The Golden Ticket: **Docker and High Security Microservices**

Aaron Grattafiori

Technical Director

dockercon 16

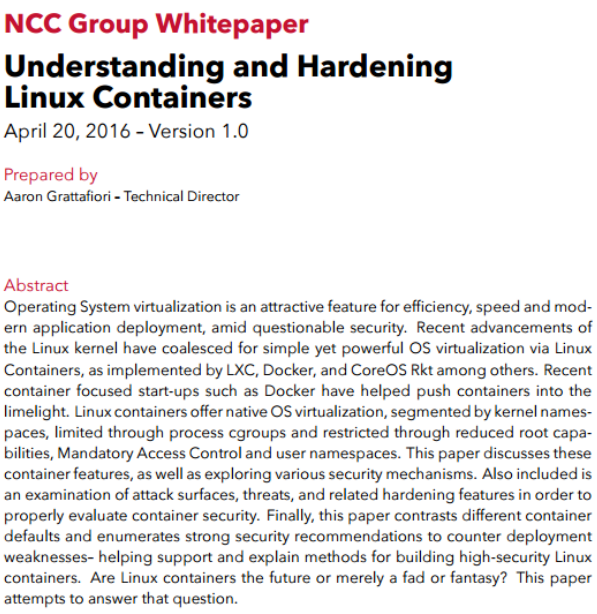
nccgroup

whoami(1)

Longtime Hacker,

Pentester, and Linux geek

Wrote a few things about Linux Containers…



NCC Group Whitepaper

Understanding and Hardening Linux Containers

April 20, 2016 - Version 1.0

Prepared by

Aaron Grattafiori - Technical Director

Abstract

Operating System virtualization is an attractive feature for efficiency, speed and modern application deployment, amid questionable security. Recent advancements of the Linux kernel have coalesced for simple yet powerful OS virtualization via Linux Containers, as implemented by LXC, Docker, and CoreOS Rkt among others. Recent container focused start-ups such as Docker have helped push containers into the limelight. Linux containers offer native OS virtualization, segmented by kernel namespaces, limited through process cgroups and restricted through reduced root capabilities, Mandatory Access Control and user namespaces. This paper discusses these container features, as well as exploring various security mechanisms. Also included is an examination of attack surfaces, threats, and related hardening features in order to properly evaluate container security. Finally, this paper contrasts different container defaults and enumerates strong security recommendations to counter deployment weaknesses- helping support and explain methods for building high-security Linux containers. Are Linux containers the future or merely a fad or fantasy? This paper attempts to answer that question.

dockercon 16

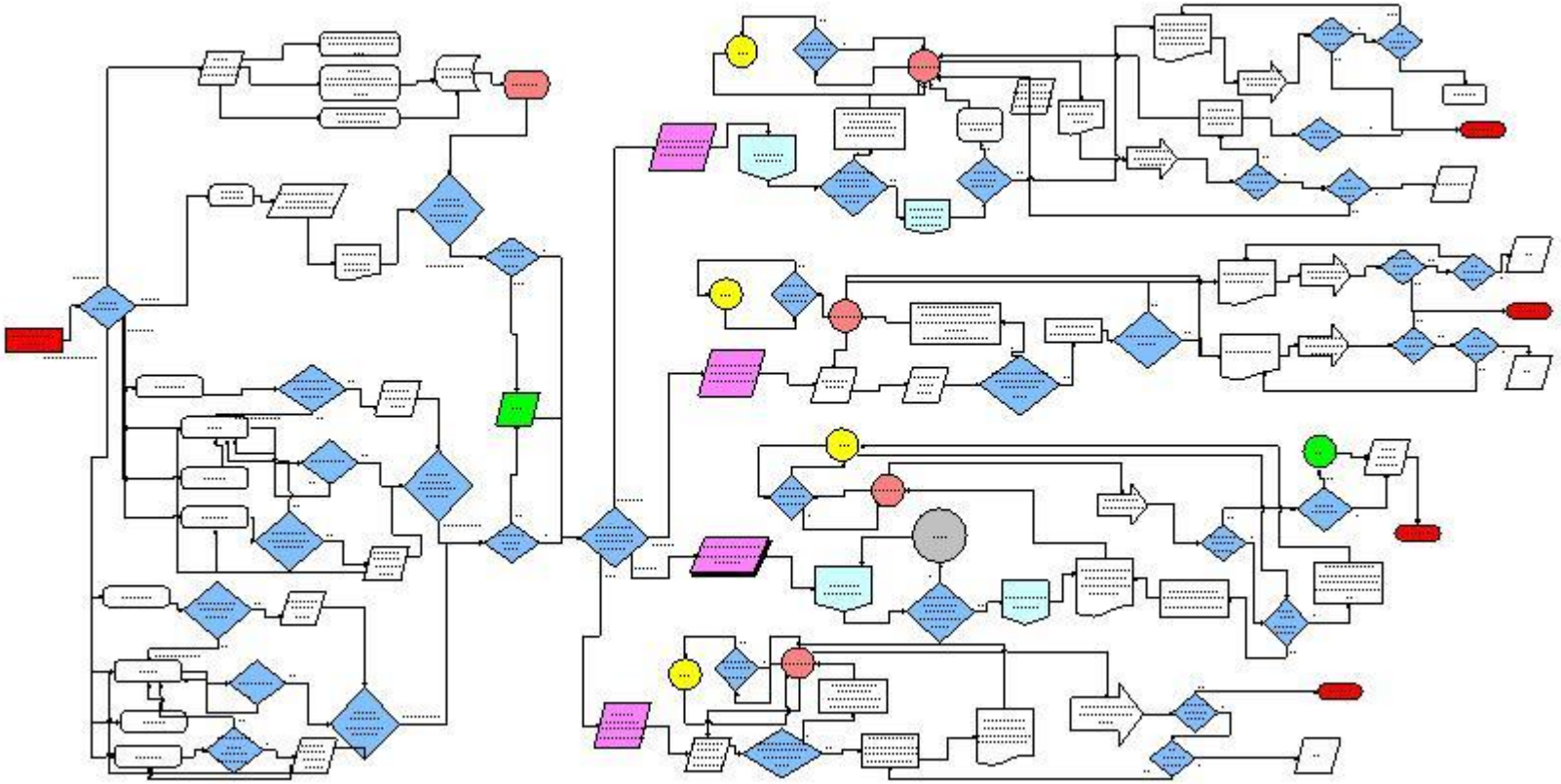

the Bay
the Bay

a disclaimer…

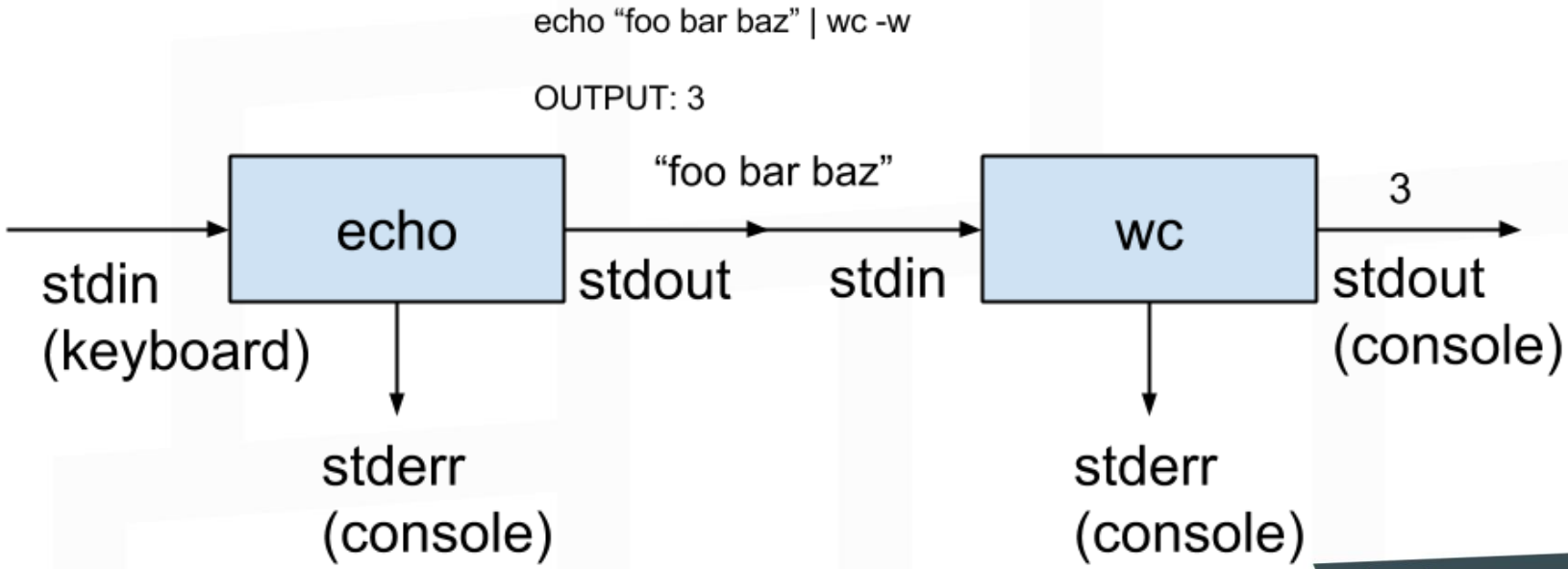
Microservices are **not for everyone**

Microservices can be **hard**

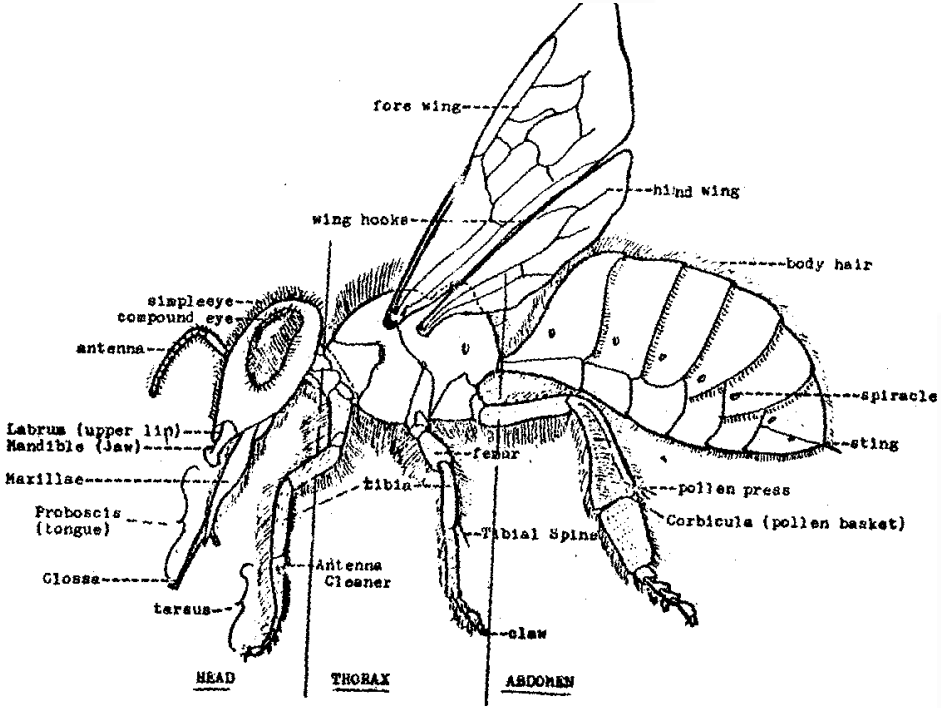
Microservices often **distribute complexity**



You've seen Microservices before

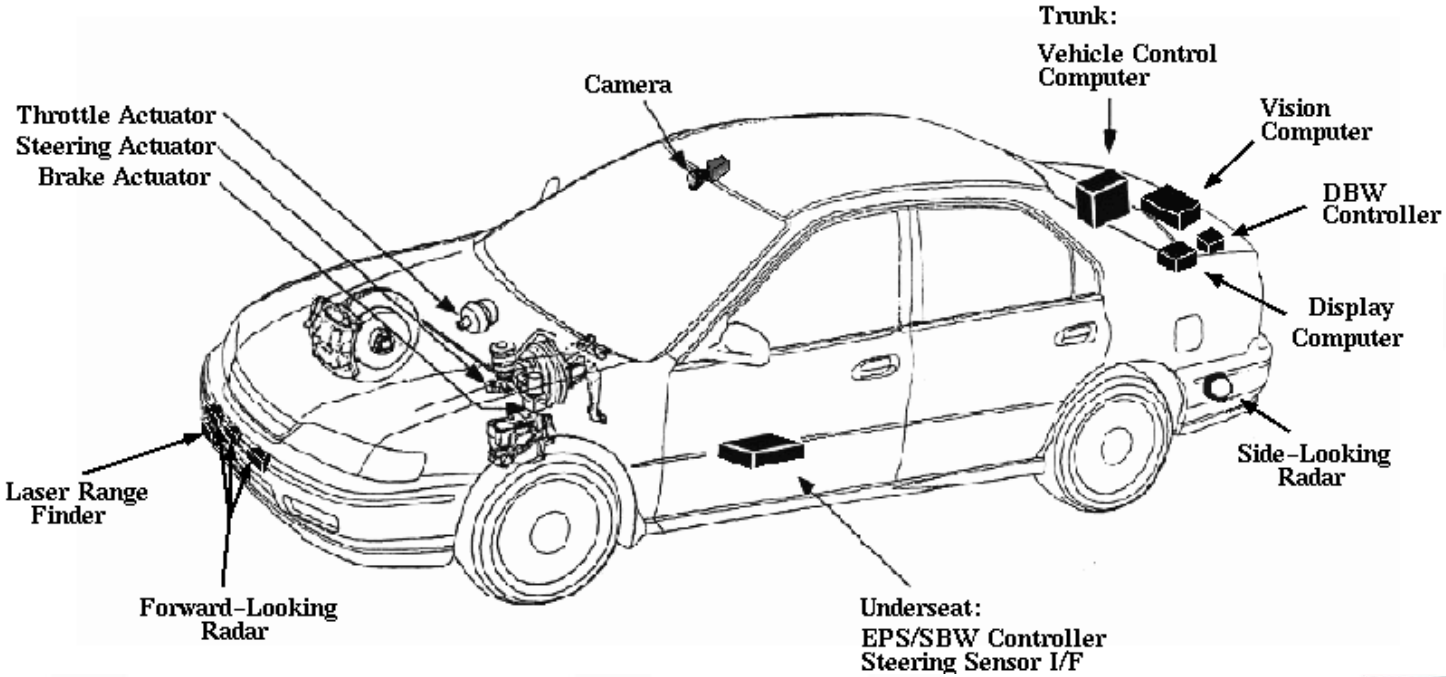


You've seen Microservices before

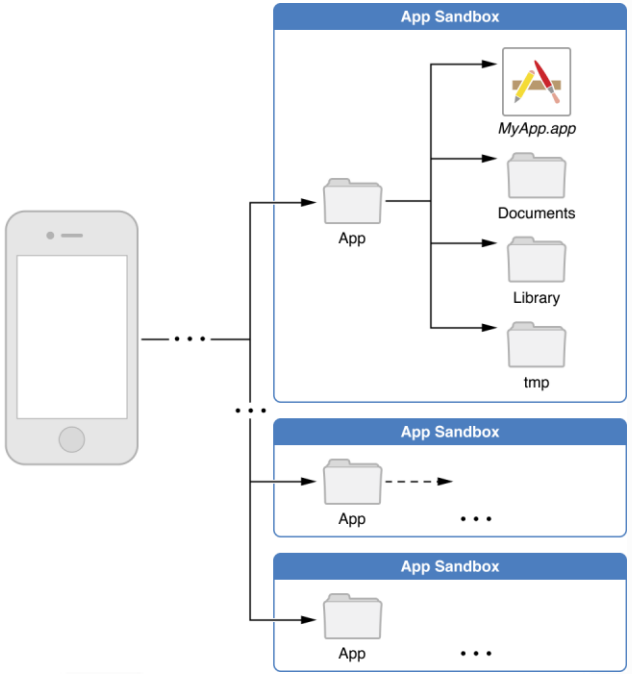


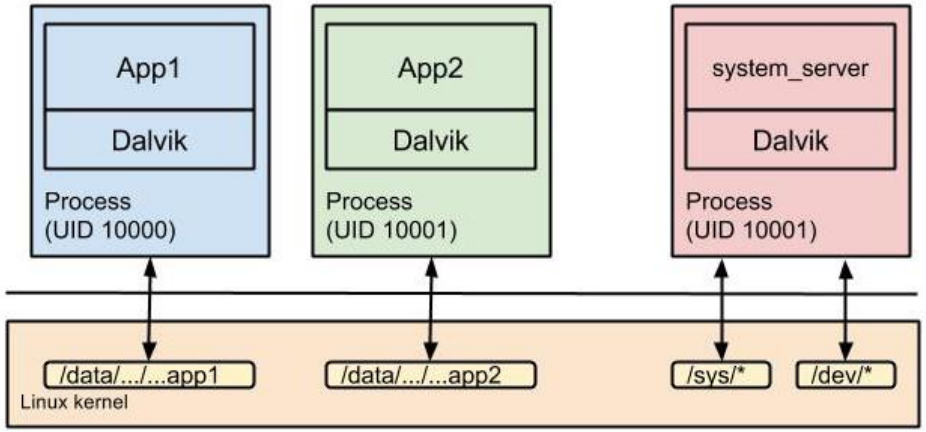
"Worker bees can leave.
Even drones can fly away.
The Queen is their slave."

You've seen Microservices before

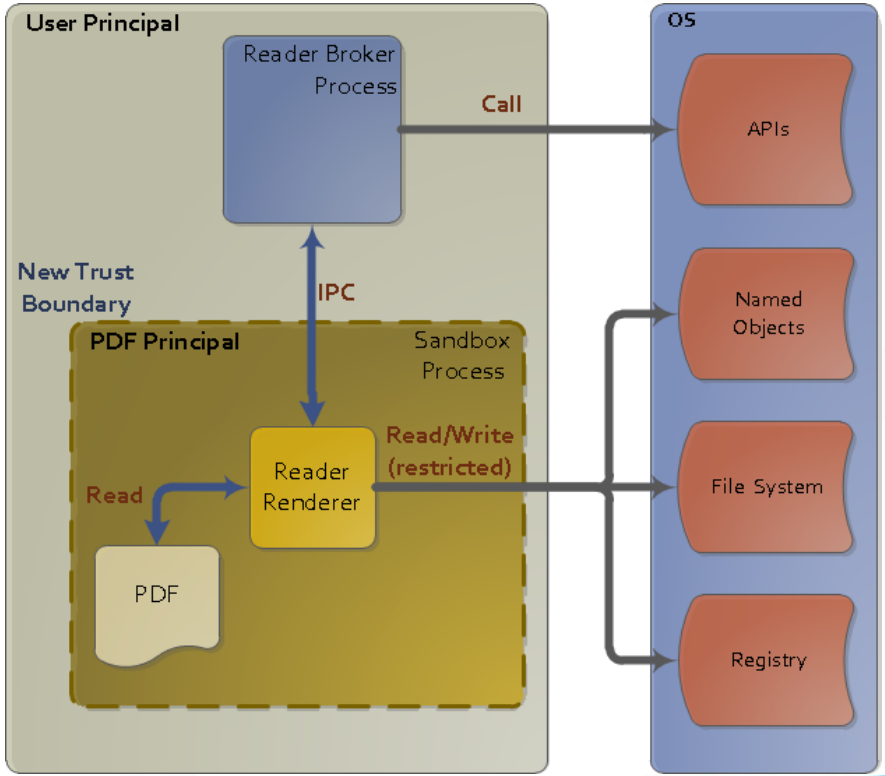


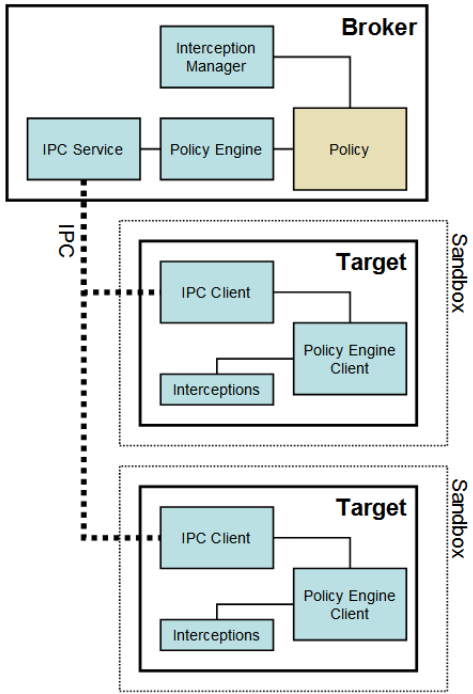
You've seen Microservices before





You've seen Microservices before



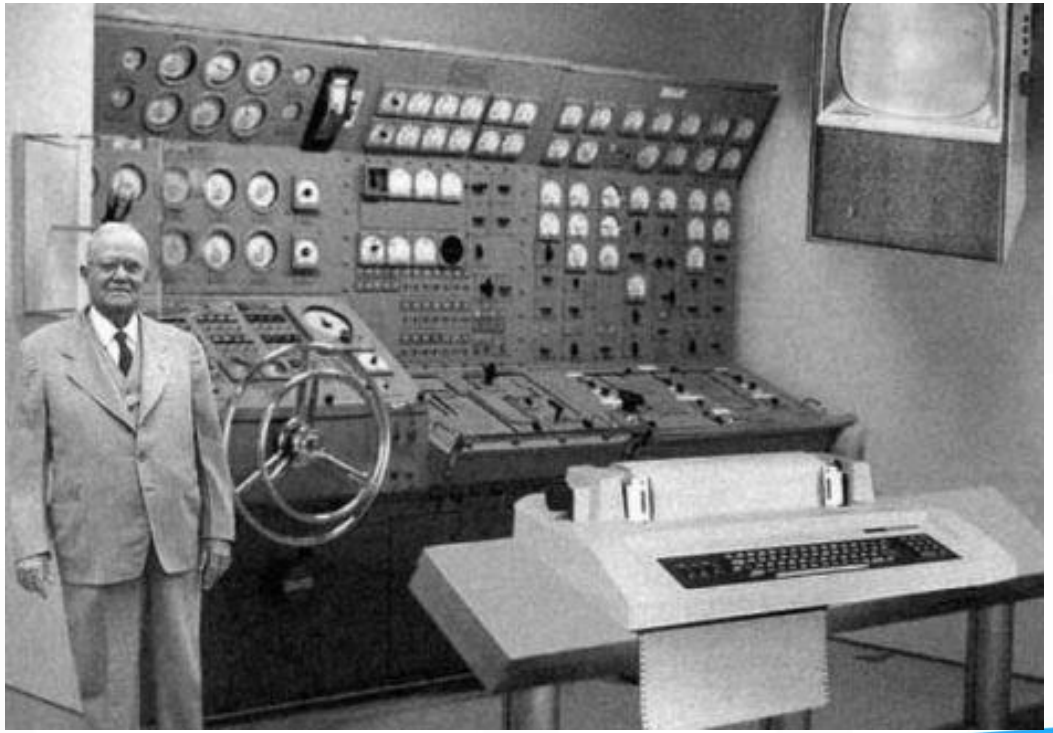


You've seen Microservices before

```
int main() {
   printf("foo\n");
      return 0;
}
```

```
void DoSomething() {
   printf ("foo\n");
}
int main() {
   DoSomething();
      return 0;
}
```

Your Legacy Application



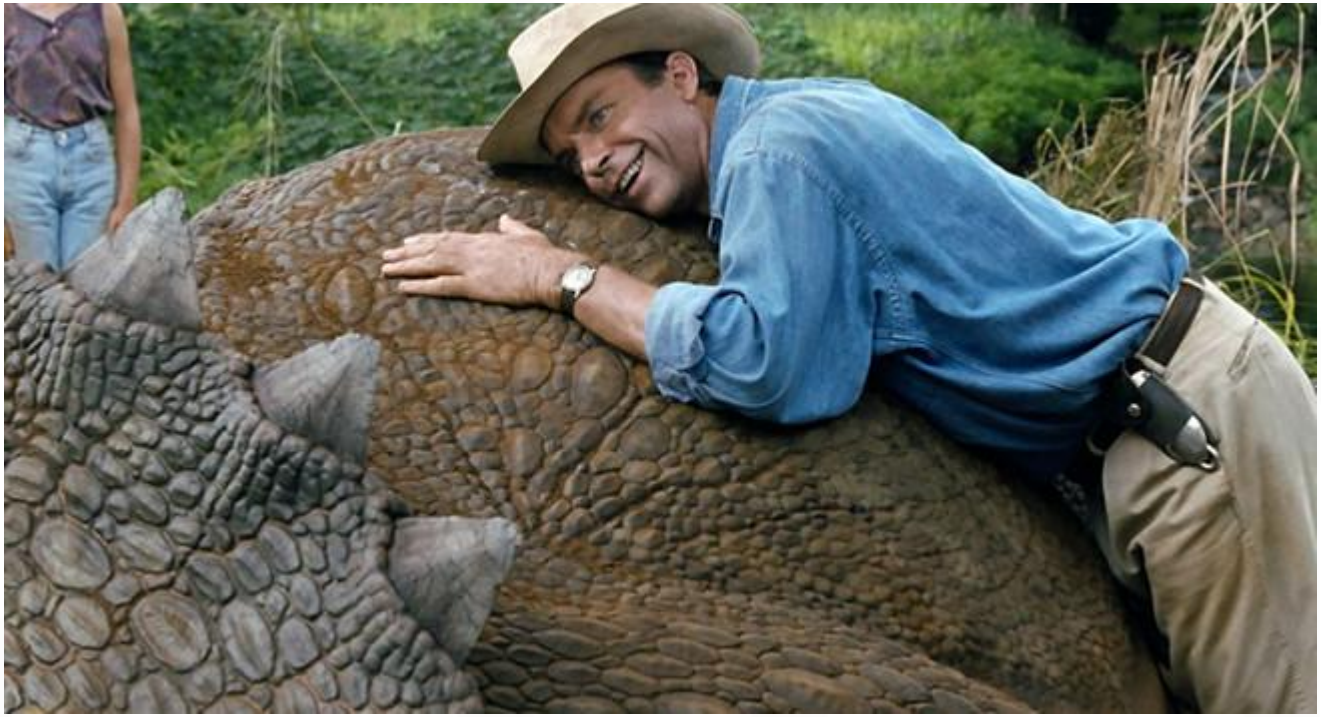
What the sales team sees…

What the developers see

What ops/sysadmins see

What hackers see…

What hackers think



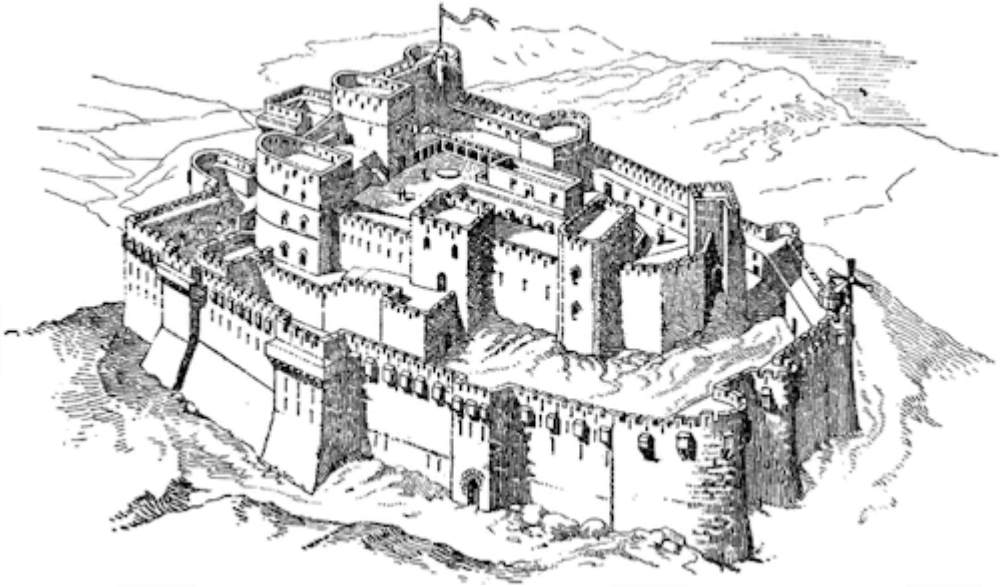
Applied Security Principles

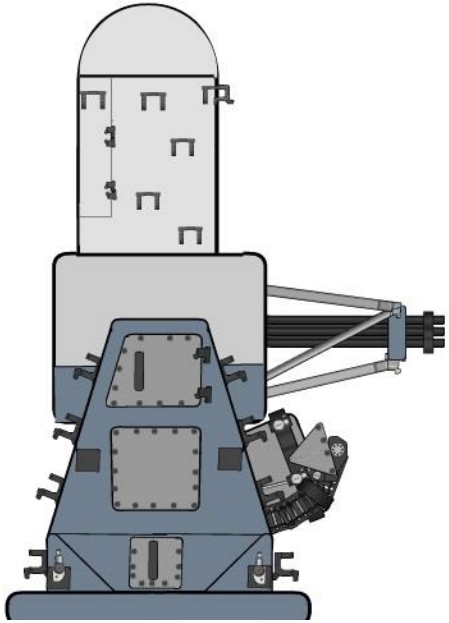
The Principle of Defense in Depth

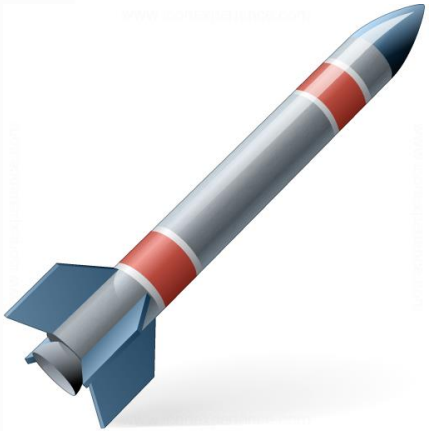
A **layered defense**…

Shrink attack surfaces and present

harden those which remain








U.S.NAVY
U.S.NAVY
U.S.NAVY
U.S.NAVY
U.S.NAVY
U.S.NAVY

The Principle of Defense in Depth

Monolithic applications *make this difficult*

and Microservices *make this easy*

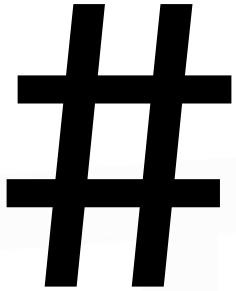
Key: not having a single point of *security* failure

The Principle of Least Privilege

"Absolute power corrupts absolutely"

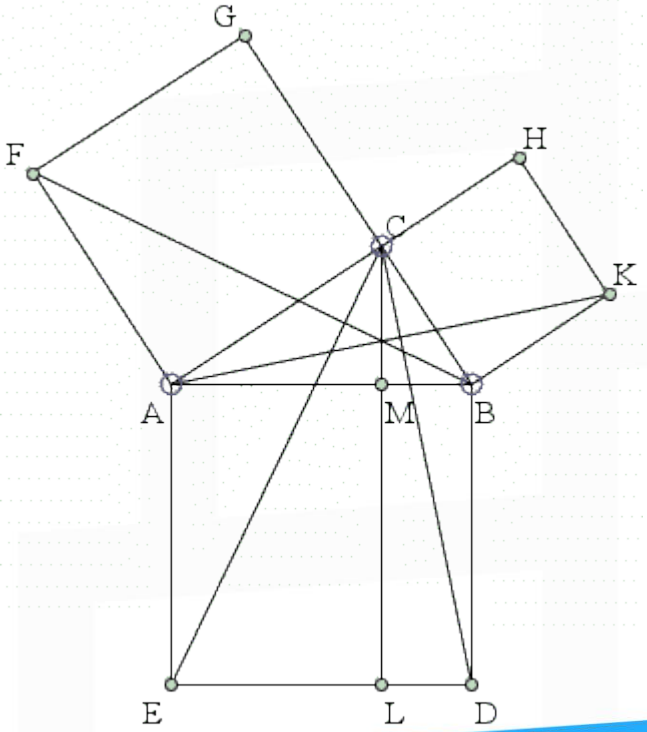
"Avoid running as root" is about shrinking trust boundaries.

Monoliths make this difficult and
Microservices make this easier



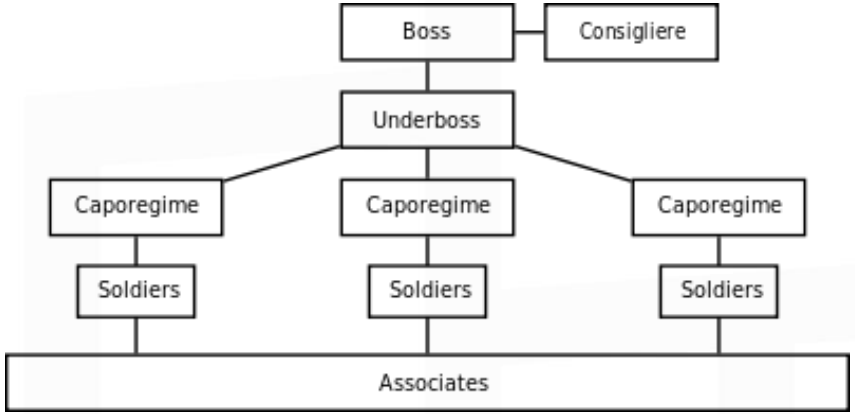
The Principle of Least Surprise

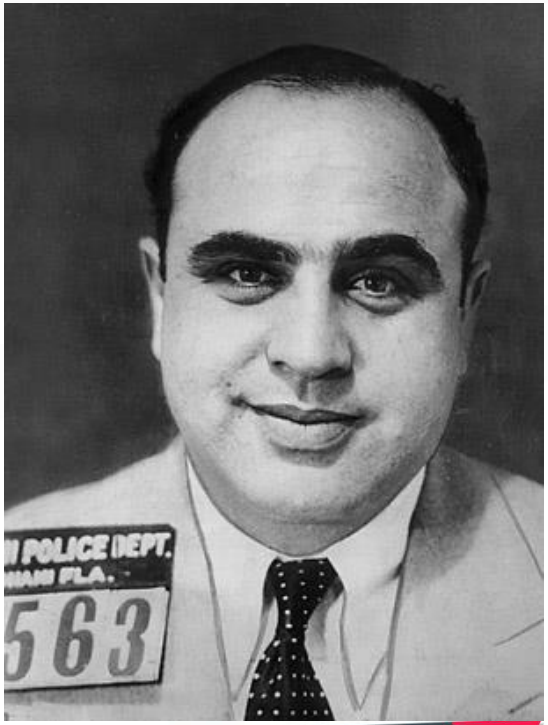
"Sane defaults, Isolate by trust"



The Principle of Least Access

"Access provided on a need to know basis"





Military security, Mafia management,
database connections, defense in depth, …

“Least” is Common to all…

This will be become a theme, *a war against excess and complexity*…

Microservices have complexity “at scale”

Microservices still reduce *overall* complexity

.. Because “Least” is a Good Idea!

1) Establish trust boundaries
2) Identify, minimize and harden attack surfaces
3) Reduce scope and access
4) Layer protections / defenses

See *“Stop Buying Security Prescriptions”* by Justin Schuh of Google

Compare and Contrast

Upsides of Monolith AppSec

Building and Standing it up is a **“known known”**

Architecture is quite **simple**

Existing culture of dev/business/compliance/sales

Downsides of Monolith AppSec

Compromise of a single point often means compromise of the **entire application or network**

Authentication requirements/creds are **global in scope**

Simple to build, but **security is hard to tailor**

Upsides of Microservices AppSec

UNIX model works well

Highly application/function specific containers permit highly application **specific security**

Establishing a Trusted Computing Base (**TCB**)

Upsides of Microservices AppSec

Least privilege and **app-specific security is much easier**

Greatly **reduced attack surfaces**

Independent patching/updates/versions

Downsides of Microservices AppSec

A “Known unknown”

Requires good understanding of application

Legacy applications are not easily adapted

Downsides of Microservices AppSec

May require devops and other culture changes

Complexity breeds insecurity

“Requires” dumb pipes networking architecture

Exploring Architectures

Exploring Real World Compromise

"Imagetragick" **RCE**

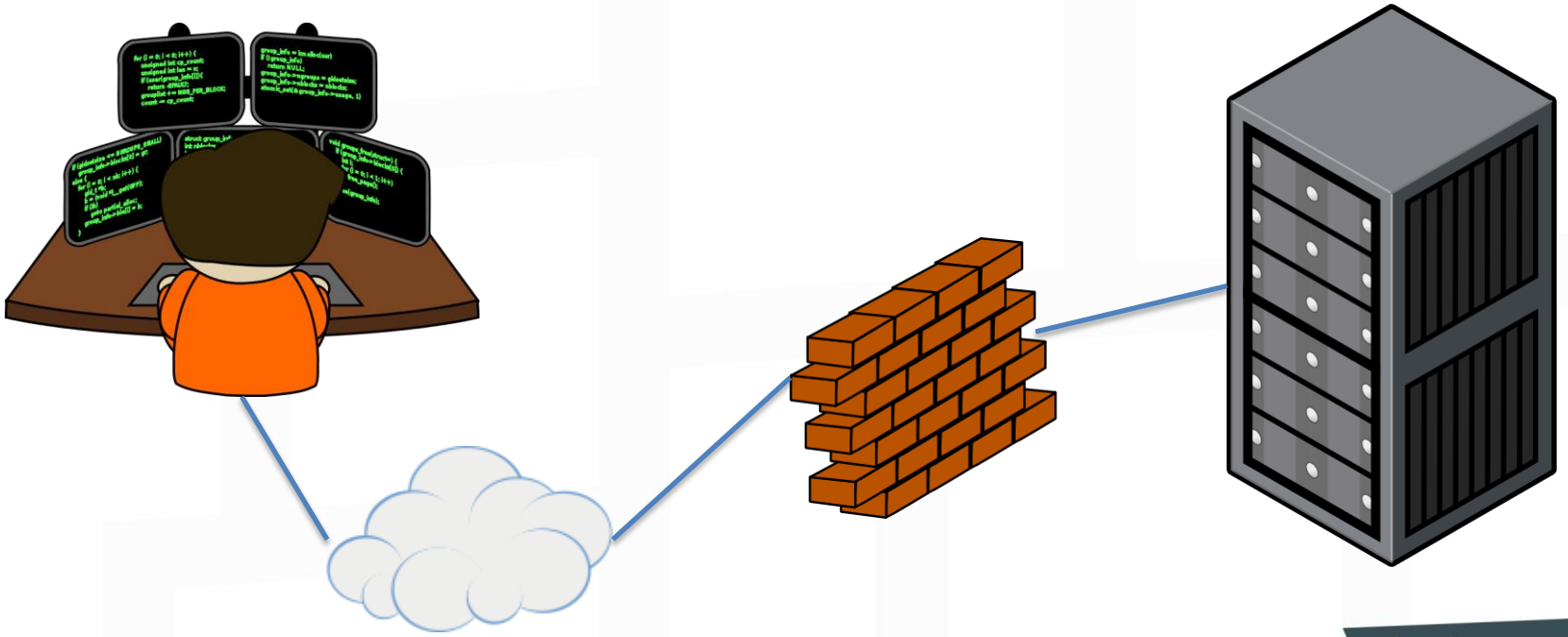
"Shellshock" **RCE**

General **Command Injection**

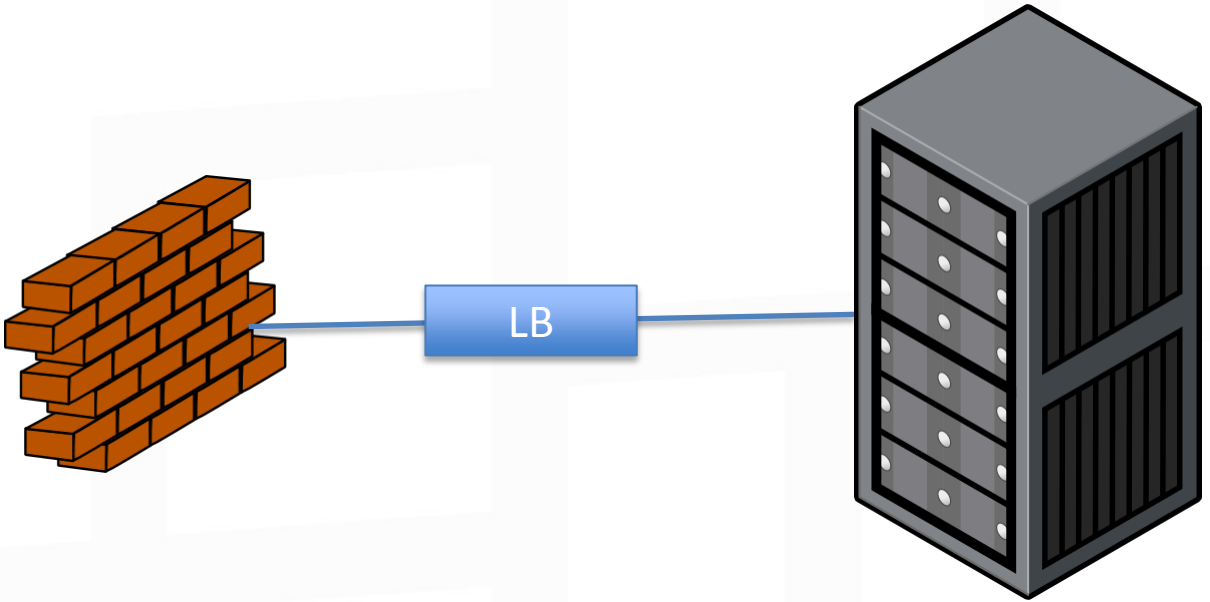
General LFI/**LFR**

Denial of Service

Exploring Real World Compromise



Exploring Real World Compromise



API Web Service

Front-end
Presentation

Back-end
Application

Database

Exploring Real World Compromise

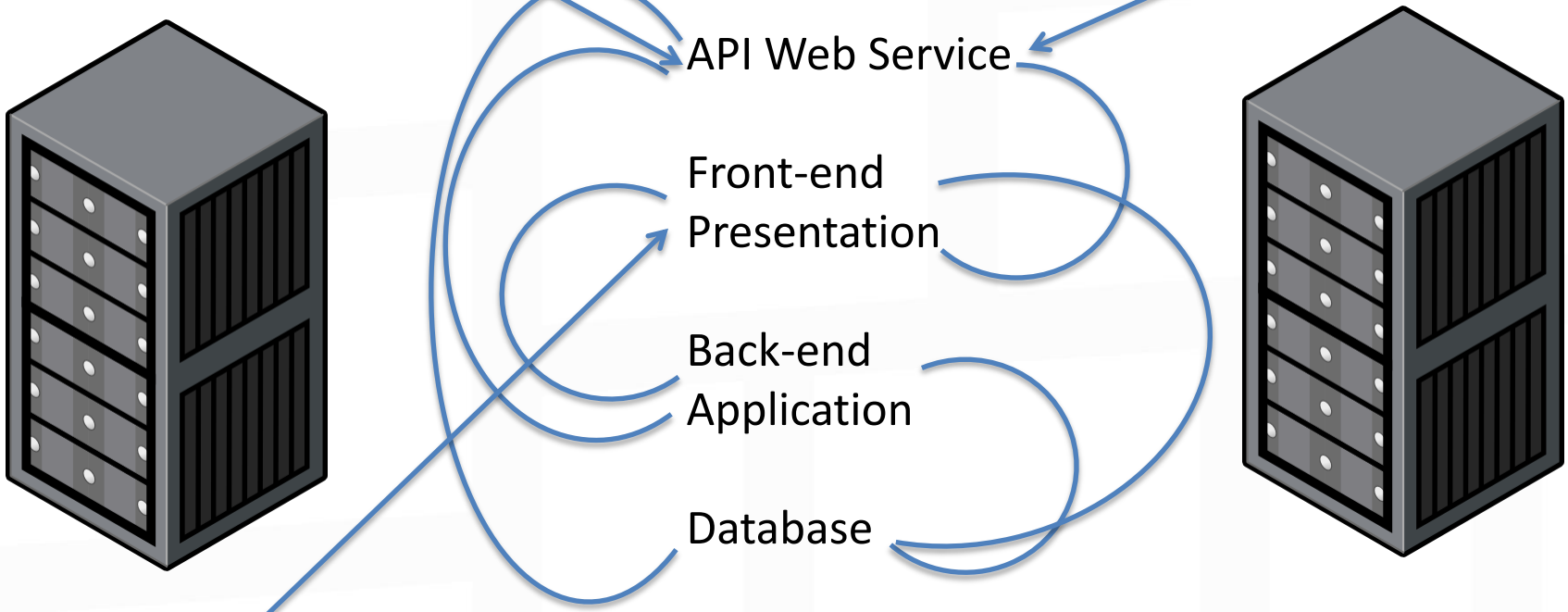
API Web Service

Front-end
Presentation

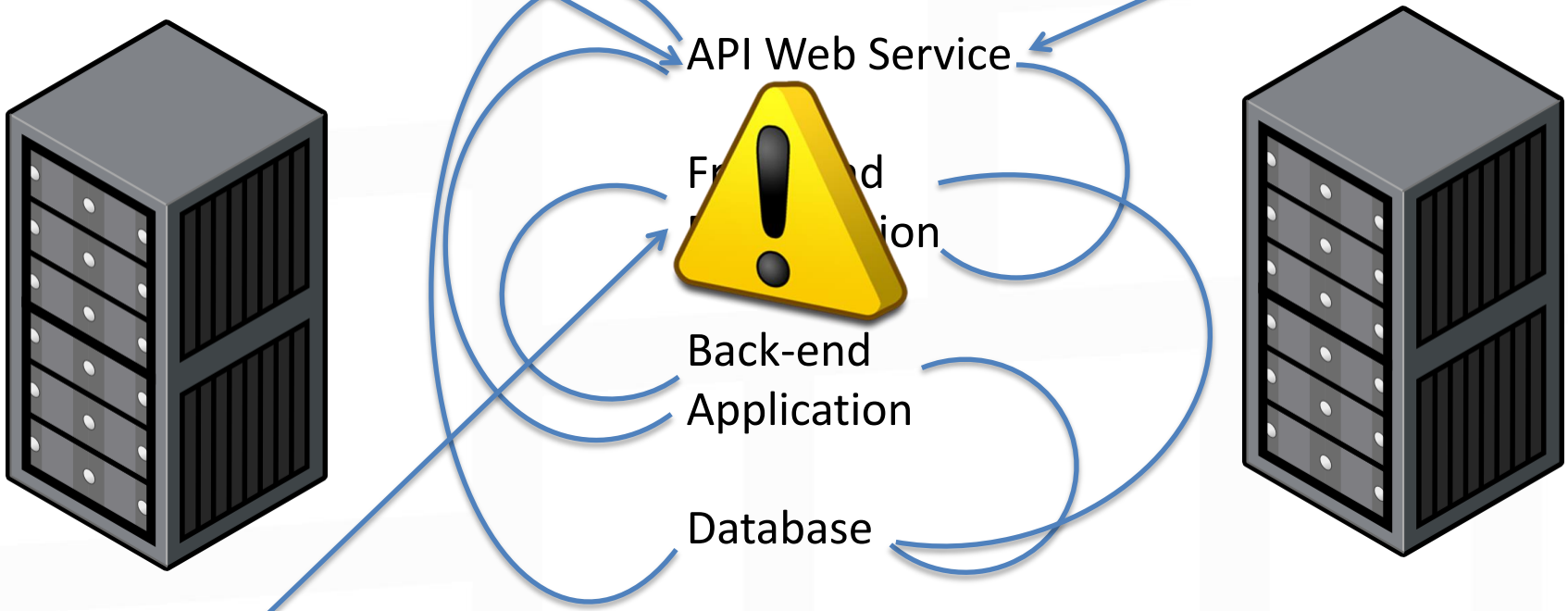
Back-end
Application

Database

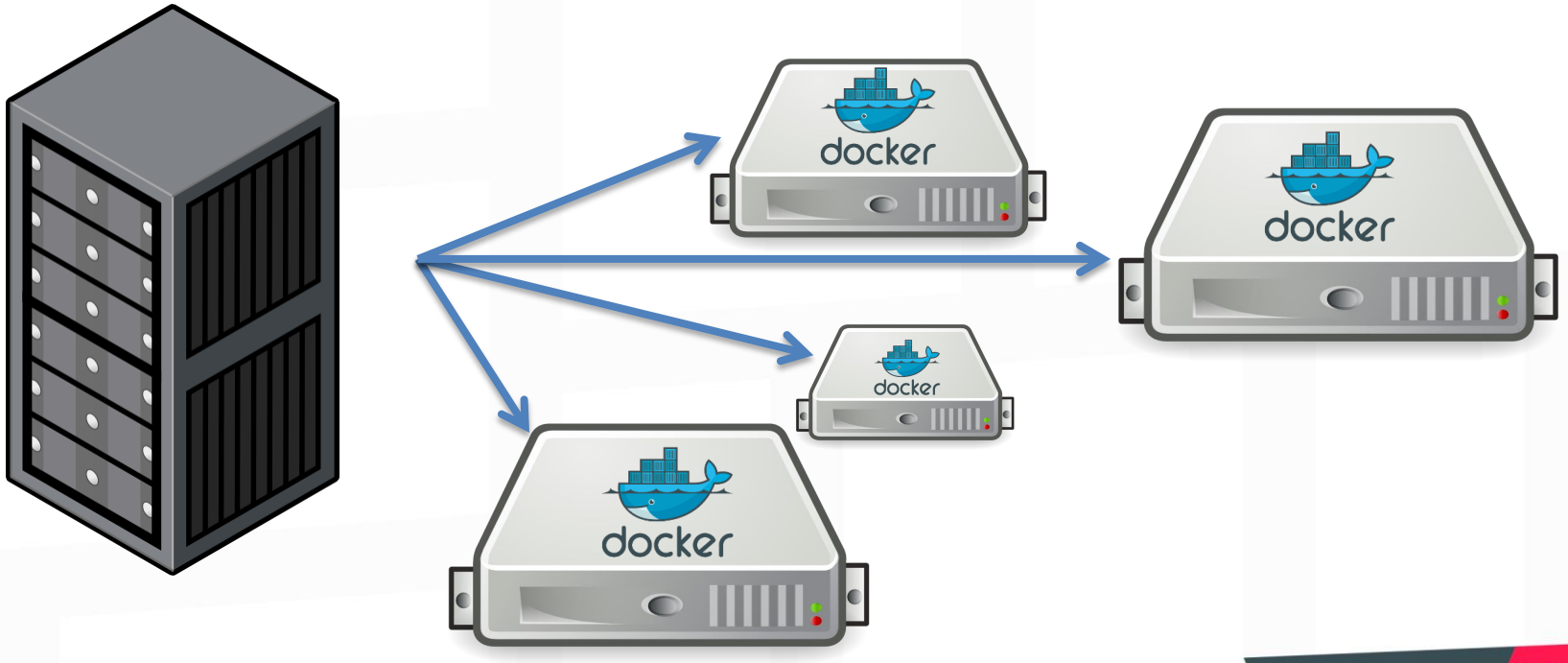
Exploring Real World Compromise



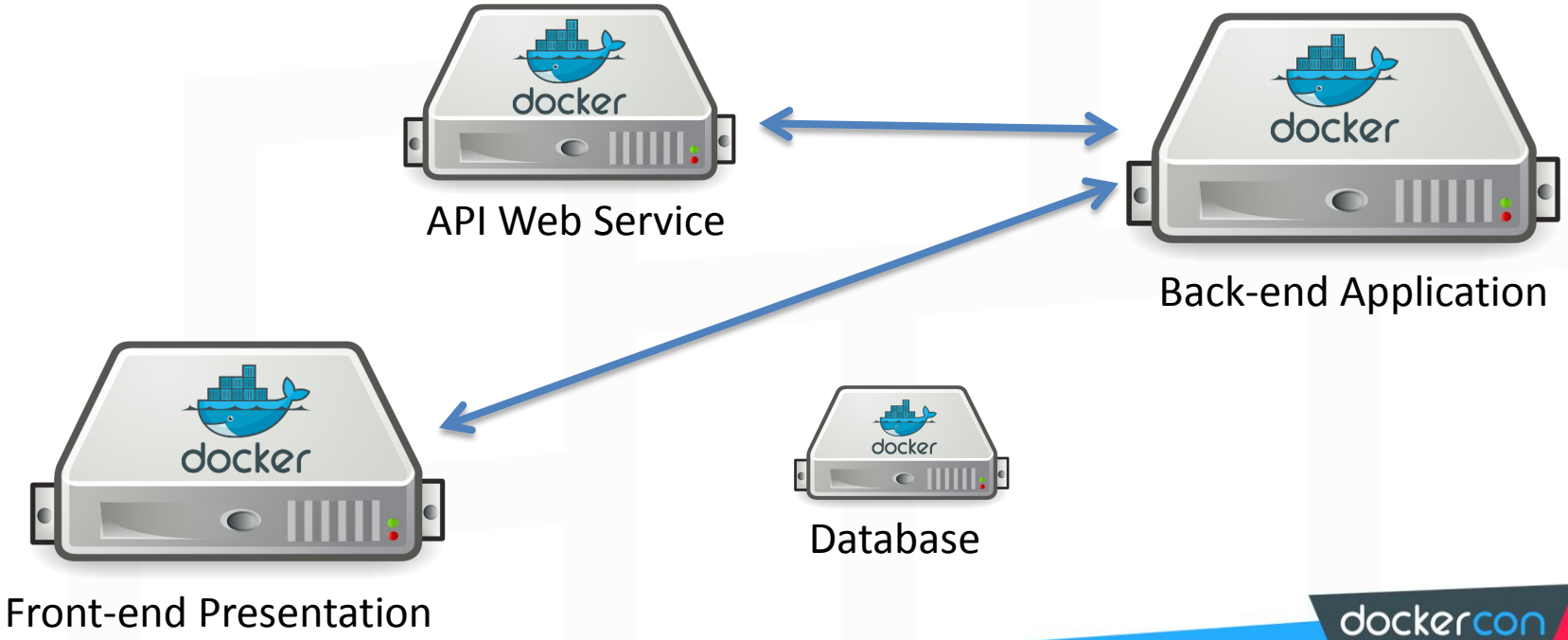
Exploring Real World Compromise



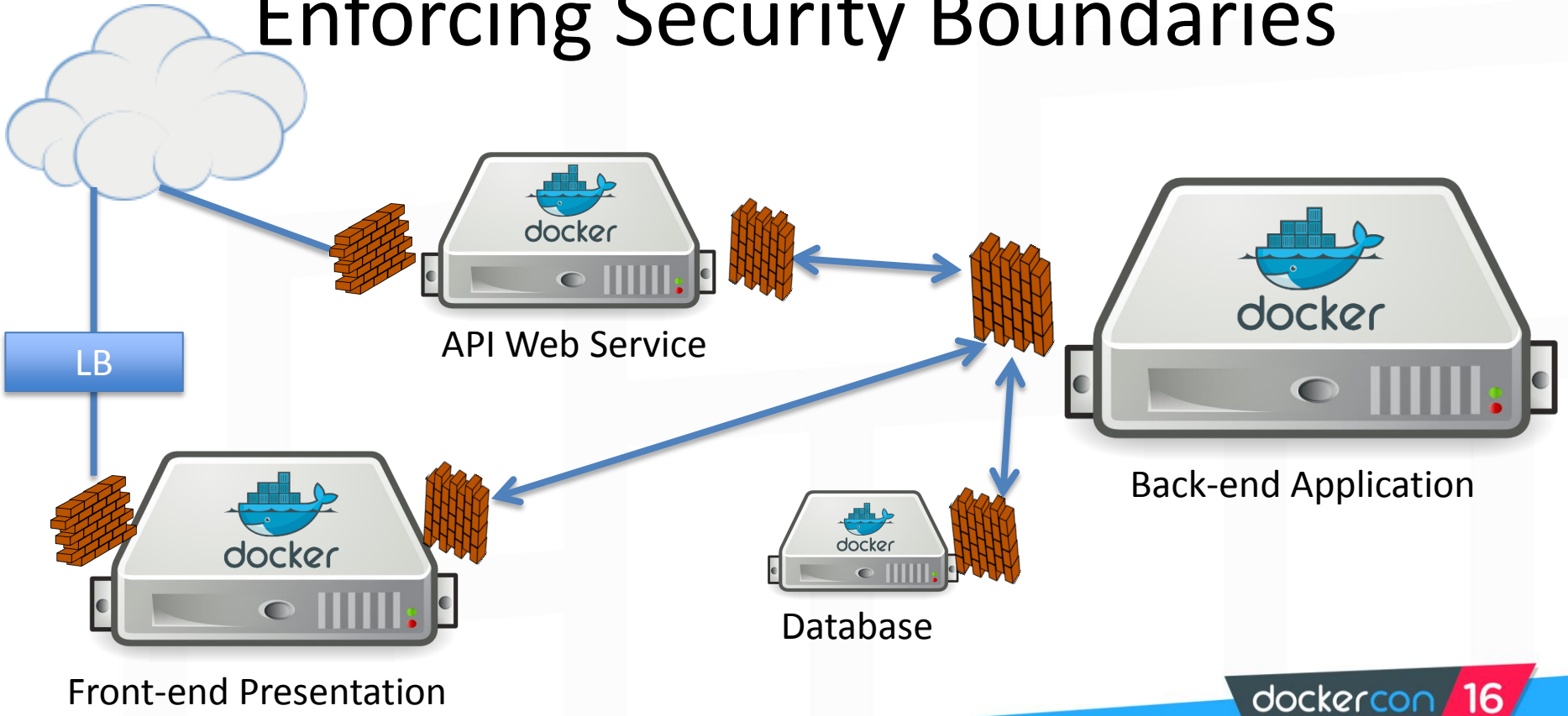
Relationship Status: “Its Complicated”



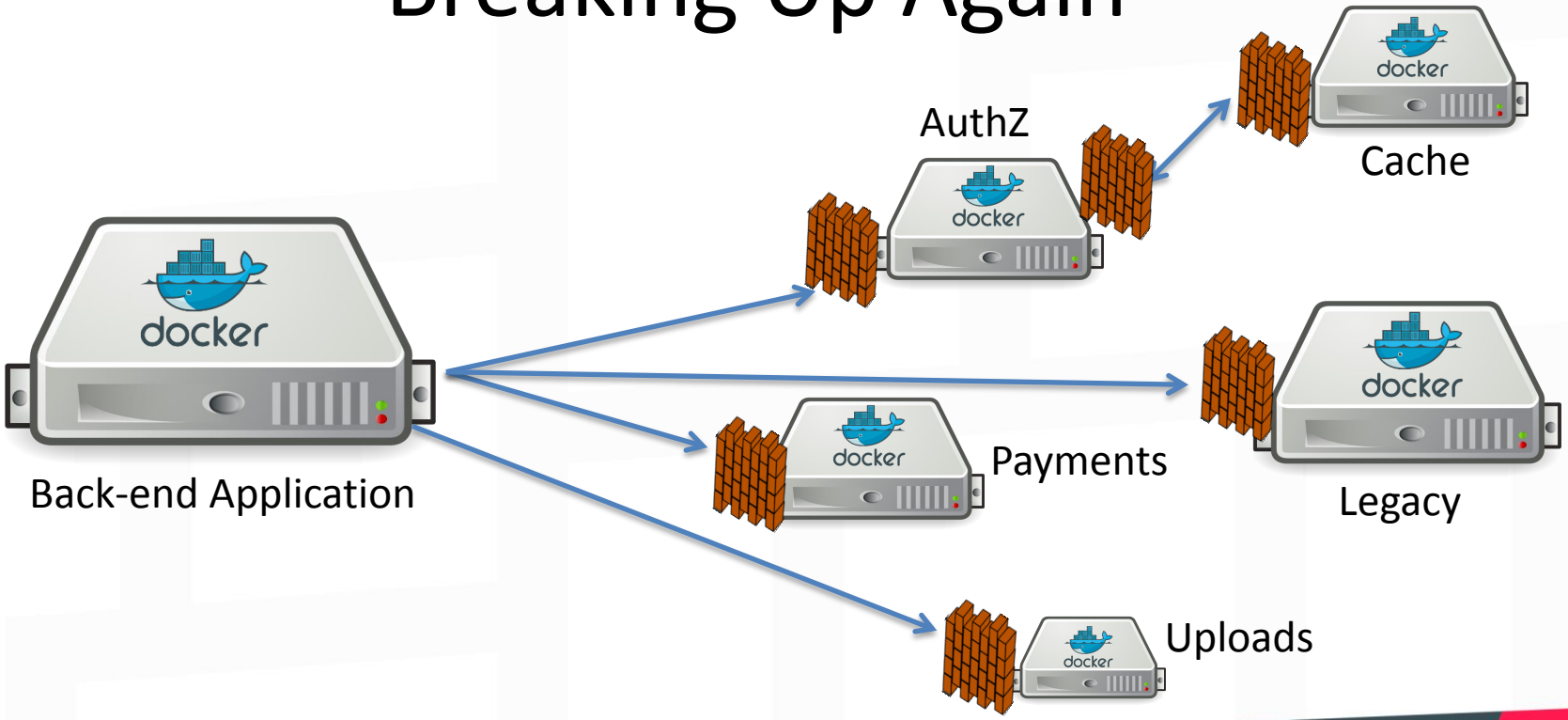
Exploring Real World Solutions



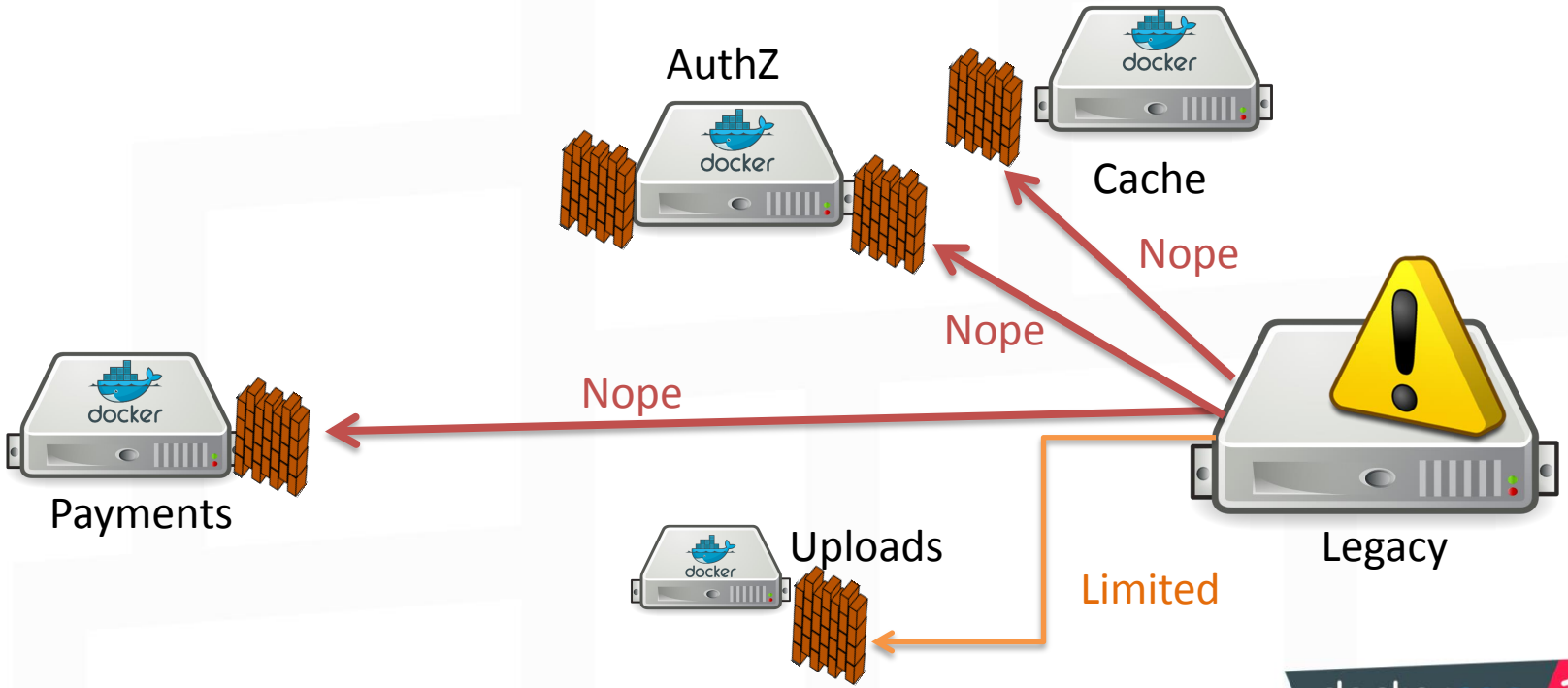
Enforcing Security Boundaries



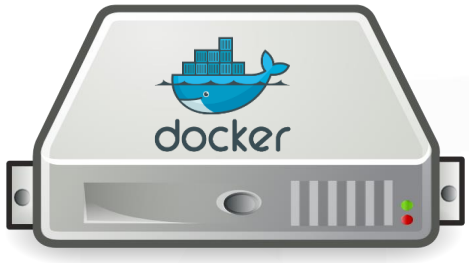
Breaking Up Again



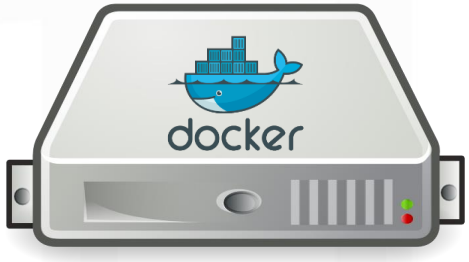
Limit Compromises



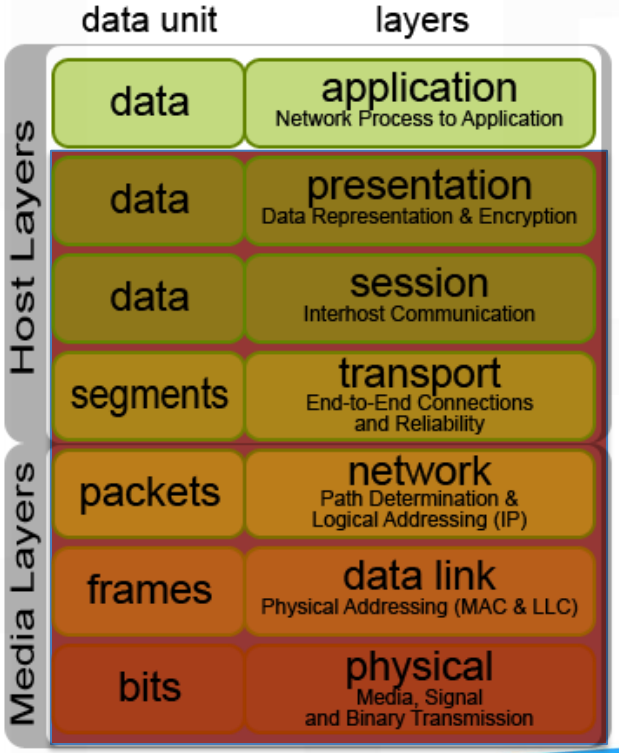
Limit Compromises: OSI Edition

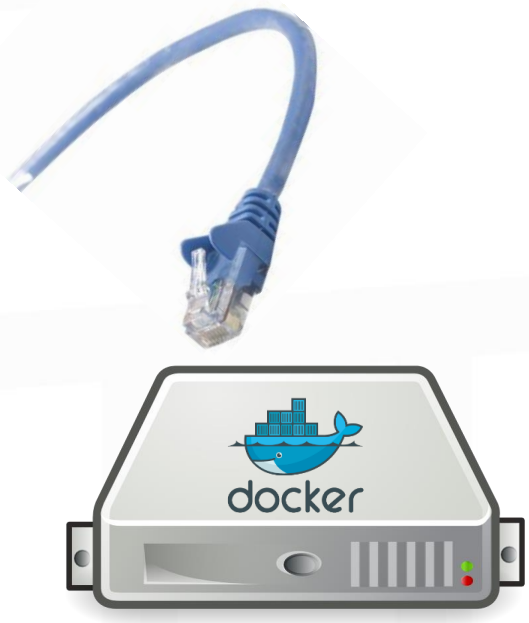


	data unit	layers
Host Layers	data	application Network Process to Application
	data	presentation Data Representation & Encryption
	data	session Interhost Communication
	segments	transport End-to-End Connections and Reliability
Media Layers	packets	network Path Determination & Logical Addressing (IP)
	frames	data link Physical Addressing (MAC & LLC)
	bits	physical Media, Signal and Binary Transmission



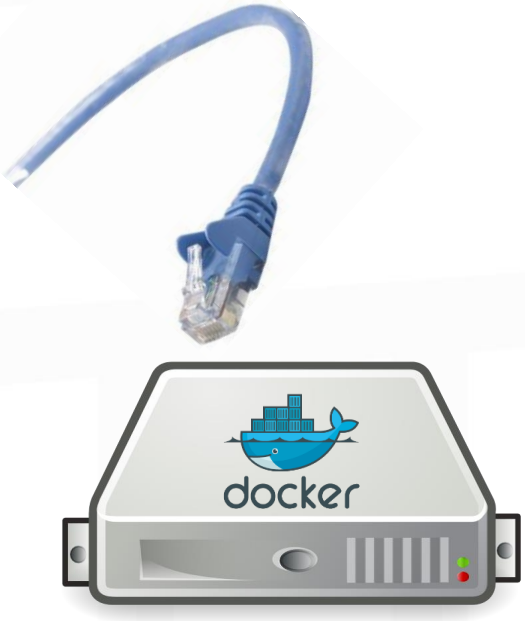
Layer 7 Authentication: Application





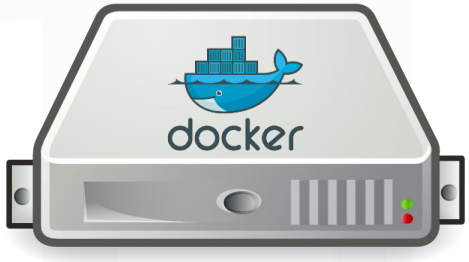
Layer 4/5 (7) Authentication: TLS

	data unit	layers
Host Layers	data	application Network Process to Application
	data	presentation Data Representation & Encryption
	data	session Interhost Communication
	segments	transport End-to-End Connections and Reliability
Media Layers	packets	network Path Determination & Logical Addressing (IP)
	frames	data link Physical Addressing (MAC & LLC)
	bits	physical Media, Signal and Binary Transmission



Layer 3 Authentication: IPSEC

	data unit	layers
Host Layers	data	application Network Process to Application
	data	presentation Data Representation & Encryption
	data	session Interhost Communication
	segments	transport End-to-End Connections and Reliability
Media Layers	packets	network Path Determination & Logical Addressing (IP)
	frames	data link Physical Addressing (MAC & LLC)
	bits	physical Media, Signal and Binary Transmission



Thinking about it another way

What's the attack surface of:

an SSL VPN using username+pw auth

vs.

an SSL VPN using TLS client certs

vs.

an IPSEC-only endpoint using key-based auth ?

Thinking about it another way

Layer 7 Authentication/Encryption? aka XML Encryption or HTTP Header auth

Or Layer 4/5 Authentication/Encryption? aka TLS/mTLS

Or Layer 3 Authentication/Encryption? aka IPSEC

In general…

Layer 7 authentication **should be required**

Less than layer 7 authentication is **highly desirable**

Don’t forget **Authentication != Authorization**

Don’t forget about **transport security**

Containers Map to Microservices

Container Model	Microservices Model
Root Capabilities	Capability based security
Different network namespaces	Overlay networks or SDN
One application per container	One core function per application
…	…

A Few Attacks of Container Systems

Cross Container Attacks on Host

Poisoning Service Discovery

Exploiting Vulnerable Application

Cross Container Attacks on Network

Targeting Docker API Configuration

Exploiting Vulnerable Library

Exploiting Out of Date Docker Engine

Exploiting Host Kernel

Targeting Container Configuration

Attacking Container Management

Pruning The Attack Tree

Application attack? Container itself, capabilities, immutable files, mount flags, MAC, read-only rootfs, defensive coding, etc.

Kernel/syscall exploit ?

Seccomp and kernel hardening

Seccomp weakness?

Manditory Access Control

Pruning The Attack Tree

Compromised kernel/host OS?

Network hardening, isolation on trust, logging, least privilege, least access

Compromised employee credentials?

Least access, least privilege, key-based authentication

Microservices in Docker or runc

Why Docker / runc ?

Helps with standardization, testing, security

Strong security defaults (seccomp, caps) and options (user namespace)

App containers help follow and develop the Microservice model

Why use runC for MS?

Lightweight and minimal!

Open Container Initiative

Docker’s libcontainer

Why not runC ?

Lightweight and minimal/manual

Weak documentation/examples/debugging

Warning: Apparmor **not a default build tag** and **Apparmor support / User namespace support not enabled by default**

Security starts with the base OS

Minimal Distro (CoreOS/RancherOS/etc)?

Plain old Ubuntu Server?

Clear Linux?

… Unikernel?

Minimal: Distro

CoreOS

alpine Linux

PROJECT ATOMIC

RANCHEROS

Security starts with the base OS

It’s important to understand how a distribution:

handles updates,

binary package compilation,

security defaults (MAC),

default kernel options,

sysctl settings, ….

Minimal: Kernel

Common sense configuration

make menuconfig: configuration, modules

grsecurity

Minimal Container?

… Why stop at shrinking just the host?

Don’t we want least access? Less updates?

Minimal: Container Images

docker-nginx

```
FROM debian:jesse
```

mongo

```
FROM debian:wheezy
```

redis

```
FROM debian:jesse
```

node

```
FROM buildpack-deps:jessie
```

Minimal: Container Images

So what exactly is `FROM debian:jesse` ?

```
FROM scratch
ADD rootfs.tar.xz /
CMD ["/bin/bash"]
```

= 137 MB

Minimal: Container Images

So what about `FROM debian:wheezy` ?

97 MB!

So what about Jenkins? ElasticSearch? Postgres?

`FROM debian:jesse`

Not Minimal Enough!

Even **before an apt-get**, there are still tons of libraries, executables, **perl**, even language files our process doesn't need!

That's more patching, disk space, **attack surface and post-exploitation utilities** we don't need

General idea for **Docker**

$ cat Dockerfile

FROM scratch

ADD stuff

CMD [“/stuff”]

$ docker build

General idea for **runC**

$ mkdir rootfs

$ **cp stuff rootfs/**

$ runc spec

$ sudo runc run foo

Reality for **runC (and Docker)**

$ ldd /path/to/program

$ mkdir –p rootfs /usr /lib # … etc

$ **cp /path/to/program rootfs/**

$ **cp /path/to/lib.so.0 rootfs/lib/**

$ ***cross fingers***

Minimal Container Images: Go + Docker

Create The Smallest Possible Docker Container

By Adriaan de Jonge | July 4, 2014 | docker, golang | 30 Comments

When you are playing around with Docker, you quickly notice that you are downloadin
of megabytes as you use preconfigured containers. A simple Ubuntu container easily e
and as software is installed on top of it, the size increases. In some use cases, you do n
everything that comes with Ubuntu. For example, if you want to run a simple web serv
there is no need for any tool around that at all.

http://blog.xebia.com/create-the-smallest-possible-docker-container/

Building Minimal Docker Containers for Go Applications

2015-04-23

by Nick Gauthier | 28 Comments

Development

There are several great official and community-supported containers for many programming languages, including Go, but these containers can be quite large. Let's walk through a comparison of methods for building containers for Go applications, then I'll show you a way to statically build Go apps for containerization that are extremely small.

https://blog.codeship.com/building-minimal-docker-containers-for-go-applications/

Golang wiki server example

$ ls -l rootfs
rootfs/wiki
rootfs/usr/lib/libpthread.so.0
rootfs/usr/lib/libc.so.6
rootfs/lib64/ld-linux-x86-64.so.2

$ du –hs rootfs

7.3M

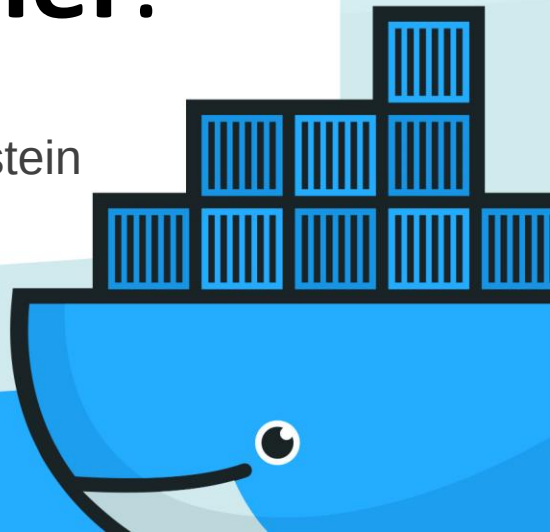
Minimal Container Images: Nginx + Docker

Roughly 32 libs + /usr/sbin/nginx + a few config files and dirs…

= 15 MB!

“Make things **as simple as possible, but not simpler.**”

— Albert Einstein



nanosleep(2)

Minimal (hardened) Host OS: ✔

Minimal (hardened) Kernel: ✔

Minimal (hardened) Root FS: ✔

The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments

Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor,
S. Jeff Turner, John F. Farrell
tos@epoch.ncsc.mil
National Security Agency

Abstract

Although public awareness of the need for security in computing systems is growing rapidly, current efforts to provide security are unlikely to succeed. Current security efforts suffer from the flawed assumption that adequate security can be provided in applications with the existing security mechanisms of mainstream operating systems. In reality, the need for

reality, operating system security mechanisms play a critical role in supporting security at higher levels. This has been well understood for at least twenty five years [2][54][39], and continues to be reaffirmed in the literature [1][35]. Yet today, debate in the research community as to what role operating systems should play in secure systems persists [11]. The computer industry has not accepted the critical role of the operating system to security, as evi-

Mandatory Access Control

MAC as a Linux Security Module (LSM)- **AppArmor**, **SELinux**, etc. **Grsecurity RBAC** is also nice

OSX uses **TrustedBSD**, Microsoft has **MIC**

Mandatory Access Control

Default Docker AppArmor policy is very good

Monolithic MAC policy must include a large number of files, permission grants, complexity

Microservices *again* allow for specific security

Nested AppArmor

```
Profile /bin/bash {
      /bin/cat Px,
      profile /bin/cat {
        /etc/hosts r,
      }
}
```

Simplified Example!

Going back to Golang HTTPd PoC

```
Profile /wiki {
      network inet stream,
      /usr/lib/* mr,
      /lib64/* mr,
      /wiki mixr,
}
```

Real Example!

Custom AppArmor Profiles

Generating profiles with **aa-genprof**

Understanding/using application is key, **profiling is still required**

Custom AppArmor Profiles

Common mistakes/problems:

- Providing too much access
- Dealing with wildcards
- Path-based ACLs

Custom AppArmor Profiles

Easier Mode for Docker/runC: **Using Bane for profiles** by Jessie Frazelle

Hard Mode: **FullSystemPolicy**

Avoid deny lists

AppArmor Profile Gotchas

Profiles must be loaded by AppArmor first

Abstractions may be more verbose than you would like

Exercise your app is key, run unit/QA/regression tests

Profiling is difficult within a container itself

Seccomp BPF

Seccomp Profiles

MAC is vulnerable to kernel attacks… **kernel is huge attack surface**

Seccomp **default filter** enabled in Docker Engine >= 1.10 !!

Seccomp is widely used in security, recently now with **BPF**

Putting the BPF into Seccomp BPF

General BPF pseudocode:

1) Check architecture

2) Get syscall number

N) ALLOW or DENY syscall

N+1) KILL, TRAP, TRACE, ALLOW process

Why Custom Profiles?

Default seccomp filter permits 304 syscalls :/

Minimal applications need minimal syscall set

We want to practice least privilege!

Methods for Generating Seccomp Profiles

1) strace/ltrace

2) Kernel help (sysdig or systemtap)

3) Auditd/auditctl

4) Seccomp itself w/ SECCOMP_RET_TRACE and PTRACE_O_TRACESECCOMP

Seccomp Profiles *using strace*

strace via `ptrace(2)`

Weird timing bugs can occur, count will miss some calls

Tracing outside of Docker should be fine

Some programs just don’t want to be traced

strace (and also ltrace –S)

```
$ strace -f /bin/ls
execve("/bin/ls", ["ls"], [/* 21 vars */]) = 0
brk(0)                                  = 0x8c31000
mmap2(NULL, 8192, PROT_READ, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0xb78c7000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No
such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=65354, ...}) = 0
…
```

Seccomp Profiles *using sysdig*

Sysdig is cool! *Really* cool

Requires a kernel module :/

But it’s really cool!

sysdig

```
$ sysdig evt.type=open and proc.name=niginx


$ sysdig -p “%evt.type” proc.name=niginx

$ sysdig -p “%evt.type” | sort -u > \
                            nginix.syscalls.txt
```

Seccomp Profiles *using auditd*

Auditd is a nice system for… auditing

Requires a daemon and complex-ish config

Very cool power built-in but not ideal for this!

(Could collect calls over time)

Seccomp Profiles *using Seccomp*

Basic C example by Kees Cook and Will Drewry:
(https://outflux.net/teach-seccomp/)

- Profiling via **SECCOMP_RET_TRAP is slow** going

Using **SECCOMP_RET_TRACE is better**!

Seccomp Profiles *using Seccomp*

But why?

ALLOW, TRAP, TRACE, KILL, ERRNO are limited,
all of them block the call *except for TRACE*

(and default ALLOW is not a good model)

Seccomp Profiles *using Seccomp*

- Create a given syscall whitelist
- Set the default action to `SECCOMP_RET_TRACE`
- use `ptrace(2)` with `PTRACE_O_TRACESECCOMP`
- Log appropriate info
- Repeat until no soft-failures!

Ptrace PoC

```
$ seccomp-trace -f filter.txt /path/to/program
...
called sycall # 157 : prctl
called sycall # 157 : prctl
SECCOMP WOULD BLOCK 56
called sycall # 56 : clone
SECCOMP WOULD BLOCK 14
SECCOMP WOULD BLOCK 13
called sycall # 14 : rt_sigprocmask
called sycall # 13 : rt_sigaction
```

General Seccomp Pitfalls

Profiles are **fragile**!

Profiles are **architecture dependent**- limiting container portability

Seccomp in Docker

Default (large) whitelist or:

```
--security-opt seccomp=<profile>
```

```
"seccomp": {
    "defaultAction": "SCMP_ACT_KILL",
    "architectures": [ "SCMP_ARCH_X86" ],
        "syscalls": [
                    {   "name": "getcwd",
                   "action": "SCMP_ACT_ALLOW"
                    }
            ]
}
```

Seccomp in runC

Part of the default build!

```
"seccomp": {
    "defaultAction": "SCMP_ACT_KILL",
    "architectures": [ "SCMP_ARCH_X86" ],
        "syscalls": [
                    {   "name": "getcwd",
                   "action": "SCMP_ACT_ALLOW"
                    }
            ]
}
```

Seccomp notes

Remember that **seccomp filters are not sandboxes**

Libseccomp is nice and has Go bindings

see also: oz sandbox train mode by Subgraph OS

Be careful implementing it from scratch, blacklists /denylists are risky! (don't allow ptrace!)

Seccomp Pro Tips for Docker/runC

https://github.com/konstruktoid/Docker/blob/master/Scripts/genSeccomp.sh can save time or JSON pain !

You can always confirm Seccomp by looking in:

```
grep Seccomp /proc/<PID>/status
```

nanosleep(2)

Minimal (hardened) Host OS: ✔

Minimal (hardened) Kernel: ✔

Minimal (hardened) Root FS: ✔

Minimal AppArmor/SELinux: ✔

Minimal Seccomp whitelist: ✔

Security Recommendations

High Security Docker Microservices

- **Enable User namespace**
- Use app specific AppArmor if possible
- Use app specific Seccomp whitelist if possible
- Harden host system
- Restrict host access
- Consider network security

High Security runC Microservices

- **Configure user namespace**
- Enable AppArmor + use app specific if possible
- Use app specific Seccomp whitelist if possible
- Harden host system
- Restrict host access
- Consider network security

The Problem of Managing Secrets

Avoid environment variables/flat files

- *Why?* … because exec**ve**()

Use temporary secret “injection”

- Temp bind mount, load into mem-only, unmount

Vault | Keywiz

Immutable Containers also Help

Extending the basic idea of non-executable memory pages or OpenBSD's W^X

- data-only containers, app-only containers

Frustrating vulnerability exploitation and limiting system post-exploitation

Networking and AuthN/AuthZ

TLS. TLS. TLS.

It's 2016 and all network traffic should be encrypted (and authenticated) *Just do it.*

Authentication/Access at the lowest possible layer. **Network access controls go against "dumb pipes"** but they're important!

Networking and AuthN/AuthZ

But I *want* dumb pipes!?

Then use SDN / Overlay networks on top…

Beware of cross-container networking / multi-tenant risks with a single bridge interface

Other Security Recommendations

Have a specification / *Dock*umentation

Generate **application-specific and overall threat models**

Don’t forget about **application security** itself (containers and MS can’t help if your app itself is still vulnerable)

Other Security Recommendations

Orchestration / Service Discovery things… Use security

Generate **application-specific and overall threat models**

Don’t forget about **application security** itself (containers and MS can’t help if your app itself is still vulnerable)

Other Security Recommendations

Microservice logging and accountability is important, **collect and keep logs centrally** (and actually look at them once in while)

Security is much easier to do if you make it part of your lifecycle in the beginning *“build it in don’t patch it on”*

Checkout my whitepaper for more general container security info and more info on many of the topics covered here.

Thank you!

Aaron.Grattafiori /at/ nccgroup dot trust
@dyn___