CSE 543: Computer Security

Fall 2024
Project 3: Implementing Integrity Access Control Monitor
Due: 11:59 pm (eastern time), November 10, 2024

October 28, 2024

1 Introduction

In this project, you will implement a reference monitor that enforces various methods for protecting integrity of data and process using Mandatory Integrity protection systems. More specifically your implementation should accommodate the **Biba Integrity policy**, **Windows Mandatory Integrity Control (MIC) policy** and the **Low Water-Mark Mandatory Access Control (LOMAC) policy**.

You are already given the project repository with few important missing pieces. First, you will understand the code structure and implementation of the reference monitor. Finally, with your understanding of the integrity access control policies, you will complete the missing pieces in the project with your own code. How to run and test your code will be explained in the later sections of this handout.

NOTE: Please read the entire report in its entirety before starting any part of the project!

2 Prerequisites

It is necessary to know the **three access control policies** in detail. Knowledge about how users, processes and files with different integrity levels interact with and affect each other is required to successfully complete this project. We have summarized it in the Background section, however, it is advisable to refer to course slides and other online resources to get a better understanding. If you have any doubts related to the rules in each policy, you may reach out to the course staff during office hours or through Canvas/Piazza.

It is advisable to brush up some concepts in C including `struct`, *pointers*, `malloc`/`free`, `typedef` and *type casting.* The code heavily uses these concepts to set up the platform to handle multiple different access control policies.

The code heavily uses *linked lists.* It is highly recommended (if you're not already comfortable with) to understand the data structure and its implementations in C before proceeding to understand the code.

3 Background

The goal of integrity protection is to prevent a system's processes from receiving untrusted inputs and files from being updated or read by untrusted processes. However, allowing no interactions between trusted and untrusted processes at all might prevent many useful interactions from occurring in a system. Therefore, many researchers have in the past come up with different approaches to this problem by proposing various mandatory access control policies. In this project, we will only focus on three of them:

Biba Integrity Policy:

- Every Process is assigned an integrity level.
- Every File is assigned an integrity level.
- All the levels in a system form a Lattice structure. In this project, for Biba policy we have a 2-level lattice – Low and High.
- Read Policy: A process/user is allowed to read from a file of the same integrity level or above. **“Read Up”**
- Write Policy: A process/user is allowed to write to a file of the same integrity level or below. **“Write Down”**
- Exec Policy: Same as Read Policy.

Windows Mandatory Integrity Control (MIC) Policy:

- Every Process is assigned an integrity level.
- Every File is assigned an integrity level.
- All the levels in a system form a Lattice structure. In this project, for MIC policy we have a 3-level lattice - Internet(low), User(high) and System(highest).
- Read Policy: All reads are allowed.
- Write Policy: A process/user is allowed to write to a file of the same integrity level or below. **“Write Down”**
- Exec Policy: A process/user is allowed to execute a file of the same integrity level or above. **“Exec Up”**

Low Water-Mark Mandatory Access Control (LOMAC) Policy

- Every Process is assigned an integrity Level.
- Every File is assigned an integrity level.
- All the levels in a system form a Lattice structure. In this project, for LOMAC policy we have a 2-level lattice - Low and High.
- Read Policy: A process's integrity level is changed to the read file's level that has the lowest integrity level.
- Exec Policy: A process's integrity level is changed to the executed file's level that has the lowest integrity level.
- Write Policy: A file's integrity level is changed to the writing process's level that has the lowest integrity level.

4 Code and Compiling

The initial code for the project is provided here: `https://classroom.github.com/a/DrCHL8S4`. You are only allowed to edit `monitor.c` and `lattice.c`. Editing any other file for the sake of finishing the project might work locally, but for grading your code files will be run in a different environment. Please refrain from changing any other file apart from these two.

Once you have some changes done in your files, to compile the code run this command:

```
make
```

The Makefile provided in the project directory will handle all the necessary compiling for your code. Once compiled, an updated binary is created with the name cse543-p3.

The reference monitor takes in Policy files and Test instruction files as input. As an output, the reference monitor creates a results file. For example:

```
Input Policy file - ./policies/biba.policy
Input Test file - ./test_cases/test
Output file path - ./results/biba.results

Commands to execute the test:
./cse543-p3 ./policies/biba.policy ./test_cases/test ./results/biba.results
./cse543-p3 ./policies/mic.policy ./test_cases/test ./results/mic.results
./cse543-p3 ./policies/lomac.policy ./test_cases/test ./results/lomac.results
```

We have provided results in the sample results directory. The files provided are:

```
sample_results/biba.sample.results
sample_results/mic.sample.results
sample_results/lomac.sample.results
```

Your code works exactly as expected if your output files **EXACTLY** match the results we have provided. E.g., your biba.results should match our biba.sample.results exactly. You can use diff or any other tool to compare your results with the provided sample results.

NOTE: Do not edit any fprintf statements provided in multiple files within the code repository as it may affect your grade. The test results are directly related to these fprintf statements.

5 Tasks

Most of your efforts would be spent in understanding the way the reference monitor is already implemented in the repository. Then, you need to finish five functions that have missing pieces of code to complete this assignment successfully. Particularly, the missing pieces of code are marked by:

```
/* YOUR CODE GOES HERE */
```

There are code comments in most places to help you understand the flow of the program. However, these are some points to remember:

1. The file lattice.c defines three global lists (doubly linked lists) namely lattice, label_mapping and trans_mapping.
2. The file monitor.c defines one global list (doubly linked list) namely system_mapping.
3. Every list is made up of elements and each element is of the type element. Definitions and other utility functions for lists are provided in linked_list.c and linked_list.h files.
4. The reference monitor accepts policy files that express the protection state (pstates), labeling state (lstates), and transition state (tstates).

5. Adding protection states into the system is handled by the function `addLattice` and this has already been implemented for you.
6. Adding labeling and transition states into the system are handled by the functions `addLabelPolicy` and `addTransPolicy` respectively, and **you have to implement this function.**

You are asked to implement the following functions:

1. `addLabelPolicy` function in `lattice.c`
   - Assigns a mapping from `name` to `level_char` based on the policy file's `lstate` statements. This is the labeling state of the reference monitor's mandatory protection system.
   - It is assigned to the global variable `label_mapping`, which is a list of name-level mappings.
   - In this function, you just need to find the lattice `level` object for the level provided from lattice, create a `map` object (defined in `lattice.h`) that assigns the level to the name, and add a new element (element defined in `linked_list.h`).
2. `addTransPolicy` function in `lattice.c`
   - Assigns a mapping in a global variable `trans_mapping`, which defines a transform rule for the reference monitor's mandatory protection system. It is based on the policy file's `tstate` statements.
   - In this function, you need to create a trans object that describes the rule – if a *subject level* performs operation `op` on an *object level*, then what is the ***new level*** of the subject/object.
3. `labelProcess` function in `monitor.c`
   - Assigns a level to a process named `proc` using `name` from `mapping`.
   - At runtime, every new process must be labeled when it starts. This function applies the labeling state to produce such a label.
   - Processes are created in two ways: During login based on authenticated user name and during process create (fork) based on the parent process name.
   - The difference is determined by the mapping passed into this function. You simply have to retrieve the `level` associated with that mapping name and create a new level assignment for this process name `proc`.
   - Runtime level assignments are stored in the global variable `system_mapping`.
4. `checkAccess` function in `monitor.c`
   - Authorizes operations requested by processes.
   - Operations include open, exec, read, and write. All operations are on file names, including read and write.
   - The permissions required for the various operations are specified in `execCmds` in `monitor.c`
   - From the `system_mapping` list, retrieve the element corresponding to the given process name.
   - From the `system_mapping` list, retrieve the element corresponding to the given file name.
   - Extract the level elements from the process element and the file element.
   - Write code to check if the given operation on the file by the process is allowed or not based on the retrieved levels information.

- Note that all reads are allowed in MIC policy

5. `checkTrans` function in `monitor.c`
    - Determines if the level of a particular process or file should be changed based on the transition state (`tstate` specifications in the policy).
    - From the `system_mapping` list, retrieve the element corresponding to the given process name.
    - From the `system_mapping` list, retrieve the element corresponding to the given file name.
    - Retrieve a transition state entry for the combination of subject level, object level, and operation.
    - If found, apply the transition.

6 Questions

1. In Biba policy testing, why do not we let process `p2` read the file `f1`?
2. In MIC policy testing, why do we let process `p2` read the file `f1`?
3. In MIC policy testing, why do not we let process `p11` execute the file `f1`?
4. In LOMAC policy testing, why do we let process `p11` execute the file `f1`?
5. In LOMAC policy, process `p2` is able to write to file `f3` sometimes. But other times it is denied to write. Explain all such occurrences and reason out.

7 Deliverables

All tasks in this project need to be completed individually without any help. Please commit your changes relatively frequently so that we can see your work clearly. Do not forget to push changes to the remote repository in Github! You can get as low as zero points if we do not see any confirmation of your work. Do not attempt to falsify time stamps, this may be viewed as an academic integrity violation! Finally, include "report.pdf" in your Github repository containing answers to project questions in Section 6.

In your Canvas submission, please indicate the GIT commit number corresponding to the final submission. You should also specify your github username. The submission format in plain text

`<your_github_username>:GIT commit`

For example, mustbastani:936c332e7eb7feb5cc751d5966a1f67e8089d331

8 Grading

The assignment is worth 100 points total broken down as follows.

1. **Biba Policy:** 10 points for Policy setup. 15 points for Test case traces.
2. **MIC Policy:** 10 points for Policy setup. 15 points for Test case traces.
3. **LOMAC Policy:** 10 points for Policy setup. 15 points for Test case traces.
4. Questions 1–5: 5 points each.