelhante, o Ghostunnel no modo cliente recebe um SVID do agente local e usa este SVID para representar o cliente que não possuia um SVID. Além disso, ele também pode filtrar com quais servidores pode se comunicar validando o SVID do servidor.

A Tabela 3.5 lista alguns comandos[18] que permitem reproduzir o caso de uso ilustrado na Figura 3.10. Para fins de simplificação, são usadas cargas de trabalho Unix. Suponha que um usuário com uid 1000 esteja executando uma carga de trabalho Unix. Além de iniciar o serviço A (cliente) o usuário precisaria iniciar um processo do Ghostunnel no modo cliente. Em outro nó é necessário instanciar o serviço B (servidor) e um processo Ghostunnel no modo servidor. Ambos os processos Ghostunnel seriam atestados pelos agentes de seus respectivos nós e receberiam seus SVIDs.

A próxima seção introduz o terceiro pilar para construção de aplicações distribuídas e seguras contra vazamentos de dados: a computação confidencial. Além de apresentar os conceitos básicos de computação confidencial, a seção apresenta o arcabouço SCONE

3.4. Computação confidencial

O crescente número de casos envolvendo o uso indevido de dados pessoais, ou até mesmo o vazamento desses dados, tem aumentado o interesse da comunidade por técnicas que auxiliem a preservar a privacidade dos dados. Para proteger dados em repouso, uma abordagem bastante conhecida e eficiente é criptografar os dados. Os principais cuidados a serem tomados nesses casos são a escolha de um algoritmo de criptografia robusto, uma

[18]Uma descrição completa da demonstração, incluindo detalhes de instalação e configuração de SPIRE e Ghostunnel, pode ser encontrada em `https://github.com/ufcg-lsd/minicurso-sbseg-2021` no diretório `demo-ghostunnel`.

```
# Iniciando o Servidor SPIRE e obtendo tokens de entrada para os agentes
spire-server run -config conf/server/server.conf
spire-server token generate -socketPath ./data/server.sock
    -spiffeID spiffe://example.org/agent1
spire-server token generate -socketPath ./data/server.sock
    -spiffeID spiffe://example.org/agent2

# Iniciando os Agentes em nós diferentes com seus respectivos tokens
spire-agent run -config conf/agent/agent.conf -joinToken "<token>"

# Criando registros de entrada para cargas de trabalho com user id 1000
spire-server entry create -socketPath ./data/server.sock
    -selector unix:uid:1000
    -spiffeID spiffe://example.org/proxy/ghostunnel-client
    -parentID spiffe://example.org/agent1
spire-server entry create -socketPath ./data/server.sock
    -selector unix:uid:1000
    -spiffeID spiffe://example.org/proxy/ghostunnel-server
    -parentID spiffe://example.org/agent2

# Iniciando carga de trabalho nos agentes
# Agente 1: Ghostunnel no modo cliente
ghostunnel-v1.6.0-linux-amd64 client
    --use-workload-api-addr "unix://${PWD}/data/agent.sock"
    --listen=localhost:9001 --target=10.11.19.207:8081

# Agente 2: Ghostunnel no modo servidor
ghostunnel-v1.6.0-linux-amd64 server
    --use-workload-api-addr "unix://${PWD}/data/agent.sock"
    --listen=10.11.19.207:8081 --target=localhost:9001
    --allow-uri spiffe://example.org/proxy/ghostunnel-client

# Demonstração: comunicação mTLS via GHostunnel
# Agente 2
socat TCP-LISTEN:9001,fork STDOUT
# Agente 1
echo "SBSEG2021" | socat STDIN TCP:localhost:9001
```

Tabela 3.5. Comandos para reproduzir demonstração ilustrada na Figura 3.10.

senha forte e o modo de armazenamento dessa senha. Contudo, por mais cuidadosa que seja a aplicação da criptografia, envolvendo todas as etapas de transporte e persistência de dados, os sistemas convencionais vão precisar descriptografar esses dados para o processamento (a realização de consultas, a análise ou transformação dos dados). Essa situação cria uma vulnerabilidade que é a possibilidade de roubo desses dados durante a computação, pois durante a computação é preciso que os dados estejam descriptografados na memória. Deste problema surgiu uma área de estudo chamada computação confidencial,

que trata de propor estratégias seguras para processamento de dados sensíveis.

Há duas principais abordagens para computação confidencial. Uma abordagem utiliza recursos de *hardware*, como por exemplo, a tecnologia Intel Software Guard Extensions (SGX) [Costan and Devadas 2016] que é considerada neste trabalho. A outra é baseada em técnicas algorítmicas, como computação homomórfica [Gentry 2009] (HE, do inglês, *Homomorphic Encryption*) e computação multi-parte [Cramer et al. 2015] (MPC, do inglês, *Multi-Party Computation*).

HE e MPC são técnicas que realizam processamento sobre dados numéricos criptografados. Portanto, o dado só é entregue a entidades não confiáveis após ser criptografado, que neste caso consiste em transformar cada valor numérico em um polinômio. Embora não possam visualizar os dados originais, essas entidades conseguem realizar operações sobre os dados criptografados. Uma das principais diferenças entre HE e MPC é que para HE a computação pode acontecer em um único servidor, enquanto que um aspecto básico do MPC é que múltiplas instâncias de processamento realizam a computação de forma colaborativa. Embora não dependam de *hardware* específico para sua utilização, estas abordagens têm limitações consideráveis, tais como: i) elevado tempo de processamento, pois simples operações aritméticas são convertidas em onerosas operações polinomiais; ii) a complexidade de adaptar uma aplicação para usar essas técnicas [Naehrig et al. 2011, Hayward and Chiang 2015], pois não há ferramentas de uso geral para conversão de código; e, iii) a falta de proteção de integridade para o código, incluindo a falta de um mecanismo de atestação remota.

Assim, as abordagens baseadas em *hardware* têm sido crescentemente utilizadas para resolver o problema de processamento de dados sensíveis. Estas abordagens utilizam tecnologias de computação confiável (TEE, do inglês *Trusted Execution Environments*). Durante a computação, TEEs isolam o código e os dados sensíveis em áreas protegidas da memória, tipicamente chamadas enclaves. Os dados sensíveis e a aplicação que estejam dentro do enclave não são acessíveis por programas não autorizados, ou por usuários com o mais alto nível de privilégio (*e.g.*, usuários administradores), o que permite a computação segura mesmo em ambientes não confiáveis (sejam ambientes remotos, operados por terceiros, ou ambientes locais, mas onde se quer proteger contra ataques internos) [Costan and Devadas 2016]. Do ponto de vista de desempenho, computação confidencial por TEEs é ordens de magnitude mais rápida do que HE e MPC, podendo ter desempenho semelhante ao de aplicações não-confidenciais. Um aspecto negativo é que portar aplicações para serem executadas em TEEs também não é tarefa trivial. A Intel, por exemplo, disponibiliza um kit de desenvolvimento de *software* (SDK, do inglês *Software Development Kit*) para SGX. No entanto, o desenvolvedor precisa entender uma série de aspectos técnicos para incorporar o SDK à sua aplicação. Detalhes de desenvolvimento de aplicações confidenciais usando o Intel SGX SDK podem ser encontrados em [Severinsen 2017].

Uma alternativa que torna mais simples o desenvolvimento de aplicações confidenciais é o uso de abordagens conhecidas como *lift-and-shift*, onde aplicações podem ser executadas em modo confidencial sem a necessidade de modificações. Alguns dos arcabouços *lift-and-shift* mais populares para desenvolvimento de aplicações confidenciais são o GrapheneSGX [Tsai et al. 2017], Fortanix [Leiserson 2018] e SCONE

[Arnautov et al. 2016]. Para este trabalho decidimos utilizar o arcabouço SCONE pois, ao contrário do GrapheneSGX, tem suporte a mecanismos de orquestração de contêineres e suporte ao framework SPIRE. Além disso, ao contrário do Fortanix, possui documentação abrangente[19].

As próximas seções apresentam os principais conceitos relacionados ao arcabouço SCONE.

3.4.1. SCONE

O objetivo do SCONE (abreviação do inglês, *Secure CONtainer Environment*) é facilitar o caminho de um desenvolvedor no processo de migrar aplicações confidenciais para Intel SGX de forma transparente. Nessa abordagem, os desenvolvedores não precisam entender detalhes técnicos de quais partes do código executam dentro do enclave e quais executam fora, pois todo o código do usuário é inserido dentro do enclave e o SCONE injeta código adicional para abstrair operações que não poderiam ser executadas dentro do enclave (como chamadas de sistema).

Além do suporte em termos de *runtime*, são disponibilizados contêineres com as dependências de baixo nível como `musl-libc`, `glibc`, e o próprio compilador `gcc`, já adaptadas para usar o Intel SGX. Portanto, se a linguagem de programação for compilada, desenvolvedores precisam apenas implantar suas aplicações nesses contêineres, usando as bibliotecas adaptadas ou, no pior caso, recompilar as aplicações com o compilador disponibilizado no contêiner. No caso de linguagens interpretadas como Python e Java, imagens estão disponíveis com versões SGX dos interpretadores. Combinando compiladores e interpretadores, SCONE suporta uma variedade de linguagens de programação como C, C++, Java, Python, Go e JavaScript. Durante suas execuções, as aplicações executarão totalmente dentro dos enclaves, protegendo dados e a própria aplicação contra vazamentos ou acessos não autorizados.

Abordagens *lift-and-shift* como o SCONE podem ser usadas para criar microsserviços confidenciais, e isso possibilita que continuemos a explorar as vantagens da utilização de provedores de nuvem sem a preocupação com a confidencialidade da aplicação e dados. Neste trabalho, aplicações confidenciais construídas com SCONE estão sendo chamadas de aplicações de Computação Confidencial Nativa da Nuvem (ou ConfCNC, do inglês *Confidential Cloud Native Computing*). Pelo fato de serem executadas na forma de microsserviços e em contêineres, aplicações ConfCNC são fáceis de serem integradas com uma série de ferramentas populares do ecossistema CNC, incluindo o SPIFFE e SPIRE.

Para entendermos como o SCONE deve ser usado para a migração de aplicações, precisamos entender dois conceitos principais: i) o provisionamento de segredos para aplicações atestadas; e, ii) proteção do sistema de arquivos.

A atestação remota em SCONE é o processo em que uma porção de código da plataforma SCONE garante que a carga de trabalho está íntegra e executando em um processador equipado com Intel SGX. A atestação remota envolve dois componentes: o CAS (do inglês, *Configuration and Attestation Service*) e o LAS (do inglês, *Local Attestation*

[19] `https://sconedocs.github.io`

Service). O CAS é o responsável por atestar uma aplicação SCONE antes de provisionar segredos e configurações de forma segura. O LAS, por sua vez, é o agente responsável pela atestação local da aplicação SGX, gerando um *quote* de atestação que é, então, enviado ao CAS. O *quote* contém informações do enclave a ser atestado, em especial, o MRENCLAVE, um SHA256 da imagem de memória esperada do enclave (incluindo código e as páginas de memória) e o MRSIGNER, que indica o desenvolvedor que assinou aquele código. Além das informações do enclave, o *quote* inclui informações da plataforma, incluindo dados sobre funcionalidades que influenciam a atestação, como a versão do *firmware* do processador e o estado de funcionalidades como o *hyperthreading*. Para gerar o *quote* o LAS precisa executar na mesma máquina que a carga de trabalho. Assim, enquanto um LAS serve um nó, um CAS pode servir um ou mais nós.

O fluxo de atestação funciona da seguinte forma. Ao ser executada, a carga de trabalho aciona código do próprio arcabouço SCONE que foi injetado durante a compilação do código ou do próprio interpretador. Este código injetado aciona o LAS e com o *quote* resultante, entra em contato com o CAS. Cada carga de trabalho deve ser previamente registrada no CAS e esse registro inclui informações do estado esperado do ambiente e informações das configurações esperadas pela aplicação específica. Como parte do estado do ambiente, podemos citar o MRENCLAVE do binário a ser executado e um *hash* da parte do sistema de arquivos relevante para a aplicação (por exemplo, contendo bibliotecas a serem carregadas dinamicamente). Já a parte das informações de configuração de uma aplicação específica podem incluir a definição de variáveis de ambiente, parâmetros de linha de comando ou arquivos secretos, como certificados e chaves.

É importante notar que as informações sensíveis precisam ser provisionadas para a aplicação via CAS e somente depois da atestação, pois só assim se pode garantir que as configurações não foram modificadas e que nenhum segredo foi entregue a uma aplicação que não passaria no teste de integridade.

O cadastro de aplicações no CAS é feito usando as chamadas sessões. Um arquivo de sessão simples é mostrado no Código 3.1. Neste arquivo podemos ver os seguintes componentes:

- O nome do serviço (`alo-mundo`) e o MRENCLAVE do binário que será executado (neste caso, o interpretador Python).
- Os parâmetros a serem passados para o binário (neste caso, o código a ser interpretado).
- Uma variável de ambiente contendo um segredo, variável esta que somente será visível dentro do enclave).
- O caminho para um arquivo que descreve os *hashes* dos arquivos que precisam ser protegidos (`fspf_path`). Neste caso, o próprio código-fonte da aplicação do usuário (`programa.py`) deveria estar protegido, assim como a chave e o *hash* base da parte do sistema de arquivos protegida.

```
1 name: sessao-exemplo
```

```
2  version: 0.3
3  services:
4      - name: alo-mundo
5        mrenclaves: [$MRENCLAVE]
6        command: python3 /app/programa.py
7        pwd: /
8        environment:
9          VARIAVEL_SECRETA: "conteudosecreto"
10       fspf_path: /fspf-file/volume.fspf
11       fspf_key: $SCONE_FSPF_KEY
12       fspf_tag: $SCONE_FSPF_TAG
13
14 security:
15     attestation:
16       tolerate: [debug-mode, hyperthreading, outdated-tcb]
17       ignore_advisories: "*"
```

Código 3.1. Arquivo `sessao.yml` submetido ao CAS descrevendo uma carga de trabalho simples.

A próxima seção ilustra a utilização dos conceitos acima através de um exemplo.

3.4.2. Exemplo de uso do SCONE

Esta seção descreve resumidamente os passos para a execução de uma aplicação SCONE, mais detalhes podem ser encontrados em [Brito et al. 2020]. Assumimos que os componentes básicos como o *driver* SGX e o Docker estão instalados[20].

Para executar um exemplo com o SCONE vamos começar com um código Python (Código 3.2).

```
1 import os
2 print("Alo, mundo!")
3 secret_env = os.environ.get("VARIAVEL_SECRETA")
4 print(f"O segredo é: {secret_env}")
```

Código 3.2. Arquivo `programa.py`, com um programa Python que será executado com SCONE.

Em uma máquina confiável, por exemplo, a máquina local do desenvolvedor, podemos então gerar o sistema de arquivos protegido. O Código 3.3 descreve um *script* para facilitar o trabalho. Ele indica que um diretório será protegido (`app`), mas o diretório raiz não será protegido.

```
1 scone fspf create /fspf/volume.fspf
2 scone fspf addr /fspf/volume.fspf / --not-protected --kernel /
3 scone fspf addr /fspf/volume.fspf /app --encrypted --kernel /app
4 scone fspf addf /fspf/volume.fspf /app /native-files /app
5 scone fspf encrypt /fspf/volume.fspf > /native-files/keytag
```

Código 3.3. Arquivo `fspf.sh`, para proteger o arquivo fonte.

[20]Para versões do Kernel do Linux anteriores à versão 5.11, instruções para a instalação do *driver* estão em `https://sconedocs.github.io/sgxinstall/`. Instruções para o Docker podem ser encontradas em `https://docs.docker.com/engine/install/ubuntu/`.

Em seguida, executamos o *script* gerado usando uma imagem do SCONE (Código 3.4) que inclui alguns utilitários de linha de comando. Ao fim da execução, uma versão criptografada do arquivo `programa.py` será gerada. O *hash* do sistema de arquivos e a chave estarão no diretório do código-fonte, uma vez que devem ficar na máquina do desenvolvedor.

```
# Criamos o diretório para as diferentes versões dos arquivos
# (e para o nosso arquivos de controle, fspf.sh e volume.fspf)
mkdir fspf native-files encrypted-files
cp programa.py native-files/
chmod +x fspf.sh
cp fspf.sh fspf/
export IMAGEM=spire:network-shield-python-alpha3

docker run -it --rm --device /dev/isgx \
 -v $PWD/fspf:/fspf \
 -v $PWD/native-files:/native-files \
 -v $PWD/encrypted-files:/app \
 registry.scontain.com:5050/sconecuratedimages/$IMAGEM \
 bash -c /fspf/fspf.sh
```

Código 3.4. Arquivo `fspf.sh`, para proteger o arquivo fonte.

Para executar esta carga de trabalho dentro de um contêiner Docker, precisamos a seguir de um arquivo de definição de imagem, ou Dockerfile (Código 3.5). Note que apenas a versão criptografada do programa e o arquivo com os *hashes* individuais são copiados. Os dados sensíveis não são copiados (*hash* raiz, chave e código original).

```
ARG IMAGEM=spire:network-shield-python-alpha3
FROM registry.scontain.com:5050/sconecuratedimages/${IMAGEM}
COPY encrypted-files/programa.py /app/programa.py
COPY fspf/volume.fspf /fspf-file/volume.fspf
ENTRYPOINT [ "python3", "/app/programa.py" ]
```

Código 3.5. Arquivo `scone.Dockerfile`, para gerar uma imagem Docker com o exemplo.

Ainda na máquina confiável, o Código 3.6 detalha os passos para a geração da imagem do contêiner e o registro da carga de trabalho no CAS usando a sessão anterior (Código 3.1). Para cadastrar a carga de trabalho precisamos substituir os valores do MRENCLAVE, do *hash* do sistema de arquivos (`fspf_tag`) e da chave do sistema de arquivos (`fspf_key`).

```
# Geração da imagem
docker build . -t sbseg-alo-mundo-scone -f scone.Dockerfile

# Para a primeira execução precisamos descobrir o MRENCLAVE
# Vamos fazer isso dentro do contêiner
docker run -it --rm --device /dev/isgx --entrypoint /bin/bash \
    sbseg-alo-mundo-scone

# Dentro do contêiner, descobrimos o MRENCLAVE do binário com:
```

```
10 SCONE_HASH=1 python3
11
12 # Vamos usar um servidor CAS público
13 # Antes de usar, precisamos estabelecer a confiança nele.
14 scone cas attest 5-4-0.scone-cas.cf -GCS
15     --only_for_testing-trust-any --only_for_testing-ignore-signer \
16     --only_for_testing-debug
17
18 # Finalmente, enviamos a sessão para o CAS público
19 scone session create sessao.yml
```

Código 3.6. Passos para a geração da imagem e registro da carga de trabalho.

O último passo, descrito no Código 3.7 é a execução da aplicação, na imagem do contêiner gerado. Como mencionado anteriormente, para executar uma aplicação SCONE, a máquina precisa ter um componente LAS executando localmente. Assim, o trecho de código mostra como executar um LAS e configura algumas variáveis relevantes: o endereço do LAS (local via Docker), o endereço do CAS (remoto, estamos usando um servidor público) e o nome do serviço a ser executando (o nome do serviço e da sessão onde ele está definido).

```
1  # Configurando o CAS remoto, o LAS local e o nome da sessão
2  export SCONE_CAS_ADDR=5-4-0.scone-cas.cf
3  export SCONE_LAS_ADDR=172.17.0.1
4  export SCONE_CONFIG_ID=sessao-exemplo/alo-mundo
5
6  # A máquina precisa ter um LAS executando localmente, por exemplo:
7  docker run -dt --rm --name las --device /dev/isgx -p 18766:18766 \
8    registry.scontain.com:5050/sconecuratedimages/spire:las-scone5.4.0
9
10 # Execução remota (a imagem deveria ser disponibilizada)
11 docker run -it --rm --device /dev/isgx \
12   -e SCONE_CAS_ADDR=$SCONE_CAS_ADDR \
13   -e SCONE_LAS_ADDR=$SCONE_LAS_ADDR \
14   -e SCONE_CONFIG_ID=$SCONE_CONFIG_ID \
15   sbseg-alo-mundo-scone
```

Código 3.7. Executando a aplicação remotamente.

Como resultado da execução o código de exemplo terá acesso à variável de ambiente com o segredo. Caso o código do interpretador não passe na atestação, o código fonte ficará indisponível, pois o binário não terá acesso à chave de descriptografia.

3.5. Estudo de Caso

Os conceitos e técnicas providos pelo paradigma de computação nativa da nuvem, o que inclui o padrão SPIFFE e ferramentas Kubernetes e SPIRE, tornam mais eficientes a construção e operação de aplicações distribuídas a serem servidas na nuvem. O SPIFFE facilita a implantação do modelo confiança-zero, facilitando a autenticação dos serviços e permitindo comunicação sobre canais seguros através do protocolo TLS. Finalmente,

tecnologias de computação confidencial podem tornar tais aplicações ainda mais seguras visto que protegem o dado durante seu processamento, mesmo que o atacante tenha controle do provedor de infra-estrutura.

Para um aprendizado prático, apresentamos um estudo de caso que considera uma aplicação distribuída construída sobre esses três pilares fundamentais. A aplicação é composta por um microsserviço produtor não-confidencial, e outro microsserviço consumidor confidencial utilizando Intel SGX através do arcabouço SCONE. A ferramenta SPIRE é utilizada para gerenciamento das identidades SPIFFE que são empregadas para autenticação e comunicação segura. Para a comunicação usamos o Apache Kafka, e portanto o serviço produtor (não-confidencial) envia mensagens para o serviço consumidor (confidencial) através de um barramento de mensagens. A Figura 3.11 ilustra os microsserviços, os componentes básicos do SPIRE, e o fluxo que permite a emissão de identidades aos microsserviços

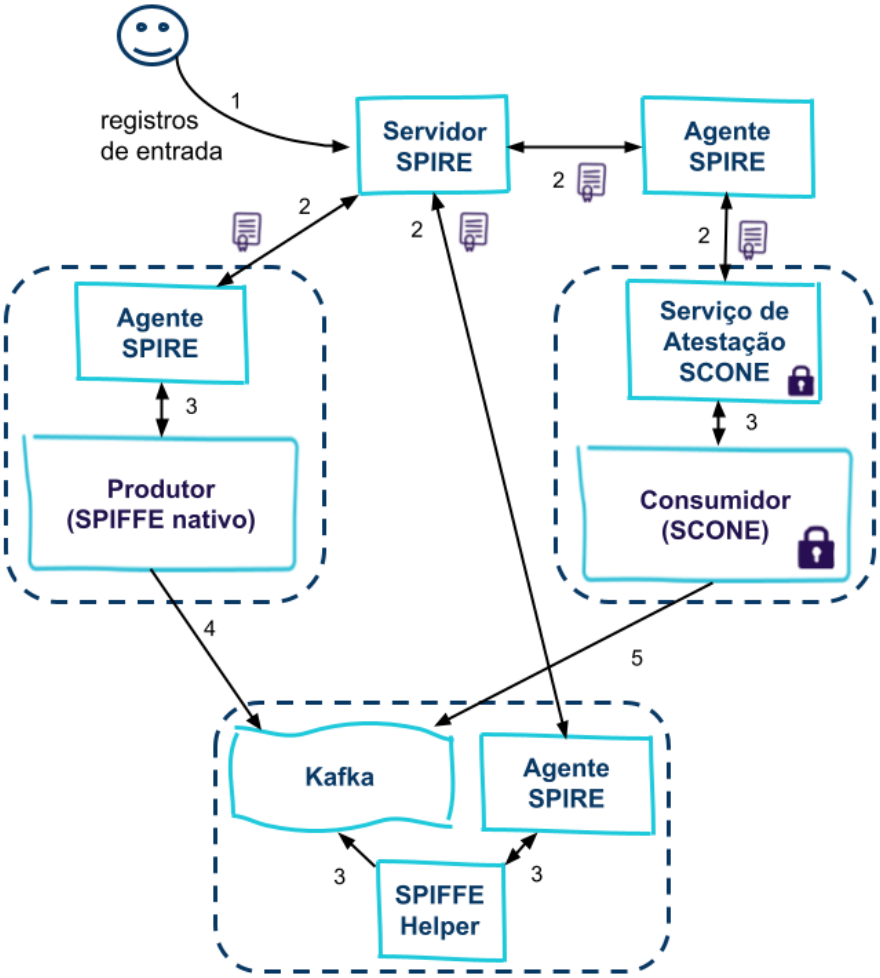

Figura 3.11. Fluxo do estudo de caso.

O propósito da aplicação distribuída foi simplificado para facilitar a compreensão dos aspectos técnicos relacionados às identidades SPIFFE. O serviço produtor gera periodicamente identificadores universalmente únicos (UUID, do inglês *universally unique*

identifier) e os publica no Kafka. O serviço consumidor se inscreve no tópico e recebe os *UUIDs* produzidos. Ambas as aplicações se comunicam com o Kafka e empregam identidades SPIFFE para estabelecimento de um canal protegido por TLS mutuamente autenticado.

O produtor é um serviço escrito em *Golang*, projetado para consultar de forma autônoma a API de Carga de Trabalho para recuperar sua identidade antes de começar a publicar mensagens no Kafka (seta 4 da Figura 3.11). O Código 3.8 apresenta um trecho da implementação do produtor responsável por configurar a obtenção do SVID junto ao SPIRE (seta 3 da Figura 3.11), além de configurar o SVID para comunicação segura com o Kafka.

```
1  import (
2          //outros pacotes
3          "github.com/spiffe/go-spiffe/v2/spiffeid"
4          "github.com/spiffe/go-spiffe/v2/spiffetls/tlsconfig"
5          "github.com/spiffe/go-spiffe/v2/workloadapi"
6          kafka "github.com/segmentio/kafka-go"
7  )
8
9  // Configuração da API de Carga de Trabalho
10 source, err := workloadapi.NewX509Source(ctx,
11     workloadapi.WithClientOptions(workloadapi.WithAddr(APISocketPath)))
12
13 // SPIFFE ID do Kafka e configuração do TLS
14 spiffeID := spiffeid.Must(spiffeTrustDomain, kafkaID)
15 tlsConfig := tlsconfig.MTLSClientConfig(source, source,
16     tlsconfig.AuthorizeID(spiffeID))
17
18 // Usando o dialer do Kafka com configurações de TLS
19 dialer := &kafka.Dialer{
20         Timeout:   10 * time.Second,
21         DualStack: true,
22         TLS:       tlsConfig,
23 }
```

Código 3.8. Serviço produtor: trecho com configuração para identidades SPIFFE.

O Kafka, por ser uma aplicação legada, não possui integração nativa com SPIFFE. Para esses casos de código legado recomenda-se a implementação de um *sidecar*: um módulo implantado próximo à aplicação com a responsabilidade de monitorar a API de Carga de Trabalho para solicitar um SVID e configurá-lo de modo que a aplicação consiga usá-lo de forma transparente. Na seção 3.3.4 foi explicado como utilizar o Ghostunnel como *sidecar*, mas para apresentar outra alternativa, utilizamos o *SPIFFE Helper* neste estudo de caso. O *SPIFFE Helper* é, portanto, uma aplicação *sidecar* que basicamente obtém SVIDs e os disponibiliza para a aplicação legada (setas 2 e 3 da Figura 3.11), atualizando-os quando necessário. O código-fonte do *SPIFFE Helper* é aberto[21]. Como o *SPIFFE Helper* foi pensado para facilitar a integração do SPIFFE com código legado, sua exe-

[21] https://github.com/spiffe/spiffe-helper

cução é simples: `spiffe-helper -config <arquivo_de_configuracao>`. O arquivo de configurações, apresentado no Código 3.9, contém uma série de valores associados à localização e nomes de chaves e certificados, além de um *script* que é executado sempre que o *SPIFFE Helper* obtém as informações do SPIRE. Para o estudo de caso apresentado, o *script* `helper_cmd.sh` (linha 2 do Código 3.9) codifica o certificado em um formato legível pelo Kafka, e notifica o Kafka para que ele atualize seus certificados.

```
1 agentAddress = "$AGENT\_SOCKET"
2 cmd = "/entrypoint/helper_cmd.sh"
3 certDir = "/store"
4 renewSignal = ""
5 svidFileName = "kafka-server-cert.pem"
6 svidKeyFileName = "kafka-server-key.pem"
7 svidBundleFileName = "ca-cert.pem"
8 addIntermediatesToBundle = false
```

Código 3.9. Arquivo de configuração do *SPIFFE Helper*.

Finalmente, o consumidor é um microsserviço confidencial, escrito em *Python*, que executa dentro de um enclave SGX através do arcabouço SCONE. O SPIRE não possuía nenhuma implementação de *plugin* para atestação de cargas de trabalho SGX. Algumas abordagens para atestação de cargas de trabalho SGX foram sugeridas recentemente por Silva et. al (2021), e reutilizamos uma das propostas no estudo de caso apresentado. Em linhas gerais, Silva et. al (2021) propõem que sempre que um Agente SPIRE receba um registro de entrada associado a uma carga de trabalho SCONE ele deve publicar os seletores, chaves e certificados em um serviço de atestação SCONE (seta 2 da Figura 3.11). O serviço de atestação SCONE possui alguns módulos responsáveis por realizar a atestação da carga de trabalho SCONE e apenas conceder acesso aos segredos (SVID e chaves) mediante sucesso da atestação (seta 3 da Figura 3.11).

Para facilitar a implantação e orquestração dos serviços usamos o Kubernetes. Por isso, tanto os serviços produtor como consumidor são atestados usando seletores do Kubernetes. Adicionalmente, o serviço consumidor também é atestado de acordo com seletores próprios de cargas de trabalho SCONE. As próximas seções detalham a atestação de aplicações CNC com Kubernetes e aplicações ConfCNC com SCONE.

3.5.1. Atestação de Cargas Nativas da Nuvem

O SPIRE possui *plugins* que o permitem uma integração com o orquestrador Kubernetes e a atribuição de identidades a partir de atributos desse orquestrador. Atualmente, quatro *plugins* compõem essa integração: dois *plugins* para atestação de nós, um *plugin* para atestação de cargas de trabalho, e um *plugin* para atualização do pacote de confiança.

Há duas formas de atestar agentes implantados em nós Kubernetes. A primeira abordagem, o *plugin k8s_sat*, atesta nós Kubernetes a partir das contas de serviço (*ServiceAccounts* providas pelo Kubernetes aos agentes. O segundo *plugin*, o *k8s_psat*, usa contas serviço projetadas. Para a atestação de cargas de trabalho no Kubernetes, o SPIRE dispõe de um *plugin* de atestação chamado *k8s*. O *plugin k8s* possui uma diversidade de seletores que podem ser utilizados no processo de atestação das cargas de trabalho. Tais

seletores são gerados a partir de consultas ao *kubelet*. A Tabela 3.6 apresenta os seletores e uma breve descrição de cada um. Por fim, o *plugin* para atualização do pacote de confiança (*k8sbundle*) é responsável por atualizar um *ConfigMap* no Kubernetes sempre que o pacote de confiança do servidor for rotacionado.

Seletor	Descrição
k8s:ns	*Namespace* da carga de trabalho
k8s:sa	Conta de serviço da carga de trabalho
k8s:container-image	Etiqueta da imagem de contêiner da carga de trabalho
k8s:container-name	Nome do contêiner da carga de trabalho
k8s:node-name	Nome do nó onde a carga de trabalho executa
k8s:pod-label	Nome de uma etiqueta atribuída ao Pod da carga de trabalho
k8s:pod-owner	Nome do dono do Pod da carga de trabalho
k8s:pod-owner-uid	UID do dono do Pod da carga de trabalho
k8s:pod-uid	UID do Pod da carga de trabalho
k8s:pod-name	Nome do Pod da carga de trabalho
k8s:pod-image	Etiqueta de qualquer imagem de contêiner no Pod da carga de trabalho
k8s:pod-image-count	Número de imagens de contêiner no Pod da carga de trabalho
k8s:pod-init-image	Etiqueta de qualquer imagem de contêiner de inicialização no Pod da carga de trabalho
k8s:pod-init-image-count	Número de imagens de contêineres de inicialização no Pod da carga de trabalho

Tabela 3.6. Seletores dos *plugins* Kubernetes de atestação de carga de trabalho.

Esses seletores podem ser combinados para definir um conjunto de características, dentro do contexto do Kubernetes, que uma carga de trabalho precisa ter para receber seu SVID.

3.5.1.1. Modelo de Ameaça do SPIFFE

O modelo de ameaça do SPIFFE é construído levando em conta três fronteiras de confiança: uma fronteira entre as cargas de trabalho e os agentes, outra fronteira entre os agentes e os servidores, e uma fronteira entre servidores de diferentes domínios de confiança.

A fronteira de confiança entre as cargas de trabalho e os agentes existe pois cargas de trabalho são consideradas, a princípio, não-confiáveis. Dessa maneira, as cargas de trabalho, consideradas como possivelmente comprometidas, só recebem uma identidade após o processo de atestação realizado pelo agente e seus *plugins*.

A fronteira de confiança entre os agentes e servidores existe por uma motivação similar. Na atestação de nó, agentes são considerados não confiáveis e possivelmente comprometidos até que o processo de atestação termine com sucesso. Em outras palavras,

antes da verificação das características de interesse de um nó, o agente não está apto para operar em pleno funcionamento, expondo a API de atestação de cargas de trabalho.

Já a fronteira entre servidores de diferentes domínios de confiança é importante para os cenários onde existe uma federação de servidores. Servidores devem ser capazes de distribuir os pacotes de confiança federados para que as cargas de trabalho consigam verificar identidades de outros domínios de confiança. Contudo, servidores de outros domínios não devem ser capazes de alterar ou gerar identidades pertencentes a outros domínios.

É importante observar que o modelo de ameaça do SPIFFE assume confiança na pilha de *software* e *hardware* utilizada pelos *plugins* (de atestação de cargas de trabalho ou de nós) para a geração dos seletores SPIFFE. Isso significa que diferentes combinações de *plugins* podem gerar diferentes superfícies de ataque.

3.5.2. Atestação de Cargas Confidenciais Nativas da Nuvem

Devido ao modelo de ameaça de aplicações ConfCNC, os *plugins* de atestação de cargas de trabalho disponíveis não são adequados para aplicações SCONE. Uma nova alternativa, que inicialmente foi projetada para cenários onde não é possível implantar agentes SPIRE, como é o caso de aplicações *serverless*, foi utilizada para entrega de identidades para tais aplicações confidenciais. Essa nova abordagem consiste em um novo tipo de *plugin* para agentes SPIRE chamados de *StoreSVID*.

O funcionamento geral de um *plugin StoreSVID* é simples. Quando uma nova identidade é atualizada no agente SPIRE, se essa identidade está marcada como responsabilidade de um *plugin* do tipo *StoreSVID*, esse *plugin* é invocado. Todos os artefatos relacionados à identidade são repassados para o *plugin* que tem a responsabilidade de codificar os artefatos em um formato apropriado e enviá-los para um armazenador seguro, que será a ponta final para entrega da identidade às cargas de trabalho.

Nesse contexto, o Laboratório de Sistemas Distribuídos da UFCG desenvolveu um novo *plugin StoreSVID*, especializado em entrega de identidades para aplicações SCONE, utilizando o SCONE CAS. Dessa maneira, o trabalho de realizar a atestação remota das cargas de trabalho confidenciais é delegado a um CAS. A autorização de acesso às identidades funciona com base em dois seletores SPIFFE: `CAS_SESSION_NAME` e `CAS_SESSION_HASH`. Uma vez que uma carga de trabalho SCONE é atestada, se ela tiver cadastrada na sessão cujo nome e cujo o *hash* correspondam aos seletores, ela receberá a identidade SPIFFE registrada via API de registro do SPIRE.

O fluxo de trabalho do plugin *StoreSVID* para aplicações SCONE é mostrado na Figura 3.12.

No passo 1, o operador registra a carga de trabalho SCONE no CAS, passando as configurações iniciais e parâmetros para a atestação da carga de trabalho no momento da implantação através de uma sessão SCONE. Esse primeiro passo retorna um *hash* para a sessão configurada no CAS. Com esse *hash* em mãos, o operador pode registrar uma entrada para a carga de trabalho no servidor SPIRE, usando os seletores `CAS_SESSION_NAME` e `CAS_SESSION_HASH` (passo 2). Uma vez que uma entrada é registrada no servidor SPIRE, o agente responsável recupera a identidade (passo 3) e

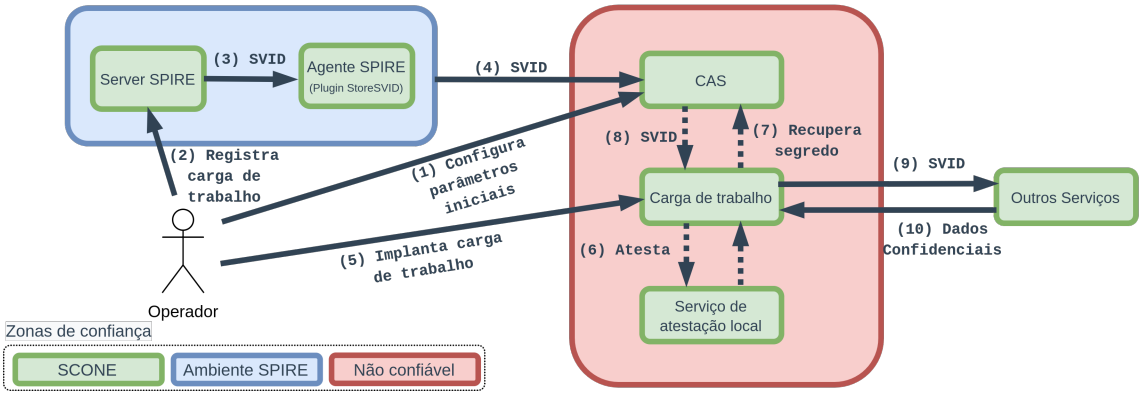

Figura 3.12. Fluxo de trabalho do plugin de atestação SCONE (StoreSVID).

pode então armazená-la no CAS, executando o passo 4.

Após o operador disparar a implantação da carga de trabalho, no passo 5, a carga de trabalho inicia um processo de atestação junto ao serviço de atestação local (passo 6). De posse de um *quote* verificável (como discutido na seção 3.4), a carga de trabalho finaliza o processo de atestação junto ao CAS, no passo 7. No passo 8, mediante a um processo de atestação bem sucedido, o CAS retorna um SVID que corresponde àquela sessão SCONE. A carga pode então utilizar a identidade recebida para se comunicar com outros serviços, receber e enviar dados confidenciais, como ilustrado nos passos 9 e 10.

A próxima seção apresenta os comandos para reproduzir o estudo de caso.

3.5.3. Execução do Estudo de Caso

Para a execução do estudo de caso utilizamos um *cluster* Kubernetes. Informações sobre instalação e configuração do Kubernetes para testes estão disponíveis no repositório de código do minicurso[22].

Os recursos completos, como arquivos de implantação e configuração estão disponíveis no repositório do minicurso, no diretório `demo-produtor-consumidor`. Lá também são encontrados todos os passos para levantar a infraestrutura, incluindo todos os componentes necessários. A seguir são apresentados os passos para registrar e implantar ambos produtor e consumidor.

```
1 git clone https://github.com/ufcg-lsd/minicurso-sbseg-2021
2 cd demo-produtor-consumidor
```

Código 3.10. Baixando o repositório.

Antes de implantar o produtor, é necessário registrar a identidade junto ao servidor SPIRE.

```
1 export AGENTE_ID=spiffe://lsd.ufcg.edu.br/agente-k8s
2 bin/spire-server entry create \
3     -parentID $AGENTE_ID \
4     -spiffeID spiffe://lsd.ufcg.edu.br/produtor \
```

[22] https://github.com/ufcg-lsd/minicurso-sbseg-2021

```
5       -selector k8s:container-name:produtor-kafka
```

Código 3.11. Registrando a identidade do produtor.

Com a identidade registrada, a implantação do produtor é feita utilizando a ferramenta *kubectl*.

```
1 kubectl apply -f produtor/deployment.yaml
2 # verificar o estado da aplicação
3 kubectl get pods
```

Código 3.12. Utilizando kubectl para implantar o produtor.

O comando `kubectl logs <nome-do-pod>` pode ser utilizado para verificar a saída da aplicação.

Para começar o processo de registro do consumidor SCONE é necessário postar a seção SCONE e obter o *hash* correspondente. O registro da seção pode ser feito com o uso do aplicativo de linha do comando do SCONE, o `scone-cli`. Para ter acesso à `scone-cli` basta usar a imagem Docker disponível no repositório do minicurso, como detalhado na seção 3.4. O comando a seguir cria uma seção SCONE a partir do arquivo `session.yaml`.

```
1 # Seção disponível no repositório em consumidor/scone-session.yaml
2 scone session create scone-session.yaml
```

Código 3.13. Utilizando a scone-cli para criar uma seção SCONE.

Após criar uma seção, a `scone-cli` retorna o *hash* dessa seção que pode ser usado para registrar uma identidade no servidor SPIRE. O exemplo a seguir ilustra a criação da identidade do consumidor que executa em um enclave com a ajuda do ambiente SCONE.

```
1  # Registrar entrada SPIRE com um hash e nome de seção
2  export SESSION_HASH=a91ed304958530306f0cab3a2977cbd84e139352ed3c
3                      d2002b6145ee4c4d722f
4  export SESSION_NAME=svid-session
5  export AGENTE_ID=spiffe://lsd.ufcg.edu.br/agente-k8s
6
7  ./bin/spire-server entry create -parentID $AGENTE_ID \
8          -spiffeID spiffe://lsd.ufcg.edu.br/consumidor \
9          -selector svidstore:type:scone_cas_secretsmanager \
10         -selector cas_session_hash:$SESSION_HASH \
11         -selector cas_session_name:$SESSION_NAME
```

Código 3.14. Criação de uma entrada no servidor SPIRE para o consumidor SCONE.

Após criar a entrada, a implantação do consumidor utilizando o comando `kubectl` é similar à implantação do produtor.

```
1 kubectl apply -f consumidor/deployment.yaml
2 # verificar o estado da aplicação
3 kubectl get pods
```

Código 3.15. Utilizando kubectl para implantar o consumidor.

A próxima seção encerra o trabalho apresentando os principais desafios e direções de pesquisa.

3.6. Desafios e Direções de Pesquisa

O padrão SPIFFE juntamente com a ferramenta SPIRE permite que aplicações CNC e ConfCNC se autentiquem de forma transparente, o que facilita a implementação do modelo confiança-zero. Para isto, o SPIFFE resolveu o problema gerado pelo processo de autenticação, que consiste em armazenar informações sensíveis, como um par identificador-senha, substituindo essa abordagem por mecanismos de atestação de nó e carga de trabalho. Com relação à atestação de nós, a maioria dos *plugins* se baseia em evidências que conseguem identificar o nó mas não são robustas para assegurar a integridade do nó, que por sua vez é o ambiente de execução do agente e cargas de trabalho. Além de prover as informações necessárias para emissão do SVID, a atestação poderia ser usada para verificar a integridade e segurança do ambiente de execução. Portanto, futuras pesquisas poderiam envolver a implementação de novos *plugins* de atestação usando diferentes tecnologias, especialmente tecnologias de computação confiável, pois são capazes de coletar informações suficientes para determinar a integridade de um ambiente de execução.

Silva et. al (2021) e Tassyani et. al (2021) conceberam recentemente *plugins* de atestação de carga de trabalho usando tecnologias de computação confiável. Silva et. al (2021) usaram o Intel SGX e arcabouço SCONE para emitir SVIDs para cargas de trabalho com MRENCLAVE específico. Tassyani et. al (2021) criaram um *plugin* de carga de trabalho que usa o chip *Trusted Platform Module* (TPM) como raiz de confiança para medir todos os executáveis e arquivos de configuração importantes carregados no sistema operacional e nos contêineres. As mesmas abordagens também poderiam gerar versões semelhantes dos *plugins* para atestação de nó, possivelmente com pouco esforço. É interessante perceber que identidades geradas a partir de tecnologias de computação confiável carregam consigo uma identificação única da aplicação, que poderia ser utilizada em auditorias. Portanto, estratégia semelhante poderia ser usada para rastrear os dados que nós e cargas de trabalho com identidades dessa natureza consomem, para fins de transparência no uso de dados e aderência à Lei Geral de Proteção de Dados.

Um aspecto do SPIRE que precisa ser aprimorado é a segurança relacionada a diferentes registros de entrada associados ao mesmo SPIFFE ID. Se o administrador (pessoa ou *software*) encarregado por cadastrar esses registros de entrada no servidor for malicioso ele poderá cadastrar um ou mais registros de entrada com o mesmo SPIFFE ID com o objetivo de obter um SVID com um SPIFFE ID específico. Mesmo que originalmente um determinado SVID com SPIFFE ID tivesse associado a um *plugin* que considera seletores rigorosos do ponto de vista de segurança, como por exemplo o *plugin* SCONE usado nesse trabalho, se outro registro de entrada for associado a um *plugin* que entrega SVID para cargas de trabalho atestadas de forma insuficiente, então a mesma identidade também poderia ser emitida para esta última carga de trabalho, mesmo que ela não esteja executando dentro de enclaves SGX. Atualmente, a solução é limitar o acesso à API de registro de identidades. Mas esse é um problema que permite soluções variadas, como permitir que o conjunto de registros de entradas de um servidor SPIRE não contenha duplicatas de SPIFFE ID, ou então cunhar nos SVIDs além do SPIFFE ID o seletor de forma a refletir o *plugin* empregado para coleta de informações no nó ou carga de trabalho.

Outro desafio consiste em portar aplicações legadas para autenticar-se com identidades SPIFFE. A organização e equipe técnica envolvidas precisam ter domínio do padrão SPIFFE e sobretudo do modelo de ameaça SPIRE, de modo que saibam quais são os componentes críticos e possam implantá-los tomando os devidos cuidados. Do ponto de vista técnico, considerando cargas de trabalho que ainda não foram adaptadas para usar a API de Carga de Trabalho do SPIRE, uma alternativa é usar *sidecars* que realizem esta tarefa. Ferramentas como *Ghostunnel* e *Envoy* podem ser usadas para autenticar SVIDs e estabelecer comunicação segura, facilitando portanto a integração de cargas de trabalho já existentes.

Agradecimentos

Este trabalho foi financiado através do projeto ZTPO, uma colaboração entre a Hewlett Packard Enterprise (Brasil), com recursos da Lei de Informática (Lei 8.248 de 23/10/1991), e a unidade CEEI-EMBRAPII na Universidade Federal de Campina Grande.

Referências

[Arnautov et al. 2016] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., and Fetzer, C. (2016). SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA. USENIX Association.

[Bellare et al. 1996] Bellare, M., Canetti, R., and Krawczyk, H. (1996). Keying hash functions for message authentication. pages 1–15. Springer-Verlag.

[Brito et al. 2020] Brito, A., Souza, C., Silva, F., Cavalcante, L., and Silva, M. (2020). Processamento confidencial de dados de sensores na nuvem. *Minicursos do XX SBSEG*.

[Costan and Devadas 2016] Costan, V. and Devadas, S. (2016). Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118.

[Cramer et al. 2015] Cramer, R., Damgård, I. B., et al. (2015). *Secure multiparty computation*. Cambridge University Press.

[Diffie and Hellman 1976] Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654.

[Dobbelaere and Esmaili 2017] Dobbelaere, P. and Esmaili, K. S. (2017). Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, DEBS '17, page 227–238, New York, NY, USA. Association for Computing Machinery.

[Dragoni et al. 2017] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). *Microservices: Yesterday, Today, and Tomorrow*, pages 195–216. Springer International Publishing, Cham.

[Eugster et al. 2003] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.

[Feldman et al. 2020] Feldman, D., Fox, E., Gilman, E., Haken, I., Kautz, F., Khan, U., Lambrecht, M., Lum, B., Fayó, A. M., Nesterov, E., Vega, A., and Wardrop, M. (2020). *Solving the Bottom Turtle: a SPIFFE way to establish trust in your infrastructure via universal identity*.

[Gentry 2009] Gentry, C. (2009). *A fully homomorphic encryption scheme*. Stanford university.

[Hayward and Chiang 2015] Hayward, R. and Chiang, C.-C. (2015). Parallelizing fully homomorphic encryption for a cloud environment. *Journal of applied research and technology*, 13(2):245–252.

[Hoffman 2016] Hoffman, K. (2016). *Beyond the Twelve-factor App: Exploring the DNA of Highly Scalable, Resilient Cloud Applications*. O'Reilly Media.

[Khan 2017] Khan, A. (2017). Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Computing*, 4(5):42–48.

[Leiserson 2018] Leiserson, A. (2018). Side channels and runtime encryption solutions with intel® sgx. Whitepaper. Acesso: 27/07/2021.

[Mell and Grance 2011] Mell, P. and Grance, T. (2011). The nist definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD.

[Naehrig et al. 2011] Naehrig, M., Lauter, K., and Vaikuntanathan, V. (2011). Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, page 113–124, New York, NY, USA. Association for Computing Machinery.

[Rose et al. 2020] Rose, S., Borchert, O., Mitchell, S., and Connelly, S. (2020). Zero trust architecture.

[Sampaio et al. 2019] Sampaio, L., Souza, C., Vinha, G., and Brito, A. (2019). Asperathos: Running qos-aware sensitive batch applications with intel sgx. In *Anais Estendidos do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 89–96, Porto Alegre, RS, Brasil. SBC.

[Severinsen 2017] Severinsen, K. M. (2017). Secure programming with intel sgx and novel applications. Master's thesis.

[Silva et al. 2021] Silva, M. S. L. d. S., Brito, A. E. M., and Brasileiro, F. (2021). Integrating spiffe and scone to enable universal identity support for confidential workloads. Master's thesis.

[Tassyany et al. 2021] Tassyany, M., Sarmento, R., Falcão, E., Gomes, R., and Brito, A. (2021). Um mecanismo de aprovisionamento de identidades para microsserviços baseado na integridade do ambiente de execução. In *Anais do XXXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 714–727, Porto Alegre, RS, Brasil. SBC.

[Tsai et al. 2017] Tsai, C., Porter, D. E., and Vij, M. (2017). Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA. USENIX Association.

[Verma et al. 2015] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. (2015). Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France.

[Wiggins 2017] Wiggins, A. (2017). The twelve-factor app. `https://12factor.net/`. Acesso: 27/07/2021.