University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Krzysztof Małysa
Student no. 394442

Multi-process sandbox for unprivileged users on Linux

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz
Institute of Informatics

Warsaw, December 2022

Abstract

TODO

Keywords

sandboxing, security, Linux, secure execution, arbitrary code execution, judging system,

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

1.1. Background

Secure execution environments are commonplace these days, from containers and virtual machines on servers to sandboxes on laptop and smartphones — most of which run on Linux. They are used to securely execute untrusted code, as well as trusted programs to prevent damage escalation in the event of unknown vulnerabilities. Their key features are isolation, limiting resource usage, and accounting for resource consumption.

The features of Linux allow the creation of simple yet effective and efficient secure environments. They work at application runtime, so in most cases existing software does not need to be adapted to use them. This makes them easily applicable, and explains why their adoption is growing.

In this thesis, the most important application of sandboxing are online judge systems. Online judge systems have beneficial role in programming education and competitive programming. They allow testing user-provided solution to a specific problem. The solution is run on a predefined test cases in order to check if it is valid. In such platforms isolating the compilation and running of the solution is essential to provide security and robustness of the platform itself.

Historically, isolation techniques evolved together with the online judge platforms. The most primitive (yet insecure) was usage of `chroot(2)` [28] to restrict access to part of the filesystem. To increase isolation virtual machines were used [2]. Later, containerization became a new way to provide isolation [19, 38].

Online education platforms greatly facilitate teaching and learning programming. They provide quick feedback on the correctness of the code the user submits. They are used in schools and universities and provide great learning opportunities for all.

Moreover, a versatile sandbox has applications outside online judging platforms. For example, it can be used to sanitize compiling a PDF form LaTeX sources or for safe execution of untrusted server-side scripts in web applications.

1.2. Goal of the thesis

The goal of the thesis is to design, implement and integrate a new sandbox for the Sim project [15]. The Sim project is an online platform for preparing people for and carrying out algorithmic contests. The project started in 2012 and is developed by me since the beginning. It is used at the XIII High School in Szczecin and programming camps to teach young people programming and algorithms. It has an online judge with a sandbox specially developed for this use case. Over the years the sandbox became a limitation. It only allows running a single-threaded statically linked executable of programs written in C, C++ or Pascal. The new sandbox will allow supporting more programming languages and improve security of the solution compilation stage.

1.2.1. Requirements

The new sandbox needs to be optimized for running short-running programs as well as have minimal runtime overhead. Most of the test cases the solution is run on are small and solution completes them in less than 10ms. The goal is to allow hundreds of such sort-running runs per second, hence optimizing for short-running programs is important. However, minimizing overhead of the sandbox during the run is also important i.e. if the program runs X ms normally, the objective is that the program inside the new sandbox will also run approximately X ms.

The new sandbox needs to be versatile. It will be used to secure the compilation of the solutions as well as running of the solutions. Compilation is a complicated process that involves parsing, translating, optimizing and linking the final program. For languages like C, C++ and Rust it involves running several executables in coordination e.g. compiler and linker i.e. more than one process at a moment — the sandbox needs to support that.

Sandboxing needs to have a low overhead. Apart from small test where solution runs quickly (a matter of milliseconds), almost always the solution is run on big test cases, where it may need several seconds for it to solve the problem. Increasing this time as little as possible while the solution is running inside the sandbox is one of the primary objectives.

It often requires running several executables e.g. compiler and linker, so allowing a single process inside sandbox is not enough. Sandboxing solution is simpler, because it is a single process. But since it is often short-running, the overhead needs to be minimal.

The sandbox needs to allow limiting resources. Real time, CPU time, memory – these need to be limited not only for the robustness of the platform, but specific problems require different limits. The goal of some problems is to solve it with very restricted memory e.g. find a missing integer in a random permutation of integers $1, \ldots, n$ without one element, but in constant memory.

The sandbox needs to account resource usage. For every test, the user is presented with consumed memory and CPU time by their solution. The sandbox needs to provide this information.

The last requirement is the sandbox will not require any privileges. There is a tool called Sip [16] for preparing the problem packages for the Sim platform. One of the purposes of the tool is to run the solutions inside the same secure environment as on the Sim platform. The user should not need any privileges to run this tool, so the sandbox should not require them either.

1.2.2. Existing solutions

Approaches to form a secure execution environments differ. One of them is virtualization or emulation e.g. QEMU [26] and KVM [25], VirtualBox [27], VMWare Workstation [42]. Although powerful and effective, they come with an enormous overhead i.e. booting up an entire operating system. Moreover, emulation noticeably slows down the runtime of an emulated application, rendering such solutions inapplicable.

Containers provide much lower overhead: setup of an order of milliseconds and negligible runtime overhead. But, Docker [21], LXC [1] require root privileges to create a container.

systemd-nspawn [40] requires root privileges to run.

Rootless are containers [36] that can be created and run by an unprivileged user are the almost perfect solution to the problem. They provide almost all of the functionality of the normal containers but without the need to engage a privileged user. However, they often use setuid binaries and that is undesirable [37]. Also they are not optimized to run sequences of short-running programs. In this thesis we will create a sandbox that uses the same techniques as rootless containers but will be optimized for running sequences of short-running programs.

1.3. Structure of the Thesis

Chapter 2 contains overview of sandboxing techniques and existing implementations and comparative analysis of them. Details of design and architecture are described in chapter 3. Implementation is described in 4. Chapter 5 contains performance evaluation of the final implementation and impact of some optimizations. Later, in chapter 6 use cases and applications are discussed. Chapter 7 details integration with online judge and challenges involved. In chapter 8 future work and opportunities are discussed. Finally, chapter 9 contains the conclusion.

Chapter 2

Literature overview

2.1. Overview of Sandboxing Techniques

During the first programming competitions, the human judges manually read and verified the source code of the contestants' solutions [41]. Over time this became infeasible and gave birth to automatic judge systems.

To prevent people from interfering with the normal workflow of the competition e.g. Denial of Service Attack by exhausting memory resources, the automatic judge systems need a secure way to compile and execute a contest's solution. This is where sandboxes come into place.

First sandboxes required modification of the OS kernel [34, 6, 7, 9, 12]. While they had little run-time overhead, some of them were limited to single-threaded applications [23].

Later, as support for process tracing matured, `ptrace`-based sandboxes arose [18, 11, 10]. The problem with those solutions is the overhead that varies from around 75% [22] to 160% [19] for syscall-intensive programs. This overhead however, does not affect programming contest fairness much [17]. Supporting multi-threaded and multi-process programs while using `ptrace` is tricky, but possible [10], because of Time of Check/Time of Use (TOCTOU) problem [4]. `ptrace`-based sandbox needs to inspect syscall arguments. To do so it has to read them, but the multi-threaded or multi-process program can change the indirect argument after the reading but before the kernel uses the argument. This creates a dangerous race condition that has to be addressed.

Finally, after the kernel support for containerization materialized, namespace and cgroup based sandboxes came into place [19, 24, 35, 8, 5, 38]. Contrary to ptrace-based sandboxes, namespace-based sandboxes have negligible runtime overhead [19]. Moreover, they don't require modifications of the Linux kernel and work on major Linux distributions out of the box.

2.2. Existing Implementations

2.2.1. Modifying OS kernel

Systrace

Systrace [34] intercepts all system calls in the kernel. It then decides if the syscall is safe by first checking a static list of safe syscalls. This step exists to reduce sandboxing performance overhead. If the syscall is not on the list, Systrace consults user space for a decision.

The system avoids TOCTOU problem [4] by copying syscall arguments to kernel memory before asking user space for a decision.

Janus

Janus [6] adds a module to extend Linux `ptrace` API. Policies are defined using configuration files. By default all syscalls are denied. The configuration directive refers to the policy module that provides the logic for deciding whether to allow a particular system call or not. For example, `path` module could be used to restrict IO on certain file paths.

Ostia

Ostia [7] instead of filtering system calls it delegates them to an external agent that performs syscalls on behalf of the sandboxed process. Authors emphasize that such architecture simplifies the system and protects from TOCTOU problems [4].

Ostia is implemented as two components: a small kernel module and a user space part. The module intercepts the syscall and copies its arguments via IPC link to the user space agent. The agent decides whether the call should be allowed, executes it and returns the results back over the IPC link. Worth noting is that not all syscalls have to delegated — some can be always allowed while others always denied.

TxBox

TxBox [9] introduces system-level transaction support. Impact of the untrusted insecure code is limited by rolling back the system state after the execution. This provides strong isolation and works with arbitrary executables but requires significant out-of-tree patches of the OS kernel.

MiniBox

MiniBox [12] is a two-way sandbox that protects operating system from the application as well as application from the operating system. A modified version of TrustVisor [20] hypervisor runs OS and sandboxed application separately in a Mutually Isolated Execution Environment. The hypervisor is the only communication channel between the isolated application and the regular OS. This way application is protected from the malicious operating system. To protect the OS from the application, MiniBox uses Software Fault Isolation techniques from NaCl [43].

SACO sandbox

South African Computer Olympiad (SACO) sandbox [23] inserts a custom kernel module that hooks up to Linux Security Module infrastructure. Although it has negligible time and memory overhead, it only supports single-threaded programs.

2.2.2. `ptrace`-based

MO sandbox

MO sandbox [18, 11] allows only single-threaded programs. It simply inspects arguments using `ptrace` and uses `setrlimit` [30] to limit resources. It is used by USA Computer Olympiad (USACO).

MBOX

MBOX [10] requires no superuser privileges. It makes use of seccomp BPF system call filtering to restrict allowed syscalls. BPF filtering is effective only for non-indirect arguments. To address this issue, the installed BPF filter notifies the `ptrace` monitoring process if further argument inspection is necessary to make a decision. To avoid TOCTOU problem [4], the MBOX allocates a read-only page to which it copies the indirect arguments before inspecting them and rewrites the syscall to use the rewritten arguments. The copied arguments are protected against modification because changing page access permissions is impossible without a syscall.

2.2.3. Using Linux namespaces

Firejail

Firejail [24] uses seccomp BPF system call filtering and mount namespaces to restrict filesystem access. Similarly it uses process namespaces to limit view of running processes and network namespaces to restrict access to network devices. However, Firejail uses a `setuid` [32] helper binary to achieve that. It allows resource limiting through `prlimit` [30].

nsjail

nsjail [8] uses Linux namespaces, seccomp BPF system call filtering, `setrlimit` [30] and cgroups to limit resources. It does not require superuser privileges. However, it is not optimized for running short-running programs. Also, it does not provide statistics of the run.

nsroot

nsroot [35] does not support resource limiting. It only makes use of Linux namespaces to restrict view of the file system, IPC and network devices.

Flatpak

Flatpak [5], previously xdg-app, is a software packaging and sandboxing tool. Internally, it uses Bubblewrap sandbox. The Bubblewrap [13] is a `setuid` [32] program that uses Linux namespaces and seccomp filters.

New Contest Sandbox

New Contest Sandbox [19] uses Linux namespaces and cgroups but not seccomp filters. It is used by Moe modular contest system (2012) [19]. Linux namespaces and cgroups have negligible overhead compared to `ptrace`.

APAC

APAC (Automatic Programming Assignment Checker) [38] uses Docker for sandboxing. It sets up a container for each run. Docker uses runC under the hood. While runtime overhead of Docker is low, the setup phase is primary source of overhead for short-running programs.

runC

runC [3] uses the same features of Linux kernel as nsjail. However, configuration is stored as files instead of passed as command-line arguments. It has a special `rootless` mode which does not require superuser privileges. Given all of the above however, it is not optimized for running short-running programs. runC is used internally by Docker.

2.2.4. Other

Google Native Client

Native Client (NaCl) [43] uses static analysis and Software Fault Isolation. After the static analysis, the program runs at native speed but requires recompilation with special compiler and libraries. NaCl only works for x86 architecture.

2.3. Conclusion

Many sandboxing solutions exist. From all of the above, closest to our requirements is nsjail (see Section 2.2.3). However, it is not optimised for running short-running programs. In fact, none of the above solutions is optimised to run hundreds of short-running programs per second.

Considering the similarities of nsjail and our solution, in the performance analysis we will compare our sandbox to nsjail sandbox.

Chapter 3

Design and Architecture

3.1. Client-server architecture

From the start the sandbox was based on the client-server architecture. This was the choice to minimize process cloning overhead [14], since it is a costly operation and happens for every sandboxing request. fork/clone needs to clone the whole address space of the process and the client process could have a large address space. Server process that is executed from a separate executable has a minimal required address space size therefore the cloning overhead is minimal for every request. Moreover, this architecture easily and safely allows sharing as much work as possible between the sandboxing requests which is the key to low overhead of running short-running programs.

The client spawns the sandboxing server and sends sandboxing requests via UNIX domain socket to the server. This is illustrated on Figure 3.1. The request contains executable, arguments, namespace configuration, resource limits, seccomp BPF filters and a pipe through which the result will be sent back.

At startup, the server creates cgroup hierarchy and some namespaces so that they won't have to be created later or their creation will be faster. Other utilities are also setup here to do it once instead of for every request. Then it starts accepting requests.

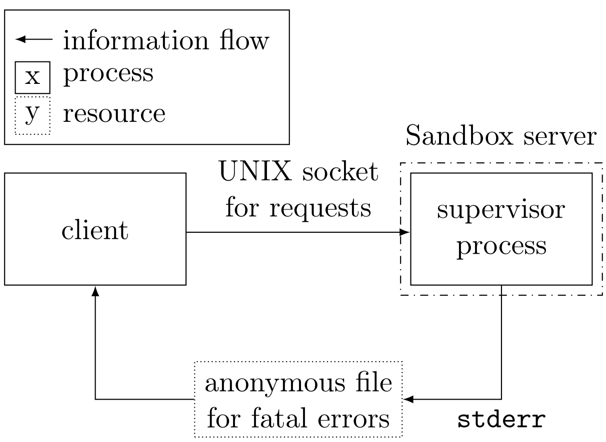

Figure 3.1: Sandbox server waits for requests. Client sends requests through the UNIX socket. Sandbox server will die on fatal error leaving the error message for the client in the anonymous file.

3.1.1. Sandboxing request handling

For each request, the server process (aka supervisor) spawns the PID 1 process of the new PID namespace. Then the init process setups namespaces and some of the resource limits. Finally the PID 1 process spawns the tracee process that finishes configuration and executes the requested executable. So for each sandboxing request we spawn exactly 2 processes. However, the executed program can spawn new processes — each of them is referred to as a "tracee process". The PID 1 process is necessary for a couple of reasons:

- It reaps the zombie processes in the tracee PID namespace.
- It allows locking mount-points in the mount namespace. The tracee process is spawned in a new user and mount namespace. Mounts are performed by the PID 1 process, therefore all mounts become locked together and cannot be individually unmounted by the tracee [31]. These mounts cannot be performed by the supervisor process instead, because it would alter the mount namespace for subsequent requests.
- Inside a PID namespace, sending signals to the PID 1 process is allowed only for signals that the PID 1 process installed signal handler for. This could change the behavior for some programs, therefore a helper PID 1 process is needed.

This is shown on Figure 3.2.

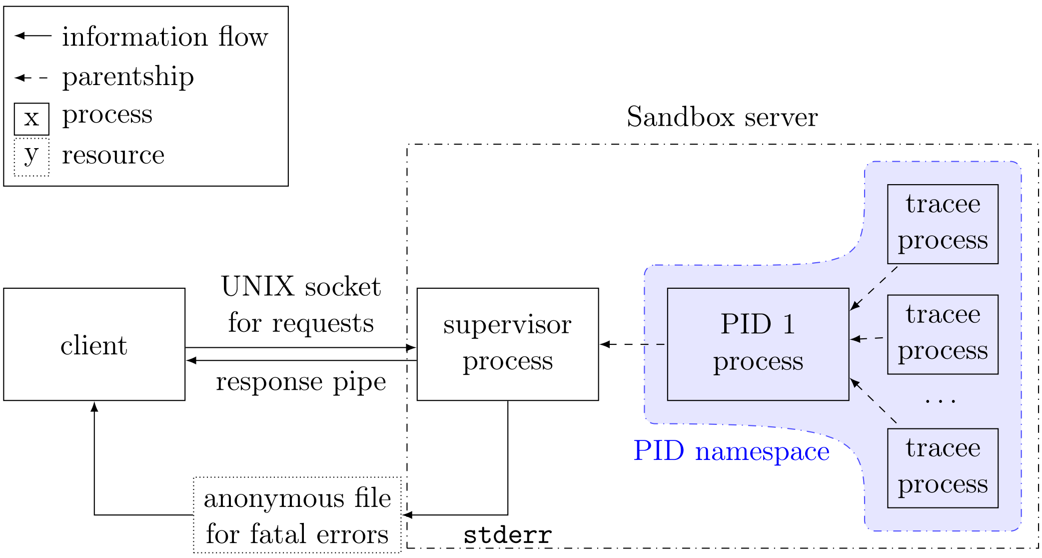

Figure 3.2: Sandbox server handles a request, at the moment after executing the requested executable. Sandbox server will die on fatal error leaving the error message for the client in the anonymous file. Sandbox server consist of the supervisor process and its child — the PID 1 process that is spawned for each request. The PID 1 process performs a role of the init process in the PID namespace of the tracee processes.

3.2. Cgroups

The server gains write access to cgroup hierarchy by being executed through `systemd-run --user --scope --property=Delegate=yes --collect`. It enables `pid`, `memory`, and `cpu` controllers for the below subgroups.

The server process creates at startup the cgroup v2 hierarchy that looks as follows:

- `/supervisor` — cgroup of the supervisor process,
- `/pid1` — cgroup of the pid1 process,
- `/tracee` — cgroup of the tracee processes.

After creation of the hierarchy it places the supervisor process in its cgroup. Subsequent processes are placed in their cgroups by making use of `CLONE_INTO_CGROUP` flag.
`/tracee` cgroup allows:

- Killing all tracee processes by writing 1 to `/tracee/cgroup.kill` file.
- Reading CPU user and CPU system time via `/tracee/cpu.stat` file.
- Reading peak memory usage via `/tracee/memory.peak` file.
- Setting process/tread number limit by writing `/tracee/pids.max`.
- Setting memory hard limit by writing `/tracee/memory.max`.
- Setting CPU usage limit by writing `/tracee/cpu.max`.
- Disabling PSI accounting to reduce the sandboxing overhead by writing 0 to `/tracee/cgroup.pressure` file.

`/tracee` cgroup needs to be deleted and recreated after each request to reset `/tracee/cpu.stat` and `/tracee/memory.peak` files.

3.3. Linux namespaces

Linux allows unprivileged users to create user namespaces only. However, after entering a new user namespace the process gains all privileges inside the namespace and can create other namespaces.
The supervisor process creates the following namespaces:

- user namespace — in order to create other namespaces and hide user ID and group ID,
- mount namespace — to allow mounting detached cgroups v2 hierarchy,
- cgroup namespace — to allow mounting detached cgroups v2 hierarchy,
- network namespace — to disconnect every tracee from network devices, done once, as it is costly,
- IPC namespace — to isolate every tracee from other processes' IPC, done once, for optimization,
- UTS namespace — to isolate every tracee from host's hostname, done once, for optimization,
- time namespace — to isolate every tracee from host's time namespace, done once, for optimization.

The pid1 process creates the following namespaces:

- user namespace — in order to create other namespaces and hide user ID and group ID,
- mount namespace — to allow mounting requested mount-point hierarchy,
- PID namespace — to isolate tracee from accessing other processes.

The tracee process creates the following namespaces:

- user namespace — in order to create other namespaces and hide user ID and group ID and lock the mount tree,
- mount namespace — in lock the mount tree created by pid1.

The listed namespaces hierarchy is illustrated on Figure 3.3.

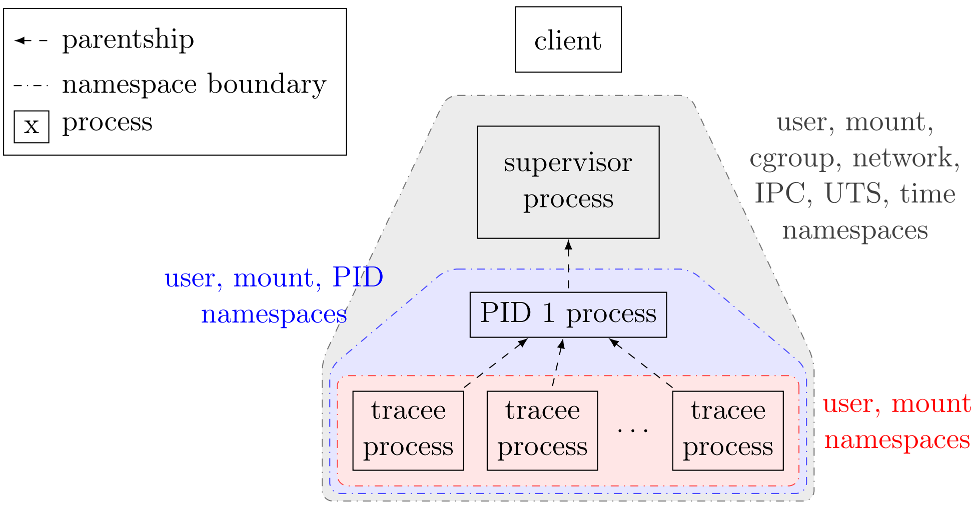

Figure 3.3: Namespaces hierarchy of the sandbox server processes.

3.4. Inter-process communication

The client sends requests via UNIX domain socket to the supervisor process. The results are sent via a pipe attached to the request. The pipe is attached to the request as a file descriptor using `SCM_RIGHTS` control message [33].

The supervisor, pid1 and tracee process communicate via shared anonymous memory page. Figure 3.4 illustrates this communication. Such communication requires no syscalls, is fast and reliable. This page is automatically unmapped upon `execveat` syscall [29] in the tracee process, so it is protected from the tracee access.

3.5. Capabilities

The supervisor process drops all capabilities, sets securebits and `NO_NEW_PRIVS` flag. This ensures minimal possible capabilities in its and ancestor user namespace and prevents gaining any new privileges.

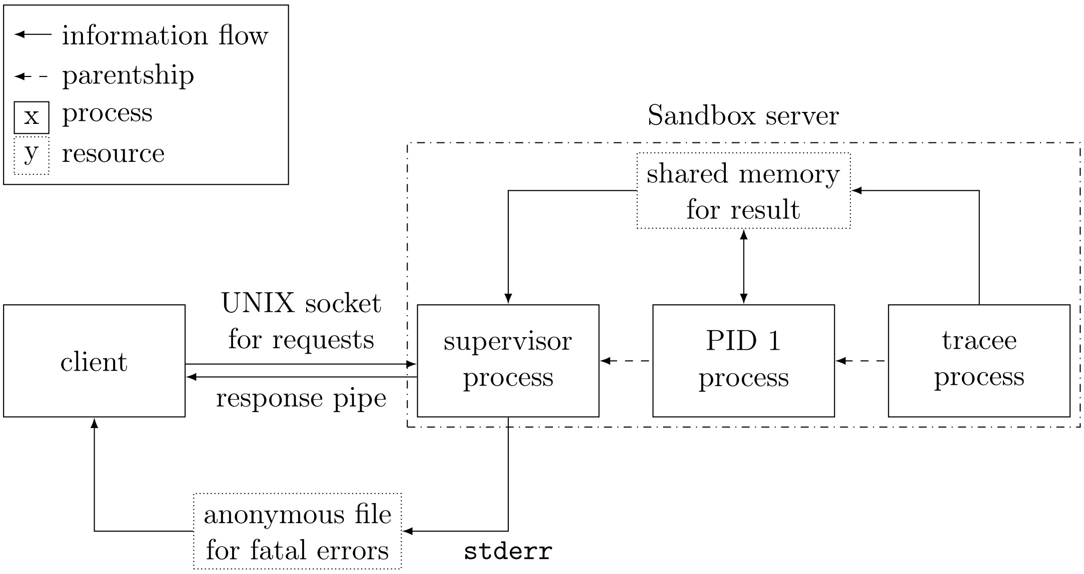

Figure 3.4: Sandbox server handles a request, at the moment before executing the requested executable. Sandbox server will die on fatal error leaving the error message for the client in the anonymous file. Sandbox server consist of the supervisor process and its child — the PID 1 process that is spawned for each request and its child — the tracee process that will execute the requested executable. The tracee process saves in the shared memory the time just before `execveat` and signals the PID 1 process. The PID 1 process reads the time saved in the shared memory and starts the real and CPU timers. After the tracee process dies, the PID 1 process writes exit code and status of the tracee process and the time it died. Moreover, the shared memory is used to communicate fatal errors to the supervisor process.

3.6. Hardening

The pid1 process, after spawning the tracee enters a new cgroup namespace to limit view of other namespaces i.e. if the tracee somehow takes control of the pid1 process, it will not be able to raise its PID and memory limit. Moreover a seccomp filter is installed to limit allowed syscalls only to those needed for reaping the orphaned zombie process, managing time limits and exiting upon tracee death.

3.7. Conclusion

Client-server architecture allows time-performance optimizations. Furthermore, it allows more common work to be done once and simplifies implementation. For instance, file descriptors do not leak to other processes because there are no threads that could fork a new process. Request handling requires creation of 2 processes — the PID 1 process and the tracee process that later executes the requested program. Resource limits and accounting is mostly performed by cgroups. Isolation is achieved by deft usage of Linux namespaces.

Chapter 4

Implementation

The project is written in C++, as it is a low-overhead, low-level language, but more convenient than C. Git is used as a Version Control System to track incremental implementation. Invaluable tool used during development was `strace` [39] that allowed easy inspection of system calls and their return value.

4.1. Interface

The client has to spawn the sandbox server — the supervisor process.

```
SupervisorConnection spawn_supervisor();
```

spawn_supervisor()

Through `SupervisorConnection` object the client has access to requesting secure execution of programs.

```
class SupervisorConnection {
public:
    SupervisorConnection(const SupervisorConnection&) = delete;
    SupervisorConnection(SupervisorConnection&& sc) noexcept;
    SupervisorConnection& operator=(const SupervisorConnection&) = delete;
    SupervisorConnection& operator=(SupervisorConnection&&) noexcept = delete;
    // Cancels pending and in-progress requests
    ~SupervisorConnection() noexcept(false);

    class RequestHandle;
    class KillRequestHandle;

    /// Sends the request and returns immediately without waiting for the request
    /// to complete. Multiple requests can be send before awaiting completion
    /// with await_result(). Throws if there is any error with the supervisor.
    [[nodiscard]] RequestHandle
    send_request(int executable_fd, Slice<std::string_view> argv,
                 const RequestOptions&);

    /// Sends the request and returns immediately without waiting for the request
    /// to complete. Multiple requests can be send before awaiting completion
    /// with await_result(). Throws if there is any error with the supervisor.
    [[nodiscard]] RequestHandle
    send_request(std::string_view executable_path, Slice<std::string_view> argv,
                 const RequestOptions&);
```

```cpp

    /// Sends the request and returns immediately without waiting for the request
    /// to complete. Multiple requests can be send before awaiting completion
    /// with await_result(). Throws if there is any error with the supervisor.
    [[nodiscard]] RequestHandle
    send_request(Slice<std::string_view> argv,
                 const RequestOptions& options = {});

    /// Throws if there is any error with the supervisor.
    Result await_result(RequestHandle&& request_handle);
};
```

SupervisorConnection class

`send_request()` returns `RequestHandle` object that can be used to obtain result of the execution, cancel execution or get the `KillRequestHandle`. Canceling execution is useful in case of errors in the client, where result of the execution as well as the execution itself is no longer needed. Tracee processes if they are executing are immediately killed and the result is discarded. Canceling an already finished request discards the result. Canceling the request before the server started handling it causes it to be skipped.

```cpp
class SupervisorConnection {
public:
    class [[nodiscard]] RequestHandle {
    public:
        RequestHandle(const RequestHandle&) = delete;
        RequestHandle(RequestHandle&& rh) noexcept;
        RequestHandle& operator=(const RequestHandle&) = delete;
        RequestHandle& operator=(RequestHandle&&) noexcept = delete;
        // Cancels the request
        ~RequestHandle() noexcept(false);
        // no-op on already cancelled request
        void cancel();
        // returned fd can be polled with POLLIN
        [[nodiscard]] int pollable_fd() const noexcept;
        // can be called only once
        KillRequestHandle get_kill_handle() noexcept;
    };
};
```

SupervisorConnection::RequestHandle class

The `KillRequestHandle` object allows killing all tracee processes. Killing an already finished request is no-op. Killing the request before the server started handling it causes tracee to be immediately killed after `execveat`.

```cpp
class SupervisorConnection {
public:
    class [[nodiscard]] KillRequestHandle {
    public:
        KillRequestHandle(const KillRequestHandle&) = delete;
        KillRequestHandle(KillRequestHandle&& krh) noexcept;
        KillRequestHandle& operator=(const KillRequestHandle&) = delete;
        KillRequestHandle& operator=(KillRequestHandle&&) noexcept = delete;
        ~KillRequestHandle() noexcept(false);
        // no-op on already killed or cancelled request
        void kill();
```

```
    };
};
```

SupervisorConnection::KillRequestHandle class

`await_result()` returns result of the execution of the request. It can either be an error with textual description. This error is not fatal to the supervisor process i.e. new requests can be sent. The result consist of the exit's code and status, and runtime statistics:

- real time aka `result::Ok::runtime`,
- CPU user and CPU system time aka `result::Ok::cgroup::cpu_time::user` and `result::Ok::cgroup::cpu_time::system`,
- peak tracee cgroup memory usage aka `result::Ok::cgroup::peak_memory_in_bytes`.

```
/// Describes exit status of the sandboxed process
struct Si {
    int code; // siginfo_t::si_code from waitid() of the root process
    int status; // siginfo_t::si_status from waitid() of the root process

    [[nodiscard]] std::string description() const;
};

namespace result {
struct Ok {
    Si si;
    std::chrono::nanoseconds runtime; // from CLOCK_MONOTONIC_RAW

    struct Cgroup {
        struct CpuTime {
            std::chrono::microseconds user;
            std::chrono::microseconds system;

            [[nodiscard]] std::chrono::microseconds total() const noexcept {
                return user + system;
            }
        } cpu_time;

        uint64_t peak_memory_in_bytes;
    } cgroup;
};

struct Error {
    std::string description;
};
} // namespace result

using Result = std::variant<result::Ok, result::Error>;
```

SupervisorConnection::await_result() return type

The `RequestOptions` allows specifying:

- `stdin`, `stdout`, and `stderr` file descriptors. If `std::nullopt` is specified `/dev/null` is opened as the file descriptor.

- Environment as array view of string views.
- Linux namespace configuration:
  - user ID mapping,
  - mounts and new root mount,
- Cgroup resource limits: process and thread limit, memory limit, CPU maximum bandwidth.
- `prlimit` hard limits.
- Real time limit.
- CPU time limit.
- Seccomp BPF filter as a file descriptor. The decision to pass it as file descriptor is that it lowers the overhead of reusing the same filter multiple times — a common scenario in a judge system. Only the file descriptor needs to be send with each request instead of the whole BPF filter contents. This allows the filter to be compiled once and passed for multiple requests with minimal overhead and simple API. An alternative is to extend API to save seccomp filters but it was considered unnecessary given the how small is the overhead of passing a single file descriptor.

```
struct RequestOptions {
    std::optional<int> stdin_fd = std::nullopt;
    std::optional<int> stdout_fd = std::nullopt;
    std::optional<int> stderr_fd = std::nullopt;
    Slice<const std::string_view> env = {};

    struct LinuxNamespaces {
        struct User {
            // nullopt will leave it the same as the outside effective
            // user ID (euid)
            std::optional<uid_t> inside_uid = std::nullopt;
            // nullopt will leave it the same as the outside effective
            // group ID (egid)
            std::optional<gid_t> inside_gid = std::nullopt;
        } user = {};

        struct Mount {
            struct MountTmpfs {
                std::string_view path;
                // Will be rounded up to page size multiple; 0 rounds up to
                // page size due to technical limitations.
                std::optional<uint64_t> max_total_size_of_files_in_bytes = 0;
                std::optional<uint64_t> inode_limit = 0;
                mode_t root_dir_mode = S_0755;
                bool read_only = true; // make the mount read-only
                // Do not allow programs to be executed on this mount
                bool no_exec = true;
            };

            struct MountProc {
```

```
                std::string_view path;
                bool read_only = true; // make the mount read-only
                // Do not allow programs to be executed on this mount
                bool no_exec = true;
            };

            struct BindMount {
                // Path to file or directory to bind mount
                std::string_view source;
                std::string_view dest; // path at which to bind mount
                bool recursive = true; // create a recursive bind mount
                bool read_only = true; // make the mount read-only
                // Do not allow programs to be executed on this mount
                bool no_exec = true;
                // Do not follow symbolic links if source points to one
                bool symlink_nofollow = false;
            };

            struct CreateDir {
                std::string_view path;
                mode_t mode = S_0755;
            };

            struct CreateFile {
                std::string_view path;
                mode_t mode = S_0644;
            };

            using Operation = std::variant<
                MountTmpfs, MountProc, BindMount, CreateDir, CreateFile
            >;

            Slice<const Operation> operations = {};
            std::optional<std::string_view> new_root_mount_path = std::nullopt;
        } mount = {};
    } linux_namespaces = {};

    struct Cgroup {
        // Every process or thread counts as 1
        std::optional<uint32_t> process_num_limit = std::nullopt;
        std::optional<uint64_t> memory_limit_in_bytes = std::nullopt;

        struct CpuMaxBandwidth {
            uint32_t max_usec;
            uint32_t period_usec;
        };

        // Writes "$max_usec $period_usec" to cpu.max cgroup file
        std::optional<CpuMaxBandwidth> cpu_max_bandwidth = std::nullopt;
    } cgroup = {};

    struct Prlimit {
        // RLIMIT_AS
        std::optional<uint64_t> max_address_space_size_in_bytes = std::nullopt;
        // RLIMIT_CORE
```

```cpp
        std::optional<uint64_t> max_core_file_size_in_bytes = std::nullopt;
        // RLIMIT_CPU
        std::optional<uint64_t> cpu_time_limit_in_seconds = std::nullopt;
        // RLIMIT_FSIZE
        std::optional<uint64_t> max_file_size_in_bytes = std::nullopt;
        // RLIMIT_NOFILE
        std::optional<uint64_t> file_descriptors_num_limit = std::nullopt;
        // RLIMIT_STACK
        std::optional<uint64_t> max_stack_size_in_bytes = std::nullopt;
    } prlimit = {};

    std::optional<std::chrono::nanoseconds> time_limit = std::nullopt;
    std::optional<std::chrono::nanoseconds> cpu_time_limit = std::nullopt;
    // For starting tracee sandbox needs allowed syscalls:
    // - kill()
    // - execveat()
    // - write()
    // - clock_getttime()
    // - sched_yield()
    // - pread64()
    std::optional<int> seccomp_bpf_fd = std::nullopt;
};
```

SupervisorConnection::await_result() return type

4.2. Time limits

The PID 1 process controls the time limits. The tracee process, just before `execveat` saves current real time from `CLOCK_MONOTONIC_RAW` and CPU time from `cpu.stat` tracee cgroup file to the shared memory (see Figure 3.4). The problem with `cpu.stat` file is that it is updated infrequently and just reading it in the young tracee process often gives 0 as the consumed CPU time of the tracee. Fortunately, executing `sched_yield()` system call forces recalculation of the file and the values are no longer 0. This is why this syscall is required as allowed in the seccomp BPF filter.

4.2.1. Real time limit

After saving the current real time the tracee process signals the PID 1 process with `SIGUSR2`. The PID 1 process reads the saved real time and sets up a POSIX timer to expire at the moment of saved time + real time limit. Upon the timer expiration, `SIGUSR1` is sent by the kernel to the PID 1 process and it terminates all tracee processes by writing 1 to `cgroup.kill` file of the tracee cgroup.

4.2.2. CPU time limit

In case the tracee is not restricted to one process, the setup is analogous to real time except that there is no CPU timer for a cgroup of processes. Instead we calculate minimal period of time in which the CPU time limit could expire as follows: $\frac{\text{remaining cpu time}}{\text{max parallelism}}$, where max parallelism equals: $\min(\text{available threads}, \texttt{process\_num\_limit}, \texttt{cpu\_max\_bandwidth} \text{ in threads})$. Upon the timer expiration the remaining cpu time is recalculated and the timer is rescheduled if the remaining cpu time is greater than 0. Timer expiration is signaled by the kernel as signal SIGXCPU. To prevent polling, the minimal timer expiration period is capped to have minimum value of 1ms — this gives at most 1000 checks per second.

In case the tracee is restricted to one process, the setup is different. After saving the current CPU time the tracee process signals the PID 1 process through a pipe. A signal cannot be used because timer_create syscall and clock_getcpuclockid library function are not marked async-signal-safe — they are not specified to be safe to call inside a signal handler. An eventfd cannot be used either, because if tracee dies before writing the eventfd, the PID 1 process will wait indefinitely on the read syscall. With pipe read returns 0 when the other end becomes closed. With limit to one process we use the CPU timer of the tracee process and set up a timer to expire when the tracee exceeds the CPU time limit.

In both cases, when the CPU time limit is exceeded, the PID 1 process and it terminates all tracee processes by writing 1 to cgroup.kill file of the tracee cgroup.

4.3. Runtime statistics

After the main tracee process (the first spawned) exists, the init process saves the current real time and the exit status in the shared memory unless the tracee set an error and exists. The kernel kills all remaining tracee processes (because the PID namespace's init process died). After the PID 1 process exists, the supervisor process reads the shared memory (see Figure 3.4). It checks if there is an either tracee or PID 1 error. If there is one, it becomes the result of the request. If there is none, the supervisor process calculates:

- real time using formula: time of tracee death − saved execveat real time,
- CPU time using formula: CPU time read from cpu.stat file−saved execveat CPU time,
- Peak memory usage by reading memory.peak tracee cgroup file.

4.4. Error handling

Errors in supervisor process are considered fatal and are reported by writing to stderr. After writing errors, the supervisor process exists immediately. When the client tries to read the request result, the read will fail with read returning unexpected value 0. The client then ensures the supervisor process is dead and tries to read the error the supervisor wrote. If the client finds one it throws is, otherwise it throws read error.

The PID 1 process and tracee process write error to the shared memory (see Figure 3.4) and exit immediately. The supervisor process reports these errors as a request result.

4.5. Request sending and receiving

The request is sent via UNIX domain socket (see Figure 3.1). The request is serialized to custom binary format and sent: constant-length header with file descriptors and the body. On the other end (supervisor process) the header with file descriptors and later the request body. Then the request is deserialized and handled.

4.6. TODO...

Chapter 5

Performance Evaluation

Chapter 6

Use Cases and Applications

Chapter 7

Integration with Online Judge

Chapter 8

Future work and Opportunities

Chapter 9

Conclusion

Bibliography

[1] David Beserra et al. “Performance Analysis of LXC for HPC Environments.” In: *CISIS*. IEEE Computer Society, 2015, pp. 358–363. ISBN: 978-1-4799-8870-9. URL: http://dblp.uni-trier.de/db/conf/cisis/cisis2015.html#BeserraMEBSF15.

[2] Sander van der Burg and Eelco Dolstra. “Automating System Tests Using Declarative Virtual Machines”. In: *2010 IEEE 21st International Symposium on Software Reliability Engineering*. 2010, pp. 181–190. DOI: 10.1109/ISSRE.2010.34.

[3] Henrique Zanela Cochak et al. “RunC and Kata runtime using Docker: a network perspective comparison”. In: *2021 IEEE Latin-American Conference on Communications (LATINCOM)*. IEEE. 2021, pp. 1–6.

[4] *CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition*. URL: https://cwe.mitre.org/data/definitions/367.html (visited on 2023-10-19).

[5] Flatpak. *Flatpak - the future of application distribution*. URL: https://flatpak.org/ (visited on 2023-10-17).

[6] T Garfinkel. “Janus: A practical tool for application sandboxing”. In: *http://www. cs. berkeley. edu/daw/janus* (2004).

[7] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. “Ostia: A Delegating Architecture for Secure System Call Interposition.” In: *NDSS*. 2004.

[8] Google. *A light-weight process isolation tool, making use of Linux namespaces and seccomp-bpf syscall filters*. URL: https://github.com/google/nsjail (visited on 2023-10-17).

[9] Suman Jana, Donald E Porter, and Vitaly Shmatikov. “TxBox: Building secure, efficient sandboxes with system transactions”. In: *2011 IEEE Symposium on Security and Privacy*. IEEE. 2011, pp. 329–344.

[10] Taesoo Kim and Nickolai Zeldovich. “Practical and effective sandboxing for non-root users”. In: *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 2013, pp. 139–144.

[11] Rob Kolstad. “Infrastructure for contest task development”. In: *Olympiads in Informatics* 3 (2009), pp. 38–59.

[12] Yanlin Li et al. “{MiniBox}: A {Two-Way} Sandbox for x86 Native Code”. In: *2014 USENIX annual technical conference (USENIX ATC 14)*. 2014, pp. 409–420.

[13] *Low-level unprivileged sandboxing tool used by Flatpak and similar projects*. URL: https://github.com/containers/bubblewrap (visited on 2023-10-19).

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

[15] Krzysztof Małysa. *Sim project*. URL: https://github.com/varqox/sim (visited on 2023-03-15).

[16] Krzysztof Małysa. *Sip – a tool for preparing problem packages for the Sim platform*. URL: https://github.com/varqox/sip (visited on 2023-03-15).

[17] Martin Mareš. "Fairness of Time Constraints." In: *Olympiads in Informatics* 5 (2011), pp. 92–102.

[18] Martin Mareš. "Perspectives on grading systems". In: *Olympiads in Informatics* (2007), pp. 124–130.

[19] Martin Mareš and Bernard Blackham. "A New Contest Sandbox." In: *Olympiads in Informatics* 6 (2012), pp. 100–109. URL: https://ioi.te.lv/oi/pdf/INFOL094.pdf.

[20] Jonathan M McCune et al. "TrustVisor: Efficient TCB reduction and attestation". In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 143–158.

[21] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux J.* 2014.239 (2014-03). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241.

[22] Bruce Merry. "Performance analysis of sandboxes for reactive tasks". In: *Olympiads in Informatics* 4 (2010), pp. 87–94.

[23] Bruce Merry. "Using a Linux security module for contest security". In: *Olympiads in Informatics* 3 (2009), pp. 67–73.

[24] netblue30/firejail. *Linux namespaces and seccomp-bpf sandbox*. URL: https://github.com/netblue30/firejail (visited on 2023-10-17).

[25] *Official website of Kernel Virtual Machine*. URL: https://www.linux-kvm.org/ (visited on 2022-11-23).

[26] *Official website of QEMU — A generic and open source machine emulator and virtualizer*. URL: https://www.qemu.org/ (visited on 2022-11-23).

[27] Oracle. *Official website of VirtualBox*. URL: https://www.virtualbox.org/ (visited on 2022-11-23).

[28] Vassilis Prevelakis and Diomidis Spinellis. "Sandboxing Applications." In: *Usenix annual technical conference, freenix track*. Citeseer. 2001, pp. 119–126.

[29] man-pages project. *execveat - execute program relative to a directory file descriptor*. URL: https://man7.org/linux/man-pages/man2/execveat.2.html (visited on 2023-10-20).

[30] man-pages project. *getrlimit, setrlimit, prlimit - get/set resource limits*. URL: https://man7.org/linux/man-pages/man2/prlimit.2.html (visited on 2023-10-18).

[31] man-pages project. *mount_namespaces - overview of Linux mount namespaces*. URL: https://man7.org/linux/man-pages/man7/mount_namespaces.7.html (visited on 2023-10-20).

[32] man-pages project. *setuid - set user identity*. URL: https://man7.org/linux/man-pages/man2/setuid.2.html (visited on 2023-10-18).

[33] man-pages project. *unix - sockets for local interprocess communication*. URL: https://man7.org/linux/man-pages/man7/unix.7.html (visited on 2023-10-20).

[34] Niels Provos. "Improving Host Security with System Call Policies." In: *USENIX Security Symposium*. 2003, pp. 257–272.

[35] Inge Alexander Raknes, Bjørn Fjukstad, and Lars Ailo Bongo. "nsroot: Minimalist process isolation tool implemented with linux namespaces". In: *arXiv preprint arXiv:1609.03750* (2016).

[36] rootlesscontaine.rs. *Rootless Containers*. URL: https://rootlesscontaine.rs (visited on 2022-11-28).

[37] Giuseppe Scrivano. *Rootless containers with Podman and fuse-overlayfs*. 2019-06-04. URL: https://indico.cern.ch/event/757415/contributions/3421994/attachments/1855302/3047064/Podman_Rootless_Containers.pdf (visited on 2022-11-28).

[38] František Špaček, Radomír Sohlich, and Tomáš Dulík. "Docker as Platform for Assignments Evaluation". In: *Procedia Engineering* 100 (2015). 25th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2014, pp. 1665–1671. ISSN: 1877-7058. DOI: https://doi.org/10.1016/j.proeng.2015.01.541. URL: https://www.sciencedirect.com/science/article/pii/S1877705815005688.

[39] *strace - trace system calls and signals*. URL: https://man7.org/linux/man-pages/man1/strace.1.html (visited on 2023-10-20).

[40] systemd. *systemd-nspawn — Spawn a command or OS in a light-weight container*. URL: https://www.freedesktop.org/software/systemd/man/systemd-nspawn.html (visited on 2022-11-28).

[41] Tocho Tochev and Tsvetan Bogdanov. "Validating the Security and Stability of the Grader for a Programming Contest System." In: *Olympiads in Informatics* 4 (2010), pp. 113–119.

[42] VMWare. *Official website of VMWare Workstation*. URL: https://www.vmware.com/products/workstation/ (visited on 2022-11-23).

[43] Bennet Yee et al. "Native client: A sandbox for portable, untrusted x86 native code". In: *Communications of the ACM* 53.1 (2010), pp. 91–99.