Table of Contents

* [Network Policy Lab](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#_network_policy_lab)
* [1. Migrate Projects](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#labexercises)
* [2. Use Network Policy](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#_use_network_policy)
  + [2.1. Change Cluster to Use **ovs-networkpolicy**](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#_change_cluster_to_use_code_ovs_networkpolicy_code)
  + [2.2. Review Example Application](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#_review_example_application)
  + [2.3. Install & Deploy Example Application](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#_install_deploy_example_application)
  + [2.4. Test Application](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#_test_application)
  + [2.5. Exploit Application](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#_exploit_application)
  + [2.6. Deploy Network Policy Objects to Secure Application](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#_deploy_network_policy_objects_to_secure_application)
* [3. Summary](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#_summary)
* [4. Clean Up Environment](https://www.opentlc.com/labs/ocp_advanced_deployment/03_2_NetworkPolicy_Solution_Lab.html#_clean_up_environment)

Network Policy Lab

**Goals**

* Set up a microservices application
* Demonstrate an exploit of the application
* Protect the microservices application with network policy

1. Migrate Projects

In this section, you migrate your projects from the **ovs-multitenant** plug-in to the **ovs-networkpolicy** plug-in.

Before migrating from the **ovs-multitenant** plug-in to the **ovs-networkpolicy** plug-in, make sure that every namespace has a unique NetID. This means that if you have previously joined projects together or made projects global, you need to undo that before switching to the **ovs-networkpolicy** plug-in, or the **NetworkPolicy** objects may not function correctly.

A helper script is available that fixes NetIDs, creates **NetworkPolicy** objects to isolate previously isolated namespaces, and enables connections between previously joined namespaces.

Use the following steps to migrate to the **ovs-networkpolicy** plug-in using this helper script, while still running the **ovs-multitenant** plug-in:

|  |  |
| --- | --- |
|  | This is **only** necessary before switching from **ovs-multitenant** to **ovs-networkpolicy**. |

1. Download the script from <https://raw.githubusercontent.com/openshift/origin/master/contrib/migration/migrate-network-policy.sh>and add the execute file permission:
2. curl -o ./migrate-network-policy.sh https://raw.githubusercontent.com/openshift/origin/master/contrib/migration/migrate-network-policy.sh

chmod a+x migrate-network-policy.sh

1. Run the script (requires the cluster administrator role):

./migrate-network-policy.sh

* + After running this script, every namespace is fully isolated from every other namespace, therefore connection attempts between pods in different namespaces fail until you complete the migration to the **ovs-networkpolicy** plug-in.
  + If you also want newly created namespaces to have the same policies by default, you can set default **NetworkPolicy** objects to be created matching the **default-deny** and **allow-from-global-namespaces** policies created by the migration script.

|  |  |
| --- | --- |
|  | In case of script failures or other errors, or if you later decide you want to revert back to the **ovs-multitenant**plug-in, you can use the [un-migration script](https://raw.githubusercontent.com/openshift/origin/master/contrib/migration/unmigrate-network-policy.sh). This script undoes the changes made by the migration script and rejoins previously joined namespaces. |

2. Use Network Policy

2.1. Change Cluster to Use **ovs-networkpolicy**

1. On the bastion, change **os\_sdn\_network\_plugin\_name** to **redhat/openshift-ovs-networkpolicy** in the Ansible inventory file.
2. You have two choices in your method to update all the masters and nodes in the cluster:
   * 1) Run the **deploy\_cluster** playbook to update all of the nodes in the cluster. It takes about 20 minutes.
   * 2) Alternatively, change the plug-in directly in the **node-config.yaml** and **master-config.yaml** configuration files:

|  |  |
| --- | --- |
|  | This saves about 30 minutes of running the Ansible **deploy\_cluster** playbook. However, you need to make sure to update the **/etc/ansible/hosts** file to prevent accidental overwrites in case you need to run the playbook again. |

* + ansible masters -m shell -a "sed -i -e 's/openshift-ovs-multitenant/openshift-ovs-networkpolicy/g' /etc/origin/master/master-config.yaml"
  + ansible nodes -m shell -a "sed -i -e 's/openshift-ovs-multitenant/openshift-ovs-networkpolicy/g' /etc/origin/node/node-config.yaml"
  + *# stop openshift*
  + ansible masters -m shell -a"systemctl stop atomic-openshift-master-api"
  + ansible masters -m shell -a"systemctl stop atomic-openshift-master-controllers"
  + ansible nodes -m shell -a"systemctl stop atomic-openshift-node"
  + ansible nodes -m shell -a"systemctl stop docker"
  + ansible nodes -m shell -a"systemctl restart openvswitch"
  + ansible nodes -m shell -a"systemctl start docker"
  + ansible masters -m shell -a"systemctl start atomic-openshift-master-api"
  + ansible masters -m shell -a"systemctl start atomic-openshift-master-controllers"
  + ansible masters -m shell -a"systemctl start atomic-openshift-node" *# (make sure masters are up before nodes)*
  + ansible nodes -m shell -a"systemctl start atomic-openshift-node"

2.2. Review Example Application

This application has four microservices written using different technologies just to showcase how polyglot microservices talk to each other. Some of these microservices have application logic and a database. All of the components run as containers on RHOCP.

* [Front-End Service](https://github.com/newgoliath/microservices-on-openshift/tree/master/php-ui): A service that provides a UI to register users, allows you to log in, displays the tweets for registered users, and is written in PHP.
* [Email Service](https://github.com/newgoliath/microservices-on-openshift/tree/master/python-email-api): A service that sends emails to registered users, is written in Python, and saves data in a MySQL database.
* [Twitter Service](https://github.com/newgoliath/microservices-on-openshift/tree/master/java-twitter-feed-api): A service that pulls tweets for registered users, is written in Java running on Tomcat, and is called by the front-end HTML directly.
* [User Registration Back-End](https://github.com/newgoliath/microservices-on-openshift/tree/master/nodejs-users-api): A service that registers users, saves them in a MongoDB, and is written in Node.js. This service is called by the front-end HTML and does the job of user registration. Once the user is registered, it calls the email service to send an email confirmation.

There are more details available in the [application’s Git repository](https://github.com/newgoliath/microservices-on-openshift). For this network policy use case, you deploy these microservices into three different projects:

* **msclient**: The project that hosts the front-end service.
* **msservices**: The project that hosts the user registration back-end service and the Twitter service.
* **msinfra**: The project that hosts the email service.

The deployment model for these microservices and the expected interactions (in order) are shown in this diagram:

2.3. Install & Deploy Example Application

1. Clone the repository to your bastion host:

git clone https://github.com/newgoliath/microservices-on-openshift

1. Change to the **microservices-on-openshift/installscripts** folder:

cd microservices-on-openshift/installscripts

1. Create three projects:
2. oc new-project msclient
3. oc new-project msservices

oc new-project msinfra

1. Edit the [1.setVariable.sh](https://github.com/newgoliath/microservices-on-openshift/blob/master/installscripts/1.setVariable.sh) script to set the following values:
   * **OSE\_DOMAIN** to your OpenShift domain name—for example, **apps.GUID.example.opentlc.com**
   * **OSE\_CLIENT\_PROJECT** to **msclient**
   * **OSE\_SERVICES\_PROJECT** to **msservices**
   * **OSE\_INFRA\_PROJECT** to **msinfra**
   * **FROM\_GMAIL** to **opentlcdemo@gmail.com**
   * **FROM\_GMAIL\_PASSWORD** to **XirZVyDjVvodKnMXDGwsw4JJsrzKpTby**
   * Dummy values for **TWITTER\_CONSUMER\_KEY**, **TWITTER\_CONSUMER\_SECRET**, **TWITTER\_OAUTH\_ACCESS\_TOKEN**, and **TWITTER\_OAUTH\_ACCESS\_TOKEN\_SECRET**
2. Run the script as follows:

source 1.setVariable.sh

1. Deploy microservices by running the following scripts:
   * Run the script to build and deploy the email service and the MySQL database in the **msinfra** project:

source 2.deployEmailSvc-PythonMySQL.sh

* + Run the script to build and deploy the Twitter service in the **msservices** project:

source 3.deployTwitter-Tomcat.sh

* + Run the script to build and deploy the user registration back-end service in the **msservices** project:

source 4.deployUserRegBackend-NodejsMongo.sh

* + Run the script to build and deploy the front-end service in the **msclient** project:

source 5.deployFrontend-PHP.sh

1. Wait a few minutes for all of the services to be up and running, then execute the following:

$ oc get pods -n msclient

**Sample Output**

NAME READY STATUS RESTARTS AGE

userreg-1-build 0/1 Completed 0 1m

userreg-1-tm3x8 1/1 Running 0 1m

1. Check the **mservices** pods:

$ oc get pods -n msservices

**Sample Output**

NAME READY STATUS RESTARTS AGE

mongodb-1-2zx66 1/1 Running 0 2m

twitter-api-1-build 0/1 Completed 0 2m

twitter-api-1-srvvm 1/1 Running 0 1m

userregsvc-1-b4xq3 1/1 Running 0 1m

userregsvc-1-build 0/1 Completed 0 2m

1. Check the **msinfra** pods:

$ oc get pods -n msinfra

**Sample Output**

NAME READY STATUS RESTARTS AGE

emailsvc-1-build 0/1 Completed 0 3m

emailsvc-1-grb00 1/1 Running 0 2m

mysql-2-ffj05 1/1 Running 0 3m

|  |  |
| --- | --- |
|  | Make sure to verify that the MySQL pod is running in the **msinfra** project. If it is not running, you need to redeploy the application using **oc rollout latest mysql -n msinfra**. |

2.4. Test Application

1. Get the route to the user registration service from the **msclient** project:

oc get route -n msclient

**Sample Output**

NAME HOST/PORT PATH SERVICES PORT TERMINATION WILDCARD

userreg userreg-msclient.apps.GUID.example.opentlc.com userreg 8080-tcp None

1. Access the UI for the front end using the route you just retrieved.
2. Register a few users (using an email address from which you can receive emails).
   * Expect the registered user to receive an email.

So far, you have deployed a few microservices and used them on OpenShift.

2.5. Exploit Application

There is an exploitable feature in the front-end service code that enables the following: If you access the **hack.php** page, it talks to the MySQL database in the **msinfra** project. Your application shouldn’t talk to MySQL at all, regardless of what errors MySQL might throw.

1. Access the **hack.php** page, you’ll get an error from MySQL. So, this app is talking to MySQL, which is undesireable. We need to shut down access from hack to MySQL entirely.

curl http://userreg-msclient.apps.$GUID.example.opentlc.com/hack.php

**Sample Output**

[root@bastion ~]# curl http://userreg-msclient.apps.b90e.example.opentlc.com/hack.php

Here is the list of emails sent: <br><br />

<b>Warning</b>: mysqli\_fetch\_assoc() expects parameter 1 to be mysqli\_result, boolean given in <b>/opt/app-root/src/hack.php</b> on line <b>12</b><br />

|  |  |
| --- | --- |
|  | Ideally the UI should not reach a database behind a microservice, but instead access it via the APIs exposed by the microservice. It is a well-known architecture practice that if you are running an application in a multi-tiered architecture, your UI layer does not talk to the database directly but accesses it instead via APIs exposed by business logic. In fact, the database is generally firewalled, and only ports from the services that are supposed to call the database are allowed. If you are using a flat network such as **ovs-subnet**, these fine-grained network controls between services are not possible. |

2.6. Deploy Network Policy Objects to Secure Application

In this section, you add network policy controls to the microservices so that the abuse described above is not allowed. You add policies that allow specific traffic to your services.

2.6.1. Deploy Network Policy to MySQL Database

First, you protect the MySQL database behind the email service so that it is only accessible from the email service.

1. Get the labels for the **msinfra** service:

oc get svc -n msinfra --show-labels

**Sample Output**

NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE LABELS

emailsvc 172.30.42.184 8080/TCP 1h app=emailsvc,microservice=emailsvc

mysql 172.30.43.224 3306/TCP 1h app=mysql,microservice=emailsvc

1. Determine the labels for the pods in the **msinfra** project:

oc get pods -n msinfra --show-labels | grep Running

**Sample Output**

emailsvc-1-grb00 1/1 Running 0 1h app=emailsvc,deployment=emailsvc-1,deploymentconfig=emailsvc,microservice=emailsvc

mysql-2-ffj05 1/1 Running 0 1h app=mysql,deployment=mysql-2,deploymentconfig=mysql,microservice=emailsvc

* + Expect the network policy to allow traffic coming from pods that have labels matching **app=emailsvc** to reach the MySQL pod that has a **app=mysql** label.

1. Create a **NetworkPolicy** file to allow only email pods to talk to MySQL (port 3306).
2. echo 'kind: NetworkPolicy
3. apiVersion: networking.k8s.io/v1
4. metadata:
5. name: allow-3306
6. spec:
7. podSelector:
8. matchLabels:
9. app: mysql
10. ingress:
11. - from:
12. - podSelector:
13. matchLabels:
14. app: emailsvc
15. ports:
16. - protocol: TCP

port: 3306' >allow-3306.yaml

1. Apply this policy to the **msinfra** project.

oc create -f allow-3306.yaml -n msinfra

**Sample Output**

networkpolicy "allow-3306" created

1. Once the policy is added, test the exploit again:

curl http://userreg-msclient.apps.$GUID.example.opentlc.com/hack.php

**Sample Output**

504 Gateway Time-out

The server didn't respond in time.

* + This is the expected result now that you have a network policy in place.

2.6.2. Create Network Policy for Remaining Services

Now that you have secured the MySQL pod, you need to create policies that allow only the correct traffic between pods.

1. Write a policy called **default-deny.yaml** to completely deny all traffic to the **msclient** project.
2. echo 'kind: NetworkPolicy
3. apiVersion: networking.k8s.io/v1
4. metadata:
5. name: default-deny
6. spec:

podSelector:' >default-deny.yaml

1. Apply the policy to the **msclient** project.

oc create -f default-deny.yaml -n msclient

**Sample Output**

networkpolicy "default-deny" created

1. Test your UI and verify that it returns "Application is not available."
2. Confirm that the route (from **loadbalancer**) is reaching the service at port 8080:

oc get route -n msclient --show-labels

**Sample Output**

NAME HOST/PORT PATH SERVICES PORT TERMINATION WILDCARD LABELS

userreg userreg-msclient.apps.GUID.example.opentlc.com userreg 8080-tcp None app=userreg,microservice=userreg

1. Create and apply a policy called **allow-8080-frontend.yaml** to allow port 8080 for the **userreg** service to enable communication with the UI.
2. echo 'kind: NetworkPolicy
3. apiVersion: networking.k8s.io/v1
4. metadata:
5. name: allow-8080-frontend
6. spec:
7. podSelector:
8. matchLabels:
9. app: userreg
10. ingress:
11. - ports:
12. - protocol: TCP

port: 8080' >allow-8080-frontend.yaml

oc create -f allow-8080-frontend.yaml -n msclient

1. Confirm that your front-end URL works again.
2. Next, apply the previously created **default-deny.yaml** policy to deny all traffic to the **msservices** project.

oc create -f default-deny.yaml -n msservices

**Sample Output**

networkpolicy "default-deny" created

1. Create a policy called **allow-8080-userregsvc.yaml** to allow port 8080 for the user registration back-end services.
2. echo 'kind: NetworkPolicy
3. apiVersion: networking.k8s.io/v1
4. metadata:
5. name: allow-8080-userregsvc
6. spec:
7. podSelector:
8. matchLabels:
9. app: userregsvc
10. ingress:
11. - ports:
12. - protocol: TCP

port: 8080' >allow-8080-userregsvc.yaml

1. Create a similar policy called **allow-8080-twitter.yaml** to allow port 8080 for the Twitter services.
2. echo 'kind: NetworkPolicy
3. apiVersion: networking.k8s.io/v1
4. metadata:
5. name: allow-8080-twitter
6. spec:
7. podSelector:
8. matchLabels:
9. app: twitter-api
10. ingress:
11. - ports:
12. - protocol: TCP

port: 8080' >allow-8080-twitter.yaml

1. Apply the policies to the **msservices** project.
2. oc create -f allow-8080-userregsvc.yaml -n msservices

oc create -f allow-8080-twitter.yaml -n msservices

1. Create and apply a policy called **allow-27017.yaml** to allow MongoDB, port 27017, to be accessed only by the user registration service.
2. echo 'kind: NetworkPolicy
3. apiVersion: networking.k8s.io/v1
4. metadata:
5. name: allow-27017
6. spec:
7. podSelector:
8. matchLabels:
9. app: mongodb
10. ingress:
11. - from:
12. - podSelector:
13. matchLabels:
14. app: userregsvc
15. ports:
16. - protocol: TCP

port: 27017' >allow-27017.yaml

oc create -f allow-27017.yaml -n msservices

1. Using the **default-deny.yaml** policy, deny all traffic to the **msinfra** project.

oc create -f default-deny.yaml -n msinfra

2.6.3. Connect Across Projects

Previously, you configured a policy that only the email service is allowed to access the MySQL database. You now have to allow connection to the email service in the **msinfra** project from the user registration service in the **msservices** project.

While you can select the destination in this case, you cannot select a specific pod on the source side when the namespaces are different. So you need to open up the **msservices** namespace to be able to reach the email service in **msinfra**.

1. First, add a label to identify the (traffic) source project, **msservices**:

oc label namespace msservices project=userregservices

**Sample Output**

namespace "msservices" labeled

1. Now create and add a policy called **allow-8080-emailsvc.yaml** that allows services in the **msservices** project to talk to the email service in the **msinfra** project.
   * Be sure to use **namespaceSelector**, as you are opening the traffic flow from the **msservices** namespace in this case.
   * echo 'kind: NetworkPolicy
   * apiVersion: networking.k8s.io/v1
   * metadata:
   * name: allow-8080-emailsvc
   * spec:
   * podSelector:
   * matchLabels:
   * app: emailsvc
   * ingress:
   * - from:
   * - namespaceSelector:
   * matchLabels:
   * project: userregservices
   * ports:
   * - protocol: TCP

port: 8080' >allow-8080-emailsvc.yaml

oc create -f allow-8080-emailsvc.yaml -n msinfra

* + You now have policies in place that prevent services crossing boundaries.

1. Test the application to make sure the calls between microservices are all going through.

3. Summary

You should now understand the following:

* Why network policy objects are important.
* How to create policies that deny all the traffic.
* How to open up traffic to specific destinations based on labels.
* How to open up traffic that crosses projects.

4. Clean Up Environment

1. Delete the three projects:
2. oc delete project msinfra
3. oc delete project msservices

oc delete project msclient

Build Version: c3147ce9f77191e30b447cc423f2f68a0c40fc03 : Last updated 2018-07-31 01:29:02 EDT