Network filtering in a distributed app

Brian Cheung and Sam Wang

ABSTRACT

Container orchestration technologies like Kubernetes provide a simple way to deploy, scale, and manage containerized applications [4]. However, as with any production-level system, these distributed systems must be properly secured at each appropriate level. In this paper, we examine common vulnerabilities of distributed applications and demonstrate how we can help mitigate them through Pod Security Polices and network proxies. Pod Security Policies are policies that define rules/conditions that a pod must run with in order to function within the whole cluster of pods. Network Proxies behave as a gateway between some internet user and the network, like a middleman for some network. Proxies, like Envoy, can be configured to fortify an application's security by filtering out malicious requests and encrypting accepted requests. We first implement Pod Security Policies to improve the base security of the application itself by limiting the pod's capabilities and only allowing appropriate privileges. By defining the least amount of capabilities required to run the application, we reduce the attack surface of the application. Then we implement Envoy with Open Policy Agent to reinforce the network security by only accepting specific requests based on client privileges. This is achieved by filtering certain packets based on information such as the source/destination IP addresses and protocols, which further reduces the attack surface and strengthens current security practices. As demonstrated in our project, the additional layers of security can help mitigate common vulnerabilities.

1. INTRODUCTION

As container orchestration technologies and cloud platforms like Kubernetes and Docker Hub simplify the deployment and scaling process and increase the accessibility of open-sourced software, developers must be cautious of unintentional vulnerabilities when deploying large-scale distributed applications. Common vulnerabilities of these distributed applications can stem from misconfiguration and misuse of legitimate privileges, unknown vulnerabilities (often from open-source software), and an overall lack of security. These vulnerabilities may allow attackers to gain privileged control of the system. This issue presents the need for preventative measures that can be implemented with Kubernetes' Pod Security Policies.

Kubernetes provides a framework that allows developers to create Pod Security Policies (PSP) which defines the rules that pods in a cluster must follow in order to be allowed to operate. This ensures that pods can and will run only if it maintains the correct privileges and resources defined in the PSPs. When a pod is deployed, the PSP acts as a gatekeeper that will compare the pod security configuration to what is defined in the PSP. Some examples of how PSPs can limit pod behaviors include, preventing privilege escalation, restricting pods from accessing host namespaces, filesystem, and networks, restrict the amount of user/groups that a pod can run, and more.

Systems that involve networking may need additional layers of security in order to further reduce the attack surface. Common threats and vulnerabilities in networks can include denial of service attacks and snooping to extract confidential and private information [5]. These threats can be mitigated through network filtering, which is the practice of monitoring the inflow/outflow of packets in a network.

There are two types of filtering, ingress and egress filtering. Ingress filtering is the technique of monitoring incoming packet data. It is considered the first-line of defense in a network because it blocks out unwanted inflow traffic to the network. While this isn't a robust and complete form of defense, it's beneficial because it can greatly reduce the load on some proxy or firewall, and it's an effective for getting rid of the majority of unwanted traffic. Egress filtering is the technique of monitoring the flow of outbound network traffic and prevents any outbound connections to potential threats/unwanted hosts. Egress filtering can be used to disrupt malware, block unwanted services, and gives greater awareness of network traffic.

For this experiment, we implemented two separate distributed applications with Kubernetes to demonstrate the possible ways Pod Security Policies and network proxies can mitigate common vulnerabilities and threats. The first part demonstrates how Pod Security Policies can prevents privilege escalation on distributed applications deployed with Kubernetes. The second part demonstrates the effectiveness and use case of an open source network proxy, called Envoy, that is used alongside a Kubernetes deployment.

Through our paper, we will show the effectiveness of and

the need to implement robust Pod Security Policies as well as a Network Proxy to aide with network filtering, in order to improve the overall security of a distributed application.

2. MOTIVATION

It is important to continue iterating upon known threats and ways to improve security measures to better defend the inherent vulnerabilities within applications and cloud services and counter intentional malicious attacks. With the increasing popularity of distributed applications deployed with container orchestration technologies like Kubernetes, there comes a need to understand how to reinforce the security of these pods that applications will be deployed on. Additionally, we must consider additional layers of security, which depend on the type of application and its exposure to other services. In this case we examine the need for securing the application's network communications through a network proxy to filter out requests and to better protect clients and services. Knowing how to implement a strong initial network filter that can effectively identify a majority of attacks dramatically decreases the chance of attackers successfully compromising the system. If most of the attempted attacks are initially thwarted, there are few attacks that can get through the security in place. Again, this results in a smaller attack surface.

3. ARCHITECTURE

As mentioned above, this experiment is divided into two parts. Part one demonstrates how Pod Security Policies can prevent a common vulnerability, and part two demonstrates how Envoy can be configured to better fortify the network security of an application.

3.1 Part 1: Pod Security Polices

For part one, we will be demonstrating how Pod Security Policies can prevent privilege escalation. We wrote two Dockerfiles that create docker images that use the Debian Linux distribution as the base. The `root-debian` image runs as the root user with root privileges by default, while the `non-root-debian` image runs as `appuser` without root privileges. Both images have a secret text file (`/root/secrets.txt`) that only the root user has access to. These two were built using the `docker-compose` in the `app` directory.

We also created two Pod Security Policies to enforce specific rules in the Kubernetes cluster. The permissive (`psp-permissive.yaml`) policy allows pods to run as `root`, while the restrictive (`psp-restrictive.yaml`) policy does not allow pods to run as `root`, and prevents privilege escalation.

We first deployed both applications into separate pods without Pod Security Polices applied. Then we applied the restrictive policy to the cluster and redeployed the applications. For each of the deployments, we observed whether or not these distributed applications successfully deployed, and if they did, we attempted to escalate privileges to the root user and access the secret in `/root/secrets.txt`. Follow the `/psp/README.md` file for a step-by-step setup and procedure to reproduce the experiment results detailed in the next section.

3.2 Part 2: Network Proxy - Envoy

For part two, we will specifically be looking into Envoy, which is an open source proxy that is designed for cloud-based applications/services. There are a number of external services, such as Open Policy Agent, that Envoy can be customized with that will implement different forms of security measures.

The Envoy proxy configuration is configured using a YAML file and consists of listeners, filters, and clusters. A listener configures the IP addresses and ports that the proxy will listen to for network requests. With Envoy specifically, it is run within a Docker container. Filters are defined by `filter_chains`, and handle finding matches with incoming network request to a destination. A filter can be unique to a listener or can be shared amongst multiple listeners. After a filter finds the match between request and destination, that request is passed onto a cluster. The cluster defines the host of the proxy.

We applied an SSL Certificate to an API request and redirect HTTP to an HTTPS request, and utilized Envoy's external authorization function alongside Open Policy Agent to define policies for network requests that Envoy handles. For applying a SSL Certificate and redirecting network traffic to HTTPS, we will be following along the example in "Securing traffic with HTTPS and SSL/TLS" [**?**]. For Envoy's use with Open Policy Agent, we will be following the example given in "Envoy" [**?**]. Through these demonstrations, we showcase envoy's ability to be customized to better secure an application.

4. EXPERIMENTAL RESULTS

4.1 Part 1: Pod Security Polices

4.1.1 Without Pod Security Policies

Both pods successfully deploy and run in the cluster as shown in Figure 1. On the `non-root-debian` pod, the user can escalate privilages to run as the root user as shown in Figure 2.

4.1.2 With Pod Security Policies

With a restrictive PSP that requires the user to run as a non-root user, the `non-root-debian` pod successfully runs while the `root-debian` pod is prevented from running with a `CreateContainerConfigError` as shown in Figure 3 and Figure 4. The PSP prevents the `root-debian` pod from running because the policies does not allow the pod to run as the root user.

The PSP also prevents privilege escalation as well. The `non-root-debian` pod successfully runs as a non-root user. The same privelege escalation method is tried with this pod after applying PSP, and as a result, the privilege escalation fails as shown in Figure 5.

```
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE      NOMINATED NODE   READINESS GATES
pod/non-root-debian-5b8bbd6c5-xx966   1/1     Running   0          15s   10.1.56.99    bcheung   <none>           <none>
pod/root-debian-69dcfd6f4c-7sv4g      1/1     Running   0          14s   10.1.56.100   bcheung   <none>           <none>

NAME                          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE     SELECTOR
service/kubernetes            ClusterIP   10.152.183.1     <none>        443/TCP        3d20h   <none>
service/non-root-debian-svc   NodePort    10.152.183.37    <none>        80:30000/TCP   14s     app=non-root-debian
service/root-debian-svc       NodePort    10.152.183.70    <none>        80:30001/TCP   15s     app=root-debian

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS        IMAGES                                        SELECTOR
deployment.apps/non-root-debian     1/1     1            1           15s   non-root-debian   localhost:32000/app_non-root_debian:k8s       app=non-root-debian
deployment.apps/root-debian         1/1     1            1           14s   root-debian       localhost:32000/app_root_debian:k8s           app=root-debian

NAME                                          DESIRED   CURRENT   READY   AGE   CONTAINERS        IMAGES                                        SELECTOR
replicaset.apps/non-root-debian-5b8bbd6c5     1         1         1       15s   non-root-debian   localhost:32000/app_non-root_debian:k8s       app=non-root-debian,pod-template-hash=5b8bbd6c5
replicaset.apps/root-debian-69dcfd6f4c        1         1         1       14s   root-debian       localhost:32000/app_root_debian:k8s           app=root-debian,pod-template-hash=69dcfd6f4c
```

Figure 1: Screenshot of the Kubernetes resources after deploying the pods without Pod Security Policies

```
bcheung@bcheung:~/Documents/EE379K-Final-Project$ microk8s.kubectl exec -it non-root-debian-5b8bbd6c5-xx966 bash
appuser@non-root-debian-5b8bbd6c5-xx966:/$ ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
appuser@non-root-debian-5b8bbd6c5-xx966:/$ cd root
bash: cd: root: Permission denied
appuser@non-root-debian-5b8bbd6c5-xx966:/$ su root
Password:
root@non-root-debian-5b8bbd6c5-xx966:/# cd root
root@non-root-debian-5b8bbd6c5-xx966:~# cat secrets.txt
This is a secret that only root has access to.
```

Figure 2: Screenshot of the privilege escalation on the non-root-debian pod

```
NAME                                READY   STATUS                       RESTARTS   AGE     IP            NODE      NOMINATED NODE   READINESS GATES
pod/non-root-debian-5b8bbd6c5-cjtl4   1/1     Running                      0          2m18s   10.1.56.103   bcheung   <none>           <none>
pod/root-debian-69dcfd6f4c-b8kgz      0/1     CreateContainerConfigError   0          2m17s   10.1.56.104   bcheung   <none>           <none>

NAME                          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE     SELECTOR
service/kubernetes            ClusterIP   10.152.183.1     <none>        443/TCP        4d1h    <none>
service/non-root-debian-svc   NodePort    10.152.183.21    <none>        80:30000/TCP   2m17s   app=non-root-debian
service/root-debian-svc       NodePort    10.152.183.31    <none>        80:30001/TCP   2m18s   app=root-debian

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS        IMAGES                                        SELECTOR
deployment.apps/non-root-debian     1/1     1            1           2m18s   non-root-debian   localhost:32000/app_non-root_debian:k8s       app=non-root-debian
deployment.apps/root-debian         0/1     1            0           2m17s   root-debian       localhost:32000/app_root_debian:k8s           app=root-debian

NAME                                          DESIRED   CURRENT   READY   AGE     CONTAINERS        IMAGES                                        SELECTOR
replicaset.apps/non-root-debian-5b8bbd6c5     1         1         1       2m18s   non-root-debian   localhost:32000/app_non-root_debian:k8s       app=non-root-debian,pod-template-hash=5b8bbd6c5
replicaset.apps/root-debian-69dcfd6f4c        1         1         0       2m17s   root-debian       localhost:32000/app_root_debian:k8s           app=root-debian,pod-template-hash=69dcfd6f4c
```

Figure 3: Screenshot of the Kubernetes resources after applying Pod Security Policies and deploying the pods

```
Conditions:
  Type              Status
  Initialized       True
  Ready             False
  ContainersReady   False
  PodScheduled      True
Volumes:
  default-token-6clvb:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-6clvb
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type     Reason     Age              From               Message
  ----     ------     ----             ----               -------
  Normal   Scheduled  <unknown>        default-scheduler  Successfully assigned default/root-debian-69dcfd6f4c-b8kgz to bcheung
  Normal   Pulled     7s (x2 over 7s)  kubelet, bcheung   Container image "localhost:32000/app_root_debian:k8s" already present on machine
  Warning  Failed     7s (x2 over 7s)  kubelet, bcheung   Error: container has runAsNonRoot and image will run as root
```

Figure 4: Screenshot of the root-debian pod's status after applying the restrictive PSP and deploying

```
bcheung@bcheung:~/Documents/EE379K-Final-Project$ microk8s.kubectl exec -it non-root-debian-5b8bbd6c5-cjtl4 bash
appuser@non-root-debian-5b8bbd6c5-cjtl4:/$ ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
appuser@non-root-debian-5b8bbd6c5-cjtl4:/$ cd root
bash: cd: root: Permission denied
appuser@non-root-debian-5b8bbd6c5-cjtl4:/$ su root
Password:
su: Authentication failure
```

Figure 5: The non-root-debian pod fails to escalate privileges and login as the root user with the restrictive PSP

4.2 Part 2: Network Proxy - Envoy

We demonstrate just some of the features of Envoy, including the ability to secure HTTP traffic by applying SSL certificates and redirecting HTTP to HTTPS traffic as well using Envoy's External authorization filter used with Open Policy Agent to enforce security policies over API requests that the proxy receives.

4.2.1 SSL Certificate and HTTPS Redirect

In order to apply SSL certificates to a request and redirect traffic to HTTPS, first a SSL certificate needs to be created. Ideally, you would generate the certificate from some service, but a simple certificate can be generated using the following command

```
mkdir certificates;
cd certificates;
openssl req - nodes -new -x509 \
  -keyout "domainname-com.key -out
     domainname-com.crt \
  -days 365 \
  -subj '/CN=domainname.com/O=Company Name
     LTD./C=US';
```

After creating the SSL certificate, the envoy.yaml file needs to be edited in order to apply a TLS context in the filter. A TLS context will specify the certificate(s) to be used with the proxy domains. The following lines will apply the TLS context

```
tls_context:
   common_tls_context:
   tls_certificates:
      - certificate_chain:
         filename: "/filepath_certificate.crt"
      private_key:
         filname: "/filepath_certificate.key"
```

After defining the certificates to be used, the proxy will be able to redirect traffic to HTTPS. In order to complete the redirect, we need to set the `https_redirect` flag to true.

```
redirect:
   path_redirect: "/"
   https_redirect: true
```

These configurations will enable the envoy proxy to attach a SSL Certificate to network requests as well as redirect any HTTP traffic to HTTPS.

4.2.2 Envoy Proxy Configuration with OPA

In this section, we examine envoy's ability to use external authorization services, specifically checking if incoming requests have certain authorization privileges. In this example, we have two clients, Brian and Sam, that want to access an endpoint that has employees. Brian is assigned a guest role, and thus can only get employees but cannot create an employee. Sam is assigned an admin role, which means Sam can get employees as well as create an employee. This policy is implemented with Open Policy Agent (OPA). After creating an envoy.yaml configuration file and mapping it to kubernetes, the OPA policy needs to be created. Below are some code snippets from the policy.rego file

```
action_allowed {
  http_request.method == "GET"
  token.payload.role == "guest"
  glob.match("/people", [], http_request.path)
}
action_allowed {
  http_request.method == "GET"
  token.payload.role == "admin"
  glob.match("/people", [], http_request.path)
}
action_allowed {
  http_request.method == "POST"
  token.payload.role == "admin"
  glob.match("/people", [], http_request.path)
  lower(input.parsed_body.firstname)
     != base64url.decode(token.payload.sub)
}
```

This defines the actions that are allowed. As shown, the admin role is allowed to execute GET and POST requests to the "/people" endpoint, while a guest can only execute a GET request. Below, there are screenshots of Brian, our guest, executing both GET and POST requests, and the GET request is allowed while the POST request is denied. The same requests are executed by Sam, our admin, and both requests are allowed.

```
class@class-VirtualBox:~$ curl -i -H "Authorization: Bearer "$BRIAN_TOKEN"" http://$SERVICE_URL/people
HTTP/1.1 200 OK
content-type: application/json
date: Sat, 14 Dec 2019 19:51:57 GMT
content-length: 151
x-envoy-upstream-service-time: 0
server: envoy

[{"id":"1","firstname":"John","lastname":"Doe"},{"id":"2","firstname":"Jane","lastname":"Doe"},{"id":"498081","firstname":"Charlie","lastname":"Opa"}]
class@class-VirtualBox:~$ curl -i -H "Authorization: Bearer "$BRIAN_TOKEN"" -d '{"firstname":"Charlie", "lastname":"OPA"}' -H "Content-Type: application/json" -X POST http://$SERVICE_URL/people
HTTP/1.1 403 Forbidden
date: Sat, 14 Dec 2019 19:52:24 GMT
server: envoy
content-length: 0
```

Figure 6: Guest role executing both GET and POST requests

```
class@class-VirtualBox:~$ curl -i -H "Authorization: Bearer "$SAM_TOKEN"" http://$SERVICE_URL/people
HTTP/1.1 200 OK
content-type: application/json
date: Sat, 14 Dec 2019 19:52:33 GMT
content-length: 151
x-envoy-upstream-service-time: 0
server: envoy

[{"id":"1","firstname":"John","lastname":"Doe"},{"id":"2","firstname":"Jane","lastname":"Doe"},{"id":"498081","firstname":"Charlie","lastname":"Opa"}]
class@class-VirtualBox:~$ curl -i -H "Authorization: Bearer "$SAM_TOKEN"" -d '{"firstname":"Charlie", "lastname":"Opa"}' -H "Content-Type: application/json" -X POST http://$SERVICE_URL/people
HTTP/1.1 200 OK
content-type: application/json
date: Sat, 14 Dec 2019 19:52:41 GMT
content-length: 55
x-envoy-upstream-service-time: 0
server: envoy

{"id":"727887","firstname":"Charlie","lastname":"Opa"}
```

Figure 7: Admin role executing both GET and POST requests

5. RELATED WORK

[7] The article talks about how more and more enterprises are moving their workloads onto the cloud and while security has evolved over time, is still a major concern. The paper goes into details about the various forms of threats and vulnerabilities of the cloud, specifically listing and detailing 17 threats. This paper provides a good foundation for understanding common threats that exploit cloud vulnerabilities.

[5] This paper discusses the threats that networks face and the current network security practices to counteract these attacks. The paper begins by detailing security attacks, security measures, and security tools. The paper goes into great detail about different security methods, such as application gateways and packet filtering. The paper discusses different things that organizations can do to prepare for these attacks and the various technology options.

[2] This paper further discusses the vulnerabilities of cloud computing services. The paper details cloud service models and talks about the 3 layers of cloud computing: system layer (IaaS), platform layer (PaaS), and application layer (SaaS). The paper then analyzes the various security issues that each layer faces and talks about the threats that exploit those vulnerabilities.

[3] This paper discusses the advantages of using cloud services and also reveals the dangers and risks of those services.

[1] This is a known vulnerability, CVE-2019-5736 [1], that allows attackers to execute commands as root within two types of containers, a new container with an attack-controlled image and an already existing container that an attacker has had access to in the past.

[6] This article goes into detail about what packet filtering is and how it is used as a network security tool. The paper details the benefits of packet filters and gives a simple implementation of it and discusses the limitations of packet filtering.

6. CONCLUSIONS

While Pod Security Policies and Network Proxies have similar goals of maintaining/strengthening the strength of the security of an application, they do have different use cases and have unique strengths when it comes to application/network security. Pod Security Policies secure the base application and the Kubernetes cluster that the application is hosted on. On the other hand, a network proxy is the middleman between the client and the network and thus is met with every network request. Furthermore, designing a network proxy that can effectively handle these network requests, whether that be filtering out malicious requests, or assigning security policies onto API requests is extremely important in protecting the overall security integrity of the application as well as reducing the load that the rest of the application has to deal with. Network proxies can be viewed as the outer line of defense, while Pod Security Policies act as the base-line defense that protects the system if a threat manages to bypass the network proxy. Both services add a layer of security that helps mitigate threats at different levels of the application, which ultimately minimizes the attack surface of the application.

7. REFERENCES

[1] Cve-2019-5736.

[2] T.-S. Chou. Security threats on cloud computing vulnerabilities.

[3] M. Kemal. Cloud security.

[4] Kubernetes. Production-grade container orchestration. https://kubernetes.io/.

[5] S. Pandey. Modern network security: Issues and challenges.

[6] D. Strom. The packet filter: A basic network security tool.

[7] P. S. Suryateja. Threats and vulnerabilities of cloud computing: A review.