CSC034U4E: Software Engineering
IIT Jammu, Autumn Semester, 2022-23
Lab Assignment No 2: Static Analysis using FlawFinder

Date uploaded/emailed :18th August 2022

Instructions:

I. *Date of Final submission: Will be specified two days before the due. This would then be a hard deadline.*
II. ***Maximum Points 200.***
III. *Submission Guidelines:*
   (a) *You may have to show/share your screen of the program that you are asked to show by the examiner.*
   (b) *You have to write a .tex file. Copy the source of each of your programs therein, along with brief analysis/inference of executing that program. The name of the file must containing the full extension used for writing the program file, when executing.*
   (c) *You will have to write a brief README.txt file - explaining how to execute the programs.*
   (d) *Compile all of the above in a .zip or .tar file.*
   (e) *Upload our .tar or .zip file as your submission on the Classroom.*
   (f) *Most important to note - you may NOT attempt all the problems from Sr no 1 to 47. That is, having attempted one problem, if you observe that any other problem after that illustrates the SAME vulnerability, you may not attempt the second (and similar other) problems.*

FlawFinder Installation::

1. Use the instructions at the page `https://dwheeler.com/flawfinder/andinstalthesoftware`.

Assignment Problems:

1. Run the code in Listing 1 and identify the errors. Modify the code to correct it, so that your static analysis tool does not flag any errors when compiled, with it. This example uses only interfaces present in C99, although the gets() function has been deprecated in C99 and eliminated from C11.

Listing 1: Problem 1

```
// improperly bounded string copies
#include <stdio.h>
#include <stdlib.h>
/* As above */

void get_y_or_n(void) {
        char response[8];
        printf("Continue? [y] n: ");
        gets(response);
        if (response[0] == 'n')
                exit(0);
                return;
}

void main {
        /* write appropriate code for the driver function main*/
        /* check the behaviour when more than 8 characters
        are entered at the prompt. */
}
```

2. Run the code in Listing 2 and identify the errors. Modify the code to correct it - i.e. Test the length of the input using strlen() and dynamically allocate the memory - so that your static analysis tool does not flag any errors when compiled, with it.

Listing 2: Problem 2

```
char *gets(char *dest) {
        int c = getchar();
        char *p = dest;
        while (c != EOF && c != '\n') {
                *p++ = c;
                c = getchar();
        }
        *p = '\0';
        return dest;
}

void main {
/* write appropriate code for the driver function main*/
/* check the behaviour when more than 20 characters
are entered at the prompt. */
}
```

3. Run the code in Listing 3 and identify the errors. Modify the code to correct it - i.e. check whether each of a[] nor b[] is properly terminated or not - so that your static analysis tool does not flag any errors when compiled, with it.

Listing 3: Problem 3

```
int main(void) {

        char a[16];
        char b[16];
        char c[32];
        strncpy(a, "0123456789abcdef", sizeof(a));
        strncpy(b, "0123456789abcdef", sizeof(b));
        strncpy(c, a, sizeof(c));

}
```

4. Run the code in Listing 4 and identify the errors. Modify the code to correct it - i.e. check whether the strings are properly terminated or not - so that your static analysis tool does not flag any errors when compiled, with it. Because nullterminated byte strings are character arrays, it is possible to perform an insecure string operation without invoking a function.

Listing 4: Problem 4

```
/*  */

int main(int argc, char *argv[]) {
        char buff[128];
        int i = 0;
}

char *arg1 = argv[1];
while (arg1[i] != '\0' ) {
        buff[i] = arg1[i];
        i++;
}
```

5. Run the code in Listing 5 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. Check for off-by-one error in the code snippet. OBO error or off-by-one bug (known by acronyms OBOE, OBO, OB1 and OBOB) is a logic error involving the discrete equivalent of a boundary condition. It often occurs in computer programming when an iterative loop iterates one time too many or too few. This

problem could arise when a programmer makes mistakes such as using "is less than or equal to" where "is less than" should have been used in a comparison, or fails to take into accountthat a sequence starts at zero rather than one (as with array indices in many languages).

Listing 5: Problem 5

```
/* As above */

iint main(void) {
        int i;
        char source[10];
        strcpy(source, "0123456789");
        char *dest = malloc(strlen(source));

        for (i=1; i <= 11; i++) {
                dest[i] = source[i];
        }
        dest[i] = '\0';
        printf("dest = %s", dest);
        }
```

6. Run the code in Listing 6 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. As the earlier code snippets, this code snippet is also meant to evaluate some specific aspect of a static analysis scanner's performance. Overall, the evaluation programs can be categorized as programs used to evaluate the detection of potential vulnerabilities and those used to evaluate resilience against false alarms. This code snippet deals specifically with the issues pertaining to the buffer overflow using a custom version of the strcpy() function.

Listing 6: Problem 6

```
/* As above  */
        char *stringcopy(char *str1, char *str2) {
        while (*str2)
                *str1++=*str2++;
        return str2;
        }

        int main(int argc, char **argv) {
                char *buffer=(char *) malloc(16*sizeof(char));
                stringcopy(buffer, argv[1]);
                printf("%s", buffer);
                return 1;
        }
```

7. Run the code in Listing 7 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. As the earlier code snippets, this code snippet is also meant to evaluate some specific aspect of a static analysis scanner's performance. Overall, the evaluation programs can be categorized as programs used to evaluate the detection of potential vulnerabilities and those used to evaluate resilience against false alarms. In this snippet the code does check for file descriptor tricks. If this proram is run as a setuid program, the attacker can exec() it after closing file descriptor 2. The next time the program opens a file, the file is associated with file descriptor 2, which is stderr. All output directed to stderr will go to the newly opened file. In this example, the attacker creates a symbolic link to the file that is to be overwritten. The name of the link contains the data to be written. When the program detects the symbolic link, it prints an error message and exits (line 32), but the error message, which contains the symbolinkname supplied by the attacker, is written into the targeted file.

Listing 7: Problem 7

```
/* As above */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main (int argc, char * argv []) {
        struct stat st;
        int fd;
        FILE * fp;

        if (argc != 3) {
                fprint(stderr, "usage: %s file message", argv[0]);
                exit(EXIT_FAILURE);
        }
        if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
                fprintf(stderr, "Can't open %s", argv[1]);
                exit(EXIT_FAILURE);
        }

        fstat(fd, &st);

        if (st.st_uid != getuid ()) {
                fprint(stderr, "%s not owner !", argv[1]);
                        exit(EXIT_FAILURE);
        }

        if (! S_ISREG (st . st_mode)) {
                fprintf(stderr, "%s not a normal file", argv[1]);
                                        // line 32
                exit(EXIT_FAILURE);
        }

        if ((fp = fdopen (fd, "w")) == NULL) {
                fprintf(stderr, "Can't open");
                exit(EXIT_FAILURE);
        }

        fprintf(fp, "%s", argv[2]);
        fclose(fp);
        fprintf (stderr, "Write Ok");
        exit(EXIT_SUCCESS);
}
```

8. Run the code in Listing 8 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. As the earlier code snippets,

this code snippet is also meant to evaluate some specific aspect of a static analysis scanner's performance. Overall, the evaluation programs can be categorized as programs used to evaluate the detection of potential vulnerabilities and those used to evaluate resilience against false alarms. This iproblem code snipet is a simple illustration of race condition, allowing the attacker to change the file named in argv[1] to a symbolic link after it's tested but before the file is opened. Many scanners detect the call to stat() on line 23, and while stat() is almost certainly a sign of trouble in this particular context, it needn't always be. A better scanner would actually detect the race condition between the open on line line 14 and the stat on line 23.

Listing 8: Problem 8

```
/* As above */
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

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

        struct stat st;
        int fd;
        FILE * fp;

        while((fd = open("/dev/null", O_RDWR)) == 0 || fd == 1 ||
                        fd == 2); //ln 14
        if (fd > 2)
                close(fd);

        if (argc != 3) {
                fprintf (stderr, "usage : %s file message ", argv [0]);
                exit(EXIT_FAILURE);
        }

        stat (argv[1], & st);  // line 23

        if (st.st_uid != getuid ()) {
                fprintf (stderr, "%s not owner !", argv [1]);
                exit(EXIT_FAILURE);
        }

        if (! S_ISREG (st.st_mode)) {
                fprintf (stderr, "%s not a normal file", argv[1]);
                exit(EXIT_FAILURE);
        }

        if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
                fprintf (stderr, "Can't open %s ", argv [1]);
                exit(EXIT_FAILURE);
        }

        if ((fp = fdopen (fd, "w")) == NULL) {
                fprintf (stderr, "Can't open");
                exit(EXIT_FAILURE);
        }

        fprintf (fp, "%s", argv [2]);
        fclose (fp);
        fprintf (stderr, "Write Ok");
        exit(EXIT_SUCCESS);
}
```

9. Run the code in Listing 9 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. As the earlier code snippets, this code snippet is also meant to evaluate some specific aspect of a static analysis scanner's performance. Overall, the evaluation programs can be categorized as programs used to evaluate the detection of potential vulnerabilities and those used to evaluate resilience against false alarms.This code snippet contains an exploitable format string vulnerability. The idea of this test to see whether the scanner can track taint through the exception handler. Ideally, the scanner should repor ta format string vulnerability on line 32, but not report the unexploitable format string vulnerability in the complementary program unexcept.c.

Listing 9: Problem 9

```
/* As above */

#include <stdio.h>
#include <ctype.h>
#include <string.h>

void func() {

        char buffer[1024];
        printf("Please enter your user id :");
        fgets(buffer, 1024, stdin);

        if (!isalpha(buffer[0])) {
                char errormsg[1044];
                strncpy(errormsg, buffer, 1024);
                                // guaranteed to be terminated
                strcat(errormsg, "is not a valid ID");
                                // we have room for this
                throw errormsg;
        }
}

main() {
        try {
                func();
        }
        catch(char * message) {
                fprintf(stderr, message);  // line 32
        }
}
```

10. Run the code in Listing 10 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. As the earlier code snippets, this code snippet is also meant to evaluate some specific aspect of a static analysis scanner's performance. Overall, the evaluation programs can be categorized as programs used to evaluate the detection of potential vulnerabilities and those used to evaluate resilience against false alarms. If the code snippet here is run as a setuid program, the attacker can exec() it after closing file descriptor 2. The next time the program opens a file, the file is associated with file descriptor 2, which is stderr. All output directed to stderr will go to the newly opened file. In this example, the attacker creates a symbolic link to the file that is to be overwritten. The name of the link contains the data to be written. When the program detects the symbolic link, it prints an error message and exits, but the error message, which contains the symbolic link name supplied by the attacker, is written into the targeted file. This isn't much different from the one in Listing 7 earlier, but the latter program was found on the web claiming to be a secure way of opening files. This program is somewhat simpler and, for some scanners, might make it easier to tell what the scanner is printing warnings about.

Listing 10: Problem 10

```
/* as above */

#include <stdio.h>
#define DATAFILE "/etc/aDataFile.data"

main(int argc, char **argv) {
        FILE *sensitiveData = NULL;
        FILE *logFile = NULL;

// Forgot to account for files 02, could be opening stderr;


        sensitiveData = fopen(DATAFILE, "w");

        if (!sensitiveData) {
                fprintf(stderr, "%s:failed to open %s", argv[0],
                                        DATAFILE);
                exit(1);
        }
        logFile = fopen(argv[1], "w");

        if (!logFile) {
                fprintf(stderr, "%s: failed to open %s ", argv[0],
                                        argv[1]);
                exit(1);
        }
}
```

11. Run the code in Listing 11 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. As the earlier code snippets, this code snippet is also meant to evaluate some specific aspect of a static analysis scanner's performance. Overall, the evaluation programs can be categorized as programs used to evaluate the detection of potential vulnerabiliti- ies and those used to evaluate resilience against false alarms. This particular code snippet is supposed to test the scanner's ability to handle preprocessor directives.

Listing 11: Problem 11

```
/* as above*/

#include <stdio.h>

#define SAFESTRCPY(a,b,c) strncpy(a, b, c)
#define FASTSTRCPY(a,b,c) strcpy(a, b)

main(int argc, char **argv) {
        size_t size = strlen(argv[3]);

char *buffer = (char *)malloc(1024);

#ifdef PARANOID
        SAFESTRCPY(buffer, argv[3], size+sizeof(char));
#else
        FASTSTRCPY(buffer, argv[3], size+sizeof(char));
#endif
}
```

12. Run the code in Listing 12 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. In this program, an attacker can supply a large value of len which overflows to zero on line 14. Since the subsequent read on line 15 uses the original value of len, the read can overflow the buffer. Many scanners will flag the read no matter what, which is useful but doesn't reflect what this program is trying to test. The complementary program notoverflow.c (coming later) is meant to check whether a scanner is actually detecting the possible overflow.

Listing 12: Problem 12

```
#include <stdlib.h>

void func(int fd) {
/* 3) integer overflow */
char *buf;
size_t len;

read(fd, &len, sizeof(len));
/* we forgot to check the maximum length */

buf = malloc(len+1);
read(fd, buf, len);

buf[len] = '\0';
}
```

13. Run the code in Listing 13 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This code snippet is from SecureProgramsHOWT/Odangersc.html. In this example, the attacker controlled number "len" is read as an integer, and even though there is a test to check if it's greater than the length of the buffer, a negative value for len will be converted to a large positive value when it gets cast to an unsigned integer in the second call to read.

Listing 13: Problem 13

```
void func(int fd) {
        /* 1) signed-ness DO NOT Do THIS */
        char *buf;
        int i, len;

        read(fd, \&len, sizeof(len));
        /* OOPS! We forgot to check for < 0 */
        if (len > 8000) {
                error("too large length");
                return;
        }

        buf = malloc(len);
        read(fd, buf, len); /* len casted to unsigned and overflows */
}
```

14. Run the code in Listing 14 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This code snippet illustrates a simple resource spoofing vulnerability where the characteristics of a fopened file are completely unchecked. Often this would be called a race condition as well, but technically it isn't since the necessary checks are missing entirely. Any first generation static analysis tool would be expected to generate warnings on this file because of the fopen(). What does FlawFinder say ?

Listing 14: Problem 14

```
#include <stdio.h>
void func() {
        FILE *aFile = fopen("/tmp/tmpfile", "w");

        fprintf(aFile, "%s", "hello world");
        fclose(aFile);
}
```

15. Run the code in Listing 15 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. In this code snippet, the goal is to test whether the scanner understands the preprocessor directives? This file tries to fool the scanner by making "strcpy" look like a variable instead of a function.

Listing 15: Problem 15

```
#define STRINGCOPY strcpy

int main(int argc, char **argv) {
        char *buffer = (char *)malloc(1024);
        STRINGCOPY(buffer, argv[3]);
}

void func() {
/* ideally this should not generate a warning because "strcpy" is just
being used as the name of a variable (and in fact it's dead code).
What does your scanner say? */
        int strcpy = 0;
        strcpy = strcpy + 1;
}
```

16. Run the code in Listing 16 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. Technically the buffer in this program has enough room for all the strncats, but the programmer forgot to terminate the buffer before the strncats start. Therefore line 7 contains a potential buffer overflow. Does your scanner detect ?

Listing 16: Problem 16

```
main(int argc, char **argv) {
        char *buffer = (char *)malloc(101);
        int i;

        for (i = 0; i < 10; i++)
                strncat(buffer, argv[i], 10);
}
```

17. Run the code in Listing 17 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. In this code snippet, strncat is called ten times in a loop, but the buffer receiving that data isn't big enough, so there's a potential buffer overflow on line 9. Does your scanner detect it ?

Listing 17: Problem 17

```
main(int argc, char **argv) {
        char *buffer = (char *)malloc(11);
        int i;
        buffer[0] = 0;

        for (i = 0; i < 10; i++)
                strncat(buffer, argv[i], 10);
}
```

18. Run the code in Listing 18 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This code snippet illustrates that strncpy doesn't automatically null terminate the string being copied into. In this example, the attacker supplies an argv[1] of length ten or more. In the subsequent strncat, data is copied not to buffer[10] as the code suggests, but to the first location to the left of buffer[0] that happens to contain a zero byte.

Listing 18: Problem 18

```
main(int argc, char **argv) {
        char *buffer = (char *)malloc(101);
        strncpy(buffer, argv[1], 10);
        strncat(buffer, argv[2], 90);
}
```

19. Run the code in Listing 19 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This code snippet is another illustration of a strncat used into an unterminated buffer. What is the message given by your FlawFinder?

Listing 19: Problem 19

```
main(int argc, char **argv) {
        char *buffer = (char *)malloc(101);
        strncat(buffer, argv[2], 90);
}
```

20. Run the code in Listing 20 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. Consider this code snippet. What is the effect of third argument to strncpy - especially how can the attacker control the third argument to make it unsafe.

Listing 20: Problem 20

```
#include <stdio.h>
main(int argc, char **argv) {
        int incorrectSize = atoi(argv[1]);
        int correctSize = atoi(argv[2]);
        char *buffer = (char *)malloc(correctSize+1);
        /* number of characters copied is based on user supplied value*/

        strncpy(buffer, argv[3], incorrectSize);
}
```

21. Run the code in Listing 21 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. In this code snippet, the program contains an integer truncation error. Superficially it looks like a safe program even though the variable len is tainted and len is used to determine the number of bytes read on line 18. It seems as though the buffer is large enough to accommodate whatever data ends up being placed there by the read statement. However, the program has a customized malloc function that takes an int argument, so in reality the malloc on line 3 doesn't always see the same argument as the read on line 18. A value of len larger than 2*MAXINT allows a buffer overflow on line 18. This example is somewhat contrived because of the large amount of memory that would have to be allocated for an exploit to succeed. On many architectures, len cannot be greater than 2*MAXINT.

Listing 21: Problem 21

```
#include <stdlib.h>
void *mymalloc(unsigned int size) {
        return malloc(size);    // line 3
}

void func(int fd) {
/* An example of an ERROR for some 64-bit architecures,
if "unsigned int" is 32 bits and "size_t" is 64 bits: */

        char *buf;
        size_t len;
        read(fd, &len, sizeof(len));

/* Has check for the maximum length been done ?*/

/* 64-bit size_t gets truncated to 32-bit unsigned int */
        buf = mymalloc(len);
        read(fd, buf, len); // line 18
}
```

22. Run the code in Listing 22 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This code snippet deals with umask system call. umask() system call controls the permissions of files created by the open call, but the permission mask is passed to the child process in an exec(). If this is a setuid program, the attacker can set a permission mask that makes these files "world-writable", but the new file may be a system critical one. In this program, the programmer uses the umask that existed when the program was exec()ed, but that umask might be controlled by an attacker.

Listing 22: Problem 22

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main() {
        int fd;
        FILE *fp;

        while((fd=open("/dev/null", O_RDWR)) == 0 || fd == 1
                                                || fd == 2);
        /* no file descriptor confusion */
        if (fd > 2)
                close(fd);
                /* files is in user-unwritable directory */
        fp = fopen("/etc/importantFile", "w");
        fclose(fp);
}
```

23. Run the code in Listing 23 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This program code interestingly illustrates the refutal of the statement in the man pages of umask (if at all so, i.e. not corrected) viz. "umask sets the umask to mask & 0777." In reality umask sets the mask to 0777 & mask, which is also contrary to the convention for chmod that most people are accustomed to. However, the correct usage is given lower down on the umask man page. In the snippet, the programmer uses umask to give the rest of the world full access to the newly created file while denying access to him or herself, which can safely be assumed to be a programming error.

Listing 23: Problem 23

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main() {
        int fd;
        FILE *fp;
        /* no file descriptor confusion */

        while((fd=open("/dev/null", O_RDWR))==0 || fd==1 || fd==2);
        if (fd > 2)
                close(fd);
        umask(700);   /* set permissions to /*---rwxrwx */

        /* file is in user unwritable directory */

        fp = fopen("/etc/importantFile", "w");

        fclose(fp);
}
```

24. Run the code in Listing 24 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This snippet illustrates a case where the user is getting a pathname as an argument and wants to find the first path component. The error is that the path in str might start with a '/', in which case len is zero. len 1 is the largest value possible for a size_t. In that particular case, the strncpy in the else clause is no safer than a strcpy. Note and analyze the responses of the FlawFinder you use.

Listing 24: Problem 24

```
#include <stdlib.h>

void func(char *str) { char buf[1024];

size_t len;

char *firstslash = strchr(str, '/');

if (!firstslash) strncpy(buf, str, 1023); /* leave room for the zero */
        buf[1023] = 0;
else {
        len = firstslash-str;
        /* length of the first path component */
        if (len > 1023)
                len = 1023;
        strncpy(buf, str, len-1);
        /* cut the slash off. Only copy len1 characters
                 to avoid zero padding */
        buf[len] = 0;
}
}
```

25. Run the code in Listing 25 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This code snippet focuses to illustrate the principle that incorrectly casting a pointer to a C++ object potentially breaks the abstraction represented by that object, since the (nonvirtual) methods called on that object are determined at compile time, while the actual type of the object might not be known until runtime. In this example, a seemingly safe strncpy causes a buffer overflow. In gcc the buffer overflows into object itself and then onto the stack, for this particular program. With some compilers the overflow might modify the object's virtual table. Check whether your FlawFinder flags in this test file.

Listing 25: Problem 25

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

class Stringg {
};

class LongString: public Stringg {
        private:
                static const int maxLength = 1023;
                char contents[1024];
        public:
                void AddString(char *str) {
                        strncpy(contents, str, maxLength);
                        contents[strlen(contents)] = 0;
                }
};

class ShortString: public Stringg {
        private:
                static const int maxLength = 5;
                char contents[6];
        public:
                void AddString(char *str) {
                        strncpy(contents, str, maxLength);
                        contents[strlen(contents)] = 0;
                }
};

void func(Stringg *str) {
        LongString *lstr = (LongString *)str;
        lstr->AddString("hello world");
}

main(int argc, char **argv) {
        ShortString str;
        func(&str);
}
```

26. Run the code in Listing 26 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This code snippet is meant to test whether FlawFinder can perform pointer alias analysis. Since that capability is generally only useful if the scanner provides some dataflow analysis capabilities, dataflow analysis is needed too. The variable that determines the size of a string copy is untainted, but alias analysis is needed to determine this.

Listing 26: Problem 26

```
int main(int argc, char **argv) {
        int len = atoi(argv[1]);
        int *lenptr_1 = &len;

        int *lenptr_2 = lenptr_1;
        char buffer[24];
        *lenptr_2 = 23;

        strncpy(buffer, argv[2], *lenptr_1);
}
```

27. Run the code in Listing 27 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This program contains a buffer overflow, but the overflowing data isn't controlled by the attacker. Ideally, a scanner should either not report a buffer overflow associated with this strcpy, or at most report a problem with lower severity than a strcpy whose argument is attacker controlled.

Listing 27: Problem 27

```
main() {
        char *buffer = (char *)malloc(10 * sizeof(char));
        strcpy(buffer, "fooooooooooooooooooooooooooooooooooooooooooooooooooooo");
}
```

28. Run the code in Listing 28 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. The program in this code snippet, contains a buffer overflow, but the overflowing data isn't controlled by the attacker. Ideally, a scanner should either not report a buffer overflow associated with this strcpy, or at most report a problem with lower severity than a strcpy whose argument is attacker controlled. The program is similar to the one in problem 27, but it presents a slightly harder problem for the scanner. In the program in Listing 28, the scanner could notice that the argument to strcpy is a constant string by looking for the quote symbol that follows the open parenthesis after the name of the function. In this program, some sort of dataflow analysis is needed (taint checking should be enough).

Listing 28: Problem 28

```
void func(char *foo) {
        char *buffer = (char *)malloc(10 * sizeof(char));
        strcpy(buffer, foo);
}

main() {
        func ("fooooooooooooooooooooooooooooooooooooooooooooooooooooo");
}
```

29. Run the code in Listing 29 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This code snippet represents another example of buffer overflow using a non-user defined. Here, the constant string is placed into a variable rather than being passed as a function argument like in the Listing 27. However, taint analysis should still be enough to let the scanner recognize that the overflowing string is not usercontrolled.A scanner should either not report a buffer overflow associated with this strcpy, or report a problem with lower severity than a strcpy whose argument is attackercontrolled.

Listing 29: Problem 29

```
main() {
        char *foo = "fooooooooooooooooooooooooooooooooooooooooooooooooooooo";

        char *buffer = (char *)malloc(10 * sizeof(char));
        strcpy(buffer, foo);
}
```

30. Run the code in Listing 30 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. The problem illustrates whether the FlawFinder flags the TOCTOU vulnerabilities, file descriptor vulnerabilities or umaskrelated vulnerabilities. This program ensures that stdin, stdout and stderr are accounted for, and then opens a file, ensuring that access checks are performed on the actual object being opened. The program doesn't set the umask, but that isn't necessary because the umask only affects the permissions of newly created files, and in this program open is called without the O_CREAT flag and therefore will only open a pre-existing file.

Listing 30: Problem 30

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main (int argc, char * argv []) {
        struct stat st;
        int fd;
        FILE * fp;

        while((fd = open("/dev/null", O_RDWR)) == 0 ||
                                        fd == 1 || fd == 2);i
        if (fd > 2) close(fd);
        if (argc != 3) {
                fprintf (stderr, "usage : %s file message ", argv [0]);
                exit(EXIT_FAILURE);
        }
        if ((fd = open (argv [1], O_WRONLY, 0)) < 0) {
                fprintf (stderr, "Can't open %s ", argv [1]);
                exit(EXIT_FAILURE);
        }

        fstat (fd, & st);

        if (st.st\_uid != getuid ()) {
                fprintf (stderr, "%s not owner ! ", argv [1]);
                exit(EXIT_FAILURE);
        }

        if (! S_ISREG (st.st_mode)) {
                fprintf (stderr, "%s not a normal file", argv[1]);
        exit(EXIT_FAILURE);
        }

        if ((fp = fdopen (fd, "w")) == NULL) {
                fprintf (stderr, "Can't open");
                exit(EXIT_FAILURE);
        }

        fprintf (fp, "%s", argv [2]);
        fclose (fp);
        fprintf (stderr, "Write Ok");

        exit(EXIT_SUCCESS);
}
```

31. Run the code in Listing 31 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This code snippet uses a variable sized buffer that syntactically looks like fixed sized buffer #1. Many security scanners generate a warning when they see a fixed-sized buffer. This test program declares a variable sized buffer based on the length of the string that's going to be copied into it, but it uses a syntax more commonly associated with fixed sized buffers. It's meant to determine whether a scanner detects fixed sized buffers by looking for square brackets after the variable name or whether it actually parses the declaration. What does your FlawFinder show? A scanner should not complain about a fixed sized buffer being used in this program.

Listing 31: Problem 31

```
#include <string.h>

void func(char *src) {
        char dst[(strlen(src) + 1) * sizeof(char)];
        strncpy(dst, src, strlen(src) + sizeof(char));
        dst[strlen(dst)] = 0;
}
```

32. Run the code in Listing 32 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This code snippet is another variant of the problem in the Listing 31. It also uses a variable sized buffer that syntactically looks like fixed sized buffer. It has the added twist the buffer might be to small if useString is called incorrectly, in spite of which there is no buffer overflow here because useString is called correctly (and is inaccesible from other source files).What does your FlawFinder show? A scanner should not complain about a fixed sized buffer being used or a potential buffer overflow, in this program.

Listing 32: Problem 32

```
#include <string.h>

static void useString(size_t len, char *src) {
        char dst[(len+1) * sizeof(char)];
        strncpy(dst, src, strlen(src));
        dst[strlen(src)] = 0;
}

void func(char *src) {
        size_t len = strlen(src);
        useString(len, src);
}
```

33. Run the code in Listing 33 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This program opens a file with a fixed name in a directory that shouldn't normally be accessible to an attacker. If, for some reason, the attacker has gained write access to /etc, this program could be used to overwrite files in other places, but the vulnerability is less serious than it would be if it opened a file in a directory that's normally writable.

Listing 33: Problem 33

```
#include <stdio.h> #include <sys/types.h> #include <sys/stat.h>
#include <fcntl.h>
main() {
        int fd;
        FILE *fp;  /* no file descriptor confusion */
while((fd = open("/dev/null", O_RDWR)) == 0 || fd == 1 || fd == 2);
        if (fd > 2)
                close(fd);
        umask(022); /* set umask */
        /* file is in user-unwritable directory */
        fp = fopen("/etc/importantFile", "w");
        fclose(fp);
}
```

34. Run the code in Listing 34 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. Insert additional other code in this snippet including the driver function main() to illustrate that the static analyzer must resolve the typedef to determine the data type of an array.

Listing 34: Problem 34

```
typedef char gchar;
void func() {
        gchar buf[10];
}
```

35. Run the code in Listing 35 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This program doesn't contain an integer overflow on line 15 because the length of the variable len is checked. It's meant to complement the code in the Listing 8, to check if buffer overflow warnings for that program are just mindlessly triggered by the read() call or if the static analyzer is actually spotting the overflow. A scanner shouldn't complain about an integer overflow on line 15 or a buffer overflow on line 16.

Listing 35: Problem 35

```
void func(int fd) {
        char *buf;
        size_t len;
        read(fd, &len, sizeof(len));
        /* check the maximum length. No need to check for negative
                numbers since size_t is unsigned already. */
        if (len > 1024)
                return;
        buf = malloc(len+1);
        read(fd, buf, len);
        buf[len] = '?';
```

36. Run the code in Listing 36 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This program uses strncpy and strncat safely, without introducing a buffer overflow. It's meant to check whether a scanner warns vacuously (i.e. without)about strncpy and strncat, or if it actually checks whether the sizes are OK and whether the buffer is terminated after the strncpy. A scanner should not report a buffer overflow on line 3 or line 5. */

Listing 36: Problem 36

```
main(int argc, char **argv) {
        char *buffer = (char *)malloc(25);
        strncpy(buffer, argv[1], 10); // line 3
        buffer[10] = 0;
        strncat(buffer, argv[2], 10); // line 5
}
```

37. Run the code in Listing 37 and identify the errors. This code snippet is believed to be safe invocation of strcpy. This use of strcpy ensures that the buffer is large enough to accommodate the string being copied. The data-flow analysis needed to verify this may be too complex to be accomplished with simple taint checking. A normal static analysis tool should not warn of a buffer overflow error on line 7.

Listing 37: Problem 37

```
#include <string.h>
char *strsave(char *src) {
        size_t len = strlen(src);
        char *result = (char *)malloc((len+1) * sizeof(char));
        if (result)
                strcpy(result, src);
        return result;  // line 7

}
```

38. Run the code in Listing 38 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This program complements the snippet in the listing 17, which is taken from the linux secure programming HOWTO. It avoids the integer truncation problem of the one in listing 17 and it's meant to test whether a scanner that reports a buffer overflow for that listing is doing so monotonously or whether it actually noticed the possible integer truncation. In this program, the developer has defined a custom version of the malloc function which takes an int argument, and thereby creates the possibility of an integer truncation vulnerability, but bounds checking prevents the malloc on line 1 from seeing a different length value than the read on line 16. The listing 39 differs from this one because both mymalloc and read take the original usercontrolled size_t len as an argument, but those calls are unreachable for values of len that would cause truncation problems. Ideally, a security scanner should not report a possible bounds violation on line 15 or a buffer overflow on line 16.

Listing 38: Problem 38

```
#include <values.h> #include <stdlib.h>
void *mymalloc(unsigned int size) {
        return malloc(size);
} // line 1
void func(int fd) {
        char *buf;
        size_t len;
        read(fd, &len, sizeof(len));

        if (len > MAXINT)
                return;
        buf = mymalloc(len);
        read(fd, buf, len);
// line 15 // line 16
}
```

39. Run the code in Listing 39 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. This program complements the snippet in the listing 17, which is taken from the linux secure programming HOWTO. It avoids the integer truncation problem of 17, and it's meant to test whether a scanner that reports a buffer overflow for truncated.c is doing so without using any intelligent checking or whether it actually noticed the possible integer truncation. The code has a statement that reads a tainted integer and use it to determine the size of a subsequent read of a tainted string. But the buffer receiving the data during the second read is allocated according to user provided length, and read will only put that many bytes in the buffer, so there should be no overflow. In this particular variant of the program, the user has defined his own version of the malloc function which takes an int argument and thereby creates the possibility of an integer truncation vulnerability. However, the program casts "len" to an integer and thereby ensures that the second argument to read (line 18) is the same number as the argument of mymalloc on line 17. Ideally, a security scanner should not report a possible bounds violation on line 17 or a buffer overflow on line 18.

Listing 39: Problem 39

```
#include <values.h> #include <stdlib.h>
void *mymalloc(unsigned int size) {
        return malloc(size);
}

void func(int fd) {
        char *buf;
        size_t len;
        int actual_len;

        read(fd, &len, sizeof(len));
        actual_len = len;

        buf = mymalloc(actual_len); \\line 17
        read(fd, buf, actual_len); \\ line 18
}
```

40. Run the code in Listing 40 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. The code snippet in this program avoids a sign error by checking of the variable len is negative. Write appropriate driver function for this program. It complements a program seen before, where an attacker can create a buffer overflow by specifying a negative number for len. A scanner should not report a buffer overflow on line 11.

Listing 40: Problem 40

```
void func(int fd) {
        char *buf;
        int i, len;
        read(fd, &len, sizeof(len));
        if (len < 0 || len > 7999) {
                error("too large length");
                return;
        }
        buf = malloc(8000);
        read(fd, buf, len);  //line 11
}
```

41. Run the code in Listing 41 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. In this program, the target string is properly terminated but the terminating null is added before the strncpy(), which might fool a scanner into thinking that the buffer is unterminated. A scanner should not complain about an unterminated strncpy(). Write appropriate driver function for this program.

Listing 41: Problem 41

```
void func(char *str) {
        char target[(strlen(str) + 1) * sizeof(char)];
        target[strlen(str)] = 0;
        strncpy(target, str, strlen(str));
}
```

42. Run the code in Listing 42 and identify the errors. Modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. The catch block in this program contains an unexploitable formatstring vulnerability. The idea of this test to see whether the scanner can track taint through the exception handling mechanism. Ideally, the warning given by the scanner for line 31 should have lower severity than the corresponding (exploitable) format string vulnerability in an earlier such program seen before.

Listing 42: Problem 42

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void func() {
        char buffer[1024];
        printf("Please enter your user id :");
        fgets(buffer, 1024, stdin);
        if (!isalpha(buffer[0])) {
                char errormsg[1044];
                strcpy(errormsg, "that isn't a valid ID");t
                hrow errormsg;
        }
}
main() {
        try {
                func();
        } catch(char * message) {
                fprintf(stderr, message);        // line 31
        }
}
```

43. Revisit the code attempted in the Stack Analysis exercise no1 - reproduced below for the sake of

convenience. But now, run the code in Listing 6 giving "1234567890123456j->*!" as the input see what happens. Identify the errors. Is this a security vulnerability? Explain how could it be exploited. Can you modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it.

Listing 43: Problem 43

```
bool IsPasswordOK(void) {
        char Password[12]; // Memory storage for pwd
        gets(Password); // Get input from keyboard
        return 0 == strcmp(Password, "goodpass");
}

int main(void) {
        bool PwStatus; // Password status
        puts("Enter Password:"); // Print
        PwStatus=IsPasswordOK(); // Get and check password
        if (!PwStatus) {
                puts("Access denied"); // Print
                exit(-1); // Terminate program
        }
        else
        puts("Access granted");// Print
}
```

44. Run the code in Listing 44(a) after writing appropriate driver function (i.e. main()). This code snippet is meant to illustrate the vulnerability that arises when changing working directories without verifying the success is security-sensitive i.e. gives rise to security issues. Note that the purpose of changing the current working directory is to modify the base path when the process performs relative path resolutions. When the working directory cannot be changed, the process keeps the directory previously defined as the active working directory. Thus, verifying the success of chdir() type of functions is important to prevent unintended relative paths and unauthorized access. Does your static analysis tool flag an error. Note that SonarLint WILL surely flag an error here. Now, modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it. Lastly, run the code in Listing 44(b) (after writing appropriate driver function (i.e. main()) and observe the consequences.

Listing 44: Problem 44(a)

```
// Listing 44(a)
const char* any_dir = "/any/";
chdir(any_dir); // Sensitive: missing check of the return value

int fd = open(any_dir, O_RDONLY | O_DIRECTORY);
fchdir(fd); // Sensitive: missing check of the return value


// Listing 44(b)
const char* root_dir = "/jail/";
if (chdir(root_dir) == -1) {
        exit(-1);
} // Is this Compliant, now ?

int fd = open(any_dir, O_RDONLY | O_DIRECTORY);
if(fchdir(fd) == -1) {
        exit(-1);
} // s this Compliant, now ?
```

45. Run the code in Listing 45 - after inserting appropriate driver function - and identify the errors, if there are any. Modify the code, i to correct it, f required, so that your static analysis tool does not flag any errors when compiled, with it.

Listing 45: Problem 45

```
for (int i = 0; i < 10; i++) { // IS this compliant - how many times
                                  does the loop execute ?
        printf("i is %d", i);
        break;
}
/* some other code here */
... ... ...
for (int i = 0; i < 10; i++) { // S this compliant - how many times
                        does the loop execute ?
        if (i == x) {
                break;
        }
        else {
                printf("i is %d", i);
                return;
        }
}
```

46. How to mitigate the attacks arising out of the buffer overflows? This exercise triggers the thought process arising out of this issue. The mitigation approaches include strategies designed to

    (a) prevent buffer overflows from occurring

    (b) detect buffer overflows and securely recover without allowing the failure to be exploited

    (c) rather than completely relying on a given mitigation strategy, it is often advantageous to follow a defense-in-depth tactic that combines multiple strategies.

    (d) a common approach is to consistently apply a secure technique to string handling (a prevention strategy) and back it up with one or more runtime detection and recovery schemes.

    Note that Microsoft proposed the bounds-checking interfaces as alternative library functions that promote safer, more secure programming. These are defined and known as C11 Annex K library functions OR also as _s functions, such as scanf_s(), printf_s() OR the strcpy_s(), strcat_s(), strncpy_s(), and strncat_s() functions as replacements for scanf(), printf(), strcpy(), strcat(), strncpy(), and strncat(). These are also optional standard. MSVC has implemented these functions, but gcc hasn't. However, without being bothered about whether accepted by non-Microsoft community or not, let us try to understand the logic behind proposing these functions. These alternative functions verify that **output buffers are large enough for the intended result** and return a failure indicator if they are not. The result is data is never written past the end of an array. And, all string results are null terminated. Now, look at the code shown in Listing 46. Modify this code for gcc, while replacing gets_s() with gets(), but preserving the purpose of gets_s i.e. ensuring the array bounds are checked when executing gets().

Listing 46: Problem 46

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

void get_y_or_n(void) {
        char response[8];
        size_t len = sizeof(response);
        printf("Continue? [y] n: ");
        gets_s(response, len);  \\ this statetment is to be replaced
                                   with appropriate code

        if (response[0] == 'n')
                exit(0);
}
```

47. This is continuation of the Problem 46. Analyze the Listing 47. The function getline() used here was originally a GNU extension. In fact, getline() and getdelim() were originally GNU extensions. They were standardized in POSIX.1-2008. getline() represents a memory management model in which the Callee (i.e. the function which is called) allocates the memory whereas the

caller (i.e. the calling function) frees. When using it, it must be made sure that there is enough memory available (except when a call to malloc() fails).

Run the code in Listing 47 and see whether it works or not in your compiler. This code is intended to support dynamically allocated memory while ensuring that buffer overflow does not occur. Does it run properly with your compiler ? Then, run it with splint as well as with FlawFinder and comment on the responses given. In case these tools crib, then modify the code to correct it so that your static analysis tool does not flag any errors when compiled, with it.

Listing 47: Problem 47

```
#define #include <stdio.h>
#include <stdlib.h>
void get_y_or_n(void) {
        char *response = NULL;
        size_t size;
        printf("Continue? [y] n: ");

        if ((getline(&response, &size, stdin) < 0) ||
        (size && response[0] == 'n')) {
        free(response);
        exit(0);
        }
        free(response);
}
```

48. The next task is to be attempted after the topics on Dynamic Testing have been covered in the class. You have to do analysis on any ten appropriate programs from all of the programs implemented above, using a dynamic analysis tool. Discuss with me the dynamic analysis tool that you decide to use, first. Your job is to figure out how far one succeeds with the static analysis - how much the static analysis helps. This analysis and comparison should help you in coming out with an abstract that helps one understand, analyze and compare the static and dynamic analysis approaches.

*****ends here*****