UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology

Void Processes: Minimising privilege by default

Jake Hillion

Queens' College

June 2022

Submitted in partial fulfillment of the requirements for the
Computer Science Tripos, Part III

Total page count: 46

Main chapters (excluding front-matter, references and appendix): 37 pages (pp 7–43)

Main chapters word count: 8291

Methodology used to generate that word count:

```
£ texcount report.tex | grep Words
Words in text: 8070
Words in headers: 93
Words outside text (captions, etc.): 128
```

`texcount` macros are used to ensure counting begins on the first content page and ends on the last content page.

Abstract

The important of privilege separation - separating the parts of an application with the most risk of attack from the parts with the most reward for an attacker - for protection of resources in a modern operating system cannot be understated. As Linux has grown into the behemoth of an operating system that it is today, many privileges and attack vectors have been enabled, large amounts of which are given passively to new processes. New methods for protecting applications and processes have come along at nearly the same rate. This paper presents void processes: a framework to restrict Linux processes, removing access to ambient resources by default and providing APIs to systematically unlock abilities that applications require. Void processes solve the problem of needing to know what your privilege is in order to reduce it, as an application developer can begin from a clean slate.

This project built a system, the void orchestrator, to enable application developers to build upwards from a point of zero-privilege, rather than removing privilege that they don't need. This report gives the background and technical details of how to achieve this on modern Linux. I present a summary of the privilege separation techniques currently employed in production and details on how to create an empty set of namespaces to remove all privilege in Linux, a technique named entering the void. The shortcomings of Linux when creating empty namespaces are discussed, before setting forth the methods for re-adding features in each of these domains. Finally, two example applications are built and their performance evaluated to show the utility of the system. This report aims to demonstrate the value of a paradigm shift from reducing an arbitrary amount of privilege to adding only what is necessary.

Acknowledgements

This project would not have been possible without the wonderful support of ... [optional]

Contents

1 Introduction **7**

2 Privilege Separation **9**

2.1 Privilege separation by process 9
2.2 Privilege separation by time 11
2.3 Privilege separation by ownership 11
2.4 Privilege separation by using another machine 12
2.5 Privilege separation by perspective 12
2.6 Summary 13

3 Entering the Void **14**

3.1 ipc namespaces 14
3.2 uts namespaces 15
3.3 time namespaces 16
3.4 network namespaces 16
3.5 pid namespaces 18
3.6 mount namespaces 19
3.7 user namespaces 24
3.8 cgroup namespaces 25
3.9 Summary 26

4 Filling the Void **27**

4.1 mount namespace 27
4.2 network namespace 28
4.3 user namespace 29
4.4 Remaining namespaces 30
4.5 Summary 31

5 Building Applications **32**

5.1 System Design 32
5.2 Fibonacci 32
5.3 TLS Server 34

5.4 Summary 38

6 Evaluation **39**

6.1 Startup costs 39

6.2 Runtime impact 39

6.3 Summary 39

7 Conclusions **42**

7.1 Future Work 42

Chapter 1

Introduction

Newly spawned processes on modern Linux are exposed to a myriad of attack vectors and privilege, whether the hundreds of system calls available, `procfs`, exposure of filesystem objects, or the ability to connect to arbitrary hosts on the Internet. This paper presents void processes: a framework to restrict Linux processes, removing access to ambient resources by default and providing APIs to systematically unlock abilities that applications require. Explicit privilege designation with void processes could have saved many applications from the threat of CVE-2021-44228 with Log4j2 by ensuring that the processes which do dangerous user data processing are sufficiently deprivileged to prevent remote code execution (§5.2). Moreover, adding explicit privilege with each change encourages consideration of privilege separation whenever new privilege is added, rather than when flaws are exposed.

This project built a system, the void orchestrator, to enable application developers to build upwards from a point of zero-privilege, rather than removing privilege that they don't need. This report gives the background and technical details of how to achieve this on modern Linux. I present a summary of the privilege separation techniques currently employed in production (§2) and details on how to create an empty set of namespaces to remove all privilege in Linux (§3), a technique named entering the void. The shortcomings of Linux when creating empty namespaces are discussed (§3.6,§3.7,§3.8), before setting forth the methods for re-adding features in each of these domains (§4). Finally, two example applications are built (§5) and evaluated (§6) to show the utility of the system. This report aims to demonstrate the value of a paradigm shift from reducing an arbitrary amount of privilege to adding only what is necessary.

Much prior work exists in the space of privilege separation, including: virtual machines (§2.4); containers (§2.5); object capabilities (§2.3); unikernels; and applications which run directly on a Linux host, potentially employing privilege separation of their own (§2.1, §2.2). These alternative environments are plotted in Figure 1.1, in which the difference between applications written for the environment and the attack surface remaining are

compared. Void processes contribute a strong compromise between providing a rich Linux-like interface for applications, reducing necessary code changes, and significantly reducing the attack surface (demonstrated in §3).

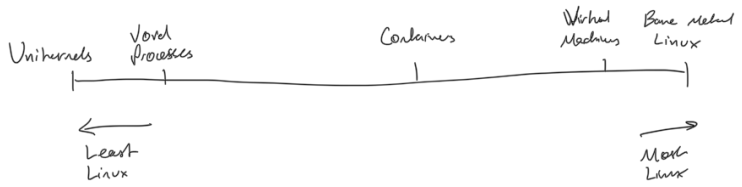

Figure 1.1: Privilege separated environments plotted to compare the number of application changes required against the remaining attack surface of the environment.

Chapter 2

Privilege Separation

Many attack vectors exist in software, notably in argument processing and deserialisation [5, 6]. Creating security conscious applications requires 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 separated from the portion which is likely to be attacked, and that the interface between them is correct. This chapter details what privilege separation is, why it is useful, and a summary of some of the privilege separation techniques available in modern Unices. Many of these techniques are included in some form in the final design for void processes.

2.1 Privilege separation by process

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 that process's privilege. Reducing the privilege of a process therefore reduces the benefit of attacking that process. One solution to reducing privilege in the process is to take a previously monolithic application and split it into multiple smaller processes. Consider a TLS supporting web server that must have access to the certificate's private keys and also process user requests. These elements can be split into different processes. This means that if the user data handling process is compromised the attacker cannot access the contents of the private keys.

Application design in this paradigm is similar to that of a distributed system, where multiple asynchronous systems must interact over various communication channels. As an application becomes more like a networked system, serialisation and deserialisation

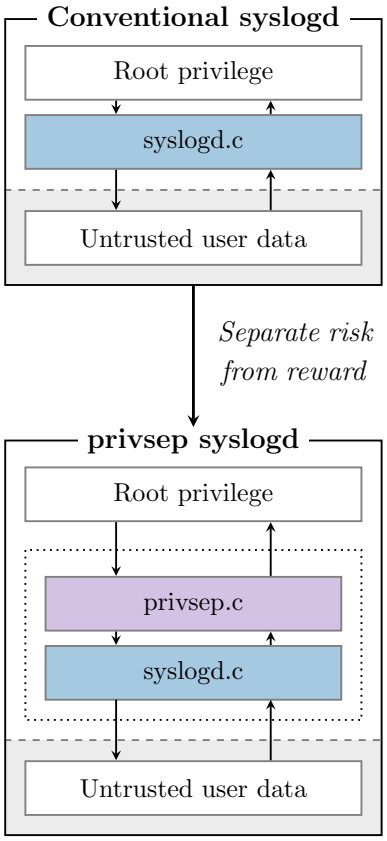

Figure 2.1: Separation of privileged access from untrusted user data in OpenBSD's privilege separated syslogd design compared to the previous. The process which handles untrusted data is separated from the privileged process and uses RPC to communicate.

becomes a common occurrence. As deserialisation is a very common source of exploits [5], this adds the potential for new flaws in the application.

OpenBSD is a UNIX operating system with an emphasis on security. A recent bug in OpenBSD's `sshd` highlights the utility of privilege separation [15]. An integer overflow in the pre-authentication logic of the SSH daemon allowed a motivated attacker to exploit incorrect logic paths and gain access without authentication. Privilege separation ensures that the process with this bug, the pre-authentication process, is separated from the process which is able to be exploited, the highly privileged daemon. Moreover, privilege separation being mandatory in the software ensures that bugs which are not exploitable due to the privilege separation monitor's checks are not exploitable anywhere.

In 2003, privilege separation was added to the `syslogd` daemon of OpenBSD [21]. 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.

2.2 Privilege separation by time

Many applications can privilege separate by using a single process which reduces its level of privilege as the application makes progress. This is effectively privilege separation over time. 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 simplest 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 process, known as dropping privilege. As the privilege has been irreversibly dropped, any attacker who gains control after the privilege has been dropped cannot take advantage of it.

After dropping privilege, it becomes difficult to do things such as reloading the configuration. The application process no longer has the required privilege to restart the application, and if it could gain it back then dropping it would have had no effect. This avoids having to treat the application as a distributed system as there continues to be only a single process to manage, which is often an easier paradigm to work in. The difficulty in implementing privilege dropping is ensuring that you know what privilege you hold, and drop it as soon as it is no longer required.

2.3 Privilege separation by ownership

The previous methods shown each suffer from having to know what their initial privilege is in order to correctly deprivilege. An alternative method to enable the principle of least privilege in applications are object capabilities. An object capability is an unforgeable token of authority to perform some particular set of actions on some particular object.

While the methods looked at until now of privilege separation by process and time are supported by all Unices, object capabilities are a more niche system. Capsicum added object capabilities and was included in FreeBSD 10, released in January 2014 [30]. These capabilities may be shared between processes as with file descriptors. Capability mode removes access to all global namespaces from a process, allowing only operations on capabilities to continue. These capabilities are commonly those opened before the switch to capability mode, but they can also be sent and received (as file descriptors) or converted from a capability with more privilege to a capability with less.

Although object capabilities still require some additional work to ensure that only intentional capabilities remain accessible when entering capability mode, they come a lot closer to easy deprivileging than the previous solutions. However, their adoption remains limited at this point. They are unavailable in the latest Linux kernel release (5.17.7) at

the time of writing, and there are no plans for their adoption.

2.4 Privilege separation by using another machine

One of the older methods of privilege separation is placing parts of an application on entirely different machines. If developing a web application, one might place the PHP backend on one machine and the database server on another. This means that even if a bad actor achieves remote access to the exposed PHP backend, they can only access the database server over its exposed API on the network, rather than having control of the machine itself. This allows features such as the database's access control to remain working, limiting the potential damange of an attacker controlling the PHP server.

Virtual machines [1, 29] made the separation of privilege by machine a much more optimal use of hardware. Rather than requiring two full servers, one might instead provide both the application backend and the database server on a single physical machine but different virtual machines. This increased hardware usage in a time when hardware speed seemed in excess, and provided very strong isolation (presuming one couldn't escape the hypervisor). Though the isolation is strong, there are overheads associated with full virtualisation, and a more performant solution was sought.

2.5 Privilege separation by perspective

Linux approaches increased process separation using namespaces. Namespaces control the view of the world that a process sees. Processes remain the primary method of separation, but utilise namespaces to increase the separation between them. 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 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. 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. 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.

2.6 Summary

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. There is never a need to drop privileges as processes are created with the absolute minimum privilege necessary to perform correctly. In Chapter 3 we discuss each namespace's role in Linux and how to create one which is empty, before explaining in Chapter 4 how to reinsert just enough Linux for each process in an application to be able to complete useful work. These combine to form an architecture which minimises privilege by default, motivating highly intentional privilege separation.

Chapter 3

Entering the Void

Isolating parts of a Linux system from the view of certain processes is achieved using namespaces. Namespaces are commonly used to provide isolation in the context of containers, which provide the appearance of an isolated Linux system to contained processes. Instead, with void processes, we use namespaces to provide a view of a system that is as minimal as possible, while still sitting atop the Linux kernel. In this chapter each namespace available in Linux 5.15 LTS is discussed. The objects each namespace protects are presented and security vulnerabilities discussed. Then the method for entering a void with each namespace is given along with a discussion of the difficulties associated with this in current Linux. Chapter 4 goes on to explain how necessary features for applications are added back in.

The full set of namespaces are represented in Table 3.1, 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 with absolute separation. We start with those that exhibit the clearest behaviour when it comes to entering the void, working up to the namespaces most difficult to separate from their parents.

3.1 ipc namespaces

Inter-Process Communication (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 use System V IPC objects which collide [1, §4.3]. IPC namespaces solve this effectively for containers by creating a new scoped namespace. Processes are a

Table 3.1: 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 of each namespace, and CVEs that each namespace protects against.

ns	date	kernel ver.	ns CVEs	prot. CVEs
`mount`	Feb 2001 [27]	2.5.2 [23]	2020-29373	2021-23021 2021-45083 2022-23653
`ipc`	Oct 2006 [19]	2.6.19 [8]		2015-7613
`uts`	Oct 2006 [17]	2.6.19 [8]		
`user`	Jul 2007 [20]	2.6.23 [9]	2021-21284	2021-43816
`network`	Oct 2007 [3]	2.6.24 [10]	2009-1360	2021-44228
`pid`	Oct 2006 [2]	2.6.24 [10]	2019-20794	2012-0056
`cgroup`	Mar 2016 [18]	4.6 [24]	2022-0492	
`time`	Nov 2019 [25]	5.6 [11]		

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 [12]:

"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. IPC namespaces represent a relatively small attack surface and appear to function well as a namespace (a series of searches revealed no results). Similarly, the historical SysV IPC and POSIX message queues that are isolated show very few bugs. One was found (CVE-2015-7613) which describes a race condition leading to escalated privilege. From the limited information available, it seems that namespacing and hence void processes protect well against this, as the escalated privilege is isolated to the calling namespace.

3.2 uts namespaces

Unix-Time Sharing (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 inherit their values. Each of the

hostname and domain name in the new namespace is initialised to the values of the parent namespace.

As the inherited 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. Although the implementation of this is trivial, it highlights how easy the information passing elements of each namespace are to miss if manually implementing isolation with namespaces.

3.3 time namespaces

Time namespaces are the final namespace added at the time of writing, added in kernel version 5.6 [11]. The motivation for adding time namespaces is given in the manual page [14]:

"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. They do not attempt to virtualise 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.

Searching the list of released CVEs for both "clock" and "time linux" (time itself revealed significantly too many results to parse) shows no vulnerabilities in the time subsystem on Linux, or the time namespaces themselves. This supports not including time namespaces at this stage, as their range is very limited, particularly in terms of isolation from vulnerabilities.

3.4 network namespaces

Network namespaces on Linux isolate the system resources related to networking. These include network interfaces themselves, IP routing tables, firewall rules and the `/proc/net` directory. This level of isolation allows a network stack that operates completely independently to exist on a single kernel.

Similarly to IPC, network namespaces present the optimal namespace for running a void

```
#                                                  # unshare -n
#                                                  # ip netns attach test ff
# 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 addr add 192.168.0.2/24 dev veth1
# ip link set up dev veth0                         # ip link set up dev veth1
# ping -c 1 192.168.0.2                            # ping -c 1 192.168.0.1
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of da PING 192.168.0.1 (192.168.0.1) 56(84) bytes of da
64 bytes from 192.168.0.2: icmp_seq=1 ttl=64 time 64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time
```

Figure 3.1: Parallel shell sessions showing the creation of a virtual Ethernet pair between the root network namespace and a newly created and completely empty network namespace.

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 (Figure 3.1). Alternatively, one can create a Wireguard adapter with sending and receiving sockets in one namespace and the VPN adapter in another [7, §7.3]. These methods allow for very high levels of separation while still maintaining access to the primary resource - the Internet or wider network. Further, this design places the management of how connected a namespace is to the parent in user-space. This is a significant difference compared to some of the namespaces discussed later in this chapter.

Network namespaces are also the first mentioned to control access to `procfs`. `/proc` holds a pseudo-filesystem which controls access to many of the kernel data structures that aren't accessed with system calls. Achieving the intended behaviour here requires remounting `/proc`, which must be done with extreme care so as not to overwrite it for every other process. In a void process this is handled by automatically voiding the mount namespace, meaning that this does not need to be intentionally taken care of.

Network namespaces have significantly more to isolate than the namespaces mentioned thus far. We see with CVE-2009-1360 that this hasn't been bug free, though the issues are few and far between. That particular vulnerability references a user triggering a kernel null-pointer dereference via passing vectors of IPv6 packets. However, the ability to revoke Internet and network access could have prevented almost an infinite amount of flaws in the time since. Most notable is CVE-2021-44228, a remote code execution bug that took the world by storm recently. Empty network namespaces for applications which don't require networking protect very well against remote code execution, as the ability for remote access is lost.

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, named init, PID namespace. This isolation behaviour is different to that of the namespaces discussed thus far, as each process within the namespace represents a process in the parent namespace too, albeit with different identifiers.

As with network namespaces, PID namespaces have a significant effect on `/proc`. Further, they cause some unusual behaviour regarding the PID 1 (init) process in the new namespace. These behaviours are shown in Listing 3.5. The first behaviour shown is that an `unshare(CLONE_PID)` call followed immediately by an `exec` does not create a working shell. 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, both of which ensure that the correct process is created first in the new namespace. 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 system call.

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, meaning that any access to `procfs` must be either freshly mounted or bound to outside the namespace intentionally. Remounting a fresh `procfs` is unfortunately not trivial on most systems, and will be discussed with user namespaces (§3.7).

PID namespaces are also of increased complexity as they enable something completely new in Linux: PID 1 processes that may terminate without the system. That is, the init process of an ordinary Linux systems survive until reboot, whereas the init process of a container survives only until the container exits. This raises issues with cleanup, such as CVE-2019-20794 where FUSE filesystems aren't correctly cleaed up on PID namespace exit. Vulnerabilities that PID protects from are quite hard to find, but a good example is CVE-2012-0056. A bug existed where a `setuid` binary could be coereced into writing to arbitrary process's memory. However, if one can't see the processes in their `/proc` because of the protection of PID namespaces, this bug is avoided.

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

£ unshare --fork --pid
# (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 --pid
# (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
```

Listing 1: Unshare behaviour with pid namespaces, with and without forking and remounting proc. Spawning a process without explicitly forking creates a broken shell. Forking creates a shell that works, but the PID namespace appears unchanged to processes that inspect it. Remounting proc and forking provides a working shell in which processes see the new pid namespace.

3.6 mount namespaces

One of the defining philosophies of Unix is everything's a file. This perhaps explains why mount namespaces, the namespaces which control the single file hierarchy, would be the most complex. This section presents a case study of the implementation of voiding the most difficult namespace and an analysis of why things were so much more difficult to implement than with others. We first look at the inheritance behaviour, and the link maintained between a freshly created namespace and its parent (§3.6.1). Secondly, I present shared subtrees and the reasoning behind them (§3.6.2), before finishing with a discussion of lazy unmounting in Linux and the weakness of the userspace utilities (§3.6.3). This culminates in a namespace that is successfully voided, but presents a huge burden to userspace programmers attempting to work with these namespaces in their own projects.

The filesystem on Linux provides access to most of the system. It follows that a correctly isolated mount namespace would protect against a horde of filesystem bugs. Most commonly the protection is against incorrectly set DAC, where a file will have permissions `0644` (guest read) while containing private API keys (CVE-2021-23021). Bugs to escape the mount namespace still crop up, though at this stage it is relatively stable.

3.6.1 Filesystem inheritance

Compared to network namespaces, there is 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.

```c
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
...
```

Listing 2: Reading the same file before and after unsharing the mount namespace demonstrates no observable change in behaviour, showing that more work must be done to create an empty namespace.

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

```
# 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
```

Listing 3: Parallel shell sessions showing highly separated behaviour without shared subtrees between mount namespaces. A folder in the parent namespace that is bound may still show different results in each namespace if the mounts have changed.

Shared subtrees [22] were introduced to provide a consistent view of the unified hierarchy between namespaces. Consider the example in Listing 3.6.2. `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 [13]:

"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. That is, if a mount is unmounted in the new namespace, it is also unmounted in 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 Listing 3.6.2, 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 Listing 3.6.2: the existing

```c
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
```

Listing 4: Attempting to unmount the private root directory after an unshare results in an error that the resource is busy when no files have been opened on it in the new namespace.

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

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

Listing 5: Parallel shell sessions demonstrating the behaviour in the parent namespace when attempting to lazily unmount the root filesystem from an unshared shell with a shared mount. The mount of procfs in the parent is lost even though the unmount was performed in a different namespace.

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 [4]. 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 [7, §7.3].

Something which behaves differently is the memory mapping of a currently running process's binary. Consider the example in Listing 3.6.3, 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 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 system call is incredibly dangerous when combined with shared subtrees. This behaviour is shown in Listing 3.6.3, 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 [26]:

"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 (sic) 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 [26], as mounts were not initially shared but duplicated between namespaces. However, when shared subtrees were added in January 2005 [28], 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. 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.

User namespaces again interact with `procfs`, which brings up an interesting limitation to the capabilities available in user namespaces. On most systems, `procfs` has a variety of mounts over parts of it. This might be to interact with a hypervisor such as Xen, to support `binfmt_misc` for running special applications, or Docker protecting the host from container mishaps. Most interestingly with Docker, these mounts are used to protect the host from the container accessing certain files. The series of mounts on one of my machines are shown in Listing 3.7. The objects mounted over include `/proc/kcore`, which presents direct access to all of the kernel's allocatable memory. Linux protects these mounts by enforcing that `procfs` with mounts below it can only be mounted in a new place if the user has root privilege in the init namespace. Fortunately, one can instead perform a small dance of first binding `/proc` to the parent namespace before remounting it, which is allowed with mounts below. Further, by running the void process with restricted authority (limited to that of the calling user even as root), the dangerous files in `/proc` are protected using discretionary access control. This avoids the requirement of adding extra mounts in the void orchestrator.

User namespaces act as both a blessing and a curse for security. In the case of Docker, with CVE-2021-21284, a remapped user may be able to alter the initial source of the

```
# docker run --rm ubuntu cat /proc/mounts | grep proc
proc /proc proc rw,nosuid,nodev,noexec,relatime 0 0
proc /proc/bus proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/fs proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/irq proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/sys proc ro,nosuid,nodev,noexec,relatime 0 0
proc /proc/sysrq-trigger proc ro,nosuid,nodev,noexec,relatime 0 0
tmpfs /proc/asound tmpfs ro,relatime 0 0
tmpfs /proc/acpi tmpfs ro,relatime 0 0
tmpfs /proc/kcore tmpfs rw,nosuid,size=65536k,mode=755 0 0
tmpfs /proc/keys tmpfs rw,nosuid,size=65536k,mode=755 0 0
tmpfs /proc/timer_list tmpfs rw,nosuid,size=65536k,mode=755 0 0
tmpfs /proc/scsi tmpfs ro,relatime 0 0
```

Listing 6: The mounts at and below /proc in a Ubuntu Docker container demonstrate the many additional mounts on top of procfs.

mappings, causing them to be overridden and gaining root access. In contrast with containerd, with CVE-2021-23021, an always root containerd daemon mounts files that shouldn't be accessible with DAC due to a logic error. Mapped user namespaces preserve DAC, protecting against this sort of incorrect code compared to a root daemon.

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. Running the shim with ambient autrhoirty here presents an issue, as the cgroup hierarchy relies on discretionary access control. 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 present.

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 init 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.

There are two problems when working with cgroups namespaces in user-space: needing sufficient discretionary access control, and leaving the control of individual application processes in a global namespace. An alternative kernel design would increase the utility by solving both of these problems. A process in a new cgroups namespace could instead create a detached hierarchy with the process as a leaf of the root and full permissions in the user-namespace that created it. The main cgroups hierarchy could then still see a single application to control, while the application itself would have full access over sharing its resources. This presents the ability for mechanisms of managing cgroups to clash between the namespaces, as the outer namespace would now have control over what resources are delegated to the application rather than each process in the application. Such a system would also provide improved behaviour over the current, which requires a delegation flag to be handed to the manager informing it to go no further down the tree. This would be significantly better enforced with namespaces. That is, the main namespace could be handled by `systemd`, while the `/docker` namespace could be internally managed by docker. This would allow `systemd` to move the `/docker` namespace around as required, with no awareness of the choices made internally.

3.9 Summary

In this chapter I presented the 8 namespaces available in Linux 5.15. What each namespace protects against, how to completely empty each created namespace, and the constraints in doing so were presented. For cgroup and mount namespaces, alternative designs that increase the usability of the namespaces were discussed.

Now that the motivation for emptying namespaces has been shown with the avoidance of vulernabilities, facilities to re-expose some of the system must be introduced in order to make useful applications. The methods for reintroducing parts of the system are given in Chapter 4, before demonstrating how to build useful applications in Chapter 5.

Chapter 4

Filling the Void

Now that a completely empty set of namespaces are available for a void process, the ability to reinsert specific privileges must be added to support non-trivial applications. To allow for running applications as void processes with minimal kernel changes, this is achieved using a mixture of file-descriptor capabilities and adding elements to the empty namespaces. Capabilities allow for very explicit privilege passing where suitable, while adding elements to namespaces supports more of Linux's modern features.

4.1 mount namespace

There are two options to provide access to files and directories in the void. Firstly, for a single file, an opened 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 only retain the path of the directory when in a different root or namespace. This means that a process with a directory file descriptor in another namespace cannot use it to access files below the namespace, removing all utility as capabilities in the void. 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 some backwards compatibility with applications that are more difficult to adapt.

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 subsystem 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. These request capabilities can be dealt with in the same process or handed back to the shim to be distributed to another void process.

Outbound networking is more difficult to re-add to a void process than inbound networking. The approach that containerisation solutions such as Docker take by default is using NAT with bridged adapters [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. 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 exposed inbound sockets of its own.

Consideration is given to providing outbound access with statically created and passed sockets, the same as inbound access. 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 could 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 [16]. This seems like a good fit, though operating outside of the operating system makes the implementation very different. Acceptable sockets are specified in the application specification, then an interaction socket is 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

```
£ ls -ld repos owned_by_root
-rw-r--r-- 1 root  root     0 May  7 22:13 owned_by_root
drwxrwxr-x 7 alice alice 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
```

Listing 7: A directory listing before and after entering a user namespace with mapped root demonstrates filesystem objects owned by the mapped (calling) user shown as being owned by root and any other filesystem objects shown as being owned by nobody.

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 each socket requested.

4.3 user namespace

Filling a user namespace is a slightly odd concept compared to the namespaces already discussed in this section. A user namespace comes with no implicit mapping of users whatsoever (§3.7). To enable applications to be run with bounded 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 4.3, where a directory owned by the calling user, `alice`, 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.

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 `0707`). 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 is well advised 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 demonstrates that it is not. Linux itself may utilise 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 assist void processes in providing minimum privilege, so is absent.

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, as ipc namespaces are created empty (§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.

4.5 Summary

Included in the goal of minimising privilege is providing new APIs to support this. A mixed solution of capabilities, capability creating capabilities, and file system bind mounts is used to re-add privilege where necessary. Moreover, a form of interface thinning is used to ban APIs which do not well fit the model. Now that void processes with useful privilege can be created, Chapter 5 presents a set of three example applications which make use of them for privilege separation.

Chapter 5

Building Applications

This section discusses the process of creating applications which utilise void processes. Firstly I present the structure of the system used to engage with void processes, the void orchestrator. Then an application which requires no privilege is demonstrated (§5.2), showing how to put together a simple application that takes advantage of void processes to start with no privilege. Finally, a basic HTTP file server with TLS support is designed and built from the ground up for void processes (§5.3).

5.1 System Design

The central development of void processes is the void orchestrator, a shim that uses an application binary and a text specification to set up the series of processes required for privilege separation. The specification describes a series of entrypoints, each of which contain three things: a trigger to create the process, a list of arguments, and extra elements for the environment. Specifications for the example applications are listed through the rest of this chapter.

There are two types of entrypoints: those spawned at startup, and those spawned when triggered by an event. This event, as shown in the TLS server example (§5.3) is most commonly sending one or more file descriptors from a different void process. This allows effective high performance communication.

5.2 Fibonacci

To begin displaying the power of the void orchestrator system we will develop an application that requires completely minimal privilege. The application and its fixed output are shown, unmodified, in Listing 5.2. The application is written in Rust, my language of choice, but there is no such requirement - an equivalent program would look very similar in C. The limited code of this example makes the privilege requirements quite clear.

```rust
fn main() {
    println!("fib(1) = {}", fib(1));
    println!("fib(7) = {}", fib(7));
    println!("fib(19) = {}", fib(19));
}

fn fib(i: u64) -> u64 {
    let (mut a, mut b) = (0, 1);
    for _ in 0..i {
        (a, b) = (b, a + b);
    }
    a
}
```

```
fib(1) = 1
fib(7) = 13
fib(19) = 4181
```

Listing 8: A basic Fibonacci application. The application computes elements of the Fibonacci sequence on static indices and does not process any user input.

Computing `fib` requires no privilege at all, operating purely on numbers on the stack. Once the values are computed they are printed using the `println!` macro, which prints to stdout. Therefore the only privilege this application requires to correctly run is access to stdout.

To run this application as a void process we require a specification (§5.1) to detail how the processes of the application should be set up. The specification for the Fibonacci application is given in Listing 5.2. When specifying an entrypoint for an application every privilege needed must be specified explicitly. In this case, as discussed, the application only requires special access to stdout. This is specified in the environment section of the entrypoint. We also see in the specification a variety of libraries made available, required for the application to successfully dynamically link. This information is decidable from the binary, but implementing that is left for future work (§7.1.2). We also see that no arguments are specified, although they are a part of the specification. No specified arguments defaults to no arguments, as the void orchestrator minimises privilege by default. The application void process therefore receives no arguments - including `arg0` as the binary name.

More of the advanced features of the system will be shown in the future examples, but this is enough to get a basic application up and running. We can see that the Rust application looks exactly like it would without the shim, at least for now. The application is also fully deprivileged. Of course, for an application as small as this example, we can verify by hand that the program has no foul effects. We can imagine a trivial extension that would make this program more dangerous: using a user argument (a privilege the program does not currently have) to take a value on which to execute fib. One way this user input could cause damage is with flawed usage of a logging library. The recent example of Log4j2 with CVE-2021-44228 springs to mind, enabling an attacker with string control to

```
{"entrypoints": { "fib": { "environment": [
    "Stdout",
    {
        "Filesystem": {
            "host_path": "/lib/x86_64-linux-gnu/libgcc_s.so.1",
            "environment_path": "/lib/libgcc_s.so.1"
        }
    },
    {
        "Filesystem": {
            "host_path": "/lib/x86_64-linux-gnu/libc.so.6",
            "environment_path": "/lib/libc.so.6"
        }
    },
    {
        "Filesystem": {
            "host_path": "/lib64/ld-linux-x86-64.so.2",
            "environment_path": "/lib64/ld-linux-x86-64.so.2"
        }
    }
]}}}
```

Listing 9: The specification for the void orchestrator to run the application shown in Listing 5.2. A single entrypoint is provided with a minimal environment, including only the content to dynamically link the binary and standard output.

execute arbitrary code from the Internet. A void process with privilege of only arguments and stdout would protect well against this vulnerability, as not only is there no Internet access to pull remote code, but there is nothing to take advantage of in the process even if remote code execution is gained.

5.3 TLS Server

Rather than presenting the complete applications as shown in the previous two sections, the TLS server presents instead a case study on designing applications from the ground up to run as void processes. The thought process behind data flow design and taking advantage of the more advanced void orchestrator features is given. This results in the process separation presented in Figure 5.1. First we must accept TCP requests from the end user (§5.3.1). Then, to be able to check that all is working so far, we respond to these requests (§5.3.2). Finally, we add an encryption layer using TLS (5.3.3). This results in a functional TLS file server with strong privilege separation, with each stage having no more privilege than it needs.

5.3.1 TCP listener

The special privilege required by a process which accepts TCP connections is a listening TCP socket. As discussed in Section 4.2, TCP listening sockets are handed already bound to void processes. This enables a capability model for network access, otherwise restricting inbound and outbound networking entirely. The specification for this listener is given in

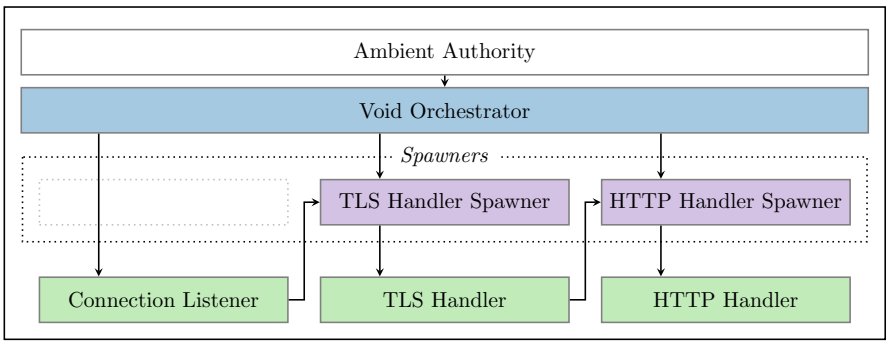

Figure 5.1: The final process design for a TLS server running under the void orchestrator. The figure is split into processes running void orchestrator code and processes running user code. Arrows represent a passing of privilege from one process to another.

```
{"entrypoints": { "tcp_listener": {
        "args": [
                { "TcpListener": { "addr": "0.0.0.0:8443" } }
        ]
}}}
```

Listing 10: The void orchestrator specification for the TCP listener endpoint of the TLS application. The privilege to use a TCP listener is requested as an argument. Dynamic linking binds are omitted for brevity.

Listing 5.3.1, where the TCP listener is requested as an argument already bound. No other permissions are required to accept connections from a TCP listener. Although the code at each stage is omitted for brevity, the resulting program has to parse the argument back into an integer and then a `TcpStream` before looping to receive incoming connections. When building and debugging software it is often useful to have access to the `stdout` or `stderr` streams, even though they won't be utilised in production. The void orchestrator provides useful `--stdout` and `--stderr` flags to temporarily privilege an application for debugging without modifying its specification. Of course, we can't do much useful with them without more privilege. Thus we move on to developing the HTTP handler.

5.3.2 HTTP handler

When attempting to add the HTTP handler, we immediately require more privilege. As this is intended to be a file server, we need some files. Although it would be easy to add files to the existing entrypoint, the principle of least privilege is highly encouraged when developing a void process. One should always ask whether an entrypoint needs a new privilege that they are about to add to it, or whether they would be better served with a new entrypoint.

In this case, we are going to add a new entrypoint for two reasons: multiprocessing and privilege separation. This allows the TCP listener entrypoint to continue in a tight loop, accepting requests very quickly and fanning them out to new processes. These new processes have only their required privileges: the files they wish to serve, and the

```
{"entrypoints": {
    "tcp_listener": {
        "args": [
            "Entrypoint",
            { "FileSocket": { "Tx": "http" } },
            { "TcpListener": { "addr": "0.0.0.0:8443" } }
        ]
    },
    "http_handler": {
        "trigger": { "FileSocket": "http" },
        "args": [ "Entrypoint", "Trigger" ],
        "environment": [{ "Filesystem": {
            "host_path": "/var/www/html",
            "environment_path": "/var/www/html"
        }}]
    }
}}
```

Listing 11: The void orchestrator specification for the TCP listener endpoint and HTTP handler endpoint of the TLS application. This extends on Listing 5.3.1 by adding the HTTP handler endpoint. A new File Socket is used to link the two entrypoints together. Dynamic linking binds are omitted for brevity.

```
fn main() {
    match std::env::args().next() {
        Some(s) => match s.as_str() {
            "connection_listener" => connection_listener_entrypoint(),
            "http_handler" => http_handler_entrypoint(),
            _ => unimplemented!(),
        },
        None => unimplemented!(),
    }
}
```

Listing 12: The main function for the TLS server. This matches on the entrypoint arg0 to determine which entrypoint the application has been run for.

`TcpStream` to serve them down. We take advantage here of another feature of the void orchestrator, file socket based triggers. These allow a statically defined socket to be setup which the void orchestrator will listen on and create new void processes on demand. Further, this ensures isolation between requests too, meaning that a single failed request that causes a process to fail will not affect any others, and a compromised process can't leak information about any other requests either.

The HTTP handler entrypoint is added to the specification in Listing 5.3.2. As well as adding a single extra argument to trigger the HTTP handler, we must also add an entrypoint argument to differentiate between the two entrypoints. Much like the usage of `arg0` for symlinked binaries, we utilise `arg0` to find which intended use of the binary is being called.

```
{"entrypoints": {
    "connection_listener": {
        "args": [
            "Entrypoint",
            { "FileSocket": { "Tx": "tls" } },
            { "TcpListener": { "addr": "0.0.0.0:8443" } }
        ]
    },
    "tls_handler": {
        "trigger": { "FileSocket": "tls" },
        "args": [
            "Entrypoint",
            { "FileSocket": { "Tx": "http" } },
            { "File": "/etc/ssl/certs/example.com.pem" },
            { "File": "/etc/ssl/private/example.com.key" },
            "Trigger"
        ]
    },
    "http_handler": {
        "trigger": { "FileSocket": "http" },
        "args": [ "Entrypoint", "Trigger" ],
        "environment": [{ "Filesystem": {
            "host_path": "/var/www/html",
            "environment_path": "/var/www/html"
        }}]
    }
}}
```

Listing 13: The void orchestrator specification for the final TLS application. This extends on Listing 5.3.1 by adding the HTTP handler endpoint. A new File Socket is used to link the two entrypoints together. Dynamic linking binds are omitted for brevity.

5.3.3 TLS handler

The final stage is to add the TLS handling into the mix. Once again we have the choice of whether to add this to an existing entrypoint or create a new one. This decision is very similar to HTTP handling, but perhaps more important. Rather than adding the `www` directory that we intend to serve publicly anyway, we are entrusting a process with the private keys of the TLS certificate, allowing anyone who takes over the process to impersonate us. This is again an excellent time for more privilege separation, so the TLS handling will be added as an additional entrypoint.

The resulting specification is given in Listing 5.3.3. The TLS handler is added in a very similar manner to the previous HTTP handler. It is triggered by a file socket, but this time receives another file socket to trigger the next stage. It receives file descriptor capabilities to each the certificate and private key files, along with the TCP stream. This process receives nothing but highly restricted capabilities, ensuring that there is very little attack surface for compromise.

We now have a full specification for a TLS server. In this section I have focused entirely on building up the specification and not the code behind it. There are two reasons for this: the code has a lot of boilerplate argument processing, and a variety of code

implementations are available. The boilerplate argument processing could be addressed with future work using features like proc macros in Rust which dynamically generate code based on the code that is already there (§7.1.3). As for varying implementations, I chose to use the static library `rustls` to implement my TLS server. Perhaps someone else would prefer OpenSSL or LibreSSL, which is of course fine. For the HTTP part I use a random library I found on the Internet to parse HTTP headers before responding only to GET requests. Of course this approach is hugely error prone, but the separation of the HTTP handler from the sensitive TLS material and other parts of the filesystem increases my confidence. The implementation therefore matters very little in this analysis, but is made available at `https://github.com/JakeHillion/void-orchestrator/tree/main/examples/tls` and along with this dissertation.

5.4 Summary

While avoiding looking at the internals, I've demonstrated how void processes can both run a standard process with no privilege requirements and define a structure for a new application. Explicit definitions of privilege can make it very clear to the programmer where privilege boundaries are, leading to effective privilege separation. In Chapter 6 we will look at the performance changes caused by these designs, where the use of standard file descriptors as capabilities will highlight how performant this design can be.

Chapter 6

Evaluation

Privilege separation often presents a trade-off between performance and security. This evaluation attempts to quantify that overhead, first discussing the cost of creating void processes (§6.1), both theoretically and on the Fibonacci test application (5.2). Secondly, we take a look at the runtime overhead of privilege separation (§6.2), specifically on the TLS server example (5.3).

6.1 Startup costs

Every void process created requires a set of 7 unique namespaces, which is a lot of work compared to a standard `fork(2)`/`vfork(2)` call. Here I evaluated the overhead of such operations, first on the raw clone calls, and secondly on launching the basic Fibonacci application (§5.2).

6.2 Runtime impact

6.3 Summary

Evaluation: summary.

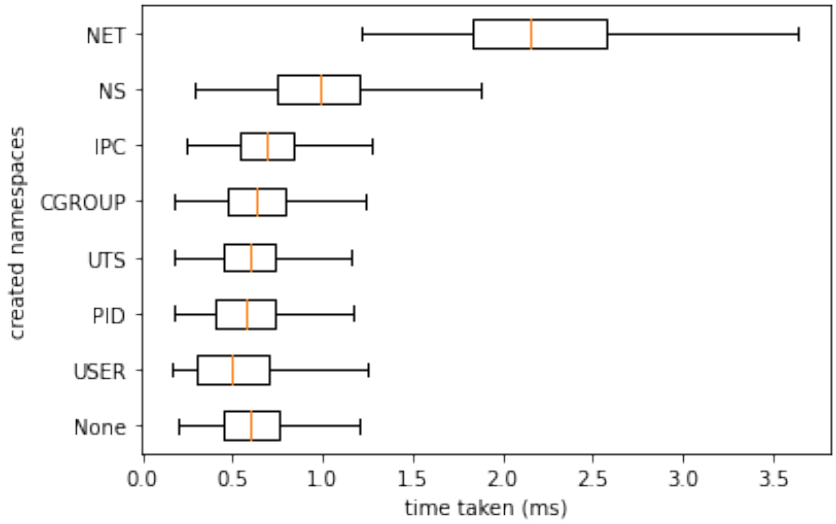

Figure 6.1: Performance of making the `clone(2)` system call with varying namespace creation flags. The test is run in a tight compiled C loop with high precision timings taken before and after each new process is cloned and waited for. `clone(2)` presents very noisy results on a system with background activity.

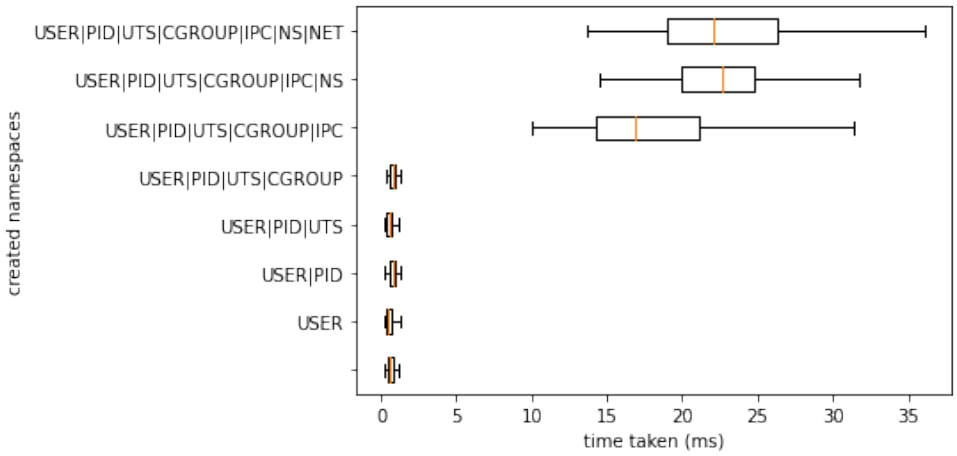

Figure 6.2: Performance of making the `clone(2)` system call with increasing amounts of namespace creation flags. The effects of Figure 6.1 are amplified when creating multiple namespaces in a single call this frequently. There is a clear divide between the time taken for user, pid, uts, and cgroup namespaces and ipc, ns and net namespaces.

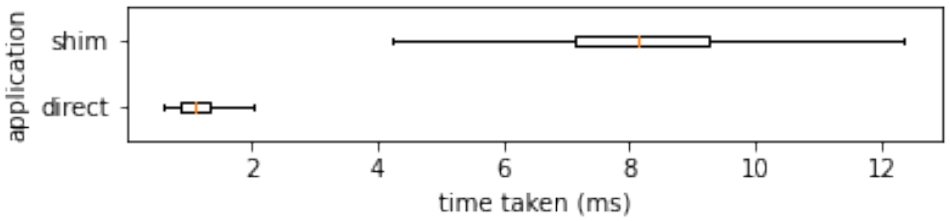

Figure 6.3: A box plot comparing the performance of the Fibonacci example (§5.2 under the shim and called directly. The median time to run under the shim is approximately 800% the time without. The inter-quartile range and range of results is also much larger.

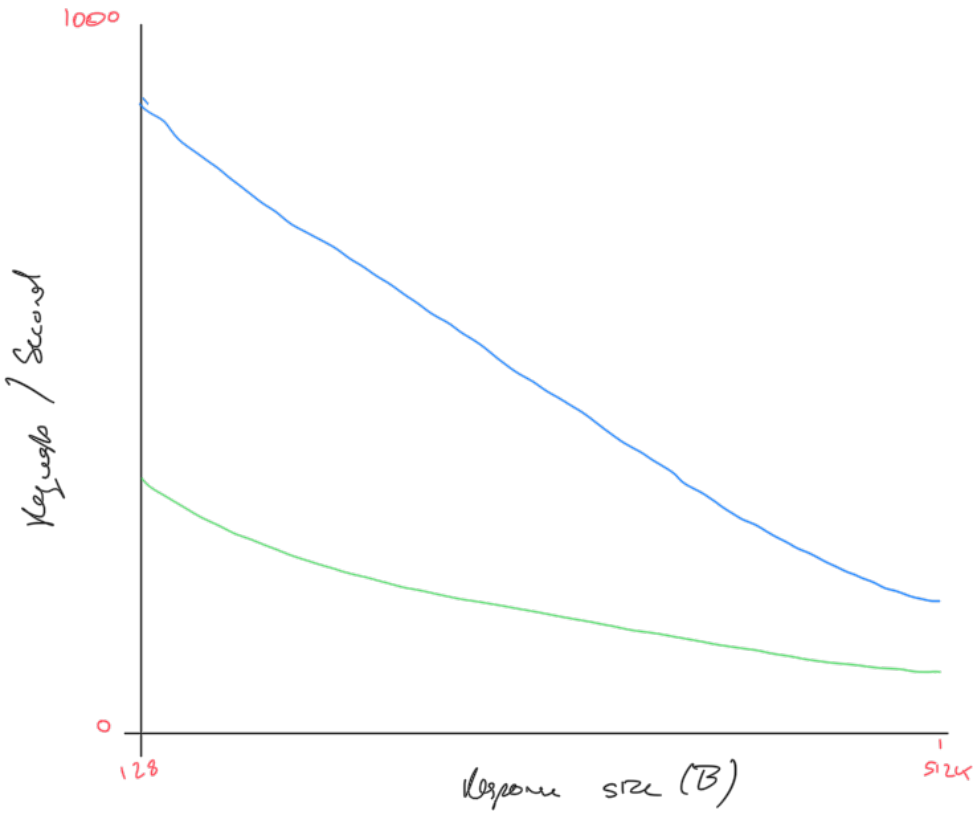

Figure 6.4: `a2bench` requests per second results over 10 seconds with 100 simultaneous requests on varying response sizes. As the response size increases, the gap between the `apache2` TLS web server and the void process TLS web server decreases.

Chapter 7

Conclusions

The system built in this project enables running applications with minimal privilege in a Linux environment in a novel way. Performance is shown to be comparable, and demonstrates where the existing kernel setup provides inadequate performance for such applications. Design choices in the user-space kernel APIs for namespaces are discussed and contextualised, with suggestions offered for alternate designs.

Void processes offer a new paradigm for application development which prioritises privilege separation above all else. Rather than focusing on limiting backward compatibility, applications often need to be completely rewritten in order to take advantage of improved isolation. The system is designed to support effective static analysis on applications, though this is not implemented at this stage.

Finally, void processes provide a seamless experience without making kernel level changes, allowing for ease of deployment. Moreover, it runs on the Linux kernel, a production kernel and not a research kernel. Although the current kernel structure limits the performance of the work with namespace creation being the bottleneck, the feasibility of namespaces for process isolation is effectively demonstrated in a system that encourages application writers to develop with privilege separation as a first principle.

7.1 Future Work

7.1.1 Kernel API improvements

The primary future work to increase the utility of void processes is better performance when creating empty namespaces. Section 6.1 showed that the startup hit when creating the namespaces for a void is very high. This shows a limitation of the APIs, as creating a namespace that has no relation to a parent should involve a small amount of work. Secondly, an API similar to network namespaces adding paired interfaces between namespaces should be added for binding in mount namespaces, allowing mount names-

paces to also be created completely empty. This would also benefit containers which by default have no connection to the parent namespace, but need to mount in their own root filesystem.

7.1.2 Dynamic linking

Dynamic linking works correctly under the shim, however, it currently requires a high level of manual and static input. If one assumes trust of the binary as well as the specification, 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 with no additional effort.

7.1.3 Building specifications from code

Much of the information given in the specification and the code is shared. For example, the specification may list the arguments and also imply their type. This means that a function signature for an entrypoint implies almost all of the specification of an entrypoint, which would allow effective code generation with some supplementary information. This would remove many of the boilerplate argument processing lines from the examples and increase the usability of the system. Combining this with the dynamic linking work (§7.1.2) would remove a huge amount of the manual effort in creating the specification, making the system more user friendly.

7.1.4 Dynamic requests

A system for dynamically requesting statically specified network sockets was presented (§4.2). This system of requests back to the shim could be extended to more dynamic behaviour for software that requires it. Some software, particularly that which interfaces with the user, is not able to statically specify its requirements before starting. By specifying instead a range of requests which are legal then making them dynamically, void processes would be able to support more software.

Bibliography

[1] 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. Available from: https://doi.org/10.1145/1165389.945462.

[2] Sukadev Bhattiprolu. **[PATCH] Define struct pspace**, October 2006. Available from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3fbc96486459324e182717b03c50c90c880be6ec.

[3] Eric W. Biederman. **[NET]: Basic network namespace infrastructure.**, October 2007. Available from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=5f256becd868bf63b70da8f2769033d6734670e9.

[4] Eric W. Biederman. **Re: netns : close all sockets at unshare ?**, October 2007. Available from: https://lore.kernel.org/all/m1r6kccexw.fsf@ebiederm.dsl.xmission.com/.

[5] The MITRE Corporation. **Deserialization of Untrusted Data (4.7)**, July 2006. Available from: https://cwe.mitre.org/data/definitions/502.html.

[6] The MITRE Corporation. **Improper Neutralization of Argument Delimiters in a Command ('Argument Injection') (4.7)**, July 2006. Available from: https://cwe.mitre.org/data/definitions/88.html.

[7] Jason A. Donenfeld. **WireGuard: Next Generation Kernel Network Tunnel**. In *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, 2017. Internet Society. Available from: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/wireguard-next-generation-kernel-network-tunnel/.

[8] Linux Kernel Newbies Editors. **Linux Version 2.6.19 Changelog**, November 2006. Available from: https://kernelnewbies.org/Linux_2_6_19.

[9] Linux Kernel Newbies Editors. **Linux Version 2.6.23 Changelog**, October 2007. Available from: https://kernelnewbies.org/Linux_2_6_23.

[10] Linux Kernel Newbies Editors. **Linux Version 2.6.24 Changelog**, January 2008. Available from: https://kernelnewbies.org/Linux_2_6_24.

[11] Linux Kernel Newbies Editors. **Linux Version 5.6 Changelog**, March 2020. Available from: https://kernelnewbies.org/Linux_5.6.

[12] Free Software Foundation. **ipc_namespaces(7)**, 2021. Available from: https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html.

[13] Free Software Foundation. **mount_namespaces(7)**, 2021. Available from: https://man7.org/linux/man-pages/man7/mount_namespaces.7.html.

[14] Free Software Foundation. **time_namespaces(7)**, 2021. Available from: https://man7.org/linux/man-pages/man7/time_namespaces.7.html.

[15] The OpenBSD Foundation. **OpenSSH 8.9 Release Notes**, February 2022. Available from: https://www.openssh.com/txt/release-8.9.

[16] The OpenBSD Foundation. **pledge(2)**, February 2022. Available from: https://man.openbsd.org/pledge.2.

[17] Serge E. Hallyn. **[PATCH] namespaces: utsname: implement utsname namespaces**, October 2006. Available from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4865ecf1315b450ab3317a745a6678c04d311e40.

[18] Tejun Heo. **[GIT PULL] cgroup namespace support for v4.6-rc1**, March 2016. Available from: https://lore.kernel.org/all/20160318190919.GF20028@mtj.duckdns.org/#r.

[19] Kirill Korotaev. **[PATCH] IPC namespace core**, October 2006. Available from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=25b21cb2f6d69b0475b134e0a3e8e269137270fa.

[20] Cedric Le Goater. **user namespace: add the framework**, July 2007. Available from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=acce292c82d4d82d35553b928df2b0597c3a9c78.

[21] Anil Madhavapeddy. **privsep.c**, July 2003. Available from: https://cvsweb.openbsd.org/cgi-bin/cvsweb/src/usr.sbin/syslogd/privsep.c?rev=1.1&content-type=text/x-cvsweb-markup.

[22] Ram Pai and Alexander Viro. **Shared Subtrees**, November 2005. Added in commit 9cfcceea8f7e8f5554e9c8130e568bcfa98a3a64. Available from: https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt.

[23] Linus Torvalds. **Linux Kernel Version 2.5.2 Changelog**, January 2002. Available from: https://mirrors.edge.kernel.org/pub/linux/kernel/v2.5/ChangeLog-2.5.2.

[24] Linus Torvalds. **Linux 4.6-rc1**, March 2016. Available from: https://lore.kernel.org/all/CA+55aFzBncC+F8TEb5KU1QVwA=PCA89mDp1VLNq008oZq8vpJQ@mail.gmail.com/.

[25] Andrei Vagin. **ns: Introduce Time Namespace**, January 2020. Available from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=769071ac9f20b6a447410c7eaa55d1a5233ef40c.

[26] Alexander Viro. **[PATCH] lazy umount (1/4)**, September 2001. Available from: https://lore.kernel.org/all/Pine.GSO.4.21.0109141427070.11172-100000@weyl.math.psu.edu/.

[27] Alexander Viro. **[PATCH][CFT] per-process namespaces for Linux**, February 2001. Available from: https://lore.kernel.org/all/Pine.GSO.4.21.0102242253460.24312-100000@weyl.math.psu.edu/.

[28] Alexander Viro. **[RFC] shared subtrees**, January 2005. Available from: https://lore.kernel.org/all/20050113221851.GI26051@parcelfarce.linux.theplanet.co.uk/.

[29] Inc. VMware. **Understanding Full Virtualization, Paravirtualization and Hardware Assist**. Technical report, March 2008. Available from: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf.

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