DETASSELING DOCKER AND OTHER KERNEL RELATED PROTECTIONS

Agenda

So Much content, “I know we aren’t going to get to all of it”

- ✔ About Me
- ✔ Our App. Scenario
- ✔ Quick Primer
- ✔ Builder: (Developing / maintaining apps via containers)
- ✔ Defender: (Container Security)
- ✔ Breaker: (Common attacks)
- ✔ Breaker: (Less common attacks)
- ✔ Wrap-Up
- ✔ Questions
- ✔ Extra Content (If we have time)

$ whoami

THE ORIGIN STORY
OF
Zach

2016

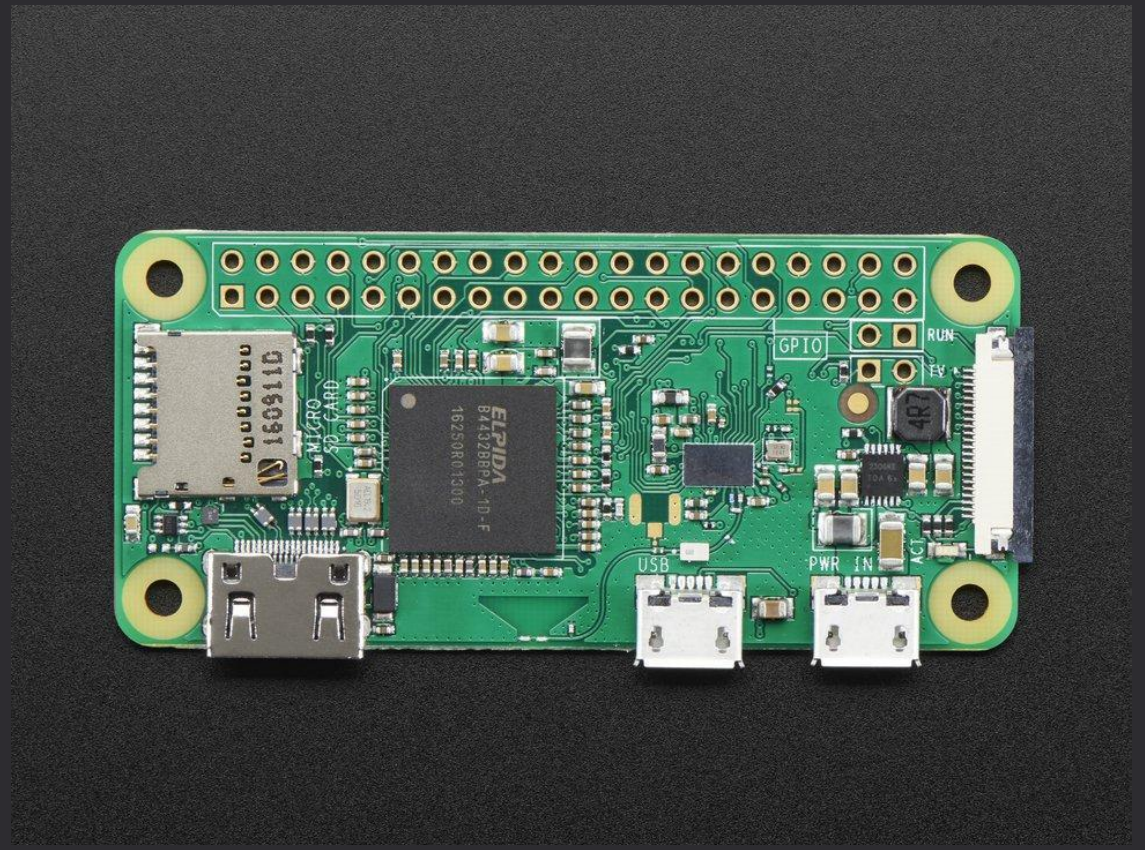
Application Security

Bigger Credit Card Processer

Kernelcon

Quick poll


47 Networks Found
Batt 100%
Ch RSSI Encryption SSID
SELECT START
A
B

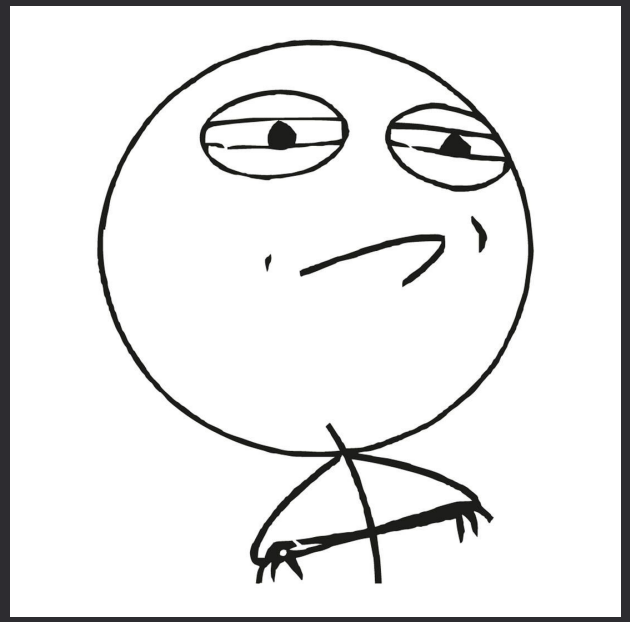

GPIO
RUN
USB
PWR IN
ACT
MICRO SD CARD
ELPIDA

Scenario

Protect an ageing business critical application during it's planned sunset lifespan.

Some assumptions:

- Sunset is relative. (Who really know how long this thing will be around)
- Some functionality is still needed / wanted for future projects
- Minimal development needed. (bug fix, security patch)
- System stability is paramount



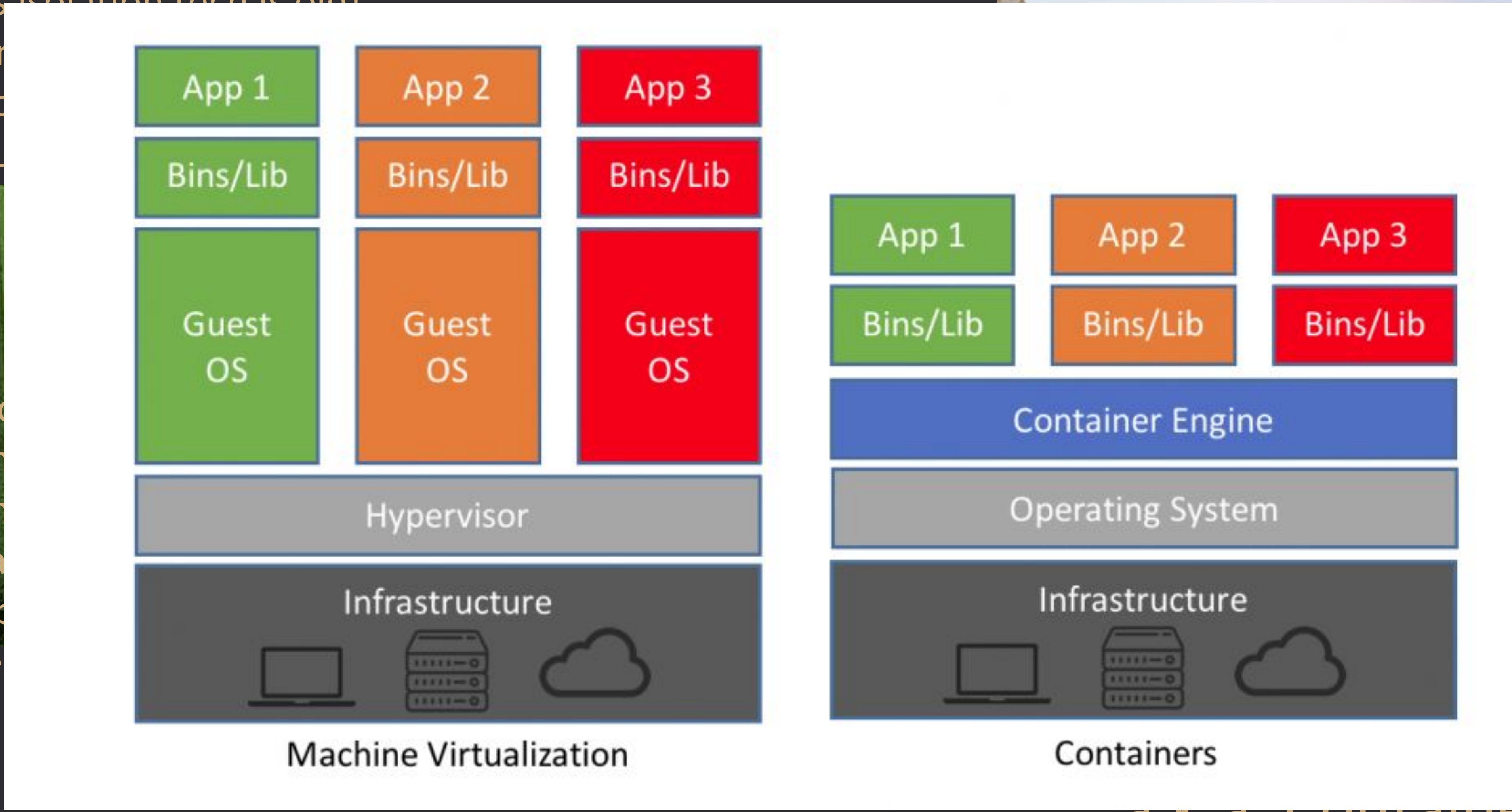
Why Choose Microservices?

Challenges due to this decision.

- Development challenges
- Deployment challenges
- Security challenges

Quick Primer

- Process isolation tech is old!
  Remem
- Used co
- 2005 Su



- Resourc
- Contain
- Contain
- Declara
  develop
  may be

on
a.k.a. Containers

Why?, Why not!

- Chroot was used extensively in the past to assist with build environments. As kernel capabilities and protections got better, so did their adoption and use in the build process.
- Eventually someone just asked the question “Why duplicate all the work?”
- It’s cheap, personal dev. Stacks or team stacks.
- Control dependency and “setup” hell.
- docker-compose integration into your IDE

Reduce container image size

- Use multi-stage builds
- Use “FROM scratch” within Dockerfile “if you can”
- Add “—no-install-recommends” to apt-get calls
- Use reduction tools like “Docker-Slim”


DEMO GODS
PLEASE MAKE THIS DEMO WORK

Start with a good host

- Leverage existing hardening systems like:
  - TOMOYO
  - AppArmor (great doc's on Docker site)
  - SELinux (Extensive documentation)
  - GRSEC

- Networks
  - Avoid "--link"
  - Make use of "—icc=false" if the container isn't hosting anything …
    You can always force container processes to use exposed ports.
  - Use isolated networks or encrypted overlay networks if using multiple hosts
    docker network create -d overlay --attachable my-attachable-overlay

User context

- Docker runs as root!! (non-root can access it but …)
- Remap inner container users to non-root users
  dockerd --userns-remap="testuser:testuser"
- Support for the “--user” command is limited to “run”
  CI/CD systems may need to augment there yaml via bash wrapping. i.e. “${THIS_UID}”
  THIS_UID=$(id -u):$(id -g) docker-compose up

- Storage
  - Avoid using “--bind”
  - Use centralized (managed/backed up) storage drivers
    “for the love of God no NFS, or NFSv4 if you have to”

Security / Defense

- Cgroups
  - control what a process can use.
    - Syscslls, (clone,unshare,setns)
- Namespaces
  - control what a process can see.
  - UTS,NET,MNT,IPC (USER)
- Seccomp
  - What you can “do”
  - Docker managed via profiles
    - --security-opt seccomp=deny.json
- Kernel Capabillitys
  - --cap-drop=all --cap-add=SYS_TIME
- Orchestration Security
  - (RBAC, segmentation, encryption)
- Host Security
  - Remove ssh and local console

Cgroups (Control Groups)

- Docker resource flags like “--cpus” or “--memory”

  docker run -it --rm --cpus="1" busybox
- Doesn't always work! (only if one container per host)

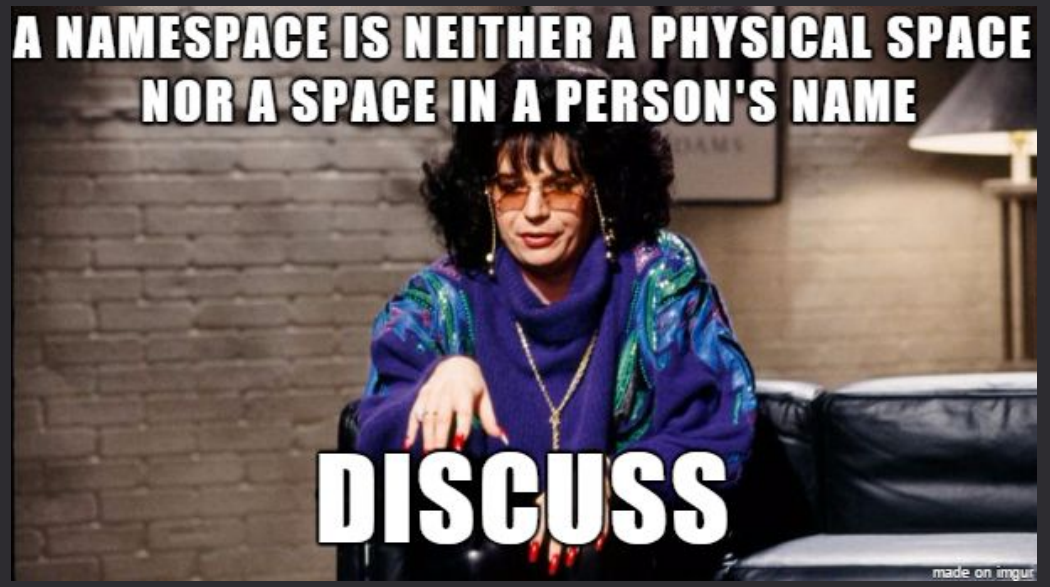
  / # cat /sys/fs/cgroup/cpu/cpu.cfs_quota_us
- Docker’s “backup plan” the “—cgroup-parent” option

  docker run -it --rm --cgroup-parent=/cgroup-limit/ <<image-name>>

/ # cgcreate -g cpu:cgroup-limit
/ # echo 100000 > /sys/fs/cgroup/cpu/cigroup-lmit/cpu.cfs_quota_us
/ # echo 100000 > /sys/fs/cgroup/cpu/cgroup-limit/cpu.cfs_period_us

Namespaces

Kernel enforced user space views.



Peer

- **UTS** – No really “this is your hostname
- **NET** - Docker using this to allow C-2-C and C-2-H IP communication. (own stack, routing etc. i.e. veth)
- **MNT** - used to manage filesystem mount points.
- **IPC** - Widely unused by anyone. “--ipc private”

Hierarchies:

- **PID** - Docker makes use of this to setup the inner process tree, i.e. starting at PID # 1 etc.
- **User** - Docker allows you to map or re-map inner container users to “NODE” or “HOST” users.
- **Cgroups** - Designed to “hide” system limits / limitations.

Seccomp (Secure computing mode)

As stated this is currently a Linux only technology.
If you wanted to achieve a similar technique on windows you would have to make use on EMET and HIPS.

- Kernel dependent. (>= 2.6.29) armv6,7,8 too 😊
- Must be compiled in.

$ grep CONFIG_SECCOMP= /boot/config-$(uname -r)

Look for:

CONFIG_SECCOMP=y
CONFIG_SECCOMP_FILTER=y

- On a 64bit system there are something like 548 syscalls
- There are 306 syscall filters in the docker “default” BPF profile.

Beware: --security-opt seccomp=unconfined

So how do we create a policy for system calls?

- Manually create the policy (Guess)
- Leverage existing tools

https://github.com/david942j/seccomp-tools

https://github.com/docker-slim/docker-slim

Great Advice

- N.I.S.T. 800-190 (April 10 2017)
- Docker C.I.S. (docker-bench-security) [https://github.com/docker/docker-bench-security]
- Kubernetes C.I.S. (kube-bench) [https://github.com/aquasecurity/kube-bench]
- Use a runtime security / monitoring framework
  Falco / anchore Twistlock / Aqua
- Privately store and continuously scan your container images. (Registry)
  Sonatype (Nexus OSS) / Jfrog (Artifactory)
- “--read-only” if you can “get away with it”

Common Attacks

CVE-2019-5736
--userns-remap

- Misconfigurations
  - --privileged AND --security-opt label=disable
  - host insecure, container runtims run as root
- Defaults
  - Doing nothing, not protecting endpoints
  - ”I use StackOverflow as my RunBook” 😊
- Ignorance
  - Not checking source images (poisoned images) [Think BitTorrent or Warez for dev’s]

Less Common Attacks:

- Orchestration leverage
  Improper RBAC
- Kernel vulnerabilities

Demo:
(Time permitting)

(Where do we go from here?)

Builder:

1. Reduce Dependencies
2. 2. Run the C.I.S. benchmarks on your container env. (don't get stuck when your images won't run in prod)
3. Add HEALTHCHECK inside your images

Breaker:

1. Start building a curated list of sites with "DockerFiles" Setup to test vulnerabilities.
2. Start digging into the communications between orchestration systems and container runtimes.
3. Brush up on your kernel exploitation techniques.

Defender:

1. Curate container images (Vuln. Scanning) (set them up for your developers)
2. Require and Audit log outputs (centralized)
3. Maintain Realtime visibility of you container env.
4. Run the C.I.S. benchmarks on your container env.

Questions?

Slides: http://bit.ly/2UEaCJ6

Kernelcon

Fin!

Slides: http://bit.ly/2UEaCJ6

Kubernetes:

R.B.A.C.
Namespace isolation
Pod Security Policy
Network Policy
Mutual Authentication between kubernetes components and pods (SPIFFE / ISTIO)
Securely store secrets (Kubernetes secrets backed by conjure / Vault)
Use TLSv1.2 for EVERYTHING (inside and outside of the cluster)

kubelet:
--anonymous-auth=false and --authorization-mode=Webhook

Simple Container Forensics: (Docker specific)

By leveraging a few docker commands you can do some rudimentary forensics.

- Docker diff –h
- Docker history -h

```
docker run -it --rm –p 1337 --name node-hack jerbi/node-hack

docker network disconnect demo6_default demo6_node-hack_1

docker export --output=”hacked_site.tar" demo6_node-hack_1

docker diff demo6_node-hack_1
```