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

We introduce a new sandbox for unprivileged Linux users that requires no kernel modifications. It takes advantage of several Linux mechanisms used elsewhere — cgroups, namespaces, ptrace and seccomp among others. The sandbox was optimized to run dozens of untrusted programs in a sequence with minimal overhead while preserving the safety. It is capable of running both multi-threaded and multi-process programs. It is able to record the peek memory usage and CPU execution time of multithreaded program, alas these statistics are unavailable for multi-process programs. We describe the encountered limitation and challenges around enforcing safety and collecting statistics. Further, we examine its ability to run complex multi-process programs like C++ compiler and the overhead when running a series of short-running programs.

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

Security and privacy – Systems security – Operating systems security

Tytuł pracy w języku polskim

Sandbox wielu procesów dla nieuprzywilejowanych użytkowników systemu Linux

Contents

Chapter 1

Introduction

TODO

1.1. Assumptions

The primary assumption is that the validation and enforcement during the interaction of an untrusted program and the Linux kernel is enough to prevent the program from doing anything unwanted. Thus satinization of all syscalls either directly (using `ptrace` and `seccomp`) or indirectly (using capabilities, cgroups, namespaces e.g. mount namespaces, and limits e.g. `prlimit()`). Thereby executing the program while no syscalls are invoked is considered safe. In general however, this is not true, e.g. `io_uring` can be used to "call syscalls" without actually executing any syscall [1, 2, 3, 4]. Another example is writing to memory mapped file just by writing to the memory region where the file is mapped to. Such dangerous situations need to be prevented by properly forbidding or sanitizing syscalls that are necessary for such situations to happen i.e. preventing `io_uring` syscalls and forbidding mapping of the unwanted files.

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 [5].

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

3.1. Overview

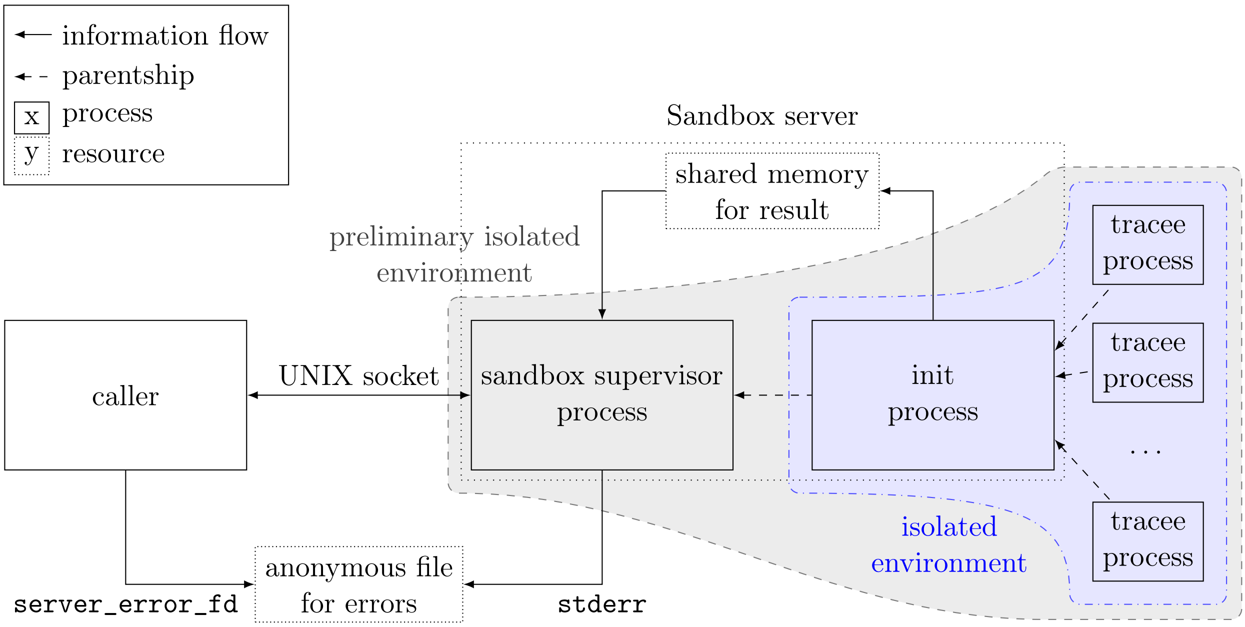

Figure 3.1: Caller requests and receives results of executing untrusted programs through UNIX socket. Sandbox server dies on error leaving the error message for the caller in an anonymous file. Sandbox server consist of the supervisor process and its child — the init process that is spawned for each request. Init process performs role of the init process in the PID namespace of tracee processes. Init process passes errors and results to the supervisor process using shared memory. To reduce overhead of setting up the isolated environment for each request, common work is done only once to create the preliminary isolated in the supervisor process.

Sandbox works as a set of separate processes. Caller spawns the sandbox server, which consists of the supervisor process that for each request spawns init process that manages tracee processes of the request. A request consist of an isolated environment configuration along with the program to execute and its arguments. A separate init process is required by PID namespaces. To minimize the attack surface in case of a compromise, it is better to separate the init process from the supervisor process. Figure 3.1 illustrates this separation.

For each request, an isolated environment needs to be prepared for the executed program. Preliminary isolated environment that is managed by sandbox supervisor process is there to reduce overhead of preparing the isolated environment during handling of the subsequent requests. Sandbox is designed for handling many subsequent requests, and setting up the isolated environment again and again requires repeating the same steps. Steps that can be done once for all requests without a security or an isolation trade-off are performed once and make up the preliminary isolated environment. One of such steps is creating a cgroup hierarchy for the init and tracee processes.

Communication is performed differently in different places. The caller sends requests and receives results via UNIX socket connected with the sandbox supervisor process. Fatal errors in the sandbox server are reported to the caller using an anonymous file for errors. This could be done over the UNIX socket but, to simplify the communication protocol it is separated. This way we do not have to deal with problematic cases like reporting error about problems with sending data over the socket — the error would have to be reported via the problematic socket. Init process reports request results to supervisor using shared memory mapping.

The init process is the parent of the main tracee process and orphaned tracee processes as a property of PID namespaces [7]. Sandbox supervisor process is the parent of the init process. Although the caller is the parent process of the sandbox supervisor process, sandbox server is implemented without this assumption. Instead of dying upon the caller process death, the sandbox supervisor watches the UNIX socket for a other-end-closure event of the connection. When this happens it exits immediately. Init process is configured to be killed as soon as the supervisor becomes dead, and trace processes will be killed by the Linux kernel when the init process dies or is killed. Therefore the sandbox server is not bound to the caller process but the UNIX socket instead.

3.2. TODO

3.3. Caller

3.4. Sandbox server

THIS SECTION NEEDS AND WILL BE REWORED

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.

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 [6] - 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.

3.5. Limiting the number of processes and threads

TODO: cgroup

3.6. Isolating filesystem

TODO

3.7. Limiting memory

TODO: prlimit + cgroup

3.8. Limiting execution time

TODO

3.9. Collecting statistics

3.9.1. Execution real time

TODO

3.9.2. Execution CPU time

TODO

3.9.3. Peek memory usage

TODO

TODO NOTES:

It seems that the PID namespace init process cannot exit unless all the processes are dead and *waited* i.e. exit_group() blocks. Here it only helps if we kill the parent. The parent of pid2 is not pid which it may seem a bit strange from inside the new namespace as there are two roots in the process hierarchy. Also init inherits orphaned children of processes inside the namespace e.g. of process pid2. So that orphaned children don't have to be adopted by an ancestor process.

RLIMIT_CPU: limit is inherited and preserved across execve but is not shared with child processes i.e. process can spawn a child that uses up limit of X seconds, and later spawn another child that now has X seconds of cpu time, not 0 seconds.

Chapter 4

Evaluation

Bibliography

[1] Jens Axboe. *Efficient IO with io_uring*. Oct. 15, 2019. URL: https://kernel.dk/io_uring.pdf (visited on 11/14/2022).

[2] Jens Axboe. *Kernel Recipes 2019 – Faster IO through io_uring*. Sept. 27, 2019. URL: https://kernel-recipes.org/en/2019/talks/faster-io-through-io_uring/ (visited on 11/14/2022).

[3] Jonathan Corbet. *Ringing in a new asynchronous I/O API*. Jan. 15, 2019. URL: https://lwn.net/Articles/776703/ (visited on 09/10/2022).

[4] Jonathan Corbet. *The rapid growth of io_uring*. Jan. 24, 2020. URL: https://lwn.net/Articles/810414/ (visited on 11/14/2022).

[5] Ram Pai linuxram@us.ibm.com. *Shared Subtrees*. Nov. 7, 2005. URL: https://www.kernel.org/doc/Documentation/filesystems/sharedsubtree.txt (visited on 09/09/2022).

[6] Redis Ltd. *Diagnosing latency issues: Latency generated by fork*. Sept. 8, 2011. URL: https://redis.io/docs/reference/optimization/latency/#latency-generated-by-fork (visited on 09/08/2022).

[7] man-pages project. *pid_namespaces(7)*. URL: https://man7.org/linux/man-pages/man7/pid_namespaces.7.html (visited on 11/13/2022).