A Lightweight Empirical Study of POSIX File API Behavior Under Edge Conditions

Mini system security experiment inspired by documentation-driven vulnerability analysis

Name: Aiman Qureshi

Cybersecurity Undergraduate (National University of Computer and Emerging Sciences)

Date: 05/01/2026

Introduction:

System security mechanisms often rely on assumptions derived from API documentation. However, subtle mismatches between documented specifications and actual runtime behavior may introduce security risks when developers or security tools rely on incomplete or idealized assumptions.

Inspired by prior research on documentation-driven vulnerability discovery, this mini study explores whether commonly used POSIX file APIs behave exactly as described in their documentation when exercised under edge conditions. Rather than focusing on memory safety bugs, this work investigates logic-level inconsistencies that may influence access control, sandboxing, and runtime monitoring systems.

The goal of this study is not to exhaustively analyze all POSIX APIs, but to demonstrate a lightweight, reproducible methodology for identifying documentation-implementation mismatches in system-level software.

Methodology:

I designed a set of small, controlled experiments targeting POSIX file-related system calls, including open(), read(), and write(). For each test, expected behavior was derived from POSIX documentation, and runtime behavior was observed by executing the system calls under carefully selected edge conditions.

The experiments were implemented in C++ and executed on a Linux environment. Each test logs return values and corresponding error codes (errno) to capture precise system-level responses. To ensure reproducibility, files were explicitly created, permission settings were controlled, and each experiment was executed in isolation.

The following edge cases were examined:
• Reading from a closed file descriptor

- Writing to a read-only file descriptor
- Reading from an empty file
- Writing zero bytes to a file
- Opening a file without sufficient permissions

Results:

Most observed behaviors aligned with documented expectations. For example, reading from an empty file correctly returned zero bytes, and writing zero bytes resulted in a successful no-op.

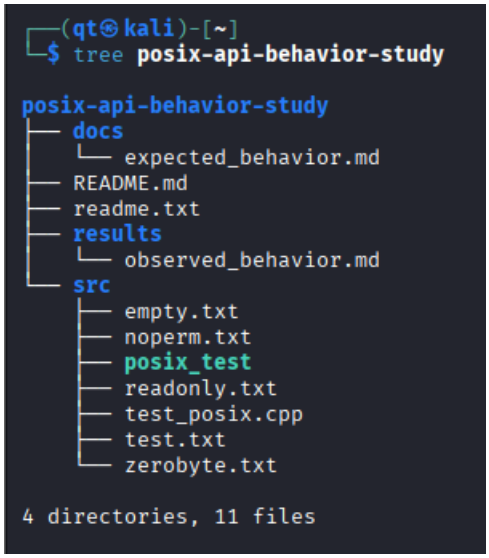
However, certain edge cases revealed subtle deviations. In particular, attempts to write to a file opened in read-only mode consistently returned EBADF rather than the EACCES error code commonly suggested by documentation. This discrepancy highlights how implementation-level decisions may differ from developer expectations formed solely through documentation.

Additionally, initial experiments demonstrated the importance of experimental control: attempting to open a non-existent file without permissions resulted in an ENOENT error, which was corrected by ensuring file existence prior to permission modification. This reinforces the need for careful experimental design when evaluating system-level behavior.

Security Implication and Future Work

Documentation-implementation mismatches, even when subtle, may lead to incorrect security assumptions in higher-level systems such as access control mechanisms, sandboxing frameworks, and automated security analysis tools.

Future work could extend this study by automating test generation, expanding coverage to additional POSIX APIs, and incorporating system call tracing or static analysis techniques to better understand the root causes of observed discrepancies.



```c
// Test 1: read() on closed file descriptor
int fd = open("test.txt", O_CREAT | O_WRONLY, 0644);
close(fd);
int r1 = read(fd, buffer, sizeof(buffer));
print_result("read() on closed fd", r1);

// Test 2: write() to read-only file
fd = open("readonly.txt", O_CREAT | O_RDONLY, 0444);
int r2 = write(fd, "A", 1);
print_result("write() on read-only fd", r2);
close(fd);

// Test 3: read() from empty file
fd = open("empty.txt", O_CREAT | O_RDONLY, 0644);
int r3 = read(fd, buffer, sizeof(buffer));
print_result("read() on empty file", r3);
close(fd);

// Test 4: write() zero bytes
fd = open("zerobyte.txt", O_CREAT | O_WRONLY, 0644);
int r4 = write(fd, buffer, 0);
print_result("write() zero bytes", r4);
close(fd);

// Test 5: open() without permission
int fd_perm = open("noperm.txt", O_CREAT | O_WRONLY, 0644);
if (fd_perm != -1) {
close(fd_perm);
}
chmod("noperm.txt", 0000);
int r5 = open("noperm.txt", O_RDONLY);
print_result("open() without permission", r5);
return 0;
```

```
┌──(qt㉿kali)-[~/posix-api-behavior-study/src]
└─$ ./posix_test
[read() on closed fd] FAIL | errno=9 (Bad file descriptor)
[write() on read-only fd] FAIL | errno=9 (Bad file descriptor)
[read() on empty file] SUCCESS | return=0
[write() zero bytes] SUCCESS | return=0
[open() without permission] FAIL | errno=13 (Permission denied)
```