A Project Report on

Automatic Sandboxing with Secure Process Isolation

Submitted in partial fulfillment for the award of the degree of

BACHELOR OF TECHNOLOGY

in

INFORMATION TECHNOLOGY

by

Shubhrodipto De	231IT070
Sankalp Saharia	231IT063
Revar Harsh	231IT055
Nikhil Agarwal	231IT044

IV Sem B.Tech (IT)

Department of Information Technology

National Institute of Technology Karnataka, Surathkal.

April 2025

CERTIFICATE

This is to certify that the project entitled "**Automatic Sandboxing with Secure Process Isolation**" has been presented by ***Shubhrodipto De (231IT070), Sankalp Saharia (231IT063), Revar Harsh (231IT055) and Nikhil Agarwal (231IT044)*** students of **IV semester B.Tech (IT)**, Department of Information Technology, National Institute of Technology Karnataka, Surathkal, during the even semester of the academic year **2024 – 2025.** It is submitted to the Department in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

Place: _____

Date: (Signature of the Examiner1)

Place: _____

Date: (Signature of the Coordinator)

DECLARATION BY THE STUDENT

I hereby declare that the Project on Operating Systems (IT253) entitled "**Automatic Sandboxing with Secure Process Isolation**" was carried out by us during the even semester of the academic year 2024 – 2025 and submitted to the department of IT, in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in the department of Information Technology, is a bonafide report of the work carried out by us. The material contained in this seminar report has not been submitted to any University or Institution for the award of any degree.

Place: _____

Date: _____

_____

(Name and Signature of the Student)

Place: _____

Date: _____

_____

(Name and Signature of the Student)

Place: _____

Date: _____

_____

(Name and Signature of the Student)

Place: _____

Date: _____

_____

(Name and Signature of the Student)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

In an era where system security is constantly being challenged by sophisticated malware and malicious code execution, the need for efficient, automated, and secure process isolation mechanisms has never been more pressing. Sandboxing, a security technique that isolates processes from the rest of the system, plays a crucial role in preventing potential damage caused by untrusted or dangerous software.

Our project, titled **"Automatic Sandboxing for Secure Process Isolation,"** focuses on automating the detection and containment of suspicious processes using two robust technologies: **Docker** for containerization and **Seccomp** (Secure Computing Mode) for system call filtering. This hybrid approach ensures that dangerous processes are not only identified but also confined to a secure and controlled execution environment.

Traditionally, sandboxing has required manual configuration and intervention, often making it inefficient and inconsistent across varying threat scenarios. To address this, we developed a script (sandbox.py) capable of automating the identification, termination, and containerization of flagged processes. Once detected, a harmful process is immediately terminated, its associated files are deleted, and it is re-launched inside a Docker container, where a predefined **Seccomp profile** strictly limits the system calls it can make.

The use of Docker ensures lightweight virtualization, allowing isolated execution without the overhead of full-fledged virtual machines. Seccomp complements this by allowing only a minimal set of safe system calls, thereby significantly reducing the attack surface. Together, these technologies help establish a robust line of defense against threats originating from insecure or vulnerable applications.

This solution is **scalable and modular**, supporting multiple types of source files—including C, C++, and Python—which makes it practical for diverse development environments. The current implementation provides an automated workflow and achieves strong isolation with minimal performance impact, making it suitable for integration into larger cybersecurity frameworks or operating system architectures.

As cyberattacks continue to evolve in sophistication, the need for intelligent and proactive containment strategies becomes critical. Automated sandboxing solutions like ours contribute toward this goal by eliminating human error, increasing speed of response, and reducing the risk posed by potentially harmful code.

CHAPTER 2: OBJECTIVE

The objective of this project is to develop an automated, secure, and efficient sandboxing system that isolates potentially harmful processes. The system should identify flagged processes, terminate them, remove their residual files, and re-execute them in a secure Docker container with restricted system calls using Seccomp.

The core aim is to:

- Automate the sandboxing workflow from detection to secure re-execution.
- Utilize Docker to provide lightweight containerization and isolation from the host environment.
- Enforce system-level restrictions through a custom JSON-based Seccomp profile.
- Maintain support for multiple programming languages such as C, C++, and Python.
- Achieve minimal performance overhead while ensuring system safety.
- Design the architecture to be modular, scalable, and extensible for future enhancements like machine learning-based detection.

By fulfilling these objectives, the project aims to contribute a proactive defense mechanism that can be integrated into broader system security frameworks for real-world deployment.

CHAPTER 3: METHODOLOGY

The methodology of this project revolves around designing and executing an automated, scalable framework for process isolation using Docker and Seccomp. The workflow involves several systematic phases from detection to secure execution in a sandboxed environment. The entire methodology is broken down as follows:

3.1 Flagged File Generation and Simulation

To test the sandboxing system, a flagged file simulating malicious behavior is created. This includes files in C, C++, or Python that mimic real-world harmful operations like file creation, unauthorized access, or continuous looping. These demo files serve as the foundation for simulating potential threats.

3.2 Detection via Monitoring Script (sandbox.py)

sandbox_launcher.py is responsible for running the flagged file, identifying its process ID (PID), and then invoking sandbox.py to sandbox the process accordingly. This setup separates the execution and monitoring phases, allowing sandbox.py to focus solely on analyzing and sandboxing the process once it is active.

3.3 Process Termination and Data Cleanup

Upon detection:

- The sandbox.py script terminates the suspicious process using system calls.
- The script deletes the flagged file and scans for any newly created files associated with the process and deletes them.
- This step ensures no harmful residual data remains on the host system.

3.4 Containerization Using Docker

After cleanup, the same flagged application is relaunched in a Docker container. This container is:

- Built dynamically using a Dockerfile.
- Configured to run only specific binaries or scripts with limited access to system resources.

Docker's native resource control via cgroups and namespaces is leveraged to further confine the flagged process.

3.5 Seccomp Profile Enforcement

A Seccomp (Secure Computing Mode) profile written in JSON format is loaded during Docker container initialization. This profile:

- Defines a whitelist of allowed Linux system calls.
- Blocks dangerous system calls like ptrace, clone, or direct I/O system calls.
- Triggers termination or alerts on disallowed system call attempts.

This adds an extra security layer to the container, making it suitable even for running semi-trusted code.

3.6 Multi-language Compatibility Testing

To ensure flexibility, the methodology includes testing with programs written in multiple languages:

- **C/C++:** Compiled binaries tested for file system and memory access operations.
- **Python:** Scripts that dynamically generate or access data tested under container and Seccomp restrictions.

Test results verify that the sandbox system supports varied execution models and runtime behaviors.

3.7 Logging and Auditing

Each stage of the methodology includes logging mechanisms:

- Logs of process detection and termination.
- Records of deleted files and cleanup actions.
- Docker container logs of executed commands.
- Seccomp violation logs for audit and debugging.

These logs are valuable for analyzing behavior trends and forensics.

3.8 Automation Pipeline

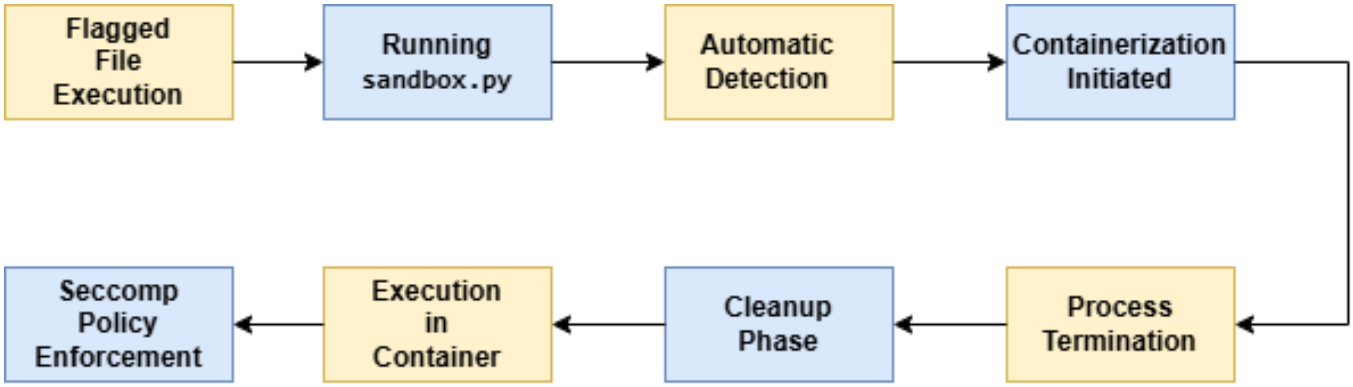
The entire methodology is automated. A single execution of sandbox.py triggers the complete lifecycle:

1. Detection
2. Termination
3. Cleanup
4. Container build
5. Execution in sandbox
6. Monitoring and logging

This automation reduces human error and response time in a real-world security setup.

This comprehensive methodology ensures that the sandboxing mechanism can reliably identify, neutralize, and securely run potentially harmful code with minimal system disruption. Future enhancements may include dynamic learning-based detection and real-time threat classification modules.

CHAPTER 4: SYSTEM DESIGN FLOW CHART



Fig[1] System Design Flow Chart

Component	Technology
Containerization	Docker
System Call Filtering	Seccomp (JSON)
Monitoring & Automation	Python Scripts
OS Interaction	Linux system calls
Logging	Python logging module, Docker logs
Compatibility	GCC for C/C++, Python 3.x

Table[1] Overview of Key Components and Technologies

CHAPTER 5: IMPLEMENTATION

In this chapter, we detail the methodology and design choices behind building our sandboxing framework. This section elaborates on the tools, components, and steps involved in setting up the system, as well as the execution flow, which makes it possible to isolate and analyze potentially malicious files in a controlled environment. The approach used for this implementation leverages automation, security practices, and containerization technologies to ensure that the system is both effective and scalable.

4.1 Tools & Environment

The sandboxing framework was constructed using a combination of reliable, well-supported tools and technologies. Each tool was selected to maximize the security, automation, and efficiency of the environment.

- **Operating System (OS): Ubuntu 22.04 LTS**
  Ubuntu 22.04 LTS was chosen as the operating system due to its stability and long-term support. It is a popular choice for server-side applications and provides an optimal environment for building secure, scalable systems. Furthermore, it is compatible with the majority of open-source tools required for containerization and system monitoring.

- **Programming Languages: Python 3, C/C++**

  - Python 3: Python was used for higher-level orchestration and automation tasks. Its simplicity and the availability of various libraries made it an ideal choice for writing scripts like sandbox_launcher.py and sandbox.py to automate execution, monitoring, and logging.

  - C/C++: C and C++ were utilized for performance-critical components that interact directly with the system, such as process management, system call handling, and container setup. These languages offer fine-grained control over system resources, essential for building a secure sandboxing environment.

- **Containerization: Docker**
  Docker was used to create isolated environments for executing test files. By leveraging Docker, we ensured that potentially harmful operations would be contained within a controlled environment, reducing the risk of malicious activity

affecting the host system. Docker containers are lightweight and efficient, providing a quick way to spin up isolated environments for each test.

- **Security: Seccomp (JSON profiles)**
  Seccomp (Secure Computing Mode) was applied at the container runtime to restrict the system calls available to the processes running within the container. This is crucial for reducing the attack surface and preventing any unauthorized or harmful actions that could compromise the host system, such as network access, file system manipulation, or privilege escalation.

- **Key Modules:** sandbox_launcher.py, sandbox.py

  - **sandbox_launcher.py**: This Python script acts as the orchestrator of the framework. It is responsible for executing the test file, initializing the sandbox environment, and triggering the monitoring process.

  - **sandbox.py**: This Python script serves as the core of the monitoring and containment process. It runs the test file inside a Docker container, watches for signs of malicious behavior, and terminates the process if it detects anything suspicious.

4.2 Core Components

The framework is composed of several key components that work together to provide the functionality of automated execution, monitoring, and isolation. These core components ensure that malicious activities can be detected, terminated, and analyzed in a safe environment.

- **sandbox_launcher.py:**
  This script acts as the initial point of entry for the framework. It takes a test file as input, sets up any necessary environment variables, and executes the file within the sandbox. This module also initializes logging mechanisms and provides outputs for review. It helps to automate the process and eliminates the need for manual intervention.

- **sandbox.py:**
  The core monitoring script, sandbox.py, is responsible for overseeing the execution of the file within the container. It checks for abnormal behaviors or system calls that

could signal malicious activity. If the script detects any suspicious activity (such as file modification, unauthorized access, or unusual system calls), it immediately terminates the process to prevent further potential damage.

- **Dockerfile Generator:**
  To ensure consistency and minimalism in container environments, a Dockerfile generator was developed. This tool dynamically generates a Dockerfile that creates a restricted environment specifically tailored to the test file being executed. The Dockerfile ensures that only the necessary dependencies are included, further minimizing the attack surface and keeping the container as lightweight as possible.

- **Seccomp Profile:**
  A Seccomp profile is applied to the Docker container during runtime. This profile restricts the system calls that the test file can make, ensuring that potentially harmful actions (such as accessing system files or initiating network connections) are blocked. This layer of security helps contain any malicious behavior within the Docker container, protecting the host system from exploitation.

4.3 Malicious File Simulation

In order to validate the effectiveness of the sandboxing framework, test files simulating malicious behavior were created. These files were written in a variety of languages, including C, C++, and Python. The files were designed to mimic different types of attacks, allowing us to evaluate the framework's ability to detect and prevent various forms of malicious behavior.

Examples of malicious behaviors simulated include:

- **File Creation and Deletion:**
  Test files that create or delete files on the system without appropriate authorization.

- **Infinite Loops:**
  Files that run in infinite loops, consuming excessive CPU and system resources, simulating a Denial-of-Service (DoS) attack.

- **Unauthorized Access:**
  Files attempting to access protected or sensitive data, such as system files, passwords, or environment variables.

These test files were critical for evaluating the sandbox's ability to detect and prevent different types of attacks, and helped ensure the system’s robustness in real-world scenarios.

4.4 Execution Flow

The following describes the execution flow of the sandboxing framework, from the moment a test file is executed until the results are logged and analyzed:

1. **File Execution by sandbox_launcher.py:**
   The sandboxing process begins when sandbox_launcher.py is executed. This script receives the test file and sets up the necessary environment for its execution. It then launches the file within the controlled sandbox environment.

2. **Monitoring by sandbox.py:**
   Once the file starts executing, the sandbox.py script begins monitoring the process. It checks for unusual or suspicious behaviors, such as attempts to create or delete files, excessive system calls, or memory consumption that exceeds predefined thresholds.

3. **Termination of Suspicious Behavior:**
   If any suspicious activity is detected during execution, the sandbox.py script will immediately terminate the process to prevent any potential harm to the host system. This is done to stop the file from causing further damage or from executing harmful operations.

4. **Docker Container Creation and Launch:**
   As part of the sandboxing process, a Docker container is dynamically created and launched for each test file. The container is built according to the restrictions specified in the generated Dockerfile, ensuring that the test environment is isolated and minimized to reduce the risk of system compromise.

5. **Seccomp Profile Enforcement:**
   The Seccomp profile is applied at runtime, ensuring that the test file is restricted in terms of which system calls it can make. This adds an additional layer of security, preventing the file from making dangerous system calls that could potentially compromise the host machine.

6. **Log Collection for Review:**
   Throughout the entire process, detailed logs are generated to capture information

about the execution and any detected behaviors. These logs are saved and made available for review and further analysis. This logging system allows for post-execution analysis of the test files to understand how the sandbox environment reacted and what behaviors were detected.

4.5 Outcomes

The sandboxing framework was successfully implemented with the following key outcomes:

- **Full Automation:**
  The process of executing, monitoring, and analyzing test files is fully automated. Once a file is submitted, the system takes care of the rest, eliminating the need for manual intervention and ensuring consistency in behavior analysis.

- **Multi-Language Support:**
  The framework supports test files written in multiple programming languages (Python, C, and C++). This provides flexibility and ensures that the sandbox can be used to analyze a wide range of potential threats.

- **Robust Isolation:**
  The use of Docker containers and Seccomp profiles ensures that the sandbox environment is secure and isolated. By restricting the system calls available to the test files, the system minimizes the risk of malicious behavior affecting the host system.

- **Scalability and Flexibility:**
  The framework can be easily scaled to analyze a variety of files with different characteristics. Additionally, the system is flexible enough to incorporate future advancements, such as machine learning-based detection systems or kernel-level monitoring for even more sophisticated threats.

CHAPTER 6: Results, Conclusion and Future Works

Results

The project achieved its goal of developing a fully functional and automated sandboxing system. The following results were observed:

- **Successful Detection and Containment:** The sandbox.py script was able to detect flagged processes based on predefined criteria and initiate sandboxing automatically.
- **Effective Isolation:** Flagged files were securely executed in Docker containers, eliminating interaction with the host system.
- **Restricted System Calls:** The Seccomp JSON policy effectively limited the system calls, preventing access to potentially dangerous operations.
- **Post-Detection Cleanup:** All generated files from flagged processes were successfully deleted to prevent lingering threats.
- **Multi-language Compatibility:** The system handled test files written in C, C++, and Python with equal efficiency.
- **Minimal Overhead:** The overall system exhibited minimal performance overhead, making it suitable for real-time or production environments.
- **Network Monitoring:** Expand the system to monitor and restrict suspicious outbound network activity from sandboxed containers.

This project lays the groundwork for an adaptive and scalable security architecture that can be extended for enterprise-level deployments or integrated into modern OS environments for better threat management.

Conclusion

This project successfully presents a robust, automated sandboxing framework capable of detecting, isolating, and securely executing potentially malicious code using Docker containers and Seccomp profiles. By simulating flagged files across multiple languages (C, C++, Python), the system demonstrates versatility and adaptability in real-world security scenarios.

Key accomplishments include:

- **Effective Process Isolation:** Leveraging Docker's namespace and cgroup features.
- **System Call Filtering:** Utilizing Seccomp profiles to enforce a whitelist-based security model.
- **Automated Workflow:** A single script (sandbox.py) orchestrates the entire lifecycle—detection, termination, cleanup, containerization, and monitoring.
- **Multi-language Support:** Ensures wide compatibility for diverse threat types.
- **Comprehensive Logging:** Enables detailed audit trails and forensic analysis.

The methodology ensures that suspicious programs are handled with minimal impact on the host environment, while maintaining transparency, modularity, and reproducibility.

Future Works

To further enhance the system's intelligence, performance, and security coverage, the following advancements are proposed:

1. Integration with eBPF (Extended Berkeley Packet Filter)

- **Purpose:** eBPF allows running sandboxed programs directly in the Linux kernel without changing kernel source code or loading kernel modules.
- **Benefits:**
    - Real-time, low-overhead monitoring of process behavior.
    - Fine-grained event filtering, including system calls, network activity, and file I/O.
    - Enhanced visibility into runtime operations for deeper anomaly detection.

2. Machine Learning-based Threat Detection

- **Objective:** Build and train an ML model that can classify flagged files based on static and dynamic features (e.g., code patterns, syscall traces).

- **Pipeline Includes:**
    - Feature extraction from flagged file behavior and metadata.
    - Training on labeled datasets (benign vs. malicious).
    - Integration with sandbox_launcher.py to predict threat likelihood before full execution.
- **Expected Outcome:** A proactive system capable of flagging potentially malicious files before execution, enhancing response time and reducing risk.

3. Web-Based Dashboard & Analytics

- Develop a frontend dashboard for:
    - Real-time log visualization.
    - Seccomp violations and system alerts.
    - Historical trend analysis and threat heatmaps.

These future developments will push the project beyond a static sandbox into a **smart, kernel-aware, and scalable threat containment platform** suited for modern cybersecurity challenges.