# Multi-Entrypoint Applications: allowing the binary to self-regulate an application deployment with multiple levels of privilege separation

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

## Abstract

Operating systems are providing more facilities for process isolation than ever before, realised in technologies such as Containers [CN] and systemd slices [CN]. These systems separate the design of the program from the systems that create privilege separation.

Multi-Entrypoint Applications bring the privilege separation back into the program itself. By using a trusted shim and binfmt_misc [CN], an application started with minimal privileges can achieve full separation. High-level language features provide an easy interface to privilege separation.

I present a summary of the privilege separation features in modern Linux, the system design of multi-entrypoint applications, the language front-ends to support it, 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. Multi-Entrypoint Applications: allowing the binary to self-regulate an application deployment with multiple levels of privilege separation. *J. ACM* 37, 4, Article 111 (August 2022), 4 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 Introduction

TODO

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


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

## 2 Motivation

TODO

### 2.1 Threat Model

I present a threat model in which application binaries are trusted absolutely. That is, the software provider had no ill intent, and once the binary is on disk, it will not change without permission. This means that one can trust the binary to set up its own security, as it is protecting not against malice by its own developers, but instead bugs in the software.

## 3 Background

### 3.1 File Descriptor Passing

File descriptor passing is the act of handing a file descriptor (in this case a capability) to a foreign process. As the memory representation of a file descriptor on Linux is a simple number, passing it to a different process does not immediately confer access to that file descriptor. To pass the file descriptor in a meaningful way, there must be some kernel intervention.

Modern Linux provides multiple methods for file descriptor passing. Three will be studied here: `fork`, `CLONE_FILES`, and `sendmsg`. When a new process is forked, the file-descriptor table of the forking process is duplicated and given to the new child process. Altering these tables in each process, for example by opening a new file, does not alter the table in the other process. This is known as copy-on-write. This behaviour can be altered by calling `clone(2)` with the `CLONE_FILES` flag. This keeps the tables in sync, rather than allowing them to diverge. Finally, file descriptors can be sent as ancillary data in a `sendmsg` call. This requests that the kernel create a new file descriptor in the foreign table when `recvmsg` is called, allowing for more targeted file-descriptor transfers.

As file-descriptor passing is fundamental to this application, understanding the performance of these techniques was highly important. Each of send and receive are studied, as the asynchronous protocols allow for these to differ in performance. That is, the send may be cheap, but require more work to be done when the file descriptor is received. The results of this study are given in Figure [FN].

TODO: Complete performance study.

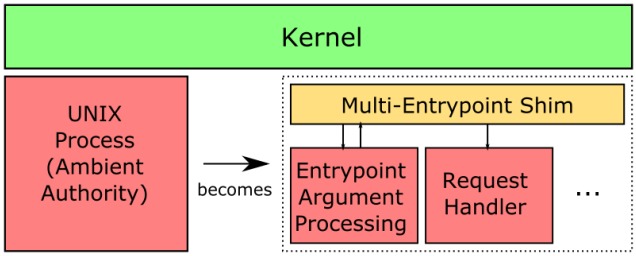


**Figure 1.** Interaction between the application and the environment.

## 4 System Design

An example of running a multi-entrypoint application is given in Figure 1. What was originally a monolithic application becomes a set of applications that communicate with a new shim. The shim does not replace the kernel, and instead supplements it with new higher-level abilities. Each entrypoint receives input from the shim, and may be able to return data if appropriate. Most of this data is in the form of file descriptors, which are treated as capabilities in this system.

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

The shim takes advantage of high levels of privilege to be able to more effectively deprivilege an application than an application with ambient authority could. For example, creating a new network namespace requires `CAP_SYS_ADMIN`, which would give many applications more privilege than they require. By deferring to a shim with extra privileges, this trusted code can be written only once, and avoid conferring more privileges than otherwise required.

## 5 Language Frontends

The language frontends are an extremely important part of this project, closing the gap between a static privilege separation solution like SELinux [CN] and a dynamic one like Capsicum [5]. I have implemented a language frontend in Rust and will describe it in this section.

### 5.1 Rust

The Rust frontend uses macros to wrap functions with high-level primitives into multi-entrypoint compatible entrypoints. Further, it allows calling these functions using the new interface via the shim. Consider the example in Listing 1.

Firstly, the encrypt entrypoint is created. This is a regular Rust function which takes two high-level File objects, a wrapped file descriptor. The entrypoint macro wraps this function, providing in its place an `extern "C"` function that

**Listing 1.** A sample application using the Rust language frontend.

```
#[entrypoint]
fn encrypt(mut in: File, mut out: File)

#[entrypoint]
fn main() {
  let input_file = ...;
  let output_file = ...;

  encrypt(input_file, output_file);
}
```

**Listing 2.** An application that requires only stdout and stderr.

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

is unmangled and takes argc/argv. This allows functions with high-level arguments to be used as normal, with the argument parsing abstracted away by the library.

Second is the ordinary main function for the application. This is also tagged as an entrypoint, allowing the library to help out with more calls. The example given here is that of the encrypt method, which uses the API seen above. The use of macros here allows the call to encrypt to remain type safe, even though the call must pass through an external interface (the shim itself).

A significant benefit to this approach is the ease of disabling the multi-entrypoint application. By turning the entrypoint macro into identity with a crate feature, the code is compiled without the aid of the multi-entrypoint shim. This allows for significantly easier debugging, as the application follows a single execution path, rather than needing to be debugged as a distributed application.

## 6 Example Applications

### 6.1 No Permissions

The cornerstone of strong process separation is an application that is completely deprivileged. Listing 2 shows an application which, when run under the shim, drops all privileges except `stdout`. This is easy to achieve under the shim.

### 6.2 gzip

GNU gzip [2] 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 [5].

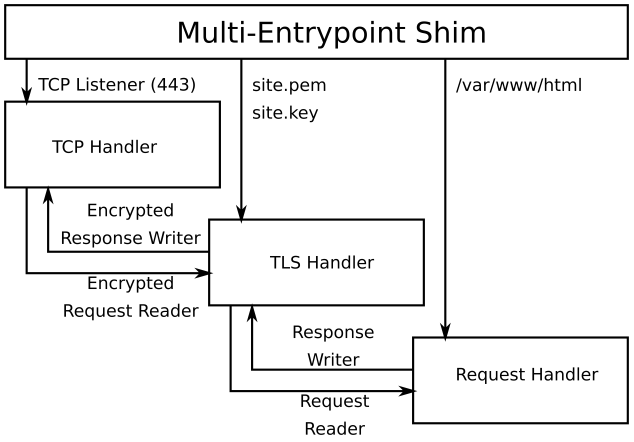


**Figure 2.** Process separation in a TLS server.

As C is not an adapted language 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 permissions to be able to handle this permissioning, as the required permissions are provided by the shim.

### 6.3 TLS Server

Finally, a rudimentary TLS server is created to show the rich privilege separation abilities of multi-entrypoint applications. An example structure is shown in Figure 2. 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.

## 7 Evaluation

TODO

## 8 Related Work

### 8.1 Virtual Machines and Containers

Virtual Machine solutions [1, 4] 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 [CN], 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 [3].

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

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

### 8.2 systemd

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

### 8.3 Capsicum

Capsicum [5] 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 (§3.1). 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.

## 9 Future Work

### 9.1 Dynamic Linking

TODO

## 10 Conclusion

TODO

## Acknowledgments

TODO: acknowledgements

## References

[1] 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

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

[3] 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

[4] 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

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