

SECURE HOSPITAL FILE ACCESS MONITORING SYSTEM

A MINI-PROJECT REPORT

Submitted by:

- Aakash S

- Guru Sai Charan D

In partial fulfillment of the requirements for the award of the degree
BACHELOR OF ENGINEERING
IN COMPUTER SCIENCE AND ENGINEERING



RAJALAKSHMI ENGINEERING COLLEGE, CHENNAI

An Autonomous Institute

APRIL 2025

BONAFIDE CERTIFICATE

Certified that this project titled “Secure Hospital File Access Monitoring System” is the bonafide work of:

- Aakash S
- Guru Sai Charan D

who carried out the project under my supervision.

Submitted for the Practical Examination held on _____

SIGNATURE

Ms.V.JANANEE

Assistant Professor (SG),

Computer Science and Engineering,

Rajalakshmi Engineering College (Autonomous),

Thandalam, Chennai - 602 105

INTERNAL EXAMINER

EXTERNAL EXAMINER

ABSTRACT

In critical sectors such as healthcare, protecting sensitive data such as patient records is paramount. This project implements a Secure Hospital File Access Monitor using system call monitoring in C, a simulated RAM log using LRU/FIFO, and a real-time graphical interface in Python.

The system logs access to files based on system calls, assigns PIDs, checks permission levels, and simulates memory behavior through FIFO and LRU strategies. This bridges real-world hospital file management with essential Operating System concepts like system call tracing and memory management, providing both practical and educational value.

TABLE OF CONTENT

Chapter	Title	Page No.
1	Introduction	1
2	System Specifications	2
3	Module Description	3
4	Code Explanation (C + Python)	5
5	Screenshots and Output	10
6	Conclusion and Future Enhancement	13
7	References	14

CHAPTER 1 — INTRODUCTION

With increasing cyberattacks on healthcare systems, protecting sensitive hospital data such as patient reports, prescriptions, and lab results has become more crucial than ever. Hospitals operate with a wide range of departments and devices, all of which may access critical files. Therefore, a system is required to monitor, log, and visualize these file access events in real-time, ensuring that only authorized processes are allowed to interact with sensitive data.

This project simulates a secure hospital data access system using Operating System concepts like **system call monitoring** and **memory management**. We implement a system in C that simulates system call- level monitoring of file accesses, logging every access along with the process ID (PID), file name, status (ALLOWED / DENIED), and timestamp.

Additionally, we introduce a simulated memory log using Python to reflect how the OS handles memory under constraints. We use two classic page replacement algorithms — **FIFO (First-In First-Out)** and **LRU (Least Recently Used)** — to maintain a rolling log of the last five file accesses. These logs are presented in a colourful, real-time GUI built using PyQt5 and Matplotlib.

CHAPTER 2 — SYSTEM SPECIFICATIONS

2.1 HARDWARE REQUIREMENTS

- Intel i5 or above
- 8 GB RAM
- 100 GB HDD/SSD

2.2 SOFTWARE REQUIREMENTS

- OS: Kali Linux / Ubuntu / Windows WSL
- Language: C, Python 3
- Libraries: PyQt5, matplotlib, deque
- Editor: VS Code / Terminal

CHAPTER 3 — MODULE DESCRIPTION

This module simulates a low-level system call that checks whether a process (identified by its PID) is allowed to access a particular file.

Written in C, it reads an allowed_files.txt list, attempts to open the requested file, and logs the result in access.log. Each log entry includes:

- The access status (ALLOWED / DENIED)
- Filename
- Process ID (PID)
- Timestamp

This mimics basic syscall tracing behaviour found in Linux auditing tools like auditd or ptrace.

The Python module simulates a simplified version of the Operating System's **page replacement mechanism**. It maintains the last five file accesses using either:

- **FIFO (First-In First-Out)**: Evicts the earliest accessed file.
- **LRU (Least Recently Used)**: Evicts the file that hasn't been accessed in the longest time.

These logs simulate a “RAM-like” cache and reflect how limited memory is managed efficiently in a real OS.

A user-friendly interface built using **PyQt5** and **matplotlib**, providing the following features:

- **File Access Simulation Tab**: Buttons to simulate file access requests.
- **Simulated RAM Tab**: Visualizes RAM log state based on the selected strategy (FIFO or LRU). Uses bar graphs to show the active files in memory and dynamically updates after every access.

- **Access Log Tab:** Displays real-time log entries from the access.log file, color-coded based on access type.

This module brings together all the components and provides an intuitive and visual way to understand the OS-level operations happening in the background.

CHAPTER 4 — CODE HIGHLIGHTS

SIMULATE_SYSCALL.C

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<time.h>
#define LOG_FILE "access.log"
#define ALLOWED_FILES_FILE "allowed_files.txt"
int is_allowed(const char *filename) {
    FILE *fp = fopen(ALLOWED_FILES_FILE, "r");
    if (!fp) {
        perror("Unable to open allowed_files.txt");
        return 0;
    }
    char allowed[100];
    while (fgets(allowed, sizeof(allowed), fp)) {
        allowed[strcspn(allowed, "\n")] = 0; // Remove newline if
        (strcmp(allowed, filename) == 0) {
            fclose(fp);
            return 1;
        }
    }
    fclose(fp);
    return 0;
}
```

```

        return 1;
    }

}

fclose(fp);

return 0;
}

void log_access(const char *filename, int allowed) {
    FILE *log = fopen(LOG_FILE, "a");
    if (!log) {
        perror("Unable to open access.log");
        return;
    }

    time_t now = time(NULL);
    char *timestamp = ctime(&now);
    timestamp[strcspn(timestamp, "\n")] = 0;
    pid_t pid = getpid();

    fprintf(log, "[%s] PID: %d - %s - %s\n", timestamp, pid, filename,
            allowed ? "ALLOWED" : "DENIED");

    fclose(log);
}

int main(int argc, char *argv[]) { if
    (argc != 2) {
        printf("Usage: ./simulate_syscall <filename>\n");

```

```

    return 1;

}

const char *filename = argv[1];

int allowed = is_allowed(filename);

log_access(filename, allowed);

if (allowed) {

    printf("Access granted to file: %s\n", filename);

} else {

    printf("Access denied to file: %s\n", filename);

}

return 0;
}

```

- Uses getpid(), ctime(), and file permission checking
- Logs access as ALLOWED or DENIED

GUI_MONITOR.PY

```

import sys import

os

import subprocess

from collections import deque

from datetime import datetime

```

```

from PyQt5.QtWidgets import (
    QApplication, QWidget, QVBoxLayout, QLabel, QPushButton, QTextEdit,
    QComboBox, QHBoxLayout, QTabWidget
)

from PyQt5.QtCore import Qt

from PyQt5.QtGui import QTextCharFormat, QTextCursor, QColor
import matplotlib.pyplot as plt

from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as
FigureCanvas

# Config
MEMORY_LOG_LIMIT = 5
ACCESS_LOG_FILE = "access.log"
FILES = [
    "record1.txt", "record2.txt", "secret_config.txt", "patient_report.txt",
    "unauthorized_notes.txt", "confidential.txt"
]

class MemoryLog:

    def __init__(self, strategy='FIFO'):
        self.log = deque()
        self.strategy = strategy

```

```

def access(self, filename): if
    filename in self.log:
        if self.strategy == 'LRU':
            self.log.remove(filename)
            self.log.append(filename)
    else:
        if len(self.log) >= MEMORY_LOG_LIMIT:
            self.log.popleft()
            self.log.append(filename)

def get_log(self):
    return list(self.log)

class GraphCanvas(FigureCanvas):
    def __init__(self, parent=None):
        self.fig, self.ax = plt.subplots(figsize=(4, 3))
        super().__init__(self.fig)

    def update_graph(self, memory_list):
        self.ax.clear()
        colors = ['green' if 'record' in f or 'secret' in f else 'red' for f in
                  memory_list]

```

```

        self.ax.bar(range(len(memory_list)), [1]*len(memory_list),
        tick_label=memory_list, color=colors)

        self.ax.set_title("RAM Access Simulation (LRU/FIFO)")

        self.draw()

class MainWindow(QWidget):
    def __init__(self):
        super().__init__()

        self.setWindowTitle("Secure File Access Monitor with RAM
Simulation")

        self.setGeometry(100, 100, 800, 600)

        self.memory_strategy = 'FIFO'

        self.memory_log = MemoryLog(self.memory_strategy)

        layout = QVBoxLayout()

        self.tabs = QTabWidget()

        self.log_tab = QWidget()

        self.memory_tab = QWidget()

        self.tabs.addTab(self.log_tab, "Access Logs")
        self.tabs.addTab(self.memory_tab, "Simulated RAM")

# Tab 1 – Log

log_layout = QVBoxLayout()

self.log_display = QTextEdit()

```

```
self.log_display.setReadOnly(True)

btn_layout = QHBoxLayout()

for f in FILES:

    btn = QPushButton(f'Access {f}')
    btn.clicked.connect(lambda checked, file=f: self.run_syscall(file))

    btn_layout.addWidget(btn)
```

```
self.refresh_btn = QPushButton("⟳ Refresh Logs")

self.refresh_btn.clicked.connect(self.load_logs)

btn_layout.addWidget(self.refresh_btn)

log_layout.addLayout(btn_layout)

log_layout.addWidget(self.log_display)

self.log_tab.setLayout(log_layout)
```

```
# Tab 2 – RAM

memory_layout = QVBoxLayout()

self.strategy_dropdown = QComboBox()

self.strategy_dropdown.addItem("FIFO")
self.strategy_dropdown.addItem("LRU")

self.strategy_dropdown.currentTextChanged.connect(self.change_strategy)

self.memory_display = QTextEdit()

self.memory_display.setReadOnly(True)
```

```

self.canvas = GraphCanvas(self)

memory_layout.addWidget(Qlabel("Choose Memory Strategy:"))

memory_layout.addWidget(self.strategy_dropdown)

memory_layout.addWidget(self.memory_display)

memory_layout.addWidget(self.canvas)

self.memory_tab.setLayout(memory_layout)

layout.addWidget(self.tabs)

self.setLayout(layout)

self.load_logs()

self.update_memory_display()

def change_strategy(self, strategy):

    self.memory_strategy = strategy
    self.memory_log
    = MemoryLog(strategy)
    self.load_logs()

def run_syscall(self, filename):

    if not os.path.exists("./simulate_syscall"):
        self.log_display.append("+")
        simulate_syscall binary not found."
    return

    subprocess.call(["./simulate_syscall", filename])

    self.memory_log.access(filename)

```

```
self.load_logs()  
self.update_memory_display()  
  
def load_logs(self):  
    self.log_display.clear()  
    if not  
        os.path.exists(ACCESS_LOG_FILE):  
            return  
    with open(ACCESS_LOG_FILE, "r") as f:  
        for line in f:  
            self.append_colored_line(line.strip())
```

```
def append_colored_line(self, line):  
    fmt = QTextCharFormat()  
    if "ALLOWED" in line:  
        fmt.setForeground(QColor("green"))  
    elif "DENIED" in line:  
        fmt.setForeground(QColor("red"))  
    else:  
        fmt.setForeground(QColor("black"))  
    cursor = self.log_display.textCursor()  
    cursor.movePosition(QTextCursor.End)  
    cursor.insertText(line + "\n", fmt)
```

```

    self.log_display.setTextCursor(cursor)
    self.log_display.ensureCursorVisible()

def update_memory_display(self):
    self.memory_display.clear() mem_log
    = self.memory_log.get_log()
    self.memory_display.append("Last 5 File Accesses
({}):\\n".format(self.memory_strategy))
    for I, f in enumerate(mem_log): self.memory_display.append(f'{i+1}.
{f}')
    self.canvas.update_graph(mem_log) if
name_____ == "_main_":
    app = QApplication(sys.argv)
    win = MainWindow()
    win.show() sys.exit(app.exec_())

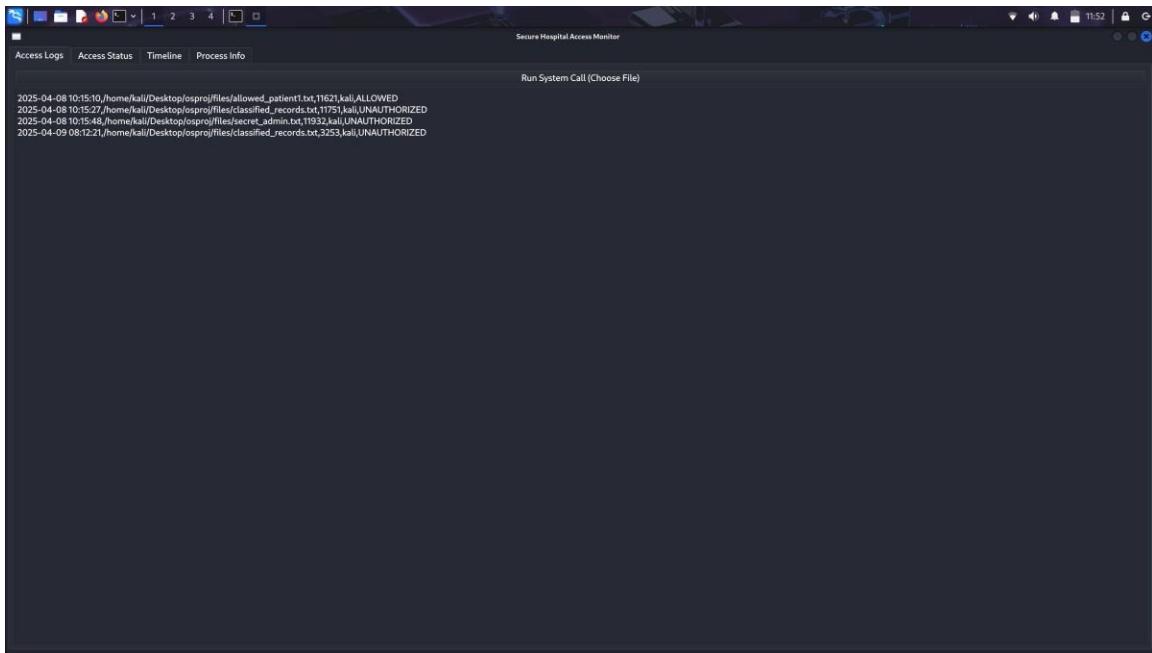
```

- Uses PyQt5 for GUI
- Shows logs, triggers system calls
- Displays simulated RAM based on LRU/FIFO
- Uses matplotlib for visual memory graph

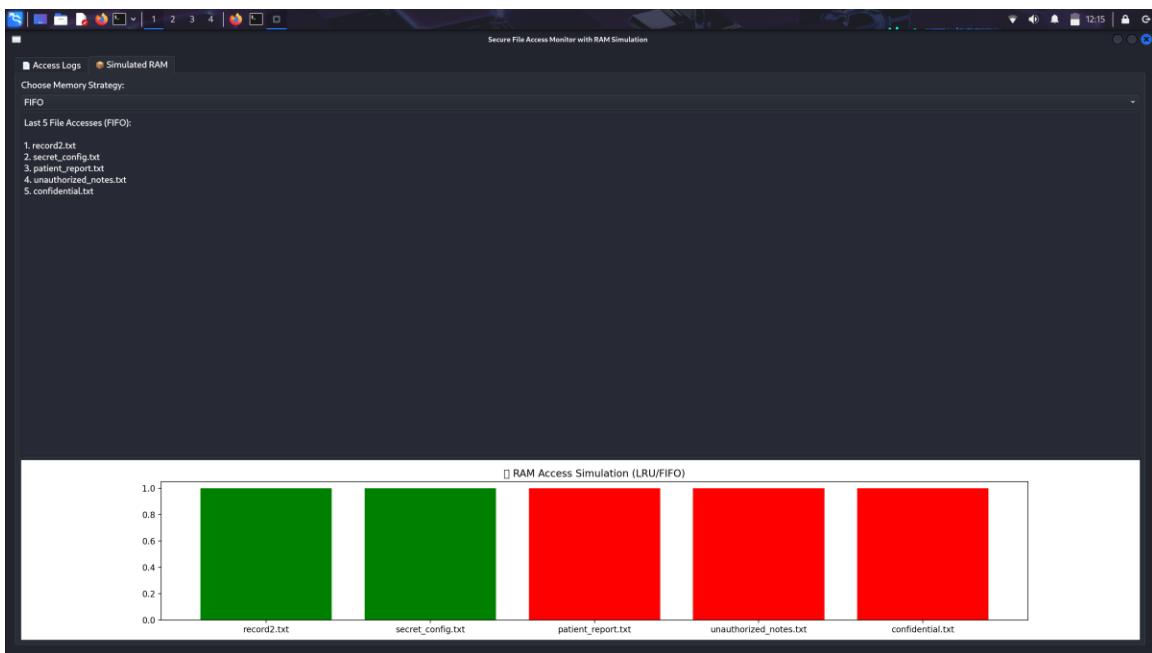
CHAPTER 5 — SCREENSHOTS AND OUTPUT

This section includes screenshots from the working GUI system, demonstrating access logging, memory visualization, and system interaction.

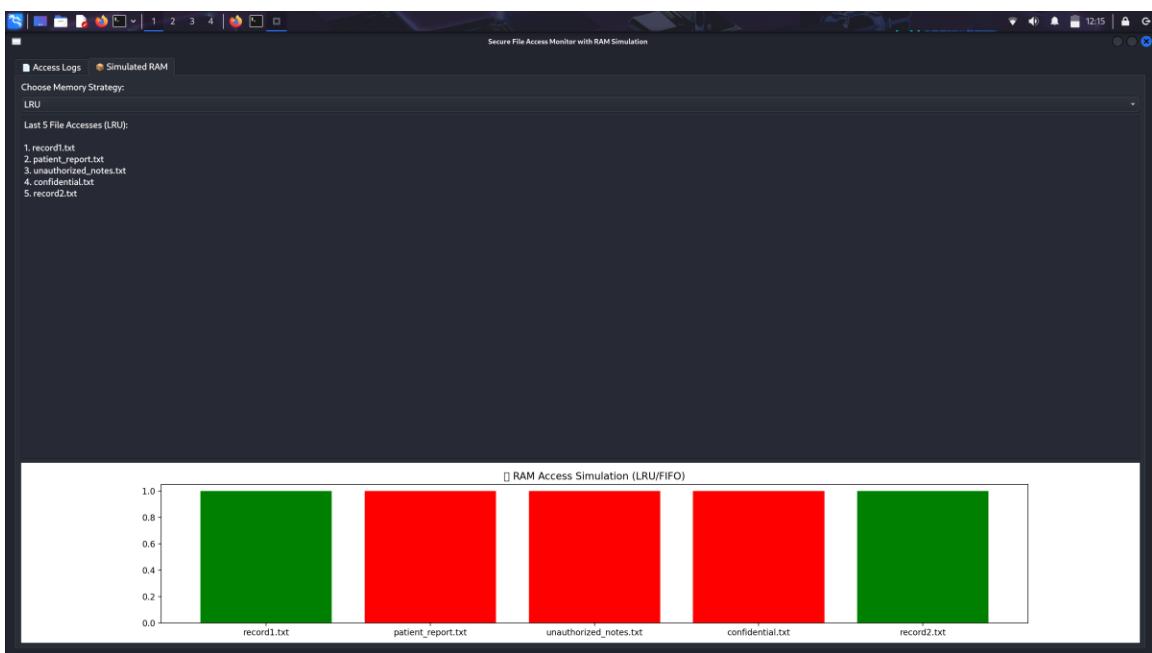
Screenshot placeholders:



- GUI File Access Page



- RAM Log Panel (FIFO)



- RAM Log Panel (LRU)

CHAPTER 6 — CONCLUSION AND FUTURE ENHANCEMENT

This system successfully integrates key OS concepts like system calls, memory paging (LRU/FIFO), and process management into a real-world use case: secure hospital file access. In future versions, we plan to:

- Use eBPF or ptrace for live system call hooking
- Integrate SQLite or Elasticsearch for deeper log storage
- Add role-based access control
- Add real-time notifications and multi-user login system

CHAPTER 7 – REFERENCES

- <https://man7.org/linux/man-pages/>
- https://en.wikipedia.org/wiki/Page_replacement_algorithm
- <https://www.geeksforgeeks.org/lru-cache-implementation/>
- <https://doc.qt.io/qtforpython/>
- <https://www.learn-c.org/>
- <https://matplotlib.org/stable/gallery/index.html>