PES UIVERSITY EC Campus
1 KM before Electronic City, Hosur Road,
Bangalore-100

A Project Report on

An Operating System Safeguarded by CERT Recommendations – Security Solutions th(at) work

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

For the Academic year 2020

by

Bhavin G Chennur

PES2201800682

Anirudh R

PES2201800068

PES UNIVERSITY EC Campus
1 KM before Electronic City, Hosur Road,
Bangalore-100

Department of Computer Science Engineering

CERTIFICATE

Certified that the project work entitled **An Operating System Safeguarded by CERT Recommendations – Security Solutions th(at) work** carried out by **Bhavin G Chennur** bearingSRN **PES2201800682** and **Anirudh R** bearing SRN **PES2201800068**,bonafide students of **IV semester** in partial fulfillment for the award of **Bachelor of Engineering** in PES University during the year **2020** .

The project report has been approved as it satisfies the academic requirements in respect of Project work prescribed for the course **Secured Programming with C**.

……..	……..
GUIDE	**Dr. Sandesh**
Dr. Kiran D C	**HOD, CSE**

Declaration

I hereby declare that the project entitled **"An Operating System Safeguarded by CERT Recommendations – Security Solutions th(at) work"** submitted for Bachelor of Computer Science Engineering degree is my original work and the project has not formed the basis of the awards of any degree, associate ship, fellowship or any other similar titles.

Signature of the Student: Bhavin G Chennur

Place: Bengaluru

Date: 13/05/2020

Signature of the Student: Anirudh R

Place: Bengaluru

Date: 13/05/2020

An Operating System Safeguarded by CERT recommendations

- Security Solutions th(at) work

Authors

Bhavin G Chennur bearing SRN PES2201800682 in PES University

Anirudh R bearing SRN PES2201800068 in PES University

Abstract

This project demonstrates four fundamental features of any Operating System, CPU Scheduling, Security, Process Management and File System. Highest Response Ratio Next (HRRN) algorithm is used to demonstrate CPU Scheduling, given ‘n’ processes this algorithm schedules them to use the CPU time. Data Sanitization, uniformity in password size, encryption for numeric passwords are a few security features with a blend of various recommendations that have been demonstrated in this project. The use of a Carry Lookahead adder (CLA) to execute processes faster has been demonstrated for Process Management. The maintenance of an Open_File_Table to keep track of the files and processes under use is demonstrated along with the use of suitable safety recommendations. An artificial Error detection system. A simulation of a process control block. This way this projects demonstrates a way to increase the protection of an Operating System with the use of safety recommendations.

Table of Content

1.0 INTRODUCTION

1.1 Problem definition

1.2 Need of safe programming

2.0 Requirement Specification

2.1 Establishment

2.2 Risk Assessment

2.3 Software Requirement Specification

2.3.1 Modules

2.3.2 Users

2.3.3 Time

2.4 Safety Requirement Specification

2.4.1 Influence of the following

2.4.1.1 Programming Language

2.4.1.2 Compiler

2.4.1.3 Memory

2.4.2 Software features

2.4.2.1 Data Types used

2.4.2.2 Functions used and its descriptions

2.5 List of recommendations used in the project

2.6 Justification for the recommendations used in the software

3.0 Design
- **3.1 Modularity**
- **3.2 Readability**
- **3.3 Abstraction**

4.0 Implementation (Entire Code)

5.0 Testing and Verification

6.0 Maintenance
- **6.1 Modification**
- **6.2 Creating Multiple Versions**
- **6.3 Scalability**

7.0 Conclusion

8.0 Blog

9.0 References

Introduction

Problem Definition and Need :

Problem Definition:

To increase the protection of an Operating System with the use of safety recommendations provided by CERT C.

Need of safe programming:

With the good the technology has done to our world it has also brought in the bad, hackers can easily plant a bug in your software and your software will be in control of bad hands. For a simple example, when your software ask a new user to create an account hackers can give command prompt instructions instead of the details required to make an account and when your software tries to process the input it might end up executing vulnerable instructions. It is that simple to destroy your entire software. Secure programming makes a noticeable attempt in protecting your software, the risk of being hacked reduces by a huge amount but not completely.
This opens up the scope for two things, one to use secure programming to your software to protect it and two to develop higher security standards. This project can do both,

we could use secure programming to develop a secure software and we also had the scope to wisely use the recommendations such that we could rise the security standards implemented in our project.

We needed safe programming for consistent results for mathematical calculations, encryption systems, safe string processing and many other core applications.

The preferences and recommendations prescribed by the CERT has helped us in writing a safe and secure code. Starting from variable description to the complicated memory allocation,the CERT rules have played a major role in minimizing the syntax and run-time errors and also provided a solution to encounter them. Our application needs a safe program and CERT recommendations is of high technical importance to us.

Requirement Specification

Establishment :

We’ll establish the security requirements of the project here, this project aims at simulating the basic features of any OS, during the development of the project the first thing we did was to accumulate all the recommendations we wanted to use, then we went on to blend it with our concept. Because requirements were infused in the first layer itself, the project is consistent with security.

We needed security for consistent results for mathematical calculations, encryption systems, safe string processing and many other core applications. Hence we avoided the software doing what it should not have done.

Data sanitization for any user input was focused thoughout the project to make sure that no bug could enter the software, this process is widely known as the security bug tracking process.

When it comes to architectural planning, we wanted every case to be independent from any other case hence we accumulated a different set of recommendations each time we took upon a new case for the software.

User-made and inbuilt functions like clearenv(), data sanitisation, password verification system were reinforced to build quality gates or bug bars around the project. Minimum standard set for the performance is to make sure that the software never crashes, for any reason whatsoever, from stack overflow to edge case input everything is carefully taken care of, this also makes the quality criteria for our project.

In case of a security breach the software blocks it and put out on an appropriate message to the user, this can be seen extensively in the error detection systems we've developed in our software. Privacy risk shouldn't ever be a concern, any user is asked for a unique password as soon as the software is launched giving no room for privacy breach.

Risk Assessment

Any recommendation of high priority risk is developed with an extra care. Below is a risk assesment analysis for some of our core functions which relate to deep security and privacy systems in our software:

Function name: Data Sanitization

Recommendation: STR02-C. Sanitize data passed to complex subsystems

Severity	High
Likelihood	Likely
Remediation Cost	Medium
Priority	P18
Level	L1

Function name: Password_verification_system

Recommendation: STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator.

Severity	High
Likelihood	Likely
Remediation Cost	Medium
Priority	P18
Level	L1

These were a few important risk assesment analysis we had to make as they form a core part of our deep security analysis.

Software Requirement Specification:

Modules:

At the outset of the project we started drafting out the kind of modules we wanted and we developed the following concepts:

An artificial error detection system which could predict errors beforehand and solve them, we further split this into two parts so that a procedure could only concentrate on strings.

A secure password system, which focused on data sanitisation, null and vulnerable inputs. Many procedures blended with tens of recommendations were built around this concept. We also went a step ahead to build an encryption system for numeric passwords.

A real CPU scheduling system which could optimise process handling with CPU, this was later integrated with CLAs and a few memory fixes.

Users:

Who are the users for our application? That's obvious, who wouldn't want an Operating System! Jokes apart, any computing system in this world needs an OS, from a basic program to add 2 numbers to a complex rocket launch system software you need an Operating System.

Time:

The time complexity for most of the cases our application deals is O(n), n is the user input in some cases and hard coded in some cases but in either possibilities time complexity doesn't exceed O(n).
A few functions doesn't even have to process anything, for instance error detection system procedures only check with the recommendations for the tasks specified and don't need any processing.

Safety Requirement Specification:

Influence of the following:

Programming Language

C is a programming language that is very close to the memory (hardware) and does not have a heavy software built around it. C language assumes that the user has taken care of his program and doesn't check for anything, as a developer it is so much more challenging to develop safe programs. For instance if the input exceeds the memory size allocated C language does not throw an error and there is high risk of the data being lost, therefore we have had to take care of minute details of all possible edge cases in developing our application.

Compiler

We use cc compiler which stands for C Compiler on the terminal provided by Microsoft Visual Studio Code, it is very similar to a gcc compiler just that cc compiler is

unique to the C language. This is possibly the best compiler that we found that would suit our application, it gives the maximum number of warnings when compared to its counterparts, for example, during the testing and verification phase our application when executed on gcc compiler gave no warnings or errors and when executed on cc compiler gave two warnings. This way we've found an optimal way to make the safest possible code powered by CERT recommendations and rules.

Memory

We use a queue ADT for storing passwords, string arrays for data processing and error detection system. We use pointers to const when referring to string literals(STR05-C), whenever we don't want to manipulate important strings even by an accident.
We've taken care of overflow conditions during dynamic memory allocations with the help of the recommendations, INT30-C. Ensure that unsigned integer operations do not wrap, MEM36-C. Do not modify the alignment of objects by calling realloc() and MEM32-C Detect and handle critical memory allocation errors.

Software features

Data Types used:

Various abstract data types have been used in the project , like queues, enums, etc. Every ADT was chosen because the application demanded it.

Here is the list of ADTs used:

typedef int size_of_cla;

typedef int no_of_blocks_in_cla;

typedef int time_variable;

typedef int password_verifier;

typedef char dangerous_characters;

typedef int integer_type_variable;

```
typedef int counter;

typedef int size_variable;

typedef char* string_pointer;

typedef char string_array;

typedef int integer_array;

typedef long long_type_variable;

typedef long* long_type_pointer;

typedef int numeric_encryption_procedure_result;

typedef int numeric_encryption_variable;

typedef int result_variable;

typedef int error_number_variable;

typedef int scheduler;

typedef int scheduler_variable;
```

```
typedef int number_of_observations;

typedef int process_identification_number;

typedef int* process_identification_number_pointer;

typedef int time_elapsed;

typedef float executioncontext;

typedef float* executioncontext_pointer;

typedef int random_a;

typedef int validation;

typedef float* processingtime_pointer;

typedef float* waitingtime_pointer;

typedef float* processtime_pointer;

typedef float* waittime_pointer;

typedef float* float_pointer;
```

```
typedef char** process_state_pointer;

typedef int* parent_pointer;

typedef int** parent_pointer_pointer;

typedef int integer_variable;

typedef int size_of_n;

typedef char char_variable;

typedef int* integer_pointer;

typedef const struct buffer* detection_pointer;

typedef struct buffer* buffer_structure_pointer;

typedef struct buffer buffer_structure_variable;

typedef const struct buffer
constant_buffer_structure_variable;

typedef struct flex_array_struct*
flex_array_struct_pointer;
```

```
typedef struct flex_array_struct
flex_array_struct_variable;

typedef unsigned int non_negative_integer;

enum processes{ process1, process2, process3,
process4, process5};

// QUEUE ADT

//This compliant solution makes use of type definitions but does not declare a pointer type and so cannot be used in a const-incorrect manner

struct obj {
  int front, rear;
  char queue[1000][10];

};
typedef struct obj PASSWORD;


typedef const struct buffer {

integer_type_variable x;
```

```
integer_type_variable y;

} BUFFER;



struct flex_array_struct {
  size_variable num;
  int data[];
};
```

Functions used and its descriptions:

Here is the list of functions used in our application:

1) extern password_verifier password_check_using_sanitization(string_pointer str, size_variable size);

--- for sanitizing the password input by the user.

2) extern void time_taken_carry_lookahead_adder();

--- a function for demonstrating the use of CLAs

3) extern password_verifier string_validity(string_pointer destination_string, size_variable destination_size, string_pointer source_string, size_variable source_size);

--- To make sure the password does not exceed the specified size or is not null.

4) extern numeric_encryption_procedure_result multiple_of_sum(numeric_encryption_variable no1, numeric_encryption_variable no2, numeric_encryption_variable multiple);

--- To encrypt a numeric password.

5) extern void NUMERIC_PASSWORD_ENCRYPTION();

--- To encrypt a numeric password.

6) extern scheduler Response_Ratio(scheduler_variable waiting_time, scheduler_variable burst_time);

--- To find the response ratio

7) extern void open_file_table();

--- To demonstrate a recommendation for error handling.

8) extern void CPU_scheduling();

--- To simulate CPU Scheduling.

9) extern void
Password_verification_system(PASSWORD *ptr);

--- A wrapper function to accept and check string passwords.

10) extern void string_processing_error_detection();

--- To simulate error detection in strings.

11) extern void
Artificial_error_detection_system(detection_pointer buf, flex_array_struct_pointer p1,
flex_array_struct_pointer p2);

--- To simulate error detection system

12) static validation valid(float_pointer pntr);

--- to validate the pointer

13)extern processtime_pointer
pcb1(number_of_observations n);

--- To calculate processing time.

12) extern waittime_pointer pcb2(number_of_observations n);

--- To process waiting time

13) extern void pcb(process_identification_number_pointer processID,executioncontext_pointer exc,time_elapsed time,processingtime_pointer ptime,waitingtime_pointer stime,process_state_pointer state,parent_pointer ptr,number_of_observations n);

--- To take all the inputs and make a process table.

List of recommendations used in the project:

We have used each and every recommendation taught in the course. We have 55 recommendations used in total.

Below is the list of recommendations used in the application:

RECOMMENDATIONS USED:

1) RECOMMENDATION: PRE02-A. Macro expansion should always be parenthesized for function-like macros.

2) RECOMMENDATION: PRE00-A. Prefer inline functions to macros.

3) RECOMMENDATION: STR02-C. Sanitize data passed to complex subsystems.

4) RECOMMENDATION: STR03-A. Do not inadvertently truncate a null terminated byte string.

5) RECOMMENDATION: Pass the size of the array whenever the array is passed to a function.

6) RECOMMENDATION: PRE01-A. Use parentheses within macros around variable names.

7) RECOMMENDATION: PRE04-A. Do not reuse a standard header file name.

8) (BEYOND_SYLLABUS) RECCOMENDATION: PRE08-C. Guarantee that header file names are unique.

9) RECOMMENDATION: PRE30-C. Do not create a universal character name through concatenation.

10) RECOMMENDATION: PRE03-A. Avoid invoking a macro when trying to invoke a function.

11) RECOMMENDATION: PRE05-A. Avoid using repeated question marks.

12) RECOMMENDATION: DCL02-A. Use visually distinct identifiers.

13) RECOMMENDATION: DCL23-C. Guarantee that mutually visible identifiers are unique.

14) RECOMMENDATION: DCL03-A. Place const as the rightmost declaration specifier.

15) RECOMMENDATION: DCL04-C. Do not declare more than one variable per declaration.

16) RECOMMENDATION: DCL07-C. Include the appropriate type information in function declarators.

17) RECOMMENDATION: DCL00-C. Const-qualify immutable objects.

18) RECOMMENDATION: DCL01-C. Do not reuse variable names in subscopes.

19) RECOMMENDATION: DCL05-C. Use typedefs of non-pointer types only.

20) RECOMMENDATION: DCL06-C. Use meaningful symbolic constants to represent literal values.

21) RECOMMENDATION: DCL08-A. Declare function pointers using compatible types.

22) RECOMMENDATION: DCL09-A. Declare functions that return an errno with a return type of errno_t.

23) RECOMMENDATION: DCL10-C. Maintain the contract between the writer and caller of variadic functions.

24) RECOMMENDATION: DCL30-C. Declare objects with appropriate storage durations.

25) RECOMMENDATION: EXP03-C. Do not assume the size of a structure is the sum of the sizes of its members.

26) RECOMMENDATION: EXP43-C. Avoid undefined behavior when using restrict-qualified pointers.

27) RECOMMENDATION: STR05-C. Use pointers to const when referring to string literals.

28) RECOMMENDATION: STR06-C. Do not assume that strtok() leaves the parse string unchanged.

29) RULE: STR30-C. Do not attempt to modify string literals.

30) (BEYOND_SYLLABUS) RECOMMENDATION: STR11-C. Do not specify the bound of a character array initialized with a string literal.

31) RECOMMENDATION: STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator.

32) RECOMMENDATION: STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string.

33) RECOMMENDATION: STR10-C. Do not concatenate different type of string literals.

34) RECOMMENDATION: MEM03-C. Clear sensitive information stored in reusable resources.

35) RECOMMENDATION: MEM06-A. Do not use user-defined functions as parameters to allocation routines.

37) (BEYOND_SYLLABUS) RECOMMENDATION: MEM36-C. Do not modify the alignment of objects by calling realloc().

38) RECOMMENDATION: MEM32-C Detect and handle critical memory allocation errors.

39) RECOMMENDATION: MEM33-C. Allocate and copy structures containing a flexible array member dynamically.

40) RECOMMENDATION: MEM35-C. Allocate sufficient memory for an object.

41) (BEYOND_SYLLABUS) RECOMMENDATION: INT30-C. Ensure that unsigned integer operations do not wrap.

42) RECOMMENDATION: ENV03-C. Sanitize the environment when invoking external programs.

43) RECOMMENDATION: STR00-A. Use TR 24731 for remediation of existing string manipulation code.

44) RECOMMENDATION: STR01-A. Use managed strings for development of new string manipulation code.

45) RECOMMENDATION: STR07-A. Take care when calling realloc() on a null terminated byte string.

46) MEM10-C. Define and use a pointer validation function

47) MEM02-A. Do not cast the return value from malloc()

48) MEM05-C. Avoid large stack allocations

49) MEM04-C. Beware of zero-length allocations

50) MEM00-C. Allocate and free memory in the same module, at the same level of abstraction.

51) MEM30-C. Do not access freed memory

52) MEM01-C. Store a new value in pointers immediately after free()

53) MEM31-C. Free dynamically allocated memory when no longer needed

54) MEM34-C. Only free memory allocated dynamically

55) MEM07-C. Ensure that the arguments to calloc(), when multiplied, do not wrap

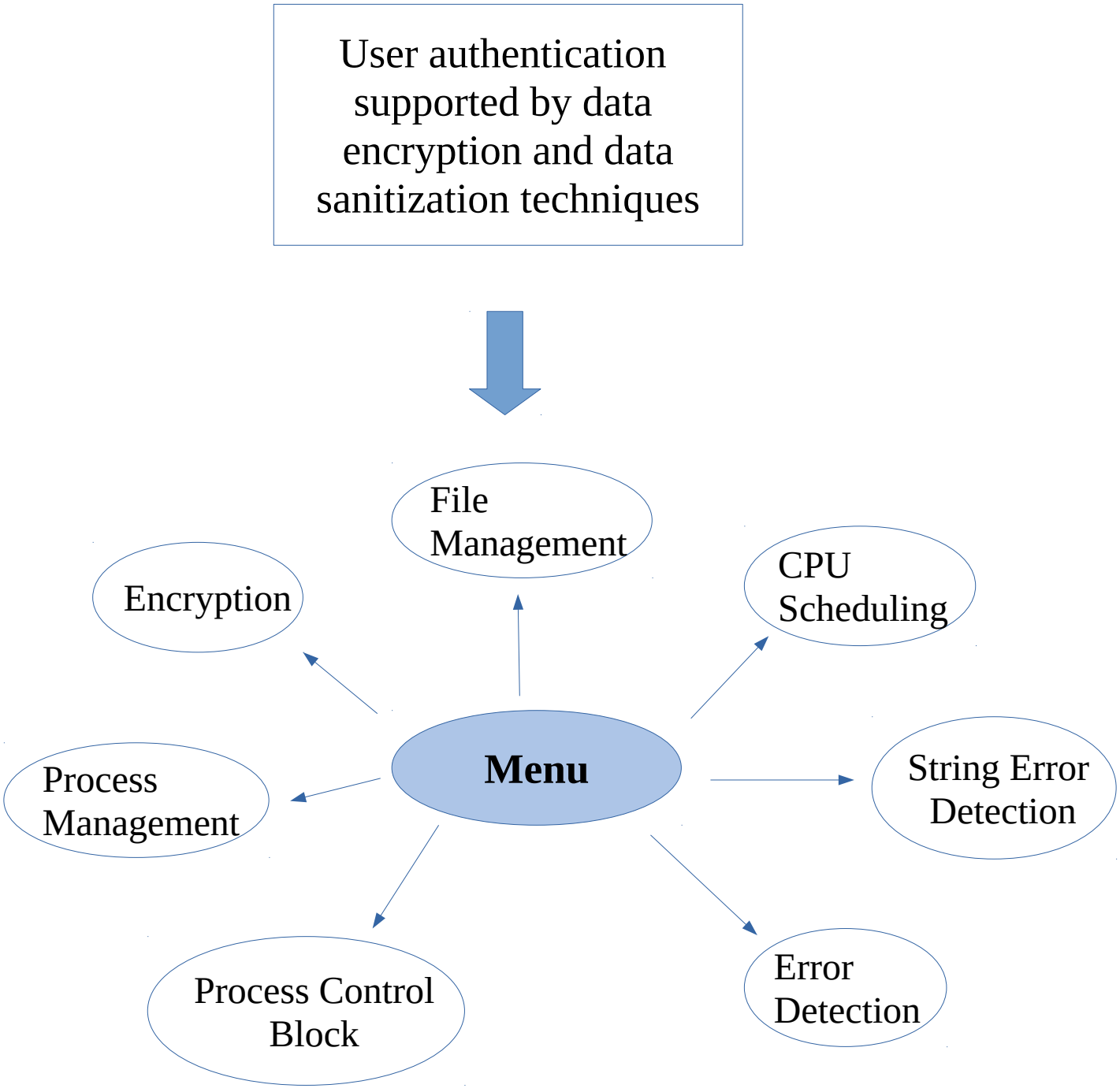
Justification for the recommendations used in the software:

We listed down the programming scenarios for every recommendation first and we then developed functions that suited both the recommendation and our project concept.

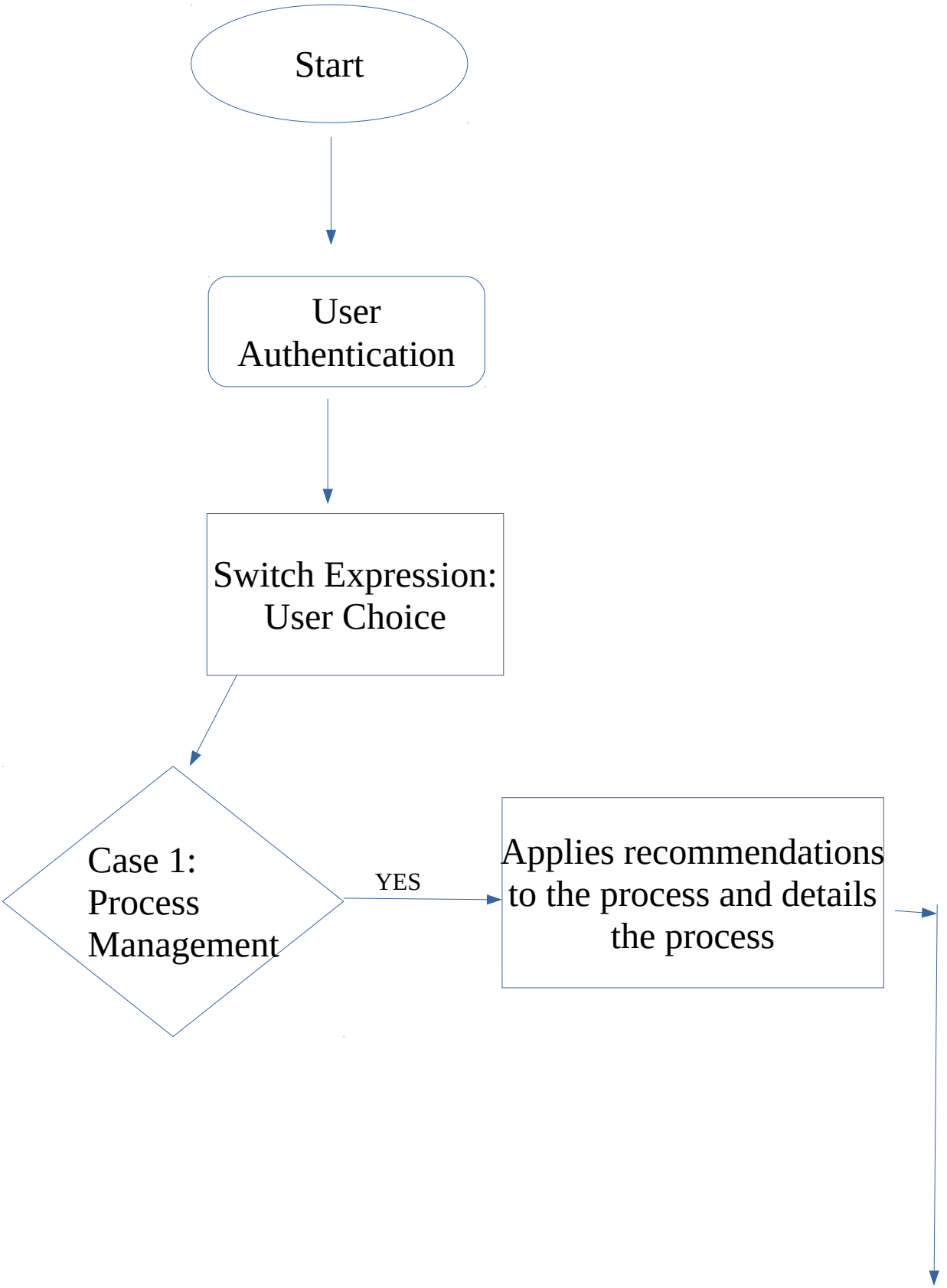
We have commented out the use of the recommedation wherever needed and the recommendation is specified at the point of usage and anyone can easily make out how the recommendation is used by looking at the scene it is used at.

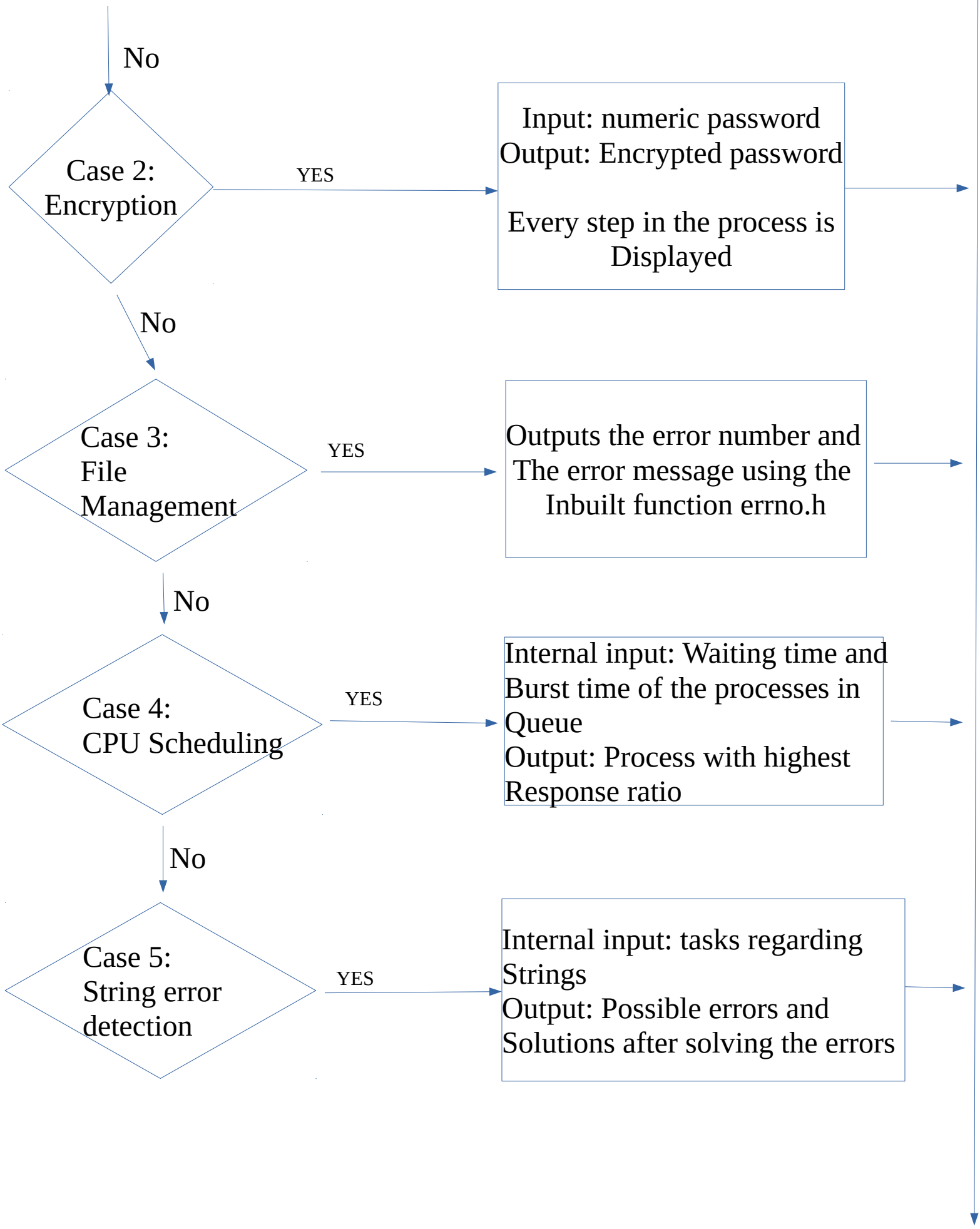
Design

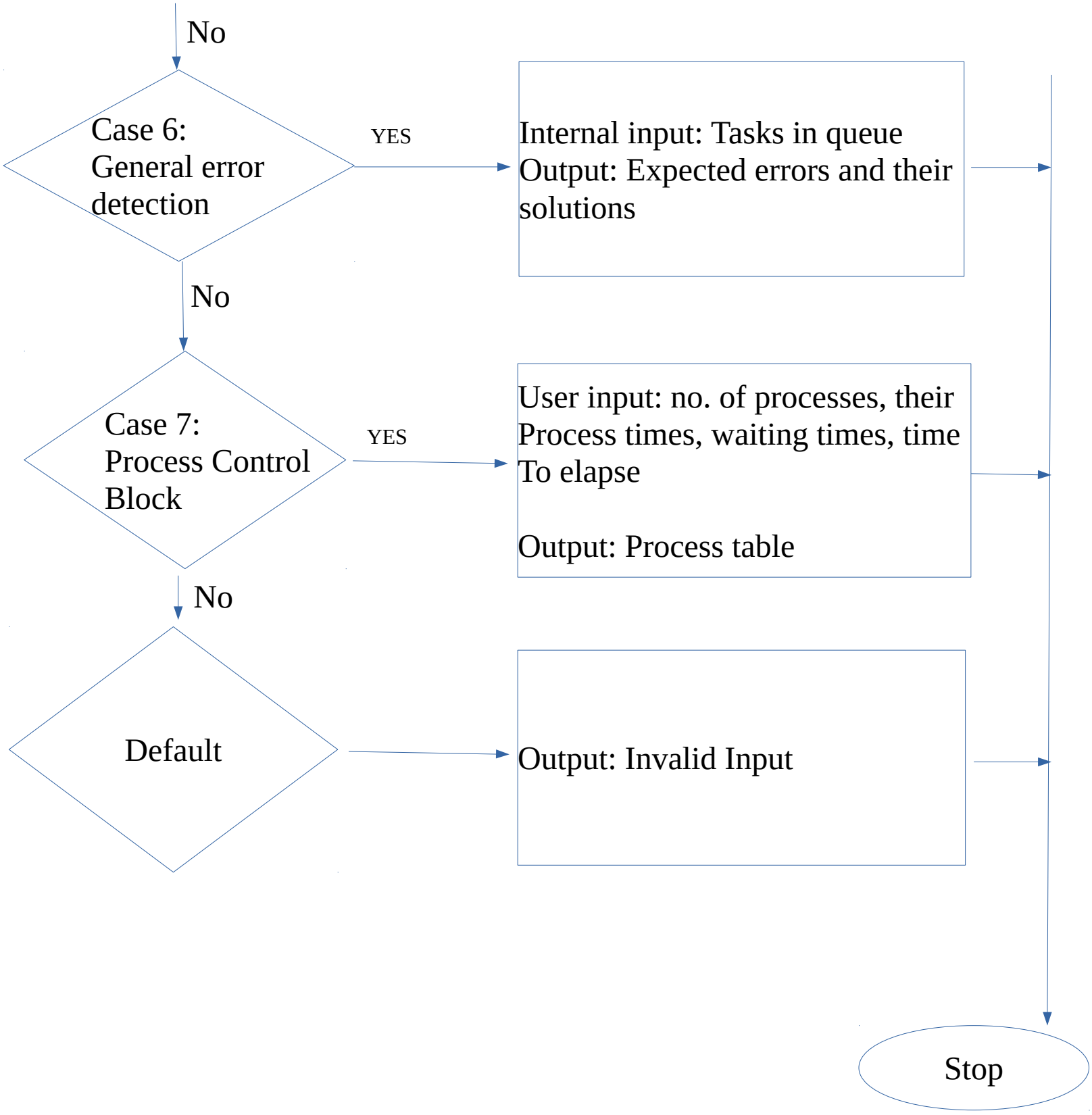
The pictorial diagram of the layout of our project is:



Workflow of the project:




No
Case 2:
Encryption
YES
Input: numeric password
Output: Encrypted password
Every step in the process is
Displayed
No
Case 3:
File
Management
YES
Outputs the error number and
The error message using the
Inbuilt function errno.h
No
Case 4:
CPU Scheduling
YES
Internal input: Waiting time and
Burst time of the processes in
Queue
Output: Process with highest
Response ratio
No
Case 5:
String error
detection
YES
Internal input: tasks regarding
Strings
Output: Possible errors and
Solutions after solving the errors


No
Case 6:
General error
detection
YES
Internal input: Tasks in queue
Output: Expected errors and their
solutions
No
Case 7:
Process Control
Block
YES
User input: no. of processes, their
Process times, waiting times, time
To elapse
Output: Process table
No
Default
Output: Invalid Input
Stop

Modularity:

Modularity refers to the extent to which a software can be divided into smaller modules. In our project every function performs a particular operation, this also gives the programmers an additional feature of **re-usability**. We have used the concept of wrapper functions whenever we had to develop an operation that involved multiple sub-operations, thereby increasing the modularity of our code.

The design of our project comprises of many systems which contains several sub-systems and those sub-systems further contains their sub-systems. So, designing a complete system in one go comprising of each and every required functionality is a hectic work and the process can have many errors because of its vast size. Thus in order to solve this problem we had to breakdown the complete software into various modules. Every module is independent and can be modified without disturbing any other module in the program.

Our project focuses on developing a more protected system, if developing a useful software solution is one side of a coin protecting it from hackers is another side. The integration of safety recommendations into the project can be meaningful when every recommendation applied suits the scene it is applied to, modular programming approach makes it that much more easier to suit a recommendation into a particular situation.

When every module talks about a specific part of the project, safeguarding it, applying safety recommendations on it becomes easier. Unlike a situation where the entire code is dumped into the main function, where everything becomes messy and safeguarding it then becomes a tough challenge.
If the application of any safety recommendation causes an error as a side effect, modularity helps us in solving it.

Security of the computer is one of the most fundamental tasks of any operating system, our project demonstrates security in three ways: Data sanitization, data validation and the most important of all **Encryption**, all the three of the above ways are demonstrated as a separate module. Applying recommendations on them became that much more easier for us, we could achieve a feat of eleven recommendations only for the case of numeric encryption. This was possible because numeric encryption was a separate module by itself and hence we could identify all the recommendations that suited the situation, and that part of our software became stronger to vulnerabiltities.

Using macros simplifies the task of programming a problem, using inline functions speeds up the execution of the program and in our software we've used these two extensively, placing a macro or an inline function on top of a function that solves the problem increases the readability of the program, and this is possible only if a function doesn't solve many problems together, that's modularity helping us use macros and inline functions on demand.

Readability:

We have put in a lot of our efforts to increase the readability of our code. ADTs were of immense help for us in doing this. We've used enums, for example

enum processes{ process1, process2, process3, process4, process5};

This helps us use process1 for the 1st process instead of the nos. 1,2,…

We've made sure that no datatype is seen in the entire code, every place the datatype appears is replaced with an user-defined datatype, hence increasing the readability of the code.

For instance,

DCL02-A. Use visually distinct identifiers.

DCL23-C. Guarantee that mutually visible identifiers are unique.

Implementing such recommendations has increased the readability of the code.

Abstraction:

Data structures like queues are defined in the ADT file, and only a user-defined name is used in the functions thereby increasing the abstraction levels. We have more than 1500 lines of code just for the functions and nowhere will you find a primitive datatype, a user-defined datatype is built for every occurrence of a datatype.

We have used different files for functions, driver program, function prototypes and ADTs, this architecture increases the abstraction by fourfold.
Hiding relevant data reduces the complexity and increases the efficiency of the program because of reusability.

We've processed recommendations that are applicable only for the abstraction, for instance

DCL05-C. Use typedefs of non-pointer types only.

This recommendation is used for

```
struct obj {
  int front, rear;
  char queue[1000][10];

};
typedef struct obj PASSWORD;
```

This is the abstraction for a QUEUE, CERT recommendations have helped us in abstraction at various instances and the above code is one such situation.

Implementation

Code for the Compliant Version :

main_for_compliant_code.c

```
#include "abstract_data_types_for_compliant_code.h"
#include "function_prototypes_for_compliant_code.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "mystdio.h"


integer_variable main()
{


printf("Welcome!!\n");
//printf("Enter 1 to sign in
integer_type_variable x;

PASSWORD ptr;
ptr.front = 0;
```

```
ptr.rear = 0;

Password_verification_system(&ptr);

constant_buffer_structure_variable buf = {10,20};

flex_array_struct_variable struct_a;
flex_array_struct_variable struct_b;

flex_array_struct_pointer p1 = &(struct_a);
flex_array_struct_pointer p2 = &(struct_b);



struct_a.data[0] = 1;
struct_a.data[1] = 2;
struct_a.data[2] = 4;


number_of_observations n;
        waitingtime_pointer stime;
        process_identification_number processID[10];
        executioncontext exc[5];
        time_elapsed time=0;
        //float stime[5]={0,-2,-3,-4,-1};
        //float ptime[5]=(4,6,7,3,8};

        process_state_pointer state =
(process_state_pointer)malloc(sizeof(char_variable)*10);
        integer_variable ptr1[5];
        processingtime_pointer ptime;
```

```
printf("Enter   \n1 to find the time taken by a carry_look_ahead_adder       \n\n2 to ENCRYPT a numeric password       \n\n3 to open a file \n\n4 to see CPU scheduling  \n\n5 to see ERROR DETECTION IN STRINGS \n\n6 to see a general case of ERROR DETECTION      \n\n7 to see the status of the processes at any instant of time     \n\n  ");
scanf("%d",&x);


integer_type_variable clear_array[12];

switch(x)
{
case 1:

time_taken_carry_lookahead_adder();

break;



case 2:

NUMERIC_PASSWORD_ENCRYPTION();

break;

case 3:
```

```
open_file_table();

break;


case 4:

CPU_scheduling();

break;


case 5:

string_processing_error_detection();

break;


case 6:

Artificial_error_detection_system(&buf, p1, p2);

break;

case 7:
printf("enter the number of processes\n");
            scanf("%d",&n);
            printf("enter the proceesor time of %d observations\n",n);
            ptime=pcb1(n);
            printf("enter the waiting time of %d observations\n",n);
            stime=pcb2(n);
```

```
            pcb(processID,exc,time,ptime,stime,state,ptr1,n);
            printf("-----------------------------------------\n");
            printf("PROCESSID\tPROCESSINGTIME\n");
            for(integer_variable i=0;i<n;i++)
            {
                  printf("%d\t\t%f\n",processID[i],ptime[i]);
            }

                                    //MEM30-C. Do not access freed
memory

            free(ptime);
                                 //MEM01-C. Store a new value in pointers
immediately after free()
            ptime=NULL;
//MEM31-C. Free dynamically allocated memory when no longer
needed
            free(stime);
            stime=NULL;

                //MEM34-C. Only free memory allocated dynamically

break;

default:
printf("Invalid case\n");

}
return 0;
}
```

abstract_data_types_for_compliant_code.h

```
typedef int size_of_cla;

typedef int no_of_blocks_in_cla;

typedef int time_variable;

typedef int password_verifier;

typedef char dangerous_characters;

typedef int integer_type_variable;

typedef int counter;

typedef int size_variable;

typedef char* string_pointer;

typedef char string_array;

typedef int integer_array;

typedef long long_type_variable;

typedef long* long_type_pointer;

typedef int numeric_encryption_procedure_result;

typedef int numeric_encryption_variable;
```

```
typedef int result_variable;

typedef int error_number_variable;

typedef int scheduler;

typedef int scheduler_variable;

typedef int number_of_observations;

typedef int process_identification_number;

typedef int* process_identification_number_pointer;

typedef int time_elapsed;

typedef float executioncontext;

typedef float* executioncontext_pointer;

typedef int random_a;

typedef int validation;

typedef float* processingtime_pointer;

typedef float* waitingtime_pointer;

typedef float* processtime_pointer;

typedef float* waittime_pointer;

typedef float* float_pointer;

typedef char** process_state_pointer;
```

```
typedef int* parent_pointer;

typedef int** parent_pointer_pointer;

typedef int integer_variable;

typedef int size_of_n;

typedef char char_variable;

typedef int* integer_pointer;

typedef const struct buffer* detection_pointer;

typedef struct buffer* buffer_structure_pointer;

typedef struct buffer buffer_structure_variable;

typedef const struct buffer constant_buffer_structure_variable;

typedef struct flex_array_struct* flex_array_struct_pointer;

typedef struct flex_array_struct flex_array_struct_variable;

typedef unsigned int non_negative_integer;

typedef FILE* filepointer;

enum processes{ process1, process2, process3, process4, process5};

// QUEUE ADT

// 19) RECOMMENDATION: DCL05-C. Use typedefs of non-pointer types only.
```

//This compliant solution makes use of type definitions but does not declare a pointer type and so cannot be used in a const-incorrect manner

```
struct obj {
  int front, rear;
  char queue[1000][10];

};
typedef struct obj PASSWORD;


typedef const struct buffer {

integer_type_variable x;
integer_type_variable y;

} BUFFER;



struct flex_array_struct {
  size_variable num;
  int data[];
};
```

function_prototypes_for_compliant_code.h

```
//#include "abstract_data_types_for_compliant_code.h"

extern password_verifier password_check_using_sanitization(string_pointer str, size_variable size);


extern void time_taken_carry_lookahead_adder();


extern password_verifier string_validity( string_pointer destination_string, size_variable destination_size, string_pointer source_string, size_variable source_size);


extern numeric_encryption_procedure_result multiple_of_sum(numeric_encryption_variable no1, numeric_encryption_variable no2, numeric_encryption_variable multiple);


extern void NUMERIC_PASSWORD_ENCRYPTION();


extern scheduler Response_Ratio(scheduler_variable waiting_time, scheduler_variable burst_time);


extern void open_file_table();


extern void CPU_scheduling();

extern void Password_verification_system( PASSWORD *ptr);
```

extern void string_processing_error_detection();

extern void test18();

extern void Artificial_error_detection_system(detection_pointer buf, flex_array_struct_pointer p1, flex_array_struct_pointer p2); // Here const struct should also be typedefed

extern void delay(int number_of_seconds);

static validation valid(float_pointer pntr);

extern processtime_pointer pcb1(number_of_observations n);

extern waittime_pointer pcb2(number_of_observations n);

extern void pcb(process_identification_number_pointer processID,executioncontext_pointer exc,time_elapsed time,processingtime_pointer ptime,waitingtime_pointer stime,process_state_pointer state,parent_pointer ptr,number_of_observations n);

mystdio.h

void test1();

functions_for_compliant_code_part1.c

/*
RECOMMENDATIONS USED:

1) RECOMMENDATION: PRE02-A. Macro expansion should always be parenthesized for function-like macros.

2) RECOMMENDATION: PRE00-A. Prefer inline functions to macros.

3) RECOMMENDATION: STR02-C. Sanitize data passed to complex subsystems.

4) RECOMMENDATION: STR03-A. Do not inadvertently truncate a null terminated byte string.

5) RECOMMENDATION: Pass the size of the array whenever the array is passed to a function.

6) RECOMMENDATION: PRE01-A. Use parentheses within macros around variable names.

7) RECOMMENDATION: PRE04-A. Do not reuse a standard header file name.

8) (BEYOND_SYLLABUS) RECCOMENDATION: PRE08-C. Guarantee that header file names are unique.

9) RECOMMENDATION: PRE30-C. Do not create a universal character name through concatenation.

10) RECOMMENDATION: PRE03-A. Avoid invoking a macro when trying to invoke a function.

11) RECOMMENDATION: PRE05-A. Avoid using repeated question marks.

12) RECOMMENDATION: DCL02-A. Use visually distinct identifiers.

13) RECOMMENDATION: DCL23-C. Guarantee that mutually visible identifiers are unique.

14) RECOMMENDATION: DCL03-A. Place const as the rightmost declaration specifier.

15) RECOMMENDATION: DCL04-C. Do not declare more than one variable per declaration.

16) RECOMMENDATION: DCL07-C. Include the appropriate type information in function declarators.

17) RECOMMENDATION: DCL00-C. Const-qualify immutable objects.

18) RECOMMENDATION: DCL01-C. Do not reuse variable names in subscopes.

19) RECOMMENDATION: DCL05-C. Use typedefs of non-pointer types only.

20) RECOMMENDATION: DCL06-C. Use meaningful symbolic constants to represent literal values.

21) RECOMMENDATION: DCL08-A. Declare function pointers using compatible types.

22) RECOMMENDATION: DCL09-A. Declare functions that return an errno with a return type of errno_t.

23) RECOMMENDATION: DCL10-C. Maintain the contract between the writer and caller of variadic functions.

24) RECOMMENDATION: DCL30-C. Declare objects with appropriate storage durations.

25) RECOMMENDATION: EXP03-C. Do not assume the size of a structure is the sum of the sizes of its members.

26) RECOMMENDATION: EXP43-C. Avoid undefined behavior when using restrict-qualified pointers.

27) RECOMMENDATION: STR05-C. Use pointers to const when referring to string literals.

28) RECOMMENDATION: STR06-C. Do not assume that strtok() leaves the parse string unchanged.

29) RULE: STR30-C. Do not attempt to modify string literals.

30) (BEYOND_SYLLABUS) RECOMMENDATION: STR11-C. Do not specify the bound of a character array initialized with a string literal.

31) RECOMMENDATION: STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator.

32) RECOMMENDATION: STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string.

33) RECOMMENDATION: STR10-C. Do not concatenate different type of string literals.

34) RECOMMENDATION: MEM03-C. Clear sensitive information stored in reusable resources.

35) RECOMMENDATION: MEM06-A. Do not use user-defined functions as parameters to allocation routines.

37) (BEYOND_SYLLABUS) RECOMMENDATION: MEM36-C. Do not modify the alignment of objects by calling realloc().

38) RECOMMENDATION: MEM32-C Detect and handle critical memory allocation errors.

39) RECOMMENDATION: MEM33-C. Allocate and copy structures containing a flexible array member dynamically.

40) RECOMMENDATION: MEM35-C. Allocate sufficient memory for an object.

41) (BEYOND_SYLLABUS) RECOMMENDATION: INT30-C. Ensure that unsigned integer operations do not wrap.

42) RECOMMENDATION: ENV03-C. Sanitize the environment when invoking external programs.

43) RECOMMENDATION: STR00-A. Use TR 24731 for remediation of existing string manipulation code.

44) RECOMMENDATION: STR01-A. Use managed strings for development of new string manipulation code.

45) RECOMMENDATION: STR07-A. Take care when calling realloc() on a null terminated byte string.

*/

```
#include "abstract_data_types_for_compliant_code.h"
#include "function_prototypes_for_compliant_code.h"
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>



// CASE 1


// 1) RECOMMENDATION: PRE02-A. Macro expansion should always be parenthesized for function-like macros.


#define no_of_AND_OR(N,k) (N/k-1)

// 9) RECOMMENDATION: PRE30-C. Do not create a universal character name through concatenation.
#define assign(ucn, val) ucn = val


void time_taken_carry_lookahead_adder()
{
no_of_blocks_in_cla  k;
size_of_cla N;

// 13) RECOMMENDATION: DCL23-C. Guarantee that mutually visible identifiers are unique.
//2 RANDOM LONG VARIABLES WHICH HAVE A LONG COMMON PART CAN STILL BE IDENTIFIED EASILY IF THE DISTINCT VARIABLE IS IN THE BEGINNING.  the significant characters in each identifier must differ.


// 12) RECOMMENDATION: DCL02-A. Use visually distinct identifiers.
time_variable CLA_time_consumption;
```

```
time_variable pg_time_consumption;
time_variable pg_block_time_consumption;
time_variable AND_OR_time_consumption;
time_variable FA_time_consumption;

pg_time_consumption = 100; //100ps is the delay of each two-input gate.
pg_block_time_consumption = 600; // 6*100 ps, because there are 6 two-input gates in a block.
N = 32; // here we are talking about a 32-bit CLA



integer_variable \u0401;
assign(\u0401, 4); // here the 32-bit CLA is made up of 4-bit blocks

k = 4;

AND_OR_time_consumption = 200 ; //2*100, because there are 2 2-input gates, one AND and one OR

FA_time_consumption = 300; //300ps is the full adder delay

CLA_time_consumption = pg_time_consumption +
pg_block_time_consumption + no_of_AND_OR(N,\
u0401)*AND_OR_time_consumption+ \u0401*FA_time_consumption ;
        //CALLING MACRO

printf("Time taken by the Carry look ahead adder is %d \
n",CLA_time_consumption);
}
```

//CASE 2:

// 7) RECOMMENDATION: PRE04-A. Do not reuse a standard header file name.

// 8) (BEYOND_SYLLABUS) RECCOMENDATION: PRE08-C. Guarantee that header file names are unique.

// 8) BECAUSE WE ARE GIVING THE LOCAL LIBRARY A UNIQUE NAME.

// The name is misleading and hence the recommendation asks us not to use the standard header file name to a local header file. IT IS TAKEN CARE OF HERE IN THE COMPLIANT VERSION.

```
#include "mystdio.h"

//5) RECOMMENDATION: Pass the size of the array whenever the array is passed to a function.

//This function is declared in the local version stdio.h file named mystdio.h
void clear(integer_pointer array,size_variable size)
{
integer_type_variable i;
for (i = 0; i < size; i++)
array[i] = 0;

}
```

// 6) RECOMMENDATION: PRE01-A. Use parentheses within macros around variable names

#define mul(a,b) ((a)*(b))

// 16) RECOMMENDATION: DCL07-C. Include the appropriate type information in function declarators.
//In the non-compliant version the variables are declared outside the function definition.
numeric_encryption_procedure_result
multiple_of_sum(numeric_encryption_variable no1,
numeric_encryption_variable no2, numeric_encryption_variable multiple)
{
result_variable result;

result = mul(no1+no2, multiple);

return result;

}

//2) RECOMMENDATION: PRE00-A. Prefer inline functions to macros

// inline functions don't need a function prototype

static inline numeric_encryption_procedure_result square(int a)// static keyword forces the compiler to consider this inline function in the linker.

```
{
return (a*a);
}


// 10) RECOMMENDATION: PRE03-A. Avoid invoking a macro when trying to invoke a function.


#define puts(x) printf("I am in macro\n")
#undef puts

void NUMERIC_PASSWORD_ENCRYPTION()
{

// 20) Recommendation: DCL06-C. Use meaningful symbolic constants to represent literal values.
enum { MAX_PASSWORD_SIZE=4 };
integer_array arr[MAX_PASSWORD_SIZE];

//we will initialise this array now

clear(arr, (sizeof(arr) / sizeof(arr[0])));

integer_type_variable i;

printf("We've created an array of 4 characters\n");

printf("The array after initialization is: \n");

for(i=1;i<=MAX_PASSWORD_SIZE;i++)
printf("%d) \t %d\n",i,arr[i-1]);
```

```
printf("An ENCRYPTION SYSTEM asks the OS to do the following tasks:
        \n1) add 1 to every number \n2)find the square of the new number \n3)
adding the no. with 19 and multiply it by 8\n");

printf("Please ENTER A 4 DIGIT PASSWORD (give spaces b/w the nos.):
");
for(i=0;i<MAX_PASSWORD_SIZE;i++)
scanf("%d",&arr[i]);


//21) RECOMMENDATION: DCL08-A. Declare function pointers using
compatible types.
integer_type_variable (*multiple_of_sum_fn_ptr) (integer_type_variable,
integer_type_variable, integer_type_variable) ; // * CANNOT BE AVOIDED
BY AN ADT IN THIS CASE
multiple_of_sum_fn_ptr = &multiple_of_sum;      //SAME NO. OF
PARAMETERS



for(i=0;i<MAX_PASSWORD_SIZE;i++)
{
printf("For %d : \n",arr[i]);

arr[i] = square(++arr[i]);

printf("After adding 1 and finding the square of it : %d\n",arr[i]);

arr[i] = multiple_of_sum_fn_ptr(arr[i],19,8); // (arr[i] + 19) * 8

printf("After adding the no. with the next number in the password and
multiply it by 8 : %d\n",arr[i]);
}
```

```
// 11) RECOMMENDATION: PRE05-A. Avoid using repeated question
marks.

puts("Finally the ENCRYPTED PASSWORD is, can you believe it ?""?! \
n");//A TRIGRAPH SEQUENCE IS AVOIDED BY CONCATENATING 2
STRINGS.

for(i=0;i<MAX_PASSWORD_SIZE;i++)
printf("%d",arr[i]);

printf("\n");



}




//CASE 3:



// 22) RECOMMENDATION: DCL09-A. Declare functions that return an
errno with a return type of errno_t.

#include <errno.h>
#include <string.h>
//THIS FUNCTION HAS TO BE USED LATER

void open_file_table() {
error_number_variable errno ;
file_pointer pf;
error_number_variable errnum;
pf = fopen ("unexist.txt", "rb");
```

```
   if (pf == NULL) {

      errnum = errno;
      fprintf(stderr, "Value of errno: %d\n", errno);
      perror("Error printed by perror");
      fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
   } else {

     printf("OS maintains an Open File Table, this file's name unexist.txt will
be added to the table\n");

   }

}
```

//CASE 4:

// 1) RECOMMENDATION: PRE02-A. Macro expansion should always be parenthesized for function-like macros

//calculating response ratio,//Response Ratio = (Waiting Time + Burst time) / Burst time

```
#define resp(wt,bt) (wt+bt)
```

scheduler Response_Ratio(scheduler_variable waiting_time, scheduler_variable burst_time)

```
{


scheduler Resp_Ratio;

Resp_Ratio=resp(waiting_time,burst_time)/burst_time;

return(Resp_Ratio);

}
```

```
void CPU_scheduling()
{

// 17) RECOMMENDATION:  DCL00-C. Const-qualify immutable objects.
// 14) RECOMMENDATION: DCL03-A. Place const as the rightmost
declaration specifier.
//EVEN THOUGH ANY 2 VARIABLES ARE CONST QUALIFIED,
THEIR DATATYPES ARE VISUALLY EASY TO IDENTIFY.

counter const no_of_processes = 5;// no_of_processes cannot be modified
again, its value is protected from accidental modification.

// CONST IS NOT MADE AN ADT TO ILLUSTRATE THE
RECOMMENDATION


integer_type_variable j;
```

```
integer_array waiting_time[] = { 0, 2, 4, 6, 8 };
integer_array burst_time[] = { 3, 6, 4, 5, 2 }; //Burst time is the amount of time required by a process for executing on CPU.

printf("The details of the processes waiting are: \n\n");
printf("Waiting time \tBurst time\n");
for(j= process1;j<no_of_processes;j++)
printf("%d\t\t%d\n",waiting_time[j], burst_time[j] );


integer_type_variable
i,max_response_ratio=0,index_of_max_response_ratio=0;

max_response_ratio = Response_Ratio(waiting_time[0],burst_time[0]);



for(i= process1;i<no_of_processes;i++)
{

// 18) RECOMMENDATION:  DCL01-C. Do not reuse variable names in subscopes.
integer_type_variable temp;

temp = Response_Ratio(waiting_time[i],burst_time[i]);

if(temp > max_response_ratio)
{
max_response_ratio = temp;
index_of_max_response_ratio = i;
}
}

printf("The process that will be given CPU time is the one with: \n waiting time = %d \n burst time = %d \n It's response ratio = %d\n",
```

```
waiting_time[index_of_max_response_ratio],
burst_time[index_of_max_response_ratio], max_response_ratio);

}
```

```
// STRING PASSWORD SECTION


//3) RECOMMENDATION: STR02-C. Sanitize data passed to complex
subsystems
password_verifier password_check_using_sanitization(string_pointer str,
size_variable size)
{

dangerous_characters badchars[] = {'%','^','&','*','$','#'};

integer_type_variable i, j;

counter count = 0;

for(i=0;i<size;i++)
{
for(j=0;j<6;j++)
{
if(str[i]==badchars[j])
count++;
}
```

```
}

if(count==0)
return 1;
else
return 0; // password is NOT safe
}
```

```
//4) RECOMMENDATION: STR03-A. Do not inadvertently truncate a null terminated byte string
password_verifier string_validity( string_pointer destination_string, size_variable destination_size, string_pointer source_string, size_variable source_size)
{
```

```
if (source_string == NULL) {
printf("Source string is NULL\n");
  return 0; //
}
else if (source_size >= destination_size) {
printf("size not matching\n");

  return 0; //
}
else {
  strcpy(destination_string, source_string);   //THE LAST CHARACTER IN THE DESTINATION STRING IS '\0'
```

```
  return 1;
}
}
```

```
//THE STRUCTURE PASSWORD IS IN ADT'S .h FILE
void Password_verification_system( PASSWORD *ptr) // pointer should not be avoided by using an ADT, because that's a recommendation.
{

// STRING AND NUMERIC PASSWORDS HAS TO BE ADDED TO THE QUEUE IN THE END

integer_type_variable  size=0, check_bit, sanitize_bit=0;

printf("Enter the SIZE of the password, LESS THAN 9 CHARACTERS\n");

scanf("%d",&size);


// 15) RECOMMENDATION: DCL04-C. Do not declare more than one variable per declaration.

// Arrays and pointers can get confusing

string_array source_array[10];

// 31) RECOMMENDATION: STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator.
```

```
string_array str[size+1]; // extra space is for '\0'

string_pointer destination_array = (string_pointer )calloc(10,1);

printf("Enter the password, DO NOT USE any of '^','&','*','$','#',... \n");

scanf("%s",str);
size=sizeof(str);
sanitize_bit = password_check_using_sanitization(str, size);


strcpy(source_array,str) ; // size is less than the array size

check_bit = string_validity(destination_array,10,source_array,size);

if( check_bit == 1 && sanitize_bit == 1)
{
printf("The password is accepted \n"); //The string is successfully copied to
the new location destination_array



// 24) RECOMMENDATION: DCL30-C. Declare objects with appropriate
storage durations.

// object created and initialised in main file, defined in abstract file and used
in functions file, possible because object declared in one file is made to be
visible in all the files.


// ADDING PASSWORD TO THE QUEUE

str[size] = '\0';

strcpy(ptr->queue[ptr->front++], str);
```

// 32) RECOMMENDATION: STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string.

```
printf("Your password: %s is added to the queue\n",str);


}
else
printf("The password is NOT accepted\n");

}
```

// 23) RECOMMENDATION: DCL10-C. Maintain the contract between the writer and caller of variadic functions.

```
#include <stdarg.h>


enum { va_eol = -1 };
 non_negative_integer average(integer_variable first, ...);

non_negative_integer average(integer_variable first, ...)
{
  non_negative_integer count = 0;
  non_negative_integer sum = 0;
  integer_type_variable i = first;
  va_list args;

  va_start(args, first);

  while (i != va_eol) {
```

```
    sum += i;
    count++;
    i = va_arg(args, integer_type_variable);
  }

  va_end(args);
  return(count ? (sum / count) : 0);
}
```

```
//func takes a pointer to a const struct obj
//func does not take a const pointer to a struct obj.


void Artificial_error_detection_system(detection_pointer buf,
flex_array_struct_pointer p1, flex_array_struct_pointer p2)
{
// Cannot modify o's contents


// 25) RECOMMENDATION: EXP03-C. Do not assume the size of a
structure is the sum of the sizes of its members.

buffer_structure_pointer buf_cpy =
    (buffer_structure_pointer)malloc(sizeof(buffer_structure_variable));

  if (buf_cpy == NULL) {
    // Handle malloc() error
printf("FAILED to create an array\n");
 }
```

```
  memcpy(buf_cpy, buf, sizeof(buffer_structure_variable));
```

// 37) RECOMMENDATION: MEM36-C. Do not modify the alignment of objects by calling realloc().

// 38) RECOMMENDATION: MEM32-C Detect and handle critical memory allocation errors.

// This compliant solution allocates resize bytes of new memory with the same alignment as the old memory, copies the original memory content, and then frees the old memory. This solution has implementation-defined behavior because it depends on whether extended alignments in excess of _Alignof (max_align_t) are supported and the contexts in which they are supported. If not supported, the behavior of this compliant solution is undefined.

```
#include <stdlib.h>
#include <string.h>
// now let's create 2 pointers: ptr and ptr1

// PART 6:
size_t resize = 1024;
  size_t alignment = 1 << 12;
  integer_pointer ptr;
  integer_pointer ptr1;

  if (NULL == (ptr = (integer_pointer)aligned_alloc(alignment,
                                    sizeof(integer_variable)))) {
    // Handle error
printf("FAILED to create an array\n");
```

```
  }

  if (NULL == (ptr1 = (integer_pointer)aligned_alloc(alignment,
                                    resize))) {
    // Handle error
printf("FAILED to create an array\n");
  }

  if (NULL == (memcpy(ptr1, ptr, sizeof(integer_variable)))) {
    // Handle error
printf("FAILED to copy an array\n");
  }

// 26) RECOMMENDATION: EXP43-C. Avoid undefined behavior when
using restrict-qualified pointers.

// WE CREATED A COPY BECAUSE WE CAN'T ASSIGN A CONST
POINTER.
ptr = &(buf_cpy->x);
ptr1 = &(buf_cpy->y);



printf("TASK 1: To find the average of the values in 2 structures and store
them in an array: \n\n");

printf("EXPECTED ERRORS: \n\n1) CONSTANT STRUCTURE CANNOT
BE MANIPULATED \n\n2) STACK OVERFLOW DUE TO GARBAGE
VALUES IN FLEXIBLE ARRAYS IN STRUCTURES\n\n");

printf("ERROR DETECTION SYSTEM HAS RESOLVED POSSIBLE
ERRORS\n\n");

int t1,t2;
```

// 23) RECOMMENDATION: DCL10-C. Maintain the contract between the writer and caller of variadic functions.

// 35) RECOMMENDATION: MEM06-A. Do not use user-defined functions as parameters to allocation routines.

```
t1 = average(*ptr, *ptr1, va_eol);
printf("The average of the 1st structure is %d \n",t1);
```

// 39) RECOMMENDATION: MEM33-C. Allocate and copy structures containing a flexible array member dynamically.

```
memcpy(p2, p1,
        sizeof(flex_array_struct_variable) + (sizeof(int)
          * 3));

t2 = average(p2->data[0], p2->data[1], p2->data[2], va_eol);

printf("The average of the 2nd structure is %d \n",t2);
```

```
// to create a long array
integer_type_variable len = 2;
```

// 40) RECOMMENDATION: MEM35-C. Allocate sufficient memory for an object.

// 41) (BEYOND_SYLLABUS) RECOMMENDATION: INT30-C. Ensure that unsigned integer operations do not wrap.

```
long_type_pointer p;
long_type_variable SIZE_MAX = 2147483647;
  if (len == 0 || len > SIZE_MAX / sizeof(long_type_variable)) {
    // Handle overflow
  }
  p = (long_type_pointer)malloc(len * sizeof(long_type_variable)); // long is
used to create an array of long, no chance of overflow.
  if (p == NULL) {
    // Handle error
    printf("FAILED to create an array\n");
  }
  //free(p);

p[0] = average(*ptr, *ptr1, va_eol);
p[1] = average(p2->data[0], p2->data[1], p2->data[2], va_eol);




printf("\nThe elements have been added to an array\n\n");


// 42) RECOMMENDATION: ENV03-C. Sanitize the environment when
invoking external programs.

if (clearenv() != 0) {
  printf("The environment is not safe\n");
}

if(getenv("PATH")!=NULL)
{
// HANDLE ERROR
printf("Please correct your path\n");
}
```

```
  free(ptr);
}
```

```
void string_processing_error_detection()
{

printf("\nOS DETECTS ERRORS BEFOREHAND TO ENSURE A
SMOOTH PROCESSING OF ITS INSTRUCTIONS\n\n");

// 27) RECOMMENDATION: STR05-C. Use pointers to const when
referring to string literals.
// IF THERE'S NO CONST MODIFICATION WOULD RESULT IN AN
UNDEFINED BEHAVIOUR.

printf("TASK 1: To modify a constant string:\n\n");
const string_pointer c = "Hello";
printf("A constant string: %s\n\n",c);
printf("NO MODIFICATION CAN BE MADE TO A CONSTANT STRING\
n\n\n");


// 29) RULE: STR30-C. Do not attempt to modify string literals.

// 30) (BEYOND_SYLLABUS) RECOMMENDATION: STR11-C. Do not
specify the bound of a character array initialized with a string literal.

// 30) is str[14] is not needed, str[] will do
```

// As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely.

```
printf("TASK 2: A CHANGE FOR A DIFFERENCE:\n\n");
string_array str1[] = "Take Artificial Intelligence";
printf("An array type string: %s\n\n",str1);
str1[0] = 'M';
printf("Can be modified as it is not pointed by a pointer\n\n");
printf("Modified string: %s\n\n\n",str1);
```

// 35) RECOMMENDATION: MEM03-C. Clear sensitive information stored in reusable resources.

```
printf("TASK 3: To copy a password: \n\n");
strcpy(str1,"PASSWORD: UXf64hTrfd56");
printf("A string has: %s\n\n",str1);

strcpy(str1,"NA");
printf("The value in the string after use is: %s\n\n",str1);

printf("DELETED IMMEDIATELY AFTER USE, COULD NOT BE COPIED\n\n\n");
```

```
printf("TASK 4: To split first name and last name\n");
```

// 33) RECOMMENDATION: STR10-C. Do not concatenate different type of string literals.

```
string_pointer msg = "First-name "
          "Last_name";
printf("Input string: %s\n\n",msg);


printf("After processing:\n\n");

string_pointer token;

// 28) RECOMMENDATION: STR06-C. Do not assume that strtok() leaves the parse string unchanged.
// THEREFORE A COPY IS MADE

string_pointer copy = (string_pointer )malloc(strlen(msg) + 1);
if (copy == NULL) {
  // Handle error
  printf("FAILED TO ALLOCATE MEMORY\n");
}
strcpy(copy, msg);
token = strtok(copy, " ");
puts(token);


while (token = strtok(0, " ")) {
  puts(token);
}
printf("\n\n");


printf("TASK 5: To make space for the null character after reallocation\n\n");

string_pointer s1 = (string_pointer )malloc(sizeof(char)*4);
string_pointer ptr_new;
strcpy(s1,"PES");
printf("Input string : %s\n\n",s1);
```

```
// 45) RECOMMENDATION: STR07-A. Take care when calling realloc() on a null terminated byte string.
ptr_new = (string_pointer)realloc(s1, sizeof(char_variable)*5);

ptr_new[sizeof(char_variable)*4] = '\0';

printf("Null character added to %s\n\n",ptr_new);


strcpy(str1," A I ");
printf("TASK 6: TO SAFELY COPY A STRING \n\n");
// 43) RECOMMENDATION: STR00-A. Use TR 24731 for remediation of existing string manipulation code.

// 44) RECOMMENDATION: STR01-A. Use managed strings for development of new string manipulation code.
string_array str2[30];
printf("Input string = %s\n\n",str1);
#ifdef __STDC_LIB_EXT1__    // __STDC_LIB_EXT1__ checks if the ISO TR 24731 functions are supported by the compiler and executes only if the compiler supports them.

integer_type_variable r = strcpy_s(str2, sizeof(str1), str1);

#else

strcpy(str2,str1);
#endif

printf("After copying: %s\n\n",str2);
}
```

functions_for_compliant_code_part2.c

```
/*
1) MEM10-C. Define and use a pointer validation function

2) MEM02-A. Do not cast the return value from malloc()

3) MEM05-C. Avoid large stack allocations

4) MEM04-C. Beware of zero-length allocations

5) MEM00-C. Allocate and free memory in the same module, at the same
level of abstraction.

6) MEM30-C. Do not access freed memory

7) MEM01-C. Store a new value in pointers immediately after free()

8) MEM31-C. Free dynamically allocated memory when no longer needed

9) MEM34-C. Only free memory allocated dynamically

10) MEM07-C. Ensure that the arguments to calloc(), when multiplied, do
not wrap
*/


#include "abstract_data_types_for_compliant_code.h"

#include "function_prototypes_for_compliant_code.h"
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

static validation valid(float_pointer pntr);
```

```
validation valid(float_pointer
pntr)                                                   //MEM10-C. Define and use a
pointer validation function

        {       if(pntr==NULL)
                        return 0; // function checks if the ptr is valid
                else
                        return 1;

        }
processtime_pointer  pcb1(number_of_observations n)
{

        processingtime_pointer ptime;
         if (n== 0) {
    exit(1);
  }



        ptime=malloc(n*sizeof(float));

        if (!ptime) {
                                                //"MEM02-A. Do not cast the return
value from malloc() // no (int*)
    exit(1);
                                        //"MEM05-C. Avoid large stack allocations
  }

                                                        //MEM04-C. Beware of zero-
length allocations       // 0 length => exit
        for(integer_variable k=0;k<n;k++)
        {
                scanf("%f",&ptime[k]);
        }
```

```
        return ptime;

}
waittime_pointer pcb2(number_of_observations n)
{
        waitingtime_pointer stime;
         size_of_n size=2147483647; // 10) MEM07-C. Ensure that the
arguments to calloc(), when multiplied, do not wrap
         if(n>(size/sizeof(float)))
         {
                 exit(1);
         }

        //printf("%d",a);
        /*int valid(float *ptr)
        {       if(ptr==NULL)
                        return 0;
                else
                        return 1;

        }
        //if (n >(sizeof(stime)/sizeof(float)))
        //      {exit(1);
  /* Handle error condition */;
     //  }
        stime=calloc(n,sizeof(float));
        if(valid(stime))
        {
        for(integer_variable k=0;k<n;k++)
        {
                scanf("%f",&stime[k]);
        }
        }
        else{
```

```
                exit(1);
        }


        return stime;
}
void pcb(process_identification_number_pointer
processID,executioncontext_pointer exc,time_elapsed
time,processingtime_pointer ptime,waitingtime_pointer
stime,process_state_pointer state,parent_pointer
ptr1,number_of_observations n)
/*{for(int j=0;j<n;j++)
        {       printf("%f",ptime[j]);
}
for(int l=0;l<n;l++)
        {       printf("%f",stime[l]);
}*/
{       random_a r;
        printf("\t\t\t PROCESS CONTROL BLOCK\n");
        for(integer_variable s=1;s<n;s++)
        {
                printf("Add time to elapse\n");
                scanf("%d",&time);
                r=rand()%(n+1);
                for(int i=0;i<n;i++)
                {
                        (processID[i])=100+i;
                        exc[i]=((time+(stime[i]))/(ptime[i])*100);
                        ptr1[i]=(processID[i]);
                        if(r==i)
                        {
                                state[i]="running";
                        }
                        else if(exc[i]>100)
                        {       state[i]="exited";}
                else if((exc[i])>40&&(exc[i])<60)
```

```
            {
                  state[i]="Blocked";
            }
            else
            {
                  state[i]="ready";
            }
            }
            printf("PROCESS ID\tSTATE\tEXECUTIONCONTEXT\
tPARENTPOINTER\n");
                  for(int m=0;m<n;m++)
                  {

                        if(exc[m]<=0)
                        {
                              exc[m]=0;
                        }

                  printf("%d\t\t%s\t\t\%f\t\t%d\t\t\
n",processID[m],state[m],exc[m],ptr1[m]);
                  }}

                  //free(ptime);
                  //free(stime);                              //MEM00-C.
Allocate and free memory in the same module, at the same level of
abstraction

// free(ptime) is done twice at both the levels of abstraction.

                  }
```

Code for the non-compliant version :

main_for_non_compliant_code.c

```
#include "function_prototypes_for_non_compliant_code.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "string.h"


integer_variable main()
{
integer_type_variable x, size=0;

PASSWORD ptr;
ptr.front = 0;
ptr.rear = 0;


constant_buffer_structure_variable buf = {10,20};

flex_array_struct_variable struct_a;
flex_array_struct_variable struct_b;

flex_array_struct_pointer p1 = &(struct_a);
flex_array_struct_pointer p2 = &(struct_b);



struct_a.data[0] = 1;
struct_a.data[1] = 2;
struct_a.data[2] = 4;

number_of_observations n;
```

```
        waitingtime_pointer stime;
        process_identification_number processID[10];
        executioncontext exc[5];
        time_elapsed time=0;
        //float stime[5]={0,-2,-3,-4,-1};
        //float ptime[5]=(4,6,7,3,8};

        process_state_pointer state[10];
        parent_pointer_pointer ptr1[5];
        processingtime_pointer ptime;


Password_verification_system(&ptr);
string_array str[size];
string_pointer destination_array = (string_pointer )calloc(10,1); //calloc
avoids garbage values because the array is initialised with 0s
printf("Enter \n1 to find the time taken by a carry_look_ahead_adder     \n\n2
to ENCRYPT a numeric password       \n\n3 3 to open a file  \n\n4 to see
CPU scheduling\n\n5 to see ERROR DETECTION IN STRINGS \n\n6 to
see a general case of ERROR DETECTION\n\n7 to see process table     \n\n
");
scanf("%d",&x);

integer_type_variable clear_array[12];


switch(x)
{
case 1:

time_taken_carry_lookahead_adder();

break;

case 2:
```

```
NUMERIC_PASSWORD_ENCRYPTION();

break;

case 3:

open_file_table();

break;

case 4:

CPU_scheduling();

break;


case 5:

string_processing_error_detection();

break;

case 6:
Artificial_error_detection_system( buf, p1, p2);
break;

case 7:
printf("enter the observations\n");
                scanf("%d",&n);
                printf("enter the proceesor time of %d observations\n",n);
                ptime=pcb(n);
                printf("enter the waiting time of %d observations\n",n);
                stime=pcb2(n);

        pcb1(processID,exc,time,&ptime[0],&stime[0],&state[0],&ptr1[0],n);
```

```
		printf("-----------------------------------\n");
		printf("PROCESS ID\t\tPROCESSING TIME\n");
		for(int i=0;i<n;i++)
		{
			printf("%d\t\t%f\n",processID[i],ptime[i]);			//
MEM30-C. Do not access freed memory // freed in function
		}
		/*free(ptime);
				//MEM01-C. Store a new value in pointers
immediately after free() // assign it to null
		ptime=NULL;						//MEM31-C.
Free dynamically allocated memory when no longer needed
		free(stime);
		stime=NULL;
		free(ptr1);						//MEM34-C.
Only free memory allocated dynamically
*/
break;

default:
printf("Invalid case\n");



}
return 0;
}
```

abstract_data_types_for_non_compliant_code.h

// abstract_data_types_for_non_compliant_code.h is also included for the functions file of non-compliant code, we've already shown it in the above segment of the compliant version, we are not including it here to avoid repitition.

// 16) VIOLATING RECOMMENDATION: DCL05-C. Use typedefs of non-pointer types only.

//The following type definition improves readability at the expense of introducing a const-correctness issue. In this example, the const qualifier applies to the typedef instead of to the underlying object type. Consequently, func does not take a pointer to a const struct obj but instead takes a const pointer to a struct obj.

typedef const struct buffer {

integer_type_variable x;
integer_type_variable y;

} *BUFFER;

function_prototypes_for_non_compliant_code.h

```
#include "abstract_data_types_for_non_compliant_code.h"

extern void time_taken_carry_lookahead_adder();


extern numeric_encryption_procedure_result
multiple_of_sum(numeric_encryption_variable no1,
numeric_encryption_variable no2, numeric_encryption_variable multiple);


// 13) VIOLATING RECOMMENDATION: DCL07-C. Include the
appropriate type information in function declarators.

extern numeric_encryption_variable no1,no2,multiple;
numeric_encryption_procedure_result multiple_of_sum(no1, no2, multiple);


extern void NUMERIC_PASSWORD_ENCRYPTION();


extern scheduler Response_Ratio(scheduler_variable waiting_time,
scheduler_variable burst_time);


extern void open_file_table();


extern void CPU_scheduling();

extern void string_processing_error_detection();

extern void Password_verification_system( PASSWORD *ptr);

extern void
Artificial_error_detection_system(constant_buffer_structure_variable buf,
flex_array_struct_pointer p1, flex_array_struct_pointer p2);
```

extern processtime_pointer pcb(number_of_observations n);

extern waittime_pointer pcb2(number_of_observations n);

extern void pcb1(process_identification_number_pointer processID,executioncontext_pointer exc,time_elapsed time,processingtime_pointer ptime,waitingtime_pointer stime,process_state_pointer state, parent_pointer ptr,number_of_observations n);

string.h

void clear(int array[]);

functions_for_non_compliant_code_part1.c

/*
RECOMMENDATIONS VIOLATED:

1) VIOLATING RECOMMENDATION: PRE00-A. Prefer inline functions to macros.

2) VIOLATING RECOMMENDATION: PRE02-A. Macro expansion should always be parenthesized for function-like macros.

3) VIOLATING RECOMMENDATION: Pass the size of the array whenever the array is passed to a function.

4) VIOLATING RECOMMENDATION: PRE01-A. Use parentheses within macros around variable names.

5) VIOLATING RECOMMENDATION: PRE04-A. Do not reuse a standard header file name.

6) VIOLATING RECOMMENDATION: PRE30-C. Do not create a universal character name through concatenation.

7) VIOLATING RECOMMENDATION: PRE03-A. Avoid invoking a macro when trying to invoke a function.

8) VIOLATING RECOMMENDATION: PRE05-A. Avoid using repeated question marks.

9) VIOLATING RECOMMENDATION: DCL02-A. Use visually distinct identifiers.

10) VIOLATING RECOMMENDATION: DCL23-C. Guarantee that mutually visible identifiers are unique.

11) VIOLATING RECOMMENDATION: DCL03-A. Place const as the rightmost declaration specifier.

12) VIOLATING RECOMMENDATION: DCL04-C. Do not declare more than one variable per declaration.

13) VIOLATING RECOMMENDATION: DCL07-C. Include the appropriate type information in function declarators.
(THE VIOLATION OF THIS RECOMMENDATION CAN BE SEEN IN 'function_prototypes_for_non_compliant_code.h' file)

14) VIOLATING RECOMMENDATION: DCL00-C. Const-qualify immutable objects.

15) VIOLATING RECOMMENDATION: DCL01-C. Do not reuse variable names in subscopes.

16) VIOLATING RECOMMENDATION: DCL05-C. Use typedefs of non-pointer types only.

17) VIOLATING RECOMMENDATION: DCL06-C. Use meaningful symbolic constants to represent literal values.

18) VIOLATING RECOMMENDATION: DCL08-A. Declare function pointers using compatible types.

19) VIOLATING RECOMMENDATION: DCL09-A. Declare functions that return an errno with a return type of errno_t.

20) VIOLATING RECOMMENDATION: DCL10-C. Maintain the contract between the writer and caller of variadic functions.

21) VIOLATING RECOMMENDATION: DCL30-C. Declare objects with appropriate storage durations.

22) VIOLATING RECOMMENDATION: EXP03-C. Do not assume the size of a structure is the sum of the sizes of its members.

23) VIOLATING RECOMMENDATION: EXP43-C. Avoid undefined behavior when using restrict-qualified pointers.

24) VIOLATING RECOMMENDATION: STR05-C. Use pointers to const when referring to string literals.

25) VIOLATING RECOMMENDATION: STR06-C. Do not assume that strtok() leaves the parse string unchanged.

26) VIOLATING RULE : STR30-C. Do not attempt to modify string literals.

27) VIOLATING RECOMMENDATION: STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator.

28) VIOLATING RECOMMENDATION: STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string.

29) VIOLATING RECOMMENDATION: STR10-C. Do not concatenate different type of string literals.

30) VIOLATING RECOMMENDATION: MEM03-C. Clear sensitive information stored in reusable resources.

31) VIOLATING RECOMMENDATION: MEM06-A. Do not use user-defined functions as parameters to allocation routines.

33) (BEYOND_SYLLABUS) VIOLATING RECOMMENDATION: MEM36-C. Do not modify the alignment of objects by calling realloc().

34) VIOLATING RECOMMENDATION: MEM32-C Detect and handle critical memory allocation errors.

35) VIOLATING RECOMMENDATION: MEM33-C. Allocate and copy structures containing a flexible array member dynamically.

36) VIOLATING RECOMMENDATION: MEM35-C. Allocate sufficient memory for an object.

37) (BEYOND_SYLLABUS) VIOLATING RECOMMENDATION: INT30-C. Ensure that unsigned integer operations do not wrap.

38) VIOLATING RECOMMENDATION: ENV03-C. Sanitize the environment when invoking external programs.

39) VIOLATING RECOMMENDATION: STR00-A. Use TR 24731 for remediation of existing string manipulation code.

40) VIOLATING RECOMMENDATION: STR01-A. Use managed strings for development of new string manipulation code.

41) VIOLATING RECOMMENDATION: STR07-A. Take care when calling realloc() on a null terminated byte string.
*/

```
#include "function_prototypes_for_non_compliant_code.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


// have all structures, macros, inline functions, ... seperately for both compliant and non-compliant .c files




// 2) VIOLATING RECOMMENDATION: PRE02-A. Macro expansion should always be parenthesized for function-like macros


#define no_of_AND_OR(N,k) N/k-1

void time_taken_carry_lookahead_adder()
{

// 10) VIOLATING RECOMMENDATION: DCL23-C. Guarantee that mutually visible identifiers are unique.
```

```
//2 RANDOM LONG VARIABLES WHICH HAVE A LONG COMMON PART CANNOT BE IDENTIFIED EASILY IF THE DISTINCT VARIABLE IS NOT IN THE BEGINNING.

// 9) VIOLATING RECOMMENDATION: DCL02-A. Use visually distinct identifiers.

time_elapsed time_consumed_by_CLA, time_consumed_by_pg,
time_consumed_by_pg_block,time_consumed_by_AND_OR,
time_consumed_by_FA;

no_of_blocks_in_cla k;

size_of_cla N;

time_consumed_by_pg = 100; //100ps is the delay of each two-input gate.
time_consumed_by_pg_block = 600; // 6*100 ps, because there are 6 two-input gates in a block.
N = 32; // here we are talking about a 32-bit CLA
k = 4; // here the 32-bit CLA is made up of 4-bit blocks

time_consumed_by_AND_OR = 200 ; //2*100, because there are 2 2-input gates, one AND and one OR

time_consumed_by_FA = 300; //300ps is the full adder delay

time_consumed_by_CLA = time_consumed_by_pg +
time_consumed_by_pg_block +
no_of_AND_OR(N,k)*time_consumed_by_AND_OR +
k*time_consumed_by_FA ;//CALLING MACRO

printf("Time taken by the Carry look ahead adder is %d \
n",time_consumed_by_CLA);
}
```

// Here in the non-compliant version DATA SANITIZATION AND STRING TRUNCATION WILL NOT BE TAKEN CARE OF

// 5) VIOLATING RECOMMENDATION: PRE04-A. Do not reuse a standard header file name.

// The name is misleading and hence the recommendation asks us not to use the standard header file name to a local header file.

```
#include "string.h"
// 3) VIOLATING RECOMMENDATION: Pass the size of the array whenever the array is passed to a function. NOT FOLLOWED HERE.
void clear(integer_pointer array)
{
integer_type_variable i;
printf("The 4 elements in the array are: \n");
//Wrong size of the array will be generated.
for (i = 0; i < sizeof(array) / sizeof(array[0]); i++)
array[i] = 0;

}
```

// 4) VIOLATING RECOMMENDATION: PRE01-A. Use parentheses within macros around variable names.

```
#define mul(a,b) (a*b)

// 13) VIOLATING RECOMMENDATION: DCL07-C. Include the appropriate type information in function declarators.
numeric_encryption_variable no1,no2,multiple;
numeric_encryption_procedure_result multiple_of_sum(no1, no2, multiple)
{

// 11) VIOLATING RECOMMENDATION: DCL03-A. Place const as the rightmost declaration specifier.
//IF MANY VARIABLES ARE CONST QUALIFIED, THEIR DATATYPES WILL NOT BE VISUALLY EASY TO IDENTIFY.
constant_numeric_encryption_procedure_result result = mul(no1+no2, multiple);

//result = mul(no1+no2, multiple);

return result;

}

/*

THIS CANNOT BE COMPILED IN EITHER THE LATEST VERSION OF GCC OR VISUAL STUDIO.

// 6) VIOLATING RECOMMENDATION: PRE30-C. Do not create a universal character name through concatenation.
```

```
#define assign(uc1, uc2, val) uc1##uc2 = val

void test2() {
  int \u0401;

  assign(\u04, 01, 4);
printf("The ucn character \u0401 has the value %d\n",\u0401);
}


*/




// 1) VIOLATING RECOMMENDATION: PRE00-A. Prefer inline functions to macros
#define square(a) ( (a) * (a) )

// 7) VIOLATING RECOMMENDATION: PRE03-A. Avoid invoking a macro when trying to invoke a function.

// 8) VIOLATING RECOMMENDATION: PRE05-A. Avoid using repeated question marks.

#define puts(x) printf("I am in macro\n")



void NUMERIC_PASSWORD_ENCRYPTION()
{
```

```
// 17) VIOLATING RECOMMENDATION: DCL06-C. Use meaningful symbolic constants to represent literal values.
// We could have done enum { MAX_PASSWORD_SIZE=4 }
integer_array arr[4]; // array is not masked by an adt because the array is declared here.

//we will initialise this array now

clear(arr);// 3) VIOLATING RECOMMENDATION: Pass the size of the array whenever the array is passed to a function. NOT FOLLOWED HERE.



integer_type_variable i;

//7) VIOLATING RECOMMENDATION: PRE03-A. Avoid invoking a macro when trying to invoke a function.

puts("We've created an array of 4 characters\n");



printf("The array after initialization is: \n");

for(i=1;i<=4;i++)
printf("%d) \t %d\n",i,arr[i-1]);
printf("An ENCRYPTION SYSTEM asks the OS to do the following tasks:
        \n1) add 1 to every number \n2)find the square of the new number \n3) adding the no. with 19 and multiply it by 8\n");

printf("Please ENTER A 4 DIGIT PASSWORD (give spaces b/w the nos.):
");
for(i=0;i<4;i++)
scanf("%d",&arr[i]);
```

```
//18) VIOLATING RECOMMENDATION: DCL08-A. Declare function pointers using compatible types.

integer_type_variable (*multiple_of_sum_fn_ptr) (integer_type_variable, integer_type_variable) ; // POINTER NOT MADE AN ADT TO ILLUSTRATE THE RECOMMENDATION.

multiple_of_sum_fn_ptr = &multiple_of_sum;// NO. OF PARAMETERS ARE NOT SAME


for(i=0;i<4;i++)
{
printf("For %d : \n",arr[i]);

arr[i] = square(++arr[i]);

printf("After adding 1 and finding the square of it : %d\n",arr[i]);

arr[i] = multiple_of_sum_fn_ptr(arr[i],19); // (arr[i] + 19) *garbage_value instead of (arr[i] + 19) * 8

printf("After adding the no. with the next number in the password and multiply it by 8 : %d\n",arr[i]);
}

// PARANTHESES AVOIDS 7) VIOLATING RECOMMENDATION: PRE03-A. Avoid invoking a macro when trying to invoke a function.
(puts)("Finally the ENCRYPTED PASSWORD is, can you believe it ??! \n");
// ??! is THE TRIGRAPH SEQUENCE WHICH GIVES |


for(i=0;i<4;i++)
printf("%d",arr[i]);

printf("\n");
```

```
}


// 19) VIOLATING RECOMMENDATION: DCL09-A. Declare functions that return an errno with a return type of errno_t.


//THIS FUNCTION HAS TO BE USED LATER

void open_file_table()
{

file_pointer pf;

pf = fopen ("unexist.txt", "rb");

   if (pf == NULL) {
   printf("Could not open the file, this non-compliant version does not return an errno\n");
       } else {

      printf("OS maintains an Open File Table, this file's name unexist.txt will be added to the table\n");
   }

}
```

// 1) VIOLATING RECOMMENDATION: PRE02-A. Macro expansion should always be parenthesized for function-like macros

//calculating response ratio,//Response Ratio = (Waiting Time + Burst time) / Burst time
#define resp(wt,bt) wt+bt

scheduler Response_Ratio(scheduler_variable waiting_time, scheduler_variable burst_time)
{	scheduler Resp_Ratio;

	Resp_Ratio=resp(waiting_time,burst_time)/burst_time;
	return(Resp_Ratio);

}

void CPU_scheduling()
{

// 14) VIOLATING RECOMMENDATION: DCL00-C. Const-qualify immutable objects.

counter no_of_processes = 5;// no_of_processes can be modified again, its value is not protected from accidental modification, using a const would be the solution.

```
integer_type_variable j;

integer_array waiting_time[] = { 0, 2, 4, 6, 8 };
integer_array burst_time[] = { 3, 6, 4, 5, 2 }; //Burst time is the amount of time required by a process for executing on CPU.

printf("The details of the processes waiting are: \n\n");
printf("Waiting time \tBurst time\n");

// USE OF ENUM FOR 'FOR LOOP'

for(j=process1;j<no_of_processes;j++)
printf("%d\t\t%d\n",waiting_time[j], burst_time[j] );


integer_type_variable
i,max_response_ratio=0,index_of_max_response_ratio=0;

max_response_ratio = Response_Ratio(waiting_time[0],burst_time[0]);



for(i= process1;i<no_of_processes;i++)
{

// 18) VIOLATING RECOMMENDATION:  DCL01-C. Do not reuse variable names in subscopes.
integer_type_variable j;

j = Response_Ratio(waiting_time[i],burst_time[i]);

if(j > max_response_ratio)
{
max_response_ratio = j;
index_of_max_response_ratio = i;
```

```
}
}

printf("The process that will be given CPU time is the one with: \n waiting
time = %d \n burst time = %d \n It's response ratio = %d\n",
waiting_time[index_of_max_response_ratio],
burst_time[index_of_max_response_ratio], max_response_ratio);

}




// STRING PASSWORD SECTION




//THE STRUCTURE PASSWORD IS IN ADT'S .h FILE
void Password_verification_system( PASSWORD *ptr) // NO ADT FOR *
BECAUSE THAT'S THE RECOMMENDATION
{
integer_type_variable size;
printf("Enter the size of the password, LESS THAN 9 CHARACTERS\n");

scanf("%d",&size);

// 27) VIOLATING RECOMMENDATION: STR31-C. Guarantee that
storage for strings has sufficient space for character data and the null
terminator.

// Off-by-One Error
```

```
string_array str[size]; // no space for '\0'

printf("Enter the password, DO NOT USE any of '^','&','*','$','#',... \n");

scanf("%s",str);

// 21) VIOLATING RECOMMENDATION: DCL30-C. Declare objects with appropriate storage durations.
char_variable front1 = 0; // Now we have front1 declared here and not in structure, MORE THAN ONE PASSWORD WILL NEVER STAY IN THE QUEUE.
strcpy(ptr->queue[front1++], str);


// 28) VIOLATING RECOMMENDATION: STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string.
printf("Your password: %s is added to the queue\n",str); //not null terminated


}




//-----------------------------------------------


void string_processing_error_detection()
{

printf("\nOS DETECTS ERRORS BEFOREHAND TO ENSURE A SMOOTH PROCESSING OF ITS INSTRUCTIONS\n\n");
```

```
// 24) VIOLATING RECOMMENDATION: STR05-C. Use pointers to const when referring to string literals.
// IF THERE'S NO CONST MODIFICATION WOULD RESULT IN AN UNDEFINED BEHAVIOUR.

printf("TASK 1: To modify a constant string:\n\n");
string_pointer c = "Hello";
printf("A constant string: %s\n\n",c);
printf("NO MODIFICATION CAN BE MADE TO A CONSTANT STRING\
n\n\n");



// 26) VIOLATING RULE: STR30-C. Do not attempt to modify string literals.


//In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior:



printf("TASK 2: A CHANGE FOR A DIFFERENCE:\n\n");
string_pointer str1 = "Take Artificial Intelligence";
printf("An array type string: %s\n\n",str1);
printf("CANNOT be modified as it is pointed by a pointer and would cause UNDEFINED BEHAVIOUR\n\n");
```

```
// 31) VIOLATING RECOMMENDATION: MEM03-C. Clear sensitive
information stored in reusable resources.
string_array stri[20];
printf("TASK 3: To copy a password: \n\n");
strcpy(stri,"PASSWORD: UXf64hTrfd56");
printf("A string has: %s\n\n",stri);

printf("Copied string : %s\n\n",stri);




printf("TASK 4: To split first name and last name\n");

// 29) VIOLATING RECOMMENDATION: STR10-C. Do not concatenate
different type of string literals.

string_pointer msg = L"First-name "
          "Last_name";
printf("Input string: %s\n\n",msg);


printf("After processing:\n\n");

string_pointer token;

// 25) VIOLATING RECOMMENDATION: STR06-C. Do not assume that
strtok() leaves the parse string unchanged.



token = strtok(msg, " ");
puts(token);
```

```
while (token = strtok(0, " ")) {
  puts(token);
}
printf("\n\n");


printf("TASK 5: To make space for the null character after reallocation\n\n");

string_pointer s1 = (string_pointer )malloc(sizeof(char_variable)*4);
string_pointer ptr_new;
strcpy(s1,"PES");
printf("Input string : %s\n\n",s1);
// 41) VIOLATING RECOMMENDATION: STR07-A. Take care when
calling realloc() on a null terminated byte string.

ptr_new = (string_pointer )realloc(s1, sizeof(char_variable)*5);

// ptr_new[sizeof(char)*4] = '\0'; NOT GIVEN IN NON-COMPLIANT
VERSION.

printf("Null character added to %s\n\n",ptr_new);



// 39) VIOLATING RECOMMENDATION: STR00-A. Use TR 24731 for
remediation of existing string manipulation code.

// 40) VIOLATING RECOMMENDATION: STR01-A. Use managed strings
for development of new string manipulation code.

strcpy(stri," A I ");
printf("TASK 6: TO SAFELY COPY A STRING \n\n");

string_array str2[30];
printf("Input string: %s\n\n",stri);
```

```
strcpy(str2,stri);

printf("After copying: %s\n\n",str2);

}
```

// 20) VIOLATING RECOMMENDATION: DCL10-C. Maintain the contract between the writer and caller of variadic functions.

```
#include <stdarg.h>


enum { va_eol = -1 };
 non_negative_integer average(integer_variable first, ...);

non_negative_integer average(integer_variable first, ...)
{
  non_negative_integer count = 0;
non_negative_integer sum = 0;
  integer_type_variable i = first;
  va_list args;

  va_start(args, first);

  while (i != va_eol) {
    sum += i;
    count++;
    i = va_arg(args, int);
  }
```

```
  va_end(args);
  return(count ? (sum / count) : 0);
}
```

```
void Artificial_error_detection_system(constant_buffer_structure_variable
buf, flex_array_struct_pointer p1, flex_array_struct_pointer p2)
{
// Can modify o's contents
```

```
// PART 5: make a copy of the pointer given to the function

// 22) VIOLATING RECOMMENDATION: EXP03-C. Do not assume the
size of a structure is the sum of the sizes of its members.

buffer_structure_pointer buf_cpy = (buffer_structure_pointer)malloc(
    sizeof(integer_type_variable)*2);


  if (buf_cpy == NULL) {
    // Handle malloc() error
printf("FAILED to create an array\n");
  }
```

```
  memcpy(buf_cpy, &(buf), sizeof(struct buffer));
```

// 33) (BEYOND_SYLLABUS) VIOLATING RECOMMENDATION: MEM36-C. Do not modify the alignment of objects by calling realloc().

// 34) VIOLATING RECOMMENDATION: MEM32-C Detect and handle critical memory allocation errors.

// This noncompliant code example returns a pointer to allocated memory that has been aligned to a 4096-byte boundary. If the resize argument to the realloc() function is larger than the object referenced by ptr, then realloc() will allocate new memory that is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement but may not preserve the stricter alignment of the original object.

```
/* FOR UNDERSTANDING OUTPUT
memory aligned to 4096 bytes
ptr = 0x1621b000

After realloc():
ptr1 = 0x1621a010

ptr1 is no longer aligned to 4096 bytes.*/
```

```
// now let's create 2 pointers: ptr and ptr1

size_t resize = 1024;
size_t alignment = 1 << 12;
```

```
  integer_pointer restrict ptr;
  integer_pointer restrict ptr1;

  if (NULL == (ptr = (integer_pointer)aligned_alloc(alignment,
                                    sizeof(integer_type_variable)))) {
    // Handle error
printf("FAILED to create an array\n");
  }

  if (NULL == (ptr1 = (integer_pointer)aligned_alloc(alignment,
                                     resize))) {
    // Handle error
printf("FAILED to create an array\n");
  }



// 23) VIOLATING RECOMMENDATION: EXP43-C. Avoid undefined
behavior when using restrict-qualified pointers.


// WE CREATED A COPY BECAUSE WE CAN'T ASSIGN A CONST
POINTER.
ptr = &(buf_cpy->x);
ptr1 = &(buf_cpy->y);



printf("TASK 1: To find the average of the values in 2 structures and store
them in an array: \n\n");

printf("EXPECTED ERRORS: \n\n1) CONSTANT STRUCTURE CANNOT
BE MANIPULATED \n\n2) STACK OVERFLOW DUE TO GARBAGE
VALUES IN FLEXIBLE ARRAYS IN STRUCTURES\n\n");
```

printf("ERROR DETECTION SYSTEM HAS RESOLVED POSSIBLE ERRORS\n\n");

integer_type_variable t1,t2;

// 20) VIOLATING RECOMMENDATION: DCL10-C. Maintain the contract between the writer and caller of variadic functions.

// 31) VIOLATING RECOMMENDATION: MEM06-A. Do not use user-defined functions as parameters to allocation routines.

printf("The average of the 1st structure is %d \n",average(*ptr, *ptr1));// NO -1 IN THE END

// 35) VIOLATING RECOMMENDATION: MEM33-C. Allocate and copy structures containing a flexible array member dynamically.

*p2 = *p1;
// array is not copied because it is a flexible member of the structure

// 20) VIOLATING RECOMMENDATION: DCL10-C. Maintain the contract between the writer and caller of variadic functions.

t2 = average(p2->data[0], p2->data[1], p2->data[2]); // NO -1 IN THE END

printf("The average of the 2nd structure is %d \n",t2);

```
// to create a long array
integer_type_variable len = 2;



// 36) VIOLATING RECOMMENDATION: MEM35-C. Allocate sufficient memory for an object.

// 37) (BEYOND_SYLLABUS) VIOLATING RECOMMENDATION: INT30-C. Ensure that unsigned integer operations do not wrap.


long_type_pointer p;
long_type_variable SIZE_MAX = 2147483647;
  if (len == 0 || len > SIZE_MAX / sizeof(long_type_variable)) {
    // Handle overflow
  }
  p = (long_type_pointer)malloc(len * sizeof(integer_type_variable)); // int is used to create an array of long, chance of overflow.
  if (p == NULL) {
    // Handle error
    printf("FAILED to create an array\n");
  }
  //free(p);

p[0] = average(*ptr, *ptr1, va_eol);
p[1] = average(p2->data[0], p2->data[1], p2->data[2], va_eol);
```

```
printf("\nThe elements have been added to an array\n\n");


// 38) VIOLATING RECOMMENDATION: ENV03-C. Sanitize the
environment when invoking external programs.
/*
if (clearenv() != 0) {
  printf("The environment is not safe\n");
}*/ //NOT SANITIZING THE ENVIRONMENT

if(getenv("PATH")!=NULL)
{
// HANDLE ERROR
printf("Please correct your path\n");
}

  free(ptr);
}
```

functions_for_non_compliant_code_part2.c

/*
1) MEM10-C. Define and use a pointer validation function

2) MEM02-A. Do not cast the return value from malloc()

3) MEM05-C. Avoid large stack allocations

4) MEM04-C. Beware of zero-length allocations

5) MEM00-C. Allocate and free memory in the same module, at the same level of abstraction.

6) MEM30-C. Do not access freed memory

7) MEM01-C. Store a new value in pointers immediately after free()

8) MEM31-C. Free dynamically allocated memory when no longer needed

9) MEM34-C. Only free memory allocated dynamically

10) MEM07-C. Ensure that the arguments to calloc(), when multiplied, do not wrap
*/

```
#include "abstract_data_types_for_non_compliant_code.h"
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
//static void pcb1(int processID[],float executioncontext[],int time,float *ptime,float stime[],int r,char *state,int *ptr[]);

// process control blog - pcb
```

```
processtime_pointer pcb(number_of_observations n)
{


        processingtime_pointer ptime;
        /* if (n== 0) {
    exit(1);
  }*/



        ptime=(float_pointer)malloc(n*sizeof(float_variable));
                                //"MEM02-A. Do not cast the return value from
malloc()
        //              avoid using --->float ptime[n];
             //"MEM05-C. Avoid large stack allocations

                                //MEM04-C. Beware of zero-length allocations
        if (!ptime) {
    exit(1);
  }
        for(integer_variable k=0;k<n;k++)
        {
                scanf("%f",&ptime[k]);
        }
        return ptime;

}
waittime_pointer pcb2(number_of_observations n)
{
        waitingtime_pointer stime;
        /*size_of_n size=2147483647;
         if(n>(size/sizeof(float)))                                    //MEM07-C.
Ensure that the arguments to calloc(), when multiplied, do not wrap
         {
                 exit(1);
```

```
        }*/

        //if (n >(sizeof(stime)/sizeof(float)))
        //      {exit(1);
  /* Handle error condition */;
     //  }
        stime=calloc(n,sizeof(float_variable));
        for(integer_variable k=0;k<n;k++)
        {
                scanf("%f",&stime[k]); // TIP : GIVE NEGATIVE INPUTS
        }
        return stime;
}
void pcb1(process_identification_number_pointer
processID,executioncontext_pointer exc,time_elapsed
time,processingtime_pointer ptime,waitingtime_pointer
stime,process_state_pointer state,parent_pointer  ptr,number_of_observations
n)

{
random_a r;
        printf("\t\t\t PROCESS CONTROL BLOCK\n");
        for(integer_variable s=1;s<n;s++)
        {
                printf("Add time to elapse\n");
                scanf("%d",&time);
                r=3;
                for(integer_variable i=0;i<n;i++)
                {
                        (processID[i])=100+i;
                        exc[i]=time+stime[i]/ptime[i]*100;      // main formula
                        ptr[i]=(&processID[i]);
                        if(r==i)
                        {
                                state[i]="running";
                        }
```

```
                else if(exc[i]>100)     // execution context > 100 =>
completed
                {       state[i]="exited";}
            else if((exc[i])>40&&(exc[i])<60)
            {
                state[i]="Blocked";
            }
            else
            {
                state[i]="ready";
            }
            }
            printf("PROCESS ID\tSTATE\tEXECUTIONCONTEXT\
tPARENTPOINTER\n");
                for(integer_variable m=0;m<n;m++)
                {

                        if(exc[m]<=0)
                        {
                            exc[m]=0;
                        }

                printf("%d\t\t%s\t\t\%f\t\t%d\t\t\
n",processID[m],state[m],exc[m],&ptr[m]);
                }
        }
                free(ptime);
                free(stime);          //MEM00-C. Allocate and free memory in
the same module, at the same level of abstraction
                }
```

Testing And Verification

The following menu is common for all the test cases:

Enter
1 to find the time taken by a carry_look_ahead_adder

2 to ENCRYPT a numeric password

3 to open a file

4 to see CPU scheduling

5 to see ERROR DETECTION IN STRINGS

6 to see a general case of ERROR DETECTION

7 to see process table

Test case 1:

Compliant case:

1
Time taken by the Carry look ahead adder is 3300

Non-compliant case:
1
Time taken by the Carry look ahead adder is 1708

EXPLANATION:

Compliant and non-compliant versions give different answers because the macro expansion is not parenthesized in the non-compliant version.

Test case 2:

Compliant case:

2
We've created an array of 4 characters
The array after initialization is:
1) 0
2) 0
3) 0
4) 0
An ENCRYPTION SYSTEM asks the OS to do the following tasks:
1) add 1 to every number
2)find the square of the new number
3) adding the no. with 19 and multiply it by 8
Please ENTER A 4 DIGIT PASSWORD (give spaces b/w the nos.): 1 2 3 4
For 1 :
After adding 1 and finding the square of it : 4
After adding the no. with the next number in the password and multiply it by 8 : 184
For 2 :
After adding 1 and finding the square of it : 9
After adding the no. with the next number in the password and multiply it by 8 : 224
For 3 :
After adding 1 and finding the square of it : 16
After adding the no. with the next number in the password and multiply it by 8 : 280
For 4 :
After adding 1 and finding the square of it : 25
After adding the no. with the next number in the password and multiply it by 8 : 352
Finally the ENCRYPTED PASSWORD is, can you believe it ??!

184224280352

Non-compliant case:

2
The 4 elements in the array are:
I am in macro
The array after initialization is:
1) 0
2) 0
3) -349771504
4) 32764
An ENCRYPTION SYSTEM asks the OS to do the following tasks:
1) add 1 to every number
2)find the square of the new number
3) adding the no. with 19 and multiply it by 8
Please ENTER A 4 DIGIT PASSWORD (give spaces b/w the nos.): 1 2 3 4
For 1 :
After adding 1 and finding the square of it : 6
After adding the no. with the next number in the password and multiply it by 8 : 120
For 2 :
After adding 1 and finding the square of it : 12
After adding the no. with the next number in the password and multiply it by 8 : 240
For 3 :
After adding 1 and finding the square of it : 20
After adding the no. with the next number in the password and multiply it by 8 : 400
For 4 :
After adding 1 and finding the square of it : 30
After adding the no. with the next number in the password and multiply it by 8 : 600
Finally the ENCRYPTED PASSWORD is, can you believe it ??!

120240400600

EXPLANATION:

Mainly because of the violation of the recommendation:
DCL08-A. Declare function pointers using compatible types.

One of the parameters of the function take a garbage value.
The encryption function is cleverly designed to demonstrate the violation of 12 other recommendations.

Test case 3:

Compliant case:

3
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory

Non-compliant case :

3
Could not open the file, this non-compliant version does not return an errno .

EXPLANATION:

This is a classic case of the violation of the recommendation:

DCL09-A. Declare functions that return an errno with a return type of errno_t.

Test case 4 :

Compliant case:

4
The details of the processes waiting are:

Waiting time	Burst time
0	3
2	6
4	4
6	5
8	2

The process that will be given CPU time is the one with:
waiting time = 8
burst time = 2
It's response ratio = 5

Non-compliant case :

4
The details of the processes waiting are:

Waiting time	Burst time
0	3
2	6
4	4
6	5
8	2

The process that will be given CPU time is the one with:
waiting time = 8
burst time = 2
It's response ratio = 9

EXPLANATION:

Response ratio is different because of the violation of the recommendation:

DCL01-C. Do not reuse variable names in subscopes.

Test case 5 :

Compliant case:

5

OS DETECTS ERRORS BEFOREHAND TO ENSURE A SMOOTH PROCESSING OF ITS INSTRUCTIONS

TASK 1: To modify a constant string:

A constant string: Hello

NO MODIFICATION CAN BE MADE TO A CONSTANT STRING

TASK 2: A CHANGE FOR A DIFFERENCE:

An array type string: Take Artificial Intelligence

Can be modified as it is not pointed by a pointer

Modified string: Make Artificial Intelligence

TASK 3: To copy a password:

A string has: PASSWORD: UXf64hTrfd56

The value in the string after use is: NA

DELETED IMMEDIATELY AFTER USE, COULD NOT BE COPIED

TASK 4: To split first name and last name
Input string: First-name Last_name

After processing:

First-name
Last_name

TASK 5: To make space for the null character after reallocation

Input string : PES

Null character added to PES

TASK 6: TO SAFELY COPY A STRING

Input string = A I

After copying: A I

Non-compliant case :

5

OS DETECTS ERRORS BEFOREHAND TO ENSURE A SMOOTH PROCESSING OF ITS INSTRUCTIONS

TASK 1: To modify a constant string:

A constant string: Hello

NO MODIFICATION CAN BE MADE TO A CONSTANT STRING

TASK 2: A CHANGE FOR A DIFFERENCE:

An array type string: Take Artificial Intelligence

CANNOT be modified as it is pointed by a pointer and would cause UNDEFINED BEHAVIOUR

TASK 3: To copy a password:

A string has: PASSWORD: UXf64hTrfd56

Copied string : PASSWORD: UXf64hTrfd56

TASK 4: To split first name and last name
Input string: F

After processing:

I am in macro

TASK 5: To make space for the null character after reallocation

Input string : PES

Null character added to PES

TASK 6: TO SAFELY COPY A STRING

Input string: A I

After copying: A I

EXPLANATION :

The output by itself explains the recommendations used.

Test case 6:

Compliant case:

6
TASK 1: To find the average of the values in 2 structures and store them in an array:

EXPECTED ERRORS:

1) CONSTANT STRUCTURE CANNOT BE MANIPULATED

2) STACK OVERFLOW DUE TO GARBAGE VALUES IN FLEXIBLE ARRAYS IN STRUCTURES

ERROR DETECTION SYSTEM HAS RESOLVED POSSIBLE ERRORS

The average of the 1st structure is 15
The average of the 2nd structure is 2

The elements have been added to an array.

Non-compliant case :

6
TASK 1: To find the average of the values in 2 structures and store them in an array:

EXPECTED ERRORS:

1) CONSTANT STRUCTURE CANNOT BE MANIPULATED

2) STACK OVERFLOW DUE TO GARBAGE VALUES IN FLEXIBLE ARRAYS IN STRUCTURES

ERROR DETECTION SYSTEM HAS RESOLVED POSSIBLE ERRORS

The average of the 1st structure is 73899345
The average of the 2nd structure is 58266631

The elements have been added to an array

Please correct your path

EXPLANATION :

The average is different in both the versions because we are not passing -1 at the end to the variadic function, the function continues taking garbage values till it encounters a -1, similarly there are 10 other recommendations used in this case and here we have shown both the compliant and the non-compliant case.

Test case 7:

Compliant case:

```
7
enter the number of processes
3
enter the proceesor time of 3 observations
4
5
6
enter the waiting time of 3 observations
1
2
3
                    PROCESS CONTROL BLOCK
Add time to elapse
4
PROCESS ID      STATE   EXECUTIONCONTEXT        PARENTPOINTER
100             exited          125.000000              -1945887296
101             exited          120.000008              -1945887288
102             exited          116.666664              -1945887280
Add time to elapse
1
PROCESS ID      STATE   EXECUTIONCONTEXT        PARENTPOINTER
100             Blocked          50.000000               -1945887296
101             ready           60.000004              -1945887288
102             running          66.666672               -1945887280
-----------------------------------------
PROCESSID       PROCESSINGTIME
100             4.000000
101             5.000000
102             6.000000
```

Non-compliant case :

```
 7
enter the observations
3
enter the proceesor time of 3 observations
4
5
6
enter the waiting time of 3 observations
1
2
3
     PROCESS CONTROL BLOCK
Add time to elapse
4
PROCESS ID      STATE   EXECUTIONCONTEXT
PARENTPOINTER
100             ready         29.000000             1484194048
101             Blocked         44.000000             1484194056
102             Blocked         54.000000             1484194064
Add time to elapse
1
PROCESS ID      STATE   EXECUTIONCONTEXT
PARENTPOINTER
100             ready         26.000000             1484194048
101             Blocked         41.000000             1484194056
102             Blocked         51.000000             1484194064
-----------------------------------
PROCESS ID              PROCESSING TIME
100             0.000000
101             0.000000
102             6.000000
```

EXPLANATION :

This case includes 10 memory recommendations, the non-compliant version gives wrong results because of the violation of the rules and recommendations., explaining 10 recommendations here is not viable, please refer the code for detailed usage. The above table is a process table that lets you keep a track of the time consumed by various processes.

Maintenance:

Modfication:

The design philosophy used is that no two cases of the application are dependent on each other, this independency makes it easy for the modification of the code in the future. We have made options for reusability like in the case of ADTs and password processing procedures, but this shouldn’t stop us from wanting to modify our code, the connection of any procedure with these common platforms is highly transparent and it shouldn’t be a matter of concern for developers in modifying the code.

We have made use of defined constants declared in the beginning of the code, so if you have to change the value of a constant you just have to change the value of the constant in the beginning and this is so much more easier when compared to changing the value at every occurrence of the constant value.

We have enumerated data types defined in a different file, the file which houses all the abstract data types, and you can easily manipulate the variables as per your requirements without even having to open the functions file, this is a classic case of abstraction as well as easy accessibility for modification.

Creating Multiple Versions:

The architecture of the code favours creation of multiple versions, in fact we've already done this in a vague way:
We use two files for functions and a single driver file, similarly you can give different views for the user using the same content, every user will then get a different

view, this is possible because we've stitched levels of abstraction at every layer of development.

The architecture supports modular programming, the problem can be divided into smaller parts and various developers can develop their version, we will then have multiple versions and all these versions can be integrated in the end to get the final software.

If there are multiple uses of recommendations, multiple versions can be used to show the various recommendations.

If a developer wants to create a new version then he can look at the explanation we've given with every recommendation used and the developer can make use of the recommendations with a lot more ease because we've given him real world examples.

Scalability

Scalability is the property of a system to handle a growing amount of work by adding resources to the system. Because of the architecture we've developed, the levels of abstraction we've given scalability is easy.

Our concept is an using CERT recommendations in Operating system , the number of functions that can be added to the code tends to infinity!
We can add more functionalities to the code and we can get security solutions to more parts of an operating system.

If we were addressing a single problem statement like, 'Write a program to add 100 numbers' the program would not be scalable, but because an operating system can be made to perform any viable task we have a scalable code.

Whenever we've used arrays or any data structures we allocate memory for the data structure based on the user input and hence we are scalable even when a user provides a large input.

Conclusion

Why did we use CERT recommendations in our project? To prevent hackers from hacking our software. How will they hack it? They would give vulnerable inputs. How did we solve the problem? We used data sanitization at every occurrence of an user input. How did we get to know the solution? This is where CERT recommendations come in, they have figured out such bugs and have given solutions to them, we just had to adapt their solutions to our code.

There are errors other than syntactical, memory allocation, run-time and logical errors. These errors are not because of a mistake by us, but we are still open for vulnerabilities. Despite making a correct code logically and syntactically we have to protect ourselves from vulnerabilities, this is the purpose of implementing CERT recommendations in our project.

In our introduction and abstract we highlighted that we'd develop encryption systems, scheduling systems and error detection systems. The hidden correlation between all these is securing ourselves from correlation,

we told that we’d implement these concepts because that is what CERT made their recommendations for.

Blog

We’ve made a blog that talks about the need of safe programming for our application along with our experience with this course, do look at it !

https://anirudhrajalbandi1.wixsite.com/mysite/blog-1

References

1) Secure Coding in C and C++, Second Edition, Robert C. Seacord

2) SEI CERT, C Coding Standard, Rules for Developing Safe, Reliable, and Secure Systems, 2016 Edition, Carnegie Mellon University

3) CERT C Programming Language Secure Coding Standard, Document No. N1255

4) SEI CERT C Coding Standard, wiki.sei.cmu.edu, Official website of Carnegie Mellon University

5) en.wikipedia.org/wiki/Operating_system

6) www.tutorialspoint.com/operating_system/os_overview.htm

7) www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/