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

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.

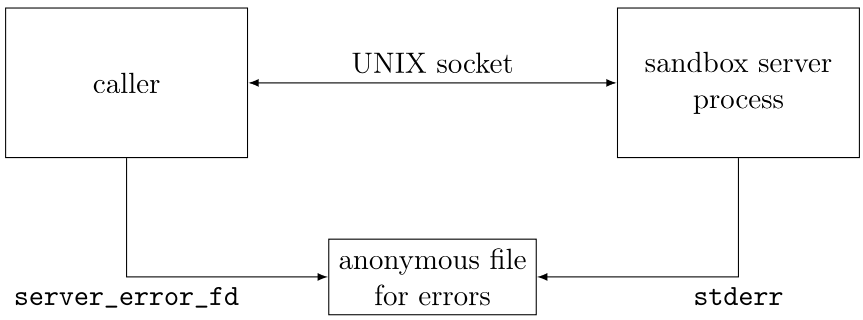

Figure 3.1: Caller requests and receives results of executing untrusted programs through UNIX socket. Sandbox server process dies on error leaving the error message for the caller in an anonymous file.

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 [1] - 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] 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).