University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Krzysztof Małysa
Student no. 394442

Multi-process sandbox for unprivileged users on Linux

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz

Warsaw, September 2022

Abstract

TODO

Keywords

sandboxing, security, container, Linux, capabilities, cgroups, user namespace, PID namespace, mount namespace, secure execution, arbitrary code execution, rlimit, seccomp, ptrace

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

10011007.10010940.10010941.10010949 — Software and its engineering – Software organization and properties – Contextual software domains – Operating systems

Tytuł pracy w języku polskim

Sandbox wielu procesów dla nieuprzywilejowanych użytkowników systemu Linux

Contents

Chapter 1

Introduction

The objective of this project is to create safe and efficient sandbox to execute short running untrusted programs, as well as complex programs e.g. compiler of a C++ program. All done with robust isolation and minimal overhead (the order of milliseconds).

The primary use case is an online judge whose job is to

- compile user-provided source code
- run it several times (even hundreds) with different inputs
- verify the output correctness usually by running another program

All of these tasks require at least ensuring that

- real time is limited
- CPU time is limited
- memory is limited
- disk space is limited
- file access is isolated
- network is isolated or disabled

Also statistics about the executed program are required

- real time used
- CPU time used
- peek memory usage

All of that is to be done as an unprivileged user.

Such combination (especially minimal overhead and recording the peek memory usage) is very uncommon e.g. Firejail (TODO: reference) does not provide memory statistics. LXC (TODO: reference) and Docker require privileges to create a container. Thereby, the listed constraints require a new solution.

Chapter 2

Useful Linux kernel mechanisms

TODO

2.1. User namespaces

TODO

2.2. PID namespaces

TODO

2.3. Mount namespaces

Mount namespaces allow for isolation of mounts i.e. process in one namespace can modify its mount list without affecting others' mount lists, or affecting or being affected by others in a controlled manner thanks to the Shared Subtrees feature of the Linux kernel [1].

2.3.1. Terminology

The most typical use case of `mount` is mounting a filesystem at some location in a filesystem e.g. mounting home directory: `mount /dev/sda2 /home/user` or mounting the temporary filesystem at `/tmp`: `mount -t tmpfs tmpfs /tmp`. Filesystem can be mounted at multiple locations e.g. `mount /dev/sda2 /a && mount /dev/sda2 /b`. Such location is called a **mount point**. As it will be explained later, a single **mount** has a single mount points. But a single `mount` operation may result in more than one mounts.

List of all mounts of a mount namespace of a process with PID `[pid]` can be examined via file `/proc/[pid]/mountinfo`.

Mount is a result of a `mount` operation and is a filesystem that is accessible at a specified location called a `mount point`.

Mount point is a location where mount in attached.

Propagation type affects how mounts that happen directly under that mount are propagated to other members of the `peer group` and its slave peer groups. It can be one of:

- `shared` Its peer group can have any size and mount events propagate to other members and from other members of the peer group.
- `slave` Its peer group has only one member — itself and has a master peer group. Mount events propagate from the master peer group, but not to the master peer group.
- `slave & shared` Its peer group can have any size and has a master peer group. Mount events propagate between members of the slave & shared peer group but not to the master peer group. Mount events from the master peer group propagate to all members of the slave & shared peer group.
- `private` Its peer group has only one member — itself. No mount events propagate from this peer group to another and vice versa.
- `unbindable` Same as `private`, but bind mounts with source inside this mount are forbidden.

Peer group is a group of mounts that propagate mounts between one another.

These notions are best illustrated in an example. First we mount `tmpfs` at `/mnt` and make its propagation type `shared`, later we examine the mount list after this operation.

```
# mount -t tmpfs tmpfs /mnt --make-shared
# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
619 27 0:69 / /mnt rw,relatime shared:274
```

Now we create a `/tmp/mnt` and bind mount there the `/mnt`.

```
# mkdir /tmp/mnt
# mount --bind /mnt /tmp/mnt
# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
619 27 0:69 / /mnt rw,relatime shared:274
773 39 0:69 / /tmp/mnt rw,relatime shared:274
```

We see that both of these mounts have `shared:274` — it means that the mount has propagation type `shared` and `274` is the id of the peer group. So both mounts are in the same peer group. Apart from the fact that these mount points have the same filesystem underneath (because of the bind mount):

```
# ls /mnt
# ls /tmp/mnt
# touch /mnt/a
# touch /tmp/mnt/b
# ls /mnt
a  b
# ls /tmp/mnt
a  b
```

They also propagate mount events between them (because of the propagation type `shared`):

```
# mkdir /mnt/c
# mount -t tmpfs tmpfs /mnt/c
# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
619 27 0:69 / /mnt rw,relatime shared:274
773 39 0:69 / /tmp/mnt rw,relatime shared:274
794 619 0:71 / /mnt/c rw,relatime shared:415
795 773 0:71 / /tmp/mnt/c rw,relatime shared:415
```

We can see that mount at `/mnt/c` propagated to `/tmp/mnt` as `/tmp/mnt/c`.

E.g. with a private propagation type mounts are not propagated.

```
# mount -t tmpfs tmpfs /mnt --make-private
# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
619 27 0:69 / /mnt rw,relatime
```

```
# mkdir /tmp/mnt
# mount --bind /mnt /tmp/mnt
# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
619 27 0:69 / /mnt rw,relatime
773 39 0:69 / /tmp/mnt rw,relatime shared:274
```

```
# ls /mnt
# ls /tmp/mnt
# touch /mnt/a
# touch /tmp/mnt/b
# ls /mnt
a  b
# ls /tmp/mnt
a  b
```

```
# mkdir /mnt/c
# mount -t tmpfs tmpfs /mnt/c
# cat /proc/self/mountinfo | grep '/mnt' | sed 's/ - .*//'
619 27 0:69 / /mnt rw,relatime
773 39 0:69 / /tmp/mnt rw,relatime shared:274
794 619 0:71 / /mnt/c rw,relatime
```

`slave` propagation type allows for propagation only in one direction, from the master peer group to the slave peer group.

More details about all propagation types and the semantics of all `mount` operations are be described in the following subsections.

2.3.2. Semantics

TODO

2.4. cgroups

TODO

2.5. cgroup namespaces

TODO

2.6. Capabilities

TODO

2.7. ptrace

TODO

2.8. seccomp

TODO

Chapter 3

Sandbox design

Sandbox is spawned as a separate process and this process executes sandboxing requests e.g. execute program A with configuration B. Communication between the caller and the sandbox server process uses UNIX domain socket. Errors regarding handling a specific request are reported through the UNIX socket as a response to the sandbox request. A separate anonymous file (created using `memfd_create()`) is used for reporting fatal errors of the sandbox server process - it fills the file with an error description and dies afterwards. Such separation allows for a simpler protocol to be used for communicating through the UNIX socket e.g. reporting errors about writing to the socket are reported using the anonymous file instead of the socket itself. Figure 3.1 illustrates the design.

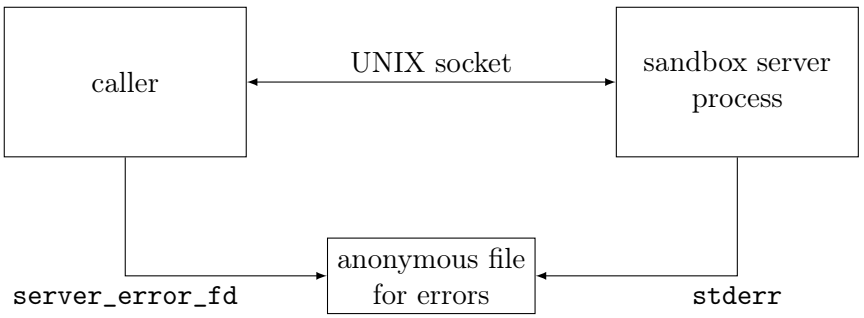

Figure 3.1: Caller requests and receives results of executing untrusted programs through UNIX socket. Sandbox server process dies on error leaving the error message for the caller in an anonymous file.

Sandbox needs to execute an untrusted executable. To do this it needs to `fork()` a child process and call `execve()` in the child process. Our use case involves executing short-running programs frequently. `fork()` syscall may take a long time [2] - the bigger RSS (resident set size - RAM pages that are actually in use) the longer time `fork()` needs. To reduce `fork()` latency, the caller spawns sandbox server process that executes a separate executable – containing only the sandbox, therefore reducing the RSS to the minimum and speeding up `fork()`. Additional benefits of this approach are setting up all common work before running executing the untrusted executable once i.e. when the sandbox server starts e.g. closing stray file descriptors not marked with `O_CLOEXEC` flag and setting up cgroups. The only overhead is

passing data and file descriptors through the UNIX socket – from caller to the sandbox server process and back.

Bibliography

[1] Ram Pai linuxram@us.ibm.com. *Shared Subtrees*. URL: `https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt` (visited on 09/09/2022).

[2] Redis Ltd. *Diagnosing latency issues: Latency generated by fork*. URL: `https://redis.io/docs/reference/optimization/latency/#latency-generated-by-fork` (visited on 09/08/2022).