NCC Group Whitepaper

Understanding and Hardening Linux Containers

June 29, 2016 – Version 1.1

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.

Table of Contents

nccgroup

1 Introduction . . . 5
1.1 Motivation . . . 6
1.2 Virtualization Background . . . 7
1.3 Benefits of An OS-Virtualization System . . . 10
1.4 Drawbacks of an OS-Virtualization system . . . 11
2 Linux Containers Overview . . . 13
2.1 A Brief History of OS Containers . . . 13
2.2 Linux Containers: where are they now? . . . 13
2.3 Prior Art: Linux Container Security, Auditing and Presentations . . . 15
2.4 TL;DR Linux Containers . . . 18
3 Namespaces . . . 20
3.1 Namespaces Background . . . 20
3.2 Namespaces Implementation . . . 20
3.3 Mount Namespace . . . 21
3.4 IPC Namespace . . . 21
3.5 UTS Namespace . . . 22
3.6 PID Namespace . . . 22
3.7 Network Namespace . . . 23
3.8 User Namespace . . . 26
4 Control Groups . . . 27
4.1 Cgroups Background . . . 27
4.2 Working with Vanilla cgroups . . . 28
4.3 Containers and cgroups . . . 29
4.4 Future of cgroups . . . 29
5 Capabilities . . . 30
5.1 Capabilities Background . . . 30
5.2 Additional Introductory Resources . . . 31

5.3 Understanding Capabilities 31
5.4 Exploring Capabilities 37
5.5 Capabilities and User Namespaces 39
5.6 Capability Defaults In Modern Containers 40
5.7 A World Without Root 42

6 Configuration and Basic Use 43

6.1 LXC 43
6.2 Docker 45
6.3 CoreOS Rocket 46

7 Understanding Container Threats 49

7.1 The Linux Kernel Itself 49
7.2 Exploring Container Threats 51
7.3 LXC Specific Threats 61
7.4 Docker Specific Threats 62
7.5 CoreOS Rkt Specific Threats 63
7.6 Indirect or Unexpected Threats 64

8 Recent Security Advancements 66

8.1 The User Namespace 66
8.2 Mandatory Access Control 69
8.3 Syscall Filtering with Seccomp 74

9 LXC, Docker and CoreOS Rocket 82

9.1 LXC 82
9.2 LXC Background 82
9.3 LXC Components 83
9.4 Brief LXC Security Analysis 83
9.5 Docker 85
9.6 Docker Background 85
9.7 Docker Components 86

9.8 Brief Docker Security Analysis 87
9.9 CoreOS Rocket 92
9.10 CoreOS and Rkt Background 92
9.11 Rkt Components 93
9.12 Rkt Security Analysis 94
9.13 Container Defaults 97

10 Security Recommendations 98

10.1 Generation Container Recommendations 98
10.2 LXC Specific Recommendations 105
10.3 Docker Specific Recommendations 106
10.4 CoreOS Rkt Specific Recommendations 109
10.5 Relevant Kernel Hardening 110

11 The Future 113

11.1 Containers on the Desktop 113
11.2 New Potential Namespaces 114
11.3 Additional Lightweight Isolation and Sandbox Platforms 114
11.4 The Open Container Initiative 116
11.5 Containers In Other Platforms 117
11.6 Container Specific Operating Systems 117
11.7 Unikernels and Microhypervisors and Hybrid Models 118
11.8 The Big Idea Of Microservices 120

12 The End 122

12.1 Conclusion 122
12.2 Acknowledgements 122
12.3 About The Author 123

1 Introduction

nccgroup

"I am large, I contain multitudes" – Walt Whitman

Linux containers have recently developed into a production-ready technology for OS level virtualization,[1] yet very little security research or best practices have been made public, and concerns for deployment[2] and security[3, 4] abound. This whitepaper seeks to expand on what little security information exists on what container technologies are capable of and can ultimately provide. This paper starts by examining how containers compare to hardware virtualization[5] and what prior vulnerabilities have occurred in these systems. This paper explores Linux container security underpinnings and discusses building and leveraging containers to provide strong application isolation. The paper also explores future areas of potential vulnerability or security research.

Much more than "chroot on steroids", Linux containers and the underlying features which power them offer an entire ecosystem of software. Various pieces form the ability to build a operating system jail or application sandbox, or to simply better package applications. This can all be done while offering extremely low resource overhead and many features typically restricted to hardware virtualization such as snapshots, pausing virtual machines (VMs) or live VM migration. Linux kernel namespaces, capabilities and resource limits via cgroups offer excellent tools for building defense in depth, a strongly encouraged security paradigm for all types of applications. From web applications and network services to desktop applications and thick clients, many of the container methods or software discussed within this paper also support different versions of the Linux kernel on almost any supported hardware. This can offer much needed security improvements in embedded or Internet of Things (IoT) devices. Finally, Mandatory Access Controls (MAC) and system call (syscall) filtering, given new life by container deployments, offer additional and formidable protections against application or container to host compromise.

Containers are quickly growing in popularity due to the ongoing shift in application or entire datacenter deployments from once traditional three-tier architectures on bare metal to large, powerful computers running a number of virtual machines. Deployments are now shifting from service oriented architectures (SOA) to Platform as a Service (PaaS) or "microservices", distilling services down. Microservices and PaaS are, as part of their core design, highly available, fault tolerant, easily scalable, service oriented and container driven. It is hardly surprising, given this general movement from hardware to software, there is a surge in the popularity of containers via LXC, Docker and CoreOS Rocket (rkt), which are helping push the use of Linux containers out of data centers at Google and IBM and into servers and desktops everywhere. Over the past several years, container focused companies such as Docker and IBM, along with a growing number of others, have greatly raised the profile and ease-of-use for Linux containers.

This paper is intended for a wide technical audience, with sections benefiting everyone from security consultants and researchers to enterprise blue teams, application developers, and devops teams. Readers may be looking to either evaluate container implementations, security risks and challenges, or understand the hardening features offered by modern container systems. Before proceeding, NCC Group warns against taking statements, configuration options or other details within this paper as verbatim. Containers are a hot topic and new developments are announced seemingly every week. In addition to the evolving container offerings, OS virtualization is a quickly moving target. New potential threats will undoubtedly be uncovered, hopefully kept in line by proactive security enhancements. Along with a general defense in depth design, assumed protections should constantly be re-evaluated, explored, tested and improved upon.

[1] https://en.wikipedia.org/wiki/Operating-system-level_virtualization
[2] http://www.csoonline.com/article/2984543/vulnerabilities/as-containers-take-off-so-do-security-concerns.html
[3] http://www.infoworld.com/article/2923852/security/containers-have-arrived-and-no-one-knows-how-to-secure-them.html
[4] http://www.eweek.com/security/security-is-a-key-concern-for-container-users.html
[5] https://en.wikipedia.org/wiki/X86_virtualization

Containers offer many overall advantages. From a security perspective, they create a method to reduce attack surfaces and isolate applications to only the required components, interfaces, libraries and network connections. In an age where complexity and the lines of code for applications such as Microsoft Office in 2013 are greater than the entirety of Windows XP,[6] application sandboxing should be the rule, not the exception. Moving fully to memory safe native code languages such as Golang and Rust will likely take many more years, and code fuzzing or qualified security review can only catch so many vulnerabilities. Inner-application sandboxing and outer-application containers offer a method to cut losses and draw a line in the virtual sand. When performance differences are negligible, is there a realistic reason to not isolate applications? Apart from time and effort, the security due diligence is encouraged.

Before continuing, NCC Group also cautions readers that OS-level virtualization will always be fundamentally less secure than hardware-level virtualization, which itself is less secure than physical server isolation. However, such security absolutes should offer little discouragement. Containers chiefly offer a method to decrease attack surface and improve isolation with a relatively small added complexity and hardware resources when compared to "traditional" virtualization. Any security solution should be well hardened and vetted, follow the principal of least privilege, the principal of least access and defense in depth.[7] Finally, it should also be noted that several components used by Linux containers and discussed in this document are relatively new. This includes the User Namespace and unprivileged containers, as well as seccomp-bpf filtering. As some of these features deal with complex, high-risk areas of the kernel, additional vulnerabilities or weaknesses are likely to occur within these components.

1.1 Motivation

The motivation for using containers varies depending on the ultimate and intended use case. This may consist of decreasing power or storage costs related to hardware virtualization inefficiency, allowing developers to more easily ship and test code or increasing security on a workstation platform through application sandboxing (such as ChromeOS). It appears through industry observations and repeated security engagements by NCC Group, by far the most popular motivation is essentially application packaging, testing and deploying entire stacks on a Platform-as-a-Service (PaaS). Motivations for using containers may involve providing internal clouds or production infrastructure on top of bare metal or virtual machines. Docker, Google, Amazon AWS, Rackspace, Heroku, CloudFoundry, Stackato, CoreOS, Flockport, Rancher among others are an example of the use, and rising popularity of, minimal-overhead Linux OS virtualization.

Modern Linux use stretches from servers, desktops and laptops to mobile phones, routers, IoT and a vast number of other embedded devices. From micro form-factor devices to supercomputers, Linux is everywhere. Security and application isolation, going beyond simple Discretionary Access Controls (DAC) or standard Kernel hardening should be available to all, regardless of the hardware architecture of their intended platform. Containers or the technologies that power them can be used to help achieve improved security with little to no performance overhead.

Future use of the individual features powering containers (such as namespaces, capabilities and cgroups), or Linux containers as a whole, may allow for increased security, application sandboxing and isolation anywhere modern versions of the Linux kernel are supported. This allows for strong security and helps developers to implement security ideals such as privilege separation, the principle of least access, isolated root capabilities and resource limits. Many of these container security features have yet to be realized in Linux based embedded devices in much need of security improvements. This ranges from consumer and commercial grade routers, smart phones such as Google Android, as well as the general Internet of Things (IoT).

[6] http://www.informationisbeautiful.net/visualizations/million-lines-of-code/
[7] https://en.wikipedia.org/wiki/Layered_security

Just as Mandatory Access Control can attempt to limit process capabilities, there exists little to no reason why a modern web browser needs unfettered access to the users' computer. Slowly major operating system vendors are implementing such limitations by default, along with other inner-application sandboxing, requiring exploits to use privilege escalation or force attackers to use multi-vulnerability chaining. However, these protections thus far are largely unimplemented outside of web browsers and document readers,[8] in addition to mainly being implemented by non-Linux operating systems.

Containers and their supporting features help support an overall model of defense in depth through layered security[9] for applications on a server or a desktop. In the age of Advanced or Persistent Threats and nation state attackers, defense in depth as a principal is only possible method which may realistically prevent successful attacks. It should be clear, containers alone do not offer a perfect solution, but they can and should be used to quickly raise the bar and frustrate system compromises through application exploits, primarily by adding isolation and reducing system attack surfaces.[10]

1.2 Virtualization Background

Before exploring Linux containers and their security, it is important to understand the fundamentals of what the software or system is capable of providing, in addition to general security considerations. Largely borrowing terminology from virtualization, the term *host* in this paper will be used to indicate the primary Operating System (OS) or device on which the container exists (where LXC is setup, where the Docker daemon or engine is running, etc). The term *guest* or *container* will refer to the collection of processes or application container itself, running within the host. Finally, the term *escape* will correspond to a guest interacting with, or otherwise compromising, the host in a manner not intended. Escaping will often take the form of violating the core security principals of isolation, such as the guest breaking out of the container.

1.2.1 Full-Virtualization

Since roughly 2006 most commodity x86, x86-64 and ARMv7 microprocessors[11] from Intel, ARM and AMD offer a hardware-assisted virtualization through special CPU instructions. This provides essentially what is complete isolation between guest kernels and the host, and allows running many different operating systems within the same physical host.

VMware's ESX, the Xen HVM and KVM within Linux are examples of "Virtual Machine (VM)" technology or "Hypervisors". This hardware mode allows the host to support different guest operating systems (such as a Microsoft Windows or FreeBSD guest on a Linux KVM host). While speed is often comparable to "bare metal" execution, full virtualization is still the slowest of the three types discussed within this paper. Although this form of virtualization requires a number of virtualized hardware devices, the security is quite robust. In some cases, data is passed through directly to hardware devices, however the attack surface is typically quite small.

This security robustness is largely due to well vetted virtual hardware, often presenting a minimal hypervisor attack surface when compared to other virtualization methods. When discussing the security and implementation of full virtualization, the attack surface may differ between so called "Type one" hypervisors on bare metal (*e.g.* Citrix Xen, VMWare ESXi and KVM) vs "Type two" hypervisors, which are implemented on top of a normal kernel (*e.g.* VirtualBox, VMware Workstation, and QEMU).

[8] Provided, this is the first logical step, as web browsers and document readers offer significant attack surfaces, are often reachable by remote attackers and facilitate an ease of exploitation through heap massaging and other factors.
[9] https://en.wikipedia.org/wiki/Layered_security
[10] With the exception of the Linux kernel, although tools such as seccomp-bpf can help for most hardware platforms.
[11] The IBM POWER, AS400, OS/2 and other CPU architectures were designed specifically for hardware virtualization. These methods and systems are out of scope for this paper, as they are often implemented within large organizations with specific requirements (banks, universities, supercomputing labs and research centers).

Security considerations: In terms of security guarantees, OpenBSD's often gruff leader, when speaking on hardware virtualization via hypervisors takes a particularly pessimistic stance:

> **''x86 virtualization is about basically placing another nearly full kernel, full of new bugs, on top of a nasty x86 architecture which barely has correct page protection. Then running your operating system on the other side of this brand new pile of shit. – You are absolutely deluded, if not stupid, if you think that a worldwide collection of software engineers who can't write operating systems or applications without security holes, can then turn around and suddenly write virtualization layers without security holes. You've seen something on the shelf, and it has all sorts of pretty colours, and you've bought it. That's all x86 virtualization is.''**
> - An openbsd-misc email by Theo de Raadt

Theo's opinion aside, escaping from hardware virtual machines is considered quite difficult and rare, although surely possible. Weakness have been discovered in several major platforms, typically in the areas of virtualized device drivers. Most recently in QEMU (as used by Xen) a large number of vulnerabilities have been discovered, causing regular cloud hosting providers to perform painful host reboots.[12, 13] For instance, the RTL8139 driver contained a heap overflow (CVE-2015-5165), and an emulated block device contained a use after free issue (CVE-2015-5166). Issues were even discovered within the floppy controller and the PCNET NIC driver (CVE-2015-3209)[14] . Vulnerabilities have been discovered within the QEMU/Xen IDE subsystem (CVE-2015-5154) and within the hvm_msr_read_-intercept function (CVE-2014-7188). Each of the above issues risked guest escape, Denial of Service (DoS) or allowed reading data from the hypervisor, depending on the configuration. Historically, within VMware Workstation (which is a type two hypervisor[15]), a guest could escape and execute arbitrary code on the host (CVE-2009-1244). Finally, Microsoft's Hyper-V has also contained at least one known escape (MS15-068).

While the paper is now fairly dated, An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments by Tavis Ormandy offers a strong security overview. Additionally an Analysis of Hypervisor Breakouts by Insinuator found, somewhat unsurprisingly, that increase in attack surface through drivers, graphics shaders, DMA and other features which travel from guest to hypervisor or guest to hardware risks additional vulnerabilities, especially in type two hypervisors.[16] Despite these examples, the risk of escape is much lower and the difficulty of host or guest-to-guest exploitation much higher than other forms of virtualization (excluding various network attacks). Apart from physically separate hardware, this method offers the strongest guest isolation.[17]

Deployment considerations: Full virtualization is typically less energy and storage efficient than other virtualization methods. Due to the special CPU instructions, this technology is also only supported on distinct hardware (*e.g.* x86, x86_64, ARM), limiting the deployment scenarios. As such, this virtualization type is also not subtable for low power devices and only recently supported on ARM.[18] Finally, full virtualization often follows the model of virtualizing the entire operating system; adding additional security to individual applications within the system is left up to traditional hardening best practices. This includes

[12] http://vmblog.com/archive/2014/09/29/rackspace-joins-amazon-in-cloud-reboot-over-xen-hypervisor-bug.aspx
[13] http://www.theregister.co.uk/2015/02/28/new_xen_vuln_causes_cloud_reboot/
[14] This arguably overhyped vulnerability was also marketed as ''VENOM'' by an adversarial ''threat'' focused security company.
[15] http://www.golinuxhub.com/2014/07/comparison-type-1-vs-type-2-hypervisor.html
[16] This type one vs type two issue is also easily illustrated by the number of VMware Workstation escapes when compared to ESX/ESXi.
[17] Unrelated to x86 and x86-64 Hypervisors, the PS3 hack and VM escape was particularly impressive. See How the PS3 Hypervisor was hacked by Nate Lawson for an excellent write-up.
[18] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438i/CHDCHAED.html

using least or multi privilege models, using strong authentication, least privilege, controlling system access and other security standards. Reboots of the host hardware for full virtualization, although rarely required, are extremely painful (as each VM needs to boot up after the host). Reboots cannot be performed gracefully unless live migration is supported or another high availability or fault tolerant system is in place.

In use by: Amazon AWS via EC2 when selecting "private virtualization", the Google Compute Engine (KVM), OpenStack deployments, and VMware vCloud Air (ESX) hosting. Also used by a vast majority of private companies with internal server hosting, traditional or modern Platform as a Service (PaaS), various internal datacenters or QA environments. Type two hypervisors, which use many of the same hardware technologies, are very common on development workstations, with IT personnel, security researchers and modern power-user systems (this is especially the case given the wide CPU support in modern workstations and laptops).

1.2.2 Paravirtualization

Paravirtualization (PV) requires a custom OS kernel, with patches for the specific para-virtualization APIs, also referred to as hypercalls. This also allows for different virtual hardware, at the expense of a modified kernel. Xen was one of the first to implement paravirtualization, and is the long standing and primary hypervisor platform for many cloud providers or high-security desktop operating systems.[19]

Xen offers a powerful production and battle tested paravirtualized environment, as well as support for non-paravirtualized operating systems via the hardware virtual machine (HVM) mode, similar to the full virtualization method as previously discussed. A large amount of device emulation code within the Xen HVM is based on code from QEMU project.

Security considerations: This custom patch-set may lag behind the kernel with regard to security issues and may be incompatible with some hardening protections such as grsecurity. Many updates and patches will require a reboot of the host, similar to OS virtualization which may add pressure to avoid updating. Escaping from PV guests may be easier than full virtualization, and has occured several times, notably and quite recently in Xen, such as XSA 148. Finally, PV guests may introduce weaknesses if configured alongside PVHVMs, such as a mixed Xen environment.[20]

Deployment considerations: Hardware support and kernel support may be lacking, or may require additional updates or modification before a guest can paravirtualized. Security for PV may place hosts at the mercy of vendor updates, such as the case with Xen. Reboots may also be extremely difficult depending on the infrastructure.

In use by: Amazon AWS/EC2, Rackspace, and many other hosting providers.

1.2.3 OS-Virtualization

This method provides the highest performance,[21] fastest "start-up" time, and is widely considered the most efficient virtualization method. It uses a shared kernel across both the host and guest. Without virtualization of hardware devices, speed and efficiency are greatly increased, although due to this shared kernel, the guest is limited to an identical OS as the host (such as Linux on Linux). This method is typically referred to as "Containers" on Linux, "Jails" on FreeBSD or "Zones" on Solaris.

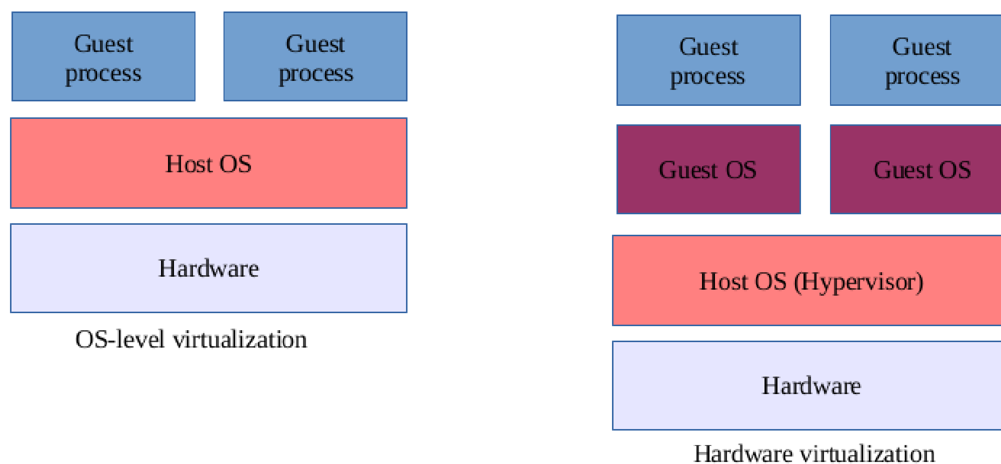
[19] https://www.qubes-os.org/
[20] http://xenbits.xen.org/xsa/advisory-148.html
[21] http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/$File/rc25482.pdf

Security considerations: The shared kernel, which permits the desired efficiency and speed, also gives way to an increased risk of host compromise or guest escapes. This is due to the fundamentally shared resources and significantly larger attack surface compared to paravirtualization or full virtualization. With this shared kernel, syscalls, shared networking, direct device or disk access, information leaks and the vast majority of kernel vulnerabilities are typically the source of OS virtualization security failures.

Deployment considerations: OS virtualization can be used anywhere a modern Linux kernel is supported. This allows the capability to use containers equally from non-hardware virtualized big iron to small embedded systems. Speed, boot time, storage efficiency and flexibility are key advantages. For large amounts of I/O or network-bound processing, OS-virtualization may be comparable to hardware virtualization, due to poorly optimized and default NAT or custom storage such as AUFS in Docker.[22]

In use by: Linux Containers (LXC), Docker, CoreOS Rkt, OpenVZ, Heroku Dynamos, RancherOS, FreeBSD Jails, Solaris Zones, Illumos/SmartOS, the defunct OpenBSD Sysjail and a number of other systems such as Mirage OS (which borrowed some ideas from Solaris Zones).[23]

The following image[24] helps illustrate OS virtualization, using a single kernel / ring0 vs hardware (full) virtualization with multiple instances on top of a hypervisor:



1.3 Benefits of An OS-Virtualization System

Overall, the hardware vs software virtualization discussion can easily be easily understood in the context of RAID (Redundant Array of Independent Disks) solutions, as both hardware and software RAID implementations exist and are in widespread use. The hardware version typically offers improved performance and reliability but comes at a cost of yet another kernel driver, electronics cost, restricted disk types, limited motherboard layout and other form factor requirements. On the other hand, a software RAID offers increased portability, flexibility, source code customization, almost zero hardware requirements, improved block device support and reasonable performance across a number of hardware platforms.

Another more specific analogy comes from Docker's Jérôme Petazzoni, who likens this virtualization dif-

[22] http://domino.research.ibm.com/library/cyberdig.nsf/papers/0929052195DD819C85257D2300681E7B/File/rc25482.pdf
[23] http://media.ccc.de/browse/congress/2014/31c3_-_6443_-_en_-_saal_2_-_201412271245_-_trustworthy_secure_modular_operating_system_engineering_-_hannes_-_david_kaloper.html
[24] Image from Wikimedia Commons.

ference to brick walls (hard, slow to setup, messy to add or move) vs room dividers (fragile, deployed in seconds, moved and added quickly). It also safe to say, even with a cumbersome configuration management, hardware Virtual Machines (VMs) and other hypervisors will diverge in version, patch freshness, and configuration which can often result in security gaps or vulnerabilities due to difficult upgrades. On the other hand, minimal OS virtualization can be easier to keep organized, may offer no additional cost, and DevOps or system administrators typically better understand the ''threat landscape'' of these environments (as opposed to complex Hypervisors or other virtualization management solutions).

Speed: The shared kernel architecture offers almost non-existent startup speed, as starting containers is little more than launching new processes. This can speed up testing or deployment, and reduces idle OS time. These quick start-up times also allows for desktop application containers to be a realistic and attainable goal with little end-user awareness.

Management: A number of tools already exist to monitor, explore, configure and troubleshoot containers. While orchestration and service discovery is an ongoing development area (and potential security pain-point for major container platforms or deployments), there are many emerging and successful platforms.

Disk Footprint: Required storage is relatively small for most base images and deltas can require an extremely minimal additional footprint. Advanced filesystem options allow for only deltas to be saved and multiple containers to share the vast majority of the root filesystem through Copy on Write (CoW) filesystems and Union filesystems such as OverlayFS. OS virtualization can also reduce applications to their required code, libraries, components and interfaces, a key differentiator from full hardware virtual machines.

Implement anywhere: Containers and many (but not all) of the supporting kernel or security features work anywhere Linux can run, compared to only x86, x86-64, and newer versions of ARM for HW virtualization.

Ability to freeze and unfreeze containers: This can allow for quick power saving on a massive, data-center wise scale or to save laptop battery power. By essentially sending a `SIGSTOP` to all of the processes bound to a container (for a specific cgroup), these processes will halt or ''freeze'', providing easy parity with hardware virtualization.

Containers can easily be moved easily: To any Linux host, from the cloud to local machines, spin-up will work quickly compared to HW virtualization which may have incompatible image or file formats (requiring moving a much larger image than an application container). Note that moving containers is currently limited to non-running containers, although some recent developments by Docker and LXD are attempting to solve this limitation.

1.4 Drawbacks of an OS-Virtualization system

Security: Discussions on the risks of OS-Virtualization system almost always center on the relatively immature implementation of Namespaces, the problem of root (capabilities) and the shared kernel, which is a fundamental risk and unfortunately bountiful attack surface. In the words of noted hardware and virtualization security researcher Joanna Rutkowska, of the Invisible Things Lab, when speaking on process isolation of Operating Systems[25]:

[25] Joanna is also a core contributor and author of the high security Xen hypervisor desktop "QubesOS". See the article How QubesOS is different for more information.

> **"Sure, the inter-process isolation provided by a monolithic kernel such as Windows or Linux could never be compared to the inter-VM isolation offered even by the most lousy hypervisors. This is simply because the sizes of the interfaces exposed to untrusted entities (processes in case of a monolithic kernel; VMs in case of a hypervisor) are just incomparable."**
>
> - Shattering the myths of Windows security by Joanna Rutkowska

Software all the way down: The lack of CPU/hypervisor enforced isolation pushes security 100% into software. While this makes the solution more flexible, it fundamentally increases the potential for attack, especially if hardware must be exposed directly (such as the sound or video card interfaces) or the kernel attack surface cannot be reduced (such as through system call filtering). This is essentially an extension of "all of the eggs in one software basket", which can sometimes be a difficult concept to accept.

Problems with legacy code: Several areas of the kernel still lack support or lack their own namespaces, a key isolation mechanism for OS virtualization. For example in Linux, the procfs and sysfs pseudo-filesystems are not namespace aware. This is easily illustrated by examining the required steps for protecting from privileged containers (that is, containers without the User namespace), various information leaks and other vulnerabilities. The problem of procfs is also readily apparent in sysstat related programs[26] which for instance will not report cgroup resource isolations, but instead reflect the "free" memory of the host OS.

Live Migration: The current inability to "live migrate", moving a running container from one host to another, when compared to some hardware Virtual Machine (VM) platforms or OpenVZ. It should be mentioned, developments are underway within Docker to support this[27] and with LXD for LXC.[28]

A homogeneous environment: The inability to virtualize other Operating Systems, as the shared kernel must remain the same for the userland processes, may be a blocker for some deployments. While a homogeneous environment has some security advantages such as less overall patching, more consistent configuration management and less "edge cases", it risks mass exploitation or vulnerability.

Single point of failure: It goes without saying, a single kernel is a single point of failure, be that performance, stability or security. If the host platform is disabled, crashes, needs to be rebooted or is compromised, all of the guest containers are affected. In the case of a host or kernel compromise, the containers must also be considered compromised. See Software compartmentalization vs. physical separation for more information and examples by Joanna Rutkowska.

Complexity at scale: Orchestration frameworks (Rancher, MESOS/Aurora, Docker Swarm, LXD, OpenStack Containers, Kubernetes/Borg, etc) are only recently catching up to the container craze, there are too many competing models to list. Many have questionable or unaudited security or leave major requirements out, such as secret management. While containers may be easy to get working within a workstation or a few servers, scaling them up to production deployment is another challenge altogether, even assuming your application stack can be properly "containerized".

For those interested in the comparison between these different virtualization approaches and want more information, see all three parts of Hypervisors are Dead, Long Live the Hypervisor on osv.io for a good overview of the problems and solutions.

[26] http://fabiokung.com/2014/03/13/memory-inside-linux-containers/
[27] http://www.slideshare.net/Docker/live-migrating-a-container-pros-cons-and-gotchas
[28] https://insights.ubuntu.com/2015/05/06/live-migration-in-lxd/

2 Linux Containers Overview

2.1 A Brief History of OS Containers

A review of past container methods starts with the infamous chroot written by Bill Joy. Included in version 7 Unix in 1979 and BSD in 1982 chroot, this can be thought of as the first "OS container". Due to the weak implementation chroot security is quickly broken[29] if an adversary or compromised process can gain superuser or "root" access.[30] Skipping forward in time includes FreeBSD Jails,[31] Linux OpenVZ, User Mode Linux (UML), Solaris Zones (in 2005), and AIX Workload Partitions (in 2007[32]). While these are also shared-kernel virtualization systems, this paper is focused on modern, native Linux solutions and will offer minimal comparisons going forward.

Linux VServers, were introduced around 2001 and represented a leap forward in usability and speed when paravirtualization was objectively more popular, and hardware support for hypervisors was still weakly supported or prohibitively expensive. Armed with a basic Linux kernel patchset and some userland tools, VServers allowed for most, if not all, of what we think of as containers today. This solution broke out different running applications implemented within instances of Linux distributions into different "security contexts". Despite all the advancements of VServers, kernel namespaces, a topic that is further discussed in Section 3 on page 20, were weakly supported. Many namespaces were still undergoing active development or yet to be implemented entirely. A lack of cgroups also resulted in difficult performance isolation across different servers, where existing tools were inadequate to easily manage process groups. In general, security was nowhere near as complete (or as incomplete, depending on your current perspective of container security).

Jump forward to 2016, where Linux kernel technologies such as namespaces, cgroups, and capabilities separately and in concert support LXC, Docker, CoreOS Rocket/rkt, Heroku, Joyent, SubgraphOS, RacherOS and countless other container solutions and PaaS systems. One only needs to view the list of companies supporting the Open Container Initiative to witness the seriousness of containers. The current push to move to Microservices as a platform also uses containers as a primary driver and key component. Finally, new and intriguing efforts to create a hybrid of hardware and OS virtualization, such as Intel's Clear Containers offers one possible future combining the best of both virtualization strategies.

2.2 Linux Containers: where are they now?

2.2.1 Servers

At a basic level, container-related systems and sandboxes are used in everyday software, even chroot is still widely used simply due to its simplicity. Many common Open Source daemons contain out of the box support for a "chrooted" environment, such as Apache or Postfix. Some network daemons are almost always chrooted. If privilege separation is enabled in OpenSSH, which is the default, an unprivileged helper process will be chrooted into an empty directory to handle pre-authentication network traffic for each client.[33] The newest versions of OpenSSH support a "sandbox" directive for `UsePrivilegeSeparation`, this offers "additional protections" using three different methods for different platforms (systrace for OpenBSD, seatbelt for OSX, seccomp for Linux and POSIX rlimits (as a fallback and for other platforms). While these solutions are not "containers", many of the security features and goals are shared.

Moving to the containers we normally think of such as LXC, Docker, and CoreOS Rkt, many companies are planning a massive increase in container use and deployment. These transitions are happening for a number of reasons, including better economy for (PaaS) providers, enabling better development pipelines,

[29] This may partially be due to security not being a key design goal, but testing.

[30] By creating a directory, "chrooting" into it, then using a directory traversal sequence (*e.g.* ../) to escape the "outer" chroot.

[31] Which had their own serious vulnerabilities: CAN-2005-2218, CVE-2007-0166, CVE-2010-2022, and CVE-2014-3001.

[32] http://www.ibm.com/developerworks/aix/library/au-workload/

[33] http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man5/sshd_config.5?query=sshd_config

establishing better parity between dev/test environments, adding additional security layers, cutting energy costs and other reasons. Cloud providers such as Rackspace and Amazon[34] now officially support support Docker containers.

Google, a major technology leader, has stated "everything at Google runs in a container", and Google is no small operation (reportedly starting and turning over two billion containers each week[35]) as everything from search to gmail is packaged and run inside unique containers.[36] Although the exact details are sparse, Google's containerization is likely powered at at least one layer by Linux KVM,[37] cgroups and historically a variant of LXC called "lmctfy" (Let Me Contain Than For You)[38] are also assumed to be key components. It should be noted that recent efforts have been applied to Docker's libcontainer (now part of the Open Container Initiative), according to the lmctyf README. Recently, Google has also released "Kubernetes"[39] for managing containers, based off their Borg platform, as well as cAdvisor for container resource usage and statistics. To further support containers in their provided cloud environment, the Google App Engine supports Docker images in managed VMs.[40]

Traditional hardware virtualization companies, such as VMware, are also quickly ramping up support for containers. VMware Photon / VMware Cloud supports several Linux Container formats through Photon[41] and allows managing containers alongside existing hypervisors and virtual machines[42] through "Project Bonneville".[43] Openstack has also added support for containers through their Nova Virt API.[44, 45]

In parallel to these big names, a number of companies are publicly developing and releasing container tools, container focused operating systems, are offering hosting services and in some cases powering their entire infrastructure via containers. This includes (in no particular order) Intel, Google, VMware, Docker, CoreOS, Amazon, Rackspace Cloud, IBM, Engine Yard, Joyent, Cloud Foundry, Heroku, Sandstorm.io, Red-Hat OpenShift, eBay, Cloud Foundry, HP (Stackato), StackEngine, OpenStack, DigitalOcean, ClusterHQ, Spotify, and many more. Countless others which remain unnamed, unpublished or cannot be mentioned here are deploying containers or using the features that power them in an ad-hoc fashion internally. Heroku, one of the first major PaaS platforms, has built their business from a container model of minimally executing, tightly controlled instances called "Dynos". According to the Heroku documentation:

> *"Dynos execute in complete isolation from one another, even when on the same physical infrastructure. This provides protection from other application processes and system-level processes consuming all available resources. The dyno manager uses a variety of technologies to enforce this isolation, most notably LXC for subvirtualized resource and process table isolation, independent filesystem namespaces, and the* `pivot_root` *syscall for filesystem isolation. These technologies provide security and evenly allocate resources such as CPU and memory in Heroku's multi-tenant environment.*
>
> - Heroku Dyno isolation and security

[34] Amazon EC2 offers their ECS system. See https://aws.amazon.com/containers/ for more information.
[35] https://speakerdeck.com/jbeda/containers-at-scale
[36] http://googlecloudplatform.blogspot.com/2014/06/an-update-on-container-support-on-google-cloud-platform.html
[37] Andrew Honig of the cloud security team at Google has stated in presentations "KVM it is the killer feature".
[38] https://github.com/google/lmctfy
[39] http://blog.kubernetes.io/2015/04/borg-predecessor-to-kubernetes.html
[40] https://cloud.google.com/compute/docs/containers
[41] https://vmware.github.io/photon/
[42] https://blogs.vmware.com/cto/vmware-containers-containers-without-compromise/
[43] http://venturebeat.com/2015/06/22/vmware-previews-project-bonneville-a-docker-runtime-that-works-with-vsphere/
[44] http://docs.openstack.org/liberty/config-reference/content/lxc.html
[45] https://wiki.openstack.org/wiki/Docker

2.2.2 Clients

Although containers in data centers and servers are where almost all of the focus is (largely due to the ease with which containers allow 'shipping'' software), servers should not be the only focus. As many security professionals have known for years, attackers have long since switched to targeting clients and workstations. Advancing the state of Linux sandboxing through application containers, dropping privileges, reducing or eliminating suid binaries and other isolation mechanisms employed by containers can help improve Linux application security. Application containers can also easily limit CPU, Disk and Memory for everything from resource hungry (and highly exploited) web browsers to anonymous liberation technology systems which can be critical to secure from indirect information leaks.

Unsurprisingly, Google is also a major player for client-side container technologies, although they don't use actual containers. Both the Chrome OS distribution[46] and the Chromium/Google Chrome web browser heavily utilize the protections mechanisms powering containers, such as kernel Namespaces (Network and PID), Seccomp-bpf, SUID sandboxing, or in newer versions full user namespaces.[47] Unfortunately for security, and somewhat paradoxically, Google Android, one of the most widely deployed Linux distributions (if you can call it a distribution) is surprisingly missing many of the modern container features provided by the kernel, apart from the mount Namespace and Mandatory Access Control via SELinux.[48] Android still chiefly relies on Discretionary Access Controls (DAC) aka UNIX permissions and other enforcement via a normal UID and process isolation based security model.

The recently released high-security Linux distribution SubgraphOS, (currently Alpha) offers application containers/sandboxing via Oz[49] for a number of security sensitive applications. This is the default system along with a grsecurity patched kernel, Tor based routing, X11 isolation via Xpra, gated outbound connections via custom firewalls and a host of other features. Additional solutions for Linux containers as well as simple application sandboxes using a subset of container technologies (namespaces, seccomp-BPF, piviot_root, capabilities, unshare, etc) are explored and further discussed within Section 11.1 on page 113.

2.3 Prior Art: Linux Container Security, Auditing and Presentations

While Linux Container systems (LXC, Docker, CoreOS Rocket, etc) have undergone fast deployment and development, security knowledge has lagged behind.[50] The number of people focused on container security, and within those who publish security research for containers seems disproportionately small, given their advantages, ongoing deployment and demand for security knowledge among companies large and small. That said, a number of presentations, articles, and papers touch on or explore the various security subjects, although often at a high level or for a specific container platform.

Included below is a list of compiled resources, almost solely focused on container security. Some in which, despite their age, served as inspiration for this paper and yet other articles and presentations offer a good overview of Container security strengths, weaknesses and adoption. If you are looking to brush up before continuing with this paper, I suggest many of the following resources. However, as with any technology publication, readers should keep in mind the resources included below may not contain the most up-to-date information, such as User namespace support in Docker v1.10, using seccomp within LXC or exploring new security features.

[46] https://www.chromium.org/chromium-os/chromiumos-design-docs/system-hardening
[47] https://chromium.googlesource.com/chromium/src/+/master/docs/linux_sandboxing.md#User_namespaces_sandbox.md
[48] This may be due to the somewhat unicast nature of Intents multicast of broadcasts for IPC.
[49] https://github.com/subgraph/oz
[50] http://www.infoworld.com/article/2923852/security/containers-have-arrived-and-no-one-knows-how-to-secure-them.html

2.3.1 Multi-Container

Abusing Privileged and Unprivileged Linux Containers by NCC Group's own Jesse Hertz explores several historical attacks against LXC and Docker. An in-depth discussion of countermeasures for these vulnerabilities, as well as proof-of-concept code is included.

Linux Containers: Future or Fantasy? by Aaron Grattafiori, the author of this paper, was presented at DEF CON 23. This talk explores the background of containers, how the different elements are secured, some common flaws in common container systems (LXC, Docker, Rkt), what they provide, general container recommendations and formed the basis for expanding on much of it within this white paper.

Linux Containers (LXC), Docker, and Security by Jérôme Petazzoni, a Docker staff member, offers a basic overview of some of the challenges faced by containers and a look into some security mechanisms and potential architectures. Jérôme also has a great overview of Implementing a "separation of concerns" with Docker containers.

Thoughts on interoperable containers by Fabio Kung of Heroku explores the security aspects, among other features of Linux containers, specifically Docker and interactions with developers.

Linux Container Security by Matthew Garrett brings up some good points on Hypervisors vs containers. A related discussion on LWN offers additional commentary, including a discussion of Sandstorm.io strategy.

Doger.io by Jay Coles contains an excellent and conscience overview of Linux Containers with a number of links to other good resources. This site can be a good starting point.

Seven problems of Linux Containers by Kirill Kolyshkin of Parallels, Inc. explores how OpenVZ solved and explored container problems in the Kernel.

2.3.2 LXC Specific

LXC Security Analysis by Roman Fiedler, Austrian Institute of Technology, offers a recent evaluation of LXC security which explores several vulnerabilities and underlying risks with LXC, some which apply to Linux containers in general.

Hard Containers - LXC and GrSecurity by Diego Elio Pettenò explores running LXC with grsecurity patches enabled.

Secure Linux containers cookbook by IBM offers an overview of security mechanisms and examples for LXC with SELinux and SMACK Mandatory Access Controls (MAC).

Security of Linux containers in the cloud by Dobrica Pavlinušić of University of Zagreb offers an overview of LXC security mechanisms and how these can be combined with MAC systems.

2.3.3 Docker Specific

The Golden Ticket: Docker and High Security Microservices by Aaron Grattafiori, the author of this paper, explores several security principles and features as they apply to Microservices and hardened containers for both Docker and runC. A video of the presentation can also be found on Youtube.[51]

Docker & Security by Florian Barth and Matthias Luft, is a recent presentation that explores the security basics and past vulnerabilities of Docker. The slides also include some discussion on Microservices, devops vs security and provides and overview of other Docker services.

Green font, Black background: Docker Security by Example by Diogo Mónica of Docker at Dockercon EU in

[51] https://www.youtube.com/watch?v=346WmxQ5xtk

2015 explores Docker security in depth and by-example and with many embedded demos.

A Docker Image Walks in to a Notary by Diogo Mónica of Docker at ContainerCamp 2015 covers Docker Notary and Docker image security.

Vulnerability Exploitation In Docker Container Environments by Anthony Bettini of Flawcheck, was presented at Blackhat Europe 2015[52] explores the insecurity and common issues around Docker container images.

Docker, Docker Give Me The News: I Got A Bad Case Of Securing You was presented at DEF CON 23 by David Mortman of Dell and explores and touches on the security costs and potential risks of deploying Docker and other container systems.

Are Docker containers really secure? by Dan Walsh, of RHEL and SELinux fame, is semi-involved with Docker development, and can often found strongly and consistently advocating for SELinux.[53]

Docker Security: Who can we trust, what should we verify? by Nils Magnus at goto; conference explores the basics of Docker security, attack surfaces and offers some good basic recommendations.

Least-privilege Microservices by Diogo Mònica and Nathan McCauley, both Docker security leads, provides a basic overview of how how Docker Containers can support least-privilege Microservices.

Introduction to Container Security by an unknown author is a short Docker white paper, released in May of 2015 that includes discusses the available security features for Docker and discusses some general risks and best practices when using it.

Security Properties of Containers Managed by Docker is a non-free Gartner research paper by Joerg Fritsch. This paper primarily asks the question, can your Docker containers actually contain..[54] Apparently Docker faired decently enough, according to The Register and InformationWeek although many of the security complaints revolve around security maturity and administration, not fundamental risks of OS virtualization and attack surface analysis.

Analysis of Docker Security by Thanh Bui of the Aalto University School of Science, this document from late 2014 covers a concise overview of Docker isolation mechanisms, security features of an unknown but older Docker version.

Docker, DevOps, Security by Chris Swan of cohesiveFT explores how security, containers and DevOps meet.

CIS Docker 1.11 Benchmark v1.0.0 by Pravin Goyal of VMware, Inc. with contributions from various container companies, including Docker itself. This includes a comprehensive, if extremely dry, best practices security document. Some of the best practices can also be explored and tested via the "Docker Bench" tool,[55] created by Docker's Diogo Mónica.

Docker and SELinux by Daniel Walsh is a basic overview of Docker security issues and recommendations, although his opinions may not be widely shared. (Takes half of the talk to explain SELinux basics)

Docker Security by Nathan McCauley and Diogo Mónica of Docker is a gentle overview video relating to security by the lead Docker security team members.

Docker Security: Are Your Containers Tightly Secured To The Ship? and Docker Security: Secure container

[52] https://www.blackhat.com/docs/eu-15/materials/eu-15-Bettini-Vulnerability-Exploitation-In-Docker-Container-Environments.pdf
[53] https://lwn.net/Articles/515034/
[54] http://blogs.gartner.com/joerg-fritsch/can-you-make-your-containers-contain/
[55] https://github.com/docker/docker-bench-security

deployment on Linux by Michael Boelen, CISOfy, covers some basic risks, mechanisms and recommendations for Docker security.

Container security: Kernel internals by an unknown author offers a kernel viewpoint of Container security, exploring from the hardware "up". This presentation has some good information and overall perspective.

Creating Containers by Michael Crosby of Docker is an excellent and in-depth exploration of Linux Containers, while there isn't a focus on security, this blog post (and the others on his website) are worth reviewing.

On the Security of Containers by Eric Windisch formerly of Docker offers a good overview behind why lightweight application containers offer additional security and how Virtual Machines can augment where trust is weak or the risk is high. "Containers are not contradictory to other, existing best-practices".

2.4 TL;DR Linux Containers

As we will explore within this paper, Linux containers (LXC, Docker, Rkt, etc) are typically comprised of five major components:

1. **Kernel namespaces** are the major building block of Linux containers, which isolate the applications within different "userspaces" such as network, processes, users themselves and the file system. Further information, examples and exploration is included within Section 3 on page 20.

2. **Control Groups** also known as cgroups are essentially ulimit on steroids, which limit various host hardware resources. This currently includes CPU count and usage, disk performance, memory, and other process limits. Further information can be found within Section 4.1 on page 27.

3. **Root Capabilities** help enforce namespaces in so-called "privileged" containers by reducing the power of root, in some cases to no power at all. Further information, such as the background, implementation, use and defaults for major container platforms are included within Section 5.1 on page 30.

4. **Pivot_root** is a syscall to "pivot" into the new container environment, by changing the root file system. Although the use and implementation of `pivot_root(2)` and related initialization steps of the container's init is not explicitly discussed within this paper, "pivoting" correctly can be crucial for security.[56]

5. **Manditory Access Controls (MAC)** such as AppArmor and SELinux are not required for creating containers, but are often a key element to their security. MAC helps enforce the security controls implemented by other container features, adding defense in depth and general platform security for any permission level within the container, privileged user or otherwise. Further information can be found within 8.1.6 on page 69.

Container security threats, largely aimed at escaping confinement can originate from many sources. Attacks could originate from a compromised container, a maliciously uploaded container, the applications within a container or attacks against the local network from within the container. The various threats to containers is included and explored in Section 7 on page 49.

To counter the identified threats, several recent Linux security advancements (not solely limited to containers), such as the User namespace and Seccomp are discussed in Section 8.1 on page 66 and Section 8.3 on page 74 respectively. Not to be left out, the historically under-configured, and I would argue under-appreciated, Mandatory Access Controls (MAC) systems are also covered 8.1.6 on page 69. These features offer a great reduction in attack surface and defense in depth within supported container platforms.

[56] File descriptors must be closed, `ptrace(2)` must be limited, and confused deputy attacks understood.

After understanding the new security features, the historical and current threats, potential risks (and available security features) I have included a brief overview of each of the major container platforms explored in this paper (LXC, Docker, Rkt). This may help understand and evaluate the motivations, project priorities, security threats (past and present) and available security options. This overview, background and security analysis of each platform starts in Section 9 on page 82. To further support this information, a table of secure defaults and support options can be found in Section 9.13 on page 97.

The cursory exploration of the current security strengths and weaknesses in Section 9 referenced above, is largely to set the stage for this paper's recommendations section. Hardening your container deployment and configuration against many of the earlier identified threats is discussed in Section 10 on page 98. The recommendations are grouped this section first as general Linux and container platform agnostic terms as well as specific recommendations section for each of the three platforms covered within this paper: LXC in Section 10.2 on page 105, Docker in Section 10.3 on page 106 and finally CoreOS Rkt in Section 10.4 on page 109.

Looking forward, an overview of potential future container platforms, other minimal sandboxing techniques, unikernels, microservices is included in Section 11 on page 113 and finally, the paper's overall conclusion can be found in Section 12.1 on page 122.

3 Namespaces

3.1 Namespaces Background

Linux kernel namespaces are the fundamental building block of containers on Linux. The idea of namespaces as a logical construct to deal with scope or segmentation is a common idea in computer science.[57] For Operating Systems, Plan 9 introduced[58] in 1992 the idea of namespaces, among other interesting concepts such as network or union filesystems and many other computing advancements outside containers. In Linux, kernel namespaces form a foundational isolation layer that allows for the implementation of Linux containers by creating different userland views. The Namespaces in Operation series on Linux Weekly News by Michael Kerrisk offers a great overview and explores each namespace. The Resource Management: Linux kernel namespaces and cgroups presentation by Rami Rosen offers a long and in-depth exploration of namespaces and cgroups. Readers interested in additional background and information should start with these resources.

Largely instrumented via the `CLONE_NEW` flags during process creation, namespaces split the traditional kernel global resource identifier tables and other structures into their own instances. This partitions processes, users, network stacks and other components into separate analogous pieces in order to provide processes a unique view. The distinct namespaces can then be bundled together in any frequency or collection to create a filter across resources for how a process, or collection thereof, views the system as a whole. Methods to help enforce namespace isolation are crucial, as each kernel resource exposed by a namespace must be wrapped with enough knowledge and direction to determine and help implement the appropriate access control. The implementation of these controls still proves difficult, as illustrated throughout this paper.

3.2 Namespaces Implementation

Apart from the `clone(2)`[59] syscall (similar to `fork(2)`) with accompanied `CLONE_NEW` flags during process creation, two additional syscalls were added. `setns(2)`[60, 61] and `unshare(2)` syscalls were added to facilitate namespace creation, as well as processing joining or leaving namespaces. From a security standpoint, essentially only two new syscalls were added in order to interact with or create the various kernel namespaces, which is great for keeping syscall bloat to a minimum.

Each namespace below is listed in order of introduction date within the released Linux kernel. When exploring this area of containers, it is important to keep in mind that namespaces are still a work in progress, and some key areas of the kernel still do not have their own namespace (such as devices, time,[62] syslog,[63] security keys, and the proc and sys psuedo-filesystems themselves). Additionally, as the kernel was not designed with namespaces in mind, the development is ongoing, and continues to improve. As we know from security engineering, this "bolting on" process is much more difficult and error prone process than having "security by design". Finally, the lack of completeness has also created a number weak security areas and been the source of a myriad of vulnerabilities both during and after the primary "development window".[64] Readers who want to drive right in should jump to Section 7 on page 49, which provides an overview of prior weaknesses relating to namespaces and various container threats.

With the exception of user namespaces, all namespaces require either root or the `CAP_SYS_ADMIN` capability (which is essentially root) to create them. Unprivileged containers, which are created by non-root users may

[57] https://en.wikipedia.org/wiki/Namespace
[58] http://www.cs.bell-labs.com/sys/doc/names.html
[59] http://man7.org/linux/man-pages/man2/clone.2.html
[60] http://man7.org/linux/man-pages/man2/setns.2.html
[61] The `setns(2)` syscall can be used along with the inode entries of `/proc/<pid>/ns`.
[62] Although at least one attempt was made: https://lwn.net/Articles/179825/
[63] https://lwn.net/Articles/527342/
[64] The time in which the vast majority of development and introduction take place for a specific component.

seem to be an exception to this requirement, but they are not. The kernel allocates this new user namespace first, wherein the user can then create new namespaces using this new pseudo-privileged mode.[65]

The various kernel namespaces are often used in concert to create what we know of as "Linux Containers", but they can also be used separately in order to gain additional isolation and security for specific application or security needs (for which a full container is unnecessary). This additional utility outside containers can be provided in several ways. For example, the mount namespace can also be used by Linux PAM to provide per-user or per-group filesystem views upon login. The network namespace can be used to isolate application traffic and implement complex routing scenarios. The `unshare(2)` system call also allows a running process to "disassociate" parts of its kernel execution context that are currently, and implicitly, being shared with other processes without first creating a new process.

Listed below is a section covering the basics around each namespace and, when appropriate, a short exploration of using the namespace outside of containers. Within the code sections below, many of the namespace examples use utilities which can be found in the util-linux package. This is often installed by default in many Linux distributions and the latest version of the source on any kernel mirror.[66]

3.3 Mount Namespace

Introduced in 2.4.19, the mount namespace via `CLONE_NEWNS` is the oldest and only namespace introduced in the 2.4 kernel.[67] The mount namespace provides a process, or group thereof treated as container, with a specific view of the system's mounted filesystems. This view can range from mount paths, physical or network drives, or advanced features such as union filesystems, bind mounts, or overlay filesystems (where some section of the host filesystem is directly accessible, yet other reads or writes stop at container boundaries.) The mount namespace can also indirectly secure other namespaces by restricting access to the hosts' mounted instance of /proc, which would violate the PID namespace constraints. Additional articles, reading, and resources are included below.

- Private mount points with unshare by Jon Jensen
- Introduction to Linux namespaces - Part 4: NS (FS) by Jean-Tiare Le Bigot
- Applying mount namespaces by IBM

3.4 IPC Namespace

System V IPC objects and POSIX message queues can utilize their own namespace starting in 2.6.19. As with other namespaces, `CLONE_NEWIPC` provides a method for creating objects in an IPC namespace which are visible to all other processes that are members of that namespace, but are not visible to processes in other IPC namespaces. This is typically used for shared memory segments. This isolation helps from some IPC related attacks[68] and Denial of Service scenarios. Additional articles, reading, and resources are included below.

- Introduction to Linux namespaces - Part 2: IPC by Jean-Tiare Le Bigot

[65] See Resource management: Linux kernel namespaces and cgroups Rami Rosen for more information on this area.
[66] https://www.kernel.org/pub/linux/utils/util-linux/
[67] This is also evident from the "NEWNS" part of the `clone` flag, which simply stands for "New Namespace" as there is no description of intent similar to the other namespaces. A good example of how namespaces were an "add on".
[68] http://labs.portcullis.co.uk/whitepapers/memory-squatting-attacks-on-system-v-shared-memory/

3.5 UTS Namespace

Around 2006, Linux 2.6.19 added support for separate host and domain "UTS" names via CLONE_NEWUTS. The kernel patch for this namespace was introduced by the now LXC developer Serge E. Hallyn, offers a very simple, yet necessary, namespace function. Different hostname and domain name values are crucial for many applications, identification, web services, logging and other features. Additional articles, reading, and resources are included below.

- Namespaces in operation, part 2: the namespaces API by Michael Kerrisk
- https://blog.jtlebi.fr/2013/12/22/introduction-to-linux-namespaces-part-1-uts/ by Jean-Tiare Le Bigot

3.6 PID Namespace

The Process identifier (PID) namespace groups processes and helps create containers by protecting from cross-application attacks, information leaks, malicious use of ptrace and other such potential weaknesses. The PID namespace can also be nested, and is easily the most commonly demonstrated Linux namespace in examples and presentations. Linux 2.6.24[69] introduced the CLONE_NEWPID flag around 2008 and development was largely supported by IBM and OpenVZ through a remount or separation (but not namespace) of procfs. All PID namespaces start at PID 1, the traditional "init". Subsequent calls to fork(2), vfork(2), or clone(2) will produce processes with PIDs that are unique within the namespace. While other namespaces can be created using the unshare() system call, new PID namespaces, in kernels prior to 3.8, can only be created using clone(2).

It is also important to consider the security of cgroups when discussing the PID namespace. Because cgroups can be accessed via containers, and cgroups provide an abstraction of process groups, it can allow containers, in specific vulnerability scenarios, to control PID resources outside of its own PID namespace. For example, malicious use of the Freezer cgroup to "freeze" (SIGSTOP) host processes and creates a Denial of Service condition. Finally, related to the PID namespace, the recently added PID cgroup[70] can be used to limit the maximum number of tasks, which should help prevent specific types of Denial of Service attacks and provide a fall-back for application-specific "max number of processes" configuration options.

3.6.1 PID Namespace Explorations

Using the unshare command, we can easily run a new shell within its own PID, Network and User namespace. The unshare command is also wrapped within an strace to illustrate the system calls made:

```
$ sudo strace -e unshare,clone unshare -f -p -U --mount-proc ps auxww
unshare(CLONE_NEWNS|CLONE_NEWUSER|CLONE_NEWPID) = 0
clone(child_stack=0, flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
    child_tidptr=0x7f77ddfe19d0) = 2703
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
nobody       1  0.0  0.1  10016  2252 pts/1    R+   19:25   0:00 ps auxww
--- SIGCHLD {si_signo=SIGCHLD, si_code=CLD_EXITED, si_pid=2703, si_status=0, si_utime
    =0, si_stime=0} ---
+++ exited with 0 +++
```

In another example, we can start a Docker Busybox container then sleep for 1337 seconds.

```
sudo docker run busybox sleep 1337
```

[69] https://lwn.net/Articles/259217/
[70] https://www.kernel.org/doc/Documentation/cgroup-v1/pids.txt

Now we can use the nsenter command to join a Docker PID namespace, assuming only a single sleep process is running on our system.

```
$ sudo nsenter -t `pgrep sleep` -m -p
root@ubuntu:/# ps auxww
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.1  18180  3280 ?        Ss   02:18   0:00 /bin/bash
root       145  0.0  0.0   4348   672 ?        S+   02:19   0:00 sleep 1337
root       147  0.0  0.1  18184  3364 ?        S    02:20   0:00 -bash
root       150  0.0  0.1  15572  2204 ?        R+   02:20   0:00 ps auxww
```

Additional articles, reading, and resources for the PID namespace:

- PID namespaces by Michael Kerrisk
- More on PID namespaces by Michael Kerrisk
- Intro to Linux namespaces: Part 3 PID by Jean-Tiare Le Bigot

3.7 Network Namespace

The network namespace, through CLONE_NEWNET is considered to be quite complex and required one of the longest development times.[71] Starting back in 2.6.24, various namespace features have provided the ability for different IPv4 or IPv6 stacks and everything that relates or supports these features including devices, addresses, routing, firewall rules, procfs directories, and sockets among other requirements. Modern container solutions can either create a bridged network with the host and other containers, support a fully disabled network lacking any network namespace, can combine existing network namespaces or simply share a non-isolated network interface with the host OS (which is considered risky).

Depending on the deployment situation, containers may not require a network at all. However many, if not all, container solutions default to a bridge-style virtual network interface, using Network Address Translation (NAT) for layer three connectivity. Network attacks or threats against attack surfaces present a risk for any container host. Either a container on the same bridge device as the host, locally with other containers or even locally networked systems. These connected systems may or may not, support, interconnect, or manage the container or group of containers in question.

3.7.1 Network Namespace Example

Network namespaces can easily be manipulated with the ip netns family of commands which includes displaying namespaces (as referenced within /var/run/netns, not just any network namespace), adding or removing them, executing commands within namespaces, reporting processes within a network namespace and other operations. First, we'll create a new network namespace called "torns" and bring up the loopback interface (the only interface in the namespace so far):

```
$ sudo ip netns add torns
$ sudo ip netns exec torns ip link set lo up
$ sudo ip netns exec torns ifconfig -a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
```

[71] For an example of the complexity, when when network namespaces were first developed, they were actually incompatible with sysfs. See Sysfs and namespaces on LWN for more information.

Next, we'll create a peered virtual interface (veth0), assigning one to the new network namespace (veth1), and setup a static IP address on the interface:

```
$ sudo ip link add veth0 type veth peer name veth1
$ sudo ip link set veth1 netns torns
$ sudo ip netns exec torns ip addr add 172.16.1.2/24 dev veth1
$ sudo ip netns exec torns ip link set dev veth1 up
$ sudo ip netns exec torns ifconfig -a
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

veth1     Link encap:Ethernet  HWaddr 72:5a:06:85:a9:23
          inet addr:172.16.1.2  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

After the namespace is setup, we'll finish setting up the other interface in the global network namespace:

```
$ sudo ip addr add 172.16.1.1/24 dev veth0
$ sudo ip link set dev veth0 up
```

We can now ping the peer interface within the root network namespace. Using strace we can see the setns system call, used to "enter" this network namespace:

```
$ sudo strace -f -e setns ip netns exec torns ping -c 2 172.16.1.1
setns(4, 1073741824)
PING 172.16.1.1 (172.16.1.1) 56(84) bytes of data.
64 bytes from 172.16.1.1: icmp_seq=1 ttl=64 time=0.115 ms
64 bytes from 172.16.1.1: icmp_seq=2 ttl=64 time=0.141 ms
```

If we want to gain additional connectivity, such as to the Internet or another network, we'll setup a default gateway and then either need to establish a bridge with a physical interface, or directly setup Network Address Translation (NAT), such as using iptables in the example below:

```
$ sudo ip netns exec torns route add default gw 172.16.1.1
$ iptables -t nat -A POSTROUTING -o enp0s3 -j MASQUERADE
```

Now executing a command within our new namespace, we can reach the Internet and serve an isolated Tor hidden service, for instance. Note that additional namespaces and not running Tor as root would be the next step in a more hardened system. This is purely serving as a limited example for network namespaces outside containers:

```
$ sudo ip netns exec torns tor
Feb 23 21:08:12.310 [notice] Tor v0.2.6.10 (git-71459b2fe953a1c0) running on Linux
    with Libevent 2.0.21-stable, OpenSSL 1.0.2d and Zlib 1.2.8.
Feb 23 21:08:12.311 [notice] Tor can't help you if you use it wrong! Learn how to be
    safe at https://www.torproject.org/download/download#warning
Feb 23 21:08:12.311 [notice] Read configuration file "/etc/tor/torrc".
Feb 23 21:08:12.314 [notice] Opening Socks listener on 127.0.0.1:9050
Feb 23 21:08:12.000 [notice] Parsing GEOIP IPv4 file /usr/share/tor/geoip.
Feb 23 21:08:12.000 [notice] Parsing GEOIP IPv6 file /usr/share/tor/geoip6.
Feb 23 21:08:12.000 [warn] You are running Tor as root. You don't need to, and you
    probably shouldn't.
Feb 23 21:08:12.000 [notice] Bootstrapped 0%: Starting
Feb 23 21:08:13.000 [notice] Bootstrapped 5%: Connecting to directory server
Feb 23 21:08:13.000 [notice] Bootstrapped 10%: Finishing handshake with directory
    server
Feb 23 21:08:13.000 [notice] Bootstrapped 15%: Establishing an encrypted directory
    connection
     .... SNIP ....                 .... SNIP ....
Feb 23 21:08:52.000 [notice] Tor has successfully opened a circuit. Looks like client
     functionality is working.
Feb 23 21:08:52.000 [notice] Bootstrapped 100%: Done
```

Again it should be noted, in a real deployment we would want to perform additional hardening, use Mandatory Access Controls (MAC) and isolate other global namespaces from the process, rather than just the network namespace. This overall illustration of network namespaces can be used as a simple way to gain some network isolation for applications, to easily prevent a process from using the network or force a particular network route (such as over a VPN connection or Tor circuit). Full containers, such as using one of the systems discussed within this paper offer an easier to use solution that packages in many other security features.

The unshare command can also be used to create a new network namespace:

```
$ sudo strace -e unshare unshare -n /sbin/ip link
unshare(CLONE_NEWNET)                   = 0
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default
     link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

Listing 1: Using unshare to create a new network namespace. The strace command is used to illustrate the unshare() syscall in action.

Additional articles, reading, and resources for the network namespace:

- Introduction to Linux namespaces – Part 5: NET by Jean-Tiare Le Bigot
- Namespaces in operation, part 7: Network namespaces by Michael Kerrisk
- Introducing Linux Network Namespaces by Scott Lowe

3.8 User Namespace

Development of user namespaces or "the user namespace" started in 2.6.23 and finally finished in 3.8. This namespace is the most recent to be implemented and provides a crucial security barrier for containers by preventing many potential attacks through essentially shifting or sliding UIDs. Until user namespaces, the "root" user, even when restricted using Linux capabilities, was still *uid 0* as far as the kernel was concerned. User namespaces offer the ability for processes to believe they are operating as root when inside the namespace, but outside, the process IDs belong to a non-root low-rights user. By using existing kernel or permissions based protections against privileged operations, the system has fundamentally removes or frustrates a large number of potential attacks. It also significantly reduces the attack surface from the system, and due to these advantages, it is considered by many to be critical for securing containers.

User namespaces also paved the way for fully-unprivileged containers. Normal, non-root and unprivileged users can create and manage their own user namespace based containers. Unfortunately, as with any new code dealing with a complex and highly sensitive area of the kernel, a number of high severity vulnerabilities have been unsurprisingly discovered, typically exploiting user namespaces outside the context of containers. See Section 8.1 on page 66 for more information on this namespace and 7.2.5 on page 58 for threats. Additional articles, reading and resources are included below.

- Namespaces in operation, part 5: User namespaces by Michael Kerrisk
- Namespaces in operation, part 6: more on user namespaces by Michael Kerrisk
- What's Next for Containers? User Namespaces by Scott McCarty
- Rooting out Root: User namespaces in Docker by Phil Estes

3.8.1 User Namespace Example

If LXC is installed, a simple utility, lxc-usernsexec, can be used to enter and explore (via /bin/sh) a user namespace as an unprivileged user. The same can also be done via the unshare command, but we'll just use the LXC tool below:

```
host $ id
uid=1000(aaron) gid=1000(aaron) groups=1000(aaron)
host $ lxc-usernsexec
```

While "root" in the new user namespace (but same mount, pid and other namespace as the host), we can attempt two privileged operations, both of which are denied, even it seems as if we're UID 0:

```
guest $ id
uid=0(root) gid=0(root) groups=0(root)
guest $ cat /etc/shadow
cat: /etc/shadow: Permission denied
guest $ strace -f -e delete_module rmmod btusb
...
delete_module("btusb", O_RDONLY|O_NONBLOCK) = -1 EPERM (Operation not permitted)
```

Back within the host system, in another shell we can look for instances of /bin/sh, and witness one running as UID 100000 (the default slide for LXC):

```
host>$ ps auxw | grep /bin/sh
100000   23871  0.0  0.0   4440   652 pts/26   S+   23:12   0:00 /bin/sh
```

4 Control Groups

4.1 Cgroups Background

Control Groups (cgroups) are a mechanism for applying hardware resource limits and access controls to a process or collection of processes. The cgroup mechanism and the related subsystems provide a tree-based hierarchical, inheritable and optionally nested mechanism of resource control. To put it simply, cgroups isolate and limit a given resource over a collection of processes to control performance or security. Cgroups can generally be thought of as implementing traditional ulimits/rlimits, but now operating across groups of tasks or users. A new, more powerful and more easily-configured alternative to ulimits/rlimits. To silence the naysayers and doubt over code bloat or added complexity, the documentation clearly states:

> **"The kernel cgroup patch provides the minimum essential kernel mechanisms required to efficiently implement such groups. It has minimal impact on the system fast paths, and provides hooks for specific subsystems such as cpusets to provide additional behavior as desired."**
> - Kernel documentation - cgroups.txt by Paul Menage

The analysis and configuration of cgroups is performed by mounting a special cgroup virtual filesystem. This filesystem can be used by other tools to control and view the state of namespaces and controls, although it also can be a method of container escape if it is not restricted when mounted within a guest.

In the world of containers, cgroups obviously manifest themselves as instruments to control access to resources, such as preventing runaway containers, denial of service attacks (forkbombs[72] or Out of Memory triggers) and to limit access to devices via a device whitelist (as the kernel "dev" system is not namespace-aware). Following the UNIX everything-is-a-file model, the cgroups system is implemented as a typical Linux pseudo-fileystem, similar to `/proc` or `/sys`. With this pseudo-filesystem comes typical attacks, such as unmounting or mounting-over in order to attempt to defeat various cgroup limitations.

Control Groups first appeared in Linux 2.6.24 around 2006 as a simple method for controlling tasks, first named "Process Containers" by the Google engineers who introduced it. The feature slowly expanded to allow per-cgroup statistics, task freezing via SIGSTOP, various I/O throttling, and Mandatory Access Control (MAC) systems. Most recently, support for PID limits was added to control the maximum number of processes as well as advanced network controls such as buffer limits and traffic priority levels enforced by iptables.

When reviewing cgroups, it can be easy to imagine them as a method to fill the gaps of kernel namespaces. However, it is important to remember that cgroups are not namespaces, nor are they really intended or invented for containers; cgroups are simply one property of processes. Linux systems without containers still can and will use cgroups for their different security and performance needs. Different cgroup "weights" are typically used as the metric for limiting a process or set thereof for a given hardware resource. See the list of kernel documentation[73] for more information on the weights and explore each cgroup subsystem. For more examples and reading, the excellent Control Groups Series from Niel Brown on Linux weekly news, the Resource Management Guide from RedHat, and Use cases of cgroups from Oracle offer great additional information. Finally, the Resource Management: Linux kernel Namespaces and cgroups presentation by Rami Rosen offers a in-depth exploration and the sysadmin webcast Introduction to Control Groups contains some great exercises for learning cgroups along with charts and other information.

[72] https://devlearnings.wordpress.com/2014/08/22/limiting-fork-bomb-in-docker/
[73] https://www.kernel.org/doc/Documentation/cgroups/

4.2 Working with Vanilla cgroups

Typically, cgroups will be transparent when using containers, with the configuration or management performed by container management, tools or template configurations. However, as with other container systems, these Linux features can be used separately or in concert to aid in security isolation or resource limits of typical applications. Many of the command line tools simply work by modifying the default `/sys/fs/cgroup` directory within the sysfs mount point, creating, removing or changing flat files within these directories.

For many system libraries and applications, the `/etc/cgrules.conf` and `/etc/cgconfig.conf` configuration files can help establish and configure system wide cgroups. For example, this could be used to set process limits on a given user upon login via PAM or on a user group known for hogging resources. This also can be used to limit a set of server processes which accept images or other compressed files to help limit decompression bomb or other resource attacks.

If manually interacting with the pesudo-filesystem is too tedious, several tools are available in libcgroup/libcg. The `cgcreate` command creates a cgroup, `cgexec` places a task under that cgroup and `cgclassify` will move an existing task or set of tasks to a pre-existing cgroup. CGManager[74] can also be used to help reduce the burden of management if the container solution not abstract it away. CGManager can also be used if specific customizations are required that the container framework (LXC, Docker, etc) does not support, or if a simple cgroups container is the only goal. An important take-away is that cgroups can be dynamically created, added to or removed, and are not restricted to process creation, that is to say: the rules can change at any time.

Listed below is a short description of each major cgroup subsystem, for which resources are controlled. Other minor subgroups (Hugetlb and perf_event) are not discussed within this paper.

CPU (cpu, cpuset, cpuacct): This CPU system is often used to restrict a set of processes to a specific number of CPUs or amount of "CPU time". Further details can also be found within RedHat's CPU shares documentation.

Memory: The memory subsystem controls memory allocation and limits for a group of processes. Limits can be hard, soft and have "pressure" applied (which can then dynamically change the limit strategy) among other expected features.

BLKIO: The BLKIO controls disk read or write speeds, operations per second, queue controls, wait times and other operations on an associated major and minor numbered block device. This subsystem does need to be explicitly enabled, but offers significant control over I/O when compared to other more traditional methods or filesystem specific controls. Support for this BLKIO subsystem is unfortunately weak within many container platforms, largely due to filesystem implementations.

Devices: The devices cgroup subsystem is typically a whitelist, formatted for devices based on type (char vs block) and device major and minor numbers. A special 'all' type applies to all device types, major and minor numbers and is typically used as a default deny before whitelisting explicit devices. Most containers will have access to commonly-used devices such as `/dev/null` and `/dev/urandom`.

Network: The recently-added Network classifier cgroup can provide a method to tag network packets with a "classid" value. This can be applied or acted upon by iptables for packet filtering and quality of service (QoS).[75]

Freezer: This subsystem allows a cgroup of tasks to be "frozen" by essentially sending a `SIGSTOP` signal

[74] https://s3hh.wordpress.com/2014/03/25/introducing-cgmanager/
[75] https://www.kernel.org/doc/Documentation/cgroups/net_prio.txt

and later "unfrozen" or "thawed" by sending a corresponding SIGCONT signal. This can be useful to pause a system or entire set of applications when there is no expected or intended use.[76] Applications which cannot catch a SIGSTOP may not behave normally when thawed.

PIDs: In 2015, the recently-released PID cgroup[77] allows for limiting the maximum number of tasks. This helps prevent accidental or intentional fork-bombs and help avoid hitting system-wide task limits. This can be understood as an extension of RLIMIT_NPROC and was recently added with commit 49b7...865c.

4.3 Containers and cgroups

As part of a container system, cgroup management is typically abstracted away, with the exception of LXC (which largely requires it as part of the standard template). In Docker and CoreOS Rkt, modifications can be made at container runtime, although documentation or examples are fairly weak. Fortunately, the simple UNIX-like implementation of cgroups allows for modification outside of container frameworks, should these not meet the resource or security goals of the platform in question. Working with cgroups outside of containers, for modern resource limits and device access control for everyday applications, can also be advantageous, although this is outside of the scope of this paper.

4.4 Future of cgroups

Systemd offers strong support for cgroup controls, for both labeling processes and supporting resource restrictions. We can expect to see further documented support or applications offering simple limits enforced by cgroups and managed via systemd in the near future. For further information on systemd and cgroups, the article How to get started with CGroups by CertDepot offers a good overview and examples, as well as a large number of other resources.

[76] This can play a strong role in desktop container systems to limit battery use of background applications.
[77] https://lkml.org/lkml/2015/2/22/204

5.1 Capabilities Background

On Linux and other UNIX-like operating systems[78] , the uid 0 (zero) user aka "root" has complete control over the system (one account to rule them all). This is also the case for any setuid-root binary, which if it contains a serious vulnerability such as memory corruption (leading to a code-path hijacking), root level access can be reached by a lower rights user. Over the history of Linux and related platforms, privilege escalation vulnerabilities to root have proved a recurring problem, either due to suid or simply violating the principle of least privilege by running as root in the first place. Linux capabilities were introduced in Linux 2.2[79] as a way to split this "absolute" access control model by partitioning root access. A capability privilege bitmap for each process is created, and then enforced by the kernel.

In a simple example, the common yet simple setuid root binary `/bin/ping`, risks privilege escalation for what should be a minimal privilege requirement – raw sockets. While the attack surface for privilege escalation to root is not limited to only suid binaries, it is important to note that the attack surface of `ping` is not only exposed when and if the raw sockets are being used, but also through any network parsing code, command line arguments or other potential areas of vulnerability within the suid binary.[80] Any exploitable condition within a root process or accessible suid binary allows the attacker to then act as the full root user. This attack surface for root privilege escalation is extended across all applications running as root, all suid binaries among other less obvious locations and all of the loaded libraries and directories they interact with. Obviously this is a serious and historical risk to system security as clearly illustrated by Neil Brown:

> **"The problem with this design is that programs which are running setuid exist in two realms at once and must attempt to be both a privileged service provider, and a tool available to users - much like the confused deputy."**
>
> - Ghosts of Unix past by Neil Brown

Switching to using a capabilities model, the `ping` command now has access to only what it needs the privileges for, via a raw sockets capability called `CAP_NET_RAW`. This fits the original intent of the application's requirements and practices the principal of least privilege to the letter. Further examples could be a web server outside of a container, which only needs root access in order to bind to a privileged port (< 1024), can simply use the `CAP_NET_BIND_SERVICE` capability or an NTP daemon, which can use the `CAP_SYS_TIME` capability to restrict privileged access to only time-setting, again as intended and required.

Capabilities do their work as a trait of each Linux thread or process, and are inherited from the parent through the use of `clone(2)` and `fork(2)`. The `__user_cap_data_struct{}` defines the different `effective`, `permitted`, and `inheritable` bitmasks. To fit with other privilege models, once a set of capabilities is configured, they can only be restricted further, not increased. Since the capabilities model can effectively split some root-level operations, it can make audits, traditional file permissions and security actually more complex if not within a sandbox or container environment. For example, if an application had two roles, admin and user, it would be easy to tell which operations could gain admin access. If the application has user roles in-between admin and user (such as, a network only admin) it is not immediately clear which executables would provide which escalated privileges. Great care must be taken to audit and understand permission customization and the newly developed privilege model within your system.

[78] FreeBSD is notably missing from this root versus user split to a capabilities model, as the project was a major reason why the Capabilities model was not standardized as part of POSIX.1e. The FreeBSD project considered the implementation poorly reviewed and the 32 or 64 bit mask too restrictive. See http://www.trustedbsd.org/privileges.html for more information. It should be noted FreeBSD does have a capabilities model called Capsicum (https://www.cl.cam.ac.uk/research/security/capsicum/freebsd.html which also has attempted a Linux port (https://github.com/google/capsicum-linux).
[79] Capabilities remain an optional component, enabled using the `CONFIG_SECURITY_CAPABILITIES` kernel configuration
[80] Does anyone else remember suid cdrecord exploits?

While the above sounds good and seems attainable, the capabilities model is unfortunately (many years later) still under development, and many "umbrella" or somewhat mixed capabilities exist. This is explored in detail within a post by infamous Linux security researcher and developer Brad Spengler of Grsecurity, False Boundaries and Arbitrary Code Execution. This article provides a brief security investigation across a large number of Linux capabilities and how they can be used to gain additional capabilities or full root. This particular article should be studied for anyone attempting to claim the capabilities system is not complex, half-baked or that a particular Linux capability can be used safely. This post by Brad likely spurred several mailing list discussions, and led Michael Kerrisk to write the LWN article CAP_SYS_ADMIN: the new root, which explores the various problems and options (including even renaming the `CAP_SYS_ADMIN` capability to `CAP_GOOD_AS_ROOT`). Finally, due the complex and high risk nature of privileged context switching and privileged access in general, implementation vulnerabilities have been discovered in several different capability implementations, `CAP_NET_ADMIN` historically being the worst security offender.

5.2 Additional Introductory Resources

Linux Capabilities: making them work a paper published in 2008 by Serge E. Hallyn and Andrew G. Morgan is the canonical resource for capability use and examples, alongside How Linux Capabilities Work (in 2.6.25) by Wenliang (Kevin) Du. Additional security specific info can be found in Security In-Depth for Linux Software by Julien Tinnes and Chris Evans.

5.3 Understanding Capabilities

Capabilities are enforced by the kernel for all "privileged operations" and are always a subset of a given capabilities combination. This consists of a processes' existing inheritable or permitted capabilities, with the exception of a thread possessing the `CAP_SETPCAP` capability. They can also be set and applied to either process threads or binary files using extended filesystem attributes.[81]

Process or thread properties are established by several different syscalls: `capset(2)` to set capabilities, `capget(2))` to retrieve them, and `cap_get_proc(3)`/`cap_set_proc(3)` to control them. The three main sets are as follows:

- **Effective:** The combined set of capabilities utilized by the kernel to evaluate permission checks.
- **Permitted:** Limiting superset of the effective capabilities the thread and it's children can assume. This also controls what may be used by such a thread given the `CAP_SETPCAP` capability (which allows adding or removing capabilities to other processes). This transitive trust model allows for a reduced set of administrative functions.
- **Inheritable:** The set of capabilities preserved across an `execve`.

Capabilities for files (typically ELF binaries) are set using `setcap(8)`, `cap_set_file(3)`, `cap_set_fd(3)` on filesystems which support extended attributes (xattrs) :

- **Effective:** A single bit, which during execve the permitted capabilities become the effective set.
- **Permitted:** Automatically permitted to the thread regardless of the threads inheritable capabilities.
- **Inheritable:** Joined with the threads eventual inherited capability set.

Although capabilities should follow a default deny or principal of least access model, several capabilities

[81] https://atrey.karlin.mff.cuni.cz/~pavel/elfcap.html

are obviously more dangerous than others. An attempted overview for each capability is provided below.[82] The list is in order of compromise risk for a container system (not a host in itself), so CAP_SYS_ADMIN and CAP_NET_ADMIN are near the top, whereas CAP_WAKE_ALARM has a low risk of exploitation impact. For each capability described below, the contents are largely paraphrased or wording is taken verbatim from the Linux capabilities man page.[83] However, additional details or descriptions have been added in many cases, with a focus on security and potential capability abuse. Additional information included below is also sourced from Brad Spenglers' excellent *False Boundaries and Arbitrary Code Execution* post referenced earlier. Finally, yet other information was obtained from the Grsecurity Appendix on Capabilities Names and Descriptions. The comments below for the various capabilities should not be considered exhaustive, and as an entire whitepaper just exploring the use, implementation, vulnerability, and exploitation of Linux capabilities could easily be created.

When using file capabilities, it can be important to understand that the binaries themselves are treated similarly to setuid. In this case, the loader rejects environment variables such as LD_PRELOAD and even if ptrace(2) is permitted, users are prevented from attaching to their setcap'd processes. In addition to this, also similar to setuid, and dissimilar to sudo, setcap binaries do not drop all of their environment variables.

CAP_SYS_MODULE: Allows the process to load and unload arbitrary kernel modules. This could lead to trivial privilege escalation and ring-0 compromise. The kernel can be modified at will, subverting all system security, Linux Security Modules, and container systems.

CAP_SYS_ADMIN: Largely a catchall capability, it can easily lead to additional capabilities or full root (typically access to all capabilities). A wide range of some 35 different operations,[84] including access to NVRAM, setting the hostname, setting the domainname, administration of the "random device", controlling serial ports, sending arbitrary SCSI commands, performing filesystem mounting or umounting, modifying shared memory, calling TTY ioctls, creating new namespaces, and bypassing UNIX socket credentials. CAP_SYS_ADMIN is required to perform a range of administrative operations, which is difficult to drop from containers if privileged operations are performed within the container. Retaining this capability is often necessary for containers which mimic entire systems versus individual application containers which can be more restrictive.

CAP_NET_ADMIN: Allows the capability holder to modify the exposed network namespaces' firewall, routing tables, socket permissions, network interface configuration and other related settings on exposed network interfaces. This also provides the ability to enable promiscuous mode for the attached network interfaces and potentially sniff across namespaces. It should be noted several privilege escalation vulnerabilities and other historical weaknesses have resulted from the ability to leverage this capability. This includes CVE-2011-1019 which effectively granted the CAP_SYS_MODULE capability to load arbitrary modules and was exploited trivially using ifconfig[85] CVE-2010-4655 which resulted in a sensitive heap memory disclosure and CVE-2013-4514 resulting in Denial of Service and possibly arbitrary code execution. These issues are largely due to the significant attack surface and implicit module loading for special interfaces or socket types.

CAP_SYS_CHROOT: Permits the use of the chroot(2) system call. This may allow escaping of any chroot(2) environment, using known weaknesses and escapes.

[82]The author did not fully investigate the code paths of each capabilities and warns unknown vulnerabilities likely remain.
[83]This may provide incomplete information, and the author did not have time to fully explore the potential implications of each capability.
[84]See the comment within the definition here: http://lxr.free-electrons.com/source/include/uapi/linux/capability.h#L224
[85]See this LKML message for more information https://lkml.org/lkml/2011/2/24/203 and an example.

CAP_SETUID: A process can manipulate process UID values via the following syscalls: (setuid(2), setreuid(2), setresuid(2), and setfsuid(2)). The process can also pass arbitrary UID values when passing socket credentials (SCM_CREDENTIALS), similar to the CAP_SETGID capability. This may be useful to allow privileged containers to effectively become other lower-rights users within a container. If seteuid is used, it will allow a process to regain root after privileges after dropping them. See the credentials man page for more information on process identifiers and use or risk to authenticated UNIX sockets. Overall, this capability may allow various privilege escalation attacks through editing root-owned files.

CAP_SETGID: A process can manipulate process group IDs (GID) and forge GID values when using socket credentials (SCM_CREDENTIALS) for UNIX domain sockets. This may be useful to allow privileged containers to effectively become lower-rights users within a container. See the credentials man page for more information on process identifiers and use or risk to authenticated UNIX sockets. Overall security concerns are similar to CAP_SETUID.

CAP_SYS_PTRACE: The ability to use ptrace(2) and recently introduced "cross memory attach" syscalls such as process_vm_readv(2) and process_vm_writev(2). If this capability is granted and the ptrace(2) syscall itself is not blocked by a seccomp filter (as discussed more in Section 8.3 on page 74), this will allow an attacker to bypass other seccomp restrictions.

CAP_SYS_RAWIO: This provides a number of sensitive operations including access to /dev/mem, /dev/kmem, /proc/kcore access (disclosing kernel/host memory), modify mmap_min_addr, and access ioperm(2)/iopl(2) syscalls and various disk commands. The FIBMAP ioctl(2) is also enabled via this capability, which has caused issues in the past,[86] As per the man page, this also allows the holder to descriptively "perform a range of device-specific operations on other devices." Finally, /dev/mem and /dev/kmem read and write access is permitted with this capability, although, if these are not disabled by the Linux distribution,[87] cgroups should prevent access and Mandatory Access Control (MAC) systems may further add defense in depth to procfs entries (which mirror these devices). Overall CAP_SYS_RAWIO should be considered a dangerous capability. If malicious access to /dev/mem, /dev/kmem or related procfs entries is granted, it allows old and well understood attacks.[88]

CAP_MAC_ADMIN: Allows the process to override the Mandatory Access Control (MAC) system. This capability was implemented for the SMACK Linux Security Module (LSM), and obviously can disable or weaken a crucial security protection if used by a malicious entity.

CAP_MAC_OVERRIDE: Allows the process to perform various Mandatory Access Control (MAC) configuration or state changes, similar to CAP_MAC_ADMIN, this was implemented for the SMACK Linux Security Module (LSM) and carries with it the same risks.

CAP_FOWNER: This capability allows a process to bypass permission checks on operations which normally require the filesystem process UID to match the target file in question (such as when using chmod). This capability also allows setting extended attributes on arbitrary files, set POSIX ACLs on arbitrary files and other minor related operations. It also should be noted this potential security bypass will not be permitted for operations covered by the CAP_DAC_OVERRIDE and CAP_DAC_READ_SEARCH, which allow bypassing all normal Discretionary Access Controls (DAC) such as file read, write and execute

[86] http://lkml.iu.edu/hypermail/linux/kernel/9907.0/0132.html
[87] Either by removing kmem, which is done by modern Kernels or configuring CONFIG_STRICT_DEVMEM which restricts reads and writes to a small chunk of kernel memory.
[88] https://www.blackhat.com/presentations/bh-europe-09/Lineberry/BlackHat-Europe-2009-Lineberry-code-injection-via-dev-mem-slides.pdf

permission checks for files and directories. While the mount namespace should restrict attacks using this capability to files/devices within a container, processes retaining CAP_FOWNER may permit attacks within a container if multiple users exist, or against filesystem permissions on shared mounts or volumes.

CAP_CHOWN: A process possessing this capability can make arbitrary changes to file UID or GID values. This can be understood as a similar capability to CAP_FOWNER. Both capabilities can allow for privilege escalation to UID 0 by editing specific, and highly sensitive files, such as /etc/passwd.

CAP_NET_RAW: The process will have the ability to create RAW and PACKET socket types for the available network namespaces. This will allows arbitrary packet generation and transmission through the exposed network interfaces. In many cases this interface will be a virtual Ethernet device which may allow for a malicious or compromised container to spoof packets at various network layers. A malicious process or compromised container with this capability may inject into upstream bridge, exploit routing between containers, bypass network access controls, and otherwise tamper with host networking if a firewall is not in place to limit the packet types and contents.[89] Finally, this capability will allow the process to bind to any address within the available namespaces. This capability is often retained by privileged containers to allow ping to function by using RAW sockets to create ICMP requests from a container.

CAP_DAC_OVERRIDE: A process with this capability may bypass file read, write, and execute discretionary permission checks. A properly configured and applied Mandatory Access Control (MAC) system could still protect from a compromised or malicious process.

CAP_DAC_READ_SEARCH: This capability allows a process to bypass file read, and directory read and execute permissions. While this was designed to be used for searching or reading files, it also grants the process permission to invoke open_by_handle_at(2). This syscall allows a process to view the contents of any file using the inode value, bypassing namespace restrictions. This capability was included within Docker and LXC by default and later abused by Sebastian Krahmer's shocker exploit[90] to break out of Docker version 0.11 and prior in addition to privileged LXC containers.

CAP_SYS_BOOT: The capability to use the reboot(2) syscall that allows a process to reboot, halt, power off and disable Ctrl-Alt-Delete for the system). It also allows for executing an arbitrary reboot command via LINUX_REBOOT_CMD_RESTART2, implemented for some specific hardware platforms. This capability also permits use of the kexec_load(2) system call, which loads a new "crash kernel" and as of Linux 3.17, the kexec_file_load(2) which also will load signed kernels.

CAP_SETPCAP: This capability grants or removes capabilities from the caller's permitted set against any other accessible process. This is similar to CAP_SETFCAP described below, although it deals with processes rather than files. File capabilities use extended attributes determine the capability set of a thread after an execve.

CAP_SETFCAP: Allows a process to use security.capability fields via setxattr. File capabilities, introduced within the 2.6.24 kernel are stored in extended filesystem attributes and determine the capability set of a thread after an execve. This is similar to CAP_SETPCAP described above, which operates on processes.

CAP_FSETID: This allows a process to add a set-user-id (suid) bit for a file that does not match the gid of the calling process. Additionally, when a file is modified under this capability, set-user-ID (suid) and

[89] Both a layer 3 (iptables) and layer 2 firewall (ebtables) are required.
[90] No CVE was apparently assigned for this issue.

set-group-id (sgid) bits will not be cleared.

CAP_MKNOD: Allows a process to create special files (typically for block or character devices, but named pipes, domain sockets, and normal files are also permitted) using the mknod(2) system call, although this cannot be used to create directories – a little known feature on some platforms..Created devices may conflict or be permitted via control groups (cgroups) or even the Mandatory Access Control (MAC) enforcement; attempted exploits may also be blocked by default deny cgroup rules or default MAC rules. Creation of devices within a container should not typically be required, so this capability should be dropped.

CAP_KILL: A process can send any signal to any reachable process using the kill(2) syscall and the use of the ioctl(2) KSIGACCEPT operation. In a container setup, a compromised process with this capability should not be able to kill processes outside of its PID namespace without leveraging an additional vulnerability.

CAP_IPC_OWNER: With this capability, a process is allowed to bypass permission checks for operations on System V IPC objects. If shared host networking is used within a container setup, this capability may allow a lower-rights container to communicate with privileged IPC endpoints.

CAP_SYS_NICE: A process with this capability can use the nice(2) and setpriority(2) syscalls to change the "nice" value[91] for arbitrary processes within the PID namespace. Several other syscalls related to performance control and memory page locations are also permitted via this capability. While this capability is unlikely to lead to direct vulnerabilities in itself, it may allow for influence over potential race conditions.

CAP_NET_BIND_SERVICE: This allows a process to bind sockets to "Internet domain privileged ports" (ports less then 1024). This sole privilege requirement is typically is why many exposed network services start with and retain (without reason) elevated privileges or full root. In many cases, this capability may not actually be required, as upstream load balancers, container wrappers, or iptables redirection rules can offload this low port binding requirement (by redirecting connections to a higher port).

CAP_LEASE: This capability allows a process to establish "leases" on arbitrary files. Using the fcntl(2) syscall with F_SETLEASE or F_GETLEASE flags, the calling process can establish or list the current lease for a given file descriptor. This allows a process to effectively attach a signal handler to specific file operations, which will be blocked by the kernel until handled given a specific lease flag, such as F_RDLCK. This is often used for synchronization, although it creates potential Denial of Service (DoS) conditions (within a given mount namespace).

CAP_LINUX_IMMUTABLE: A process with this capability can set or clear filesystem attributes FS_APPEND_FL (append-only file) and FS_IMMUTABLE_FL (immutable file) using the chattr command within a given mount namespace.

CAP_SYS_RESOURCE: Allows a process to override and set resource, quota or reserved space limits. Other ext2 and ext3 ioctl(2) journaling operations are also possible. This capability is generally required for modification of process and resource limits.

CAP_SYS_PACCT: The process can use the acct(2) system call to enable or disable process accounting.

CAP_IPC_LOCK: Allows the process to "lock" parts or the entirety of the virtual memory space into RAM, pre-

[91] This "niceness" value controls the CPU priority within the Kernel scheduler, where lower the value, the higher the priority (until -20)

venting it from being swapped, using the mlock(2), mlockall(2), mmap(2), and shmctl(2) syscalls. This is typically used for memory with higher than normal security requirements, where swapping to disk is undesirable, such as when performing cryptographic operations. Note that the munlock(2) and munlockall(2) syscalls are not permitted, but memory can be unlocked using shmctl(2) with the SHM_UNLOCK value.

CAP_SYS_TTY_CONFIG: A process with this capability can use the vhangup(2) syscall (simulating a "hangup" of the terminal) and employ other privileged ioctl(2) operations[92] to configure virtual terminals (VT), such as modifying VT size and font settings, keyboard settings, tick rate, mode settings.

CAP_SYS_TIME: Allows the process to set the system hardware clock via settimeofday(2), stime(2), and adjtimex(2) syscalls. Malicious actions by a process with this capability may create problems with accurate logging for incident response or create problems with time sensitive cryptographic operations. This could occur within the host or container and target protocols such as Kerberos or TLS, or other artifacts such as signatures or session tickets that can expire.

CAP_AUDIT_CONTROL: Allows a process to enable, disable, change filter rules, and view the status of "kernel auditing" via a unicast netlink socket.

CAP_AUDIT_WRITE: A process can write records to the auditing log via a unicast netlink socket. This may allow for injecting malformed audit logs.

CAP_BLOCK_SUSPEND: Starting in Linux 3.5, this new capability allows a process to use features largely designed for mobile devices such as "wake locking", which can prevent the system from suspending. This is typically exposed through procfs via /proc/sys/wake_lock.

CAP_SYSLOG: This capability was finally "forked" in Linux 2.6.37 from the dreaded CAP_SYS_ADMIN catchall, this capability allows the process to use the syslog(2) system call. This also allows the process to view kernel addresses exposed via /proc and "other interfaces" when /proc/sys/kernel/kptr_restrict is set to 1. The kptr_restrict sysctl setting was introduced in 2.6.38, and determines if kernel addresses are exposed. This defaults to zero (exposing kernel addresses) since 2.6.39 within the vanilla kernel, although many distributions correctly set the value to 1 (hide from everyone accept uid 0) or 2 (always hide). In addition, this capability also allows the process to view dmesg output, if the dmesg_restrict setting is 1. Finally, the CAP_SYS_ADMIN capability is still permitted to perform syslog operations itself for "historical reasons".

CAP_WAKE_ALARM: Starting in Linux 3.0, this allows a process to trigger a wake up of the system, in addition to setting the CLOCK_REALTIME_ALARM and CLOCK_BOOTTIME_ALARM timers. See CAP_BLOCK_SYSPEND and the Linux Weekly News article "Waking systems from suspend"[93] for more information.

CAP_NET_BROADCAST: This capability is currently marked as "unused", and is not referenced outside of the capability kernel header file. If used, this would allow a process to generate socket broadcasts or listen to multicasts.

[92] See drivers/tty/vt/vt_ioctl.c for more information.
[93] https://lwn.net/Articles/429925/

5.4 Exploring Capabilities

The capset(2) and capget(2) Linux syscalls can set or get thread capabilities through defined structures on specific process IDs. The capget(2) syscall can probe the capabilities of any process within the PID namespace (this can also manually be parsed by decoding values from the status entry of any PID in /proc).

To avoid directly using these system calls, the libcap–ng library by Steve Grubb is "intended to make programming with POSIX capabilities much easier than the traditional libcap library[94]". This library includes the filecap utility to analyze all the currently running applications and print all retained capabilities. This library also includes other helpful utilities for printing capabilities for running processes (pscap), testing capabilities (captest) and a network related processes using capabilities (netcap).

The libcap library offers a simple interface for, and example utilities, for launching processes with specific capabilities (such as capsh -drop, illustrated above). Automatically inheriting or controlling capabilities can be performed in several ways. Either through systemd via the CapabilityBoundingSet directive, or via PAM (Pluggable Authentication Modules), this can be used by system administrators to limit access. As part of the libcap library "pam_cap", can grant capabilities to a users' inherited set.[95]

A simple capabilities example can be demonstrated through the /bin/ping command, a classic and ever-present setuid-root binary. This helps illustrate exactly why capabilities are a good security model, as /bin/ping should only need the network capabilities required to function, namely, RAW sockets.[96]

Our example starts by dropping the setuid root permission, easily done by simply copying the ping binary to a new location as a low-rights user:

```
$ ls -l /bin/ping
-rwsr-xr-x 1 root root 44168 Nov  7  2015 /bin/ping
$ cp /bin/ping /tmp/
$ ls -l /tmp/ping
-rwxr-xr-x 1 aaron aaron 44168 Nov 25 13:58 /tmp/ping
```

Attempting to ping the localhost address using the newly placed binary will result in a permission denied error for socket() with SOCK_RAW, as illustrated by strace:

```
$ strace -e socket /tmp/ping 127.0.0.1
socket(PF_INET, SOCK_RAW, IPPROTO_ICMP) = -1 EPERM (Operation not permitted)
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
ping: icmp open socket: Operation not permitted
...
```

Now, with sudo access (or if granted CAP_SETFCAP) and the setcap command, we can use extended filesystem attributes to add the CAP_NET_RAW capability to the new non-suid root /bin/ping binary. Using the getcap command to list the files capabilities, we can see this was successful:

```
$ sudo setcap cap_net_raw=p /tmp/ping
$ getcap /tmp/ping
/tmp/ping = cap_net_raw+p
```

[94] https://people.redhat.com/sgrubb/libcap-ng/
[95] https://kernel.googlesource.com/pub/scm/linux/kernel/git/morgan/libcap/+/libcap-2.24/pam_cap/capability.conf
[96] If you're wondering why SOCK_RAW is required for ICMP echo requests, see this write-up from 1996: http://www.tldp.org/LDP/khg/HyperNews/get/khg/18/1.html.

Using the new ping binary now works as a low rights user, without suid, and given any vulnerability within the ping binary itself, even complete arbitrary code execution, will only result in the capabilities provided by CAP_NET_RAW (opposed to the entire set of root permissions):

```
$ /tmp/ping -c1 localhost
PING localhost (127.0.0.1) 56(84) bytes of data
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.047 ms
```

In a slightly more complex example, we can grant the tcpdump command the ability to sniff packets[97] without requiring any root access:

```
$ sudo setcap cap_net_raw,cap_net_admin=eip /usr/sbin/tcpdump
$ /usr/sbin/tcpdump -i lo
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on lo, link-type EN10MB (Ethernet), capture size 65535 bytes
...
```

Using the proc filesystem, processes can easily list other processes' capabilities (within the same PID namespace or procfs mount) using the /status file within any /proc/<pid> directory. The example below lists the capabilities of the grep process itself, then using capsh, decodes the bitmask fields within the status file to a more human readable format.

```
$ grep ^Cap /proc/self/status
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000001fffffffff
$ capsh --decode=0000001fffffffff
0x0000001fffffffff=cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,
cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable
,cap_net_bind_service,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,
cap_ipc_owner,cap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,
cap_sys_pacct,cap_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,
cap_sys_time,cap_sys_tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit
_control,cap_setfcap,cap_mac_override,cap_mac_admin,cap_syslog,35,36
```

By using the getpcaps command from libcap2, we can easily inspect the capabilities of a process or set of processes within the system, such as dhclient:

```
$ ps ax | grep dhclient
2223 ?        Ss     0:04 dhclient -v eth1
$ getpcaps 2223
Capabilities for `2223': =
cap_dac_override,cap_net_bind_service,cap_net_admin,cap_net_raw,cap_sys_module+ep
```

The capsh utility provided by libcap2 includes a simple feature for launching applications and dropping capabilities appropriately. In the example below, all capabilities are dropped before executing /bin/sh. A ping command illustrates how supposedly even root/uid 0 cannot use RAW sockets unless it retains the

[97] Note that tcpdump can also execute arbitrary commands via the -z flag.

capability. Inspecting our permitted, inherited, and effective sets lists no permissions at all. Although all root capabilities are dropped, root-owned files and directories can still be accessed (such as /etc/shadow).

```
sudo capsh --drop=all --secbits=1 --
$ id
uid=0(root) gid=0(root) groups=0(root)
$ ping -c1 localhost
ping: icmp open socket: Operation not permitted
$ grep ^Cap /proc/self/status
CapInh: 0000000000000000
CapPrm: 0000000000000000
CapEff: 0000000000000000
CapBnd: 0000000000000000
$ cat /etc/shadow
root:$6$fRJmXGaQ$gd6N9vv....:16409:0:99999:7::
...
```

If we use capsh to grant the correct limited capability of CAP_NET_RAW, now the ping command works.

```
$ sudo capsh --inh="cap_net_raw=+ep" --drop=all --
$ id
uid=0(root) gid=0(root) groups=0(root),127(pkcs11)
$ ping -c1 localhost
PING localhost (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.074 ms
...
```

5.5 Capabilities and User Namespaces

Before reviewing the default capabilities below, some readers may first be wondering about the intersection of Linux capabilities and user namespaces, which allow for a pseudo-root inside of the container (discussed further in Section 8.1 on page 66). If all capabilities are granted to root users, and for many purposes a user appears to be "root" within a user namespace, can this be leveraged to escape the container or used to impact other namespaces? Fortunately, these somewhat conflicting security models hold up. Executing a setuid-root program within a user namespace still gives a process all capabilities, but it's limited within such a namespace (at least, that is the security design).

> **"The child process created by clone(2) with the CLONE_NEWUSER flag starts out with a complete set of capabilities in the new user namespace. Likewise, a process that creates a new user namespace using unshare(2) or joins an existing user namespace using setns(2) gains a full set of capabilities in that namespace. On the other hand, that process has no capabilities in the parent (in the case of clone(2)) or previous (in the case of unshare(2) and setns(2)) user namespace, even if the new namespace is created or joined by the root user (i.e., a process with user ID 0 in the root namespace)."**
>
> - The user_namespaces(7) man page

This is also the case, according to the manual page documentation, for any operation that may affect other namespaces or otherwise allow namespace reassociation. As user namespaces have their own full set of capabilities, and those capabilities can interact within their namespace, affecting or joining other namespaces (at least via capabilities) is only permitted if the process in question retains the CAP_SYS_ADMIN in the target namespace. While capabilities are indeed an important area of modern containers (in addition

to more general Linux hardening opportunities), user namespaces as discussed earlier create a very unique situation. However, while it is intended that the user namespace will be restricted to other namespaces within a container, vulnerabilities may be uncovered in this semi-conflicting security model.[98] Jump to Section 8.1 on page 66 for more information on user namespaces and Section 10 on page 98 for recommendations which cover capabilities for privileged containers and use of user namespaces.

5.6 Capability Defaults In Modern Containers

When examining the defaults table on the following page, it is important to keep in mind the goals of each container platform. LXC is understood to commonly run entire virtual operating systems, and expects the administrator to tune the templates appropriately – not only for security. For Docker, developers are expected to follow the newly established convention of running single applications, also referred to as "App VMs" and Docker itself must appeal to the developer masses by allowing the largest default set of capabilities that does not put the system at a risk. What should be allowed for usability, and what should be restricted for security obviously has been discussed some, back[99] and forth[100] in addition to vulnerabilities.[101] CoreOS Rkt remains under active development and must deal with systemd limitations (or advantages, depending on your perspective) for default or inherited capabilities.

Within LXC, capability defaults largely depend on the template used. For example, Ubuntu retains CAP_SYS_RAWIO while the CentOS template[102] drops it. For the table included below, LXC defaults are sourced from the Ubuntu template, assumed to be the most common Linux distribution. This Ubuntu template includes the base template,[103] from which all LXC templates source their defaults. While LXC retains a large number of capabilities by default, the AppArmor profiles (enabled by default in Ubuntu) largely work as a fallback safety net against attacks leveraging powerful capabilities such as CAP_SYS_ADMIN. This again speaks to the core difference in philosophy of application verses OS containers[104] which can make hardening significantly more difficult.

Docker defaults were recorded from the daemon's default Linux template[105] and although not explicitly covered within this section, the Open Containers specification (runC), is identical to the Docker capabilities list according to the most recent version of the specification.[106] CoreOS Rkt defaults are actually from the systemd-nspawn defaults,[107] as Rkt is almost always deployed with systemd. Additionally, the retained capabilities by Rkt may be different when using LKVM as part of stage 1, but it is not exactly clear at this time, and with the added hardware isolation, they may not be relevant. Finally, with respect to the list on the following page, no attempt has been made within this paper to capture potential capabilities that effectively permit access to yet other capabilities or operations, such as those permitted by CAP_SYS_ADMIN or CAP_NET_ADMIN. The list below is merely a list of defaults, and not a complete vulnerability assessment of these permitted capabilities.

[98] This is likely an area not fully explored, examined or tested. Intersecting security models such as capabilities and user namespaces, could have unforeseen consequences, especially in areas of the kernel not fully namespace aware, or which have vulnerable namespace isolation (such as using the user namespace to gain CAP_NET_ADMIN, according to Andy Lutomirski). Other issues may occur when any container namespace is shared with the host system, such as the network namespace.
[99] https://github.com/docker/docker/issues/5661
[100] https://github.com/docker/docker/issues/5887
[101] http://stealth.openwall.net/xSports/shocker.c
[102] https://github.com/lxc/lxc/blob/master/config/templates/centos.common.conf.in
[103] https://github.com/lxc/lxc/blob/master/config/templates/common.conf.in
[104] While LXC defaults assume a container is a full operating system, they are no less capable that Docker when establishing a secure single-application container.
[105] https://github.com/docker/docker/blob/master/daemon/execdriver/native/template/default_template_linux.go
[106] https://github.com/opencontainers/runc/blob/master/libcontainer/SPEC.md
[107] http://cgit.freedesktop.org/systemd/systemd/tree/src/nspawn/nspawn.c?id=v219#n146

Default Capabilities (red for known high-risk capabilities)			
Linux Capability	**LXC**	**Docker**	**CoreOS Rkt**
CAP_AUDIT_CONTROL	True	False	True
CAP_AUDIT_WRITE	True	True	True
CAP_BLOCK_SUSPEND	True	False	False
CAP_CHOWN	True	True	True
CAP_DAC_OVERRIDE	True	True	True
CAP_FSETID	True	True	True
CAP_FOWNER	True	True	True
CAP_IPC_OWNER	True	False	True
CAP_IPC_LOCK	True	False	False
CAP_KILL	True	True	True
CAP_LEASE	True	False	True
CAP_LINUX_IMMUTABLE	True	False	True
CAP_MAC_OVERRIDE	False	False	False
CAP_MAC_ADMIN	False	False	False
CAP_DAC_READ_SEARCH	True	False	True
CAP_MKNOD	True	True	True
CAP_NET_ADMIN	True	False	False
CAP_NET_RAW	True	True	True
CAP_NET_BIND_SERVICE	True	True	True
CAP_NET_BROADCAST	True	False	True
CAP_SETUID	True	True	True
CAP_SETGID	True	True	True
CAP_SYS_ADMIN	True	False	True
CAP_SETPCAP	True	True	True
CAP_SETFCAP	True	True	True
CAP_SYSLOG	True	False	False
CAP_SYS_BOOT	True	False	True
CAP_SYS_CHROOT	True	True	True
CAP_SYS_NICE	True	False	True
CAP_SYS_RESOURCE	True	False	True
CAP_SYS_RAWIO	True	False	False
CAP_SYS_PACCT	True	False	False
CAP_SYS_MODULE	False	False	False
CAP_SYS_PTRACE	True	False	True
CAP_SYS_TIME	False	False	False
CAP_SYS_TTY_CONFIG	True	False	True
CAP_WAKE_ALARM	True	False	False

5.6.1 Modifying Container Defaults

LXC default capabilities are controlled through the appropriate template configuration, using the `lxc.cap.keep` and `lxc.cap.drop` directives. It is recommended to keep the smallest set of capabilities required by the application via the whitelist approach `lxc.cap.keep` which specifies the capabilities to be retained in the container, and all others are dropped (see the lxc.container.conf manapge for more information). Docker capabilities can be kept and dropped via command line options --cap-add and --cap-drop respectively when launching containers. CoreOS Rkt capabilities can be dropped using the Isolator settings. See the rkt_caps_test.go testing code for example info on setting and using the `capabilities-retain-set`. Finally, more information and recommendations can be found within the security recommendations in Section 10 on page 98.

5.7 A World Without Root

Since the start of capabilities support for processes and executables, Linux has added a set of per-thread `securebits` flags which can be used to disable special handling of capabilities provided to root/UID 0 user. While this can be configured for various levels, the `SECBIT_NOROOT` effectively removes the automatic capabilities provided to root and any suid root owned executables. This creates an environment completely controlled by granted capabilities and in theory, has removed all of the typical power from root,[108] only those capabilities set are granted. Like almost all flags, these are preserved across forks and in the case of `securebits`, the set properties can be ''locked''. Discretionary Access Controls (DAC) and any configured Mandatory Access Control (MAC) are now more important when using `SECBIT_NOROOT`, and care should be taken not to allow lower rights users to execute privileged executables. For example, stripping `tcpdump` of the root requirement is good to avoid remote attack surfaces and local privilege escalation attacks, but it could allow non-root users to sniff traffic (if they can execute the binary).

In addition to `SECBIT_NOROOT` flags, using `PR_SET_NO_NEW_PRIVS` with `prctl(2)` [109] is a great way to further strengthen the principal of least privilege, as also discussed by Kees Cook in his blog post Keeping your process unprivileged. This further and concisely illustrates the benefits and potential to limit programs who need "no new privileges" (NNP), even beyond an `execv(2)` of a setuid binary. Docker in 1.11 has added in support for ''no new privileges'' via security option flags.

Further ''no new privileges'' information and a detailed description can be found within the Linux kernel documentation:

> **''Any task can set no_new_privs. Once the bit is set, it is inherited across fork, clone, and execve and cannot be unset. With no_new_privs set, execve promises not to grant the privilege to do anything that could not have been done without the execve call. For example, the setuid and setgid bits will no longer change the uid or gid; file capabilities will not add to the permitted set, and LSMs will not relax constraints after execve.''**
> - Linux kernel Documentation/prctl/no_new_privs.txt by Kees Cook

It is important to note, the AppArmor child profiles (or nested AppArmor) are unfortunately not currently compatible with NNP. This is due to the lack of any profile ''stacking'' support, such that the switch from the first context to another security context cannot be guaranteed to not expand privileges. This conflict is unfortunately only slightly hinted at in the NNP documentation. For the time being, those wishing to use NNP for Seccomp will have to avoid also using child-profiles within AppArmor.[110]

Overall, Linux capabilities are no silver bullet; special attention should be paid to which capabilities are granted, and unfortunately how those capabilities are implemented or how they allow for multiple privileged operations. It is not trivial to understand or explore which capabilities allow for subsequent privilege or capability escalation. This complexity has lead to container escapes, and other capability vulnerabilities. Despite these risks, capabilities can greatly reduce the potential privilege escalation, help restrict attack surfaces, and limit the impact of successful privileged process exploitation.

[108] Some areas of the kernel may be unaware of SECBIT features, which may introduce privilege-escalation vulnerabilities.
[109] https://lwn.net/Articles/478062/
[110] This problem will hopefully be resolved in the future.

6 Configuration and Basic Use

The following sections explore the basic use and configuration of the three container platforms covered within this paper: LXC, Docker and CoreOS Rkt. As this paper is focused primarily around the security of containers, this section is not intended to explain how to configure and run each system. As such, users should always refer to the specific project's documentation and guidelines, which are also guaranteed to be more up-to-date.

As with many large deployments the configuration, security aspects, threats and recommendations for container management and deployment across a large infrastructure is out of scope. These "orchestration" frameworks (systems such as Openstack and Kubernetes), which help manage software defined data centers are complex and can be misconfigured, insecure or incomplete for containers. Newly released orchestration systems typically involve a complex system that allows for service discovery built using specific tools. This includes offerings such as Docker Swarm, CoreOS Fleet or CoreOS etcd and other systems such as Mesos, Aurora and Apache ZooKeeper. This whitepaper does not explore the security or configuration of container management or orchestration frameworks.

6.1 LXC

LXC is typically configured by hand, if the possibly insecure defaults are to be modified, through editing of the specific container template. Both Libvirt[111] and OpenStack contain support[112] for LXC management which, in addition to other options such as the up and coming LXD "container hypervisor", may offer easier configuration and management. Before starting out, in order to check the system's configuration and support for various kernel and container features, the `lxc-checkconfig` utility can be very helpful. This initial bash script (which examines the kernel's configuration via /proc/config.gz), is typically installed as part of the distributions LXC installation.

6.1.1 The LXC template

The template configuration files control the aspects of an LXC runtime, apart from any command line options. Following typical "UNIXisms", a local (/.config/lxc/) and global (/etc/lxc/) configuration file is used for both privileged and unprivileged containers, often containing the default base template options for networking or uid maps.

Per the lxc.conf manpage: "a basic configuration is generated at container creation time with the default's recommended for the chosen template". These defaults are typically not the most secure configuration for a given container, and often skewed towards treating the container as an entire running system (with various daemons, gettys, users, etc). While the default AppArmor profile and some default protections add some security barriers, additional configuration efforts are strongly encouraged. The section below, along with Section 10.2 on page 105 contains additional security recommendations.

As the security relevant configuration items are listed below, special consideration outside of the listed items, should be performed for any hosting company or any other consumer of third party or user-controlled LXC templates. Documenting and enumerating potential attacks from untrusted LXC profiles is left as an exercise for the reader and is not in scope for this paper. See `lxc.container.conf(5)` for more information on any of the following:

lxc.network.type: The type of network virtualization used. This defaults to a bridge mode, allowing inner-container and container to host networking. Using none will cause the container to share the host's network namespace. This should be avoided if at all possible for a number of reasons, including risks of data snooping, packet crafting, attacks against host traffic or local network. By using veth, a

[111] https://libvirt.org/drvlxc.html
[112] http://docs.openstack.org/liberty/config-reference/content/lxc.html

network bridge is created and attached to a host network device. This bridge is established on the host system prior to the container running. While this allows for trivial container network monitoring, the risk of arbitrary container to host and container to container network communication is critical. Serious cross-container and potentially container to host attacks are possible if a firewall is not applied both at layer 2 (using ebtables, protecting from ARP spoofing attacks) and layer 3 (using iptables, for general networking). If the ebtables layer 2 firewall configuration is cumbersome, consider simply using a separate bridge for each container.

lxc.cgroup.devices.deny: Specifies which devices to deny access to via cgroups. This is typically all devices by using the value a, then explicitly allowing devices using the type, major and minor numbers within `lxc.cgroup.devices.allow`. This whitelist, "default deny" model is strongly recommended and embraces a common security best practice.

lxc.cgroup.devices.allow: Each entry within an LXC template permits the creation of device instances and allows access based on type, major and minor numbers. This typically includes devices such as `/dev/null`, and `/dev/zero`. Note that while exposing `/dev/random` allows for strong random numbers to be generated by the container, it does risk exposing the host's entropy state and pool (allowing for potentially misbehaving or compromised containers to drain the pool, or influence other applications which have blocked read calls). When granting access to devices, keep in mind each device is a path to the kernel, exposes potential vulnerable `ioctls(2)` and if it involves any devices outside of the standard set, should be reviewed extremely carefully.

lxc.mount.auto: This directive specifies kernel filesystems (/proc, /sys) which are automatically mounted within the container. While the complexity is out of scope for this paper, key concerns involve read-only vs read-write. If read-only specifiers cannot be used, mixed (which is partially read and partially write) is strongly preferred over full read-write access.

lxc.cap.drop: This specifies the list of capabilities which are removed during container start-up. As this directive is a blacklist not a whitelist, it is not recommended, see **lxc.cap.keep** below. In many Ubuntu templates, this merely consists of dropping the following capabilities: `sys_module`, `mac_admin`, `mac_override`, and `sys_time`. For Fedora/CentOS, the template drops `mac_admin`, `mac_overrid`, `setfcap`, `sys_module`, `sys_nice`, `sys_pacct`, `sys_rawio` and `sys_time`. Obviously a number of capabilities remain for privileged containers in either case. For more information on capabilities see Section 5.1 on page 30.

lxc.cap.keep: Specifies the list of capabilities to remain for the container, and all other capabilities will be dropped. This option is strongly preferred over the older **lxc.cap.drop** which is a blacklist-style specifier for dropping explicit capabilities such as `CAP_SYS_MODULE`. This approach is not used by any default Ubuntu provided LXC templates. For more information on capabilities see Section 5.1 on page 30.

lxc.aa_profile: Controls the AppArmor profile which is loaded upon starting the container. It is strongly recommended to use a distribution and kernel which support AppArmor, and use a strong a default LXC profile or consider developing a custom per-container profile. The Ubuntu profile, which is applied by default, can be found within: `/etc/apparmor.d/abstractions/lxc/container-base`.

lxc.se_context: This directive sets the SELinux context in which all processes within a container will be restricted to. This is typically generated by hand or using a tool such as refpolicy.

lxc.seccomp: This option specifies a seccomp configuration (starting in version 1.0). Both version one (a whiltelist) and version two (a whitelist or blacklist) are supported. On Ubuntu, the example seccomp profile is a blacklist which blocks `kexec_load`, `open_by_handle_at`, `init_module`, `finit_module`

and `delete_module`. See Section 8.3 on page 74 for more information on seccomp.

lxc.id_map Allows the container to take advantage of the user namespace. This directive specifies a mapping to "shift" all UID values up, typically by tens of thousands. This has an effect of limiting uid 0 within the container, to an arbitrary (typically unprivileged) UID outside in the host. See Section 8.1 on page 66 for more information on the user namespace.

On Ubuntu, unprivileged container rootfs images are often downloaded from http://images.linuxcontainers.org which is actually hosted and run by Stéphane Graber, a core LXC developer.[113] This makes for fast deployment, and testing although with limited transparency or trust (prior to inspecting the rootfs image). If the LXC command-line tools are invoked by a low-rights user, unprivileged containers will be used by default. Such LXC tools are typically command line executables which operate on an abstract per-container UNIX socket and arguments or operations almost always require a container name. Example tools include `lxc-start`, `lxc-stop`, and `lxc-ls` among others.

6.2 Docker

Docker, which encompasses a system daemon running as root, which executes and manages containers on behalf of the Docker Engine that itself is controlled through a Docker client (CLI or REST interface). This trio of software is typically used to download and run different Docker images or create and run images built from Dockerfiles. A Dockerfile is a set of commands and directives which will be interpreted by the Docker Engine. A complete exploration of non-security related Docker options, configuration directives, Dockerfile details, tuning and performance considerations and other advanced configuration directives such as non-default networking or custom volume mounts is outside the scope of this paper. However, more information and a brief security analysis of Docker can be found in Section 9.5 on page 85.

6.2.1 The Dockerfile

A simple plaintext file, with a number of different configuration directives each instrumenting the build process to generate an image and the resulting container itself. For more information see Dockerfile best practices and the Dockerfile reference by Docker. An example Dockerfile is included below, this pulls from the latest ubuntu base image, updates the container and installs the Nginx webserver, then creates a base file and sets the Nginx binary to run when the container starts, exposing port 80 to the host:

```
FROM       ubuntu
MAINTAINER Dade Murphy <dade@cyberdelia.net>

RUN apt-get update && apt-get install nginx
RUN bash -c 'echo "<html>hi</html>" >> /var/www/html/index.html'

CMD    ["nginx", "-extra", "-arguments", "-here"]
EXPOSE 80
```

This Dockerfile can then be built into a Docker image with the `docker build` command, which will parse and run the various Dockerfile directives:

```
$ docker build -t my-nginx-test
```

Once the image is built and the commands within the Dockerfile are run within the container, we can launch as many instances of the image (my-nginx-test) as desired, with each one listening on different TCP ports:

[113] This may or may not be acceptable security risk for some users.

```
$ sudo docker run -p 80:80 --name n-one my-nginx-test
$ sudo docker run -p 8080:80 --name n-two my-nginx-test
```

By using the `docker ps` command, we can see a list of the running containers and then execute commands within them with `docker exec` (provided an environment exists to do so). To easily gain access to the container image, the `--volumes-from` directive can be used (along with the container name) in order to easily browse, modify and read from the image using yet another small container (such as a Busybox shell). While this is appropriate for first users, it is not the recommended practice for editing container contents. Finally, using the `docker exec` command, along with the appropriate arguments for a chosen container, will launch an interactive shell or command inside the Docker container.

6.2.2 Docker Hub

The Docker Hub, a major area of typical Docker use, is similar to github in nature and allows downloading and hosting Dockerfiles, building and testing of images, working within a repository, and collaborating with other Docker users. Docker Hub, often paired with a registry server, is typically operated by Docker themselves hosting the main official base or container images. However, internal company "on premise" solutions or internal registries do exist and are widely used. Internal registries allow cloud and highly risk-averse enterprises or users the benefits of Docker Hub while maintaining specific control over the infrastructure, Docker image hosting and included base images.

The hub can be searched online at https://hub.docker.com or via the command line `docker search` command, as in the following example for "tor":

```
$ sudo docker search tor
NAME                         STARS   OFFICIAL   AUTOMATED   DESCRIPTION
nagev/tor                      20                 [OK]                \
                                   Download and compile Tor software. Starts ...
jess/tor-browser               11                 [OK]
rbubhas/tor-privoxy-alpine     7                  [OK]                \
                                   The smallest (15 MB) docker image with Tor ..
jess/tor                       5                  [OK]
...
```

Docker hub is almost always "behind the scenes" of many Docker examples or images, such as downloading an existing OS image and executing it as a container in the example below:

```
$ sudo docker run -it ubuntu bash
```

6.3 CoreOS Rocket

As Rocket (Rkt) has simplicity as a key design goal, the use of Rkt is typically via command-line options and subcommands to the `rkt` command itself (there is no long-running daemon as with Docker). Rkt itself is an implementation of the CoreOS App Container Specification which is in many ways a competitive standard to key Docker initiatives such as the Dockerfiles and Docker images, in addition to competing with the newly formed Open Container Initiative of which CoreOS is also a member. As with the LXC and Docker sections, a complete exploration of non-security related Rkt options, configuration, implementation details, performance considerations, or LKVM use is outside the scope of this paper.

Rkt supports two configuration files for two major directories, `/etc/rkt`, the local directory and `/usr/lib/rkt`,

confusingly referred to as the "system" directory.

6.3.1 Rkt example

After downloading and installing Rkt, it can quickly be used to download and launch an image provided by CoreOS. The example below downloads and executes the etcd app container, used as a distributed key-value store for service discovery (similar to Apache ZooKeeper and a number of recent options for resilient service discovery). With the exception of `rkt fetch`, all subcommands must be run as root. The example below (borrowed from coreos.com) adds trust for the etcd image from CoreOS:

```
$ sudo rkt trust --prefix coreos.com/etcd
prefix: "coreos.com/etcd"
key: "https://coreos.com/dist/pubkeys/aci-pubkeys.gpg"
gpg key fingerprint is: 8B86 DE38 890D DB72 9186  7B02 5210 BD88 8818 2190
    CoreOS ACI Builder <release@coreos.com>
Trusting "https://coreos.com/dist/pubkeys/aci-pubkeys.gpg" for prefix "coreos.com/
   etcd" without fingerprint review.
Added key for prefix "coreos.com/etcd" at "/etc/rkt/trustedkeys/prefix.d/coreos.com/
   etcd/8b86de38890ddb7291867b025210bd8888182190"
```

After adding trust, we can download the etcd image, verifies it against the trusted certificate and launch it:

```
$ sudo rkt run coreos.com/etcd:v2.0.9
rkt: searching for app image coreos.com/etcd:v2.0.9
prefix: "coreos.com/etcd"
key: "https://coreos.com/dist/pubkeys/aci-pubkeys.gpg"
gpg key fingerprint is: 8B86 DE38 890D DB72 9186  7B02 5210 BD88 8818 2190
    CoreOS ACI Builder <release@coreos.com>
Key "https://coreos.com/dist/pubkeys/aci-pubkeys.gpg" already in the keystore
Downloading signature from https://github.com/coreos/etcd/releases/download/v2.0.9/
   etcd-v2.0.9-linux-amd64.aci.asc
Downloading signature: [=======================================] 819 B/819 B
Downloading ACI: [=============================================] 3.79 MB/3.79 MB
rkt: signature verified:
  CoreOS ACI Builder <release@coreos.com>
2015/08/18 16:10:03 Preparing stage1
2015/08/18 16:10:07 Loading image sha512-91
   e98d7f1679a097c878203c9659f2a26ae394656b3147963324c61fa3832f15
2015/08/18 16:10:08 Writing pod manifest
2015/08/18 16:10:08 Setting up stage1
2015/08/18 16:10:08 Writing image manifest
2015/08/18 16:10:08 Wrote filesystem to /var/lib/rkt/pods/run/6b85a91a-73b9-4f1b-96c2
   -009ae9dc45e1
2015/08/18 16:10:08 Writing image manifest
2015/08/18 16:10:08 Pivoting to filesystem /var/lib/rkt/pods/run/6b85a91a-73b9-4f1b
   -96c2-009ae9dc45e1
2015/08/18 16:10:08 Execing /init
[188748.162493] etcd[4]: 2015/08/18 14:10:08 etcd: no data-dir provided, using
   default data-dir ./default.etcd
[188748.163213] etcd[4]: 2015/08/18 14:10:08 etcd: listening for peers on http://
   localhost:2380
.... snip ....
```

Note that the "signature verified" line above will only be displayed if the image signer is previously marked as trusted using the `rkt trust` command. This may create a security issue if this initial step is bypassed, as the image will still be downloaded and can be run, although the UI will provide negative warnings.

Rkt also has experimental support for the Linux Kernel Virtual Machine (LKVM) within "stage1", a term for the Rkt bootstrapping portion. This can be used in place for Linux Namespaces and cgroups for isolation.[114] This full virtualization offers strong security controls, launching a minimal Linux kernel under KVM which then uses systemd and chroot to start the container within the Virtual Machine. Using the `--stage1-image` flag, Rkt can dynamically load different container images using either LKVM or standard OS virtualization, separately or in parallel. See Getting started with Ubuntu Vivid for more information and examples for using Rkt.

6.3.2 Rkt images

Similar to Docker Images, Rkt uses an "App Container Image"[115] which is downloaded if it does not exist within the local cache. The images are, by default, authenticated using GPG-signed image manifests, which can be trusted by CoreOS signing keys.

Chaining trust in Rkt is mainly established via App Container Image manifest signing (detached GPG signatures). The Signing and Verification of images in Rkt reference within the Rkt github repository explores how to trust new image producers, trusting custom images and generally provides information about the signing and verification options. As CoreOS still supports docker, and Rkt users may want to use Docker images, these can be downloaded directly and are automatically converted to an ACI via `docker2aci`.[116] However, it should be explicitly noted, when using Rkt to run and download a Docker image, historically both HTTPS TLS certificate verification and Docker image verification were disabled, although very recently only image verification is disabled.[117] To this day within Rkt "Docker images do not support signature verification",[118] and the author is not aware of an out of band verification method apart from checking hash values manually.

[114] https://coreos.com/rkt/docs/latest/running-lkvm-stage1.html
[115] https://github.com/appc/spec/blob/master/spec/aci.md
[116] https://github.com/appc/docker2aci
[117] https://github.com/coreos/rkt/issues/912
[118] https://coreos.com/rkt/docs/latest/running-docker-images.html

7 Understanding Container Threats

Container security concerns are almost always at the top of the list for enterprises, those implementing Platforms as a Service (PaaS) or those deploying containers. This is to be expected given their relatively small numbers of production deployment and relative immaturity of some implementations when compared to hardware virtualization. Most importantly, the fundamental risks of a shared kernel and lack of complete kernel namespaces cannot be ignored and deserves real security skepticism. As we will see in the following sections, containers and container deployments face a myriad of different threats, historical weaknesses and other vulnerabilities which must be considered and defended against.

As with other sections of this paper, the risks and potential attacks against container-related components within a deployment are out of scope. This includes management systems, user administration, service discovery, advanced networking and other related items. Each particular container deployment requires specific configuration, security review and consideration with respect to general security principals such as a layered defense, least access and least privilege as well as reviewing authentication, authorization, encryption and trust isolation. These guidelines help secure the deployment of any such system container or otherwise. For recommended solutions, compensating access controls and other security recommendations which help defend against the threats below, see Section 10 on page 98.

7.1 The Linux Kernel Itself

> **"Kernel security wasn't suddenly horrible in 2009, I simply showed how horrible it's always been".**
> -Brad Spengler, grsecurity lead developer

Increased attention on finding and discovering kernel vulnerabilities, in addition to a general increase in security awareness and exploitation techniques continues to lead to a regular stream of public vulnerabilities which likely affect containers asa form of OS virtualization. Beyond typical attacker behavior, exploitation of vulnerabilities in the Linux kernel is now commonly used by both security researchers and every-day users for "jailbreaking" a myriad of smart phones, TVs, routers and other embedded devices which leverage Linux to various degrees. These devices often ship Linux in a "locked-down" form, often without root access, which encourages researchers with varying motivations to discover local kernel vulnerabilities.[119] The majority of this jailbreaking research specifically focuses on arbitrary code execution and privilege escalation, which are direct threats to the security in other platforms as well.

Kernel vulnerabilities can take various forms, from information leaks and Denial of Service (DoS) risks to privilege escalation and arbitrary code execution. Of the roughly 400 Linux system calls, a number have contained privilege escalation vulnerabilities, as recently as of 2016 with `keyctl(2)`. Over the years this included, but is not limited to: `futex(2)`, `vmsplice(2)`, `mremap(2)`, `unmap(2)`, `do_brk(2)`, `splice(2)`, and `modify_ldt(2)`. In addition to system calls, old or obscure networking code including but not limited to SCTP, IPX, ATM, AppleTalk, X.25, DECNet, CANBUS, Econet and NETLINK has contributed to a great number of privilege escalation vulnerabilities through various use cases or socket options. Finally, the "perf" subsystem, used for performance monitoring, has historically contained a number of issues, such as `perf_swevent_init` (CVE-2013-2094).

As recently as last year, loading arbitrary kernel modules via the crypto API[120] was possible, just as it was in 2013.[121] This bypassed many security features, allowing for potentially low-rights users to subvert all secu-

[119] The legality and ethics behind jailbreaking research and related motivations in general, the author would personally argue, are almost always in good faith. However manufacturers, who have various (and sometimes valid) reasons for locking systems down may freely disagree.
[120] https://plus.google.com/+MathiasKrause/posts/PqFCo4bfrWu
[121] https://lkml.org/lkml/2013/3/4/70

rity. Also, the recent CVE-2014-9322 aka BatIRET[122] vulnerability was particularly bad, as it is a key kernel feature. Kernel vulnerabilities, often (with enough effort) resulting in privilege escalation (or at minimum DoS) have also been discovered in filesystem implementations (reiserfs, cifs, binfmt) and different device-specific drivers. While exploitation of the vanilla Linux kernel is slowly becoming more difficult through distribution specific hardening, additional security patches such as grsecurity, or even CPU features (SMEP, SMAP) developers are always adding new complexity and new attack surfaces. When hardening containers, the goal should be to reduce the kernel attack surface however possible; as the kernel should be strongly considered a "known unknown".[123]

As listed above, several privilege escalation vulnerabilities in the kernel have resulted from trivial loading of very old, unmaintained or uncommon packet and socket types (such as Econet, SCTP, ATM, and CAN bus). These network stacks can by dynamically loaded and culminate to a large attack surface, resulting in several high profile and publicly released exploits. Fortunately, versions of Ubuntu Linux starting several years ago have added protections from users loading ancient or uncommon socket/packet types,[124] building on ideas from grsecurity's MODHARDEN patch. For more information on kernel hardening, see Section 10.5 on page 110.

One core threat to kernel security is unfortunately the development team itself, as the Linux kernel development team does not have a great reputation for security prioritization.[125] The kernel team was a winner of the 2009 Pwnie Award[126] for "lamest vendor response",[127] after numerous vulnerabilities were discounted as purely being "Denial of Service" issues. Linus Torvalds, the original creator of the Linux kernel and continued lead developer has opinions sometimes at odds with well understood security principals, believing security researchers are crazy,[128] and security bugs are less important than normal bugs, in addition to numerous commits containing shadow patches or downplayed vulnerabilities[129, 130, 131, 132] going back as far as 2008.[133]

Jon Oberhide's SOURCE Boston presentation Linux Kernel Exploitation: Earning Its Pwnie a Vuln at a Time from 2010 offers a great overview of kernel security, while some of the vulnerability trends and other information is now dated, this presentation is a good kernel security resource. All of that said, the principals of open source still apply, hopefully more vulnerabilities simply means more issues being fixed (hopefully faster than or outpacing new vulnerabilities being introduced). Configuration can help minimize attack surfaces and patches are at least extremely quick when needed. A newly announced effort is fortunately underway to merge long-standing grsecurity exploit mitigation techniques or protections with the vanilla kernel[134] and was covered in the recent Washington Post article The kernel of the argument and Linux Weekly News in Kernel security: beyond bug fixing.

[122] http://labs.bromium.com/2015/02/02/exploiting-badiret-vulnerability-cve-2014-9322-linux-kernel-privilege-escalation/
[123] https://en.wikipedia.org/wiki/There_are_known_knowns
[124] Via the "Blacklist Rare Protocols" modprobe blacklist: https://wiki.ubuntu.com/Security/Features#blacklist-rare-net
[125] https://lwn.net/Articles/313765/
[126] https://en.wikipedia.org/wiki/Pwnie_Awards
[127] http://www.networkworld.com/article/2261206/network-security/twitter--linux--red-hat--microsoft--honored--with-pwnie-awards.html
[128] http://www.networkworld.com/article/2274866/lan-wan/torvalds--fed-up-with-the--security-circus-.html
[129] https://lwn.net/Articles/476947/
[130] http://arstechnica.com/security/2013/05/critical-linux-vulnerability-imperils-users-even-after-silent-fix/
[131] https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=320b2b8de12698082609ebbc1a17165727f4c893
[132] http://tk-blog.blogspot.com/2008/09/linux-kernel-and-silent-fixes.html
[133] http://seclists.org/fulldisclosure/2008/Jul/276
[134] http://www.openwall.com/lists/kernel-hardening/2015/11/05/1

7.2 Exploring Container Threats

As illustrated earlier in this paper, both the namespaces and capabilities systems are still under development or can be considered incomplete. Many kernel features are still not namespace-aware and may present a risk of attack or information exposure. This includes but is not limited to devfs, procfs, system time, kernel ring buffer (dmesg) and LSMs among other minor features such as per UID RLIMITs, pending signals and the max number of processes. New kernel features and requirements for namespaces and containers also risk introducing new vulnerabilities or creating new exploit paths for prior issues. Risks are especially acute with any particularly complex system dealing with namespace isolation, such as the user or network namespaces.

In order to explore how containers can be vulnerable despite consistent efforts, it is first key to understand that, with the exception of using the user namespace, root within a container is the same as root within the host. Privileged users within a privileged container represent the greatest risk to the security assumptions of the container system.[135] The following sections cover a generic overview of prior Linux container escapes or threats to often overlooked cross-container attacks. The sections also explore specific threats to LXC, Docker, CoreOS Rkt and finally cover some indirect threats such as new attack surfaces, and/or malicious images.

7.2.1 Escaping

Escaping the container or, borrowing terms from hardware virtual machines, escaping from the *guest* into the *host*, is typically the worst case scenario from many security perspectives. The following section focuses on understanding and enumerating the root causes used for prior container escapes, as this is one method to help enumerate which container and kernel attack surfaces should be restricted or hardened, and those likely to contain additional yet undiscovered vulnerabilities. The following general list is not ordered in any way, as each threat should be reviewed individually and the specifics of different weaknesses to container escapes may change depending on the deployment or use cases.[136]

- **Lack of user namespaces or privileged with capabilities:** A primary method of escape is simply allowing privileged operations, such as those provided by dangerous capabilities or the single `CAP_SYS_ADMIN` capability. This can also be seen as general lack of user namespaces which (depending on the capabilities list) can effectively undo the vast majority of container hardening, namespaces and protections (and is often, at least within LXC, only contained by careful MAC configuration). For instance, the guest may be able to remount specific system directories critical to security enforcement (cgroups, procfs, sysfs) or the host's devpts can be exposed, allowing the guest to remount it and control it. Capabilities outside of `CAP_SYS_ADMIN` may allow escape onto the local network, raw disks (in order to mount the host disk or boot image) or allow modification of various host settings depending on the granted rights. Finally, user namespaces, as discussed later within this paper reduces or entirely eliminates many of the threats included below (particularly those effecting procfs and sysfs) although it still should always be paired with a MAC solution for improved security.
- **Insecure defaults or a weak configuration:** A container solution which is weakly or insecurely configured by an administrator, or a container which uses insecure defaults, will undoubtedly enable attacks and expose vulnerabilities which can allow for guest escape. This ranges from enabling additional root capabilities and having weak host firewalls or poor cgroup restrictions to simply exposing container host information such as the kernel ring buffer via `dmesg` (which can assist in kernel exploitation or information leaks). For instance, weak cgroup restrictions could allow for local disk access, even within user namespaces and mount restricted namespaces via raw disk, device and `mknod(2)` access.

[135] This fact holds true for any system which is basically intended for least privilege, yet is flexible to handle almost any use-case.

[136] Threats may also shift given different or ever-changing container defaults, and the list included in this paper should be used as a reference for attack surfaces and prior escapes, not as a complete list of all possible threats.

- **Not removing or "dropping" all possible capabilities:** While avoiding a full or normal root user is strongly suggested, not dropping the correct capabilities can easily allow for escape. One example is `CAP_NET_RAW` which can be used to perform network attacks. This capability remains enabled by default in all container platforms, although it it is largely required for ping to function. Likewise, `CAP_READ_DAC_SEARCH` was also a prior source of escape in Docker, also remained enabled by default due to the assumed lack of threat. CoreOS Rkt and LXC retain several security sensitive capabilities, as illustrated in Section 9.13 on page 97.

- **Weak network defaults:** Another source of potential escape comes from default networking, typically allowing unfettered access from the container to host and between containers. This threat often manifests itself via different services which are bound to "all interfaces" (0.0.0.0), which will of course include the bridge interface which is connected to the containers own virtual Ethernet device. This inadvertently exposes different network daemons, such as OpenSSH or unauthenticated Web servers to potentially compromised or malicious containers. This can also allow for cross-container ARP spoofing attacks, further discussed in 7.2.2 on page 55.

- **Unsafe exposure of procfs:** Due to the lack of namespace support, the exposure of `/proc/` offers a source of significant attack surface and information disclosure. Numerous files within the procfs offer a risk for container escape, host modification or basic information disclosure which could facilitate other attacks. Several examples are included below, although it should be noted the following list is not exhaustive, and mainly focuses on the largest risks rather than driver specific mistakes:[137]
  - `/proc/sys/` typically allows access to modify kernel variables, often controlled through `sysctl(2)`. This also contains other sensitive settings, including but not limited to:
    - `/proc/sys/kernel/core_pattern`: This defines a program which is executed on core-file generation (typically a program crash) and is passed the core file as standard input if the first character of this file is a pipe symbol. This program is run by the root user and will allow up to 128 bytes of command line arguments. This would allow trivial code execution within the container host given any crash and core file generation (which can be simply discarded during a myriad of malicious actions)..[138]
    - `/proc/sys/kernel/modprobe`: Controls the path to the "kernel module loader", which is called when loading a kernel module such as via the `modprobe` command, and may lead to trivial privilege escalation and/or escape from the container.
    - `/proc/sys/vm/panic_on_oom`: Will instantly trigger a kernel panic when encountering an Out of Memory (OOM) condition. This is more of a Denial of Service (DoS) attack than container escape, but it no less exposes an ability which should only be available to the host.
  - `/proc/config.gz` depending on `CONFIG_IKCONFIG_PROC` settings, this exposes a compressed version of the kernel configuration options for the running kernel. This may allow a compromised or malicious container to easily discover and target vulnerable areas enabled in the kernel.
  - `/proc/sysrq-trigger`: Sysrq is an old mechanism which can be invoked via a special "SysRq" keyboard combination. This can allow an immediate reboot of the system, issue of `sync(2)`, remounting all filesystems as read-only, invoking kernel debuggers, and other operations. If the guest is not properly isolated, it can trigger the sysrq commands by writing characters to this file, such as: `echo "b" >/proc/sys/kernel/sysrq` to reboot the host.

[137] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=8040835760adf0ef66876c063d47f79f015fb55d
[138] http://seclists.org/oss-sec/2011/q4/158

- `/proc/kmsg`: The kmsg file can expose kernel ring buffer messages typically accessed via `dmesg`. Exposure of this information can aid in kernel exploits, trigger kernel address leaks (which could be used to help defeat the kernel Address Space Layout Randomization (KASLR)), and be a source of general information disclosure about the kernel, hardware, blocked packets and other system details.

- `/proc/kallsyms`: This contains a list of kernel exported symbols and their address locations for dynamic and loadable modules. This also includes the location of the kernel's image in physical memory, which is helpful for kernel exploit development. From these locations, the base address or offset of the kernel can be located, which can be used to overcome kernel Address Space Layout Randomization (KASLR). For systems with `kptr_restrict` set to "1" or "2", this file will exist but not provide any address information (although the order in which the symbols are listed is identical to the order in memory).

- `/proc/<pid>/mem`: This interface exposes interfaces to the kernel memory device (`/dev/mem`). While the PID Namespace may protect from some attacks via this procfs vector, this area of has been historically vulnerable, then thought safe[139, 140] and again found to be vulnerable[141] for privilege escalation.

- `/proc/kcore`: This special file represents the physical memory of the system and is in an ELF core format (typically found in core dump files). The large reported file size represents the maximum amount of physically addressable memory for the architecture, and can cause problems when reading it (or crashes depending on the fragility of the software). The ability to read this file (restricted to privileged users) can leak memory contents from the host system and other containers.

- `/proc/kmem` and `/proc/mem`: Both pseudo-files are typically restricted interfaces for modifying kernel memory, often also inaccessible via `/dev/kmem` and `/proc/mem` respectively. Either via DAC permission restrictions or MAC protections.

- `/proc/sched_debug`: This special file returns process scheduling information for the entire system. This information includes process names and process IDs from all namespaces in addition to process cgroup identifiers. This effectively bypasses the PID namespace protections and is other/world readable, so it can be exploited in unprivileged containers as well. This issue was discovered by NCC Group's own Jesse Hertz and responsibly disclosed, and resolved.[142]

With the exception of some `/proc/sys/` entries and `/proc/sched_debug`, all of the above files are only readable or writable by the root user. Privileged Docker containers protect from many of these issues by using read-only bind mounts for sensitive procfs files and directories. Other container solutions will use a read-only or mixed permission procfs, specific read-only bind mounts or use the bundled MAC system to prevent unintended reads or writes to sensitive files, in addition to preventing remounting or mounting a new procfs.

Some readers may be thinking "if procfs isn't namespace aware, the user namespace allowing root within in a container could allow access to these sensitive files", however user namespaces provides only a fake UID zero user. Access controls will still be enforced, illustrating the defense-in-depth user namespaces can provide, even if there is a failure in other features, such as bind-mounts or the implemented MAC policy. See Section 8.1 on page 66 for more information and an example.

- **General information disclosure:** May enable further exploitation of related systems, allow probing of firewall rules, disclose information on running processes and other details which support probing of the host. This

[139] https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=198214a7
[140] https://lwn.net/Articles/433326/
[141] http://blog.zx2c4.com/749
[142] https://github.com/docker/docker/pull/21263

can come from several sources:

- As illustrated within the procfs threats section above (7.2.1 on page 52), various files within the procfs, expose information that can be used to perform yet other attacks or learn information about the system. This includes discovering the location of the container rootfs or image (such as /proc/<pid>/mountinfo), exposing kernel memory or the kernel configuration, listing running processes across all namespaces, and other informational issues which may facilitate exploitation or escape.
- Kernel ring buffer exposure, via `dmesg` exposure. Additionally, if not restricted using capability dropping (`CAP_SYSLOG` and `CAP_SYS_ADMIN`), it may also allow clearing of the host kernel buffer.[143]
- Access and exposure of kernel keyrings[144] is a known Kernel Namespace weakness for Linux Containers which is to be documented for Docker.[145] This may leak or allow tampering with the key material for disk encryption or network authentication such as Kerberos. Syscalls such as `add_key`, `request_key`, and `keyctl(2)` can also be blocked in order to prevent access from userspace applications.
- Exposure of sysfs, including `/sys/kernel` and `/proc/sys/kernel/` exposes a vast array of host information, many files are readable by all users, allowing unprivileged users to gain knowledge of network interfaces, hardware devices and a wide array of system state information.
- Debugfs mounted on `/sys/kernel/debug` can allow for tracing of some kernel details, risks potential attack surfaces and exposes various user-space information. Kernel developers can expose an unknown amount of information within the various traces. See An updated guide to debugfs by LWN for more information.

- **Exposure or lack of protections for sysfs:** Similar to the procfs (essentially the same as /proc/sys/), the sysfs interface can leak information about the host kernel, hardware within the system and the attack surface can be used for various attacks against the host. Access to the following files should be considered a risk:
  - The `uevent_helper` trivially allows a given binary within the Guest on the Host. Exploitation may be easier on LXC, given the known path to the container rootfs (either through disclosure or looking in /proc/pid/mountinfo) and trivial or predictable filesystem layout. The path to the executable is placed in "uevent_helper", similar to the example below which will invoke evil-helper.sh within the host:

```
echo /var/lib/lxc/test/rootfs/tmp/evil-helper > /sys/kernel/uevent_helper
echo change > /sys/class/mem/null/uevent
```

  - `/sys/class/thermal`: Access to ACPI and various hardware settings for temperature control, typically found in laptops or gaming motherboards. This may allow for DoS attacks against the container host, which may even lead to physical damage.
  - `/sys/kernel/vmcoreinfo`: This file can leak kernel addresses which could be used to defeat KASLR.
  - `/proc/sys/security`: The securityfs area of sysfs, this primarily controls the configuration and use of various Linux Security Modules (LSMs) such as AppArmor or SELinux. Mount-over attacks typically target this area of sysfs, and vulnerabilities with successful exploits have occurred in past LXC and Docker versions.
  - `/sys/firmware/efi/vars`: Exposes interfaces for interacting with EFI variables in NVRAM. While this is

[143] As Docker discovered in 2014: https://github.com/docker/docker/issues/5491.
[144] https://lwn.net/Articles/639523/
[145] https://github.com/docker/docker/issues/10939

not typically relevant for most servers, EFI is becoming more and more popular. Permission weaknesses have even lead to some "bricked" laptops.[146]

- /proc/sys/fs/binfmt_misc: Allows executing "miscellaneous binary formats", which typically means various interpreters can be registered for non-native binary formats (such as Java) based on their magic number. While this path is typically writable by AppArmor rules, NCC is not aware of any exploits, although it is not likely required for most container applications.

- **Networking exposure due to shared host network namespaces:** While this is a documented weakness[147] shared networking can be desirable for many deployment scenarios (such as having a container add network routes or modify routing information) there are risks to localhost-bound services (assuming trusted callers) and non-TCP or UDP connections, such as abstract namespaces. As discussed elsewhere within this paper, skipping a single namespace within a container deployment can have serious security side-effects and can subvert the security of other namespaces.

7.2.2 Cross-container Attacks

While threats to the host may present the largest risk for lateral movement or additional compromise in deployed systems, cross-container attacks may be equally significant. Attacks between containers on the same host or same local network may enable more direct vulnerabilities or not require weaknesses within the container implementation itself. For example, compromising the database container on the same host due to a weak password, or using ARP spoofing to capture credentials from a neighboring container may achieve an attackers goals without using a container escape. The following section explores potential attacks between two or more containers on the same host or network (typically the same broadcast domain).

Without user namespaces or running each container within a different user id (uid/gid), containers separated by a PID namespace may still conflict with resource limits (such as the global user struct) and potentially other more security-sensitive areas of the kernel not yet namespace aware. However these issues are often restricted to DoS attacks, in the same way filling a shared disk or volume mount, using all task resources via a forkbomb degrade the host system.

As discussed earlier, cross-container attacks may largely be enabled through weak network defaults. This is due to the common and typically default "bridge" configuration, allowing the host system to act as a switch for arbitrary cross-container and container to host networking. Both Docker,[148] LXC and Rkt allow for cross-container communication without it being explicitly enabled. Linux bridges are susceptible to ARP spoofing, just like most real-world switches. This is especially the case when the `CAP_NET_RAW` capability is retained by a container, the default for all containers or when user namespaces are in use, which re-grants all capabilities (clearly the intent to only interact within their own namespaces can be violated). This is considered a "known issue" for LXC,[149] and has been referenced in some prior presentations.[150]

Resource limits, beyond required storage space limits, are typically not a default configuration item for any container system. DoS attacks may happen indirectly via connected systems, or are a result of framework vulnerabilities, such as the "billion laughs" attack in XML. DoS can also be a result of misbehaving applications or innocuous application misconfiguration. These risks are further explored in 7.2.4 on page 57. Beyond shared resources, if any mount point or even underlying partitions are shared between containers, it may allow a compromised or malicious container to influence other systems.

[146] http://linux.slashdot.org/story/16/02/01/1357237/running-rm--rf--is-now-bricking-linux-systems
[147] https://github.com/docker/docker/issues/14767
[148] https://docs.docker.com/articles/networking/#between-containers
[149] https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1548497
[150] http://events.linuxfoundation.org/sites/events/files/slides/secure-lxc-networking.pdf

Local network access also may expose container management systems, which may indirectly then trigger cross-container attacks by targeting known information, such as a hostname or IP address of target containers. Unfortunately enumerating the different systems and threats are outside of the scope of this white paper. When performing an assessment of these systems, consider how different attacks may be enabled if direct access to any service discovery, orchestration and management systems is gained (either through a host escape or local network attacks). The majority of these systems offer poor or missing authentication, authorization or do not use transport layer security such as TLS.

7.2.3 Inner-container Attacks

Before a compromised or malicious container can attempt to access the host or target different cross container vulnerabilities, an attacker may be required to first gain unauthorized access to the container itself. This will typically manifest itself via non-container related flaws, such as plain web application vulnerabilities, weak administration controls, command injection, arbitrary file writes, and even memory corruption or other vulnerabilities. The following inner-container attacks section enumerates attacks within the container itself, with the explicit goal of either gaining additional privilege, violating the security assumptions of the provided application or platform, and obtaining a system foothold or reverse shell. This foothold can provide access or other information supporting yet other attacks, such as those focused on the local shared kernel. Exploitation of these vulnerabilities may or may not be required for cross-container attacks or escaping to the host, depending on the application and container configuration.

- **Out of date software:** One of the primary methods for container compromise may be due to simply not keeping the container up to date or not deploying new containers when vulnerabilities are discovered. This may risk known and pre-packaged exploits being easily used by attackers in drive-by attacks or advanced attackers targeting complex but known vulnerabilities for a high value target.

- **Exposing the container to insecure or untrusted networks:** Exposure risks attacks against other container systems given a single "bad apple". This occurs from not isolating containers by trust via their attack surfaces or hardening. Not limiting access to unique network zones or accessibility as well as limiting container security based on application content (such as PII or Payment related). All of these architecture or design risks can subvert a strong security model and otherwise exacerbate the impact of any compromise.

- **Use of large base images:** The tendency or default to use large base images allows for a number of potential attacks. This greatly increases the "tools" available for compromise indication (that is, letting the attacker know their exploit was successful), or remote access. The inclusion of interpretive languages (Python, Ruby, Perl, PHP, etc) can also be a source of almost unlimited exploit tooling and development. Finally, large base images may require more patching, which on production critical systems, if a system is not place for ephemeral images, many vulnerabilities may be found within a single long-running container. This issue is exacerbated by thinking of containers as "sealed systems" when that is only one part of the security problem.

- **Weak application security:** The application within the container may contain any number of vulnerabilities which allow an attacker to gain a foot-hold into the container or expose the very information without performing any container attacks (such as exposing sensitive system log files which contain leaked cloud authentication tokens, such as AWS access keys).

- **Lack of user namespaces or use of the actual root user:** This may allow a single vulnerable application in a container or such an application within a container OS to be more easily exploited to gain a foothold onto the system, network or tamper with files to remain persistent within the container. Privileged attacks can then be explored and attempted against the other processes within a container, the host or other containers.

7.2.4 Denial of Service and Resource Consumption

Apart from specific security attacks against the host, other guest containers or the container itself, Denial of Service attacks against the system can be surprisingly powerful and effectively have the similar business or downtime impacts to exploitation when successful. While other attacks receive the majority of security focus, DoS attacks are extremely effective, and often employed by threat actors. These attacks are often used either as a direct means of attack, with the sole purpose of disruption, or as a cover for additional more serious exploit targets. DoS attacks in a container environment may include, but not be limited to attacks such as:

- **Infinitely replicating or duplicating functions:** Classic forkbombs are a trivial attack given any ability to write code (either interpreted, native or otherwise) in order to quickly consume hardware resources (typically CPU and Memory). A great bash example, which should always be tested carefully, is the following recursive function: `(:(){ :|:& };:)`
- **Direct exposure of hardware resources:** Any hardware devices which are directly exposed to the container present an opportunity for DoS attack, assuming the container has such access legitimately. This includes unbound CPU, memory, network or disk access.
- **Indirect exposure of hardware resources:** This could include network mapped drives, container management, service discovery solutions or other systems connected on the same network. If an attacker can generate a large amount of requests, even for innocuous requests or files, it may amplify logging information or local network traffic to specific systems, and indirectly create a DoS.
- **Random devices:** Exposing the strong cryptography PRNG `/dev/random` to malicious or compromised containers can lower or deplete the kernel's entropy pool.[151] While the need for strong randomness during key generation is required, a compromised container with access to `/dev/random` can disrupt, slow down or possibly entirely halt future cryptographic operations (due to blocking behavior) within the same container host by issuing a large number of read calls to the device. Note that this potential DoS can also be performed using the new `getrandom(2)` syscall, even if `/dev/random` is not exposed by cgroups.

7.2.5 The General Problem of New Code

As with any software, introducing new code risks new and undiscovered mistakes or bugs, where undoubtedly some will manifest themselves as security vulnerabilities. Vulnerabilities can either be from incorrect design, implementation errors, or unexpected interactions with existing code. While the likelihood of bugs is a known constant by security professionals, the risk is increased when the code modifications deal with sensitive areas of a codebase such as privilege or access control. In order to illustrate this point, a short list of public CVE vulnerabilities stemming from new security features (with goals such as reduced root capabilities) is included below.

- **Vulnerabilities involving privileged capabilities include but are not limited to:**
  - CVE-2014-7975: The `do_umount` function in fs/namespace.c in the Linux kernel through 3.17 does not require the `CAP_SYS_ADMIN` capability for `do_remount_sb()` calls that change the root filesystem to read-only, which allows local users to cause a denial of service (loss of writability) by making certain unshare system calls, clearing the MNT_LOCKED flag, and making an MNT_FORCE `umount(2)` system call.
  - CVE-2013-4588: Multiple stack-based buffer overflows in net/netfilter/ipvs/ip_vs_ctl.c in the Linux kernel before 2.6.33, when CONFIG_IP_VS is used, allow local users to gain privileges by leveraging the

[151] See `/proc/sys/kernel/random/entropy_avail` for the current pool size.

CAP_NET_ADMIN capability for (1) a `getsockopt(2)` system call, related to the `do_ip_vs_get_ctl()` function, or a `setsockopt(2)` system call, related to the `do_ip_vs_set_ctl()` function.

- CVE-2013-6383: The `aac_compat_ioctl()` function in drivers/scsi/aacraid/linit.c in the Linux kernel before 3.11.8 does not require the `CAP_SYS_RAWIO` capability, which allows local users to bypass intended access restrictions via a crafted `ioctl(2)` call.
- CVE-2011-2517: Multiple buffer overflows in net/wireless/nl80211.c in the Linux kernel before 2.6.39.2 allow local users to gain privileges by leveraging the `CAP_NET_ADMIN` capability during scan operations with a long SSID value.
- CVE-2011-1019: The `dev_load()` function in net/core/dev.c in the Linux kernel before 2.6.38 allows local users to bypass an intended `CAP_SYS_MODULE` capability requirement and load arbitrary modules by leveraging the `CAP_NET_ADMIN` capability.

- **Vulnerabilities within namespaces themselves:** It is important to keep in mind, several of the issues below likely derive from namespaces and capabilities being "added-on" later within the kernel as opposed to building them in from the ground up (obviously other more important things were on Linus' mind back in 1991). The vast majority of issues included below are related to user namespaces themselves. With the somewhat recent addition (stable in 3.8) of user namespaces, a number of vulnerabilities have been discovered. It is an unfortunate reality to see a component intended to add security for containers through reduced privileges turned around, in order to gain unauthorized privilege. Exploits may come from local attackers in the host or compromised applications. These threats against namespaces, and specifically attacks using user namespaces, often originate from malicious actions on systems without any containers or with only a partial implementation thereof. This largely impacts or affects administrators without the knowledge of user namespaces being enabled or without the intent to use containers or user namespaces. See Anatomy of a user namespaces vulnerability for one in-depth exploration by Linux Weekly News (LWN).

  > **"We're looking back on three years of vulnerabilities around `CLONE_NEWUSER` with no end in sight, and we have an obligation to help the end users that don't want to be exposed to this any more."**
  > - kernel-hardening post by Kees Cook

  As user namespace vulnerabilities often present themselves outside the context of containers, or on systems where other namespaces are not applied, distributions have taken to adding custom sysctl patches[152] in order to allow for user namespaces to be disabled (without having a custom kernel be applied[153]). Aware of these developments and distribution tweaks kernel security developers, such as Kees Cook, have proposed an official patch to allow privileged users to disable user namespaces for non-container systems.[154] Significant dissent followed[155, 156] which was well summarized in the Controlling access to user namespaces article by LWN. Finally, there are also potential patches for adding a new user namespace specific capability (`CAP_SYS_USER_NS`[157]) which also may help resolve this problem, although it is likely problematic as well.[158]

  - CVE-2013-1956: The `create_user_ns()` function in kernel/user_namespace.c in the Linux kernel be-

[152] Such as adding a `kernel.unprivileged_userns_clone` patch
[153] http://www.openwall.com/lists/kernel-hardening/2016/01/23/8
[154] http://www.openwall.com/lists/kernel-hardening/2016/01/22/21
[155] http://www.openwall.com/lists/kernel-hardening/2016/01/24/10
[156] http://www.openwall.com/lists/kernel-hardening/2016/01/25/11
[157] https://lkml.org/lkml/2015/10/17/94
[158] http://www.openwall.com/lists/kernel-hardening/2016/01/25/16

fore 3.8.6 does not check whether a chroot directory exists that differs from the namespace root directory, which allows local users to bypass intended filesystem restrictions via a crafted clone system call.

- CVE-2013-1957: The `clone_mnt()` function in fs/namespace.c in the Linux kernel before 3.8.6 does not properly restrict changes to the MNT_READONLY flag, which allows local users to bypass an intended read-only property of a filesystem by leveraging a separate mount namespace.
- CVE-2013-1959: kernel/user_namespace.c in the Linux kernel before 3.8.9 does not have appropriate capability requirements for the uid_map and gid_map files, which allows local users to gain privileges by opening a file within an unprivileged process and then modifying the file within a privileged process. At least one public exploit is easily found[159] or could be created.
- CVE-2013-1858: The clone system-call implementation in the Linux kernel before 3.8.3 does not properly handle a combination of the `CLONE_NEWUSER` and `CLONE_FS` flags, which allows local users to gain privileges by calling chroot and leveraging the sharing of the / directory between a parent process and a child process.
- CVE-2014-4014: The capabilities implementation in the Linux kernel before 3.14.8 does not properly consider that namespaces are inapplicable to inodes, which allows local users to bypass intended chmod restrictions by first creating a user namespace, as demonstrated by setting the setgid bit on a file with group ownership of root.
- CVE-2014-5206: The `do_remount` function in fs/namespace.c in the Linux kernel through 3.16.1 does not maintain the `MNT_LOCK_READONLY` bit across a remount of a bind mount, which allows local users to bypass an intended read-only restriction and defeat certain sandbox protection mechanisms via a "mount -o remount" command within a user namespace.
- CVE-2014-5207: fs/namespace.c in the Linux kernel through 3.16.1 does not properly restrict clearing `MNT_NODEV`, `MNT_NOSUID`, and `MNT_NOEXEC` and changing `MNT_ATIME_MASK` during a remount of a bind mount, which allows local users to gain privileges, interfere with backups and auditing on systems that had atime enabled, or cause a DoS (excessive filesystem updating) on systems that had atime disabled via a "mount -o remount" command within a user namespace.[160]
- CVE-2014-7970: The `pivot_root` implementation in fs/namespace.c in the Linux kernel through 3.17 does not properly interact with certain locations of a chroot directory, which allows local users to cause a denial of service (mount-tree loop) via . (dot) values in both arguments to the `pivot_root` system call.
- CVE-2015-2925: Was an interesting and some may argue unexpected vulnerability that effected both Linux containers via bind mounts and OpenVZ, which allowed escaping from bind mounts by using a similar method to a double-chroot attack.[161]
- CVE-2015-4176, CVE-2015-4177, and CVE-2015-4178: As well explained by Kernel developer Eric W. Biederman: *"An unprivileged user could call umount(MNT_DETACH) and in the right circumstances gain access to every file on essentially any filesystem in the mount namespace. So in a kernel with user namespaces enabled and you are running a sandbox like Docker that has a real root user inside. That root user could with a little work remove every ro bind mount on top of proc. Such as /proc/sys/. Allowing a user that simply has uid 0 and no caps access to do all kinds of interesting things."*

[159] http://pastebin.com/8vQnNtfZ
[160] Testing code: http://www.openwall.com/lists/oss-security/2014/08/13/9
[161] http://www.openwall.com/lists/oss-security/2015/04/03/7

- CVE-2015-1328: Discovered by Philip Pettersson, where a privilege escalation vulnerability when using overlayfs mounts inside of user namespaces. A local user could exploit this flaw to gain administrative privileges on the system. Exploits can be found for many versions of Ubuntu Linux.[162]
- CVE-2014-8989: The Linux kernel through 3.17.4 does not properly restrict dropping of supplemental group memberships in certain namespace scenarios, which allows local users to bypass intended file permissions by leveraging a POSIX ACL containing an entry for the group category that is more restrictive than the entry for the other category, aka a "negative groups" issue, related to kernel/groups.c, kernel/uid16.c, and kernel/user_namespace.c. The LWN article User namespaces and setgroups() has more information, in addition to a post on OSS-Security.[163]
- CVE-2016-4997: Allows for kernel memory corruption, leading to elevation of privileges or kernel code execution. This occurs in a call that is normally restricted to root, however, Linux Kernels that support user and network namespaces can allow an unprivileged user to trigger this functionality.
- CVE-2016-4998: Resulted in out of bounds heap memory access, leading to a Denial of Service (or possibly heap disclosure or further impact).

7.2.6 Attacks Against The Host Container Management

While exploring all threats or attacks against container management (such as the LXC configuration suite, command line utilities, installation tools, the Docker daemon, REST API and other related software) is largely out of scope for this paper, it is important to understand prior vulnerabilities. This may help when deploying containers to consider trust for container images, Dockerfile parsing or building and general configuration hardening. In some cases, exploitation may take the model of a confused deputy attack. For example, the host container software mistakenly trusts the containers own rootfs data (such as supplying an empty AppArmor ruleset), which can be seen as the basic risk of untrusted input and requirement of data validation. Docker in particular has contained a number of vulnerabilities related to image processing or verification.

As a meta-container management issue, for container hosts in cloud environments, restricting access from the container to upstream metadata endpoints (such as 169.254.169.254 for AWS) should be added to any host firewall rules or other access restrictions. These systems can leak sensitive information which is typically used for host management itself.

The following section briefly examines the configuration, use, or trust of host container platforms to potentially compromised container images. Weaknesses within this section often revolve around vulnerabilities within the hardening and security configuration of various containers. As with similar sections within this container threats exploration, the following list should not be considered exhaustive, but is intended to provide an idea of the types of prior vulnerabilities and risks of handling untrusted image contents or performing operations on potentially untrusted containers. Finally, it should be noted while LXC and Docker are the only examples listed below, a lack of Rkt vulnerabilities or CVEs should not indicate a lack of vulnerabilities, only a reduced attack surface and likely a lack of security research.

- **Weaknesses from a procfs unmount, overwrite or overmount in Docker and LXC:** These simple attacks have proven effective against SELinux, AppArmor, Docker and LXC (among other software) on several occasions. For example CVE-2015-1334 allows for a fake procfs to bypass SELinux domain transitions: *"A malicious container can create a fake proc filesystem, possibly by mounting tmpfs on top of the container's /proc, and wait for a lxc-attach to be ran from the host environment. lxc-attach incorrectly trusts the container's /proc/PID/attr/current,exec files to set up the SELinux domain transitions, which may result in no*

[162] https://www.exploit-db.com/exploits/37292/
[163] http://www.openwall.com/lists/oss-security/2014/11/17/19

confinement being used." In CVE-2015-3661 Tõnis Tiigi and Eric Windisch discovered the Docker Engine before 1.6.1 allowed local users to set arbitrary Linux Security Modules (LSM) and docker_t policies via an image that allows volumes to override files in /proc.

- **Symlink and Hardlink attacks in Docker:** For CVE-2014-6407 and CVE-2014-6408 discovered by Florian Weimer of Red Hat Product Security and an independent researcher, Tõnis Tiigi. Docker prior to 1.3.2 was vulnerable to extracting files to arbitrary paths on the host during "docker pull" and "docker load" operations. This was caused by symlink and hardlink traversals present in Docker's image extraction. This vulnerability could be leveraged to perform remote code execution and privilege escalation. Docker also allowed security options to be applied to images, allowing images to modify the default run profile of containers executing these images. This vulnerability could allow a malicious image creator to loosen the restrictions applied to a container processes, potentially facilitating a break-out.
- **Chroot strikes back:** For CVE-2014-9357 Tõnis Tiigi found the introduction of chroot for archive extraction in Docker 1.3.2 had introduced a privilege escalation vulnerability. Malicious images or builds from malicious Dockerfiles could escalate privileges and execute arbitrary code as a privileged root user on the host.
- **Symlinks strike Docker again:** For CVE-2014-9356 Tõnis Tiigi discovered path traversal attacks within Docker before 1.3.3 were possible in the processing of absolute symlinks. In checking symlinks for traversals, only relative links were considered. This allowed path traversals to exist where they should have otherwise been prevented. This was exploitable via both archive extraction when building Dockerfiles and through volume mounts.
- **Directory traversal in rootfs:** For CVE-2015-1331 and CVE-2015-1334 discovered by Roman Fiedler. LXC included a directory traversal flaw that allows arbitrary file creation as the root user as a local attacker (non-guest). LXC also allowed a malicious container to create a fake proc filesystem (possibly by mounting tmpfs on top of the container's existing /proc), and wait for a lxc-attach to be executed in the host environment. This will then bypass the expected SELinux or AppArmor enforcement, facilitating other attacks depending on the container configuration.
- **Symlink attacks on LXC:** CVE-2015-1335 allows a malicious container to escape AppArmor confinement via a symlink attack on a (1) mount target or (2) bind mount source. See Roman Fiedler's original report for an example exploit[164] and the summary by Tyler Hicks[165] for a good summary.
- **Symlink attacks strike three for Docker:** In CVE-2015-3627 Tõnis Tiigi found libcontainer and Docker Engine before 1.6.1 opened the file-descriptor passed to the pid-1 process before performing the chroot, which allowed local users to gain privileges via a symlink attack in an image.
- **Sensitive files writable in Docker:** For CVE-2015-3630 Eric Windisch of Docker Security discovered several paths underneath /proc were writable from containers before Docker 1.6.1, allowing global system manipulation and configuration. These paths included /proc/asound, /proc/timer_stats, /proc/latency_stats, and /proc/fs.

7.3 LXC Specific Threats

While LXC retains the most flexibly and configuration choices for a major container platform, it can be undermined by insecure defaults (such as overly-permissive root-capability sets) or default bridge networking. This fundamental or defaults issue may be due to a core philosophy of treating containers as minimal operation

[164] https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1476662
[165] http://www.openwall.com/lists/oss-security/2015/09/29/4

systems as opposed to encompassing a single application container, as with Docker and less so CoreOS Rkt, which is a bit of a middle-ground due to the use of systemd, and other "helper" executables. This philosophy does not support an idea of least privilege or least access, and makes security configuration, through specific AppArmor or Seccomp profiles difficult.

Fortunately for most users, if a non-root user creates containers, the user namespace will be used by default which adds a great degree of defense in depth from container escapes, however the default security issue and core philosophy of OS containers vs App containers remains. In addition to weak defaults, LXC tools (such as `lxc-start` and `lxc-attach`) were recently found to contain a number of critical security flaws, as illustrated by the LXC security analysis performed by Roman Fiedler[166] and posted to OSS-Security by infamous researcher Solar Designer. For additional details on the strengths, weaknesses and risks for LXC, see Section 9.4.

7.4 Docker Specific Threats

While Docker likely has the largest and most diverse user-base of any container system, the default security settings are quite good, especially starting in Docker Engine 1.10 which has support for user namespaces and seccomp-bpf. However, user namespaces are not enabled by default, and the base seccomp support is also implemented as a blacklist not a whitelist. This is likely due to the task of balancing general use cases, historical root-user assumptions for various Docker subsystems, overall security, and strong defaults.

Before exploring other threats, we should first mention using `--privileged` is considered extremely dangerous. Although it can enable some cool tricks[167] which could be used to add defense in depth, this option essentially disables all security:

> **"Docker will enable access to all devices on the host as well as set some configuration in AppArmor or SELinux to allow the container nearly all the same access to the host as processes running outside containers on the host."**
> - Docker command documentation by Docker

A large base image size, implemented in many Dockerfile examples, abstracted through other `FROM` calls or simply through lack of user knowledge can be an overall risk. Large base images not only risks including a large additional attack surface within the underlying system, but risks having the container inherit security vulnerabilities, for which then must be patched. For example, due to default and aggressive dependency requirements for common Ubuntu packages, a high risk but unused application is included within container images. Due to security requirements to stay up to date, the container images must now be upgraded when in all likelihood, the application or library is not required for the application in the first place.

As the Docker daemon runs as the root user, and performs various privileged namespace operations, it is required to execute Docker commands via sudo, directly as the root user or be placed into the "docker" group. This long-running root process may allow for privilege escalation given any number of vulnerabilities, although root access itself is typically required for Docker access (outside of using any confused deputy attacks). As simply using the root account or having all users within "docker" group is not recommended for a number of security reasons, but mostly because it allows any compromised user or process in that group to gain root access.[168] Unfortunately, due to real world demands for development team access, application debugging, testing or other reasons, users which are not intended to have root-level access

[166] https://service.ait.ac.at/security/2015/LxcSecurityAnalysis.txt
[167] https://blog.docker.com/2013/09/docker-can-now-run-within-docker/
[168] Mostly through known attacks, such as bind-mounting the rootfs into a new container image, then entering that image to edit specific system files or create a new suid root shell on the host

(or docker group access, which can be equated with root access) are granted such access. This may also risk an attacker targeting development employees due to their ability to have near-instant root access on any host where such access to the "docker" group is granted. Other attacks may take the form of compromising an application (such as CI software or other devops tools) which is also within the docker group.

Due to the need to be "one size fits all" nature of Docker, a number of default capabilities remain enabled, and user namespaces remains disabled due to some features not working correctly. In addition to these issues, Docker users often perform somewhat risky practices, such as placing potentially sensitive values within environment variables, which can be auto exposed through container links or inherited by all processes. This threat may be a side effect of the core product being focused on applications and developers or devops.

Finally, in order for some deployments to allow for container introspection, the Docker daemon REST interface can sometimes be reachable. This high risk activity is sometimes performed to help with container management and inner-application debugging or in order to support monitoring, or intelligent load balancing. Access can be intentionally or unintentionally granted through the connected or exposed network interfaces or in some cases, the host Docker daemon socket (`/var/run/docker.sock`) is directly mounted inside the container. Almost any access to the Docker daemon should be considered as a full container escape, in addition to likely privilege escalation (as access to the Docker daemon is equal to root access on the host). For additional details on the strengths and weaknesses or risks of Docker, see Section 9.8 on page 88.

7.5 CoreOS Rkt Specific Threats

It is safe to say that CoreOS Rocket (Rkt) is still under very active development and may not be ready for production, even if the February 2016 release of 1.0 is marked as production ready. The overall platform, documentation, security defaults are still very immature when compared to LXC and Docker. Rkt configuration can be compared to LXC, where a number of security options should be tweaked, improved or added in order to obtain better base security (especially for privileged containers). The system also currently lacks strong MAC support, uses an extremely weak seccomp-bpf blacklist, and does not (or can not) drop many dangerous capabilities as it is heavily linked with systemd defaults. Threats to Rkt when configured with a stage1 LKVM are considered out of scope for this security consideration, however the full virtualization offers a number of security improvements over kernel namespaces.

While Rkt does not use a long-running root privilege daemon in the same way Docker does, it does require root to use almost all `rkt` commands. This contains the same risk as running as root, and enables the same potential threats. As with Docker, work is ongoing to break apart privileged operations from non-privileged operations. At the time of this writing, only the `rkt fetch` command can be run without root. Although CoreOS does attempt to prevent from procfs and sysfs exposures or attacks through read-only mounts, these protections can be easily revered by exploiting `CAP_SYS_ADMIN` to remount them as writable.

Within Rkt, Docker image verification is missing, and until very recently, TLS certificate validation was also required to be disabled when using Docker images. As discussed within this paper, the `--insecure-skip-verify` flag, required when installing Docker images, historically skipped prompting the user to trust a key, allowed HTTPS to HTTP downgrades, and chiefly disabled all verification even for TLS certificates on the upstream server connection. While the Rkt team was aware of the issue, and warnings were then added, it took roughly nine months to fix this issue, breaking apart `--insecure-skip-verify` into two separate options.

The rkt fetch command, used to download potentially signed images, risks potential image spoofing, tam-

pering or modification via upstream attackers if image creators are not trusted explicitly, or are trusted prior to downloading an image. The current UI defaults poorly indicate the trust level of an image for images "downloaded but not verified", or when keys can be trusted automatically (such as if the connection is performed over TLS when using the metadata service). For images downloaded from a private repository, `docker://` and `https://` protocols are supported, although for Docker, only HTTP Basic authentication is supported which risks exposing credentials within configuration files.[169] See Section 9.12 on page 94 for more information on Rkt security risks.

7.6 Indirect or Unexpected Threats

The sections below explore attacks which are related to containers but may not involve the container, host or other containers directly. These threats involve setting up the container in the first place, or must be accepted risks of the cloud storage platform.

7.6.1 Github All The Way down

The risks of using fully Github backed projects such as Docker and CoreOS is an underlying consideration of any Github user or project. This risk or potential threat may come from malicious developer contributions, compromised developer accounts. Attackers may attempt maliciously covert commits, hiding bugs which can be later exploited. Even Github itself could be compromised, which has occurred several times in the past. Finally, attacks or exposure of SSH keys used for major accounts[170] is a reoccurring vulnerability, as is attacking locations with shared OAuth tokens or SSH keys for connected CI infrastructure. Docker manages several core libraries through Github and accepts various commits from upstream GitHub vendors, which themselves accept commits from yet other, potentially weakly authenticated or semi-anonymous GitHub users with levels of verification or commit review likely ranging from none to several users. While this is generally a risk we must fundamentally accept, it is no doubt something to consider.

7.6.2 Denial of Service

Malicious or compromised containers can leverage any available resource to perform various DoS attacks. If cgroups and other limits are not in place, attackers could fill the entire disk space, trigger file descriptor limits, hit max process limits, open the max number of open file descriptors and a number of related attacks on global kernel resources (regardless of namespaces). While many of these DoS attacks require command or code execution within a container, they often require little to no privileges and will likely impact the host as well, which can make remediation and even performing any action on the host difficult as well.

7.6.3 The Problem Of Patching

The software, supporting libraries and various binaries within containers almost always need to be regularly upgraded, or newly built containers need to replace old ones. As new vulnerabilities are found all the time, the larger amount of software running within a container leads to a greater risk of a vulnerability being discovered within said container. Vulnerabilities may not only affect the containerized application, but may also risk exploitation in a basic form or, in the worst case, allow for a foothold into the container itself. While defense in depth may mitigate some issues, or the vulnerable code path may not be reachable, a clear plan for upgrading containers is required, even apart from security bugs. As upgrading containers "in place" isn't a recognized or recommended practice (that is to say, running a command such as apt-get update), and the alternative of immutable, data-only and app-only containers can be challenging, there is no particularly simple solution for this. Moving to an immutable architecture is also particularly difficult if the overall stack is not Highly Available (HA) and or otherwise Load Balanced. See 9.8.2 on page 91 and Section 10.3 on page 107 for more information and recommendations.

[169] The configuration also supports OAuth Bearer Tokens for registries which support it.
[170] https://blog.benjojo.co.uk/post/auditing-github-users-keys

7.6.4 Advanced Hardware Attacks

Exposing devices directly via cgroups may invite attacks against specific kernel modules, non-standard device drivers or even system hardware itself. In cases where special devices are exposed through a device, and such a device is allowed to be accessed via a container, this may allow for specific DoS or other attacks where none would have existed previously. Other "fringe" areas of attack may target TCP segmentation offloading, system non-ECC memory via Rowhammer[171] or even advanced CPU instructions[172] and CPU L3 cache timing attacks.[173] Such risks are generally increased when using so called "bare metal" containers, but many of the normal security threats and recommendations remain involving the principal of least access, even for device hardware.

7.6.5 Image Attacks Via A Poisoned Apple

When downloading images from LXC rootfs download repositories, Docker Hub, third party repositories or CoreOS repositories, the image or rootfs is rarely inspected (as long as the resulting behavior and output is as expected). Although some container platforms such as Docker have a curating process[174] the threat remains. There may come a time where malicious images are inadvertently produced and/or hosted by container companies and developers, discovered by unknown actors, implemented by questionable security researchers, or merely for testing[175] and mistakenly used by end-users. When will we see "backdoored" container images and do they exist now? It's hard to know for sure, but it is a threat to be considered during deployment. Amazon AMIs with malicious backdoors have been discovered in the past[176, 177] and backdoors are extremely difficult to spot[178] and in some cases, more-or-less impossible[179] depending on the technique.

Making potential backdoors more difficult to spot, is the "large base image problem" which occurs mostly in LXC[180] and Docker.[181] A number of risks exist for inheriting unknown vulnerabilities or high risk libraries within the required package dependencies. While the Docker philosophy is a single "app container", it is rarely actually the case. Common examples of image "bloat" include pulling in base Ubuntu Linux images of several hundred megabytes, or using another `FROM` command which pulls yet another unknown base image. While only one application may be running as part of the `CMD` or `ENTRYPOINT` flag, numerous others often exist, including full interpreters such as Perl or Python which allow for attacks and potential container escapes. See Section 10.1 on page 99 and Section 10.3 on page 107 for security recommendations on base or rootfs images.

7.6.6 Going Forward

Vulnerabilities are more likely to be discovered going forward in disparate areas, such as those of the Linux kernel or supporting systems which were not written with capabilities or namespaces as part of the design. The `dac_read_search(2)` inode access issue and the exposure of process names via the world readable `/proc/sched_debug` information leak are good examples of this key problem.

[171] http://www.halfdog.net/Security/2015/SafeRowhammerPrivilegeEscalation/
[172] https://www.pagerduty.com/blog/the-discovery-of-apache-zookeepers-poison-packet/
[173] https://www.kb.cert.org/vuls/id/976534
[174] See more information within https://github.com/docker-library.
[175] https://twitter.com/mubix/status/576592666294628353
[176] https://www.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/BNPSS11.pdf
[177] http://www.forbes.com/sites/andygreenberg/2011/11/08/researchers-find-amazon-cloud-servers-teeming-with-backdoors-and-other-peoples-data/
[178] http://dvlabs.tippingpoint.com/blog/2011/04/11/cloud-security-amazons-ec2-serves-up-certified-pre-owned-server-images
[179] https://www.ece.cmu.edu/~ganger/712.fall02/papers/p761-thompson.pdf
[180] With LXC, this is mostly intentional due to the expected use case of supporting multiple applications within a single container.
[181] Docker is increasingly switching to Alpine Linux to sidestep this large base image problem.

8.1 The User Namespace

With the release of the user namespace in Linux 3.8[182], [183] root (uid/gid 0) within a container is no longer considered by the kernel as root outside of that user namespace (*e.g.* root in the container is no longer root on the host). This obviously is a great security advancement, and removed a long existing weakness of Linux containers by allowing for "privileged" operations within a container, yet limited (non-root) access in the case of an access control breakdown or container escape. The kernel and related procfs/sysfs attack surfaces are less of a concern with the user namespace, as root now lacks elevated privileges to perform typically sensitive operations. The user namespace is built by using a one to one mapping of userspace UID values to kernel 'kuids'. This is fundamentally different than suid, because it takes place transparently for the root user as well as the normal 'struct user' being a different kernel structure.

User namespaces are also great for resource control and to help isolate containers on the same host from each-other. While this isolation could previously be achieved through using a different uid/gid per container instance, the user namespace offers a more consistent map. User namespaces also offer defense in depth against a privilege escalation vulnerability within a multi-process or multi-user container. This new user namespace also allows the creation of fully unprivileged containers by unprivileged users. While this allows for great security benefits that fully embraces the principle of least privilege, and helps support the development and security of desktop application containers, this obviously opens up the door for potential security risks. Vulnerabilities may occur within the user namespace implementation or the at the intersection with other system components, as the case has been a number of times, see 7.2.5 on page 58 for more information.

8.1.1 Unprivileged containers

Fully unprivileged containers, added alongside user namespace support, allow for unprivileged users to create and run OS and application containers. This obviously expands the opportunities for non-server application containers and allows for transparent sandboxing of applications via unprivileged containers or individual container features if full containers are too cumbersome. On Ubuntu, unprivileged containers are the default if LXC commands are invoked using an unprivileged user. Rootfs images are pulled in by using the "download" template..[184] Docker in version 1.10 added support for the user namespace, although it is not enabled by default, and the Docker daemon still requires root in order to create containers. CoreOS Rkt has experimental support for user namespaces[186] and will likely require root interaction as well. For unprivileged containers without using a container framework, the `unshare`, `runuser` and `lxc-usernsexec` commands among others can be used directly (or at an even lower level, directly using the system calls is also an option). Finally, Unprivileged containers in Go by Alexander Morozov of Docker is also a great resource for those implementing User namespaces directly in Golang.

8.1.2 Exploring User Namespaces

User Namespaces are basically achieved by using a "uid/gid shift", such that all UID values, including UID 0, are remapped for each instance of the user namespace. If not controlled by the container framework of choice, this will be setup through global configuration files `/etc/subuid` and `/etc/subgid`. For any process within a user namespace, the `/proc/<pid>/uid_map` file can be used to examine the respective offset, which also can confirm the presence of a user namespace. For instance, inside the user namespace, the file will

[182] https://lwn.net/Articles/491310/

[183] http://kernelnewbies.org/Linux_3.8#head-fc2604c967c200a26f336942caee2440a2a4099c

[184] It may be prudent to note, the "download template"[185] uses Stéphane Graber's own server for a build environment (images.linuxcontainers.org RDNS: rproxy.stgraber.org). The security of image delivery (assuming trust of Stéphane Graber) should be GPG signed and verified and the download performed over HTTPS, although the script will fail open with a warning in both cases.

[186] https://coreos.com/rkt/docs/latest/devel/user-namespaces.html

read `0 100000 65536` while outside of the user namespace, the file will contain `0 0 4294967295`. Due the relatively new feature and various distribution implementations or support, if the new UID values are not automatically allocated, the `usermod` tool can be used to add them. For further information on the implementation and some use outside of containers, see Namespaces in operation, part 5: User namespaces by Michael Kerrisk.

8.1.3 User Namespaces in LXC

For LXC, starting in v1.0, user namespaces are enabled for any container by specifying the `lxc.id_map` directive within the respective LXC configuration file.[187] On Ubuntu, this can be illustrated by using unprivileged containers to issue a command inside the container as "root", and inspect the process from outside the container within the host. Note the following example assumes the basic subuid map is already setup:

Creating and starting a busybox container, using the Ubuntu template download to quickly obtain the rootfs method) named "foo":

```
user$ lxc-create -n foo -t busybox
user$ lxc-start -n foo
udhcpc (v1.21.1) started
Sending discover...
Sending select for 192.168.32.126...
Lease of 192.168.32.126 obtained, lease time 3600

Please press Enter to activate this console. <enter>
```

Within the unprivileged container, we execute `id` command to show we are "root", then sleep for 9999 seconds and disconnect the console:

```
BusyBox v1.21.1 (Ubuntu 1:1.21.0-1ubuntu1) built-in shell (ash)
Enter 'help' for a list of built-in commands.

/ # id
uid=0(root) gid=0(root)
/ # sleep 9999 &
/ # ^D
```

Outside of the container, we'll list the processes belonging to the user with UID 100000 (the start of the uid shift) and see the sleep command running as non-root. We can also see the UID map for this particular process:

```
user$ ps --User 100000 -u
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
100000     7199  0.0  0.0   2184   512 ?        Ss   22:51   0:00 init
100000     7246  0.0  0.0   2188   256 pts/15   Ss+  22:51   0:00 /bin/getty -L tty1
    115200 vt100
100000     7247  0.0  0.0   2188   500 pts/23   Ss   22:51   0:00 /bin/sh
100000     7248  0.0  0.0   2184   260 pts/23   S+   22:51   0:00 sleep 9999

user$ cat /proc/7248/uid_map
    0     100000      65536
```

[187] https://linuxcontainers.org/lxc/manpages/man5/lxc.container.conf.5.html#lbBA

8.1.4 User Namespaces in Docker

User namespace support has been a long-time security desire for many Docker adopters and users. Early-on, within the initial Docker releases (after LXC released version 1.0) user namespaces could be achieved by using the LXC engine, although this was short lived, due to Docker switching to their in-house written and Golang powered libcontainer.[188] Docker was then prevented from implementing user namespaces within libcontainer due to actual limitations within Golang itself.[189] After this limitation was resolved, Docker seemed to have focused on different or business efforts while continuing to slowly work with the community of a user namespace implementation, however implementing user namespaces in Docker was no small task due to the large number of Docker use cases and the previous core architectural decisions made by the Docker daemon, Dockerfiles, various metadata, and other details.

As Docker deployment exploded in popularity, a large focus was placed on security in the past 12-18 months. Progress was eventually made and the user namespace support moved from experimental[190] to stable[191, 192] in Docker 1.10, released in February of 2016. This was largely by an effort of IBM's Phil Estes[193] (although he debates that effort[194]) to bring together several different approaches for the implementation in Docker. This implementation, which essentially uses a single user namespace rather than a per-container user namespace, is currently considered as "phase 1" which also only remaps the root user.[195] Future support in "phase 2" will allow for full per-container UID remapping.

User namespaces in Docker are not enabled by default, but will be used whenever the `--userns-remap` daemon flag is present. Some limitations do exist[196] related to read-only containers, namespace sharing (which should also be considered high risk), and external drivers. Hopefully as these limitations are resolved, user namespaces will become the default. It should also be noted that while user namespaces are presently supported, it still requires a privileged user to interact with the Docker engine, or the unprivileged user must be within the docker group (which is effectively allowing full root to that user, given any compromise or malicious actions).

8.1.5 User Namespaces in Rkt

CoreOS Rkt added experimental support for user namespaces in version v0.8.0. This is likely due to existing or added user namespace support in systemd (via systemd-nspawn), which Rkt makes heavy use of. The user namespace can be enabled by using `--private-users` and `--no-overlay` (as OverlayFS is not compatible with user namespaces in Rkt) on the command line.

8.1.6 Here Be Dragons

Despite the large upsides the user namespace provides in terms of security, due to the sensitive nature of the user namespace, somewhat conflicting security models and large amount of new code, several serious vulnerabilities have been discovered and new vulnerabilities have unfortunately continued to be discovered. These deal with both the implementation of user namespaces itself or allow the illegitimate or unintended use of the user namespace to perform a privilege escalation. Often these issues present themselves on systems where containers are not being used, and where the kernel version is recent enough to support

[188] https://github.com/opencontainers/runc/tree/master/libcontainer
[189] https://github.com/golang/go/issues/8447
[190] http://integratedcode.us/2015/10/13/user-namespaces-have-arrived-in-docker/
[191] https://blog.docker.com/2016/02/docker-engine-1-10-security/
[192] https://github.com/docker/docker/pull/19187
[193] https://github.com/docker/docker/pull/12648
[194] http://integratedcode.us/2015/10/16/its-the-community-stupid/
[195] https://events.linuxfoundation.org/sites/events/files/slides/User%20Namespaces%20-%20ContainerCon%202015%20-%2016-9-final_0.pdf
[196] https://docs.docker.com/engine/reference/commandline/daemon/#user-namespace-known-restrictions

user namespaces.

User namespaces also allows for "interesting" intersections of security models, whereas full root capabilities are granted to new namespace. This can allow `CLONE_NEWUSER` to effectively use `CAP_NET_ADMIN`[197] over other network namespaces as they are exposed, and if containers are not in use. Additionally, as we have seen many times, processes with `CAP_NET_ADMIN` have a large attack surface and have resulted in a number of different kernel vulnerabilities. This may allow an unprivileged user namespace to target a large attack surface (the kernel networking subsystem) whereas a privileged container with reduced capabilities would not have such permissions. See Section 5.5 on page 39 for a more in-depth discussion on this topic.

For these reasons, among other risks, the grsecurity patches default to disabling the user namespace for unprivileged users. Linux distributions have also shipped custom modifications to disable it and kernel developers have discussed patches to disable it's capabilities for server administrators who want an easy method to disable it, without having to recompile their kernel. See sysctl: allow CLONE_NEWUSER to be disabled for a lengthy and contentious kernel-hardening mailing list thread and the container threats in section 7.2.5 on page 58 for more information and examples of prior vulnerabilities. Finally, subgraphOS, a high-security Linux distribution also ships with a disabled user namespace for security reasons.[198]

If we understand that kernel namespaces are incomplete (and more of a logical attempt at isolation rather than a designed security barrier), and that Linux capabilities must be dropped or are also incomplete, then we need yet something else for security. Enter Mandatory Access Control – keeping root, and everyone else in check.

8.2 Mandatory Access Control

While Mandatory Access Controls (MAC) are not a recent security advancement they are finding a new utility and rate of adoption along with the popularity of Linux containers. In 1977, the US Air Force commissioned an unclassified paper by the MITRE corporation titled *"Integrity Considerations for Secure Computing Systems"*[199] by Kenneth J. Biba. This paper (also released a few years earlier by UC Davis in 1975[200]) outlined different so-called "water marks" for secure enforcement of information access; the paper also discusses the idea of policies, domains, subjects and objects which focused on the "integrity" of secure data within the system. Almost ten years later in 1998, the National Security Agency (NSA) published an infamous paper titled *"The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments"*.[201] This paper gave Mandatory Access Control a major (or at least public) start within Operating System circles. Within Linux this was kickstarted by the NSA via SELinux, a set of Open Source patches released directly by the NSA which added a Multi Level Security (MLS) type enforcement system.[202]

[197] https://lwn.net/Articles/673613/
[198] https://github.com/subgraph/oz/issues/11#issuecomment-163396758
[199] http://www.dtic.mil/dtic/tr/fulltext/u2/a039324.pdf
[200] http://seclab.cs.ucdavis.edu/projects/history/papers/biba75.pdf
[201] http://csrc.nist.gov/nissc/1998/proceedings/paperF1.pdf
[202] While outside the scope of this paper, it should be noted the earlier MITRE solution from Kenneth J. Biba is called the Biba model and the later NSA solution the so-called "inverse" Bell LaPadula model which is implemented within the MLS portion of SELinux. Wikipedia puts the differences between the models as: "The Bell–LaPadula model focuses on data confidentiality and controlled access to classified information, in contrast to the Biba Integrity Model which describes rules for the protection of data integrity." The primary SELinux model however is Domain Type Enforcement.

A particular Phrack article offers a concise overview of what MAC provides:

> **"Type Enforcement is a simple concept: Mandatory Access Control takes precedence over a Discretionary Access Control (DAC) to contain subjects (processes, users) from accessing or manipulating objects (files, sockets, directories), based on the decision made by the security system upon a policy and subject's attached security context. A subject can undergo a transition from one security context to another (for example, due to role change) if it's explicitly allowed by the policy. This design allows fine-grained, albeit complex, decision making. Essentially, MAC means that everything is forbidden unless explicitly allowed by a policy. Moreover, the MAC framework is fully integrated into the system internals in order to catch every possible data access situation and store state information."**
> - Linux Kernel Heap Tampering Detection by Larry H. in Phrack 66

The use within Linux containers is immediately clear. Prior to the user namespace, the capabilities model and other kernel namespaces were the only mechanism (aside from MAC) for limiting privileged containers and preventing escape. This can be found in mailing list postings[203] and security articles.[204, 205] While MAC systems can be cumbersome to configure, they offer strong additional security assurances and defense in depth,[206] provided kernel hardening is also applied. While there are several native methods of MAC enforcement for Linux, only two will be discussed within this section, as they are the arguably the most popular and most commonly supported within container environments.

8.2.1 Security-Enhanced Linux (SELinux)

SELinux is a generalized system to establish fine-grained policy and type enforcement, isolated in separate components or labels. SELinux essentially employs the Bell-LaPaula Model (BLP), commonly used for access control in government and military applications where such restriction is more easily enforced[207] or where type enforcement must follow data classification levels such as only increasing in classification. Configuration of SELinux primarily involves applying this type enforcement across different labels, and appropriately labeling both processes and data.

The extremely complex[208, 209] policy language is one of the reasons SELinux is not widely accepted, even among many security-conscious system administrators. In order for SELinux type enforcement to be "correct", the correct Multi Level Security (MLS) labels must be applied and fine-grained. Due to this complexity, lack of up-to-date policies and general lack of understanding, SELinux suffers from what the author personally refers to the "setenforce 0 principal".[210] Disabling SELinux is such a common trend, it even has a website created to stopping the practice, stopdisablingselinux.com with an associated "setenforce 1" t-shirt, put together by infamous SELinux advocate and Red Hat employee Dan Walsh.

SELinux is well supported within Linux distributions, including being enabled by default in modern versions of Google Android and RedHat/CentOS Linux. When a Linux kernel has `CONFIG_SECURITY_SELINUX` enabled, and SELinux has well configured policies, it can achieve a MAC solution. For containers, support is also fairly widespread, with implementations in LXC, Docker and CoreOS Rkt. Within LXC container tem-

[203] http://www.mail-archive.com/lxc-users@lists.sourceforge.net/msg00992.html
[204] http://www.ibm.com/developerworks/linux/library/l-lxc-security/
[205] https://blog.docker.com/2013/08/containers-docker-how-secure-are-they/
[206] This is especially the case if paired with the user namespace and other kernel hardening or attack surface reductions.
[207] Large budgets allow for creation of complex policies, although we've seen how effective they can be against even a single well motivated adversary or system administrator.
[208] https://www.rsbac.org/_media/documentation/rsbac_handbook/architecture_implementation/functional_diagram_gfac_rsbac2.png
[209] http://cecs.wright.edu/~pmateti/Courses/7900/Lectures/Security/NSA-SE-Android/Figs/selinux%20architecture.png
[210] "The likelihood of SELinux being completely disabled, set to not enforce loaded policies or not have an adequate policy quickly approaches 100% within various NCC Group pentests (and likely in general)."

plates, the `lxc.se_context` directive specifies the specific context to run the container under. If not set, the default in SELinux supported and enabled systems is the `unconfined_t` context, which is to say no SELinux confinement is performed. To aid with specific policy development, a simple SELinux example policy and additional information can often be found in /usr/share/lxc/selinux/lxc.te. For Docker, see RedHat's Project Atomic documentation for more information and the Docker SELinux security policy, also by RedHat for an in-depth discussion. CoreOS adds support for SELinux primarily through SVirt, in order to provide independent SELinux contexts.[211] Documentation or examples for SELinux within LXC, Docker and Rkt is fairly sparse.

Vulnerabilities and weaknesses within SELinux, apart from it being disabled or not enforcing a policy, are typically found within the policy file itself or inappropriately applied labels. However as at least one prior exploit by Brad Spengler[212] CVE-2015-1815 illustrates, even security software such as SELinux can introduce weaknesses or even could lead to a system compromise.[213] A lack of restrictions for system calls or other kernel edge-cases, as with any MAC system, also contributes to significant vulnerabilities, which either subvert the security system and in some cases disable it entirely within the first steps of an exploit. See 8.2.4 on page 73 for more information.

8.2.2 AppArmor

AppArmor offers a pathname based access control (as opposed to filesystem inodes within SElinux), which typically focuses on processes and is often data-centric. AppArmor, originally called ''subDomain'', was essentially released with Immunix Linux in 2001 and was created[214] as an easy solution to the complex setup required for SELinux. The SUSE AppArmor Quickstart documentation offers a good overview of how it works. AppArmor policies are based on a default deny and it can be used in a non-enforcing mode (similar to SELinux) in order to develop an application or process specific profile. In Linux kernels with `CONFIG_SECURITY_APPARMOR` configured one can confirm AppArmor is actually enabled by using the aa–status command or look for a ''Y'' within `/sys/module/apparmor/parameters/enabled`.[215]

AppArmor is typically found, and used by default, within a number of Linux distributions such as Debian and Ubuntu, as well as high-security distributions such as SubgraphOS (currently alpha) in order to protect various applications and network deamons.[216] Ubuntu has continued to add default profiles for a number of widely deployed packages from CUPS and tcpdump to Apache2 and even Firefox.[217] For container systems, AppArmor provides a MAC system that focuses on augmentation or defense in depth of normal container systems (namespaces, capabilities, and cgroups). Take a look at the default base profile for LXC containers (/etc/apparmor.d/abstractions/lxc/container-base [218]) for a well-tuned example.

Although profile generation is much easier compared to SELinux, it is not a trivial task, requiring an understanding of an application's requirements and ''exercising'' the application appropriately. A profile generator written by AppArmor developers, aa–`genprof`, can be used to develop a profile for a specific application or process. For Docker containers, bane[219] by Jess Frazelle can also be used to develop application and container-specific Docker AppArmor profiles. In all cases, profile generation is unfortunately not an oper-

[211] https://coreos.com/blog/container-security-selinux-coreos.html
[212] https://grsecurity.net/~spender/exploits/exploit2.txt
[213] In this case the vulnerability allowed for arbitrary command execution, possibly even exploitable remotely, via shell metacharacters within a file name (http://seclists.org/oss-sec/2015/q1/1011).
[214] http://wiki.apparmor.net/index.php/AppArmor_History
[215] As astute readers may guess, some exploitation methods have used the referenced /proc/sys/ entries to disable or allow for unconfined access via ''overmounting'' and other attacks.
[216] Interestingly, AppArmor contains a "severity" database of various files and Linux capabilities. See http://apt-browse.org/browse/debian/wheezy/main/i386/apparmor-utils/2.7.103-4/file/etc/apparmor/severity.db for an example
[217] https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/AppArmorProfiles
[218] https://github.com/lxc/lxc/blob/master/config/apparmor/abstractions/container-base
[219] https://github.com/jfrazelle/bane

ation that can be performed through static-analysis of either the binary application or source code – the application must be exercised appropriately to generate a complete profile.[220] Information on container specific AppArmor profiles for LXC and Docker can be found within Section 10.2 on page 105. See AppArmor Documentation and the Core Policy Reference for information on building profiles, different AppArmor commands, and various tutorials not specifically related to containers.

While AppArmor has received significant support and is widely used within most popular Linux distributions and container solutions (Docker and LXC), it does contain some underlying risks or vulnerabilities. AppArmor can be subverted in several ways, including but not limited to:

- **Path modification:** Using filesystem hardlinks, overmounting on top of existing folders, or remounting filesystems in different folders can support bypassing path-based rules. In a contrived example, mounting a new procfs in a new location will bypass any procfs AppArmor rules.
- **Inappropriate trust:** If the policy configuration details are sourced from the container's own filesystem or procfs mount, an attacker can rewrite the policy.
- **Profile weaknesses;** For an application of any complex size, or in the case of LXC using OS-style containers, the profile can be complex and therefore likely contain flaws. See Poking Holes in AppArmor Profiles by Azimuth Security. Finally, profiles that take advantage of "abstractions" may allow for unintended consequences.
- **Issues outside of MAC control:** AppArmor is also not designed to address some weaknesses, such as direct execution of system calls.[221]

Attacks leveraging the trust of the container's rootfs have also resulted in AppArmor bypasses for both LXC and Docker, as illustrated by the following description by Tyler Hicks for CVE-2015-1334 found by Roman Fiedler: *"A malicious container can create a fake proc filesystem, possibly by mounting tmpfs on top of the container's /proc, and wait for a lxc-attach to be ran from the host environment. lxc-attach incorrectly trusts the container's /proc/PID/attr/current,exec files to set up the AppArmor profile and SELinux domain transitions which may result in no confinement being applied."*

8.2.3 Other Mandatory Access Control Implementations

While AppArmor and SELinux are the most widely used Linux Security Modules (LSMs), several other Linux MAC implementations exist and offer different capabilities and configuration.[222] While these are out of scope of this paper due to their niche implementations or lack of support in most container frameworks, two such implementations deserve special mention:

- **Simplified Mandatory Access Control Kernel or SMACK:** The SMACK project can be seen as the antithesis of SELinux, focusing on being uncomplicated and easy to use. Existing, and possibly outdated, documentation by IBM within the Secure Linux containers cookbook explores SMACK as used in LXC. Outside of containers, the "SMACK MAC" is used today in everything from mobile Operating Systems such as Samsung Tizen to Phillips Smart TVs.

- **Grsecurity's Role Based Access Control or RBAC:** Grsecurity's RBAC offers an excellent framework for a MAC system, and one that is not implemented as a LSM, so it can work alongside others. Similar to AppArmor and SELinux, the RBAC system can be used in a training mode, developing a policy automatically based on exercised application features and "learned" functionality. Grsecurity's policy rules are based on three

[220] Areas which are not desired to be operational could be avoided, and will be blocked within the application, although this may trigger unstable application behavior depending on the level of error handling.
[221] http://comments.gmane.org/gmane.comp.security.apparmor/5184
[222] It is also worth noting, LSMs may become "stackable" in the future, although that remains a hot debate. See https://lwn.net/Articles/393008/ and https://lwn.net/Articles/518345/.

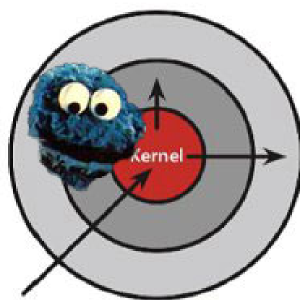
main distinctions: subjects, roles and objects. The policy is a mix of path based (similar to AppArmor) and inode-based checks (similar to SELinux), but it also contains simple rules for specific socket operations (which can limit specific network behavior against target hosts), resource restrictions, kernel capability controls, and many other options.[223] The training or "learning" support[224] within `gradm` offers a quick method to develop what are often complex policies, even on complex applications or applications for which no source code is available. While the learned or trained policy file may not be a perfect model of "correct" application behavior, it offers an excellent starting point for manual review of the specific operations or accesses an application, user group, or entire container performs according to the generated policy.

Unfortunately RBAC does not work well with containers, as the RBAC system is specifically not namespace-aware it may need to be properly configured for pathname based enforcement.[225] While fixing this was at one time on Brad Spengler's TODO list, a forum reply for requested support was in 2011 and has received no updates.[226] While little work has been done for containers and grsec RBAC and learning-based operations, it could be an effective MAC method for container solutions. Hopefully those with more time and funding can help develop this solution.

8.2.4 The "Flawed Assumption" of Mandatory Access Control Systems

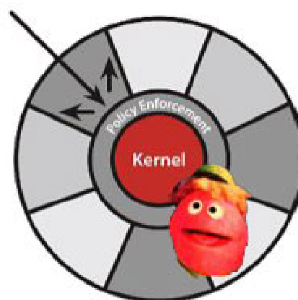
While Mandatory access control systems seem like an excellent and highly secure solution for enforcing security, a single weakness in the MAC policy, or the policy coverage itself can open up a significant attack surface and undermine the entire system. Generally, all MAC systems which are implemented in the kernel will be defeated by a kernel vulnerability, often exploited via exposed attack surfaces. This is a fundamental limitation which must be understood and kept in mind when exploring the effectiveness of any MAC system, in addition to exploring the potential attack surfaces and weaknesses of a given MAC policy or rule-set.

Making a parody of Dan Walsh's 2007 post on the SELinux implementation within RHEL 5, Brad Spengler of grsecurity posted[227] the following image, which comically illustrates the underlying flawed assumption of MAC:



Discretionary Access Control

Once a security exploit gains access to priveleged system component, the entire system is compromised.



Mandatory Access Control

Kernel policy defines application rights, firewalling applications from compromising the entire system.

Blackhats with kernel exploits

Basement dwelling 12-year olds armed with kernel exploit released past Tuesday. A SELinux disabling payload in the exploit turns your entire MAC policy into laughing stock. You spend the rest of the weekend removing SSH backdoors.

Red Hat and Security-Enhanced Linux (SELinux): It's really about the neat diagrams.

[223] https://en.wikipedia.org/wiki/Grsecurity#Role-based_access_control
[224] https://en.wikibooks.org/wiki/Grsecurity/The_Administration_Utility#Learning_Mode
[225] https://forums.grsecurity.net/viewtopic.php?f=3&t=2527&p=10296&hilit=lxc#p10300
[226] https://forums.grsecurity.net/viewtopic.php?f=5&t=2971
[227] https://lwn.net/Articles/357907/

This fundamental limitation of MAC systems is problematic, as the large kernel attack surface remains an Achilles' heel, proving that MAC systems alone cannot be the sole protection against system compromise. Historic vulnerabilities in syscalls, pipes, procfs and even implementation flaws in filesystems[228] have allowed for exploits to easily disable MAC systems, allowing for trivial further system exploitation. An example can be found in Phrack 66 referenced above, "Linux Kernel Heap Tampering Detection".

This basic weakness, along with other general hardening recommendations when considering the shared attack kernel surfaces of containers strongly encourages yet another layer of security: hardening the kernel itself. This includes but is not limited to keeping up-to-date on patches or using recent versions, removing the myriad of features which are not often required, and applying a hardening patchset such as grsecurity and PaX if at all possible. See kernel hardening recommendations in Section 10.5 on page 110 for more information.

8.3 Syscall Filtering with Seccomp

Seccomp or "SECure COMPuting" offers a method to reduce the number of system calls available for an application to interface with the kernel. While this may seem a recent advancement, this idea is not new. As early as 1996, Janus[229] was created by several researchers at UC Berkeley to limit system calls and provide a "restricted execution environment".

However, seccomp solved a core problem which plagued many prior implementations. These older implementations of syscall filtering often employed syscall "wrapping" or "tracing" such as BSD's deprecated systrace, and were repeatedly found to be vulnerable to concurrency issues such as TOCTOU (Time of Check - Time of Use)[230] and even several privilege escalations.[231, 232] Other ptrace-based syscall filters, such as those historically attempted by vsftp, and systrace are not ideal for the reasons mentioned above, not to mention they are very complex to implement. It should be noted systrace has now been replaced in OpenBSD by `tame()`,[233] a new and quite rational approach to filtering and reducing the syscall attack surface. See Domesticating applications, OpenBSD style for more information on a competing approach to Seccomp.

A limited seccomp was implemented in Linux as early as 2.6.12[234] and was enabled by writing directly to procfs. This was initially intended to provide for CPU sharing of fully untrusted applications, but that never fully developed. This "basic" seccomp was chiefly used within the Google Chrome browser[235] and limited syscalls to just `read(2)`, `write(2)`, `sigreturn(2)`, and `_exit(2)`, with a `SIGKILL` signal sent to the process when attempting other syscalls. This highly restricted set of calls is now referred to as `SECCOMP_MODE_STRICT`. Limitations in flexibility, complications in the implementation of disparate microprocesses, heavy IPC requirements, and risks of using those syscalls for special pseudo file systems (procfs) lead to further seccomp development efforts. After a number of failed trials and tribulations[236] the Linux community accepted a patch for seccomp-BPF. This introduced a means of configuring which syscalls are available to a process via a Berkeley Packet Filter (BPF)[237] and was written by Will Drewry of Google.

[228] Issues with reiserfs: https://www.exploit-db.com/exploits/12130/.
[229] http://www.cs.berkeley.edu/~daw/janus/
[230] http://www.watson.org/~robert/2007woot/2007usenixwoot-exploitingconcurrency.pdf
[231] https://www.provos.org/index.php?/categories/2-Systrace&/archives/33-Local-Privilege-Escalation.html
[232] http://undeadly.org/cgi?action=article&sid=20070809201304
[233] https://lwn.net/Articles/651701/
[234] https://lwn.net/Articles/346902/
[235] See https://lwn.net/Articles/347547/ and https://code.google.com/p/seccompsandbox/wiki/overview
[236] See https://lwn.net/Articles/332974/ and https://lwn.net/Articles/450291/
[237] https://lwn.net/Articles/475043/

8.3.1 Seccomp With Berkeley Packet Filters

For the first version of seccomp (SECCOMP_MODE_STRICT), the limit of system calls is often overly restrictive for non-trivial applications, or too restrictive for those who do not want to develop millions of broker processes and IPC calls. Seccomp BPF uses a Berkeley Packet Filter (BPF) to filter calls made by the restricted program.[238] The BPF pseudo-language was designed for high-speed, in-kernel bytecode[239] evaluation in a simple and safe language.[240] By using BPF to evaluate system call IDs and their arguments, instead of the fields of IP packets, seccomp-bpf is able to reuse this mechanism for purposes other than firewalling. With the creation of a seccomp-bpf syscall filter-set, in either a whitelist or blacklist, syscalls (and in some cases their arguments) can be restricted.

As best stated by the original patch author for seccomp-BPF:

> **"The goal of the patchset is straightforward: To provide a means of reducing the kernel attack surface. In practice, this is done at the primary kernel ABI: system calls."**
> - dynamic seccomp policies (using BPF filters) by Will Drewry

Beware of trying to use seccomp-bpf as a general security mechanism or as the core of a sandbox implementation, as this is not its intended use. The documentation clearly states it should be used for defense in depth via attack surface reduction:

> **"System call filtering isn't a sandbox. It provides a clearly defined mechanism for minimizing the exposed kernel surface. It is meant to be a tool for sandbox developers to use. Beyond that, policy for logical behavior and information flow should be managed with a combination of other system hardening techniques and, potentially, an LSM of your choosing."**
> - Linux kernel Documentation/prctl/seccomp_filter.txt by Will Drewry

Seccomp-bpf also avoids problems typical with traditional system call interposition frameworks such as TOCTOU referenced above:

> **"BPF makes it impossible for users of seccomp to fall prey to time-of-check-time-of-use (TOCTOU) attacks that are common in system call interposition frameworks. BPF programs may not dereference pointers which constrains all filters to solely evaluating the system call arguments directly."**
> - Linux kernel Documentation/prctl/seccomp_filter.txt by Will Drewry

However, a currently understood limitation of seccomp relates to the `ptrace(2)` syscall. The official documentation[241] clearly states: *"seccomp-based sandboxes MUST NOT allow use of ptrace, even of other sandboxed processes, without extreme care; ptracers can use this mechanism to escape"*. If `ptrace(2)` is allowed, the tracer can modify the process' system call in order to bypass the filter and then call blocked or restricted system calls (further examples are provided in seccomp documentation). See seccomp_ptrace_escape.c on github for a proof-of-concept.

Seccomp-BPF has two different operating modes, enabled via `prctl(2)` or `seccomp(2)` syscalls. In either case, the BPF program is passed as a pointer which is then installed in the kernel and called on each and every system call (for threads which are using seccomp-bpf). Once the filter is setup, it cannot be removed (similar to root capabilities) and filters can only become more strict. This allows for filtered applications to further remove syscalls from their own permitted sets, allowing for a true least privilege model.

[238] http://www.tcpdump.org/papers/bpf-usenix93.pdf
[239] https://blog.cloudflare.com/bpf-the-forgotten-bytecode/
[240] BPF programs are directed acyclic graphs, all instructions are the same size and can be confirmed to exit.
[241] See SECCOMP_RET_TRACE within https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

`SECCOMP_MODE_STRICT`: The first version of seccomp, also called "mode one", enables the most basic seccomp implementation. This only allows processes to call `read(2)`, `write(2)`, `_exit(2)`, and `sigreturn(2)`. As stated in the documentation, this can be useful for minimal "number-crunching applications" or very small processes such as renderers in Google Chrome. This mode should be applied if at all possible, although for containers this will rarely be appropriate. Attempting to access syscalls outside of the set above results in a `SIGKILL`.

`SECCOMP_MODE_FILTER`: Also called "mode two", this version was added by Will Drewry in Linux 3.5. A pointer to a Berkeley Packet Filter (BPF) which defines allowed or blocked system calls is passed as an argument when using `prctl(2)`. As seccomp itself is preserved across an `execve(2)`, `clone(2)` or a `fork(2)`, syscall filtering can effectively follow a least privilege model, continuing to create new levels of restrictions "down" a sandbox or container path as long as `prctl(2)` is in the allow list at the highest level. To avoid unhandled behavior and weak error checking by applications denied access to system calls, filters can raise specific signals upon violation,[242] opposed to the forced `SIGKILL` in mode one.

8.3.2 Invoking Seccomp-BPF

It may be helpful to understand system calls[243] and how they are implemented[244] within the Linux Kernel before further implementing your own seccomp policy. After deciding to use ether the `FILTER` or `STRICT` mode, seccomp is triggered using `seccomp(2)` and `prctl(2)` syscalls. An example syscall in C is included below, where prog is a pointer to the BPF:

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, prog);
```

If the kernel has `CONFIG_FTRACE_SYSCALLS` enabled, syscall arguments can be filtered within the seccomp policy. This argument filtering is carefully limited to non-pointer, often numerical arguments, due to potential TOCTOU attacks.[245] For more information on implementing and exploring seccomp-bpf, Kees Cook has created an excellent seccomp teaching and tutorial page.

To see these options in action, consider reviewing some some sample programs and reading additional in depth information and examples. The libseccomp library also has great documentation, interfaces, man pages, a golang implementation and examples. The go-seccomp package from the excellent Subgraph team offers the ability to parse Chromium BPF policy files for review or implementation, and supports Golang. Subgraph is also moving their code to will use the more flexible gosecco.

8.3.3 The Problems and Setbacks of Seccomp BPF

Generating the correct and minimal syscall filter set is difficult. This is a complex problem, if not the core problem, of seccomp-bpf use. As discussed by Chromium OS authors: *"Determining policy for seccomp filter can be time consuming. System calls are often named in arch-specific, or legacy tainted, ways (e.g., geteuid(2) versus geteuid32(2))."*

While using strace (via `ptrace(2)`) based measurements can allow for building rulesets may work for simple programs, more complex issues may arise due to timing, threading or the inability to trace an entire container. Fortunately, advanced in-kernel tools such as Systemtap or Sysdig can be used to monitor an entire user (for which the container or collection of processes can run as) or to allow for non-ptrace based syscall

[242] There may also be a reason to use SIGKILL vs SIGTRAP or SIGERROR depending on the threat model and logging intentions.
[243] https://sysdig.com/fascinating-world-linux-system-calls/
[244] https://0xax.gitbooks.io/linux-insides/content/SysCall/syscall-1.html
[245] See the minijail documentation for more information.

measurements. For example, to trace all the syscalls from the "nobody" user, you can use the following sysdig: `sysdig -p"%evt.type" user.name=nobody`. The other main kernel auditing tool, Systemtap, can also be used for system call and resource access monitoring; it supports the development of MAC policies as well. Generating filter sets for LXC and Docker can also be aided by using the genSeccomp.sh[246] and mkseccomp scripts provided by the Docker project.[247]

Advanced seccomp filter generation can also be explored by defining a whitelist of no system calls then raising a specific signal that allows logging to be created, as seccomp mode 2 allows control over the triggered signal. A more advanced version is also discussed by Mozilla within their wiki page advanced use cases. This page discusses a "warn-only" mode, created by always allowing syscalls from a specific address, and using `SIGSYS` to log the system call. The using simple seccomp filters post by Kees Cook also offers a good solution and example code for syscall reporting by using a similar catch and log method.

The problem with the above methods for profile generation is they are extremely slow and iterative, as syscalls are always blocked (unless the `SECCOMP_RET_TRACE` is the default filter action). The Subgraph team, as part of the oz sandbox have exploited this functionality to create a trainer program.[248] This uses the special `PTRACE_O_TRACESECCOMP` flag with `ptrace(2)` to dynamically test and profile applications for seccomp generation. While the current code is fairly oz-sandbox specific, a similar effort could be created which would be a standalone application. Finally when developing a seccomp policy, the Docker seccomp documentation[249] lists a number of potentially high risk system calls which should be excluded in any filter policy.

Missing support for CPU architectures other than x86 and x86_64 may prevent adoption on some platforms. Additional kernel hardware support for seccomp on other platforms is slow going. However, due to widespread Linux support, the nature of open source software, and the large development community around containers, it is only a matter of time for other architectures. ARM seccomp-bpf support is already technically in the kernel.[250] Note that lack of CPU support may introduce security vulnerabilities due to syscall numbers being different, and other soft failures such as a lack of kernel support being silently ignored. Seccomp-bpf code should always be written with this in mind and offer warnings if it cannot be activated.

The difficultly of a whitelist vs blacklist model. For seccomp-bpf, as with most access control systems, a whitelist is typically preferred. The list of syscalls a container should be allowed to make may be easy to generate, depending on application, deployment situation and container size. However in the case of syscalls, this may quickly break down or be very difficult to generate. A blacklist approach may be appropriate due to difficulties with static profiling, exercising program features, dynamic testing and complex applications. This list of high risk, possibly vulnerable, known dangerous or explicitly disallowed syscalls may be easier to establish. This may include syscalls which allow for loading kernel modules, rebooting, triggering mount operations and other administrative calls.

A good example of why a whitelist should be preferred is a recent local privilege escalation vulnerability was found within `keyctl(2)`, a system call that is unlikely to be blacklisted.[251] This relatively under-utilized and under-explored kernel key management facility contained an exploitable use-after-free vulnerability (CVE-2016-0728). An excellent write-up for exploiting this vulnerability can be found on the Perception

[246] https://github.com/konstruktoid/Docker/blob/master/Scripts/genSeccomp.sh
[247] See https://github.com/docker/docker/blob/master/contrib/mkseccomp.pl and https://github.com/docker/docker/blob/master/contrib/mkseccomp.sample.
[248] https://github.com/subgraph/oz/blob/master/oz-seccomp/tracer.go
[249] https://github.com/docker/docker/blob/master/docs/security/seccomp.md
[250] https://lkml.org/lkml/2012/11/1/512
[251] This syscall was not in the newly released Docker seccomp default whitelist, making Docker invulnerable to CVE-2016-0728.

Point article: Analysis and Exploitation of a Linux Kernel Vulnerability. If a blacklist will suffice, it should ideally be a temporary solution until a whitelist policy can be carefully generated. In another example, a seccomp blacklist could temporally mitigate a kernel vulnerability until patches are available, such as CVE-2015-3290. This vulnerability can be mitigated by filtering the `modify_ldt(2)` or `perf_event_open(2)` syscalls.

A final example of why to avoid blacklists comes in the difficulty of supporting different architectures. Buried within the manpage (and not mentioned in the kernel documentation) is the following text:

> This means that in order to create a seccomp-based blacklist for system calls performed through the x86-64 ABI, it is necessary to not only check that arch equals `AUDIT_ARCH_X86_64`, but also to explicitly reject all system calls that contain `__X32_SYSCALL_BIT` in *nr*. - seccomp(2) manpage

Jann Horn of Google Project Zero documented[252] these "more-or-less surprising things about seccomp", creating two example bypasses for weak blacklist implementations. Finally, performance may be a factor, and unfortunately in some environments, force a small blacklist as opposed to a larger whitelist.

Required system calls may include their own vulnerabilities. While many applications can be reduced to a small number of the available system calls, greatly reducing the kernel's attack surface, vulnerabilities may still exist within the allowed calls, and complexity of argument filtering may prove cumbersome or impossible. This may also differ depending on if a container solution is using an "app container" approach as in Docker, with a single containerized application vs an entire OS container, as LXC is often used. Seccomp filter sets apply to the entire container, so it is advantageous to keep the container as minimal as possible. The somewhat recent `futex(2)` requeue kernel privilege escalation vulnerably was particularly bad, as `futex(2)` is often thought to be safe and is otherwise a hard requirement within sandboxes, containers, and likely any application of moderate complexity, particularly those with multiple threads. The vulnerability, captured by CVE-2014-3153, was discovered by Pinkie Pie aka Comex[253] which worked as a Google Chrome Sandbox bypass[254, 255, 256] and was utilized by the infamous Geohot TowelRoot exploit.[257] See Exploiting the Futex Bug and uncovering Towelroot by Yohanes Nugroho for a complete walkthrough.

Seccomp-bpf contains problematic and non-trivial performance tradeoffs which may not be ignored in some deployments. However fast BPF is, a lot of system calls are required to be extremely performant or may be called inadvertently in tight loops. As researched by Michael Kerrisk and discussed by Jake Edge of Linux Weekly News:

> **"The performance cost for the filters is not insubstantial. (Michael Kerrisk) tested his simple "deny open" example, which is six BPF instructions, in a program that continually called getppid()—one of the cheapest system calls. That resulted in 25% more execution time than running it without the filter."**
> - A seccomp overview by Jake Edge

[252] https://lkml.org/lkml/2015/3/16/868
[253] https://hackerone.com/reports/13388
[254] https://groups.google.com/a/chromium.org/forum/#!topic/chromium-os-reviews/TP2YQBfL0ns
[255] https://codereview.chromium.org/314903002/
[256] https://code.google.com/p/chromium/issues/detail?id=377392
[257] http://blog.nativeflow.com/the-futex-vulnerability

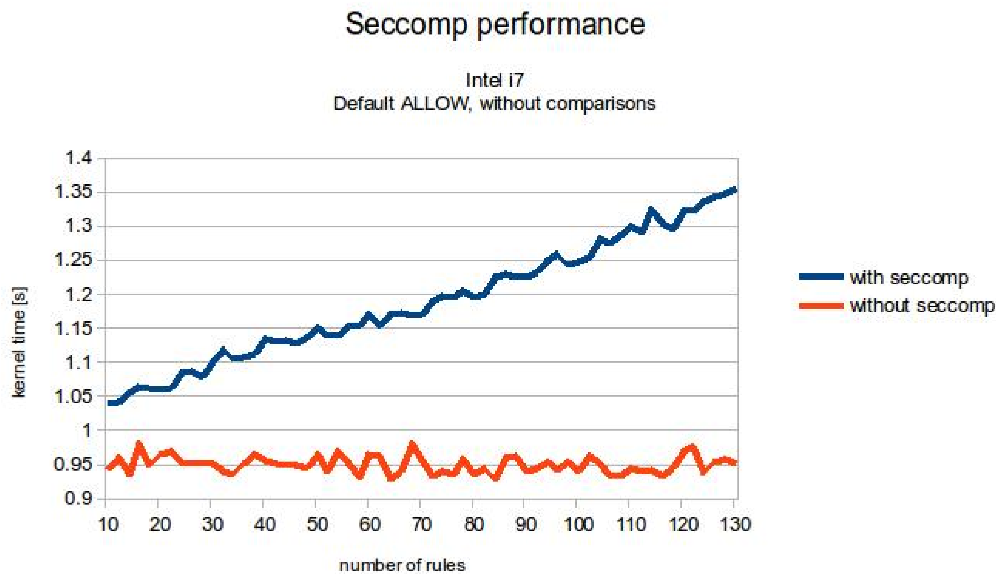


Figure 1: Image from the Tizen Project wiki which contains other performance information.

The order of filter operations and optimizations of BPF bytecode are also non-trivial, especially for applications which themselves install additionally increased filters via further `prctl(2)` calls. Performance on ARM is especially problematic, including a 20% overhead for a single seccomp-bpf rule.[258] Performance also drops with the number of rules linearly, using a default allow policy, even without argument comparisons as illustrated by the following graph:

To offset this performance hit, the filter should be ordered with the most-utilized syscalls last, as all filters are unintuitively executed in reverse order. In addition, deployments should likely enable the Kernel's build-in JIT compiler for BPF. This JIT compiler can apparently achieve a 2-3x performance increases. To enable it a runtime, set `/proc/sys/net/core/bpf_jit_enable` equal to 1. See the Using seccomp to limit the kernel attack surface by Michael Kerrisk for more excellent information. *Note:* As with any attack surface, enabling this BPF JIT may create additional vulnerabilities or Kernel address leaks. See Attacking hardened Linux systems with kernel JIT spraying by Keegan McAllister for an attack against Intel SMEP or PaX's KERNEXEC which cleverly leverages the Linux BPF JIT for a JIT spraying attack.

8.3.4 Seccomp within Linux Containers

As the kernel syscall interface is a significant attack surface for containers, it makes perfect sense to use seccomp to attempt and further isolate containers. Before continuing, it should be noted that, as of the time of this writing, Docker appears to by the only container platform to support filtering of syscall arguments.

In LXC, starting with version 1.0, seccomp-bpf can be enabled using the `lxc.seccomp` declaration within the respective container configuration file.[259] This should point to a specific file which, depending on the version specified, is either a whitelist of allowed syscalls (with a default deny) or a blacklist of disallowed syscalls (and default permit). A few defaults are also included as a blacklist,[260] and typically packaged with the Linux distribution upon install.

[258] https://wiki.tizen.org/wiki/Security:Seccomp
[259] https://linuxcontainers.org/lxc/manpages/man5/lxc.container.conf.5.html#lbAZ
[260] https://github.com/lxc/lxc/blob/master/config/templates/common.seccomp

Within Docker, seccomp-bpf support is now provided by default, within libcontainer as of Docker Engine v1.10 released in February of 2016, with initial support merged into experimental builds during the summer of 2015.[261] The development of seccomp overall within Docker is an interesting one. Docker started working on a default blacklist with optional whitelist, but hit licensing and library problems.[262] The development team then moved to a pure Golang implementation of a BPF ruleset generator[263] which was recently merged/added.[264] Prior to Docker version 1.10, in order to gain seccomp-bpf support within Docker, the lxc-backend must be used, and docker must be configured correctly. This older backend is no longer maintained, and many Docker features may not work with the LXC driver.

The seccomp-bpf support within Docker, implemented as a large whitelist, is now included and enabled by default.[265] The syscall whitelist contains 310 system calls in order to be generic across a great range of applications and to allow a low barrier for basic adoption. More information and examples can be found within the Docker Github project documentation security/seccomp.md and the full whitelist, 310 syscalls in all (roughly allowing 3 in 4 syscalls) can be found within default.json. Related to Docker is the "runC" project, powered by Docker's libcontainer. As of this writing, seccomp is a default build tag, as opposed to AppArmor which is optional. Unfortunately documentation on use or examples is quite scarce and will likely be added once the Open Container Foundation (OCF) specification is finished. CoreOS Rkt unfortunately does not directly support seccomp or seccomp-bpf, although the issue has been raised.[266] Support is currently implemented as part of systemd-nspawn, however the configured blacklist is extremely weak, blocking only ten syscalls and leaving many other dangerous and potentially high risk syscalls available.[267]

8.3.5 Beyond Containers: Other Implementations

Apart from containers, the advantages of seccomp-bpf for high-risk software such as web browsers and high-security software such as OpenSSH, vsftpd and anonymity systems such as Tor is clear. Many of these software packages have implemented syscall filtering and No New Privileges (NNP). Just as container hosts want defense in depth against unknown weaknesses within system calls or other kernel features powered by system calls, individual applications can equally take advantage of this least privilege solution. While some attacks are seemingly from the future (such as Rowhammer) the ability to reduce the attack surface will always make such exploitation more difficult.[268] Included below is a short, non-exhaustive list of open source applications currently using seccomp (either in `STRICT` or `FILTER` modes):

vsftpd: First implemented seccomp in version 3.0.0 in 2012. The implementation within vsftp carefully allows for different states of "trust" by limiting system calls as a function of application state (mainly tied to the authentication process). This expands privilege as required, which is an excellent strategy which follows the principle of least privilege.

OpenSSH: First implemented seccomp within version 6.0 in 2013. This uses a default deny filter and only permits a set of roughly 25 system calls.[269] Note that mode is off by default, but can be enabled by adding: `UsePrivilegeSeparation sandbox` to the configuration file.

[261] https://github.com/docker/libcontainer/pull/613
[262] https://github.com/docker/libcontainer/pull/384
[263] https://github.com/docker/libcontainer/pull/529
[264] https://github.com/docker/libcontainer/pull/613
[265] This is enabled on supported Linux kernels and when seccomp 2.2.1 is present. Older distribution versions, such as Ubuntu Trusty will not enable seccomp, even if there is kernel support.
[266] https://github.com/coreos/rkt/issues/1614
[267] https://github.com/systemd/systemd/blob/09541e49ebd17b41482e447dd8194942f39788c0/src/nspawn/nspawn.c#L1564
[268] https://twitter.com/chrisrohlf/status/575059136955740160
[269] brk(2), clock_gettime(2), close(2), exit(2), exit_group(2), getpgid(2), getpid(2), getrandom(2), gettimeofday(2), madvise(2), mmap(2), mmap2(2), mremap(2), munmap(2), _newselect(2), poll(2), pselect6(2), read(2), rt_sigprocmask(2), select(2), shutdown(2), sigprocmask(2), time(2), and write(2) as of OpenSSH 7.1

Google Chrome OS: The core design and security model[270] makes heavy use of seccomp-bpf for GPU sandboxing, the Google Chrome renderer, services which access ''external'' devices (such as USB), and within minijail (the built-in application sandbox).

Google Chrome browser: Uses seccomp-bpf for Flash and minimal rendering processes. See A safer playground for Linux and Chrome's next generation sandbox for more information.

Mozilla Firefox: Makes use of seccomp-bpf for some plugins although it is still missing for the core browser engine and renderer.

Tor (The Onion Router): Has enabled support for seccomp-bpf[271] although it defaults to disabled (in the future, this will likely be enabled by default as supported Kernel versions are more widespread due to Tor project's security focus). A list of permitted syscalls (for x86_64) is available to review[272] and illustrates the unfortunate complexity involved.

MBOX Sandbox: Makes use of seccomp-bpf[273] to do syscall interpositioning for application sandboxing. This system uses ptrace handler to then hook only the necessary system calls. MBOX has mitigated TOCTOU risks introduced via this method of syscall interpositioning and seccomp shimming.

[270] http://www.chromium.org/chromium-os/chromiumos-design-docs/system-hardening
[271] See https://trac.torproject.org/projects/tor/ticket/5756 and https://www.torproject.org/docs/tor-manual.html.en#Sandbox
[272] https://trac.torproject.org/projects/tor/attachment/ticket/10943/tor-messenger-seccomp-amd64.policy.sorted
[273] https://taesoo.gtisc.gatech.edu/pubs/2013/mbox/mbox.pdf

9.1 LXC

> **"Containers which offer an environment as close to possible as the one you'd get from a VM but without the overhead that comes with running a separate kernel and simulating all the hardware."**
>
> – LXC Documentation

9.2 LXC Background

The main idea of Linux containers (LXC) really started during Linux VServers, an early implementation of namespaces and placing entire systems within different "security contexts". As namespaces advanced within the kernel, and the precursor to cgroups was added,[274] developers from IBM started the Linux Containers Project around 2008 Adding a number of userspace tools and documentation[275] further advanced the development of LXC, including some of the initial security recommendations for using SELinux and Smack Mandatory Access Control systems[276] in parallel to user namespace developments. The Ubuntu Linux distribution by Canonical has largely led the recent development and advancement of strong defaults for LXC deployments.[277] However, apart from AppArmor and user namespaces, many hardening options remain disabled in most templates for ease of use or to remain generic.

LXC is also one of several different yet closely related projects:

LXC: The primary Linux Container userspace tools, security patches, standard defaults and templates. Found on the Linux container website[278] and Github.[279] Notably, LXC 1.0 will receive 5 years of security updates and bug fixes (2019 at the time of this writing).

LXD: A form of container hypervisor which is under very active initial development, offering integration with OpenStack, managed containers through REST APIs and other options. LXD can be seen as a sort of "docker for LXC", with similar command line flags, image repositories and other container management features. Future versions may support advanced CPU features in order to benefit from increased security.

LXCFS: A FUSE (Filesystem in Userspace) module for addressing the shortcomings of proc, cgroupfs and sysfs for container support. This is performed through custom overlay files for cpuinfo, meminfo, stat and uptime.

CGManager: Manages cgroups through a Dbus API, supports nested and unprivileged containers. CGManager is provided by default in Ubuntu and used in recent versions of LXC for cgroup management. It can also be used independently of LXC.

[274] https://lkml.org/lkml/2006/10/20/251
[275] http://www.ibm.com/developerworks/library/l-lxc-containers/
[276] http://www.ibm.com/developerworks/library/l-lxc-security/
[277] https://wiki.ubuntu.com/LxcSecurity
[278] https://linuxcontainers.org/
[279] https://github.com/lxc/lxc

9.3 LXC Components

LXC is primarily configured via configuration templates and command line utilities. Containers can be auto-started via integrations with system boot utilities (typically systemd). Support for advanced LXC features, such as unprivileged containers, LXC support and different cgroup management can vary across Linux distributions,[280] with Ubuntu Linux being the most well-supported platform. See Section 6.1 on page 43 for more information, example use.

9.4 Brief LXC Security Analysis

The following brief assessment of security should not be considered in-depth, but is intended to provide the reader with an idea of positive security controls, hardening and design. Also included is many prior issues, outstanding risks or vulnerabilities, known weaknesses in deployment and additional items for consideration which can aid in understanding security.

9.4.1 LXC Strengths

AppArmor for Mandatory Access Control (MAC) by default. If you're using Ubuntu, and likely some other Debian-based distributions, you'll have an AppArmor-isolated container by default. The default rules offer a number of defense in depth protections for various areas of the system not namespace aware, such as procfs and sysfs. Unprivileged containers, used by default if LXC is started by an unprivileged user, further enhance any default MAC rules.

Support for Seccomp-BPF, enabled by default with a minimal blacklist and with added support for different filter strategies. Seccomp support has been a long supported option within LXC. An allow or "white" list is permitted in addition to a simple deny or "black" list. Examples for each can be found in the example LXC documentation, in addition to the base blacklist.[281, 282]

Historical and continued user namespace support is available by default. Introduced within LXC 1.0, user namespace support on modern kernels offer a strong security barrier and additional defense in depth against malicious or compromised containers. LXC was the first major container management solution to offer stable support for user namespaces.

Strong configuration and control, straightforward templates. LXC offers a well documented and well understood method for configuration and setup of containers, with the vast majority of options coming from a standard configuration file rather than a mix of command line parameters. The templates for creating containers are simple shell or python scripts, which build or download root filesystems. These filesystems typically start out as tarballs or flat files.

Explicitly enabled container external network exposure. Apart from networking within a host or between containers via the default bridge, access to or exposure of listening services within a container must be explicitly granted via manual iptables forwarding. This default security control can help containers isolate applications from even weak or missing host firewall hardening.

Significant user base and community support offers indirect security benefits. The large number of LXC users indirectly contributes to success as an Open Source project, speed of patches (security or otherwise) and early feature support. Docker enjoys similar successes and deployment numbers, although some development efforts may be less transparent due to company governance or priority.[283]

[280] https://www.flockport.com/lxc-and-lxd-support-across-distributions/
[281] https://github.com/lxc/lxc/blob/master/doc/examples/seccomp-v1.conf
[282] https://github.com/lxc/lxc/blob/master/doc/examples/seccomp-v2-blacklist.conf
[283] The governance issue is also the case for runC and libcontainer, although this may be less so due to Open Container Initiative.

9.4.2 LXC Weaknesses

See Section 10.2 on page 105 for LXC specific security recommendations to help counter some of the following risks.

Philosophy of "system containers" is counter to security fundamentals of least access, least privilege and reduced complexity. The idea of using virtual systems over single app containers may provide some comfort for system administrators or other devops teams familiar with traditional virtualization and administration tasks. However, this philosophy comes with increased security risk, software bloat, and difficulty with hardening. For example, in order to permit administrative tasks within a container, increased privileges must be retained from the host system. This also increases the likelihood of a vulnerability being discovered within the large base images required to support an entire system. Finally, application-specific AppArmor profiles cannot be applied, unless a nested AppArmor configuration is used.

High risk capability defaults, encouraged by the "whole system" philosophy. For privileged containers not using the user namespace, LXC retains the vast majority of potentially dangerous capabilities, including `CAP_SYS_ADMIN`. Mitigating the risks of this large capability set is left to Mandatory Access Controls (MAC) via AppArmor, however repeated gaps in this configuration have been discovered. See Section 5.6 on page 40 for more information and a comparison table against other container platforms.

Default bridged networking. Default networking uses a bridge mode, this allows container to host and container to container traffic by default. Due to Linux bridges being virtual switches, layer two attacks such as ARP spoofing also work. This allows a single malicious or compromised container to hijack traffic for other containers within the same host, due to the often singular bridge interface. See 7.2.1 on page 52 for more information.

Outstanding seccomp-bpf issues and the `CAP_SYS_PTRACE` capability remaining enabled presents a risk to enabled security features. While seccomp-bpf is enabled by default, the blacklist is extremely minimal. Compared to the roughly 60 known dangerous calls the base Docker seccomp-bpf profile restricts, LXC only blocks five syscalls: `kexec_load(2)`, `open_by_handle_at(2)`, `init_module(2)`, `finit_module(2)` and `delete_module(2)`. Github issue 571 also undermines security due to the mixed CPU architecture support. Additionally, while not specific to LXC, `CAP_SYS_PTRACE` (granted to privileged containers) can be used to undermine seccomp-bpf, as the `ptrace(2)` system call is not within the filter list. This could allow for a privileged or unprivileged container escape in most scenarios despite seccomp-bpf and AppArmor.

Weak support for LXC via libvirt or svirt. While these can be used to launch and manage containers on supporting distributions, support for libvirt may be missing, currently deprecated[284] or incomplete[285] in some Linux distributions.

A history of issues within container management command-line software. Many different CVEs have been released for the various LXC utilities and init processes. If privileged containers are used, this may allow for container to host attacks by targeting LXC itself, as opposed to the kernel isolation features. See Security Issues in LXC and CVE-2015-1335 for further details.

[284] https://access.redhat.com/articles/1365153

[285] https://www.flockport.com/lxc-and-lxd-support-across-distributions/

9.5 Docker

> **"At the core of the Docker platform is Docker Engine, a lightweight runtime and robust tooling that builds and runs your Docker containers. Docker Engine runs on Linux to create the operating environment for your distributed applications. The in-host daemon communicates with the Docker client to execute commands to build, ship and run containers."**
>
> – Docker Documentation

9.6 Docker Background

Docker is undoubtedly a very hot topic in the container world. A large amount of Linux container support, efforts, development and overall "buzz"[286, 287] can be attributed to Dockers popularity. As the Platform as a Service (PaaS) offerings grew with the advent of hardware virtualization, bringing costs down significantly, the popularity of "the cloud" greatly increased between 2008 and 2010. A company called dotCloud was building, shipping software and using systems powered by Linux containers internally. This system was later released in 2013 as an Open Source project called Docker, which today is a rapidly expanding company with a number of different container related products and Open Source projects. What started with the release of a container solution has now expanded to Docker Hub, Registry, Swarm, Compose and different supporting software such as Docker Notary for trusted content and Docker Machine for provisioning containers.

A key difference Docker instituted, starting from the very beginning[288] from existing container philosophies was the focus on packing/shipping individual software through building container images. This major philosophy continues to be a focus of Docker today, as the founders intended to change the face of "shipping software" via containers, hence the name. This philosophy strongly encourages a single application per container, more easily allows containers without root or elevated privilege and increases the ease of use for application developers by removing typical system administration or "devops" work; while at the same time, providing more control and reliability. This overall movement is sometimes referred to as a "application containers" or "app VMs", rather than full OS virtualization. Docker is also not just limited to servers[289] as the desktop can easily benefit, although deployment and development has lagged behind the server focus significantly.

In the spring of 2014, Docker made a key change to the platform. They switched from using LXC to libcontainer written in Golang. This was designed from the ground up explicitly for containers and Docker as the new default "execution engine". This library directly makes the syscalls and performs other work on behalf of the Docker client, in order to create the required kernel namespaces, cgroups, manage capabilities and other required functions. While 64-bit Linux is the only "officially" supported platform, FreeBSD and even Microsoft Windows are adding support for Docker, as well as many cloud providers or virtualization software manufacturers, illustrating the general user demand for such systems.

[286] http://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/
[287] http://iops.io/blog/docker-hype/
[288] https://www.youtube.com/watch?v=wW9CAH9nSLs
[289] https://blog.jessfraz.com/post/docker-containers-on-the-desktop/

9.7 Docker Components

Docker has moved from a single container interface to an entire software ecosystem, changes rapidly progressing in the last two years as the company quickly expanded. This includes the Docker image hosting and distribution platform and other subscription-only or supported products such as the Docker Trusted Registry, which supports securely distributing signed Docker images. Many of these additional features are not in scope for this paper, which focuses purely Linux containers and related security. When discussing Docker, it should be clear there are several main components:

Docker Client: This client interacts with the Docker daemon, typically via the CLI "docker" command. This command actually interacts the Docker daemon's REST API, using a UNIX socket[290] by default (or optionally a TCP socket)[291] to communicate with the Docker Daemon. As the Docker daemon runs as root, access to the CLI (or the dockerd socket directly) effectively requires root privileges, or to be within the "docker" group. Untrusted users should never be in the docker group, or be allowed to communicate with the REST API unless they are intended to have root permissions on the host.[292]

Docker Daemon: Accepts Docker client connections from the REST interface or UNIX socket and exposes Docker Engine functionality. The Docker daemon also deals with monitoring, running and generally exposing Docker containers, acting essentially as the "init" for all running containers. The default listener is the UNIX socket, and it is encouraged for various security reasons[293] to be the only form of connection unless the API is required to be exposed outside of the host.

Docker Engine: The heavy-lifting behind the Docker daemon, the Docker Engine is written in Golang implemented via libcontainer now under the runC project[294] which implements the Open Container Specification v1.[295] This creates the required kernel namespaces, cgroups, handles capabilities and filesystem access controls.

Docker containers are composed primarily of Docker container "images". These images often start as a Dockerfile which can be thought of as a Makefile for the container image. These Dockerfiles are then compiled and built to different layers to provide several optimizations, which then results in an image. Each state-changing command within a Dockerfile typically creates a new image layer, which can be visualized by the imagelayers.io project. Images are often directly downloaded from a Docker registry or hub (also called Docker hub, which works similar to GitHub). Docker "official repositories"[296] contain a select set of base OS images which are analogous to "ISOs" when installing a new virtual machine or AMI's when deploying on Amazon EC2. This saves time rather than building all of the image layers or other included software from scratch (similar to Debian/GNU Linux packages for a distribution as opposed to using Gentoo Linux). It is also worth pointing out, all official Docker images are signed.

Running Docker containers are managed and exist within the host they were first started on as a collection of namespaced processes, similar to LXC and CoreOS Rkt. While Docker does not currently support checkpointing, restoring or live migrating running containers between hosts (think vMotion), this may be coming

[290] https://docs.docker.com/articles/basics/#bind-docker-to-another-hostport-or-a-unix-socket
[291] http://blog.trifork.com/2013/12/24/docker-from-a-distance-the-remote-api/
[292] Many public examples can be found to illustrate how to gain root access via Docker. This is also cautioned in the Docker security documentation: *"only trusted users should be allowed to control your Docker daemon"*. See the article Docker security for more information on access control or design assumptions.
[293] This includes Server Side Request Forgery (SSRF) protections, weaknesses in the TCP API defaults, required firewalls and authentication as well as binding to the correct interfaces.
[294] https://github.com/opencontainers/runc/blob/master/libcontainer/
[295] https://github.com/opencontainers/runc/blob/master/libcontainer/SPEC.md
[296] https://docs.docker.com/docker-hub/official_repos/

in the future.[297] Similar efforts are also in the works for LXC via new LXD features.[298]

At the disk level, Docker uses a Copy-on-Write (CoW) filesystem called AUFS, often by default (although Ubuntu may now default to DeviceMapper). Similar to the use of CoW within Virtual Machines and expensive external storage, CoW filesystems have an excellent advantage of disk space savings and quick creation time. While AUFS is not included within the Linux kernel by default, many modern distributions have chosen to include it (such as Debian and Ubuntu). The Overlay filesystem, overlayfs, is also becoming popular with Docker (and LXC) which is a fast[299] and efficient[300] "union" filesystem (another idea borrowed from Plan9[301]). This allows mixed "over" and "under" for the CoW, which can be nested in other overlay filesystems.

By using a filesystem built on layers, quick modifications can be performed in seconds, such as modifications or updates to a Dockerfile. This also allows for images to be inspected at each layer-based modification.[302]

9.8 Brief Docker Security Analysis

The following assessment of security should not be considered in-depth, but is intended to provide the reader with an idea of positive security controls, hardening and design in addition to prior significant issues, outstanding risks, known weaknesses in deployment and additional items for consideration.

Docker adds a number of features that set it apart from vanilla Linux containers or LXC, but the core philosophy can set it apart. Docker revolves around being application developer centric, with strong container versioning, image repositories, Dockerfile sharing, and other "fire and forget" features. The upside of application-specific containers is simplicity, least access, least privilege and other core benefits. The downsides of this ease of use involve pressures to reduce developer friction, keep generic options as defaults and make sure developers, not system administrators, can still easily "ship" containers and their software. This core trade-off between the ease of use and detailed configuration (which is strongly recommended, although not required for LXC) plays a key role in the current security settings, options and platform defaults. In January of 2015, a Gartner report Security Properties of Containers Managed by Docker by Joerg Fritsch, which is not publicly available and was not read the author, includes a large amount of information, although the discussion can be reduced to, according to The Register:

> **"Linux containers are mature enough to be used as private and public PaaS but disappoint when it comes to secure administration and management, and to support for common controls for confidentiality, integrity and availability."**
> - Docker Security Immature but not Scary by The Register/Gartner

In February of 2016, Docker Engine 1.10 introduced two long awaited key security features[303] for defense in depth: User namespaces[304] and seccomp filtering[305] via a generic syscall whitelist. Both of these key security features are supported in 1.10, assuming the features are present in the Linux kernel.

[297] http://blog.kubernetes.io/2015/07/how-did-quake-demo-from-dockercon-work.html
[298] https://events.linuxfoundation.org/sites/events/files/slides/Live%20Migration%20of%20Linux%20Containers.pdf
[299] https://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/
[300] https://twitter.com/burkelibbey/status/566314803225186304
[301] http://doc.cat-v.org/plan_9/4th_edition/papers/names
[302] This also can help support or encourage the development of host based monitors or security-related container watchdogs, which are becoming more popular.
[303] https://blog.docker.com/2016/02/docker-engine-1-10-security/
[304] https://github.com/docker/docker/issues/15187
[305] https://github.com/docker/docker/issues/17142

9.8.1 Docker Strengths

Strong container security defaults. Despite a large degree of use cases, Docker offers strong defaults for common applications, especially when this is compared to Linux capabilities of LXC and several default weaknesses of CoreOS Rkt. The strong momentum, large community, and somewhat recent security team,[306] and key security addicted developers[307] help drive key issues. These strong defaults help not only with respect to security, but support and tie-ins for other container platforms, containers on the desktop, defense in depth and generally support the security of software-focused data centers.

A base philosophy which supports security principals. Docker's "single application" philosophy, as discussed earlier, encourages simplicity, least privilege and least access. This simplicity attempts to package only what an application needs, limit potential attacks and reduce the inherited potential for various types of vulnerabilities. Another advantage of Dockers "modernity" is the use of Golang for many Docker components. Use of this programming language can avoid many traditional native code vulnerabilities related to memory corruption[308] and it directly supports kernel namespace functionality among other features required by the Docker Engine.

Built-in support for different Mandatory Access Control (MAC) systems: MAC systems are robustly supported by Docker. AppArmor support is well documented[309] and is used by default for defense in depth, with many borrowed rules from the LXC AppArmor base. Per-container AppArmor profiles are also supported via `--security-opt="apparmor:<profile>`. Recently, Docker also added an AppArmor policy for the Docker engine itself[310] and amid growing dissatisfaction with the always-root Docker daemon [note: runC. i think docker is starting to push to have runC be the default execution agent and do away with dockerd], have begun transition to break-out privileged functionality, although this is a long term goal and a large effort is required. SElinux support[311] is built in, in addition to being supported by RedHat as part of Project Atomic.[312]

Image and filesystem behavior supports auditing and specific security controls. The default copy-on-write filesystem isolates changes made by one container to another instance of the same container image, containers can also be made immutable which provides audit trails for incident response and makes restoring to known good possible (assuming the integrity of the root filesystem can be trusted). Apart from these features, storage drivers are more a concern of performance[313] or auditing, and apart from volume exposure via poor configuration, have little impact on the security of Docker apart from the occasional bug,[314] and some hardening issues[315]

With Docker 1.10 seccomp filtering is enabled using a default base profile. Within Docker, seccomp-bpf support is now provided within libcontainer as of Docker Engine v1.10 released in February of 2016. The filter

[306] http://blog.docker.com/2015/03/secured-at-docker-diogo-monica-and-nathan-mccauley/

[307] https://github.com/jfrazelle

[308] This includes automatic bounds checking and other features, such as banning pointer math: https://golang.org/doc/faq#no_pointer_arithmetic

[309] https://docs.docker.com/engine/security/apparmor/

[310] https://github.com/docker/docker/commit/39dae54a3f40035b1b7e5ca86c53d05dec832ed2

[311] http://opensource.com/business/14/7/docker-security-selinux

[312] http://www.projectatomic.io/docs/docker-and-selinux/

[313] http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/

[314] https://github.com/docker/docker/issues/10216

[315] AUFS also has had prior compatibility issues with Grsecurity patched kernels, although some patches have resolved these, switching to the devicemapper storage driver (which may be the default, depending on the Linux distribution) is a simple solution to avoid this conflict. Additionally, btrfs is not compatible with SELinux, which should be kept in mind if SELinux will be used for MAC.

functionality is implemented as a whitelist of 310 different system calls[316] (in order to remain quite generic), and blocks roughly 100 other known-dangerous or high-risk syscalls.[317] Prior to v1.10, syscall filtering via seccomp was in the contrib directory of the Docker Engine (libcontainer), marked as experimental. Prior to that inclusion mid 2015, seccomp-bpf was not supported within Docker's libcontainer. It could be used by using the LXC driver, however this often broke other Docker functionality expecting libcontainer and was not recommended as it is unmaintained.

Security profile developments for per-container security. It appears the Docker development and security team's intent is to eventually create the idea of a security profile (Github issue #17142) which includes an application/container specific seccomp filterset and possibly a specific AppArmor profile for each Docker container. This could be part of the official image for specific applications and allow for highly secure defaults with almost no developer or administrator interaction (assuming a fairly basic configuration). Unfortunately this strategy could be problematic going forward. Getting it "right" the first time is likely why it's taking the Docker team so long, as deployment complications grow, legacy requirements are in place, or Docker priorities shift, the solution could prove problematic. The age old Blacklist vs Whitelist for seccomp debate[318] continues to evolve, performance concerns thus far appear to be taking a back seat to security.

Explicitly enabled container external network exposure. Apart from networking within a host or between containers via the default bridge, access to or exposure of listening services within a container must be explicitly granted or "exposed". This default security control can help containers isolate applications from even weak or missing host firewall hardening.

Container image trust and integrity are ongoing priorities. A major feature of Docker (and CoreOS Rkt) is container image security itself. Although signed docker images were initially quite weak[319, 320, 321] recent advancements have created a robust solution (thanks to the hard work of the Docker security team and developers). A related improvement started as libtrust which was moved, merged and improved to later become Docker Notary (a system which also helps distribute data in general securely, not just containers).

Significant user base and community support offers indirect security benefits. The large number of Docker users, developers and individual contributors indirectly contributes to success as an Open Source project, speed of patches (security or otherwise) and support for various add-ons such as custom networking layers or storage back-ends. Some development efforts may be less transparent due to company governance, priorities or be released as subscription-only or non-open source projects.

9.8.2 Docker Weaknesses

See Section 10.3 on page 106 for Docker specific security recommendations to help counter some of the following risks.

Prior to Docker v1.10, a lack of the user namespace is a key weakness. As 1.10 was recently released, many prior deployed versions of Docker lack this support, however many other base security features offer good base security. The user namespace, when used in conjunction with other namespaces for containers, offers strong defense in depth against both known and unknown attack surfaces. This is currently implemented via an OS-wide or global UID remapping as part of "Phase One".[322] It is also worth noting the UID remapping

[316] https://raw.githubusercontent.com/docker/docker/master/profiles/seccomp/default.json
[317] https://github.com/docker/docker/blob/master/docs/security/seccomp.md
[318] https://github.com/docker/libcontainer/pull/263
[319] https://news.ycombinator.com/item?id=9419470
[320] https://titanous.com/posts/docker-insecurity
[321] https://lwn.net/Articles/628343/
[322] https://github.com/docker/docker/pull/12648

is performed once per daemon/Docker engine instance, a compromise due to shared image layer caching. **The user namespace is not used by default.** While Docker has an excellent default set of security, including as of 1.10 seccomp support as well, it does not use newly released user namespaces unless the daemon is started with the `--userns-remap` flag. As the user namespace disables some Docker features (due to current incompatibilities), it is likely not enabled by default. Hopefully as these limitations are resolved in the future, user namespaces will be enabled by default.

The REST API has a number of security problems. Weak defaults, missing security roles, historical vulnerabilities,[323] all access is read/write and the refactor is taking some time.[324, 325] The primary issue is the RESTful API is unauthenticated by default .[326] If the API is enabled by mistake, exposed outside of a trusted environment or exposed to all interfaces on the Docker host, it will allow unauthenticated attackers to fully compromise the server.[327] This may dangerously include attacks from compromised or malicious containers themselves, depending on the network configuration and hardening.

Default capabilities may present a security risk, especially with older Docker versions. While Docker does have the strongest default capability set (or put another way, the least number of retained capabilities) of the three major platforms examined, this is only a recent change. In order to make Docker work easily for the vast majority of use cases, the bounding capability set must include a mixed list of capabilities. The historical Docker guest escape via `CAP_DAC_READ_SEARCH`[328] was an unfortunately required wake-up call to further restrict the default capabilities,[329] and update the AppArmor policy.[330] While a number of capabilities were retained in earlier versions, recently the bounding set is described as only "those needed", and the rest are dropped by default. However, there are still a large number of capabilities enabled by default which may commonly not be required. In presentations by Docker, it is discussed Docker retains "less than half the normal capabilities", the system still retains a number of root capabilities which are not required for typical applications. Docker still retains 14 different root capabilities, including some potentially dangerous capabilities such as `CAP_NET_RAW`, `CAP_MKNOD` and `CAP_FOWNER`.

For these retained capabilities, `CAP_NET_RAW` which allows ping to work from a container (likely a major reason why it remains enabled), also unfortunately allows any RAW socket types, in addition to allowing the container to bind to any address within the exposed network namespaces. This capability could create vulnerabilities on the local network or within the host depending on the implementation details and risks of other adjacent network systems, which may include container management or orchestration software. The `CAP_MKNOD` capability, not likely to be required after an image has been setup, has some additional restrictions. The default AppArmor policy and a lack of other capabilities (such as `CAP_SYS_RAWIO` or `CAP_SYS_ADMIN`) may limit the potential risks for this retained capability, but both should be dropped if possible. Finally, it should be noted the capabilities restrictions and related discussion is only relevant to containers not using the user namespace, although some issues may remain regardless.

Network ports will bind to all interfaces by default. When networking ports for a Docker container are explicitly enabled, via -p or -P when running containers, the ports will be bound by the Docker daemon to all host network interfaces by default. This risks exposure of the ports to unintended network interfaces or hosts.

[323] https://github.com/docker/docker/issues/9413
[324] https://github.com/docker/docker/issues/7358
[325] https://github.com/docker/docker/issues/5893
[326] http://blog.james-carr.org/2013/10/30/securing-dockers-remote-api/
[327] As of 1.10, a new authorization framework is in place to limit and control access to specific areas.
[328] In June of 2014, Sebastian "stealth" Krahmer, a prolific security researcher and member of the SuSE Security team announced a Docker guest escape by using the `DAC_CAP_READ_SEARCH` capability.
[329] https://medium.com/@fun_cuddles/docker-breakout-exploit-analysis-a274fff0e6b3
[330] https://github.com/docker/libcontainer/pull/256

See the Docker documentation on exposing incoming ports for more information.

Risks of Dockerfile complexity and Docker image handling. Dockerfiles, the building blocks of almost all Docker images, have allowed for several vulnerabilities,[331] most notably CVE-2014-9357 which allowed arbitrary code execution. Dockerfiles themselves are fairly restrictive from the Docker host perspective, potential risks of Server Side Request Forgery (SSRF) could present themselves via malicious ADD or COPY directives. Docker image verification and integrity was only recently implemented properly (in Docker 1.8), with the integration of The Update Framework or TUF.[332] See Docker Content Trust for more information.

The seccomp filterset is extremely broad. While the seccomp filter has an enabled by default base profile, and does block roughly 50 high risk or dangerous system calls, it is effectively implemented as a blacklist of "known and potentially bad but unused". Even if the whitelist of roughly 300 calls is actually the technical implementation, the average application container likely requires a much smaller subset. Over time, revisions and improvements will likely take place for this policy or for "security profiles" during development.

As currently implemented, the Docker daemon must run as root. In order to perform the namespace requests or modifications against the kernel as well as filesystem controls, the Docker daemon and therefore client runs as root. While efforts are slowly underway to remove this root requirement, and the user namespace introduction within 1.10 helps this effort along, a large amount of code must be modified, and it must be performed in a secure fashion. Typical Docker installs will also create a "docker" group. This privileged docker group is often used for docker administration or integration by various users or applications. This often and unknowingly provides what is effectively root access to any user within the docker group (despite warnings in Docker's documentation). Any user who can execute the docker CLI command, or any user who can connect to the REST interface, can compromise the system and any container within it.[333] Finally, future efforts by Docker and the runC project may remove this root restriction, although this is still in the planning stage.

The default Docker networking within the host allows containers to communicate between each-other, due to shared network bridge. The `--icc` configuration option which creates a blanket FORWARD ACCEPT iptables rule by default, risks cross-container and container to host network connectivity. This inner Docker host network communication may not be intuitive during deployments or expected for users or developers new to Linux containers. This communication may pose a security risk, depending on the types of network services and the overall trust model for the deployment. An example could be understood as front-end API servers (directly exposed to the Internet) deployed via dynamic resource scheduling alongside back-end databases, with caching services for API sessions or other stores of sensitive information. Within many application-backends, debug interfaces or health and monitoring ports are also commonly bound to all interfaces then protected at the network parameter. Another example could be the host's Docker API bound to a reachable interface and inadvertently accessible. Finally, such cross-container networking is also vulnerable to security problems on regular hardware switches, such as ARP spoofing[334] Spanning Tree Protocol (STP) or even IPv6 attacks.

Dealing with Image upgrades or stale containers is problematic. Upgrades of containers in place (via package managers) is problematic and largely discouraged by the community in lieu of immutable images[335, 336].

[331] http://seclists.org/fulldisclosure/2014/Dec/52
[332] https://theupdateframework.github.io/
[333] This mistake has been discovered on a large number of different NCC Group container, application and network security assessments. In addition to various online recommendations, such as How to use Docker by Digital Ocean.
[334] https://nyantec.com/en/2015/03/20/docker-networking-considered-harmful/
[335] http://blog.codeship.com/immutable-deployments/
[336] http://chadfowler.com/blog/2013/06/23/immutable-deployments/

This can pose a problem for those developers or administrators moving from a more traditional server architecture to application containers, as security risks from stale images and outdated libraries place applications at risk. Large base images are more problematic to update, and in some cases strike the official images themselves. Jérôme Petazzoni of Docker has a great write-up and exploration on this topic.[337]

Large Docker base images using in FROM directives. When considering Docker images, it is important to consider how most Docker base images are still quite large. Using "FROM ubuntu" or "FROM centos" will pull in a large base set of packages. Hundreds of megabytes, containing unknown numbers of likely unrequited binaries, dependencies and libraries containing an unknown number of vulnerabilities. While the intent is often to be running just a single application per container, many Dockerfiles or base images pull in a gamut of other libraries, tools and system applications. The large base image problem, along with stale images, was the root cause for the article 30 percent of Docker Hub images contain vulnerabilities. It is also worth noting the unofficial Docker response to that article[338] dispels several, but not all myths. Finally, it seems Docker has recently started to switch some base images over to Alpine.

The Docker client does not verify the TLS certificate by default, leading to a potential false sense of security. When using the TCP socket connection for the Docker command line utility, the `DOCKER_TLS_VERIFY` environment variable or `--tlsverify` is required for certificate/host validation. While there is admittedly little potential for exploitation with many use cases (such as local-only connections), if the Docker client is used to connect to other Docker daemons across the network, this could present a risk of both server spoofing and client impersonation (in order to then compromise the server). See Docker documentation Secure by default for more information.

9.9 CoreOS Rocket

> **"Rkt is the next-generation container manager for Linux clusters. Designed for security, simplicity, and composability within modern cluster architectures, rkt discovers, verifies, fetches, and executes application containers with pluggable isolation."**
>
> – CoreOS Rkt documentation

9.10 CoreOS and Rkt Background

CoreOS started out as a fork of Google's Chrome OS, and is intended to be a minimal Linux distribution for hosting Linux containers. CoreOS was known specifically for hosting Docker as the popularity of containers (and Docker itself) grew rapidly starting in late 2013. Much more than a Linux distribution, CoreOS and the development team are now an ecosystem of software and standards. This includes but is not limited to the Application Container Framework (appc) intended to be a generic specification for Linux containers, the Rocket (Rkt) container which is an implementation of the App Container Framework and etcd which is a distributed key-value store for a clustered and redundant deployment of services.

[337] https://jpetazzo.github.io/2015/05/27/docker-images-vulnerabilities/
[338] http://jpetazzo.github.io/2015/05/27/docker-images-vulnerabilities/

CoreOS released Rocket (Rkt)[339, 340] in December of 2014. The design goals included simplicity (via the UNIX philosophy), speed and security. When launching Rkt, CoreOS largely pointed to Docker as devolving from their original mission and manifesto[341] "as they moved from Docker Containers to the Docker Platform". Rkt implements the Application Container Runtime which, alongside the App Container Image portion for container images and are strongly tied to both the Application Container Specification or "AppC".

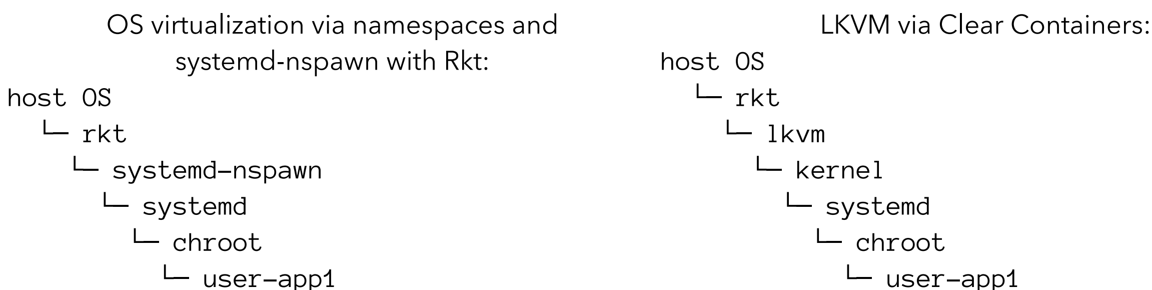
As with both LXC and Docker, Rkt is just one part of the container puzzle. Further integration with Kubernetes is also underway, which itself was released by Google for container management, deployment, clustering and other features (after years[342] of using similar methods via Borg, Google's internal container system). Integration of Kubernetes is provided through Tectonic, although these features and solutions are largely outside the scope of this paper. It is likely that CoreOS will remain a solid platform for Docker or LXC deployments, even as Rkt is intended to be the focus and key implementation of the App Container Spec and essentially competes with Docker and LXC.[343] As CoreOS only uses systemd for init, a key difference between Rkt, Docker and LXC is that Rkt also uses systemd as the supervisor daemon whereas the Docker Daemon is effectively the "supervisor" for Docker containers. LXC has no supervisor process (although it does have a related command socket).

9.11 Rkt Components

A key component of Rocket (Rkt) is intended to be simplicity, with no running daemon or real dependences apart from supported Linux kernel versions and CPU architecture. By executing a single command (rkt), users can create an entire container, pull down images (which can be signed and stored in internal or external repositories) and use many, although not all, of the standard Linux container or kernel features. CoreOS also is partnering with VMware for the App Container Specification work[344] and with Intel Open Source developers on "Clear Containers" by using a modified KVM to create a minimal, container-focused, hardware-implemented hypervisor.[345]

Rkt works through executing multiple "stages"[346] which perform different "heavy lifting" functions, analogous to a real rocket. Different stages allow for flexibility of implementation and can be understood as a design decision key to the App Container Specification. This flexibility was part of why Intel chose Rkt to create their proof of concept "Clear Containers", the power of a LKVM hypervisor applied to container isolation. The following ASCII diagram was recreated to illustrate the application hierarchy:

OS virtualization via namespaces and systemd-nspawn with Rkt: | LKVM via Clear Containers:



This goal of simplicity for Rkt is discussed by CoreOS, is also a key component of security. This is especially

[339] https://github.com/coreos/rkt
[340] https://coreos.com/blog/rocket/
[341] https://github.com/docker/docker/commit/0db56e6c519b19ec16c6fbd12e3cee7dfa6018c5
[342] https://queue.acm.org/detail.cfm?id=2898444
[343] https://coreos.com/blog/app-container-and-docker/
[344] https://coreos.com/blog/vmware-ships-rkt/
[345] https://coreos.com/blog/rkt-0.8-with-new-vm-support/
[346] https://github.com/coreos/rocket/blob/master/Documentation/architecture.md

the case as Rkt relates to Docker and what the CoreOS Rkt team sees as Docker becoming overly complex or requiring Docker specific tools and formats. According to project co-founder Alex Polvi, during an interview[347] on Rkt vs Docker, a major focus of Rkt is apparently security itself (despite some slow real-world improvements):

> Libby Clark: *"Will it (Rkt) be more secure?"*
>
> Polvi: *"I mean, if they're using our products (laughs). I don't say that to be overconfident I say that because we are sincerely focused on securing the backend of the Internet. If we don't deliver on that mission, we fail as a company. We sincerely are putting everything we can into security, going beyond just turning it on and making it easy to use. More and more companies are recognizing a lot of the different security technologies are very difficult to use. It's time to get serious about security."*

The Application Container specification and image format, key CoreOS initiatives, are also intended to be included within the Open Container Initiative (OCI).[348] The team also hopes to make CoreOS Rkt an implementation of the standard chosen or supported by the OCI going forward.[349] However, apart from a stream of additional members over several months, little public information has been released apart from the draft implementation by Docker's runC powered by libcontainer.[350]

9.12 Rkt Security Analysis

Before assessing the security of Rkt, it's important to understand the project is still in the early "very stages". This is easily illustrated by the types of issues opened within the public GitHub tracker, such as unsafe image extraction[351] or download,[352] requiring root,[353] the lack of documentation and relatively small number of core project contributors and developers. Although some security issues have been closed, many should not have been opened in the first place, especially for a project who is apparently so focused on security. Other critical issues, such as the unexpected disabling of TLS certificate validation[354] when installing Docker images remained open for more than nine months despite relatively trivial fixes.

Recently CoreOS also introduced "Clair"[355] which scans images (Docker and CoreOS) for vulnerabilities present within the supported package managers, all without executing the actual container image. In the Spring of 2016, Clair 1.0 was launched,[356] which is described as production ready and contains a number of helpful features. However, while Clair offers easy detection for known vulnerabilities which are patched upstream within platform repositories, it will only detect known vulnerabilities in packages installed by platform package managers, although this is a known limitation.[357] As with the cursory reviews of LXC and Docker, the following assessment of security should not be considered extremely in-depth, but is intended to provide the reader with an idea of positive security controls, hardening and design. Many prior issues, outstanding risks or vulnerabilities, known weaknesses in deployment and additional items for consideration have also been included.

[347] http://www.linux.com/news/featured-blogs/200-libby-clark/806347-collaboration-summit-keynote-alex-polvi-coreos
[348] https://www.opencontainers.org
[349] http://techcrunch.com/2015/06/22/docker-coreos-google-microsoft-amazon-and-others-agree-to-develop-common-container-standard/
[350] https://github.com/opencontainers/runc
[351] https://github.com/coreos/rkt/issues/904
[352] https://github.com/coreos/rkt/issues/194
[353] https://github.com/coreos/rkt/issues/539
[354] https://github.com/coreos/rkt/issues/912
[355] https://coreos.com/blog/vulnerability-analysis-for-containers/
[356] https://coreos.com/blog/clair-v1.html
[357] https://github.com/coreos/clair/issues/58

9.12.1 Rkt Strengths

Key goal of simplicity aids security implementations and reduces attack surfaces. The simplicity of design will help maintain good visibility, auditing and understanding, although feature creep is a problem for many Open Source projects even those with good check-in review.[358]

Clear Containers support via KVM. In August of 2015, Rkt version 0.8 was released which added LKVM or Intel Clear Container support.[359] This swaps "stage1" with a full hardware Virtual Machine, offering increased security. This security feature is largely unique to the three platforms and offers significant defense in depth, however Docker will soon have support for pluggable runtimes via the new containerd.

Isolator concept supports key security controls. CoreOS has a concept of an "isolator". These have started to be implemented with respect to resources[360] and "Linux Isolators" via capabilities (os/linux/capabilities-remove-set) although they may be established for other security functions in the future such as SELinux, AppArmor and system calls.[361]

TPM Support within Rkt for container image security. Support for TPMs as part of "trusted computing"[362] within Rkt is an interesting addition which is not supported by other container platforms. This offers a cryptographic binding which can help in some secret distribution scenarios, incident response and offers unique benefits for strong container to hardware binding. However, it remains to be seen how many Linux servers with supported TPMs this feature will be effective on.

SELinux support via specific SVirt integrations. This support was added in early 2015,[363] Each container can run within a different SELinux context or a custom defined context for additional, application-specific restrictions. Although documentation is currently quite weak and support requires the use of SVirt, SELinux is automatically enabled by default on kernels which support it. However SELinux may not work with recent versions of systemd (impacting Rkt) when set to enforcing mode[364] due to a systemd bug.[365]

9.12.2 Rkt Weaknesses

See Section 10.4 on page 109 for Rkt specific security recommendations to help counter some of the following risks.

User namespaces within Rkt disabled by default and remain experimental. Using the `--private-users` with `rkt run` will enable experimental support for user namespaces. As with Docker, user namespaces are not enabled by default. However Docker drops many more capabilities, has a seccomp filter and default Mandatory Access Controls, all of which significantly raise the difficulty of container escape or Linux kernel code execution. CoreOS Rkt has only basic support or no support at all for some of these security features, making user namespaces all the more necessary.

Rkt retains dangerous capabilities in containers. Due to the use of systemd, dangerous capabilities are still inherited by containers.[366] This includes `CAP_SYS_ADMIN`, which is understood by many to be a trivial pathway to root. Other high risk capabilities also remain enabled which may due to the integration or complications of systemd.

[358] Just revisit the DTLS implementation of and default inclusion within OpenSSL which lead to heartbleed.
[359] https://coreos.com/blog/rkt-0.8-with-new-vm-support/
[360] https://github.com/appc/spec/blob/master/spec/ace.md#resource-isolators
[361] https://github.com/coreos/rkt/issues/1614
[362] https://coreos.com/blog/coreos-trusted-computing.html
[363] https://coreos.com/blog/rkt-0.7.0-with-selinux-and-new-build-system/
[364] https://github.com/coreos/rkt/issues/2264
[365] https://bugzilla.redhat.com/show_bug.cgi?id=1317928
[366] https://github.com/coreos/rkt/issues/576

Weak trust establishment for image signing keys. While Rkt does support image verification via GPG signatures, if the `rkt trust` command is not issued before a `rkt fetch`, the key will be automatically downloaded and trusted without user interaction (if the endpoint is hosted over HTTPS) by using the "meta discovery" functionality. This is performed via a meta HTML tag in the page which points to a different URI on the website hosting the CoreOS ACI itself. Some improvements are also required to establish better trust of official Rkt images.[367] Finally, it is worth noting that Rkt signatures do not have timestamps, which may allow for downgrade or replay attacks depending on transport security and other factors.

Rkt currently requires root for all subcommands. Although the goal is to have a least privilege model, Rkt still requires root for almost all operations. Some progress is being made[368] and a full discussion is available.[369, 370] Currently the only non-root command is when downloading images[371] and is an optional component when setting up Rkt. Requiring root encourages elevated privileges by programs which must interact with Rkt or users running the various Rkt subcommands.

If Docker images are used, image signature verification is disabled. Docker image verification is not supported within Rkt, however this may be a common use case and some development is underway to bridge this gap.[372] Until recently both TLS certificate verification and image verification were disabled.[373] The documentation has now made clear, and warnings provided, with separate flags for disabling different types of security. Fortunately, apart from Docker image verification being disabled, other security features for image fetching are not disabled by default.

Seccomp support not integrated within Rkt. The App Container Specification and current Rkt implementation currently do not support seccomp-bpf directly, but instead rely on systemd configuration.[374] Seccomp support is currently claimed by using seccomp within systemd-nspawn. When enabled,[375] systemd-nspawn drops the following ten system calls: `iopl(2)`, `ioperm(2)`, `kexec_load(2)`, `swapon(2)`, `swapoff(2)`, `open_by_handle_at(2)`, `init_module(2)`, `finit_module(2)`, `delete_module(2)`, and `syslog(2)`. Compared to the roughly 60 known dangerous calls the base Docker seccomp-bpf profile restricts, this should be considered an extremely weak seccomp implementation. It should be noted that `ptrace(2)` is not dropped, which can allow seccomp to be subverted in many attack scenarios. Finally, this "outsourced" seccomp support may complicate a given container configuration and prove difficult for integration with the OCI.

Weak or missing support for Mandatory Access Controls (MAC). Due to the large number of root capabilities that remain enabled, MAC systems not enabled by default, and only experimental support for user namespaces, the kernel attack surface should be considered "highly available". SELinux is also the only Mandatory Access Control (MAC) solution supported, and support and documentation should be considered weak. With strong support for AppArmor by both LXC and Docker, it would be helpful to have the support within Rkt as well.[376] While SELinux is enabled by default, the profile is extremely generic and may not be effective for a particular application. SELinux is also recommended to actually be disabled when trying out Rkt.[377] This mirrors the typical fact that SELinux is often disabled by many devops or system administrators.

[367] https://github.com/coreos/rkt/issues/2234
[368] https://github.com/coreos/rkt/issues/1585
[369] https://github.com/coreos/rkt/issues/1585
[370] https://github.com/coreos/rkt/issues/820
[371] https://github.com/coreos/rkt/blob/master/Documentation/trying-out-rkt.md
[372] https://github.com/coreos/rkt/issues/2188
[373] https://github.com/coreos/rkt/issues/912
[374] https://github.com/coreos/rkt/issues/1614
[375] https://github.com/systemd/systemd/blob/09541e49ebd17b41482e447dd8194942f39788c0/src/nspawn/nspawn.c#L1564
[376] As everything is Open Source, support could always be added manually, but some official profiles for Rkt would be a good start for the community.
[377] https://github.com/coreos/rkt/blob/v1.0.0/Documentation/trying-out-rkt.md

Weak or missing procfs and sysfs limits by default. Rkt is effectively missing a number of limits for procfs (`/proc`) and sysfs (`/sys`), allowing information to leak from the container host or easily allowing attacks from the guest container. This includes but is not limited to the following exploits discussed within 7.2.1 on page 52: uevent_helper, sysrq-trigger, core_pattern, and modprobe. While some protections are enabled by default via read-only bind mounts, these can be easily subverted by using `CAP_SYS_ADMIN` to remount the mounts as read-write.

9.13 Container Defaults

Listed below are the relevant security features for the three major container platforms explored within this paper. Each security feature is covered directly or indirectly within this paper and the title can be clicked, for those which are covered in detail, in order to jump to the relevant section. To avoid any misconceptions, the following parameters are defined as to their use in the table below:

- Default: The security feature is enabled by default.
- Strong Default: The most secure configuration is enabled by default.
- Weak Default: A less secure configuration is enabled by default.
- Optional: The security feature can be optionally configured. This is not a given weakness unless no other equivilant feature can be configured or enabled.
- Not Possible: The security feature cannot be configured in any way, no documentation exists, the feature is still under development, or the feature is not planned to be implemented.

Available Container Security Features, Requirements and Defaults			
Security Feature	**LXC 2.0**	**Docker 1.11**	**CoreOS Rkt 1.3**
User Namespaces	Default	Optional	Experimental
Root Capability Dropping	Weak Defaults	Strong Defaults	Weak Defaults
Procfs and Sysfs Limits	Default	Default	Weak Defaults
Cgroup Defaults	Default	Default	Weak Defaults
Seccomp Filtering	Weak Defaults	Strong Defaults	Optional
Custom Seccomp Filters	Optional	Optional	Optional
Bridge Networking	Default	Default	Default
Hypervisor Isolation	Coming Soon	Coming Soon	Optional
MAC: AppArmor	Strong Defaults	Strong Defaults	Not Possible
MAC: SELinux	Optional	Optional	Optional
No New Privileges	Not Possible	Optional	Not Possible
Container Image Signing	Default	Strong Defaults	Default
Root Interation Optional	True	False	Mostly False

10.1 Generation Container Recommendations

Security recommendations for containers are complex, and greatly vary depending on the type of system used (LXC, Docker, Rkt, etc.), the deployment scenario, if running untrusted containers is supported, what basic internal or cloud network hardening is present. The recommendations can also differ if it's being deployed to Dev/Test vs Production or containers are being used for desktop/clients as a form of sandboxing. Realistic capabilities and time of the devops or system administration team, as well as architecting a new solution can also impact what security recommendations are actionable. Finally, the level of configuration and customization required within a deployment, even apart from any technical limitations, may help dictate the container platform of choice.

In short, many different recommendations can be made for any container platform, and in almost any deployment scenario. However, these generally boil down to similar security recommendations for almost any system, platform or service:

1. Reduce all attack surfaces to only those required and harden what surfaces must be exposed.
2. Attempt to isolate based on trust, risk, exposure, in addition to network "zone" such as development vs production vs corporate.
3. Apply and enable all security relevant and supported configuration options for the platform.
4. Keep the host and container up-to-date and follow all industry or container-specific standards.
5. Regularly test and review the above security recommendations (1-4) for security gaps or implementation weaknesses.
6. Remember that complexity breeds insecurity, always try to keep implementations and code as simple as possible.
7. Consider and evaluate the security and trust placed in third party code, various cloud platforms and other elements common to containers such as continuous integration (CI).

The attack surfaces for containers and their applications must be carefully and systematically reduced or removed through the available kernel namespaces cgroups, capabilities, Mandatory Access Control (MAC), and syscall filtering. When deploying container solutions or when becoming certified, reviewed and or otherwise audited for various forms of compliance, it is also important to consider how some third party businesses or auditors may assess the system. Some older (in Internet years) institutions may have problems understanding the "container" or "microservices" model, as common deployments (container or otherwise) are traditional 3-tier monolithic systems.[378] The traditional "DMZ" is largely missing from microservices, and although it can now be built ever-stronger, it may be a pain point during auditing and compliance.

The following list of container-agnostic recommendations explore and expand upon the above high-level or generic recommendations that should apply to all containers. Following these, additional in depth recommendations which are container-specific for LXC (Section 10.2 on page 105), Docker (Section 10.3 on page 106), and Rkt (Section 10.4 on page 109). As with other sections within this paper, the order in which the recommendations are listed is not reflective of their importance.

[378] That is, a single web front-end or load balancer, web application or API within a DMZ and finally database in-line.

Follow standard hardening best practices for non-container systems. These generic recommendations for system or application hardening can easily be applied to container deployments.

- Continuously review the container configuration options and security features, strengthening defaults whenever possible.
- Practice and attempt to follow or encourage simplicity of mechanism, avoiding complexity when possible.
- Consider how authentication and authorization are handled within the system.
- Use strong encryption whenever possible, and especially for transport security. Standard encryption algorithms (such as AES and RSA) and protocols (such as TLS and IPsec) should always be used over in-house developed solutions.
- Reduce component scope when possible, isolating operations into discreet elements.
- Examine how the principle of least privilege is implemented within any container solution and overall architecture:
  - For privileged containers, drop all possible capabilities and use whitelist model whenever possible.
  - If writing new system daemons from scratch, if they must run with some form of elevated privileges, consider using `prctl(2)` to set no_new_privs.[379] See Section 5.7 on page 42 for more information or the Linux Weekly News article System call filtering and no_new_privs.
  - Consider hardening the container host itself to avoid highly privileged operations or users, as these users or daemons may be inadvertently exposed to containers through network access control failures or other misconfigurations.
- Investigate how the container follows the principle of least access. This can be both physical access, the location within a network or container to container and container to host network access.
- Develop a threat model for the deployment or architecture and understand how the containers and container hosts fit within it. This should be established through posing questions for the design such as:
  - If a malicious container is present on a host, what actions could be performed?
  - If a container host is compromised, what additional access within the environment could be gained?
  - If the local network between container hosts is compromised, what attacks would succeed?
  - How is the container prevented from accessing data, networks or interfaces which should be highly restricted?

Reduce the available attack surfaces. This can be performed through configuration hardening, security features such as seccomp and Mandatory Access Control, minimal Linux kernel builds as well as via small container base images or rootfs structures. Containers can even be used for a single binary..[380] See the specific Docker recommendations below (Section 10.3 on page 106) for an exploration of small base images.

Apply Linux kernel hardening, key for defense in depth and ensuring container isolation. The kernel is arguably the most key security component of any container solution, and the most difficult to secure. This recommendation is also explored in more depth within Section 10.5 on page 110.

[379] https://www.kernel.org/doc/Documentation/prctl/no_new_privs.txt
[380] https://medium.com/@kelseyhightower/optimizing-docker-images-for-static-binaries-b5696e26eb07

- Consider using a custom host kernel with a minimal set of loaded modules and compiled-in options. In an ideal case, only the required features should be present. When building this kernel, consider using compile-time hardening protections such as CONFIG_CC_STACKPROTECTOR_STRONG[381]
- Keep the kernel as up-to-date as possible, having a process in place for upgrading container hosts on a regular basis and a process for emergency updates. In some cases, such as leveraging KSPLICE[382] it may be possible to perform kernel updates without rebooting. This can also help when a known flaw is released but is not patched within upstream kernels.
- If at all possible, strongly consider using grsecurity and PaX patches for any custom kernel. This significantly hardens the kernel against a wide range of exploit techniques and known weaknesses. However, for containers to operate or run properly alongside grsecurity, a number of defaults may need to be modified using sysctl before locking the settings down. This includes but is not limited to different chroot restrictions which default to enabled. See "Hard Containers"[383] additional information as well as 10.5.2 on page 111.
- Typical sysctl hardening should be applied.[384] Specifically for containers, the following few options should be enabled at minimum (beyond other defaults and network sysctl hardening):
  - `kernel.dmesg_restrict=1` - Preventing access to the kernel ring buffer for non-administrative users, unprivileged user namespaces containers will also be included in this restriction.
  - `kernel.randomize_va_space=2` - Enable the strongest form of Address Space Randomization (ASLR) within the vanilla Linux kernel for userland processes. This chiefly randomizes the heap/brk between executions.
  - `kptr_restrict=2` - Restrict kernel symbol addresses from being easily discovered by even privileged users. Disclosure of these addresses undermines KASLR and are often used within kernel exploits.
  - `kernel.sysrq=0` - Disable system rescue mode, unlikely to be used on modern systems.

Apply traditional disk and storage limits and security. Consider using separate physical storage block devices, or partitions for containers and their related volume mounts, metadata, rootfs images and other container data. This can increase speed, allow for better isolation and can provide defense in depth against DoS attacks targeting the host.

- Standard mount security options should use `nodev`, `nosuid` and `noexec` should be applied where possible. More advanced options such as bind mounts, using overlay filesystems, and temporary volume mounts can also be used once the basics have been applied.
- Consider using extended filesystem attributes such as immutable flags on critical configuration files or append-only flags on sensitive log files for additional restrictions and defense in depth.

Control device access and limit resource usage using Control Groups (cgroups). While the configuration of cgroups is often left to defaults, this is typically only related to devices themselves. These container defaults can be increased through tailored resource limits, (are often disabled by default for "out of box" usability reasons).

- Containers should carefully expose host and kernel devices, only doing so as required. The default deny

[381] This may add a performance penalty, but offers better security over CONFIG_CC_STACKPROTECTOR_REGULAR which is used in Ubuntu by default.
[382] http://www.ksplice.com/
[383] https://blog.flameeyes.eu/2012/04/hard-containers
[384] https://github.com/konstruktoid/ubuntu-conf/blob/master/misc/sysctl.conf

model for devices within cgroups should be the bases for any access controls. This also helps prevent attacks which leverage `CAP_MKNOD` in privileged containers to create new devices dynamically.

- By using container management software or direct configuration, cgroups for resource limits on CPU, memory, and disk usage should be applied to avoid potential Denial of Service attacks.[385]

If compiling native code for use within a container, always apply compile-time hardening options. For Ubuntu and Debian Linux systems, consider using the hardening-wrapper virtual package which applies many of the following features by default. This includes but is not limited to:

- Complier flag `-fstack-protector-strong`: Enables "strong" stack protection via canary values. This was released by Google in GCC 4.9, which heuristically protects more functions than the older version `-fstack-protector`, yet it may still miss protecting some. To avoid missing stack protections for any functions, use the `-fstack-protector-all` declaration. This protects all functions regardless of the stack buffer size, however this comes at a possible performance cost. See the "Strong" stack protection for GCC article by Jake Edge of Linux Weekly News for more information.
- Complier flag `-D_FORTIFY_SOURCE=2`: Provides a number of runtime protections for unsafe areas of libc (such as format strings) as well as some buffer related protections. Note this is only activated when code is compiled with -O1 or higher optimization.
- Compiler flags `-Wformat -Wformat-security`: Enabling warnings which may catch coding mistakes related to format strings.
- Linker flags `-Wl,-z,relro`: Providing read-only relocation tables for produced ELF binaries.
- Linker flags `-fPIE -fpie`: In order to fully support Address Space Layout Randomization (ASLR) and PIE (Position Independent Executables[386]). This is required for ASLR to be effective at protecting binaries, see A look at ASLR in Android Ice Cream Sandwich 4.0 for examples and more information by security researcher Jon Oberheide.

Limit the network attack surfaces from several different perspectives. Due to the default use of bridge networking, containers can often freely communicate with other containers as well as with any network daemon on the host which is listening or bound to all interfaces or otherwise bound to 0.0.0.0, a common misconfiguration. There are also issues relating to ARP spoofing within Docker and LXC, as the default bridge interfaces work similar to normal networking switches, each attached virtual interface corresponds to a single "physical port".
Protections include cross-container layer three traffic access control via iptables, cross-container layer two traffic limits via ebtables[387] and general container to host traffic via the bridge interface. If the bridge interface is not used, and instead shared-host networking is in place, attempt to limit the host attack surface via MAC systems.

- The container should be isolated from the host network daemons first and foremost in order to eliminate potential escapes and prevent access to potentially sensitive services (such as mistakenly exposed Docker Daemon).
- Access control should be restricted between containers on the same host or different hosts in order to prevent lateral movement or a compromised container affecting other systems.

[385] Some advanced features for resource control may be missing direct support from the container platform. For instance, disk performance controls or restrictions are not currently supported in Docker.
[386] https://en.wikipedia.org/wiki/Position-independent_code
[387] http://ebtables.netfilter.org/

- Restrictions should be in place for traffic bound for local or upstream networks, only allowing the required communication, ports or protocols. This is especially the case for any container management or monitoring solutions which may offer a bridge to then unrestricted networks. This is also critical for upstream networks, host-related management systems or cloud metadata services (*e.g.* http://169.254.169.254), which may contain sensitive secrets or allow for lateral movement if compromised.
- For LXC and Docker, consider implementing different bridges based on trust as opposed to the default lxcbr0 or docker0 bridge for all containers. This may help isolate some attacks such as ARP spoofing.[388, 389, 390] The most secure configuration would use a separate bridge for each container or if that is not possible, a separate bridge based on trust or container type.

Managing security artifacts such as secrets, keys, passwords or other sensitive information should be a key security design consideration. While the managing, storage, rotation, distribution and overall security of secrets within a container environment may not be a concern for some deployments, it can be a major consideration in others. Secret storage and distribution is a subject of considerable ongoing debate for container platforms,[391] although it is not limited to just containers or their hosts. Depending on the container environment, types of secrets and duration of their use, several best practices should be followed:

- Avoid storing security artifacts in source code, as these can leak in various ways. Secrets within source code are also difficult to rotate, can reduce the overall uniqueness of any secret and can easily be reverse-engineered or otherwise extracted.
- For Docker, avoid storing security artifacts within Dockerfiles, these could be exposed within Docker container history, image information or via malicious registry access. Secrets within Dockerfiles also have many of the same problems as storing secrets within source code.
- Security artifacts within environment variables risk exposure in a number of different areas. This includes but may not be limited to application features, log files or any local file read or file disclosure vulnerabilities through `/proc/self/environ`. Environment variables are also typically passed to any child processes, which may increase their exposure depending on the application and system.
- Temporary volume mounts or temporary files within a volume mount can be used to inject security information into a running container before removing the files or mount and keeping the information in-memory. This helps eliminate long-term access and reduce risk of local file read vulnerabilities.
- Specifically designed secret distribution systems such as Vault by Hashicorp or Keywhiz by Square should be strongly considered. These purpose built security artifact distribution tools have varied support for containers but a large amount of documentation or success stories.
- Message Queue (MQ) systems can provide for temporary security distribution "topics", although bootstrapping these securely can be problematic. If MQ systems are used within the overall architecture, consider using the same or similar mechanisms for managing security tokens to help reduce complexity.
- If key material security is a critical concern, consider the use of a Hardware Security Module (HSM) or a similar well-vetted software implementation. These systems typically have antiquated APIs, so access may need to be wrapped in yet another layer or API. If this is performed, ensure the overall security model is maintained and strong access controls, rate limiting and auditing are in place.

Use the relatively new hardening features of modern Linux kernels such as user namespaces for defense in depth and Seccomp-BPF for a reduced kernel attack surface. These features are further discussed and implementations explored within Section 8.1 on page 66 and Section 8.3 on page 74 for each platform respectively. It

[388] http://events.linuxfoundation.org/sites/events/files/slides/secure-lxc-networking.pdf
[389] https://lists.linuxcontainers.org/pipermail/lxc-users/2011-May/002025.html
[390] https://nyantec.com/en/2015/03/20/docker-networking-considered-harmful/
[391] https://github.com/docker/docker/issues/13490

should be noted the seccomp ruleset should follow a whitelist approach and specifically avoid allowing for `ptrace(2)` due to known risks.[392]

- **For LXC containers:** Seccomp-bpf is supported by default starting in version 1.0 but only a limited blacklist filterset is applied by default. The `lxc.seccomp` template directive should be used which allows for both syscall whitelists and blacklists based on the system call number. If profiling the application or system to be deployed within the container is not feasible, at the very least consider implementing a blacklist based on known dangerous or privileged syscalls.
- **For Docker containers:** Seccomp-bpf is enabled by default starting in version 1.10 (on Linux kernels which support it and when using seccomp version 2.2.1, not part of Ubuntu Trusty)). The seccomp filter is using a whitelist of roughly 300 system calls. While this default whitelist offers attack surface reduction, custom per-container profiles should be used by passing the following directive on the command line: `--security-opt seccomp:/path/to/seccomp/profile.json`.
- **For Rkt containers:** No direct support of either version of seccomp is currently supported, although systemd use is encouraged. The systemd configuration appears straightfoward[393] although documentation is lacking.

Apply a Mandatory Access Control (MAC) System which will likely be AppArmor or SELinux. MAC systems are crucial for containers not taking advantage of user namespaces, or failing to drop capabilities. With a user namespace in place, MAC systems contribute to greater defense in depth. For AppArmor, Docker and LXC have strong support and many examples of both base default profiles, and extended container-specific profiles. For Docker, the Bane tool can be used to develop specific profiles and for LXC, traditional AppArmor tools can help profile the application. The grsecurity project also offers an excellent Role Based Access Control (RBAC) although most container platforms do not easily support or integrate with it. Finally, SELinux is supported within all three platforms, although documentation is often weak.

Consider access controls to the container environment. For example, avoid using OpenSSH within all containers. While this can be difficult for development or QA teams attempting to debug issues, consider creating a separate staging environment which allows quick and easy SSH, and then using a production environment which has some additional verification steps, bastion hosts, multi-factor authentication, restricted access, and other security measures for access to container hosts. The same security should be in place for container orchestration and management software.

Have a plan and procedure in place for regular container updates in addition to emergency fixes, taking into account potential upstream lag time. With LXC, this could be as simple as running apt-get upgrade, as containers are often full Linux distributions. For Docker, this style of image updates is considered by many as "bad form". Container related infrastructure is intended to move towards being immutable and images should not be modified once built. For Docker, upgrading should involve building and deploying a new container image, then having services switched over (obviously easier in a high-availability environment). See 9.8.2 on page 91 for more information. In addition to container updates, the container host and kernel will need to be updated, traditionally requiring a reboot. The Ksplice project allows "rebootless" container project hosts via in-memory kernel patches. The project is open source, although it is now owned by Oracle. Instructions for Ubuntu[394] or docker[395] can be found on the Ksplice and Oracle websites respectively.

[392] https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt
[393] https://plus.google.com/+LennartPoetteringTheOneAndOnly/posts/cb3uNFMNUyK
[394] https://www.ksplice.com/uptrack/download-ubuntu
[395] https://blogs.oracle.com/wim/entry/oracle_linux_containers_and_docker

Isolate containers based on trust and exposure. Avoid having containers share the same network or host when conflicting trust, security models or container ownership is involved. This can be the case with web applications and cache sytems or multi-tenant systems, where "company owned" containers are used alongside customer container. In yet other cases, this may involve development running alongside production. For security, resource control and isolation should limit attempt to the effect of any compromise.

Access controls at the most basic level should be well understood, such as read or write access. This can involve exploring the security model and posing questions. What does a container specifically need write or read access to? Can a container be entirely read-only or write-only as a strong security barrier? For example, simple containers which handle untrusted data and return the contents (such as parsing and resizing an image, uploaded from a potentially untrusted user). Other container deployments, such as a web server with static files or an SFTP server for a "dropbox", could leverage read-only or write-only volume mounts.

Strongly consider having a single application per container. "App VMs" should be preferred over "OS VMs". Not only does this avoid fully virtualized Linux distributions, it keeps attack surfaces down, reduces complexity, and increases efficiency. It also encourages application specific security options or hardening. MAC rules or syscall filters can now be tailored for single application.

Privileged container actions should be always be established during container start-up and executed by binaries in the host. By trusting any data within the container, or by requiring the container system to perform some privileged operation a number of risks present themselves. Vulnerabilities can be introduces by increasing capabilities to risking attacks against the container management software. All data within the guest should be treated as untrusted. Recent Docker and LXC AppArmor bugs (CVE-2015-3631 and CVE-2015-1334) for examples or libvirt issues such as CVE-2013-6456 and CVE-2014-0179.

Carefully expose containers on specific network interfaces and understand what potential networks or hosts can reach the container service. Only expose container interfaces to the specific host interfaces or bridges required, as opposed to all interfaces. As with all network security, other standard security recommendations apply. This includes, but is not limited to, understanding how the exposed service is authenticated by clients clients, how the server authenticates clients (if required) and how encryption offers confidentiality and integrity protections.

When downloading or building LXC rootfs images, building Dockerfiles, downloading Docker images or Rkt App Container Images, always consider where images are downloaded from and the security of the related network traffic. Cryptographic image verification has come a long way since the inception for several of these container projects, but interoperability (in the case of Rkt), continued vulnerabilities, or Docker images being opaque, mandates taking care when downloading and verifying images.

Logging, auditing and monitoring is important for container deployment. With modern storage and processing, logging and monitoring should be verbose (without exposing sensitive values) and performed everywhere possible. Operations within container hosts, and the logging of applications themselves is key for testing, development, debugging and other efforts. Centralized logging and time synchronization is also critical in the event of any breach, compromise or other incident response.

Use hardware virtualization along application trust zones. Using a single container per VM (if performance or efficiency is not a key concern) is a nice way to leverage the advantages of both containers and hypervisors. In multi-tenant environments, this could allow for tenant isolation via hardware VMs and inner-tenant application isolation via containers. This is one example of isolating by trust (obviously, per-tenant physical machines would offer the best isolation, but that may not be realistic).

Application hardening itself, of the application being contained, is also a key step which cannot be overlooked

when deploying containers. By preventing an attacker from gaining any access to the system (usually by obtaining code or command execution within the application), the entire system benefits via this first line of what is hopefully a defense in depth strategy. First breaking into the container itself should be the initial hurdle of attacking any container system. Application vetting can be performed through code auditing, hardened tool-chains, strong permissions, least privileges, defensive design and internal or third-party application security testing.

10.2 LXC Specific Recommendations

LXC security recommendations largely encompass two key areas. Avoiding weak defaults through additional configuration and avoiding treating containers as full operating systems. As new defaults and security improvements are added as time goes on, always consult with the current documentation, best practices and third-party security guides.

Follow the above general security recommendations. Ensure layer two and layer three firewall rules are applied to limit container to host and guest to guest communication. Use user namespaces to further isolate the container from the host when appropriate. Enabling and using the user namespace within LXC is simply defining the required UID mapping files[396] within the LXC template, invoking LXC commands as an unprivileged user and using a supported kernel. These two security features are critical for additional LXC hardening.

Consider creating custom AppVMs or single-application containers as opposed to full Operating Systems. This will require additional development effort, however the lxc-sshd (as an example) and lxc-busybox templates can be used as a starting point for minimal container images. This allows for more specific security controls, less disk space, reduced update requirements and a minimal attack surface.

For privileged containers, drop any capabilities which are not required for the application or applications within the container. Use `lxc.cap.keep` within the LXC template or even apply it to the default configuration file, which will follow a whitelist model. Performing capability restrictions is much easier when using App VMs instead of fully containerized operating systems. Privileged containers should always be avoided if possible, instead taking advantage of user namespaces.

Ensure default AppArmor profiles, providing Mandatory Access Controls (MAC) are applied. The base AppArmor profile (typically `/etc/apparmor.d/abstractions/lxc/container-base`, or found on github[397]) offers a good defense in depth against container escape. The `lxc.aa_profile` directive should not be set to "unconfined", which disables AppArmor. In many ways, LXC defaults for privileged containers require AppArmor as part of the security model, enforcing many security options. This is mainly due to many default containers retaining the highly privileged `CAP_SYS_ADMIN` capability, as some OS operations are used within the container (such as the container fstab mounting `/dev/shm` in some default templates). It is also recommended to explore developing container-specific profiles.[398]

Consider using seccomp-bpf to reduce the attack surface exposed by the Linux kernel. The `lxc.seccomp` directive supports both blacklists and whitelists and multiple versions. While some syscall blacklisting is applied by default, this is not comprehensive. Explicitly blocking x86 vs x86-64 system calls within this configuration file is non-trivial[399] and even some outstanding weaknesses exist.[400] Finally, be sure any seccomp-bpf profile filters block access to `ptrace(2)`, in addition to dropping the related capability (`CAP_SYS_PTRACE`) if

[396] Using a different UID shift or slide for each container is also a good idea, although it is largely an additional defense in depth measure.
[397] https://github.com/lxc/lxc/tree/master/config/apparmor
[398] https://help.ubuntu.com/lts/serverguide/lxc.html#lxc-apparmor
[399] https://plus.google.com/+MathiasKrause/posts/Mj4j7UK6NSn
[400] https://github.com/lxc/lxc/issues/571

possible.

Use cgroups to restrict access to all devices, then only permit those required. The lxc.cgroup.devices.deny parameter should be set to "a" for all (the default). The LXC template should explicitly allow only the required devices via lxc.cgroup.devices.allow.

Use custom mount options to increase defense in depth. Each lxc.mount.entry directive should use a selection from or combination of the following security related mount options (nodev, nosuid, noexec and bind). For example, no device files should ever be created within /lib or no binaries executed from the web server directory root.

10.3 Docker Specific Recommendations

The Docker ecosystem is a complex collection of software. Recommendations may change for specific versions and generic yet actionable recommendations for a broadly-used system a difficult task. Regardless of this moving target, security recommendations for Docker are included below.

Follow the above general security recommendations. Ensure layer two and layer three firewall rules are applied to limit container to host and guest to guest communication. Enable and use user namespaces and the base seccomp profile to further isolate the container from the host when appropriate. These three security features are critical for Docker hardening over the defaults.

Closely follow Docker development and upgrade whenever possible. At the time of this writing, user namespace and seccomp-bpf support are now included within the Docker 1.10 release. Both security features are implemented at a basic level, although neither feature are enabled by default. The Docker Engine, daemon and client should always be upgraded upon release, especially in the case of a security vulnerability or additional security feature.

In Docker 1.11 and later, consider using the "No New Privileges" feature. GitHub Pull 20727 merged optional support for the new "No New Privileges" feature, better supporting a least privileges model. This can be enabled via the --security-opt=no-new-privileges flag for docker run. This security feature is further explained within Section 5.7 on page 42 and relevant kernel documentation for prctl(2).[401]

In Docker 1.11 and later, consider using the PID cgroup to limit the maximum number of processes. This will help prevent fork-bomb style attacks via intentional resource consumption or a legitimate runaway process. Additionally, the --kernel-memory memory flag can be used to limit the maximum memory. See Issue 6479 and PR 14006 for more information.

Avoid using Docker with the --privileged flag at all costs. As discussed within the Docker configuration section (Section 6.2 on page 45), the --privileged option effectively disables several major container isolation features and provides little or no security.

Avoid running as root within privileged non-user namespace containers. Dockerfiles should almost always include a useradd command and USER directive. This can add some defense in depth, similar to not running as root for normal host operations. However, now with user namespaces released as part of Docker 1.10 it is not necessarily required.

Develop container specific AppArmor rules for containers. Apply the --security-opt=apparmor:<profile> flag when running Docker. For help developing a policy, see the Bane tool by Jessie Frazelle who worked for a long time at Docker. The genSeccomp.sh script by Thomas Sjögren can additionally be used to help

[401] https://www.kernel.org/doc/Documentation/prctl/no_new_privs.txt

generate an AppArmor policy, however Bane was more recently released (and may be better supported).

Use `--read-only` when running containers and overall consider building an overall immutable architecture. Immutable containers offer many benefits such as limiting attack scenarios, helping prevent compromise, simplifying deployment and allowing for easier upgrade paths. Although this new paradigm may require rearchitecture, retooling, adjustment and refactoring, the end result will be easier to manage and secure. The idea of immutable images is growing as a deployment trend, in virtualization, cloud and container architectures. Some also advocate for not storing application data within application containers. See Data-only container madness for additional information. See Making Docker read-only in production and Immutable Infrastructure with Docker and Containers by Jérôme Petazzoni of Docker, Building a glass house by Jason Chan of Netflix or Immutable Infrastructure with Docker and EC2 by Michael Bryzek of Gilt for more information.

Avoid providing access to the docker user or docker group. As discussed within Docker specific threats (Section 7.4 on page 62) and Docker specific configuration (Section 6.2 on page 45), it is widely accepted[402] a user with docker privileges can trivially escalate to root. While it can be tempting to allow access to the docker group, this is essentially the same thing as providing root. Only provide such access for users which are expected to be able to gain root access on the host where such Docker permissions exist. See Docker daemon attack surface within the Docker security documentation. Privileged access will be required until future Docker versions or runC[403] allow unprivileged users. Always grant access to the Docker daemon carefully.

Avoid providing access to the Docker UNIX socket or REST API to potentially untrusted callers or containers. Providing access to the Docker UNIX socket within a container should always be avoided. This exposure may be due or even recommended in some cases for introspection or to allow management of the Docker daemon from within a Docker container itself. Just as access to the docker user or group can provide a trivial path to root, access directly to the REST API or UNIX socket can just as easily compromise the security of the entire host (and therefore and all containers running within it).

If the RESTful API is exposed, always enable TLS and authentication (both of which default to disabled). See the Docker specific threats in Section 7.4 on page 62 as well as Protecting the Docker daemon socket on the Docker site for more information. TLS verification for the Docker command-line client can also be set via the `DOCKER_TLS_VERIFY` environment variable.

If user namespaces are not in use, containers should only retain the required capabilities using `--cap-add`. If for some reason user namespaces cannot be used, drop all but the required capabilities for Docker containers. If this cannot be done, drop potentially risky capabilities which remain enabled by default using `--cap-drop` such as the default `CAP_NET_RAW` and `CAP_MKNOD`. Note that using `--cap-add` implies dropping all other capabilities.

Consider using the docker-bench-security tool by Docker. The docker-bench-security tool checks for "dozens of common best practices around Docker containers in production". As this is a simple bash script, it could easily be extended for specific needs, security regression tests or additional company security requirements. It should be noted the docker-bench-security system is required to run with an extreme level of privilege, apparently required for the verification. This dictates sensitive access on deployment of this privileged container.

Attempt to use small base images within Dockerfiles. This includes replacing `FROM` calls to large distributions

[402] One of just many examples: http://reventlov.com/advisories/using-the-docker-command-to-root-the-host.
[403] https://github.com/opencontainers/runc/issues/38

such as ubuntu or centOS which themselves use large rootfs images with dependency friendly package managers with smaller, purpose-built or minimal distributions. For example, replacing Ubuntu Linux with Alpine Linux, which also offers its own package management.[404] This reduces attack surfaces, reduces complexity, image footprint size, and likely future patching requirements. Minimal container base images also more closely support the "App VM" model, rather than having a large container image with lots of different applications and only one running. Continuing the trend of specific container base images, minimal containers can be made even smaller by using the `FROM scratch` within Dockerfiles or simply using `docker import` on few files a binary requires. See Create the smallest possible Docker container for more information. Finally, some offerings may even auto-generate seccomp profiles and perform other hardening along with supporting minimal containers.[405]

If using SELinux for Manditory Access Control (MAC), use different SELinux labels for each container with `--security-opt`. This allows specific control and custom labels to be applied to different containers on the same host. See Adjusting SELinux labels by Daniel Walsh for more information.

Exercise caution when exporting ports or exposing containers to the network. Docker defaults expose the container on all system interfaces. This may allow a container to break expected network controls or open a container to unexpected network attacks. This includes command line parameters –p and –P in addition to `EXPOSE` directives within created or downloaded Dockerfiles.

Avoid using the LXC runtime via `-lxc-conf` flags for the Docker Daemon. This is currently unsupported, and is likely to create inconsistencies with expected Docker image behavior and security assertions, such as restrictions to procfs not being applied when in LXC mode. Prior reasons to use LXC included support for seccomp and UID mapping for the user namespaces, however this is now unnecessary.

Explore Docker and container auditing tools. Tools such as drydock, CoreOS Clair, docker-bench-security, and Docker Project Nautilus offer methods to audit the security of your Docker configuration and containers. Drydock offers configurable templates, Clair can scan for known issues patched in upstream repositories and docker-bench-security runs a number of standard security benchmarks (although some implemented security checks may be outdated). Docker Project Nautilus takes the idea of Clair but goes a bit further, scanning within all container binaries themselves rather than using package metadata.[406]

Follow best practices when writing DockerFiles. Dockerfiles are a key area of security-related configuration and the resulting Image security for any Docker container. See Dockerfile best practices by Docker and Dockerfile best practices take 2 by Michael Crosby for specific information and great overall recommendations, not just for security impacting decisions.

Follow the development of Security Profiles and consider assisting if possible and implementing when ready. Outlined within Github Issue 17142 there is a desire to develop a "security profile" for Docker containers which will use a combination of seccomp, capabilities and MAC to restrict the operations of processes within a container and limit potential attack surfaces. This could improve the default security of many widely used Docker images.

If possible limit the container to container communication by using `-icc=false`. This disables the blanket inner-container communication by applying a default DROP iptables policy, however layer two communication may still be permitted (as iptables only controls layer three traffic). Containers should always be restricted

[404] Alpine Linux is a small distribution or development team, with unknown build-chain hardening for packages and without HTTPS repositories (although packages are signed, this exposes an additional attack surface).
[405] NCC Group has not evaluated the effectiveness of this tool and has no relationship with CloudImmunity https://github.com/cloudimmunity/docker-slim.
[406] There are no details yet on how exactly the system will work, but they will be released soon (as of 4/06/2016) according to Docker.

from the host and each-other whenever possible to avoid trivial cross-host attacks (via each container within a network), and cross-container attacks within the same host. Note this should be configured in conjunction with dropping the `CAP_NET_RAW` capability, and possibly using per-container bridges to avoid ARP spoofing attacks.[407] See Communication between containers within the Docker Networking documentation.

Follow existing hardening guides to supplement or augment this whitepaper. Several Docker hardening guides and configuration benchmarks have been released over the last two years. As with anything related to documenting technology (and particularly security), some recommendations may be outdated. This includes but is not limited to:

- The CIS Docker Engine 1.6 benchmarks provides good information about hardening and explores a myriad of configuration settings. This was a collaboration between Docker security team members, Jérôme Petazzoni and VMware, Rakuten, Cognitive Scale and the International Securities Exchange.
- An Introduction to Container Security by the Docker team explores the basics of Docker, process, file and device restrictions as part of Linux containers powered by the Docker Engine (libcontainer). A brief overview of Docker Image security is also included. See Docker security and best practices on the Docker blog for more information.
- Docker Secure Deployment Guidelines by Gotham Digital Science offers a clear list of hardening items and direct recommendations based on public presentations and other information. While this document is roughly a year old at time of writing this paper, and the version of Docker discussed is 1.4 (11/12/14), many of the guidelines still apply.

10.4 CoreOS Rkt Specific Recommendations

As discussed within the main overview, the Rkt container framework is under is under active development, and many key security features remain merely planned or under discussion. As of Febuary 2016, Rkt claims it is ready for production. RedHat, Intel and VMware are supporting the CoreOS Rkt in various parts, but some supporters (RedHat) do not think Rkt is "ready for the enterprise".[408]

Follow the above general security recommendations. Most if not all of the above generic recommendations will apply to Rkt containers.

Always use the newest Rkt version. Due to the rapid development, new or expected security features and bugfixes (both vulnerabilities and otherwise) are crucial to obtain as soon as possible.

If possible, avoid sharing the network namespace with the host. Rkt users should not use the `--net=host` flag. The risks are numerous and range from accessible network daemons, interface sniffing, RAW packet injection and access to abstract sockets. For Rkt, if layer two protections are not setup for protecting from against potential attacks, consider implementing a per-container network bridge.

Consider using the LKVM stage1 for high risk containers and normal kernel namespaces for others. This allows the isolation and security to match the risk profile. This could even be conditionally applied to the same container image, depending on where in the network it is deployed or the user accessing the system. See Running rkt with an LKVM stage1 for more information and a walkthrough.

Use SELinux with Rkt, at least until other Manditory Access Control solutions become available. SELinux can be used in order to help contain different Rkt containers on the same system from accessing the host or

[407] https://nyantec.com/en/2015/03/20/docker-networking-considered-harmful/
[408] http://rhelblog.redhat.com/2015/05/05/rkt-appc-and-docker-a-take-on-the-linux-container-upstream/

from other malicious or compromised containers. Currently the documentation is sparse at best[409] and the use of SVirt is required (likely due to RedHat integrations). This may create implementation difficulties on non-Redhat based Linux distributions.

Always use `rkt trust` before issuing `rkt fetch` in order to ensure the Image is verified correctly, as opposed to purely trusting TLS and the specified endpoint automatically. The defaults when using `rkt fetch` will download the image using "meta discovery" (a meta tag `ac-discovery-pubkeys` on the target server) and trust the key associated with it automatically without user interaction or specification.[410] The automatic trust functionality is still under continued work by the developers, see Github issue 481 for more information. Additionally, users can set the `TrustKeysFromHttps` flag to false and avoid this behavior.

Follow the instructions for basic privilege separation. Although the majority of Rkt commands must be run as root, it is possible to run image downloading commands as an unprivileged user. In order to follow the principal of least access, even this small portion of code should run with reduced privileges. For `rkt fetch`, non-root users can be created by executing the shell instructions found within the Rkt getting started page.

Carefully add users to the "rkt" group. Similar to the "docker" user and group for Docker, the "rkt" group should be considered privileged and will allow privilege escalation in the host through malicious use of containers. Avoid providing access to users you intend to be restricted on the container host (this also includes the "core" user within CoreOS).

Due to a lack of image verification for Docker images, consider manual verification. When using Docker images within Rkt, be sure to not disable TLS verification in order to create a secure connection to the Docker server and then attempt to verify the image out of band. Some efforts are underway to enable this feature.[411]

10.5 Relevant Kernel Hardening

> **"Containers will always (by design) share the same kernel as the host. Therefore, any vulnerabilities in the kernel interface, unless the container is forbidden the use of that interface (i.e. using seccomp)"**
> - LXC Security Documentation by Serge Hallyn, Canonical

As the Linux kernel is the underpinning of security or isolation features, hardening the Kernel against attack and limiting the attack surface should be considered one of the top priorities. While the history of Linux kernel vulnerabilities should be a whitepaper in itself, understanding some of the higher profile vulnerabilities and historical issues helps us to understand kernel attack surfaces and guide attempts to eliminate or isolate potential weak areas and restrict what is not required for container operation.

10.5.1 Hardening Actions

As a large number of exploits resulting in local arbitrary, ring-0 code execution resulted via two interfaces: syscalls such as `do_brk(2)`, `do_remap(2)`, `vmsplice(2)` to name a famous few or and network protocols (snmp, econet, CAN, SCTP, UDP). Both of these interfaces can now be restricted via seccomp-BPF, module whitelisting or a minimal kernel build. Additionally, a small number of additional weaknesses in procfs, debugging and filesystems have also been discovered, and these can be restricted by removing support or limiting access. See Section 7.1 on page 49 for a more in-depth overview of the various kernel threats and past attacks.

Current kernel exploitation methods are non-trivial (especially so given the default hardening (such as KASLR,

[409] https://github.com/coreos/rkt/blob/master/Documentation/selinux.md
[410] https://github.com/coreos/rkt/pull/1239
[411] https://github.com/coreos/rkt/issues/2188

ACPI protections and the closing of many address leaks which some distributions apply,[412] but not overly difficult. When faced with a hardened, Grsecurity kernel, attackers must utilize a stack information leak and then use "stackjacking".[413] This is in addition to chaining several weaker vulnerabilities into a single, combined privilege escalation vulnerability. Similar to recent Google Chrome browser exploits are built, which require the combination of multiple and often very specific vulnerabilities to achieve a complete system compromise. A tiered sandbox, which models defense in depth, both decreases the likelihood of escape, increases the complexity of the required exploit and narrows the possibility of a successful exploit chain. Security can often be reduced to making attackers work hard, or in some cases, as the common saying goes "You don't have to outrun the bear".

Creating a minimal hardened kernel[414] is not typically done (or is even a default option) with commodity Linux distributions, which must support a wide array of hardware, software and use cases from servers to laptops. Selective and specific kernel options can reduce the significant attack surfaces and decreases the potential vulnerability window for exposed systems by simply including less code in the built kernel. The kernel itself can also be built with compile-time hardening, both `CONFIG_CC_STACKPROTECTOR_STRONG` to protect a reasonable number of functions (20%)[415] vs `CONFIG_CC_STACKPROTECTOR_REGULAR` at just 2.81% of functions protected. Using the STRONG option may add a performance penalty, but offers better security over REGULAR (which is used in Ubuntu by default). It is worth adding Brad Spengler of Grsecurity does not believe in kernel stack smashing protection, as illustrated by several LWN discussions.[416, 417, 418, 419]

When building a new kernel, reviewing the configuration of sysctl values, performing hardening and implementing additional security features, I would recommend starting by reviewing the Ubuntu hardening steps incrementally and increasingly implemented by the Ubuntu security team for default kernels, Hardening Debian or the Gentoo Hardened projects. When building any custom kernel, it is still important to keep it consistently kept up to date, so the process for building, testing and deployment should be practiced and well staffed. Finally, various regressions and kernel security features can be tested by using Ubuntu's test-kernel-security.py script.[420] This script checks for around 60 different kernel security related regressions or misconfigurations.

10.5.2 Grsecurity

The Grsecurity/PaX project creates the opportunity for a significant barrier against successful kernel exploitation through their available patchset. This protection should be considered required hardening for any highly security sensitive or at risk system, but especially so for well-hardened OS-virtualization environments.

The core focus of Grsecurity/PaX is to harden the kernel via the "prevention and containment"[421] of exploitation techniques. This core idea introduced the first version of non-executable pages `NOEXEC`[422] and Address Space Layout Randomization (ASLR) from pwnie-winning Grsecurity/PaX team member pipacs[423] which is

[412] https://wiki.ubuntu.com/Security/Features
[413] Brad Spengler mentioned via email this is no longer an available attack. See slide 22 of https://grsecurity.net/the_case_for_grsecurity.pdf
[414] http://cecs.wright.edu/~pmateti/Courses/4420/HardenOS/#sec-7
[415] https://lwn.net/Articles/584225/
[416] https://lwn.net/Articles/354454/
[417] https://lwn.net/Articles/354481/
[418] https://lwn.net/Articles/354462/
[419] https://lwn.net/Articles/269532/
[420] http://bazaar.launchpad.net/~ubuntu-bugcontrol/qa-regression-testing/master/view/head:/scripts/test-kernel-security.py
[421] https://pax.grsecurity.net/docs/pax.txt
[422] This would later become Data Execution Prevention (DEP) on Microsoft Windows.
[423] "Microsoft today has announced a challenge, giving out $200,000 for work very similar to that that has been done and given away for free by pipacs, a decade ago" stated Dino Dai Zovi, who awarded pipacs with a lifetime achievement pwnie award: http://www.

now implemented in a large number of Operating Systems. This ranges from OpenBSD and FreeBSD to Apple OSX, Apple iOS, Microsoft Windows and of course the vanilla Linux kernel. For those developers or administrators considered with potential performance impacts, some positive information (for at least the majority of grsecurity protections as of 2002) can be found in provided overview slides by Grsecurity's lead Brad Spengler. More recent Grsecurity additions can be found in The case for grsecurity.

Grsecurity is fully compatible with various Linux container kernel features, including Mandatory Access Control (MAC) frameworks, and can easily be established in server environments (which require less drivers and typically less complexity is found). See Compile and patch your own secure Linux kernel with PaX and grsecurity by InsanityBit for a good overview of the process. Then Gentoo Hardened project also contains a great Gentoo PaX Quckstart guide. Finally, grsecuriy itself has a comprehensive Wikibooks series for further information on the features and options the patchset provides.

Often times, creating or running containers can violate existing default grsecurity hardening, you can temporarily disable these settings via sysctl for testing. Typically, the conflicting options are chroot_deny_chroot, chroot_caps, chroot_deny_mount, chroot_deny_mount, chroot_deny_chroot and chroot_deny_chmod although others may be involved depending on the container system in use and specifically enabled grsecurity features. Many protection features can be temporarily disabled, tuned and tested via sysctl then finally locked down (setting kernel.grsecurity.grsec_lock equal to 1) once the issues are resolved.

After 14 years, grsecurity stable patches are no longer publicly available. This is unfortunately due to "several very large companies in the embedded Linux world", who are "not playing by the same rules as every other company using our software violates users' rights, misleads users and developers, and harms our ability to continue our work", largely though false claims and continued trademark violations.[424] The test series, which is unfit for production use according to the grsec team, remains available "in order to avoid impacts to the Gentoo Hardened and Arch Linux hardened communities". Stable patches can be obtained by becoming a sponsor or commercial support. If Linux containers are used currently within your organization, or deployment of grsecurity is merely a security goal, I strongly suggest becoming a corporate sponsor to support this excellent work which benefits us all, as history as shown.

darkreading.com/attacks-and-breaches/pwnie-award-highlights-sony-epic-fail-and-more/d/d-id/1099384
[424] https://grsecurity.net/announce.php

11 The Future

While some may say containers are just a fad and too many have jumped on the hype train, it's hard to argue with the advantages, from efficiency to security. Containers are just starting to see deployment in large numbers to production, making inroads where hardware virtualization is king. Their use and implementations are also only recently reaching maturity in terms of security or documentation. Containers are clearly one future[425] OS-virtualization platform which can offer reduced attack surfaces and improve security controls.

The world of technology is also seeing a move to software-everything, just as application security is outpacing hardware solutions.[426] For example, software defined networking and overlay networks can architect designs and offer microsegmentation in ways never thought possible before, as well as offer dynamism that would be impossible in any hardware solution. Entire software defined data centers, powered by containers and hardware virtualization can achieve flexibility, scalability, and prototyping which was simply not possible (at least without large amounts of money), especially before the advent of cloud computing.

It should be noted that the following sections briefly discuss newly released, beta versions or otherwise upcoming technology or applications. While very interesting from a security, sandboxing or container standpoint, they have not been reviewed (nor is any one technology endorsed) by the author or NCC Group.

11.1 Containers on the Desktop

Desktop application containers[427] and the use of kernel features that power containers are a possible future for isolation on modern Linux or security-focused distributions. Ubuntu's ever-increasing base AppArmor profiles[428] and Gnome Desktop's Sandboxed Apps[429] project also illustrates similar efforts to sandbox the modern Linux desktop. Google's ChromeOS can also be used an example of the power of "containerized" client-focused Operating Systems.

When using containers on the desktop, unprivileged containers are a perfect fit.[430] Introduced in Linux 3.12 and LXC 1.0, fully unprivileged containers offer a solution for low-rights, non-root users a method to "containerize" desktop applications and retain a model of least privilege, although LXC often retains large base images for full "OS virtualization" unless something such as the busybox template is used to support "App VMs". Desktop application containers can also easily limit CPU, disk and memory for applications which like to monopolize these resources, such as web browsers.

A number of Open Source developers or container focused companies have demonstrated[431] GUI applications in containers or systems to isolate containers.[432] Past Docker employee and Linux junkie Jessie Frazelle also has given numerous talks and created articles discussing the use of containers on the desktop.[433] It should be noted that while X11 within a container is possible[434] the design of X11 remains a risk. Many Desktop container solutions do not attempt to isolate X11 at all, opting to expose the X11 magic socket into the container for transparent and full-speed X.[435]

However a new high-security and privacy focused Linux distribution, Subgraph OS, is attempting to offer OS virtualized applications and as well as address the core X11 risk. Subgraph OS also comes with a default grse-

[425] http://blog.circleci.com/its-the-future/
[426] https://www.youtube.com/watch?v=2OTRU--HtLM
[427] http://venturebeat.com/2015/07/09/cloud-on-your-desk-this-docker-container-comes-in-a-mini-shipping-container/
[428] https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/AppArmorProfiles
[429] https://wiki.gnome.org/Projects/SandboxedApps
[430] https://www.stgraber.org/2014/01/17/lxc-1-0-unprivileged-containers/
[431] http://www.flockport.com/run-gui-apps-in-lxc-containers/
[432] https://blog.philippklaus.de/2011/06/sandboxing-on-ubuntu-with-arkose-using-lxc/
[433] https://blog.jessfraz.com/post/docker-containers-on-the-desktop/
[434] https://web.archive.org/web/20140728171918/http://box.matto.nl/lxcxserver.html
[435] https://www.stgraber.org/2014/02/09/lxc-1-0-gui-in-containers/

curity patched kernel, is under active development,[436] and has a recently released alpha version which works surprisingly well, including many working security features such as a metaproxy app fiirewall, AppArmor profiles, seccomp-bpf and the OZ sandbox. Subgraph OS represents the next generation model of a high-security Linux distribution with client-side sandboxing architectures. You can explore their artistic design through their graph.[437] Many high risk applications within Subgraph OS are restricted via the "OZ" sandbox. This sandbox is described as "a sandboxing system targeting everyday workstation applications"[438] which uses Xpra[439] for X11 isolation, setting it apart from other implementations. Finally, Subgraph is developed by a respected set of researchers, committed to making Subgraph OS both secure and usable.

Subuser, developed by Timoth Hobbs, is another newly released client-focused container solution described as "docker on the desktop" or "QubesOS light", where applications are sandboxed using Docker containers, and X11 is again isolated using Xpra.[440] While Subuser is similar to "OZ" by the Subgraph OS team, it is not a full distribution. Subgraph OS is also a more comprehensive and security focused project with tight Tor integration and a grsecurity patched kernel, whereas Subuser is simply application containers via Docker with Xpra. Subuser is overall an interesting idea, and the permissions model may allow it to be usable and generic. However, it currently requires the main user to be in the docker group, which effectively means you're always running with elevated privileges. Maybe future versions will leverage runC, when it no longer requires root for unprivileged containers.

Overall, with modern smartphone platforms using application sandboxing by default such as Seatbelt within Apple iOS and UID sandboxing within Google Android, we may see a time soon where unsandboxed code is the exception, not the other way around. Sandboxing, namespaces, dropped capabilities and other restrictions are the security controls that modern Linux needs, in order to keep pace with other mainstream closed-source Operating Systems. See The State of the Art of Application Restrictions and Sandboxes: A survey of Application-oriented Access Controls and their Shortfalls by Z. Cliffe Schreuders Z, Tanya McGill, Christian Payne for more information on this overall subject.

11.2 New Potential Namespaces

While the current set of namespaces is adequate, a number of security gaps must be mitigated directly through security such as Mandatory Access Control or cgroups which, in many cases, should remain in place for defense-in-depth. Future Namespaces may include a device namespace[441],[442] a time namespace,[443] and a security namespace (for LSMs). With each new namespace, Linux containers will gain increased default security or can be more easily isolated for specific kernel provided features.

11.3 Additional Lightweight Isolation and Sandbox Platforms

While full LXC, Docker and Rkt are often the focus of Container discussions, there are a number of options available for simple sandboxes which leverage many of the same Linux kernel technologies (cgroups, namespaces, and capabilities) as containers. These lightweight solutions may be more adaptable to a specific need or are focused on desktop application sandboxing. For a more generic exploration of sandboxing, outside of using Linux containers, see The Methods of Sandboxing and Isolation by Ján Lieskovský.

[436] https://subgraph.com/sgos/index.en.html
[437] https://subgraph.com/sgos/graph/index.en.html
[438] https://github.com/subgraph/oz
[439] https://www.xpra.org/
[440] http://subuser.org/news/0.3.html
[441] https://lwn.net/Articles/564854/
[442] https://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2013_0.pdf
[443] https://lwn.net/Articles/179825/

11.3.1 Google Minijail

The Google ChromeOS team developed their own micro-container and sandbox platform "minijail" as a "library/application launcher for quickly creating restricted jails to run daemons inside of ChromeOS". Further information and implementation details can be found in the System design and hardening documentation, and the code can be found within the newer platform2 repository.[444]

11.3.2 MBOX

MBOX is "a lightweight sandboxing mechanism that any user can use without special privileges in commodity operating systems", and similar to other desktop solutions it can be invoked by an unprivileged user. MBOX achieves this sandbox through a layered sandbox filesystem and interposing on system calls to add hooks, user acceptance/confirmation for actions and other security checks. The MBOX sandbox can easily disable all network access for a target binary, control various signal handling, and present a fake filesystem to measure or snapshot changes (making it also useful for reverse-engineering or malware analysis). The syscall interpositioning/hooking supports both seccomp and ptrace, however ptrace is the default. As discussed earlier, syscall interposition frameworks are often vulnerable to various timing attacks such as Time-of-Check-Time-of-Use (TOCTOU). MBOX avoids this problem by simply and elegantly using a read-only memory page:

> **"MBOX avoids TOCTOU problems by mapping a page of read-only memory in the tracee process. When MBOX needs to examine, sanitize, or rewrite an in-memory data structure, such as a path name, used as a system call argument, MBOX copies the data structure to the read-only memory (using `PTRACE_POKEDATA` or the more efficient `process_vm_writev(2)`), and changes the system call argument pointer to point to this copy."**
> - Practical and effective sandboxing for non-root users by Taesoo Kim and Nickolai Zeldovich

The MBOX whitepaper Practical and effective sandboxing for non-root users and associated presentation slides offer an in-depth explanation on features or use and examples for different sandboxing goals. For more information on more general forms of TOCTOU attacks, see the iSEC Partners/NCC Group whitepaper Recognizing and Preventing TOCTOU by Chris Hacking.

11.3.3 Unshare

As discussed above, in some cases, a full container solution may be overkill and installing separate applications or sandboxing frameworks too burdensome, depending on the platform or storage constraints. As discussed earlier within this whitepaper, the `unshare(2)` system call allows for namespace "disassociation". This can be easily explored using the `unshare` command from the utils-linux package and with minimal effort, can allow for an extremely lightweight application container with support for all of the major namespaces. Other unshare-based container platforms have been developed such as a version written in the rust high-security programming language[445] or divehttps://github.com/vi/dive written in C. These essentially container methods, which directly use `unshare(2)`, may be ideal on embedded platforms where space, simplicity and compatibility are key concerns.

11.3.4 runC

The runC project, released in the summer of 2015, is a "a CLI tool for spawning and running containers according to the OCP specification" (OCP referring to the older name for the Open Container Initiative). This Docker project is under very active development and is powered by Docker's own libcontainer and is the first instance of an Open Container Project initiative and governed platform, confirming to version one of the various specifications.[446] The runC tool also supports running Docker images, and does not use the

[444] https://chromium.googlesource.com/chromiumos/platform2/+/factory-rambi-6420.B/libchromeos/chromeos/minijail/
[445] https://github.com/tailhook/unshare
[446] https://github.com/opencontainers/specs

Docker daemon, so it can be easily managed by other systems or integrated with systemd. As runC is backed by libcontainer, many of the libcontainer security features will work out of the box.

11.3.5 Systemd itself

The systemd init system can also be directly used to create containers. By using systemd-nspawn, lightweight application containers can be created, just as systemd-nspawn is used within CoreOS Rkt. See the Flockport article A Quick Look at Systemd Nspawn Containers for a basic overview of the use and configuration of systemd for containers. For additional analysis, see the LWN article Systemd vs. Docker by Josh Berkus for more information.

11.3.6 Firejail

Firejail[447] is a "SUID sandbox program that reduces the risk of security breaches by restricting the running environment of untrusted applications using Linux namespaces, seccomp-bpf and Linux capabilities". This simple sandbox offers a straightforward way to sandbox different Linux applications, although it is still under very active development. As is especially true with software providing security protections, make sure you have the latest release as Release 0.9.30 introduced a local privilege escalation bug[448] which was fixed in 0.9.32.

The numerous Firejail sandbox examples show that it can be used to secure a web server or a modern web browser or create a restricted shell. Firejail also supports user-defined "profiles" which can create rules for everything from filesystem controls (blacklisting, whitelisting and private directories), syscall filtering, and user namespace options to resource limits, environment variables, and network namespaces. Default profiles have been created for a large number of common desktop Linux software including Firefox, Chromium, Evince, Dropbox, Pidgin, Transmission, Spotify, Steam and VLC.[449]

11.3.7 Pflask

Pflask is another lightweight container solution which, similar to the unshare command, does not require configuration and supports unprivileged containers via user namespaces (when supported and permitted by the kernel).[450] Pflask is written in C and offers ephemeral containers, disabled capabilities, optional network and filesystem configuration as well as backgrounded containers and systemd-machined integration.[451]

11.4 The Open Container Initiative

The Open Container Project (OCP) later renamed to the Open Container Initiative (OCI), is a large multi-industry and multi-stakeholder "open governance structure" for container formats and runtimes. The OCI has over 35 sponsors and run under the auspices of the Linux Foundation. Standards and specifications are expected to slowly be publicly released in the coming months, with various implementations to follow. As referenced above, OCI has released, in partnership with Docker, the "runC" framework,[452] a lightweight wrapper around the native Golang libcontainer (which started under Docker and is now maintained as part of runC itself). Future versions of CoreOS Rkt are also intended to be OCI compliant. For more information, see the Open Containers project website.

[447] https://github.com/netblue30/firejail
[448] https://l3net.wordpress.com/2015/10/14/firejail-0-9-32-rc1-release-announcement/#more-4783
[449] https://github.com/netblue30/firejail/tree/master/etc
[450] https://github.com/ghedo/pflask
[451] https://www.freedesktop.org/wiki/Software/systemd/machined/
[452] https://github.com/opencontainers/runc

11.5 Containers In Other Platforms

One major benefit of Linux is it's flexibility, which includes a long history of very different hardware platforms. The use of Linux within the "Internet of Things" (IoT) is a direct testament of it's dynamic configuration, value, and support, as a vast majority of IoT and other embedded devices use Linux. Apart from the IoT, Google Android could also use more container features than simply the mount namespace and SELinux for MAC, either using additional namespaces, developing a custom Android-specific kernel namespace or using seccomp-BPF. All of these features could be used to further isolate processes and improve security through sandboxing or security beyond typical UID limits.

Containers, powered by kernel features, or simply different security and hardening options discussed within this paper (such as root capabilities and specific namespaces for one-off isolation techniques) are not commonly found on embedded systems and specifically, not found within the IoT landscape. Numerous vulnerabilities within these devices are routinely discovered, many of which may be mitigated, at least in part, by additional sandboxing, namespaces, filtering, and reduced root capabilities. Lightweight containers and related sandboxes can and should be used within various embedded devices, often which have extremely poor application security or run all processes with elevated privileges. Examples range from Linux powered routers and security hardware to kiosks and payment systems, as well as augmenting existing connected vehicle or automotive sandboxes, gaming systems or even infotainment systems in airplanes.

11.5.1 Microsoft Windows

Not to be outdone by the Open Source community and to keep pace with many enterprise customers, Microsoft has even released support for Docker images[453] using a custom runtime. According to one rather comical Hacker News comment, this means Linux containers are serious now:

> **"This will be what takes containers into the mainstream businesses. Companies may adopt docker or other instead of this, but Microsoft creating their own version of it means its a viable technology."**
> - Containers are serious now by Kirinan

The Microsoft fork of Docker started with a single, very important commit[454] and associated pull request.[455] Apart from Docker, Microsoft also has future plans to implement native OS-virtualization within the Windows platform.[456] See the Microsoft article Microsoft announces new container technologies for more information on native Windows containers.

11.6 Container Specific Operating Systems

Ubuntu Core, RedHat's Atomic Host, boot2docker, CoreOS and systems such as RancherOS are particularly interesting for hosting Linux containers. RancherOS in particular runs two different Docker engines, one for system services, which are each within Docker containers, and another for normal user applications. The author nor NCC Group has examined the security or hardening of these container Operating Systems, their patch cycles, Linux kernel configuration and other considerations, as this was deemed outside the scope of this whitepaper. See the Docker blog post The new minimalist operating systems and Tiny Docker Operating Systems for more information. Additionally, as a spin-out from RancherOS, plain "Rancher" offers a number of container orchestration features, Docker management and an overlay network (optionally protected by IPSec) all while coupling-in RancherOS. While this platform is in it's infancy, it may offer an "all in one" solution for self-hosted PaaS, similar to OpenStack but focused on containers.

[453] http://www.infoworld.com/article/2834122/application-virtualization/windows-server-is-getting-docker-says-microsoft-and-docker.html
[454] https://github.com/Microsoft/docker/commit/bad29cf9adb8fcc78347eb0a8c154c04a7b36e2e
[455] https://github.com/Microsoft/docker/pull/1
[456] https://msdn.microsoft.com/en-us/virtualization/windowscontainers/quick_start/container_setup

11.7 Unikernels and Microhypervisors and Hybrid Models

An idea to augment the OS virtualization model with Hardware Virtualization, attempting to create a hybrid platform with additional security is an evolution well underway. Joyent, public cloud container host also uses a "purpose-built container hypervisor" which is likely based on KVM, and follows the model of Solaris Zones as implemented with Illumos/SmartOS[457]). CoreOS Rkt, as discussed earlier can use Linux KVM for stage1, as co-developed by the Intel Clear Containers project.

11.7.1 Intel ClearContainers

> **"The impetus for the container effort is to embed security using VT-x technology"**
> - Intel Looks to Secure Containers by Imad Sousou

The Intel Clear Containers project is part of Intel's Clear Linux which provides a stripped down hardware VM (via KVM) for wrapping containers. This is exciting for many reasons, as it allows for hardware virtualization speed, security and isolation to be paired with OS virtualization efficiency and ease of use.[458] See the Clear containers project page for more information and a few different examples.

The Clear Containers project claims only 18-20 MB of memory overhead per container and on modern SSDs it can be started in less than 150 milliseconds. Clear Containers were implemented first within CoreOS Rkt, starting in 0.8. Docker support is still under development. See An introduction to Clear Containers by Arjan van de Ven on Linux Weekly News for more information.

11.7.2 Canonical LXD

LXD can be compared to many basic Docker features. To put this more simply, LXD is to LXC what Docker is to runC. LXD is working towards a "container hypervisor" that also offers live migration, unprivileged containers and at some point, hardware-assisted isolation. It is stated the primary goal of LXD is to "extend containers into process based systems that behave like virtual machines". It is rumored LXD will also support Intel's new Software Guard Extensions (SGX).[459, 460]

The following statement from the team further illustrates the LXD roadmap:

> **"We're working with silicon companies to ensure hardware-assisted security and isolation for these containers, just like virtual machines today. We'll ensure that the kernel security cross-section for individual containers can be tightened up for each specific workload. We will make sure you can live-migrate these containers from machine to machine. And we're adding the ability to bind storage and network interfaces to the containers, just like virtual machines."**
> - Is LXD a real Linux hypervisor? by the LXD team

It's important to note, LXD is not yet an actual hypervisor, it is currently implemented in Golang and provides a REST API, again very similar to Docker. The LXD live migration, possibly a key differentiator[461] has been successfully demonstrated.[462] See Where does LXD fit in and Where are we going with LXD for more information.

[457] https://www.joyent.com/blog/triton-docker-and-the-best-of-all-worlds
[458] https://blogs.intel.com/evangelists/2015/05/19/fostering-new-data-center-usages-with-clear-containers/
[459] https://twitter.com/grsecurity/status/530490056893825024
[460] https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2015/january/intel-software-guard-extensions-sgx-a-researchers-primer/
[461] http://tycho.ws/blog/2015/04/lxd-live-migration.html
[462] https://www.youtube.com/watch?v=a9T2gcnQg2k&t=1189

11.7.3 Siemens Jailhouse

Jailhouse is a partitioning Hypervisor "based on Linux". This solution apparently can run bare-metal applications or as a sort of paravirtualization layer via "adapted operating systems besides Linux". Jailhouse is said to focus on simplicity, rather than feature richness and offers support for x86, x86-64 and ARM processors. The main project also states the "aim of Jailhouse is to keep the amount of code responsible for establishing and maintaining cell isolation as small as possible".[463] Overall this is new and interesting project which is still under very active development. The core ideas behind Jailhouse are explored within the 2013 presentation Static System Partitioning and KVM by project lead Jan Kiszka.

11.7.4 Full Unikernels: App VMs on Steroids

The next step beyond container-focused hypervisor platforms is fully isolated unikernels deployed alongside an immutable architecture or infrastructure.[464] This fundamental shift towards single-purpose appliances and single-application containers begs the question, can micro containers simply run within their own hardware backed hypervisor? This basic idea is implemented in OSv and MirageOS and is explored in the ACM article Unikernels: Rise of the Virtual Library Operating System. For specific applications, such as networking middleware (firewalls, software routers, monitoring devices, etc) ClickOS is yet another example for performance focused unikernels.[465] Finally, Xen, a longtime platform of choice for testing new virtualization platforms is itself leading the charge: this time on exploring the idea of "Cloud Operating Systems".[466]

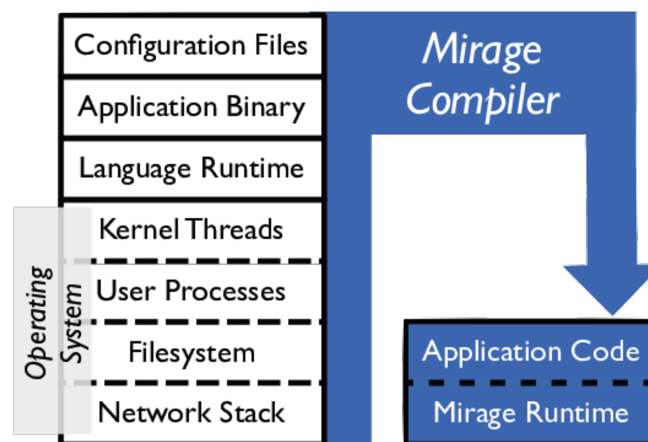


Figure 2: A high-level meta-example of reducing down an entire stack into a Mirage runtime and the application code in order to run the minimal unikernel image inside of a traditional hypervisor.

Full unikernels offer simplicity benefits, hardware-backed isolation and extreme levels of attack surface reduction[467] but, overall, they are not a well known solution for most devops or IT shops. This lack of support and general understanding can lead to misimplementations that introduce other potential security risks. Several arguments can also be made that they are still unfit for production[468] even if they can be deployed and manged successfully. Another key concern is large mounts of security-sensitive code may need to be re-implemented in unikernel specific libraries. To use "Rump kernels" as an example[469] this includes

[463] https://lwn.net/Articles/574273/
[464] https://medium.com/@darrenrush/after-docker-unikernels-and-immutable-infrastructure-93d5a91c849e
[465] https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-martins.pdf
[466] http://www.linux.com/news/enterprise/cloud-computing/751156-are-cloud-operating-systems-the-next-big-thing-
[467] https://twitter.com/dinodaizovi/status/702209109567733760
[468] https://www.joyent.com/blog/unikernels-are-unfit-for-production
[469] http://rumpkernel.org/

TCP/IP stacks, filesystem drivers, random number generators, cryptographic primitives and other areas typically provided by well vetted Operating System implementations. Overall these reimplementations will likely introduce new vulnerabilities or risks and other security regressions. Unikernel applications based on provably secure micro kernels[470] are left as an exercise to the reader.

At Dockercon EU, in the winter of 2015, Anil Madhavapeddy a lead developer for MirageOS presented Unikernels meet Docker. This demo involved developing Docker into a unikernel microservice, such that different applications were deployed atop hardware-virtualization backed unikernels on KVM. For more information see Contain Your Unikernels! and the code repository for the demo.[471] For more performance-specific information on Unikernels and several related implementations such as OSv, see the Linux Foundation presentation A survey of high performance virtualization techniques by Mike Day of IBM.

11.8 The Big Idea Of Microservices

The microservices model is much more than an underground devops movement to place a REST API on every single application. From a security perspective, microservices can encourage a logical breakdown of application components, isolate high-risk or sensitive services and help establish an overall architecture of least access and least privilege.[472] The Nginx blog article Introduction to Microservices offers a concise overview of many advantages and disadvantages. In short, microservices offer a model of reducing traditionally large applications into discrete components, just as classes or namespaces within an application isolate features, microservices can separate authentication, authorization, data processing, and logging. Microservices also break-down different application features and disparate functions into specific APIs or services. Payment services can be well isolated from email notifications code, front-end minimal token caching systems restricted from back-end databases. Each application "micro service" can now also be scaled, upgraded, fault tolerant, distributed and (most importantly) secured separately.

Microservices can operate similar to secure networks and modern application sandboxes, translating IPC calls or inner-OS API calls into cross-container or cross-host API calls. This embraces networking and authentication based application segmentation based on trust, data handling and required access. Within application sandboxes (such as the Google Chrome browser), broker processes and isolated services are based on risk, trust, attack surfaces and a combination of related security requirements. This can offer a very robust design with defense-in-depth monitoring, auditing and control where traditionally it was difficult to achieve or not chiefly encouraged. Security and isolation can now be part of the design, development or deployment process. To put it simply, the line diagram drawn on whiteboards can now be logically deployed, with security enforced at the application and network level rather than a much less defined, isolated or restricted, as is often the case.[473]

Logically isolating application components into different services can also help a system adapt to performance requirements at scale, which may be yet another reason to consider a microservices architecture. Breaking down different application requirements such as authentication, creating database cursors, or performing token generation is logical. Isolating areas of the application which convert between different binary formats, parse untrusted data or other high-risk operations helps the application, logging and related container hardening move towards specificity vs generality. One example could be web application with an image processing component. This can be split into separate, highly locked down and minimal containers connected via a message bus or API. The web application UI container can communicate with a back-end web API container, in turn, this container can control and log access to, as well as authenticate service-specific

[470] https://sel4.systems/
[471] https://github.com/Unikernel-Systems/DockerConEU2015-demo/
[472] https://www.gartner.com/doc/2974417
[473] https://pbs.twimg.com/media/CY3NsQjUwAE6Afz.png:large

connections. Finally, potentially dangerous image verification or format conversion then be performed within another separate and tightly restricted or resource limited container.

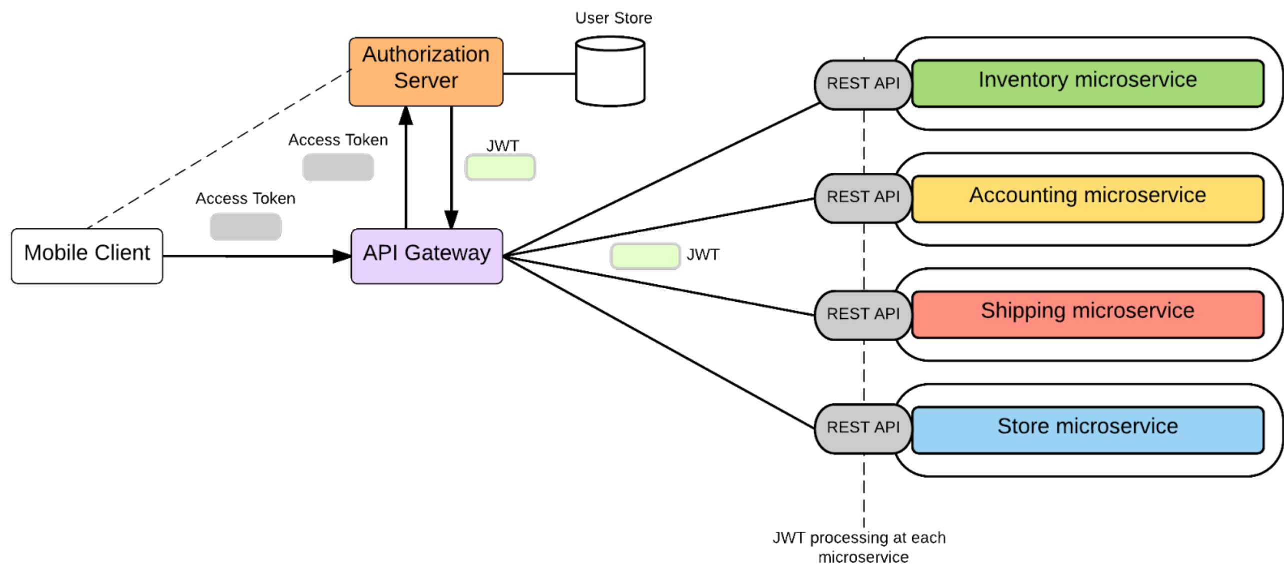


Figure 3: One potential Microservices model, using JSON Web Tokens (JWT) for authentication. Image from Microservices in Practice: From Architecture to Deployment.

Container orchestration is often discussed when considering moving an existing platform to containers, using a large number of containers, or migrating to a Microservices model. Orchestration is also a major difficulty point of many deployments. This is not only due to different requirements, interoperability, configuration and competing platforms, but the complexity and overall immaturity of many of these systems. Orchestration systems or service discovery either via Docker Swarm, CoreOS Fleet, Apache Mesos/Aurora or a myriad of other newcomers have a high impact if they contain various security risks. However, these risks may prove less serious than introducing inconsistency or a general lack of configuration management, as often present without orchestration. Inconsistency within application deployments often leads to security gaps, either through misconfiguration or lack of patching, as NCC Group commonly witness during various types of security assessments.

Overall, containers and microservices were destined to support each-other. As applications moved from large monolithic untamed beasts to service oriented and API decorated designs, the logical progression is to further isolate and reduce service size. This fits perfectly with distilling applications into their required libraries and Operating System needs. Potential concerns related to rearchitecture costs, development time and other requirements may dictate how, when and if an organization will switch to microservices. In any case, the microservices model should be considered for any deployment or platform rearchitecture, as it offers a number of benefits including performance, scale, security.

12 The End

12.1 Conclusion

As I hope that this whitepaper has made clear, effort is still required for containers to be secure. However, this is largely due to the relative immaturity of implementations, either the container systems themselves or kernel features which support containers, and the nature of OS virtualization itself. Recent advances such as Linux Clear Containers or Siemens Jailhouse aim to create a hybrid approach of application VMs alongside hardware virtualization. This may offer an excellent solution for high security applications and improved container trust, at the same time retaining packaging, flexibility and efficiency.

In this modern age, I believe that there is little excuse for not running a Linux application in some form of a Linux container, MAC or lightweight sandbox. Even if readers of this paper don't take the efforts described to further harden container systems, raising the bar with default container or sandbox deployments can be a simple exercise that greatly increases attacker time and effort. Containers can also provide an organization the ability to have minimal and consistent application deployments, without the need for a heavy handed configuration management system such as Puppet or Chef (for applications as well as system administration).

Containers can't provide perfect security; as rational people we know this is impossible for any system of moderate complexity and usability. Security through Linux containers is all about making attackers simply work harder through least privilege, least access, attack surface reduction, and simplicity through basic (yet often logical) process restrictions. As with any risk consideration, all of the eggs shouldn't be in one basket. Software virtualization places 100% of the security into the container system; as such, implementations should endeavor to prevent a single security failure from compromising an entire data center, cloud deployment or even simply the container host.

Are Linux containers the future? I think they're one piece of the puzzle that helps both application developers and security teams. As with any technology stack, security vigilance and an ongoing evaluation process is required. Key container features and kernel namespaces are still missing or in progress and overall have been added-on rather than built-in. Does this mean we should abandon containers and stick with full virtual machines? No, but it takes some additional effort to secure containers. Fortunately, the majority of these security efforts align to existing best or recommended practices, hopefully making the integration easy with existing programs. If security is the utmost concern, above all else, then containers are not what should be used. For those looking to balance the scales, use physical isolation, hardware virtualization and containers along well-audited trust boundaries.

> **''We can only see a short distance ahead, but we can see plenty that needs to be done.''**
>
> - Alan Turing

12.2 Acknowledgements

I would like to thank NCC Group coworkers and peer reviewers Jesse Hertz, Jon Barber, Raphael Salas, Mark Manning, Jake Heath, Max Burkhardt. Nathan McCauley of the Docker security team also deserves thanks, as do close friends Josh ''hex'' Dukes, and Kyle ''greenfly'' Rankin, all of who commented on a pre-released version of this document. As I work in this space, I also try to keep in mind we're standing on the shoulders of giants. I would like to thank hardworking Linux kernel or container developers, fellow security researchers, the Google Chrome team, the Grsecurity team, original Plan9 developers, and open source security efforts everywhere for continuing to advance, and break, the state of the art.

12.3 About The Author

Aaron (@dyn___) Grattafiori is a Technical Director and research lead at NCC Group. With over ten years of experience, he regularly performs application security engagements, Linux and container assessments, network penetration tests and many other tests for a large variety of NCC Group clients. Aaron also stays busy with research on a number of topics involving Linux security, automotive systems, network protocols, fuzzing and privacy or liberation technologies. Aaron has spoken on a number of security topics at both regional and national security conferences such as Blackhat, DEFCON, and Toorcon.