A PRACTICAL INTRODUCTION TO CONTAINER SECURITY

Thursday, June 30, 2016, 3:30PM-5:30PM, Room 3018, Lab 3

Presenters

Bob Kozdemba, Principal Solutions Architect, Red Hat, Inc.

Bob Kozdemba is a field architect who specializes in open source container application platforms. Bob is a Red Hat Certified Architect (RHCA) and holds an MS in Information Technology and an MBA from the University of Maryland. In his spare time he enjoys studying jazz and blues and performs with the Austin Classical Guitar Ensemble.

Dan Walsh, Consulting Software Engineer, Red Hat, Inc.

Dan Walsh has worked in the computer security field for over 30 years. Dan joined Red Hat in August 2001. Dan leads the RHEL Docker enablement team since August 2013, but has been working on container technology for several years. He has led the SELinux project, concentrating on the application space and policy development. Dan helped developed sVirt, Secure Vitrualization. He also created the SELinux Sandbox, the Xguest user and the Secure Kiosk. Previously, Dan worked Netect/Bindview's on Vulnerability Assessment Products and at Digital Equipment Corporation working on the Athena Project, AltaVista Firewall/Tunnel (VPN) Products.

Abstract

Linux containers provide convenient application packing and run-time isolation in multi-tenant environments. However, the security implications of running containerized applications is often taken for granted. For example, today it's very easy to pull docker images from the internet and run them in a datacenter without examining their content and authenticity. During this lab, students will complete hands-on exercises to understand the concepts and challenges associated with deploying containers at an enterprise scale in a secure fashion. You'll learn how to run containers with elevated privileges, use atomic tools to scan, verify, and compare containers and images for common vulnerabilities and exposures(CVEs). You'll also experiment with how SELinux works with containers.

Overview and Prerequisites

This session is a self paced, hands introduction to container security using Red Hat Enterprise Linux 7. The prerequisites for this lab include basic Linux command line and text editing skills. A basic knowledge of Docker is helpful.

Lab Environment

Your workstation is configured with the following virtual machines running RHEL7 Server connected via a private libvirt network.

- rhserver0.example.com (192.168.0.100)
- rhserver1.example.com (192.168.0.101)
- rhserver2.example.com (192.168.0.102)

Open a terminal window on your workstation and use the `ssh` client to login to the various systems as **root** (password is **redhat**). Before you get started with the labs. run a quick test to make sure you can connect between the virtual servers on their private network.

For example:

```
$ ssh root@rhserver0.example.com
[root@rhserver0 ~]# ping -c1 rhserver1.example.com
[root@rhserver0 ~]# ping -c1 rhserver2.example.com
```

Now go ahead and dive in. If you have a question, flag one of us down and we'll be happy to help.

Lab 1: Registry and Docker Configuration

Most of the exercises in this hands on lab will be completed on the *rhserver0* system. The *rhserver1* and *rhserver2* systems will be used only as Docker registry servers.

Registry Configuration

Configure and start the Docker registry services on the *rhserver1* and *rhserver2* systems.

Hints:

- Enable and start the docker registry.
  ```
  # systemctl enable docker-registry
  # systemctl start docker-registry
  # systemctl status docker-registry
  ```

- The registry listens on port 5000 by default so you'll want to open up the firewall ports.
  ```
  # firewall-cmd --add-port 5000/tcp
  # firewall-cmd --add-port 5000/tcp --permanent
  ```

Open a separate terminal window, login to each registry server and follow the registry logs.

```
# journalctl -u docker-registry --follow
```

Run a quick test from each server.

```
# curl http://localhost:5000
```

The curl command should return a `"docker-registry server\"` string. You should also see a `GET` entry in the logs from the `curl`.

Docker Configuration

On *rhserver0*, configure the Docker service to trust the registry running on *rhserver1*.

Hints:
Using the `vim` or `nano` text editor, edit /etc/sysconfig/docker on rhserver0 and add rhserver1.example.com:5000 as an INSECURE_REGISTRY then restart the docker service.

```
# systemctl restart docker
```

Run a quick test from rhserver0.

```
# curl http://rhserver1.example.com:5000
```

The curl command should return a `"docker-registry server\"` string. This will confirm that your firewall settings on the registry servers are working. You should also see a `GET` in the logs from the `curl`.

Is there another method that you can think of to test the registry?

Pushing images to a remote registry

Login to *rhserver0*. Notice that a few images are present in the local Docker image cache. Run the following command to see them.

```
# docker images
```

Test the registry service on both *rhserver1* and *rhserver2* using the docker command line from *rhserver0*. Use the docker command to tag a local image and push it to the registry running on *rhserver1*. Then remove your local copy and pull the image from the remote registry. For this exercise tag, and push all of the images in the local docker cache on *rhserver0* to both registry servers.

Hints:

```
# man docker-tag
# docker tag <image-name> rhserver1.example.com:5000/<image-name>
# docker images
# man docker-push
# docker push rhserver1.example.com:5000/<image-name>
```

If the push was successful, make a backup copy then delete (untag) the local cached copy and pull a new copy from the remote registry.

```
# man docker-tag
# docker tag rhserver1.example.com:5000/<image-name>:latest
rhserver1.example.com:5000/<image-name>:backup
# docker rmi rhserver1.example.com:5000/<image-name>:latest
# docker pull <registry-host>:<port>/<image-name>
```

Login to *rhserver1* and check the registry logs.

```
# journalctl -udocker-registry --follow
```

Lab 2: Authorization

The Docker software that ships with RHEL has the ability to block remote registries. For example, in a production environment you might want to prevent users from pulling random containers from the public internet by blocking Docker Hub (docker.io). During this lab you will configure docker on *rhserver0* to block the registry on *rhserver*2, then try to pull or run the image from the blocked registry.

Perform the following by editing `/etc/sysconfig/docker` on *rhserver0*:

- Configure docker to use *rhserver2* as an insecure registry.
- Configure docker to block *rhserver2* (see `BLOCK_REGISTRY=`).
- Restart the docker daemon.
- Try to pull or run the image that was pushed to the registry on *rhserver2*. It should fail.

Lab 3: Isolation

Containers provide a certain degree of process isolation via kernel namespaces. In this lab, we'll examine the capabilities of a process running in a containerized namespace. Begin by running a container and looking at it's capabilities.

```
# docker run --rm -ti --name temp registry.access.redhat.com/rhel7 grep
Cap /proc/self/status
```

A non-null `CapEff` value indicates the process has capabilities.

Now run the same command as a non-root user and compare the results.

```
# docker run --rm -ti --name temp --user 32767
registry.access.redhat.com/rhel7 grep Cap /proc/self/status
```

Run as root but drop all capabilities.

```
# docker run --rm -ti --name temp --cap-drop=all
registry.access.redhat.com/rhel7 grep Cap /proc/self/status
```

Let's dig in a bit deeper. Run the `sleep` command in a container as a daemon (-d) then run some additional commands on the host to examine capabilities.

```
# docker run -d --name sleepy registry.access.redhat.com/rhel7 sleep
9999
```

Check to make sure the sleepy container is running.

```
# docker ps
```

Run the `pscap` command on the container host to produce a report of process capabilities. If the application has any capabilities, they will be listed in the report. You should see that the *sleep* process has a number of capabilities. If a process is not in the report, it has dropped all capabilities.

```
# pscap
```

When you are finished, stop then remove the sleepy container.

```
# docker stop sleepy
# docker rm sleepy
```

Now, repeat the procedure above but run the container as a non-root user. Again, run the `pscap` command again and compare the difference in capabilities.

```
# docker run -d --name sleepy --user 32767
registry.access.redhat.com/rhel7 sleep 9999
```

```
# pscap
```

When you are finished, stop and remove the sleepy container.

```
# docker rm --force sleepy
```

SELinux Basics

In this section, we'll cover the basics of SELinux and containers. SELinux policy prevents a lot of break out situations where the other security mechanisms fail. With SELinux on Docker, we write policy that says that the container process running as `svirt_lxc_net_t` can only read/write files with the `svirt_sandbox_file_t` label.

On rhserver0, create the following directories.

```
# mkdir /data /shared /private
```

Run bash in a rhel7 container and volume mount the /data directory on rhserver0 to the /data directory in the container's file system. Once the container is running, verify the volume mount and try to list the contents of /data and the files.

```
# docker run --rm -it --name rhel7 -v /data:/data
registry.access.redhat.com/rhel7 bash
```

Notice the bash prompt changes when you enter the container's namespace. Did the mount succeed? Can you examine the /data directory?

```
[container \]# df
[container \]# ls /data
[container \]# date > /data/date.txt
```

Can you create a file in the /data directory? The container ran as root, correct?

Open a second window on rhserver0 and examine the selinux labels on the host.

```
# ls -dZ /data
```

Find the selinux context of bash in the container.

```
# ps -eZ | grep bash
```

Find the selinux file context associated with containers.

```
# semanage fcontext --list | grep svirt
```

Change the context of `/data/file2` to match the container's context.

```
# chcon -Rt svirt_sandbox_file_t /data
```

Now try to create a file again from the container shell.

```
[container \]# date > /data/date.txt
```

Exit the container.

```
[container \]# exit
```

Private Mounts

Now let Docker create the SELinux labels. Repeat the scenario above but instead add the `:Z` option for the bind mount the `/private` directory then try to create a file in the /private directory from the container's namespace.

```
# docker run -d --name sleepy -v /private:/private:Z
registry.access.redhat.com/rhel7 sleep 9999
```

Note the addition of a unique Multi-Category Security (MCS) label to the directory. SELinux takes advantage of MCS separation to ensure that the processes running in the container can only write to svirt_sandbox_file_t files with the same MCS Label `s0`.

```
# ls -dZ /private
```

Shared Mounts

Repeat the scenario above but instead add the `:z` option for the bind mount then try to create a file in the `/shared` directory from the container's namespace.

```
# docker run -d --name sleepy -v /shared:/shared:z
registry.access.redhat.com/rhel7 sleep 9999
```

Notice the SELinux label.

```
# ls -dZ /shared
```

Lab 4: Inspecting Content

Docker images can easily be pulled from any public registry and run on a container host but is this good practice? Do we trust this image and what are its contents? A better approach would be to inspect and scan the image first. The atomic command ships with both RHEL7 Server and RHEL7 Atomic Host.

atomic diff

Hints:

```
# atomic diff --help
# man atomic-diff
```

Run the rhel7 image and connect to its namespace with bash. Then make some change like creating a file or something.

```
# docker run --rm -it --name my_container
registry.access.redhat.com/rhel7 bash
[container /]# date > /usr/tmp/date.txt
```

Now, open a new terminal window, ssh into rhserver0 and run atomic diff to see the differences between the rhe7 image and the running container.

```
# atomic diff registry.access.redhat.com/rhel7 my_container
```

Atomic should report a list of differences between the two file systems. The `/usr/tmp/date.txt` file should appear in the report.

Exit the container namespace when you're finished.

```
[container /]# exit
```

atomic mount

Next we'll use the atomic command to inspect a container's filesystem by mounting it to the host.

```
# mkdir /mnt/image
# atomic mount registry.access.redhat.com/rhel7 /mnt/image
# cat /mnt/image/etc/redhat-release
```

How might you search a container for all programs that are owned by root and have the SETUID bit set? Sound like a good idea for a custom container scanner?

```
# find /mnt/image -user root -perm -4000 -exec ls -ldb {} \;
```

Unmount when finished.

```
# atomic umount /mnt/image
```

Live mount

Use atomic to live mount a running a container. This option allows the user to modify the container's contents as it runs or update the container's software without rebuilding the container.

```
# docker run --rm --name sleepy registry.access.redhat.com/rhel7 sleep
9999
```

Open a second window and mount the running container's file system from the host.

```
# mkdir /mnt/live
# atomic mount --live sleepy /mnt/live
# date > /mnt/live/usr/tmp/date.txt
```

Now exec into the container's namespace and examine the file that was created above.

```
# docker exec -it sleepy bash
```

```
[container \] # cat /usr/tmp/date.txt
```

Before unmounting, open another terminal window on *rhserver0* and take note of the SELinux label on the mount point.

```
# ls -dZ /mnt/live
# atomic umount /mnt/live
```

Shared mount

This option mounts a container with a shared SELinux label.
This seems to work but what is a good demo of shared mounting?

```
# atomic mount --shared sleepy /mnt/live
# ls -dZ /mnt/live
```

Compare the SELinux label of the mount point to the live mount in the step above.

```
Unmount the container.
```

```
# atomic umount /mnt/live
```

Exit from the container namespace.

```
[container \] # exit
```

Atomic verify

This option is currently under development so we'll leave it as a homework exercise. For now, have a quick look at the `atomic-verify(1)` man page. Refer to https://bugzilla.redhat.com/show_bug.cgi?id=1341347 for details.

Skopeo

Skopeo is an atomic project that ships as a technology preview with RHEL7. Example use cases include retrieving information from remote registries and signing content. You can read more about it on github but for now, try the following simple example.

Run the `skopeo` command from rhserver0 and inspect one of the images that you pushed to the registry on *rhserver1*.

Hints:

```
# skopeo docker://<remote-registry-host:port>/<image> | jq-linux64
```

Lab 5: Atomic Scanner

Before containers are run, it makes good sense to be able to scan container images for known vulnerabilities and configuration problems. A number of container scanning tools are beginning to appear including RHEL's `atomic scan` command.

OpenSCAP scanner

Get started by running the built-in atomic scanner that ships with RHEL.

Hints:

```
# atomic scan --help
# atomic scan --list
```

Scan the rhel7 image using the OpenSCAP scanner. This will use the default scan type (more about that later).

```
# atomic scan registry.access.redhat.com/rhel7
```

In addition to container images, running containers can also be scanned. Scan the sleepy container that maybe still running from the previous lab.

How would you scan all running containers on a given host?

Try running the scanner on an image in one of the remote registries.

Finally, have a look at the contents of the `/var/lib/atomic/atomic_scan_openscap` directory on the host. The scanner itself runs as a container and writes its results in the host's file system. The scanning tools do not run as privileged containers but they are able to mount up a read only rootfs along with a writeable directory on the host's file system so the scanner can place its output. You'll lean more about this feature in the final lab.

Scan Types

Scanners can support a number of scan types. In the section, configure atomic to enable the openscap scanner's standard compliance scan type.

Hints:

First, make a backup copy of the scanner configuration file into /usr/tmp.

```
# cp /etc/atomic.d/atomic_scan_openscap /usr/tmp
```

Edit `/etc/atomic.d/atomic_scan_openscap and a`dd an element to the scans array with the following properties:

```
{ name: compliance,
        args: ['oscapd-evaluate', 'scan',  '--no-cve', '--targets',
'chroots-in-dir:///scanin',  '--output', '/scanout'],
        description: "Performs a standard compliance scan."}
```

Verify the scanner now supports the *compliance* scan type.

```
# atomic scan --list
```

Run the scanner using the new compliance scan type.

```
# atomic scan --scanner atomic_scan_openscap --scan_type compliance
registry.access.redhat.com/rhel7
```

Lab 6: Custom Scanners

The atomic scanner was designed with a pluggable architecture to allow developers to write custom scanners using any programming language of supported by RHEL. Adding a scanner plugin involves the following:

- Make atomic aware of your plug-in.
- Ensure the plugin obtains the proper input from the `/scanin` directory.
- Ensure the plugin writes the results to the `/scanout` directory.

Installing a custom scanner

Hints:

```
# cd /root/custom-scanner
```

Build a docker image that contains the new scanner.

```
# docker build --rm=true --force-rm=true --tag=example_plugin .
```

The example_plugin image should appear in the docker image cache.

```
# docker images
```

Now install the scanner and confirm it is configured.

```
# atomic install --name example_plugin example_plugin
# atomic scan --list
```

It should report 2 scanners each with 2 scan types. Also, look in the `/etc/atomic.d` directory.

Edit `/etc/atomic.conf` and set a *default_scanner* field to the new example_plugin scanner.

Confirm the default setting.

```
# atomic scan --list
```

Run the new scanner using the default scan type.

```
# atomic scan <image>
```

Use a specific scanner and scan type to find out more about the mystery image that you pushed to the registry on *rhserver1*.

```
# atomic scan --scanner example_plugin --scan_type=get-os
rhserver1.example.com:5000mystery
```

Writing a custom scanner

As an example of how to create a custom scanner, you'll make changes to the custom scanner source code and rebuild its container image.

```
# cd custom-scanner
```

Have a look at the scanner source code in the list_rpms.py source file. The `atomic scan` command will bind mount directories so the scanner container can read from its `/scanin` directory and write to its `/scanout` directory.

Begin by making a backup copy of the `list_rpms.py` file and modify the custom scanner python source code. A simple, recommended change would be to insert an 'etc/lsb-release' element into the array at line 39 before the 'etc/debian_version' element.

Also make a backup copy of the original scanner image by change the image tag. Build a docker image and install the scanner as you did above.

```
# docker tag example_plugin:latest example_plugin:v1
# docker images
# docker build --rm=true --force-rm=true --tag=example_plugin .
```

Now run the modified example_plugin scanner on the mystery image again. If everything worked, the scanner should help you solve the mystery.

```
# atomic scan --scanner example_plugin --scan_type=get-os
rhserver1.example.com:5000/mystery
```

Extra Credit

Tag the new scanner image so it can be pushed to the registry on rhserver1. On rhserver0, configure the scanner to pull the scanner image directly from the registry on rhserver1 by editing the scanner configuration file in `/etc/atomic.d`

Lab 7: Read-only Containers

Imagine a scenario where an application gets compromised. The first thing the bad guy wants to do is to write an exploit into the application, so that the next time the application starts up, it starts up with the exploit in place. If the container was read-only it would prevent leaving a backdoor in place and be forced to start the cycle from the beginning.

Docker added a read-only feature but it presents challenges since many applications need to write to temporary directories like `/run` or `/tmp` and when these directories are read-only, the apps fail. Red Hat's approach leverages tmpfs. It's a nice solution to this problem because it eliminates data exposure on the host. As a best practice, run all applications in production with this mode.

At the time of this writing, the --tmpfs feature is only available via the *docker-latest* RHEL package so you will need to swap out the Docker runtime in order to complete the lab.

Confirm that you have pushed the docker images to the registries on rhserver1 or rhserver2.

```
# docker tag registry.access.redhat.com/rhel7
rhserver1.example.com:5000/rhel7
# docker push rhserver1.example.com:5000/rhel7
```

Configure rhserver0 to use the docker-latest environment.

```
# systemctl stop docker
# systemctl disable docker
# yum remove docker
# yum reinstall /root/dist/docker-latest-1.10.3-22.el7.x86_64.rpm
```

Edit `/etc/sysconfig/docker-latest` and configure *rhserver0* as an insecure registry as you did in the first lab then enable and start the docker-latest service.

```
# systemctl enable docker-latest
# systemctl start docker-latest
# systemctl status docker-latest
```

Now run a read-only container and specify a few writable file systems using the `--tmpfs` option.

```
# docker run --rm -ti --name test --read-only --tmpfs /run --tmpfs /tmp
rhserver1.example.com:5000/rhel7 bash
```

Now, try to the following. What fails and what succeeds? Why?

```
[container \]# mkdir /newdir
[container \]# mkdir /run/newdir
```

This concludes the lab on container security. We hope you had fun and learned something in the process.Thanks for attending and please complete the course survey.