Using OPA to Implement Security Policies on Thunder ADC

By John D. Allen
Global Solutions Architect – Cloud, IoT, & Automation
March, 2022

Table of Contents

Introduction

Open Policy Agent, or OPA, is a Cloud-Native CNCF project to provide a common source of policy for Cloud-based solutions. While designed more for User Authentication within Cloud applications, its design lends itself to many different ways to implement Policy and to be used by many different solutions.

This document will cover using OPA to provide Security Policies to a Thunder node through the use of a Thunder Cloud Agent (TCA) container. This TCA will query the OPA node on a periodic basis and update a defined Thunder ADC or CFW node with these policies. Using this method, any number of Thunder nodes can be updated at the same time by just making one policy change on the OPA node.

To implement this, we created a Proof-of-Concept Container that looks for a Connection-Rate-Limit Policy on OPA and creates a Virtual-Server Template and applies it to an SLB Virtual-Server.

How we did it

For our POC, we used a OPA Container running on a Kubernetes Cluster, and loaded in our Policy Files. We then wrote an a10-thunder-opa-policy Container that calls the OPA Container every minute or so and looks for these policies and updates a specified Thunder ADC node with the values found on the OPA Container.

OPA Configuration

For our POC, we used the instructions to deploy an OPA Container from the OPA website: https://www.openpolicyagent.org/docs/latest/deployments/#kubernetes We also hard-wired a NodePort of 30181 inside the service-opa.yaml file.

Deploy the OPA Container

First, we created an example policy, following the instructions on the OPA website:

example.rego

```
package example

greeting = msg {
  info := opa.runtime()
  hostname := info.env["HOSTNAME"]
  msg := sprintf("Hello from pod %q!", [hostname])
}
```

And created a Kubernetes ConfigMap from this:

```
kubectl create configmap example-policy --from-file example.rego
```

Next, we create a ‘deployment-opa.yaml’ file like so:

```
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
  name: opa
  namespace: policy
  labels:
    app: opa
spec:
  replicas: 1
  selector:
    matchLabels:
      app: opa
  template:
    metadata:
      labels:
        app: opa
      name: opa
    spec:
      containers:
      - name: opa
        image: openpolicyagent/opa:0.34.0
        ports:
        - name: http
          containerPort: 8181
        args:
        - "run"
        - "--ignore=.*"  # exclude hidden dirs created by Kubernetes
        - "--server"
        - "/policies"
        volumeMounts:
        - readOnly: true
          mountPath: /policies
          name: example-policy
      volumes:
      - name: example-policy
        configMap:
          name: example-policy
```

And then deploy it:

```
kubectl create -f deployment-opa.yaml
```

Then we create a Service manifest file:

```
kind: Service
apiVersion: v1
metadata:
  name: opa
  namespace: policy
  labels:
    app: opa
spec:
  type: NodePort
  selector:
    app: opa
  ports:
    - name: http
      protocol: TCP
      port: 8181
      targetPort: 8181
      nodePort: 30181
```

And deploy it so we can access the OPA Container from outside Kubernetes, if we want to:

```
kubectl create -f service-opa.yaml
```

Now we can do a quick check to see if it is up and running:

```
# curl http://localhost:30181/v1/data/example?pretty=true
{
  "result": {
    "greeting": "Hello from pod \"opa-fcbd7b8b8-qq2hv\"!"
  }
}
```

Of course 'localhost' will need to be either the FQDN or an IP address of your Kubernetes Cluster…I just happened to run the command on the Master node of my Kubernetes Cluster.

Add our Network Policies to the OPA Container

For our POC, we are going to load in two different Network Policies: Bandwidth Limits, and Connection Rate Limits. Our TCA currently only implements the Connection Rate Limits, but the TCA code does have some hooks already in it for the Bandwidth Limits policy…I just ran out of time to complete the implementation ;)

Since we are not using any security on our POC, we can just directly load up our policies into the OPA Container.

First, we need to setup our policy files:

bw-plan.json

```
{
  "red": [ "0" ],
  "yellow": [ "1" ],
  "orange": [ "10" ],
  "green": [ "100" ]
}
```

bw-nodes.json

```
{
  "thunder-1": [ "orange" ],
  "thunder-2": [ "green" ]
}
```

cps-plan.json

```
{
  "red": [ "0" ],
  "yellow": [ "10" ],
  "orange": [ "100" ],
  "blue": [ "1000" ],
  "green": [ "10000" ]
}
```

cps-nodes.json

```
{
  "thunder-1": [ "orange" ],
  "thunder-2": [ "blue" ],
  "tester1": [ "yellow" ]
}
```

So for each “Policy”, we have two files; the ‘-plan’ file has the specific rates for the policies, and a plan name associated with it. The second file is the Node-IDs and the Plan that is assigned to them.

So for example, our TCA for the ‘thunder-1’ node will first query OPA for what plan name is assigned to it – for CPS, its “orange”, then it will query OPA for the CPS “orange” plan, which is a rate of “100”….so “thunder-1” will have a conn-rate-limit of 100.

Then, we need to define our actual OPA Policies that will make use of this Data:

net-policy.rego

```
package net

import data.net.bw
import data.net.cps
import data.net.bwnodes
import data.net.cpsnodes
import input

bwrate = msg {
  plan := data.net.bwnodes[input.node][0]
  msg := data.net.bw[plan][0]
}

cpsrate = msg {
  plan := data.net.cpsnodes[input.node][0]
  msg := data.net.cps[plan][0]
}
```

We will be placing all our Data and Policy under the ‘net’ leaf in OPA. So when we query for these policies, we will start our query strings with ‘net.’

Once we define these files, we then load them up into OPA. For the POC, we are just using this simple script:

```
#
# Data
curl -s -X PUT http://localhost:30181/v1/data/net/bw --data-binary @bw-plan.json
curl -s -X PUT http://localhost:30181/v1/data/net/cps --data-binary @cps-plan.json
curl -s -X PUT http://localhost:30181/v1/data/net/bwnodes --data-binary @bw-nodes.json
curl -s -X PUT http://localhost:30181/v1/data/net/cpsnodes --data-binary @cps-nodes.json

#
# Policy
curl -s -X PUT http://localhost:30181/v1/policies/net --data-binary @net-policy.rego |
jq .
```

Loading in the net-policy.rego file will produce some JSON output, so that is why we have the " | jq ." on the end of the curl command where we load it in. If all works, it will generate a blank JSON structure…if there are errors, they will print out in a more readable form:

```
# ./LOAD_OPA.sh
{}
```

We can now check to make sure it is all loaded in:

```
# curl -s http://localhost:30181/v1/data?pretty=true
{
  "result": {
    "example": {
      "greeting": "Hello from pod \"opa-fcbd7b8b8-qq2hv\"!"
    },
    "net": {
      "bw": {
        "green": [
          "100"
        ],
        "orange": [
          "10"
        ],
        "red": [
          "0"
        ],
        "yellow": [
          "1"
        ]
      },
      "bwnodes": {
        "thunder-1": [
          "orange"
        ],
        "thunder-2": [
          "green"
        ]
      },
      "cps": {
        "blue": [
          "1000"
        ],
        "green": [
          "10000"
        ],
        "orange": [
          "100"
        ],
        "red": [
          "0"
        ],
        "yellow": [
          "10"
        ]
      },
      "cpsnodes": {
        "tester1": [
          "yellow"
        ],
```

```
        "thunder-1": [
          "orange"
        ],
        "thunder-2": [
          "blue"
        ]
      }
    }
  }
}
```

You should get an output that looks like the above text.

OPA Proxy Configuration

Our TCA can run stand-alone if you want (for things like testing and tweaking the configuration), or run as a Container or Pod on a Kubernetes Cluster, which is what we will be documenting here.

First, you will need to create the a10-opa-proxy Container and place it somewhere it can be accessed from your K8s Cluster. My Dockerfile looks like this:

```
FROM golang:1.17.3 AS builder

RUN mkdir /app
RUN mkdir /app/axapi
RUN mkdir /app/config
ADD ./axapi /app/axapi
ADD go.* /app
ADD opaproxy.go /app
ADD ./config/config.yaml /app/config
ADD Dockerfile /app
ADD NOTES.md /app

WORKDIR /app
RUN CGO_ENABLED=0 GOOS=linux go build -o /app/opaproxy opaproxy.go

##
## Build Final Container
FROM alpine:latest AS production
COPY --from=builder /app .

CMD ["./opaproxy"]
```

Note that there is the *axapi* Golang module that is required to talk to the Thunder node. This is NOT an A10 Networks written module, but one that I wrote. This Module is maintained in a separate Github repo.

You will then need to create a ConfigMap for the configuration file, along with a Deployment manifest. Here is the file I created:

a10-opa-proxy.yaml

```
---
apiVersion: v1
```

```
kind: ConfigMap
metadata:
  name: a10-opa-policy-config
  namespace: policy
data:
  config.yaml: |+
    ## Debug level 0 -> 10
    debug: 10
    ## OPA Node info
    OPA_IP: 10.1.1.220
    OPA_PORT: 30181
    ## Thunder ADC Node info
    THND_IP: 10.1.1.33
    THND_PORT: 443
    THND_USER: admin
    THND_PASSWD: a10
    THND_ID: thunder-1
    # yaml array, but Unmarshalled as JSON...yes, it works in Golang :)
    vs: [
      {"name": "ws-vip", "policy": "bw"},
      {"name": "ws-vip", "policy": "cps"}
    ]
    # CHECK_INTERVAL is in seconds.
    CHECK_INTERVAL: 120
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: a10-opa-proxy
  namespace: policy
  labels:
    app: a10-opa-proxy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: a10-opa-proxy
  template:
    metadata:
      labels:
        app: a10-opa-proxy
      name: a10-opa-proxy
    spec:
      containers:
      - name: a10-opa-proxy
        image: 10.1.1.30:5000/a10-opa-proxy:latest
        volumeMounts:
          - name: a10-opa-policy-config
            mountPath: /config/config.yaml
            subPath: config.yaml
            readOnly: true
      volumes:
      - name: a10-opa-policy-config
        configMap:
          name: a10-opa-policy-config
          items:
          - key: config.yaml
            path: config.yaml
```

The *config.yaml* file is created as a K8s ConfigMap, and then mapped to the a10-opa-proxy Pod. Most of the items in the config.yaml file should be fairly obvious as to what they do and what they should be set to. The Deployment is rather simple, and only requires the config file volume to be mapped to the Pod.

Deploying the OPA Proxy

Once this is set, you can deploy the Pod:

```
# kubectl create -f a10-opa-proxy.yaml
configmap/a10-opa-policy-config created
deployment.apps/a10-opa-proxy created
```

Assuming everything was setup correctly, you should see the Pod start and be Running:

```
# kubectl get pods -A
NAMESPACE     NAME                                       READY   STATUS    RESTARTS
AGE
kube-system   hostpath-provisioner-5c65fbdb4f-zcdzj      1/1     Running   38
369d
kube-system   kube-multus-ds-amd64-ckbh7                 1/1     Running   34
369d
kube-system   calico-kube-controllers-847c8c99d-rzv5w    1/1     Running   41
369d
policy        opa-fcbd7b8b8-qq2hv                        1/1     Running   2
130d
kube-system   coredns-86f78bb79c-c6z29                   1/1     Running   34
369d
kube-system   calico-node-rf9vh                          1/1     Running   0
107d
kube-system   kube-multus-ds-amd64-z9vvc                 1/1     Running   23
303d
kube-system   calico-node-npvhd                          1/1     Running   2
107d
cyan          webserver-6c48bb8c8b-kdzf8                 1/1     Running   0
8m28s
cyan          webserver-6c48bb8c8b-bqwwj                 1/1     Running   0
8m28s
cyan          webserver-6c48bb8c8b-c5qld                 1/1     Running   0
8m28s
default       thunder-kubernetes-connector-69b7547bd4-7dbm7   1/1     Running   0
8m20s
policy        a10-opa-proxy-6fd79cd4d9-x8q4r             1/1     Running   0
8s
```

For my demo, I put the a10-opa-proxy in the same namespace where I put the OPA Pod…you can run it from anywhere – even another K8s Cluster or Docker server – as long as it can connect to both the OPA and the Thunder node.

Checking the logs will show quite a bit of detail if you have the debug level set to 10:

```
# kubectl logs a10-opa-proxy-6fd79cd4d9-x8q4r -n policy
time="2022-03-25 17:15:15" level=info msg="A10 Thunder OPA Proxy Starting..."
```

```
debug: 10
opaip: 10.1.1.220
opaport: 30181
thunderip: 10.1.1.33
thunderport: 443
thunderid: thunder-1
time="2022-03-25 17:15:15" level=info msg="Connected to Thunder Device"
time="2022-03-25 17:15:15" level=info msg="Network Policies found on OPA Server"
policy = orange
rate = 10
time="2022-03-25 17:15:15" level=info msg="Updating BW Policy Template"
>>>{"server": {"name": "opa-policy-bw", "bw-rate-limit": 10, "bw-rate-limit-resume": 8,
"bw-rate-limit-duration": 20} }
policy = orange
rate = 100
time="2022-03-25 17:15:15" level=info msg="Updating CPS Policy Template"
>>>{"virtual-server": {"name": "opa-policy-cps", "conn-limit": 100, "conn-rate-limit":
100} }
policy = orange
rate = 10
>>>{"server": {"name": "opa-policy-bw", "bw-rate-limit": 10, "bw-rate-limit-resume": 8,
"bw-rate-limit-duration": 20} }
time="2022-03-25 17:17:16" level=info msg="Updating BW Policy Template"
policy = orange
rate = 100
>>>{"virtual-server": {"name": "opa-policy-cps", "conn-limit": 100, "conn-rate-limit":
100} }
time="2022-03-25 17:17:16" level=info msg="Updating CPS Policy Template"
policy = orange
rate = 10
>>>{"server": {"name": "opa-policy-bw", "bw-rate-limit": 10, "bw-rate-limit-resume": 8,
"bw-rate-limit-duration": 20} }
time="2022-03-25 17:19:16" level=info msg="Updating BW Policy Template"
policy = orange
rate = 100
time="2022-03-25 17:19:16" level=info msg="Updating CPS Policy Template"
>>>{"virtual-server": {"name": "opa-policy-cps", "conn-limit": 100, "conn-rate-limit":
100} }
```

This shows what was retrieved from OPA. You can then go check your Thunder node configuration for the correct Templates:

```
slb template server opa-policy-bw
  bw-rate-limit 10 resume 8 duration 20
!
slb server 10.1.1.220 10.1.1.220
  port 30265 tcp
!
slb server 10.1.1.221 10.1.1.221
  port 30265 tcp
!
slb service-group ws-sg tcp
  health-check ws-mon
  member 10.1.1.220 30265
  member 10.1.1.221 30265
!
slb template virtual-server opa-policy-cps
  conn-limit 100
  conn-rate-limit 100
!
```

```
slb virtual-server ws-vip 10.1.1.44
  template virtual-server opa-policy-cps
  port 80 http
    aflex http-error-status-log
    source-nat auto
    service-group ws-sg
!
```

Notice that the a10-opa-proxy has created two Templates: *opa-policy-bw* and *opa-policy-cps.* For this Proof-of-Concept, we are only applying the *opa-policy-cps* policy to the SLB Virtual-Server that was defined in the *config.yaml* file. The *opa-policy-bw* Template would need to be applied at the SLB Server level.

Congratulations, you are now controlling a Security Policy on a Thunder node using OPA!

Removing the OPA Proxy

This is just deleting the two configuration items from the Kubernetes Cluster:

```
# kc delete -f a10-opa-proxy.yaml
configmap "a10-opa-policy-config" deleted
deployment.apps "a10-opa-proxy" deleted
```

In our Proof-of-Concept, we did not include any code to remove the Templates that were defined and attached when the Pod is stopped.

Appendix 1: Resources

a10-opa-proxy Github: https://github.com/jdallen-a10/a10-opa-proxy