UE18CS257C

SECURED PROGRAMMING WITH C

Static Code Analysis Assignment - Parabolic Encryption

Team:

Name	SRN
Aditeya Baral	PES1201800366
Vishesh P	PES1201800314
Vinay Kirpalani	PES1201800218

Introduction:

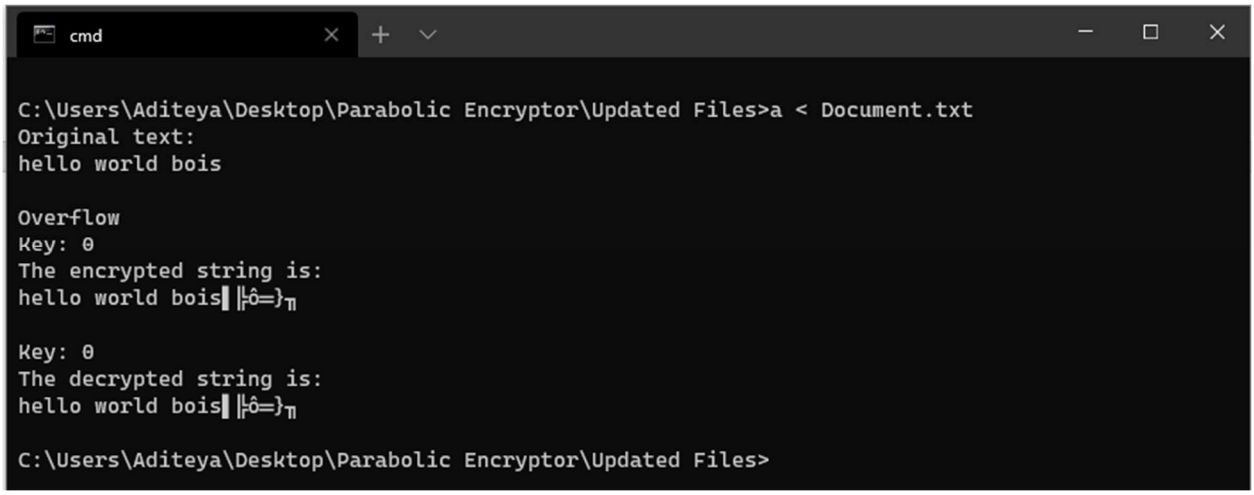
Our project implements the parabolic encryption algorithm, which takes in a string input, converts each character of the string into some other character based on the equation of an upward parabola $\boldsymbol{y^2 = 4ax}$, where ***x*** is the index of that particular input character, ***a*** is a the encryption distance calculated during runtime and ***y*** is the function of ***x***, which is added to the ASCII value of the original character to get the encrypted character.

Our project also implements the decryption algorithm which takes in the encrypted string obtained using the above encryption algorithm, applies the inverse function of the parabola function : $\boldsymbol{x = -\sqrt{4ay}}$ on each character of the string, where ***y*** is the index of the encrypted input character, ***a*** is the same encryption distance as before and ***x*** is the function of ***y***, which is added to the ASCII value of the encrypted character to get the decrypted character.

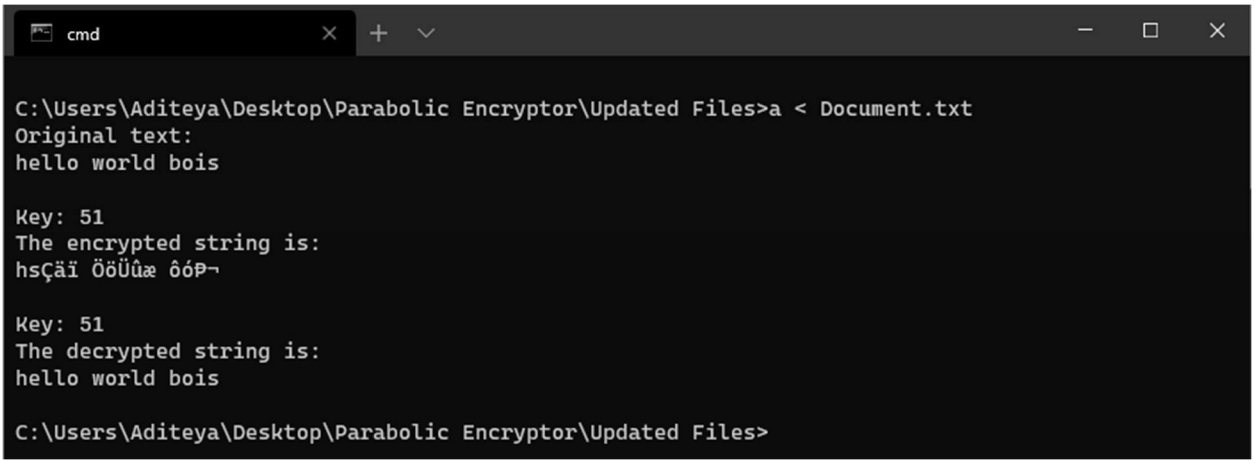
We have used various static analysis tools such as cppcheck, valgrind and splint since each tool works differently and picks up specific types of issues, so using a combination of them helps us to analyse all kinds of vulnerabilities in our code.

Execution:

(i) Vulnerable files:



(ii) Secure Files:



Vulnerabilities and Mitigations:

1. Failing to specify the type of the variable ‘sum’:

If we do not provide a declaration to the sum variable, it will lead to type casting as an integer. Although we do want it as an integer, this should not be done to avoid type mismatches and possible loss of information.

```
sum = 0;

/*
If we do not provide a declaration to the sum variable, it will lead to type casting as an ineteger.
Although we do want it as an integer, this shouldn't be done to avoid type mismatches and possible loss of informarion.
*/
```

Mitigation:

Variable sum has been declared as an integer.

```
int sum = 0;

// sum has been declared as an integer
```

2. Incorrect declaration of the string pointer 'str':

If the input size is larger than allocated size, it will result in a buffer overflow. However, if the size if greater than 10^10, it will lead to segmentation faults in functions used later like strlen(). We should also use calloc() to ensure the entire array has been initialized to null string.

```
char* str = (char*)malloc(sizeof(char)*100);

/*
If the input size is larger than allocated size, it will result in a buffer overflow.
However, if the size if greater than 10^10, it will lead to segmentation faults in functions used later like strlen.
We should also use calloc to ensure the entire array has been initialised to null string.
*/
```

Mitigation:

A large size has been used and calloc() has been used to initialize all locations to null string.

```
char* str = (char*)calloc(100000,sizeof(char));

// A large size has been used and calloc has been used to initialise all locations to null string
```

3. The use of the gets() function when we know it has security flaws:

gets() is not safe since it is unbounded.

```
gets(str);
/* gets() is not safe since it is unbounded. Should be replaced with scanf */
```

What *splint* shows:

```
Main-vulnerable.c: (in function main)
Main-vulnerable.c:18:5: Use of gets leads to a buffer overflow vulnerability.
                          Use fgets instead: gets
  Use of function that may lead to buffer overflow. (Use -bufferoverflowhigh to
```

Mitigation:

gets() has been replaced with scanf() since it is safe.

```
scanf("%100000[^\n]s",str);     // gets() has been replaced with scanf since it is safe
```

4. Incorrect declaration of the string pointer ‘enstr’:

Since the encrypted string size is lesser than the input size, it will result in an overflow. The allocated size must be equal to or more than the input size.

```
char* enstr = (char*)malloc(sizeof(char)*10);

/*
Since the encrypted string size is lesser than the input size, it will result in an overflow.
The allocated size must be equal to or more than the input size.
*/
```

Mitigation:

The allocated size has been set to the input size and calloc() has been used to initialize it to null string.

```
char* enstr = (char*)calloc(100000,sizeof(char));

// The allocated size has been set to the input size and calloc has been used to initialize it to null string.
```

5. Double-freeing the pointer ‘str’ and failing to free the pointer ‘enstr’:

There are multiple free statements for the same pointer. There should only be one free statement. Also, dynamic array enstr has not been freed. This is going to result in a memory leak.

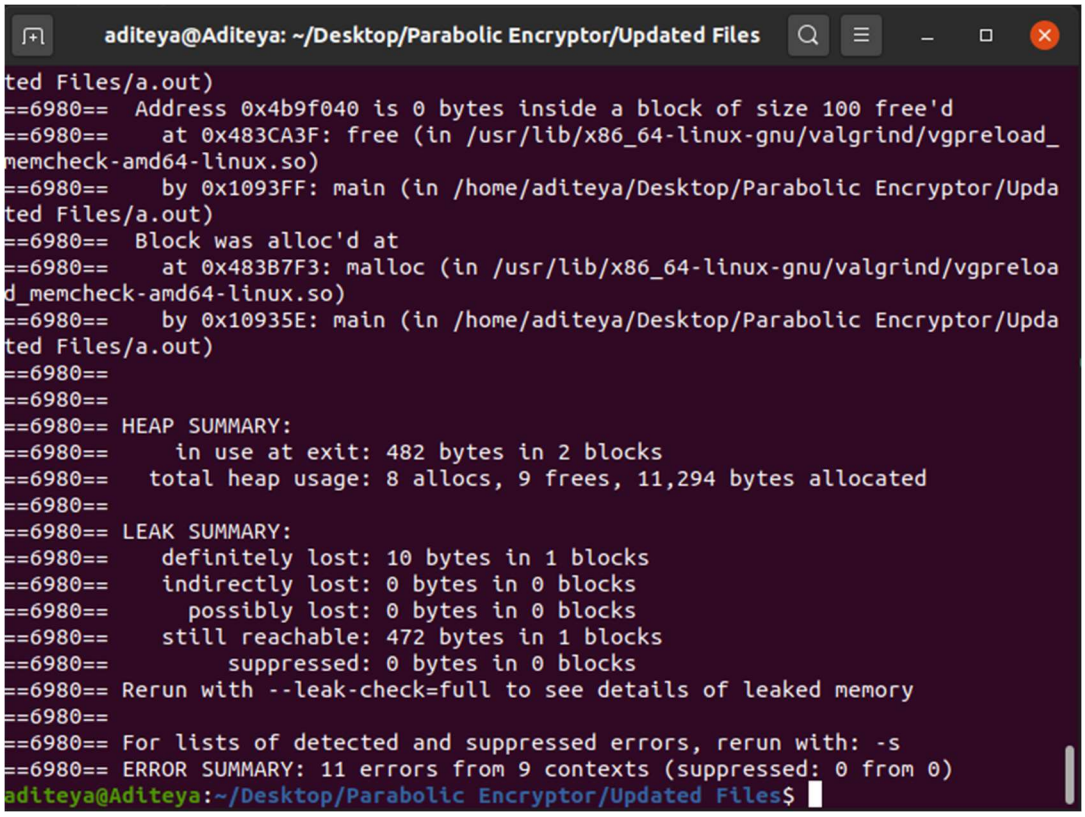
```
free(str);
free(str);

/*
There are multiple free statements for the same pointer. There should only be one free statement.
Also, dynamic array enstr has not been freed. This is going to result in a memory leak.
*/
```

What *splint* shows:

```
Main-vulnerable.c:41:10: Dead storage str passed as out parameter to free: str
  Memory is used after it has been released (either by passing as an only param
  or assigning to an only global). (Use -usereleased to inhibit warning)
   Main-vulnerable.c:40:10: Storage str released
```

```
Main-vulnerable.c:48:14: Fresh storage enstr not released before return
  A memory leak has been detected. Storage allocated locally is not released
  before the last reference to it is lost. (Use -mustfreefresh to inhibit
```

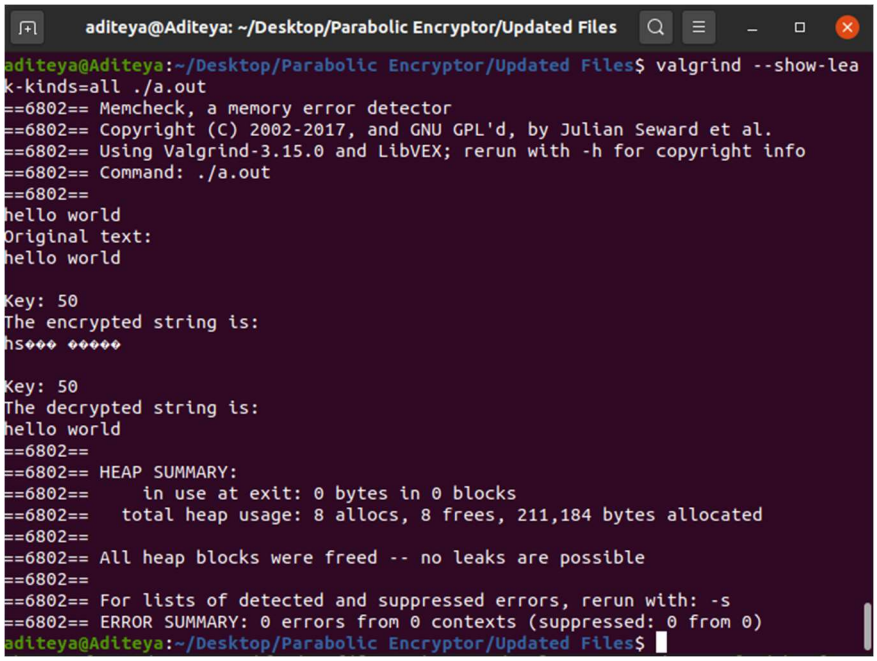
What *valgrind* shows:



Mitigation:

str has been freed only once and enstr has been freed, preventing a memory leak.

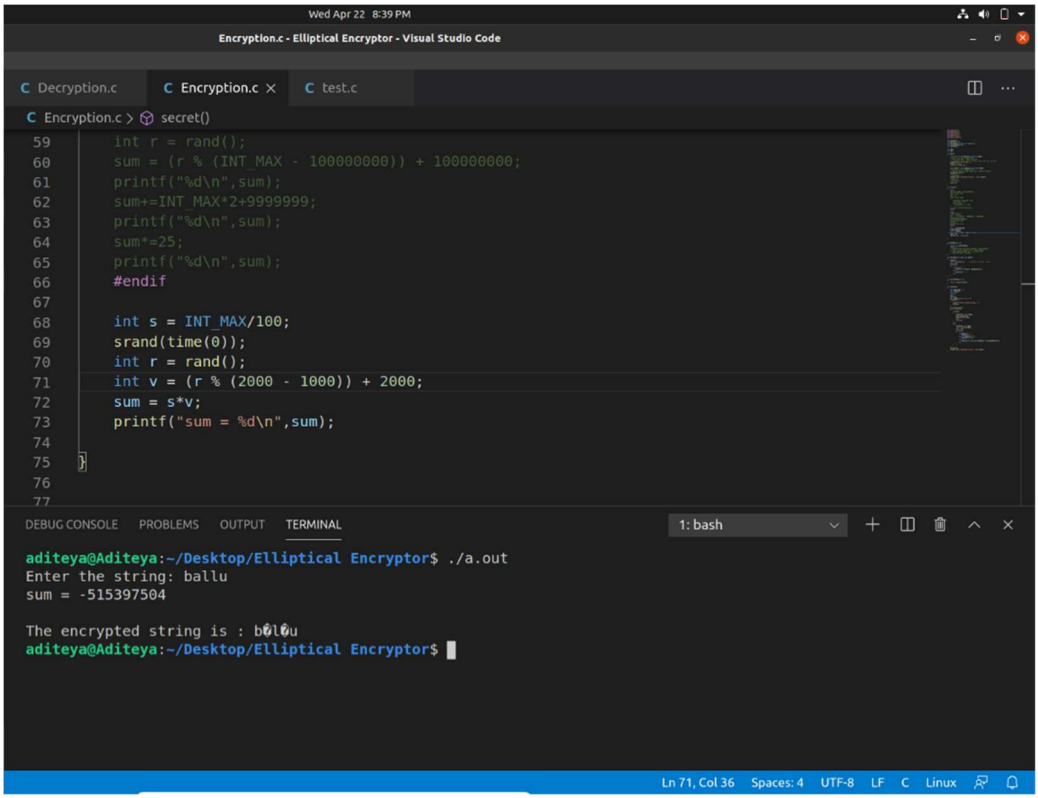
```
free(str);      // str has been freed only once
free(enstr);    // enstr has been freed, preventing a memory leak
```



6. Flawed algorithm used to calculate the value of the variable ‘sum’:

The algorithm in the secret() function is capable of resulting in integer overflow based on the random value, and hence some characters in the text do not get encrypted. The flaw can be shown by including a precondition before the operation s*v takes place below by checking if s*v>INT_MAX or s>INT_MAX/v. The algorithm needs to be redesigned such that no overflow can occur irrespective of the value randomly generated.

```
void secret()
{
    int s = INT_MAX/100;
    srand(time(0));
    int r = rand();
    int v = (r % (2000 - 1000)) + 2000;
    if (s>INT_MAX/v)
        printf("Overflow\n");
    else
        sum = s*v;

    /*
    The algorithm in the secret() function is capable of resulting in integer overflow based on the random value,
    and hence some characters in the text do not get encrypted. The algorithm needs to be redesigned such that no overflow
    can occur irrespective of the value randomly generated.
    */
}
```



Mitigation:

The algorithm in the secret() function has been modified to use the sum of the digits of the time(seconds) elapsed since the first epoch, i.e. 00:00:00 hours, GMT (Greenwich Mean Time), January 1, 1970. (Number will change with each run) instead of a randomly generated value. This prevents the sum variable from taking in a large value and encountering an integer overflow since this sum of digits will never exceed INT_MAX.

```
void secret()
{
    int local_seed = (int)time(NULL);
    seed = local_seed;
    int i = 0;
    sum = 0;
    while (local_seed)
    {
        int temp = local_seed % 10;
        local_seed /= 10;
        sum += temp;
    }

    /*
    The algorithm in the secret() function has been modified to use the sum of the digits of the time(seconds) elapsed since
    00:00:00 hours, GMT (Greenwich Mean Time), January 1, 1970. (Number will change with each run)
    instead of a randomly generated value. This prevents the sum variable from taking in a large value and encountering an integer
    overflow since this sum of digits will never exceed INT_MAX.
    */
}
```

7. Domain error and type mismatch:

This problem carries over from the secret() function. If the value generated for sum is too large, it will result in an overflow, causing it to take a negative value greater than INT_MIN. But this value is extremely small, and the square root of this value will be zero as it is a domain error. Hence the original string will NOT be decrypted since the sum value determines the decryption distance.

sqrt() also returns a double, and we are returning an integer to the calling function since the distance has to be an integer.

```c
int parabola(int c)
{
    double x = sqrt(4*sum*c);
    return x;

    /*
    This problem carries over from the secret() function. If the value generated for sum is too large, it will
    result in an overflow, causing it to take a negative value greater than INT_MIN. But this value is extremely small,
    and the square root of this value will be zero as it's a domain error. Hence the original string will NOT be decrypted since the sum value
    determines the decryption distance.

    sqrt() also returns a double, and we are returning an integer to the calling function since the distance has to
    be an integer
    */
}
```

```c
int inv_parabola(int c)
{
    double x = -1*sqrt(4*sum*c);
    return x;

    /*
    This problem carries over from the secret() function. If the value generated for sum is too large, it will
    result in an overflow, causing it to take a negative value greater than INT_MIN. But this value is extremely small,
    and the square root of this value will be zero as it's a domain error. Hence the original string will NOT be decrypted since the sum value
    determines the decryption distance.

    sqrt() also returns a double, and we are returning an integer to the calling function since the distance has to
    be an integer
    */
}
```

What *splint* shows:

```
Dependencies-vulnerable.c:32:12: Return value type double does not match
                                    declared type int: x
Dependencies-vulnerable.c: (in function encrypt)
```

```
Dependencies-vulnerable.c:68:12: Return value type double does not match
                                    declared type int: x
```

Mitigation:

Since the value generated by sum is always a positive value lesser than INT_MAX, the previous issues have been removed and the sqrt() function no longer returns zero. Hence the encryption distance is always a positive value and all the characters in the original string get encrypted.

```c
int parabola(int c)
{
    double x = sqrt(4*sum*c);
    return (int)x;

    /*
    Since the value generated by sum is always a positive value lesser than INT_MAX, the previous issues have been removed
    and the sqrt() function no longer returns zero. Hence the encryption distance is always a positive value and all the
    characters in the original string get encrypted.
    */
}
```

```c
int inv_parabola(int c)
{
    double x = -1*sqrt(4*sum*c);
    return x;

    /*
    Since the value generated by sum is always a positive value lesser than INT_MAX, the previous issues have been removed
    and the sqrt() function no longer returns zero. Hence the decryption distance is always a positive value and all the
    characters in the original string get decrypted.
    */
}
```

8. Null termination Error:

A '\0' needs to be appended at the end of the encrypted string to mark the end of the string. Without this delimiter, printing the string might result in an overflow.

```c
void encrypt(char *str,char *enstr)
{
    secret();
    int l = strlen(str);    // If the length of str is more than 10^10, it will seg fault here
    int i = 0;
    while(i<l)
    {
        if(str[i]!=' ')
            enstr[i] = (str[i] + parabola(i));
        else
            enstr[i] = ' ';
        i++;
    }

    /*
    A '\0' needs to be appended at the end of the encrypted string to mark the end of the string.
    Without this delimiter, printing the string might result in an overflow.
    */
}
```

Mitigation:

'\0' delimiter has been appended at the end of the encrypted string.

```c
void encrypt(char *str,char *enstr)
{
    secret();
    int l = strlen(str);
    int i = 0;
    while(i<l)
    {
        if(str[i]!=' ')
            enstr[i] = (str[i] + parabola(i));
        else
            enstr[i] = ' ';
        i++;
    }
    enstr[i] = '\0';    // '\0' delimiter has been appended at the end of the encrypted string
}
```

9. File pointer Error (checking the value of fp):

No check is performed to see if the file exists. The later sections of this function will crash if the file is not present. Hence, we need to add in a check to ensure the file exists and the pointer fp did not return NULL.

```c
FILE *fp = fopen("Encrypted.txt","r");

/*
No check is performed to see if the file exists. The later sections of this function will crash if the file is
not present. Hence we need to add in a check to ensure the file exists and the pointer fp did not return NULL.
*/
```

What *splint* shows:

```
Dependencies-vulnerable.c:99:20: Possibly null storage fp passed as non-null
                                   param: fscanf (fp, ...)
  A possibly null pointer is passed as a parameter corresponding to a formal
  parameter with no /*@null@*/ annotation.  If NULL may be used for this
  parameter, add a /*@null@*/ annotation to the function parameter declaration.
```

```
Dependencies-vulnerable.c:107:20: Possibly null storage fp passed as non-null
                                    param: fscanf (fp, ...)
   Dependencies-vulnerable.c:88:16: Storage fp may become null
```

```
Dependencies-vulnerable.c:141:13: Possibly null storage fp passed as non-null
                                    param: fprintf (fp, ...)
   Dependencies-vulnerable.c:140:16: Storage fp may become null
```

Mitigation:

A check has been added to ensure the file exists and terminate if it does not return NULL.

```c
FILE *fp = fopen("Encrypted.txt","r");

if(fp==NULL)        // A check has been added to ensure the file exists and terminate if it does not
{
    printf("File problem,exiting.\n");
    exit(0);
}
```

10. File pointer Error (not closing the file):

fp was not closed after the file reading operation. This must always be done to ensure the buffered data gets flushed.

```c
/*
fp wasn't closed after the file reading operation.
This must always be done to ensure the buffered data gets flushed
*/
```

Mitigation:

fp has been closed after all the file reading operations.

```c
fclose(fp);          // fp has been closed after all the file reading operations
```

11. Freeing statically allocated arrays:

temp and destr are static arrays. We have attempted to free them. This cannot be done.

```c
free(temp);
free(destr);
// temp and destr are static arrays. We have attempted to free them. This cannot be done
```

What *gcc* shows when we compile the code:

```
Dependencies-vulnerable.c: In function ‘decrypt’:
Dependencies-vulnerable.c:133:5: warning: attempt to free a non-heap object ‘temp’ [-Wfree-nonheap-object]
     free(temp);
     ^~~~~~~~~~
Dependencies-vulnerable.c:134:5: warning: attempt to free a non-heap object ‘destr’ [-Wfree-nonheap-object]
     free(destr);
     ^~~~~~~~~~~
```

Mitigation:

The two free statements to static pointers have been removed.

```c
//The two free statements to static pointers have been removed
```

12. Failing to check whether pointers str and enstr are NULL:

If str is NULL, then the program will crash every time we try to access it. Hence, we should check that before using it.

```
// If str is NULL, then the program will crash everytime we try to access it. Hence we should check that before
// using it.
```

If enstr is NULL, then the program will crash every time we try to access it. Hence, we should check that before using it.

```
// If enstr is NULL, then the program will crash everytime we try to access it. Hence we should check that before
// using it.
```

What *splint* shows:

```
Main-vulnerable.c:18:10: Possibly null storage str passed as non-null param:
                           gets (str)
  A possibly null pointer is passed as a parameter corresponding to a formal
  parameter with no /*@null@*/ annotation.  If NULL may be used for this
  parameter, add a /*@null@*/ annotation to the function parameter declaration.
  (Use -nullpass to inhibit warning)
   Main-vulnerable.c:6:17: Storage str may become null
Main-vulnerable.c:33:17: Possibly null storage enstr passed as non-null param:
                           encrypt (..., enstr)
   Main-vulnerable.c:23:19: Storage enstr may become null
Main-vulnerable.c:33:17: Passed storage enstr not completely defined (*enstr is
                           undefined): encrypt (..., enstr)
  Storage derivable from a parameter, return value or global is not defined.
  Use /*@out@*/ to denote passed or returned storage which need not be defined.
  (Use -compdef to inhibit warning)
```

Mitigation:

The program will only be executed further if enstr and str both are not NULL here, else it will terminate execution.

```
if(str==NULL)
{
    printf("\nMemory allocation error, exiting.\n");
    exit(0);
}
// The program will only be executed further if str is not NULL here, else it will terminate execution.
```

```
if(enstr==NULL)
{
    printf("\nMemory allocation error, exiting.\n");
    exit(0);
}
// The program will only be executed further if enstr is not NULL here, else it will terminate execution.
```