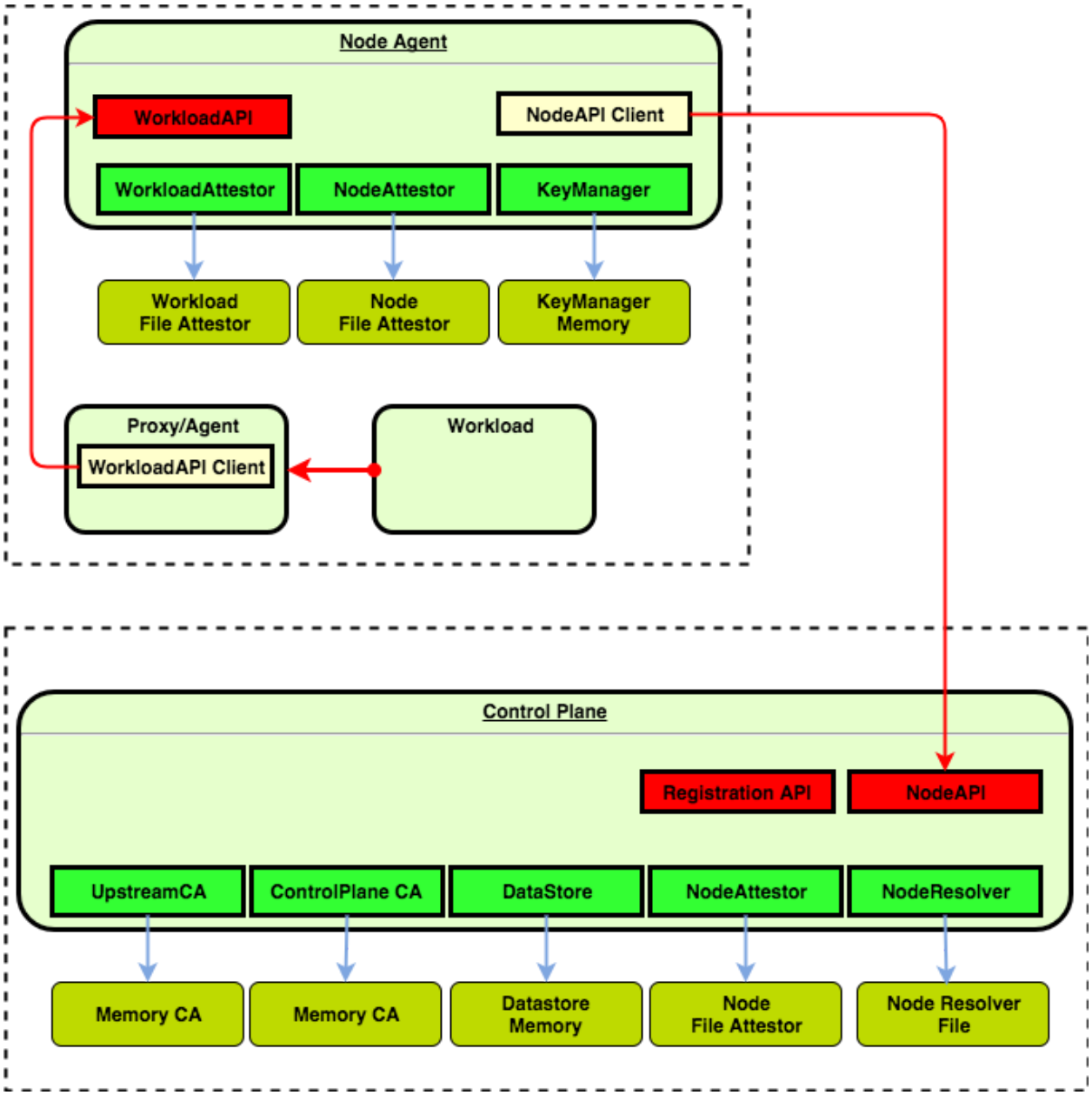
SPIFFE on Kubernetes

Last edit: May 10th, 2018

Authors: Enrico Schiattarella, Vipin Jain, *<put your name here if you add content>*

Introduction

This document describes SPIRE (SPIFFE Runtime Environment) design for Kubernetes environment. SPIRE Architecture document identifies a number of plugins, as indicated in the diagram below, that allows for integration with a variety of target environment, such as Kubernetes. The goal of this document is to be prescriptive, identifying specific integration points for providing identities to applications within Kubernetes environment.

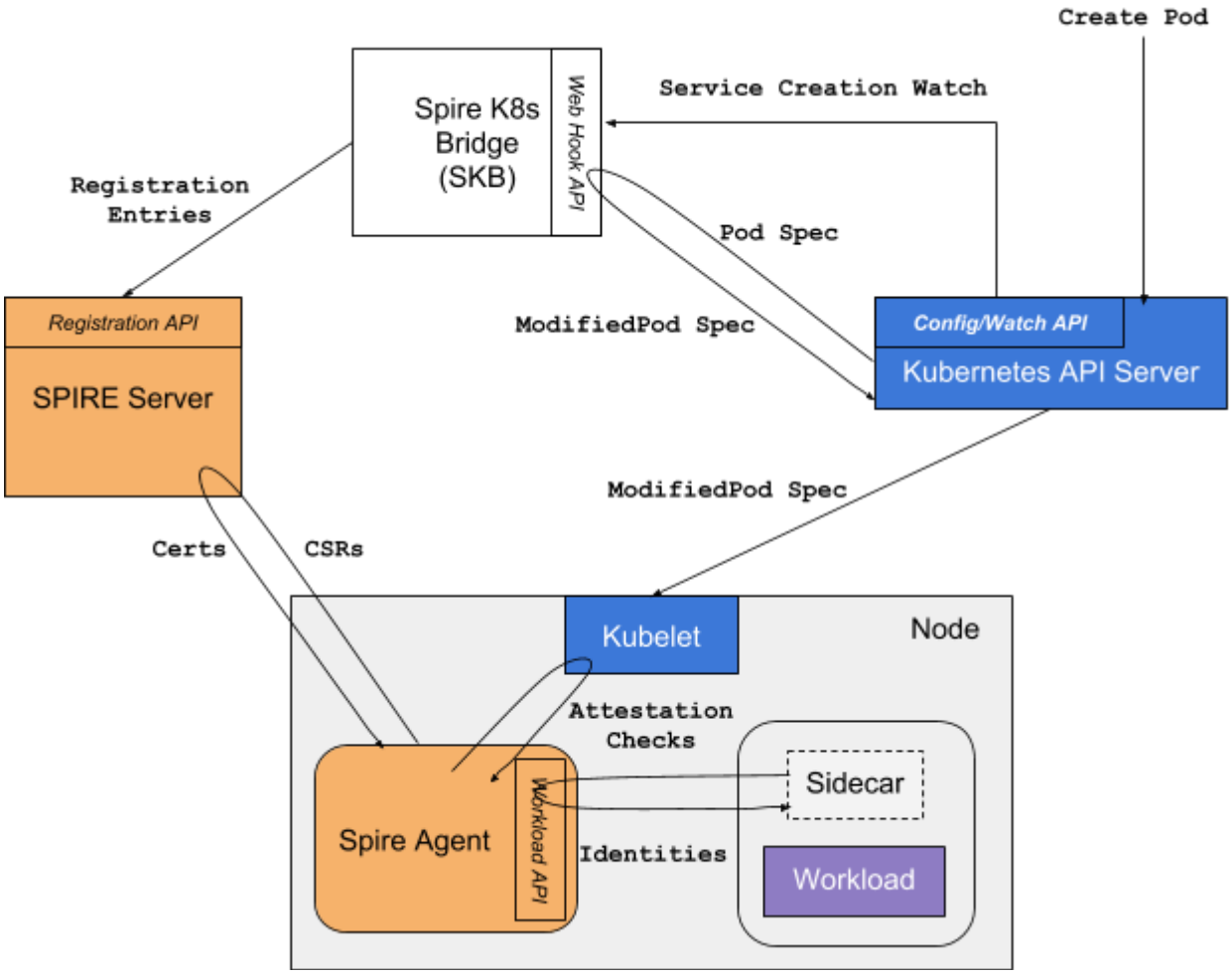


The design elements in this document are a result of discussions in ongoing meetings within SPIFFE-Kubernetes working group - details of the meeting notes and recordings can be found here.

The design and work discussed here is also relevant to the activities of the Kubernetes Container Identity Working Group with a goal of providing identities to workloads using various mechanisms, including using SPIFFE standard.

Architectural Components

Following diagram describes the integration points and role of various functional blocks.



Above design proposes a new module called Spire K8s Bridge, SKB (or better name TBD), that is responsible for interfacing between Kubernetes API Server and SPIRE Control Plane.

The integration with Kubernetes in achieved for following functions:

- **Create Registration Entries:** Creation of registration entries is done based on
  - Interception of pod creation to mount a unix-domain-socket, and optionally insert tls-proxy side-car
  - Notifications from API Server about service and service-account creation in order to create registration entries
- **Update Pod Spec**: Pod Spec is modified using the webhook APIs exposed by API Server to
  - Mount a unix-domain-socket to be used by workload API
  - Insert tls-proxy sidecar container/process
- **Pod/Workload Attestation**: Kubelet API is used to get the pod details. This can be the API to query pod's pid, or a formal way as defined in #60661 to validate the pid of the caller process.

SPIRE Kubernetes Bridge authenticates with SPIRE Server using the prescribed method as defined in SPIRE Architecture.

Node Attestation

Introduction

Node attestation is the process of bootstrapping trust between the SPIRE server and a SPIRE agent running on a node. The agent is pre-provisioned with the public key of the server, so the focus is on how the agent can prove its identity to the server. The agent can do so in multiple ways, for example using a join token that is shared manually with the server or a platform-specific mechanism like AWS Instance Identity Documents or GCP Instance Identity.
In this document we propose a Kubernetes-native way of achieving attestation, which provides both benefits automation and platform independence. It allows the Kubernetes administrator to bring new cluster nodes online and have them automatically recognized by the SPIRE server without involving the SPIRE server administrator.
The solution is based on the mechanism that has been developed to facilitate authentication of Kubelets to the ApiServer [ref]. This mechanism is fairly new and so it still has shortcomings. However, it is under active development, so it is expected to improve over time in terms of functionality and security.

Kubernete Cluster Setup (one-time)

The Kubernetes cluster administrator creates a user, clusterrole and clusterrolebinding for the SPIRE agent.

The user consists of a key, certificate pair signed by Kubernetes CA.

```
openssl genrsa -out spire-agent.key 2048

openssl req -new -key spire-agent.key -out spire-agent.csr -subj
"/CN=spire-agent/O=spire"

openssl x509 -req -in spire-agent.csr -CA CA_LOCATION/ca.crt -CAkey
CA_LOCATION/ca.key -CAcreateserial -out spire-agent.crt -days 500

kubectl config set-credentials spire-agent
--client-certificate=./spire-agent.crt  --client-key=./spire-agent.key

kubectl config set-context spire-agent-context --cluster=minikube
--user=spire-agent
```

The clusterrole and clusterrolebinding give privileges to this user to access the Certificates API only.

File csr-role.yml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: csr-create-get
rules:
- apiGroups:
  - certificates.k8s.io
  resources:
  - certificatesigningrequests
  verbs:
  - create
  - get


kubectl create -f csr-role.yml

kubectl create clusterrolebinding node-client-auto-approve-csr
--clusterrole=system:certificates.k8s.io:certificatesigningrequests:nodeclient
--user=spire-agent
```

User can only create and read CSRs, it cannot perform any other cluster operation.

New Agent Deployment

Agent is pre-provisioned with the following:

- address:port of the Kubernetes APIServer
- key for user spire-agent
- address:port of the SPIRE server
- public key of the SPIRE server

The agent can, but it does not have to, run as a Kubernetes DaemonSet. The only requirement is connectivity to Kubernetes ApiServer in addition to SPIRE server.

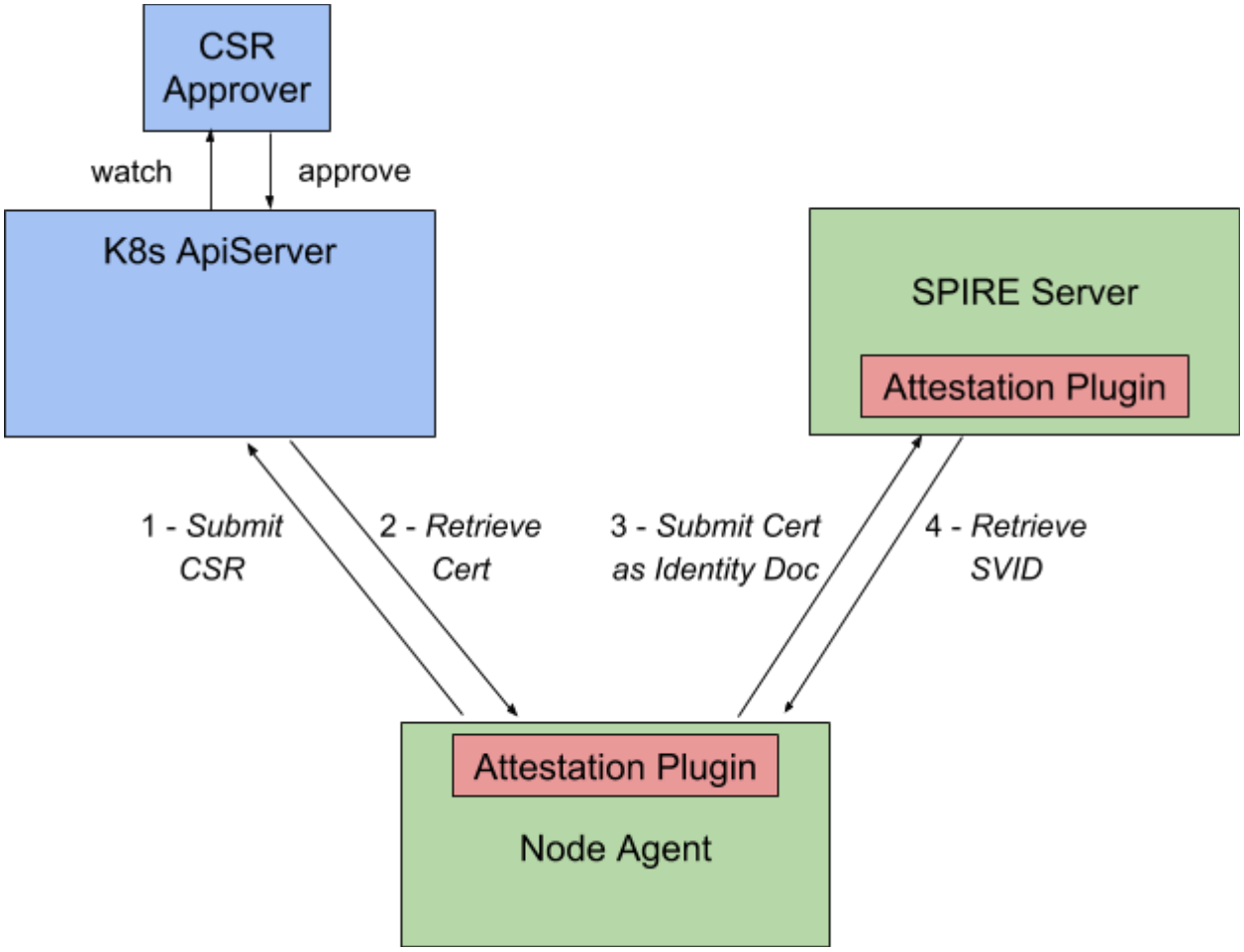
Attestation steps:

1. Agent creates a CSR and submits it to the K8s ApiServer

2. If the CSR conforms to the “Kubelet TLS Bootstrap spec” Kube-controller-manager will auto-approve it and generate the certificate. If not, Kubernetes admin need to approve it with “`kubectl certificate approve <csr-name>`”.
   Conforming CSRs have:
   Subject: `O="system:nodes" CN="system:node: <agent-pod-or-ip>"`
   KeyUsage: {`"digital signature", "key encipherment", "client auth"`}

3. The Agent retrieves the certificate issued by the Kubernetes API Server and sends it over to the SPIRE server

4. The SPIRE server verifies the certificate, start trusting the agent and provides it its own SVID



Security considerations:

1. This mechanism is not able to distinguish between the SPIRE agent performing this steps and other software entity. It is assumed that the cluster administrator has provided keys for the spire-agent user only to SPIRE agents.

2. The embedded CSR approver currently does not perform any check on the content of the CSR. As long as the user that submitted has the privileges and the CSR is conforming, it will be signed.
   There are various ways and plans to enhance this:
   - use an external approver with more sophisticated policies. An example is here. Another proposal which would leverage TPM is here. Custom approver logic can be embedded in SKBridge, so no code changes are required on Kubernetes side
   - perform extra checks when the agent talks to the server. A SPIRE Server plugin could check the IP address of the peer and challenge it to prove ownership of the private key before issuing the SVIDs. This would allow pinning the agent to a specific node, preventing an agent from impersonating another one. This is also being discussed in the context of Kubelet TLS bootstrap [ref], so if it gets implemented we can leverage it.

3. Automatic signing can be disabled by deleting the clusterrolebinding above. In this case the human operator that signs the CSR can inspect it to verify the accuracy of the agent host. This provides pretty strong guarantee, but does not allow autoscaling the cluster.

4. Step 3 is subject to replay attacks, so the certificate should be submitted over TLS (server-side only)

5. This attestation mechanism is somewhat similar to the one used on AWS. However, there are 2 important differences:
   1. In AWS case there is no guarantee about the identity of the process, as any process can request an IID, so server essentially does trust-on-first-use (TOFU); in K8s case only agent can request the CSR because it has spire-agent user credentials (unless Kubernetes admin grants the same privilege to other users)
   2. In AWS there is guarantee of node identity, because it is in the IID, whereas in K8s case the agent needs to be trusted

Workload API from Pod

Sidecar injection

Workloads that are not SPIFFE-aware require the use of a helper process to retrieve their identities and possibly to initiate/terminate TLS connections. This functionality can be performed by a process running into a *sidecar* container. The sidecar container is automatically deployed in the workload pod by using Kubernetes Mutating Webhook Admission Controllers (available as beta feature in Kubernetes 1.9). Essentially, when the Kubernetes API Server receives a request to create a Pod object (either from user or from a controller), it takes the Pod spec, submits it to an external HTTP/HTTPs endpoint and gets back a modified Pod spec or an error. It then proceeds to persist the new spec and sends out the corresponding watch notifications. Once the Pod object is created, it is picked up by the scheduler which selects a node. The Kubelet sees that a new Pod has been assigned to it and tries to start it.
This mechanism can be used to inject both init containers and normal containers. Init containers are special containers that get executed before the other containers in the pod. Other containers in the pod do not start until init containers have terminated successfully. This is useful to execute one-time initialization functions that have to be complete before workloads can start. Normal containers injected using Webhook admission controllers, on the contrary, are long-running and no different from other containers described in the original Pod spec.
In our case the external HTTPs endpoint is implemented by the SK Bridge, which also uses the admission controller requests to inject registration entries in the SPIRE server, thus guaranteeing that by the time the Pod is running, the identities are available. The connection

between Kubernetes ApiServer and SK Bridge is authenticated by providing SK Bridge with a certificate issues by ApiServer.

UDS injection

The agent exposes the Workload API on a UDS socket that is automatically mounted into workload pods when they are deployed. A single socket is exposed by the agent and is available to the sidecar or the workload itself in a well-known location. As the socket is shared, the SPIRE Agent needs to authenticate incoming requests. It can do so by using kernel-level mechanisms to retrieve PID, network namespace, etc. and cross-check the information with Kubelet and possibly with the docker runtime.
The mounting of the UDS socket can also be done automatically and transparently using mutating webhook admission controllers. The SK Bridge endpoint injects a hostPath volume mount for the well-known UDS location in the Pod spec.

Kubernetes Webhook Admission Controller Configuration

The configuration consists of a single MutatingWebhookConfiguration object. The configuration can point to a URL or to a Kubernetes server and can include a namespace selector to limit the scope of the hook.
A minimal working configuration is (remove comments!):

```
kind: MutatingWebhookConfiguration
metadata:
  name: spire-sidecar-injector
webhooks:
  - name: sidecar-injector.spiffe.io
    clientConfig:
      url: "https://skbridge.local:9999/inject"
        caBundle: "..." // base64-enc. trust bundle to verify webhook
cert
    rules:
      - operations: [ "CREATE" ]
        apiGroups: [""]
        apiVersions: ["v1"]
        resources: ["pods"]
    failurePolicy: Fail // do not proceed if webhook is not available
```

See Appendix B for an example of a webhook request sent by ApiServer on Pod creation

This an example of the sections that SKBridge add to a PodSpec for sidecar and UDS injection:

```
containers:
```

```
  - name: spire-sidecar
    image: spire-sidecar:latest
    volumeMounts:
    - mountPath: /spire
      name: spire-wl-api
volumes:
  - name: spire-wl-api
    hostPath:
      path: /tmp/spire
      type: Directory
```

The spire agent creates the directory /tmp/spire and the Workload API server UDS as soon as it starts. When pods are created, the sidecar container is injected and the local node directory /tmp/spire under /spire. All parameters are constant (i.e. do not depend on the type or name of the workload or the way it is deployed: deployment, replication controller, daemonset, …).
The additions to the PodSpec are returned to the K8s ApiServer in the form of a RFC 6902 JSON patch.

SK Bridge configuration

The SK Bridge configuration consists of:

- Knobs that control the spec rewrite (on/off, UDS mount path, etc.)
- Knobs that control the formation of registration entries
- A key/cert pair accepted by K8s ApiServer
- SPIRE Server credentials

It can be suppliedd as:

- Config file
- Configmap (if running on K8s)
- Kubernetes 3rd party resource (if running on K8s)

Workload Attestation

Workload attestation is the process by which an agent authenticates a workload trying to access the workload API.
It consists of two steps:

1. Identifying the caller process OS-level metadata
2. Associating the process with a Kubernetes pod scheduled on the local node

Process Identification

A process that receives a connection request on a Unix Domain Socket can use the Go `GetsockoptUcred` function from package `syscall` to retrieve peer process credentials, which consist of PID, UID and GID. If the connection is received on a local TCP socket, the function `Getpeername` allows retrieval of the peer process identity in the form “IP:port”.
Once the PID of a process is known, it is possible to derive more information using procfs mounted under `/proc`.

Pod Identification

Information about a Pod and its containers can be obtained from Kubernetes, assuming that either a container name or IP address is known.
Kubernetes ApiServer knows about all pods on all nodes, but accessing the Pod list requires credentials.

The Kubelet exposes two network endpoints on a node:

1. An authenticated read/write REST API on port 10250
2. An unauthenticated, read-only REST API on port 10255

The simplest way for the agent to retrieve the list of Pods running on the node is to access the URL “http://localhost:10255/pods”. No authentication is required and there is no need to filter out pods running on other nodes.

Once the list of pods is retrieved, agent needs to identify the one associated with the workload. If the workload request is received on a local TCP socket, the agent can simply look for a pod that has the corresponding IP address in the `Status.podIP` field.

If the request was received on a UDS socket, only the process PID is available and the corresponding Pod cannot be identified without interrogating the Docker runtime and using procfs.

The lookup sequence for the UDS case is:

1. Get the list of all pods using the Docker SDK client
2. Check all containers in the list until you find one whose `State.PID` attribute matches workload PID or any of its ancestor PIDs. Ancestor PIDs are retrieved from procfs.
3. If a container is found, extract the container ID and go to step 4, otherwise stop
4. Retrieve the list of pods from the Kubelet, for each Pod check the list of containers in `Status.ContainerStatuses` until you find one whose `ContainerID` value matches the ID obtained from the Docker API
5. If a Pod is found the Agent can use pod metadata (labels, image ID, etc.) to authorize the request.

Since this sequence can be fairly expensive, Agent can build a cache and maintain it using the event notification mechanisms offered by Docker and Kubernetes APIs. There's a library by Boz that does this, but it talks to K8s ApiServer instead of Kubelet, so it needs ApiServer credentials. It has its own threading model and various things that we don't need, like a CLI. Eventually we may decide to keep our own custom implementation or augment Boz's library.

While this mechanism works, it has few limitations and shortcomings. Ideally we would like something that:

1. Does not have a dependency on the docker runtime (dep. on CRI probably would be ok)
2. is standardized, robust, and ideally works without a shared kernel between attestor and workload (to support kata containers and other VM-based isolation mechanisms)

Kubernetes container identity working group has been discussing this topic here but as of now there doesn't seem to be anything coming in the near future.

Automating Registration Entries

Registration entries in SPIRE represent a record of an entity and its associated properties being identified by SPIFFE ID. The API interface is defined in Spire architecture document defines four registration APIs:

- `Create(registeredEntry) registrationID`
- `Delete(registrationID) registeredEntry`
- `Fetch(registrationID) registeredEntry`
- `Update(registrationID, registeredEntry) registeredEntry`

```
Where registeredEntry is defined as
{
      []Selector{type, value},
      ParentID,
      SpiffeID,
      TTL,
      []FederatedBundleSpiffeID
}

registrationID string uniquely identifies a registration entry.
```

The SpiffeID chosen for Kubernetes workloads is of the form:

spiffe://<domain>/ns/<namespace>/sa/<serviceaccount>
spiffe://<domain>/ns/<namespace>/service/<service-name>

ParentID is spiffe ID of parent cluster if any is likely of the form:

spiffe://<*domain*>/k8s

TTL specifies the expiration for the entry, i.e. the entry unless updated/renewed is considered invalid. TBD: would this be a user input?

Selectors define the match criteria to select SPIFFE ID based on labels defined for the workloads. For K8s, these labels are provided during creation of registration entry based on the pod/service information obtained from Kubernetes API Server. A new selector entry is created by SK Bridge for every Pod that is created in K8s. Type in selector is set to *k8s.*

For example, a kubernetes service that selects the backend pods in the following definitions

Service

```
kind: Service
metadata:
  name: prod-frontend
spec:
  Selector:
    env: prod
    tier: frontend
  Ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

Pod

```
kind: Pod
metadata:
  name: fe-pod-23
  labels:
    env:prod
    tier: frontend
spec:
  containers
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

It would create a registration entry for the Service with following parameters in registeredEntry:

[]Selector{type, value}: []{k8s, env:prod}, {k8s, tier: frontend}
ParentID: nil (or spiffe://prod.xyz.com/k8s)
SpiffeID: spiffe://prod.xyz.com/ns/default/service/prod-frontend
TTL: 3200
[]FederatedBundleSpiffeID: nil

SK Bridge Failure Handling

SKB maintains the in-memory state w.r.t. Pod information and corresponding registered SPIFFE IDs. As a result it must handle following failure scenarios:

- SKB's disconnectivity to Kubernetes API Server: It is possible that some Pods/workloads get scheduled during the disconnectivity. In order to handle this, SKB would resync all Pod/Service information from API Server after the connection is re-establish, and submit new registration entries accordingly.

- SKB's disconnectivity with SPIRE Server: Upon such an event, SKB is expected to cache all POD entries received from API Server and re-push the registration entries as its connectivity is established again.
- SKB process crash: Upon such an event SKB fetches all registration entries from Spire Server and get all Pod/Service information from API Server, reconcile the differences and create/delete/update any registration entries as required. <TBD: Spire doesn't define ListAll APIs for registration entries>

Appendix A: Development Environment

TB Filled

Appendix B: Sample Webhook call from ApiServer


```
{
    "apiVersion": "admission.k8s.io/v1beta1",
    "kind": "AdmissionReview",
    "request": {
        "kind": {
            "group": "",
            "kind": "Pod",
            "version": "v1"
        },
        "namespace": "default",
        "object": {
            "metadata": {
                "creationTimestamp": null,
                "generateName": "nginx-deployment-64ff85b579-",
                "labels": {
                    "app": "nginx",
                    "pod-template-hash": "2099416135"
                },
                "ownerReferences": [
                    {
                        "apiVersion": "extensions/v1beta1",
                        "blockOwnerDeletion": true,
                        "controller": true,
                        "kind": "ReplicaSet",
                        "name": "nginx-deployment-64ff85b579",
                        "uid": "63e04274-3ea5-11e8-bf3c-080027b3e9f0"
                    }
                ]
            },
```

```
"spec": {
    "containers": [
        {
            "image": "nginx:latest",
            "imagePullPolicy": "Always",
            "name": "nginx",
            "ports": [
                {
                    "containerPort": 80,
                    "protocol": "TCP"
                }
            ],
            "resources": {},
            "terminationMessagePath": "/dev/termination-log",
            "terminationMessagePolicy": "File",
            "volumeMounts": [
               {
     "mountPath": "/var/run/secrets/kubernetes.io/serviceaccount",
                    "name": "default-token-zrrkt",
                    "readOnly": true
                }
            ]
        }
    ],
    "dnsPolicy": "ClusterFirst",
    "restartPolicy": "Always",
    "schedulerName": "default-scheduler",
    "securityContext": {},
    "serviceAccount": "default",
    "serviceAccountName": "default",
    "terminationGracePeriodSeconds": 30,
    "tolerations": [
        {
            "effect": "NoExecute",
            "key": "node.kubernetes.io/not-ready",
            "operator": "Exists",
            "tolerationSeconds": 300
        },
        {
            "effect": "NoExecute",
            "key": "node.kubernetes.io/unreachable",
            "operator": "Exists",
            "tolerationSeconds": 300
        }
    ],
    "volumes": [
        {
            "name": "default-token-zrrkt",
```

```
                        "secret": {
                            "secretName": "default-token-zrrkt"
                        }
                    }
                ]
            },
            "status": {}
        },
        "oldObject": null,
        "operation": "CREATE",
        "resource": {
            "group": "",
            "resource": "pods",
            "version": "v1"
        },
        "uid": "2a916b18-3eb4-11e8-bf3c-080027b3e9f0",
        "userInfo": {
            "groups": [
                "system:serviceaccounts",
                "system:serviceaccounts:kube-system",
                "system:authenticated"
            ],
            "uid": "e062443f-3de2-11e8-a7eb-080027b3e9f0",
         "username": "system:serviceaccount:kube-system:replicaset-controller"
        }
    }
}
```


Appendix C: Sample Container Metadata from Docker Runtime


```
{
    "Id": "8c2c3fe46e0a90cb912c18ec490c2bd433752321e278eb255ab2961abceb8ba4",
    "Created": "2018-04-17T23:42:01.863177412Z",
    "Path": "nginx",
    "Args": [
      "-g",
      "daemon off;"
    ],
    "State": {
      "Status": "running",
      "Running": true,
      "Paused": false,
      "Restarting": false,
```

```
      "OOMKilled": false,
      "Dead": false,
      "Pid": 14925,
      "ExitCode": 0,
      "Error": "",
      "StartedAt": "2018-04-17T23:42:02.010608383Z",
      "FinishedAt": "0001-01-01T00:00:00Z"
    },
    "Image":
"sha256:b175e7467d666648e836f666d762be92a56938efe16c874a73bab31be5f99a3b",
    "ResolvConfPath":
"/var/lib/docker/containers/603689188dd28889705d0d25be3be11634d59fb283b1df2185
04aabf4415653c/resolv.conf",
    "HostnamePath":
"/var/lib/docker/containers/603689188dd28889705d0d25be3be11634d59fb283b1df2185
04aabf4415653c/hostname",
    "HostsPath":
"/var/lib/kubelet/pods/ebf54896-4298-11e8-bf3c-080027b3e9f0/etc-hosts",
    "LogPath":
"/var/lib/docker/containers/8c2c3fe46e0a90cb912c18ec490c2bd433752321e278eb255a
b2961abceb8ba4/8c2c3fe46e0a90cb912c18ec490c2bd433752321e278eb255ab2961abceb8ba
4-json.log",
    "Name":
"/k8s_spire-sidecar_kuard-b75468d67-hnvk2_default_ebf54896-4298-11e8-bf3c-0800
27b3e9f0_0",
    "RestartCount": 0,
    "Driver": "overlay2",
    "Platform": "linux",
    "MountLabel": "",
    "ProcessLabel": "",
    "AppArmorProfile": "",
    "ExecIDs": [
      "08728a090bb5ec2030a59f5f0e263d7ec6805d453a04fd0905fe3e67695377ae"
    ],
    "HostConfig": {
      "Binds": [
        "/tmp/spire:/spire:rslave",

"/var/lib/kubelet/pods/ebf54896-4298-11e8-bf3c-080027b3e9f0/etc-hosts:/etc/hos
ts",

"/var/lib/kubelet/pods/ebf54896-4298-11e8-bf3c-080027b3e9f0/containers/spire-s
idecar/a038f9f3:/dev/termination-log"
      ],
      "ContainerIDFile": "",
      "LogConfig": {
        "Type": "json-file",
        "Config": {}
```

```
      },
      "NetworkMode":
"container:603689188dd28889705d0d25be3be11634d59fb283b1df218504aabf4415653c",
      "PortBindings": null,
      "RestartPolicy": {
        "Name": "",
        "MaximumRetryCount": 0
      },
      "AutoRemove": false,
      "VolumeDriver": "",
      "VolumesFrom": null,
      "CapAdd": null,
      "CapDrop": null,
      "Dns": null,
      "DnsOptions": null,
      "DnsSearch": null,
      "ExtraHosts": null,
      "GroupAdd": null,
      "IpcMode":
"container:603689188dd28889705d0d25be3be11634d59fb283b1df218504aabf4415653c",
      "Cgroup": "",
      "Links": null,
      "OomScoreAdj": 1000,
      "PidMode": "",
      "Privileged": false,
      "PublishAllPorts": false,
      "ReadonlyRootfs": false,
      "SecurityOpt": [
        "seccomp=unconfined"
      ],
      "UTSMode": "",
      "UsernsMode": "",
      "ShmSize": 67108864,
      "Runtime": "runc",
      "ConsoleSize": [
        0,
        0
      ],
      "Isolation": "",
      "CpuShares": 2,
      "Memory": 0,
      "NanoCpus": 0,
      "CgroupParent":
"/kubepods/besteffort/podebf54896-4298-11e8-bf3c-080027b3e9f0",
      "BlkioWeight": 0,
      "BlkioWeightDevice": null,
      "BlkioDeviceReadBps": null,
      "BlkioDeviceWriteBps": null,
```

```
      "BlkioDeviceReadIOps": null,
      "BlkioDeviceWriteIOps": null,
      "CpuPeriod": 0,
      "CpuQuota": 0,
      "CpuRealtimePeriod": 0,
      "CpuRealtimeRuntime": 0,
      "CpusetCpus": "",
      "CpusetMems": "",
      "Devices": [],
      "DeviceCgroupRules": null,
      "DiskQuota": 0,
      "KernelMemory": 0,
      "MemoryReservation": 0,
      "MemorySwap": 0,
      "MemorySwappiness": null,
      "OomKillDisable": false,
      "PidsLimit": 0,
      "Ulimits": null,
      "CpuCount": 0,
      "CpuPercent": 0,
      "IOMaximumIOps": 0,
      "IOMaximumBandwidth": 0
    },
    "GraphDriver": {
      "Data": {
        "LowerDir":
"/var/lib/docker/overlay2/ce90e898041a0ae39243117c408553fca1223de5cf1528299db6
5150600cc888-init/diff:/var/lib/docker/overlay2/6edacee19082bcdd036a61399508fe
216a28959284423d82c2ae2a146bc9f14e/diff:/var/lib/docker/overlay2/2e397dc7c3e82
9448e9126a174f00
6772e2302e6926932fde70f141d85882a97/diff:/var/lib/docker/overlay2/3157f32386d9
bb200bbf62eb8bba0173129ac1a68c0daf731ee46474343a9ceb/diff",
        "MergedDir":
"/var/lib/docker/overlay2/ce90e898041a0ae39243117c408553fca1223de5cf1528299db6
5150600cc888/merged",
        "UpperDir":
"/var/lib/docker/overlay2/ce90e898041a0ae39243117c408553fca1223de5cf1528299db6
5150600cc888/diff",
        "WorkDir":
"/var/lib/docker/overlay2/ce90e898041a0ae39243117c408553fca1223de5cf1528299db6
5150600cc888/work"
      },
      "Name": "overlay2"
    },
    "Mounts": [
      {
        "Type": "bind",
```

```
        "Source":
"/var/lib/kubelet/pods/ebf54896-4298-11e8-bf3c-080027b3e9f0/etc-hosts",
        "Destination": "/etc/hosts",
        "Mode": "",
        "RW": true,
        "Propagation": "rprivate"
      },
      {
        "Type": "bind",
        "Source":
"/var/lib/kubelet/pods/ebf54896-4298-11e8-bf3c-080027b3e9f0/containers/spire-s
idecar/a038f9f3",
        "Destination": "/dev/termination-log",
        "Mode": "",
        "RW": true,
        "Propagation": "rprivate"
      },
      {
        "Type": "bind",
        "Source": "/tmp/spire",
        "Destination": "/spire",
        "Mode": "rslave",
        "RW": true,
        "Propagation": "rslave"
      }
    ],
    "Config": {
      "Hostname": "kuard-b75468d67-hnvk2",
      "Domainname": "",
      "User": "0",
      "AttachStdin": false,
      "AttachStdout": false,
      "AttachStderr": false,
      "ExposedPorts": {
        "80/tcp": {}
      },
      "Tty": false,
      "OpenStdin": false,
      "StdinOnce": false,
      "Env": [
        "NGINX_PORT_80_TCP_ADDR=10.100.25.154",
        "KUBERNETES_SERVICE_HOST=10.96.0.1",
        "KUBERNETES_PORT=tcp://10.96.0.1:443",
        "KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1",
        "NGINX_SERVICE_PORT=80",
        "NGINX_PORT_80_TCP_PROTO=tcp",
        "KUBERNETES_SERVICE_PORT_HTTPS=443",
        "NGINX_PORT=tcp://10.100.25.154:80",
```

```
        "NGINX_PORT_80_TCP=tcp://10.100.25.154:80",
        "KUBERNETES_PORT_443_TCP_PROTO=tcp",
        "NGINX_SERVICE_HOST=10.100.25.154",
        "KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443",
        "KUBERNETES_PORT_443_TCP_PORT=443",
        "NGINX_PORT_80_TCP_PORT=80",
        "KUBERNETES_SERVICE_PORT=443",
        "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
        "NGINX_VERSION=1.13.12-1~stretch",
        "NJS_VERSION=1.13.12.0.2.0-1~stretch"
      ],
      "Cmd": [
        "nginx",
        "-g",
        "daemon off;"
      ],
      "Healthcheck": {
        "Test": [
          "NONE"
        ]
      },
      "ArgsEscaped": true,
      "Image":
"nginx@sha256:18156dcd747677b03968621b2729d46021ce83a5bc15118e5bcced925fb4ebb9
",
      "Volumes": null,
      "WorkingDir": "",
      "Entrypoint": null,
      "OnBuild": null,
      "Labels": {
        "annotation.io.kubernetes.container.hash": "60a483c4",
        "annotation.io.kubernetes.container.restartCount": "0",
        "annotation.io.kubernetes.container.terminationMessagePath":
"/dev/termination-log",
        "annotation.io.kubernetes.container.terminationMessagePolicy": "File",
        "annotation.io.kubernetes.pod.terminationGracePeriod": "30",
        "io.kubernetes.container.logpath":
"/var/log/pods/ebf54896-4298-11e8-bf3c-080027b3e9f0/spire-sidecar/0.log",
        "io.kubernetes.container.name": "spire-sidecar",
        "io.kubernetes.docker.type": "container",
        "io.kubernetes.pod.name": "kuard-b75468d67-hnvk2",
        "io.kubernetes.pod.namespace": "default",
        "io.kubernetes.pod.uid": "ebf54896-4298-11e8-bf3c-080027b3e9f0",
        "io.kubernetes.sandbox.id":
"603689188dd28889705d0d25be3be11634d59fb283b1df218504aabf4415653c",
        "maintainer": "NGINX Docker Maintainers
\u003cdocker-maint@nginx.com\u003e"
 },
```

```
      "StopSignal": "SIGTERM"
    },
    "NetworkSettings": {
      "Bridge": "",
      "SandboxID": "",
      "HairpinMode": false,
      "LinkLocalIPv6Address": "",
      "LinkLocalIPv6PrefixLen": 0,
      "Ports": {},
      "SandboxKey": "",
      "SecondaryIPAddresses": null,
      "SecondaryIPv6Addresses": null,
      "EndpointID": "",
      "Gateway": "",
      "GlobalIPv6Address": "",
      "GlobalIPv6PrefixLen": 0,
      "IPAddress": "",
      "IPPrefixLen": 0,
      "IPv6Gateway": "",
      "MacAddress": "",
      "Networks": {}
    }
}
```

Appendix E: Sample Kubelet Pod Description

```
{
            "metadata": {
                "annotations": {
                    "kubernetes.io/config.seen":
"2018-04-17T17:11:40.274703011Z",
                    "kubernetes.io/config.source": "api"
                },
                "creationTimestamp": "2018-04-17T17:11:40Z",
                "generateName": "kuard-b75468d67-",
                "labels": {
                    "pod-template-hash": "631024823",
                    "run": "kuard"
                },
                "name": "kuard-b75468d67-v4mgm",
                "namespace": "default",
                "ownerReferences": [
                    {
                        "apiVersion": "extensions/v1beta1",
```

```
                        "blockOwnerDeletion": true,
                        "controller": true,
                        "kind": "ReplicaSet",
                        "name": "kuard-b75468d67",
                        "uid": "64e42976-4262-11e8-bf3c-080027b3e9f0"
                    }
                ],
                "resourceVersion": "134977",
                "selfLink":
"/api/v1/namespaces/default/pods/kuard-b75468d67-v4mgm",
                "uid": "64f2af56-4262-11e8-bf3c-080027b3e9f0"
            },
            "spec": {
                "containers": [
                    {
                        "image": "gcr.io/kuar-demo/kuard-amd64:1",
                        "imagePullPolicy": "IfNotPresent",
                        "name": "kuard",
                        "resources": {},
                        "terminationMessagePath": "/dev/termination-log",
                        "terminationMessagePolicy": "File",
                        "volumeMounts": [
                            {
                                "mountPath":
"/var/run/secrets/kubernetes.io/serviceaccount",
                                "name": "default-token-zrrkt",
                                "readOnly": true
                            }
                        ]
                    },
                    {
                        "image": "nginx:latest",
                        "imagePullPolicy": "Always",
                        "name": "spire-sidecar",
                        "resources": {},
                        "terminationMessagePath": "/dev/termination-log",
                        "terminationMessagePolicy": "File",
                        "volumeMounts": [
                            {
                                "mountPath": "/spire",
                               "name": "spire-wl-api"
                            }
                        ]
                    }
                ],
                "dnsPolicy": "ClusterFirst",
                "nodeName": "minikube",
                "restartPolicy": "Always",
```

```
            "schedulerName": "default-scheduler",
            "securityContext": {},
            "serviceAccount": "default",
            "serviceAccountName": "default",
            "terminationGracePeriodSeconds": 30,
            "tolerations": [
                {
                    "effect": "NoExecute",
                    "key": "node.kubernetes.io/not-ready",
                    "operator": "Exists",
                    "tolerationSeconds": 300
                },
                {
                    "effect": "NoExecute",
                    "key": "node.kubernetes.io/unreachable",
                    "operator": "Exists",
                    "tolerationSeconds": 300
                }
            ],
            "volumes": [
                {
                    "name": "default-token-zrrkt",
                    "secret": {
                        "defaultMode": 420,
                        "secretName": "default-token-zrrkt"
                    }
                },
                {
                    "hostPath": {
                        "path": "/tmp/spire",
                        "type": "Directory"
                    },
                    "name": "spire-wl-api"
                }
            ]
        },
        "status": {
            "conditions": [
                {
                    "lastProbeTime": null,
                    "lastTransitionTime": "2018-04-17T17:11:40Z",
                    "status": "True",
                    "type": "Initialized"
                },
                {
                    "lastProbeTime": null,
    "lastTransitionTime": "2018-04-17T17:11:43Z",
                    "status": "True",
```

```
                        "type": "Ready"
                    },
                    {
                        "lastProbeTime": null,
                        "lastTransitionTime": "2018-04-17T17:11:40Z",
                        "status": "True",
                        "type": "PodScheduled"
                    }
                ],
                "containerStatuses": [
                    {
                        "containerID":
"docker://af9fd21d196fb2fb302995aa600edca8ba916297aafbbf049b537ec9b4594fa4",
                        "image": "gcr.io/kuar-demo/kuard-amd64:1",
                        "imageID":
"docker-pullable://gcr.io/kuar-demo/kuard-amd64@sha256:3e75660dfe00ba63d0e6b5d
b2985a7ed9c07c3e115faba291f899b05db0acd91",
                        "lastState": {},
                        "name": "kuard",
                        "ready": true,
                        "restartCount": 0,
                        "state": {
                            "running": {
                                "startedAt": "2018-04-17T17:11:40Z"
                            }
                        }
                    },
                    {
                        "containerID":
"docker://cd0104e9e5f9b7a2845e5629c86063b005708971769b5a944c9ed0f9f5abf746",
                        "image": "nginx:latest",
                        "imageID":
"docker-pullable://nginx@sha256:18156dcd747677b03968621b2729d46021ce83a5bc1511
8e5bcced925fb4ebb9",
                        "lastState": {},
                        "name": "spire-sidecar",
                        "ready": true,
                        "restartCount": 0,
                        "state": {
                            "running": {
                                "startedAt": "2018-04-17T17:11:42Z"
                            }
                        }
                    }
                ],
                "hostIP": "10.0.2.15",
                "phase": "Running",
                "podIP": "172.17.0.3",
```

```
        "qosClass": "BestEffort",
        "startTime": "2018-04-17T17:11:40Z"
    }
}
```