Void Processes: Minimising privilege by default

Jake Hillion
University of Cambridge
United Kingdom
jake.hillion@cl.cam.ac.uk

Abstract

Operating systems are providing more facilities for process isolation than ever before, realised in technologies such as Docker containers [21] and systemd slices [5]. These systems separate the design of the program from the systems that create privilege separation. Void Processes take these techniques to the extreme, removing access to everything but syscalls from a process by default. This work focuses on adding back slivers of privilege to achieve functional applications with minimal privilege.

I present a summary of the privilege separation features in modern Linux, the system design of void processes, and an evaluation on a series of example applications.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Keywords: datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Jake Hillion. 2022. Void Processes: Minimising privilege by default. *J. ACM* 37, 4, Article 111 (August 2022), 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Void processes take advantage of modern Linux namespaces to run applications with minimal exposure to the system itself. Void processes use a mixture of Linux namespaces and file descriptor based capabilities to allow running purpose-built applications without expecting the support of the standard Linux system. During the process of building such a system, gaps in the kernel were exposed. Namespaces were intended to emulate an ordinary Linux system, rather than creating 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 the learnings of what features are missing in the user-space kernel APIs to succeed in creating processes this way.

This work explores the question of what is an operating system by taking a novel approach to running applications with the system exposed in a very different way. Rather than limiting the access of a process or set of processes to the operating system, such as in containers, we instead limit the access to the operating system with more explicit methods per process. Interaction between processes is allowed by specifying such interaction statically at compile time, removing any separation between the application developer and the system controlling access to the application, unlike solutions such as SELinux [20].

Author's address: Jake Hillion, University of Cambridge, United Kingdom, jake.hillion@cl.cam.ac.uk.

0004-5411/2022/8-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

2 Background

2.1 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.

2.1.1 Copy-on-Write. Comparing to network namespaces, a slightly more modern namespace [Table 1], 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.

Table 1. Table showing the date and kernel version each namespace was added. The date provides the first commit where they appeared date of creation, 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 Version		CVEs
`mount`	24th February 2001	[28]	2.5.2	[24]	2020-29373
`ipc`	2nd October 2006	[18]	2.6.19	[1]	
`uts`	2nd October 2006	[16]	2.6.19	[1]	
`user`	15th July 2007	[19]	2.6.23	[2]	2021-21284
`network`	10th October 2007	[8]	2.6.24	[3]	2011-2189
`pid`	2nd October 2006	[7]	2.6.24	[3]	2019-20794
`cgroup`	18th March 2016	[17]	4.6	[25]	2022-0492
`time`	12th November 2019	[26]	5.6	[4]	

Unmount shows all files demo (maybe open /etc/-passwd).

2.1.2 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.

Shared subtrees [22] were introduced to provide a consistent view of the unified hierarchy between namespaces. Consider the example in Figure 1. `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 [12]:

"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. Further, it means that mounts and unmounts are propagated out of the namespace. This can be highly confusing behaviour, and `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 new unshared tree. The reasoning for this is that containers created should not present the behaviour given in Figure 1, and this behaviour is unavoidable unless the parent mounts are shared, while it is possible to disable the behaviour where necessary.

2.1.3 Lazy unmounting. Mount namespaces present further interesting behaviour when unmounting the initial root filesystem. Although this may initially seem isolated to void processes, it is also a problem in a container type system. Consider again the container created in Figure 1 - the existing root must be unmounted after pivoting, to avoid keeping the container 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 [9]. 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 [10, §7.3].

Something which behaves differently is the memory mapping of a currently running process's binary. Consider the example in Listing 1, 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. While this initially seems like a good solution, this syscall is incredibly dangerous when combined with shared subtrees. This behaviour is shown in Figure 2, where a lazy (and hence recursive) unmount is combined with a shared subtree to disastrous effect.

```
# 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 1. Highly separated behaviour without shared subtrees between mount namespaces.

```
# 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 2. Behaviour when attempting to unmount / from an unshared shell with a shared mount.

Listing 1. 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
```

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) [27]:

"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 [27], as mounts were not initially shared but duplicated between namespaces. However, when shared subtrees were added in January 2005 [29], 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, in contradiction to 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.

3 System Design

An example of running a void process application is given in Figure 3. 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

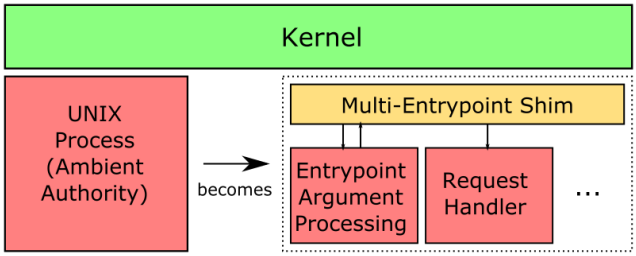

Figure 3. Interaction between the application and the environment.

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.

3.1 Building the Void

Preparing a void process takes advantage of the namespaces feature in Linux. However, many of the namespaces are not designed for this purpose, so this is a more difficult prospect than one might hope. Details of when each namespace was added and some of the relevant features are given in Table 1.

3.1.1 Mount namespaces. Mount namespaces were the first [Table 1] namespaces introduced to Linux, in kernel version 2.5.2 [24]. In contrast to network namespaces, 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 (more details in §2.1.2), 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.

3.1.2 IPC namespaces. Creating a void process with IPC namespaces is pleasantly easy in comparison. From the manual page [11]:

"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, particularly because it is not copy-on-write. IPC objects are visible within a namespace if and only if they are created within that namespace. Therefore, a new namespace is an entirely empty void.

3.1.3 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. Unlike IPC namespaces, UTS namespaces are copy-on-write. That is, the value of each of these in the parent namespace is the same in the child.

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.

3.1.4 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.

3.1.5 Network namespaces. Network namespaces were added in kernel version 2.6.24 [3], some time after the initial namespace boom. They present the optimal namespace for

creating a void. Creating a new network namespace immediately creates an entirely empty namespace. That is, the new network namespace has no link whatsoever to the creating network namespace. To add a link, one can create a virtual Ethernet pair, with one adapter in each namespace [RN]. Alternatively, one can create a Wireguard adapter with sending and receiving sockets in one namespace and the VPN adapter in another [10, §7.3]. This allows for very high levels of separation while still maintaining access to the primary resource - the Internet or wider network.

3.1.6 PID namespaces. pid namespaces add 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.

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 2.

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/fork`, 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 combines the two into a single syscall.

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, or freshly mounted.

3.1.7 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:

Listing 2. 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
```

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.

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.

3.1.8 time namespaces. Time namespaces are the final namespace added at the time of writing, added in kernel version 5.6 [4]. The motivation for adding time namespaces is given in the manual page [13]:

"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 it in one specific 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 `CLONE_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 debatable and the quantity of bespoke syscalls would slow down the application. Time namespaces are thus avoided in this implementation.

3.2 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.

3.2.1 Files and directories. 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.

3.2.2 Networking. 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 [14]. 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.

Listing 3. An application that requires only stdout and stderr.

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

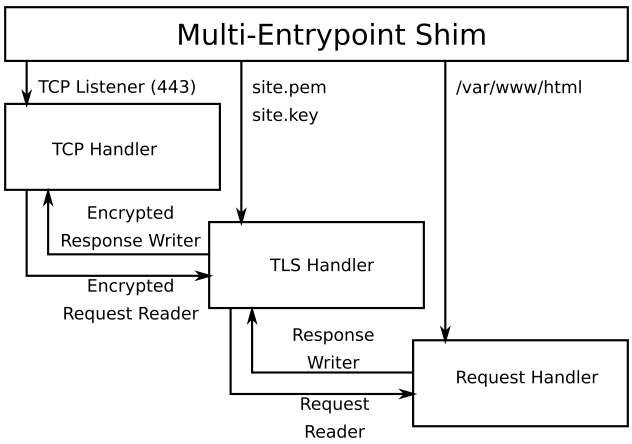

Figure 4. Process separation in a TLS server.

4 Example Applications

4.1 No Permissions

The cornerstone of strong process separation is an application that is completely deprivileged. Listing 3 shows an application which, when run under the shim, drops all privileges except `stdout`. This is easy to achieve under the shim.

4.2 gzip

GNU gzip [15] 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 [31].

As C does not have high-level language features for multi-entrypoint applications, 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.

4.3 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 4. 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.

5 Evaluation

Write evaluation

6 Related Work

6.1 Virtual Machines and Containers

Virtual Machine solutions [6, 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 [21], 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 [23].

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.

6.2 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.

6.3 Capsicum

Capsicum [31] extends UNIX file descriptors in FreeBSD to reflect the rights on the object they hold. These capabilities may be shared between processes as other file descriptors. The goals of both software are the same: make privilege separated software better. However, we take quite different approaches. Multi-entrypoint applications focus on building a static definition really close to the code, while Capsicum allows processes to dynamically privilege separate. This allows applying static analysis to the policies, while also keeping the definition close to the code.

7 Future Work

7.1 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.

8 Conclusion

Write conclusion

Acknowledgments

Write acknowledgements

References

[1] 2006. Linux Version 2.6.19 Changelog. https://kernelnewbies.org/Linux_2_6_19

[2] 2007. Linux Version 2.6.23 Changelog. https://kernelnewbies.org/Linux_2_6_23

[3] 2008. Linux Version 2.6.24 Changelog. https://kernelnewbies.org/Linux_2_6_24

[4] 2020. Linux Version 5.6 Changelog. https://kernelnewbies.org/Linux_5.6

[5] The Systemd Authors. 2022. systemd.slice. https://www.freedesktop.org/software/systemd/man/systemd.slice.html

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review* 37, 5 (Oct. 2003), 164–177. https://doi.org/10.1145/1165389.945462

[7] Sukadev Bhattiprolu. 2006. [PATCH] Define struct pspace. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=3fbc96486459324e182717b03c50c90c880be6ec

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

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

[10] Jason A. Donenfeld. 2017. WireGuard: Next Generation Kernel Network Tunnel. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2017.23160

[11] Free Software Foundation. 2021. ipc_namespaces(7). https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html

[12] Free Software Foundation. 2021. mount_namespaces(7). https://man7.org/linux/man-pages/man7/mount_namespaces.7.html

[13] Free Software Foundation. 2021. time_namespaces(7). https://man7.org/linux/man-pages/man7/time_namespaces.7.html

[14] The OpenBSD Foundation. 2022. pledge(2). https://man.openbsd.org/pledge.2

[15] Jean-loup Gailly. 2020. Gzip. https://www.gnu.org/software/gzip/

[16] Serge E. Hallyn. 2006. [PATCH] namespaces: utsname: implement utsname namespaces. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4865ecf1315b450ab3317a745a6678c04d311e40

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

[18] Kirill Korotaev. 2006. [PATCH] IPC namespace core. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=25b21cb2f6d69b0475b134e0a3e8e269137270fa

[19] Cedric Le Goater. 2007. user namespace: add the framework. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=acce292c82d4d82d35553b928df2b0597c3a9c78

[20] Pete Loscocco. 2000. Security-enhanced Linux available at NSA site. https://lore.kernel.org/all/200012221402.JAA11421@coalstack.epoch.ncsc.mil/

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

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

[23] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. 2007. 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)*. Association for Computing Machinery, New York, NY, USA, 275–287. https://doi.org/10.1145/1272996.1273025

[24] Linus Torvalds. 2002. Linux Kernel Version 2.5.2 Changelog. https://mirrors.edge.kernel.org/pub/linux/kernel/v2.5/ChangeLog-2.5.2

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

[26] Andrei Vagin. 2020. ns: Introduce Time Namespace. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=769071ac9f20b6a447410c7eaa55d1a5233ef40c

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

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

[29] Alexander Viro. 2005. [RFC] shared subtrees. https://lore.kernel.org/all/20050113221851.GI26051@parcelfarce.linux.theplanet.co.uk/

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

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