Void Processes: minimising privilege by default

Jake Hillion
Queens' College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for Part III of the Computer Science Tripos.

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: Jake.Hillion@cl.cam.ac.uk

May 11, 2022

Declaration

I, Jake Hillion of Queens' College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 5

Signed:

Date: May 11, 2022

Abstract

Write a summary of the whole thing. Make sure it fits in one page.

Contents

1 Introduction **1**

2 Privilege Separation **3**
- 2.1 Dropping Privileges 4
- 2.2 Object Capabilities 6
- 2.3 Namespaces 6

3 Entering the Void **9**
- 3.1 ipc namespaces 10
- 3.2 uts namespaces 10
- 3.3 time namespaces 11
- 3.4 network namespaces 11
- 3.5 pid namespaces 12
- 3.6 mount namespaces 14
- 3.7 user namespaces 20
- 3.8 cgroup namespaces 21

4 Filling the Void **23**
- 4.1 mount namespace 23
- 4.2 network namespace 24
- 4.3 user namespace 25
- 4.4 Remaining namespaces 27

5 System Design **29**

6 Building Applications **31**
- 6.1 No Permissions 31
- 6.2 gzip 31
- 6.3 TLS Server 32

7 Evaluation **35**

8 Related Work **37**
8.1 Virtual Machines and Containers 37
8.2 systemd 38

9 Future Work **39**
9.1 Dynamic Linking 39

10 Conclusion **41**

List of Figures

1.1 Privilege separated environments plotted from least to most like bare metal Linux. 2

2.1 Data flow with the two processes in OpenBSD's privilege separated syslogd design. 5

3.1 Creating a virtual Ethernet pair between the root network namespace and a newly created network namespace. 12
3.2 Highly separated behaviour without shared subtrees between mount namespaces. 16
3.3 Behaviour when attempting to unmount / from an unshared shell with a shared mount. 18

5.1 Interaction between the application and the environment. . . . 30

6.1 Process separation in a TLS server. 32

List of Tables

1.1 Table showing the date and kernel version each namespace was added. The date provides the date of the first commit where they appeared, and the kernel version the kernel release they appear in the changelog of. Namespaces are ordered by kernel version then alphabetically. Some examples are provided of CVEs for each namespace. 2

Chapter 1

Introduction

Void Processes allow running purpose-built applications without all of the features that a full Linux system makes available, and encourage privilege separation by default. This is achieved using a mixture of Linux namespaces and file descriptor based capabilities. During the process of building the system gaps in the kernel were exposed - namespaces were intended to emulate an ordinary Linux system rather than build something new. This work will go on to detail the mechanisms for creating Void Processes themselves, re-adding features that these processes need to do useful work, and describe which features are missing in the user-space kernel APIs to successfully create processes this way.

The question of what makes an operating system has been asked many times. There have previously been many attempts to redefine an operating system. Here we compare this work with two of those: unikernels and containers. Unikernels abandon the monolithic kernel in favour of a slimmed down kernel that only provides the features the user needs, limiting the trusted computing base but requiring special purpose applications to be written. Containers provide a view of an isolated system while sharing a monolithic kernel with the host, allowing almost any application that can run on Linux to run in a Linux Container, but including all of the features and security holes that come with running a monolithic kernel. Void Processes lie between the

Table 1.1: Table showing the date and kernel version each namespace was added. The date provides the date of the first commit where they appeared, and the kernel version the kernel release they appear in the changelog of. Namespaces are ordered by kernel version then alphabetically. Some examples are provided of CVEs for each namespace.

Namespace	Date	Kernel Ver.	CVEs
`mount`	Feb 2001 [1]	2.5.2 [2]	2020-29373
`ipc`	Oct 2006 [3]	2.6.19 [4]	
`uts`	Oct 2006 [5]	2.6.19 [4]	
`user`	Jul 2007 [6]	2.6.23 [7]	2021-21284
`network`	Oct 2007 [8]	2.6.24 [9]	2011-2189
`pid`	Oct 2006 [10]	2.6.24 [9]	2019-20794
`cgroup`	Mar 2016 [11]	4.6 [12]	2022-0492
`time`	Nov 2019 [13]	5.6 [14]	

two. While they still rely on the monolithic kernel for isolation and inter-process communication, further reliance on the kernel is limited as much as possible. While much of the Linux experience is made unavailable the core calls remain the same, such as operations on file descriptors. By having nothing available at all by default, an environment where every privilege required must be explicitly added is created. When combined with inter-process communication, a feature not as ingrained in unikernels, high levels of privilege separation are achieved. These methods are plotted in Figure 1.1.

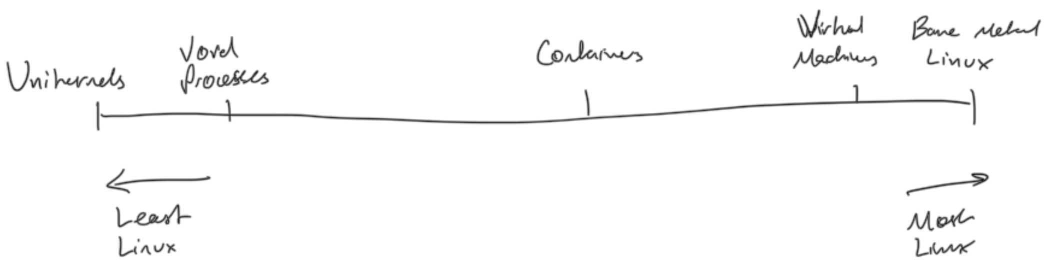

Figure 1.1: Privilege separated environments plotted from least to most like bare metal Linux.

Chapter 2

Privilege Separation

Privilege separation became a necessity as systems became more complex. Many attack vectors existed in software, notably in argument processing and deserialisation [15, 16]. This made it clear that large applications didn't exhibit enough decomposition when it came to their privileges. Creating security conscious applications would require one of two things: creating applications without security bugs, or separating the parts of the application with the potential to cause damage from the parts most likely to contain bugs. Though many efforts have been made to create correct applications [CN], the use of such technology is far from widespread and security related bugs in applications are still frequent [CN]. Rather than attempting to avoid bugs, the commonly employed solution is privilege separation: ensuring that the privileged portion of the application is correct and separated from the portion which is likely to be attacked.

The basic unit of privilege separation on Unix is a process. If it's possible for an attacker to gain remote code execution in a process, the attacker gains access to all of the process's privilege. Reducing the privilege of a process therefore reduces the attack surface available. The first solution to privilege separating an application on UNIX is to place elements into different processes. One can separate, for example, the process of a TLS server which has access to the private keys from the part of the server which processes user input. What becomes obvious in this technique is the programmer overhead of handling the newly found inter-process communication in the system. Application design in this paradigm is similar to that of a distributed system, where many asynchronous systems must interact.

OpenBSD is a UNIX operating system with an emphasis on security. In 2003, privilege separation was added to the `syslogd` daemon of OpenBSD [17]. The system is designed with a parent process that retains privilege and a network accepting child process that goes through a series of states, dropping privilege with each state change. This pattern allowed for restarting of the service while keeping the section which processed user data strongly separated from the process which remains privileged, by enabling the child process to cause its own restart while not holding enough privilege to execute that restart itself. An overview of the data flow is provided in Figure 2.1. This section goes on to provide an overview of privilege separation techniques in modern Unices, many of which are included in some form in the final design for Void Processes.

2.1 Dropping Privileges

It is possible for many applications to utilise a single process which reduces its level of privilege as the application makes progress. The approach is commonly to begin with high privilege for opening, for example, a listening socket below port 1000. After this has been completed, the ability to do so is dropped. One of the earliest ways to do this is to change user using `setuid(2)` after the privileged requirements are complete. An API such as OpenBSD's `pledge(2)` allows only a pre-specified set of system calls after the call to `pledge(2)`. A final alternative is to drop explicit capabilities on Linux. Each of these solutions irreversibly reduce the privilege of the application. This is known as dropping privilege.

After dropping privilege, it becomes difficult to do things such as reloading the configuration. The application process no longer have the required privilege to restart the application, and if you could gain it back then dropping

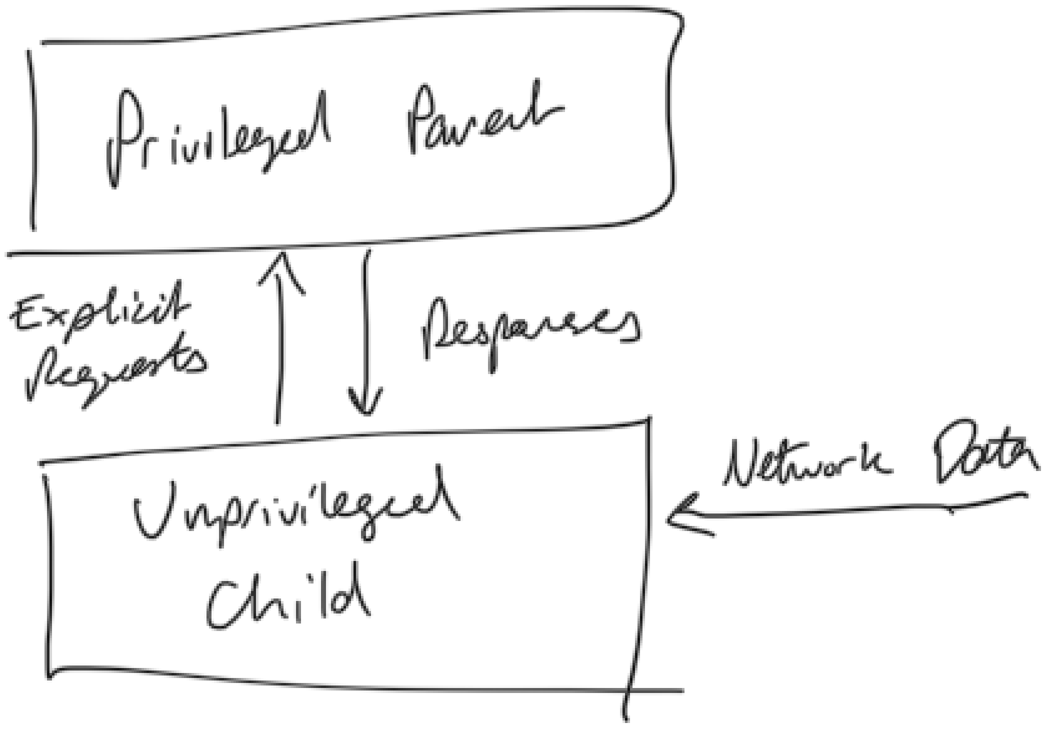

Figure 2.1: Data flow with the two processes in OpenBSD's privilege separated syslogd design.

it would have had no effect. Further, as an application becomes more like a networked system, serialisation and deserialisation becomes a common occurrence. As deserialisation is a very common source of exploits [16], this adds the potential for new flaws in the application.

2.2 Object Capabilities

Object capabilities attempt to enable least privilege operation by providing more tangible tokens of authority to applications. Capabilities are described as unforgeable tokens of authority, providing an application with an object and a set of rights on it. For example, a file descriptor and the right to read from it. Capsicum [18] adds object capabilities to FreeBSD. These capabilities may be shared between processes as with file descriptors. Capability mode reduces a process's privilege to operating only on capabilities, and not creating new ones.

2.3 Namespaces

The Linux approach to increased process separation is namespaces. Namespaces alter the view of the world that a process sees. Processes remain the primary method of separation, but can increase how separated the processes are from both each other and the underlying host system. The intended and most common use case of namespaces is providing containers. Containers approximate virtual machines, providing the appearance of running on an isolated system while sharing the same host. Containers, however, have to implement privilege separation in a very different way to the privilege of separation we've seen previously. Rather than spawning multiple processes and employing privilege separation techniques to limit the attack vector in each, one spawns multiple containers to form a more literal distributed system instead. It is common to see, for example, a web server and the database that backs it deployed as two separate containers. These separate containers interact entirely over the network. This means that if a user achieves remote code execution of the database, it does not extend to the web server. The design pattern of building multiple processes which exist independently and communicate externally is called microservices. This presents an interesting paradigm of small applications which can and often do run on separate physical hosts combining to provide a unified application experience.

This work focuses on the application of namespaces to more conventional privilege separation. Working with a shim which orchestrates the process and namespace layout, Void Applications seek to provide a completely pruned minimal Linux experience to each Void Process within the application. This builds on much of the prior work to severely limit the access of processes in the application as much as possible, reducing the impact of a vulnerability in each part to the fullest. There is never a need to drop privileges as applications are created with the absolute minimum necessary to perform correctly. In Section 3 we discuss each namespace's role in Linux and how to create one which is empty, before explaining in Section 4 how to reinsert just enough Linux for each process in an application to be able to complete useful work.

Chapter 3

Entering the Void

Isolating parts of a Linux system from the view of certain processes is achieved by using namespaces. Namespaces are commonly used to provide isolation in the context of containers, which provide the appearance of an isolated complete Linux environment to contained processes. Instead, with Void Processes, we target complete isolation. Rather than using namespaces to provide a view of an alternate full Linux system, they are used to provide a view of a system that is as minimal as possible, while still sitting atop the Linux kernel. In this section each namespace available in Linux is detailed, including how to take a fresh namespace of each kind and completely empty it for a Void Process. Section 4 goes on to explain how necessary features for applications are added back in.

The full set of namespaces are represented in Table 1.1, in chronological order. The chronology of these is important in understanding the thought process behind some of the design decisions. The ease of creating an empty namespace varies massively, as although adding namespaces shared the goal of containerisation, they were completed by many different teams of people over a number of years. Some namespaces maintain strong connections to their parent, while others are created completely separate. We start with those that are most trivial to add, working up to the namespaces most intensely linked to their parents.

3.1 ipc namespaces

IPC namespaces isolate two mechanisms that Linux provides for IPC which aren't controlled by the filesystem. System V IPC and POSIX message queues are each accessed in a global namespace of keys. This has created issues in the past with attempting to run multiple instances of PostgreSQL on a single machine, as both instances tried to create a System V IPC entry with the same key [CN]. IPC namespaces solve this effectively for containers by creating a new scoped namespace. Processes are a member of one and only one IPC namespace, allowing the familiar global key APIs. IPC namespaces are optimal for creating Void Processes. From the manual page [19]:

"Objects created in an IPC namespace are visible to all other processes that are members of that namespace, but are not visible to processes in other IPC namespaces."

This provides exactly the correct semantics for a Void Process. IPC objects are visible within a namespace if and only if they are created within that namespace. Therefore, a new namespace is entirely empty, and no more work need be done.

3.2 uts namespaces

UTS namespaces provide isolation of the hostname and domain name of a system between processes. Similarly to IPC namespaces, all processes in the same namespace see the same results for each of these values. This is useful when creating containers. If unable to hide the hostname, each container would look like the same machine. Unlike IPC namespaces, UTS namespaces are copy-on-write. Each of these values in the child is initialised as the same as the parent.

As the copied value does give information about the world outside of the Void Process, slightly more must be done than placing the process in a new namespace. Fortunately this is easy for UTS namespaces, as the host name and domain name can be set to constants, removing any link to the parent.

3.3 time namespaces

Time namespaces are the final namespace added at the time of writing, added in kernel version 5.6 [14]. The motivation for adding time namespaces is given in the manual page [20]:

"The motivation for adding time namespaces was to allow the monotonic and boot-time clocks to maintain consistent values during container migration and checkpoint/restore."

That is, time namespaces virtualise the appearance of system uptime to processes, rather than attempting to virtualise the wall clock time. This is important for processes that depend on time in primarily one situation: migration. If an uptime dependent process is migrated from a machine that has been up for a week to a machine that was booted a minute ago, the guarantees provided by the clocks `CLOCK_MONOTONIC` and `CLOCK_BOOTTIME` no longer hold. This results in time namespaces having very limited usefulness in a system that does not support migration, such as the one presented here. Perhaps randomised offsets would hide some information about the system, but the usefulness is limited. Time namespaces are thus avoided in this implementation.

3.4 network namespaces

Similarly to IPC, they present the optimal namespace for running a Void Process. Creating a new network namespace immediately creates a namespace containing only a local loopback adapter. This means that the new network namespace has no link whatsoever to the creating network namespace, only supporting internal communication. To add a link, one can create a virtual Ethernet pair with one adapter in each namespace (see Figure 3.1). Alternatively, one can create a Wireguard adapter with sending and receiving sockets in one namespace and the VPN adapter in another [21, §7.3]. These methods allow for very high levels of separation while still maintaining access to the primary resource - the Internet or wider network.

```
#
#
# ip link add veth0 type veth peer veth1
# ip link set veth1 netns test
# ip addr add 192.168.0.1/24 dev veth0
# ip link set up dev veth0
# ping -c 1 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time=0.317 ms
```

```
# unshare -n
# ip netns attach test $$
#
#
# ip addr add 192.168.0.2/24 dev veth
# ip link set up dev veth1
# ping -c 1 192.168.0.1
PING 192.168.0.1 (192.168.0.1) 56(84)
64 bytes from 192.168.0.1: icmp_seq=1
```

Figure 3.1: Creating a virtual Ethernet pair between the root network namespace and a newly created network namespace.

3.5 pid namespaces

PID namespaces create a mapping from the process IDs inside the namespace to process IDs in the parent namespace. This continues until processes reach the top-level PID namespace. This isolation behaviour is different to that of some other namespaces, as each process within the namespace represents a process in the parent namespace too, albeit with different identifiers.

Although PID namespaces work quite well for creating a Void Process from the perspective of the inside process, some care must be taken in the implementation, as the actions of PID namespaces are highly affected by others. Some examples of this slightly unusual behaviour are shown in Listing 3.1.

The first behaviour shown is that an `unshare(CLONE_PID)` call followed immediately by an `exec` does not have the desired behaviour. The reason for this is that the first process created in the new namespace is given PID 1 and acts as an init process. That is, whichever process the shell spawns first becomes the init process of the namespace, and when that process dies, the namespace can no longer create new processes. This behaviour is avoided by either calling `unshare(2)` followed by `fork(2)`, or utilising `clone(2)` instead. The `unshare(1)` binary provides a fork flag to solve this, while the implementation of the Void Orchestrator uses `clone(2)` which has the semantics of combining the two into a single syscall.

Listing 3.1: Unshare behaviour with PID namespaces, with and without forking and remounting proc.

```
$ unshare -p
-bash: fork: Cannot allocate memory
# (new shell in new pid namespace)
# ps ax | tail -n 3
-bash: fork: Cannot allocate memory

$ unshare --fork -p
# (new shell in new pid namespace)
# ps ax | tail -n 3
2645 ?       I     0:00 [kworker/...]
2689 pts/1   R+    0:00 ps ax
2690 pts/1   S+    0:00 tail -n 2

$ unshare --fork --mount-proc -p
# (new shell in new pid namespace)
# ps ax | tail -n 3
 1 pts/1   S     0:00 -bash
15 pts/1   R+    0:00 ps ax
16 pts/1   S+    0:00 tail -n 3
```

Secondly, we see that even in a shell that appears to be working correctly, processes from outside of the new PID namespace are still visible. This behaviour occurs because the mount of `/proc` visible to the process in the new PID namespace is the same as the init process. This is solved by remounting `/proc`, available to `unshare(3)` with the `---mount-proc` flag. Care must be taken that this mount is completed in a new mount namespace, or else processes outside of the PID namespace will be affected. The Void Orchestrator again avoids this by voiding the mount namespace entirely, so any access to proc must be either bound to outside the namespace deliberately or freshly mounted.

3.6 mount namespaces

Mount namespaces were by far the most challenging part of this project. When adding new features, they continuously raised problems in both API description, expected behaviour, and availability of tools in user-space. A comparison will be given in this section to two other namespaces, network and UTS, to show the significant differences in the design goals of mount namespaces. Many of the implementation problems here comes from a fundamental lack of consistency between mount namespaces and other namespaces in Linux.

3.6.1 Copy-on-Write

Comparing to network namespaces, we see a huge difference in what occurs when a new namespace is created. When creating a new network namespace, the ideal conditions for a Void Process are created - a network namespace containing only a loopback adapter. That is, the process has no ability to interact with the outside network, and no immediate relation to the parent network namespace. To interact with alternate namespaces, one must explicitly create a connection between the two, or move a physical adapter into the new (empty) namespace. Mount namespaces, rather than creating a new and empty namespace, made the choice to create a copy of the parent namespace, in a copy-on-write fashion. That is, after creating a new mount namespace, the mount hierarchy appears much the same as before. This is shown in Listing 3.2, where the file `/etc/passwd` is shown before and after an unshare, revealing the same content.

3.6.2 Shared Subtrees

While some other namespaces are copy-on-write, for example UTS namespaces, they do not present the same problem as mount namespaces. Although UTS namespaces are copy-on-write, it is trivial to create the conditions for a Void Process by setting the hostname of the machine to a constant. This removes any relation to the parent namespace and to the outside machine.

Listing 3.2: Reading the same file before and after unsharing the mount namespace.

```
int main() {
int fd;

if ((fd = open("/etc/passwd", O_RDONLY)) < 0)
     perror("open");
print_file(fd);
if (close(fd))
     perror("close");

if (unshare(CLONE_NEWNS))
     perror("unshare");
printf("----- unshared -----\n");

if ((fd = open("/etc/passwd", O_RDONLY)) < 0)
     perror("open");
print_file(fd);
if (close(fd))
     perror("close");
}
```

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
...
----- unshared -----
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
...
```

```
# unshare -m
# mount_container_root /tmp/a
# mount --bind \
    /mnt/cdrom /tmp/a/mnt/cdrom
# pivot_root /tmp/a /tmp/a/oldroot
# umount /tmp/a/oldroot
#
# ls /mnt/cdrom
```

```
#
#
#
#
#
#
# mount /dev/sr0 /mnt/cdrom
# ls /mnt/cdrom
file_1  file_2
```

Figure 3.2: Highly separated behaviour without shared subtrees between mount namespaces.

Mount namespaces instead maintain a shared pointer with most filesystems, more akin to not creating a new namespace than a copy-on-write namespace.

Shared subtrees [22] were introduced to provide a consistent view of the unified hierarchy between namespaces. Consider the example in Figure 3.2. `unshare(1)` creates a non-shared tree, which presents the behaviour shown. Although `/mnt/cdrom` from the parent namespace has been bind mounted in the new namespace, the content of `/mnt/cdrom` is not the same. This is because the filesystem newly mounted on `/mnt/cdrom` is unavailable in the separate mount namespace. To combat this, shared subtrees were introduced. That is, as long as `/mnt/cdrom` resides on a shared subtree, the newly mounted filesystem will be available to a bind of `/mnt/cdrom` in another namespace. `systemd` made the choice to mount `/` as a shared subtree [23]:

"Notwithstanding the fact that the default propagation type for new mount is in many cases `MS_PRIVATE`, `MS_SHARED` is typically more useful. For this reason, `systemd(1)` automatically remounts all mounts as `MS_SHARED` on system startup. Thus, on most modern systems, the default propagation type is in practice `MS_SHARED`."

This means that when creating a new namespace, mounts and unmounts are propagated by default. More specifically, it means that mounts and unmounts are propagated both from the parent namespace to the child, and from the child namespace to the parent. This can be highly confusing behaviour, as it provides minimal isolation by default. `unshare(1)` considers this behaviour inconsistent with the goals of unsharing - it immediately calls `mount("none", "/", NULL, MS_REC|MS_PRIVATE, NULL)` after `unshare(CLONE_NEWNS)`, detaching the newly unshared tree. The reasoning for enabling `MS_SHARED` by default is that containers created should not present the behaviour given in Figure 3.2, and this behaviour is unavoidable unless the parent mounts are shared, while it is possible to disable the behaviour where necessary.

3.6.3 Lazy unmounting

Mount namespaces present further interesting behaviour when unmounting the old root filesystem. Although this may initially seem isolated to Void Processes, it is also a problem in a container system. Consider again the container created in Figure 3.2: the existing root must be unmounted after pivoting, else the container remains fully connected to the outside root.

Referring again to network namespaces, sockets continue to exist in their initial namespace, allowing for regular file-descriptor passing semantics [24]. Extending upon this socket behaviour is Wireguard, which creates adapters that may be freely moved between namespaces while continuing to connect externally from their initial parent [21, §7.3].

Something which behaves differently is the memory mapping of a currently running process's binary. Consider the example in Listing 3.3, which shows a short C program and the result of running it. It is seen that the `/` mount is busy when attempting the unmount. Given that the process was created in the parent namespace, the behaviour of file descriptors would suggest that the process would maintain a link to the parent namespace for its own memory mapped regions. However, the fact that the otherwise empty namespace has a busy mount shows that this is not the case.

A feature called lazy unmounting or `MNT_DETACH` exists for situations where a busy mount still needs to be unmounted. Supplying the `MNT_DETACH` flag

Listing 3.3: Behaviour when attempting to unmount / after an unshare.

```
int main() {
if (unshare(CLONE_NEWNS))
        perror("unshare");
if (mount("none", "/", NULL,
  MS_REC|MS_PRIVATE, NULL))
        perror("mount");
if (umount("/"))
        perror("umount");
}
--
umount: Device or resource busy
```

```
# cat /proc/mounts | grep udev
udev /dev devtmpfs rw,nosuid,relati...
#
#
# cat /proc/mounts | grep udev
cat: /proc/mounts: No such file or...
```

```
#
#
# unshare --propagation unchanged -m
# umount -l /
#
#
```

Figure 3.3: Behaviour when attempting to unmount / from an unshared shell with a shared mount.

to `umount2(2)` causes the mount to be immediately detached from the unified hierarchy, while remaining mounted internally until the last user has finished with it. Whilst this initially seems like a good solution, this syscall is incredibly dangerous when combined with shared subtrees. This behaviour is shown in Figure 3.3, where a lazy (and hence recursive) unmount is combined with a shared subtree to disastrous effect.

This behaviour raises questions about why a shared subtree, which exists as an object, would need to be detached recursively - decreasing the reference count to the shared subtree itself would seem sufficient. The inconsistency is best explained by looking at the development timeline for the three features here: mount namespaces, shared subtrees, and recursive lazy unmounts. When lazy unmounting was added, in September 2001, the author said the following (sic) [25]:

“There are only two things to take care of - a) if we detach a parent we should do it for all children b) we should not mount anything on ”floating” vfsmounts. Both are obviously staisfied for current code (presence of children means that vfsmount is busy and we can’t mount on something that doesn’t exist).”

This logic held even in the presence of namespaces, with the initial patchset in February 2001 [25], as mounts were not initially shared but duplicated between namespaces. However, when shared subtrees were added in January 2005 [26], this logic stopped holding.

When setting up a container environment, one calls `pivot_root(2)` to replace the old root with a new root for the container. Then, the old root may be unmounted. Oftentimes the solution is to exec a binary in the new root first, meaning that the old root is no longer in use and may be unmounted. This works, as old root is only a reference in this namespace, and hence may be unmounted with children - the `vfsmount` in this namespace is not busy, contradicting an assertion in the quotation.

If, instead, one wishes to continue running the existing binary, this is possible with lazy unmounting. However, the kernel only exposes a recursive lazy unmount. With shared subtrees, this results in destroying the parent tree. While this is avoidable by removing the shared propagation from the subtree before unmounting, the choice to have `MNT_DETACH` aggressively cross shared subtrees can be highly confusing, and perhaps undesired behaviour in a world with shared subtrees by default.

The API is particularly unfriendly to creating a Void Process. The creation of mount namespaces is copy-on-write, and many filesystems are mounted shared. This means that they propagate changes back through namespace boundaries. As the mount namespace does not allow for creating an entirely empty root, extra care must be taken in separating processes. The method taken in this system is mounting a new `tmpfs` file system in a new namespace, which doesn’t propagate to the parent, and using the `pivot_root(8)` command to make this the new root. By pivoting to the `tmpfs`, the old root exists as the only reference in the otherwise empty `tmpfs`. Finally, after ensuring the old root is set to `MNT_PRIVATE` to avoid propagation, the old root can be lazily detached. This allows the binary from the parent namespace, the shim in this case, to continue running correctly. Any new processes only have access to the materials in the empty `tmpfs`. This new `tmpfs` never appears in the parent namespace, separating the Void Process effectively from the parent namespace.

3.7 user namespaces

User namespaces provide isolation of security between processes. They isolate uids, gids, the root directory, keys and capabilities. This provides massive utility for rootless containers [CN], and also this shim. Rather than the shim being a `setuid` or `CAP_SYS_ADMIN` binary, it can instead operate with ambient authority. This vastly simplifies the logic for opening file descriptors to pass the child processes, as the shim itself is already operating with correctly limited authority.

Similarly to many other namespaces, user namespaces suffer from needing to limit their isolation. For a user namespace to be useful, some relation needs to exist between processes in the user namespace and objects outside. That is, if a process in a user namespace shares a filesystem with a process in the parent namespace, there should be a way to share credentials. To achieve this with user namespaces a mapping between users in the namespace and users outside exists. The most common use-case is to map root in the user namespace to the creating user outside, meaning that a process with full privileges in the namespace will be constrained to the creating user's ambient authority.

To create an effective Void Process content must be written to the files `/proc/[pid]/uid_map` and `/proc/[pid]/gid_map`. In the case of the shim uid 0 and gid 0 are mapped to the creating user. This is done first such that the remaining stages in creating a Void Process can have root capabilities within the user namespace - this is not possible prior to writing to these files. Otherwise, `CLONE_NEWUSER` combines effectively with other namespace flags, ensuring that the user namespace is created first. This enables the other namespaces to be created without additional permissions.

3.8 cgroup namespaces

cgroup namespaces provide limited isolation of the cgroup hierarchy between processes. Rather than showing the full cgroups hierarchy, they instead show only the part of the hierarchy that the process was in on creation of the new cgroup namespace. Correctly creating a Void Process is hence as follows:

1. Create an empty cgroup leaf.
2. Move the new process to that leaf.
3. Unshare the cgroup namespace.

This process excludes the cgroup namespace from the initial `clone(3)` call, as the cloned process must be moved before creating the new namespace. By following this sequence of calls, the process in the void can only see the leaf which contains itself and nothing else, limiting access to the host system. This is the approach taken in this piece of work. This presents the one point where running the shim with ambient authority rather than high capabilities is potentially limiting. In order to move the process into a leaf the shim must have sufficient authority to modify the cgroup hierarchy. On systemd these processes will be launched underneath a user slice and will have sufficient permissions, but this may vary between systems. This leaves cgroups the most weakly implemented namespace at the moment.

Although good isolation of the host system from the Void Process is provided, the Void Process is in no way hidden from the host. There exists only one cgroups v2 hierarchy on a system (cgroups v1 are ignored for clarity), where resources are delegated through each. This means that all processes contained within the hierarchy must appear in the primary hierarchy, such that the distribution of the single set of system resources can be centrally controlled. This behaviour is similar to the aforementioned pid namespaces, where each process has a distinct PID in each of its parents, but does show up in each. Hiding from the host has little value as a root user there can inspect each namespace manually.

An alternative implementation that would make implementing with the cgroups namespace easier would be one that condenses all of the processes in the sea groups name space into one parent process in the parent main space. This would have the effect of hiding underlying processes from the parent name space, while still allowing control over the sea groups tree as a whole. It would further provide better isolation of the child, as a newly spawned cgroups space would show an empty route that only contains the child process. This would also allow more effective interaction with user namespaces, as the child namespace would only have control over itself, allowing for full control without risking the rest of the tree. This is opposed to the current limited view of the cgroups tree, which appears to have limited usefulness.

Chapter 4

Filling the Void

Once a set of namespaces to contain the Void Process have been created the goal is to reinsert enough to run the application, and nothing more. To allow for running applications as Void Processes with minimal kernel changes, this is done using a mixture of file-descriptor capabilities and adding elements to the namespaces. Capabilities allow for a clean experience where suitable, while adding elements to namespaces creates a more Linux-like experience for the application.

4.1 mount namespace

There are two options to provide access to files and directories in the void. Firstly, for a single file, an already open file descriptor can be offered. Consider the TLS broker of a TLS server with a persistent certificate and keyfile. Only these files are required to correctly run the application - no view of a filesystem is necessary. Providing an already opened file descriptor gives the process a capability to those files while requiring no concept of a filesystem, allowing that to remain a complete void. This is possible because of the semantics of file descriptor passing across namespaces - the file descriptor remains a capability, regardless of moving into a namespace without access to the file in question.

Alternatively, files and directories can be mounted in the Void Process's namespace. This supports three things which the capabilities do not: directories, dynamic linking, and applications which have not been adapted to use file descriptors. Firstly, the existing `openat(2)` calls are not suitable by default to treat directory file descriptors as capabilities, as they allow the search path to be absolute. This means that a process with a directory file descriptor in another namespace can access any files in that namespace [RN] by supplying an absolute path. Secondly, dynamic linking is best served by binding files, as these read only copies and the trusted binaries ensure that only the required libraries can be linked against. Finally, support for individual required files can be added by using file descriptors, but many applications will not trivially support it. Binding files allows for a form of backwards compatibility.

4.2 network namespace

Reintroducing networking to a Void Process follows a similar capability-based paradigm to reintroducing files. Rather than providing the full Linux networking view to a Void Process, it is instead handed a file descriptor that already has the requisite networking permissions. A capability for an inbound networking socket can be requested statically in the application's specification, which fits well with the earlier specified threat model. This socket remains open and allows the application to continuously accept requests, generating the appropriate socket for each request within the application itself, which can be dealt with through the mechanisms provided - specifically file descriptor based sockets.

Outbound networking is more difficult to re-add to a Void Process than inbound networking. The approach that containerisation solutions such as Docker take is using NAT with bridged adapters by default [RN]. That is, the container is provided an internal IP address that allows access to all networks via the host. Virtual machine solutions take a similar approach, creating bridged Ethernet adapters on the outside network or on a private

NAT by default. Each of these approaches give the container/machine the appearance of unbounded outbound access, relying on firewalls to limit this afterwards. This does not fit well with the ethos of creating a Void Process - minimum privilege by default. An ideal solution would provide precise network access to the void, rather than adding all access and restricting it in post. This is achieved with inbound sockets by providing the precise and already connected socket to an otherwise empty network namespace, which does not support creating inbound sockets of its own.

Consideration is given to providing outbound access in the same way as inbound - with statically created and passed sockets. For example, a socket to a database could be specified in the specification, or even one per worker process. The downside of this approach is that the socket lifecycle is still handled by the kernel. While this would work well with UDP sockets, TCP sockets can fail because the remote was closed or a break in the path caused a timeout to be hit.

Given that statically giving sockets is infeasible and adding a firewall does not fit well with creating a void, I sought an alternative API. `pledge(2)` is a system call from OpenBSD which restricts future system calls to an approved set [27]. This seems like a good fit, though operating outside of the operating system makes the implementation very different. Acceptable sockets can be specified in the application specification, then an interaction socket provided to request various pre-approved sockets from the shim layer. This allows limited access to the host network, approved or denied at request time instead of by a firewall. That is, access to a precisely configured socket can be injected to the void, with a capability to request such sockets and a capability given for the socket.

4.3 user namespace

Filling a user namespace is a slightly odd concept compared to the namespaces already discussed in this section. As stated in Section 3.7, a user namespace comes with no implicit mapping of users whatsoever. To enable

Listing 4.1: A directory listing before and after entering a user namespace with mapped root.

```
$ ls -ld repos owned_by_root
-rw-r--r-- 1 root  root      0 May
7 22:13 owned_by_root
drwxrwxr-x 7 jsh77 jsh77 4096 Feb 27 17:52 repos

$ unshare -U --map-root

# ls -ld repos owned_by_root
-rw-r--r-- 1 nobody nogroup     0 May
7 22:13 owned_by_root
drwxrwxr-x 7 root root 4096 Feb 27 17:52 repos
```

applications to be run with constrained authority, a single mapping is added by the Void Orchestrator of `root` in the child user namespace to the launching UID in the parent namespace. This means that the user with highest privilege in the container, `root`, will be limited to the access of the launching user. The behaviour of mapping `root` to the calling user is shown with the `unshare(1)` command in Listing 4.1, where a directory owned by the calling user, `jsh77`, appears to be owned by `root` in the new namespace. A file owned by `root` in the parent namespace appears to be owned by `nobody` in the child namespace, as no mapping exists for that file's user.

We can see then that the way user namespaces are currently used creates a binary system: either a file appears as owned by `root` if owned by the calling user, or appears as owned by `nobody` if not (ignoring groups for clarity, though their behaviour is similar). One questions whether more users could be mapped in, but this presents additional difficulties. Firstly, `setgroups(2)` system call must be denied to achieve correct behaviour in the child namespace. This is because the `root` user in the child namespace has full capabilities, which include `CAP_SETGID`. This means that the user in the namespace can drop their groups, potentially allowing access to materials which the creating user did not (consider a file with permissions 707). This limits the utility of switching user in the child namespace, as the groups must remain the same. Secondly, mapping to users and groups other than oneself requires `CAP_SETUID` or `CAP_SETGID` in the parent namespace. Avoiding this seems a good idea to reduce the ambient authority of the shim.

Voiding the user namespace initially provides the ability to create other namespaces with ambient authority, and hides the details of the Void Process's ambient permissions from inside. Although this creates a binary system of users which may at first seem limiting, applying the context of Void Processes helps realise that it is not. Linux itself may utility users, groups and capabilities for process limits, but Void Processes only provide what is absolutely necessary. That is, if a process should not have access to a file owned by the same user, it is simply not made available. Running only as `root` within the Void Process is therefore not a problem - multiple users is a feature of Linux which doesn't well serve the cause.

4.4 Remaining namespaces

4.4.1 uts namespace

uts namespaces are easily voided by setting the two controlled strings to a static string. However, if one wishes for them to hold specific values, they can be set in one of two ways: either calling `sethostname(2)` or `setdomainname(2)` from within the Void Process, or by providing static values within the Void Process's specification.

4.4.2 ipc namespace

Filling ipc namespaces is also not possible in this context. An ipc namespace is created empty, as stated in Section 3.1. IPC objects exist in one and only one ipc namespace, due to sharing what they expect to be a global namespace of keys. This means that existing IPC objects cannot be mapped into the Void Process's namespace. However, the process within the ipc namespace can use IPC objects, for example between threads. This is potentially inadvisable, because different Void Processes would provide stronger isolation than IPC within a single Void Process. Alternative IPC methods are available which use the filesystem namespace and are better shared in a controlled fashion between Void Processes.

4.4.3 pid namespace

A created pid namespace exists by itself, with no concept of mapping in PIDs from the parent namespace. The first process created in the namespace becomes PID 1, and after that other processes can be spawned from within. As such there is no need to fill pid namespaces, instead applications can be restructured to not expect seeing other process's IDs.

4.4.4 cgroup namespace

cgroup namespaces present some very interesting behaviour in this regard. What appears to be the root in the new cgroup namespace is in fact a subtree of the hierarchy in the parent. This again provides a quite strange concept of filling - elements of the tree cannot be cloned to appear in two places, by design. To provide fuller interaction with the cgroups system, one can instead bind whichever subtree they wish to act on from the parent mount namespace to the child mount namespace. This provides the control of any section of the cgroups subtree seen fit, and is unaffected by the cgroups namespace of the child. That is, the cgroups namespace is used only to provide a void, and the mount namespace can be used to operate on cgroups.

Chapter 5

System Design

An example of running a Void Process application is given in Figure 5.1. What was originally a monolithic application becomes a set of applications that communicate with a new shim. The shim does not replace the kernel, and instead supplements it with new higher-level abilities. Each entrypoint receives input from the shim, and can return data to the shim where appropriate. Most of this data is in the form of file descriptors, which are treated as capabilities in this system.

A Void Process application stores the requirements for running it as static data in the ELF of the binary. When launched, `binfmt_misc` is used to launch the application with the multi-entrypoint shim. The shim decodes this data and sets up processes and inter-process communication (IPC) accordingly.

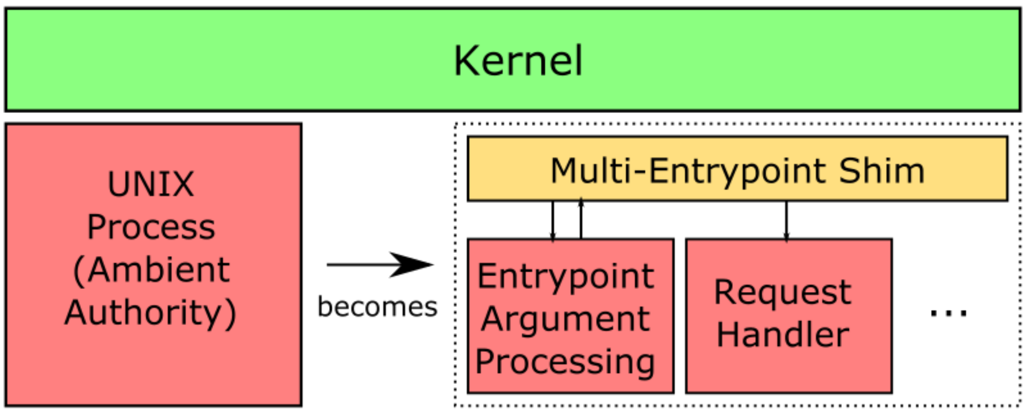
Kernel
UNIX Process (Ambient Authority)
becomes
Multi-Entrypoint Shim
Entrypoint Argument Processing
Request Handler
...

Figure 5.1: Interaction between the application and the environment.

Chapter 6

Building Applications

6.1 No Permissions

The cornerstone of strong process separation is an application that is completely deprivileged. Listing 6.1 shows an application which, when run under the shim, drops all privileges except `stdout`. This is easy to achieve under the shim.

6.2 gzip

GNU gzip [28] is well structured for privilege separation, though doesn't implement it by default. There is a clear split between the processing logic, selecting the items to do work on, and the compression/decompression routines, each of which are handed a pair of input and output file descriptors. This is shown by Watson et al. in [18].

As C does not have high-level language features for multi-entrypoint appli-

Listing 6.1: An application that requires only stdout and stderr.

```
#[entrypoint(stdout)]
fn main() { println!("hello world!"); }
```

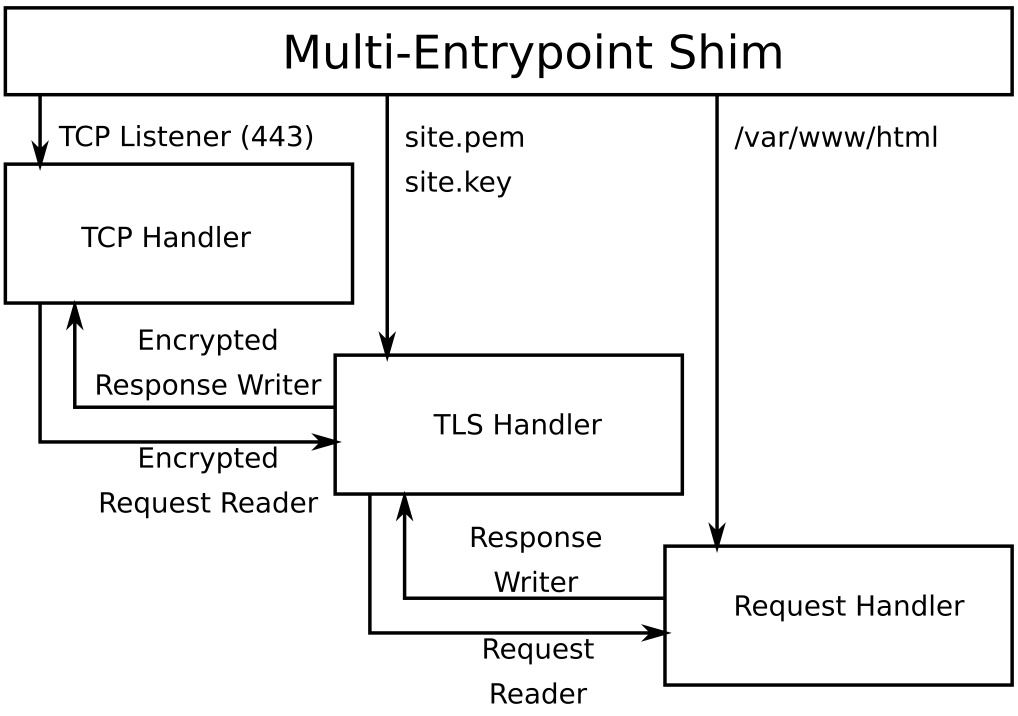

Figure 6.1: Process separation in a TLS server.

cations, adapting it is slightly more verbose than the other examples seen. However, the resulting code change is still only X lines, if a bit more intricate. This places the risky compression and decompression routines in full sandboxes, while still allowing the simpler argument processing code ambient authority. The argument processing code needs no additional Linux capabilities to manage this permissioning, as the required capabilities are provided by the shim.

6.3 TLS Server

Finally, a rudimentary TLS server is created to show the rich privilege separation abilities of multi-entrypoint applications. An example structure is shown in Figure 6.1. Rather than being provided with a view of the network, the initial TCP handling process is given an already bound socket listener by the shim. This allows the TCP handler to live in an extremely restricted zero-access network namespace, while still performing the tasks of receiving new TCP connections.

Next, the TCP handler hands off the new TCP connections to the shim. Though the figure shows this as a direct connection between the TCP handler and the TLS handler, they are passed through the shim, from which the shim spawns a fresh TLS handler for each connection. The TLS handler is handed file descriptors to the certificate and key files that it requires, and hands back a decrypted request reader and an empty response writer file descriptor to the shim.

Finally, this pair of decrypted request reader and response writer are handed to a new process which handles the request. In the example case, this new process is handed a dirfd to `/var/www/html`, which is bind-mounted into an empty file system namespace by the shim. This allows the request handler enough access to serve files, while restricting access to anything else.

Chapter 7

Evaluation

Write evaluation

Chapter 8

Related Work

8.1 Virtual Machines and Containers

Virtual Machine solutions [29, 30] provide the ability to split a single machine into multiple virtual machines. When placing a single application in each virtual machine, they are effectively isolated from one another. Full fat container solutions such as Docker [31], containerd [CN], and systemd-nspawn [CN] provide mechanisms to isolate an application almost completely from other applications running on a single machine. Some have claimed that this provides isolation superior to virtual machines [32].

Both of these solutions are less effective at isolating parts of an application from itself [CN with research]. Consider running only a TLS web server in a virtual machine. Although other applications will be unable to access the certificates, as they are in different virtual machines, methods within the application that should not be able to access the certificates still can.

While virtual machines and containers provide a strong isolation at the application level, they are not a compelling solution to intra-application privilege separation.

8.2 systemd

`systemd` [CN] provides a declarative interface to all of the process separation techniques used in this work. Rather than the responsibility of the programmer, creating these declarative descriptions is most commonly left to the package maintainers. This work seeks to provide similar capabilities to the people best suited to privilege separating an application: the developers.

Chapter 9

Future Work

9.1 Dynamic Linking

Dynamic linking works correctly under the shim, however, it currently requires a high level of manual input. Given that the threat model in Section **??** specifies trusted binaries, it is feasible to add a pre-spawning phase which appends read-only libraries to the specification for each spawned process automatically before creating appropriate voids. This would allow anything which can link correctly on the host system to link correctly in Void Processes.

Chapter 10

Conclusion

Write conclusion

Bibliography

[1] Alexander Viro. [PATCH][CFT] per-process namespaces for Linux, February 2001.

[2] Linus Torvalds. Linux Kernel Version 2.5.2 Changelog, January 2002.

[3] Kirill Korotaev. [PATCH] IPC namespace core, October 2006.

[4] Linux Version 2.6.19 Changelog, November 2006.

[5] Serge E. Hallyn. [PATCH] namespaces: utsname: implement utsname namespaces, October 2006.

[6] Cedric Le Goater. user namespace: add the framework, July 2007.

[7] Linux Version 2.6.23 Changelog, October 2007.

[8] Eric W. Biederman. [NET]: Basic network namespace infrastructure., October 2007.

[9] Linux Version 2.6.24 Changelog, January 2008.

[10] Sukadev Bhattiprolu. [PATCH] Define struct pspace, October 2006.

[11] Tejun Heo. [GIT PULL] cgroup namespace support for v4.6-rc1, March 2016.

[12] Linus Torvalds. Linux 4.6-rc1, March 2016.

[13] Andrei Vagin. ns: Introduce Time Namespace, January 2020.

[14] Linux Version 5.6 Changelog, March 2020.

[15] The MITRE Corporation. Improper Neutralization of Argument Delimiters in a Command ('Argument Injection') (4.7), July 2006.

[16] The MITRE Corporation. Deserialization of Untrusted Data (4.7), July 2006.

[17] Anil Madhavapeddy. privsep.c, July 2003.

[18] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical Capabilities for UNIX. In *USENIX Security Symposium*, volume 46, page 2, 2010.

[19] Free Software Foundation. ipc_namespaces(7), 2021.

[20] Free Software Foundation. time_namespaces(7), 2021.

[21] Jason A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. In *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, 2017. Internet Society.

[22] Ram Pai and Alexander Viro. Shared Subtrees, November 2005. Added in commit 9cfcceea8f7e8f5554e9c8130e568bcfa98a3a64.

[23] Free Software Foundation. mount_namespaces(7), 2021.

[24] Eric W. Biederman. Re: netns : close all sockets at unshare ?, October 2007.

[25] Alexander Viro. [PATCH] lazy umount (1/4), September 2001.

[26] Alexander Viro. [RFC] shared subtrees, January 2005.

[27] The OpenBSD Foundation. pledge(2), February 2022.

[28] Jean-loup Gailly. Gzip, August 2020.

[29] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, October 2003.

[30] Inc. VMware. Understanding Full Virtualization, Paravirtualization and Hardware Assist. Technical report, March 2008.

[31] Dirk Merkel. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2:2, March 2014.

[32] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, New York, NY, USA, March 2007. Association for Computing Machinery.