CSC034U4E: Software Engineering
IIT Jammu, Autumn Semester, 2022-23
Lab Assignment No 1: Static Analysis using Splint

Date uploaded/emailed :17th 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.*

1. Analyze and briefly discuss how Rice′*s* theorem applies to Static Analysis.

2. Run the code in Listing 1 with splint and identify the error. Modify the code to correct it, so that splint does not flag any errors when compiled.

Listing 1: Problem 2

```
int main() {
        char *p;
        int x, s;
        char* foo();
        printf("Please enter the value of x \n");
        scanf("%d", &x);
        if (x==0)
                p = 0;
        else
           p = foo();
        if (x !=0)
                s=*p;
        else
        printf("The value of x entered is %d\n", x);
        return;
}
```

3. Run the code in Listing 2 with splint. Compile all with gcc and state whether there are any errors flagged or not. Modify the code to correct it, so that splint does not flag any errors when compiled, if at all it flags errors earlier.

Listing 2: Problem 3: Double Free Errors

```
        #include <stdlib.h>
        void test(void *ptr)
        {
        free(ptr);
        free(ptr);
        }
```

4. Run the the code in Listing 3 with splint and identify the error. Modify the code to correct it, so that splint does not flag any errors when compiled.

Listing 3: Problem 4

```
#include <stdio.h>
#include <stdlib.h>
#define  MAXTMP 80
void static foo(void)
{
        char *tmp;
        tmp = (char *) malloc(MAXTMP);
        *tmp = 'X';
        free(tmp);
}
int main()
{
        foo();
        return 0;
}
```

5. Run the code in Listing 4 with splint. Compile all with gcc and state whether there are any errors flagged or not. Modify the code to correct it, so that splint does not flag any errors when compiled, if at all it flags errors earlier.

Listing 4: Problem 5: Code without sanity check on the maximum memory allocated

```
int main(int argc, char *argv[]) {
        if (argc == 42) {
                char *p,*q;
                p = NULL;
                printf("%s",p);
                q = (char *)malloc(100);
                p = q;
                free(q);
                *p = 'x';
                free(p);
                p = (char *)malloc(100);
                p = (char *)malloc(100);
                q = p;
                strcat(p,q);
                assert(argc > 87);
        }
}
```

6. Run the code in Listing 5, 6 and 7 with splint. Find out the purposes of each incrementally improved program. Compile all with gcc and state whether there are any errors flagged or not. Modify the code to correct it, so that splint does not flag any errors when compiled, if at all it flags errors earlier.

Listing 5: Problem 6: Code without sanity check on the maximum memory allocated

```
int main(int argc, char **argv) {
        char *str;
        int len;
        if ((str = (char *)malloc(BUFSIZE)) == NULL) {
                return FAILURE_MEMORY;
        }
        len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
        if (len >= BUFSIZE) {
                free(str);
                if ((str = (char *)malloc(len + 1)) == NULL) {
                        return FAILURE_MEMORY;
        }
        snprintf(str, len + 1, "%s(%d)", argv[0], argc); }
        printf("%s\n", str);
        free(str);
        str = NULL;
        return SUCCESS;
}
```

Listing 6: Problem 6: Code with sanity check on the maximum memory allocated

```
int main(int argc, char **argv) {
        char *str;
        int len;
        if ((str = (char *)malloc(BUFSIZE)) == NULL) {
                return FAILURE_MEMORY;
        }
        len = snprintf(str, BUFSIZE, "%s(%d)", argv[0], argc);
        if (len >= BUFSIZE) {
                free(str);
        if (len >= MAX_ALLOC) {
                return FAILURE_TOOBIG;
        }
        if ((str = (char *)malloc(len + 1)) == NULL) {
                return FAILURE_MEMORY;
        }
        snprintf(str, len + 1, "%s(%d)", argv[0], argc);
        }
        printf("%s\n", str);
        free(str);
        str = NULL;
        return SUCCESS;
}
```

Listing 7: Problem 6: Tracking Buffer Sizes

```
typedef struct{
        char* ptr;
        int bufsize;
} buffer;
//define all the constants used in the program appropriately
int main(int argc, char **argv) {
        buffer str;
        int len;
        if ((str.ptr = (char *)malloc(BUFSIZE)) == NULL) {
                return FAILURE_MEMORY;
        }
        str.bufsize = BUFSIZE;
        len = snprintf(str.ptr, str.bufsize, "%s(%d)", argv[0], argc);
        if (len >= BUFSIZE) {
                free(str.ptr);
                if (len >= MAX_ALLOC) {
                        return FAILURE_TOOBIG;
        }
        if ((str.ptr = (char *)malloc(len + 1)) == NULL) {
                return FAILURE_MEMORY;
        }
        str.bufsize = len + 1;
        snprintf(str.ptr, str.bufsize, "%s(%d)", argv[0], argc);
        }
        printf("%s\n", str.ptr);
        free(str.ptr);
        str.ptr = NULL;
        str.bufsize = 0;
        return SUCCESS;
}
```

7. Run the the code in Listing 8 with splint and identify the error. Modify the code to correct it, so that splint does not flag any errors when compiled.

Listing 8: Problem 7

```
map(l,f,z) {
        int  r;
        if (l==null)
                r=z;
        else
                r=f(map(*l,f,z));
        return r;
}
foo(i) {
        return i+1;
}

void main() {
        int  h,t,n;
        t = null;
        n = 42;
        while (n>0) {
                n = n-1;
                h = (int *) malloc sizeof(int);
                *h = t;
                t = h;
        }
        return map(h,foo,0);
}
```

8. Run the the code in Listings 9 to 12 with splint and identify the error. Modify the code to correct it, so that splint does not flag any errors when compiled.

Listing 9: Problem 8

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

static jmp_buf env;

static void inner(void)  {
        longjmp(env, 1);
}

static void middle(void) {
        void *ptr = malloc(1024);
        inner();
        free(ptr);
}

void outer(void) {
        int i;
        i = setjmp(env);
        if (i == 0)
        middle();
}

int main() {
/* write appropriate body of main calling
        all the three previous functions */
}
```

Listing 10: Problem 8

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

void test(const char *filename)  {
        FILE *f = fopen(filename, "r");
        void *p = malloc(1024);
        /* do stuff */
}
```

Listing 11: Problem 8

```
#include <stdlib.h>
struct link { struct link *next; };

int free_a_list_badly(struct link *n)  {
        while (n) {
                free(n);
                n = n->next;
        }
}
```

Listing 12: Problem 8

```
#include <stdlib.h>
void test(int n)  {
        int buf[10];
        int *ptr;
        if (n < 10)
                ptr = buf;
        else
                ptr = (int *)malloc(sizeof (int) * n);
        /* do stuff.  */
        /* oops; this free should be conditionalized.  */
                free(ptr);
}
```

9. Run the the code in Listing 13 with splint and identify the error. Modify the code to correct it.

Listing 13: Problem 9

```
#include <stdio.h>
#include <signal.h>
extern void body_of_program(void);
void custom_logger(const char *msg)  {
        printf(stderr, "LOG: %s", msg);
}
static void handler(int signum) {
        custom_logger("got signal");
}
int main(int argc, const char *argv) {
        custom_logger("started");
        signal(SIGINT, handler);
        body_of_program();
        custom_logger("stopped");
        return 0;
}
```

10. Run splint on the code shown in Listing 14 and explain the errors:.

Listing 14: Problem 10

```
#include <stdlib.h>
static int* f1() {
        int value;
        printf("Input Number: ");
        (void) scanf("%d", &value);
        return &value;
}
static char* f2() {
        return "TESTING";
}
int main() {
        int *retvalue;
        char *str = (char *)malloc(sizeof(char));
        retvalue = f1();
        if      (*retvalue > 0 && str != NULL)  {
                strcpy(str, f2());
                printf("String: %s \n", str);
        }
        if(str != NULL)
                free(str);
                if(retvalue != NULL)
                        free(retvalue);
                return(1);
}
```

11. Using the /@*null*@/, the /@*observer*@/, /@*only*@/ annotations, correct the program in Problem #10 above, here.

12. Run the the code in Listing 15 with splint and identify the errors, if any. Modify the code with the splint buffer overflow check invariant maxSet(), appropriately. Comment your program code to explain how splint models blocks of contiguous memory using two properties: maxSet and maxRead, as also how are the invariants inserted by splint as pre and post conditions to check of anomalous code.

Listing 15: Problem 12

```
/*include the files <stdio.h> <string.h> <stdlib.h>. <stddef.h>
void static updateEnv(char *str, size_t size) {
        char *tmp;
        tmp = getenv("HOME");
        if (tmp != NULL) {
        strncpy(str,tmp,size -1);
        str[size -1] = '\0';
        }
}
int main() {
        char *str;
        size_t size;
        str = "Hello World";
        size = strlen(str);
        updateEnv(str, (size -1));
        printf("\nThe Environment variable copied\n");
        return 0;
}
```

13. Run the the code in Listing 16 with splint and identify the error. Modify the code to correct it.

Listing 16: Problem 13

```
#include <stdlib.h>
#include <string.h>
struct check {
        char *sname;
        size_t ncount;
};
static int f1(struct check *testc) {
        char *b = (char *) malloc(sizeof(char));
        if(b == NULL)    return 0;
        printf("Input String: ");
        (void) scanf("%s", b);
        testc->sname = b;
        testc->ncount = strlen(b);
        return 1;
}
static char* f2(){
        char *str = (char*) malloc(sizeof(char));
        if(str != NULL)
                strcpy(str, "TESTING");
        return str;
}
int main(){
        struct check *c = (struct check*)malloc(sizeof(struct check));

        if(c==NULL)
                exit(0);
        if(f1(c) == 0) {
                if(c->sname != NULL)
                        free(c->sname);
                c->sname = f2();
                if(c->sname != NULL)
```

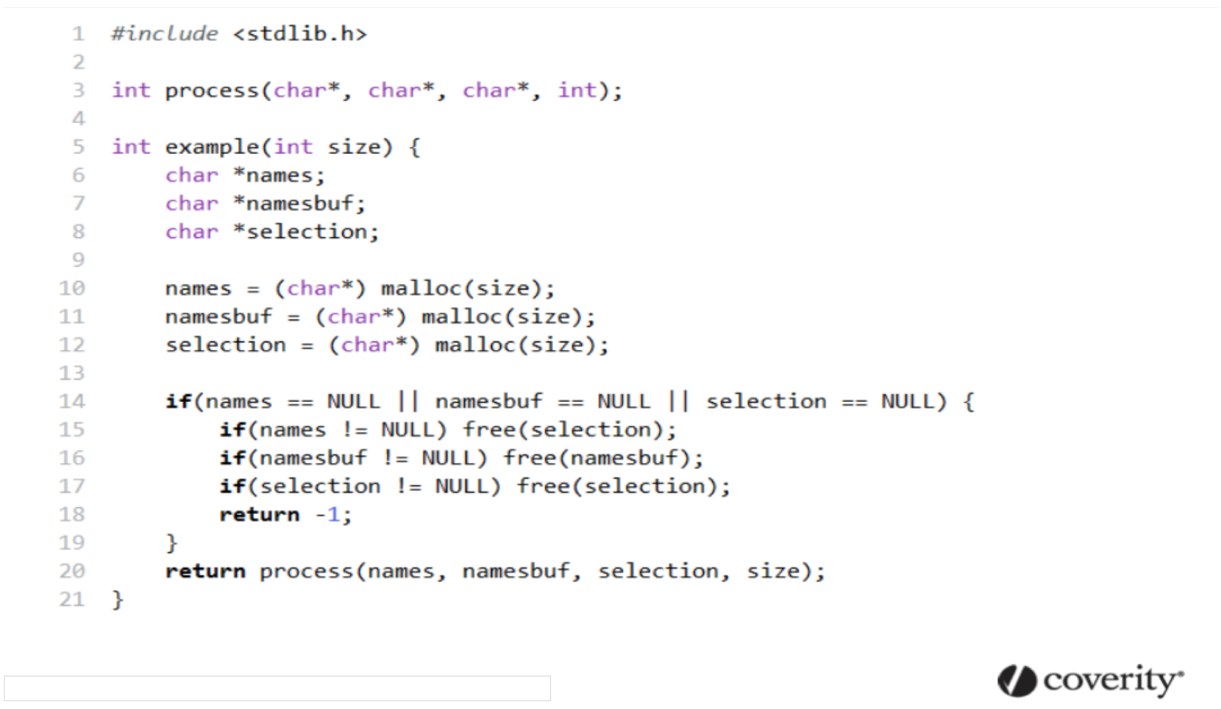
```
                        c->ncount = strlen(c->sname);
        }
        if(c != NULL)
                free(c);
        return(1);
}
```

14. Use the /@*null*@/ and the /@*in*@/, /@*out*@/ annotations to ensure correctness in execution in Problem #13, above in this assignment.

15. Use the program uploaded viz. Problem10Assign1.c and explain all the errors flagged of by splint.

16. Use the program uploaded viz. Problem11Assign1.c and explain all the errors flagged of by splint.

17. Consider the code shown in Figure 1 here. First logically analyze the code and comment whether it contains any errors or not. Insert appropriate main() function for the code to compile correctly with gcc. Then, compile the modified code splint and comment on the errors flagged. Use your observations to answer the following question: *Can the lifetime of a variable exceed its scope ?* Justify your answer with appropriate illustration(s).



```
#include <stdlib.h>

int process(char*, char*, char*, int);

int example(int size) {
    char *names;
    char *namesbuf;
    char *selection;

    names = (char*) malloc(size);
    namesbuf = (char*) malloc(size);
    selection = (char*) malloc(size);

    if(names == NULL || namesbuf == NULL || selection == NULL) {
        if(names != NULL) free(selection);
        if(namesbuf != NULL) free(namesbuf);
        if(selection != NULL) free(selection);
        return -1;
    }
    return process(names, namesbuf, selection, size);
}
```

Figure 1: Code for Problem 17

18. Run the the code in Listing 16 with splint and identify the errors, if any. Which type of testing and analysis is best useful to find out the errors in such code.

Listing 17: Problem 12

```
/*include the appropriate header files and
write the driver function main also */
function getFullName(firstName) {
if (firstName == "Sachin")
        return "Sachin Tendulkar"
if (firstName == "Rahul")
        return "Ganguly" /* Is this  correct business logic ? */

if (firstName != "Sachin or Rahul")
        return "Star bowlers of Indian Cricket Team"
}

int main() {
        ... ...
        /*write appropriate code to run the program*/
}
```

19. Run the the code in Listing 18 with splint and identify the errors, if any.

Listing 18: Problem 19

```
/*include the appropriate header files ,  some function body
(wherever missing) and  write the driver function main also */
void foo(int x) {
        if (x == 0) {
                bar();
                cli();
        }
        else {
                baz();
                cli();
        }
        while (x > 0) {
                sti();
                do_work();
                cli();
        }
        sti();
}
```

20. Run the the code in Listing 18 with splint and identify the errors, if any.

Listing 19: Problem 20

```
/*include the appropriate header files ,  some function body
(wherever missing) and  write the driver function main also */
void foo(unsigned n) {
        char str = new char[n+1];
        int idx = 0;
        if (n > 5)
                idx = n;
        else
                idx = n+1;
        str[idx] = 'c';
}
```