Fortify Source Code Analyzer Software

Engineer

Candidate Homework

# Project 1

You will define, implement and test a Sudoku solver.

**Sudoku rules:**

The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contains all of the digits from 1 to 9.

**Sudoku examples:**

|  |  |
| --- | --- |
| Easy | Difficult |
| |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | 1 | 3 | 8 |  |  | 4 |  | 5 | |  | 2 | 4 | 6 |  | 5 |  |  |  | |  | 8 | 7 |  |  |  | 9 | 3 |  | | 4 | 9 |  | 3 |  | 6 |  |  |  | |  |  | 1 |  |  |  | 5 |  |  | |  |  |  | 7 |  | 1 |  | 9 | 3 | |  | 6 | 9 |  |  |  | 7 | 4 |  | |  |  |  | 2 |  | 7 | 6 | 8 |  | | 1 |  | 2 |  |  | 8 | 3 | 5 |  | | |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  |  | 2 |  |  |  |  | 4 | 1 | |  |  |  |  | 8 | 2 |  | 7 |  | |  |  |  |  | 4 |  |  |  | 9 | | 2 |  |  |  | 7 | 9 | 3 |  |  | |  | 1 |  |  |  |  |  | 8 |  | |  |  | 6 | 8 | 1 |  |  |  | 4 | | 1 |  |  |  | 9 |  |  |  |  | |  | 6 |  | 4 | 3 |  |  |  |  | | 8 | 5 |  |  |  |  | 4 |  |  | |

**Requirements**

* Write an architecture document explaining the choices you have made to implement this solver.
* Implement the solver in Java.
  + Document the issues you have encountered and how you resolved them
  + Include the source code and other necessary files, setup instructions in your response.
* Check your solver with at least the 2 examples provided above

Project 1 Solution

The provided code is a solution to the Sudoku puzzle problem. It uses a backtracking algorithm to solve the puzzle by trying different numbers in each empty cell until a valid solution is found.

Components:-

* Solution Class: This class contains the main logic for solving the Sudoku puzzle. It has two methods: solveSudoku and solve.
* solveSudoku: This method is the entry point for solving the Sudoku puzzle. It calls the solve method with the initial row and column indices.
* solve: This method is a recursive function that tries to fill in the Sudoku puzzle. It takes a 2D array board and two integer indices row and col as input.
* Main Class: This class contains the main method that creates an instance of the Solution class and calls the solveSudoku method with a sample Sudoku puzzle.
* checkIfValid Method: This method checks if a given number is valid in a given cell

The algorithm used is a backtracking algorithm.

The provided code is a solution to the Sudoku puzzle problem using a backtracking algorithm. It uses a recursive function to try different numbers in each empty cell until a valid solution is found. The checkIfValid method is used to validate each number before trying it in the puzzle.

Issues faced:-

* In checkIfValid, Initially It was a bit difficlut to figure out how check box of 3x3, Later I figured that since box is of 3x3 hence division by 3 will give which box we have to look for.
* Since I did not have much practice with java recently, took a bit to figure out syntax for java code.

Source code:-

class Solution {

public void solveSudoku(char[][] board) {

solve(board, 0, 0);

}

public boolean solve(char[][] board, int row, int col) {

// If the col is 9, that means we have filled out a whole row. Start the search on the next row by resetting column and incrementing the row by 1

if (col == board[0].length) {

col = 0;

row += 1;

}

// If you've reached 9, that means you didn't run into any errors with your blocks in the previous rows

if (row == board.length) {

return true;

}

// If this piece already has a value, check the next square

if (board[row][col] != '.') return solve(board, row, col+1);

// Trying every number from 1 to 9 for the block

for (char num = '1'; num <= '9'; num++) {

if (checkIfValid(board, row, col, num))

{

board[row][col] = num;

boolean solved = solve(board, row, col+1);

if (solved)

{

return true;

}

// If our board isn't solved, We backtrack to try the next number

else

{

board[row][col] = '.';

}

}

}

return false;

}

public boolean checkIfValid(char[][] board, int row, int col, char value) {

for (int i = 0; i < board.length; i++) {

// Check the column for duplicates

if (board[i][col] == value) return false;

// Check the row for duplicates

if (board[row][i] == value) return false;

}

int rowBox = row / 3;

int colBox = col / 3;

for (int i = rowBox \* 3; i < (rowBox + 1) \* 3; i++) {

for (int j = colBox \* 3; j < (colBox + 1) \* 3; j++) {

// Check the box for duplicates

if (board[i][j] == value) return false;

}

}

return true;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

char[][] board = {

{'.', '.', '2', '.', '.', '.', '.', '4', '1'},

{'.', '.', '.', '.', '8', '2', '.', '7', '.'},

{'.', '.', '.', '.', '4', '.', '.', '.', '9'},

{'2', '.', '.', '.', '7', '9', '3', '.', '.'},

{'.', '1', '.', '.', '.', '.', '.', '8', '.'},

{'.', '.', '6', '8', '1', '.', '.', '.', '4'},

{'1', '.', '.', '.', '9', '.', '.', '.', '.'},

{'.', '6', '.', '4', '3', '.', '.', '.', '.'},

{'8', '5', '.', '.', '.', '.', '4', '.', '.'}

};

solution.solveSudoku(board);

// Printing the solved board

for (char[] row : board) {

for (char c : row) {

System.out.print(c + " ");

}

System.out.println();

}

}

}

# Project 2

The following code in C language contains some vulnerabilities. Analyze the code, report in the code as comments the vulnerabilities you have found and explain why these are vulnerabilities.

#include <stdio.h>

#include <stdlib.h>

#include <wchar.h>

#define PASSWORD "ABCD1234!"

/\*You need not worry about other include statements if at all any are missing \*/

void func1()

{

char \* data;

char \* dataBuffer = (char \*)ALLOCA(100\*sizeof(char)); //ALLOCA is a non standard extension

memset(dataBuffer, 'A', 100-1);

dataBuffer[100-1] = '\0';

data = dataBuffer - 8; //data points to memory that is 8 bytes before data buffer.

{

char source[100];

memset(source, 'C', 100-1);

source[100-1] = '\0';

strcpy(data, source); //while copying source to data it will go out of memory for data, as memory allocated for data does not start at a valid string

//also in above line we are are not checking if size of data is greater than or equal to source. If source is a greater than 100 bytes it will create an issue.

if(data != NULL)

{

printf("%s\n", data);

}

}

}

void func2()

{

char \* data; //pointer is not initialized to NULL at the time of declaration. Causing it to be an unsafe pointer and can lead to undefined behaviour.

data = NULL;

data = (char \*)calloc(100, sizeof(char)); //We should check the return type of calloc against null to be sure that memory is allocated.

strcpy(data, "A String");//Here again we are using strcpy without checking if data memory is large enough to store the source string.

if(data != NULL)

{

printf("%s\n", data);

}

}

void func3()

{

char \* password;

char passwordBuffer[100] = "";//Non secure memory where attackers can attack. We should use encryption techniques in case of password.

password = passwordBuffer; //password is an char pointer which is not secure and any other process could handle it

strcpy(password, PASSWORD); //Similar issue while using strcpy we are not comparing the size of passwordBuffer and PASSWORD

{

HANDLE pHandle;

char \* username = "User";

char \* domain = "Domain";

/\* Let's say LogonUserA is a custon authentication function\*/

if (LogonUserA(

username,

domain,

password, /\* we should pass passwordBuffer directly to make it more secure, as it is difficult for attacker to find memory location of password.

While in case of password they can easily find location password is pointing to.\*/

&pHandle) != 0)

{

printf("User logged in successfully.\n");

CloseHandle(pHandle);

}

else

{

printf("Unable to login.\n");

}

}

}

static void func4()

{

char \* data; //pointer is pointing to an unsafe memory and can result in undefined behaviour.

data = NULL;

data = (char \*)calloc(20, sizeof(char));

if (data != NULL)

{

strcpy(data, "Initialize");//Here again the issue of overflow can happend because memory size of data is not checked.

if(data != NULL)

{

printf("%s\n", data);

}

free(data);

}

}

void func5()

{

int i = 0;

do

{

printf("%d\n", i);

i = (i + 1) % 256;

} while(i >= 0); //loop condition is wrong, Since 'i' will never be negative according to loop body, Therefore it loop will get into infinite execution.

}

void func6()

{

char dataBuffer[100] = "";

char \* data = dataBuffer;

printf("Please enter a string: ");

if (fgets(data, 100, stdin) < 0)

{

printf("fgets failed!\n");

exit(1);

}

if(data != NULL)

{

printf("%s\n", data);

}

}

void func7()

{

char \* data; //pointer pointing to unsafe memory

data = "Fortify";

data = NULL;

printf("%s\n", data); //Here we should check if data points to null or not, Then print data accordingly so that user get desired result.

}

int main(int argc, char \* argv[])

{

printf("Calling func1\n");

func1();

printf("Calling func2\n");

func2();

printf("Calling func3\n");

func3();

printf("Calling func4\n");

func4();

printf("Calling func5\n");

func5();

printf("Calling func6\n");

func6();

printf("Calling func7\n");

func7();

return 0;

}