Attack Surface and Hardening Report for `time_daemon.c`

Scott Nguyen

1 Introduction

This report presents an analysis of the attack surface of a simple Linux daemon program written in C, `time_daemon.c`, which logs the current time to syslog every second. The daemon is launched at boot, detaches from the terminal, and gracefully terminates on signal. This report outlines the potential attack surfaces, their associated risks, and presents a hardened version of the daemon with security best practices in place.

2 Attack Surface Model

An attack surface is the total set of points where an unauthorized user could try to enter data into or extract data from a program. For `time_daemon`, the attack surface includes:

- **Signal Handling:** The daemon listens for `SIGTERM` and `SIGINT`. These signals can be sent externally, potentially allowing unauthorized shutdowns.
- **Environment Inheritance:** Any untrusted or malicious environment variables passed to the process could be misused internally or leak information.
- **Syslog Logging:** Unescaped or poorly formatted log messages may allow log injection.
- **File Descriptors:** The daemon closes STDIN, STDOUT, and STDERR, but may still inherit other file descriptors from the parent.
- **Privileges:** If the daemon starts as root, it retains root access throughout its execution, increasing risk in the event of compromise.
- **Resource Usage:** Unbounded resource use (e.g., memory, CPU) could degrade system performance or be exploited for denial-of-service.

3 Security Risks Identified

1. **Lack of Privilege Dropping:** The daemon did not previously drop root privileges after daemonization.
2. **Unsafe Signal Handling:** Use of `signal()` was non-portable and less secure.
3. **No Resource Limits:** There were no constraints on memory, file descriptors, or CPU usage.

4. **Insecure Logging:** Log messages were not verified or sanitized, potentially allowing log injection.
5. **Thread-Unsafe Time Conversion:** The use of `localtime()` was not thread-safe.
6. **No Umask Set:** The default umask could allow world-writable files to be created.

4 Hardened Implementation Summary

The following changes were implemented to reduce the daemon's attack surface:

- **Replaced `signal()` with `sigaction()`:** Provides safer and more predictable signal handling.
- **Dropped privileges after daemonization:** The process switches to the `nobody` user using `setuid(65534)`.
- **Set Resource Limits:** Memory and file descriptor usage are now capped using `setrlimit()` to prevent abuse.
- **Used `localtime_r()`:** The thread-safe alternative to `localtime()`.
- **Enabled safer logging:** Formatting logic ensures structured messages and prevents malformed entries.
- **Set restrictive `umask`:** Prevents created files from being world-readable or writable.

These changes ensure that the daemon is more resilient against both accidental misuse and targeted attacks, following secure coding best practices in embedded Linux environments.

5 Additional Attack Surface Reflections

In this section, I address instructor feedback to gain a deeper understanding of Attack Surfaces.

5.1 Operational Misuse: Fork Bombing

The daemon currently does not restrict multiple simultaneous executions. A malicious or misconfigured user could repeatedly start the daemon, leading to a denial-of-service via resource exhaustion (a fork bomb). While resource limits (`setrlimit()`) help constrain each instance, they do not prevent uncontrolled forking system-wide. A more robust defense would include the use of a PID or lock file to ensure a single running instance, or configuring the service manager (e.g., `init`) to disallow duplicate launches.

5.2 Dynamic Library Hijacking

Since the daemon links dynamically against shared libraries (e.g., `libc`, `syslog`), it may be vulnerable to shared object hijacking if an attacker manipulates environment variables like `LD_LIBRARY_PATH`, or replaces system libraries in writable paths. While this is unlikely in locked-down embedded environments, further protection can be achieved by clearing sensitive environment variables at startup, avoiding unsafe runtime paths, and optionally using static linking for critical binaries.

5.3 Lifecycle Malware Insertion

The daemon's security assumes integrity of its build and deployment lifecycle. Without safeguards, a compromised build environment or update mechanism could allow an attacker to introduce malicious logic. To defend against this, production systems should adopt secure build pipelines, use cryptographic signing of binaries, and verify checksums during deployment or updates. This ensures that only trusted software versions are executed on the target system.

6 Conclusion

By examining and reducing the attack surface of `time_daemon.c`, the system is now hardened against common threats like privilege misuse, uncontrolled signals, resource exhaustion, and logging vulnerabilities. This exercise demonstrates how even simple programs can be made significantly more secure with careful analysis and incremental improvements.

Appendix: Source Code

```c
// cc -D_FORTIFY_SOURCE=2 -O2 -Wall -Wextra -o time_daemon time_daemon.c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <syslog.h>
#include <time.h>
#include <signal.h>
#include <string.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/resource.h>

static volatile sig_atomic_t running = 1;

void signal_handler(int sig) {
    if (sig == SIGTERM || sig == SIGINT) {
        running = 0;
    }
}

// Minimal privilege drop function
void drop_privileges() {
    if (setuid(65534) != 0) {  // 65534 = nobody user on most systems
        syslog(LOG_ERR, "Failed to drop privileges: %s", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

// Set resource limits (defensive)
void set_limits() {
    struct rlimit rl;

    rl.rlim_cur = 10 * 1024 * 1024;  // 10 MB max memory usage
    rl.rlim_max = 10 * 1024 * 1024;
    setrlimit(RLIMIT_AS, &rl);

    rl.rlim_cur = 1024;  // max open files
    rl.rlim_max = 1024;
    setrlimit(RLIMIT_NOFILE, &rl);
}

int main() {
    pid_t pid, sid;

    // Fork the parent process
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
```

```
        exit(EXIT_FAILURE);
    }
    if (pid > 0) {
        exit(EXIT_SUCCESS);
    }

    // Create a new session ID
    sid = setsid();
    if (sid < 0) {
        perror("setsid failed");
        exit(EXIT_FAILURE);
    }

    // Change working directory to /
    if (chdir("/") < 0) {
        perror("chdir failed");
        exit(EXIT_FAILURE);
    }

    // Close file descriptors
    close(STDIN_FILENO);
    close(STDOUT_FILENO);
    close(STDERR_FILENO);

    // Set restrictive umask
    umask(027);

    // Signal handling
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = signal_handler;
    sigaction(SIGTERM, &sa, NULL);
    sigaction(SIGINT, &sa, NULL);

    // Set resource limits
    set_limits();

    // Open syslog
    openlog("time_daemon", LOG_PID | LOG_CONS, LOG_DAEMON);
    syslog(LOG_INFO, "Daemon started securely");

    // Drop privileges
    drop_privileges();

    while (running) {
        time_t now = time(NULL);
        if (now == ((time_t) -1)) {
            syslog(LOG_ERR, "Failed to get current time");
        } else {
            struct tm ptm;
            char time_str[64];

            if (localtime_r(&now, &ptm) == NULL) {
                syslog(LOG_ERR, "Failed to convert time");
```

```
106             } else if (strftime(time_str, sizeof(time_str), "%Y-%m-%d %H:%
                    M:%S", &ptm) == 0) {
107                 syslog(LOG_ERR, "strftime returned 0");
108             } else {
109                 syslog(LOG_INFO, "Current time: %s", time_str);
110             }
111         }
112         sleep(1);
113     }
114
115     syslog(LOG_INFO, "Daemon terminated");
116     closelog();
117
118     return EXIT_SUCCESS;
119 }
```

Listing 1: `time_daemon.c` (Hardened Version)