rojocOS - An attempt at an efficient OS for cryptography

Romain de Coudenhove, Johan Utterström and Constantin Vaillant-Tenzer

May 15, 2025

Table of Contents

Making an OS

"No one who isn't already a seasoned developer with years of experience in several languages and environments should even be considering OS Dev yet. A decade of programming, including a few years of low-level coding in assembly language and/or a systems language such as C, is pretty much the minimum necessary to even understand the topic well enough to work in it." (`wiki.osdev.org/Beginner_Mistakes`)

Our goal

Modify and improve an WeensyOS system to make it compatible with instant disk encryption (Sabt, Achemlal, and Bouabdallah 2015).

- **Instant Disk Encryption**: Files encrypted with AES using public keys derived from entropy pool.
- **File Deletion**: Quickly and securely remove files
- **Randomness**: Enhanced entropy by a user 16 bits key generation.
- **Secure Memory**: Kernel allocator with zeroing and alignment.

Open source code available on `https://github.com/cvt8/rojocos` (from May 18, 2025).

rojocOS Overview

- **Filesystem**: Tree-based, 1376-byte nodes, AES encryption with public keys, supports file deletion.
- **Randomness**: Entropy from keystrokes and TSC, used for public key generation.
- **Memory Allocator**: Secure kernel heap with zeroing and alignment.
- **User Programs**: Linux-like commands (`ls`, `cat`, `plane`) and shell.

Table of Contents

Why we upgraded randomness?

- **Old state**: ‘rand()‘ was a Linear congruential generator that was always seeded with a fixed constant ⇒ identical “random” stream each boot.
- **Our requirements**: unpredictable seeds for safe file storage, potential crypto applications, and still run on QEMU with *no extra hardware*.
- **Solution sketch**:
  1. Collect human entropy from keyboard timing and `rdtsc` (CPU cycle count)
  2. Keep 16-byte pool in kernel; and blend in with new cycle count randomness.
  3. New syscall **sys_getrandom**
  4. Backward compatible: kernel and user
- **Key design choice**: simple XOR mixing + feed into LCG algorithm for speed; but note that this is *not* a cryptographically secure random number generator.

Kernel implementation (entropy path)

- `entropy.c/h` *new module*
  - `request_user_entropy()`: blocking boot prompt, gathers 16 keystrokes.
  - `refresh_entropy()`: re-prompts after 10 000 outputs (for convenience, should be lower for security).
  - `get_entropy_value()`:
    - Each key byte $b_i = \text{ASCII} \oplus \text{TSC}_{7:0} \oplus \text{TSC}_{15:8} \oplus \text{timing}$
    - i.e. keystroke + low and next-low bits at moment key is read + software counter incremented while polling for key
    - XOR: random if at least one is random
    - Diffusion pass: $b_i \mathrel{\hat{=}} b_{(i+7) \bmod 16}$ to avoid local bias
    - On read, select 4 bytes starting at (TSC & 15) and XOR with live TSC.
- `kernel.c`
  - Boot: call `request_user_entropy()` before paging test.

User–space integration

- `lib.c`
  - First call to `rand()` now:

    kernel build: srand(get_entropy_value()) | user build: srand(sys_getrandom())

  - LCG step from original OS retained: $x_{n+1} = 1664525\,x_n + 1013904223 \pmod{2^{32}}$.
- **Demo program** `p-entropy.c`: prints 32-bit numbers, yields every 50 iterations—verifies stream differs across boots. Asks for new entropy when it is used up (we can set this parameter manually)
- **Build changes**: add `entropy.o` to kernel objects; add `p-entropy` target to process list.

Limitations and future work

Downsides

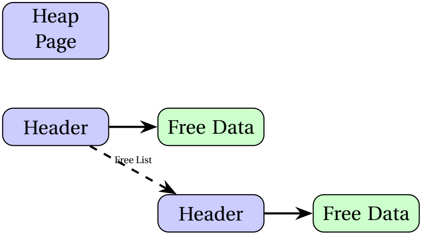
- Requires human keystrokes ⇒ blocks unattended boots / CI.
- 128-bit pool with XOR mixing is not cryptographically secure
- Not safe from attacks. A single sys getrandom() call (or any kernel-memory leak) reveals the raw XOR-based pool

Next steps

- Gather extra sources of entropy (disk IRQ, network jitter, hardware randomness).
- Ideally: background entropy daemon; boot proceeds without blocking Entropy pool continually updates

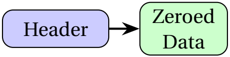
Overview and Architecture

- **Purpose**: Secure memory allocator for WeensyOS kernel, inspired by iPhone's memory protection.
- **Heap Structure**:
  - 4KB pages via `page_alloc(PO_KERNEL_HEAP)`.
  - Blocks with headers (`size`, `next`).
  - Free list with first-fit strategy.
- **Functions**: `kernel_malloc`, `kernel_free`, `extend_heap`.



Security Features

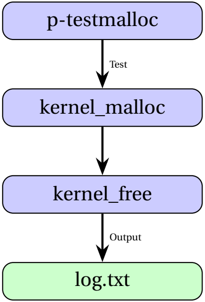
- **Memory Zeroing**: Allocated and freed memory is zeroed to prevent data leaks.
- **16-Byte Alignment**: Supports encryption (e.g., AES) and SIMD operations.
- **Validation**: Checks sizes and pointers to prevent overflows and double-frees.
- **Encryption Compatibility**: Designed for future cryptographic integration.



Inspired by iPhone's Secure Enclave

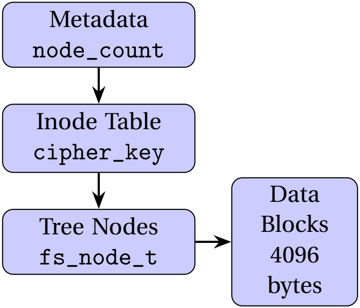
Testing the Implementation

- **Test Function**: `testmalloc` runs:
  - Single allocation/free (100 bytes).
  - Multiple allocations (200, 300 bytes).
  - Large allocation (2048 bytes).
  - Stress test (10x 50 bytes).
  - Zero-size and custom-size tests.
- **Automatic Execution**: Via `p-testmalloc` on boot.
- **Verification**: Logs in `log.txt`, memory maps on CGA console.



Key Features of rojocOS Filesystem

- **Tree-Based Organization**: Hierarchical structure rooted at '/'.
- **Fixed-Size Nodes**: `fs_node_t` (1376 bytes), up to 32 children.
- **AES Encryption**: Files encrypted with 256-byte `cipher_key` using public keys from entropy pool.
- **File Deletion**: Securely remove files via `INT_SYS_UNLINK`.
- **Rich System Calls**: `open`, `read`, `write`, `mkdir`, `unlink`, etc.
- **Linux-Like Commands**: `ls`, `touch`, `cat`, `cd`, `mkdir`, `plane`.

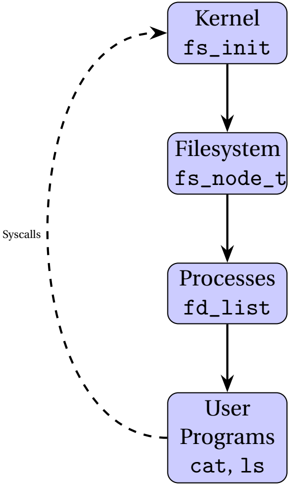


Benefits and Design Goals

- **Simplicity**: Minimalist design and small codebase.
- **Efficiency**: Fixed-size nodes (1376 bytes) and 4096-byte blocks optimize disk usage.
- **Security**: AES encryption with public keys (partial implementation via `cipher_key`).
- **Extensibility**: Easy to add new syscalls or user programs.

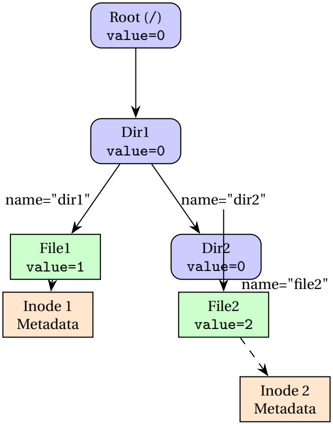
Filesystem Integration

- **Kernel Initialization**: `fs_init` sets up filesystem with disk callbacks.
- **Process Access**: File descriptors (`proc_fdlist_t`) manage open files.
- **Encryption**: Public keys from entropy pool used for AES key exchange.
- **User Programs**: `cat`, `ls`, etc., use `sys_open()`, `sys_read()`, `sys_write()`.



Filesystem Structure

- **Tree Structure**: Hierarchical, rooted at `/`.
- **Node (`fs_node_t`, 1376B)**:
  - `used` (1B): Node status.
  - `value` (4B): Inode index or 0 (dirs).
  - `children` (32 entries): 32B name, 4B index each.
- **Storage**:
  - `tree_usage_offset`: Tracks node availability.
  - `tree_offset`: Stores nodes.
- **Inodes**: Metadata (start block, block count, 256B AES key).
- **Data Blocks**: 4096B, AES-encrypted.



System Calls

- **INT_SYS_OPEN**: Opens a file, returns file descriptor; uses `fs_getattr`.
- **INT_SYS_READ/WRITE**: Reads/writes encrypted file data; updates descriptor offset.
- **INT_SYS_MKDIR**: Creates directory via `fs_touch` (`value=0`).
- **INT_SYS_TOUCH**: Creates file, allocates inode.
- **INT_SYS_UNLINK**: Deletes file, frees inode and blocks.
- **INT_SYS_LISTDIR**: Lists directory contents.
- **INT_SYS_CHDIR/GETCWD**: Manages working directory.

Example: File Operations

syscall_open("/file1") → open file.
syscall_unlink("/file1") → deletes file.

User Programs

- **plane**: Text editor; opens files, reads/writes content.
- **shell**: Executes commands via `INT_SYS_EXECV`.
- **mkdir**: Creates directories with `INT_SYS_MKDIR`.
- **ls**: Lists files using `INT_SYS_LISTDIR`.
- **cd**: Changes directory with `INT_SYS_CHDIR`.
- **touch**: Creates files via `INT_SYS_TOUCH`.
- **cat**: Displays file contents using `INT_SYS_READ`.

Shell Example

$ ls
file1 dir1
$ touch file2
$ cat file1
Hello, WeensyOS!

Process Management: Fork Syscall

- **INT_SYS_FORK**: Creates a child process:
  - Allocates new page table for child.
  - Copies parent's memory pages; shared pages (e.g., kernel) are mapped directly.
  - Child inherits parent's registers, cwd, but returns 0; parent returns child PID.
- **File Descriptors**: Managed via `proc_fdlist_t`, copied during fork.
- **INT_SYS_EXECV**: Loads new program, preserves arguments.
- **INT_SYS_WAIT/FORGET**: Waits for child exit, frees child resources.

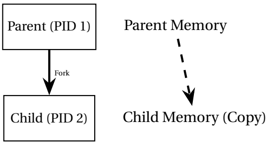


Table of Contents

Performances tests

Let's make a demo !

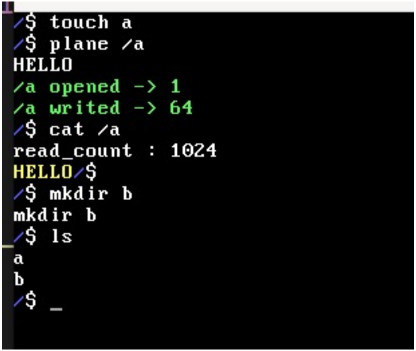


Table of Contents

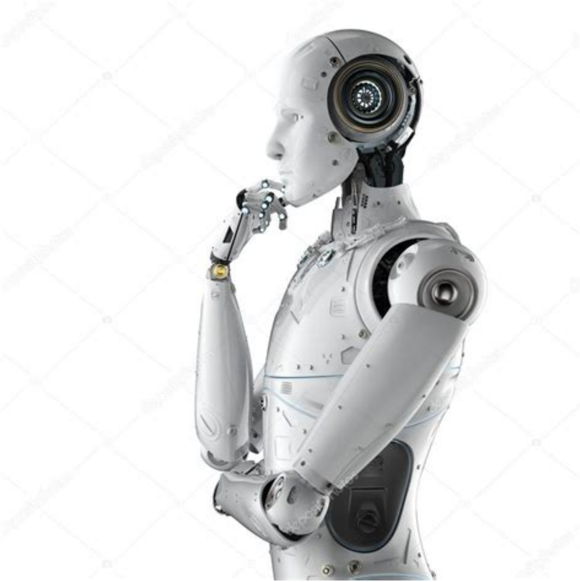
Key Takeaways

- **Secure Filesystem**: Tree-based, AES-encrypted files with public keys, supports deletion.
- **Randomness**: Entropy pool for public key generation, though not cryptographically secure.
- **Memory Allocator**: Secure kernel heap with zeroing and alignment.
- **User Experience**: Linux-like commands and shell for usability.
- **Foundation for Cryptography**: First steps toward a secure OS.

References I

Sabt, Mohamed, Mohammed Achemlal, and Abdelmadjid Bouabdallah (2015). “Trusted execution environment: What it is, and what it is not”. In: *2015 IEEE Trustcom/BigDataSE/Ispa*. Vol. 1. IEEE, pp. 57–64.

Questions



Discussion