Process Sandbox: Design Document

Overview

A defense-in-depth process sandbox that leverages multiple Linux security primitives (namespaces, seccomp, cgroups, capabilities) to create isolated execution environments for untrusted code. The key architectural challenge is orchestrating these security mechanisms in the correct order while maintaining usability and performance.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

Context and Problem Statement

> **Milestone(s):** This section provides foundational context for all milestones (1-5)

The challenge of safely executing untrusted code represents one of the most fundamental problems in computer security. Whether running user-submitted scripts, processing uploaded files, or executing third-party plugins, systems must balance functionality with security. Traditional approaches to this problem have significant limitations that make them unsuitable for many modern use cases.

Mental Model: The High Security Prison

Think of a process sandbox like a **high-security prison designed to contain dangerous inmates**. Just as a maximum-security facility uses multiple layers of containment—perimeter walls, guard towers, internal barriers, restricted movement, limited resources, and constant monitoring—a robust process sandbox must employ multiple security mechanisms that work together to prevent escape.

In this analogy, the **untrusted code is the dangerous prisoner** who cannot be trusted to follow rules and will actively attempt to break out. A single security measure, like a simple fence, is insufficient because prisoners are clever and motivated. They might dig tunnels (exploit filesystem paths), bribe guards (exploit setuid binaries), impersonate staff (privilege escalation), or exhaust prison resources (denial of service attacks).

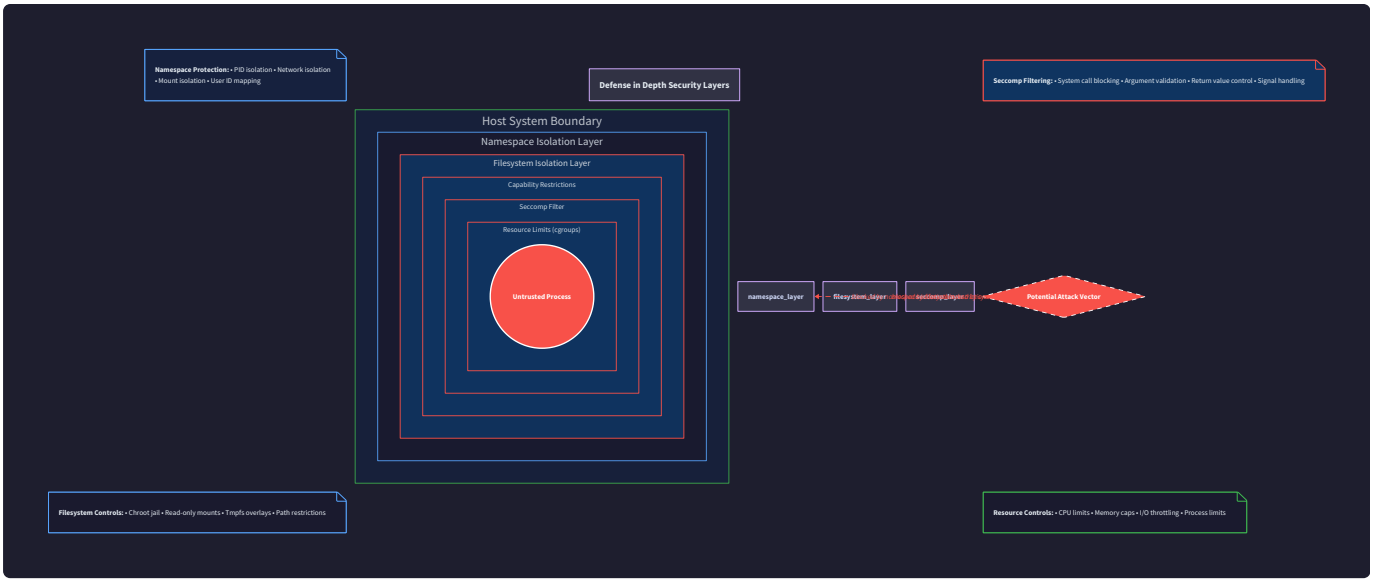
The **defense-in-depth approach** mirrors how real prisons operate. Even if an inmate bypasses one security layer—say, they pick a cell lock—they still face additional barriers: locked corridor doors, guard checkpoints, perimeter walls, and external monitoring. Each layer has a different mechanism of action: physical barriers, human oversight, technological detection, and resource constraints.

Similarly, our process sandbox employs multiple complementary security primitives:

- **Namespaces** act like the prison's **separate wings and buildings**, creating isolated views where the prisoner believes they're in a different facility entirely
- **Filesystem isolation** serves as **restricted movement areas**, limiting which locations the prisoner can access
- **Seccomp filtering** functions like **strict rule enforcement**, blocking specific dangerous actions entirely
- **Cgroups** operate as **resource rationing**, preventing any single prisoner from consuming all available food, water, or electricity

- **Capability dropping** resembles **privilege revocation**, removing special permissions that could enable escape or harm

The key insight is that **security layers must be independent and non-bypassable**. If the prisoner finds a way around one restriction, the others still contain them. This redundancy is not wasteful—it's essential for high-assurance security.



Existing Sandboxing Approaches

Before designing our process sandbox, we must understand existing approaches and their limitations. Each represents different trade-offs between security, performance, and complexity.

Approach	Security Level	Performance Overhead	Complexity	Resource Usage	Typical Use Cases
Virtual Machines	Very High	High (10-30%)	High	High (GB RAM)	Cloud isolation, OS-level sandboxing
System Containers	High	Medium (5-15%)	Medium	Medium (MB RAM)	Application packaging, microservices
Language Sandboxes	Medium	Low (1-5%)	Low	Low (KB RAM)	Browser scripts, embedded scripting
Process Isolation	High	Low (1-3%)	Medium	Low (KB RAM)	Plugin systems, code execution services

Virtual Machines

Virtual machines provide the **strongest isolation** by running untrusted code in a completely separate operating system instance. The hypervisor creates hardware-level isolation between the guest and host systems. Even if untrusted code achieves complete compromise of the guest OS, it remains contained within the virtual machine boundaries.

However, VMs have significant drawbacks for many use cases. They require **substantial resource overhead**—typically hundreds of megabytes of RAM and multiple CPU seconds for startup. This makes them impractical for short-lived tasks or high-frequency operations. Additionally, VM management introduces operational complexity around disk images, network configuration, and snapshot management.

VMs excel for **long-running workloads** where the startup cost can be amortized, such as cloud computing instances or development environments. They're less suitable for rapid code execution services or embedded sandboxing within applications.

System Containers

Technologies like Docker provide **OS-level virtualization** using Linux namespaces and cgroups. Containers share the host kernel but present isolated userspace environments. This approach offers better performance than VMs while maintaining strong isolation boundaries.

Container orchestration platforms like Kubernetes have made containers the standard for **application deployment and scaling**. However, containers have limitations for untrusted code execution. The shared kernel creates potential attack surfaces through system call interfaces. Container escape vulnerabilities, while rare, can provide full host system access.

Containers work well for **packaging and isolating trusted applications** but may not provide sufficient security guarantees for truly untrusted code. They also carry overhead from container runtime systems and image management.

Language Sandboxes

Many programming languages provide built-in sandboxing mechanisms. JavaScript's same-origin policy, Java's security manager, and Python's restricted execution modes attempt to limit code capabilities at the language runtime level.

Language sandboxes offer **excellent performance** since they operate within the same process as the host application. They enable fine-grained control over API access and can sandbox code with minimal overhead.

However, language sandboxes have fundamental limitations. They depend entirely on the **correctness of the runtime implementation**. Bugs in the interpreter or compiler can create bypass opportunities. Additionally, they typically cannot prevent resource exhaustion attacks or protect against vulnerabilities in native libraries.

Language sandboxes work well for **limited scripting scenarios** where performance is critical and the threat model is moderate, such as browser-based applications or configuration scripting.

Process Isolation

Process-level sandboxing uses operating system primitives to isolate untrusted code within separate processes. This approach leverages the OS kernel's built-in security boundaries while avoiding the overhead of full virtualization.

Process sandboxes can achieve **strong security guarantees** by combining multiple OS-level isolation mechanisms. They provide better performance than VMs while offering more robust security than language sandboxes. The kernel enforces isolation boundaries, making them less dependent on userspace code correctness.

The main challenge with process sandboxes is **complexity of configuration**. Properly configuring multiple security mechanisms requires deep understanding of OS internals and careful attention to interaction effects between different isolation layers.

> **Key Insight**: No single sandboxing approach is universally superior. The choice depends on specific requirements around security assurance, performance constraints, operational complexity, and resource availability. Process-level sandboxing occupies a sweet spot for many applications requiring strong security with reasonable performance.

Linux Security Primitives Overview

Linux provides several powerful security primitives that can be combined to create robust process sandboxes. Each primitive addresses different aspects of the isolation problem and operates through different kernel mechanisms.

Namespaces: Resource View Isolation

Linux **namespaces** provide isolated views of global system resources. When a process runs in a namespace, it sees a filtered or virtualized view of system state that can be completely different from the host system's view.

Think of namespaces as **alternate reality generators**. Just as science fiction depicts parallel universes where the same location contains different buildings and inhabitants, namespaces present alternative views where the same system appears to contain different processes, files, or network interfaces.

Namespace Type	Isolates	Security Benefit	Example Use
PID	Process IDs and process tree	Prevents process enumeration and signaling	Sandboxed process sees itself as PID 1
Mount	Filesystem mount points	Restricts filesystem access	Private /tmp directory
Network	Network interfaces and routing	Blocks network access	Isolated network stack
UTS	Hostname and domain name	Prevents hostname disclosure	Custom hostname inside sandbox
User	User and group IDs	Maps privileged IDs to unprivileged	Root inside maps to nobody outside
IPC	System V IPC objects	Isolates shared memory and semaphores	Private message queues

Namespaces are created using the `clone()` system call with special flags, or applied to existing processes with `unshare()`. The kernel maintains separate namespace instances and translates resource references based on the calling process's namespace membership.

The **security value** of namespaces comes from their ability to make system resources invisible or inaccessible to sandboxed processes. A process in a PID namespace cannot signal or inspect processes outside its namespace, even if it somehow learns their PIDs.

Seccomp: System Call Filtering

The **seccomp** (secure computing) mechanism allows processes to restrict which system calls they can make. Seccomp-BPF extends this with Berkeley Packet Filter programs that can examine system call arguments and make fine-grained filtering decisions.

Think of seccomp as a **strict gatekeeper** at the kernel boundary. Every time the sandboxed process wants to make a system call—the equivalent of requesting a service from the kernel—the seccomp filter examines the request. Like a bouncer at an exclusive club, it can allow entry, deny access, or even call security (kill the process) if the request is inappropriate.

Filter Action	Effect	When to Use
ALLOW	Permit system call	Safe operations like read/write
ERRNO	Return error code	Operations that should fail gracefully
TRAP	Send signal to process	Debugging or custom handling
KILL	Terminate process immediately	Dangerous operations like ptrace

Seccomp filters are **immutable once installed** and inherited by child processes. This prevents sandboxed code from removing its own restrictions. The BPF program runs in kernel space with minimal overhead, making filtering very efficient.

The **security benefit** is preventing sandboxed processes from using dangerous system calls that could enable privilege escalation, information disclosure, or system manipulation. Even if other isolation layers are bypassed, seccomp provides a kernel-enforced backstop.

Cgroups: Resource Limiting

Control groups (cgroups) provide hierarchical resource management and accounting. They allow setting limits on CPU time, memory usage, I/O bandwidth, and process counts for groups of processes.

Think of cgroups as a **resource allocation department** in a large organization. Just as departments get budget allocations for different expense categories—travel, equipment, personnel—cgroups assign resource quotas to process groups. When a group tries to exceed its allocation, the system either throttles the usage or denies the request entirely.

Controller	Limits	Enforcement Mechanism	Security Benefit
Memory	RAM and swap usage	OOM kill or throttling	Prevents memory exhaustion
CPU	Processing time and cycles	Scheduling priority reduction	Limits CPU consumption
PID	Maximum process count	Fork failures	Prevents fork bombs
I/O	Disk read/write bandwidth	Request queuing	Controls storage impact

Cgroups organize processes into a **hierarchy** where resource limits are inherited and can be subdivided. The kernel enforces limits automatically—processes cannot bypass or negotiate around their assigned quotas.

The **security value** comes from preventing resource exhaustion attacks. Without cgroups, a sandboxed process could consume all available memory, CPU time, or disk bandwidth, creating denial-of-service conditions that affect the entire host system.

Capabilities: Privilege Subdivision

Traditional Unix systems have a binary privilege model: processes run either as root (with all privileges) or as regular users (with limited privileges). Linux **capabilities** subdivide root privileges into discrete units that can be granted independently.

Think of capabilities as **security clearance levels** in a classified environment. Instead of having just "classified" and "unclassified" access, organizations have specific clearances like "secret communications," "top secret weapons," and "compartmentalized intelligence." Each clearance grants access to specific resources without providing universal access.

Capability	Privilege Granted	Typical Use	Security Risk if Granted
CAP_NET_RAW	Raw network socket access	Network diagnostics	Network sniffing, spoofing
CAP_SYS_ADMIN	System administration	Mount operations	Full system compromise
CAP_SETUID	Change user ID	Process privilege changes	Privilege escalation
CAP_SYS_PTRACE	Process tracing and debugging	Debuggers, profilers	Process memory access

Processes can **drop capabilities** permanently, preventing future privilege escalation even if vulnerabilities are exploited. The `PR_SET_NO_NEW_PRIVS` flag ensures that execve() cannot grant additional privileges.

The **security benefit** is implementing the principle of least privilege. Sandboxed processes can run with minimal capabilities, reducing the impact of successful exploits. Even if an attacker achieves code execution, they inherit the restricted capability set.

> **Architecture Principle**: These four primitives are complementary, not redundant. Namespaces control *what* processes can see, seccomp controls *what* they can do, cgroups control *how much* they can consume, and capabilities control *how privileged* their actions can be. Effective sandboxing requires orchestrating all four mechanisms in the correct sequence with appropriate configurations.

Common Integration Challenges:

⚠️ **Pitfall: Ordering Dependencies** The sequence of applying security mechanisms matters critically. For example, capabilities must be configured before entering certain namespaces, and seccomp filters must be installed before dropping the privileges needed to configure them. Incorrect ordering can result in partially-configured sandboxes or setup failures.

⚠️ **Pitfall: Privilege Requirements** Many security operations require elevated privileges to configure but should not grant those privileges to the sandboxed process. This creates a bootstrapping problem where the sandbox setup process needs careful privilege management to avoid granting unnecessary capabilities to the final sandboxed environment.

⚠️ **Pitfall: Resource Cleanup** Failed sandbox creation can leave orphaned namespaces, cgroup entries, or other kernel resources. Proper error handling must ensure cleanup of partially-configured sandboxes to prevent resource leaks and security inconsistencies.

The next sections will explore how to architect a system that properly orchestrates these primitives while avoiding these common pitfalls.

Implementation Guidance

This section provides practical guidance for understanding and working with Linux security primitives in preparation for building the process sandbox.

Technology Recommendations

Component	Simple Approach	Advanced Approach
System Calls	Direct syscalls with libc wrappers	Raw syscalls with custom wrappers
Configuration	Command-line flags and environment variables	Structured config files (JSON/YAML)
Error Handling	Simple perror() and exit()	Structured error codes with cleanup
Logging	Printf to stderr	Structured logging with levels
Testing	Manual verification scripts	Automated unit and integration tests

Development Environment Setup

Before implementing the sandbox, ensure your development environment supports the required kernel features:

```c
// Feature detection utility - save as check_features.c

#define _GNU_SOURCE

#include <unistd.h>

#include <sys/syscall.h>

#include <stdio.h>

#include <errno.h>


int main() {

    // Check namespace support

    if (syscall(SYS_unshare, 0) == -1 && errno != EINVAL) {

        printf("ERROR: Namespace support not available\n");

        return 1;

    }



    // Check seccomp support

    if (syscall(SYS_seccomp, 0, 0, NULL) == -1 && errno != EINVAL) {

        printf("ERROR: Seccomp support not available\n");

        return 1;

    }



    // Check cgroups v2 mount

    if (access("/sys/fs/cgroup/cgroup.controllers", R_OK) != 0) {

        printf("WARNING: Cgroups v2 not available, falling back to v1\n");

    }



    printf("All required features available\n");

    return 0;

}
```

Compile and run: `gcc -o check_features check_features.c && ./check_features`

Essential Header Files and Includes

```c
// sandbox_common.h - Common includes and definitions

#ifndef SANDBOX_COMMON_H

#define SANDBOX_COMMON_H


#define _GNU_SOURCE

#include <unistd.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <sys/mount.h>

#include <sys/prctl.h>

#include <sys/capability.h>

#include <sys/syscall.h>

#include <linux/seccomp.h>

#include <linux/filter.h>

#include <linux/sched.h>

#include <sched.h>

#include <signal.h>

#include <errno.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>


// Sandbox configuration structure

typedef struct {

    char *program_path;           // Path to program to execute

    char **program_args;          // Arguments to pass

    char *chroot_path;           // Root directory for filesystem isolation

    unsigned long memory_limit;   // Memory limit in bytes

    unsigned int cpu_percent;     // CPU percentage limit

    int enable_network;          // Whether to allow network access
```

```
    char **allowed_syscalls;     // Whitelist of allowed system calls
} sandbox_config_t;

// Sandbox runtime state
typedef struct {
    pid_t child_pid;             // PID of sandboxed process
    int namespace_fd[6];         // File descriptors for namespace references
    char *cgroup_path;           // Path to cgroup directory
    int cleanup_needed;          // Flag indicating cleanup is required
} sandbox_state_t;

// Error codes
#define SANDBOX_SUCCESS 0
#define SANDBOX_ERROR_CONFIG 1
#define SANDBOX_ERROR_PRIVILEGE 2
#define SANDBOX_ERROR_NAMESPACE 3
#define SANDBOX_ERROR_FILESYSTEM 4
#define SANDBOX_ERROR_SECCOMP 5
#define SANDBOX_ERROR_CGROUP 6
#define SANDBOX_ERROR_CAPABILITY 7
#define SANDBOX_ERROR_EXEC 8

#endif // SANDBOX_COMMON_H
```

Core Utility Functions

```c
// sandbox_utils.c - Utility functions for sandbox implementation

#include "sandbox_common.h"


// Initialize sandbox configuration with defaults

sandbox_config_t* sandbox_config_init() {

    sandbox_config_t *config = calloc(1, sizeof(sandbox_config_t));

    if (!config) return NULL;



    config->memory_limit = 64 * 1024 * 1024;  // 64MB default

    config->cpu_percent = 10;                  // 10% CPU default

    config->enable_network = 0;               // No network by default

    config->chroot_path = "/tmp/sandbox_root"; // Default chroot location



    return config;

}


// Clean up sandbox configuration

void sandbox_config_free(sandbox_config_t *config) {

    if (!config) return;



    free(config->program_path);

    free(config->chroot_path);



    if (config->program_args) {

        for (int i = 0; config->program_args[i]; i++) {

            free(config->program_args[i]);

        }

        free(config->program_args);

    }
```

```
    if (config->allowed_syscalls) {
        for (int i = 0; config->allowed_syscalls[i]; i++) {
            free(config->allowed_syscalls[i]);
        }
        free(config->allowed_syscalls);
    }

    free(config);
}

// Initialize sandbox runtime state
sandbox_state_t* sandbox_state_init() {
    sandbox_state_t *state = calloc(1, sizeof(sandbox_state_t));
    if (!state) return NULL;

    // Initialize all namespace FDs to -1 (invalid)
    for (int i = 0; i < 6; i++) {
        state->namespace_fd[i] = -1;
    }

    return state;
}

// Error handling utility
void sandbox_error(const char *message, int error_code) {
    fprintf(stderr, "SANDBOX ERROR [%d]: %s", error_code, message);
    if (errno) {
        fprintf(stderr, " - %s", strerror(errno));
    }
    fprintf(stderr, "\n");
}
```

```c
// Check if running as root (needed for many sandbox operations)
int check_root_privileges() {
    if (getuid() != 0) {
        sandbox_error("Root privileges required for sandbox setup", SANDBOX_ERROR_PRIVILEGE);
        return 0;
    }
    return 1;
}
```

Skeleton Implementation Structure

```c
// sandbox_main.c - Main sandbox orchestrator (to be implemented)

#include "sandbox_common.h"


// Main sandbox creation and execution function

int create_sandbox(sandbox_config_t *config, sandbox_state_t *state) {

    // TODO: Implement complete sandbox creation sequence

    // This function will orchestrate all security layers



    // TODO 1: Validate configuration parameters

    // - Check that program_path exists and is executable

    // - Verify chroot_path is a directory

    // - Validate resource limits are reasonable



    // TODO 2: Check privileges and capabilities

    // - Ensure running as root or with necessary capabilities

    // - Prepare for privilege dropping after setup



    // TODO 3: Create child process with clone() and namespace flags

    // - Use CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWNET | CLONE_NEWUTS

    // - Handle parent and child execution paths



    // TODO 4: Set up filesystem isolation (child process)

    // - Create minimal root filesystem

    // - Mount essential directories (/proc, /dev, /sys)

    // - Perform chroot or pivot_root



    // TODO 5: Configure seccomp filter (child process)

    // - Build BPF program from allowed_syscalls list

    // - Install seccomp filter with SECCOMP_SET_MODE_FILTER
```

```
    // TODO 6: Set up cgroups (parent process)
    // - Create cgroup directory
    // - Write memory and CPU limits
    // - Add child PID to cgroup

    // TODO 7: Drop capabilities (child process)
    // - Drop all capabilities except minimum required
    // - Set PR_SET_NO_NEW_PRIVS flag
    // - Change to unprivileged user/group

    // TODO 8: Execute target program (child process)
    // - execve() with provided program and arguments

    // TODO 9: Monitor and wait (parent process)
    // - Wait for child completion
    // - Handle signals and cleanup

    return SANDBOX_ERROR_CONFIG; // Placeholder - remove when implemented
}

// Clean up sandbox resources
void cleanup_sandbox(sandbox_state_t *state) {
    // TODO: Implement cleanup for all created resources
    // - Remove cgroup entries
    // - Close namespace file descriptors
    // - Clean up temporary filesystem
    // - Kill child process if still running
}

int main(int argc, char *argv[]) {
```

```
    if (argc < 2) {
        printf("Usage: %s <program> [args...]\n", argv[0]);
        return 1;
    }

    // TODO: Parse command line arguments into config
    sandbox_config_t *config = sandbox_config_init();
    sandbox_state_t *state = sandbox_state_init();

    if (!config || !state) {
        sandbox_error("Failed to allocate sandbox structures", SANDBOX_ERROR_CONFIG);
        return 1;
    }

    // TODO: Set config->program_path = argv[1] and config->program_args

    int result = create_sandbox(config, state);

    if (result == SANDBOX_SUCCESS) {
        printf("Sandbox executed successfully\n");
    } else {
        printf("Sandbox execution failed with code %d\n", result);
    }

    cleanup_sandbox(state);
    sandbox_config_free(config);
    free(state);

    return result;
```

```
}
```

Recommended Project Structure

Organize your sandbox implementation using this directory structure:

```
process-sandbox/
├── src/
│   ├── sandbox_common.h        ← Common definitions and structures
│   ├── sandbox_utils.c         ← Utility functions (provided above)
│   ├── sandbox_main.c          ← Main orchestrator (skeleton above)
│   ├── namespace_manager.c     ← Milestone 1: Namespace operations
│   ├── filesystem_isolator.c   ← Milestone 2: Filesystem isolation
│   ├── seccomp_filter.c        ← Milestone 3: System call filtering
│   ├── cgroup_manager.c        ← Milestone 4: Resource limits
│   └── capability_dropper.c    ← Milestone 5: Capability management
├── tests/
│   ├── test_basic.sh          ← Basic functionality tests
│   ├── test_isolation.sh      ← Isolation verification tests
│   └── test_security.sh       ← Security boundary tests
├── rootfs/                    ← Minimal root filesystem template
│   ├── bin/                   ← Essential binaries (sh, cat, ls)
│   ├── lib/                   ← Required libraries
│   ├── dev/                   ← Device nodes
│   └── proc/                  ← Will be mounted
└── Makefile                   ← Build configuration
```

Milestone Checkpoints

After reading this section, verify your understanding:

Environment Check:

1. Compile and run `check_features.c` to verify kernel support
2. Confirm you can run commands as root (needed for namespace creation)
3. Verify `/sys/fs/cgroup` exists (needed for cgroup management)

Code Setup:

1. Create the project structure above
2. Copy the provided utility functions and verify they compile cleanly: `gcc -c sandbox_utils.c`
3. Review the skeleton `sandbox_main.c` and understand the TODO sequence

Conceptual Understanding:

- Explain the prison analogy and how it maps to Linux security primitives
- Compare the trade-offs between VMs, containers, and process sandboxes
- Describe how namespaces, seccomp, cgroups, and capabilities provide complementary security

Next Steps: You're now ready to begin Milestone 1 (Process Namespaces). The skeleton code provides the structure you'll fill in, and the utility functions handle common operations you'll need throughout the implementation.

Goals and Non-Goals

> **Milestone(s):** This section provides foundational context for all milestones (1-5)

The success of any process sandbox depends on clearly defining what it must accomplish, what performance characteristics it should maintain, and equally importantly, what threats and use cases it explicitly does not address. This section establishes the boundaries and expectations for our defense-in-depth process sandbox, ensuring that both implementers and users understand the system's capabilities and limitations.

Mental Model: The Security Contract

Think of these goals as a security contract between the sandbox system and its users. Just as a legal contract specifies what services will be provided, under what conditions, and what circumstances void the agreement, our goals define the security guarantees we provide, the performance characteristics users can expect, and the attack scenarios that fall outside our protection model. This contract helps users make informed decisions about when and how to deploy the sandbox, while giving implementers clear success criteria for each milestone.

The process sandbox operates under the principle of **defense-in-depth**, where multiple independent security mechanisms work together to contain untrusted code. Each layer provides specific security guarantees, and the combination of all layers creates a robust isolation environment that can withstand various attack vectors.

Functional Goals

The functional goals define the core security and usability requirements that the process sandbox must satisfy. These goals directly map to the security mechanisms implemented across our five milestones and establish the fundamental capabilities that users can rely upon.

Process Isolation and Resource Visibility Control

The sandbox must create complete isolation of the sandboxed process's view of system resources, making it appear as if the process is running on a dedicated system while actually sharing hardware with the host. This isolation prevents untrusted code from discovering information about other processes, users, network configuration, or system topology that could be used for reconnaissance or lateral movement.

Isolation Type	Mechanism	Security Guarantee	Verification Method
Process Tree	PID namespace	Sandboxed process sees itself as PID 1 with no parent processes visible	Check `/proc` entries from inside sandbox
Hostname/Domain	UTS namespace	Sandboxed process cannot determine real hostname or domain information	Compare `hostname` output inside vs outside
Network Stack	Network namespace	Sandboxed process has no network interfaces except loopback	Check `ip addr` and `netstat` output
Filesystem View	Mount namespace + chroot	Sandboxed process sees only minimal filesystem with no host paths	Verify host directories are inaccessible
User/Group IDs	User namespace	Sandboxed process maps to unprivileged user on host system	Check effective UID/GID mappings

This isolation must be complete and tamper-proof. The sandboxed process should have no mechanism to discover that it's running in a container or to gather information about the host system's configuration, running processes, or network topology.

Filesystem Security and Access Control

The sandbox must provide a secure filesystem environment that prevents both data exfiltration and system compromise. This involves creating a minimal root filesystem with only essential binaries and libraries, while ensuring that the sandboxed process cannot access any host filesystem paths or sensitive system directories.

The filesystem isolation must prevent common container escape techniques such as:

- Traversing symbolic links that point outside the sandbox
- Accessing host filesystem through `/proc/*/root` or similar pseudo-filesystem entries
- Mounting host filesystems using privileged system calls
- Writing to system directories that could affect host system behavior

Filesystem Component	Configuration	Security Purpose	Implementation
Root Directory	Read-only minimal filesystem	Prevents tampering with system binaries	chroot or pivot_root
Writable Areas	tmpfs mounts at `/tmp`, `/var/tmp`	Provides scratch space without host persistence	Memory-backed filesystems
Device Access	Restricted `/dev` with minimal device nodes	Prevents access to hardware devices	Bind mount with filtered device list
Process Info	Read-only `/proc` mount	Allows process introspection within namespace	Mount with `hidepid` option
System Config	No access to `/etc`, `/sys`, `/boot`	Prevents system configuration discovery	Excluded from chroot environment

System Call Restriction and Argument Filtering

The sandbox must implement comprehensive system call filtering using seccomp-BPF to prevent dangerous operations while allowing legitimate computation. This filtering operates at two levels: blocking entire classes of system calls that should never be needed by sandboxed code, and filtering arguments of allowed system calls to prevent abuse.

The system call whitelist must be minimal but sufficient for basic computation, file I/O within the sandbox, and memory management. The filter must be architecture-aware, handling differences in system call numbers between x86_64, ARM, and other platforms.

System Call Category	Policy	Rationale	Examples
Process Management	Block dangerous calls	Prevent privilege escalation and system interference	`ptrace`, `clone` with namespace flags, `unshare`
Filesystem	Allow within sandbox only	Enable legitimate I/O while preventing host access	Allow `open` / `read` / `write`, block `mount` / `umount`
Network	Block all network calls	Prevent network-based attacks and data exfiltration	Block `socket`, `connect`, `bind`, `sendto`
Kernel Interface	Block kernel modification	Prevent system configuration changes	Block `sysctl`, `kexec_load`, `init_module`
Signal Handling	Allow basic signals only	Enable process control without system interference	Allow `sigaction` for SIGTERM/SIGINT, block `kill` with system PIDs

Resource Consumption Control

The sandbox must enforce strict resource limits to prevent denial-of-service attacks against the host system. These limits must be configurable but have safe defaults that prevent runaway processes from consuming all available system resources.

Resource Type	Limit Mechanism	Default Limit	Enforcement Behavior
Memory	cgroups memory controller	64MB resident set size	Kill process group when limit exceeded
CPU Time	cgroups CPU controller	10% of one CPU core	Throttle process when quota exceeded
Process Count	cgroups PID controller	32 total processes	Block `fork` / `clone` when limit reached
File Descriptors	`setrlimit` RLIMIT_NOFILE	64 open files	Return EMFILE when limit reached
Disk I/O	cgroups I/O controller	10MB/s read/write bandwidth	Throttle I/O operations when limit exceeded
Disk Space	Filesystem quotas on tmpfs	16MB total scratch space	Return ENOSPC when quota exhausted

Privilege Minimization

The sandbox must operate with the minimum privileges necessary for its functionality, dropping all Linux capabilities that are not essential for basic process execution. This implements the principle of least privilege to reduce the attack surface if other security mechanisms are bypassed.

The capability dropping must occur after all privileged setup operations are complete but before executing the untrusted code. The process must also set the `PR_SET_NO_NEW_PRIVS` flag to prevent privilege escalation through setuid binaries or other mechanisms.

Capability Category	Action	Rationale
All capabilities	Drop from effective, permitted, inheritable sets	Minimal privilege principle
Bounding set	Clear all capabilities	Prevent capability inheritance
Ambient set	Clear all capabilities	Prevent ambient capability grants
No-new-privs flag	Set to 1	Block future privilege escalation

Non-Functional Goals

The non-functional goals establish performance, resource usage, and operational requirements that ensure the sandbox is practical for real-world deployment. These goals balance security with usability, ensuring that the sandbox's security mechanisms don't create unacceptable overhead or operational complexity.

Performance and Latency Requirements

The sandbox creation and initialization process must complete quickly enough to support interactive use cases and batch processing workflows. While perfect performance is not required for a learning-focused implementation, the overhead should be reasonable for typical workloads.

Performance Metric	Target	Measurement Method
Sandbox Creation Time	Under 100ms for basic setup	Time from `create_sandbox()` call to process exec
Memory Overhead	Under 8MB additional RSS	Compare memory usage of sandboxed vs unsandboxed process
CPU Overhead	Under 5% additional CPU time	Compare execution time of identical workloads
I/O Overhead	Under 20% additional latency	Compare file read/write performance
System Call Overhead	Under 10μs per filtered call	Measure seccomp filter evaluation time

The performance targets recognize that security mechanisms inherently add overhead, but aim to keep this overhead low enough that the sandbox can be used for development, testing, and lightweight production workloads.

Resource Usage and Scalability

The sandbox system itself must use system resources efficiently, allowing multiple sandbox instances to run concurrently on the same host without excessive resource consumption. This enables use cases such as parallel test execution, multi-tenant development environments, and batch job processing.

Resource	Host Impact	Scaling Target
Memory per Sandbox	Under 16MB host memory overhead	Support 64 concurrent sandboxes on 4GB system
File Descriptors	Under 8 FDs per sandbox	Work within typical system limits
cgroup Hierarchy	One cgroup per sandbox	Clean up cgroups on sandbox termination
Namespace Resources	Namespace FDs held only during setup	Release namespace references after setup

Operational Simplicity

The sandbox must be deployable and manageable with minimal operational complexity. This includes clear configuration options, comprehensive error reporting, and automatic cleanup of system resources to prevent resource leaks.

Operational Aspect	Requirement	Implementation
Configuration	Single configuration struct	`sandbox_config_t` with sensible defaults
Error Reporting	Clear error messages with context	Specific error codes and descriptive messages
Resource Cleanup	Automatic cleanup on exit	`cleanup_sandbox()` function handles all resources
Privilege Requirements	Clear documentation of root requirements	`check_root_privileges()` validation function
Logging	Structured logging for troubleshooting	Log all security mechanism setup steps

Compatibility and Portability

The sandbox must work across common Linux distributions and kernel versions, while clearly documenting any specific requirements or limitations. This ensures that learners can complete the project on their development systems and that the resulting code is broadly applicable.

Compatibility Aspect	Requirement	Verification
Kernel Version	Linux 4.8+ (for modern cgroups and seccomp features)	Runtime kernel version check
Distribution	Ubuntu 18.04+, RHEL 8+, Debian 10+	Test on multiple distributions
Architecture	x86_64 primary, ARM64 secondary	Architecture-specific seccomp filter generation
Container Environment	Work inside Docker/Podman with privileged mode	Document container deployment requirements

Explicit Non-Goals

The explicit non-goals clearly define the security threats and use cases that fall outside the scope of this process sandbox. These limitations help set realistic expectations and prevent misuse of the sandbox in scenarios where it cannot provide adequate protection.

Hardware and Side-Channel Attacks

This process sandbox does not protect against attacks that exploit hardware vulnerabilities or side-channel information leakage. Such attacks require fundamentally different isolation mechanisms that operate below the operating system level.

Attack Class	Why Not Protected	Alternative Solutions
Spectre/Meltdown	Requires hardware-level isolation	Virtual machines with microcode updates
Cache Timing Attacks	Processes share CPU caches	Hardware partitioning or trusted execution environments
Power Analysis	Process sandbox cannot control hardware power states	Specialized secure hardware
Memory Layout Discovery	ASLR bypass techniques may still work within sandbox	Additional memory protection mechanisms

Kernel Privilege Escalation

The sandbox cannot protect against vulnerabilities in the Linux kernel itself. If untrusted code can exploit a kernel vulnerability to gain privileged access, the sandbox's user-space isolation mechanisms can be bypassed.

Vulnerability Class	Limitation	Risk Mitigation
Kernel Buffer Overflows	System call filtering cannot prevent all kernel bugs	Keep kernel updated, use kernel hardening
Use-After-Free in Kernel	Seccomp cannot prevent kernel memory corruption	Enable KASAN/KFENCE in development
Race Conditions in Kernel	Namespace isolation doesn't fix kernel races	Kernel fuzzing and static analysis
Device Driver Vulnerabilities	Hardware access restrictions don't cover all drivers	Minimize loaded kernel modules

> **Critical Limitation: Kernel Trust Boundary**
>
> The process sandbox operates entirely in user space and relies on the Linux kernel to enforce all security policies. Any vulnerability that allows untrusted code to execute with kernel privileges can completely bypass sandbox protections. This is an inherent limitation of operating system-level sandboxing.

Cryptographic and Protocol Security

The sandbox does not provide any cryptographic protections or secure communication channels. It focuses purely on process isolation and resource control, leaving encryption and authentication to higher-level application protocols.

Security Function	Not Provided	Reason
Data Encryption	No encryption of sandbox filesystem or memory	Not a process isolation concern
Authentication	No user authentication or access control	Assumes trusted operator deploying sandbox
Secure Communication	No encrypted channels between host and sandbox	Communication design left to applications
Key Management	No secure storage of cryptographic keys	Outside scope of process isolation

High-Performance Computing Workloads

The sandbox is designed for security rather than performance, and its resource restrictions and system call filtering make it unsuitable for high-performance computing applications that require direct hardware access or specialized system interfaces.

HPC Requirement	Sandbox Limitation	Impact
RDMA/InfiniBand Access	Network namespace blocks specialized networking	Cannot use high-performance interconnects
GPU Compute	Device restrictions prevent GPU access	No CUDA or OpenCL computation
Shared Memory IPC	IPC namespace isolation blocks shared memory	Cannot use MPI or similar frameworks
Real-Time Scheduling	cgroups CPU limits interfere with RT scheduling	Cannot guarantee real-time deadlines

Multi-Process Applications

While the sandbox can contain multiple processes within a single process tree, it is not designed for complex multi-process applications that require sophisticated inter-process communication or coordination across multiple sandbox instances.

Application Pattern	Limitation	Alternative Approach
Distributed Systems	Cannot coordinate across multiple sandbox instances	Use container orchestration platforms
Database Systems	Shared memory and file locking restrictions	Deploy in full virtual machines
Multi-Tenant Services	Single sandbox per tenant model only	Use application-level isolation
Legacy Applications	May require system services not available in sandbox	Refactor for sandboxed environment

Production-Grade Operational Features

This sandbox implementation focuses on demonstrating security concepts rather than providing enterprise-ready operational features. Production deployments would require additional monitoring, management, and integration capabilities.

Operational Feature	Not Included	Production Alternative
Metrics and Monitoring	Basic error reporting only	Use container runtime with monitoring
Log Aggregation	No centralized logging	Deploy with logging infrastructure
Health Checks	No automated health monitoring	Implement application-specific health checks
Auto-Scaling	No dynamic resource adjustment	Use orchestration platform auto-scaling
Backup/Recovery	No state persistence mechanisms	Design stateless applications

> **Design Philosophy: Education Over Production**
>
> This process sandbox prioritizes educational value and conceptual clarity over production-readiness. The goal is to demonstrate how Linux security primitives work together to create isolation, not to build a production container runtime. Learners should understand these limitations when considering real-world applications.

Implementation Guidance

This subsection provides practical guidance for implementing the goals and requirements defined above, with specific technology recommendations and code structure to support all project milestones.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Configuration Management	Static struct with compile-time defaults	JSON/YAML configuration file parsing
Error Reporting	Printf-style logging with error codes	Structured logging with log levels
Process Management	Direct fork/exec with basic signal handling	Process lifecycle management with proper reaping
Resource Monitoring	Basic resource limit enforcement	Real-time resource usage monitoring
Testing Framework	Simple assert-based unit tests	Comprehensive integration test suite

B. Recommended File Structure

```
process-sandbox/
├── src/
│   ├── main.c                  ← CLI entry point and argument parsing
│   ├── sandbox.h               ← Main header with all type definitions
│   ├── sandbox.c               ← Core sandbox orchestration logic
│   ├── config.c                ← Configuration management functions
│   ├── namespaces.c            ← Namespace creation and management
│   ├── filesystem.c            ← Filesystem isolation and chroot setup
│   ├── seccomp.c               ← System call filtering with BPF
│   ├── cgroups.c               ← Resource limits and cgroup management
│   ├── capabilities.c          ← Capability dropping and privilege management
│   └── utils.c                 ← Utility functions and error handling
├── tests/
│   ├── unit/                   ← Unit tests for individual components
│   ├── integration/            ← End-to-end sandbox testing
│   └── security/               ← Security validation tests
├── sandbox-root/               ← Minimal filesystem for chroot
│   ├── bin/                    ← Essential binaries
│   ├── lib/                    ← Required shared libraries
│   └── tmp/                    ← Writable scratch space
└── docs/
    └── debugging.md            ← Troubleshooting guide
```

C. Core Configuration Infrastructure

```c
#include <sys/types.h>
#include <unistd.h>

// Complete configuration structure with all required fields
typedef struct {
    char* program_path;           // Path to program to execute in sandbox
    char** program_args;          // Arguments to pass to program
    char* chroot_path;           // Path to minimal root filesystem
    unsigned long memory_limit;   // Memory limit in bytes
    unsigned int cpu_percent;     // CPU limit as percentage of one core
    int enable_network;          // 1 to allow network, 0 to block
    char** allowed_syscalls;     // NULL-terminated array of allowed system calls
} sandbox_config_t;

// Runtime state tracking for cleanup
typedef struct {
    pid_t child_pid;             // PID of sandboxed process
    int namespace_fd[6];         // File descriptors for namespace references
    char* cgroup_path;           // Path to cgroup directory
    int cleanup_needed;          // 1 if cleanup required, 0 if clean
} sandbox_state_t;

// Error codes for consistent error handling
#define SANDBOX_SUCCESS 0
#define SANDBOX_ERROR_CONFIG 1
#define SANDBOX_ERROR_PRIVILEGE 2
#define SANDBOX_ERROR_NAMESPACE 3
#define SANDBOX_ERROR_FILESYSTEM 4
#define SANDBOX_ERROR_SECCOMP 5
#define SANDBOX_ERROR_CGROUP 6
#define SANDBOX_ERROR_CAPABILITY 7
```

```
#define SANDBOX_ERROR_EXEC 8
```

D. Configuration Management Skeleton

```c
// Initialize sandbox configuration with secure defaults
sandbox_config_t* sandbox_config_init() {
    // TODO 1: Allocate memory for configuration structure
    // TODO 2: Set program_path to NULL (must be specified by user)
    // TODO 3: Set program_args to empty array
    // TODO 4: Set chroot_path to "./sandbox-root" default
    // TODO 5: Set memory_limit to 64MB (64 * 1024 * 1024)
    // TODO 6: Set cpu_percent to 10 (10% of one core)
    // TODO 7: Set enable_network to 0 (no network access)
    // TODO 8: Set allowed_syscalls to basic whitelist (read, write, exit, etc.)
    // TODO 9: Return configured struct pointer
}

// Clean up configuration memory
void sandbox_config_free(sandbox_config_t* config) {
    // TODO 1: Check for NULL pointer
    // TODO 2: Free program_path string if allocated
    // TODO 3: Free each string in program_args array
    // TODO 4: Free program_args array itself
    // TODO 5: Free chroot_path string
    // TODO 6: Free each string in allowed_syscalls array
    // TODO 7: Free allowed_syscalls array
    // TODO 8: Free config structure itself
}

// Initialize runtime state structure
sandbox_state_t* sandbox_state_init() {
    // TODO 1: Allocate memory for state structure
    // TODO 2: Set child_pid to 0 (no child process yet)
    // TODO 3: Initialize all namespace_fd entries to -1
    // TODO 4: Set cgroup_path to NULL
```

```
    // TODO 5: Set cleanup_needed to 0
    // TODO 6: Return state structure pointer
}
```

E. Main Sandbox Orchestration Skeleton

```c
// Main function that coordinates all sandbox setup steps
int create_sandbox(sandbox_config_t* config, sandbox_state_t* state) {
    // TODO 1: Validate configuration parameters (check required fields)
    // TODO 2: Check that process has root privileges for namespace operations
    // TODO 3: Fork child process using clone() with namespace flags
    // TODO 4: In child process: set up all security layers in correct order
    //         - Create and enter namespaces (PID, mount, network, UTS, user)
    //         - Set up filesystem isolation with chroot/pivot_root
    //         - Apply seccomp system call filters
    //         - Configure cgroups resource limits
    //         - Drop Linux capabilities
    //         - Execute target program
    // TODO 5: In parent process: store child PID and wait for completion
    // TODO 6: Return SANDBOX_SUCCESS or appropriate error code
}

// Clean up all sandbox resources
void cleanup_sandbox(sandbox_state_t* state) {
    // TODO 1: Kill child process if still running (SIGTERM then SIGKILL)
    // TODO 2: Wait for child process to exit (reap zombie)
    // TODO 3: Close namespace file descriptors
    // TODO 4: Remove cgroup directory and clean up cgroup resources
    // TODO 5: Free any allocated memory in state structure
    // TODO 6: Set cleanup_needed flag to 0
}
```

F. Error Handling and Validation

```c
// Centralized error reporting with context
void sandbox_error(const char* message, int error_code) {
    // TODO 1: Print error message with timestamp
    // TODO 2: Include errno information if system call failed
    // TODO 3: Log error code for automated processing
    // TODO 4: Include process PID for multi-process debugging
}

// Check if current process has required privileges
int check_root_privileges() {
    // TODO 1: Check effective UID is 0 (root)
    // TODO 2: Check for CAP_SYS_ADMIN capability
    // TODO 3: Test ability to create user namespace
    // TODO 4: Return 1 if sufficient privileges, 0 if not
    // Hint: Use getuid() and capget() system calls
}
```

G. Language-Specific Implementation Hints

- Use `clone()` system call with `CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWNET | CLONE_NEWUTS | CLONE_NEWUSER` flags for namespace creation
- Use `prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)` to prevent privilege escalation
- Use `capset()` system call to drop Linux capabilities after capability header setup
- Use `mount()` system call for filesystem operations, particularly `mount("none", "/proc", "proc", MS_RDONLY, NULL)` for read-only proc
- Use `chroot()` or `pivot_root()` for filesystem root changes - chroot is simpler but pivot_root is more secure
- Use `/sys/fs/cgroup` filesystem for cgroup manipulation, creating directories and writing limit values to control files
- Include error checking after every system call - most sandbox failures come from insufficient error handling

H. Milestone Checkpoints

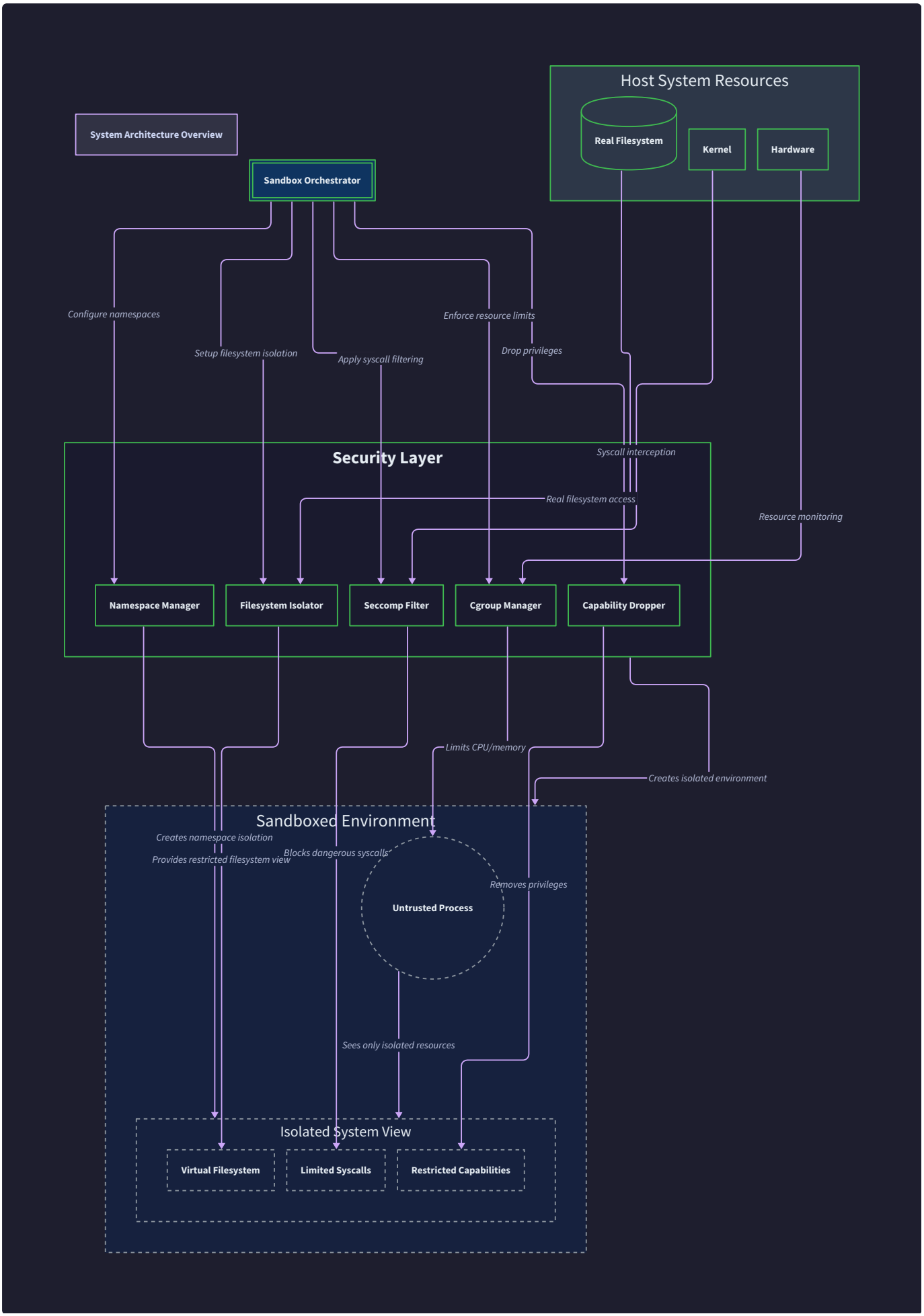
After implementing each milestone, verify functionality with these concrete tests:

Milestone	Verification Command	Expected Behavior	Failure Symptoms
1 (Namespaces)	`./sandbox --program=/bin/ps aux`	Process list shows only sandbox processes, PID 1 is the target program	See host processes or incorrect PID numbering
2 (Filesystem)	`./sandbox --program=/bin/ls /`	Only minimal filesystem visible, no host directories	Can see `/home`, `/root`, or other host paths
3 (Seccomp)	`./sandbox --program=/bin/strace -e network /bin/echo`	Network system calls blocked, process killed on violation	Network calls succeed or process hangs
4 (Cgroups)	Run memory-intensive program with 64MB limit	Process killed when exceeding memory limit	Process consumes unlimited memory
5 (Capabilities)	`./sandbox --program=/bin/mount -t tmpfs none /tmp`	Mount operation fails due to dropped capabilities	Mount succeeds (capabilities not properly dropped)

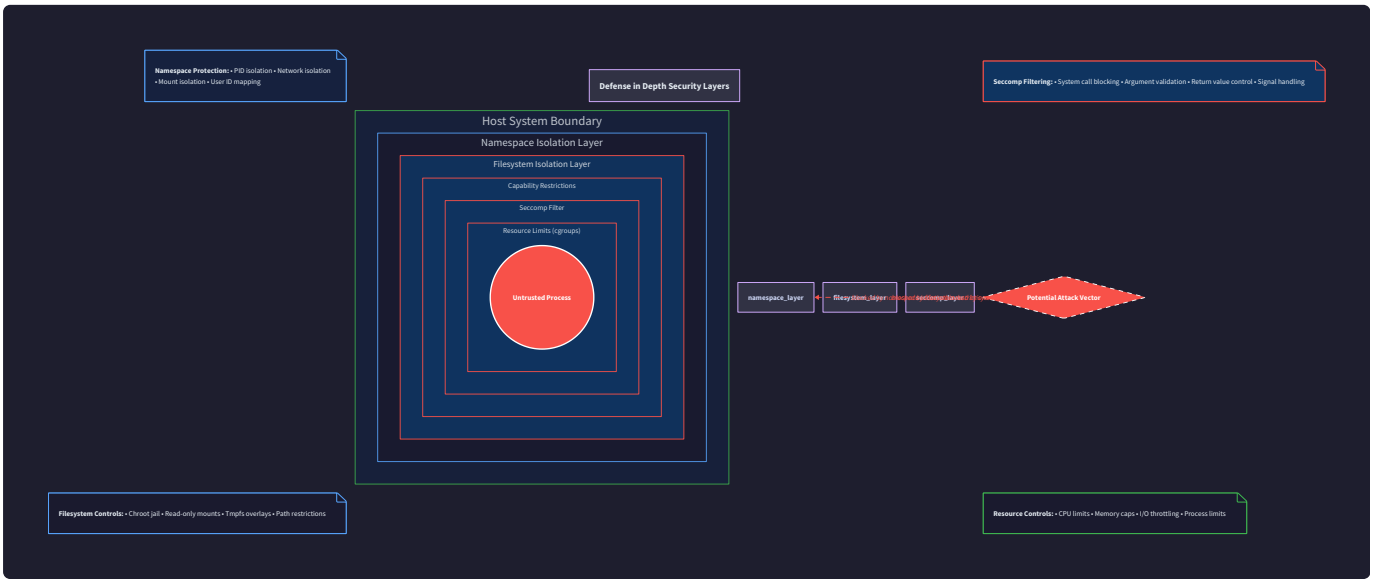
High-Level Architecture

> **Milestone(s):** This section provides architectural foundation for all milestones (1-5)

The process sandbox architecture follows a **defense-in-depth** strategy, orchestrating multiple Linux security primitives to create layered isolation around untrusted code. Think of this system as a high-security facility with multiple independent checkpoints: even if one security mechanism fails, the others continue to protect the host system. Each security layer operates independently but must be carefully coordinated to avoid conflicts and ensure proper initialization order.


System Architecture Overview
Host System Resources
Real Filesystem
Kernel
Hardware
Sandbox Orchestrator
Configure namespaces
Setup filesystem isolation
Apply syscall filtering
Enforce resource limits
Drop privileges
Security Layer
Syscall interception
Real filesystem access
Resource monitoring
Namespace Manager
Filesystem Isolator
Seccomp Filter
Cgroup Manager
Capability Dropper
Limits CPU/memory
Creates isolated environment
Sandboxed Environment
Creates namespace isolation
Provides restricted filesystem view
Blocks dangerous syscalls
Removes privileges
Untrusted Process
Sees only isolated resources
Isolated System View
Virtual Filesystem
Limited Syscalls
Restricted Capabilities

The architecture centers around a **Sandbox Orchestrator** that coordinates five specialized security components. This orchestrator acts as the conductor of a security orchestra, ensuring each component plays its part at precisely the right moment in the sandbox creation sequence. The orchestrator maintains both configuration state (what the sandbox should look like) and runtime state (tracking active resources that need cleanup).



The security layers form concentric rings of protection around the untrusted process. The outermost layer (namespaces) provides the broadest isolation by changing the process's view of system resources. Moving inward, filesystem isolation restricts file access, seccomp filters system calls at the kernel boundary, cgroups limit resource consumption, and capability dropping removes dangerous privileges. This layered approach ensures that exploiting the sandbox requires defeating multiple independent security mechanisms simultaneously.

Component Responsibilities

The process sandbox divides security responsibilities among six primary components, each with clearly defined ownership boundaries and interfaces. This separation allows each component to focus on a specific aspect of security while maintaining clean integration points.

Component	Primary Responsibility	Resources Managed	Key Dependencies
Sandbox Orchestrator	Coordinates all security components and manages sandbox lifecycle	Process state, cleanup handlers, error reporting	All other components
Namespace Manager	Creates isolated views of system resources	PID, mount, network, UTS, user namespaces	clone() system call, root privileges
Filesystem Isolator	Provides minimal isolated filesystem environment	Root filesystem, mount points, file permissions	Mount namespace, chroot/pivot_root
Seccomp Filter	Restricts allowed system calls using BPF programs	BPF filter programs, system call whitelist	prctl() system call, BPF compiler
Cgroup Manager	Enforces resource limits on CPU, memory, I/O	Cgroup hierarchies, controller limits	Cgroup filesystem, root privileges
Capability Dropper	Removes dangerous Linux capabilities	Process capability sets, user/group IDs	capset() system call, privilege flags

Sandbox Orchestrator Responsibilities

The **Sandbox Orchestrator** serves as the central coordinator and maintains the master state for each sandbox instance. It owns the `sandbox_config_t` structure containing user-specified settings and the `sandbox_state_t` structure tracking runtime resources. The orchestrator's primary responsibilities include validating configuration parameters, calling security components in the correct sequence, handling initialization failures with proper cleanup, and providing a unified error reporting interface.

The orchestrator must handle complex error scenarios where some security mechanisms succeed while others fail. For example, if namespace creation succeeds but filesystem isolation fails, the orchestrator must properly clean up the created namespaces before returning an error. This requires maintaining detailed cleanup state and implementing rollback logic for partial failures.

Orchestrator Function	Input Parameters	Return Value	Side Effects
`sandbox_config_init()`	None	`sandbox_config_t*`	Allocates configuration with defaults
`sandbox_config_free()`	`sandbox_config_t*`	void	Deallocates configuration memory
`sandbox_state_init()`	None	`sandbox_state_t*`	Allocates runtime state structure
`create_sandbox()`	`sandbox_config_t*`, `sandbox_state_t*`	int (error code)	Creates child process, applies all security layers
`cleanup_sandbox()`	`sandbox_state_t*`	void	Releases namespaces, cgroups, kills processes
`check_root_privileges()`	None	int (boolean)	Checks for required root privileges

Security Component Interface Pattern

Each security component follows a consistent interface pattern that simplifies orchestration and error handling. Every component provides initialization, application, and cleanup functions that can be called independently. This pattern allows the orchestrator to apply security mechanisms incrementally and handle failures at any stage.

The component interface pattern ensures that each security layer can be tested in isolation and that new security mechanisms can be added without modifying existing components. Each component validates its own preconditions (such as required privileges or kernel features) and reports specific error codes that help with debugging.

Security Layer Ordering

The sequence of applying security mechanisms is critical to the sandbox's effectiveness and stability. Applying these layers in the wrong order can result in privilege escalation vulnerabilities, resource leaks, or complete sandbox initialization failure. The ordering requirements stem from dependencies between security mechanisms and the way Linux kernel security checks interact.

> **Critical Insight**: Some security mechanisms require privileges that other mechanisms remove. For example, creating cgroups requires root privileges, but capability dropping removes those privileges. The ordering must ensure that privilege-requiring operations happen before capability dropping.

The correct initialization sequence follows a specific pattern based on privilege requirements and resource dependencies:

1. **Privilege Verification** - Check for required root privileges before attempting any security operations
2. **Process Forking** - Create the child process that will become the sandboxed environment
3. **Namespace Creation** - Establish isolated views of system resources while retaining privileges
4. **Filesystem Setup** - Configure isolated filesystem environment using mount namespace
5. **Cgroup Configuration** - Set resource limits while still having cgroup management privileges
6. **Seccomp Installation** - Install system call filter as late as possible to avoid blocking required setup calls
7. **Capability Dropping** - Remove dangerous privileges as the final step before executing user code

> **Decision: Apply Seccomp Filter Last Among Security Mechanisms**
>
> - **Context**: Seccomp filters can block system calls required by other security setup operations
> - **Options Considered**: Apply seccomp first (maximum protection), apply seccomp in middle (balanced), apply seccomp last (compatibility)
> - **Decision**: Apply seccomp filter just before capability dropping and program execution
> - **Rationale**: Many security setup operations require system calls that should be blocked during normal execution (like mount, setrlimit, prctl). Applying seccomp too early would block legitimate setup operations.
> - **Consequences**: Brief window where setup code runs without system call filtering, but all other security layers are active. Setup code is trusted and minimal.

Namespace Creation Dependencies

Namespace creation must happen early because other security mechanisms depend on namespace isolation. The mount namespace must exist before filesystem isolation can configure bind mounts and chroot environments. The PID namespace should be created before cgroup assignment to ensure proper process hierarchy tracking.

Namespace Type	Must Create Before	Reason
User	All other namespaces	Provides privilege mapping for other namespace operations
PID	Cgroup assignment	Ensures cgroup sees correct process hierarchy
Mount	Filesystem isolation	Required for chroot and bind mount operations
Network	Seccomp filter	Some seccomp filters check network-related system calls
UTS	Program execution	Hostname isolation should be established before user code runs

Privilege Dropping Timing

Capability dropping must be the last security operation because it removes privileges required by other security mechanisms. Once capabilities are dropped, the process cannot create new namespaces, configure cgroups, or perform

other privileged operations. The `PR_SET_NO_NEW_PRIVS` flag must be set before loading seccomp filters, but this doesn't remove existing capabilities.

The privilege dropping sequence within its own phase also matters: first change supplementary groups, then primary group, then user ID, then drop capabilities. This order prevents the process from being unable to complete privilege changes due to insufficient permissions.

Resource Limit Application Order

Cgroup limits should be applied after namespace creation but before program execution to ensure the resource limits apply to the correct process hierarchy. Memory limits must be set before the program starts to prevent initial memory allocation from exceeding limits. CPU limits can be applied at any time but are most effective when set before the program begins executing.

> **Decision: Configure Cgroups After Namespaces But Before Seccomp**
>
> - **Context**: Cgroup operations require various system calls and file system access that might be restricted
> - **Options Considered**: Configure cgroups first (early protection), configure cgroups after everything (late application), configure cgroups mid-sequence (balanced)
> - **Decision**: Configure cgroups after namespace creation but before seccomp filter installation
> - **Rationale**: Cgroup setup requires file system operations that seccomp might block, but cgroups should be active before user code runs. Namespaces must exist first to provide proper isolation context.
> - **Consequences**: Resource limits are active during seccomp installation and capability dropping, providing protection against resource exhaustion during these phases.

Recommended File Structure

The codebase organization reflects the component-based architecture and supports incremental development through the milestone sequence. The structure separates core sandbox logic from supporting utilities and provides clear integration points for testing and debugging.

```
process-sandbox/
├── src/                          # Core implementation
│   ├── main.c                    # Entry point and command-line interface
│   ├── sandbox.h                 # Public API and configuration structures
│   ├── sandbox.c                 # Main orchestrator implementation
│   ├── namespace.h               # Namespace management interface
│   ├── namespace.c               # Namespace creation and management
│   ├── filesystem.h              # Filesystem isolation interface
│   ├── filesystem.c              # Chroot/pivot_root implementation
│   ├── seccomp.h                 # System call filtering interface
│   ├── seccomp.c                 # BPF filter generation and installation
│   ├── cgroups.h                 # Resource limiting interface
│   ├── cgroups.c                 # Cgroup creation and limit enforcement
│   ├── capabilities.h            # Privilege management interface
│   └── capabilities.c            # Capability dropping implementation
├── include/                      # Shared header files
│   ├── common.h                  # Common types and constants
│   ├── errors.h                  # Error codes and reporting
│   └── utils.h                   # Utility functions
├── lib/                          # Supporting utilities
│   ├── utils.c                   # Common utility implementations
│   ├── logging.c                 # Logging and debugging support
│   └── cleanup.c                 # Resource cleanup utilities
├── tests/                        # Test suite
│   ├── unit/                     # Component unit tests
│   │   ├── test_namespace.c
│   │   ├── test_filesystem.c
│   │   ├── test_seccomp.c
│   │   ├── test_cgroups.c
│   │   └── test_capabilities.c
│   ├── integration/              # Full sandbox tests
│   │   ├── test_basic_execution.c
│   │   ├── test_security_bypass.c
│   │   └── test_resource_limits.c
│   └── fixtures/                 # Test data and configurations
│       ├── minimal_rootfs/       # Test filesystem environments
│       └── test_programs/        # Programs for testing sandbox
├── rootfs/                       # Minimal root filesystem template
│   ├── bin/                      # Essential binaries
│   ├── lib/                      # Required libraries
│   ├── dev/                      # Device nodes template
│   └── proc/                     # Proc filesystem mount point
├── examples/                     # Example configurations and usage
│   ├── basic_sandbox.c           # Simple sandbox example
│   ├── web_service_sandbox.c     # Network service sandboxing
│   └── compiler_sandbox.c        # Code compilation sandboxing
├── scripts/                      # Build and development tools
│   ├── setup_rootfs.sh          # Script to create minimal root filesystem
│   ├── test_runner.sh            # Comprehensive test execution
│   └── benchmark.sh              # Performance measurement tools
├── docs/                         # Documentation
│   ├── api.md                    # API reference
│   ├── security.md               # Security model explanation
│   └── debugging.md              # Troubleshooting guide
├── Makefile                      # Build configuration
└── README.md                     # Project overview and quick start
```

Core Implementation Organization

The `src/` directory contains the main sandbox implementation with each security component in its own source file. This organization supports the milestone-based development approach where learners can implement one security mechanism at a time. The header files define clean interfaces between components and make dependencies explicit.

The `sandbox.h` and `sandbox.c` files contain the orchestrator logic and main API. These files depend on all other components but provide a single entry point for sandbox creation. The orchestrator handles the complex initialization sequence and error handling that coordinates all security mechanisms.

File	Primary Purpose	Key Structures	Dependencies
`main.c`	Command-line interface and program entry	Command-line parsing	`sandbox.h`
`sandbox.h/c`	Orchestrator and main API	`sandbox_config_t`, `sandbox_state_t`	All component headers
`namespace.h/c`	Namespace isolation	Namespace file descriptors	`clone()`, `unshare()`
`filesystem.h/c`	Filesystem isolation	Mount points, chroot state	`mount()`, `chroot()`
`seccomp.h/c`	System call filtering	BPF programs, filter rules	`prctl()`, BPF utilities
`cgroups.h/c`	Resource limits	Cgroup paths, limit values	Cgroup filesystem
`capabilities.h/c`	Privilege dropping	Capability sets, user IDs	`capset()`, `setuid()`

Supporting Infrastructure Organization

The `lib/` directory contains utility functions that multiple components use but that aren't part of the core security logic. This includes error reporting, resource cleanup, and debugging utilities. The `include/` directory contains shared definitions and constants used across multiple components.

The separation between core logic and supporting utilities helps maintain focus on the security mechanisms while providing necessary infrastructure. The utility functions handle cross-cutting concerns like logging and memory management that every component needs.

Testing and Validation Structure

The `tests/` directory supports both unit testing of individual components and integration testing of the complete sandbox. Unit tests can verify each security mechanism in isolation, while integration tests validate that the components work together correctly and provide the expected security properties.

The `fixtures/` directory contains test data including minimal root filesystems and test programs that exercise different aspects of the sandbox. This structure supports automated testing and provides concrete examples of sandbox configurations.

Decision: Separate Unit Tests by Component

- **Context**: Each security component has distinct functionality that can be tested independently
- **Options Considered**: Single test file (simple), test per function (granular), test per component (balanced)
- **Decision**: Create separate test files for each major component with comprehensive test cases
- **Rationale**: Component-level testing matches the architecture, allows parallel test development during milestone implementation, and makes it easy to identify which security mechanism has issues
- **Consequences**: More test files to maintain, but better test organization and easier debugging when specific security mechanisms fail

Development and Deployment Support

The `scripts/` directory contains automation for common development tasks including building minimal root filesystems, running comprehensive tests, and measuring performance. The `examples/` directory demonstrates different sandbox configurations for common use cases.

This structure supports both learning (clear examples and comprehensive tests) and production usage (deployment scripts and performance tools). The examples show how to configure the sandbox for different security requirements and workload types.

Implementation Guidance

The process sandbox implementation requires careful attention to Linux-specific system calls and security mechanisms. The following guidance provides practical recommendations for building each component and integrating them effectively.

Technology Recommendations

Component	Simple Option	Advanced Option	Rationale
BPF Filter Generation	Hand-written BPF assembly	libseccomp library	Hand-written provides learning value; libseccomp for production
Configuration Format	Command-line arguments	JSON/YAML config files	Command-line for simplicity; files for complex configurations
Logging	fprintf() to stderr	Structured logging library	Basic logging sufficient for learning project
Error Handling	Integer return codes	Detailed error structures	Return codes align with C conventions and system calls
Root Filesystem	Manual directory creation	Buildroot or similar tool	Manual creation for learning; automated tools for production

Core Data Structures

The sandbox system centers around two main configuration and state structures that all components use:

```c
#include <sys/types.h>
#include <unistd.h>

// Configuration structure defining sandbox parameters
typedef struct {
    char* program_path;           // Path to program to execute
    char** program_args;          // Arguments to pass to program
    char* chroot_path;           // Root filesystem path
    unsigned long memory_limit;   // Memory limit in bytes
    unsigned int cpu_percent;     // CPU percentage limit (0-100)
    int enable_network;          // Allow network access (0 or 1)
    char** allowed_syscalls;     // Whitelist of allowed system calls
} sandbox_config_t;

// Runtime state tracking active sandbox resources
typedef struct {
    pid_t child_pid;             // PID of sandboxed process
    int namespace_fd[6];         // File descriptors for namespaces
    char* cgroup_path;           // Path to cgroup directory
    int cleanup_needed;          // Flags indicating resources to cleanup
} sandbox_state_t;

// Error codes for sandbox operations
#define SANDBOX_SUCCESS 0
#define SANDBOX_ERROR_CONFIG 1
#define SANDBOX_ERROR_PRIVILEGE 2
#define SANDBOX_ERROR_NAMESPACE 3
#define SANDBOX_ERROR_FILESYSTEM 4
#define SANDBOX_ERROR_SECCOMP 5
#define SANDBOX_ERROR_CGROUP 6
#define SANDBOX_ERROR_CAPABILITY 7
```

```
#define SANDBOX_ERROR_EXEC 8
```

Orchestrator Implementation Framework

The main orchestrator coordinates all security components and handles the complex initialization sequence:

```c
#include "sandbox.h"
#include "namespace.h"
#include "filesystem.h"
#include "seccomp.h"
#include "cgroups.h"
#include "capabilities.h"

// Initialize configuration with safe defaults
sandbox_config_t* sandbox_config_init() {
    sandbox_config_t* config = malloc(sizeof(sandbox_config_t));
    if (!config) return NULL;

    // TODO 1: Set default values for all configuration fields
    // TODO 2: Initialize program_path and program_args to NULL
    // TODO 3: Set default chroot_path to "./rootfs"
    // TODO 4: Set reasonable default memory_limit (e.g., 64MB)
    // TODO 5: Set default cpu_percent to 10
    // TODO 6: Disable network access by default
    // TODO 7: Initialize allowed_syscalls to basic whitelist

    return config;
}

// Main sandbox creation function coordinating all security layers
int create_sandbox(sandbox_config_t* config, sandbox_state_t* state) {
    // TODO 1: Validate configuration parameters for completeness and safety
    // TODO 2: Check for required root privileges using check_root_privileges()
    // TODO 3: Fork child process and store PID in state->child_pid
    // TODO 4: In parent: set up monitoring and return success
    // TODO 5: In child: Apply security layers in correct order:
    //         - Create namespaces (all types: PID, mount, network, UTS, user)
```

```
    //          - Set up filesystem isolation (chroot/pivot_root)
    //          - Configure cgroups and resource limits
    //          - Install seccomp filter
    //          - Drop capabilities
    //          - Execute target program
    // TODO 6: Handle failures at any stage with proper cleanup
    // TODO 7: Set appropriate error codes for different failure types

    return SANDBOX_SUCCESS;
}

// Clean up all sandbox resources
void cleanup_sandbox(sandbox_state_t* state) {
    // TODO 1: Kill child process if still running
    // TODO 2: Close namespace file descriptors
    // TODO 3: Remove cgroup directory and limits
    // TODO 4: Clean up any temporary filesystem mounts
    // TODO 5: Free allocated memory in state structure

    // Hint: Use kill(state->child_pid, SIGTERM) then SIGKILL if needed
    // Hint: Check state->cleanup_needed flags to see what needs cleanup
}
```

Component Integration Pattern

Each security component follows a consistent pattern for integration with the orchestrator:

```c
// Example pattern for namespace component
typedef struct {
    int pid_ns_fd;
    int mount_ns_fd;
    int net_ns_fd;
    int uts_ns_fd;
    int user_ns_fd;
    int initialized;
} namespace_state_t;

// Initialize namespace isolation
int create_namespaces(namespace_state_t* ns_state) {
    // TODO 1: Create user namespace first for privilege mapping
    // TODO 2: Create PID namespace for process isolation
    // TODO 3: Create mount namespace for filesystem isolation
    // TODO 4: Create network namespace for network isolation
    // TODO 5: Create UTS namespace for hostname isolation
    // TODO 6: Store namespace file descriptors for cleanup
    // TODO 7: Set initialized flag on success

    // Hint: Use clone() with CLONE_NEWPID | CLONE_NEWMOUNT | etc.
    // Hint: Open /proc/self/ns/* files to get namespace file descriptors

    return SANDBOX_SUCCESS;
}
```

Error Handling and Debugging Infrastructure

```c
#include <errno.h>

#include <string.h>


// Centralized error reporting with context

void sandbox_error(const char* operation, int error_code) {

    // TODO 1: Print operation name and error code

    // TODO 2: Include errno and strerror() for system call failures

    // TODO 3: Add timestamp and PID for debugging

    // TODO 4: Consider logging to file for production usage



    // Example: "SANDBOX ERROR: create_namespaces failed with code 3: Operation not permitted"

}


// Privilege checking utility

int check_root_privileges() {

    // TODO 1: Check if running as root (getuid() == 0)

    // TODO 2: Verify required capabilities are available

    // TODO 3: Test ability to create namespaces (try and rollback)

    // TODO 4: Check for cgroup write permissions

    // TODO 5: Return 1 if all privileges available, 0 otherwise



    // Hint: Use getuid(), geteuid(), and access() to check permissions

    // Hint: Try opening /proc/sys/user/max_user_namespaces for namespace support



    return 0; // Replace with actual implementation

}
```

Milestone Checkpoints

After implementing each component, verify functionality with these specific tests:

Milestone 1 - Namespace Isolation:

- Run: `sudo ./sandbox /bin/sh` and execute `echo $$` (should print "1")

- Verify: `cat /proc/self/status | grep NSpid` shows isolated PID namespace
- Test: `ip addr` should show only loopback interface (network isolation)
- Check: `hostname` should show isolated hostname different from host

Milestone 2 - Filesystem Isolation:

- Run: `sudo ./sandbox /bin/ls /` (should show minimal rootfs, not host filesystem)
- Verify: Attempting to access `/home` or other host directories fails
- Test: Create file in sandbox - should not appear on host filesystem
- Check: `mount` command shows sandbox-specific mounts only

Milestone 3 - System Call Filtering:

- Run sandbox with program that attempts forbidden system call
- Verify: Process killed with SIGSYS signal when making blocked system call
- Test: `strace -f ./sandbox /bin/echo hello` shows only whitelisted system calls
- Check: Attempt to use `ptrace()` or other dangerous calls fails immediately

Milestone 4 - Resource Limits:

- Run memory-intensive program exceeding configured limit
- Verify: Process killed by OOM killer when exceeding memory limit
- Test: CPU-intensive program should be throttled to configured percentage
- Check: `cat /proc/<pid>/cgroup` shows sandbox cgroup membership

Milestone 5 - Capability Dropping:

- Run: `capsh --print` inside sandbox shows minimal capability set
- Verify: Attempts to mount filesystems or change networking fail with permission errors
- Test: `cat /proc/self/status | grep Cap` shows dropped capabilities
- Check: Programs cannot escalate privileges even with setuid binaries

⚠️ **Common Pitfalls**:

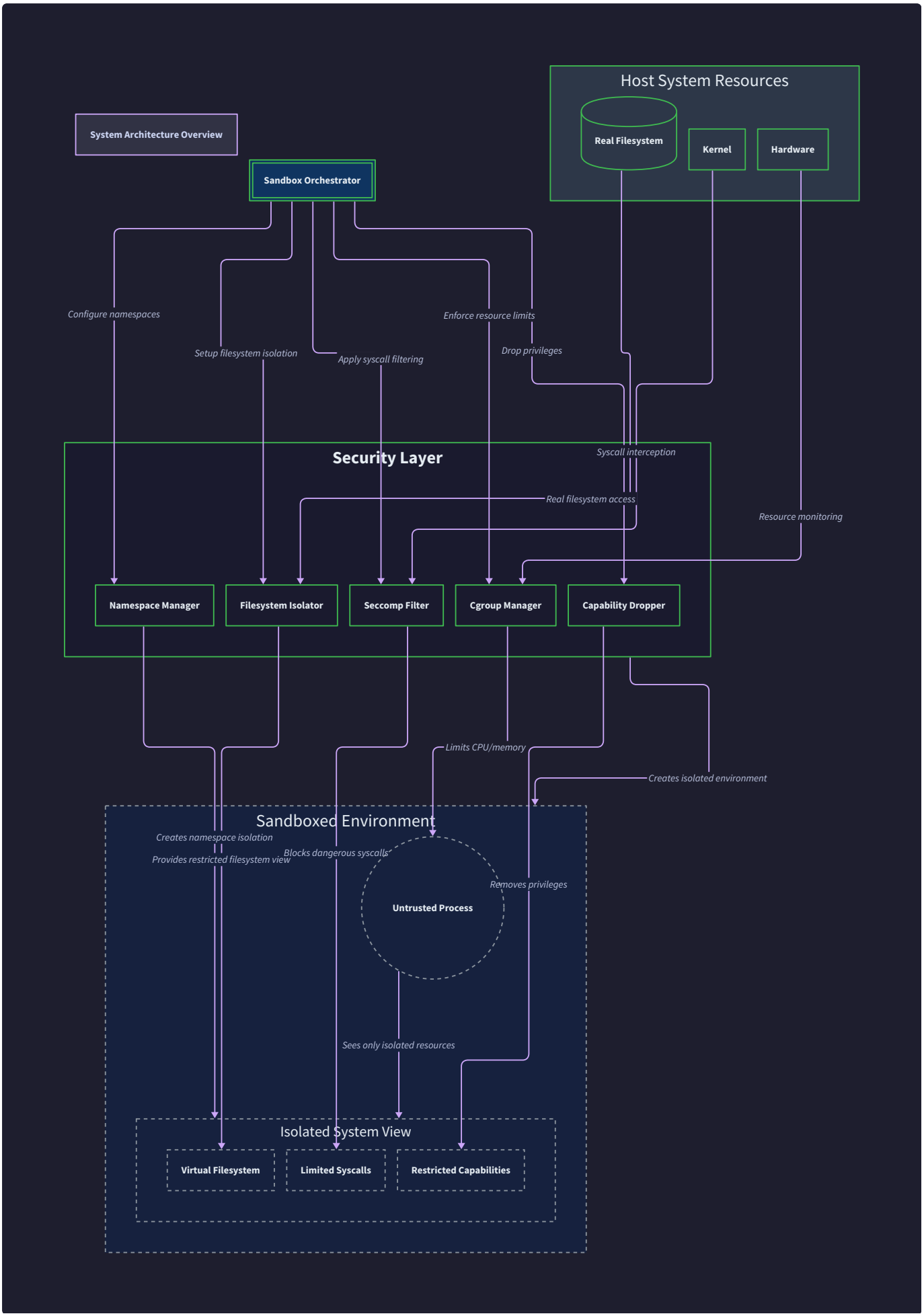
- **Forgetting SIGCHLD in clone()**: The parent process won't be notified when the child exits, leading to zombie processes. Always include `SIGCHLD` in the clone flags.
- **Not mounting /proc in new namespace**: Many utilities expect /proc to be mounted. Create the mount after entering the new mount namespace with `mount("proc", "/proc", "proc", 0, NULL)`.
- **Applying seccomp before setup is complete**: System call filtering can block legitimate setup operations. Install the seccomp filter as the last step before executing user code.
- **Missing library dependencies in chroot**: The minimal root filesystem must include all shared libraries required by the target program. Use `ldd` to identify dependencies and copy them to the chroot environment.
- **Incorrect capability dropping order**: Drop capabilities after changing user/group IDs. The sequence should be: `setgroups()` → `setgid()` → `setuid()` → `capset()`.

Data Model

> **Milestone(s):** This section provides data structures for all milestones (1-5), with specific focus on configuration (Milestones 1-2) and runtime state management (Milestones 3-5)

The data model of the process sandbox defines the fundamental structures that capture configuration intent and runtime state throughout the sandbox lifecycle. Think of the data model as the **sandbox's memory** - it stores everything the system needs to know about what the sandbox should look like (configuration) and what it actually looks like at any moment in time (runtime state). Just as a construction foreman needs blueprints (configuration) and progress tracking sheets (runtime state) to build a house, the sandbox orchestrator needs structured data to create and manage isolated execution environments.

The data model serves two critical purposes in the sandbox architecture. First, it provides a **declarative configuration interface** that allows users to specify their isolation requirements without understanding the underlying Linux primitives. Second, it maintains **runtime state tracking** that enables proper resource cleanup and monitoring throughout the sandbox lifecycle. This separation between desired state (configuration) and actual state (runtime) follows established patterns from container orchestration systems and ensures the sandbox can recover gracefully from failures.


System Architecture Overview
Host System Resources
Real Filesystem
Kernel
Hardware
Sandbox Orchestrator
Configure namespaces
Setup filesystem isolation
Apply syscall filtering
Enforce resource limits
Drop privileges
Security Layer
Syscall interception
Real filesystem access
Resource monitoring
Namespace Manager
Filesystem Isolator
Seccomp Filter
Cgroup Manager
Capability Dropper
Limits CPU/memory
Creates isolated environment
Sandboxed Environment
Creates namespace isolation
Provides restricted filesystem view
Blocks dangerous syscalls
Removes privileges
Untrusted Process
Sees only isolated resources
Isolated System View
Virtual Filesystem
Limited Syscalls
Restricted Capabilities

The complexity of the sandbox data model stems from the need to coordinate multiple independent Linux security mechanisms. Each security layer - namespaces, filesystem isolation, seccomp filtering, resource limits, and capability management - requires its own configuration parameters and runtime state tracking. The challenge lies in designing structures that are both comprehensive enough to support all security features and simple enough for users to understand and configure correctly.

Sandbox Configuration

The **sandbox configuration** represents the declarative specification of how a sandbox should be constructed. Think of it as the **architectural blueprint** for the security prison - it specifies the cell dimensions (resource limits), the permitted activities (allowed system calls), the view from the windows (namespace configuration), and the items allowed inside (filesystem access). Unlike imperative commands that describe how to build something, the configuration describes what the end result should look like, leaving the implementation details to the sandbox orchestrator.

The configuration structure must balance several competing concerns. It needs to be comprehensive enough to support all security features while remaining simple enough for typical use cases. It must provide sensible defaults that work out of the box while allowing advanced users to customize every aspect of the security policy. Most importantly, it must be validated early to prevent configuration errors from causing runtime failures after expensive setup work has already been performed.

Configuration Structure Design

The primary configuration structure, `sandbox_config_t`, encapsulates all parameters needed to create a sandbox instance. This structure follows the principle of **explicit over implicit** - every security-relevant setting is explicitly specified rather than inferred from context. This approach prevents subtle security bugs that can arise when default behaviors change or when configurations are applied in unexpected environments.

Field Name	Type	Purpose	Default Behavior
`program_path`	`char*`	Absolute path to executable that will run inside sandbox	Must be specified, no default
`program_args`	`char**`	Null-terminated array of command-line arguments	Empty array if not specified
`chroot_path`	`char*`	Path to directory that becomes the new root filesystem	Uses minimal built-in rootfs if NULL
`memory_limit`	`unsigned long`	Maximum memory in bytes that sandbox can allocate	64MB default limit
`cpu_percent`	`unsigned int`	Percentage of one CPU core (0-100) available to sandbox	10% default allocation
`enable_network`	`int`	Boolean flag controlling network namespace isolation	0 (disabled) for maximum isolation
`allowed_syscalls`	`char**`	Null-terminated array of permitted system call names	Minimal safe set if not specified

The configuration structure intentionally uses simple C types rather than complex nested structures. This design choice reflects the reality that sandbox configuration is often driven by external configuration files, command-line arguments, or

environment variables. Simple types make serialization, validation, and debugging significantly easier than deeply nested object hierarchies.

Resource Limit Configuration

Resource limits represent one of the most critical aspects of sandbox configuration because they directly impact system stability. Think of resource limits as the **power grid** for the sandbox prison - they determine how much electricity (CPU), water (memory), and other utilities each cell can consume. Without proper limits, a malicious or buggy program can consume all available resources and impact other processes on the system.

The memory limit specification uses bytes rather than more convenient units like megabytes to avoid ambiguity about binary versus decimal multipliers. A limit of `67108864` bytes is unambiguous, while "64MB" could mean either 64 * 1000 * 1000 or 64 * 1024 * 1024 bytes depending on interpretation. This precision matters because memory limits are enforced by the kernel with byte-level accuracy.

CPU limit configuration uses percentage of a single core rather than absolute CPU time to make the configuration portable across different hardware. A sandbox configured for "10% of one core" will behave consistently whether running on a dual-core laptop or a 32-core server. The cgroup implementation translates this percentage into appropriate period and quota values for kernel enforcement.

> **Key Design Insight**: Resource limits must be specified in kernel-native units to avoid translation errors and ensure consistent enforcement. The configuration layer provides the abstraction, but the underlying values must match exactly what the kernel expects.

System Call Allowlist Configuration

The system call allowlist represents the **security checkpoint** for all kernel interactions from the sandboxed process. Think of it as the **customs office** at a border crossing - every request to cross from userspace into kernel space must be explicitly approved based on a predetermined list of acceptable operations. The allowlist approach follows the security principle of **default deny** - anything not explicitly permitted is automatically forbidden.

System call names are used in the configuration rather than numeric values to improve readability and avoid architecture-specific differences. The same system call can have different numbers on x86_64 versus ARM64, but the name remains constant. The sandbox implementation translates names to numbers during seccomp filter compilation based on the runtime architecture.

The configuration supports both positive allowlists (specifying permitted calls) and negative blocklists (specifying forbidden calls), though the default behavior favors allowlists for security. A typical configuration might include essential calls like `read`, `write`, `exit`, and `exit_group` while excluding dangerous operations like `ptrace`, `mount`, or `kexec_load`.

> **Decision: System Call Configuration Format**
>
> - **Context**: Need to specify which system calls the sandboxed process can make while keeping configuration readable and architecture-independent
> - **Options Considered**:
>   1. Numeric system call IDs in configuration
>   2. String names translated to numbers at runtime
>   3. Predefined named policies (strict, moderate, permissive)
> - **Decision**: String names with runtime translation to numbers
> - **Rationale**: Names are architecture-independent, human-readable, and maintainable. Translation overhead is acceptable since it happens once during sandbox setup.
> - **Consequences**: Requires system call name-to-number translation logic and architecture detection, but improves configuration portability and debuggability.

Filesystem Configuration

Filesystem configuration determines what files and directories the sandboxed process can access. Think of this as designing the **floor plan** of the security prison - it specifies which rooms (directories) exist, what furniture (files) they contain, and whether the doors have locks (permissions). The filesystem configuration must balance security isolation with providing enough functionality for the sandboxed program to operate correctly.

The `chroot_path` configuration specifies the directory that becomes the new root filesystem for the sandboxed process. This directory must contain a complete minimal Linux filesystem with essential directories like `/bin`, `/lib`, `/proc`, `/dev`, and `/tmp`. The sandbox orchestrator validates this structure during configuration parsing to ensure the sandboxed process won't fail due to missing dependencies.

When `chroot_path` is NULL, the sandbox uses a built-in minimal root filesystem constructed in memory using tmpfs. This default behavior provides a secure baseline that works for simple programs without requiring users to construct their own root filesystem. The built-in filesystem includes only essential files and directories needed for basic program execution.

Runtime State

The **runtime state** captures the dynamic information about a running sandbox instance. Think of runtime state as the **security control room** for the prison - it contains all the monitors, switches, and communication channels needed to observe and control the sandbox throughout its lifecycle. While configuration represents the desired state, runtime state represents the actual current state of all security mechanisms.

Runtime state differs fundamentally from configuration in that it contains **handles and references to kernel objects** rather than parameters and settings. Namespace file descriptors, process IDs, and cgroup paths are all kernel-managed resources that must be properly tracked and cleaned up. The runtime state structure serves as the sandbox's **resource inventory** - it knows exactly what kernel resources are allocated and ensures they are properly released when the sandbox terminates.

The complexity of runtime state management stems from the **temporal dependencies** between different security layers. Namespaces must be created before processes can be moved into them. Cgroups must be configured before processes

are added to them. Capabilities must be dropped after all other setup is complete. The runtime state tracks not just what resources exist, but what order they were created in and what dependencies exist between them.

Process and Namespace Tracking

Process tracking represents the core of runtime state management because the sandboxed process is the central entity around which all other security mechanisms revolve. Think of the process as the **prisoner** in the security analogy - the runtime state must know where the prisoner is, what cell they're in, and how to communicate with or terminate them if necessary.

Field Name	Type	Purpose	Lifecycle
`child_pid`	`pid_t`	Process ID of the sandboxed process	Set after fork, cleared after wait
`namespace_fd`	`int[6]`	File descriptors for each namespace type	Created during namespace setup, closed during cleanup
`cgroup_path`	`char*`	Filesystem path to sandbox's cgroup directory	Allocated during cgroup creation, freed during cleanup
`cleanup_needed`	`int`	Boolean flag indicating whether cleanup is required	Set to 1 during setup, cleared after successful cleanup

The namespace file descriptor array stores handles to each namespace type in a fixed order: PID, mount, network, UTS, user, and IPC namespaces. These file descriptors allow the parent process to manipulate namespaces even after the sandboxed process has started executing. For example, the parent might need to configure network interfaces in the network namespace or add additional mount points to the mount namespace.

Namespace file descriptors are obtained by opening files in `/proc/[pid]/ns/` after creating the namespaces but before the sandboxed process calls exec. Once the exec call completes, the original process image is replaced and the namespace setup code is no longer running, making these file descriptors the only way for the parent to access the namespaces.

> **Critical Resource Management**: Namespace file descriptors must be closed by the parent process during cleanup to prevent file descriptor leaks. Each unclosed namespace file descriptor prevents the kernel from fully cleaning up the namespace even after all processes in it have terminated.

Namespace State Tracking

The namespace state structure provides detailed tracking of namespace creation and initialization status. Think of this as the **facility status board** in a control room - it shows exactly which isolation chambers are operational and which are still under construction.

Field Name	Type	Purpose	Valid Values
`pid_ns_fd`	`int`	File descriptor for PID namespace	-1 if not created, valid fd if active
`mount_ns_fd`	`int`	File descriptor for mount namespace	-1 if not created, valid fd if active
`net_ns_fd`	`int`	File descriptor for network namespace	-1 if not created, valid fd if active
`uts_ns_fd`	`int`	File descriptor for UTS namespace	-1 if not created, valid fd if active
`user_ns_fd`	`int`	File descriptor for user namespace	-1 if not created, valid fd if active
`initialized`	`int`	Boolean indicating all namespaces ready	0 during setup, 1 when complete

The namespace state structure enables partial cleanup in failure scenarios. If namespace creation fails partway through the process, the cleanup code can examine each file descriptor to determine which namespaces were successfully created and need to be cleaned up. This prevents resource leaks when errors occur during sandbox initialization.

The `initialized` flag serves as a **consistency marker** that indicates whether the namespace setup process completed successfully. The sandbox orchestrator sets this flag only after all namespaces have been created and their file descriptors have been stored. This flag prevents the sandboxed process from starting execution before the security isolation is fully established.

Cleanup Resource Tracking

Cleanup resource tracking ensures that all kernel resources allocated during sandbox creation are properly released when the sandbox terminates. Think of this as the **maintenance checklist** that facility staff use when closing down a section of the prison - every lock must be checked, every utility must be turned off, and every resource must be accounted for.

The `cleanup_needed` flag provides a **fail-safe mechanism** that ensures cleanup occurs even if the normal termination path is interrupted. This flag is set to 1 as soon as any kernel resources are allocated and cleared to 0 only after all resources have been successfully released. If the parent process receives a signal or encounters an unexpected error, it can check this flag to determine whether cleanup is necessary.

Cgroup path tracking presents unique challenges because cgroups are managed through the filesystem interface rather than file descriptors. The `cgroup_path` field stores the full filesystem path to the sandbox's cgroup directory, which must be removed during cleanup to prevent accumulation of empty cgroup directories over time. The cleanup process must also handle the case where processes are still running in the cgroup, which prevents directory removal until all processes have been terminated.

Decision: Cleanup Strategy

- **Context**: Need to ensure all kernel resources are properly released when sandbox terminates, even in failure scenarios
- **Options Considered**:
  1. Manual cleanup calls at each exit point
  2. Cleanup flag with centralized cleanup function
  3. Reference counting for each resource type
- **Decision**: Cleanup flag with centralized cleanup function
- **Rationale**: Centralized cleanup reduces code duplication and ensures consistent cleanup behavior. The flag approach is simpler than reference counting while still providing safety.
- **Consequences**: All exit paths must call the cleanup function, but resource management is centralized and easier to verify for correctness.

Configuration Validation and Defaults

Configuration validation ensures that sandbox parameters are valid and consistent before expensive setup operations begin. Think of validation as the **blueprints review process** before construction starts - it's much cheaper to catch design problems on paper than to discover them after pouring the foundation. The validation process checks individual parameter values, relationships between parameters, and system compatibility.

Individual parameter validation checks that each configuration value is within acceptable bounds and formats. Memory limits must be positive and within system limits. CPU percentages must be between 0 and 100. File paths must be absolute and accessible. System call names must be recognized on the current architecture. These checks prevent runtime failures that would waste resources and potentially leave the system in an inconsistent state.

Relationship validation ensures that configuration parameters are mutually compatible. For example, if network access is disabled, any configuration that depends on network connectivity should be flagged as invalid. If the chroot path is specified but doesn't contain required directories, the validation should detect this before attempting sandbox creation.

Validation Type	Check Performed	Error Condition	Recovery Action
Memory Limit	Value > 0 and < system memory	Zero or negative limit	Use default 64MB limit
CPU Percentage	Value between 0 and 100	Out of range value	Use default 10% limit
Program Path	File exists and is executable	Missing or non-executable	Return configuration error
Chroot Path	Directory exists with required subdirs	Missing directory structure	Use built-in minimal rootfs
System Calls	All names valid on current architecture	Unrecognized system call name	Return configuration error

Default value assignment provides a secure baseline for unconfigured parameters. The defaults are chosen to prioritize security over performance - it's better to have a sandbox that runs slowly but securely than one that runs quickly but allows privilege escalation. Users who need higher performance can explicitly configure more permissive settings after understanding the security implications.

The default system call allowlist includes only the most essential operations needed for basic program execution: `read`, `write`, `exit`, `exit_group`, `brk`, `mmap`, `munmap`, `mprotect`, and `rt_sigreturn`. This minimal set allows

simple programs to run while blocking dangerous operations like process creation, file system modification, or network access.

Data Flow and State Transitions

The sandbox data model supports a clear flow from initial configuration through runtime execution to final cleanup. Think of this as the **prisoner processing pipeline** - there are distinct stages with specific hand-off points and validation checkpoints between each stage. Understanding this flow is crucial for implementing robust error handling and resource management.

The configuration phase begins with parsing user input and ends with a validated `sandbox_config_t` structure. During this phase, all user-provided values are validated, defaults are applied for missing parameters, and system compatibility is verified. No kernel resources are allocated during configuration parsing, making this phase fast and reversible.

The initialization phase creates kernel resources and populates the `sandbox_state_t` structure. This phase includes namespace creation, filesystem setup, cgroup configuration, and other resource-intensive operations. Each step in initialization updates the runtime state structure to track what resources have been allocated, enabling partial cleanup if later steps fail.

Phase	Input	Output	Resources Allocated	Failure Recovery
Parse Config	User input	`sandbox_config_t`	None	Simple error return
Validate Config	`sandbox_config_t`	Validated config	None	Simple error return
Initialize	Config + empty state	Populated `sandbox_state_t`	Namespaces, cgroups, processes	Cleanup allocated resources
Execute	State + config	Running sandbox	None (resources already allocated)	Terminate and cleanup
Cleanup	`sandbox_state_t`	Clean system	None (resources released)	Force cleanup if needed

The execution phase monitors the running sandbox and handles communication between the parent supervisor and the sandboxed process. During this phase, the configuration is read-only and the runtime state changes only to reflect the current status of the sandboxed process (running, terminated, etc.).

The cleanup phase releases all kernel resources tracked in the runtime state structure. This phase must be robust enough to handle partial failures - if some resources cannot be released immediately (e.g., because processes are still running), the cleanup process must retry or escalate to more forceful termination methods.

Common Pitfalls

⚠️ **Pitfall: Configuration Pointer Ownership** A common mistake is unclear ownership of string pointers in the configuration structure. Fields like `program_path` and `chroot_path` point to memory that may be allocated by different parts of the system (command-line parsing, configuration file reading, etc.). If the configuration structure doesn't properly manage this memory, it can lead to double-free errors or memory leaks. The solution is to make the configuration structure own all its string data by copying values during initialization and freeing them during destruction.

⚠️ **Pitfall: Namespace File Descriptor Leaks** Namespace file descriptors are easy to leak because they remain valid even after all processes in the namespace have terminated. Unlike process IDs which become invalid when processes exit, namespace file descriptors keep the namespace alive indefinitely. Always close these file descriptors in the cleanup function and set them to -1 to prevent accidental reuse.

⚠️ **Pitfall: Partial Cleanup State** Runtime state structures can be left in inconsistent states if cleanup is interrupted or fails partway through. For example, if the parent process is killed while cleaning up cgroups, some resources may be released while others remain allocated. Design the cleanup function to be **idempotent** - calling it multiple times should be safe and should eventually clean up all resources.

⚠️ **Pitfall: Configuration Validation Race Conditions** Validating configuration at parse time doesn't guarantee that the same validation will hold at execution time. For example, checking that a chroot directory exists during configuration parsing doesn't prevent the directory from being deleted before sandbox creation. Critical validations must be repeated during the initialization phase when resources are actually being allocated.

⚠️ **Pitfall: Integer Overflow in Resource Limits** Memory and resource limits are often specified in human-readable units (MB, GB) but stored in bytes. Converting "1GB" to bytes requires multiplication by 1073741824, which can overflow 32-bit integers. Always use appropriate integer types (`unsigned long` or `uint64_t`) for byte-based limits and validate that conversions don't overflow.

Implementation Guidance

The data model implementation requires careful attention to memory management, error handling, and resource lifecycle management. The structures serve as the foundation for all sandbox operations, so bugs in the data model can cascade throughout the entire system.

Technology Recommendations

Component	Simple Option	Advanced Option
Configuration Parsing	Manual string parsing with `strtol` / `strtoul`	JSON parsing with `cjson` library
String Management	Fixed-size buffers with `strncpy`	Dynamic allocation with `strdup`
Error Handling	Integer return codes	Structured error objects
Memory Management	Manual `malloc` / `free` with cleanup functions	Memory pools or smart pointers

Recommended File Structure

```
sandbox/
  include/
    sandbox.h               ← public API and data structures
  src/
    config.c                ← configuration management
    state.c                 ← runtime state management
    cleanup.c               ← resource cleanup utilities
  tests/
    test_config.c           ← configuration validation tests
    test_state.c            ← state management tests
```

Configuration Management Infrastructure

The configuration management code handles parsing, validation, and memory management for sandbox configuration structures. This infrastructure provides a clean API for other components while encapsulating the complexity of parameter validation and default assignment.

```c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include "sandbox.h"

// Default resource limits for security and stability
#define DEFAULT_MEMORY_LIMIT (64UL * 1024 * 1024)  // 64MB
#define DEFAULT_CPU_PERCENT 10
#define MAX_MEMORY_LIMIT (1UL << 30)  // 1GB maximum

// Default minimal system call allowlist
static const char* default_syscalls[] = {
    "read", "write", "exit", "exit_group",
    "brk", "mmap", "munmap", "mprotect",
    "rt_sigreturn", NULL
};

// Initialize configuration structure with secure defaults
sandbox_config_t* sandbox_config_init(void) {
    // TODO: Allocate sandbox_config_t structure using malloc
    // TODO: Initialize all string pointers to NULL
    // TODO: Set memory_limit to DEFAULT_MEMORY_LIMIT
    // TODO: Set cpu_percent to DEFAULT_CPU_PERCENT
    // TODO: Set enable_network to 0 (disabled for security)
    // TODO: Copy default_syscalls array to allowed_syscalls field
    // TODO: Return pointer to initialized structure
    // Hint: Use strdup() to copy strings so they can be freed later
}

// Validate configuration parameters for security and correctness
int sandbox_config_validate(sandbox_config_t* config) {
```

```
    // TODO: Check that program_path is not NULL and file exists
    // TODO: Use stat() to verify program_path is executable
    // TODO: Validate memory_limit is > 0 and < MAX_MEMORY_LIMIT
    // TODO: Validate cpu_percent is between 0 and 100
    // TODO: If chroot_path provided, check directory exists
    // TODO: Validate each syscall name in allowed_syscalls array
    // TODO: Return SANDBOX_SUCCESS if valid, appropriate error code if not
    // Hint: Use access(path, X_OK) to check if file is executable
}

// Clean up all memory allocated for configuration
void sandbox_config_free(sandbox_config_t* config) {
    // TODO: Free program_path string if not NULL
    // TODO: Free each string in program_args array, then array itself
    // TODO: Free chroot_path string if not NULL
    // TODO: Free each string in allowed_syscalls array, then array itself
    // TODO: Free the config structure itself
    // TODO: Set config pointer to NULL to prevent reuse
    // Hint: Always check for NULL before calling free()
}
```

Runtime State Management Infrastructure

The runtime state management code tracks kernel resources and handles cleanup operations. This infrastructure ensures that all allocated resources are properly released even in failure scenarios.

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
#include "sandbox.h"

// Initialize runtime state structure
sandbox_state_t* sandbox_state_init(void) {
    // TODO: Allocate sandbox_state_t structure using malloc
    // TODO: Set child_pid to -1 (invalid PID)
    // TODO: Initialize all namespace_fd entries to -1 (invalid FDs)
    // TODO: Set cgroup_path to NULL
    // TODO: Set cleanup_needed to 0 (no cleanup required yet)
    // TODO: Return pointer to initialized structure
}

// Mark that cleanup will be needed for this sandbox
void sandbox_state_mark_cleanup_needed(sandbox_state_t* state) {
    // TODO: Set cleanup_needed flag to 1
    // TODO: This should be called as soon as any kernel resources are allocated
}

// Store namespace file descriptors for later cleanup
int sandbox_state_store_namespaces(sandbox_state_t* state, const namespace_state_t* ns_state) {
    // TODO: Copy all file descriptors from ns_state to state->namespace_fd array
    // TODO: Validate that all file descriptors are valid (>= 0)
    // TODO: Mark cleanup as needed since we now have resources to clean
    // TODO: Return SANDBOX_SUCCESS on success, error code on failure
}

// Clean up all resources tracked in state structure
void cleanup_sandbox(sandbox_state_t* state) {
```

```
    // TODO: Check cleanup_needed flag - return early if no cleanup needed

    // TODO: If child_pid is valid, send SIGTERM then wait for process

    // TODO: Close all valid namespace file descriptors in namespace_fd array

    // TODO: If cgroup_path is set, remove the cgroup directory

    // TODO: Free cgroup_path string if allocated

    // TODO: Set cleanup_needed to 0 and reset all fields to initial values

    // Hint: Use kill(child_pid, SIGTERM) followed by waitpid() to clean up process

    // Hint: Use rmdir() to remove empty cgroup directory

}
```

Error Handling and Validation Utilities

The error handling infrastructure provides consistent error reporting and recovery throughout the sandbox system. These utilities ensure that errors are detected early and reported with sufficient detail for debugging.

```c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include "sandbox.h"

// Report sandbox errors with context and errno information
void sandbox_error(const char* operation, int error_code) {
    // TODO: Print error message including operation that failed
    // TODO: Include error_code and its meaning (use strerror if errno-based)
    // TODO: Include current errno value if relevant
    // TODO: Format message consistently for easier debugging
    // Hint: Use fprintf(stderr, ...) for error output
}

// Check if current process has root privileges
int check_root_privileges(void) {
    // TODO: Check if effective UID is 0 (root)
    // TODO: Return 1 if running as root, 0 if not
    // TODO: Consider checking for specific capabilities if needed
    // Hint: Use geteuid() to get effective user ID
}

// Validate that a directory contains required subdirectories for chroot
int validate_chroot_structure(const char* chroot_path) {
    // TODO: Check that chroot_path directory exists and is accessible
    // TODO: Verify required subdirectories exist: /bin, /lib, /tmp, /proc, /dev
    // TODO: Check that essential files exist (may be empty but structure required)
    // TODO: Return SANDBOX_SUCCESS if valid, SANDBOX_ERROR_FILESYSTEM if not
    // Hint: Use opendir() and readdir() to examine directory contents
}
```

Milestone Checkpoints

After implementing the data model infrastructure, verify correct behavior with these checkpoints:

Configuration Management Test:

```bash
# Compile test program
gcc -o test_config src/config.c tests/test_config.c -I include

# Run configuration tests
./test_config
# Expected output:
# Configuration initialization: PASSED
# Default value assignment: PASSED
# Parameter validation: PASSED
# Memory cleanup: PASSED
```

Runtime State Management Test:

```bash
# Compile state management test
gcc -o test_state src/state.c tests/test_state.c -I include

# Run state management tests
./test_state
# Expected output:
# State initialization: PASSED
# Resource tracking: PASSED
# Cleanup verification: PASSED
# Error handling: PASSED
```

Integration Test: Create a simple test program that initializes configuration, sets up runtime state, and cleans up resources. Verify that no memory leaks occur and all resources are properly released.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Segmentation fault on config access	Null pointer or uninitialized structure	Use gdb to check structure contents	Initialize all pointers to NULL, validate before use
Memory leak in configuration	String pointers not freed properly	Run with valgrind to find leaks	Ensure all malloc'd strings are freed in cleanup
File descriptor leak warnings	Namespace FDs not closed	Check `/proc/[pid]/fd/` for open FDs	Close all FDs in cleanup function
Cleanup function hangs	Process won't terminate	Use `ps aux` to check process state	Send SIGKILL after SIGTERM timeout
Cgroup removal fails	Processes still in cgroup	Check cgroup.procs file contents	Ensure all processes terminated before rmdir

Namespace Isolation Component

> **Milestone(s):** This section corresponds to Milestone 1 (Process Namespaces) and provides foundational isolation for all subsequent milestones

The namespace isolation component forms the first and most fundamental layer of the process sandbox's defense-in-depth strategy. By leveraging Linux namespaces, this component creates isolated views of system resources that make the sandboxed process believe it exists in its own private universe, completely separate from the host system. This isolation is so convincing that a process running inside the sandbox sees itself as process ID 1 in its own system, with its own network stack, filesystem mounts, and even hostname—all while being just another process on the host system.

Mental Model: Separate Universes

> Think of Linux namespaces as creating parallel universes for processes. Imagine you have a magic mirror that shows a completely different reality to anyone looking into it. From the host system's perspective, the sandboxed process is just another process with a regular PID, using the normal filesystem, and sharing the same network interfaces. But from inside the sandbox, the process sees itself as the only important process (PID 1), living on a system with its own private filesystem and isolated network stack.

This "separate universe" effect is achieved through the kernel's namespace mechanism, which maintains multiple independent views of the same underlying system resources. When a process is created in a new namespace, the kernel essentially creates a new mapping table for that category of resource. For example, in a new PID namespace, the kernel maintains a separate process ID mapping where the first process gets PID 1, even though it might be PID 23847 from the host's perspective.

The power of this approach lies in its completeness—the sandboxed process cannot even discover that other realities exist. It cannot see host processes, cannot access host mount points, and cannot communicate over the host's network

interfaces. This isolation occurs at the kernel level, making it extremely difficult for malicious code to break out of the sandbox through normal system call interfaces.

Consider a concrete scenario: when a sandboxed process calls `getpid()`, the kernel consults the PID namespace mapping and returns 1. When it calls `ps aux` to list processes, the kernel only shows processes within the same PID namespace. When it attempts to connect to network addresses, the kernel routes those requests through the network namespace's isolated network stack, which by default has no external connectivity.

Namespace Types and Usage

Linux provides several types of namespaces, each isolating a different category of system resource. The process sandbox leverages five key namespace types to create comprehensive isolation. Understanding each namespace type and its security benefits is crucial for implementing effective sandbox isolation.

Namespace Type	System Resource	Security Benefit	Implementation Challenge
PID	Process IDs and process tree	Hides host processes, prevents process manipulation attacks	Requires proper /proc mounting, zombie reaping
Mount	Filesystem mount points	Isolates filesystem view, prevents unauthorized file access	Complex mount propagation, dependency resolution
Network	Network interfaces, routing tables, firewalls	Blocks network access, prevents network-based attacks	Interface creation, connectivity management
UTS	Hostname and domain name	Isolates system identity, prevents information leakage	Minimal complexity, mainly cosmetic isolation
User	User and group IDs	Maps root inside container to unprivileged user outside	Most complex, requires careful UID/GID mapping

PID Namespace Isolation provides process-level security by creating an isolated process tree where the sandboxed process becomes PID 1. This isolation prevents the sandboxed process from discovering, signaling, or manipulating host processes. The sandboxed process cannot use `kill`, `ptrace`, or other process control system calls to affect processes outside its namespace. Additionally, the isolated process tree means that when the sandbox terminates, all child processes are automatically cleaned up since they lose their parent process.

The PID namespace also affects the `/proc` filesystem, which must be mounted separately within the namespace to reflect the isolated process view. Without proper `/proc` mounting, tools like `ps`, `top`, and `pidof` will either fail or show incorrect information from the host namespace.

Mount Namespace Isolation creates an independent filesystem view by providing a separate mount table. This isolation prevents the sandboxed process from accessing host mount points, even if it knows their paths. The sandbox can have its own `/tmp`, `/dev`, and other critical directories without affecting or being affected by the host system.

Mount namespaces support different propagation modes that control how mount and unmount events propagate between namespaces. The sandbox uses private propagation to ensure that mount changes within the sandbox do not affect the host system. This isolation extends to preventing access to sensitive host directories like `/root`, `/home`, and system configuration directories.

Network Namespace Isolation provides complete network isolation by creating a separate network stack with its own interfaces, routing tables, and firewall rules. By default, a new network namespace contains only a loopback interface, effectively cutting off all external network access. This isolation prevents network-based attacks, data exfiltration, and unauthorized communication.

The network namespace can be selectively configured to allow specific types of network access if required. This might involve creating virtual network interfaces, setting up network address translation, or configuring specific firewall rules. However, the default configuration provides maximum security by denying all network access.

UTS Namespace Isolation isolates the system hostname and domain name, preventing the sandboxed process from discovering or modifying the host system's identity. While this provides minimal security benefit compared to other namespaces, it contributes to the overall isolation illusion and prevents information leakage that might aid in fingerprinting or reconnaissance attacks.

User Namespace Isolation provides the most complex but potentially most valuable isolation by mapping user and group IDs between the host and sandbox. This mapping allows a process to have root privileges (UID 0) inside the sandbox while running as an unprivileged user outside the sandbox. This isolation dramatically reduces the impact of privilege escalation vulnerabilities within the sandbox.

User namespace mapping requires careful configuration of UID and GID mappings in `/proc/PID/uid_map` and `/proc/PID/gid_map` files. These mappings define how user and group IDs translate between the namespace and the host system. Incorrect mapping can either break functionality or compromise security.

Namespace Creation and Ordering

Creating namespaces in the correct sequence is critical for both functionality and security. The kernel imposes certain ordering constraints, and the sandbox's security model depends on establishing isolation layers in a specific order. The namespace creation algorithm must handle these dependencies while providing comprehensive error handling and cleanup.

The namespace creation process follows this carefully orchestrated sequence:

1. **Verify Root Privileges**: Check that the calling process has sufficient privileges to create namespaces. Most namespace creation requires `CAP_SYS_ADMIN` capability or root privileges. The process should verify these privileges early to provide clear error messages rather than cryptic system call failures.
2. **Create User Namespace First**: If using user namespaces, create them before any other namespaces. User namespaces are special because they can grant capabilities needed for creating other namespaces, even to non-root processes. Creating the user namespace first establishes the privilege context for subsequent namespace creation.
3. **Set Up UID/GID Mappings**: Immediately after creating the user namespace, configure the UID and GID mappings by writing to `/proc/self/uid_map` and `/proc/self/gid_map`. These mappings must be established before creating other namespaces that depend on user identity checks.
4. **Create PID Namespace**: Create the PID namespace to establish process isolation. The PID namespace affects process creation and signaling, so it should be established before creating the actual sandboxed process. Note that the creating process does not enter the PID namespace—only its children will be in the new namespace.
5. **Create Mount Namespace**: Create the mount namespace to establish filesystem isolation. This namespace affects all subsequent filesystem operations, including the mounting of `/proc`, `/dev`, and other special filesystems that other namespaces might require.

6. **Create Network Namespace**: Create the network namespace to establish network isolation. Network namespaces are relatively independent but should be created before UTS namespaces for consistency and to ensure network-related hostname resolution works correctly within the isolated environment.
7. **Create UTS Namespace**: Create the UTS namespace last among the isolation namespaces. UTS namespaces have few dependencies and primarily provide cosmetic isolation, making them safe to create at any point in the sequence.
8. **Validate Namespace Creation**: After creating all namespaces, validate that they were created successfully by checking namespace file descriptors and reading namespace identifiers from `/proc/self/ns/` entries. This validation catches subtle failures that might not be immediately apparent.

The algorithm must handle partial failures gracefully. If namespace creation fails at any step, the process should clean up any successfully created namespaces and return to the original namespace context. This cleanup prevents resource leaks and ensures the system remains in a consistent state.

```
Algorithm: Namespace Creation Sequence

Input: namespace_state_t* ns_state (structure to track namespace file descriptors)
Output: SANDBOX_SUCCESS or appropriate error code

1. Initialize namespace_state_t structure with invalid file descriptors (-1)
2. Call check_root_privileges() to verify required privileges
3. If using user namespaces:
   a. Call unshare(CLONE_NEWUSER) to create user namespace
   b. Store namespace fd in ns_state->user_ns_fd
   c. Write UID mapping to /proc/self/uid_map
   d. Write GID mapping to /proc/self/gid_map
4. Call unshare(CLONE_NEWPID) to create PID namespace
5. Store namespace fd in ns_state->pid_ns_fd
6. Call unshare(CLONE_NEWNS) to create mount namespace
7. Store namespace fd in ns_state->mount_ns_fd
8. Call unshare(CLONE_NEWNET) to create network namespace
9. Store namespace fd in ns_state->net_ns_fd
10. Call unshare(CLONE_NEWUTS) to create UTS namespace
11. Store namespace fd in ns_state->uts_ns_fd
12. Validate all namespace creation by reading /proc/self/ns/ entries
13. Set ns_state->initialized = 1
14. Return SANDBOX_SUCCESS

Error Handling:
- If any step fails, call cleanup_namespaces(ns_state) before returning error code
- Log specific error messages indicating which namespace creation failed
- Ensure all file descriptors are properly closed during cleanup
```

Architecture Decisions

The namespace isolation component requires several critical architecture decisions that significantly impact both security and functionality. Each decision involves trade-offs between security, complexity, performance, and compatibility that must be carefully evaluated.

Decision: Use unshare() vs clone() for Namespace Creation

- **Context**: Linux provides two primary mechanisms for creating namespaces: the `unshare()` system call that moves the calling process into new namespaces, and the `clone()` system call that creates a new process in new namespaces. The choice affects process creation flow and error handling complexity.
- **Options Considered**:
  - Use `unshare()` to create namespaces then `fork()` to create sandboxed process
  - Use `clone()` with namespace flags to create process and namespaces atomically
  - Hybrid approach using `unshare()` for some namespaces and `clone()` for others
- **Decision**: Use `unshare()` followed by `fork()` for namespace creation
- **Rationale**: The `unshare()` approach provides better error handling and cleanup capabilities. If namespace creation fails, the parent process remains in the original namespaces and can clean up resources. The `clone()` approach creates the process and namespaces atomically, making error recovery more complex since the child process might be created even if some namespaces fail. Additionally, `unshare()` allows for step-by-step namespace creation with validation at each step.
- **Consequences**: This approach requires two separate system calls (`unshare()` then `fork()`) instead of one atomic `clone()` call. However, it provides cleaner error handling, better logging of namespace creation failures, and allows for partial namespace creation if needed for testing or debugging.

Approach	Error Handling	Complexity	Atomic Creation	Resource Cleanup
unshare() + fork()	Excellent - parent can recover	Medium	No	Simple
clone() with flags	Limited - child handles errors	Low	Yes	Complex
Hybrid approach	Complex - depends on namespace	High	Partial	Variable

Decision: User Namespace Mapping Strategy

- **Context**: User namespaces require explicit UID and GID mappings between the host and container. These mappings determine what user and group identities the sandboxed process sees versus what identities it has on the host system. Incorrect mappings can compromise security or break functionality.
- **Options Considered**:
  - Map container root (UID 0) to host process UID with single mapping
  - Create comprehensive mapping covering multiple UIDs and GIDs
  - Skip user namespaces entirely and rely on other isolation mechanisms
- **Decision**: Map container root (UID 0) to host process UID with single identity mapping
- **Rationale**: Single identity mapping provides maximum security by ensuring the sandboxed process has no privileges outside the namespace, regardless of what UID it believes it has inside. Comprehensive mappings increase complexity and potential attack surface. Skipping user namespaces entirely misses a valuable security layer that allows other namespace creation without root privileges.
- **Consequences**: Sandboxed processes can run as root inside the container while being unprivileged outside. This approach limits functionality for applications that need multiple user identities but provides strong security guarantees. File ownership and permissions must be carefully managed since files created inside the sandbox will appear owned by the mapped host UID.

Mapping Strategy	Security	Complexity	Functionality	Root Required
Single identity	Excellent	Low	Limited	No
Multiple mappings	Good	High	Full	Sometimes
No user namespace	Poor	Low	Full	Yes

Decision: Namespace File Descriptor Management

- **Context**: Linux namespaces can be referenced through file descriptors obtained from `/proc/PID/ns/` entries. These file descriptors can be used to enter namespaces later or to ensure namespaces persist even after all processes exit. The sandbox must decide whether to keep these file descriptors open and how to manage their lifecycle.
- **Options Considered**:
  - Keep namespace file descriptors open for entire sandbox lifetime
  - Close file descriptors immediately after namespace creation
  - Selectively keep only critical namespace file descriptors
- **Decision**: Keep all namespace file descriptors open and store them in `namespace_state_t`
- **Rationale**: Keeping namespace file descriptors open allows the parent supervisor to monitor namespace health, perform cleanup operations, and potentially enter namespaces for debugging or management. File descriptors also prevent namespace destruction if all processes exit unexpectedly. The resource cost of keeping a few file descriptors open is minimal compared to the operational benefits.
- **Consequences**: The sandbox must carefully manage file descriptor cleanup to prevent resource leaks. File descriptors must be closed when the sandbox terminates. This approach increases memory usage slightly but provides better observability and control over namespace lifecycle.

FD Management	Resource Usage	Observability	Cleanup Complexity	Namespace Persistence
Keep open	Low increase	Excellent	Medium	Guaranteed
Close immediately	Minimal	Limited	Simple	Process-dependent
Selective keeping	Variable	Partial	High	Partial

Decision: Privilege Requirements and Capability Handling

- **Context**: Different namespaces have different privilege requirements. Some require root privileges, others require specific capabilities like `CAP_SYS_ADMIN`. The sandbox must handle privilege requirements consistently while minimizing the attack surface of running with elevated privileges.
- **Options Considered**:
  - Require root privileges for all namespace operations
  - Use capabilities to minimize required privileges
  - Create user namespace first to gain privileges for other namespace creation
- **Decision**: Use user namespace creation to gain privileges for other namespaces when possible, fall back to checking root privileges
- **Rationale**: User namespaces provide a mechanism to gain the capabilities needed for creating other namespaces without requiring the parent process to run as root. This approach follows the principle of least privilege while maintaining functionality. Falling back to root privilege checking ensures compatibility with systems where user namespaces might be disabled.
- **Consequences**: The sandbox can be used by non-root users on systems with user namespace support, improving security. However, the code must handle both privilege models and provide clear error messages when neither approach works. Some systems disable user namespaces for security reasons, requiring root privileges.

Privilege Strategy	Security	Compatibility	Complexity	User Access
User namespace first	Excellent	Good	Medium	Yes
Root required	Good	Excellent	Low	No
Capabilities only	Good	Limited	High	Depends

Common Pitfalls

Understanding and avoiding common pitfalls in namespace creation is crucial for building a reliable process sandbox. These pitfalls often manifest as subtle bugs that only appear under specific conditions or with certain workloads, making them particularly dangerous for security-critical systems.

⚠️ Pitfall: Forgetting SIGCHLD Handling in PID Namespaces

When creating PID namespaces, the first process in the namespace (PID 1) takes on special responsibilities similar to the init process on a normal system. One critical responsibility is reaping zombie child processes. If the sandbox creates child processes that terminate, they become zombies unless properly reaped by their parent.

In a PID namespace, the PID 1 process must handle `SIGCHLD` signals and call `wait()` or `waitpid()` to clean up terminated children. Failure to do this results in zombie processes accumulating within the namespace, potentially exhausting process limits and causing resource leaks.

Why it's wrong: Zombie processes consume kernel resources (process table entries) and can eventually exhaust the system's process limit. In a sandbox environment, this can be exploited as a denial-of-service attack where malicious code creates many short-lived processes to exhaust resources.

How to fix: Install a `SIGCHLD` signal handler that calls `waitpid(-1, NULL, WNOHANG)` in a loop to reap all available zombie children. Alternatively, use `signalfd()` or `sigwaitinfo()` for synchronous signal handling that integrates better with event loops.

⚠️ Pitfall: Not Mounting /proc in New PID Namespace

After creating a PID namespace, the `/proc` filesystem still reflects the host system's process view unless explicitly remounted. Many programs and system tools rely on `/proc` to discover running processes, system information, and resource usage. Without proper `/proc` mounting, these tools show incorrect information or fail entirely.

Why it's wrong: Programs inside the sandbox see host system processes through `/proc`, breaking the isolation illusion and potentially revealing sensitive information about the host system. Additionally, process management tools like `ps` and `top` show incorrect information, making debugging and monitoring difficult.

How to fix: After entering the new mount and PID namespaces but before executing the sandboxed program, mount a new `proc` filesystem: `mount("proc", "/proc", "proc", 0, NULL)`. Ensure the mount point exists and is properly cleaned up when the sandbox terminates.

⚠️ Pitfall: Incorrect User Namespace UID/GID Mapping

User namespace mapping requires writing specific formats to `/proc/self/uid_map` and `/proc/self/gid_map` files. The format is strict: "inside-id outside-id length" with specific whitespace requirements. Incorrect format, invalid ranges, or permission issues cause mapping failures that can compromise security or functionality.

Common mapping errors include: writing mappings after creating other namespaces (mappings must be written immediately), using incorrect UID ranges, forgetting to map GID in addition to UID, and not handling the single-write requirement (mappings can only be written once per namespace).

Why it's wrong: Failed UID/GID mapping can leave the process without proper user identity, cause file permission issues, or in worst cases, grant unintended privileges. The process might not be able to access files it should own or might retain privileges it should have dropped.

How to fix: Write UID and GID mappings immediately after creating the user namespace. Use the exact format: `printf("0 %d 1", getuid())` for single-identity mapping. Check write return values and handle errors explicitly. Test mappings by reading back the files and verifying the process's effective UID/GID.

⚠️ Pitfall: Namespace Creation Without Proper Error Handling

Namespace creation can fail for various reasons: insufficient privileges, kernel features disabled, resource exhaustion, or policy restrictions. Failing to handle these errors properly can leave the process in an inconsistent state with partial namespace isolation, compromising security.

Why it's wrong: Partial namespace creation might give a false sense of security while leaving attack vectors open. For example, if PID namespace creation succeeds but network namespace creation fails, the sandboxed process has process isolation but retains network access, potentially allowing data exfiltration.

How to fix: Check the return value of every `unshare()` call and handle failures appropriately. If any namespace creation fails, clean up successfully created namespaces and return to the original state. Provide clear error messages indicating which specific namespace creation failed and why. Test namespace creation under various failure conditions to ensure robust error handling.

⚠️ Pitfall: Race Conditions in Namespace Setup

Namespace creation involves multiple system calls that must be coordinated carefully. Race conditions can occur when multiple threads attempt namespace operations simultaneously, when the parent and child processes interact during setup, or when namespace file descriptors are accessed concurrently.

Why it's wrong: Race conditions can lead to processes ending up in wrong namespaces, namespace file descriptors becoming invalid, or cleanup operations failing. These issues are particularly dangerous because they might only manifest under specific timing conditions, making them difficult to detect during testing.

How to fix: Use proper synchronization mechanisms like mutexes or semaphores when namespace operations might be accessed concurrently. Ensure parent-child process synchronization during namespace setup using mechanisms like pipes or signals. Avoid sharing namespace file descriptors between threads without proper synchronization.

⚠️ Pitfall: Not Handling Nested Namespace Restrictions

Some systems or container environments already run inside namespaces, creating nested namespace scenarios. Creating namespaces within existing namespaces can have unexpected restrictions, capability requirements, or behavior differences. Additionally, some security policies explicitly prevent nested namespace creation.

Why it's wrong: Nested namespace creation might fail silently or create namespaces with different isolation properties than expected. Security policies might prevent namespace creation entirely, causing the sandbox to fail in production environments where it worked during development.

How to fix: Detect existing namespace contexts by comparing namespace identifiers in `/proc/self/ns/` with `/proc/1/ns/`. Provide clear error messages when nested namespace creation is attempted but not supported. Test the sandbox in various deployment environments, including containers and systems with restrictive security policies.

Implementation Guidance

The namespace isolation component requires careful orchestration of Linux system calls and proper resource management. This guidance provides complete infrastructure code and detailed skeletons for the core namespace creation logic.

Technology Recommendations:

Component	Simple Option	Advanced Option
Namespace Creation	Direct `unshare()` system calls	Custom namespace library with error recovery
File Descriptor Management	Simple array storage	Hash table with namespace type indexing
Process Synchronization	Pipe-based parent-child sync	Event file descriptors (eventfd)
Error Handling	Return codes with global errno	Structured error types with context
Privilege Detection	`getuid() == 0` check	Capability checking with `libcap`
Configuration	Static compile-time settings	Runtime configuration files

Recommended File Structure:

```
sandbox/
  src/
    namespace.h           ← namespace types and function declarations
    namespace.c           ← core namespace creation logic
    namespace_util.c      ← helper functions for namespace management
    config.h              ← configuration structures and defaults
    error.h               ← error handling utilities
    main.c                ← sandbox orchestrator using namespace component
  tests/
    test_namespace.c      ← unit tests for namespace creation
    test_integration.c    ← integration tests with real processes
  scripts/
    setup_test_env.sh     ← script to set up testing environment
```

Infrastructure Starter Code (Complete):

```c
// error.h - Error handling utilities

#ifndef SANDBOX_ERROR_H

#define SANDBOX_ERROR_H


#include <stdio.h>

#include <stdlib.h>

#include <errno.h>

#include <string.h>


// Error codes - use exactly these values

#define SANDBOX_SUCCESS 0

#define SANDBOX_ERROR_CONFIG 1

#define SANDBOX_ERROR_PRIVILEGE 2

#define SANDBOX_ERROR_NAMESPACE 3

#define SANDBOX_ERROR_FILESYSTEM 4

#define SANDBOX_ERROR_SECCOMP 5

#define SANDBOX_ERROR_CGROUP 6

#define SANDBOX_ERROR_CAPABILITY 7

#define SANDBOX_ERROR_EXEC 8


// Error reporting utility - logs error and optionally exits

void sandbox_error(const char* message, int error_code) {

    fprintf(stderr, "[SANDBOX ERROR %d] %s", error_code, message);

    if (errno != 0) {

        fprintf(stderr, ": %s", strerror(errno));

    }

    fprintf(stderr, "\n");

    if (error_code != SANDBOX_SUCCESS) {

        exit(error_code);

    }

}
```

```c
// Privilege checking utility
int check_root_privileges() {
    if (getuid() != 0 && geteuid() != 0) {
        // Check if user namespaces are available as alternative
        if (access("/proc/self/ns/user", F_OK) == 0) {
            return SANDBOX_SUCCESS; // Can use user namespaces
        }
        sandbox_error("Root privileges or user namespace support required",
                      SANDBOX_ERROR_PRIVILEGE);
        return SANDBOX_ERROR_PRIVILEGE;
    }
    return SANDBOX_SUCCESS;
}

#endif // SANDBOX_ERROR_H
```

```c
// config.h - Configuration structures
#ifndef SANDBOX_CONFIG_H
#define SANDBOX_CONFIG_H

#include <sys/types.h>

// Main sandbox configuration structure
typedef struct {
    char* program_path;           // Path to program to execute
    char** program_args;          // Arguments for program
    char* chroot_path;           // Path for chroot directory
    unsigned long memory_limit;   // Memory limit in bytes
    unsigned int cpu_percent;     // CPU percentage limit
    int enable_network;          // 1 to allow network, 0 to block
    char** allowed_syscalls;     // NULL-terminated list of allowed syscalls
} sandbox_config_t;

// Namespace state tracking structure
typedef struct {
    int pid_ns_fd;      // PID namespace file descriptor
    int mount_ns_fd;    // Mount namespace file descriptor
    int net_ns_fd;      // Network namespace file descriptor
    int uts_ns_fd;      // UTS namespace file descriptor
    int user_ns_fd;     // User namespace file descriptor
    int initialized;    // 1 if namespace creation completed successfully
} namespace_state_t;

// Runtime sandbox state
typedef struct {
    pid_t child_pid;           // PID of sandboxed process
    int namespace_fd[6];       // Array of namespace file descriptors
    char* cgroup_path;         // Path to cgroup directory
```

```c
    int cleanup_needed;        // 1 if cleanup is required
} sandbox_state_t;

// Configuration initialization with defaults
sandbox_config_t* sandbox_config_init() {
    sandbox_config_t* config = malloc(sizeof(sandbox_config_t));
    if (!config) return NULL;

    config->program_path = NULL;
    config->program_args = NULL;
    config->chroot_path = "/tmp/sandbox_root";
    config->memory_limit = 64 * 1024 * 1024; // 64MB default
    config->cpu_percent = 10; // 10% CPU default
    config->enable_network = 0; // Network disabled by default
    config->allowed_syscalls = NULL;

    return config;
}

// Configuration cleanup
void sandbox_config_free(sandbox_config_t* config) {
    if (!config) return;
    free(config->program_path);
    // Free string arrays if allocated
    if (config->program_args) {
        for (int i = 0; config->program_args[i]; i++) {
            free(config->program_args[i]);
        }
        free(config->program_args);
    }
    free(config->chroot_path);
```

```
    if (config->allowed_syscalls) {
        for (int i = 0; config->allowed_syscalls[i]; i++) {
            free(config->allowed_syscalls[i]);
        }
        free(config->allowed_syscalls);
    }
    free(config);
}

// Runtime state initialization
sandbox_state_t* sandbox_state_init() {
    sandbox_state_t* state = malloc(sizeof(sandbox_state_t));
    if (!state) return NULL;

    state->child_pid = -1;
    for (int i = 0; i < 6; i++) {
        state->namespace_fd[i] = -1;
    }
    state->cgroup_path = NULL;
    state->cleanup_needed = 0;

    return state;
}

#endif // SANDBOX_CONFIG_H
```

Core Logic Skeleton Code:

```c
// namespace.c - Core namespace creation implementation

#define _GNU_SOURCE

#include <sched.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/mount.h>

#include <fcntl.h>

#include "config.h"

#include "error.h"


// Create all required namespaces for sandbox isolation

int create_namespaces(namespace_state_t* ns_state) {

    /* TODO 1: Initialize namespace_state_t structure

     * - Set all file descriptors to -1

     * - Set initialized flag to 0

     * - Return SANDBOX_ERROR_CONFIG if ns_state is NULL

     */



    /* TODO 2: Check root privileges or user namespace availability

     * - Call check_root_privileges()

     * - Return SANDBOX_ERROR_PRIVILEGE if privileges insufficient

     * - Log clear error message about privilege requirements

     */



    /* TODO 3: Create user namespace first (if supported)

     * - Call unshare(CLONE_NEWUSER)

     * - Store namespace fd by opening /proc/self/ns/user

     * - Handle EPERM or ENOSYS errors gracefully

     * - Write UID/GID mappings immediately after creation

     * Hint: UID mapping format is "0 %d 1\n" where %d is getuid()
```

```
     */

    /* TODO 4: Create PID namespace
     * - Call unshare(CLONE_NEWPID)
     * - Store namespace fd in ns_state->pid_ns_fd
     * - Remember: calling process stays in old PID namespace
     * - Only children will be in new PID namespace
     * - Check for EPERM error and provide helpful message
     */

    /* TODO 5: Create mount namespace for filesystem isolation
     * - Call unshare(CLONE_NEWNS)
     * - Store namespace fd in ns_state->mount_ns_fd
     * - Set mount propagation to private: mount(NULL, "/", NULL, MS_REC|MS_PRIVATE, NULL)
     * - This prevents mount changes from propagating to host
     */

    /* TODO 6: Create network namespace for network isolation
     * - Call unshare(CLONE_NEWNET)
     * - Store namespace fd in ns_state->net_ns_fd
     * - New network namespace has only loopback interface by default
     * - No external network connectivity unless explicitly configured
     */

    /* TODO 7: Create UTS namespace for hostname isolation
     * - Call unshare(CLONE_NEWUTS)
     * - Store namespace fd in ns_state->uts_ns_fd
     * - Optionally set custom hostname with sethostname()
     * - This is mainly for information hiding, not security
     */
```

```
    /* TODO 8: Validate namespace creation success
     * - Read /proc/self/ns/ entries to verify namespace IDs changed
     * - Compare with initial namespace IDs to confirm isolation
     * - Log namespace IDs for debugging purposes
     * - Set ns_state->initialized = 1 if all validations pass
     */

    /* TODO 9: Error handling and cleanup
     * - If any step fails, clean up successfully created namespaces
     * - Close all opened file descriptors
     * - Use setns() to return to original namespaces if needed
     * - Return appropriate SANDBOX_ERROR_* code with descriptive message
     */

    return SANDBOX_SUCCESS;
}

// Helper function to write user namespace UID/GID mappings
static int setup_user_mapping() {
    /* TODO 1: Write UID mapping to /proc/self/uid_map
     * - Open /proc/self/uid_map for writing
     * - Write mapping: "0 %d 1\n" where %d is getuid()
     * - Handle EPERM errors (already mapped, or no permission)
     * - Close file descriptor
     */

    /* TODO 2: Write GID mapping to /proc/self/gid_map
     * - Open /proc/self/gid_map for writing
     * - Write mapping: "0 %d 1\n" where %d is getgid()
```

```
     * - Handle same error conditions as UID mapping
     * - Both mappings required for proper user namespace function
     */

    /* TODO 3: Verify mappings took effect
     * - Read back the mapping files to confirm format
     * - Check that getuid() and geteuid() return expected values
     * - Return SANDBOX_ERROR_NAMESPACE if mappings failed
     */

    return SANDBOX_SUCCESS;
}

// Clean up namespace resources
void cleanup_namespaces(namespace_state_t* ns_state) {
    /* TODO 1: Close all namespace file descriptors
     * - Check each fd in ns_state for validity (>= 0)
     * - Call close() on valid file descriptors
     * - Set file descriptors back to -1 after closing
     * - Handle EBADF errors gracefully (already closed)
     */

    /* TODO 2: Reset namespace state
     * - Set initialized flag to 0
     * - Clear any cached namespace information
     * - Log cleanup completion for debugging
     */
}

// Main sandbox creation function that orchestrates all components
int create_sandbox(sandbox_config_t* config, sandbox_state_t* state) {
```

```
    /* TODO 1: Validate input parameters
     * - Check that config and state are not NULL
     * - Verify config->program_path is set and executable
     * - Return SANDBOX_ERROR_CONFIG for invalid parameters
     */

    /* TODO 2: Create namespace isolation
     * - Allocate namespace_state_t structure
     * - Call create_namespaces() to set up isolation
     * - Store namespace file descriptors in state->namespace_fd array
     * - Handle namespace creation failures appropriately
     */

    /* TODO 3: Fork child process for sandbox execution
     * - Call fork() to create child process
     * - Child will inherit namespace isolation from parent
     * - Parent stores child PID in state->child_pid
     * - Handle fork() failure (resource exhaustion, limits)
     */

    /* TODO 4: Child process setup (in child branch)
     * - Mount new /proc filesystem for PID namespace
     * - Set up SIGCHLD handler for zombie reaping (child becomes PID 1)
     * - Set custom hostname if UTS namespace was created
     * - Prepare for filesystem isolation (next milestone)
     */

    /* TODO 5: Parent process monitoring (in parent branch)
     * - Set state->cleanup_needed = 1
     * - Optionally wait for child setup completion
```

```
     * - Return control to caller for additional configuration
     * - Child will continue with exec in later milestones
     */

    return SANDBOX_SUCCESS;
}

// Cleanup all sandbox resources
void cleanup_sandbox(sandbox_state_t* state) {
    /* TODO 1: Terminate child process if still running
     * - Send SIGTERM to state->child_pid
     * - Wait for graceful termination with timeout
     * - Send SIGKILL if process doesn't terminate
     * - Use waitpid() to reap child process
     */

    /* TODO 2: Clean up namespace file descriptors
     * - Close all file descriptors in state->namespace_fd array
     * - Namespace cleanup happens automatically when last process exits
     * - Set file descriptors to -1 after closing
     */

    /* TODO 3: Clean up other resources
     * - Free state->cgroup_path if allocated
     * - Reset cleanup_needed flag
     * - Log cleanup completion
     */
}
```

Language-Specific Implementation Hints:

- **System Call Error Handling**: Always check return values from `unshare()`, `fork()`, and file operations. Use `errno` and `strerror()` to provide meaningful error messages.
- **File Descriptor Management**: Use `open()` with `/proc/self/ns/*` entries to obtain namespace file descriptors. Remember to close them during cleanup.
- **Process Synchronization**: Use `pipe()` or `socketpair()` for parent-child synchronization during namespace setup if needed.
- **Signal Handling**: Install `SIGCHLD` handler in PID 1 process using `sigaction()` for robust zombie reaping.
- **Memory Management**: Always `free()` allocated configuration structures and handle allocation failures gracefully.

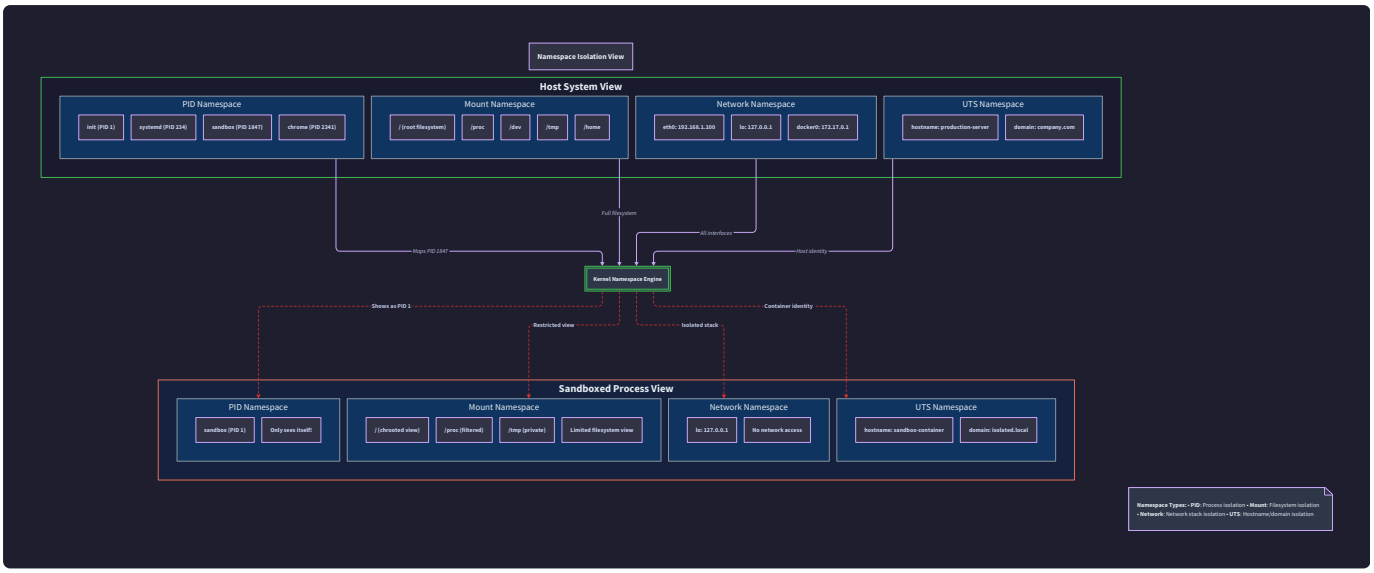
Milestone Checkpoint:

After implementing the namespace isolation component, verify correct behavior:

1. **Compile and run**: `gcc -o sandbox src/*.c && sudo ./sandbox /bin/bash`
2. **Expected behavior**:
   - Child process should see itself as PID 1: `echo $$` returns `1`
   - Process list isolation: `ps aux` shows only processes in namespace
   - Network isolation: `ip addr` shows only loopback interface
   - Hostname isolation: `hostname` shows isolated or custom hostname
   - Filesystem namespace: mount changes don't affect host system
3. **Validation commands** (run inside sandbox):

```bash
echo $$                    # Should print: 1
ps aux | wc -l             # Should show very few processes
cat /proc/self/ns/pid      # Different namespace ID than host
ping 8.8.8.8               # Should fail (network isolated)
hostname                   # Should show custom/isolated hostname
```

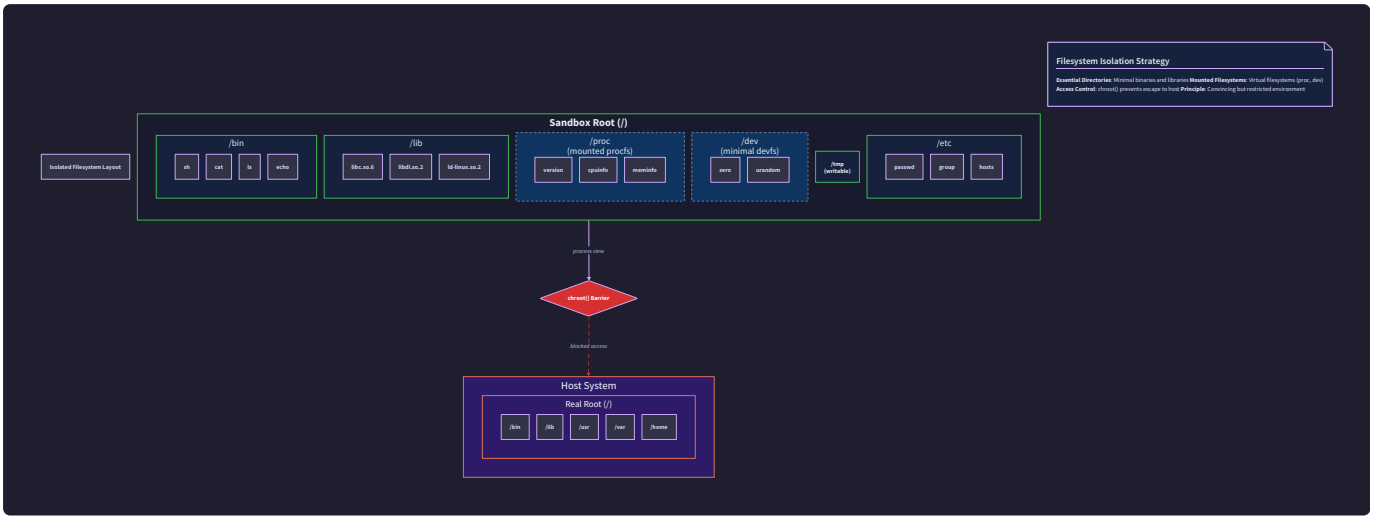
4. **Signs of problems**:
   - **Child PID not 1**: PID namespace creation failed or `/proc` not remounted
   - **Can see host processes**: Mount namespace not created or `/proc` issue
   - **Network connectivity works**: Network namespace creation failed
   - **Same hostname as host**: UTS namespace creation failed
   - **Permission errors**: Insufficient privileges or user mapping problems



Filesystem Isolation Component

> **Milestone(s):** This section corresponds to Milestone 2 (Filesystem Isolation) and builds upon the namespace isolation from Milestone 1, providing the foundation for secure system call filtering in Milestone 3

The filesystem isolation component creates a **convincing but restricted filesystem environment** for sandboxed processes. While namespace isolation provides the container for our security boundaries, filesystem isolation defines what the sandboxed process can actually see and access within those boundaries. This component transforms a potentially dangerous process that expects full system access into one that operates within a carefully crafted minimal environment.



Mental Model: The Fake Building

Understanding filesystem isolation requires thinking about it like constructing a **fake building** for a movie set. When filming a scene that takes place in a hospital, the production team doesn't need to build an actual functioning hospital—they only need to construct the rooms and corridors that will appear on camera, fill them with convincing props, and ensure the actors can't accidentally wander off the set into the real world.

Similarly, filesystem isolation creates a **convincing but minimal environment** that contains everything the sandboxed process expects to find (essential binaries, libraries, device files) while ensuring it cannot escape to access the real host

filesystem. The sandboxed process believes it's running on a complete Linux system, but it's actually operating within a carefully constructed façade that contains only what it needs and nothing more.

Just as a movie set has controlled entry and exit points, **chroot** or **pivot_root** operations define the boundaries of this fake environment. The process can explore freely within its artificial world, but the walls are designed to be impenetrable—it cannot climb out of its confined space to reach the production equipment (real host system) beyond the set.

The key insight is that **most processes don't actually need access to the entire filesystem hierarchy**. A typical application might only need access to its own executable, a few shared libraries, basic device files like `/dev/null`, and a temporary scratch space. By providing exactly these resources and nothing more, we create an environment that's functionally complete for the application but severely limited from an attacker's perspective.

Minimal Root Filesystem Construction

Building the minimal root filesystem requires understanding **exactly what components are necessary** for the sandboxed process to function while excluding everything that could be exploited for privilege escalation or information disclosure. This construction process involves identifying dependencies, creating directory structures, and populating the environment with only essential files.

The foundation of any Linux filesystem environment consists of several **critical directory structures** that processes expect to find. The `/bin` directory must contain essential executables that the sandboxed process might invoke, such as `/bin/sh` if the process needs to spawn shell commands. The `/lib` and `/lib64` directories house shared libraries that executables depend on—these are discovered by analyzing the dynamic linking requirements of all binaries included in the sandbox.

Library dependency resolution represents one of the most complex aspects of minimal filesystem construction. Every executable and shared library can depend on additional libraries, creating a dependency tree that must be fully resolved to avoid runtime failures. The `ldd` command reveals immediate dependencies, but this analysis must be performed recursively since dependencies can have their own dependencies. Dynamic linking also involves the runtime linker itself (typically `/lib64/ld-linux-x86-64.so.2` on x86_64 systems), which must be present and functional.

The `/dev` directory requires careful population with **device files that applications commonly expect**. Most processes assume they can access `/dev/null` for discarding output, `/dev/zero` for obtaining null bytes, `/dev/random` and `/dev/urandom` for entropy, and `/dev/stdin`, `/dev/stdout`, `/dev/stderr` for standard I/O operations. However, dangerous device files like `/dev/mem` (direct memory access) or `/dev/kmem` (kernel memory) must be explicitly excluded to prevent privilege escalation.

Configuration and data files represent another category that requires selective inclusion. Many applications expect to find configuration files in `/etc`, such as `/etc/passwd` for user information, `/etc/group` for group definitions, or `/etc/resolv.conf` for DNS resolution. However, these files should be created specifically for the sandbox environment rather than copied from the host system, as host configuration files might contain sensitive information or references to resources outside the sandbox.

The minimal filesystem construction process follows a **systematic approach** to ensure completeness while maintaining security:

1. **Analyze the target executable** using `ldd` and similar tools to identify all shared library dependencies
2. **Create the essential directory structure** with appropriate permissions: `/bin`, `/lib`, `/lib64`, `/dev`, `/proc`, `/tmp`, `/etc`

3. **Copy required executables** to `/bin`, ensuring they have appropriate permissions (typically 755) and are owned by root
4. **Recursively copy all library dependencies** to `/lib` and `/lib64`, following the same ownership and permission patterns as the host system
5. **Create essential device files** using `mknod` with appropriate major and minor numbers, limiting to safe devices only
6. **Generate minimal configuration files** in `/etc` with sandbox-specific content that doesn't expose host information
7. **Set up temporary and writable areas** that the process can use for scratch space without affecting the host system

The **size optimization** of the minimal filesystem becomes critical when running multiple sandbox instances. Each sandbox requires its own filesystem copy to maintain isolation, so minimizing the size reduces memory usage and creation time. This optimization involves identifying which libraries are truly necessary versus those that are transitively included but never used, and potentially using static linking for simple executables to eliminate library dependencies entirely.

Directory	Purpose	Security Considerations	Typical Contents
`/bin`	Essential executables	Include only necessary binaries	`sh`, target executable
`/lib`, `/lib64`	Shared libraries	Resolve all dependencies	`libc.so.6`, `ld-linux.so.2`
`/dev`	Device files	Include only safe devices	`null`, `zero`, `random`, `urandom`
`/etc`	Configuration files	Create sandbox-specific configs	`passwd`, `group`, `hosts`
`/tmp`	Temporary storage	Writable, size-limited	Empty directory with 1777 permissions
`/proc`	Process information	Mount read-only when possible	Kernel-provided filesystem

Chroot vs Pivot Root Decision

The choice between **chroot** and **pivot_root** for filesystem isolation represents a fundamental architectural decision that affects both security robustness and implementation complexity. Both mechanisms change the apparent root directory for processes, but they operate through different principles and provide different levels of isolation guarantees.

Chroot operates by changing the root directory reference for a process and its children without affecting the underlying mount namespace structure. When a process calls `chroot("/new/root")`, the kernel updates the process's root directory pointer, making all absolute paths resolve relative to the new location. However, the old root filesystem remains mounted and potentially accessible through various escape mechanisms, particularly if the process has sufficient privileges or can manipulate file descriptors that point outside the chroot environment.

Pivot_root, conversely, operates at the mount namespace level by atomically swapping two mount points—making the new root the actual root of the mount namespace while moving the old root to a specified location within the new hierarchy. This approach provides stronger isolation because it actually changes the mount namespace structure rather than just adjusting process-level path resolution.

Decision: Use pivot_root for filesystem isolation

- **Context**: Need robust filesystem isolation that prevents sandbox escape attempts while maintaining clean separation from host filesystem
- **Options Considered**: chroot-only approach, pivot_root with old root cleanup, hybrid approach using both mechanisms
- **Decision**: Implement pivot_root with immediate cleanup of old root references
- **Rationale**: Pivot_root provides stronger security guarantees by actually restructuring the mount namespace rather than relying on path resolution changes. It eliminates many classical chroot escape vectors and provides cleaner separation between sandbox and host environments.
- **Consequences**: Requires mount namespace isolation (already provided by Milestone 1), increases implementation complexity, but significantly improves security posture against filesystem-based escape attempts.

Approach	Security Strength	Implementation Complexity	Escape Vectors	Mount Namespace Required
chroot only	Moderate	Low	fd-based, /proc traversal	No
pivot_root	High	Moderate	Limited, requires root access	Yes
hybrid	High	High	Minimal	Yes

The **pivot_root implementation sequence** requires careful coordination with mount namespace creation to ensure proper isolation. The process begins by creating the minimal root filesystem in a temporary location, then setting up a new mount namespace where pivot_root can operate safely. The old root must be moved to a location within the new root hierarchy before being unmounted and removed to prevent any references from persisting.

Security implications of this choice extend beyond simple path resolution. Chroot environments can be escaped through various mechanisms: processes with sufficient privileges can create device files to access raw disk devices, file descriptors opened before the chroot can provide access to external files, and /proc filesystem entries can potentially be traversed to reach the original root. Pivot_root eliminates most of these vectors by actually restructuring the filesystem view rather than layering path resolution changes on top of the existing structure.

Error handling complexity differs significantly between approaches. Chroot failures are typically straightforward—either the target directory exists and is accessible, or the operation fails. Pivot_root failures can be more complex, involving mount namespace state, filesystem busy conditions, or cleanup of partially completed operations. The implementation must handle these failure modes gracefully while ensuring that failed sandbox creation doesn't leave mount artifacts on the host system.

Mount Management

Effective mount management ensures that the sandbox filesystem contains necessary kernel-provided filesystems while preventing escape vectors through filesystem traversal or mount manipulation. The sandbox requires access to **pseudo-filesystems** like `/proc`, `/dev`, and `/sys` for normal operation, but these mounts must be configured to prevent information disclosure and privilege escalation.

Proc filesystem mounting provides essential process and system information that applications frequently require. However, the standard `/proc` mount reveals extensive information about the host system, including all running processes, kernel configuration, and system resource usage. The sandbox implementation must mount `/proc` within the isolated filesystem while ensuring that the **PID namespace isolation** from Milestone 1 limits the visible process information to only processes within the same namespace.

The `/proc` mount also contains sensitive entries that can be exploited for container escape, particularly `/proc/*/root` symlinks that point to the root directory of arbitrary processes. If a sandboxed process can access `/proc/1/root` (assuming PID 1 exists outside the sandbox), it might be able to traverse back to the host filesystem. **Proper PID namespace isolation** prevents this by ensuring that external processes are not visible within the sandbox's `/proc` view.

Dev filesystem population requires creating device files that applications expect while excluding dangerous devices that could enable privilege escalation. The sandbox needs basic devices like `/dev/null`, `/dev/zero`, and `/dev/random`, but it must not include devices like `/dev/mem` or `/dev/kmem` that provide raw system access. Rather than mounting the host's `/dev` directly, the implementation creates a new tmpfs mount at `/dev` and populates it with only the necessary device files.

Tmpfs mounts provide writable storage areas within the sandbox without affecting the host filesystem. These mounts are backed by memory and swap space rather than persistent storage, ensuring that any modifications made by the sandboxed process are automatically cleaned up when the sandbox terminates. Common tmpfs mount points include `/tmp` for temporary files and potentially `/var` or application-specific directories that need write access.

The **mount sequence** must be carefully ordered to avoid conflicts and ensure proper layering of filesystem views:

1. **Create and mount the new root filesystem** using bind mounts or copying files to a tmpfs
2. **Set up pivot_root preparation** by creating the old root directory within the new root
3. **Execute pivot_root operation** to swap the root filesystem
4. **Mount /proc within the new root** to provide process information within the PID namespace
5. **Create and populate /dev** as a tmpfs with necessary device files
6. **Mount additional tmpfs filesystems** for writable areas like `/tmp`
7. **Unmount and remove the old root** to complete the isolation
8. **Apply mount options** like read-only, noexec, nosuid, nodev where appropriate

Mount option security plays a crucial role in preventing various attack vectors. The `noexec` option prevents execution of binaries from specific mount points, which is particularly important for writable areas like `/tmp`. The `nosuid` option prevents set-user-ID and set-group-ID bits from taking effect, limiting privilege escalation attempts. The `nodev` option prevents device files from being interpreted as such, even if they have the correct major and minor numbers.

Mount Point	Type	Options	Purpose	Security Considerations
`/`	tmpfs or bind	ro, nodev	Root filesystem	Read-only prevents modification
`/proc`	proc	ro, nosuid, nodev, noexec	Process information	Limited by PID namespace
`/dev`	tmpfs	rw, nosuid, noexec	Device files	Manually populated with safe devices
`/tmp`	tmpfs	rw, noexec, nosuid, nodev	Temporary storage	Prevent execution and privilege escalation
`/var/tmp`	tmpfs	rw, noexec, nosuid, nodev	Application temp	Size-limited scratch space

Bind mount restrictions prevent the sandboxed process from accessing sensitive host filesystem locations even if it manages to create new mount points. The implementation must ensure that no bind mounts provide access to locations like `/etc/shadow`, `/proc` from the host mount namespace, or any directory containing sensitive files. This restriction is typically enforced by ensuring that the sandbox root filesystem is completely separate from the host filesystem rather than layering access controls on top of host directories.

Architecture Decisions

The filesystem isolation component requires several critical architectural decisions that balance security, performance, and maintainability. These decisions affect how the sandbox creates, manages, and cleans up filesystem environments across multiple concurrent instances.

> **Decision: Use tmpfs-backed sandbox filesystems with COW optimization**
>
> - **Context**: Need isolated filesystem for each sandbox instance while minimizing memory usage and creation time for multiple concurrent sandboxes
> - **Options Considered**: Directory copying per sandbox, tmpfs with full file copying, COW filesystem overlays, shared read-only base with per-instance writable layers
> - **Decision**: Implement tmpfs-based sandboxes with copy-on-write overlays for shared components
> - **Rationale**: Tmpfs provides automatic cleanup and memory-backed performance, while COW overlays allow sharing of common files (libraries, binaries) across multiple sandbox instances, significantly reducing memory footprint
> - **Consequences**: Requires overlay filesystem support in kernel, increases implementation complexity, but provides excellent scalability for multiple concurrent sandboxes

Filesystem lifecycle management determines how sandbox filesystems are created, maintained, and destroyed throughout the sandbox lifetime. The implementation must handle creation failures gracefully, ensure proper cleanup even when sandboxed processes terminate unexpectedly, and provide isolation between concurrent sandbox instances sharing the same host system.

The **creation process** begins with establishing a new mount namespace (provided by the namespace isolation component) and then constructing the minimal filesystem within that namespace. This construction can follow several

strategies: copying files from a template directory, extracting from a prepared archive, or assembling from shared components using overlay mounts. Each approach offers different trade-offs between creation time, memory usage, and customization flexibility.

Template-based creation involves preparing a complete minimal filesystem template during sandbox initialization and then copying or bind-mounting this template for each new sandbox instance. This approach provides fast startup times since the dependency resolution and file copying is done once during initialization rather than for each sandbox. However, it requires more disk space and doesn't allow for per-sandbox customization of the filesystem contents.

> **Decision: Implement read-only base with per-sandbox writable overlays**
>
> - **Context**: Balance between sharing common components across sandboxes while allowing per-instance modifications and maintaining strict isolation
> - **Options Considered**: Complete filesystem copying, shared read-only base, overlay filesystems, union mounts
> - **Decision**: Create shared read-only base filesystem with per-sandbox tmpfs overlays for writable areas
> - **Rationale**: Maximizes sharing of common files (executables, libraries) while ensuring complete isolation for writable areas. Tmpfs overlays provide automatic cleanup and prevent persistent storage of sandbox modifications
> - **Consequences**: Requires careful management of overlay mount points and cleanup, but provides optimal memory efficiency and strong isolation guarantees

Writable area management addresses the need for sandboxed processes to create temporary files, modify application state, or generate output while preventing these modifications from affecting other sandbox instances or persisting beyond the sandbox lifetime. The implementation uses **tmpfs mounts** for writable areas, ensuring that all modifications are memory-backed and automatically cleaned up when the sandbox terminates.

The overlay approach requires **careful mount point management** to ensure that writable areas are properly isolated while shared areas remain accessible. The implementation creates tmpfs mounts at specific locations (`/tmp`, `/var/tmp`, potentially application-specific directories) and uses bind mounts or overlay filesystems to compose the final filesystem view that the sandboxed process observes.

Cleanup and resource management becomes critical when running multiple sandbox instances concurrently. Each sandbox creates mount points, temporary directories, and potentially overlay filesystems that must be properly cleaned up to prevent resource leaks. The cleanup process must handle cases where the sandboxed process terminates cleanly, crashes unexpectedly, or becomes unresponsive.

The **cleanup sequence** follows a specific order to avoid conflicts and ensure complete resource deallocation:

1. **Terminate the sandboxed process** if it's still running
2. **Unmount all tmpfs and overlay mounts** in reverse order of creation
3. **Remove temporary directories** created for the sandbox instance
4. **Clean up any bind mounts** that were created for the sandbox
5. **Verify that all resources are freed** and log any cleanup failures for debugging

Architecture Aspect	Chosen Approach	Alternative	Trade-off Rationale
Base filesystem	Shared read-only template	Per-sandbox copying	Memory efficiency for concurrent instances
Writable areas	Per-sandbox tmpfs overlays	Shared writable space	Complete isolation between instances
Library sharing	Shared base with COW	Individual copies	Reduced memory footprint
Cleanup strategy	Automatic tmpfs cleanup	Manual file deletion	Guaranteed cleanup even on crashes
Mount namespace	Per-sandbox isolation	Shared with restrictions	Complete filesystem isolation

Performance optimization considerations affect how the filesystem isolation component handles concurrent sandbox creation and teardown. Creating a new sandbox involves multiple filesystem operations (mounting, copying, device file creation) that can become a bottleneck when launching many sandboxes simultaneously. The implementation should pre-compute as much as possible and use efficient copying mechanisms like hard links where security permits.

Error recovery mechanisms ensure that partially created sandbox filesystems don't leave artifacts on the host system. If sandbox creation fails partway through the process, the cleanup logic must be able to identify and remove any mount points, temporary directories, or other resources that were created before the failure. This requires maintaining state about what resources have been allocated and ensuring that the cleanup code can handle incomplete resource sets.

Common Pitfalls

Filesystem isolation implementation involves several complex interactions between mount namespaces, file permissions, and process privileges that create opportunities for subtle but dangerous mistakes. Understanding these common pitfalls helps avoid security vulnerabilities and operational issues that can compromise the entire sandbox system.

⚠️ Pitfall: Incomplete Library Dependency Resolution

Many implementations fail to properly resolve all shared library dependencies, leading to runtime failures when the sandboxed process attempts to load missing libraries. This issue often manifests intermittently because some libraries are loaded only when specific code paths are executed, making the problem difficult to detect during initial testing.

The root cause typically involves **incomplete recursive dependency analysis**. Using `ldd` on the main executable reveals immediate dependencies, but those libraries may themselves depend on additional libraries that are loaded dynamically. For example, `libc.so.6` might depend on `ld-linux.so.2`, and various libraries might require plugins or modules that are loaded at runtime based on configuration or feature usage.

Detection: The sandboxed process fails with "cannot open shared object file" errors, often occurring only when specific features are used rather than immediately at startup.

Solution: Implement recursive dependency scanning that follows the entire dependency tree, use tools like `objdump -p` to examine dynamic section requirements, and consider using `strace` on the target executable in a non-sandboxed environment to identify all libraries actually loaded during typical operation.

⚠️ Pitfall: Insufficient /dev Device Population

Sandboxed processes frequently fail because they cannot access expected device files in `/dev`, but the symptoms often appear as generic I/O errors rather than clearly indicating missing devices. Applications assume that standard devices like `/dev/null`, `/dev/zero`, and `/dev/urandom` are always available, and their absence can cause unexpected behavior.

Critical devices that are commonly required include `/dev/null` (for output redirection), `/dev/zero` (for memory mapping), `/dev/random` and `/dev/urandom` (for entropy), and the standard I/O devices (`/dev/stdin` , `/dev/stdout` , `/dev/stderr`). Missing any of these can cause applications to fail in non-obvious ways.

Detection: Applications fail with permission denied errors when attempting I/O operations, random number generation fails, or processes hang when trying to access entropy sources.

Solution: Create a comprehensive device population function that creates all standard devices with correct major/minor numbers and permissions. Use `ls -la /dev` on the host system to identify the correct device parameters, but be selective about which devices to include.

```c
// Essential devices with major/minor numbers and permissions
struct device_spec {
    char *name;
    mode_t mode;
    dev_t device_number;
};

static struct device_spec essential_devices[] = {
    {"/dev/null", S_IFCHR | 0666, makedev(1, 3)},
    {"/dev/zero", S_IFCHR | 0666, makedev(1, 5)},
    {"/dev/random", S_IFCHR | 0444, makedev(1, 8)},
    {"/dev/urandom", S_IFCHR | 0444, makedev(1, 9)},
    // ... additional devices
};
```

⚠️ Pitfall: Mount Point Cleanup Race Conditions

Improper cleanup of mount points can lead to resource leaks, permission errors, or in severe cases, interference between different sandbox instances. The cleanup process must handle cases where mount points are busy, processes are still accessing mounted filesystems, or the cleanup itself is interrupted.

Race conditions commonly occur when the sandboxed process is still running when cleanup begins, when multiple threads are accessing the filesystem simultaneously, or when the cleanup process itself fails partway through the sequence. These races can leave orphaned mount points that consume system resources and potentially interfere with subsequent sandbox creation.

Detection: Mount points remain visible in `/proc/mounts` after sandbox termination, "device busy" errors during cleanup, or mounting failures in subsequent sandboxes due to existing mount points.

Solution: Implement a robust cleanup sequence that first ensures all processes using the mount points have terminated, then unmounts in reverse order of creation, and finally verifies that all mount points have been successfully removed.

⚠️ Pitfall: Inadequate /proc Mount Security

Mounting `/proc` within the sandbox without proper consideration of PID namespace isolation can leak information about host processes or provide escape vectors through `/proc` entries that reference external resources. The `/proc` filesystem contains numerous entries that can be exploited if not properly restricted.

Dangerous /proc entries include `/proc/*/root` (symlinks to process root directories), `/proc/*/cwd` (current working directories), `/proc/*/fd/*` (file descriptor symlinks), and various entries under `/proc/sys` that can affect system behavior. If the sandboxed process can see processes outside its PID namespace, these entries provide potential escape routes.

Detection: The sandboxed process can see host processes in `/proc`, can traverse `/proc/*/root` to access the host filesystem, or can access `/proc/sys` entries that should be restricted.

Solution: Ensure that PID namespace isolation is properly configured before mounting `/proc`, consider mounting `/proc` read-only, and potentially use mount options or bind mounts to restrict access to sensitive `/proc` subdirectories.

⚠️ Pitfall: Writable Area Permission Confusion

Incorrect permissions on writable areas can either prevent legitimate application operation or provide unintended privilege escalation opportunities. The permission model must balance functional requirements (allowing the application to create and modify files) with security constraints (preventing exploitation of set-user-ID bits or device file creation).

Permission inheritance from the host system can create unexpected security holes. If writable directories inherit permissive permissions or special flags from their parent directories, sandboxed processes might be able to create files with unintended capabilities.

Detection: Applications cannot create necessary temporary files, or conversely, applications can create files with elevated privileges or special capabilities that should not be permitted.

Solution: Explicitly set permissions on all mount points and directories rather than inheriting from templates, use mount options like `nosuid`, `nodev`, and `noexec` on writable areas, and verify permissions after filesystem setup is complete.

⚠️ Pitfall: Pivot Root State Confusion

The pivot_root operation requires specific preconditions about mount point states and can fail with cryptic error messages if these conditions are not met. The operation must be performed within a mount namespace, the new root must be a mount point, and the old root destination must exist within the new root hierarchy.

State requirements for pivot_root include: both old and new root must be mount points, the old root must not be the same as the new root, the new root and its parent must not be on the same mount as the current root, and the calling process must have CAP_SYS_ADMIN capability.

Detection: pivot_root fails with "Invalid argument" or "Device or resource busy" errors, or the operation appears to succeed but the filesystem layout is incorrect.

Solution: Carefully prepare the mount namespace state before attempting pivot_root, verify that all preconditions are met, and implement comprehensive error handling that can diagnose and report specific failure conditions.

Pitfall	Symptom	Root Cause	Prevention
Missing libraries	Runtime "cannot open shared object" errors	Incomplete dependency resolution	Recursive dependency scanning
Missing devices	Generic I/O failures or hangs	Unpopulated /dev directory	Comprehensive device creation
Mount leaks	Resource exhaustion, mounting failures	Improper cleanup ordering	Robust cleanup with verification
/proc exposure	Information disclosure, escape vectors	Inadequate namespace isolation	Proper PID namespace before /proc mount
Permission errors	Application failures or privilege escalation	Incorrect writable area permissions	Explicit permission setting with mount options
pivot_root failure	Filesystem isolation setup failure	Unmet mount state preconditions	State verification before pivot_root

Implementation Guidance

The filesystem isolation component requires careful orchestration of mount operations, file system construction, and cleanup procedures. This implementation guidance provides concrete code structure and examples for building a robust filesystem isolation system.

Technology Recommendations

Component	Simple Option	Advanced Option
Filesystem Construction	Direct file copying with `cp -r`	Overlay filesystems with `mount -t overlay`
Mount Management	Basic `mount` / `umount` syscalls	`libmount` for complex mount operations
Device File Creation	Manual `mknod` calls	`udev` rules with restricted device sets
Dependency Resolution	Static `ldd` analysis	Dynamic tracing with `LD_TRACE_LOADED_OBJECTS`
Template Management	Directory-based templates	Compressed filesystem images

Recommended File Structure

```
project-root/
├── src/
│   ├── sandbox.c                    ← main sandbox orchestration
│   ├── filesystem/
│   │   ├── fs_isolation.c           ← this component (core logic)
│   │   ├── fs_isolation.h           ← filesystem isolation interface
│   │   ├── mount_manager.c          ← mount operations and cleanup
│   │   ├── template_builder.c       ← minimal filesystem construction
│   │   └── dependency_resolver.c    ← library dependency analysis
│   ├── namespace/
│   │   └── ns_isolation.c           ← from previous milestone
│   └── common/
│       ├── error_handling.c         ← error reporting utilities
│       └── privilege_utils.c        ← privilege checking functions
├── templates/
│   └── minimal_rootfs/              ← pre-built filesystem template
│       ├── bin/
│       ├── lib/
│       ├── lib64/
│       └── dev/
└── tests/
    ├── test_filesystem.c            ← filesystem isolation tests
    └── integration/
        └── test_full_sandbox.c      ← end-to-end testing
```

Infrastructure Starter Code

Mount Management Utilities (Complete implementation):

```c
// mount_manager.c - Complete mount management utilities

#include <sys/mount.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>


// Mount point tracking for cleanup

struct mount_entry {

    char *path;

    struct mount_entry *next;

};


static struct mount_entry *mount_list = NULL;


// Add mount point to tracking list

static void track_mount(const char *path) {

    struct mount_entry *entry = malloc(sizeof(struct mount_entry));

    entry->path = strdup(path);

    entry->next = mount_list;

    mount_list = entry;

}


// Create and mount tmpfs with specified options

int create_tmpfs_mount(const char *target, const char *options, size_t size_mb) {

    char mount_opts[256];

    snprintf(mount_opts, sizeof(mount_opts), "%s,size=%zum",

             options ? options : "rw,nosuid,nodev", size_mb);



    if (mkdir(target, 0755) < 0 && errno != EEXIST) {
```

```
        perror("mkdir for tmpfs mount");
        return SANDBOX_ERROR_FILESYSTEM;
    }

    if (mount("tmpfs", target, "tmpfs", 0, mount_opts) < 0) {
        perror("tmpfs mount failed");
        return SANDBOX_ERROR_FILESYSTEM;
    }

    track_mount(target);
    return SANDBOX_SUCCESS;
}

// Cleanup all tracked mount points
void cleanup_all_mounts(void) {
    struct mount_entry *current = mount_list;
    while (current) {
        if (umount2(current->path, MNT_DETACH) < 0) {
            printf("Warning: failed to unmount %s: %s\n",
                   current->path, strerror(errno));
        }
        struct mount_entry *next = current->next;
        free(current->path);
        free(current);
        current = next;
    }
    mount_list = NULL;
}
```

Device File Creation Utilities (Complete implementation):

```c
// device_manager.c - Complete device file creation

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>


struct device_spec {

    const char *name;

    mode_t mode;

    int major;

    int minor;

};


// Essential devices for sandbox environment

static const struct device_spec essential_devices[] = {

    {"null", S_IFCHR | 0666, 1, 3},

    {"zero", S_IFCHR | 0666, 1, 5},

    {"full", S_IFCHR | 0666, 1, 7},

    {"random", S_IFCHR | 0444, 1, 8},

    {"urandom", S_IFCHR | 0444, 1, 9},

    {"tty", S_IFCHR | 0666, 5, 0},

    {NULL, 0, 0, 0} // terminator

};


int create_essential_devices(const char *dev_path) {

    char device_path[PATH_MAX];


    for (int i = 0; essential_devices[i].name != NULL; i++) {

        snprintf(device_path, sizeof(device_path), "%s/%s",

                 dev_path, essential_devices[i].name);
```

```
        dev_t dev = makedev(essential_devices[i].major, essential_devices[i].minor);
        if (mknod(device_path, essential_devices[i].mode, dev) < 0) {
            if (errno != EEXIST) {
                perror("mknod failed");
                return SANDBOX_ERROR_FILESYSTEM;
            }
        }

        // Ensure correct ownership and permissions
        if (chown(device_path, 0, 0) < 0) {
            perror("chown device failed");
            return SANDBOX_ERROR_FILESYSTEM;
        }
    }

    // Create standard I/O symlinks
    snprintf(device_path, sizeof(device_path), "%s/stdin", dev_path);
    if (symlink("/proc/self/fd/0", device_path) < 0 && errno != EEXIST) {
        perror("stdin symlink failed");
        return SANDBOX_ERROR_FILESYSTEM;
    }

    snprintf(device_path, sizeof(device_path), "%s/stdout", dev_path);
    if (symlink("/proc/self/fd/1", device_path) < 0 && errno != EEXIST) {
        perror("stdout symlink failed");
        return SANDBOX_ERROR_FILESYSTEM;
    }

    snprintf(device_path, sizeof(device_path), "%s/stderr", dev_path);
    if (symlink("/proc/self/fd/2", device_path) < 0 && errno != EEXIST) {
```

```
        perror("stderr symlink failed");
        return SANDBOX_ERROR_FILESYSTEM;
    }

    return SANDBOX_SUCCESS;
}
```

Core Logic Skeleton Code

Filesystem Isolation Main Interface:

```c
// fs_isolation.h - Main filesystem isolation interface
#ifndef FS_ISOLATION_H
#define FS_ISOLATION_H

#include "../common/sandbox_types.h"

// Filesystem isolation state
typedef struct {
    char *sandbox_root;          // Root directory of sandbox filesystem
    char *old_root_path;         // Path where old root is moved during pivot
    int mount_ns_fd;             // Mount namespace file descriptor
    int cleanup_needed;          // Whether cleanup is required
} fs_isolation_state_t;

// Initialize filesystem isolation for sandbox
int setup_filesystem_isolation(sandbox_config_t *config, fs_isolation_state_t *fs_state);

// Clean up filesystem isolation resources
void cleanup_filesystem_isolation(fs_isolation_state_t *fs_state);

// Create minimal root filesystem from template
int create_minimal_rootfs(const char *template_path, const char *sandbox_root);

#endif
```

Main Filesystem Isolation Logic:

```c
// fs_isolation.c - Core filesystem isolation implementation

#include "fs_isolation.h"

#include "../common/error_handling.h"

#include <sys/mount.h>

#include <sys/stat.h>

#include <unistd.h>

#include <stdlib.h>

#include <string.h>


int setup_filesystem_isolation(sandbox_config_t *config, fs_isolation_state_t *fs_state) {

    // TODO 1: Create temporary directory for sandbox root filesystem

    // Use mkdtemp() to create unique directory like /tmp/sandbox_XXXXXX

    // Store the path in fs_state->sandbox_root



    // TODO 2: Create minimal root filesystem in sandbox root

    // Call create_minimal_rootfs() with template path and sandbox root

    // Handle failure by cleaning up created directory



    // TODO 3: Prepare old root directory within new root

    // Create directory like sandbox_root/old_root for pivot_root

    // Store path in fs_state->old_root_path



    // TODO 4: Perform pivot_root operation

    // Call pivot_root(fs_state->sandbox_root, fs_state->old_root_path)

    // This swaps the root filesystem and moves old root to specified location



    // TODO 5: Set up essential mount points in new root

    // Mount /proc with: mount("proc", "/proc", "proc", MS_NOSUID|MS_NODEV|MS_NOEXEC, NULL)

    // Create and populate /dev with tmpfs and essential devices

    // Create tmpfs mount for /tmp with size limits
```

```
    // TODO 6: Clean up old root references
    // Unmount old root: umount2("/old_root", MNT_DETACH)
    // Remove old root directory: rmdir("/old_root")

    // TODO 7: Apply read-only restrictions where appropriate
    // Remount root as read-only if config specifies: mount(NULL, "/", NULL, MS_REMOUNT|MS_RDONLY, NULL)
    // Apply noexec, nosuid, nodev flags to appropriate mount points

    fs_state->cleanup_needed = 1;
    return SANDBOX_SUCCESS;
}

void cleanup_filesystem_isolation(fs_isolation_state_t *fs_state) {
    // TODO 1: Check if cleanup is needed
    // Return early if fs_state->cleanup_needed is 0

    // TODO 2: Unmount all sandbox mount points
    // Call cleanup_all_mounts() to unmount tracked mount points
    // Handle umount failures gracefully - log warnings but continue cleanup

    // TODO 3: Remove sandbox root directory
    // Use recursive directory removal to clean up fs_state->sandbox_root
    // Handle permission errors that might require elevated privileges

    // TODO 4: Free allocated memory
    // Free fs_state->sandbox_root and fs_state->old_root_path
    // Set pointers to NULL and cleanup_needed to 0
}
```

```c
int create_minimal_rootfs(const char *template_path, const char *sandbox_root) {
    // TODO 1: Verify template directory exists and is accessible
    // Check that template_path exists and contains required subdirectories
    // Return SANDBOX_ERROR_CONFIG if template is invalid

    // TODO 2: Create directory structure in sandbox root
    // Create essential directories: /bin, /lib, /lib64, /dev, /proc, /tmp, /etc
    // Set appropriate permissions (755 for directories)

    // TODO 3: Copy essential binaries from template
    // Copy files from template_path/bin to sandbox_root/bin
    // Preserve permissions and ownership where possible

    // TODO 4: Copy library dependencies
    // Copy shared libraries from template to sandbox lib directories
    // Ensure all dependencies are included for copied binaries

    // TODO 5: Create essential configuration files
    // Generate minimal /etc/passwd, /etc/group with sandbox user
    // Create basic /etc/hosts, /etc/resolv.conf if needed

    // TODO 6: Set up device directory structure
    // Create /dev directory and call create_essential_devices()
    // Ensure device files have correct permissions and ownership

    return SANDBOX_SUCCESS;
}
```

Language-Specific Hints

C-Specific Implementation Notes:

- Use `pivot_root()` syscall directly rather than calling external mount command
- Handle `errno` values carefully - `EBUSY` often indicates processes still using mount points
- Use `MNT_DETACH` flag with `umount2()` for lazy unmounting during cleanup
- `makedev()` macro creates device numbers from major/minor values
- Use `mkdtemp()` for creating unique temporary directories safely

Error Handling Patterns:

- Always check return values from mount operations - they fail frequently due to permissions or state issues
- Use `strerror(errno)` to get descriptive error messages for debugging
- Implement cleanup functions that can handle partial failure states
- Log mount operations for debugging - knowing what was mounted helps with troubleshooting

Memory Management:

- Use `strdup()` for copying path strings that need to persist
- Free all allocated paths in cleanup functions
- Be careful with stack-allocated buffers for path construction - use `PATH_MAX`
- Consider using `asprintf()` for dynamic string formatting of paths

Milestone Checkpoint

After implementing the filesystem isolation component, verify the following behavior:

Test Command: Compile and run the sandbox with a simple test program that tries to access various filesystem locations.

```bash
gcc -o sandbox src/sandbox.c src/filesystem/*.c src/common/*.c

sudo ./sandbox /bin/ls /
```

Expected Output: The sandboxed `ls` command should show only the minimal filesystem contents (bin, lib, dev, proc, tmp) rather than the full host filesystem.

Verification Steps:

1. **Filesystem Isolation**: The sandboxed process should not see host directories like `/home`, `/root`, `/boot`
2. **Device Access**: Test that `/dev/null`, `/dev/zero` work: `echo test > /dev/null` succeeds
3. **Process Information**: `/proc/self/` should exist and show the sandboxed process information
4. **Write Permissions**: Writing to `/tmp` should succeed, writing to `/bin` should fail
5. **Library Resolution**: Complex programs should still run despite minimal library set

Common Issues and Debugging:

- **"No such file or directory" for libraries**: Check dependency resolution in template creation
- **"Permission denied" on device files**: Verify device creation and permissions
- **"Invalid argument" from pivot_root**: Check mount namespace creation and directory setup
- **Processes hanging**: Usually indicates missing `/dev/random` or similar device files
- **Mount cleanup failures**: Check that no processes are still running in the namespace

System Call Filtering Component

> **Milestone(s):** This section corresponds to Milestone 3 (Seccomp System Call Filtering) and builds upon the namespace and filesystem isolation from Milestones 1-2, providing system call-level restrictions that will be enhanced by resource limits (Milestone 4) and capability dropping (Milestone 5)

The system call filtering component represents the third layer in our defense-in-depth sandbox architecture. While namespace isolation creates separate views of system resources and filesystem isolation restricts access to files and directories, system call filtering operates at the most fundamental level—controlling which kernel operations the sandboxed process can invoke. This component uses Linux's **seccomp** (secure computing) mechanism with Berkeley Packet Filter (BPF) programs to create a kernel-enforced whitelist of permitted system calls.

Think of seccomp as installing a strict security checkpoint between user space and kernel space. Every time the sandboxed process attempts to invoke a system call—whether to open a file, allocate memory, or communicate over a network—the seccomp filter program examines the request and decides whether to allow it, block it, or terminate the process entirely. This filtering happens at the kernel level, making it impossible for malicious code to bypass through clever programming tricks or exploits.

Mental Model: The Strict Gatekeeper

Understanding seccomp requires thinking about the fundamental boundary between user space and kernel space in Linux systems. Every program runs in user space, which provides a safe, virtualized environment with limited privileges. When a program needs to perform privileged operations—reading files, allocating memory, creating network connections—it must request these services from the kernel through **system calls**.

Imagine the kernel as a high-security government building, and system calls as different types of access requests. Normally, any program with appropriate permissions can enter through various doors: the "file access" entrance, the "network communication" lobby, the "process creation" wing. Each system call represents a different way to enter the kernel and request services.

Now imagine installing an extremely strict gatekeeper at every entrance to this building. This gatekeeper has been given a detailed list of who is allowed through which doors, under what circumstances, and with what identification. The gatekeeper doesn't just check identity—they inspect every detail of each access request: which door the visitor wants to use, what they're carrying, where they claim to be going, and what they intend to do once inside.

This is exactly how seccomp-BPF works. The **BPF filter program** acts as the gatekeeper, examining every system call attempt. For each system call, the filter can inspect:

- The **system call number** (which "door" they're trying to use)
- The **arguments** passed to the system call (what they're "carrying")
- The **architecture** of the calling process (their "identification type")
- Other **process characteristics** available in the seccomp context

Based on this inspection, the gatekeeper makes one of several decisions:

- **SECCOMP_RET_ALLOW**: "You're on the approved list—go right in"
- **SECCOMP_RET_ERRNO**: "Access denied—here's an error code explaining why"
- **SECCOMP_RET_KILL**: "You're not supposed to be here at all—security breach!"

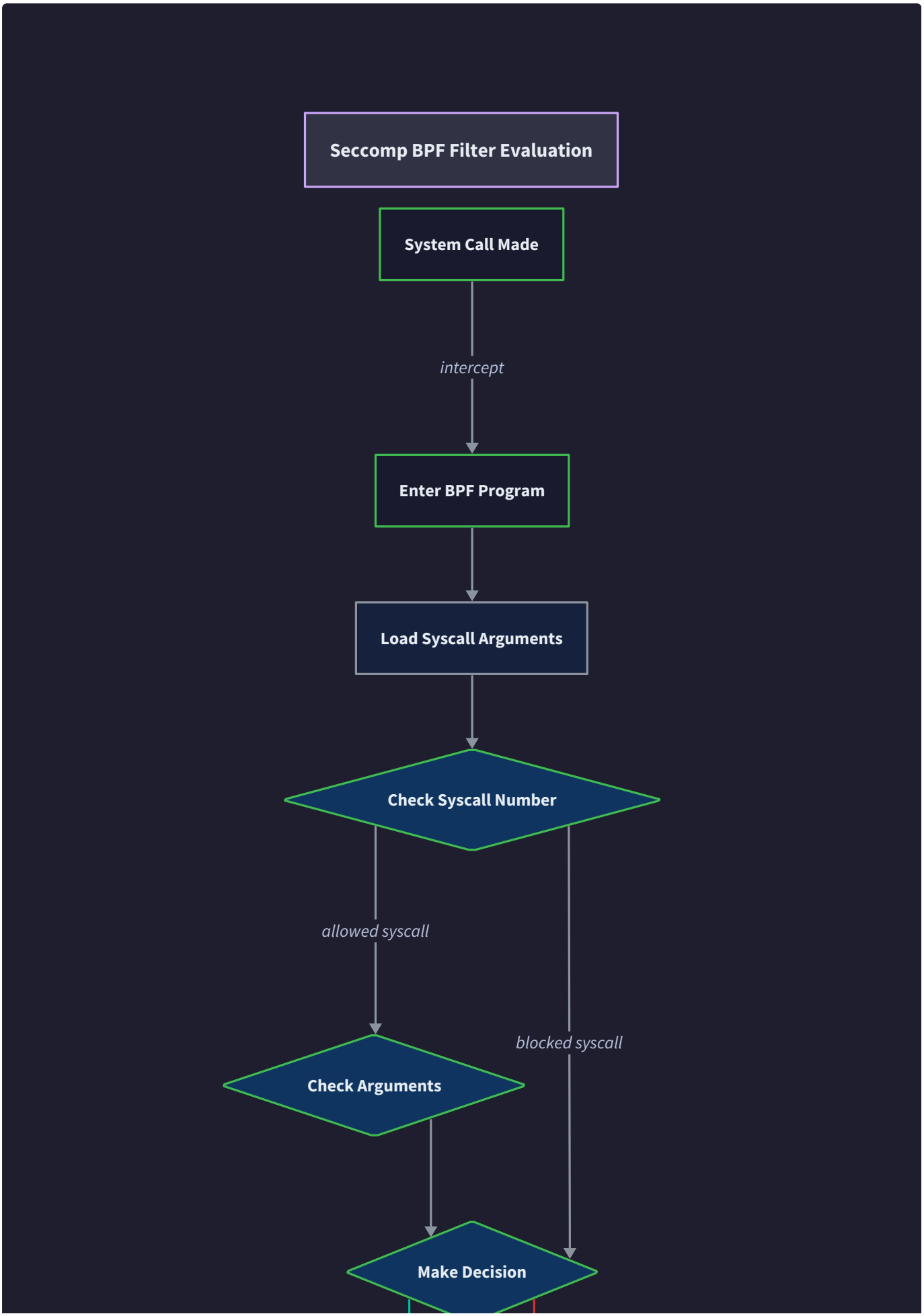
- **SECCOMP_RET_TRAP**: "Hold on—let me call the supervisor to handle this special case"

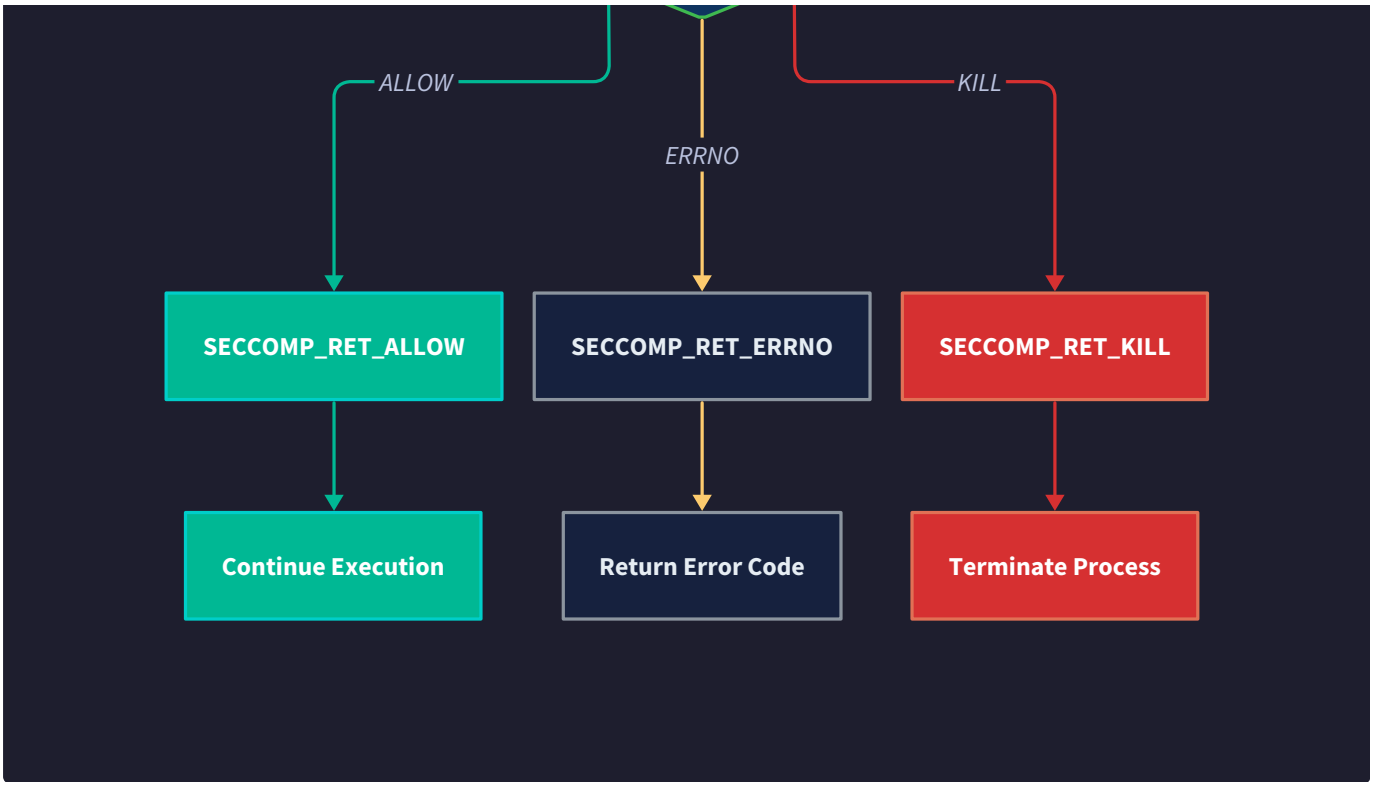
The key insight is that this gatekeeper operates at the kernel level, not in user space. Once a seccomp filter is installed and the `PR_SET_NO_NEW_PRIVS` flag is set, there's no way for the sandboxed process to remove or bypass the filter. Even if an attacker gains complete control of the sandboxed process, they're still constrained by the system call restrictions enforced by the kernel itself.

This creates a powerful security boundary that complements our namespace and filesystem isolation. An attacker might find ways to escape chroot or manipulate filesystem paths, but they still cannot invoke forbidden system calls like `ptrace` (for debugging other processes), `mount` (for manipulating filesystems), or `socket` (for creating network connections) if these are blocked by the seccomp filter.

BPF Filter Program Design

Berkeley Packet Filter programs provide the mechanism for implementing sophisticated system call filtering logic. Originally designed for high-performance network packet filtering, BPF has evolved into a general-purpose kernel programming interface. In the context of seccomp, BPF programs examine system call invocations and return decisions about whether to allow or block them.


Seccomp BPF Filter Evaluation
System Call Made
intercept
Enter BPF Program
Load Syscall Arguments
Check Syscall Number
allowed syscall
blocked syscall
Check Arguments
Make Decision



A **BPF filter program** consists of a sequence of instructions that operate on a virtual machine within the kernel. This virtual machine provides a safe execution environment where filter programs can examine data without risking kernel stability. The BPF instruction set includes arithmetic operations, comparison operations, memory access operations, and control flow operations, but is deliberately constrained to ensure that programs always terminate and cannot cause kernel crashes.

When a system call occurs, the kernel creates a **seccomp data structure** containing information about the system call attempt. This structure includes the system call number, the architecture identifier, and all six system call arguments. The BPF program receives a pointer to this data structure and can examine any fields within it to make filtering decisions.

The fundamental structure of a seccomp BPF program follows this pattern:

1. **Architecture Validation**: First, check that the system call comes from the expected architecture (x86_64, ARM, etc.). Different architectures use different system call numbers for the same operations, so this validation prevents bypass attempts using alternative architectures.
2. **System Call Number Extraction**: Load the system call number from the seccomp data structure and use it as the primary filtering criterion.
3. **Whitelist Comparison**: Compare the system call number against a list of permitted operations. This typically involves a series of conditional jumps that either allow the system call or continue to the next check.
4. **Argument-Level Filtering**: For system calls that require additional scrutiny, examine the arguments to apply more sophisticated filtering rules. For example, an `openat` system call might be allowed only when opening files under specific directory paths.
5. **Default Action**: If no explicit allow rule matches, apply the default action—typically either returning an error or killing the process.

The BPF instruction format uses a compact representation where each instruction contains an operation code, addressing information, and immediate values or jump offsets. Instructions can perform operations like:

- **BPF_LD**: Load data from the seccomp structure

- **BPF_JMP**: Conditional or unconditional jumps for control flow
- **BPF_ALU**: Arithmetic and logical operations on loaded values
- **BPF_RET**: Return a seccomp action code

Building effective BPF programs requires understanding both the instruction set and the data layout of the seccomp structure. The seccomp data provides access to system call information through fixed offsets, allowing BPF programs to load and examine specific fields efficiently.

BPF Program Component	Purpose	Key Considerations
Architecture Check	Prevent bypass via alternative architectures	Must handle x86_64, ARM64, etc. based on target
System Call Lookup	Primary filtering by system call number	Requires architecture-specific syscall mappings
Argument Examination	Fine-grained filtering based on parameters	Limited to simple comparisons and ranges
Return Action	Decision about allowing or blocking call	ALLOW, ERRNO, KILL, or TRAP actions available
Program Validation	Kernel verification of BPF safety	Must prove termination and memory safety

One critical aspect of BPF program design is **performance optimization**. Since every system call triggers BPF program execution, inefficient filters can significantly impact application performance. Well-designed programs place the most common system calls early in the decision tree and use jump tables rather than linear searches when possible. However, premature optimization should be avoided—clarity and correctness take precedence over micro-optimizations in most sandbox scenarios.

The kernel imposes several **safety constraints** on BPF programs that influence their design. Programs must be provably terminating (no infinite loops), cannot access arbitrary kernel memory (only the provided seccomp structure), and cannot perform operations that might compromise kernel stability. The kernel's BPF verifier analyzes every program before allowing its installation, rejecting programs that violate these safety properties.

System Call Whitelist Strategy

Developing an effective system call whitelist requires balancing security restrictions with functional requirements. The challenge lies in identifying the minimal set of system calls necessary for the sandboxed application to operate while blocking all operations that could be used for malicious purposes. This decision significantly impacts both the security posture and the compatibility of the sandbox.

The **whitelist approach** forms the foundation of secure system call filtering. Rather than attempting to blacklist dangerous operations (which inevitably misses newly discovered attack vectors), a whitelist explicitly enumerates every permitted operation. This approach follows the **principle of least privilege**—start with no permissions and grant only the minimum access required for legitimate functionality.

Constructing an effective whitelist begins with **profiling the target application** to understand its system call usage patterns. This involves running representative workloads under `strace` or similar tracing tools to capture the complete set of system calls invoked during normal operation. The profiling process must cover all code paths, including error handling routines and initialization sequences, to avoid missing essential system calls.

However, raw profiling data typically reveals many more system calls than actually necessary. Modern C libraries like glibc implement high-level functions using multiple system calls and may use different implementation strategies depending on kernel version or runtime conditions. For example, file operations might use either `open` or `openat`, and memory allocation might invoke `brk`, `mmap`, or `mmap2` depending on allocation size and system configuration.

The whitelist construction process involves several refinement steps:

1. **Essential System Calls**: Start with fundamental operations required for any meaningful program execution—memory management (`brk`, `mmap`, `munmap`), process lifecycle (`exit`, `exit_group`), and basic I/O (`read`, `write`).
2. **Application-Specific Operations**: Add system calls required by the specific application being sandboxed. Web servers need socket operations, file processors need filesystem access, computational applications need mathematical libraries.
3. **Library Dependencies**: Include system calls required by shared libraries used by the application. This often includes dynamic linking operations (`mmap` for library loading), locale processing (`open` for locale data), and runtime configuration (`access` for configuration file checks).
4. **Error Handling Paths**: Ensure that error conditions don't trigger forbidden system calls. For example, failed memory allocations might trigger alternative allocation strategies that use different system calls.
5. **Platform Variations**: Account for differences between kernel versions, distributions, and architectures. System call numbers and availability vary significantly across platforms.

System Call Category	Examples	Security Considerations	Inclusion Strategy
Memory Management	`brk`, `mmap`, `munmap`, `madvise`	Required for basic operation, low risk	Always include with argument restrictions
File Operations	`open`, `openat`, `read`, `write`, `close`	High risk if unrestricted	Include with path-based filtering when possible
Process Control	`fork`, `clone`, `execve`, `wait4`	Very high risk, enables privilege escalation	Exclude unless absolutely necessary
Network Operations	`socket`, `connect`, `sendto`, `recvfrom`	High risk, enables data exfiltration	Exclude for offline processing, restrict for network services
System Information	`getpid`, `getuid`, `uname`, `clock_gettime`	Generally safe, information disclosure	Include with minimal restrictions
Signal Handling	`sigaction`, `sigprocmask`, `sigreturn`	Required for proper error handling	Include but monitor for abuse

Architecture-specific considerations add significant complexity to whitelist development. Linux supports multiple processor architectures, each with its own system call numbering scheme. The same logical operation (like reading a file) uses different system call numbers on x86_64 versus ARM64. Additionally, some architectures provide multiple system calls for similar operations—for example, x86_64 has both `open` and `openat` system calls, while newer architectures only provide `openat`.

Modern seccomp filtering addresses this by requiring **architecture validation** as the first step in every BPF program. The filter must explicitly check the architecture field in the seccomp data and either handle each supported architecture

separately or reject calls from unexpected architectures entirely. This prevents attackers from bypassing filters by switching to alternative architectures or using compatibility layers.

The **glibc complexity problem** represents one of the most challenging aspects of whitelist development. The GNU C Library implements standard library functions using various system calls, and its implementation strategy can change between versions or based on runtime conditions. For example, `malloc()` might use `brk()` for small allocations and `mmap()` for large ones, but the threshold and exact behavior depend on glibc configuration and heap state.

To address glibc complexity, many sandbox implementations adopt a **learning mode** approach. They initially run the target application with permissive logging to capture all attempted system calls, then iteratively refine the whitelist based on observed behavior. This process must be repeated across different environments and workloads to ensure comprehensive coverage.

> **Key Insight**: The most secure whitelist is often not the most restrictive one. A whitelist that's too restrictive will force developers to add exceptions or disable filtering entirely, ultimately reducing security. The goal is finding the minimal set that reliably supports legitimate application behavior while blocking meaningful attack vectors.

Argument-Level Filtering

While system call number filtering provides a foundation for sandbox security, many attacks exploit legitimate system calls with malicious arguments rather than using forbidden operations entirely. **Argument-level filtering** extends seccomp protection by examining the parameters passed to allowed system calls and applying additional restrictions based on their values. This approach enables much more sophisticated security policies that can distinguish between safe and dangerous uses of the same system call.

Consider the `openat` system call, which opens files and directories. Blocking `openat` entirely would prevent the sandboxed process from reading any files, making most applications non-functional. However, allowing unrestricted `openat` enables attackers to access sensitive system files, escape chroot restrictions, or open device files for privilege escalation. Argument-level filtering resolves this dilemma by examining the path parameter and allowing only access to safe locations.

The **seccomp data structure** provides access to all system call arguments through fixed offsets. Each system call can accept up to six arguments, stored as 64-bit values in the seccomp context. BPF programs can load these arguments and apply various filtering operations:

1. **Exact Value Matching**: Compare arguments against specific allowed values. For example, restrict file descriptors to standard input/output/error (0, 1, 2).
2. **Range Checking**: Verify that numeric arguments fall within acceptable ranges. Memory allocation sizes, signal numbers, and priority values often have safe operational ranges.
3. **Bitmask Validation**: Check that flag arguments contain only permitted bit combinations. Many system calls accept flag parameters that control behavior—filtering can ensure only safe flag combinations are used.
4. **Pointer Validation**: Limited validation of pointer arguments, typically checking for null pointers or obvious invalid values (though comprehensive pointer validation requires kernel cooperation).

However, argument-level filtering faces significant **technical limitations** that constrain its effectiveness. BPF programs operate in kernel space but cannot safely dereference user space pointers. This means that string arguments (like file

paths) cannot be directly examined by the filter program. The BPF program can check whether a pointer is null and validate its numeric value, but cannot inspect the string content it points to.

Filtering Technique	Applicable Argument Types	Security Benefit	Implementation Complexity
Exact Value Match	Integers, flags, file descriptors	High for specific values	Low
Range Validation	Sizes, counts, priorities	Medium for preventing extremes	Low
Bitmask Filtering	Flag combinations, permissions	High for controlling behavior	Medium
Pointer Null Check	All pointer types	Low but prevents crashes	Low
String Content Filtering	File paths, names	Very high but not directly possible	High (requires kernel cooperation)

This limitation has led to the development of **hybrid filtering approaches** that combine BPF-based argument filtering with user space policy enforcement. The seccomp filter handles straightforward numeric validation, while more complex string-based policies are implemented through other mechanisms:

- **User Space Helpers**: The BPF program can return `SECCOMP_RET_TRAP`, which triggers a signal handler in user space. The signal handler can examine string arguments and decide whether to allow the operation.
- **Seccomp Notify Extension**: Modern kernels support `SECCOMP_RET_USER_NOTIF`, which suspends the system call and notifies a supervisor process. The supervisor can examine all arguments, including string content, and decide whether to allow, modify, or deny the operation.
- **Path-Based Restrictions**: Combine seccomp filtering with filesystem isolation. Rather than trying to filter file paths in BPF, use chroot or mount namespaces to ensure that all accessible paths are safe.

Despite these limitations, argument-level filtering provides valuable security benefits for numeric parameters. Consider these practical examples:

File Descriptor Validation: Many attacks involve manipulating file descriptors to access unexpected files. A seccomp filter can restrict file operations to known-safe descriptors. For example, a log processing application might only be allowed to read from file descriptor 3 (the input log) and write to file descriptor 4 (the output file).

Signal Number Restriction: The `kill` system call allows sending signals to processes, but many signals can be used maliciously. Argument filtering can restrict signal sending to safe signals like `SIGTERM` while blocking dangerous ones like `SIGSTOP` or `SIGKILL`.

Memory Protection Flags: The `mmap` system call accepts protection flags that control whether memory regions are readable, writable, or executable. Argument filtering can block requests for executable memory regions, preventing certain code injection attacks.

Socket Domain Restrictions: If network access is required, socket creation can be restricted to specific domains and types. For example, allow only `AF_INET` TCP sockets while blocking `AF_UNIX` domain sockets that might bypass other restrictions.

The implementation of argument-level filtering requires careful attention to **system call ABI details**. Different architectures pass arguments in different ways, and the seccomp data structure reflects the raw argument values as received by the

kernel. This may include pointer values, packed structures, or encoded flags that require specific interpretation logic.

> **Security Principle**: Argument-level filtering is most effective when used to restrict the scope of allowed operations rather than trying to distinguish between safe and unsafe string content. Use it to limit numeric parameters to safe ranges and combine it with other isolation mechanisms for comprehensive protection.

Architecture Decisions

The design of an effective seccomp filtering system involves several critical decisions that significantly impact both security effectiveness and implementation complexity. These architectural choices must balance security goals with practical considerations like performance, maintainability, and compatibility across different environments.

> **Decision: Whitelist vs Blacklist Filtering Strategy**
>
> - **Context**: System call filtering can either explicitly allow safe operations (whitelist) or explicitly block dangerous operations (blacklist). This fundamental choice affects the entire security model and determines how new system calls and attack vectors are handled.
> - **Options Considered**:
>   1. **Blacklist Approach**: Block known dangerous system calls, allow everything else
>   2. **Whitelist Approach**: Allow only known safe system calls, block everything else
>   3. **Hybrid Approach**: Use whitelists for core functionality, blacklists for specific dangerous operations
> - **Decision**: Implement pure whitelist filtering with comprehensive application profiling
> - **Rationale**: Blacklist approaches inevitably miss new attack vectors and fail-open when encountering unknown system calls. Whitelists provide stronger security guarantees by defaulting to denial and requiring explicit justification for each permitted operation. While more difficult to implement correctly, whitelists offer better long-term security properties.
> - **Consequences**: Requires extensive profiling and testing to identify all necessary system calls. May break applications that use uncommon system calls or evolve over time. However, provides strong security guarantees and fails-safe when encountering unexpected operations.

Filtering Strategy	Security Strength	Implementation Effort	Maintenance Burden	Compatibility Risk
Blacklist Only	Low - misses new attacks	Low	High - requires tracking new threats	Low
Whitelist Only	High - fails safe	High	Medium - requires app profiling	High
Hybrid Model	Medium - complex interactions	Very High	Very High - dual complexity	Medium

Decision: BPF Program Compilation Strategy

- **Context**: BPF filter programs can be written as raw instruction arrays, generated from higher-level descriptions, or compiled from domain-specific languages. The compilation strategy affects both development productivity and runtime performance.
- **Options Considered**:
  1. **Hand-Written BPF Instructions**: Directly specify BPF instruction arrays in C
  2. **Macro-Based Generation**: Use C macros to generate instruction sequences
  3. **Runtime Compilation**: Build BPF programs from configuration at runtime
- **Decision**: Use macro-based generation with compile-time optimization
- **Rationale**: Hand-written instructions are error-prone and unmaintainable. Runtime compilation adds complexity and potential failure modes. Macro-based generation provides good balance of control and maintainability while ensuring filters are validated at compile time.
- **Consequences**: Requires learning BPF instruction patterns and macro systems. However, produces reliable, efficient filters that can be validated during development rather than deployment.

Decision: Multi-Architecture Support Strategy

- **Context**: Linux runs on multiple processor architectures with different system call numbering schemes. The sandbox must either support multiple architectures or restrict operation to specific platforms.
- **Options Considered**:
  1. **Single Architecture**: Support only x86_64, reject other architectures
  2. **Multi-Architecture**: Support x86_64, ARM64, and other common architectures
  3. **Runtime Detection**: Automatically adapt to detected architecture
- **Decision**: Support x86_64 primarily with explicit architecture validation, framework for adding others
- **Rationale**: Multi-architecture support significantly increases complexity and testing requirements. Most deployment environments use x86_64. However, the architecture must be explicitly validated to prevent bypass attacks using alternative architectures or compatibility layers.
- **Consequences**: Limits deployment flexibility but reduces complexity and testing burden. Architecture validation prevents certain bypass techniques but requires updates when adding platform support.

The **filter installation and lifecycle management** represents another crucial architectural decision. Seccomp filters are inherited across fork operations but cannot be removed once installed. This creates constraints around when and how filters are applied within the sandbox creation sequence.

The optimal filter installation strategy coordinates with other sandbox components:

1. **Pre-Namespace Installation**: Install seccomp filters before creating namespaces to ensure filtering applies to namespace creation operations themselves. However, this prevents the filter from utilizing namespace-based path restrictions.
2. **Post-Filesystem Installation**: Install filters after filesystem isolation is complete, allowing argument-level filtering to assume chroot restrictions are already active. This simplifies filter logic but allows unrestricted system calls during sandbox setup.

3. **Staged Installation**: Use multiple filter installations at different stages, with progressively restrictive policies. Initial filters allow setup operations, final filters enforce strict runtime restrictions.

The sandbox implementation adopts a **staged installation** approach that balances security and functionality. During the sandbox creation phase, a permissive filter allows necessary setup operations while blocking obviously dangerous calls like `ptrace` and `mount`. After filesystem isolation and other setup is complete, a strict runtime filter replaces the permissive one.

Installation Stage	Allowed Operations	Security Focus	Implementation Notes
Initial Setup	Namespace creation, filesystem operations, resource setup	Prevent obvious attacks during setup	Permissive but blocks dangerous operations
Transition Phase	Final setup operations, privilege dropping	Prepare for strict filtering	Brief window with intermediate restrictions
Runtime Operation	Application-specific whitelist only	Maximum restriction	Permanent strict filtering

Performance optimization considerations influence several architectural decisions. While BPF programs execute very efficiently, system call filtering still adds overhead to every kernel transition. The filter design must minimize this impact through several strategies:

- **Early Exit Optimization**: Place the most frequently used system calls early in the filter decision tree to minimize instruction execution for common operations.
- **Jump Table Organization**: Use structured jump patterns rather than linear searches when checking system call numbers against large whitelists.
- **Argument Validation Ordering**: Perform expensive argument validation only after confirming the system call number is allowed and likely to be used.

However, premature optimization should be avoided. The primary goals are correctness and security—performance optimization should only be applied to filters that demonstrably impact application performance under realistic workloads.

The **error handling strategy** for seccomp violations significantly affects both security and debuggability. When a sandboxed process attempts a forbidden system call, the filter can respond in several ways:

- **SECCOMP_RET_KILL**: Immediately terminate the process with SIGKILL
- **SECCOMP_RET_ERRNO**: Return a specific error code to the caller
- **SECCOMP_RET_TRAP**: Trigger a signal handler for custom processing

The choice between these responses involves trade-offs between security and operational considerations. Killing processes provides the strongest security guarantee but makes debugging difficult and may cause data loss. Returning errors allows graceful handling but may enable attackers to probe the filter and adapt their approach.

The implementation uses a **differentiated response strategy** based on the severity of the violation. Attempts to use obviously malicious system calls like `ptrace` result in immediate process termination, while attempts to access forbidden files return appropriate error codes. This approach provides strong protection against serious attacks while maintaining reasonable debuggability for legitimate application issues.

Common Pitfalls

Implementing effective seccomp filtering involves numerous subtle technical challenges that frequently trip up developers, even those experienced with Linux systems programming. Understanding these common mistakes helps avoid security vulnerabilities and implementation problems that could compromise the entire sandbox.

⚠️ Pitfall: Forgetting PR_SET_NO_NEW_PRIVS Before Filter Installation

One of the most critical mistakes is installing seccomp filters without first setting the `PR_SET_NO_NEW_PRIVS` flag. This flag prevents the process from gaining additional privileges through setuid binaries, file capabilities, or other privilege escalation mechanisms. The kernel requires this flag to be set before allowing unprivileged processes to install seccomp filters.

Without `PR_SET_NO_NEW_PRIVS`, seccomp filter installation will fail with `EACCES` even when running as root, because the kernel cannot guarantee that the filter will remain effective if the process later gains additional privileges. This requirement ensures that seccomp filtering cannot be bypassed through privilege escalation.

The correct implementation sequence must call `prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)` before any `prctl(PR_SET_SECCOMP, SECCOMP_MODE_FILTER, &filter)` operation. This flag setting is irreversible and inherited by child processes, ensuring that the security restriction cannot be removed later.

⚠️ Pitfall: Architecture-Specific System Call Number Confusion

System call numbers vary significantly between processor architectures, and using the wrong numbers can create massive security holes. For example, `__NR_open` has different values on x86_64 versus ARM64, and some system calls exist on one architecture but not others. A filter that works correctly on x86_64 might inadvertently allow dangerous operations on ARM64 if it uses wrong system call numbers.

Additionally, some 64-bit architectures provide compatibility layers for 32-bit applications, which use entirely different system call numbering schemes. An attacker might attempt to bypass filtering by using 32-bit system call interfaces even on 64-bit systems.

The solution requires explicit architecture validation as the first step in every BPF program. The filter must check the `arch` field in the seccomp data and either handle each supported architecture correctly or reject calls from unexpected architectures. Never assume that system call numbers are portable between architectures.

⚠️ Pitfall: Missing Essential glibc System Calls

Modern C library implementations use many more system calls than application developers typically realize. Functions like `malloc()`, `printf()`, and even simple arithmetic operations may trigger unexpected system calls for memory management, locale processing, or dynamic linking. Missing any of these essential calls can cause applications to crash or behave unpredictably.

The most problematic cases involve system calls that are used only in specific circumstances—error handling paths, memory pressure situations, or particular input patterns. An application might work fine during normal testing but fail in production when these edge cases are encountered.

Comprehensive system call profiling is essential, but it must cover all possible execution paths. Use tools like `strace -f` to trace all system calls during extensive testing, including error conditions, resource exhaustion scenarios, and various input types. Additionally, understand that different glibc versions may use different implementation strategies, requiring testing across target environments.

⚠️ Pitfall: Incorrect BPF Program Termination and Validation

BPF programs must be provably terminating—the kernel's BPF verifier will reject programs that might contain infinite loops or unbounded recursion. However, developers often create inadvertent infinite loops through incorrect jump logic or fail to handle all possible code paths properly.

Common BPF validation failures include:

- **Unreachable Code**: Instructions that cannot be reached through any execution path
- **Uninitialized Register Access**: Attempting to use registers that might not contain valid values
- **Infinite Loop Potential**: Jump patterns that could theoretically loop forever
- **Invalid Memory Access**: Attempting to access data outside the provided seccomp structure

The BPF verifier's error messages can be cryptic and difficult to debug. The best approach is to build BPF programs incrementally, testing each addition with the kernel verifier. Use systematic patterns for jump logic and ensure every execution path reaches a valid return instruction.

⚠️ Pitfall: Inadequate Testing of Filter Edge Cases

Seccomp filters often work correctly for normal application behavior but fail when applications encounter unusual conditions or use less common system call patterns. These edge cases can create security vulnerabilities or operational failures that only manifest in production environments.

Critical edge cases to test include:

- **Signal Handling**: Ensure signal delivery and handling works correctly with filtering active
- **Thread Creation**: Verify that threading libraries can function with the restricted system call set
- **Memory Pressure**: Test behavior when system memory is low and allocation strategies change
- **File Descriptor Exhaustion**: Check that applications handle file descriptor limits gracefully
- **Network Timeouts**: For network applications, verify timeout and error handling works correctly

The testing strategy should include both positive tests (verifying allowed operations work) and negative tests (confirming forbidden operations are blocked). Negative tests are particularly important because they verify the security properties of the filter.

⚠️ Pitfall: Filter Bypass Through Alternative System Call Interfaces

Linux provides multiple ways to accomplish many operations, and attackers may attempt to bypass filters by using alternative system call interfaces. For example, file operations can be performed using `open`, `openat`, `creat`, or even `memfd_create` depending on the specific requirements.

Similarly, memory mapping can be accomplished through `mmap`, `mmap2`, or `mremap`, and process creation might use `fork`, `vfork`, `clone`, or `clone3`. A filter that blocks `open` but allows `openat` might be trivially bypassed.

The solution requires understanding the complete family of system calls related to each operation and either allowing all safe variants or blocking all potentially dangerous ones. This often involves studying the kernel source code and glibc implementation to understand which system calls are actually used in different scenarios.

⚠️ Pitfall: Argument Validation Complexity and Limitations

Argument-level filtering appears more powerful than it actually is due to fundamental limitations in what BPF programs can safely examine. Developers often attempt to implement sophisticated string validation or complex data structure inspection that cannot be safely performed in kernel space.

Remember that BPF programs cannot dereference user space pointers, cannot call kernel functions, and have very limited stack space. Attempting to validate file paths, examine structure contents, or perform complex calculations will either fail BPF verification or create security vulnerabilities.

Focus argument validation on simple numeric checks—ranges, exact values, and bitmasks. Use other isolation mechanisms like filesystem namespaces and chroot for complex policy enforcement rather than trying to implement everything in BPF.

⚠️ Pitfall: Filter Installation Timing and Inheritance

The timing of seccomp filter installation relative to other sandbox setup operations critically affects both security and functionality. Installing filters too early can prevent necessary sandbox setup operations, while installing them too late allows a window where dangerous operations are permitted.

Additionally, seccomp filters are inherited by child processes but cannot be removed once installed. This inheritance behavior affects how the sandbox manages multi-process applications and may require careful coordination between parent and child processes.

Plan the filter installation sequence carefully, potentially using multiple filter stages with progressively stricter policies. Document the security properties of each stage and ensure that the transition between stages cannot be exploited.

Implementation Guidance

Building a robust seccomp filtering system requires careful attention to BPF program construction, system call profiling, and proper integration with the overall sandbox architecture. This implementation guidance provides concrete code patterns and debugging strategies for common seccomp development challenges.

Technology Recommendations Table:

Component	Simple Option	Advanced Option
BPF Program Generation	Hand-written instruction arrays with helper macros	libseccomp library for high-level policy specification
System Call Profiling	Manual strace analysis with grep/awk scripts	Automated profiling with custom ptrace-based tracer
Architecture Support	Hardcoded x86_64 with explicit validation	Runtime architecture detection with per-arch filters
Filter Testing	Shell scripts with specific test binaries	Comprehensive test framework with fuzzing capabilities

Recommended File Structure:

The seccomp filtering component integrates with the existing sandbox architecture while maintaining clear separation of concerns:

```
sandbox/
  include/
    seccomp_filter.h         ← public interface definitions
    bpf_helpers.h           ← BPF instruction generation macros
  src/
    seccomp/
      seccomp_filter.c      ← main filtering implementation
      bpf_programs.c        ← BPF program generation and compilation
      syscall_whitelist.c   ← system call whitelist definitions
      filter_install.c      ← filter installation and lifecycle management
      seccomp_filter_test.c ← unit tests for filtering logic
  tools/
    profile_syscalls.c      ← system call profiling utility
    test_filter.c           ← filter validation and testing tool
  config/
    base_whitelist.h        ← default system call whitelist
    arch_syscalls.h         ← architecture-specific system call numbers
```

Infrastructure Starter Code:

The following BPF helper infrastructure provides the foundation for building seccomp filters without requiring deep knowledge of BPF instruction formats:

```c
// bpf_helpers.h - BPF instruction generation macros

#ifndef BPF_HELPERS_H

#define BPF_HELPERS_H


#include <linux/seccomp.h>

#include <linux/filter.h>

#include <linux/audit.h>

#include <sys/syscall.h>


// BPF instruction construction macros

#define BPF_STMT(code, k) { (unsigned short)(code), 0, 0, k }

#define BPF_JUMP(code, k, jt, jf) { (unsigned short)(code), jt, jf, k }


// Load operations for seccomp data

#define LOAD_SYSCALL_NR \

    BPF_STMT(BPF_LD | BPF_W | BPF_ABS, offsetof(struct seccomp_data, nr))


#define LOAD_ARCH \

    BPF_STMT(BPF_LD | BPF_W | BPF_ABS, offsetof(struct seccomp_data, arch))


#define LOAD_ARG(n) \

    BPF_STMT(BPF_LD | BPF_W | BPF_ABS, offsetof(struct seccomp_data, args[n]))


// Comparison and jump operations

#define SYSCALL_ALLOW(syscall_nr) \

    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_##syscall_nr, 0, 1), \

    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW)


#define SYSCALL_ERRNO(syscall_nr, errno_val) \

    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, __NR_##syscall_nr, 0, 1), \

    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ERRNO | (errno_val & 0xFFFF))


#define ARCH_CHECK(arch_val) \

    BPF_JUMP(BPF_JMP | BPF_JEQ | BPF_K, arch_val, 1, 0), \
```

```
    BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL)

// Return actions
#define ALLOW_SYSCALL BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ALLOW)
#define KILL_PROCESS BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_KILL)
#define ERRNO_RETURN(err) BPF_STMT(BPF_RET | BPF_K, SECCOMP_RET_ERRNO | (err))

// Architecture definitions for common platforms
#if defined(__x86_64__)
#define CURRENT_ARCH AUDIT_ARCH_X86_64
#elif defined(__aarch64__)
#define CURRENT_ARCH AUDIT_ARCH_AARCH64
#elif defined(__arm__)
#define CURRENT_ARCH AUDIT_ARCH_ARM
#else
#error "Unsupported architecture for seccomp filtering"
#endif

#endif // BPF_HELPERS_H
```

Core Logic Skeleton Code:

The main seccomp filtering implementation provides the interface for creating and installing BPF programs based on system call whitelists:

```c
// seccomp_filter.c - Main seccomp filtering implementation

#include "seccomp_filter.h"

#include "bpf_helpers.h"

#include <sys/prctl.h>

#include <errno.h>


typedef struct {

    const char* name;

    int number;

    int arch_specific;

} syscall_entry_t;


// Core filtering interface

int install_seccomp_filter(sandbox_config_t* config) {

    // TODO 1: Validate that PR_SET_NO_NEW_PRIVS is set or can be set

    // Check current process state and set flag if not already set



    // TODO 2: Build BPF program based on allowed system calls in config

    // Use config->allowed_syscalls to generate instruction sequence



    // TODO 3: Validate BPF program structure and instruction count

    // Ensure program is well-formed and within kernel limits



    // TODO 4: Install BPF program using prctl(PR_SET_SECCOMP)

    // Handle installation errors and provide meaningful diagnostics



    // TODO 5: Verify filter installation by testing known-allowed operation

    // Perform simple system call to confirm filter is active and working



    return SANDBOX_SUCCESS;

}
```

```
int build_bpf_program(char** allowed_syscalls, struct sock_filter** filter_out, int* filter_len) {
    // TODO 1: Calculate required instruction count based on whitelist size
    // Each allowed system call requires jump and allow instructions

    // TODO 2: Allocate instruction array with proper size
    // Include space for architecture check, syscall checks, and default action

    // TODO 3: Generate architecture validation instructions
    // Add LOAD_ARCH and ARCH_CHECK instructions at program start

    // TODO 4: Generate system call whitelist checking instructions
    // For each allowed syscall, add LOAD_SYSCALL_NR and SYSCALL_ALLOW

    // TODO 5: Add default deny action at program end
    // Ensure any non-whitelisted system call results in appropriate action

    return SANDBOX_SUCCESS;
}

int validate_syscall_whitelist(char** allowed_syscalls) {
    // TODO 1: Check that whitelist contains essential system calls
    // Verify read, write, exit_group are included for basic functionality

    // TODO 2: Validate system call names against known syscall table
    // Ensure all names are valid and map to actual system call numbers

    // TODO 3: Check for architecture-specific syscall compatibility
    // Warn about system calls that don't exist on current architecture

    // TODO 4: Identify potentially dangerous allowed system calls
```

```
    // Log warnings for syscalls like execve, ptrace, mount if allowed


    return SANDBOX_SUCCESS;
}

int test_seccomp_filter() {
    // TODO 1: Attempt a known-safe system call (like getpid)
    // Verify that allowed operations work correctly after filter installation

    // TODO 2: Check that filter blocks obviously dangerous operations
    // This is tricky - cannot actually test forbidden calls without risks

    // TODO 3: Verify error codes returned for blocked operations
    // Ensure SECCOMP_RET_ERRNO returns expected error values


    return SANDBOX_SUCCESS;
}
```

System Call Profiling Utility:

A dedicated profiling tool helps identify the complete set of system calls required by target applications:

```c
// tools/profile_syscalls.c - System call profiling utility

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/ptrace.h>

#include <sys/wait.h>

#include <sys/user.h>


int main(int argc, char* argv[]) {

    // TODO 1: Fork child process to run target application

    // Child process will be traced to capture all system calls



    // TODO 2: Attach ptrace to child and configure syscall tracing

    // Use PTRACE_SETOPTIONS with PTRACE_O_TRACESYSGOOD



    // TODO 3: Main tracing loop - wait for syscall events

    // Handle both syscall entry and exit to capture full behavior



    // TODO 4: Extract syscall number from registers on each call

    // Architecture-specific register access for syscall identification



    // TODO 5: Maintain set of observed syscalls and generate whitelist

    // Output final whitelist in format suitable for seccomp filter



    return 0;

}
```

Language-Specific Hints:

BPF Instruction Management: Use static arrays for BPF instructions when the filter is known at compile time. For dynamic filters, allocate instruction arrays with `malloc()` and ensure proper cleanup. The kernel copies the filter during installation, so temporary arrays can be freed afterward.

System Call Number Portability: Always use `__NR_syscallname` macros from `<sys/syscall.h>` rather than hardcoded numbers. These macros provide architecture-specific values and are the only reliable way to ensure portability.

Error Handling for prctl(): The `prctl()` system call returns specific error codes that indicate different failure modes. `EINVAL` suggests invalid arguments or BPF program format, `EACCES` indicates missing `PR_SET_NO_NEW_PRIVS`, and `EFAULT` suggests memory access problems with the filter array.

BPF Program Size Limits: The kernel imposes limits on BPF program size and complexity. Keep programs under 4096 instructions and minimize jump complexity. If whitelist size approaches limits, consider splitting into multiple filters or using more efficient instruction patterns.

Milestone Checkpoint:

After implementing basic seccomp filtering, verify the implementation with these tests:

1. **Basic Installation Test**: Create a minimal filter that allows only `read`, `write`, and `exit_group`. Install the filter and verify a simple program can print a message and exit normally.
2. **Whitelist Enforcement Test**: Create a test program that attempts both allowed and forbidden system calls. Verify that allowed calls succeed and forbidden calls either return appropriate errors or terminate the process.
3. **Architecture Validation Test**: Verify that the BPF program correctly validates architecture and rejects calls from unexpected architectures (this is difficult to test directly but can be verified through BPF program inspection).
4. **Application Compatibility Test**: Run a realistic application (like a simple file processing tool) with comprehensive system call profiling to ensure the whitelist covers all necessary operations.

Expected behavior: Applications should run normally with appropriate whitelists but fail quickly and safely when attempting forbidden operations. The filter should be transparent for normal operation while providing strong security boundaries.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
prctl() fails with EACCES	PR_SET_NO_NEW_PRIVS not set	Check process flags in /proc/self/status	Call prctl(PR_SET_NO_NEW_PRIVS) before filter installation
Application crashes immediately	Missing essential system calls	Run with strace to identify blocked calls	Add required syscalls to whitelist
BPF program installation fails	Invalid instruction format	Use kernel BPF verifier error messages	Review instruction generation and program structure
Filter allows forbidden operations	Incorrect syscall numbers or logic	Test with known forbidden calls	Verify architecture-specific syscall numbers

Resource Limit Component

> **Milestone(s):** This section corresponds to Milestone 4 (Resource Limits with Cgroups) and builds upon the isolation mechanisms from Milestones 1-3, adding resource control to prevent resource exhaustion attacks

The resource limit component provides the fourth layer of our defense-in-depth strategy, addressing resource-based attacks that can occur even within isolated namespaces and restricted system calls. While namespace isolation, filesystem restrictions, and seccomp filtering control **what** a sandboxed process can access, cgroups control **how much** of each system resource the process can consume. This distinction is crucial because a compromised process might attempt to exhaust system resources as a denial-of-service attack or to trigger out-of-memory conditions that could lead to privilege escalation opportunities.

Mental Model: The Resource Allocation Department

Think of cgroups as a **bureaucratic resource allocation department** within the Linux kernel. Just as a corporate IT department tracks and limits how much computing resources, storage quota, and network bandwidth each department can use, cgroups create accounting ledgers and enforcement policies for system resources on a per-process-group basis.

In this analogy, the **cgroup hierarchy** represents the organizational chart of departments and sub-departments. Each department (cgroup) has its own resource budget allocated by the parent department. The **cgroup controllers** are like specialized accounting divisions - the memory controller tracks RAM usage like a facilities manager tracking office space, the CPU controller manages processor time like a project manager allocating developer hours, and the I/O controller monitors disk bandwidth like a network administrator managing traffic quotas.

When a sandboxed process attempts to consume resources, it's like an employee making a resource request. The appropriate controller checks the current usage against the allocated budget. If the request would exceed the limit, the controller acts like a strict department head - it either denies the request (returning ENOMEM for memory), throttles the resource allocation (CPU time slicing), or queues the request until resources become available (I/O bandwidth limiting).

This mental model helps explain why cgroups are hierarchical (departments can have sub-departments with smaller budgets), why limits are enforced at allocation time rather than after the fact (good accounting prevents overspending), and why different controllers operate independently (different types of resources require different accounting methods).

The key insight is that cgroups provide **proactive resource governance** rather than reactive monitoring. They prevent resource exhaustion before it happens, rather than detecting it after the system is already under stress.

Cgroup Controllers and Limits

The cgroups subsystem provides several **controllers**, each responsible for managing a specific type of system resource. Our sandbox leverages four primary controllers that address the most common resource exhaustion attack vectors: memory consumption, CPU utilization, process proliferation, and I/O bandwidth abuse.

Controller	Resource Managed	Primary Limits	Attack Vector Prevented
memory	RAM and swap usage	memory.limit_in_bytes, memory.soft_limit_in_bytes	Memory exhaustion, OOM conditions
cpu	Processor time allocation	cpu.cfs_quota_us, cpu.cfs_period_us	CPU starvation, busy loops
pids	Process and thread count	pids.max	Fork bombs, process exhaustion
blkio	Block device I/O	blkio.throttle.read_bps_device, blkio.throttle.write_bps_device	I/O flooding, disk thrashing

The **memory controller** operates by tracking page allocations and maintaining running totals of resident memory usage. When a sandboxed process requests memory allocation (through malloc, mmap, or stack growth), the kernel consults the memory cgroup's current usage against its configured limit. If the allocation would exceed the limit, the kernel triggers the Out of Memory (OOM) killer specifically within that cgroup, terminating processes within the sandbox without affecting the host system.

Memory limits are configured through two primary mechanisms: **hard limits** that trigger immediate OOM killing when exceeded, and **soft limits** that allow temporary exceedances but prioritize reclaiming memory from processes that exceed their soft allocation. For sandbox environments, hard limits provide more predictable behavior since they create deterministic boundaries that cannot be exceeded under any circumstances.

The **CPU controller** uses the Completely Fair Scheduler (CFS) bandwidth control mechanism to allocate processor time. Rather than setting absolute CPU percentages, it configures quotas and periods - for example, a quota of 50,000 microseconds within a period of 100,000 microseconds effectively limits the cgroup to 50% of one CPU core. This approach scales properly across systems with different numbers of CPU cores and provides more granular control than simple percentage-based limiting.

CPU throttling operates by tracking the cumulative CPU time consumed by all processes within the cgroup during each scheduling period. When the quota is exhausted, all processes in the cgroup are removed from the scheduler's run queue until the next period begins. This creates predictable CPU allocation that prevents CPU starvation attacks while maintaining fair scheduling for legitimate workloads.

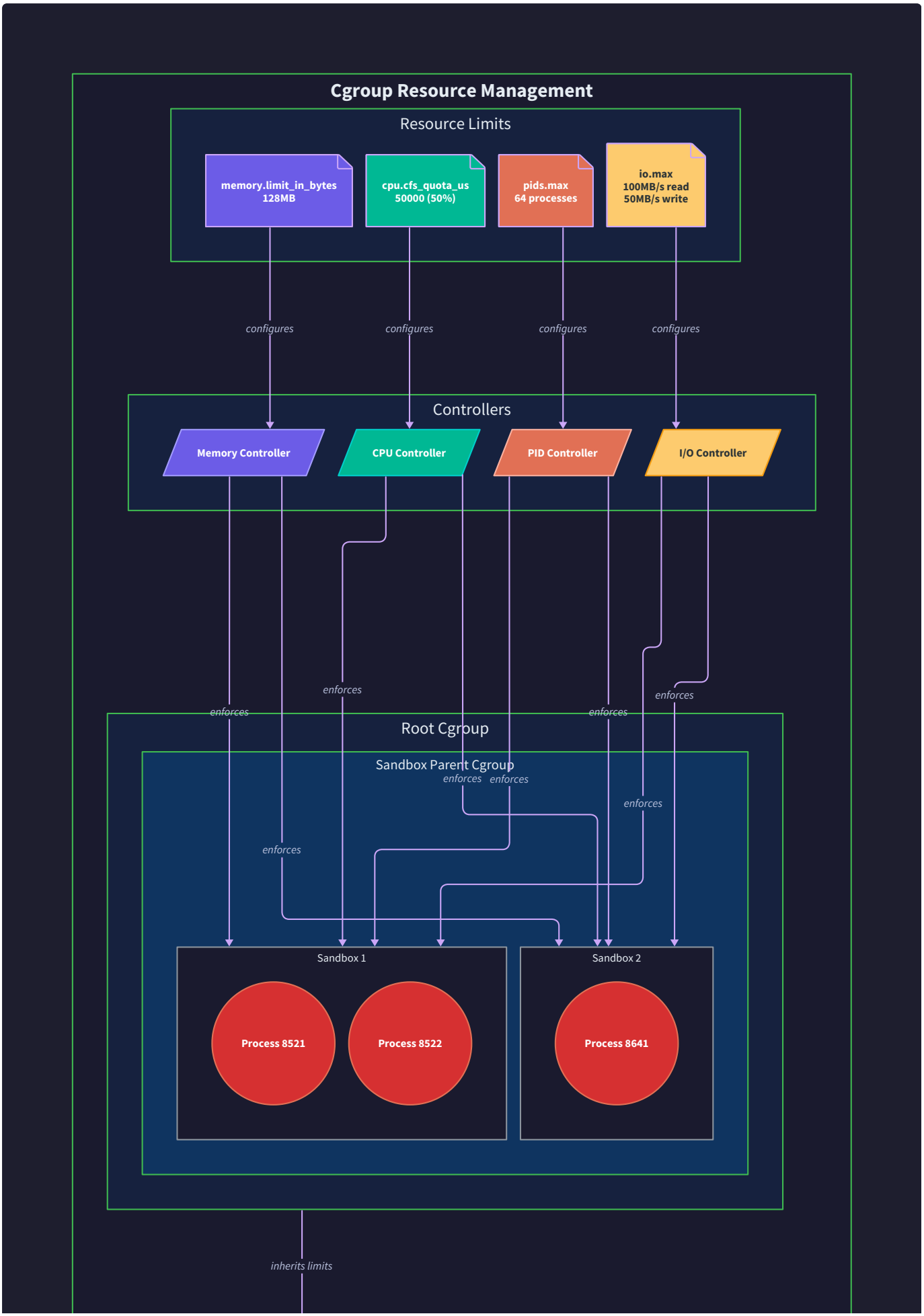
The **PID controller** provides a simple but critical safeguard against fork bomb attacks. By setting `pids.max` to a reasonable value (typically 32-128 processes for most sandbox workloads), it prevents malicious code from creating thousands of processes that could exhaust the system's process table or memory resources. The PID controller tracks both processes and threads, preventing attackers from circumventing limits by using threads instead of processes.

The **block I/O controller** manages disk bandwidth by setting per-device read and write limits measured in bytes per second. This prevents sandboxed processes from saturating disk I/O channels with large read or write operations that could impact host system performance. I/O throttling is implemented through request queuing - when a process exceeds its bandwidth allocation, additional I/O requests are queued and processed at the configured rate limit.

Limit Type	Configuration Parameter	Typical Sandbox Value	Enforcement Behavior
Memory Hard Limit	memory.limit_in_bytes	64MB - 256MB	OOM kill on exceed
CPU Quota	cpu.cfs_quota_us / cpu.cfs_period_us	10% - 50% of one core	Throttle scheduling
Process Count	pids.max	32 - 128 processes	EAGAIN on fork()
Read Bandwidth	blkio.throttle.read_bps_device	10MB/s - 100MB/s	Queue excess requests
Write Bandwidth	blkio.throttle.write_bps_device	5MB/s - 50MB/s	Queue excess requests

Cgroup Hierarchy Management

Cgroups organize resources through a **hierarchical filesystem interface** mounted at `/sys/fs/cgroup`. This hierarchy reflects resource inheritance relationships - child cgroups inherit resource limits from their parents and can further subdivide allocated resources among their own children. Understanding this hierarchy is crucial for properly isolating sandbox instances from each other and from the host system.


Cgroup Resource Management
Resource Limits
memory.limit_in_bytes
128MB
cpu.cfs_quota_us
50000 (50%)
pids.max
64 processes
io.max
100MB/s read
50MB/s write
configures
configures
configures
configures
Controllers
Memory Controller
CPU Controller
PID Controller
I/O Controller
enforces
enforces
enforces
enforces
enforces
enforces
enforces
enforces
enforces
Root Cgroup
Sandbox Parent Cgroup
Sandbox 1
Process 8521
Process 8522
Sandbox 2
Process 8641
inherits limits

The filesystem interface means that creating and configuring cgroups involves standard filesystem operations: creating directories establishes new cgroups, writing to special files configures limits, and reading from these files monitors current usage. This design provides a uniform interface for cgroup management that integrates well with existing system administration tools and scripts.

For sandbox environments, we typically create a dedicated hierarchy structure that isolates all sandbox instances under a common parent cgroup while providing individual resource allocation for each sandbox instance:

```
/sys/fs/cgroup/
├── sandbox-parent/                  ← Common parent for all sandboxes
│   ├── memory.limit_in_bytes        ← Total memory pool for all sandboxes
│   ├── cpu.cfs_quota_us             ← Total CPU allocation for sandbox system
│   └── instance-<uuid>/             ← Individual sandbox instance
│       ├── memory.limit_in_bytes    ← Per-instance memory limit
│       ├── cpu.cfs_quota_us         ← Per-instance CPU quota
│       ├── pids.max                 ← Per-instance process limit
│       └── cgroup.procs             ← Process membership file
```

This hierarchical structure provides several important benefits. First, it enables **resource isolation between sandbox instances** - one sandbox cannot consume resources allocated to another sandbox, even if both are running on the same host. Second, it provides **aggregate resource control** - the parent cgroup can limit the total resources consumed by the entire sandbox system, preventing it from impacting other host services. Third, it simplifies **cleanup and management** - removing the parent directory recursively cleans up all child cgroups automatically.

Cgroup creation requires careful attention to the **order of operations**. The kernel automatically creates certain control files when a cgroup directory is created, but these files only become writable after the appropriate controllers are enabled in the parent cgroup's `cgroup.subtree_control` file. This means the controller enablement must propagate down the hierarchy before child cgroups can configure their limits.

Process membership in cgroups is managed through the `cgroup.procs` file, which contains the PID of every process currently assigned to that cgroup. Writing a PID to this file moves the process into the cgroup, automatically removing it from its previous cgroup. This mechanism provides **atomic process migration** that ensures every process belongs to exactly one cgroup for each controller type at any given time.

Hierarchy Level	Purpose	Configuration Responsibilities	Cleanup Requirements
Root Cgroup	System-wide defaults	Enable controllers for descendants	N/A (persistent)
Sandbox Parent	Aggregate sandbox limits	Total resource pool, controller enablement	Remove after all instances
Instance Cgroup	Per-sandbox limits	Individual resource quotas, process membership	Remove after process termination

The hierarchical nature also enables **resource limit inheritance and override patterns**. A parent cgroup can set conservative default limits that apply to all children, while individual child cgroups can set more restrictive limits based on their specific workload requirements. However, child cgroups cannot exceed the resource allocations of their parents - the kernel enforces this constraint automatically.

Limit Enforcement and Monitoring

Cgroup limit enforcement operates through **kernel-level resource accounting** that intercepts resource allocation requests at the point they occur. This proactive enforcement model prevents resource exhaustion attacks by denying excessive resource requests before they can impact system stability, rather than attempting to reclaim resources after problems have already occurred.

Memory limit enforcement occurs during page allocation operations within the kernel memory management subsystem. When a process requests additional memory through system calls like `brk()`, `mmap()`, or during page fault handling, the kernel checks the requesting process's memory cgroup usage against its configured limits. If the allocation would exceed the hard limit, the kernel immediately returns ENOMEM to the requesting process and may invoke the OOM killer to terminate processes within the cgroup to free memory.

The OOM killer behavior within cgroups is **scope-limited** to the violating cgroup, meaning it only considers processes within the same cgroup as candidates for termination. This provides important isolation guarantees - a memory-exhausting sandbox cannot trigger OOM killing of host processes or processes in other sandboxes. The kernel selects OOM victims based on a scoring algorithm that considers memory usage, process age, and OOM adjustment values.

CPU limit enforcement integrates with the kernel's process scheduler to implement **bandwidth throttling**. The CFS scheduler tracks the cumulative CPU time consumed by all processes within a cgroup during each scheduling period (typically 100 milliseconds). When the configured quota is exhausted, all processes in the cgroup are removed from the scheduler's run queues and cannot be scheduled for execution until the next period begins.

This enforcement mechanism provides **guaranteed CPU isolation** between cgroups while maintaining fair scheduling within each cgroup. Processes within a throttled cgroup continue to compete fairly with each other for their allocated CPU time, but cannot consume CPU cycles beyond their quota regardless of system load or availability of idle CPU time.

Process limit enforcement occurs at `fork()` and `clone()` system call sites within the kernel. Before creating a new process or thread, the kernel increments the PID cgroup's process count and checks it against the configured maximum. If the limit would be exceeded, the system call fails with EAGAIN, preventing the process creation. This provides immediate feedback to applications that attempt to create excessive numbers of processes.

I/O bandwidth enforcement operates through the block layer's request queuing mechanism. When processes issue read or write requests that would exceed their configured bandwidth limits, the block I/O scheduler places these requests in a throttling queue and processes them at the configured rate. This creates **smooth I/O rate limiting** that prevents I/O flooding while allowing processes to make progress at their allocated bandwidth.

Resource monitoring capabilities provide real-time visibility into resource consumption patterns and limit violations. Each cgroup controller exposes usage statistics through filesystem interfaces that can be polled programmatically or monitored by system administration tools.

Monitoring File	Information Provided	Usage Pattern	Alert Thresholds
memory.usage_in_bytes	Current memory consumption	Poll every 1-5 seconds	>90% of limit
memory.failcnt	Count of allocation failures	Check after process termination	Any non-zero value
cpu.stat	CPU time and throttling events	Poll every 5-30 seconds	High throttle ratio
pids.current	Current process/thread count	Poll when fork() fails	Approaching pids.max
blkio.throttle.io_service_bytes	I/O bytes transferred	Poll every 10-60 seconds	Sustained limit hitting

Architecture Decisions

The design and implementation of cgroup resource limiting requires several critical architecture decisions that significantly impact security effectiveness, performance characteristics, and operational complexity. These decisions must balance the competing requirements of strong resource isolation, minimal performance overhead, and maintainable system architecture.

Decision: Cgroups v1 vs v2 Selection

- **Context**: Linux systems may support cgroups v1 (legacy), cgroups v2 (unified hierarchy), or both simultaneously. The sandbox system must choose which version to use for resource limiting, with implications for feature availability, performance, and compatibility.
- **Options Considered**:
  1. Cgroups v1 with separate controller hierarchies
  2. Cgroups v2 with unified hierarchy
  3. Hybrid approach detecting available version at runtime
- **Decision**: Use cgroups v2 when available, fall back to v1 for compatibility
- **Rationale**: Cgroups v2 provides simplified management through unified hierarchy, better resource control granularity, improved performance through reduced overhead, and enhanced security through better privilege delegation. However, older systems may only support v1, making fallback necessary for broad compatibility.
- **Consequences**: Enables access to advanced v2 features like enhanced memory control and improved CPU isolation while maintaining compatibility with older systems. Requires implementing dual code paths and version detection logic.

Option	Pros	Cons	Chosen?
Cgroups v1 Only	Universal compatibility, well-documented, stable API	Separate hierarchies complex, limited delegation, performance overhead	No
Cgroups v2 Only	Unified hierarchy, better performance, enhanced features	Limited compatibility, newer feature set	No
Hybrid v2/v1	Best features when available, broad compatibility	Additional complexity, dual maintenance	**Yes**

Decision: Controller Selection Strategy

- **Context**: Cgroups provides numerous controllers for different resource types (memory, CPU, I/O, network, devices, etc.). The sandbox must determine which controllers to enable and configure, balancing security coverage against complexity and performance overhead.
- **Options Considered**:
  1. Enable all available controllers for maximum security
  2. Enable minimal controller set (memory, CPU, PID only)
  3. Configurable controller selection based on workload requirements
- **Decision**: Enable core controllers (memory, CPU, PID, I/O) by default with optional additional controllers
- **Rationale**: Memory, CPU, and PID controllers address the most common resource exhaustion attack vectors with minimal overhead. I/O controller prevents disk flooding attacks. Additional controllers like network or devices can be enabled for specific high-security scenarios but add complexity and overhead for general use cases.
- **Consequences**: Provides strong protection against primary resource exhaustion attacks while maintaining good performance. Allows specialized deployments to enable additional controllers as needed.

Decision: Default Resource Limit Values

- **Context**: The sandbox system must establish default resource limits that provide security without unnecessarily constraining legitimate workloads. These limits affect both security effectiveness and usability, requiring careful balance between restriction and functionality.
- **Options Considered**:
  1. Very restrictive defaults (16MB memory, 5% CPU) prioritizing security
  2. Permissive defaults (1GB memory, 100% CPU) prioritizing compatibility
  3. Moderate defaults (128MB memory, 25% CPU) balancing security and usability
- **Decision**: Moderate defaults with easy configuration override
- **Rationale**: Very restrictive defaults would break many legitimate applications and create poor user experience. Permissive defaults provide insufficient protection against resource exhaustion attacks. Moderate defaults accommodate most legitimate workloads while providing meaningful resource constraint.
- **Consequences**: Enables sandbox system to work well "out of the box" for common use cases while maintaining reasonable security boundaries. Users can tighten limits for high-security scenarios or relax them for resource-intensive applications.

Resource Type	Conservative Default	Moderate Default	Permissive Default	Chosen
Memory Limit	16MB	128MB	1GB	**128MB**
CPU Quota	5% of one core	25% of one core	100% of one core	**25%**
Process Limit	8 processes	32 processes	256 processes	**32**
I/O Read Bandwidth	1MB/s	10MB/s	Unlimited	**10MB/s**
I/O Write Bandwidth	512KB/s	5MB/s	Unlimited	**5MB/s**

Decision: Cgroup Hierarchy Organization

- **Context**: The sandbox system must organize cgroup hierarchies to provide isolation between sandbox instances while enabling efficient management and cleanup. Different organizational patterns have implications for resource sharing, isolation guarantees, and administrative overhead.
- **Options Considered**:
  1. Flat hierarchy with all sandboxes as siblings under root
  2. Two-level hierarchy with sandbox parent containing instance children
  3. Multi-level hierarchy organized by user, application, and instance
- **Decision**: Two-level hierarchy with dedicated sandbox parent cgroup
- **Rationale**: Flat hierarchy provides no aggregate resource control and complicates cleanup. Multi-level hierarchy adds unnecessary complexity for most use cases. Two-level approach provides aggregate resource control, simplifies cleanup through parent removal, and enables resource sharing policies between sandbox instances when desired.
- **Consequences**: Simplifies resource management and cleanup while providing good isolation. Enables aggregate resource policies for the entire sandbox system. May require additional levels for complex multi-tenant scenarios.

The architecture decisions collectively create a resource limiting system that prioritizes **practical security** over theoretical maximum restriction. By choosing moderate defaults, hybrid cgroup version support, and essential controller enablement, the system provides strong resource exhaustion protection while remaining usable for a wide variety of legitimate applications.

Common Pitfalls

Resource limit implementation with cgroups involves several subtle failure modes that can compromise security effectiveness or system stability. Understanding these pitfalls and their prevention strategies is essential for building reliable sandbox systems.

⚠️ Pitfall: Insufficient Permission Management

Cgroup operations require specific privileges that vary between cgroups v1 and v2, and many developers underestimate the permission requirements for cgroup management. Simply running as root is insufficient - the process needs appropriate capabilities and must handle permission delegation correctly for child processes.

In cgroups v1, writing to controller-specific files like `memory.limit_in_bytes` requires root privileges, but the process also needs write access to `cgroup.procs` to add processes to the cgroup. This creates a chicken-and-egg problem where the sandbox launcher needs root privileges to set up limits, but the sandboxed process should run with dropped privileges.

The common mistake is attempting to configure cgroup limits from within the sandboxed process after privileges have been dropped. This fails because the process no longer has sufficient privileges to write to cgroup control files. The correct approach is to configure all cgroup settings from the privileged parent process before spawning the sandboxed child.

Prevention: Configure all cgroup limits and settings from the privileged launcher process before spawning the sandboxed child. Use file descriptor passing or other IPC mechanisms to communicate cgroup paths to child processes that need to monitor their resource usage but not modify limits.

⚠️ Pitfall: Controller Enablement Propagation

Cgroups v2 requires explicit controller enablement in parent cgroups before child cgroups can use those controllers. Many developers forget that controller availability propagates down the hierarchy - if a controller is not enabled in a parent cgroup's `cgroup.subtree_control`, child cgroups cannot use that controller even if it's available at the system level.

This creates confusing error conditions where attempting to write to `memory.max` in a child cgroup fails with "Operation not permitted" even when running as root, because the memory controller was never enabled in the parent cgroup. The error messages are often misleading and don't clearly indicate the controller enablement issue.

Prevention: Implement systematic controller enablement that propagates from the root cgroup down to the target cgroup level. Check `cgroup.controllers` to verify available controllers and write to `cgroup.subtree_control` to enable controllers for child cgroups before creating those children.

⚠️ Pitfall: Race Conditions in Process Migration

Moving processes into cgroups involves writing PIDs to `cgroup.procs`, but this operation is racy - the process might exit between the time you obtain its PID and when you write it to the cgroup file. This is particularly problematic when trying to move short-lived processes or when the target process is already under resource pressure.

Additionally, multi-threaded processes require special handling because individual threads cannot be moved to different cgroups independently. Attempting to write a thread ID (TID) to `cgroup.procs` results in moving the entire process including all its threads.

Prevention: Perform process migration immediately after fork() but before exec() when the child process is still under parent control. Use the `cgroup.threads` interface (cgroups v2) when thread-level granularity is needed, but understand the additional complexity and limitations this introduces.

⚠️ Pitfall: Resource Limit Calculation Errors

Resource limits require careful calculation to account for system architecture differences and kernel accounting overhead. Memory limits specified in bytes may not account for kernel memory overhead, page alignment requirements, or memory used by shared libraries that may be charged to the cgroup.

CPU limits using CFS quotas are specified in microseconds and require understanding the relationship between quota and period values. A common mistake is setting CPU quotas without considering the number of available CPU cores - setting a 100,000 microsecond quota with a 100,000 microsecond period limits the cgroup to 100% of one CPU core, not 100% of system CPU capacity.

Prevention: Always specify memory limits with headroom above the expected application memory usage (typically 20-30% overhead). For CPU limits, explicitly calculate quotas relative to desired percentage of total system CPU capacity. Test resource limits under realistic workload conditions to verify they provide adequate resources for legitimate operations.

⚠️ Pitfall: Incomplete Cleanup and Resource Leaks

Cgroup cleanup requires careful ordering to avoid resource leaks and permission errors. Attempting to remove a cgroup directory while processes are still assigned to it fails, but processes may continue running even after the parent process exits if they've daemonized or been inherited by init.

The cleanup process must also handle hierarchical dependencies - child cgroups must be removed before their parents, and all processes must be terminated or migrated before cgroup removal. Failing to clean up cgroups properly leads to resource leaks where system resources remain allocated to non-existent sandbox instances.

Prevention: Implement systematic cleanup that first terminates all processes in the cgroup (using SIGTERM followed by SIGKILL), waits for process exit, then removes child cgroups before parent cgroups. Use `find /sys/fs/cgroup -name "*sandbox*" -type d` to identify leaked cgroups during development and testing.

⚠️ Pitfall: Cgroups Version Detection Failures

Systems may support both cgroups v1 and v2 simultaneously, with different controllers available through each interface. Simply checking for the existence of `/sys/fs/cgroup/unified` is insufficient to determine v2 availability - the system may have v2 mounted but with limited controller support.

Detection logic must verify not only that cgroups v2 is available, but that the required controllers (memory, CPU, PID, I/O) are available and can be enabled for the sandbox hierarchy. Fallback logic must gracefully handle scenarios where some controllers are only available through v1 interfaces.

Prevention: Implement comprehensive version detection that checks controller availability in both v1 and v2 interfaces. Create capability matrices that map required functionality to available controller interfaces. Test detection logic on systems with different cgroup configurations including hybrid setups.

Implementation Guidance

The cgroup resource management component bridges the gap between Linux kernel resource control mechanisms and practical sandbox resource limiting. This implementation provides both the low-level cgroup manipulation primitives and the higher-level resource policy management needed to create secure, isolated execution environments.

Technology Recommendations

Component	Simple Option	Advanced Option
Cgroup Interface	Direct filesystem operations (/sys/fs/cgroup)	libcgroup wrapper library
Resource Monitoring	Periodic polling of cgroup stat files	Event-driven monitoring with inotify
Process Management	Signal-based process termination	Process tracking with pidfd (Linux 5.3+)
Configuration Format	Simple key-value pairs in struct	YAML/JSON configuration with validation
Error Handling	Basic error codes and logging	Structured error types with context

Recommended File Structure

The cgroup management functionality integrates with the existing sandbox architecture while providing clear separation of concerns for resource management operations:

```
project-root/
  cmd/sandbox/main.c              ← main sandbox launcher
  src/
    cgroup/
      cgroup_mgr.c                ← main cgroup management logic
      cgroup_mgr.h                ← public cgroup management interface
      cgroup_v1.c                 ← cgroups v1 specific implementation
      cgroup_v2.c                 ← cgroups v2 specific implementation
      resource_limits.c           ← resource limit calculation and validation
      resource_limits.h           ← resource limit data structures
    sandbox/
      sandbox_config.h            ← configuration structures (from previous milestones)
      sandbox_state.h             ← runtime state structures
    utils/
      file_utils.c                ← filesystem utilities for cgroup operations
      string_utils.c              ← string manipulation for cgroup paths
  tests/
    cgroup/
      test_cgroup_mgr.c           ← cgroup management unit tests
      test_resource_limits.c      ← resource limit validation tests
    integration/
      test_sandbox_limits.c       ← end-to-end resource limiting tests
```

Infrastructure Starter Code

File: src/cgroup/cgroup_mgr.h

```c
#ifndef CGROUP_MGR_H

#define CGROUP_MGR_H


#include <sys/types.h>

#include <stdint.h>


// Cgroup controller types - maps to Linux cgroup controllers

typedef enum {

    CGROUP_CONTROLLER_MEMORY = 1 << 0,

    CGROUP_CONTROLLER_CPU = 1 << 1,

    CGROUP_CONTROLLER_PID = 1 << 2,

    CGROUP_CONTROLLER_IO = 1 << 3,

    CGROUP_CONTROLLER_ALL = 0xF

} cgroup_controller_t;


// Cgroup version detection result

typedef enum {

    CGROUP_VERSION_UNKNOWN = 0,

    CGROUP_VERSION_V1,

    CGROUP_VERSION_V2,

    CGROUP_VERSION_HYBRID

} cgroup_version_t;


// Resource limit configuration structure

typedef struct {

    unsigned long memory_limit_bytes;    // Memory limit in bytes (0 = no limit)

    unsigned int cpu_quota_us;           // CPU quota in microseconds per period

    unsigned int cpu_period_us;          // CPU scheduling period in microseconds

    unsigned int pid_max;                // Maximum number of processes

    unsigned long io_read_bps;           // I/O read bandwidth limit (bytes/sec)

    unsigned long io_write_bps;          // I/O write bandwidth limit (bytes/sec)

    cgroup_controller_t enabled_controllers; // Bitmask of enabled controllers
```

```c
} cgroup_limits_t;

// Cgroup management state
typedef struct {
    char *cgroup_path;                  // Path to cgroup directory
    char *instance_id;                  // Unique identifier for this sandbox
    cgroup_version_t version;           // Detected cgroup version
    cgroup_controller_t available_controllers; // Available controllers bitmask
    int cleanup_needed;                 // Flag indicating cleanup required
    pid_t managed_pids[256];            // PIDs assigned to this cgroup
    int pid_count;                      // Number of managed PIDs
} cgroup_state_t;

// Initialize cgroup management system
cgroup_state_t* cgroup_state_init(const char *instance_id);

// Clean up cgroup management state
void cgroup_state_free(cgroup_state_t *state);

// Detect available cgroup version and controllers
int detect_cgroup_capabilities(cgroup_state_t *state);

// Create cgroup hierarchy for sandbox instance
int create_cgroup_hierarchy(cgroup_state_t *state);

// Configure resource limits for cgroup
int configure_resource_limits(cgroup_state_t *state, const cgroup_limits_t *limits);

// Add process to cgroup
int add_process_to_cgroup(cgroup_state_t *state, pid_t pid);

// Monitor resource usage and limit violations
int monitor_resource_usage(cgroup_state_t *state);
```

```
// Clean up cgroup hierarchy and terminate processes
int cleanup_cgroup_hierarchy(cgroup_state_t *state);

// Utility functions for resource limit calculations
cgroup_limits_t* create_default_limits();
int validate_resource_limits(const cgroup_limits_t *limits);
void print_resource_limits(const cgroup_limits_t *limits);

#endif // CGROUP_MGR_H
```

File: src/utils/file_utils.c

```c
#include "file_utils.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>

#include <sys/stat.h>


// Write string value to file - commonly used for cgroup control files

int write_file_string(const char *path, const char *value) {

    FILE *file = fopen(path, "w");

    if (!file) {

        return -errno;

    }



    int result = fprintf(file, "%s", value);

    fclose(file);



    if (result < 0) {

        return -errno;

    }



    return SANDBOX_SUCCESS;

}


// Read string value from file - used for monitoring cgroup status

int read_file_string(const char *path, char *buffer, size_t buffer_size) {

    FILE *file = fopen(path, "r");

    if (!file) {

        return -errno;

    }
```

```
    if (!fgets(buffer, buffer_size, file)) {
        fclose(file);
        return -errno;
    }

    fclose(file);

    // Remove trailing newline if present
    size_t len = strlen(buffer);
    if (len > 0 && buffer[len - 1] == '\n') {
        buffer[len - 1] = '\0';
    }

    return SANDBOX_SUCCESS;
}

// Check if directory exists - used for cgroup hierarchy validation
int directory_exists(const char *path) {
    struct stat st;
    if (stat(path, &st) == 0 && S_ISDIR(st.st_mode)) {
        return 1;
    }
    return 0;
}

// Create directory with parents - used for cgroup hierarchy creation
int create_directory_recursive(const char *path, mode_t mode) {
    char *path_copy = strdup(path);
    if (!path_copy) {
        return -ENOMEM;
```

```
    }

    char *p = path_copy + 1; // Skip leading slash

    while ((p = strchr(p, '/'))) {
        *p = '\0';

        if (!directory_exists(path_copy)) {
            if (mkdir(path_copy, mode) != 0 && errno != EEXIST) {
                free(path_copy);
                return -errno;
            }
        }

        *p = '/';
        p++;
    }

    // Create final directory
    if (!directory_exists(path)) {
        if (mkdir(path, mode) != 0 && errno != EEXIST) {
            free(path_copy);
            return -errno;
        }
    }

    free(path_copy);
    return SANDBOX_SUCCESS;
}
```

Core Logic Skeleton Code

File: src/cgroup/cgroup_mgr.c

```c
#include "cgroup_mgr.h"

#include "../utils/file_utils.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <errno.h>


#define CGROUP_V1_ROOT "/sys/fs/cgroup"

#define CGROUP_V2_ROOT "/sys/fs/cgroup/unified"

#define SANDBOX_CGROUP_PREFIX "sandbox"


// Initialize cgroup management state for a sandbox instance

cgroup_state_t* cgroup_state_init(const char *instance_id) {

    // TODO 1: Allocate cgroup_state_t structure and initialize fields to safe defaults

    // TODO 2: Copy instance_id into state->instance_id (validate non-NULL and reasonable length)

    // TODO 3: Initialize managed_pids array to empty and set pid_count to 0

    // TODO 4: Set cleanup_needed flag to 0 initially

    // TODO 5: Call detect_cgroup_capabilities to determine available cgroup version and controllers

    // TODO 6: If detection fails, free allocated memory and return NULL

    // TODO 7: Return initialized state structure

    // Hint: Use strdup() for string copying and check return values for allocation failures

}


// Detect available cgroup version and controllers on the system

int detect_cgroup_capabilities(cgroup_state_t *state) {

    // TODO 1: Check if /sys/fs/cgroup/unified exists and has cgroup.controllers file (cgroups v2)

    // TODO 2: If v2 available, read /sys/fs/cgroup/unified/cgroup.controllers to get available
controllers

    // TODO 3: Parse controller list and set available_controllers bitmask for memory, cpu, pids, io

    // TODO 4: If v2 not available or incomplete, check for v1 controllers in /sys/fs/cgroup/memory,
/sys/fs/cgroup/cpu, etc.
```

```
    // TODO 5: Set state->version to CGROUP_VERSION_V2, CGROUP_VERSION_V1, or CGROUP_VERSION_HYBRID

    // TODO 6: Log detected version and available controllers for debugging

    // TODO 7: Return SANDBOX_SUCCESS if at least memory and cpu controllers available,
SANDBOX_ERROR_CGROUP otherwise

    // Hint: Use directory_exists() and read_file_string() utilities, parse space-separated controller
names

}


// Create hierarchical cgroup structure for sandbox isolation

int create_cgroup_hierarchy(cgroup_state_t *state) {

    // TODO 1: Build cgroup path based on version: "/sys/fs/cgroup/sandbox-parent/instance-<id>" (v2)
or "/sys/fs/cgroup/memory/sandbox-parent/instance-<id>" (v1)

    // TODO 2: Create parent directory "sandbox-parent" if it doesn't exist (shared by all sandbox
instances)

    // TODO 3: If using cgroups v2, enable required controllers in parent's cgroup.subtree_control
file

    // TODO 4: Create instance-specific subdirectory using state->instance_id

    // TODO 5: Store full cgroup path in state->cgroup_path for future operations

    // TODO 6: Set state->cleanup_needed = 1 to indicate cleanup required on exit

    // TODO 7: Return SANDBOX_SUCCESS on success, SANDBOX_ERROR_CGROUP on failure

    // Hint: Use create_directory_recursive() and write_file_string(), handle both v1 and v2 path
structures

}


// Configure resource limits by writing to appropriate cgroup control files

int configure_resource_limits(cgroup_state_t *state, const cgroup_limits_t *limits) {

    // TODO 1: Validate that state and limits are non-NULL and cgroup_path is set

    // TODO 2: If memory controller enabled, write memory limit to memory.limit_in_bytes (v1) or
memory.max (v2)

    // TODO 3: If CPU controller enabled, configure CPU quota by writing to cpu.cfs_quota_us and
cpu.cfs_period_us (v1) or cpu.max (v2)

    // TODO 4: If PID controller enabled, write process limit to pids.max

    // TODO 5: If I/O controller enabled, configure bandwidth limits using blkio.throttle.* (v1) or
io.max (v2)

    // TODO 6: Handle controller availability - skip controllers not available on this system
```

```
    // TODO 7: Return SANDBOX_SUCCESS if all enabled limits configured, SANDBOX_ERROR_CGROUP on any failure

    // Hint: Build control file paths by concatenating state->cgroup_path with controller-specific filenames

}


// Add process to cgroup by writing PID to cgroup.procs file

int add_process_to_cgroup(cgroup_state_t *state, pid_t pid) {

    // TODO 1: Validate state is non-NULL and cgroup_path is set

    // TODO 2: Check that pid is positive and process exists (kill(pid, 0) == 0)

    // TODO 3: Build path to cgroup.procs file within the cgroup directory

    // TODO 4: Convert pid to string and write to cgroup.procs file

    // TODO 5: Add pid to state->managed_pids array and increment state->pid_count

    // TODO 6: Check for array bounds to prevent overflow of managed_pids array

    // TODO 7: Return SANDBOX_SUCCESS on successful addition, SANDBOX_ERROR_CGROUP on failure

    // Hint: Use sprintf() to convert pid to string, handle ESRCH error if process exits during operation

}


// Monitor current resource usage and detect limit violations

int monitor_resource_usage(cgroup_state_t *state) {

    // TODO 1: Build paths to usage monitoring files (memory.usage_in_bytes, cpu.stat, etc.)

    // TODO 2: Read current memory usage and compare to configured limits

    // TODO 3: Read CPU usage statistics and calculate CPU utilization percentage

    // TODO 4: Read current process count from pids.current

    // TODO 5: Check for limit violation indicators (memory.failcnt, cpu throttling events)

    // TODO 6: Log current usage statistics and any detected violations

    // TODO 7: Return SANDBOX_SUCCESS normally, SANDBOX_ERROR_CGROUP if monitoring files inaccessible

    // Hint: Parse numeric values from cgroup stat files, handle different file formats between v1 and v2

}


// Clean up cgroup hierarchy by terminating processes and removing directories
```

```c
int cleanup_cgroup_hierarchy(cgroup_state_t *state) {

    // TODO 1: Send SIGTERM to all processes in state->managed_pids array

    // TODO 2: Wait up to 5 seconds for processes to terminate gracefully

    // TODO 3: Send SIGKILL to any remaining processes that didn't terminate

    // TODO 4: Wait for all processes to exit completely

    // TODO 5: Remove cgroup directory (this automatically removes from hierarchy)

    // TODO 6: If this was the last instance, remove parent "sandbox-parent" directory

    // TODO 7: Reset state->cleanup_needed flag and clear cgroup_path

    // Hint: Use kill() for signaling, waitpid() with WNOHANG for checking exit status, rmdir() for cleanup

}


// Create default resource limit configuration suitable for most sandbox workloads

cgroup_limits_t* create_default_limits() {

    cgroup_limits_t *limits = malloc(sizeof(cgroup_limits_t));

    if (!limits) {

        return NULL;

    }



    // Set moderate defaults balancing security and usability

    limits->memory_limit_bytes = 128 * 1024 * 1024;  // 128MB

    limits->cpu_quota_us = 25000;                     // 25% of one CPU core

    limits->cpu_period_us = 100000;                   // 100ms period

    limits->pid_max = 32;                             // 32 processes max

    limits->io_read_bps = 10 * 1024 * 1024;          // 10MB/s read

    limits->io_write_bps = 5 * 1024 * 1024;          // 5MB/s write

    limits->enabled_controllers = CGROUP_CONTROLLER_ALL;



    return limits;

}
```

Language-Specific Hints

C-Specific Implementation Details:

- Use `snprintf()` for safe string formatting when building cgroup file paths to prevent buffer overflows
- Handle `EINTR` when reading from cgroup files as operations may be interrupted by signals
- Use `strtoul()` and `strtoull()` for parsing numeric values from cgroup stat files with proper error checking
- Remember that cgroup file writes may fail with `EBUSY` if the cgroup contains running processes during certain operations
- Use `access()` with `W_OK` to check write permissions before attempting to modify cgroup control files

Memory Management:

- Always `free()` the cgroup_state_t structure and its allocated strings in the cleanup function
- Use `strdup()` for copying strings into the state structure and check for allocation failures
- Consider using a memory pool or arena allocator for cgroup path strings to reduce fragmentation

Error Handling Patterns:

- Return negative errno values for system call failures to provide detailed error information
- Use consistent error codes (`SANDBOX_ERROR_CGROUP`) for cgroup-specific failures that aren't system call related
- Log error details with `perror()` or `strerror()` when system calls fail for debugging assistance

Milestone Checkpoint

After implementing the cgroup resource management component, verify correct functionality with these validation steps:

Basic Functionality Test:

```bash
# Compile and run basic cgroup test
gcc -o test_cgroup src/cgroup/*.c src/utils/*.c tests/cgroup/test_cgroup_mgr.c
sudo ./test_cgroup

# Expected output should show:
# - Detected cgroup version (v1, v2, or hybrid)
# - Available controllers (memory, cpu, pids, io)
# - Successfully created cgroup hierarchy
# - Configured resource limits without errors
# - Added test process to cgroup
# - Read back resource usage statistics
# - Cleaned up cgroup hierarchy
```

Integration Test with Previous Milestones:

```bash
# Run complete sandbox with resource limits

sudo ./sandbox --memory-limit 64M --cpu-percent 10 --pid-limit 16 /bin/sh -c 'echo "Testing resource limits"'


# Verify resource constraints:

# 1. Memory limit should be visible in /sys/fs/cgroup/.../memory.max (or memory.limit_in_bytes)

# 2. CPU limit should be configured in cpu.max (or cpu.cfs_quota_us)

# 3. Process should be listed in cgroup.procs file

# 4. Resource usage should be trackable in usage files
```

Resource Limit Enforcement Test:

```bash
# Test memory limit enforcement (should terminate with OOM)

sudo ./sandbox --memory-limit 32M /bin/sh -c 'dd if=/dev/zero of=/dev/null bs=1M count=64'


# Test CPU limit enforcement (should show throttling in cpu.stat)

sudo ./sandbox --cpu-percent 10 /bin/sh -c 'yes > /dev/null' &

sleep 5

# Check cpu.stat file for nr_throttled events


# Test process limit enforcement (should fail with EAGAIN)

sudo ./sandbox --pid-limit 4 /bin/bash -c 'for i in {1..10}; do sleep 60 & done'
```

Signs of Problems and Debugging:

- **"Operation not permitted" when writing cgroup files**: Check that you're running as root and that parent cgroups have appropriate controllers enabled
- **Cgroup creation fails**: Verify that cgroup filesystem is mounted and writable, check `dmesg` for kernel cgroup errors
- **Resource limits not enforced**: Confirm that processes are actually assigned to the cgroup by checking `cgroup.procs` file contents
- **Cleanup failures**: Check for zombie processes that prevent cgroup removal, use `kill -9` on stubborn processes
- **Controller not available**: Verify kernel configuration includes required cgroup controllers (`CONFIG_MEMCG`, `CONFIG_CPU_SETS`, etc.)

The successful completion of this milestone provides the foundation for comprehensive resource management that prevents resource exhaustion attacks while maintaining usable performance boundaries for legitimate sandbox workloads.

Capability Management Component

> **Milestone(s):** This section corresponds to Milestone 5 (Capability Dropping) and represents the final security layer, building upon all previous isolation mechanisms from Milestones 1-4 to achieve complete privilege minimization

The capability management component represents the final layer in our defense-in-depth sandbox architecture. After establishing namespace isolation, filesystem restrictions, system call filtering, and resource limits, we must address one critical remaining attack vector: **privilege escalation**. Even within a heavily restricted environment, a process running with unnecessary Linux capabilities retains pathways to break out of the sandbox or cause system-wide damage.

Mental Model: The Security Clearance System

Think of Linux capabilities as a **government security clearance system**. In traditional Unix systems, processes are either "civilians" (regular users) or "generals" (root with unlimited power). This binary model creates a dangerous situation: any process that needs even one privileged operation must be granted the equivalent of top-secret clearance, giving it access to far more than it actually requires.

Linux capabilities solve this problem by subdividing root's omnipotent powers into specific, granular permissions—like specialized security clearances in a military organization. Instead of giving someone "access to everything classified," you can grant specific clearances: "authorized to access network configuration files," "permitted to bind to privileged ports," or "allowed to modify process scheduling." Each capability represents a different type of security clearance that can be granted or revoked independently.

Just as a military organization follows the principle of least privilege—granting only the minimum clearance level needed for each job—our sandbox must systematically drop all capabilities except those absolutely essential for the sandboxed workload. A document editor doesn't need network administration capabilities, and a computational task doesn't need the ability to mount filesystems.

The complexity arises from Linux's sophisticated capability inheritance model. When a process spawns children or executes new programs, different rules govern which security clearances transfer automatically, which must be explicitly granted, and which are permanently revoked. Understanding these inheritance patterns is crucial for maintaining security across process boundaries and preventing accidental privilege escalation.

Capability Sets and Inheritance

Linux implements capabilities through five distinct capability sets that govern how privileges are managed and inherited across process execution boundaries. Each set serves a specific role in the overall security model, and understanding their interactions is essential for safely dropping privileges.

Capability Set	Purpose	Inheritance Behavior	Security Impact
Effective	Currently active privileges	Reset on exec, must be explicitly activated	Controls what the process can do right now
Permitted	Maximum possible privileges	Inherited based on file capabilities and other sets	Upper bound for effective capabilities
Inheritable	Privileges that can transfer to child processes	Preserved across exec if file allows	Determines what children can potentially inherit
Bounding	System-wide privilege ceiling	Cannot be raised, only lowered	Prevents any process from gaining specific capabilities
Ambient	Simplified inheritance for unprivileged processes	Automatically becomes effective after exec	Provides easier capability management for non-root

The **effective set** contains capabilities that are currently active and usable by the process. These are the actual privileges the process can exercise at any given moment. When a process calls a privileged operation, the kernel checks this set to determine authorization. The effective set can be modified during runtime by adding capabilities from the permitted set or removing capabilities entirely.

The **permitted set** represents the maximum capabilities the process could potentially activate. Think of this as a pool of available privileges that the process can draw from to populate its effective set. The permitted set cannot be expanded beyond its current contents, but capabilities can be permanently dropped from it. This set plays a crucial role during exec operations, as it determines what privileges survive the transition to a new program.

The **inheritable set** controls which capabilities can be passed to child processes during exec operations. However, inheritance is not automatic—it depends on complex interactions with the target executable's file capabilities and the ambient set. This set is preserved across exec boundaries but only becomes active in the child if specific conditions are met.

The **bounding set** functions as a system-wide capability ceiling. No process can acquire capabilities that are not present in the bounding set, regardless of file capabilities or inheritance rules. Dropping capabilities from the bounding set is a one-way operation that affects the current process and all its descendants, making it a powerful tool for permanent privilege reduction.

The **ambient set** provides a simplified inheritance mechanism designed for unprivileged processes. Capabilities in the ambient set automatically become effective in child processes after exec, eliminating the need for complex file capability configurations. However, ambient capabilities pose security risks if not carefully managed.

> **Architecture Decision: Capability Dropping Strategy**
>
> - **Context**: We need to determine which capability sets to modify and in what order to achieve maximum security while maintaining functionality
> - **Options Considered**:
>   1. Drop only from effective set (preserves potential for re-elevation)
>   2. Drop from all sets simultaneously (may cause unexpected failures)
>   3. Systematic dropping: bounding → ambient → inheritable → permitted → effective
> - **Decision**: Use systematic dropping with verification at each step
> - **Rationale**: This approach provides defense in depth by creating multiple barriers to privilege escalation while allowing us to detect issues early in the process
> - **Consequences**: More complex implementation but maximum security assurance and better debugging capability

The capability inheritance algorithm during exec operations follows intricate rules that combine file capabilities, process capabilities, and security policies. When a process executes a new program, the kernel calculates the new capability sets using formulas that consider the current process capabilities, the target file's capabilities, and various security flags.

For unprivileged processes (non-root), the inheritance calculation is:

1. New permitted = (current inheritable & file inheritable) | (file permitted & capability bounding set)
2. New effective = file effective ? new permitted : empty set
3. New inheritable = current inheritable & file inheritable
4. New ambient = current ambient if program allows, otherwise empty

These inheritance rules create several security implications that our sandbox must address. Most critically, any capabilities left in the inheritable set could potentially transfer to child processes if they execute programs with matching file capabilities. Similarly, capabilities remaining in the ambient set automatically become effective in child processes, creating a straightforward privilege escalation path.

Privilege Dropping Algorithm

The process of safely dropping Linux capabilities requires careful orchestration of multiple operations in a specific sequence. The order of operations is critical because certain steps are irreversible, and incorrect sequencing can leave the process in an inconsistent security state or cause legitimate operations to fail unexpectedly.

The complete privilege dropping algorithm proceeds through several phases, each with specific verification steps:

1. **Capability Assessment Phase**: The first step involves querying the current process capabilities across all five capability sets. This assessment provides a baseline for verification and helps identify which capabilities are currently active. The `capget` system call retrieves the current capability state, which we store for comparison after each dropping operation.

2. **Bounding Set Reduction Phase**: We begin by dropping capabilities from the bounding set because this operation creates a permanent ceiling that cannot be raised. Each capability not explicitly required for the sandboxed workload is removed using the `prctl` system call with `PR_CAPBSET_DROP`. This operation requires `CAP_SETPCAP` capability, so it must be performed before dropping that capability from our own sets.

3. **Ambient Set Clearing Phase**: The ambient capability set is cleared entirely because these capabilities automatically become effective in child processes. The `prctl` system call with `PR_CAP_AMBIENT` and `PR_CAP_AMBIENT_CLEAR_ALL` removes all ambient capabilities in a single operation. This prevents child processes from inheriting unexpected privileges.

4. **User and Group Transition Phase**: Before dropping the remaining capability sets, we transition to the target user and group identities. This step is crucial because changing user ID typically clears many capability sets automatically. The sequence is: `setgid`, `setgroups`, then `setuid`. Each operation must be verified to ensure it succeeded.

5. **Inheritable Set Clearing Phase**: The inheritable capability set is cleared to prevent any capabilities from transferring to child processes through the inheritance mechanism. This operation uses `capset` system call to explicitly set the inheritable set to empty.

6. **Permitted Set Reduction Phase**: Capabilities are removed from the permitted set, leaving only those absolutely necessary for the sandboxed workload. This step requires careful analysis of the target program's requirements, as dropping essential capabilities will cause the program to fail when it attempts privileged operations.

7. **Effective Set Minimization Phase**: Finally, the effective capability set is reduced to match the permitted set or further restricted based on immediate needs. This set can be modified during runtime, so we set it to the minimum required for program startup.

8. **No New Privileges Flag**: The `PR_SET_NO_NEW_PRIVS` flag is set using `prctl` to prevent the process and its children from gaining privileges through exec operations, even if they execute setuid programs or programs with file capabilities.

Each phase includes verification steps to ensure the operations succeeded and the process is in the expected security state. The verification process re-reads the capability sets and compares them against expected values, providing immediate detection of any failures in the privilege dropping process.

> The critical insight in capability dropping is that it's a one-way operation with complex interdependencies. Once capabilities are dropped from certain sets, they cannot be recovered, making verification at each step essential for debugging and security assurance.

The algorithm must also handle several edge cases and error conditions. If the target program requires capabilities that were accidentally dropped, the error typically manifests as `EPERM` errors when the program attempts privileged operations. The algorithm includes rollback provisions for non-critical failures, but certain operations like bounding set modifications cannot be undone.

No New Privileges Flag

The `PR_SET_NO_NEW_PRIVS` flag represents a fundamental security mechanism that prevents privilege escalation through exec operations. This flag creates an irreversible security boundary that blocks several common attack vectors used to escape from restricted environments.

When `PR_SET_NO_NEW_PRIVS` is set, the kernel prevents the process and all its descendants from gaining privileges in several ways. First, it disables the setuid and setgid bits on executable files. Normally, when a process executes a setuid program, the effective user ID changes to the file's owner, potentially granting additional privileges. With `PR_SET_NO_NEW_PRIVS` set, these bits are ignored, and the process continues running with its current credentials.

Second, the flag prevents file capabilities from granting new privileges during exec operations. File capabilities are extended attributes that can grant specific Linux capabilities to processes executing a program. Without the no-new-privs flag, a restricted process could potentially execute a program with file capabilities to regain privileges. The flag ensures that exec operations cannot increase the process's capability sets beyond their current contents.

Third, certain Linux Security Modules (LSMs) like AppArmor and SELinux respect the no-new-privs flag and may apply additional restrictions. These systems can implement more permissive policies for processes that have committed to not gaining new privileges, creating a trust boundary that enables more flexible security policies.

The flag also interacts with seccomp filters in important ways. When `PR_SET_NO_NEW_PRIVS` is set, unprivileged processes can install seccomp filters that would otherwise require `CAP_SYS_ADMIN`. This interaction is particularly relevant for our sandbox because it ensures that the seccomp restrictions cannot be bypassed through privilege escalation.

Privilege Escalation Vector	Without NO_NEW_PRIVS	With NO_NEW_PRIVS
Setuid executables	Process gains file owner's privileges	Setuid bit ignored, no privilege change
Setgid executables	Process gains file group's privileges	Setgid bit ignored, no privilege change
File capabilities	Process can gain capabilities from file	File capabilities cannot increase process caps
LSM transitions	May allow privilege-granting transitions	LSM policies may provide additional restrictions
Seccomp installation	Requires CAP_SYS_ADMIN for unprivileged	Allows unprivileged seccomp filter installation

The timing of setting `PR_SET_NO_NEW_PRIVS` within our privilege dropping algorithm is crucial. It should be set after the process has transitioned to its target user ID and dropped most capabilities, but before executing the final target program. Setting it too early might interfere with necessary privilege operations during sandbox setup. Setting it too late leaves a window where privilege escalation might be possible.

> **Architecture Decision: NO_NEW_PRIVS Timing**
>
> - **Context**: Determining when to set PR_SET_NO_NEW_PRIVS in the capability dropping sequence
> - **Options Considered**:
>   1. Set immediately after fork (blocks all privilege operations)
>   2. Set after user transition but before capability dropping (may interfere with capability operations)
>   3. Set after capability dropping but before exec (optimal security/functionality balance)
> - **Decision**: Set after complete capability dropping, immediately before exec
> - **Rationale**: Ensures all necessary privilege operations complete successfully while preventing privilege escalation in the target program
> - **Consequences**: Maximum compatibility with target programs while maintaining strong privilege escalation prevention

The no-new-privs flag cannot be unset once it's enabled, making it a permanent commitment for the process and all its descendants. This irreversible nature provides strong security guarantees but requires careful consideration of the target program's requirements. Some legitimate programs may fail if they depend on setuid operations or file capabilities for normal functionality.

Architecture Decisions

The capability management component requires several critical architecture decisions that significantly impact both security effectiveness and implementation complexity. These decisions must balance maximum privilege reduction against practical usability and compatibility with diverse sandboxed workloads.

> **Decision: Minimum Required Capabilities**
>
> - **Context**: Determining the absolute minimum set of capabilities that sandboxed processes need to function
> - **Options Considered**:
>   1. Drop all capabilities (maximum security, high compatibility risk)
>   2. Retain common capabilities like CAP_CHOWN, CAP_DAC_OVERRIDE (moderate security, better compatibility)
>   3. Configurable capability whitelist based on workload type (complex but flexible)
> - **Decision**: Configurable whitelist with secure defaults dropping all capabilities
> - **Rationale**: Different sandboxed programs have vastly different requirements; flexibility allows optimization while secure defaults protect against misconfiguration
> - **Consequences**: Requires careful analysis of each workload's needs but provides maximum security when properly configured

The minimum capability set decision proves particularly challenging because Linux capabilities are not well-documented from an application perspective. Many programs don't clearly specify which capabilities they require, leading to trial-and-error discovery processes. Our architecture addresses this by providing a capability discovery mode that logs attempted privileged operations, helping administrators identify necessary capabilities through testing.

Capability	Common Use Cases	Security Risk	Recommended Action
`CAP_CHOWN`	Change file ownership	Medium - can manipulate file permissions	Drop unless file management needed
`CAP_DAC_OVERRIDE`	Bypass file permission checks	High - can access any file	Drop unless specific access patterns required
`CAP_SETUID`	Change user ID	High - enables privilege escalation	Always drop after user transition
`CAP_SETGID`	Change group ID	Medium - enables group privilege escalation	Always drop after group transition
`CAP_NET_BIND_SERVICE`	Bind to privileged ports (<1024)	Medium - can impersonate system services	Drop unless network service needed
`CAP_SYS_ADMIN`	Many administrative operations	Critical - effectively equivalent to root	Always drop
`CAP_SYS_PTRACE`	Trace other processes	Critical - can access process memory	Always drop

Decision: Capability Verification Strategy

- **Context**: How to verify that capability dropping succeeded and the process is in the expected security state
- **Options Considered**:
  1. Trust system call return values (fast but unreliable)
  2. Re-read capabilities after each operation (thorough but slow)
  3. Final verification only (balanced approach)
- **Decision**: Verification after each major phase with detailed logging
- **Rationale**: Capability operations can fail silently or partially; immediate verification enables precise error diagnosis and security assurance
- **Consequences**: Slightly higher overhead but dramatically improved security confidence and debugging capability

The verification strategy implementation maintains a shadow copy of expected capability sets and compares them against actual kernel state after each modification. This approach catches several classes of errors: partial failures where some capabilities are dropped but others remain, kernel version incompatibilities that cause operations to be ignored, and race conditions where other processes modify our capabilities unexpectedly.

Decision: Capability Drop Failure Handling

- **Context**: How to respond when capability dropping operations fail
- **Options Considered**:
  1. Continue with reduced security (dangerous but functional)
  2. Abort sandbox creation entirely (secure but may impact availability)
  3. Retry with progressively relaxed requirements (complex fallback logic)
- **Decision**: Abort sandbox creation with detailed error reporting
- **Rationale**: Security boundaries must be absolute; partial security provides false confidence and unknown attack surface
- **Consequences**: Some workloads may fail to start, but security guarantees are maintained and administrators receive clear guidance for resolution

The failure handling strategy extends beyond simple abortion to include comprehensive diagnostic information. When capability operations fail, the system logs the current state of all capability sets, the attempted operation, the specific error code, and recommendations for resolution. This detailed error reporting significantly reduces the debugging burden for administrators trying to configure appropriate capability whitelists.

The capability management component also implements several performance optimizations that don't compromise security. Capability queries are batched where possible to reduce system call overhead. The verification process uses efficient bit manipulation to compare capability sets rather than iterating through individual capabilities. Error paths are optimized for fast failure detection to minimize the impact on sandbox creation performance.

Common Pitfalls

Capability management presents several subtle pitfalls that frequently trap developers implementing process sandboxes. These mistakes often result in security vulnerabilities, unexpected program failures, or difficult-to-diagnose runtime issues.

⚠️ Pitfall: Incorrect Operation Ordering

Many developers attempt to drop capabilities before completing necessary privileged operations, causing the privilege dropping process itself to fail. For example, calling `setuid()` before dropping `CAP_SETUID` from the bounding set will succeed, but attempting to drop `CAP_SETUID` from the bounding set after calling `setuid()` may fail because the process no longer has the necessary privileges to modify capability sets.

The correct sequence must complete all privileged operations that require specific capabilities before dropping those capabilities from any set. This includes operations like changing user/group IDs, setting up cgroups, modifying namespaces, and installing seccomp filters. A common mistake is forgetting that some operations require capabilities that aren't immediately obvious—for example, certain cgroup operations may require `CAP_DAC_OVERRIDE` to write to cgroup control files.

To avoid this pitfall, create a comprehensive checklist of all privileged operations your sandbox performs and identify the capabilities each operation requires. Perform all operations before beginning the capability dropping sequence, and verify each operation's success before proceeding to the next phase.

⚠️ Pitfall: Ambient Capability Oversight

Ambient capabilities are often overlooked because they're a relatively recent addition to Linux and don't appear in older documentation. Developers frequently remember to drop capabilities from the traditional four sets (effective, permitted, inheritable, bounding) but leave ambient capabilities intact. This creates a significant security vulnerability because ambient capabilities automatically become effective in child processes.

Even if a process successfully drops all capabilities from its effective and permitted sets, any capabilities remaining in the ambient set will be granted to child processes after exec operations. This means a sandboxed process could spawn a child that has more privileges than the parent, completely defeating the purpose of capability dropping.

The solution requires explicitly clearing the ambient capability set using `prctl(PR_CAP_AMBIENT, PR_CAP_AMBIENT_CLEAR_ALL)`. This operation should be performed early in the privilege dropping sequence, ideally immediately after assessing current capabilities but before beginning other dropping operations.

⚠️ Pitfall: Incomplete Capability Assessment

Many implementations fail to comprehensively assess which capabilities a target program actually requires, leading to either security vulnerabilities (retaining unnecessary capabilities) or functionality failures (dropping required capabilities). This assessment is complicated by the fact that most programs don't document their capability requirements, and the requirements may vary based on configuration or runtime behavior.

A partial assessment might identify obvious requirements like `CAP_NET_BIND_SERVICE` for programs that bind to privileged ports, but miss subtle requirements like `CAP_DAC_READ_SEARCH` for programs that need to traverse directories with restrictive permissions. These subtle requirements often only manifest under specific conditions, making them difficult to discover through casual testing.

To address this pitfall, implement a systematic capability discovery process. Run the target program in a test environment with capability auditing enabled, exercise all program functionality, and log any capability-related failures. Use tools like `filecap` and `getcap` to identify file capabilities that the program might depend on. Document the discovered requirements and validate them through comprehensive testing.

⚠️ Pitfall: Race Conditions in Multi-threaded Programs

Capability operations affect the entire process, including all threads, but the timing of these operations can create race conditions in multi-threaded programs. If the sandbox process creates threads before dropping capabilities, those threads

might perform privileged operations concurrently with capability dropping, leading to inconsistent security states or unexpected failures.

For example, if one thread is dropping `CAP_NET_ADMIN` while another thread is attempting to configure network interfaces, the network configuration might succeed or fail depending on the exact timing of the operations. This creates non-deterministic behavior that's difficult to debug and may leave the sandbox in an unexpected security state.

The solution requires careful coordination of capability dropping with thread management. Ensure that capability dropping occurs before creating any threads that might perform privileged operations, or implement explicit synchronization to prevent privileged operations during capability transitions. Consider using the `PR_SET_NO_NEW_PRIVS` flag early in the process to prevent threads from gaining new privileges even if capability dropping is delayed.

⚠️ Pitfall: Verification Logic Errors

Capability verification logic often contains subtle bugs that can mask security vulnerabilities or cause false alarms. Common errors include incorrect bit manipulation when checking capability sets, endianness issues when reading capability data structures, and failure to account for kernel version differences in capability representation.

One particularly common mistake is assuming that capability numbers are consistent across all Linux architectures and kernel versions. Capability constants like `CAP_NET_ADMIN` have the same numeric values across systems, but the internal kernel representation and the format returned by `capget()` can vary. This can cause verification logic to incorrectly interpret capability sets or fail to detect security issues.

Implement capability verification using well-tested libraries like `libcap` rather than raw system calls where possible. When using raw system calls is necessary, carefully validate the verification logic against known capability states and test on multiple kernel versions and architectures.

Implementation Guidance

The capability management component requires careful integration of Linux-specific system calls with robust error handling and verification logic. The implementation must handle the complex interactions between different capability sets while providing clear diagnostics for debugging capability-related issues.

Technology Recommendations:

Component	Simple Option	Advanced Option
Capability Operations	Raw system calls (`capget`, `capset`, `prctl`)	`libcap` library with higher-level abstractions
Verification Logic	Manual bit manipulation and comparison	Capability parsing libraries with built-in validation
Error Reporting	Simple error codes and messages	Structured logging with capability state dumps
Testing Framework	Basic unit tests with mock capabilities	Comprehensive integration tests with real privilege dropping

Recommended File Structure:

```
project-root/
  src/
    capability_manager.c     ← Core capability dropping logic
    capability_manager.h     ← Public interface and type definitions
    capability_utils.c       ← Utility functions for capability operations
    capability_utils.h       ← Utility function declarations
    test/
      test_capability_manager.c  ← Unit tests for capability operations
      test_capability_utils.c    ← Tests for utility functions
    include/
      sandbox_types.h        ← Shared type definitions
```

Infrastructure Starter Code:

```c
#include <sys/capability.h>
#include <sys/prctl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

// Capability utility functions - complete implementations
int get_current_capabilities(cap_t *caps) {
    *caps = cap_get_proc();
    if (*caps == NULL) {
        return -errno;
    }
    return 0;
}

int verify_capability_dropped(cap_value_t cap) {
    cap_t caps;
    cap_flag_value_t value;
    int result = get_current_capabilities(&caps);
    if (result < 0) {
        return result;
    }


    if (cap_get_flag(caps, cap, CAP_EFFECTIVE, &value) < 0) {
        cap_free(caps);
        return -errno;
    }


    cap_free(caps);
    return (value == CAP_CLEAR) ? 1 : 0;
}
```

```
void log_capability_state(const char* phase) {
    cap_t caps = cap_get_proc();
    if (caps != NULL) {
        char* cap_string = cap_to_text(caps, NULL);
        if (cap_string != NULL) {
            printf("Capabilities at %s: %s\n", phase, cap_string);
            cap_free(cap_string);
        }
        cap_free(caps);
    }
}
```

Core Logic Skeleton Code:

```c
// Drop all Linux capabilities except those in the whitelist
// Returns SANDBOX_SUCCESS on success, specific error codes on failure
int drop_capabilities(const char** capability_whitelist, size_t whitelist_size) {
    // TODO 1: Log initial capability state for debugging
    // TODO 2: Validate whitelist contains only safe capabilities
    // TODO 3: Drop capabilities from bounding set (all except whitelist)
    // Hint: Use prctl(PR_CAPBSET_DROP, cap_value) for each non-whitelisted capability

    // TODO 4: Clear ambient capability set completely
    // Hint: Use prctl(PR_CAP_AMBIENT, PR_CAP_AMBIENT_CLEAR_ALL)

    // TODO 5: Clear inheritable capability set
    // TODO 6: Set permitted set to whitelist only
    // TODO 7: Set effective set to match permitted set
    // Hint: Use cap_set_proc() after building capability set with cap_set_flag()

    // TODO 8: Verify each dropped capability is actually cleared
    // TODO 9: Log final capability state and return success
}

// Set the PR_SET_NO_NEW_PRIVS flag to prevent privilege escalation
// Must be called after all necessary privileged operations are complete
int set_no_new_privs() {
    // TODO 1: Set PR_SET_NO_NEW_PRIVS flag using prctl
    // TODO 2: Verify the flag was set successfully
    // TODO 3: Log the no-new-privs state for confirmation
    // Hint: Use prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)
}

// Transition to target user and group, dropping setuid/setgid capabilities
// This must be done before dropping other capabilities to avoid permission errors
```

```
int change_user_group(uid_t target_uid, gid_t target_gid) {
    // TODO 1: Change to target group ID first (setgid)
    // TODO 2: Clear supplementary groups (setgroups)
    // TODO 3: Change to target user ID (setuid)
    // TODO 4: Verify user and group transition succeeded
    // TODO 5: Verify we can no longer change back to original IDs
    // Hint: Order matters - do setgid before setuid, and verify each step
}

// Validate that a capability whitelist contains only safe capabilities
// Returns 0 if safe, specific error code if dangerous capabilities found
int validate_capability_whitelist(const char** whitelist, size_t size) {
    // TODO 1: Define list of always-dangerous capabilities (CAP_SYS_ADMIN, CAP_SYS_PTRACE, etc.)
    // TODO 2: Iterate through whitelist and check each capability name
    // TODO 3: Return error if any dangerous capabilities are whitelisted
    // TODO 4: Warn about potentially risky capabilities that are allowed
    // TODO 5: Validate capability names are recognized by the system
}
```

Language-Specific Hints for C:

- Use `libcap-dev` package for higher-level capability operations rather than raw system calls
- The `cap_t` type is opaque - always use library functions to manipulate it
- Remember to call `cap_free()` on capability structures to prevent memory leaks
- Use `errno` to get detailed error information when capability operations fail
- Test capability operations with `strace` to see exactly which system calls are being made
- The `PR_SET_NO_NEW_PRIVS` constant requires `#include <sys/prctl.h>`

Milestone Checkpoint: After implementing the capability management component, verify the following behavior:

1. **Capability Enumeration Test**: Run `getpcaps $$` before and after capability dropping to confirm capabilities are actually removed from all sets.
2. **Privilege Operation Test**: After dropping capabilities, attempt operations that should fail (e.g., binding to port 80, accessing `/etc/shadow`). Verify these operations return `EPERM`.
3. **No New Privileges Test**: After setting the no-new-privs flag, attempt to execute a setuid program. Verify the program runs with current privileges rather than elevated privileges.

4. **Child Process Test**: Fork a child process after capability dropping and verify the child inherits the restricted capability set, not the original capabilities.
5. **Integration Test**: Run the complete sandbox with capability dropping enabled and verify the target program functions correctly with minimal capabilities.

Expected output for capability verification:

```
Capabilities before dropping: = cap_chown,cap_dac_override,cap_fowner,cap_setgid,cap_setuid+eip
Dropping capabilities from bounding set...
Clearing ambient capabilities...
Setting capability sets...
Capabilities after dropping: =
No new privileges flag set successfully
Capability dropping completed successfully
```

Signs of problems and debugging steps:

- If capability operations return `EPERM`: Check that the process has sufficient privileges to perform capability operations (may need to be root initially)
- If the target program fails with permission errors: Review the capability whitelist and add necessary capabilities
- If child processes have unexpected privileges: Verify ambient and inheritable sets are properly cleared
- If setuid programs still gain privileges: Confirm the `PR_SET_NO_NEW_PRIVS` flag is set correctly

Sandbox Orchestration and Data Flow

> **Milestone(s):** This section integrates concepts from all milestones (1-5), showing how namespace isolation, filesystem restrictions, system call filtering, resource limits, and capability dropping work together to create a complete sandbox

The **sandbox orchestration** represents the conductor of a complex symphony, where each security mechanism plays its part in perfect harmony. Like a master chef following a precise recipe, the orchestration must combine ingredients (security primitives) in exactly the right order and proportions to create the desired outcome—a secure execution environment that contains untrusted code without breaking functionality.

The orchestration component serves as the **central coordinator** that understands the dependencies between security layers and ensures they are applied in the correct sequence. This is not merely about calling functions in order; it requires deep understanding of how Linux security primitives interact, what state each component needs from its predecessors, and how to handle the complex cleanup required when things go wrong.

Mental Model: The Security Assembly Line

Think of sandbox creation as a **sophisticated assembly line** in a high-security facility. Each station on the line adds another layer of protection to the product (the untrusted process). The line supervisor (orchestration component) ensures that:

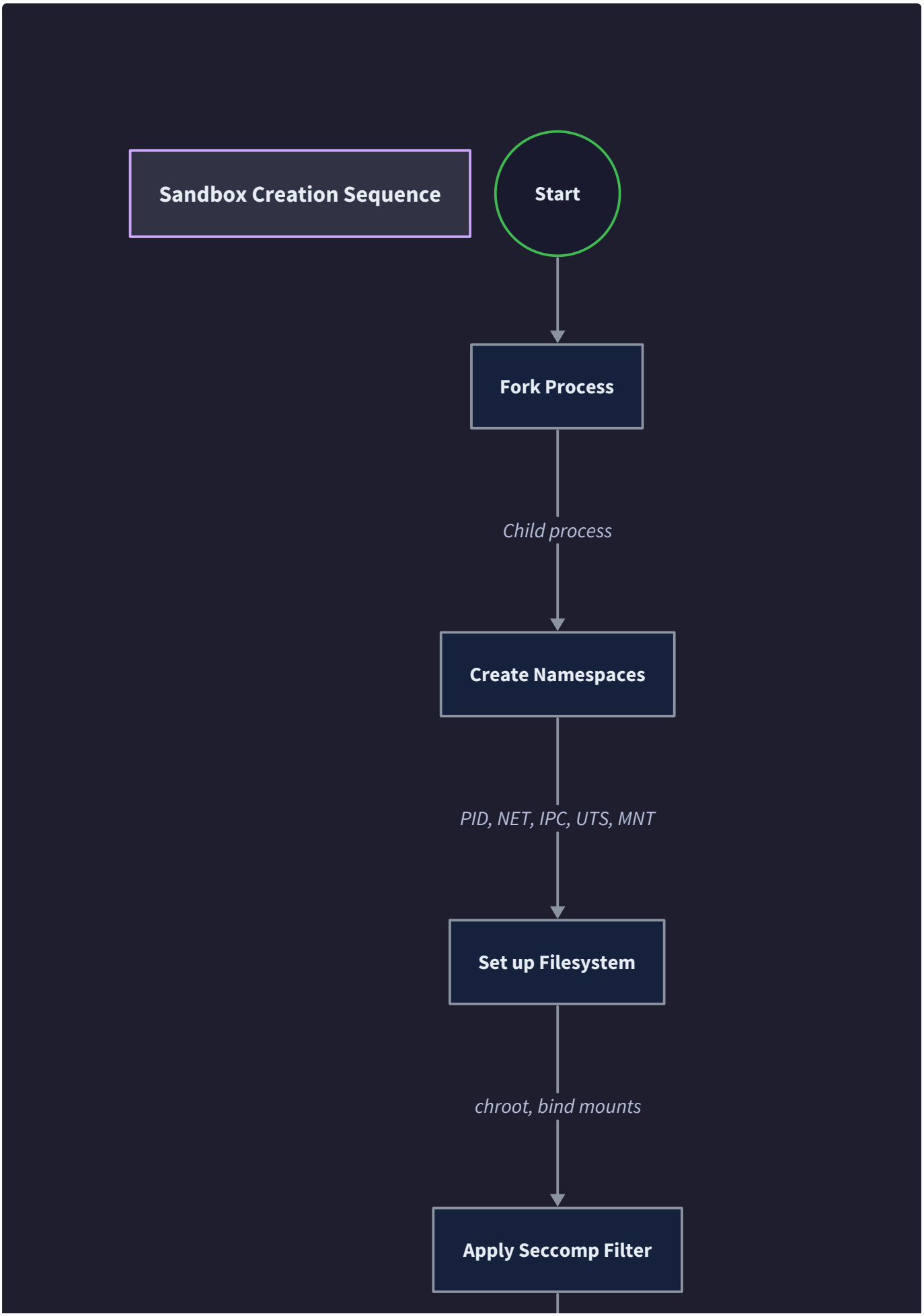
- Each station receives exactly what it needs from previous stations
- No station begins work until prerequisites are complete
- If any station fails, the entire assembly line stops and reverses all completed work

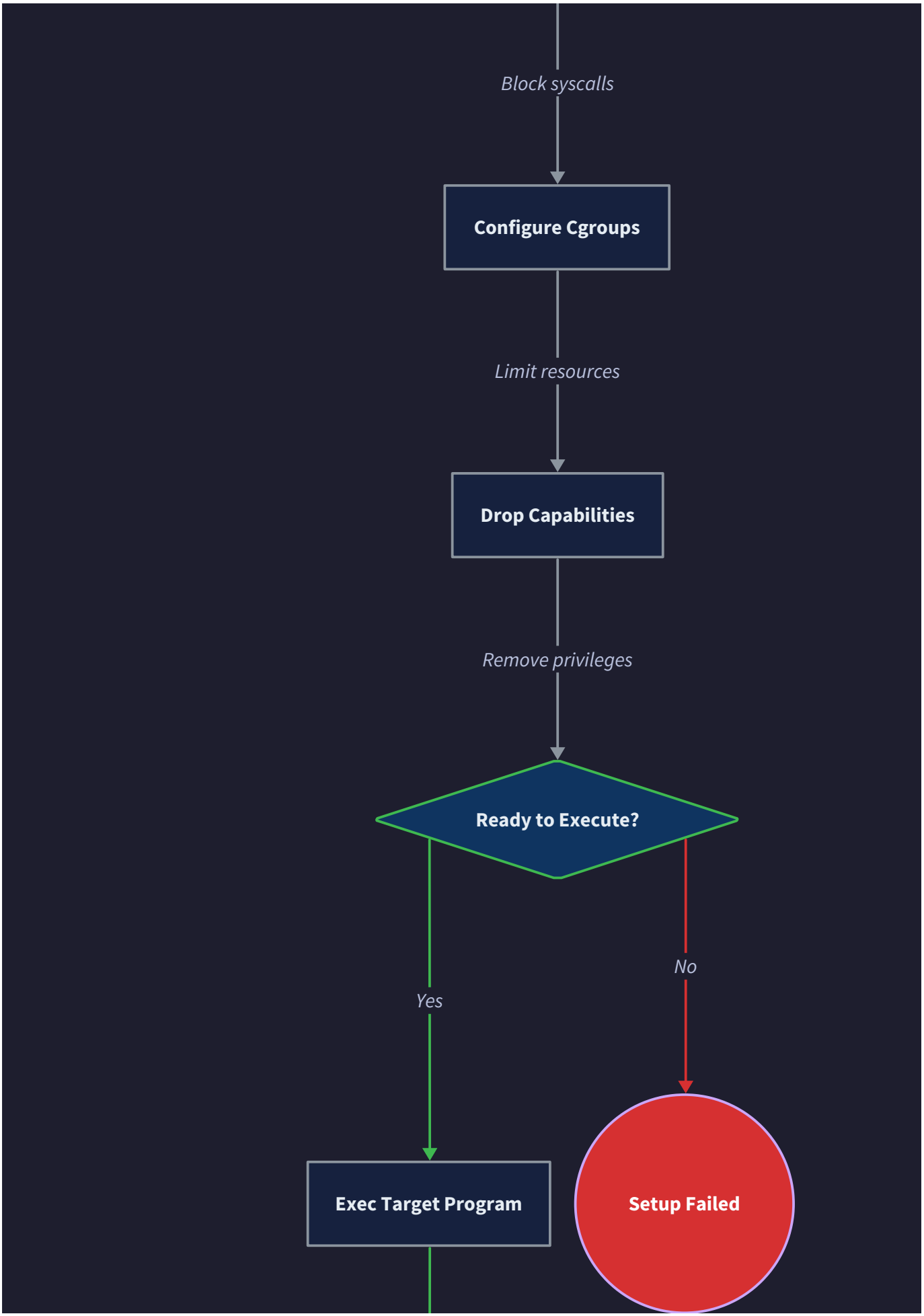
- The final product meets all security specifications before leaving the facility

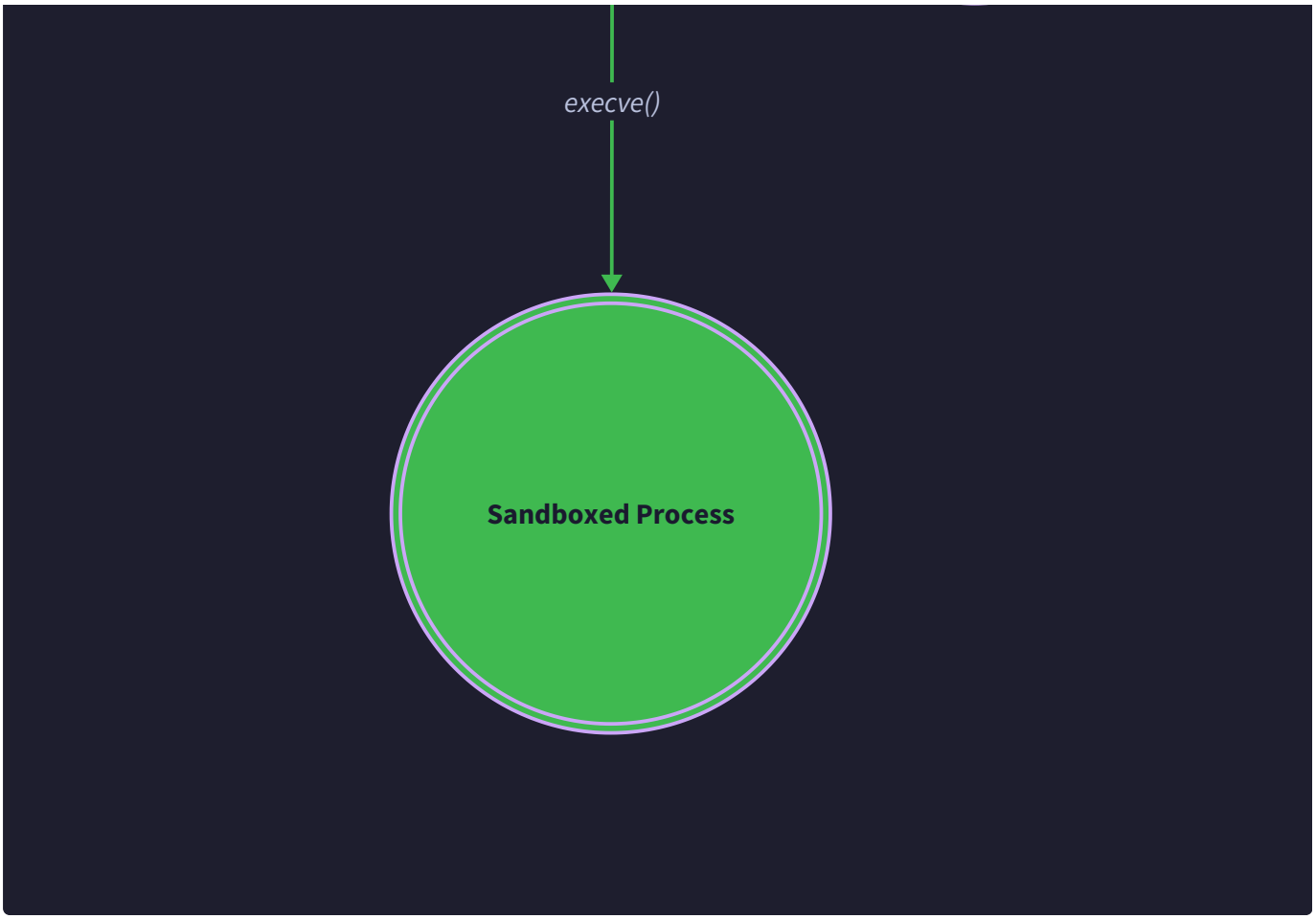
Just as an assembly line has quality control checkpoints, our sandbox orchestration includes validation steps that verify each security layer is properly configured before proceeding to the next.

Sandbox Creation Sequence

The sandbox creation follows a **strictly ordered sequence** where each step depends on the successful completion of its predecessors. This ordering is not arbitrary—it reflects the fundamental dependencies between Linux security mechanisms and the need to establish isolation before applying restrictions.


Sandbox Creation Sequence
Start
Fork Process
Child process
Create Namespaces
PID, NET, IPC, UTS, MNT
Set up Filesystem
chroot, bind mounts
Apply Seccomp Filter


Block syscalls
Configure Cgroups
Limit resources
Drop Capabilities
Remove privileges
Ready to Execute?
Yes
No
Exec Target Program
Setup Failed



The creation sequence must handle both the **happy path** (successful creation) and multiple **failure scenarios** where partial setup must be cleanly reversed. The orchestration component maintains detailed state about what has been configured, enabling precise cleanup even when failures occur in the middle of the sequence.

Step-by-Step Creation Algorithm

The sandbox creation follows this precise algorithm, where each step represents a **point of no return** that requires specific cleanup if subsequent steps fail:

1. **Privilege Verification**: The orchestration component first verifies that the calling process has sufficient privileges (typically root or specific capabilities) to perform all required operations. This early check prevents partial setup followed by privilege-related failures.
2. **Configuration Validation**: The `sandbox_config_t` structure is validated for consistency and completeness. This includes verifying that file paths exist, resource limits are reasonable, and the syscall whitelist contains required system calls for basic operation.
3. **State Initialization**: A new `sandbox_state_t` structure is allocated and initialized to track all resources that will need cleanup. This includes namespace file descriptors, cgroup paths, and process identifiers.
4. **Initial Process Fork**: The orchestration performs the first `fork()` operation to create the child process that will become the sandboxed process. The parent retains the ability to configure the child's environment before it begins execution.
5. **Namespace Creation**: The child process creates all required namespaces using `clone()` or `unshare()` system calls. This establishes the fundamental isolation boundaries that all subsequent security layers will rely upon.

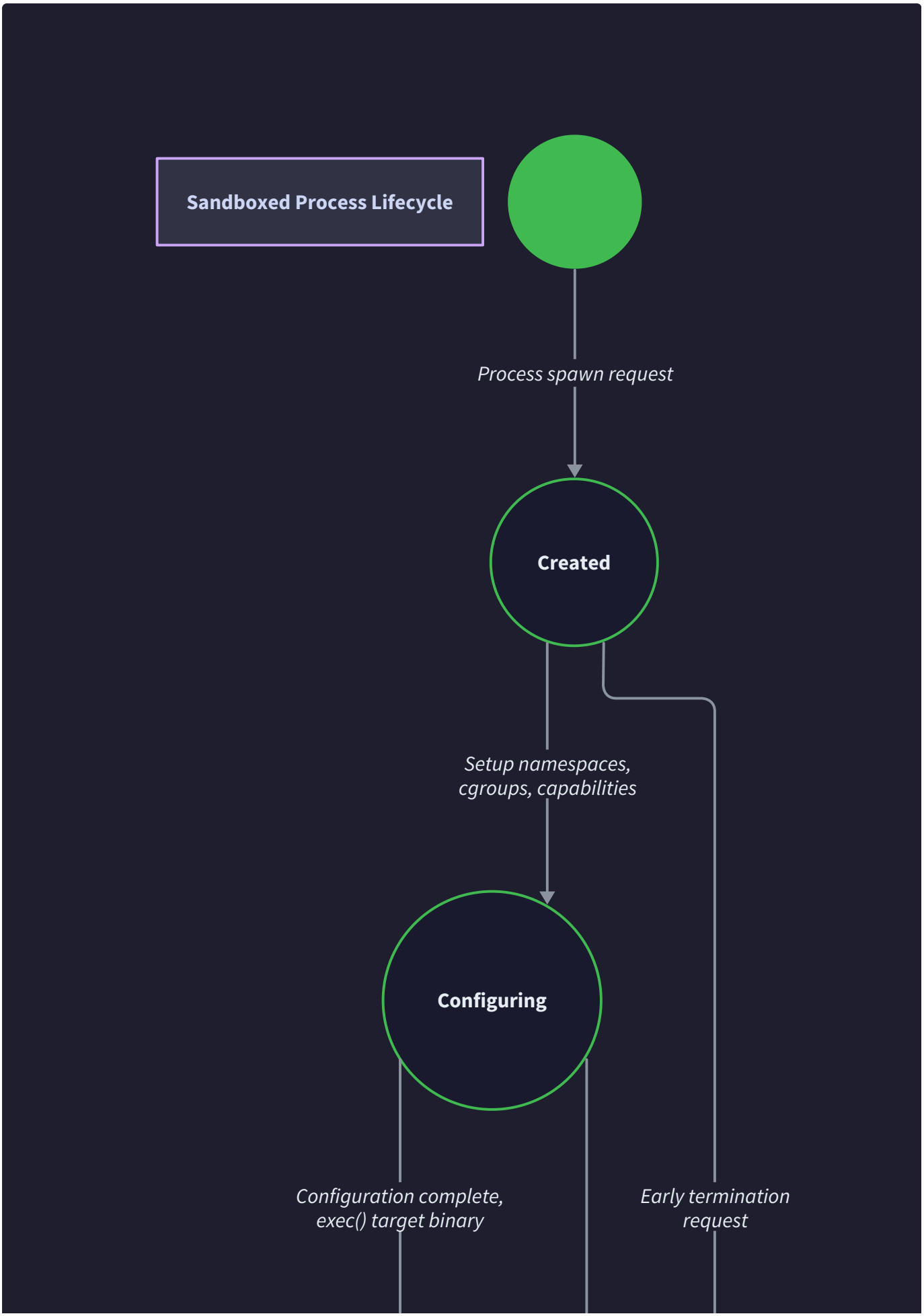
6. **User Namespace Setup**: If user namespaces are enabled, the orchestration configures UID and GID mappings to provide privilege separation. This step must occur before certain other namespace operations that require mapped privileges.
7. **Filesystem Isolation**: The filesystem isolation component creates the minimal root filesystem and performs `chroot()` or `pivot_root()` to establish the new root directory. This physically limits what files the sandboxed process can access.
8. **Resource Limit Application**: The cgroup component creates the necessary cgroup hierarchy and applies memory, CPU, and I/O limits. The sandboxed process is added to the appropriate cgroups before it begins execution.
9. **System Call Filter Installation**: The seccomp-BPF filter is compiled and installed, restricting which system calls the sandboxed process can make. This step occurs late in the sequence because the sandbox setup itself requires many privileged system calls.
10. **Capability Dropping**: All unnecessary Linux capabilities are dropped, and the `PR_SET_NO_NEW_PRIVS` flag is set to prevent privilege escalation. This represents the final hardening step before execution.
11. **Program Execution**: The `exec()` system call replaces the sandboxed process image with the target program. At this point, all security restrictions are active and cannot be circumvented.

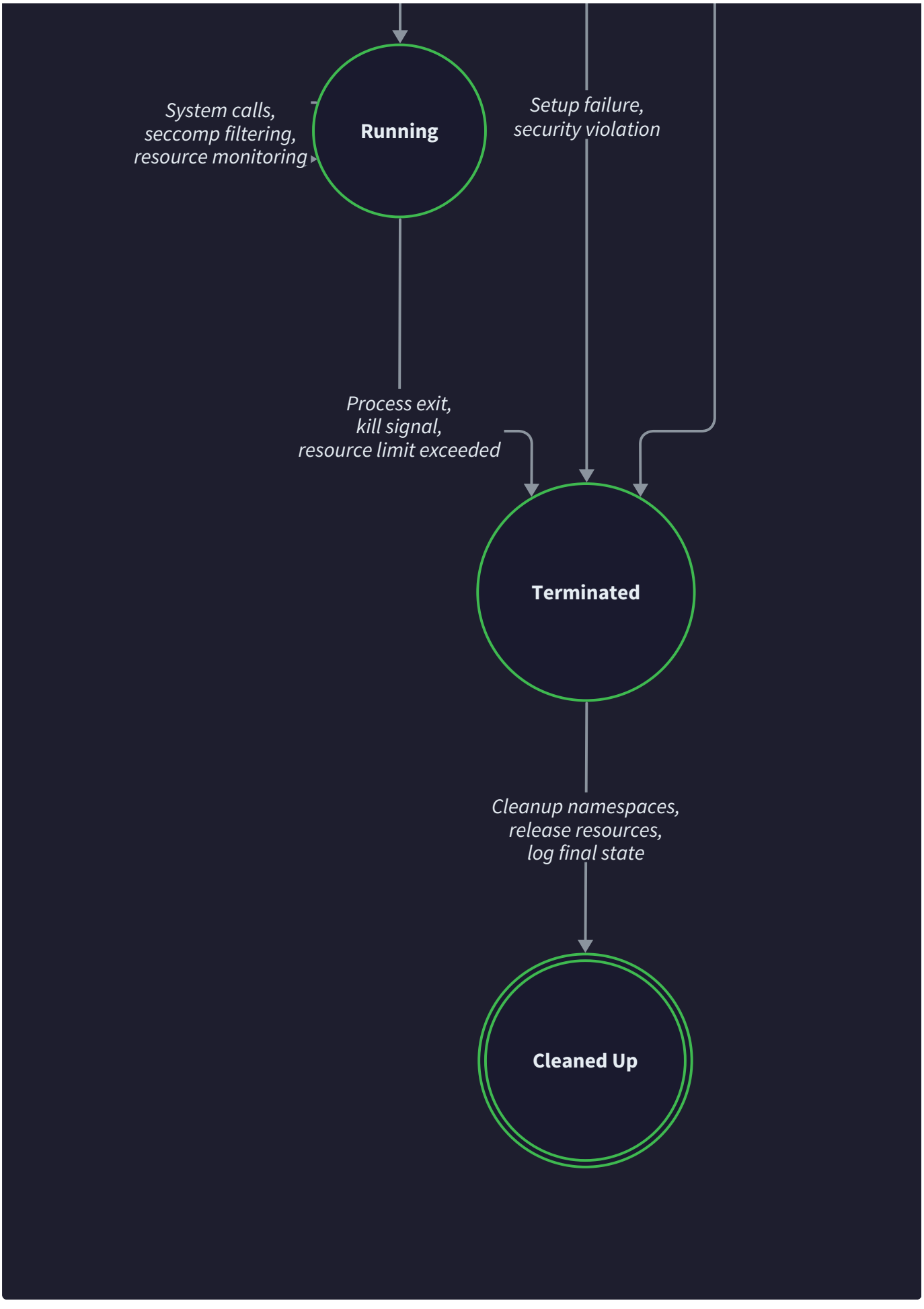
> **Architecture Decision: Fork-Then-Configure vs Configure-Then-Fork**
>
> - **Context**: We need to decide whether to fork first and then configure the child, or configure the current process and then fork
> - **Options Considered**:
>   - Fork early and configure child from parent
>   - Configure current process then fork configured child
>   - Use clone() with all flags at once
> - **Decision**: Fork early and configure child from parent
> - **Rationale**: This approach provides better error handling (parent can clean up failed child), maintains parent privileges for configuration, and allows parent to monitor child throughout setup
> - **Consequences**: Requires IPC between parent and child during setup, but provides superior error recovery and monitoring capabilities

Creation Step	Prerequisites	Failure Recovery	Critical Resources
Privilege Verification	Process must be root or have CAP_SYS_ADMIN	No cleanup needed	None
Configuration Validation	Valid config structure	Free allocated memory	Configuration object
State Initialization	Valid configuration	Free state structure	State tracking object
Process Fork	Sufficient process limits	None (fork failed)	Child process
Namespace Creation	Forked child process	Terminate child process	Namespace file descriptors
Filesystem Isolation	Mount namespace active	Unmount all sandbox mounts	Mount points, directories
Resource Limits	Cgroup controllers available	Remove from cgroups, delete hierarchy	Cgroup directories
System Call Filtering	No conflicting seccomp filters	Cannot be undone	BPF program
Capability Dropping	Process in target user namespace	Cannot be undone	Process capabilities
Program Execution	All security layers configured	Terminate sandbox process	Running program

Sandboxed Process Lifecycle

The sandboxed process experiences a **well-defined lifecycle** with clear state transitions and specific responsibilities for the orchestration component at each stage. Understanding this lifecycle is crucial for proper resource management and error handling.


Sandboxed Process Lifecycle
Process spawn request
Created
Setup namespaces, cgroups, capabilities
Configuring
Configuration complete, exec() target binary
Early termination request


Running
System calls,
seccomp filtering,
resource monitoring
Setup failure,
security violation
Process exit,
kill signal,
resource limit exceeded
Terminated
Cleanup namespaces,
release resources,
log final state
Cleaned Up

The lifecycle management ensures that the orchestration component maintains complete visibility into the sandbox state

and can perform appropriate cleanup at any stage. The parent process (orchestrator) remains responsible for monitoring and resource cleanup even after the sandboxed process begins execution.

Lifecycle State Machine

The sandbox process transitions through distinct states, each with specific entry conditions and possible exit transitions:

Current State	Entry Condition	Possible Events	Next State	Actions Taken
Created	Successful fork()	Configuration starts	Configuring	Begin security layer setup
Created	Fork failure	Process cleanup	Terminated	Release partial resources
Configuring	Security layer setup in progress	Setup completes successfully	Running	Execute target program
Configuring	Security layer setup failure	Cleanup begins	Terminating	Reverse completed setup steps
Running	Target program executing	Program exits normally	Terminating	Begin resource cleanup
Running	Security violation detected	Kill signal sent	Terminating	Force process termination
Running	Resource limit exceeded	OOM or SIGKILL	Terminating	Clean up cgroup resources
Terminating	Cleanup in progress	Cleanup completes	Cleaned Up	All resources released
Terminating	Cleanup failure	Cleanup retry	Terminating	Attempt cleanup again
Cleaned Up	All resources released	Process reaped	Destroyed	Remove from parent tracking

State Transition Details

Each state transition represents a **significant change** in the sandbox's security posture and resource allocation. The orchestration component must carefully manage these transitions to prevent resource leaks and security violations.

Created → Configuring: This transition occurs when the child process begins applying security restrictions. The parent process monitors the child's configuration progress and maintains the ability to terminate the child if configuration fails or hangs.

Configuring → Running: The most critical transition, where the sandbox becomes fully isolated and begins executing untrusted code. Once this transition completes, the sandboxed process can no longer be reconfigured—all security policies are immutable.

Running → Terminating: This transition can be triggered by several events: normal program completion, security policy violations, resource limit exceeded, or external termination requests. The orchestration component must distinguish between these causes for proper logging and response.

Terminating → Cleaned Up: The cleanup phase requires careful reversal of the creation sequence, removing the process from cgroups, cleaning up mount points, and closing namespace file descriptors. The orchestration component tracks which resources need cleanup to avoid double-free errors.

> The critical insight is that once a sandbox enters the Running state, security policies become immutable. This is why the configuration phase must be completely successful before allowing program execution.

Communication Channels

The orchestration of a complex sandbox requires **reliable communication channels** between the parent supervisor and the child sandbox process. These channels serve multiple purposes: coordinating the configuration sequence, monitoring sandbox health, and providing a controlled interface for legitimate interactions with the outside world.

Parent-Child Coordination Channel

During the configuration phase, the parent and child processes must coordinate their activities to ensure proper timing and error handling. This coordination uses a **bidirectional pipe** established before the fork operation.

Message Type	Direction	Purpose	Payload	Response Required
CONFIG_START	Parent → Child	Begin configuration sequence	Configuration step ID	ACK or ERROR
NAMESPACE_READY	Child → Parent	Namespaces created successfully	Namespace FDs	CONTINUE
FILESYSTEM_READY	Child → Parent	Filesystem isolation complete	Mount point list	CONTINUE
SECCOMP_READY	Child → Parent	System call filter active	Filter program hash	CONTINUE
LIMITS_APPLIED	Child → Parent	Resource limits configured	Cgroup path	CONTINUE
CAPS_DROPPED	Child → Parent	Capabilities dropped	Remaining capabilities	EXEC_APPROVED
CONFIG_ERROR	Child → Parent	Configuration step failed	Error code and details	TERMINATE
READY_TO_EXEC	Child → Parent	All security layers active	Final status	EXEC_APPROVED

The coordination protocol ensures that the child process never proceeds to the next configuration step without explicit approval from the parent. This **synchronous handshake** prevents race conditions and ensures proper error handling.

Runtime Monitoring Channel

After the sandbox begins executing the target program, the communication needs change from configuration coordination to **runtime monitoring**. The parent process needs visibility into sandbox behavior without compromising isolation.

The monitoring channel uses **signal-based communication** combined with `procfs` inspection to gather information about the sandbox state. Direct IPC channels are intentionally limited to prevent sandbox escape vectors.

Monitoring Method	Information Gathered	Update Frequency	Security Implications
Process Status	PID, PPID, state, exit status	On-demand	Read-only access to /proc/[pid]/stat
Resource Usage	Memory, CPU, I/O consumption	Periodic polling	Read cgroup accounting files
Security Events	Seccomp violations, capability attempts	Real-time signals	Kernel delivers SIGSYS or SIGKILL
Filesystem Activity	File access patterns, mount changes	Audit subsystem	Optional integration with auditd
Network Activity	Connection attempts, data transfer	Network namespace inspection	Limited by namespace isolation

Controlled External Interface

Some sandbox applications require **limited interaction** with external resources while maintaining security isolation. The orchestration component can provide controlled interfaces that allow specific operations without compromising the sandbox integrity.

> **Architecture Decision: Signal-Based vs Socket-Based Communication**
>
> - **Context**: Need to monitor sandbox without providing escape vectors
> - **Options Considered**:
>   - Unix domain sockets for bidirectional communication
>   - Signal-based notification with procfs inspection
>   - Shared memory with lock-free protocols
> - **Decision**: Signal-based communication with procfs inspection
> - **Rationale**: Signals cannot be used for privilege escalation, procfs provides read-only monitoring, and this approach minimizes attack surface compared to bidirectional sockets
> - **Consequences**: Limited bandwidth for monitoring data, requires polling for some information, but provides excellent security isolation

Common Pitfalls in Communication Design

⚠️ **Pitfall: Bidirectional Sockets in Sandbox** Creating Unix domain sockets between parent and child provides a potential escape vector if the sandbox can influence socket operations. Attackers may exploit socket buffer overflows, file descriptor passing, or credential passing mechanisms to escape isolation. **Fix**: Use unidirectional communication where possible, with signals from child to parent and procfs inspection by parent.

⚠️ **Pitfall: Blocking on Child Communication** If the parent process blocks indefinitely waiting for child responses, a misbehaving sandbox can cause denial of service by simply not responding to coordination messages. **Fix**: Implement timeouts for all coordination operations and terminate non-responsive sandboxes.

⚠️ **Pitfall: Information Leakage Through Error Messages** Detailed error messages sent from sandbox to parent may leak information about the host system's configuration or internal state. **Fix**: Use standardized error codes instead of free-form error messages, and sanitize any diagnostic information before transmission.

Integration Flow Example

To illustrate how all components work together, consider a complete sandbox creation scenario for executing a simple untrusted Python script:

Initial Setup: The orchestration component receives a request to execute `untrusted_script.py` with a memory limit of 64MB, CPU limit of 10%, and network access disabled.

1. **Configuration Validation**: The orchestrator verifies that Python interpreter exists in the minimal rootfs, memory limit is reasonable, and the syscall whitelist includes Python's requirements (`read`, `write`, `mmap`, etc.).
2. **Process Creation**: The orchestrator forks, creating parent (PID 1000) and child (PID 1001) processes. The child will become the sandbox.
3. **Namespace Setup**: Child 1001 creates PID, mount, network, and UTS namespaces. From child's perspective, it becomes PID 1 in its own PID namespace.
4. **Filesystem Preparation**: Child 1001 performs `chroot()` to `/sandbox/python-minimal/`, which contains only Python interpreter, required libraries, and the target script. Host filesystem becomes completely inaccessible.
5. **Cgroup Assignment**: Parent 1000 creates cgroup `/sys/fs/cgroup/sandbox-1001/` and configures memory.limit_in_bytes=67108864 (64MB) and cpu.cfs_quota_us=10000 (10% CPU). Child 1001 is added to this cgroup.
6. **Seccomp Filter**: Child 1001 installs BPF filter allowing only: `read`, `write`, `mmap`, `munmap`, `exit_group`, `rt_sigaction`, and `brk`. All other syscalls will terminate the process.
7. **Capability Dropping**: Child 1001 drops all capabilities and sets `PR_SET_NO_NEW_PRIVS=1`, preventing any privilege escalation.
8. **Program Execution**: Child 1001 executes `execve("/usr/bin/python3", ["python3", "untrusted_script.py"])`. The sandbox is now fully active.

Runtime Monitoring: Parent 1000 monitors child 1001 through `/proc/1001/stat` and cgroup accounting files. If the script attempts forbidden operations (network access, excessive memory), appropriate limits trigger termination.

Error Handling and Recovery Patterns

The orchestration component must handle failures at every stage of sandbox creation and execution. Each failure mode requires specific recovery actions to prevent resource leaks and maintain system stability.

Failure Mode	Detection Method	Immediate Action	Recovery Steps	Prevention Strategy
Configuration Validation Failure	Invalid parameters in `sandbox_config_t`	Return error to caller	Free configuration memory	Validate all parameters before creation
Fork Failure	`fork()` returns -1	Log error and return failure	No cleanup needed	Check process limits before fork
Namespace Creation Failure	`clone()` or `unshare()` fails	Terminate child process	Kill child, reap zombie	Verify kernel namespace support
Filesystem Setup Failure	`chroot()` or mount operations fail	Begin cleanup sequence	Unmount all sandbox mounts	Verify filesystem permissions and space
Cgroup Creation Failure	Cannot write to cgroup files	Remove partial cgroup hierarchy	Delete created cgroup directories	Check cgroup controller availability
Seccomp Installation Failure	`prctl(PR_SET_SECCOMP)` fails	Terminate sandbox immediately	Cannot recover - kill process	Validate BPF program before installation
Capability Dropping Failure	`capset()` fails or verification fails	Terminate sandbox immediately	Cannot recover - kill process	Verify capability support before dropping
Runtime Security Violation	SIGSYS signal or limit exceeded	Kill sandbox process	Clean up all resources	Monitor and tune security policies
Parent Process Death	Sandbox becomes orphaned	Sandbox continues until exit	Init system reaps orphan	Use process monitoring and restart

Resource Lifecycle Management

The orchestration component maintains complete responsibility for **resource lifecycle management**, ensuring that every allocated resource is properly cleaned up regardless of how the sandbox terminates.

Resource Tracking Tables

The `sandbox_state_t` structure maintains comprehensive tracking of all resources requiring cleanup:

Resource Type	Tracking Method	Cleanup Function	Cleanup Order	Failure Recovery
Child Process	`child_pid` field	`kill()` + `waitpid()`	First (prevent further resource use)	Retry with SIGKILL
Namespace FDs	`namespace_fd[]` array	`close()` file descriptors	After process termination	Close remaining FDs
Mount Points	Linked list in filesystem state	`umount()` in reverse order	Before namespace cleanup	Lazy unmount with MNT_DETACH
Cgroup Hierarchy	`cgroup_path` and process list	Remove processes, delete directories	After process termination	Force process removal
Temporary Directories	Path tracking in filesystem state	`rmdir()` recursive removal	After unmounting	Force removal with elevated privileges
Lock Files	File handle tracking	`unlink()` and `close()`	Before directory removal	Continue cleanup despite failures

Cleanup Ordering Dependencies

The cleanup sequence must respect **dependency relationships** between resources. Attempting cleanup in the wrong order can cause failures or leave resources in inconsistent states.

> The fundamental principle is to clean up resources in the reverse order of their creation, with process termination occurring first to prevent interference with cleanup operations.

1. **Process Termination**: Send SIGTERM to sandbox process, wait briefly, then SIGKILL if needed. Reap zombie process with `waitpid()`.
2. **Cgroup Cleanup**: Remove sandbox process from all cgroups, delete cgroup hierarchy directories. This must occur after process termination to avoid conflicts.
3. **Filesystem Cleanup**: Unmount all sandbox mount points in reverse order of creation. Use lazy unmount (`MNT_DETACH`) if normal unmount fails.
4. **Namespace Cleanup**: Close namespace file descriptors. The kernel automatically cleans up namespaces when all references are closed.
5. **Temporary Resource Cleanup**: Remove temporary directories, close temporary files, release any other allocated resources.

Implementation Guidance

The orchestration component serves as the integration point for all sandbox security mechanisms. This implementation guidance provides the concrete code structure needed to coordinate namespace isolation, filesystem restrictions, system call filtering, resource limits, and capability dropping into a cohesive sandbox system.

Technology Recommendations

Component	Simple Option	Advanced Option
Process Management	`fork()` + `execve()` with manual setup	`clone()` with custom namespace flags
IPC Coordination	Anonymous pipes (`pipe()`)	Unix domain sockets with SCM_RIGHTS
Error Handling	Return codes with `errno`	Structured error types with context
Configuration	Static compiled configuration	Dynamic JSON/YAML configuration files
Logging	`syslog()` integration	Structured logging with log levels
Resource Tracking	Static arrays and linked lists	Dynamic hash tables with cleanup callbacks

Recommended File Structure

```
project-root/
  cmd/sandbox/
    main.c                    ← CLI entry point and argument parsing
  src/orchestration/
    orchestrator.c            ← Main orchestration logic (THIS COMPONENT)
    orchestrator.h            ← Public interface definitions
    lifecycle.c               ← Process lifecycle management
    communication.c           ← Parent-child communication
    cleanup.c                 ← Resource cleanup and error recovery
    orchestrator_test.c       ← Integration tests
  src/namespace/              ← Namespace isolation component (Milestone 1)
    namespace.c
    namespace.h
  src/filesystem/             ← Filesystem isolation component (Milestone 2)
    filesystem.c
    filesystem.h
  src/seccomp/               ← System call filtering component (Milestone 3)
    seccomp.c
    seccomp.h
  src/cgroup/                ← Resource limits component (Milestone 4)
    cgroup.c
    cgroup.h
  src/capability/            ← Capability management component (Milestone 5)
    capability.c
    capability.h
  src/common/
    config.c                  ← Configuration management
    error.c                   ← Error handling utilities
    utils.c                   ← Common utilities
```

Infrastructure Starter Code

Configuration Management Infrastructure (`src/common/config.c`):

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include "config.h"

// Complete configuration management - learner uses this directly
sandbox_config_t* sandbox_config_init() {
    sandbox_config_t *config = calloc(1, sizeof(sandbox_config_t));
    if (!config) return NULL;

    // Set secure defaults
    config->memory_limit = 64 * 1024 * 1024;  // 64MB default
    config->cpu_percent = 10;                  // 10% CPU default
    config->enable_network = 0;               // Network disabled by default
    config->chroot_path = strdup("/tmp/sandbox-rootfs");

    // Default safe syscall whitelist
    config->allowed_syscalls = calloc(32, sizeof(char*));
    config->allowed_syscalls[0] = strdup("read");
    config->allowed_syscalls[1] = strdup("write");
    config->allowed_syscalls[2] = strdup("exit_group");
    config->allowed_syscalls[3] = strdup("rt_sigaction");
    config->allowed_syscalls[4] = strdup("mmap");
    config->allowed_syscalls[5] = strdup("munmap");
    config->allowed_syscalls[6] = strdup("brk");

    return config;
}
```

```c
void sandbox_config_free(sandbox_config_t *config) {
    if (!config) return;

    free(config->program_path);
    free(config->chroot_path);

    if (config->program_args) {
        for (int i = 0; config->program_args[i]; i++) {
            free(config->program_args[i]);
        }
        free(config->program_args);
    }

    if (config->allowed_syscalls) {
        for (int i = 0; config->allowed_syscalls[i]; i++) {
            free(config->allowed_syscalls[i]);
        }
        free(config->allowed_syscalls);
    }

    free(config);
}

int validate_config(const sandbox_config_t *config) {
    if (!config || !config->program_path) return 0;
    if (access(config->program_path, X_OK) != 0) return 0;
    if (config->memory_limit < 1024 * 1024) return 0;  // Minimum 1MB
    if (config->cpu_percent > 100) return 0;
    return 1;
}
```

Error Handling Infrastructure (`src/common/error.c`):

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <syslog.h>
#include "error.h"

// Complete error handling system - learner uses this directly
void sandbox_error(const char *message, int error_code) {
    char full_message[512];

    if (error_code == 0) {
        snprintf(full_message, sizeof(full_message), "SANDBOX ERROR: %s", message);
    } else {
        snprintf(full_message, sizeof(full_message),
                "SANDBOX ERROR: %s (errno=%d: %s)",
                message, error_code, strerror(error_code));
    }

    // Log to syslog for system monitoring
    syslog(LOG_ERR, "%s", full_message);

    // Also print to stderr for debugging
    fprintf(stderr, "%s\n", full_message);
}

int check_root_privileges() {
    if (geteuid() != 0) {
        sandbox_error("Sandbox requires root privileges", 0);
        return 0;
    }
```

```
    return 1;
}

const char* sandbox_error_string(int error_code) {
    switch (error_code) {
        case SANDBOX_SUCCESS: return "Success";
        case SANDBOX_ERROR_CONFIG: return "Configuration error";
        case SANDBOX_ERROR_PRIVILEGE: return "Insufficient privileges";
        case SANDBOX_ERROR_NAMESPACE: return "Namespace creation failed";
        case SANDBOX_ERROR_FILESYSTEM: return "Filesystem isolation failed";
        case SANDBOX_ERROR_SECCOMP: return "Seccomp filter installation failed";
        case SANDBOX_ERROR_CGROUP: return "Cgroup setup failed";
        case SANDBOX_ERROR_CAPABILITY: return "Capability management failed";
        case SANDBOX_ERROR_EXEC: return "Program execution failed";
        default: return "Unknown error";
    }
}
```

Core Orchestration Logic Skeleton

Main Orchestrator (`src/orchestration/orchestrator.c`):

```c
#include <sys/wait.h>

#include <sys/prctl.h>

#include <unistd.h>

#include <signal.h>

#include "orchestrator.h"

#include "../namespace/namespace.h"

#include "../filesystem/filesystem.h"

#include "../seccomp/seccomp.h"

#include "../cgroup/cgroup.h"

#include "../capability/capability.h"


sandbox_state_t* sandbox_state_init() {

    // TODO 1: Allocate sandbox_state_t structure using calloc()

    // TODO 2: Initialize all file descriptors to -1 (invalid)

    // TODO 3: Set child_pid to -1 (no child yet)

    // TODO 4: Set cleanup_needed to 0 (nothing to clean up yet)

    // TODO 5: Initialize namespace_fd array to all -1 values

    // Hint: Use calloc() to zero-initialize, then set FDs to -1 explicitly

}


int create_sandbox(sandbox_config_t *config, sandbox_state_t *state) {

    // TODO 1: Validate configuration using validate_config()

    // TODO 2: Check root privileges using check_root_privileges()

    // TODO 3: Create coordination pipes for parent-child communication

    // TODO 4: Fork child process and store PID in state->child_pid

    // TODO 5: In child: call setup_sandbox_child() with config and pipe FDs

    // TODO 6: In parent: call monitor_sandbox_creation() with state and pipe FDs

    // TODO 7: If any step fails, call cleanup_sandbox() and return error code

    // Hint: Use pipe() before fork(), close unused ends in each process

}
```

```c
static int setup_sandbox_child(sandbox_config_t *config, int control_pipe) {
    namespace_state_t *ns_state = NULL;
    fs_isolation_state_t *fs_state = NULL;
    cgroup_state_t *cgroup_state = NULL;

    // TODO 1: Create namespaces using create_namespaces() from namespace component
    // TODO 2: Send NAMESPACE_READY message to parent via control_pipe
    // TODO 3: Setup filesystem isolation using setup_filesystem_isolation()
    // TODO 4: Send FILESYSTEM_READY message to parent via control_pipe
    // TODO 5: Install seccomp filter using install_seccomp_filter()
    // TODO 6: Send SECCOMP_READY message to parent via control_pipe
    // TODO 7: Drop capabilities using drop_capabilities()
    // TODO 8: Send CAPS_DROPPED message to parent via control_pipe
    // TODO 9: Execute target program using execve()
    // Hint: Wait for parent approval after each message before proceeding
}

static int monitor_sandbox_creation(sandbox_state_t *state, int control_pipe) {
    char message[256];
    int step = 0;

    // TODO 1: Read NAMESPACE_READY message from control_pipe
    // TODO 2: Verify namespace creation succeeded, send CONTINUE or TERMINATE
    // TODO 3: Read FILESYSTEM_READY message from control_pipe
    // TODO 4: Verify filesystem setup succeeded, send CONTINUE or TERMINATE
    // TODO 5: Read SECCOMP_READY message from control_pipe
    // TODO 6: Verify seccomp installation succeeded, send CONTINUE or TERMINATE
    // TODO 7: Read CAPS_DROPPED message from control_pipe
    // TODO 8: Verify capability dropping succeeded, send EXEC_APPROVED
    // TODO 9: Monitor child process for successful exec or failure
```

```
    // Hint: Use select() or poll() with timeout to avoid blocking forever
}

void cleanup_sandbox(sandbox_state_t *state) {
    if (!state) return;

    // TODO 1: If child_pid > 0, send SIGTERM and wait briefly
    // TODO 2: If child still running, send SIGKILL and wait
    // TODO 3: Reap zombie child process using waitpid()
    // TODO 4: Clean up cgroup hierarchy if cgroup_path is set
    // TODO 5: Clean up filesystem mounts if any were created
    // TODO 6: Close namespace file descriptors in namespace_fd array
    // TODO 7: Free any allocated memory in state structure
    // Hint: Clean up in reverse order of creation, handle failures gracefully
}
```

Process Lifecycle Management (`src/orchestration/lifecycle.c`):

```c
#include <sys/wait.h>

#include <signal.h>

#include <time.h>

#include "orchestrator.h"


typedef enum {

    SANDBOX_STATE_CREATED,

    SANDBOX_STATE_CONFIGURING,

    SANDBOX_STATE_RUNNING,

    SANDBOX_STATE_TERMINATING,

    SANDBOX_STATE_CLEANED_UP

} sandbox_lifecycle_state_t;


int monitor_sandbox_lifecycle(sandbox_state_t *state) {

    sandbox_lifecycle_state_t current_state = SANDBOX_STATE_CREATED;

    int status;



    // TODO 1: Monitor child process state changes using waitpid(WNOHANG)

    // TODO 2: Check for configuration completion signals from child

    // TODO 3: Transition to RUNNING state when child exec() succeeds

    // TODO 4: Handle SIGCHLD to detect child termination

    // TODO 5: Transition through TERMINATING to CLEANED_UP states

    // TODO 6: Update state structure with current lifecycle state

    // Hint: Use a state machine with clear transition conditions

}


int terminate_sandbox(sandbox_state_t *state, int timeout_seconds) {

    // TODO 1: Send SIGTERM to child process if still running

    // TODO 2: Start timeout timer for graceful shutdown

    // TODO 3: Wait for child to exit voluntarily within timeout

    // TODO 4: If timeout expires, send SIGKILL for forceful termination
```

```
    // TODO 5: Reap zombie child process using waitpid()
    // TODO 6: Update state to reflect process termination
    // Hint: Use alarm() or timer_create() for timeout handling
}
```

Language-Specific Implementation Hints

C-Specific Considerations:

- Use `signalfd()` or `sigaction()` to handle SIGCHLD for child process monitoring
- Implement proper error handling with `errno` checking after system calls
- Use `strace -f` to debug parent-child coordination and system call sequences
- Be careful with file descriptor inheritance across `fork()` and `exec()`
- Use `prctl(PR_SET_PDEATHSIG, SIGKILL)` in child to ensure cleanup if parent dies

Memory Management:

- Always initialize pointers to NULL in structures
- Use `calloc()` instead of `malloc()` to zero-initialize memory
- Implement proper cleanup functions for each component state structure
- Use valgrind to detect memory leaks during testing

Error Handling Patterns:

```c
int result = some_operation();
if (result != SANDBOX_SUCCESS) {
    sandbox_error("Operation failed", errno);
    cleanup_partial_state();
    return result;
}
```

Milestone Checkpoints

Checkpoint 1: Basic Orchestration After implementing the basic orchestration framework:

- Command: `gcc -o sandbox src/orchestration/*.c src/common/*.c && sudo ./sandbox --program=/bin/echo --args="hello world"`
- Expected: Program should fork, coordinate setup, and execute echo successfully
- Verify: Check that parent receives coordination messages and child executes correctly
- Debug: Use `strace -f` to trace system calls in both parent and child processes

Checkpoint 2: Security Layer Integration After integrating all security components:

- Command: `sudo ./sandbox --program=/bin/ls --memory-limit=32M --cpu-limit=5 --no-network`

- Expected: Program runs with all security restrictions active, lists directory contents
- Verify: Check `/proc/[pid]/status` for namespace isolation, cgroup limits applied
- Debug: Examine `/sys/fs/cgroup/sandbox-*/` for resource limit configuration

Checkpoint 3: Error Handling and Cleanup After implementing complete error handling:

- Command: `sudo ./sandbox --program=/nonexistent/program` (should fail gracefully)
- Expected: Clean error message, all resources cleaned up properly
- Verify: No orphaned processes, no leftover cgroup directories, no mounted filesystems
- Debug: Check for resource leaks using `ps`, `mount`, and `lsof` commands

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Child process hangs during setup	Deadlock in parent-child coordination	Use `strace -f` to see where each process blocks	Add timeouts to all coordination operations
Sandbox escapes isolation	Security layers applied in wrong order	Check sequence of security operations	Follow strict ordering: namespaces → filesystem → seccomp → cgroups → capabilities
Resource cleanup fails	Resources cleaned up in wrong order	Examine cleanup sequence and dependencies	Clean up in reverse order of creation
Parent process becomes zombie	Child process outlives parent	Check process hierarchy with `ps axjf`	Use `prctl(PR_SET_PDEATHSIG)` in child
Permission denied during setup	Insufficient privileges for operation	Check capabilities and user ID at each step	Verify root privileges before starting, check namespace mappings

Error Handling and Edge Cases

> **Milestone(s):** This section applies to all milestones (1-5), providing error handling strategies and recovery mechanisms for namespace isolation, filesystem restrictions, system call filtering, resource limits, and capability management.

The robustness of a process sandbox depends not just on its security mechanisms, but on how gracefully it handles failures, resource exhaustion, and edge cases. A production sandbox system must anticipate and recover from dozens of failure modes while maintaining security boundaries even during error conditions. This section explores the systematic approach to error detection, classification, recovery, and cleanup that ensures the sandbox remains both secure and reliable.

Mental Model: The Emergency Response System

Think of sandbox error handling like a hospital emergency response system. Just as hospitals have protocols for every type of medical emergency—cardiac arrest, trauma, poisoning—a sandbox must have specific responses for every type of system failure. The emergency response system has three key characteristics that apply directly to sandbox error handling:

Triage and Classification: Emergency rooms immediately classify patients by severity and type of condition to determine treatment priority. Similarly, the sandbox must classify errors by severity (fatal vs recoverable) and component (namespace vs cgroup vs seccomp) to determine the appropriate response strategy.

Containment and Stabilization: Medical teams first stabilize the patient and prevent the condition from worsening before attempting curative treatment. The sandbox must first contain failures—preventing resource leaks, zombie processes, or security breaches—before attempting recovery.

Documentation and Learning: Hospitals maintain detailed incident reports to improve future responses. The sandbox logs detailed error information not just for debugging, but to identify patterns that indicate systemic issues requiring architectural changes.

This mental model emphasizes that error handling is not just "catch and log"—it requires systematic preparation, immediate containment, and long-term improvement based on failure analysis.

Common Failure Modes

The process sandbox encounters failures across multiple dimensions: system call failures during setup, resource exhaustion during execution, and permission errors during cleanup. Understanding these failure modes and their cascading effects is crucial for building robust error detection and recovery mechanisms.

System Call Failures

System call failures represent the most fundamental class of errors in sandbox creation, as every security mechanism depends on kernel operations that can fail for various reasons.

Namespace Creation Failures occur when the kernel cannot create the requested namespace isolation. The `clone()` system call with namespace flags can fail due to kernel limits, insufficient memory, or disabled namespace support. More subtly, the kernel enforces per-user limits on namespace creation—the `CLONE_NEWUSER` flag fails when a user exceeds their maximum namespace count, typically 32 per user. Network namespace creation fails when the system has exhausted network namespace slots or when network controllers are not available.

Mount Operation Failures represent another critical failure mode during filesystem isolation setup. The `mount()` system call can fail due to incorrect filesystem types, missing mount points, insufficient permissions, or kernel module loading failures. Bind mounts fail when source paths do not exist or when mount propagation settings conflict. The `pivot_root()` operation fails when the old and new root filesystems are on the same mount point or when processes still hold references to the old root.

Cgroup Management Failures occur when the kernel cannot create cgroup hierarchies or configure resource limits. Cgroup creation fails when hierarchies are full, when controllers are not available, or when the cgroup filesystem is not mounted. Setting resource limits fails when values exceed system maximums or when processes are already consuming more resources than the new limits allow.

Failure Mode	Detection Method	Immediate Effect	Recovery Strategy
`clone()` ENOMEM	Return value check	Fork failure	Release memory, retry with fewer namespaces
`clone()` EINVAL	Return value check	Invalid namespace flags	Detect kernel support, fallback configuration
`mount()` ENOENT	Return value check	Missing mount point	Create directory, retry mount
`mount()` EPERM	Return value check	Permission denied	Check capabilities, escalate privileges
`pivot_root()` EINVAL	Return value check	Invalid filesystem state	Verify mount setup, fix mount points
Cgroup creation ENOSPC	Return value check	No cgroup slots	Clean up unused cgroups, retry
Resource limit EINVAL	Return value check	Invalid limit value	Validate limits, use system defaults

Seccomp Filter Installation Failures can occur when BPF programs are malformed, when the process already has incompatible seccomp filters, or when the kernel does not support required BPF features. The `prctl(PR_SET_SECCOMP)` call fails with `EINVAL` when the BPF program contains invalid instructions or when filter size exceeds kernel limits.

> **Design Insight**: System call failures often cascade—a failed namespace creation can prevent subsequent mount operations, which then causes seccomp setup to operate in an unexpected environment. The error handling strategy must account for these dependencies and either fail fast or adapt the configuration to work with partial setup.

Resource Exhaustion Attacks

Resource exhaustion represents a class of failures where the sandboxed process or the sandbox infrastructure itself consumes excessive system resources, potentially affecting host system stability.

Memory Exhaustion can occur at multiple levels within the sandbox system. The sandboxed process can exceed its cgroup memory limit, triggering the OOM (Out of Memory) killer within the cgroup. More dangerously, the sandbox creation process itself can exhaust system memory when creating large numbers of concurrent sandboxes or when namespace setup requires significant kernel memory allocation.

Process ID Exhaustion happens when sandboxed processes spawn children faster than the PID cgroup limit allows, or when the system-wide PID limit is approached. This can cause legitimate processes to fail to fork, creating denial-of-service conditions.

File Descriptor Exhaustion occurs when sandboxed processes open files or sockets faster than they close them, eventually hitting per-process or system-wide file descriptor limits. This is particularly problematic for network-enabled sandboxes that can open many socket connections.

Disk Space Exhaustion can happen when sandboxes write to tmpfs mounts faster than space is reclaimed, or when log files from sandbox activity grow without bounds. Tmpfs mounts backed by system memory can trigger OOM conditions when they grow too large.

Resource Type	Exhaustion Trigger	Detection Method	Recovery Action
Memory	Process exceeds cgroup limit	Cgroup memory.events counter	Terminate process group
Memory	System OOM condition	Monitor /proc/meminfo	Suspend new sandbox creation
PIDs	Process fork() fails	Cgroup pids.events counter	Kill oldest child processes
File Descriptors	open() returns EMFILE	Monitor /proc/pid/fd count	Close unused descriptors
Disk Space	write() returns ENOSPC	Monitor tmpfs usage	Clean temporary files
Network Sockets	socket() returns EMFILE	Network namespace socket count	Close idle connections

> **Critical Insight**: Resource exhaustion often occurs gradually and then suddenly. Monitoring systems must track resource usage trends and implement proactive throttling before hard limits trigger emergency termination. Reactive-only approaches result in poor user experience and potential security issues.

Permission and Privilege Errors

Permission errors occur when the sandbox system lacks the necessary privileges to perform required operations, or when privilege transitions fail during sandbox setup or cleanup.

Insufficient Privileges During Setup is the most common cause of sandbox creation failure. Creating namespaces requires `CAP_SYS_ADMIN`, mounting filesystems requires `CAP_SYS_ADMIN`, and managing cgroups typically requires root privileges. The error handling must detect these permission failures and either escalate privileges through appropriate mechanisms (setuid binaries, systemd services) or gracefully degrade functionality.

Privilege Dropping Failures can occur when the capability dropping sequence is performed in the wrong order or when processes attempt to drop capabilities they do not possess. The interaction between user namespace mappings and capability inheritance creates complex failure modes where capabilities appear available but cannot be effectively used or dropped.

Cleanup Permission Failures happen when the privileged sandbox orchestrator process lacks permissions to clean up resources created by child processes. This is particularly problematic with user namespaces, where processes running as different mapped UIDs create resources that the parent process cannot remove.

Permission Error	Root Cause	Detection	Resolution
`EPERM` on `clone()`	Missing `CAP_SYS_ADMIN`	System call return code	Check effective capabilities
`EPERM` on `mount()`	Missing mount capability	System call return code	Verify filesystem permissions
`EPERM` on cgroup write	Incorrect cgroup ownership	File write failure	Fix cgroup permissions
`EPERM` on capability drop	Invalid capability state	`prctl()` failure	Verify current capability set
`EPERM` on cleanup	Mismatched ownership	File operation failure	Use appropriate cleanup privileges

Cleanup and Resource Management

Proper cleanup and resource management in a process sandbox requires systematic tracking of all allocated resources and careful attention to cleanup ordering, especially when dealing with interdependent Linux kernel resources like

namespaces, mounts, and cgroups.

Resource Tracking and Lifecycle Management

The sandbox system must maintain comprehensive tracking of all resources allocated during sandbox creation to ensure proper cleanup even when failures occur partway through setup. This requires a **resource tracking registry** that records each successful resource allocation and provides cleanup handlers for each resource type.

Namespace File Descriptor Management involves tracking namespace file descriptors obtained through `/proc/self/ns/` entries. These file descriptors pin namespaces in memory even after the creating process exits, preventing kernel cleanup until the descriptors are closed. The tracking system must record namespace file descriptors in the `sandbox_state_t` structure and ensure they are closed during cleanup.

Mount Point Tracking requires maintaining a list of all mount operations performed during filesystem isolation setup. Mounts must be unmounted in reverse order of creation to handle dependencies correctly—child mounts must be unmounted before parent mounts to avoid `EBUSY` errors. The system tracks mount points in a linked list through the `mount_entry` structure, allowing cleanup traversal in LIFO order.

Cgroup Hierarchy Management involves tracking created cgroup directories and ensuring that all processes are terminated before attempting to remove cgroup nodes. Cgroups cannot be removed while they contain processes, so cleanup must first terminate all processes in the cgroup, wait for process exit, and then remove the hierarchy from leaf nodes toward the root.

Resource Type	Allocation Tracking	Cleanup Dependencies	Cleanup Method
Namespace FDs	`namespace_state_t.pid_ns_fd` array	Close before process exit	`close()` system call
Mount Points	`mount_entry` linked list	Unmount children first	`umount2()` with `MNT_DETACH`
Cgroup Dirs	`cgroup_state_t.cgroup_path`	Kill processes first	Remove directory tree
Process IDs	`cgroup_state_t.managed_pids` array	Send SIGTERM, then SIGKILL	`kill()` system call
Temporary Files	Track in global file list	Remove before parent dirs	`unlink()` system call
Device Nodes	Track created devices	Remove after unmounting	`unlink()` special files

Cleanup Ordering and Dependencies

The order of resource cleanup operations is critical for avoiding cleanup failures and ensuring that interdependent resources are released in the correct sequence. The cleanup sequence must handle the dependency relationships between different resource types while being robust to partial failures.

Process Termination First represents the fundamental principle of sandbox cleanup. All sandboxed processes must be terminated before attempting to clean up other resources, as running processes can hold references to namespaces, mount points, and cgroup resources that prevent their removal. The termination sequence uses escalating signals: `SIGTERM` for graceful shutdown, followed by `SIGKILL` after a timeout period.

Mount Point Cleanup Ordering follows a specific hierarchy where bind mounts and overlay mounts are unmounted before their underlying filesystems. Special filesystems like `/proc`, `/dev`, and `/sys` within the sandbox namespace are unmounted before unmounting the root filesystem. The `umount2()` call with `MNT_DETACH` flag performs lazy unmounting that handles some ordering dependencies automatically.

Namespace Cleanup Dependencies require understanding that mount namespaces must be cleaned up before PID namespaces, and network namespaces can be cleaned up independently. Closing namespace file descriptors allows kernel garbage collection, but the kernel may delay actual cleanup until all references are released.

> **Decision: Cleanup Strategy**
>
> - **Context**: Resource cleanup can fail partially, leaving the system in an inconsistent state with leaked resources
> - **Options Considered**: Best-effort cleanup vs. mandatory cleanup vs. cleanup with retries
> - **Decision**: Implement cleanup with retries and best-effort fallback
> - **Rationale**: Some cleanup operations may temporarily fail due to timing issues (processes still exiting), but permanent failures should not prevent other resources from being cleaned up
> - **Consequences**: Enables robust cleanup that handles transient failures while preventing cascading cleanup failures

The cleanup algorithm follows this specific ordering:

1. **Signal Process Termination**: Send `SIGTERM` to all processes in the sandbox cgroup
2. **Wait with Timeout**: Wait up to 5 seconds for graceful process termination
3. **Force Kill**: Send `SIGKILL` to any remaining processes
4. **Wait for Process Exit**: Wait for all PIDs to be reaped by parent processes
5. **Unmount Filesystems**: Unmount all tracked mount points in reverse order
6. **Close Namespace FDs**: Close all namespace file descriptors to release references
7. **Remove Cgroup Hierarchy**: Remove cgroup directories from leaf to root
8. **Clean Temporary Files**: Remove temporary directories and files created during setup

Error Recovery During Cleanup

Cleanup operations can themselves fail, creating the possibility of resource leaks and requiring recovery strategies that handle partial cleanup failures gracefully.

Retry Logic for Transient Failures handles cleanup operations that fail due to timing issues, such as attempting to unmount filesystems while processes are still exiting. The retry mechanism uses exponential backoff with a maximum retry count to handle transient failures without infinite loops.

Best-Effort Cleanup for Permanent Failures ensures that permanent failures in one cleanup operation do not prevent other resources from being cleaned up. For example, if a mount point cannot be unmounted due to busy processes, the cleanup continues with other mount points and cgroup removal, logging the failure for administrative attention.

Resource Leak Detection and Reporting involves checking for successful completion of each cleanup operation and maintaining counters of leaked resources. This information is logged for monitoring systems and can trigger alerts when resource leaks exceed acceptable thresholds.

Cleanup Failure	Retry Strategy	Fallback Action	Leak Detection
Mount umount EBUSY	Retry 3 times with 1s delay	Log warning, continue	Check /proc/mounts
Cgroup rmdir ENOTEMPTY	Retry after killing processes	Mark for admin cleanup	Check cgroup.procs
Process kill ESRCH	No retry needed	Process already gone	Check /proc/pid
File unlink ENOENT	No retry needed	File already removed	No leak possible
Namespace close EBADF	No retry needed	FD already closed	No leak possible

⚠️ **Pitfall: Blocking on Failed Cleanup** A common mistake is making cleanup operations synchronous and blocking when they fail. This can cause the sandbox orchestrator to hang indefinitely if kernel resources are in inconsistent states. Instead, implement cleanup with timeouts and continue with other operations even if some cleanup steps fail. Use asynchronous cleanup monitoring to retry failed operations in the background.

Monitoring and Health Checks

Effective sandbox monitoring requires continuous observation of sandboxed process health, resource consumption patterns, and security boundary integrity. The monitoring system must detect both gradual degradation and sudden failures while providing actionable information for both automated recovery and human intervention.

Sandbox Health Detection

Sandbox health encompasses multiple dimensions: process liveness, security boundary integrity, resource consumption patterns, and communication channel functionality. Each dimension requires specific monitoring approaches and failure detection thresholds.

Process Liveness Monitoring tracks whether sandboxed processes are running, responsive, and behaving within expected parameters. This involves monitoring process existence through PID tracking, responsiveness through communication channel heartbeats, and behavioral patterns through system call monitoring. The monitoring system maintains process state in the `sandbox_lifecycle_state_t` enumeration and transitions between states based on observed behavior.

Security Boundary Integrity Checks verify that namespace isolation, filesystem restrictions, and capability limitations remain in effect throughout sandbox execution. These checks involve periodic verification that processes cannot access restricted resources, that namespace views remain isolated, and that seccomp filters continue blocking forbidden system calls.

Resource Consumption Pattern Analysis monitors resource usage trends to detect both gradual resource leaks and sudden consumption spikes that might indicate malicious behavior or process malfunction. This includes tracking memory usage growth rates, file descriptor accumulation, and process creation patterns.

Health Dimension	Monitoring Method	Check Frequency	Failure Threshold
Process Existence	Check /proc/pid	Every 1 second	Process not found
Process Responsiveness	Heartbeat ping	Every 5 seconds	No response in 15s
Memory Usage	Read cgroup memory.usage	Every 2 seconds	>90% of limit
CPU Usage	Read cgroup cpu.stat	Every 5 seconds	>100% sustained
File Descriptors	Count /proc/pid/fd entries	Every 10 seconds	>80% of limit
Network Activity	Monitor namespace traffic	Every 1 second	Unexpected connections
System Call Pattern	Audit log analysis	Continuous	Blocked syscall attempts

Misbehaving Process Detection

Detecting and responding to misbehaving sandboxed processes requires understanding both resource-based misbehavior (consuming excessive resources) and security-based misbehavior (attempting to violate sandbox boundaries).

Resource Abuse Detection identifies processes that consume resources in patterns inconsistent with their expected workload. This includes detecting memory leaks through monotonically increasing memory usage, CPU abuse through sustained high CPU utilization, and I/O abuse through excessive disk or network activity. The detection system uses sliding window analysis to distinguish between legitimate bursts and sustained abuse patterns.

Security Violation Detection monitors attempts to breach sandbox security boundaries, including seccomp filter violations (blocked system calls), capability escalation attempts, and filesystem access violations. These violations are typically logged by kernel security mechanisms, but the monitoring system must aggregate and analyze these logs to detect coordinated attacks or persistent probing behavior.

Communication Protocol Violations occur when sandboxed processes fail to follow expected communication patterns with the sandbox orchestrator. This includes sending malformed messages, failing to respond to control messages, or attempting to establish unauthorized communication channels.

> **Decision: Misbehavior Response Strategy**
>
> - **Context**: Misbehaving processes can affect both sandbox security and system stability, requiring immediate response
> - **Options Considered**: Immediate termination vs. graduated response vs. resource throttling
> - **Decision**: Implement graduated response with escalating severity
> - **Rationale**: Immediate termination may interrupt legitimate work during temporary resource spikes, while graduated response allows distinction between malicious behavior and temporary resource needs
> - **Consequences**: Requires more complex monitoring logic but provides better user experience and more precise security enforcement

The misbehavior detection algorithm uses a **three-tier response system**:

Tier 1: Warning and Throttling responds to minor resource violations or single security boundary probes by logging warnings and applying resource throttling. CPU throttling reduces the process's CPU quota, memory pressure applies swap pressure, and I/O throttling reduces bandwidth allocation.

Tier 2: Process Suspension responds to repeated violations or sustained resource abuse by suspending the offending process using `SIGSTOP`. This prevents further resource consumption while preserving process state for analysis or user intervention.

Tier 3: Process Termination responds to severe violations, persistent abuse after throttling, or security boundary breach attempts by terminating the entire sandbox process tree using `SIGKILL`.

Misbehavior Type	Detection Criteria	Tier 1 Response	Tier 2 Response	Tier 3 Response
Memory Growth	10% increase in 60s	Apply memory pressure	Suspend process	Kill on OOM
CPU Abuse	>95% CPU for 30s	Reduce CPU quota	Suspend process	Kill after 2 minutes
Fork Bomb	10+ processes in 5s	Block new forks	Suspend all processes	Kill process tree
File Descriptor Leak	FD count doubles	Close unused FDs	Suspend process	Kill and cleanup
Network Flooding	>1000 packets/sec	Apply bandwidth limit	Suspend process	Kill and block
Seccomp Violations	5+ blocked syscalls	Log and monitor	Suspend process	Kill on 10+ attempts

Automated Recovery and Escalation

The monitoring system must not only detect problems but also implement automated recovery strategies and escalate to human operators when automated responses are insufficient.

Automatic Restart Policies handle transient failures by restarting failed sandboxes according to configurable policies. The restart logic distinguishes between failures that warrant restart (resource exhaustion, temporary system issues) and failures that should not be retried (security violations, configuration errors). Restart attempts use exponential backoff to prevent resource thrashing.

Resource Adjustment and Healing responds to resource pressure by automatically adjusting resource limits within acceptable bounds. If a sandbox consistently approaches memory limits but otherwise behaves properly, the system can increase memory allocation up to configured maximums. Similarly, CPU quotas can be adjusted based on workload patterns.

Escalation Triggers and Notifications activate when automated responses are insufficient or when patterns suggest systemic issues requiring human attention. Escalation triggers include repeated sandbox failures, resource exhaustion at the host level, or security violation patterns that suggest coordinated attacks.

Recovery Scenario	Automatic Action	Escalation Trigger	Human Notification
Process Crash	Restart with same config	3 restarts in 5 minutes	Email alert
Memory Pressure	Increase memory limit by 25%	Host memory <10% free	Page operator
CPU Starvation	Increase CPU quota	Host load >8.0	Dashboard alert
Security Violations	Kill and restart with stricter limits	>10 violations/hour	Security team alert
Host Resource Exhaustion	Suspend new sandboxes	Cannot create new sandbox	Emergency page
Cleanup Failures	Retry cleanup with delays	>50 leaked resources	Operations alert

⚠️ **Pitfall: Alert Fatigue from Over-Monitoring** Monitoring systems often generate so many alerts that operators become desensitized to genuine problems. Design monitoring thresholds to minimize false positives by using trend analysis rather than absolute thresholds, implementing alert suppression during maintenance windows, and escalating only when multiple indicators suggest genuine problems. A single memory spike should not trigger alerts, but sustained growth combined with increased CPU usage and file descriptor accumulation should.

Implementation Guidance

The error handling and monitoring system bridges the gap between sandbox creation and production readiness, providing the infrastructure needed to detect, respond to, and recover from the wide variety of failure modes that occur in real deployments.

Technology Recommendations

Component	Simple Option	Advanced Option
Process Monitoring	Poll /proc filesystem	Use netlink process events
Resource Monitoring	Parse cgroup files	Use cgroup notification API
Log Aggregation	Write to syslog	Use structured logging with journald
Metrics Collection	Simple counters in memory	Prometheus metrics endpoint
Health Checks	Basic HTTP endpoint	Rich health check with component status
Error Reporting	Write to stderr	Structured error logs with correlation IDs

Recommended File Structure

The error handling system integrates across all sandbox components, requiring both centralized error handling infrastructure and component-specific error recovery logic.

```
sandbox/
├── src/
│   ├── error/
│   │   ├── error_handler.c        ← centralized error classification and logging
│   │   ├── cleanup_manager.c      ← resource tracking and cleanup orchestration
│   │   ├── retry_logic.c          ← exponential backoff and retry utilities
│   │   └── error_recovery.c       ← automatic recovery strategies
│   ├── monitoring/
│   │   ├── health_monitor.c       ← sandbox health checks and status tracking
│   │   ├── resource_monitor.c     ← resource usage monitoring and trend analysis
│   │   ├── process_monitor.c      ← process lifecycle and misbehavior detection
│   │   └── metrics_collector.c    ← metrics aggregation and reporting
│   ├── sandbox/
│   │   ├── sandbox_orchestrator.c ← integrates error handling with sandbox creation
│   │   └── lifecycle_manager.c    ← manages sandbox lifecycle state transitions
│   └── util/
│       ├── signal_utils.c         ← signal handling utilities for cleanup
│       └── proc_utils.c           ← /proc filesystem parsing utilities
├── include/
│   ├── error/
│   │   ├── error_codes.h          ← error code definitions and classifications
│   │   ├── cleanup_manager.h      ← resource tracking structures and APIs
│   │   └── error_recovery.h       ← recovery strategy function signatures
│   └── monitoring/
│       ├── health_monitor.h       ← health check structures and status definitions
│       └── resource_monitor.h     ← resource monitoring thresholds and limits
└── tests/
    ├── error/
    │   ├── test_error_injection.c  ← controlled failure testing
    │   └── test_cleanup_recovery.c ← cleanup and recovery testing
    └── monitoring/
        ├── test_health_checks.c    ← health monitoring validation
        └── test_misbehavior.c      ← misbehavior detection testing
```

Error Classification and Reporting Infrastructure

The error handling system requires comprehensive infrastructure for classifying, logging, and responding to different types of failures. This infrastructure provides the foundation for all component-specific error handling.

Complete Error Classification System:

```c
#include <errno.h>
#include <string.h>
#include <syslog.h>
#include <time.h>

// Error severity levels for classification and response
typedef enum {
    ERROR_SEVERITY_INFO,      // Informational, no action needed
    ERROR_SEVERITY_WARNING,   // Warning, monitoring required
    ERROR_SEVERITY_ERROR,     // Error, automatic recovery possible
    ERROR_SEVERITY_CRITICAL,  // Critical, immediate intervention required
    ERROR_SEVERITY_FATAL      // Fatal, sandbox termination required
} error_severity_t;

// Error categories for specialized handling
typedef enum {
    ERROR_CATEGORY_SYSTEM,    // System call and kernel errors
    ERROR_CATEGORY_RESOURCE,  // Resource exhaustion and limits
    ERROR_CATEGORY_SECURITY,  // Security boundary violations
    ERROR_CATEGORY_CONFIG,    // Configuration and setup errors
    ERROR_CATEGORY_NETWORK    // Network and communication errors
} error_category_t;

// Comprehensive error context structure
typedef struct {
    int error_code;              // Standard errno or custom error code
    error_severity_t severity;   // Error severity for response selection
    error_category_t category;   // Error category for specialized handling
    char component[64];          // Component that detected the error
    char operation[128];         // Operation that failed
    char description[256];       // Human-readable error description
```

```
    pid_t sandbox_pid;           // Associated sandbox process ID
    time_t timestamp;           // Error occurrence time
    char correlation_id[37];    // UUID for tracking related errors
} error_context_t;

// Global error reporting and logging function
void sandbox_error_report(error_context_t* ctx) {
    // TODO 1: Generate correlation ID if not provided
    // TODO 2: Log error to syslog with appropriate priority level
    // TODO 3: Update error metrics counters for monitoring
    // TODO 4: Trigger automatic recovery if severity level warrants it
    // TODO 5: Send notifications if error requires escalation
}

// Utility function for creating error contexts from system call failures
error_context_t* create_system_error(const char* component, const char* operation, int errno_value) {
    // TODO 1: Allocate and initialize error_context_t structure
    // TODO 2: Set error_code to errno_value
    // TODO 3: Classify severity based on errno value (ENOMEM=CRITICAL, EPERM=ERROR, etc.)
    // TODO 4: Set category to ERROR_CATEGORY_SYSTEM
    // TODO 5: Generate timestamp and correlation ID
    // Hint: Use strerror(errno_value) to populate description field
}
```

Resource Cleanup Infrastructure

The cleanup system provides centralized resource tracking and cleanup orchestration that handles the complex dependencies between different resource types.

Complete Cleanup Manager Implementation:

```c
#include <sys/queue.h>
#include <pthread.h>

// Resource types for cleanup tracking
typedef enum {
    RESOURCE_TYPE_NAMESPACE_FD,
    RESOURCE_TYPE_MOUNT_POINT,
    RESOURCE_TYPE_CGROUP_DIR,
    RESOURCE_TYPE_PROCESS_ID,
    RESOURCE_TYPE_TEMP_FILE,
    RESOURCE_TYPE_DEVICE_NODE
} resource_type_t;

// Generic resource cleanup entry
typedef struct cleanup_entry {
    resource_type_t type;           // Type of resource for cleanup dispatch
    void* resource_data;            // Resource-specific data for cleanup
    int (*cleanup_func)(void*);     // Cleanup function for this resource type
    char description[128];          // Human-readable resource description
    TAILQ_ENTRY(cleanup_entry) entries;  // Queue linkage for cleanup ordering
} cleanup_entry_t;

// Cleanup manager state with resource tracking
typedef struct {
    TAILQ_HEAD(cleanup_head, cleanup_entry) cleanup_queue;  // LIFO cleanup queue
    pthread_mutex_t queue_mutex;    // Thread safety for cleanup queue
    int total_resources;            // Counter of tracked resources
    int cleanup_failures;          // Counter of cleanup failures
    int cleanup_in_progress;       // Flag to prevent concurrent cleanup
} cleanup_manager_t;

// Global cleanup manager instance
```

```c
static cleanup_manager_t* global_cleanup_manager = NULL;


// Initialize cleanup manager

cleanup_manager_t* cleanup_manager_init(void) {

    // TODO 1: Allocate cleanup_manager_t structure

    // TODO 2: Initialize cleanup queue using TAILQ_INIT

    // TODO 3: Initialize mutex using pthread_mutex_init

    // TODO 4: Set counters and flags to zero

    // TODO 5: Install signal handlers for emergency cleanup

    // Hint: Use atexit() to ensure cleanup runs even on normal exit

}


// Add resource to cleanup tracking

int track_resource_for_cleanup(resource_type_t type, void* data, int (*cleanup_func)(void*), const
char* description) {

    // TODO 1: Allocate cleanup_entry_t structure

    // TODO 2: Copy resource data and description

    // TODO 3: Lock cleanup queue mutex

    // TODO 4: Insert entry at head of queue using TAILQ_INSERT_HEAD

    // TODO 5: Increment total_resources counter and unlock mutex

    // Hint: Resources added later are cleaned up first (LIFO order)

}


// Execute all cleanup operations with error handling

void cleanup_all_resources(void) {

    // TODO 1: Lock cleanup queue mutex and set cleanup_in_progress flag

    // TODO 2: Iterate through cleanup queue using TAILQ_FOREACH

    // TODO 3: Call cleanup_func for each entry with retry logic

    // TODO 4: Log failures but continue with remaining cleanup operations

    // TODO 5: Remove successful entries from queue, keep failed ones

    // TODO 6: Report final cleanup statistics and unlock mutex

    // Hint: Use TAILQ_FOREACH_SAFE for safe iteration during removal
```

```
}
```

Health Monitoring Core Logic

The health monitoring system provides continuous observation of sandbox state and automated detection of problems requiring intervention.

Complete Health Monitor Framework:

```c
#include <sys/time.h>
#include <pthread.h>

// Health check result structure
typedef struct {
    char check_name[64];        // Name of health check performed
    int status;                 // 0=healthy, 1=warning, 2=critical
    char message[256];          // Status message with details
    double response_time_ms;    // Time taken to perform check
    time_t last_check_time;    // Timestamp of last check
} health_check_result_t;

// Comprehensive sandbox health state
typedef struct {
    sandbox_state_t* sandbox;          // Associated sandbox state
    health_check_result_t process_health;    // Process liveness check
    health_check_result_t resource_health;   // Resource usage check
    health_check_result_t security_health;   // Security boundary check
    health_check_result_t network_health;    // Network isolation check
    int overall_health_score;          // Aggregate health score (0-100)
    time_t last_health_update;        // Timestamp of last health update
    pthread_t monitor_thread;         // Background monitoring thread
    int monitoring_active;            // Flag to control monitoring loop
} sandbox_health_t;

// Initialize health monitoring for a sandbox
sandbox_health_t* health_monitor_init(sandbox_state_t* sandbox) {
    // TODO 1: Allocate sandbox_health_t structure
    // TODO 2: Initialize all health check results with default values
    // TODO 3: Set overall_health_score to 100 (perfect health)
    // TODO 4: Create background monitoring thread
```

```
    // TODO 5: Start periodic health checks with configurable interval
    // Hint: Use pthread_create to start monitor_thread_main function
}

// Perform comprehensive process health check
int check_process_health(sandbox_state_t* sandbox, health_check_result_t* result) {
    // TODO 1: Check if sandbox child process still exists using kill(pid, 0)
    // TODO 2: Verify process is in expected cgroup by reading cgroup.procs
    // TODO 3: Check process memory usage against limits
    // TODO 4: Verify process has not created excessive child processes
    // TODO 5: Update result structure with findings and response time
    // Hint: Use gettimeofday before and after checks to measure response_time_ms
}

// Main monitoring thread function
void* monitor_thread_main(void* arg) {
    // TODO 1: Cast arg to sandbox_health_t* and extract monitoring parameters
    // TODO 2: Loop while monitoring_active flag is set
    // TODO 3: Perform all health checks with appropriate intervals
    // TODO 4: Update overall_health_score based on individual check results
    // TODO 5: Trigger alerts if health score drops below thresholds
    // TODO 6: Sleep for monitoring interval before next check cycle
    // Hint: Use different intervals for different checks (process=1s, resource=5s)
}
```

Milestone Checkpoints

After implementing error handling and monitoring infrastructure, verify that the system properly handles failure scenarios and maintains operational visibility.

Error Handling Verification:

- Run `./test_error_injection` to verify that injected system call failures are properly classified and logged
- Expected: Error messages with appropriate severity levels and correlation IDs
- Check: `journalctl | grep sandbox_error` should show structured error logs
- Verify: Partial sandbox setup failures trigger proper resource cleanup

Cleanup System Validation:

- Create sandbox with multiple namespaces and mount points, then terminate process
- Expected: All resources cleaned up in correct order without errors
- Check: `lsns` should not show orphaned namespaces after cleanup
- Verify: `/proc/mounts` should not contain sandbox mount points after cleanup

Health Monitoring Validation:

- Start sandbox with memory limit, then run memory-intensive workload
- Expected: Health monitor detects resource pressure and triggers appropriate responses
- Check: Monitor logs should show resource usage warnings before hitting limits
- Verify: Misbehaving process gets terminated when exceeding resource thresholds

Debugging Tips for Common Issues:

Symptom	Likely Cause	How to Diagnose	Fix
Cleanup hangs indefinitely	Mount point busy with processes	`lsof +D /sandbox/root` to find blocking processes	Kill processes before unmounting
Resource leaks after failures	Missing cleanup entry registration	Check cleanup queue size before/after operations	Add track_resource_for_cleanup calls
False positive health alerts	Monitoring thresholds too sensitive	Review health check logs and adjust thresholds	Increase warning thresholds, use trend analysis
Missed security violations	Insufficient seccomp audit logging	Enable seccomp audit: `echo 1 > /proc/sys/kernel/seccomp/actions_logged`	Configure audit log monitoring
Cleanup permission errors	Privilege dropping before cleanup	Check effective UID during cleanup operations	Preserve cleanup privileges until after resource cleanup

Testing Strategy

> **Milestone(s):** This section applies to all milestones (1-5), providing comprehensive testing approaches for namespace isolation, filesystem restrictions, system call filtering, resource limits, and capability management

Testing a process sandbox requires a fundamentally different approach than testing typical application code. The sandbox's primary purpose is to **contain and restrict** rather than to provide functionality, which means our tests must verify that operations fail correctly rather than succeed. Think of testing a process sandbox like **quality assurance for a maximum-security prison** - we're not just checking that the doors lock, but that every possible escape route has been blocked, every privilege has been properly restricted, and every resource limit is actively enforced.

The challenge lies in the fact that security mechanisms often work by **preventing observable behavior** rather than producing it. A successful seccomp filter kills processes before they can report what happened. A properly configured cgroup silently throttles CPU usage. A correctly dropped capability makes system calls fail with cryptic error codes. Our testing strategy must account for these "negative" behaviors while ensuring that legitimate operations still function correctly within the restricted environment.

Furthermore, the defense-in-depth architecture means that **multiple security layers interact** in complex ways. A test might pass when only namespaces are active but fail when seccomp filtering is added, or resource limits might behave differently when capabilities have been dropped. Our testing approach must validate not just individual components but their interactions and the emergent security properties of the complete system.

Unit Testing Approach

Mental Model: The Security Component Laboratory

Think of unit testing security components like **running controlled experiments in separate laboratory chambers**. Each test isolates a single security mechanism (namespaces, seccomp, cgroups, capabilities) and subjects it to precisely controlled conditions to verify it behaves correctly. Just as a laboratory experiment controls all variables except the one being studied, our unit tests activate only the security layer under test while mocking or bypassing the others.

The key insight is that security components have **dual responsibilities**: they must block malicious operations while allowing legitimate ones. Every unit test must therefore verify both the "deny" path and the "allow" path. A namespace test that only checks isolation without verifying that permitted operations still work is incomplete and may hide bugs that break legitimate functionality.

Component-Level Test Structure

Each security component requires a distinct testing approach tailored to its specific isolation mechanism and failure modes. The following table outlines the core test categories for each component:

Component	Primary Test Focus	Success Criteria	Failure Criteria	Special Considerations
Namespace Isolation	Resource view separation	Isolated process sees different PIDs, mounts, network	Host system resources remain visible	Requires root privileges, cleanup of namespaces
Filesystem Isolation	Filesystem access restriction	Only sandbox root and permitted paths accessible	Host filesystem paths remain accessible	Requires mount/unmount operations, dependency tracking
Seccomp Filtering	System call restriction	Allowed syscalls succeed, forbidden syscalls killed	Forbidden syscalls succeed or wrong error code	Architecture-specific syscall numbers, BPF compilation
Resource Limits	Resource consumption control	Limits enforced when exceeded	Unlimited resource consumption possible	Time-dependent behavior, requires resource pressure
Capability Management	Privilege restriction	Privileged operations fail after dropping	Privileged operations succeed after dropping	Order-dependent operations, ambient capability interactions

Namespace Isolation Unit Tests

Namespace testing focuses on verifying that each namespace type creates the expected isolated view of system resources. The fundamental challenge is that namespace isolation is only visible from inside the namespace, requiring tests to fork child processes and examine their environment.

The namespace test suite must verify isolation for each namespace type:

1. **PID Namespace Tests**: Fork a child process in a new PID namespace and verify it sees itself as PID 1, cannot see host processes in `/proc`, and can only signal processes within its namespace.
2. **Mount Namespace Tests**: Create a new mount namespace, perform mount operations, and verify they are not visible from the host namespace while confirming the child can access its own mounts.
3. **Network Namespace Tests**: Create a network namespace and verify the child process has no network interfaces except loopback, cannot connect to external services, and cannot see host network configuration.
4. **UTS Namespace Tests**: Modify hostname within a UTS namespace and confirm changes are isolated from the host system while remaining visible within the namespace.
5. **User Namespace Tests**: Map user IDs within a user namespace and verify the mapping is enforced for file system operations and process ownership while maintaining isolation from host user management.

Each test requires careful orchestration of parent-child communication to verify isolation properties:

Test Phase	Parent Process Actions	Child Process Actions	Verification Method
Setup	Create namespace with `create_namespaces()`	Enter namespace via `setns()`	Check namespace inode numbers
Isolation	Modify host resources	Attempt to observe host changes	Verify child sees unchanged state
Containment	Monitor host state	Perform operations in namespace	Verify host remains unaffected
Cleanup	Clean up namespace file descriptors	Exit normally	Verify namespace is destroyed

Filesystem Isolation Unit Tests

Filesystem isolation testing must verify that the sandbox's filesystem view is both complete enough to function and restricted enough to prevent access to sensitive host resources. The tests must account for the complex interactions between chroot/pivot_root operations, mount namespaces, and bind mount restrictions.

The core filesystem isolation tests include:

1. **Root Filesystem Restriction**: Verify that processes cannot access paths outside the sandbox root, cannot traverse to parent directories, and cannot follow symlinks that escape the sandbox.
2. **Essential Directory Population**: Confirm that `/proc`, `/dev`, and `/sys` are properly mounted with appropriate restrictions, essential device files are accessible, and system information is available but filtered.
3. **Mount Point Management**: Test that temporary filesystems are correctly mounted, bind mounts are properly restricted, and all mounts are tracked for cleanup.

4. **Dependency Resolution**: Verify that all required shared libraries are available within the sandbox, binary executables can load and run, and no missing dependencies prevent legitimate operations.

The filesystem test approach requires careful setup of test environments:

Test Environment	Purpose	Setup Requirements	Cleanup Requirements
Minimal Root	Test basic chroot functionality	Create directory tree, copy essential binaries	Remove temporary directory tree
Library Test Root	Test dependency resolution	Copy full library dependencies	Clean up library cache, temporary files
Device Node Test	Test `/dev` population	Create device nodes, set permissions	Remove created devices, restore permissions
Mount Test Environment	Test mount management	Prepare test mount points	Unmount all test mounts, remove directories

Seccomp Filtering Unit Tests

Seccomp filter testing requires verifying that BPF programs correctly identify and block forbidden system calls while allowing permitted ones. The challenge is that seccomp violations terminate the process, requiring careful test orchestration to capture and verify the termination behavior.

The seccomp test methodology involves:

1. **BPF Program Generation**: Test that syscall whitelists generate correct BPF instruction sequences, handle architecture-specific syscall numbers properly, and produce valid filter programs that load without errors.
2. **Allowed Syscall Verification**: Install the filter and verify that whitelisted syscalls complete successfully, return expected values, and do not trigger filter violations.
3. **Forbidden Syscall Detection**: Attempt forbidden syscalls and verify they result in process termination, appropriate exit codes, and no side effects on system state.
4. **Argument-Level Filtering**: Test filters that examine syscall arguments, verify that parameter-based restrictions work correctly, and confirm that argument validation prevents bypass attempts.

The seccomp testing approach requires managing child processes that may be killed by the filter:

Test Scenario	Parent Actions	Child Actions	Expected Outcome
Whitelist Compliance	Fork child, wait for exit	Install filter, call allowed syscalls	Child exits normally
Blacklist Violation	Fork child, expect SIGKILL	Install filter, call forbidden syscall	Child killed by kernel
Argument Filtering	Fork child, monitor exit code	Install filter, call syscall with bad args	Child killed or errno returned
Filter Bypass Attempt	Monitor system state	Try various bypass techniques	All attempts fail, no system impact

Resource Limit Unit Tests

Cgroup resource limit testing must verify that limits are correctly applied and enforced under resource pressure. These tests require generating controlled resource consumption to trigger limit enforcement and observing the system's response.

The resource limit test categories include:

1. **Memory Limit Enforcement**: Allocate memory beyond the configured limit and verify the process is terminated by the OOM killer, memory usage is capped at the limit, and other processes remain unaffected.
2. **CPU Limit Enforcement**: Generate CPU load and verify that usage is throttled to the configured percentage, other processes receive fair scheduling, and limits are maintained over time.
3. **Process Count Limits**: Fork processes beyond the PID limit and verify new forks fail with appropriate error codes, existing processes continue running, and the limit is enforced consistently.
4. **I/O Bandwidth Limits**: Generate disk I/O load and verify throughput is capped at configured rates, I/O operations are throttled appropriately, and limits apply to all file operations.

Resource limit testing requires careful coordination of resource consumption:

Resource Type	Load Generation Method	Measurement Approach	Verification Criteria
Memory	Allocate increasing buffer sizes	Monitor `/proc/*/status` RSS	Allocation fails or process killed at limit
CPU	Busy loop with configurable intensity	Monitor CPU usage via cgroup stats	Usage throttled to configured percentage
Process Count	Fork bomb with controlled rate	Count processes in cgroup	Fork fails when limit reached
I/O Bandwidth	Large file read/write operations	Monitor I/O rates via iostat	Throughput capped at limit

Capability Management Unit Tests

Capability testing verifies that Linux capabilities are correctly dropped and that privilege escalation is prevented. The tests must account for the complex interactions between different capability sets and the various mechanisms for capability inheritance.

The capability test framework must verify:

1. **Capability Dropping**: Confirm that specified capabilities are removed from all capability sets, remaining capabilities are preserved correctly, and the bounding set prevents re-acquisition of dropped capabilities.
2. **Privilege Operation Blocking**: Attempt operations that require dropped capabilities and verify they fail with permission denied errors, no privileged side effects occur, and error handling is appropriate.
3. **No New Privileges Enforcement**: Verify that `PR_SET_NO_NEW_PRIVS` prevents privilege escalation through exec, setuid binaries do not gain privileges, and capability inheritance is blocked correctly.
4. **User/Group Transition**: Test that user and group changes occur in the correct order, capabilities are appropriately adjusted during transitions, and the final privilege state matches expectations.

The capability testing approach requires privilege verification at multiple stages:

Test Stage	Capability Check	Operation Test	Success Criteria
Initial State	List all current capabilities	Verify privileged operations work	Full capability set available
After Dropping	Verify specific caps removed	Confirm dropped operations fail	Only whitelisted capabilities remain
After No-New-Privs	Check flag is set	Attempt privilege escalation	All escalation attempts fail
After User Change	Verify final capability state	Test remaining privileges work	Expected capabilities functional

Integration Testing

Mental Model: The Complete Security System Rehearsal

Integration testing for a process sandbox is like **conducting a full-scale security drill** for a high-security facility. While unit tests examine individual security components in isolation (like testing each lock and sensor separately), integration tests activate the entire security system simultaneously and verify that all layers work together harmoniously. The goal is to ensure that the combination of namespace isolation, filesystem restrictions, system call filtering, resource limits, and capability dropping creates a robust containment environment without unexpected interactions that could compromise security or break legitimate functionality.

The critical insight is that security layers can interfere with each other in subtle ways. Namespace isolation might prevent seccomp filters from accessing required kernel information. Capability dropping might break cgroup management operations. Filesystem restrictions might prevent access to libraries needed for system call handling. Integration testing must uncover these interactions and verify that the orchestration sequence produces a stable, secure environment.

Realistic Workload Testing

Integration testing focuses on running realistic programs within the complete sandbox environment to verify that all security layers cooperate effectively. The test workloads should represent the types of programs the sandbox is designed to contain: computational tasks, file processing jobs, network services, and potentially malicious code that attempts various escape techniques.

The integration test suite includes several categories of realistic workloads:

Workload Category	Program Type	Security Challenge	Expected Behavior
Computational	CPU-intensive calculations	Resource limit enforcement	Program completes within CPU limits
File Processing	Read/write file operations	Filesystem isolation + I/O limits	Only sandbox files accessible, I/O throttled
System Information	Query system resources	Namespace isolation	Only sandbox view visible
Network Operations	Socket creation, connections	Network namespace isolation	Network access blocked
Privilege Escalation	Setuid, capability manipulation	Capability dropping + seccomp	All escalation attempts fail
Resource Exhaustion	Memory/CPU/disk bombs	Cgroup resource limits	Resource consumption capped, no host impact

Multi-Process Workload Testing

Real-world programs often spawn child processes, creating additional complexity for sandbox management. Integration tests must verify that all child processes inherit the same security restrictions, resource limits are applied hierarchically, and process management operations work correctly within the sandbox environment.

The multi-process testing approach examines several scenarios:

1. **Fork and Exec Chains**: Programs that spawn multiple child processes through fork and exec operations, verifying that each child inherits the complete security context and cannot escape through process manipulation.
2. **Process Communication**: Parent and child processes that communicate through pipes, shared memory, or other IPC mechanisms, confirming that communication works within the sandbox but cannot reach external processes.
3. **Process Tree Management**: Programs that create complex process hierarchies, ensuring that resource limits apply to the entire tree and that process cleanup handles all descendants correctly.
4. **Signal Handling**: Process groups that use signals for coordination, verifying that signal delivery works within the sandbox but cannot affect external processes.

The multi-process test framework tracks process relationships and resource usage:

Test Scenario	Process Pattern	Verification Points	Success Criteria
Simple Fork	Parent forks one child	Both processes in same cgroup, namespace	Resource limits apply to both, isolation maintained
Exec Chain	Process A execs B, B execs C	Security context preserved across execs	All execed processes have same restrictions
Process Tree	Parent spawns multiple children	Hierarchical resource accounting	Tree-wide resource limits enforced
IPC Communication	Processes share data via pipes	Communication works, isolation maintained	Internal IPC works, external blocked

Performance Impact Assessment

Integration testing must verify that the security layers do not impose excessive performance overhead that would make the sandbox impractical for real workloads. Performance testing measures the overhead introduced by each security mechanism and their combined impact on system resources.

The performance testing methodology includes:

1. **Baseline Measurements**: Run workloads on the host system without any sandbox restrictions to establish performance baselines for CPU usage, memory consumption, I/O throughput, and execution time.
2. **Incremental Layer Testing**: Add security layers one at a time and measure the performance impact of each addition, identifying which mechanisms contribute most to overhead and whether the impact is acceptable.
3. **Resource Efficiency**: Monitor system resource usage during sandbox operation to verify that the sandbox infrastructure itself does not consume excessive CPU, memory, or I/O bandwidth that could affect host system performance.

4. **Scalability Testing**: Run multiple sandbox instances simultaneously to verify that the system can handle realistic concurrent workloads without performance degradation or resource exhaustion.

The performance measurement framework tracks key metrics:

Performance Metric	Measurement Method	Acceptable Overhead	Failure Threshold
CPU Overhead	Compare execution time vs baseline	< 20% increase	> 50% increase
Memory Overhead	Monitor RSS usage of sandbox processes	< 10MB per sandbox	> 50MB per sandbox
I/O Throughput	Measure disk read/write rates	< 15% reduction	> 40% reduction
Startup Time	Time from sandbox creation to program execution	< 100ms	> 500ms
Cleanup Time	Time to fully clean up sandbox resources	< 50ms	> 200ms

Security Interaction Testing

The most critical aspect of integration testing is verifying that the combination of security layers does not create unexpected vulnerabilities or bypass opportunities. Security interaction testing attempts various escape techniques that might exploit the interfaces between different security mechanisms.

The security interaction test suite includes:

1. **Layer Bypass Attempts**: Try to use features of one security layer to circumvent restrictions imposed by another, such as using namespace manipulation to escape filesystem isolation or exploiting cgroup interfaces to bypass capability restrictions.
2. **Privilege Escalation Chains**: Attempt multi-step privilege escalation attacks that combine techniques across different security layers, verifying that the defense-in-depth approach blocks sophisticated attack sequences.
3. **Resource Exhaustion Attacks**: Try to exhaust resources in ways that might disable security mechanisms or cause the sandbox to fail in an unsafe state, ensuring that resource limits protect the security infrastructure itself.
4. **Information Leakage Tests**: Attempt to extract information about the host system through various channels, verifying that namespace isolation and filesystem restrictions prevent information disclosure.

The security testing approach systematically explores potential vulnerabilities:

Attack Category	Attack Vector	Security Layers Tested	Expected Defense
Container Escape	`/proc` traversal, mount manipulation	Namespace + filesystem isolation	All escape attempts blocked
Privilege Escalation	Setuid exploitation, capability abuse	Capability dropping + seccomp	Escalation impossible
Resource Bombing	Memory/CPU exhaustion attacks	Cgroup limits + seccomp	Resource consumption capped
Information Disclosure	System resource enumeration	Namespace isolation	Only sandbox view visible
Side-Channel Attacks	Timing analysis, cache behavior	Complete security stack	Minimal information leakage

Security Validation Tests

Mental Model: The Penetration Testing Laboratory

Security validation testing transforms our sandbox into a **target for systematic penetration testing**, where we deliberately attempt to break, bypass, or exploit every security mechanism we've implemented. Think of this phase as **hiring a professional burglar to test your security system** - we assume the role of an attacker with detailed knowledge of the sandbox's implementation and systematically probe every possible weakness. The goal is not just to verify that security mechanisms work under normal conditions, but to ensure they remain effective against sophisticated attacks designed specifically to exploit their weaknesses.

The key insight is that security validation must test both **known attack vectors** and **unexpected failure modes**. Known attacks include well-documented container escape techniques, privilege escalation exploits, and resource exhaustion attacks. Unexpected failure modes arise from the complex interactions between security layers and might include race conditions in cleanup code, edge cases in BPF filter evaluation, or resource accounting errors under extreme load.

Namespace Escape Prevention Testing

Namespace isolation is often the first target for container escape attempts because it provides the most fundamental isolation layer. Security validation must verify that namespace boundaries cannot be crossed through any combination of system calls, filesystem operations, or resource manipulations.

The namespace escape testing methodology systematically probes each namespace type for vulnerabilities:

1. **PID Namespace Escape Tests**: Attempt to access or manipulate processes outside the namespace through `/proc` traversal, process signaling, shared memory segments, and process tracing interfaces. Verify that all attempts fail and that the namespace boundary is impermeable.
2. **Mount Namespace Escape Tests**: Try to access the host filesystem through mount point manipulation, symlink traversal, bind mount exploitation, and `/proc` filesystem access. Confirm that all host filesystem access is blocked and that mount operations remain contained.
3. **Network Namespace Escape Tests**: Attempt to communicate with external networks through socket operations, netlink interfaces, network filesystem mounts, and shared network resources. Verify that all external communication is blocked and that network isolation is complete.
4. **User Namespace Privilege Escalation**: Try to gain host privileges through user namespace mapping manipulation, capability inheritance, setuid program exploitation, and filesystem permission bypasses. Ensure that user mapping provides isolation without privilege escalation opportunities.

The namespace escape test framework uses systematic probing techniques:

Escape Technique	Test Methodology	Security Expectation	Failure Indicators
/proc Traversal	Access `/proc/1/root`, `/proc/*/cwd`	Access denied, symlinks blocked	Host filesystem becomes visible
Mount Manipulation	Bind mount host paths, remount read-write	Operations fail or restricted	Host mounts accessible
Netlink Exploitation	Use netlink sockets to query host network	No host network information	Host network config visible
Capability Inheritance	Exploit user namespace for capability gain	No additional privileges acquired	Privileged operations succeed

Seccomp Filter Bypass Testing

Seccomp-BPF filters represent a critical security boundary that attackers will attempt to bypass through various techniques including syscall argument manipulation, architecture-specific exploit, and timing-based attacks. Security validation must verify that the BPF filter correctly handles all edge cases and cannot be circumvented.

The seccomp bypass testing approach includes:

1. **Syscall Argument Manipulation**: Test BPF filters with edge case arguments including null pointers, invalid addresses, extreme values, and crafted data structures designed to confuse argument parsing logic.
2. **Architecture-Specific Bypasses**: Verify that syscall number mappings are correct across different architectures, that no syscalls are missed due to architecture differences, and that calling conventions are properly handled.
3. **Race Condition Exploitation**: Attempt to exploit timing windows between filter installation and process execution, verify that filters cannot be modified after installation, and ensure that filter application is atomic.
4. **BPF Program Logic Flaws**: Test the BPF program with syscalls that have multiple code paths, verify that all conditional branches lead to correct decisions, and ensure that no logic errors allow forbidden operations.

The seccomp testing framework validates filter correctness systematically:

Bypass Technique	Test Implementation	Expected Filter Behavior	Vulnerability Indicators
Argument Crafting	Call allowed syscalls with malicious args	Arguments validated, bad calls blocked	Malicious args bypass filter
Syscall Aliasing	Use alternate syscall numbers for same operation	All variants blocked consistently	Alias syscalls succeed
Timing Attacks	Install filter during syscall execution	Filter applied atomically	Race condition allows bypass
Logic Exploitation	Trigger edge cases in BPF logic	All edge cases handled correctly	Unexpected filter behavior

Resource Limit Enforcement Verification

Resource limits protect both the host system and the sandbox infrastructure from resource exhaustion attacks. Security validation must verify that limits are enforced under extreme conditions and that limit enforcement mechanisms cannot be

bypassed or disabled.

The resource limit testing approach examines enforcement under pressure:

1. **Memory Exhaustion Attacks**: Rapidly allocate memory to exceed limits, attempt to bypass limits through shared memory or memory mapping, and verify that the OOM killer terminates processes correctly without affecting the host system.
2. **CPU Consumption Attacks**: Generate maximum CPU load to test throttling mechanisms, attempt to bypass CPU limits through process spawning or nice value manipulation, and ensure that other processes receive fair scheduling.
3. **Process Spawning Attacks**: Create fork bombs to test PID limits, attempt to exhaust process table entries, and verify that process creation limits protect system stability.
4. **I/O Exhaustion Attacks**: Generate intensive disk I/O to test bandwidth limits, attempt to bypass I/O limits through different filesystem interfaces, and ensure that I/O throttling protects system responsiveness.

The resource limit validation framework tests enforcement under extreme conditions:

Attack Type	Resource Pressure Method	Limit Enforcement Test	System Protection Verification
Memory Bomb	Allocate beyond cgroup limit	Process killed by OOM	Host memory usage unaffected
CPU Bomb	Busy loops consuming 100% CPU	CPU usage throttled to limit	Host processes remain responsive
Fork Bomb	Spawn processes rapidly	Fork fails at PID limit	Host process table protected
I/O Bomb	Heavy disk read/write load	I/O rate limited to quota	Host I/O performance preserved

Capability Escalation Prevention Testing

Capability dropping provides fine-grained privilege control, but capability management has many subtle edge cases that attackers may exploit. Security validation must verify that capability restrictions cannot be bypassed through any combination of system calls or process manipulation.

The capability escalation testing methodology includes:

1. **Direct Capability Exploitation**: Attempt operations that require dropped capabilities and verify they fail consistently, test that capability sets are correctly maintained across process operations, and ensure that no capabilities can be re-acquired.
2. **Indirect Privilege Escalation**: Try to gain privileges through setuid programs, filesystem capabilities, or other mechanisms that might bypass capability restrictions, ensuring that all privilege escalation paths are blocked.
3. **Capability Inheritance Attacks**: Exploit capability inheritance mechanisms through exec operations, process spawning, and shared resources to gain additional privileges in child processes.
4. **Ambient Capability Exploitation**: Test that ambient capabilities do not provide unexpected privilege escalation opportunities and that the interaction between different capability sets is secure.

The capability escalation test framework systematically probes privilege boundaries:

Escalation Method	Attack Implementation	Expected Failure Mode	Security Breach Indicators
Direct Cap Usage	Call syscalls requiring dropped caps	Operation fails with EPERM	Privileged operation succeeds
Setuid Exploitation	Execute setuid programs	No privilege change	Process gains privileges
Exec Inheritance	Gain caps through program exec	No additional capabilities	New capabilities acquired
Ambient Cap Abuse	Exploit ambient capability set	Ambient caps remain restricted	Unexpected privileges gained

Complete System Penetration Testing

The most comprehensive security validation involves **comprehensive penetration testing** that combines multiple attack techniques in sophisticated sequences designed to exploit the interactions between different security layers. This testing phase attempts realistic attack scenarios that a skilled adversary might use against the sandbox.

The penetration testing approach includes:

1. **Multi-Stage Attack Sequences**: Chain together multiple exploit techniques across different security layers, attempting to use partial success in one area to enable attacks against another security mechanism.
2. **Information Gathering and Reconnaissance**: Systematically probe the sandbox environment to gather information that could enable more targeted attacks, testing whether the sandbox reveals implementation details that could aid an attacker.
3. **Persistence and Stealth Testing**: Attempt to establish persistent access or hide malicious activities within the sandbox environment, verifying that monitoring and cleanup mechanisms detect and prevent such activities.
4. **Host Impact Assessment**: Conduct attacks designed to impact the host system through resource consumption, information disclosure, or privilege escalation, ensuring that the sandbox provides complete isolation.

The comprehensive penetration testing framework simulates realistic attack scenarios:

Attack Scenario	Attack Chain	Security Layers Challenged	Success Criteria
Container Breakout	Info gathering → namespace escape → privilege escalation	All layers	Complete containment maintained
Resource Warfare	Resource exhaustion → monitoring evasion → persistence	Cgroup limits + monitoring	Resource attacks contained
Stealth Persistence	Minimal footprint → cleanup evasion → information exfiltration	All layers + cleanup	No persistent access achieved
Host Compromise	Privilege escalation → capability abuse → system manipulation	Capabilities + seccomp	No host system impact

Milestone Checkpoints

Mental Model: Security Clearance Checkpoints

Think of milestone checkpoints as **security clearance verification stations** where each sandbox component must pass rigorous testing before being certified for the next level of integration. Just as security clearance involves both background checks and practical demonstrations of trustworthiness, each milestone checkpoint combines automated testing with

manual verification to ensure that the component functions correctly and securely before advancing to more complex integration scenarios.

The checkpoint system serves multiple purposes: it provides **incremental confidence building** as each security layer is validated, it enables **early problem detection** before components become entangled in complex interactions, and it creates **natural debugging boundaries** that help isolate issues to specific components when integration testing reveals problems.

Milestone 1: Namespace Isolation Verification

The namespace isolation checkpoint verifies that the foundational isolation layer works correctly and provides the expected separation of system resource views. This checkpoint is critical because all subsequent security layers depend on namespace isolation to provide their security context.

The namespace checkpoint testing procedure includes systematic verification of each namespace type:

1. **PID Namespace Verification**: Create a sandbox with PID namespace isolation and verify that the sandboxed process sees itself as PID 1, cannot observe host processes in its `/proc` view, and can only send signals to processes within its own namespace.
2. **Mount Namespace Verification**: Set up mount namespace isolation and confirm that mount operations within the sandbox are invisible to the host, that the sandbox cannot see host mount points, and that mount point manipulation cannot escape the namespace boundary.
3. **Network Namespace Verification**: Establish network namespace isolation and verify that the sandbox has no network connectivity, cannot see host network interfaces, and cannot communicate with external network services.
4. **UTS Namespace Verification**: Create UTS namespace isolation and confirm that hostname changes within the sandbox do not affect the host system and that the sandbox cannot observe host system identification information.

The namespace checkpoint verification process follows a structured validation sequence:

Verification Step	Test Command	Expected Output	Failure Indicators
PID Namespace	Run `ps aux` inside sandbox	Only sandbox processes visible, init PID is 1	Host processes visible, wrong init PID
Mount Namespace	Check `/proc/mounts` inside sandbox	Only sandbox mounts listed	Host mounts visible
Network Namespace	Run `ip addr` inside sandbox	Only loopback interface present	Host network interfaces visible
UTS Namespace	Change hostname, check from host	Host hostname unchanged	Host hostname modified

Namespace Isolation Debugging Checklist

When namespace isolation tests fail, systematic debugging helps identify the root cause:

⚠️ Common Namespace Issues

- **Missing CLONE_NEWPID Flag**: If processes don't see PID 1, verify that `CLONE_NEWPID` was specified in the clone() call and that `/proc` is mounted in the new namespace.

- **Insufficient Privileges**: Namespace creation requires `CAP_SYS_ADMIN` capability. Verify that the sandbox creator process has appropriate privileges or is running as root.
- **Namespace File Descriptor Leaks**: If namespace isolation is inconsistent, check that namespace file descriptors are properly managed and that processes enter the correct namespace via `setns()`.
- **Mount Propagation Issues**: If mount operations affect the host, verify that mount propagation is set to private and that the mount namespace is properly isolated.

Milestone 2: Filesystem Isolation Verification

The filesystem isolation checkpoint ensures that the sandbox provides a complete and secure filesystem environment that prevents access to sensitive host resources while providing all necessary functionality for legitimate programs.

The filesystem checkpoint validation process examines multiple aspects of filesystem isolation:

1. **Root Filesystem Containment**: Verify that processes cannot access paths outside the sandbox root directory through any combination of path traversal, symlink following, or special filesystem interfaces.
2. **Essential Service Availability**: Confirm that essential directories like `/proc`, `/dev`, and `/tmp` are properly mounted and populated with necessary entries for program execution.
3. **Library Dependency Resolution**: Test that programs can successfully load all required shared libraries and that no library dependencies cause execution failures within the sandbox environment.
4. **Filesystem Permission Enforcement**: Verify that filesystem permissions are correctly enforced within the sandbox and that privilege escalation through filesystem manipulation is prevented.

The filesystem checkpoint verification follows a comprehensive testing protocol:

Verification Category	Test Procedure	Success Criteria	Troubleshooting Steps
Root Containment	Try to access `/host`, `/../`, `/proc/1/root`	All access attempts fail	Check chroot/pivot_root, mount namespaces
Essential Directories	Test access to `/proc`, `/dev/null`, `/tmp`	All essential paths accessible	Verify mount operations, device creation
Library Loading	Run dynamically linked programs	Programs execute successfully	Check library paths, dependency resolution
Permission Enforcement	Test file operations with various permissions	Permissions enforced correctly	Verify filesystem type, mount options

Filesystem Isolation Debugging Checklist

Filesystem isolation problems often involve complex interactions between mount operations, permission settings, and library dependencies:

⚠️ **Common Filesystem Issues**

- **Missing Device Nodes**: If programs fail with "No such device" errors, verify that essential device files like `/dev/null`, `/dev/zero`, and `/dev/random` are present and have correct permissions.

- **Library Path Problems**: If programs fail to load, use `ldd` on the host to identify required libraries and ensure they are copied to the appropriate paths within the sandbox root.
- **Mount Order Dependencies**: If mounts fail inconsistently, verify that mount operations occur in the correct order and that parent directories exist before mounting child filesystems.
- **Permission Propagation**: If permission changes affect the host, check that bind mounts are configured appropriately and that filesystem operations are properly contained.

Milestone 3: System Call Filtering Verification

The seccomp filtering checkpoint validates that system call restrictions are correctly implemented and that the BPF filter provides robust protection against unauthorized system call usage while allowing legitimate operations to proceed.

The seccomp checkpoint testing protocol includes comprehensive validation of filter behavior:

1. **Whitelist Compliance Testing**: Execute programs that use only whitelisted system calls and verify they complete successfully without filter violations or unexpected termination.
2. **Blacklist Enforcement Testing**: Attempt to execute forbidden system calls and confirm they result in immediate process termination with appropriate exit codes and no side effects on system state.
3. **Argument-Level Filter Testing**: Test system calls with various argument combinations to verify that argument-based filtering works correctly and that malicious arguments are properly detected.
4. **Filter Robustness Testing**: Subject the seccomp filter to edge case inputs and stress conditions to ensure it remains stable and secure under all operating conditions.

The seccomp verification process validates filter behavior systematically:

Filter Test Category	Test Implementation	Expected Behavior	Validation Method
Allowed Syscalls	Call whitelisted syscalls with normal args	Syscalls complete successfully	Monitor exit status, check for violations
Forbidden Syscalls	Attempt blacklisted syscalls	Process killed immediately	Verify SIGSYS signal, check exit code
Argument Filtering	Call syscalls with filtered arguments	Bad args rejected, good args allowed	Test argument validation logic
Filter Edge Cases	Test with extreme or malformed inputs	Filter remains stable and secure	Monitor for crashes, bypasses, errors

Seccomp Filter Debugging Checklist

Seccomp filter problems often involve subtle issues in BPF program logic or syscall number handling:

⚠️ **Common Seccomp Issues**

- **Missing Required Syscalls**: If legitimate programs are killed unexpectedly, use `strace` to identify required syscalls that are missing from the whitelist.
- **Architecture-Specific Numbers**: If filters behave differently on different systems, verify that syscall numbers are correctly mapped for the target architecture.

- **NO_NEW_PRIVS Flag**: If seccomp filter installation fails, ensure that the `PR_SET_NO_NEW_PRIVS` flag is set before installing the filter.
- **BPF Program Validation**: If filter installation fails, verify that the BPF program is correctly formatted and that all instruction offsets and jump targets are valid.

Milestone 4: Resource Limit Enforcement Verification

The cgroup resource limit checkpoint ensures that resource consumption is properly controlled and that limits are enforced under various load conditions without affecting host system performance or stability.

The resource limit checkpoint validation examines enforcement across all resource types:

1. **Memory Limit Enforcement**: Generate memory pressure beyond configured limits and verify that processes are terminated by the OOM killer when limits are exceeded while ensuring that memory usage remains within bounds.
2. **CPU Limit Enforcement**: Generate CPU load and verify that CPU usage is throttled to the configured percentage while confirming that other system processes continue to receive appropriate CPU time.
3. **Process Count Limit Enforcement**: Spawn processes beyond the configured PID limit and verify that fork operations fail appropriately while existing processes continue to function correctly.
4. **I/O Bandwidth Limit Enforcement**: Generate intensive disk I/O and verify that throughput is limited to the configured rate while ensuring that I/O operations remain functional within the limits.

The cgroup verification protocol tests limit enforcement under controlled conditions:

Resource Type	Load Generation	Limit Verification	System Protection Check
Memory	Allocate increasingly large buffers	Process killed at limit	Host memory usage stable
CPU	Run CPU-intensive loops	Usage capped at percentage	Host CPU responsive
Process Count	Spawn child processes rapidly	Fork fails at limit	Host process creation unaffected
I/O Bandwidth	Heavy file read/write operations	Throughput limited to quota	Host I/O performance maintained

Resource Limit Debugging Checklist

Resource limit problems often involve cgroup configuration issues or incorrect limit calculations:

⚠️ **Common Cgroup Issues**

- **Controller Availability**: If resource limits are not enforced, verify that the required cgroup controllers are available and enabled in the kernel configuration.
- **Cgroup Version Compatibility**: If cgroup operations fail, check whether the system uses cgroups v1 or v2 and ensure that the sandbox uses the appropriate interfaces.
- **Permission Problems**: If cgroup directory creation fails, verify that the process has appropriate permissions to create and manage cgroup hierarchies.
- **Limit Value Validation**: If resource limits behave unexpectedly, verify that limit values are within acceptable ranges and properly formatted for the cgroup interface.

Milestone 5: Capability Management Verification

The capability management checkpoint validates that Linux capabilities are correctly dropped and that privilege restrictions are properly enforced while ensuring that necessary functionality remains available within the restricted environment.

The capability checkpoint testing protocol comprehensively validates privilege restrictions:

1. **Capability Dropping Verification**: Confirm that specified capabilities are removed from all capability sets and that attempts to use dropped capabilities fail consistently with appropriate error codes.
2. **Privilege Operation Blocking**: Test operations that require various capabilities and verify that dropped capabilities prevent unauthorized privileged operations while retained capabilities continue to function.
3. **No New Privileges Enforcement**: Verify that the `PR_SET_NO_NEW_PRIVS` flag prevents privilege escalation through exec operations and that setuid binaries do not grant additional privileges.
4. **User Transition Verification**: Confirm that user and group ID changes occur correctly and that the final privilege state matches the expected configuration for the sandbox environment.

The capability verification process validates privilege restrictions systematically:

Capability Test	Privilege Operation	Expected Result	Verification Method
Dropped Capabilities	Attempt operations requiring dropped caps	Operation fails with EPERM	Test specific privileged syscalls
Retained Capabilities	Use operations requiring retained caps	Operation succeeds normally	Verify necessary functions work
No New Privileges	Execute setuid binaries	No privilege escalation	Check effective UID unchanged
User Transition	Change to unprivileged user	User change successful, caps adjusted	Verify final UID/GID and capability state

Capability Management Debugging Checklist

Capability management problems often involve the complex interactions between different capability sets and privilege transition operations:

⚠️ **Common Capability Issues**

- **Operation Ordering**: If capability dropping fails, verify that operations occur in the correct sequence: drop capabilities, set no-new-privs, then change user/group.
- **Capability Dependencies**: If essential functions break after dropping capabilities, identify the minimum required capability set and ensure that necessary capabilities are retained.
- **Ambient Capability Handling**: If child processes have unexpected privileges, check that ambient capabilities are properly managed and do not provide unintended privilege inheritance.
- **Verification Method Accuracy**: If capability verification reports incorrect results, ensure that capability checking uses the appropriate interfaces and examines all relevant capability sets.

Implementation Guidance

The testing strategy implementation requires a systematic approach that builds testing infrastructure alongside the sandbox components. The testing framework must be robust enough to handle the complex interactions between security

layers while providing clear diagnostic information when tests fail.

Testing Technology Recommendations

Testing Component	Simple Option	Advanced Option
Unit Test Framework	Basic C assert macros with test runner	Check framework with fixtures and mocking
Process Management	fork() with waitpid() for child testing	Process sandbox library with cleanup tracking
System Call Tracing	Manual strace analysis	Automated syscall capture and analysis
Resource Monitoring	/proc filesystem parsing	Cgroup statistics API with automated collection
Security Validation	Manual exploit attempts	Automated penetration testing framework

Recommended Testing File Structure

The testing infrastructure should be organized to support both component-level testing and integration scenarios:

```
process-sandbox/
  tests/
    unit/
      test_namespaces.c         ← Namespace isolation unit tests
      test_filesystem.c         ← Filesystem isolation unit tests
      test_seccomp.c           ← Seccomp filter unit tests
      test_cgroups.c           ← Cgroup resource limit unit tests
      test_capabilities.c       ← Capability management unit tests
    integration/
      test_workloads.c         ← Realistic program testing
      test_security.c          ← Security validation tests
      test_performance.c       ← Performance impact assessment
    common/
      test_framework.c         ← Shared testing utilities
      test_helpers.c           ← Common test helper functions
    workloads/
      cpu_bomb.c               ← CPU exhaustion test program
      memory_bomb.c            ← Memory exhaustion test program
      fork_bomb.c              ← Process spawning test program
      escape_attempt.c         ← Container escape test program
  scripts/
    run_tests.sh              ← Test execution orchestration
    analyze_results.sh        ← Test result analysis
```

Testing Infrastructure Starter Code

The testing framework requires infrastructure for managing child processes, monitoring resource usage, and validating security properties. This complete testing foundation handles the complex orchestration required for security testing:

```c
// test_framework.c - Complete testing infrastructure

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <signal.h>

#include <errno.h>

#include <string.h>

#include <time.h>


typedef struct {

    char name[128];

    int (*test_func)(void);

    int passed;

    int executed;

    char error_msg[256];

} test_case_t;


typedef struct {

    pid_t child_pid;

    int timeout_seconds;

    int expected_exit_code;

    int expected_signal;

    time_t start_time;

} child_test_context_t;


// Global test registry

static test_case_t test_registry[256];

static int test_count = 0;


// Register a test case
```

```c
void register_test(const char* name, int (*test_func)(void)) {
    if (test_count >= 256) {
        fprintf(stderr, "Too many tests registered\n");
        exit(1);
    }

    strncpy(test_registry[test_count].name, name, 127);
    test_registry[test_count].test_func = test_func;
    test_registry[test_count].passed = 0;
    test_registry[test_count].executed = 0;
    test_count++;
}

// Execute child process with timeout and monitoring
int run_child_test(child_test_context_t* ctx, int (*child_func)(void)) {
    ctx->child_pid = fork();
    if (ctx->child_pid == 0) {
        // Child process - run the test
        int result = child_func();
        exit(result);
    } else if (ctx->child_pid > 0) {
        // Parent process - monitor child
        ctx->start_time = time(NULL);
        int status;

        // Wait with timeout
        for (int i = 0; i < ctx->timeout_seconds; i++) {
            int wait_result = waitpid(ctx->child_pid, &status, WNOHANG);
            if (wait_result == ctx->child_pid) {
                // Child completed
```

```
                if (WIFEXITED(status)) {
                    return WEXITSTATUS(status) == ctx->expected_exit_code ? 0 : -1;
                } else if (WIFSIGNALED(status)) {
                    return WTERMSIG(status) == ctx->expected_signal ? 0 : -1;
                }
            } else if (wait_result == -1) {
                perror("waitpid failed");
                return -1;
            }
            sleep(1);
        }

        // Timeout - kill child
        kill(ctx->child_pid, SIGKILL);
        waitpid(ctx->child_pid, &status, 0);
        return -1;
    } else {
        perror("fork failed");
        return -1;
    }
}

// Verify process capabilities
int verify_capability_dropped(cap_value_t capability) {
    cap_t caps = cap_get_proc();
    if (caps == NULL) {
        return -1;
    }

    cap_flag_value_t value;
```

```c
    if (cap_get_flag(caps, capability, CAP_EFFECTIVE, &value) == -1) {
        cap_free(caps);
        return -1;
    }

    cap_free(caps);
    return (value == CAP_CLEAR) ? 0 : -1;
}

// Monitor resource usage
typedef struct {
    unsigned long memory_kb;
    unsigned int cpu_percent;
    int process_count;
    unsigned long io_read_bytes;
    unsigned long io_write_bytes;
} resource_usage_t;

int get_cgroup_usage(const char* cgroup_path, resource_usage_t* usage) {
    char path[256];
    FILE* file;

    // Read memory usage
    snprintf(path, sizeof(path), "%s/memory.usage_in_bytes", cgroup_path);
    file = fopen(path, "r");
    if (file) {
        fscanf(file, "%lu", &usage->memory_kb);
        usage->memory_kb /= 1024;
        fclose(file);
    }
```

```
    // Read CPU usage (simplified)
    snprintf(path, sizeof(path), "%s/cpuacct.usage", cgroup_path);
    file = fopen(path, "r");
    if (file) {
        unsigned long cpu_ns;
        fscanf(file, "%lu", &cpu_ns);
        // Convert to percentage (simplified calculation)
        usage->cpu_percent = (cpu_ns / 10000000) % 100;
        fclose(file);
    }

    // Read process count
    snprintf(path, sizeof(path), "%s/pids.current", cgroup_path);
    file = fopen(path, "r");
    if (file) {
        fscanf(file, "%d", &usage->process_count);
        fclose(file);
    }

    return 0;
}
```

Core Testing Logic Skeleton

The core testing functions provide the framework for implementing specific security validation tests:

```c
// Core testing function signatures with detailed TODOs


// Test namespace isolation effectiveness
int test_namespace_isolation() {
    // TODO 1: Create child process with namespace isolation using create_namespaces()
    // TODO 2: In child, check that PID namespace shows process as PID 1
    // TODO 3: Verify mount namespace isolates filesystem view
    // TODO 4: Confirm network namespace blocks external connectivity
    // TODO 5: Test UTS namespace isolates hostname changes
    // TODO 6: Validate that all namespace boundaries prevent escape
    // TODO 7: Clean up namespace file descriptors and child process
}

// Test filesystem containment
int test_filesystem_isolation() {
    // TODO 1: Set up minimal root filesystem using create_minimal_rootfs()
    // TODO 2: Apply chroot or pivot_root isolation
    // TODO 3: Attempt to access host filesystem paths - should fail
    // TODO 4: Verify essential directories (/proc, /dev, /tmp) are accessible
    // TODO 5: Test that symlink traversal cannot escape sandbox
    // TODO 6: Confirm library dependencies are satisfied
    // TODO 7: Clean up temporary filesystem and mounts
}

// Test seccomp filter enforcement
int test_seccomp_filter() {
    // TODO 1: Build BPF filter program using build_bpf_program()
    // TODO 2: Install seccomp filter with install_seccomp_filter()
    // TODO 3: Test that whitelisted syscalls work correctly
    // TODO 4: Attempt blacklisted syscalls - process should be killed
    // TODO 5: Test argument-level filtering with various parameters
```

```
    // TODO 6: Verify filter cannot be bypassed with edge cases
    // TODO 7: Confirm filter remains stable under stress conditions
}

// Test resource limit enforcement
int test_resource_limits() {
    // TODO 1: Create cgroup hierarchy with create_cgroup_hierarchy()
    // TODO 2: Configure memory, CPU, and process limits
    // TODO 3: Generate memory pressure - verify OOM killer enforcement
    // TODO 4: Generate CPU load - verify throttling to configured limit
    // TODO 5: Spawn processes beyond limit - verify fork failures
    // TODO 6: Generate I/O load - verify bandwidth throttling
    // TODO 7: Clean up cgroup hierarchy and test processes
}

// Test capability restriction
int test_capability_management() {
    // TODO 1: Get baseline capabilities with get_current_capabilities()
    // TODO 2: Drop specified capabilities using drop_capabilities()
    // TODO 3: Set no-new-privileges flag with set_no_new_privs()
    // TODO 4: Attempt operations requiring dropped capabilities - should fail
    // TODO 5: Verify setuid programs do not escalate privileges
    // TODO 6: Test capability inheritance in child processes
    // TODO 7: Confirm final capability state matches expectations
}

// Integration test with realistic workload
int test_complete_sandbox() {
    sandbox_config_t* config;
    sandbox_state_t* state;
```

```
    // TODO 1: Initialize sandbox configuration with sandbox_config_init()
    // TODO 2: Configure all security layers (namespaces, filesystem, seccomp, cgroups, capabilities)
    // TODO 3: Create complete sandbox with create_sandbox()
    // TODO 4: Execute realistic workload in sandbox environment
    // TODO 5: Monitor resource usage and security properties during execution
    // TODO 6: Verify all security restrictions remain effective
    // TODO 7: Clean up sandbox resources with cleanup_sandbox()
}
```

Milestone Checkpoint Implementation

Each milestone requires specific validation steps that confirm the security mechanism is working correctly:

```bash
#!/bin/bash

# Milestone checkpoint validation script


# Milestone 1: Namespace Isolation Checkpoint

validate_milestone_1() {

    echo "=== Milestone 1: Namespace Isolation Verification ==="



    # Test PID namespace isolation

    sudo ./test_sandbox --test=pid_namespace_isolation

    if [ $? -eq 0 ]; then

        echo "✓ PID namespace isolation working"

    else

        echo "✗ PID namespace isolation failed"

        return 1

    fi



    # Test mount namespace isolation

    sudo ./test_sandbox --test=mount_namespace_isolation

    if [ $? -eq 0 ]; then

        echo "✓ Mount namespace isolation working"

    else

        echo "✗ Mount namespace isolation failed"

        return 1

    fi



    # Test network namespace isolation

    sudo ./test_sandbox --test=network_namespace_isolation

    if [ $? -eq 0 ]; then

        echo "✓ Network namespace isolation working"

    else
```

```
        echo "✗ Network namespace isolation failed"
        return 1
    fi

    echo "Milestone 1 checkpoint: PASSED"
    return 0
}

# Milestone 3: Seccomp Filter Checkpoint
validate_milestone_3() {
    echo "=== Milestone 3: Seccomp Filter Verification ==="

    # Test allowed syscalls work
    sudo ./test_sandbox --test=seccomp_whitelist_compliance
    if [ $? -eq 0 ]; then
        echo "✓ Whitelisted syscalls working"
    else
        echo "✗ Whitelisted syscalls failed"
        return 1
    fi

    # Test forbidden syscalls are blocked
    sudo ./test_sandbox --test=seccomp_blacklist_enforcement
    if [ $? -eq 0 ]; then
        echo "✓ Blacklisted syscalls blocked"
    else
        echo "✗ Blacklisted syscalls not blocked"
        return 1
    fi
```

```
    echo "Milestone 3 checkpoint: PASSED"

    return 0

}
```

Security Validation Test Framework

The security validation framework provides systematic testing of attack scenarios:

```c
// Security validation test implementation
typedef struct {
    char attack_name[64];
    int (*attack_func)(void);
    int expected_failure;  // 1 if attack should fail, 0 if should succeed
} security_test_t;

// Test container escape attempts
int test_container_escape() {
    // TODO 1: Attempt /proc traversal to access host filesystem
    // TODO 2: Try mount manipulation to escape namespace
    // TODO 3: Attempt symlink traversal to host directories
    // TODO 4: Test bind mount exploitation techniques
    // TODO 5: Verify all escape attempts fail with appropriate errors
    // Return 0 if all attacks blocked, -1 if any succeed
}

// Test privilege escalation attempts
int test_privilege_escalation() {
    // TODO 1: Attempt to use dropped capabilities
    // TODO 2: Try setuid program exploitation
    // TODO 3: Test capability inheritance bypass
    // TODO 4: Attempt ambient capability manipulation
    // TODO 5: Verify all escalation attempts fail
    // Return 0 if all attacks blocked, -1 if any succeed
}

// Test resource exhaustion attacks
int test_resource_exhaustion() {
    // TODO 1: Generate memory bomb to test OOM killer
    // TODO 2: Create CPU bomb to test throttling
```

```
    // TODO 3: Launch fork bomb to test PID limits
    // TODO 4: Generate I/O bomb to test bandwidth limits
    // TODO 5: Verify host system remains stable and responsive
    // Return 0 if all attacks contained, -1 if host affected
}
```

This comprehensive testing strategy ensures that each security component works correctly in isolation and that the complete system provides robust defense against sophisticated attacks while maintaining usability for legitimate workloads.

Debugging Guide

> **Milestone(s):** This section applies to all milestones (1-5), providing systematic debugging approaches for namespace isolation, filesystem restrictions, system call filtering, resource limits, and capability management

Building a process sandbox requires orchestrating multiple complex Linux security mechanisms, each with their own failure modes and debugging challenges. The **Mental Model: The Crime Scene Investigation** - think of debugging a sandbox failure like being a detective at a crime scene. You need to systematically examine all the evidence (logs, process state, system calls, resource usage) to reconstruct what happened and identify the root cause. Just as a detective follows clues from the scene backwards to understand the sequence of events, sandbox debugging requires tracing symptoms back through the layers of security mechanisms to find where the failure originated.

The debugging process becomes particularly challenging because sandbox failures often manifest as cryptic symptoms far removed from their root causes. A process that fails to start might be blocked by seccomp, missing library dependencies, insufficient capabilities, or resource limits. A process that starts but behaves incorrectly might be operating with an incomplete filesystem view, blocked system calls, or unexpected namespace isolation. The key to effective debugging is understanding how these security layers interact and developing systematic approaches to isolate which layer is causing the problem.

Symptom-Based Diagnosis

Effective sandbox debugging starts with systematic symptom analysis. Each type of failure produces characteristic symptoms that point toward specific security layers or configuration issues. Understanding these patterns allows developers to quickly narrow down the root cause rather than randomly checking all possible failure points.

The following diagnostic matrix maps common symptoms to their most likely root causes, organized by the security layer most commonly responsible:

Symptom	Likely Root Cause	Primary Layer	Secondary Causes
Process fails to start with "No such file or directory"	Missing executable or libraries in sandbox filesystem	Filesystem Isolation	Mount namespace not properly initialized, chroot incomplete
Process starts but immediately exits with status 127	Dynamic linker cannot find shared libraries	Filesystem Isolation	Incorrect library paths, missing ld.so cache
Process killed with SIGSYS signal	Blocked system call by seccomp filter	System Call Filtering	Missing syscall in whitelist, architecture mismatch
Process hangs during startup	Blocked system call without error return	System Call Filtering	Filter returns ERRNO instead of allowing required syscall
Process fails with "Permission denied" on file access	Insufficient capabilities for operation	Capability Management	Missing CAP_DAC_OVERRIDE or other required capability
Process cannot bind to network port	Network namespace isolation or capability restriction	Namespace Isolation	Missing CAP_NET_BIND_SERVICE or isolated network namespace
Process killed by OOM killer	Memory limit exceeded	Resource Limits	Cgroup memory limit too low, memory leak in sandboxed code
Process throttled or slow performance	CPU or I/O limits reached	Resource Limits	Cgroup CPU quota exhausted, I/O bandwidth throttling active
Cannot create child processes	Process limit exceeded	Resource Limits	Cgroup PID limit reached, fork bomb protection triggered
Process sees wrong PID or hostname	Namespace isolation incomplete	Namespace Isolation	PID or UTS namespace not created properly
Mount operations fail	Missing mount capabilities or wrong namespace	Namespace Isolation	CAP_SYS_ADMIN dropped, mount namespace not isolated
Device access fails	Missing device nodes or permissions	Filesystem Isolation	/dev not populated, device files missing or wrong permissions

Process Startup Failures represent the most common category of sandbox problems. These typically manifest as immediate process termination with cryptic error codes. The key diagnostic approach involves systematically checking each prerequisite:

1. **Executable existence**: Verify the target program exists at the expected path within the sandbox filesystem
2. **Library dependencies**: Use `ldd` on the host to identify required shared libraries, then verify each exists in the sandbox
3. **Dynamic linker configuration**: Ensure `/etc/ld.so.conf` or `/etc/ld.so.cache` provides correct library search paths
4. **File permissions**: Check that executable has proper permissions and all parent directories are accessible

System Call Blocking symptoms require careful analysis of which operations are failing. The most reliable diagnostic approach uses `strace` to capture the exact system call that triggers the seccomp violation:

```
Process killed by SIGSYS at system call: openat()
Arguments: AT_FDCWD, "/etc/passwd", O_RDONLY
```

This information immediately identifies whether the problem is a missing syscall in the whitelist (openat not allowed) or argument-level filtering (specific file path blocked).

Resource Exhaustion symptoms often appear as performance degradation before triggering hard limits. The diagnostic approach involves monitoring resource usage patterns:

- Memory exhaustion: Process RSS grows until hitting cgroup limit, then OOM killer activates
- CPU throttling: Process experiences intermittent delays as CPU quota is exhausted and restored
- I/O throttling: File operations become progressively slower as bandwidth limit is reached
- Process limits: Fork or clone operations fail with EAGAIN when PID limit is exceeded

Namespace Isolation Issues typically manifest as processes seeing unexpected system state. The key diagnostic technique involves comparing the process's view of system resources with the expected isolated view:

- PID namespace: Check if process sees itself as PID 1 and only sees processes in its namespace
- Mount namespace: Verify filesystem layout matches expected sandbox structure
- Network namespace: Confirm network interfaces, routing tables, and port bindings are isolated
- UTS namespace: Check hostname and domain name are isolated from host system

> The critical debugging insight is that symptoms often appear in one security layer but originate in another. A "permission denied" error might seem like a capability problem but actually result from seccomp blocking the underlying system call before capability checks occur.

Debugging Tools and Techniques

Effective sandbox debugging requires a comprehensive toolkit of Linux debugging utilities, each specialized for examining different aspects of process isolation and security enforcement. The key principle is using the right tool for each security layer rather than relying on generic debugging approaches.

Namespace Inspection Tools provide visibility into the isolated views that sandboxed processes experience. These tools are essential for verifying that namespace isolation is working correctly:

Tool	Purpose	Usage Pattern	Key Information
`lsns`	List all namespaces on system	`lsns -p <pid>`	Shows which namespaces process belongs to
`nsenter`	Enter existing namespace	`nsenter -t <pid> -p -m /bin/bash`	Execute commands in process's namespace view
`unshare`	Create new namespaces	`unshare -p -m -n /bin/bash`	Test namespace creation independently
`ip netns`	Network namespace management	`ip netns exec <name> ip addr`	Inspect network configuration in namespace
`/proc/<pid>/ns/`	Namespace file descriptors	`ls -la /proc/<pid>/ns/`	Compare namespace IDs between processes

The most powerful namespace debugging technique involves using `nsenter` to "step inside" the sandboxed process's view of the system. This allows direct inspection of what the process actually sees:

```bash
# Enter all namespaces of sandboxed process
nsenter -t $SANDBOX_PID -p -m -n -u /bin/bash

# Now running inside sandbox's view - check what's visible
ps aux          # Should only see processes in PID namespace
mount           # Should see sandbox filesystem layout
ip addr         # Should see isolated network interfaces
hostname        # Should see sandbox hostname
```

Filesystem Debugging Tools help identify problems with chroot, mount configuration, and file access permissions within the isolated filesystem:

Tool	Purpose	Usage Pattern	Key Information
`findmnt`	Display filesystem tree	`findmnt -D`	Shows all mount points with propagation
`lsof`	List open files	`lsof -p <pid>`	Shows what files process actually has open
`strace`	Trace file operations	`strace -e trace=file <command>`	Shows file access attempts and results
`ls -la /proc/<pid>/fd/`	Open file descriptors	Direct inspection	Shows numbered file descriptors
`/proc/<pid>/mountinfo`	Process mount view	`cat /proc/<pid>/mountinfo`	Shows mounts visible to specific process

A systematic approach to filesystem debugging involves comparing the sandbox's filesystem view with expectations:

1. **Mount verification**: Use `findmnt` to confirm all required filesystems are mounted at correct locations
2. **Path resolution**: Trace file access attempts to see where path lookups fail
3. **Permission analysis**: Check that file permissions, ownership, and capabilities are correct
4. **Library dependency tracking**: Verify all shared libraries are accessible and properly linked

System Call Tracing provides the most detailed view of how processes interact with the kernel, making it indispensable for debugging seccomp issues:

Tool	Purpose	Usage Pattern	Key Information
`strace`	System call tracer	`strace -f -o trace.log <command>`	Complete syscall log with arguments
`perf trace`	Performance-oriented tracing	`perf trace -p <pid>`	Lower overhead syscall monitoring
`seccomp_tools`	Seccomp filter analysis	`seccomp-tools dump <pid>`	Shows active seccomp filters
`bpftool`	BPF program inspection	`bpftool prog show`	Lists loaded BPF programs
`/proc/<pid>/seccomp_filter`	Seccomp status	`cat /proc/<pid>/seccomp_filter`	Shows seccomp mode and filter count

The key to effective system call debugging is using filtered tracing to focus on relevant operations rather than being overwhelmed by every syscall:

```bash
# Trace only file operations
strace -e trace=file,desc -f -o file_trace.log ./sandbox_program

# Trace only network operations
strace -e trace=network -f -o network_trace.log ./sandbox_program

# Trace specific problematic syscalls
strace -e trace=openat,read,write,mmap -f -o specific_trace.log ./sandbox_program
```

Resource Monitoring Tools help diagnose cgroup-related issues by showing actual resource consumption versus configured limits:

Tool	Purpose	Usage Pattern	Key Information
`systemd-cgtop`	Live cgroup resource view	`systemd-cgtop`	Shows CPU, memory, I/O usage by cgroup
`cgget`	Read cgroup parameters	`cgget -g memory:/sandbox/<id>`	Shows current cgroup configuration
`cgexec`	Execute in cgroup	`cgexec -g memory:/test <command>`	Test resource limits independently
`/sys/fs/cgroup/`	Direct cgroup inspection	`cat /sys/fs/cgroup/memory/sandbox/memory.usage_in_bytes`	Raw cgroup statistics
`dmesg`	Kernel log messages	`dmesg -T | grep -i "killed process"`	Shows OOM killer activity

Capability Analysis Tools help identify privilege-related failures by showing exactly which capabilities a process has and needs:

Tool	Purpose	Usage Pattern	Key Information
`capsh`	Capability shell	`capsh --print`	Shows current process capabilities
`getpcaps`	Get process capabilities	`getpcaps <pid>`	Shows capabilities of specific process
`pscap`	Process capability scan	`pscap -a`	Lists capabilities of all processes
`filecap`	File capability scan	`filecap /usr/bin/`	Shows file-based capabilities
`/proc/<pid>/status`	Process status	`grep Cap /proc/<pid>/status`	Shows capability bitmasks

> The debugging strategy should always start with the most specific tools for the suspected problem area, then expand to broader system-level analysis if the specific tools don't reveal the issue.

System Call Tracing

System call tracing represents the most powerful debugging technique for sandbox issues because it provides complete visibility into how processes interact with the kernel. Every security mechanism ultimately affects system call behavior, making syscall tracing the common diagnostic pathway for all sandbox problems.

Comprehensive Tracing Strategy involves using `strace` with systematic filtering and analysis approaches. The key principle is starting broad to identify the problem area, then narrowing focus to specific system calls or failure patterns:

```bash
# Phase 1: Broad tracing to identify problem category
strace -f -o full_trace.log -T -tt -v ./sandbox_program

# Phase 2: Filter to specific syscall categories
strace -e trace=process -f -o process_trace.log ./sandbox_program    # Process creation
strace -e trace=file -f -o file_trace.log ./sandbox_program          # File operations
strace -e trace=network -f -o network_trace.log ./sandbox_program    # Network operations
strace -e trace=ipc -f -o ipc_trace.log ./sandbox_program            # IPC operations

# Phase 3: Focus on specific failing syscalls
strace -e trace=openat,read,write -f -o specific_trace.log ./sandbox_program
```

The `-f` flag is crucial for sandbox tracing because it follows child processes created by fork or clone. Many sandbox issues occur in child processes, so missing this flag can hide the actual problem. The `-T` flag shows time spent in each syscall, helping identify performance issues caused by resource throttling.

Seccomp Filter Analysis through system call tracing requires understanding how seccomp violations appear in trace output. When seccomp blocks a system call, it can either return an error code or kill the process, depending on the filter configuration:

```
# Seccomp returning EPERM (error code)
openat(AT_FDCWD, "/etc/passwd", O_RDONLY) = -1 EPERM (Operation not permitted)

# Seccomp killing process (appears as signal)
openat(AT_FDCWD, "/etc/passwd", O_RDONLY) = ?
+++ killed by SIGSYS +++
```

The key diagnostic technique involves comparing syscall traces between successful execution outside the sandbox versus failed execution inside the sandbox. Differences in syscall success/failure patterns immediately identify which operations are being blocked.

Advanced Tracing Techniques help diagnose complex interaction issues between multiple security layers:

Technique	Command	Purpose	Analysis Focus
Multi-process tracing	`strace -f -ff -o trace`	Separate log per process	Child process failures
Time-based analysis	`strace -T -tt`	Timestamp each syscall	Performance bottlenecks
Signal tracing	`strace -e signal=all`	Show signal delivery	Seccomp kills, resource limits
Memory mapping	`strace -e trace=mmap,munmap,mprotect`	Memory operations	Library loading issues
Capability tracing	`strace -e trace=capget,capset`	Privilege changes	Capability dropping problems

Filter Development and Testing uses system call tracing to iteratively develop seccomp filters. The process involves:

1. **Baseline tracing**: Run the target program outside sandbox to capture complete syscall profile
2. **Syscall cataloging**: Extract unique syscall names and analyze their necessity
3. **Iterative filtering**: Start with minimal whitelist, add syscalls as failures occur
4. **Argument analysis**: Use detailed traces to develop argument-level filtering rules

```bash
# Extract unique syscalls from trace
grep -o '^[a-z_]*(' trace.log | sort | uniq > required_syscalls.txt

# Count syscall frequency to prioritize
grep -o '^[a-z_]*(' trace.log | sort | uniq -c | sort -nr > syscall_frequency.txt

# Analyze specific syscall arguments
grep 'openat(' trace.log | grep -v ENOENT | head -20
```

Performance Impact Analysis uses tracing to identify resource bottlenecks caused by cgroup limits. CPU throttling appears as increased time spent in syscalls, memory pressure shows up as frequent memory management operations, and I/O throttling causes delays in read/write operations:

```bash
# Identify slow syscalls (potential throttling)
strace -T -f -o timing_trace.log ./sandbox_program
grep -E ' <[0-9]+\.[0-9]{3}[0-9]+>' timing_trace.log | sort -k2 -nr

# Monitor memory pressure
strace -e trace=mmap,munmap,brk,madvise -T -f -o memory_trace.log ./sandbox_program

# Track I/O patterns
strace -e trace=read,write,readv,writev -T -f -o io_trace.log ./sandbox_program
```

Common System Call Patterns that indicate specific problems:

Pattern	Syscalls Involved	Indicates	Solution
Repeated `openat("/lib64/ld-linux-x86-64.so.2")` failures	openat, access, stat	Dynamic linker problems	Fix library paths, populate /lib64
`clone()` followed by immediate `exit()`	clone, exit	Child process startup failure	Check child process requirements
Frequent `futex()` calls with timeouts	futex	Lock contention or deadlock	Analyze threading issues
`mmap()` failures with ENOMEM	mmap, brk	Memory limit reached	Increase memory limit or optimize usage
`socket()` returning EACCES	socket, bind	Network namespace or capability issue	Check network isolation, CAP_NET_BIND_SERVICE

> The most effective system call debugging approach combines automated analysis (extracting patterns from large trace files) with manual inspection of specific failure points. Start with automated tools to identify the problem area, then manually analyze the specific syscalls that are failing.

Implementation Guidance

Technology Recommendations:

Component	Simple Option	Advanced Option
System Call Tracing	strace with shell scripts	Custom ptrace-based tracer
Log Analysis	grep/awk/sed	Python with syscall parsing library
Namespace Inspection	Manual nsenter commands	Automated namespace comparison tool
Resource Monitoring	Manual cgroup file reading	Real-time monitoring with eBPF
Error Reporting	Simple logging to stderr	Structured logging with correlation IDs

Recommended File Structure:

```
project-root/
  debug/
    debug_utils.h           ← debugging utility functions
    debug_utils.c           ← implementation
    trace_analyzer.c        ← system call trace analysis
    namespace_inspector.c   ← namespace state inspection
    resource_monitor.c      ← cgroup resource monitoring
  tools/
    sandbox_debug.c         ← main debugging tool
    trace_filter.sh         ← strace filtering scripts
    namespace_diff.py       ← namespace comparison utility
  tests/
    debug_test.c           ← debugging functionality tests
```

Debugging Infrastructure Code:

```c
// debug_utils.h - Complete debugging utilities
#ifndef DEBUG_UTILS_H
#define DEBUG_UTILS_H

#include <sys/types.h>
#include <time.h>

// Debug levels for filtering output
typedef enum {
    DEBUG_LEVEL_ERROR = 1,
    DEBUG_LEVEL_WARN = 2,
    DEBUG_LEVEL_INFO = 3,
    DEBUG_LEVEL_DEBUG = 4,
    DEBUG_LEVEL_TRACE = 5
} debug_level_t;

// Namespace inspection results
typedef struct {
    pid_t pid;
    ino_t pid_ns;
    ino_t mnt_ns;
    ino_t net_ns;
    ino_t uts_ns;
    ino_t user_ns;
    ino_t cgroup_ns;
    char hostname[256];
    int mount_count;
    int process_count;
} namespace_info_t;

// Resource usage snapshot
typedef struct {
```

```
    unsigned long memory_usage_bytes;
    unsigned long memory_limit_bytes;
    double cpu_usage_percent;
    unsigned long io_read_bytes;
    unsigned long io_write_bytes;
    int process_count;
    int process_limit;
    time_t timestamp;
} resource_snapshot_t;

// System call trace entry
typedef struct {
    char syscall_name[32];
    pid_t pid;
    int return_value;
    int errno_value;
    double duration_ms;
    char arguments[512];
    time_t timestamp;
} syscall_trace_entry_t;

// Function declarations
void debug_log(debug_level_t level, const char* component, const char* format, ...);
int inspect_namespace_state(pid_t pid, namespace_info_t* info);
int get_resource_usage(const char* cgroup_path, resource_snapshot_t* snapshot);
int trace_syscalls_to_file(pid_t pid, const char* output_file, int duration_seconds);
int analyze_trace_file(const char* trace_file, syscall_trace_entry_t* entries, int max_entries);
int compare_namespace_states(const namespace_info_t* expected, const namespace_info_t* actual);
void print_debugging_report(pid_t sandbox_pid, const char* cgroup_path);

#endif
```

Core Debugging Logic Skeleton:

```c
// Main debugging entry point - learners implement the diagnosis logic
int diagnose_sandbox_failure(pid_t sandbox_pid, const char* cgroup_path,
                             const char* expected_behavior) {
    // TODO 1: Capture current namespace state and compare with expected isolation
    // Use inspect_namespace_state() to get actual state
    // Compare with expected namespace configuration

    // TODO 2: Check resource usage against limits
    // Use get_resource_usage() to get current consumption
    // Identify if any limits are being exceeded

    // TODO 3: Analyze recent system call activity
    // Use trace_syscalls_to_file() to capture recent activity
    // Look for failed syscalls, blocked operations, or unusual patterns

    // TODO 4: Check process health and responsiveness
    // Send test signals to verify process is responsive
    // Check if process is in expected state (running/sleeping)

    // TODO 5: Generate structured diagnosis report
    // Combine all findings into actionable diagnosis
    // Provide specific recommendations for fixing identified issues

    return SANDBOX_SUCCESS;
}

// Automated syscall trace analysis - learners implement pattern recognition
int analyze_syscall_patterns(const syscall_trace_entry_t* entries, int count,
                             char* diagnosis, size_t diagnosis_size) {
    // TODO 1: Count failed syscalls by type
```

```
    // Group failures by syscall name and errno value
    // Identify most common failure patterns

    // TODO 2: Detect seccomp violations
    // Look for SIGSYS kills or EPERM returns from seccomp
    // Identify which syscalls are being blocked

    // TODO 3: Identify resource exhaustion patterns
    // Look for ENOMEM, EAGAIN, or other resource-related errors
    // Correlate with resource usage trends

    // TODO 4: Analyze timing patterns
    // Identify unusually slow syscalls (potential throttling)
    // Look for timeout patterns in futex or network operations

    // TODO 5: Generate human-readable diagnosis
    // Convert patterns into actionable recommendations
    // Prioritize most likely root causes

    return SANDBOX_SUCCESS;
}

// Interactive debugging session - learners implement step-by-step diagnosis
int run_interactive_debug_session(pid_t sandbox_pid) {
    // TODO 1: Present main diagnostic menu
    // Options: namespace inspection, resource monitoring, syscall tracing, etc.

    // TODO 2: Execute selected diagnostic command
    // Run appropriate inspection function based on user choice
```

```
    // TODO 3: Display results in human-readable format
    // Format output with clear explanations of what each value means

    // TODO 4: Suggest follow-up actions based on findings
    // Recommend next diagnostic steps or potential fixes

    // TODO 5: Allow iterative investigation
    // Loop back to menu for continued diagnosis

    return SANDBOX_SUCCESS;
}
```

Language-Specific Debugging Hints:

- Use `errno` and `strerror()` to get human-readable error messages from failed system calls
- The `/proc` filesystem provides extensive debugging information: `/proc/<pid>/status`, `/proc/<pid>/ns/`, `/proc/<pid>/cgroup`
- Signal handling with `sigaction()` can catch seccomp violations (SIGSYS) for custom error reporting
- Use `getrlimit()` and `setrlimit()` to check and modify resource limits for testing
- The `clone()` system call flags can be inspected with `/proc/<pid>/status` to verify namespace membership
- File descriptor leaks can be detected by monitoring `/proc/<pid>/fd/` directory size over time

Milestone Checkpoint - Debugging Infrastructure:

After implementing the debugging utilities, verify functionality with these tests:

```bash
# Test namespace inspection
./sandbox_debug --inspect-namespaces <sandbox_pid>
# Expected: Shows PID, mount, network, UTS namespace IDs different from host

# Test resource monitoring
./sandbox_debug --monitor-resources <cgroup_path> --duration 30
# Expected: Shows memory, CPU, I/O usage updating every few seconds

# Test syscall tracing
./sandbox_debug --trace-syscalls <sandbox_pid> --output trace.log --duration 10
# Expected: Creates trace file with timestamped syscall entries

# Test automated diagnosis
echo "test failure scenario" | ./sandbox_debug --diagnose <sandbox_pid>
# Expected: Provides structured analysis with specific recommendations
```

Signs that debugging infrastructure is working correctly:

- Namespace inspection shows isolated namespace IDs for sandboxed processes
- Resource monitoring detects when limits are exceeded and processes are throttled
- Syscall tracing captures both successful and failed system calls with detailed arguments
- Automated diagnosis provides actionable recommendations rather than generic errors

Common debugging infrastructure problems:

- **Permission denied accessing /proc files**: Debugging tools need sufficient privileges to inspect other processes
- **Incomplete syscall traces**: Missing `-f` flag in strace loses child process activity
- **Resource monitoring shows zeros**: Cgroup files might not be accessible or cgroup not properly configured
- **Namespace inspection fails**: Process might have exited or debugging tool lacks required capabilities

Advanced Debugging Techniques:

For complex sandbox failures, consider these advanced approaches:

```c
// Custom ptrace-based syscall monitor
int attach_syscall_monitor(pid_t target_pid) {
    // TODO: Implement ptrace attachment for real-time syscall monitoring
    // More efficient than strace for long-running monitoring
    // Can implement custom filtering logic
}

// eBPF-based resource monitoring
int setup_ebpf_monitoring(pid_t target_pid) {
    // TODO: Use eBPF programs for low-overhead resource monitoring
    // Can track custom metrics not available through cgroups
    // Provides real-time alerts for resource violations
}

// Automated sandbox health checks
int continuous_health_monitoring(pid_t target_pid, int check_interval_seconds) {
    // TODO: Implement periodic health checks with alerting
    // Monitor for zombie processes, resource leaks, security violations
    // Automatically restart or cleanup failed sandboxes
}
```

Future Extensions

> **Milestone(s):** This section applies to all milestones (1-5), exploring advanced enhancements and operational improvements that can be built upon the foundational sandbox implementation

The process sandbox implementation represents a solid foundation for secure code execution, but the rapidly evolving landscape of Linux security features and operational requirements opens numerous opportunities for enhancement. These future extensions fall into three primary categories: advanced security features that leverage cutting-edge kernel capabilities, performance optimizations that improve sandbox efficiency and scalability, and management tooling that enables production deployment at scale.

Understanding these potential extensions serves multiple purposes for learners. First, it demonstrates that security is never a completed project—new threats emerge, new kernel features become available, and operational requirements evolve. Second, it provides a roadmap for continued learning and skill development in systems security. Third, it illustrates how

foundational knowledge of namespaces, seccomp, cgroups, and capabilities serves as a springboard for understanding more sophisticated security mechanisms.

The extensions presented here range from relatively straightforward enhancements that could be implemented immediately after completing the core milestones, to research-grade features that represent the cutting edge of container security. Each category addresses different aspects of the classic trade-off between security, performance, and usability that defines all practical security systems.

Advanced Security Features

Mental Model: The Evolving Security Arsenal

Think of advanced security features as upgrading from a basic home security system to a state-of-the-art facility protection network. Your current sandbox is like having locks, an alarm system, and security cameras—effective and proven. Advanced features are like adding biometric access controls, AI-powered threat detection, motion sensors that can distinguish between authorized personnel and intruders, and automated response systems that can isolate threats before they spread. Each new capability adds another layer of sophisticated protection, but also introduces complexity in configuration, integration, and troubleshooting.

The landscape of Linux security features has expanded dramatically in recent years, driven by the growing adoption of containers and the increasing sophistication of attack vectors. While the foundational mechanisms of namespaces, seccomp, cgroups, and capabilities remain essential, newer kernel features provide more granular control, better performance, and protection against attack classes that weren't well-understood when the original mechanisms were designed.

Linux Security Modules (LSM) Integration

Linux Security Modules represent a fundamental shift from discretionary access control to mandatory access control policies that can enforce security decisions regardless of process ownership or traditional Unix permissions. The most relevant LSMs for sandbox enhancement are SELinux, AppArmor, and the newer Landlock framework.

SELinux Integration provides label-based security that can enforce policies based on security contexts rather than user identity. In a sandbox context, SELinux can prevent sandboxed processes from accessing resources even if filesystem permissions would normally allow access. This creates an additional security layer that operates independently of namespace isolation and capability restrictions.

SELinux Feature	Security Benefit	Implementation Complexity	Performance Impact
Type Enforcement	Process-resource access control	High - requires policy development	Medium - kernel policy evaluation
Multi-Level Security	Information flow control	Very High - complex classification	Low - cached policy decisions
Role-Based Access	Administrative separation	Medium - role definition required	Low - minimal runtime overhead
Confined Domains	Process behavior restriction	High - domain transition rules	Medium - policy lookup overhead

Decision: Landlock for Filesystem Access Control

- **Context**: Need finer-grained filesystem access control than mount namespaces alone provide, without the complexity of full SELinux policy development
- **Options Considered**: SELinux type enforcement, AppArmor path-based restrictions, Landlock filesystem sandboxing
- **Decision**: Implement Landlock integration for filesystem access control
- **Rationale**: Landlock provides programmatic access control without requiring system-wide policy configuration, offers better granularity than mount namespaces, and has lower complexity than SELinux
- **Consequences**: Enables per-file access control within sandbox, requires kernel 5.13+, adds another security layer to coordinate with existing mechanisms

Landlock Integration represents the most promising LSM enhancement for process sandboxes because it was specifically designed for application-level sandboxing. Unlike SELinux or AppArmor, which require system-wide policy configuration, Landlock allows individual applications to impose additional restrictions on themselves and their children.

Landlock filesystem rules can restrict access to specific files or directories regardless of namespace configuration. This enables scenarios like allowing a sandboxed process to access `/etc/passwd` for user lookup while preventing access to `/etc/shadow`, even if both files are visible in the mount namespace. The programmatic nature of Landlock policies means they can be generated dynamically based on sandbox configuration rather than requiring pre-written policy files.

The integration approach involves creating Landlock rulesets during sandbox initialization, after namespace creation but before dropping capabilities. This ensures the sandbox orchestrator retains the ability to call the Landlock system calls, but the sandboxed process inherits the restrictions without the ability to modify them.

Seccomp Notify and User Mode Helper

Traditional seccomp filtering operates as a binary allow-or-deny decision made entirely in kernel space. The seccomp notify mechanism, introduced in Linux 5.0, enables a more sophisticated approach where filtered system calls can be intercepted and handled by a user space supervisor process.

Seccomp Notify Architecture creates a communication channel between the kernel's seccomp filter and a supervisor process. When a sandboxed process attempts a system call that matches a notify filter rule, the kernel suspends the sandboxed process and sends details about the system call to the supervisor through a file descriptor. The supervisor can then examine the system call arguments, potentially perform the operation on behalf of the sandboxed process with additional validation, and return either the successful result or an error code.

Notify Use Case	Security Advantage	Implementation Challenge	Performance Consideration
File Access Mediation	Path-based access control	Complex path canonicalization	Context switching overhead
Network Request Filtering	Protocol-aware filtering	Network protocol parsing	Socket state management
IPC Auditing	Comprehensive communication logging	Message format interpretation	Audit data volume
Resource Quota Enforcement	Dynamic limit adjustment	Cross-process state tracking	Lock contention management

This mechanism enables sophisticated policies like allowing `openat()` system calls only for files that match specific patterns, or permitting network connections only to pre-approved destinations. The supervisor can implement complex logic that would be impossible to express in a static BPF program, such as rate limiting, quota management, or integration with external policy engines.

Implementation Strategy involves modifying the BPF filter generation to include `SECCOMP_RET_USER_NOTIF` return values for system calls that require mediation, creating a supervisor thread or process that handles the notification file descriptor, and implementing a policy engine that can make access control decisions based on system call arguments and current sandbox state.

The supervisor process design presents interesting architectural choices. A threaded approach keeps the supervisor within the same process as the sandbox orchestrator, simplifying state sharing but creating potential attack surface if the supervisor is compromised. A separate process approach provides better isolation but requires more complex inter-process communication and state management.

eBPF and Advanced Monitoring

Extended Berkeley Packet Filter (eBPF) represents a significant evolution beyond traditional seccomp BPF, enabling sophisticated monitoring and enforcement capabilities throughout the kernel. While seccomp BPF is limited to system call filtering, eBPF programs can be attached to numerous kernel events including network packet processing, filesystem operations, process scheduling, and memory management.

eBPF Integration Opportunities for sandbox enhancement include runtime security monitoring that tracks sandbox behavior patterns and detects anomalous activity, performance profiling that identifies resource usage patterns and optimization opportunities, and dynamic policy enforcement that can adapt security policies based on observed behavior.

LSM hooks provide attachment points where eBPF programs can intercept and potentially modify security decisions made by the kernel. An eBPF program attached to the `security_file_permission` hook can implement custom file access policies, while programs attached to `security_task_create` can monitor process creation and enforce additional restrictions on child processes.

eBPF Program Type	Monitoring Capability	Security Enhancement	Resource Overhead
LSM Hook Programs	Security decision interception	Dynamic policy enforcement	Low - efficient JIT compilation
Tracepoint Programs	System event monitoring	Anomaly detection	Medium - event processing overhead
Kprobe Programs	Function call tracing	Detailed behavior analysis	High - frequent execution
Socket Filter Programs	Network traffic analysis	Protocol-specific filtering	Medium - packet processing

The observability benefits of eBPF extend beyond security to performance optimization and debugging. eBPF programs can collect detailed metrics about system call latency, memory allocation patterns, and I/O behavior within the sandbox. This data enables identification of performance bottlenecks and validation that security restrictions are not causing unexpected performance degradation.

Implementation Considerations include the requirement for specific kernel versions and configuration options, the complexity of eBPF program development and verification, and the need for careful resource management to prevent

eBPF programs from impacting system performance. The eBPF verifier ensures program safety but requires understanding of eBPF programming constraints and kernel internals.

Hardware Security Features

Modern processors provide hardware-assisted security features that can enhance process isolation beyond what software mechanisms alone can achieve. Intel's Memory Protection Extensions (MPX), Control-flow Enforcement Technology (CET), and Intel Memory Protection Keys (MPK) offer hardware-level enforcement of memory safety and control flow integrity.

Intel MPK Integration enables fine-grained memory protection by associating memory regions with protection keys and allowing rapid switching of access permissions for those keys. In a sandbox context, MPK can provide an additional layer of memory isolation that operates independently of virtual memory protections and can be toggled with minimal overhead.

The integration approach involves allocating separate protection keys for sandbox memory regions during process initialization, configuring initial access permissions that allow supervisor access but restrict sandboxed code, and potentially implementing dynamic permission switching based on execution context. This requires careful coordination with memory management and process lifecycle management to ensure keys are properly allocated and cleaned up.

ARM Pointer Authentication and Memory Tagging provide similar hardware-assisted security features on ARM platforms. Pointer authentication can detect return-oriented programming (ROP) and jump-oriented programming (JOP) attacks by cryptographically signing return addresses and function pointers. Memory tagging associates tags with memory allocations and pointers, enabling detection of use-after-free and buffer overflow vulnerabilities.

The heterogeneous nature of hardware security features across processor architectures requires careful abstraction layer design that can leverage available features without creating hard dependencies on specific hardware capabilities. The implementation should gracefully degrade when hardware features are not available while taking advantage of them when present.

Performance Optimizations

Mental Model: The Assembly Line Optimization

Think of performance optimizations as transforming a custom craft workshop into an efficient assembly line. Your current sandbox is like a skilled craftsperson who can create a custom security environment for each request, carefully measuring, cutting, and assembling each component. Performance optimizations are like introducing pre-cut materials, standardized templates, assembly jigs, and parallel workstations. The end result is the same high-quality product, but produced with dramatically higher throughput and lower resource consumption per unit.

Performance optimization in security systems presents unique challenges because optimizations cannot compromise security properties. Every performance enhancement must be carefully analyzed to ensure it doesn't introduce security vulnerabilities or weaken the existing security model. This creates a constraint that doesn't exist in many other performance optimization contexts.

Template-Based Sandbox Creation

The sandbox creation process involves numerous expensive operations: namespace creation, filesystem setup, cgroup configuration, and seccomp filter compilation. These operations are largely independent of the specific program being sandboxed, suggesting opportunities for template-based optimization.

Sandbox Templates involve pre-creating sandbox environments with common configurations and reusing them for multiple sandboxed processes. This approach amortizes the expensive setup operations across multiple sandbox instances and reduces the latency between sandbox request and program execution.

The template approach requires careful design to maintain security isolation. Templates cannot share writable resources between instances, and each template instantiation must receive its own isolated namespace and filesystem view. The implementation strategy involves creating "golden" sandbox environments with read-only base configurations and using copy-on-write mechanisms to provide per-instance writable areas.

Template Component	Reuse Strategy	Isolation Mechanism	Performance Benefit
Base Filesystem	Read-only mount with overlay	Overlay filesystem per instance	Eliminates filesystem construction
Namespace Configuration	Clone from template namespace	Namespace creation with CLONE_NEW* flags	Reduces namespace setup time
Seccomp Filter	Compiled BPF program reuse	Per-process filter installation	Eliminates BPF compilation
Cgroup Structure	Shared cgroup hierarchy	Per-instance cgroup directories	Reduces cgroup creation overhead

Container Image Integration extends the template concept by leveraging existing container image formats and tooling. Container images provide standardized mechanisms for packaging filesystem contents and metadata, enabling integration with existing container ecosystems while maintaining the enhanced security properties of the custom sandbox implementation.

The integration approach involves implementing container image unpacking and mounting within the sandbox filesystem isolation component, adapting container image metadata to sandbox configuration parameters, and potentially supporting container registry integration for dynamic image retrieval.

This integration enables use cases like sandboxing language-specific runtime environments (Node.js, Python, JVM) that are distributed as container images, while applying more restrictive security policies than typical container runtimes provide.

Process Pool Management

Creating new processes and configuring security restrictions involves significant overhead, particularly when sandboxes are used for short-lived tasks. Process pool management addresses this by maintaining a pool of pre-configured sandbox processes that can be rapidly assigned to new tasks.

Pool Architecture involves creating a master pool manager that maintains multiple pre-initialized sandbox processes in a ready state, a task dispatcher that assigns incoming work to available processes from the pool, and a lifecycle manager that handles process recycling and replacement.

The security implications require careful consideration. Pool processes must be reset to a clean state between tasks to prevent information leakage, and the pool manager must ensure that process failures don't compromise other pool members. The implementation strategy involves designing a task execution protocol that provides clean isolation between tasks and implementing process health monitoring to detect and replace unhealthy pool members.

Pool Management Aspect	Design Challenge	Security Consideration	Performance Impact
Process Initialization	Balancing pool size with resource usage	Limiting pool process capabilities	Startup time amortization
Task Isolation	Ensuring clean state between tasks	Preventing information leakage	Context switching overhead
Failure Recovery	Detecting and replacing failed processes	Maintaining pool security properties	Recovery time impact
Resource Management	Controlling total resource consumption	Enforcing per-task and pool-wide limits	Memory and CPU efficiency

Task Execution Protocol design involves creating a communication mechanism between the pool manager and pool processes that can safely transmit task specifications and results. The protocol must prevent task data from being accessible to subsequent tasks and ensure that task failures don't compromise the pool process.

The protocol implementation options include Unix domain sockets for high-performance local communication, shared memory with careful synchronization for bulk data transfer, and file descriptor passing for providing task-specific resource access. Each approach has different performance characteristics and security implications.

Resource Sharing and Deduplication

Multiple sandbox instances often require similar resources, particularly when executing related tasks or using common runtime environments. Resource sharing and deduplication can significantly reduce memory usage and improve cache efficiency while maintaining security isolation.

Memory Deduplication using kernel same-page merging (KSM) can identify identical memory pages across different sandbox processes and merge them into shared read-only pages. This is particularly effective for shared libraries, runtime environments, and static data that is identical across multiple sandbox instances.

The security analysis requires ensuring that memory deduplication doesn't create information leakage channels. The KSM implementation provides copy-on-write semantics that prevent one process from observing modifications made by another process, but timing-based attacks might potentially infer information about shared memory usage patterns.

Filesystem Layer Sharing extends the overlay filesystem approach to share common layers across multiple sandbox instances. When multiple sandboxes use similar base environments, the common portions can be shared while providing per-instance writable layers for modifications.

Sharing Mechanism	Resource Savings	Security Risk	Implementation Complexity
Shared Library Memory	High - common across all processes	Low - read-only sharing	Low - kernel KSM support
Filesystem Layer Sharing	Medium - depends on commonality	Medium - layer isolation required	Medium - overlay management
Seccomp Filter Sharing	Low - filters are small	Low - per-process installation	Low - compiled program reuse
Network Namespace Sharing	High - for network-isolated workloads	High - requires network isolation	High - complex namespace management

The implementation strategy involves identifying sharing opportunities during sandbox configuration, implementing reference counting and lifecycle management for shared resources, and providing monitoring and debugging capabilities to understand sharing effectiveness and detect sharing-related issues.

Asynchronous Operations and Batching

Many sandbox management operations can be performed asynchronously without impacting security properties. Asynchronous operation design can improve responsiveness and enable batching optimizations that reduce per-operation overhead.

Asynchronous Cleanup involves deferring resource cleanup operations that don't have immediate security implications to background tasks. Cgroup removal, filesystem unmounting, and namespace cleanup can often be performed after the sandboxed process has terminated without creating security risks.

The design involves implementing a cleanup queue with background worker threads, prioritizing cleanup operations based on resource pressure and security sensitivity, and providing monitoring capabilities to ensure cleanup operations don't accumulate indefinitely.

Batched Configuration Updates can optimize scenarios where multiple sandbox instances require similar configuration changes. Cgroup limit updates, seccomp filter installations, and namespace configuration changes can potentially be batched to reduce system call overhead.

The batching strategy requires careful analysis to ensure that batching doesn't delay critical security updates or create windows where security policies are inconsistently applied across related sandbox instances.

Management and Monitoring

Mental Model: The Mission Control Center

Think of management and monitoring as building a mission control center for your sandbox fleet. Your current implementation is like a skilled pilot flying a single aircraft—effective for the immediate task but limited in scope. Management and monitoring systems are like air traffic control, flight tracking systems, communication networks, and automated response protocols that enable coordination of hundreds of aircraft simultaneously. They provide visibility into the entire operation, early warning of problems, and coordinated response capabilities that no single operator could manage manually.

Production deployment of process sandboxes requires sophisticated management and monitoring capabilities that extend far beyond the core isolation mechanisms. These systems must provide operational visibility, automated response to

failures, resource planning data, and security monitoring that can detect sophisticated attacks or policy violations.

Comprehensive Metrics Collection

Effective sandbox management requires detailed metrics covering security events, performance characteristics, resource utilization, and operational health. The metrics collection system must balance comprehensive coverage with minimal performance impact on sandbox operations.

Security Metrics include tracking seccomp filter violations with detailed context about blocked system calls and their arguments, monitoring capability usage patterns to detect privilege escalation attempts, and recording namespace escape attempts or other container breakout behaviors.

The metrics collection architecture involves implementing lightweight instrumentation throughout the sandbox components, designing efficient data aggregation mechanisms that can handle high-volume metric streams, and providing configurable retention and sampling policies to manage storage requirements.

Metric Category	Key Metrics	Collection Method	Storage Requirements
Security Events	Seccomp violations, capability usage, escape attempts	Event-driven logging with structured data	High - retain for audit
Resource Utilization	Memory, CPU, I/O consumption per sandbox	Periodic sampling from cgroups	Medium - time-series data
Performance	Sandbox creation latency, operation throughput	Request/response timing	Medium - aggregated statistics
Operational Health	Process lifecycle events, error rates	State change notifications	Low - current status only

Performance Metrics provide data for optimization decisions and capacity planning. Key metrics include sandbox creation and destruction latency, resource utilization efficiency, and throughput characteristics under various load patterns. These metrics enable identification of performance bottlenecks and validation that optimizations provide expected benefits.

The collection system design involves implementing efficient metric aggregation that can handle high-frequency measurements, providing configurable metric retention policies that balance storage costs with analytical requirements, and ensuring that metrics collection doesn't significantly impact sandbox performance.

Centralized Policy Management

Production sandbox deployments often require consistent policy enforcement across multiple hosts and dynamic policy updates without service interruption. Centralized policy management provides the administrative capabilities needed for large-scale deployments.

Policy Distribution Architecture involves implementing a policy server that maintains canonical policy definitions, a distribution mechanism that ensures policy updates are consistently applied across all sandbox hosts, and a versioning system that enables rollback and audit trail maintenance.

The policy management system must handle various policy types including seccomp filter specifications that define allowed system calls for different sandbox categories, resource limit templates that provide standard resource allocations for different workload types, and filesystem policy definitions that specify allowed file access patterns.

Policy Type	Update Frequency	Distribution Latency	Rollback Requirements
Seccomp Filters	Low - security-driven updates	Fast - security critical	Immediate rollback capability
Resource Limits	Medium - capacity management	Medium - operational impact	Gradual rollback preferred
Filesystem Policies	Low - application-driven	Medium - functionality impact	Validation before rollback
Network Policies	High - dynamic environments	Fast - connectivity critical	Automatic rollback on failure

Policy Validation and Testing capabilities ensure that policy updates don't break existing functionality or introduce security vulnerabilities. The validation system includes automated testing of new policies against representative workloads, staged deployment mechanisms that apply policy changes incrementally, and monitoring integration that detects policy-related issues after deployment.

The implementation involves creating policy specification languages or configuration formats that can express complex security requirements, implementing policy compilation and optimization mechanisms that generate efficient runtime enforcement code, and providing policy simulation capabilities that can predict the impact of proposed policy changes.

Automated Incident Response

Large-scale sandbox deployments require automated response capabilities to handle security incidents, resource exhaustion, and operational failures without human intervention. Automated incident response systems can react to threats faster than human operators and provide consistent response policies across the entire deployment.

Security Incident Response involves implementing detection mechanisms for various attack patterns including container escape attempts, privilege escalation behaviors, and resource exhaustion attacks. The response system can automatically isolate affected sandbox instances, collect forensic data, and implement containment measures.

The incident response architecture includes event correlation engines that can identify attack patterns across multiple sandbox instances, automated containment mechanisms that can isolate compromised sandbox instances without impacting others, and escalation policies that engage human operators when automated responses are insufficient.

Incident Type	Detection Method	Automated Response	Escalation Criteria
Container Escape	Namespace boundary violations	Process termination and isolation	Successful escape detection
Resource Exhaustion	Cgroup limit violations	Resource reallocation or throttling	Sustained exhaustion patterns
Anomalous Behavior	Behavioral pattern analysis	Increased monitoring and logging	Patterns matching known attacks
Policy Violations	Seccomp or capability violations	Process termination and alerting	Repeated violation attempts

Operational Incident Response handles non-security failures including host resource exhaustion, network connectivity issues, and software failures. The response system can implement failover mechanisms, automatic resource rebalancing, and service degradation strategies that maintain partial functionality during outages.

The implementation strategy involves creating a rule engine that can process complex incident detection logic, implementing response action libraries that provide safe and tested incident response capabilities, and providing

comprehensive logging and audit trails that enable post-incident analysis and system improvement.

Integration with Container Orchestration

Modern deployment environments often involve container orchestration platforms like Kubernetes, which provide additional management capabilities but may require integration with custom sandbox implementations. Integration strategies can leverage orchestration platform capabilities while maintaining enhanced security properties.

Kubernetes Integration involves implementing custom resource definitions (CRDs) that represent sandbox configurations and policies, developing operators that manage sandbox lifecycle within Kubernetes clusters, and implementing admission controllers that can enforce additional security policies beyond standard Kubernetes controls.

The integration approach includes designing sandbox-aware scheduling policies that consider security requirements and resource isolation needs, implementing network policy integration that coordinates between Kubernetes network policies and sandbox network isolation, and providing monitoring integration that surfaces sandbox-specific metrics through standard Kubernetes monitoring systems.

Runtime Integration with container runtimes like containerd or CRI-O enables leveraging existing container management capabilities while applying enhanced security restrictions. This approach involves implementing runtime plugins or alternative runtime implementations that provide standard container runtime interfaces while using the custom sandbox security mechanisms.

The integration strategy includes implementing OCI runtime specification compliance that enables integration with standard container tooling, providing image management integration that can work with existing container image workflows, and ensuring that enhanced security properties are preserved when integrating with standard container management systems.

Implementation Guidance

The implementation of these advanced features requires careful planning and staged development to avoid introducing complexity that compromises the reliability and security of the core sandbox functionality. The following guidance provides concrete starting points and implementation strategies for each category of enhancement.

Technology Recommendations

Component	Simple Option	Advanced Option
LSM Integration	Landlock filesystem rules (kernel 5.13+)	Full SELinux policy integration
eBPF Monitoring	libbpf with pre-written CO-RE programs	Custom eBPF program development
Metrics Collection	Prometheus exposition format	Custom high-performance metrics system
Policy Management	JSON configuration files with validation	Distributed policy server with versioning
Template Storage	Local filesystem with overlay mounts	Container registry integration
Process Pooling	Thread pool with fork() workers	Pre-forked process pool with shared memory

Recommended File Structure

The advanced features should be organized as optional extensions that can be enabled independently without affecting core functionality:

```
project-root/
  cmd/
    sandbox/main.c              ← core sandbox implementation
    sandbox-advanced/main.c     ← advanced feature integration
  src/
    core/                       ← core sandbox components (from milestones 1-5)
      namespace.c
      filesystem.c
      seccomp.c
      cgroups.c
      capabilities.c
    extensions/
      lsm/
        landlock.c              ← Landlock integration
        landlock.h
      ebpf/
        monitor.c               ← eBPF monitoring programs
        monitor.h
        programs/               ← BPF source files
          security_monitor.bpf.c
      performance/
        templates.c             ← sandbox template management
        pool.c                  ← process pool management
        sharing.c               ← resource sharing and deduplication
      management/
        metrics.c               ← metrics collection and exposition
        policy.c                ← policy management system
        incident.c              ← automated incident response
  config/
    policies/                   ← policy configuration files
      default.json
      restricted.json
    templates/                  ← sandbox template definitions
      minimal.json
      nodejs.json
  scripts/
    setup-advanced.sh           ← system setup for advanced features
    benchmark.sh               ← performance testing utilities
```

Landlock Integration Starter Code

```c
#include <linux/landlock.h>
#include <sys/prctl.h>
#include <sys/syscall.h>

// Landlock integration for filesystem access control
struct landlock_ruleset_attr {
    __u64 handled_access_fs;
};

struct landlock_path_beneath_attr {
    __u64 allowed_access;
    __s32 parent_fd;
};

// Initialize Landlock ruleset for filesystem restrictions
int setup_landlock_filesystem_rules(const char** allowed_paths, size_t path_count) {
    // TODO 1: Check if Landlock is supported by attempting to create ruleset
    // TODO 2: Create ruleset with ABI version 1 and filesystem access types
    // TODO 3: For each allowed path, open directory and add rule to ruleset
    // TODO 4: Enforce ruleset by calling landlock_restrict_self()
    // TODO 5: Return 0 on success, negative error code on failure
    // Hint: Use landlock_create_ruleset(), landlock_add_rule(), landlock_restrict_self()
}

// Test Landlock restrictions by attempting file access
int test_landlock_restrictions(void) {
    // TODO 1: Attempt to open allowed file - should succeed
    // TODO 2: Attempt to open restricted file - should fail with EACCES
    // TODO 3: Return 0 if restrictions work correctly, 1 if bypass detected
}
```

eBPF Security Monitor Skeleton

```c
#include <bpf/bpf.h>
#include <bpf/libbpf.h>

struct security_event {
    pid_t pid;
    uid_t uid;
    char comm[16];
    int event_type;
    union {
        struct {
            int syscall_nr;
            long arg0, arg1, arg2;
        } syscall_event;
        struct {
            char filename[256];
            int access_mode;
        } file_event;
    };
};

// Initialize eBPF security monitoring
int setup_ebpf_security_monitor(void) {
    // TODO 1: Load BPF program from object file using libbpf
    // TODO 2: Attach program to LSM hooks for file access and process creation
    // TODO 3: Create perf event buffer for receiving security events
    // TODO 4: Start background thread to process events from buffer
    // TODO 5: Return file descriptor for cleanup, negative on error
    // Hint: Use bpf_object__open_file(), bpf_program__attach_lsm()
}
```

```
// Process security events from eBPF program
void* security_monitor_thread(void* arg) {
    // TODO 1: Poll perf event buffer for security events
    // TODO 2: Parse security_event structure from buffer
    // TODO 3: Apply security policies and generate alerts
    // TODO 4: Log events to metrics collection system
    // TODO 5: Continue until shutdown signal received
}
```

Sandbox Template Management

```c
#include <sys/mount.h>
#include <sys/stat.h>

typedef struct {
    char template_id[64];
    char base_path[PATH_MAX];
    char overlay_dir[PATH_MAX];
    sandbox_config_t* config;
    int reference_count;
    time_t created_time;
} sandbox_template_t;

// Create reusable sandbox template
int create_sandbox_template(const char* template_id, sandbox_config_t* config) {
    // TODO 1: Create template directory structure with base and overlay dirs
    // TODO 2: Set up read-only base filesystem with required binaries
    // TODO 3: Pre-compile seccomp filter for template
    // TODO 4: Create template metadata file with configuration
    // TODO 5: Register template in global template registry
    // Hint: Use overlayfs with lowerdir (template) and upperdir (instance)
}

// Instantiate sandbox from template
int instantiate_from_template(const char* template_id, sandbox_state_t* state) {
    // TODO 1: Locate template in registry and increment reference count
    // TODO 2: Create per-instance overlay directory structure
    // TODO 3: Mount overlayfs with template as lowerdir
    // TODO 4: Apply template seccomp filter to current process
    // TODO 5: Configure cgroups based on template resource limits
}
```

```
// Clean up template instance
void cleanup_template_instance(sandbox_state_t* state) {
    // TODO 1: Unmount instance overlay filesystem
    // TODO 2: Remove per-instance overlay directories
    // TODO 3: Decrement template reference count
    // TODO 4: Clean up any template-specific resources
}
```

Metrics Collection System

```c
#include <time.h>
#include <pthread.h>

typedef struct {
    char metric_name[128];
    double value;
    time_t timestamp;
    char labels[512];  // JSON-formatted labels
} metric_sample_t;

typedef struct {
    metric_sample_t* samples;
    size_t capacity;
    size_t count;
    pthread_mutex_t mutex;
} metrics_buffer_t;

// Initialize metrics collection system
int metrics_init(const char* output_file) {
    // TODO 1: Initialize thread-safe metrics buffer with reasonable capacity
    // TODO 2: Create background thread for periodic metrics collection
    // TODO 3: Set up signal handlers for clean shutdown
    // TODO 4: Open output file for metrics exposition
    // TODO 5: Return 0 on success, negative on error
}

// Record security metric (e.g., seccomp violation)
void record_security_metric(const char* metric_name, const char* sandbox_id,
                            const char* violation_type) {
    // TODO 1: Create metric sample with current timestamp
    // TODO 2: Format labels JSON with sandbox_id and violation_type
```

```
    // TODO 3: Add sample to thread-safe metrics buffer
    // TODO 4: Check if buffer needs flushing to output
}

// Collect resource usage metrics from cgroups
void collect_resource_metrics(void) {
    // TODO 1: Iterate through all active sandbox cgroups
    // TODO 2: Read memory, CPU, and I/O usage from cgroup files
    // TODO 3: Record metrics with sandbox labels
    // TODO 4: Calculate utilization percentages relative to limits
}
```

Performance Optimization Framework

```c
typedef struct {
    int pool_size;
    int max_pool_size;
    sandbox_state_t** ready_processes;
    sandbox_state_t** busy_processes;
    pthread_mutex_t pool_mutex;
    pthread_cond_t process_available;
} process_pool_t;

// Initialize process pool for sandbox reuse
process_pool_t* init_process_pool(int initial_size, int max_size) {
    // TODO 1: Allocate process_pool_t structure and initialize mutex/condition
    // TODO 2: Pre-create initial_size sandbox processes in ready state
    // TODO 3: Initialize process arrays for ready and busy process tracking
    // TODO 4: Start background thread for pool health monitoring
    // TODO 5: Return initialized pool, NULL on error
}

// Get process from pool for task execution
sandbox_state_t* get_pooled_process(process_pool_t* pool,
                                    sandbox_config_t* config) {
    // TODO 1: Lock pool mutex and wait for available process
    // TODO 2: Move process from ready to busy array
    // TODO 3: Reset process state for new task (clear temp files, reset limits)
    // TODO 4: Apply task-specific configuration if different from template
    // TODO 5: Return configured process, unlock mutex
}

// Return process to pool after task completion
void return_pooled_process(process_pool_t* pool, sandbox_state_t* process) {
```

```
    // TODO 1: Clean up task-specific resources (files, network connections)
    // TODO 2: Verify process health (check memory leaks, zombie children)
    // TODO 3: Move process from busy to ready array
    // TODO 4: Signal process_available condition variable
    // TODO 5: Replace process if health check failed
}
```

Milestone Checkpoints

Advanced Security Integration Checkpoint:

- Verify Landlock rules prevent access to restricted files: `./test_landlock_restrictions`
- Confirm eBPF monitoring detects security events: `sudo ./security_monitor_test`
- Test LSM integration doesn't break existing functionality: `./integration_test_suite`

Performance Optimization Checkpoint:

- Measure sandbox creation latency improvement: `./benchmark_creation_time`
- Verify template instances maintain isolation: `./test_template_isolation`
- Confirm process pool provides performance benefit: `./benchmark_pool_vs_fork`

Management and Monitoring Checkpoint:

- Check metrics collection captures all event types: `curl localhost:9090/metrics`
- Verify policy updates apply correctly: `./test_policy_distribution`
- Test incident response triggers on security violations: `./simulate_security_incident`

These checkpoints ensure that advanced features enhance rather than compromise the security and functionality of the core sandbox implementation.

Glossary

> **Milestone(s):** This section provides definitions for all milestones (1-5), ensuring clear understanding of technical terms and Linux-specific concepts used throughout the process sandbox implementation.

The process sandbox leverages numerous Linux-specific security primitives and kernel mechanisms that may be unfamiliar to developers new to systems programming or security engineering. This glossary provides comprehensive definitions of all technical terms, data structures, security concepts, and Linux kernel mechanisms used throughout the sandbox design and implementation.

Mental Model: The Technical Dictionary

Think of this glossary as a **technical dictionary for a specialized security domain**. Just as learning a new language requires understanding vocabulary before constructing complex sentences, building a process sandbox requires mastering

the terminology of Linux security primitives before orchestrating them into a complete defense system. Each term builds upon others, creating a hierarchy of concepts from basic Linux primitives to advanced security mechanisms.

The terms are organized to reflect the conceptual layers of the sandbox: fundamental Linux concepts, security primitives, data structures, operational procedures, and debugging terminology. Understanding these definitions enables precise communication about sandbox behavior and systematic troubleshooting of security mechanisms.

Core Linux Security Concepts

Process sandbox: An isolated execution environment created using operating system primitives that restricts an untrusted program's access to system resources. The sandbox combines multiple Linux security mechanisms including namespaces, seccomp, cgroups, and capabilities to create defense-in-depth isolation. Unlike virtual machines which virtualize hardware, process sandboxes use kernel features to isolate processes while sharing the same kernel.

Defense-in-depth: A security strategy employing multiple independent layers of protection, where the failure of one security mechanism does not compromise the entire system. In the context of process sandboxing, this means combining namespace isolation, filesystem restrictions, system call filtering, resource limits, and privilege dropping so that an attacker must bypass all layers to escape the sandbox.

Namespace: A Linux kernel feature that provides isolated views of system resources, making processes believe they have exclusive access to resources that are actually shared. Each namespace type isolates a different category of system resource: PID namespaces isolate process trees, mount namespaces isolate filesystem views, network namespaces isolate network stacks, UTS namespaces isolate hostname and domain, user namespaces isolate user and group IDs, and cgroup namespaces isolate cgroup hierarchies.

Seccomp: A Linux kernel security feature that allows processes to restrict which system calls they can make through Berkeley Packet Filter (BPF) programs. Seccomp operates at the kernel level, intercepting system calls before they execute and applying filtering rules. The most common mode, seccomp-BPF, enables complex filtering based on system call numbers, arguments, and calling process characteristics.

Cgroups: The Linux control groups subsystem that provides resource limiting, accounting, and isolation for collections of processes. Cgroups organize processes into hierarchical groups and apply resource controls through specialized controllers for memory, CPU time, process count, I/O bandwidth, and other resources. Both cgroups v1 and v2 are commonly deployed, with different interfaces and capabilities.

Capabilities: Linux's fine-grained privilege system that divides the traditional root privileges into distinct units that can be independently granted or revoked. Instead of running as either privileged (root) or unprivileged (user), processes can hold specific capabilities like `CAP_NET_ADMIN` for network configuration or `CAP_SYS_PTRACE` for process debugging while lacking other dangerous privileges.

Filesystem Isolation Terminology

Chroot: A Linux operation that changes the apparent root directory for a process and its children, making the process unable to access files outside the new root hierarchy. While chroot provides basic filesystem isolation, it can be escaped by processes with sufficient privileges and is primarily useful as one layer of a multi-layered security approach.

Pivot_root: An atomic operation that swaps the root filesystem of a process, moving the old root to a specified location within the new root hierarchy. Unlike chroot, pivot_root operates on mounted filesystems and provides stronger isolation guarantees, making it the preferred mechanism for container and sandbox implementations.

Tmpfs: A memory-backed temporary filesystem that stores files in virtual memory instead of persistent storage. Tmpfs provides high-performance temporary storage that automatically disappears when unmounted, making it ideal for writable areas within otherwise read-only sandbox environments.

Mount namespace: A Linux namespace that provides each process group with an isolated view of filesystem mount points. Processes in different mount namespaces can mount and unmount filesystems independently without affecting other namespaces, enabling the creation of customized filesystem layouts for sandboxed processes.

Device files: Special files that provide interfaces to kernel drivers and hardware devices. Device files appear in the filesystem (typically under `/dev`) but reading or writing them communicates with kernel drivers rather than storing data on disk. Sandboxes must carefully control which device files are accessible to prevent hardware access and privilege escalation.

Overlay filesystem: A union filesystem that combines multiple directories into a single logical filesystem, typically with a read-only lower layer and a writable upper layer. Overlay filesystems enable efficient sandbox creation by providing each sandbox instance with a writable view of a shared read-only base system without duplicating files.

System Call and BPF Terminology

BPF: Berkeley Packet Filter, originally designed for network packet filtering but extended to provide a safe in-kernel execution environment for various purposes including seccomp system call filtering. BPF programs are executed in a virtual machine within the kernel, providing security through static analysis and runtime restrictions.

Whitelist: An explicit list of allowed system calls that a sandboxed process may invoke. Whitelisting follows the security principle of default deny, where only explicitly permitted system calls are allowed and all others result in process termination or error returns. This contrasts with blacklisting approaches that attempt to enumerate dangerous system calls.

System call: The interface through which user-space programs request services from the kernel, such as reading files, allocating memory, or creating network connections. System calls represent the boundary between user and kernel space, making them a critical enforcement point for security policies in process sandboxes.

Argument-level filtering: Advanced seccomp filtering that examines not just the system call number but also the arguments passed to the system call. This enables policies like "allow `open()` but only for files under `/tmp`" or "allow `socket()` but only for Unix domain sockets" by inspecting system call parameters.

Resource Management Terminology

Resource exhaustion: An attack pattern where malicious code attempts to consume excessive system resources (memory, CPU time, file descriptors, processes) to cause denial of service or trigger resource allocation failures that might lead to privilege escalation opportunities.

Hierarchical filesystem: The tree-structured organization of cgroups where child cgroups inherit and are constrained by the resource limits of their parent cgroups. This hierarchy enables sophisticated resource management policies and delegation of administrative responsibilities.

Controller: A cgroup subsystem that manages a specific type of resource, such as the memory controller for RAM usage limits, the CPU controller for processor time quotas, or the PID controller for process count restrictions. Each controller provides its own configuration files and enforcement mechanisms.

Quota: A specific allocation of resource consumption permitted to a process or cgroup. Quotas can be absolute limits (maximum 64MB of memory) or relative shares (10% of CPU time) and are enforced by the kernel to prevent resource exhaustion attacks.

Bandwidth throttling: The technique of limiting the rate at which processes can consume resources, particularly I/O bandwidth. Throttling prevents sudden bursts of resource consumption from affecting system performance and enables predictable resource sharing among multiple sandboxes.

Privilege and Capability Terminology

Effective set: The set of Linux capabilities that are currently active and can be used by a process. The effective set determines what privileged operations the process can actually perform at any given moment.

Permitted set: The maximum set of capabilities that a process may add to its effective set. The permitted set acts as a ceiling on the capabilities a process can activate, even if it possesses them in other sets.

Inheritable set: Capabilities that can be inherited by child processes when combined with the inheritable set of an executable file during `exec()`. The inheritable set provides controlled privilege transfer across process boundaries.

Bounding set: A system-wide or process-specific ceiling on capabilities that can be gained through any mechanism. The bounding set provides an ultimate limit on privilege escalation, even for processes that might otherwise gain capabilities through setuid executables or other means.

Ambient set: A simplified capability inheritance mechanism that allows unprivileged processes to retain specific capabilities across `exec()` boundaries without requiring special file attributes. Ambient capabilities automatically transfer to child processes when certain conditions are met.

No new privileges: A process flag set via `prctl(PR_SET_NO_NEW_PRIVS)` that prevents the process and its descendants from gaining additional privileges through any mechanism, including setuid executables, file capabilities, or SELinux transitions. This flag provides strong protection against privilege escalation attacks.

Privilege dropping: The systematic process of removing unnecessary Linux capabilities and changing to a lower-privileged user account to minimize the attack surface if the sandboxed process is compromised. Proper privilege dropping follows a specific sequence to avoid accidentally retaining dangerous capabilities.

Data Structure Definitions

The process sandbox employs numerous data structures to manage configuration, runtime state, and security mechanisms. Understanding these structures is essential for implementing and debugging the sandbox system.

Structure Name	Purpose	Key Fields
`sandbox_config_t`	Configuration parameters for sandbox creation	`program_path`, `program_args`, `chroot_path`, `memory_limit`, `cpu_percent`, `enable_network`, `allowed_syscalls`
`sandbox_state_t`	Runtime state and handles for active sandbox	`child_pid`, `namespace_fd[6]`, `cgroup_path`, `cleanup_needed`
`namespace_state_t`	File descriptors for namespace isolation	`pid_ns_fd`, `mount_ns_fd`, `net_ns_fd`, `uts_ns_fd`, `user_ns_fd`, `initialized`
`fs_isolation_state_t`	Filesystem isolation configuration and state	`sandbox_root`, `old_root_path`, `mount_ns_fd`, `cleanup_needed`
`cgroup_limits_t`	Resource limit specifications	`memory_limit_bytes`, `cpu_quota_us`, `cpu_period_us`, `pid_max`, `io_read_bps`, `io_write_bps`
`cgroup_state_t`	Cgroup management and cleanup state	`cgroup_path`, `instance_id`, `version`, `available_controllers`, `managed_pids`
`error_context_t`	Comprehensive error reporting information	`error_code`, `severity`, `category`, `component`, `operation`, `description`, `sandbox_pid`
`health_check_result_t`	Individual health check outcome	`check_name`, `status`, `message`, `response_time_ms`, `last_check_time`

Process Lifecycle and State Management

Orchestration: The coordination of multiple security components in the correct sequence to create a functional sandbox. Orchestration ensures that namespaces are created before filesystem isolation, that seccomp filters are applied before executing untrusted code, and that resource limits are configured before allowing process creation.

Lifecycle management: The systematic tracking of sandbox state from initial creation through final cleanup, including all intermediate states like configuration, execution, termination, and resource cleanup. Proper lifecycle management ensures that resources are allocated and deallocated correctly and that partial failures don't leave system resources in inconsistent states.

Coordination channel: Communication mechanisms between the parent supervisor process and sandboxed child processes, typically implemented using pipes, Unix domain sockets, or shared memory. Coordination channels enable status reporting, graceful shutdown signaling, and emergency termination commands.

Resource tracking: The maintenance of references to all allocated resources (file descriptors, mount points, cgroup hierarchies, temporary files) to ensure proper cleanup even in failure scenarios. Resource tracking prevents resource leaks that could accumulate over time and affect system stability.

Testing and Validation Terminology

Unit testing: Testing individual security components in isolation to verify their specific functionality without dependencies on other components. Unit tests for sandbox components might verify that namespace creation succeeds, that seccomp

filters block specific system calls, or that cgroup limits are properly enforced.

Integration testing: Testing the complete sandbox system with realistic workloads to verify that all security components work together correctly. Integration tests execute actual programs within sandboxes and verify that the expected isolation and resource constraints are maintained.

Security validation: Systematic testing of attack scenarios and bypass attempts to verify that security mechanisms cannot be circumvented. Security validation includes testing known container escape techniques, privilege escalation attacks, and resource exhaustion scenarios.

Penetration testing: Comprehensive security testing that simulates real attacks against the sandbox system, including both automated tools and manual exploitation attempts. Penetration testing validates that the defense-in-depth approach successfully prevents sandbox escapes and privilege escalation.

Debugging and Diagnostics

Symptom-based diagnosis: A systematic debugging approach that maps observable failure symptoms to likely root causes and provides specific diagnostic steps. This approach is particularly valuable for complex systems like process sandboxes where failures can have multiple contributing factors.

System call tracing: The monitoring and recording of all system calls made by a process, typically using tools like `strace` or in-kernel tracing mechanisms. System call tracing is essential for debugging seccomp filter configurations and understanding process behavior within sandboxes.

Namespace inspection: The examination of namespace configurations and isolation effectiveness by comparing the view of system resources from inside and outside the sandbox. Namespace inspection typically involves examining `/proc` filesystem entries and namespace file descriptors.

Resource monitoring: The continuous or periodic measurement of resource consumption within cgroups to verify that resource limits are properly configured and enforced. Resource monitoring helps diagnose performance issues and validate resource limit effectiveness.

Advanced Security Features

LSM: Linux Security Module, a framework that enables additional mandatory access control systems to be integrated into the kernel. LSMs like SELinux, AppArmor, and Smack provide policy-based access controls that complement the capability and namespace-based protections of process sandboxes.

Landlock: A modern Linux Security Module that enables unprivileged processes to create fine-grained security sandboxes for themselves and their children. Landlock provides programmable access control for filesystem operations and integrates well with existing sandbox mechanisms.

eBPF: Extended Berkeley Packet Filter, a powerful in-kernel programming framework that enables safe execution of user-defined programs within the kernel. eBPF can be used for advanced security monitoring, custom system call filtering, and performance monitoring of sandboxed processes.

Seccomp notify: A modern seccomp feature that allows user-space programs to handle filtered system calls instead of simply allowing or denying them. Seccomp notify enables sophisticated policies like "allow file access but log all operations" or "allow network access but restrict destinations."

Performance and Operational Terms

Template-based optimization: A performance enhancement technique that pre-creates sandbox environments and reuses them for multiple executions. Templates reduce sandbox creation overhead by preparing common configurations and filesystem layouts in advance.

Process pool: A resource management technique that maintains a collection of pre-configured processes ready for immediate task assignment. Process pools reduce the latency of sandbox creation by avoiding the overhead of repeated process creation and configuration.

Memory deduplication: An optimization technique that identifies and shares identical memory pages across multiple processes. In sandbox contexts, memory deduplication can significantly reduce memory overhead when running many similar sandbox instances.

Metrics collection: The systematic gathering of operational and security data from sandbox instances for monitoring, alerting, and analysis purposes. Metrics collection typically includes resource usage statistics, security events, performance measurements, and operational health indicators.

Error Handling and Recovery

Cleanup ordering: The specific sequence in which sandbox resources must be deallocated to handle dependencies correctly. For example, processes must be terminated before cgroups can be removed, and mount points must be unmounted before namespaces can be cleaned up.

Health monitoring: Continuous observation of sandbox operational status to detect failures, resource exhaustion, security violations, or other problems that require intervention. Health monitoring enables proactive response to issues before they cause complete sandbox failures.

Automated recovery: Systematic response to detected failures without requiring human intervention. Automated recovery might include restarting failed processes, recreating corrupted resources, or falling back to simpler configurations when advanced features fail.

Escalation: The process of triggering human attention when automated responses are insufficient to handle detected problems. Escalation policies define when and how operational staff should be notified of sandbox issues that require manual intervention.

Implementation Guidance

The process sandbox implementation relies on numerous Linux-specific system calls, kernel interfaces, and user-space utilities. Understanding how to correctly invoke these mechanisms and handle their error conditions is essential for building robust sandbox systems.

Essential System Calls and Library Functions

Function/System Call	Purpose	Critical Parameters	Common Pitfalls
`clone()`	Create child process with namespace isolation	`CLONE_NEWPID`, `CLONE_NEWMNT`, `CLONE_NEWNET`, `CLONE_NEWUTS`, `CLONE_NEWUSER`	Forgetting `SIGCHLD` flag, insufficient privileges for user namespaces
`unshare()`	Move calling process to new namespaces	Namespace flags, timing relative to other operations	Cannot unshare PID namespace from running process
`mount()`	Mount filesystems and bind mounts	`MS_BIND`, `MS_RDONLY`, `MS_NOSUID`, `MS_NODEV`, `MS_NOEXEC`	Mount propagation, order dependencies
`pivot_root()`	Atomically change root filesystem	Old and new root paths, mount point requirements	Requires mounted filesystems, complex setup sequence
`prctl()`	Set process attributes and security flags	`PR_SET_NO_NEW_PRIVS`, `PR_SET_SECCOMP`, capability operations	Operation ordering, irreversible changes
`seccomp()`	Install BPF system call filters	Filter programs, operation modes, flag combinations	Architecture-specific syscall numbers, missing required calls

File Structure and Organization

The sandbox implementation should be organized into clearly separated modules that handle specific security mechanisms and can be tested independently:

```
sandbox/
├── cmd/
│   └── sandbox-runner/
│       └── main.c                    # CLI interface and demonstration
├── lib/
│   ├── namespace/
│   │   ├── namespace.c               # Namespace creation and management
│   │   ├── namespace.h               # Public namespace API
│   │   └── namespace_test.c          # Unit tests
│   ├── filesystem/
│   │   ├── fs_isolation.c            # Chroot/pivot_root implementation
│   │   ├── fs_isolation.h            # Filesystem isolation API
│   │   ├── minimal_rootfs.c          # Minimal rootfs construction
│   │   └── fs_isolation_test.c       # Unit tests
│   ├── seccomp/
│   │   ├── seccomp_filter.c          # BPF filter creation and installation
│   │   ├── seccomp_filter.h          # Seccomp API definitions
│   │   ├── syscall_whitelist.c       # System call whitelist management
│   │   └── seccomp_test.c            # Filter testing
│   ├── cgroups/
│   │   ├── cgroup_manager.c          # Cgroup creation and limit setting
│   │   ├── cgroup_manager.h          # Resource management API
│   │   └── cgroup_test.c             # Resource limit testing
│   ├── capabilities/
│   │   ├── capability_drop.c         # Privilege dropping implementation
│   │   ├── capability_drop.h         # Capability management API
│   │   └── capability_test.c         # Privilege verification tests
│   ├── orchestration/
│   │   ├── sandbox_orchestrator.c    # Main coordination logic
│   │   ├── sandbox_orchestrator.h    # Public sandbox API
│   │   └── orchestration_test.c      # Integration tests
│   └── common/
│       ├── error_handling.c          # Error reporting and logging
│       ├── cleanup_manager.c         # Resource cleanup coordination
│       ├── health_monitor.c          # Sandbox health checking
│       └── debug_utils.c             # Debugging and diagnostics
├── tests/
│   ├── integration/
│   │   ├── full_sandbox_test.c       # End-to-end testing
│   │   ├── security_validation.c     # Attack scenario testing
│   │   └── performance_test.c        # Resource usage measurement
│   └── security/
│       ├── container_escape_test.c   # Escape attempt testing
│       ├── privilege_escalation.c    # Privilege escalation testing
│       └── resource_exhaustion.c     # Resource limit testing
├── scripts/
│   ├── setup_test_env.sh            # Test environment preparation
│   ├── create_minimal_rootfs.sh     # Rootfs creation automation
│   └── run_security_tests.sh        # Security validation automation
└── docs/
    ├── api_reference.md             # Complete API documentation
    ├── security_model.md           # Security assumptions and guarantees
    └── troubleshooting.md          # Common issues and solutions
```

Technology Recommendations

Component	Simple Option	Advanced Option	Recommendation Rationale
Build System	`make` with simple Makefile	`cmake` or `meson` build system	Make sufficient for learning, cmake better for production
Testing Framework	Manual test functions with assertions	`cmocka` or `criterion` testing framework	Start simple, upgrade for comprehensive testing
Logging	`printf()` and `syslog()`	Structured logging with `journald`	Simple logging adequate for development
Configuration	Command-line arguments	JSON/YAML configuration files	CLI args easier for initial implementation
Inter-Process Communication	Unix pipes and signals	D-Bus or custom protocol	Pipes sufficient for sandbox coordination
Error Handling	Return codes and `errno`	Exception-like error propagation	Return codes match Linux system call conventions

Core Implementation Skeleton

The main sandbox orchestration function provides the entry point for coordinating all security mechanisms:

```c
/*
 * create_sandbox - Create a complete process sandbox with all security layers
 * @config: Sandbox configuration including program, limits, and restrictions
 * @state: Runtime state structure to populate with handles and cleanup info
 *
 * Returns: SANDBOX_SUCCESS on success, specific error code on failure
 *
 * This function orchestrates the creation of all security layers in the correct
 * sequence. It must be called with sufficient privileges (typically root) and
 * handles both success and failure scenarios with proper resource cleanup.
 */
int create_sandbox(sandbox_config_t *config, sandbox_state_t *state) {
    // TODO 1: Validate configuration parameters and check prerequisites
    // TODO 2: Initialize cleanup manager to track allocated resources
    // TODO 3: Create child process with clone() and namespace flags
    // TODO 4: In parent: set up cgroup hierarchy and add child to cgroup
    // TODO 5: In child: configure filesystem isolation with pivot_root
    // TODO 6: In child: install seccomp filter before executing target program
    // TODO 7: In child: drop capabilities and change user/group
    // TODO 8: In child: exec target program with configured environment
    // TODO 9: In parent: monitor child and handle lifecycle events
    // TODO 10: In parent: return control to caller with populated state
    // Hint: Use SIGCHLD handler to detect child termination
    // Hint: Store namespace file descriptors for later inspection
}

/*
 * setup_namespace_isolation - Create isolated namespaces for sandbox
 * @ns_state: Namespace state structure to populate with file descriptors
 *
```

```
 * Returns: SANDBOX_SUCCESS or SANDBOX_ERROR_NAMESPACE
 *
 * Creates PID, mount, network, UTS, and user namespaces to isolate the
 * sandbox from the host system. Must be called before filesystem isolation.
 */
int setup_namespace_isolation(namespace_state_t *ns_state) {
    // TODO 1: Check if we have sufficient privileges for namespace creation
    // TODO 2: Create user namespace first to enable other namespace creation
    // TODO 3: Set up user/group ID mapping in user namespace
    // TODO 4: Create PID namespace for process isolation
    // TODO 5: Create mount namespace for filesystem isolation
    // TODO 6: Create network namespace to block network access
    // TODO 7: Create UTS namespace to isolate hostname
    // TODO 8: Store namespace file descriptors for cleanup and inspection
    // Hint: Use /proc/self/ns/* to obtain namespace file descriptors
    // Hint: User namespace must be created before others in unprivileged mode
}

/*
 * build_seccomp_filter - Generate BPF program from system call whitelist
 * @allowed_syscalls: Null-terminated array of allowed system call names
 * @filter_program: Output pointer for BPF instruction array
 * @program_len: Output pointer for instruction count
 *
 * Returns: SANDBOX_SUCCESS or SANDBOX_ERROR_SECCOMP
 *
 * Translates human-readable system call names into BPF instructions that
 * implement a whitelist filter, allowing only specified calls.
 */
int build_seccomp_filter(char **allowed_syscalls, struct sock_filter **filter_program,
```

```
                      int *program_len) {
    // TODO 1: Validate that allowed_syscalls contains required basic calls
    // TODO 2: Resolve system call names to numbers for current architecture
    // TODO 3: Generate BPF instructions for whitelist logic
    // TODO 4: Add architecture validation to prevent cross-arch attacks
    // TODO 5: Add argument-level filtering for dangerous but needed syscalls
    // TODO 6: Optimize instruction order for common system calls
    // Hint: Use __NR_syscall macros for system call numbers
    // Hint: BPF_STMT and BPF_JUMP macros simplify instruction generation
}
```

Milestone Checkpoints

Milestone 1 Checkpoint (Namespace Isolation): After implementing namespace creation, verify isolation by running:

```bash
sudo ./sandbox-runner /bin/bash
# Inside sandbox, run:
ps aux              # Should only show PID 1 and children
mount | grep proc   # Should show isolated /proc mount
hostname            # Should show isolated hostname
ip addr show        # Should show only loopback interface
```

Milestone 2 Checkpoint (Filesystem Isolation): Verify filesystem containment by testing:

```bash
sudo ./sandbox-runner /bin/ls /
# Should only show minimal rootfs contents, not host filesystem
sudo ./sandbox-runner /bin/cat /proc/mounts
# Should show tmpfs, proc, and minimal mounts only
sudo ./sandbox-runner /bin/touch /tmp/test
# Should succeed - writable tmpfs
sudo ./sandbox-runner /bin/touch /bin/test
# Should fail - read-only root
```

Milestone 3 Checkpoint (Seccomp Filtering): Test system call filtering effectiveness:

```bash
sudo ./sandbox-runner /bin/cat /etc/passwd

# Should work - read operations allowed

sudo ./sandbox-runner /usr/bin/strace ls

# Should fail - ptrace family blocked

sudo ./sandbox-runner /bin/chmod +x /tmp/file

# Should fail if chmod not in whitelist
```

Debugging Tools and Techniques

Symptom	Likely Cause	Diagnostic Command	Solution
`clone()` fails with `EPERM`	Insufficient privileges for namespace creation	Check `id`, verify `CAP_SYS_ADMIN`	Run with root or configure user namespaces
Sandbox sees host PID tree	PID namespace not created or not entered	`ls -la /proc/self/ns/pid` from inside sandbox	Verify `CLONE_NEWPID` flag in `clone()`
Seccomp kills on basic operations	Missing required system calls in whitelist	`strace -f ./sandbox-runner program`	Add missing syscalls to allowed list
Cgroup limits not enforced	Controller not enabled or wrong cgroup version	`cat /proc/cgroups`, check `/sys/fs/cgroup`	Enable controllers, check cgroup mount
Capabilities not dropped	Wrong order of setuid/capability operations	`getpcaps $$` from inside sandbox	Drop capabilities before changing UID

Security Validation Tests

Essential security tests to verify sandbox effectiveness:

```c
/*
 * Test that attempts common container escape techniques and verifies
 * that they are properly blocked by the sandbox security layers.
 */
int test_container_escape_attempts(void) {
    // TODO 1: Test /proc/*/root traversal escape
    // TODO 2: Test device node creation and access
    // TODO 3: Test mount manipulation attacks
    // TODO 4: Test namespace escape via file descriptors
    // TODO 5: Test cgroup escape via cgroup.procs manipulation
    // Hint: All escape attempts should fail with permission denied
    // Hint: Use child processes to isolate dangerous test operations
}

/*
 * Verify that resource limits are properly enforced and cannot be bypassed
 * through various resource exhaustion techniques.
 */
int test_resource_limit_enforcement(void) {
    // TODO 1: Attempt to allocate memory beyond configured limit
    // TODO 2: Attempt to create processes beyond PID limit
    // TODO 3: Attempt to consume CPU time beyond quota
    // TODO 4: Attempt to exhaust I/O bandwidth beyond limit
    // TODO 5: Verify that OOM killer or throttling occurs as expected
    // Hint: Use fork bombs and memory allocation loops for testing
    // Hint: Monitor /sys/fs/cgroup/*/memory.events for OOM events
}
```