Perceptor Architecture and Development Guide

Introduction

Mission Statement: The goal of the Blackduck Perceptor project is to provide the building blocks for a composable, flexible, and highly customizable distributed system for continuous scanning of software threat sources. The canonical implementation of the OSE-Perceptor which is supported by blackduck, is OSEScanner 2.0. OSE-Perceptor can **perceive** threats running in cloud native orchestration environments based on images that are either running or building, and upon **perception** of said threats, use orchestration-specific metadata operations to instrument those objects with vulnerability data.

Perceptor: Perception, by definition, means, 'the result or product of perceiving'. The perceptor is at the heart of the architecture; its role is to solidify the events from the outside world by federating them with information from the blackduck hub. [1]

Perceivers: Perceivers are platform- or orchestration-specific entities which ultimately will result in the **'perception'** of a threat.

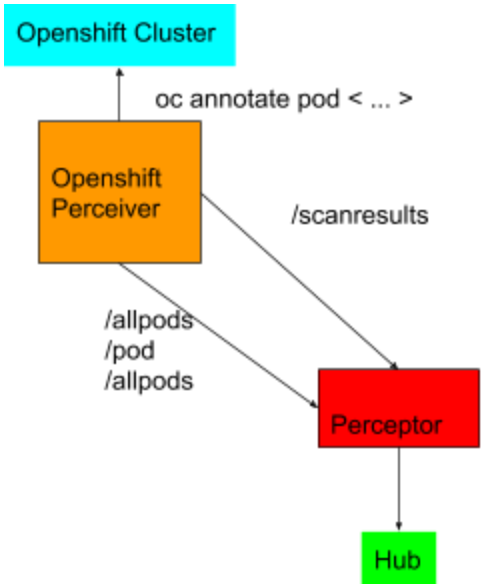

Figure 1: Perceivers are the precursor to the perception of a threat in a given cloud native platform. Once they begin the process of noticing an image, pod, or any other scannable artifact, the Perceptor starts to work on creating the final perception of the threat, based on the hub's scan results. Ultimately those scan results are provided back to the perceiver object, which in turn triggers another 'perception', of a threat on the orchestration engine itself.

[1] What's the difference ? Perceiving is a verb, it relates to an action which takes place. Perception is a noun ~ it refers to the state of something which has already occured. The ultimate perception of a threat is in the blackduck hub itself, which finds, catalogues, and collates vulnerability information.

Deployment

The architectural decisions are generally expected to hold up for clusters with anywhere from 50 to 200 pods per node, of an average size of 300 nodes. This corresponds to 15000 containers, which in a worst case scenario, could lead to roughly 50,000 scan requests.

It is expected that some clusters will exceed sizes of 1000 nodes[2], both on openshift as well as kubernetes. In such clusters, we might expect as many as 200,000 containers to be running.

Requirements

- The scanning throughput of the system must scale horizontally.
- All APIs for internal communication should be documented (and if possible, clients generated) via swagger docs.
- The ability to find new pods, images, or other source code threats must handle platform- or cloud provider-specific actions.
- API clients associated with cloud provider-specific actions should not restrict or limit one another: i.e. they need to be containerized in some fashion to avoid API incompatibilities (or, for example, in JVM languages, jar shading can be used)..
- Given that all major cloud native platforms support advanced metrics and introspective tooling for relatively low implementation cost, testing and verification should be completely automated and driven by scanning and annotation metrics.

Architecture

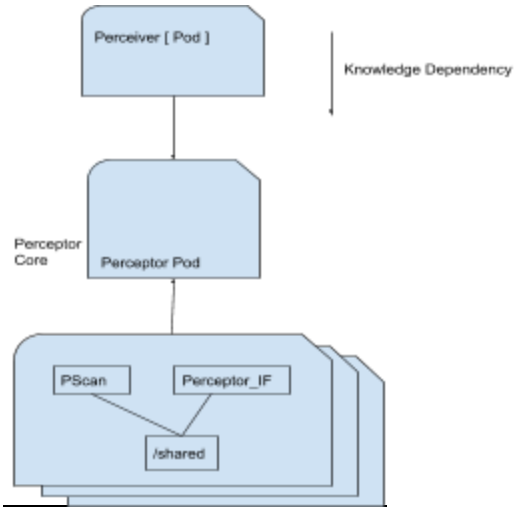

Figure 2: Technical architecture of Perceptor, Perceiver, PScan and Perceptor_IF data flow. Although there are many types of perceivers, there is only one perceiver to be considered per data center / orchestration. Similarly, we have one perceptor. The horizontal scaling component of the overall architecture is the Pod which comprises PScan and Perceptor_IF. Since the scan component only needs to naively poll Perceptor for work, it scales across a data center.

2 http://blog.kubernetes.io/2016/03/1000-nodes-and-beyond-updates-to-Kubernetes-performance-and-scalability-in-12.html

- **Perceptor (one type)** : The perceptor is a stateless cache to the blackduck hub, serves as a classical Controller [3], initiating and deciding what sort of requests should be made on behalf of perceivers. Initially designed not to scale, as a modern golang webserver can generally handle tens of thousands of requests per second.
- **Perceivers (many types) :** Perceivers are stateless interfaces to cloud orchestration frameworks, image repositories, or any other source of a threat. In a given deployment, a given Perceiver type may exist in every datacenter, or every orchestrator. Multiple perceivers can communicate threat investigations to the same perceptor ~ this is because the Perceptor has no knowledge of a perceivers, and transactions around threats are carried out using image IDs and SHA's as primary keys.
- **PScan (one type) :** If the Perceptor is the heart of OSE-perceptor, then we consider PScan to be the backbone. PScan polls perceptor for scanning workloads, and scans images (read further for image source information) that are surfaced from Perceptor. PScan is horizontally scalable (in some data centers, we suspect that it may *need* to scale to all nodes depending on how images are retrieved).
    - Note that we plan to **extend PScan to support the hub-docker-inspector** linux package scanning tool, which is a multi container deployment. T
      His would lead to a **second type** of PScan.
- **Perceptor_Image Facade (many):** The perceptor Image Facade is a container which has the ability to pull an image from a registry. We use the word 'pull' because, in fact, it is a facade, which can be pluggable. The contract between the image facade and the PScan service is that the image facade is capable of **ensuring** images are available in a shared volume mount, without necessarily **providing** those images[4].
- **Perceptor-up-e2e.sh:** Perceptor-up-e2e is a shell script which deploys all the perceptor components in their standard supported OSEScanner 2.0 configuration, implementing an end to end, completely automated, metrics-driven test which saturates a cluster with a corpus of images, and measures the time taken to discover, scan, and annotate all said images / pods.

[3] https://en.wikipedia.org/wiki/Model–view–controller

[4] This is what allows us to give any customer a mechanism to run OSEPerceptor without giving it privileged access to the docker socket, or docker access of any other type. It also allows customers to implement a push model for providing images to PScan, so they can build their own distributed scanning systems.

Component details

Perceptor

The APIs objects from perceptor are served to perceivers, and for golang applications, it can be vendored as a perceiver dependency.

Perceptor: *handles events from several different sources:*

- Perceivers
- Scanners
- Hub
- Regularly scheduled timers within perceptor

Perceptor: *receives information about pods and images* in the system from perceivers.

- Expects to be notified whenever there's a pod event -- whether a new pod is added, an existing pod is updated, or a pod is deleted from the system
- Expects to be notified by perceiver, and makes a REST API available for this purpose.
- Expects to be notified of image events, if the platform supports that.

Perceptor: *stores image scan information in the hub.*

- Requires an image as a (primary key) sha to identify a scan.
- Users the sha, to access vulnerability and policy violation information in the hub, and can make that available to perceivers ~ via scan-id.

Perceptor: *maintains a queue of images to be scanned*.

- Expects scanners to poll it for work from this queue.
- Expects that the scanner informs perceptor upon completion of the scan.

Perceptor: *manages a model which describes its state*.

- Can be reconstructed using the scan information in the hub.
- Reconstructed the pod/image coming in from perceivers. Each event is processed serially to update the model. Metrics are produced based on events coming in, the state of the model, and events being pushed out.

Perceptor: *publishes metrics for all essential cluster input and scanning operations.*

- Currently, Perceptor publishes metrics to a time series format ~ prometheus, which is based on the openTSDB data model [5] .

[5] http://opentsdb.net

Perceiver

We define an “perceivable-image-repo [PIR]” as a platform which, in some manner, either runs, stores, or curates images in some way. Examples of PIRs include: A hosted container registry with public APIs for querying newly added images, An openshift cluster (which has an emedded registry, as well as mechanisms to run artifacts from said registry), or a kubernetes cluster (which is capble of running images, as Pods).

Perceivers live in a separate repository, each with unique vendored API dependencies which allow them to talk to orchestration or image APIs. This is part of the original motivation from separating them.

Perceiver: *watches* events on a orchestration or image repository.

- Expects the PIR to provide an API that tells perceiver of new artifacts that may be running.
- Expects the PIR to support summarizing and exposing its entire state to perceiver at a given time.
- Currently expects, although this isnt a hard requirement, that the PIR suppots a watch semantic in its API[6].

PScan

Perceptor scan is part of perceptor core, has no significant external dependencies. Ultimately PScan is a container scan with an additional component: a *polling* mechanism. The polling mechanism is responsible for receiving workloads from Perceiver itself, and then leveraging API functionality of the later defined Perceptor Image_Facade , which is co-located, to acquire images that need to be scanned.

PScan: *is colocated with an image facade.*

- Expects the image facade to be able to deliver image contents to a PScan readable directory.
- Expects the underlying scan machinery to be able to trigger scans against the blackduck hub [note this could be pluggable at some point].
- Polls perceptor for work, and implements that work.

Perceptor Image_Facade

Perceptor image facade is also part of perceptor core, has no significant external dependencies. The canonical implementation of the image facade, is that it pulls docker images as tarballs into a shared directory that PScan can then read. However, the image facade is easily swapped out with a custom implementation. For example, one may write an image facde

[6] https://coreos.com/etcd/docs/latest/learning/why.html

Perceptor-up-e2e.sh

Perceptor-up-e2e.sh will live in an [as of yet] repository that is external to the perceptors and the perceivers. Presumably ose-scanner will be the integration point.

Future Ideas

- Analytics Dashboard [Prometheus SQL gateway]
- HA configuration [perceptor state serialization]
- Avoiding hub DOS [redis/memcached intermediary]
- Package scanning [hub docker inspector]
- Grafana dashboards exported for customer scan health dashboard.