Que 1. A brief description of the tool with its main features (strengths and weaknesses)

Ans1. FlawFinder is a simple yet efficient ad quick tool that scans your C/C++ source code for calls to typical vulnerable library functions. It was developed by David Wheeler, a renowned security expert. It is run from the command line. Its output can easily be customized.

- Typical error types found:
  - Calls to library functions creating buffer overflow vulnerabilities (gets, strcpy, sprintf, ...)
  - Calls to library functions potentially vulnerable to string formatting attacks (sprintf, printf, ...)
  - Potential race conditions in file handling.

Strengths and Weaknesses

Strengths

* Flawfinder is easier to use - just give flawfinder a directory name, and flawfinder will enter the directory recursively, figure out what needs analysing, and analyse it.
* Flawfinder can handle internationalized programs also it can report column numbers (as well as line numbers) of hits.

Weaknesses

* Flawfinder is sometimes run slower than other analysis tools i.e. flawfinder does not produce fast.
* Other tools like RATS, works on other programming languages also, not only on C/C++ like flawfinder does.

Que 2. A description of the output of the tool for the program.

Ans 2. Last login: Sun Apr 9 11:27:21 on ttys000
Anands-MBP:~ anandkillampalli$ flawfinder Downloads/SSprog1
Flawfinder version 1.31, (C) 2001-2014 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 169
Examining Downloads/SSprog1/prefast_exercise.cpp
Examining Downloads/SSprog1/prefast_exercise_old_SAL_syntax.cpp
Examining Downloads/SSprog1/stdafx.h

FINAL RESULTS:

Downloads/SSprog1/prefast_exercise.cpp:19: [5] (buffer) gets:
Does not check for buffer overflows (CWE-120, CWE-20). Use fgets() instead.
Downloads/SSprog1/prefast_exercise_old_SAL_syntax.cpp:19: [5] (buffer) gets:
Does not check for buffer overflows (CWE-120, CWE-20). Use fgets() instead.
Downloads/SSprog1/prefast_exercise.cpp:48: [4] (shell) system:
This causes a new program to execute and is difficult to use safely
(CWE-78). try using a library call that implements the same functionality
if available.
Downloads/SSprog1/prefast_exercise_old_SAL_syntax.cpp:48: [4] (shell) system:
This causes a new program to execute and is difficult to use safely
(CWE-78). try using a library call that implements the same functionality
if available.
Downloads/SSprog1/prefast_exercise.cpp:36: [2] (buffer) memcpy:
Does not check for buffer overflows when copying to destination (CWE-120).

Make sure destination can always hold the source data.
Downloads/SSprog1/prefast_exercise_old_SAL_syntax.cpp:36: [2] (buffer) memcpy:
Does not check for buffer overflows when copying to destination (CWE-120).
Make sure destination can always hold the source data.

ANALYSIS SUMMARY:

Hits = 6
Lines analyzed = 271 in approximately 0.01 seconds (20928 lines/second)
Physical Source Lines of Code (SLOC) = 242
Hits@level = [0] 0 [1] 0 [2] 2 [3] 0 [4] 2 [5] 2
Hits@level+ = [0+] 6 [1+] 6 [2+] 6 [3+] 4 [4+] 4 [5+] 2
Hits/KSLOC@level+ = [0+] 24.7934 [1+] 24.7934 [2+] 24.7934 [3+] 16.5289 [4+] 16.5289 [5+] 8.26446
Minimum risk level = 1
Not every hit is necessarily a security vulnerability.
There may be other security vulnerabilities; review your code!
See 'Secure Programming for Linux and Unix HOWTO'
(http://www.dwheeler.com/secure-programs) for more information.
Aakash-MBP:~ aakashnandrajog$

Que 3. A corrected version of the program where the vulnerabilities found (if any) with the tool have been removed<

Ans 3.

```
#include "stdafx.h"

#include "stdio.h"

#undef __analysis_assume

#include <CodeAnalysis\SourceAnnotations.h>



#define BUF_SIZE 100

#define STR_SIZE 200



void zeroing();



char *my_alloc(size_t size) {

        char *ch  = (char *)malloc(size);

        *ch = NULL;
```

```
        ch[size] = NULL;  // null terminate here too, to be safe
        return ch;
}

HRESULT input(char *buf) {
        return (fgets(buf ,100 ,stdin) != NULL)?SEVERITY_SUCCESS:SEVERITY_ERROR; //Fixed
}

char *do_read() {
        char *buf = my_alloc(STR_SIZE);
        printf("Allocated a string at %x", buf);
        if (!input(buf)) {
                printf("error!");
                exit(-1);
        }
        if (*buf = NULL)
                printf("empty string");
        return buf;
}

void copy_data(char *buf1,
         char *buf2) {
        memcpy(buf2,buf1,STR_SIZE*200); //FIXED
        buf2[STR_SIZE-1] = NULL; // null terminate, just in case
}
```

```
void swap(char *buf1,
          char *buf2) {
        char *x = buf1;
        buf2 = buf1;
        buf1 = x;
}

int execute(char *buf) {
        return system(buf); // pass buf as command to be executed by the OS
}

void validate([SA_Pre(Tainted=SA_Yes)][SA_Post(Tainted=SA_No)] char *buf) {

   // This is a magical validation method, which turns tainted data
   // into untainted data, for which the code not shown.
   //
   // A real implementation might for example use a whitelist to filter
   // the string.

}

_Check_return_ int test_ready() {
        // code not shown
        return 1;
```

```
}


int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int
nCmdShow) {
        char *buf1 = do_read();
        char *buf2 = my_alloc(BUF_SIZE);
        if (buf2 == NULL)
                exit(-1);
        zeroing();
        test_ready();
        execute(buf1);

    char* buf3 = do_read();
        copy_data(buf3, buf2);
        execute(buf2);

        char *buf4 = my_alloc(STR_SIZE);
    char *buf5 = do_read();
    swap(buf4, buf5);
    execute(buf4);

}

// *****************************************************************

void zero(int *buf, int len)
```

```
{
   int i;
   for(i = 0; i <= len; i++)
      buf[i] = 0;
}

void zeroboth(int *buf, int len,
           int *buf3, int len3)
{
   int *buf2 = buf;
   int len2 = len;
   zero(buf2, len2);
   zero(buf3, len3);
}

void zeroboth2(int *buf, int len,
                int *buf3, int len3)
{
           zeroboth(buf, len3, buf3, len);
}

void zeroing()
{
   int elements[200];
   int oelements[100];
```

```
    zeroboth2(elements, 200, oelements, 100);

}
```

Output

```
Last login: Sun Apr  9 14:21:26 on ttys000
Anands-MBP:~ anandkillampalli$ flawfinder Downloads\SSprong1\prefast_exercise.cpp
Flawfinder version 1.31, (C) 2001-2014 David A. Wheeler.
Number of rules (primarily dangerous function names) in C/C++ ruleset: 169
Warning: Skipping non-existent file DownloadsSSprong1prefast_exercise.cpp

FINAL RESULTS:


ANALYSIS SUMMARY:

No hits found.
Lines analyzed = 0 in approximately 0.01 seconds (0 lines/second)
Physical Source Lines of Code (SLOC) = 0
Hits@level = [0]   0 [1]   0 [2]   0 [3]   0 [4]   0 [5]   0
Hits@level+ = [0+]   0 [1+]   0 [2+]   0 [3+]   0 [4+]   0 [5+]   0

Minimum risk level = 1
There may be other security vulnerabilities; review your code!
See 'Secure Programming for Linux and Unix HOWTO'
(http://www.dwheeler.com/secure-programs) for more information.
Aakash-MBP:~ aakashnandrajog$
```