Karthikeyan G
CB.SC.P2CYS24008

Lab - 10
Rule 07. Characters and Strings (STR)

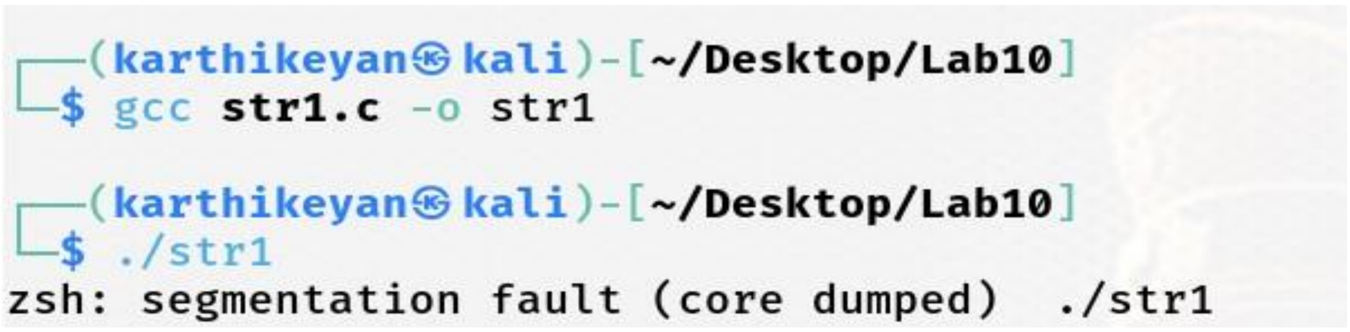
STR30-C. Do not attempt to modify string literals

- String literals are created as static arrays of characters with a terminating null character (\0) at compile time. They have static storage duration, meaning they persist for the lifetime of the program.
- String literals are usually accessed via pointers or arrays of characters. They should ideally be assigned to pointers of const char or const wchar_t to indicate they are read-only.
- It is unspecified whether identical string literals in different parts of a program share memory or are stored separately. Modifying a string literal results in undefined behavior and often causes access violations because they are typically stored in read-only memory.
- Do not assign string literals to non-const pointers or cast them to non-const types. A pointer to (or an array of) const characters must be treated as a string literal to ensure it is read-only.
- The following functions return pointers to characters within the string: **strpbrk(), strchr(), strrchr(), strstr(),** and their wide character equivalents: **wcspbrk(), wcschr(), wcsrchr(), wcsstr(),** along with memory functions like memchr() and **wmemchr()**. If the first argument is a string literal, the returned pointer must also be treated as pointing to a constant, read-only area.
- This rule enforces the broader guideline EXP40-C, which states: Do not modify constant objects. Treating string literals as read-only ensures program stability and prevents runtime errors.

Noncompliant Code Example

In this noncompliant code example, the char pointer str is initialized to the address of a string literal. Attempting to modify the string literal is undefined behavior:

```
#include<stdio.h>
int main()
{
	char *str  = "string literal";
	str[0] = 'S';
	printf("%s",str);
	return 0;
}
```

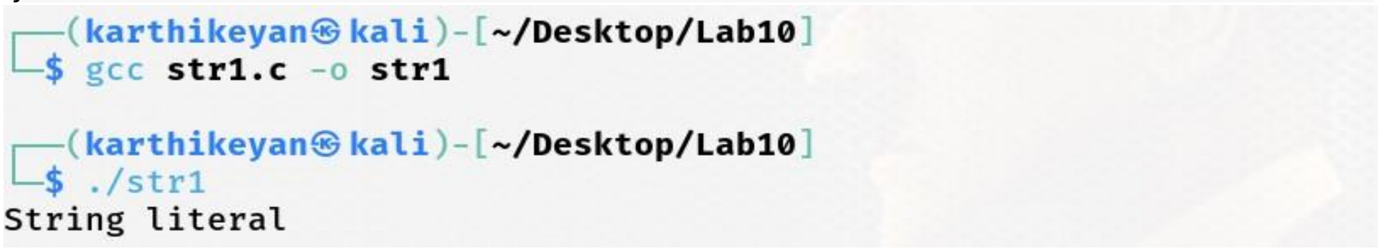


```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str1.c -o str1

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str1
zsh: segmentation fault (core dumped)  ./str1
```

Compliant Solution

As an array initializer, a string literal specifies the initial values of characters in an array as well as the size of the array. (See STR11-C. Do not specify the bound of a character array initialized with a string literal.) This code creates a copy of the string literal in the space allocated to the character array str. The string stored in str can be modified safely.

```
#include<stdio.h>
int main()
{
	char str[]  = "string literal";
	str[0] = 'S';
	printf("%s\n",str);
	return 0;
}
```

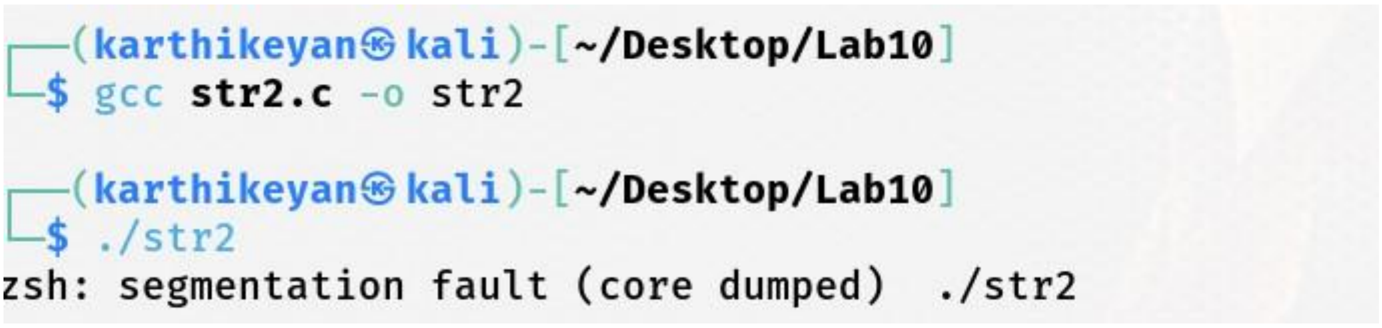


```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str1.c -o str1

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str1
String literal
```

Noncompliant Code Example (POSIX)

In this noncompliant code example, a string literal is passed to the (pointer to non-const) parameter of the POSIX function mkstemp(), which then modifies the characters of the string literal:

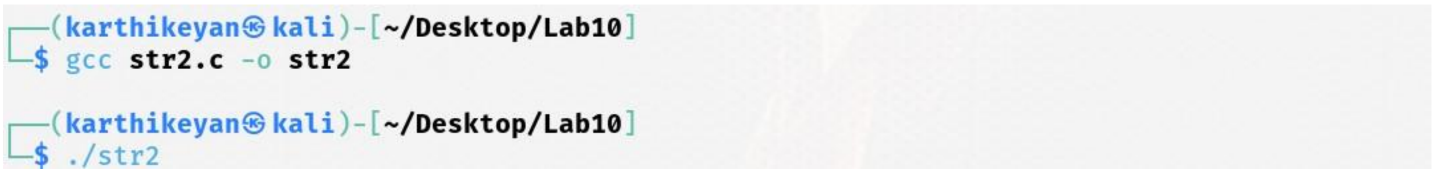
```
#include<stdio.h>
#include <stdlib.h>

void func(void)
{
	mkstemp("/tmp/edXXXXXX");
}
int main()
{
	func();
	return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str2.c -o str2

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str2
zsh: segmentation fault (core dumped)  ./str2
```

Compliant Solution (POSIX)

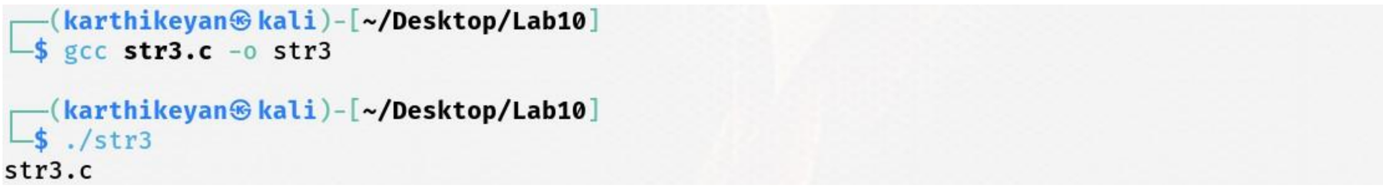
```
#include<stdio.h>
#include <stdlib.h>

void func(void)
{
  static char fname[] = "/tmp/edXXXXXX";
  mkstemp(fname);
}

int main()
{
   func();
   return 0;
}
```



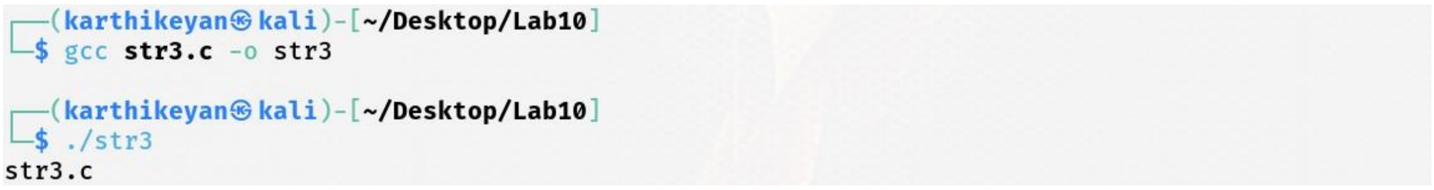
```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str2.c -o str2

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str2
```

Noncompliant Code Example (Result of strrchr())

In this noncompliant example, the char * result of the strrchr() function is used to modify the object pointed to by pathname. Because the argument to strrchr() points to a string literal, the effects of the modification are undefined.

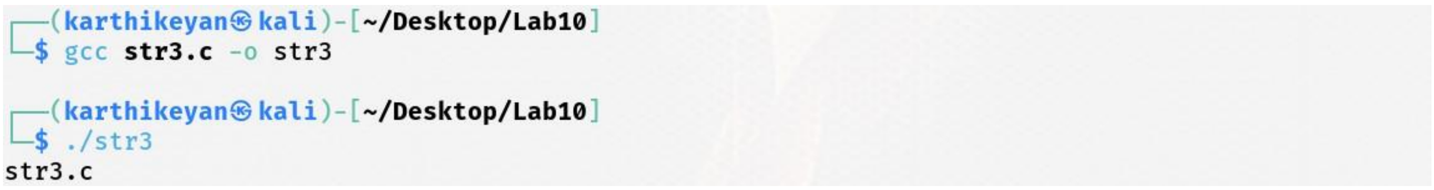


```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str3.c -o str3

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str3
str3.c
```



```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str3.c -o str3

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str3
str3.c
```

```c
#include <stdio.h>
#include <string.h>
const char *get_dirname(const char *pathname) {
  char *slash;
  slash = strrchr(pathname, '/');
  if (slash) {
    *slash = '\0'; /* Undefined behavior */
  }
  return pathname;
}

int main(void) {
  puts(get_dirname(__FILE__));
  return 0;
```



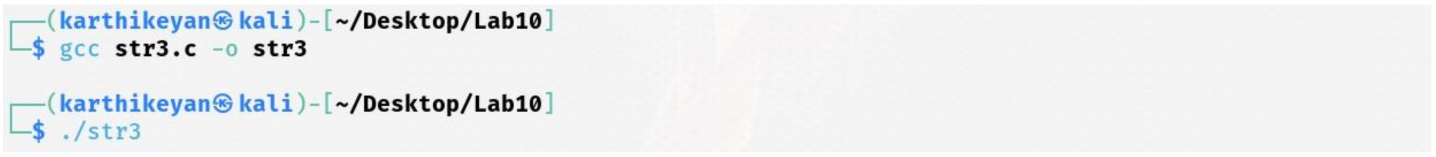
```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str3.c -o str3

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str3
str3.c
```

Compliant Solution (Result of strrchr())

This compliant solution avoids modifying a const object, even if it is possible to obtain a non-const pointer to such an object by calling a standard C library function, such as strrchr(). To reduce the risk to callers of get_dirname(), a buffer and length for the directory name are passed into the function. It is insufficient to change pathname to require a char * instead of a const char

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

char *get_dirname(const char *pathname, char *dirname, size_t size) {
  const char *slash;
  slash = strrchr(pathname, '/');
  if (slash) {
    ptrdiff_t slash_idx = slash - pathname;
    if ((size_t)slash_idx < size) { memcpy(dirname,
      pathname, slash_idx); dirname[slash_idx] =
      '\0';
      return dirname;
    }
  }
  return 0;
}

int main(void) {
  char dirname[260];
  if (get_dirname(_FILE_, dirname, sizeof(dirname))) {
    puts(dirname);
  }
  return 0;
}
```

* because conforming compilers are not required to diagnose passing a string literal to a function accepting a char *.



```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str3.c -o str3

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str3
```

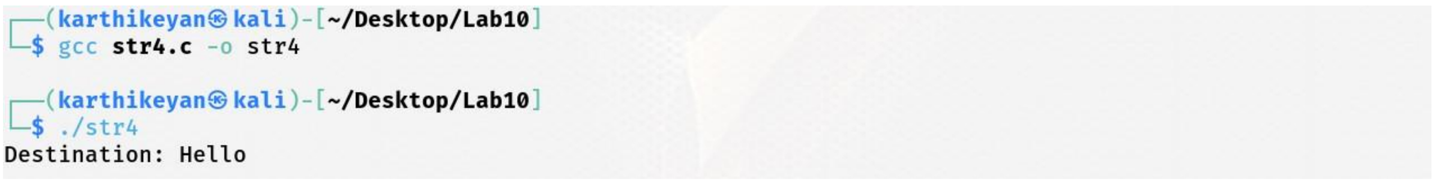
STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator

- Copying data to a buffer that is not large enough to hold the data results in a buffer overflow, which frequently occurs when manipulating strings.
- To prevent buffer overflows, either limit copies through truncation or, preferably, ensure that the destination buffer is sufficiently sized to accommodate the character data and the null-termination character.
- When strings are allocated on the heap, this rule relates to MEM35-C, which emphasizes allocating sufficient memory for an object.
- Since strings are represented as arrays of characters, this rule also connects to ARR30-C, which advises against forming or using out-of-bounds pointers or array subscripts, and ARR38-C, which ensures that library functions do not create invalid pointers.

Noncompliant Code Example (Off-by-One Error)

This noncompliant code example demonstrates an off-by-one error. The loop copies data from src to dest. However, because the loop does not account for the null-termination character, it may be incorrectly written 1 byte past the end of dest.

```c
#include <stddef.h>
#include <stdio.h>

void copy(size_t n, char src[n], char dest[n]) {
   size_t i;

   for (i = 0; src[i] && (i < n); ++i) {
     dest[i] = src[i];
   }
   dest[i] = '\0';
   printf("source : %s\n", src);
   printf("destination : %s\n", dest);
}

int main()
{
    char source[] = "Hello, World!. ......";
    char destination[12];
    size_t size= sizeof(source);
    copy(size, source, destination);
    return 0;
}
```

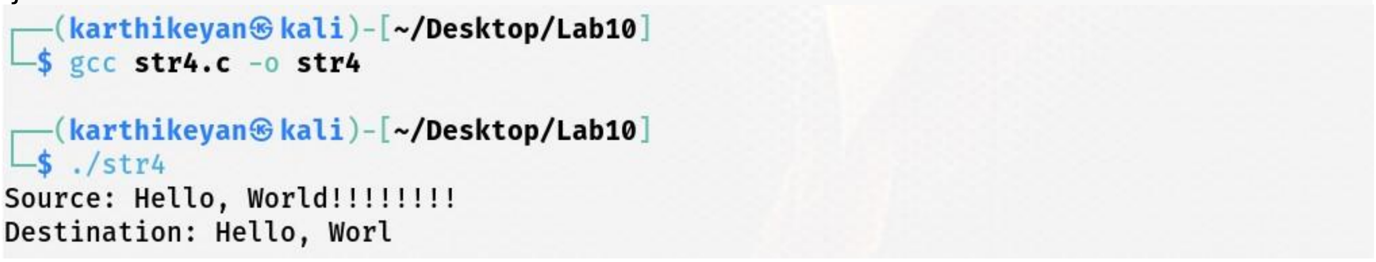


```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str4.c -o str4

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str4
Destination: Hello
```

Compliant Solution (Off-by-One Error)

In this compliant solution, the loop termination condition is modified to account for the null-termination character that is appended to dest:

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

void copy(size_t n, char src[n], char dest[n]) {
  size_t i;

  for (i = 0; src[i] && (i < n - 1); ++i) {
    dest[i] = src[i];
  }
  dest[i] = '\0';
  printf("source : %s\n", src);
  printf("destination : %s\n", dest);
}

int main()
{
   char source[] = "Hello, World!!!!!...";
   char destination[12];
   size_t size= sizeof(source);
   copy(size, source, destination);
   return 0;
}
```



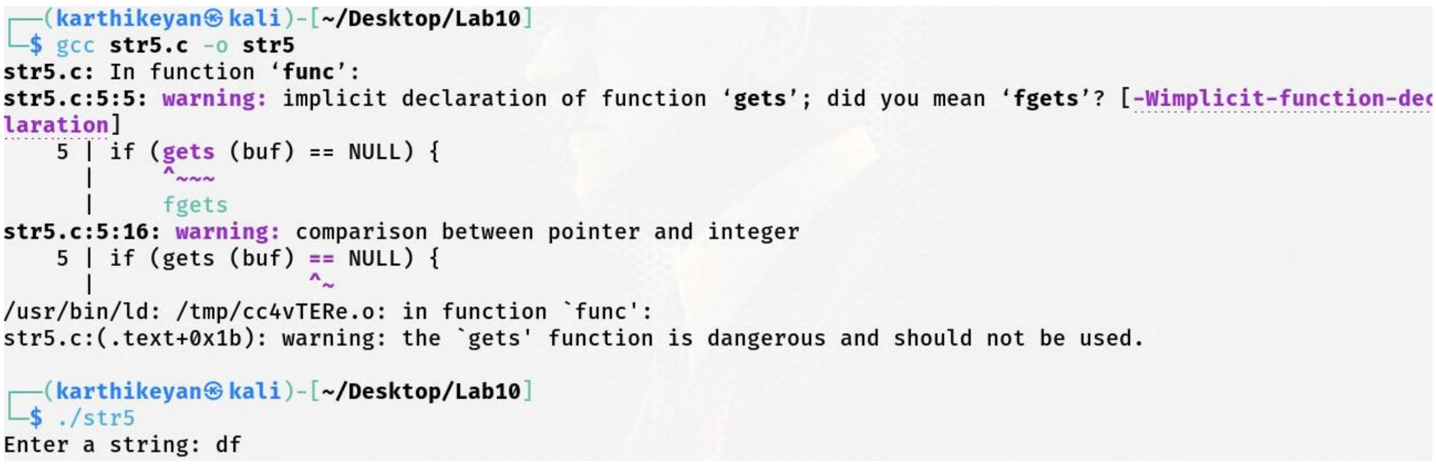
```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str4.c -o str4

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str4
Source: Hello, World!!!!!!!!
Destination: Hello, Worl
```

Noncompliant Code Example (gets())

The gets() function, which was deprecated in the C99 Technical Corrigendum 3 and removed from C11, is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from stdin. This noncompliant code example assumes that gets() will not read more than BUFFER_SIZE - 1 characters from stdin. This is an invalid assumption, and the resulting operation can result in a buffer overflow.

The gets() function reads characters from the stdin into a destination array until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array.

```c
#include <stdio.h>
#define BUFFER_SIZE 1024

void func(void) {
  char buf[BUFFER_SIZE];
  if (gets(buf) == NULL) {
    /* Handle error */
    printf("Error reading input\n");
    return;
  }
}
int main()
{
    printf("Enter a string: ");
    func();
    return 0;
}
```

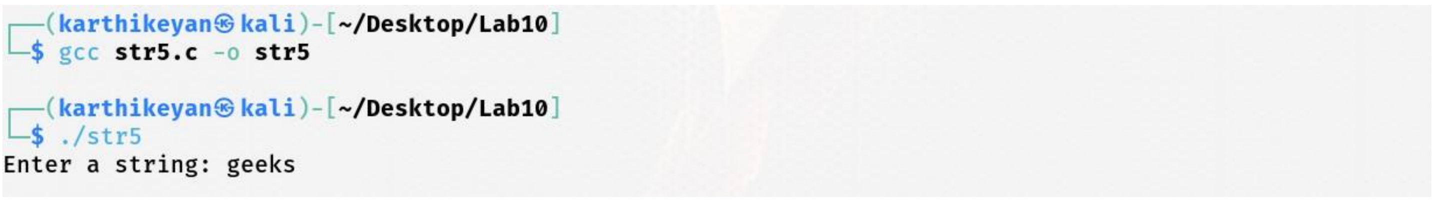


```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str5.c -o str5
str5.c: In function 'func':
str5.c:5:5: warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-dec
laration]
    5 | if (gets (buf) == NULL) {
      |     ^~~~
      |     fgets
str5.c:5:16: warning: comparison between pointer and integer
    5 | if (gets (buf) == NULL) {
      |                ^~
/usr/bin/ld: /tmp/cc4vTERe.o: in function `func':
str5.c:(.text+0x1b): warning: the `gets' function is dangerous and should not be used.

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str5
Enter a string: df
```

Compliant Solution (fgets())

The fgets() function reads, at most, one less than the specified number of characters from a stream into an array. This solution is compliant because the number of characters copied from stdin to buf cannot exceed the allocated memory:

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

enum { BUFFERSIZE = 32 };

void func(void) {
  char buf[BUFFERSIZE];
  int ch;

  if (fgets(buf, sizeof(buf), stdin)) {
    /* fgets() succeeded; scan for newline character */
    char *p = strchr(buf, '\n');
    if (p) {
      *p = '\0';
    } else {
      /* Newline not found; flush stdin to end of line */
      while ((ch = getchar()) != '\n' && ch != EOF)
        ;
      if (ch == EOF && !feof(stdin) && !ferror(stdin)) {
          /* Character resembles EOF; handle error */
      }
    }
  } else {
    /* fgets() failed; handle error */
    printf("Error reading input\n");
  }
}
int main()
{
    printf("Enter a string: ");
    func();
    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str5.c -o str5

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str5
Enter a string: geeks
```

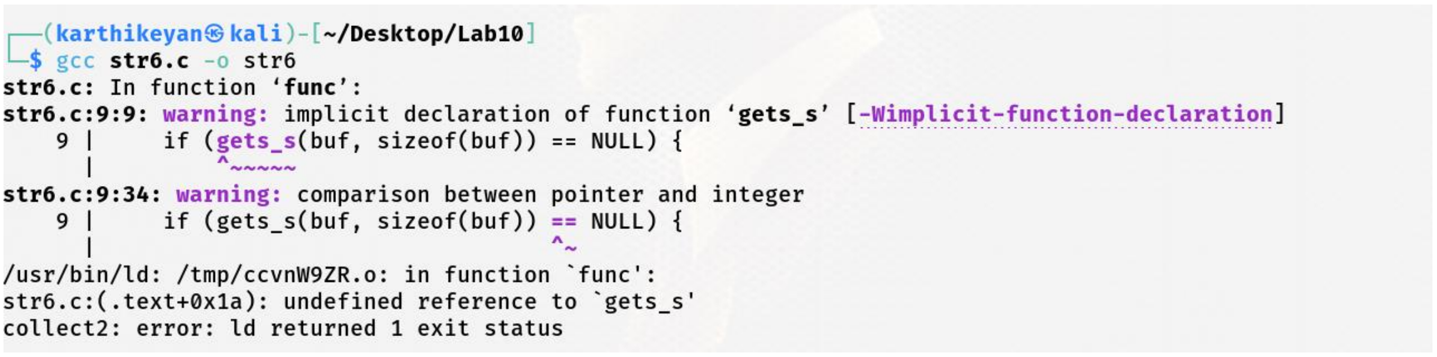
The fgets() function is not a strict replacement for the gets() function because fgets() retains the newline character (if read) and may also return a partial line. It is possible to use fgets() to safely process input lines too long to store in the destination array, but this is not recommended for performance reasons. Consider using one of the following compliant solutions when replacing gets().

Compliant Solution (gets_s())

The gets_s() function reads, at most, one less than the number of characters specified from the stream pointed to by stdin into an array.

If end-of-file is encountered and no characters have been read into the destination array, or if a read error occurs during the operation, then the first character in the destination array is set to the null character and the other elements of the array take unspecified values:

```
#define __STDC_WANT_LIB_EXT1__ 1
#include <stdio.h>

enum { BUFFERSIZE = 32 };

void func(void) {
  char buf[BUFFERSIZE];

  if (gets_s(buf, sizeof(buf)) == NULL) {
    /* Handle error */
     printf("Error reading input\n");
     return;
  }
}
int main()
{
    printf("Enter a string: ");
    func();
    return 0;
}
```



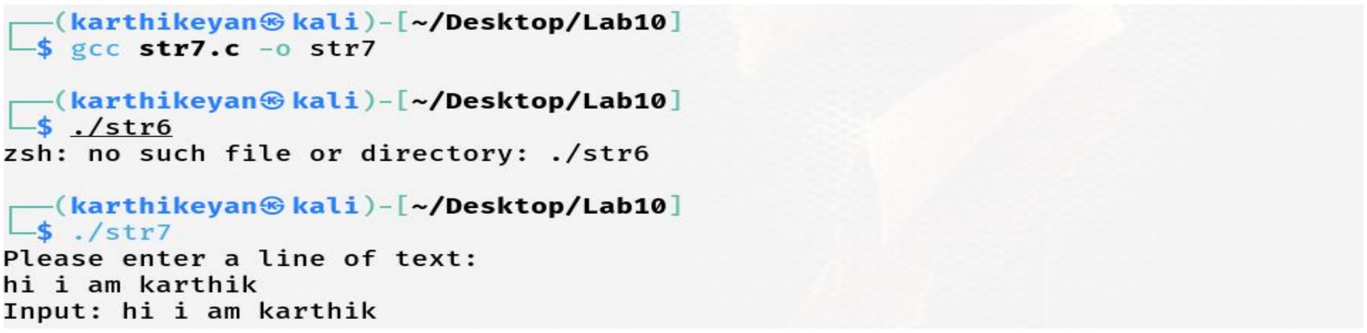
```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str6.c -o str6
str6.c: In function 'func':
str6.c:9:9: warning: implicit declaration of function 'gets_s' [-Wimplicit-function-declaration]
    9 |     if (gets_s(buf, sizeof(buf)) == NULL) {
      |         ^~~~~~
str6.c:9:34: warning: comparison between pointer and integer
    9 |     if (gets_s(buf, sizeof(buf)) == NULL) {
      |                                  ^~
/usr/bin/ld: /tmp/ccvnW9ZR.o: in function `func':
str6.c:(.text+0x1a): undefined reference to `gets_s'
collect2: error: ld returned 1 exit status
```

Compliant Solution (getline(), POSIX)

The getline() function is similar to the fgets() function but can dynamically allocate memory for the input buffer. If passed a null pointer, getline() dynamically allocates a buffer of sufficient size to hold the input. If passed a pointer to dynamically allocated storage that is too small to hold the contents of the string, the getline() function resizes the buffer, using realloc(), rather than truncating the input. If successful, the getline() function returns the number of characters read, which can be used to determine if the input has any null characters before the newline. The getline() function works only with dynamically allocated buffers. Allocated memory must be explicitly deallocated by the caller to avoid memory leaks.

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

void func(void) {
   int ch;
   size_t buffer_size = 32;
   char *buffer = malloc(buffer_size);

   if (!buffer) {
      fprintf(stderr, "Memory allocation failed\n");
      return;
   }

   // Read a line from stdin
   ssize_t size = getline(&buffer, &buffer_size, stdin);

   if (size == -1) {
      fprintf(stderr, "Error reading line\n");
      free(buffer); // Free the buffer before returning
      return;
   }
```

```
    // Remove newline character if present
    char *p = strchr(buffer, '\n');
    if (p) {
        *p = '\0';  // Replace newline with null terminator
    } else {
        // Newline not found; flush stdin to end of line
        while ((ch = getchar()) != '\n' && ch != EOF);
        if (ch == EOF && !feof(stdin) && !ferror(stdin)) {
            fprintf(stderr, "Unexpected EOF encountered\n");
        }
    }

    // Output the processed string
    printf("Input: %s\n", buffer);

    // Free allocated memory
    free(buffer);
}

int main() {
    printf("Please enter a line of text:\n");
    func();
    return 0;
}
```

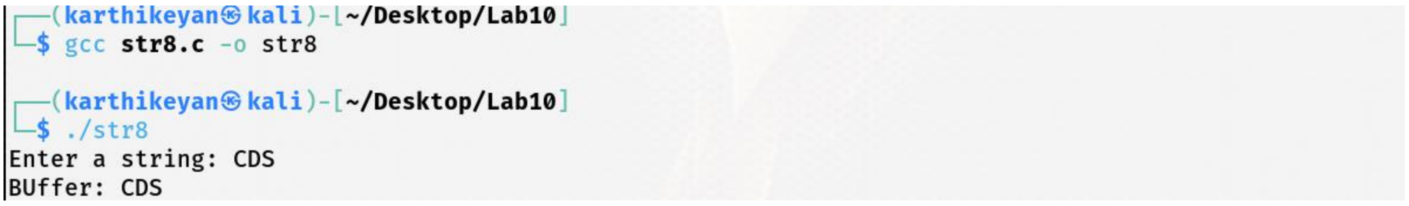


```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str7.c -o str7

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str6
zsh: no such file or directory: ./str6

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str7
Please enter a line of text:
hi i am karthik
Input: hi i am karthik
```

Noncompliant Code Example (getchar())

Reading one character at a time provides more flexibility in controlling behavior, though with additional performance overhead. This noncompliant code example uses the getchar() function to read one character at a time from stdin instead of reading the entire line at once. The stdin stream is read until end-of-file is encountered or a newline character is read. Any newline character is discarded, and a null character is written immediately after the last character read into the array. Similar to the noncompliant code example that invokes gets(), there are no guarantees that this code will not result in a buffer overflow.

```c
#include <stdio.h>
enum { BUFFERSIZE = 32 };

void func(void) {

  char buf[BUFFERSIZE];
  char *p;
  int ch;
  p = buf;
  while ((ch = getchar()) != '\n' && ch != EOF) {
    *p++ = (char)ch;
  }
  *p++ = 0;
  if (ch == EOF) {
     /* Handle EOF or error */
     printf("EOF Error\n");
  }
  else printf("BUffer: %s\n",buf);
}
int main()
{
    printf("Enter a string: ");
    func();
    return 0;
}
```



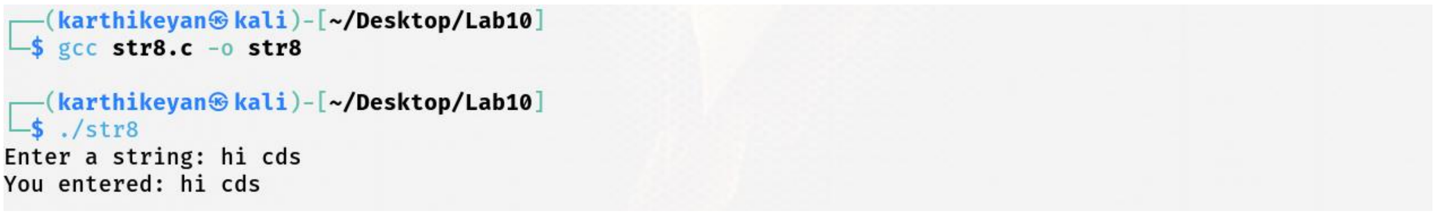
```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str8.c -o str8

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str8
Enter a string: CDS
BUffer: CDS
```

After the loop ends, if ch == EOF, the loop has read through to the end of the stream without encountering a newline character, or a read error occurred before the loop encountered a newline character. To conform to FIO34-C. Distinguish between characters read from a file and EOF or WEOF, the error-handling code must verify that an end-of-file or error has occurred by calling feof() or ferror().

Compliant Solution (getchar())

In this compliant solution, characters are no longer copied to buf once index == BUFFERSIZE - 1, leaving room to null-terminate the string. The loop continues to read characters until the end of the line, the end of the file, or an error is encountered. When truncated == true, the input string has been truncated.

```c
#include <stdio.h>
#include <stdbool.h>

enum { BUFFERSIZE = 32 };

void func(void) {
  char buf[BUFFERSIZE];
    int ch;
    size_t index = 0;
    bool truncated = false;

    while ((ch = getchar()) != '\n' && ch != EOF)
    {
      if (index < sizeof(buf) - 1) {
        buf[index++] = (char)ch;
      }
      else {
        truncated = true;
      }
    }
    buf[index] = '\0';  /* Terminate string */
    if (ch == EOF) {
      /* Handle EOF or error */
      printf("EOF Error\n");
    }

    if (truncated) {
            /* Handle truncation */
            printf("Input was truncated to fit buffer.\n");
    }
    else { printf("You entered: %s\n", buf);}
  }
  int main()
  {
      printf("Enter a string: ");
      func();
      return 0;
  }
```

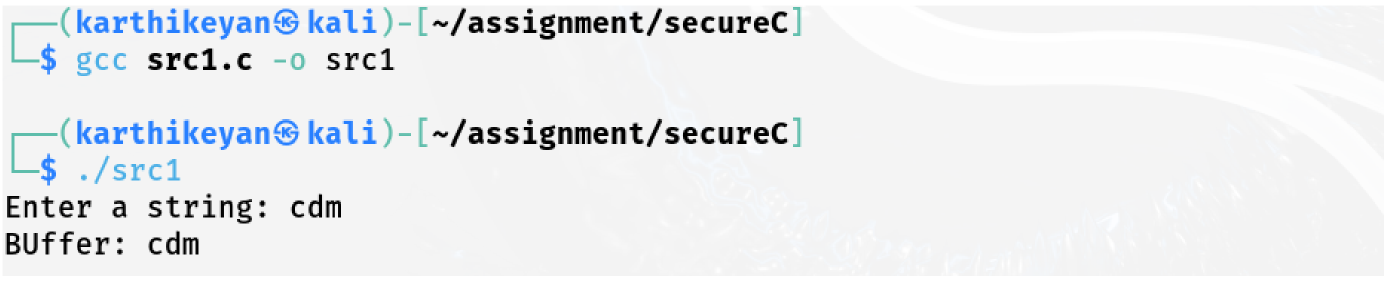


```
┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ gcc str8.c -o str8

┌──(karthikeyan㉿kali)-[~/Desktop/Lab10]
└─$ ./str8
Enter a string: hi cds
You entered: hi cds
```

Noncompliant Code Example (fscanf())

In this noncompliant example, the call to fscanf() can result in a write outside the character array buf:

```
#include <stdio.h>
#include <stdbool.h>
enum { BUF_LENGTH = 1024 };
void get_data(void) {
  char buf[BUF_LENGTH];
  if (1 != fscanf(stdin, "%s", buf)) {
    /* Handle error */
    printf("ERROR READING INPUT\n");
    return;
  }
  printf("BUffer: %s\n", buf);
}
int main()
{
    printf("Enter a string: ");
    get_data();
    return 0;
}
```

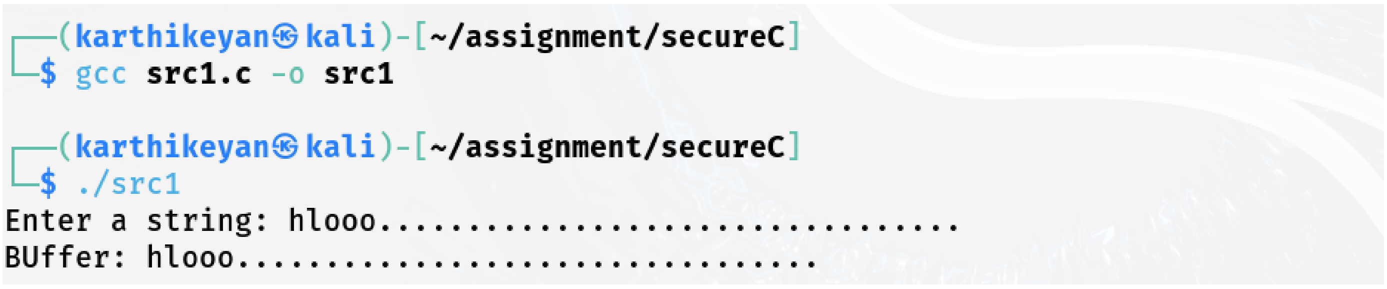


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src1.c -o src1

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src1
Enter a string: cdm
BUffer: cdm
```

Compliant Solution (fscanf())

In this compliant solution, the call to fscanf() is constrained not to overflow buf:

```c
#include <stdio.h>
#include <stdbool.h>

enum { BUF_LENGTH = 1024 };

void get_data(void) {
  char buf[BUF_LENGTH];
  if (1 != fscanf(stdin, "%1023s", buf)) {
    /* Handle error */
    printf("ERROR READING INPUT\n");
    return;
  }
  printf("BUffer: %s\n", buf);
}
int main()
{
    printf("Enter a string: ");
    get_data();
    return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src1.c -o src1

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src1
Enter a string: hlooo.................................
BUffer: hlooo................................
```

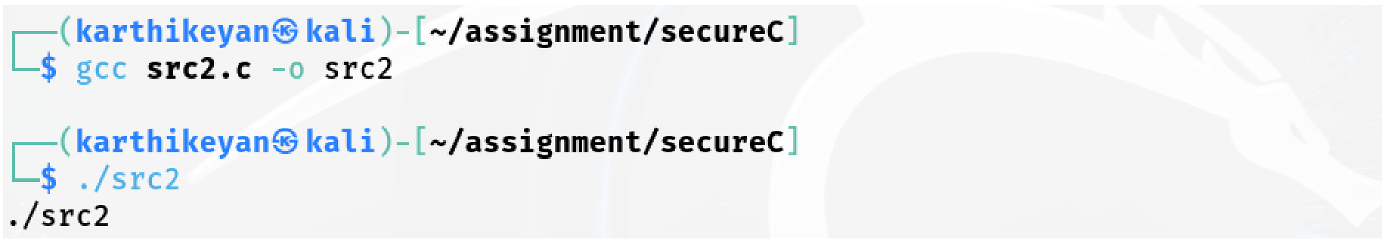
Noncompliant Code Example (argv)

In a hosted environment, arguments read from the command line are stored in process memory. The function main(), called at program startup, is typically declared as follows when the program accepts command-line arguments:

Command-line arguments are passed to main() as pointers to strings in the array members argv[0] through argv[argc - 1]. If the value of argc is greater than 0, the string pointed to by argv[0] is, by convention, the program name. If the value of argc is greater than 1, the strings referenced by argv[1] through argv[argc - 1] are the program arguments.

Vulnerabilities can occur when inadequate space is allocated to copy a command-line argument or other program input. In this noncompliant code example, an attacker can manipulate the contents of argv[0] to cause a buffer overflow:

```
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  /* Ensure argv[0] is not null */
  const char *const name = (argc && argv[0]) ? argv[0] : "";
  char prog_name[128];
  strcpy(prog_name, name);
  printf("%s\n",prog_name);
  return 0;
}
```

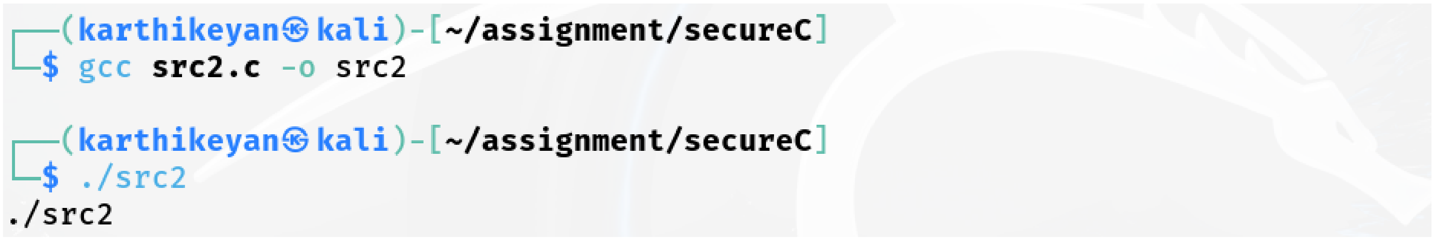


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src2.c -o src2

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src2
./src2
```

Compliant Solution (argv)

The strlen() function can be used to determine the length of the strings referenced by argv[0] through argv[argc - 1] so that adequate memory can be dynamically allocated.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
  /* Ensure argv[0] is not null */
  const char *const name = (argc && argv[0]) ? argv[0] : "";
  char *prog_name = (char *)malloc(strlen(name) + 1);
  if (prog_name != NULL) {
    strcpy(prog_name, name);
  } else {
     printf("ERROR copying prog_name\n");
     free(prog_name);
     return 0;
  }
  printf("%s\n",prog_name);
  free(prog_name);
  return 0;
}
```

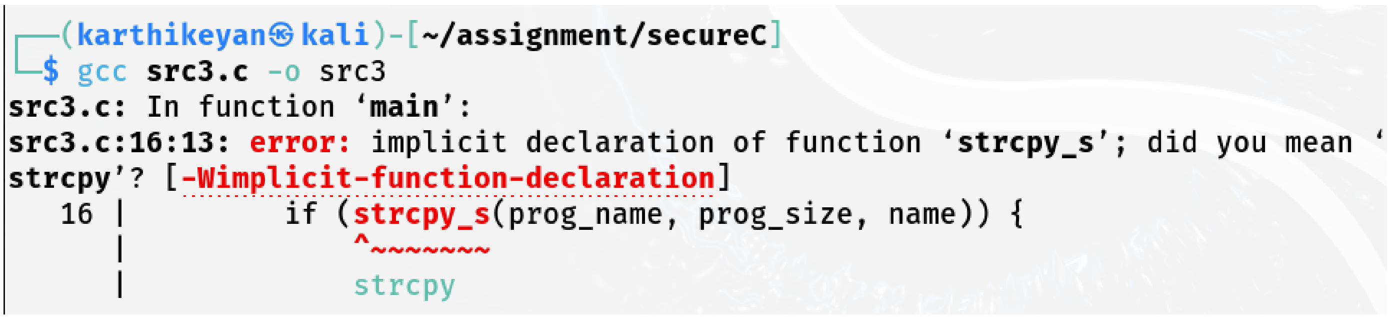


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src2.c -o src2

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src2
./src2
```

Compliant Solution (argv)

The strcpy_s() function provides additional safeguards, including accepting the size of the destination buffer as an additional argument.

```c
#define __STDC_WANT_LIB_EXT1__ 1
#include  <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {
  /* Ensure argv[0] is not null */
  const char *const name = (argc && argv[0]) ? argv[0] : "";
  char *prog_name;
  size_t prog_size;

  prog_size = strlen(name) + 1;
  prog_name = (char *)malloc(prog_size);

  if (prog_name != NULL) {
    if (strcpy_s(prog_name, prog_size, name)) {
      printf("Copy success!");
    }
  } else {
     printf("ERROR copying prog_name\n");
     return 0;
  }
   printf("%s\n",prog_name);
  free(prog_name);
  return 0;
}
```



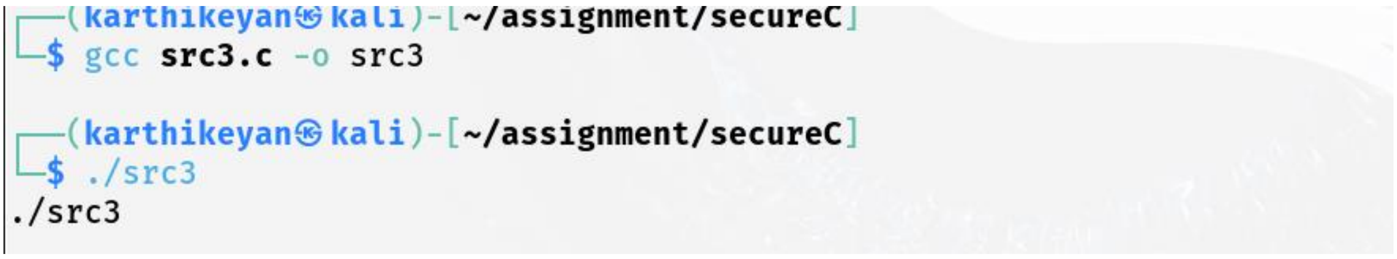
```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src3.c -o src3
src3.c: In function ‘main’:
src3.c:16:13: error: implicit declaration of function ‘strcpy_s’; did you mean ‘
strcpy’? [-Wimplicit-function-declaration]
   16 |         if (strcpy_s(prog_name, prog_size, name)) {
      |             ^~~~~~~~
      |             strcpy
```

The strcpy_s() function can be used to copy data to or from dynamically allocated memory or a statically allocated array. If insufficient space is available, strcpy_s() returns an error.

Non-Compliant Solution (argv)

If an argument will not be modified or concatenated, there is no reason to make a copy of the string. Not copying a string is the best way to prevent a buffer overflow and is also the most efficient solution. Care must be taken to avoid assuming that argv[0] is non-null.

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
/* Ensure argv[0] is not null */
const char * const prog_name = (argc && argv[0]) ? argv[0]: "";
printf("%s\n", prog_name);
return 0;
}
```

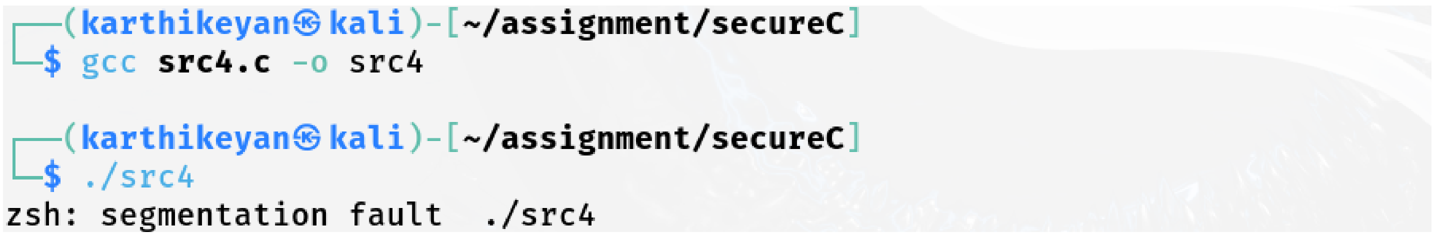


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src3.c -o src3

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src3
./src3
```

Noncompliant Code Example (getenv())

Environment variables can be arbitrarily large, and copying them into fixed-length arrays without first determining the size and allocating adequate storage can result in a buffer overflow.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
void func(void) {
char buff [256];
char *editor = getenv("EDITOR");
if (editor = NULL) {
/* EDITOR environment variable not set */
printf("EDITOR environment variable not set.\n");
} else {
strcpy(buff, editor);
printf("EDITOR is set to: %s\n", buff);
}
}
int main() {
func(); // Call the func() function
return 0;
}
```

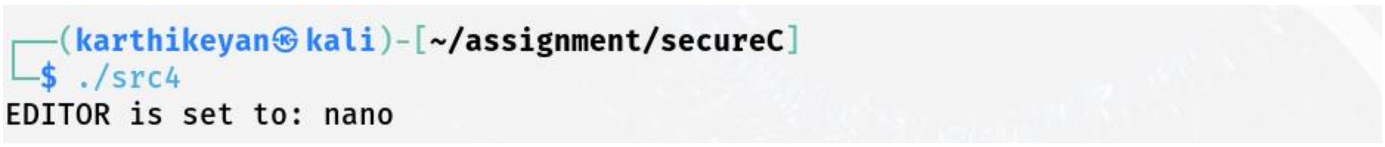


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src4.c -o src4

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src4
zsh: segmentation fault  ./src4
```

Compliant Solution (getenv())

Environmental variables are loaded into process memory when the program is loaded. As a result, the length of these strings can be determined by calling the strlen() function, and the resulting length can be used to allocate adequate dynamic memory:

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>  // For printf

void func(void) {
  char *buff;
  char *editor = getenv("EDITOR");

  if (editor == NULL) {
    /* EDITOR environment variable not set */
    printf("EDITOR environment variable not set.\n");
  } else {
    size_t len = strlen(editor) + 1;
    buff = (char *)malloc(len);
    if (buff == NULL) {
      /* Handle memory allocation error */
      printf("Memory allocation failed.\n");
      return;
    }
    memcpy(buff, editor, len);
    printf("EDITOR is set to: %s\n", buff);
    free(buff);
  }
}
int main(void) {
  func();
  return 0;
}
```

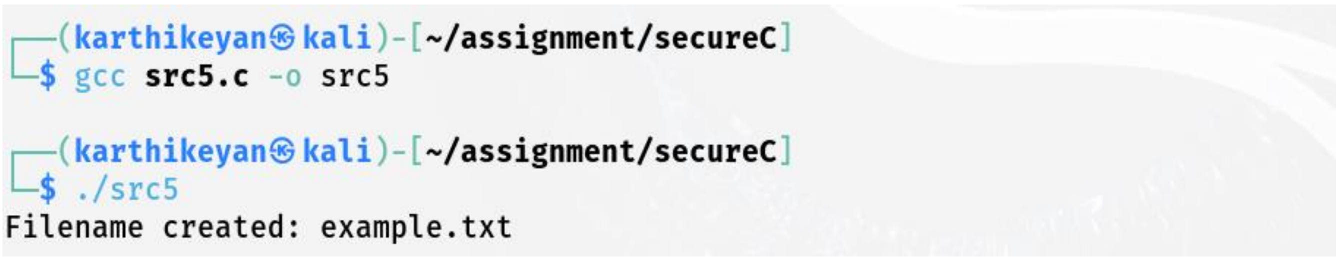


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src4
EDITOR is set to: nano
```

Noncompliant Code Example (sprintf())

In this noncompliant code example, name refers to an external string; it could have originated from user input, the file system, or the network. The program constructs a file name from the string in preparation for opening the file.

```
#include <stdio.h>
void func(const char *name) {
  char filename[128];
  sprintf(filename, "%s.txt", name);
  printf("Filename created: %s\n", filename); // Print the generated filenam
}
int main() {
  const char *name = "example"; // Sample name
  func(name); // Call the func() function with the sample name
  return 0;
}
```



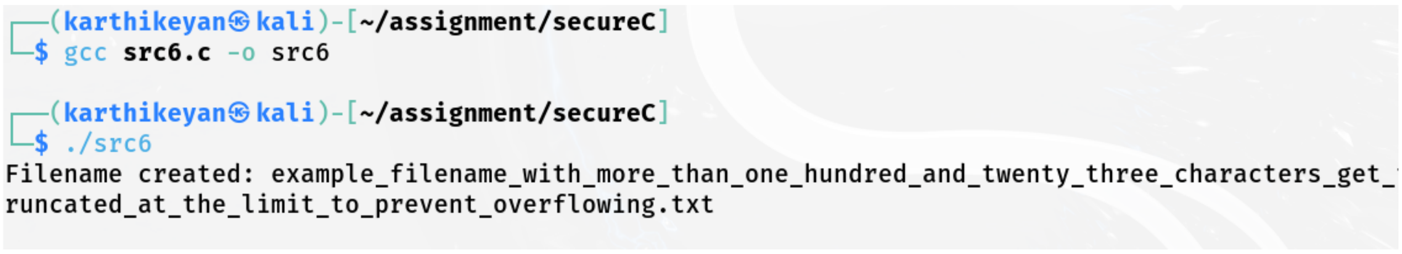
```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src5.c -o src5

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src5
Filename created: example.txt
```

Because the sprintf() function makes no guarantees regarding the length of the generated string, a sufficiently long string in name could generate a buffer overflow.

Compliant Solution (sprintf())

The buffer overflow in the preceding noncompliant example can be prevented by adding precision to the %s conversion specification. If the precision is specified, no more than that many bytes are written. The precision 123 in this compliant solution ensures that filename can contain the first 123 characters of name, the .txt extension, and the null terminator.

```c
#include <stdio.h>
void func(const char *name) {
  char filename [128];
  sprintf(filename, "%.123s.txt", name);
  printf("Filename created: %s\n", filename);
}
int main() {
  const char *name =
"example_filename_with_more_than_one_hundred_and_twenty_three_character_get_truncated";
  func(name);
  return 0;
}
```



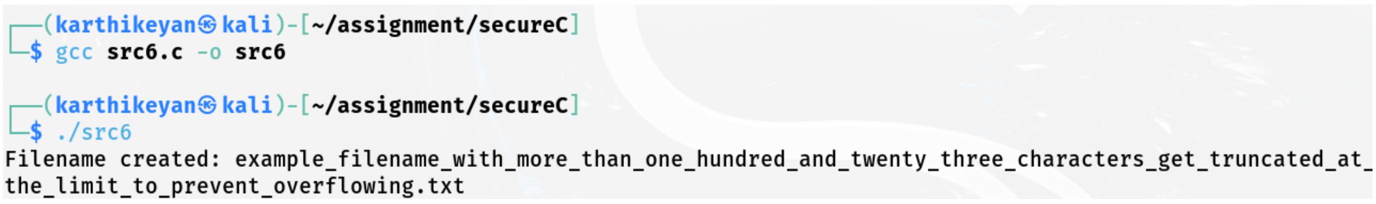
```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src6.c -o src6

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src6
Filename created: example_filename_with_more_than_one_hundred_and_twenty_three_characters_get_
runcated_at_the_limit_to_prevent_overflowing.txt
```

You can also use * to indicate that the precision should be provided as a variadic argument:

Compliant Solution (snprintf())

A more general solution is to use the snprintf() function, which also truncates name if it will not fit in the filename.

```c
#include <stdio.h>
#include <string.h>
void func(const char *name) {
  char filename[128];
  int result = snprintf(filename, sizeof(filename), "%s.txt", name);
  if (result != strlen(filename) )
  {
    printf("error copying \n");
    return;
  }
  printf("Filename created: %s\n", filename);
}
int main()
{
  const char *name =
"example_filename_with_more_than_one_hundred_and_twenty_three_character_get_truncated
";
  func(name);
  return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src6.c -o src6

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src6
Filename created: example_filename_with_more_than_one_hundred_and_twenty_three_characters_get_truncated_at_
the_limit_to_prevent_overflowing.txt
```

STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string

- Many library functions require that the strings (or wide strings) passed to them are **null-terminate.**

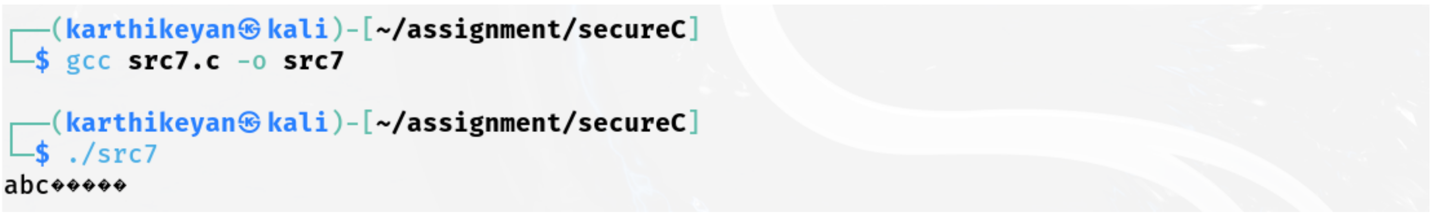
  This null termination indicates the end of the string, allowing the function to operate safely without reading beyond its bounds.
- If a string is **not null-terminated** and is passed to such a function, it may cause the function to read **beyond the allocated memory**, resulting in **undefined behavior** or **memory access violations**.

- To prevent this, **always ensure** that character sequences (or wide character sequences) are properly null terminated before passing them to functions that expect them. This is critical for maintaining program stability and avoiding security vulnerabilities like **buffer overflows**.

Noncompliant Code Example

This code example is noncompliant because the character sequence c_str will not be null-terminated when passed as an argument to printf().

```
#include <stdio.h>
void func(void) {
  char c_str[3] = "abc";
  printf("%s\n", c_str);
}
int main()
{
  func();
  return 0;
}
```

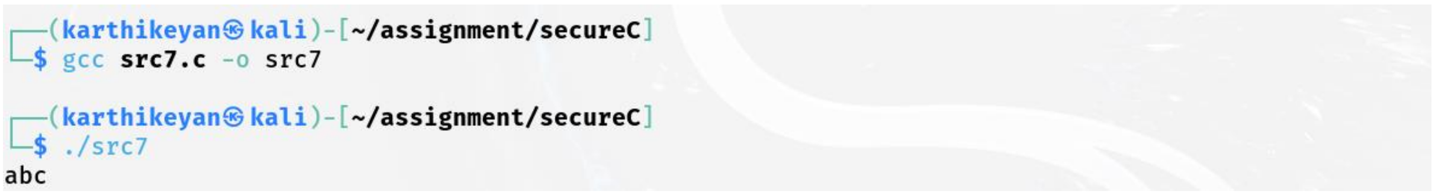


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src7.c -o src7

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src7
abc������
```

Compliant Solution

This compliant solution does not specify the bound of the character array in the array declaration. If the array bound is omitted, the compiler allocates sufficient storage to store the entire string literal, including the terminating null character.

```c
#include <stdio.h>
void func(void) {
  char c_str[] = "abc";

  printf("%s\n", c_str);
}
int main()
{
  func();
  return 0;
}
```

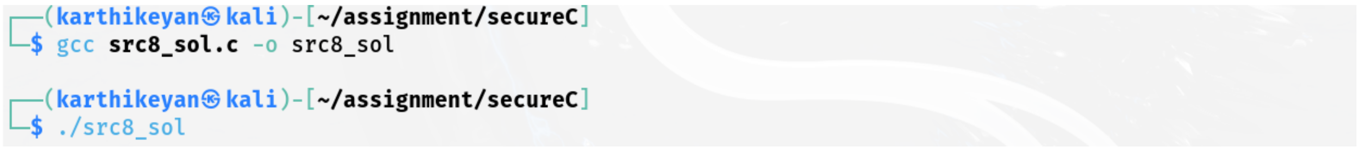


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src7.c -o src7

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src7
abc
```

Noncompliant Code Example

This code example is noncompliant because the wide character sequence cur_msg will not be null-terminated when passed to wcslen(). This will occur if lessen_memory_usage() is invoked while cur_msg_size still has its initial value of 1024.

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

wchar_t *cur_msg = NULL;
size_t cur_msg_size = 1024;
size_t cur_msg_len = 0;
void lessen_memory_usage(void) {
 wchar_t *temp;
 size_t temp_size;
/* ... */
 if (cur_msg != NULL) {
 temp_size = cur_msg_size / 2 + 1;
 temp = realloc(cur_msg, temp_size * sizeof(wchar_t));
 /* temp &and cur_msg may no longer be null-terminated */
 if (temp == NULL) {
 /* Handle error */
 printf("Error occured");
 }
 cur_msg = temp;
 cur_msg_size = temp_size;
 cur_msg_len = wcslen(cur_msg);

 }
}
int main() {
 cur_msg = (wchar_t *)malloc(cur_msg_size * sizeof(wchar_t));
 lessen_memory_usage();
 free(cur_msg);
 return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src8_sol.c -o src8_sol

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src8_sol
```

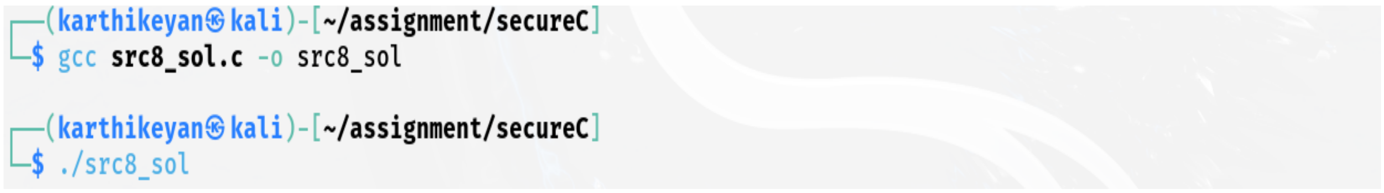
The program proceeds without printing anything, but it will still have undefined behavior due to the uninitialized memory being passed to wcslen().

Compliant Solution

In this compliant solution, cur_msg will always be null-terminated when passed to wcslen():

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

wchar_t *cur_msg = NULL;
size_t cur_msg_size = 1024;
size_t cur_msg_len = 0;
void lessen_memory_usage(void) {
 wchar_t *temp;
 size_t temp_size;
 /* ... */
 if (cur_msg != NULL) {
 temp_size = cur_msg_size / 2 + 1;
 temp = realloc(cur_msg, temp_size * sizeof(wchar_t));
 /* temp and cur_msg may no longer be null-terminated */
 if (temp == NULL) {
 /* Handle error */
 }
 cur_msg = temp;
 /* Properly null-terminate cur_msg */
 cur_msg[temp_size - 1] = L'\0';
 cur_msg_size = temp_size;
 cur_msg_len = wcslen(cur_msg);
 }
}
int main() {
```

```
cur_msg = (wchar_t *)malloc(cur_msg_size * sizeof(wchar_t));
if (cur_msg == NULL) {
/* Handle allocation error */
return 1;
}

/* Initialize cur_msg with some data */
wcsncpy(cur_msg, L"Test message", cur_msg_size - 1);
cur_msg[cur_msg_size - 1] = L'\0'; // Ensure null termination
lessen_memory_usage();
free(cur_msg);
return 0;
}
```



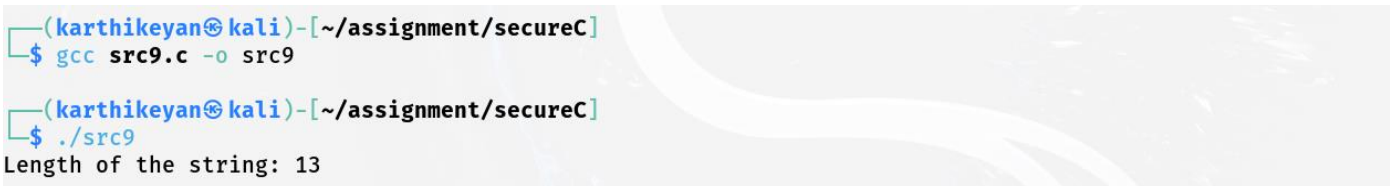
```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src8_sol.c -o src8_sol

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src8_sol
```

Noncompliant Code Example (strncpy())

Although the strncpy() function takes a string as input, it does not guarantee that the resulting value is still null-terminated. In the following noncompliant code example, if no null character is contained in the first n characters of the source array, the result will not be null-

```
#include <string.h>
#include<stdio.h>
enum { STR_SIZE = 32 };
size_t func(const char *source) {
char c_str[STR_SIZE];
size_t ret = 0;
if (source) {
c_str[sizeof(c_str) - 1] = '\0';
strncpy(c_str, source, sizeof(c_str));
ret = strlen(c_str);
} else {
/* Handle null pointer */
}
return ret;
}
int main() {
const char *source = "Sample string";
size_t length = func(source);
printf("Length of the string: %zu\n", length);
return 0;
}
```

terminated. Passing a non-null-terminated character sequence to strlen() is undefined behavior.

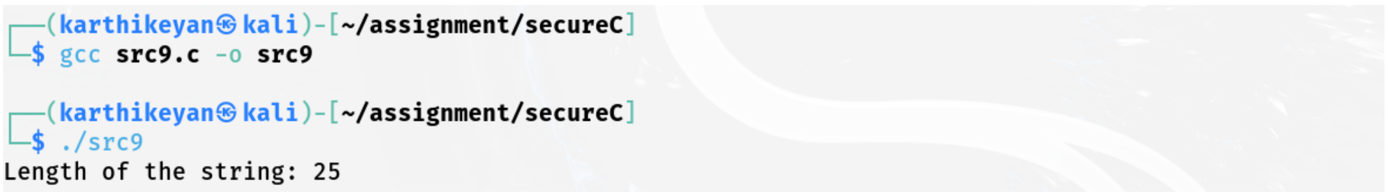


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src9.c -o src9

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src9
Length of the string: 13
```

Compliant Solution (Truncation)

This compliant solution is correct if the programmer's intent is to truncate the string:

```
#include <string.h>
#include<stdio.h>
enum { STR_SIZE = 32 };
size_t func(const char *source)
{
  char c_str[STR_SIZE];
  size_t ret = 0;
  if (source) {
    strncpy(c_str, source, sizeof(c_str) - 1);
    c_str[sizeof(c_str) - 1] = '\0';
    ret = strlen(c_str);
  }
  else {
    /* Handle null pointer */
  }
  return ret;
}
int main() {
  const char *source = "Sample string for testing";
  size_t length = func(source);
  printf("Length of the string: %zu\n", length);
  return 0;
}
```

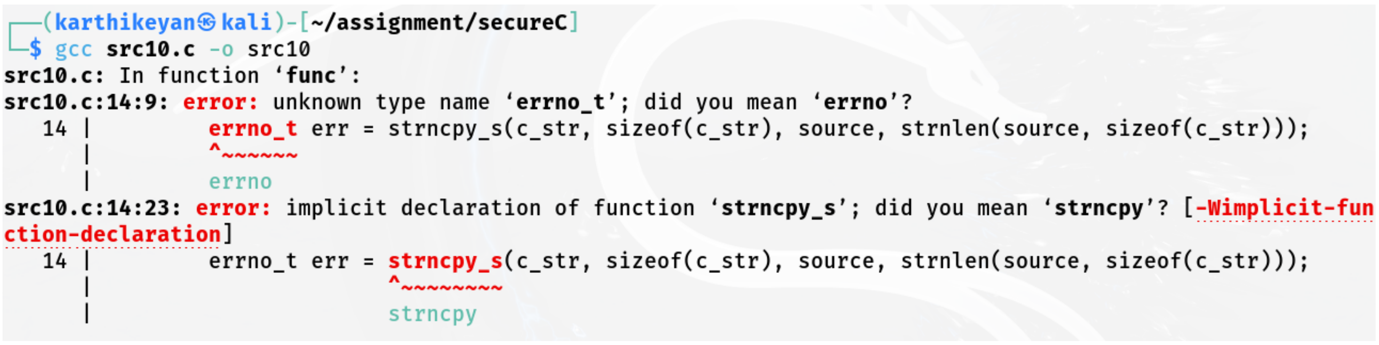


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src9.c -o src9

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src9
Length of the string: 25
```

Compliant Solution (Truncation, strncpy_s())

The C Standard, Annex K strncpy_s() function can also be used to copy with truncation. The strncpy_s() function copies up to n characters from the source array to a destination array. If no null character was copied from the source array, then the nth position in the destination array is set to a null character, guaranteeing that the resulting string is null-terminated.

```c
#include <string.h>
#include <stdio.h>
#include <errno.h>
#define __STDC_WANT_LIB_EXT1__ 1

enum { STR_SIZE = 32 };

size_t func(const char *source) {
  char c_str[STR_SIZE];
  size_t ret = 0;

  if (source) {
    errno_t err = strncpy_s(c_str, sizeof(c_str), source, strnlen(source, sizeof(c_str)));
    if (err != 0) {
      /* Handle error */
      printf("Error copying string\n");
    } else {
      ret = strnlen(c_str, sizeof(c_str));
    }
  } else {
     /* Handle null pointer */
     printf("Null pointer error\n");
  }
  return ret;
}
int main() {
const char *source = "Sample string for testing";
size_t length = func(source);
printf("Length of the string: %zu\n", length);
return 0;
}
```

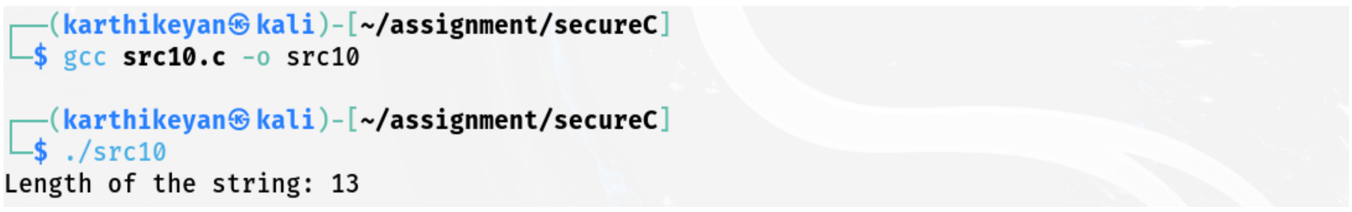


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src10.c -o src10
src10.c: In function ‘func’:
src10.c:14:9: error: unknown type name ‘errno_t’; did you mean ‘errno’?
   14 |         errno_t err = strncpy_s(c_str, sizeof(c_str), source, strnlen(source, sizeof(c_str)));
      |         ^~~~~~~
      |         errno
src10.c:14:23: error: implicit declaration of function ‘strncpy_s’; did you mean ‘strncpy’? [-Wimplicit-fun
ction-declaration]
   14 |         errno_t err = strncpy_s(c_str, sizeof(c_str), source, strnlen(source, sizeof(c_str)));
      |                       ^~~~~~~~~
      |                       strncpy
```

Compliant Solution (Copy without Truncation)

If the programmer's intent is to copy without truncation, this compliant solution copies the data and guarantees that the resulting array is null-terminated. If the string cannot be copied, it is handled as an error condition.

```
#include <string.h>
#include <stdio.h>
enum { STR_SIZE = 32 };
size_t func(const char *source) {
  char c_str[STR_SIZE];
  size_t ret = 0;
  if (source) {
  if (strnlen(source, sizeof(c_str)) < sizeof(c_str)) {
    strcpy(c_str, source);
    ret = strlen(c_str);
  }
  else {
    /* Handle string-too-large */
  }
  }
  else {
    /* Handle null pointer */
  }
  return ret;
}
int main() {
  const char *source = "Sample string";
  size_t length = func(source);
  printf("Length of the string: %zu\n", length);
  return 0;
}
```



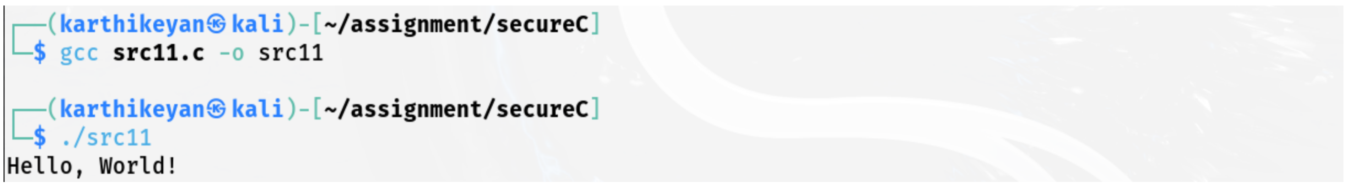
```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src10.c -o src10

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src10
Length of the string: 13
```

This code handles cases where source is NULL, a valid string, or a non-null-terminated string with at least the first 32 bytes valid. However, if source points to invalid memory or any of the first 32 bytes are invalid, the call to strnlen() will access this invalid memory, causing undefined behavior. Standard C cannot prevent or detect this without external information about source

Noncompliant Code Example

This noncompliant code example is taken from a vulnerability in bash versions 1.14.6 and earlier that led to the release of CERT Advisory CA-1996-22. This vulnerability resulted from the sign extension of character data referenced by the c_str pointer in the yy_string_get() function in the parse.y module of the bash source code:

```
#include <stdio.h>
struct {
 struct {
 char *string;
 } location;
} bash_input;
static int yy_string_get(void) {
 register char *c_str;
 register int c;
 c_str = bash_input.location.string;
 c = EOF;
 /* If the string doesn't exist or is empty, EOF found */
 if (c_str && *c_str) {
 c = *c_str++;
 bash_input.location.string = c_str;
 }
 return (c);
}
int main() {
 char input[] = "Hello, World!";
 bash_input.location.string = input;
 int ch;
 while ((ch = yy_string_get()) != EOF) {
 putchar(ch);
 }
 putchar('\n');
 return 0;
}
```



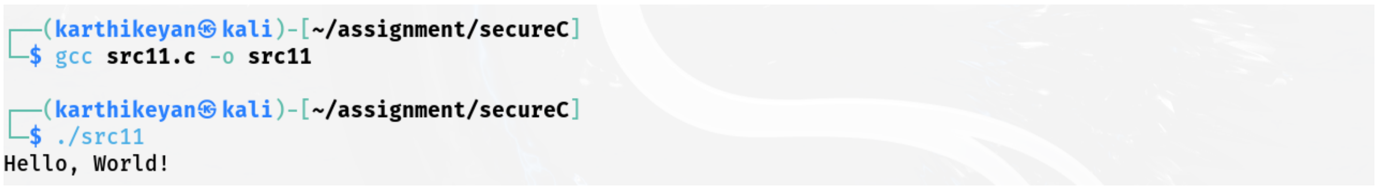
```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src11.c -o src11

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src11
Hello, World!
```

The c_str variable is used to traverse the character string containing the command line to be parsed. As characters are retrieved from this pointer, they are stored in a variable of type int. For implementations in which the char type is defined to have the same range, representation, and behavior as signed char, this value is sign-extended when assigned to the int variable. For character code 255 decimal (−1 in two's complement form), this sign extension results in the value −1 being assigned to the integer, which is indistinguishable from EOF.

Compliant Solution

In this compliant solution, the result of the expression *c_str++ is cast to unsigned char before assignment to the int variable c:

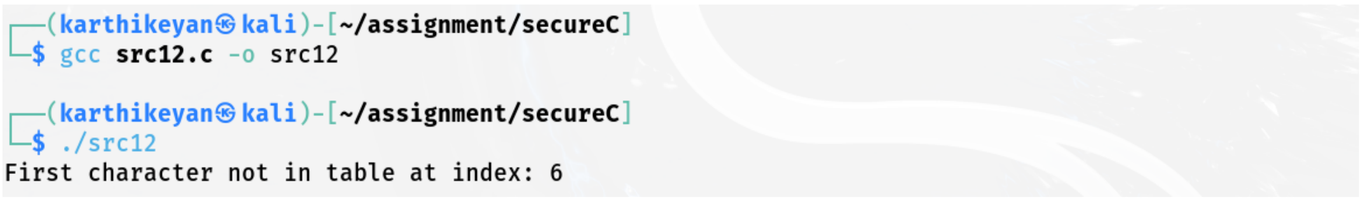
```
#include <stdio.h>
struct {
 struct {
 char *string;
 } location;
} bash_input;
static int yy_string_get(void) {
 register char *c_str;
 register int c;
 c_str = bash_input.location.string;
 c = EOF;
 /* If the string doesn't exist or is empty, EOF found */
 if (c_str && *c_str) {
 /* Cast to unsigned type */
 c = (unsigned char)*c_str++;
 bash_input.location.string = c_str;
 }
 return (c);
int main() {
 char input[] = "Hello, World!";
 bash_input.location.string = input;
 int ch;
 while ((ch = yy_string_get()) != EOF) {
 putchar(ch);
 }
 putchar('\n');
 return 0;
}
```


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src11.c -o src11

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src11
Hello, World!
```

Noncompliant Code Example

In this noncompliant code example, the cast of *s to unsigned int can result in a value in excess of UCHAR_MAX because of integer promotions, a violation of ARR30-C. Do not form or use out-of-bounds pointers or array subscripts:

```
#include   <stdio.h>
#include   <limits.h>
#include <stddef.h>
static const char table[UCHAR_MAX + 1] = {
 [0 ... UCHAR_MAX] = 0, // Initialize all entries to 0
 ['a'] = 'a',
 ['b'] = 'b',
 ['c'] = 'c',
 ['d'] = 'd',
 ['e'] = 'e',
 ['f'] = 'f', // ... initialize other characters as needed
};
ptrdiff_t first_not_in_table(const char *c_str) {
 for (const char *s = c_str; *s; ++s) {
 if (table[(unsigned int)*s] != *s) {
 return s - c_str;
 }
 }
 return -1;
}
int main() {
 const char *input = "abcdefg"; // Example input string
 ptrdiff_t result = first_not_in_table(input);
 if (result != -1) {
 printf("First character not in table at index: %td\n", result);
 } else {
 printf("All characters are in the table.\n");
 }
 return 0;
}
```

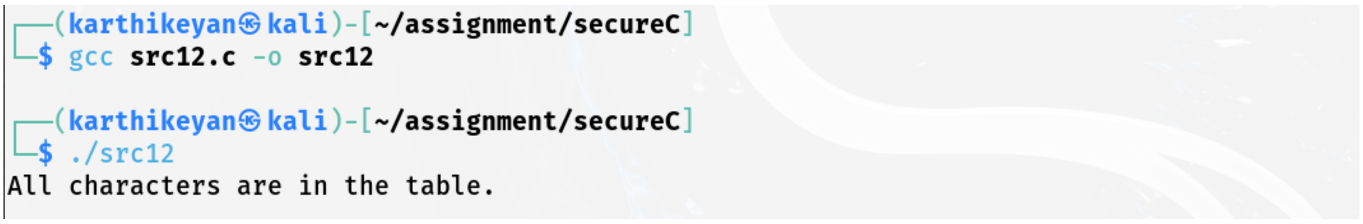


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src12.c -o src12

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src12
First character not in table at index: 6
```

Compliant Solution

This compliant solution casts the value of type char to unsigned char before the implicit promotion to a larger type:

```c
#include   <stdio.h>
#include   <limits.h>
#include <stddef.h>
static const char table [UCHAR_MAX + 1] = {
[0 ... UCHAR_MAX] = 0,
['a'] = 'a',
['b'] = 'b',
['c'] = 'c',
['d'] = 'd',
['e'] = 'e',
['f'] = 'f',
['g'] = 'g'
};
ptrdiff_t first_not_in_table(const char *c_str) {
for (const char *s = c_str; *s; ++s) {
if (table[(unsigned char)*s] != *s) {
return s-c_str;
}
}return -1;}
int main() {
const char *input = "abcdeg"; // Example input string
ptrdiff_t result = first_not_in_table(input);
if (result != -1) {
printf("First character not in table at index: %td\n", result);
} else {
printf("All characters are in the table.\n");
}
return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src12.c -o src12

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src12
All characters are in the table.
```

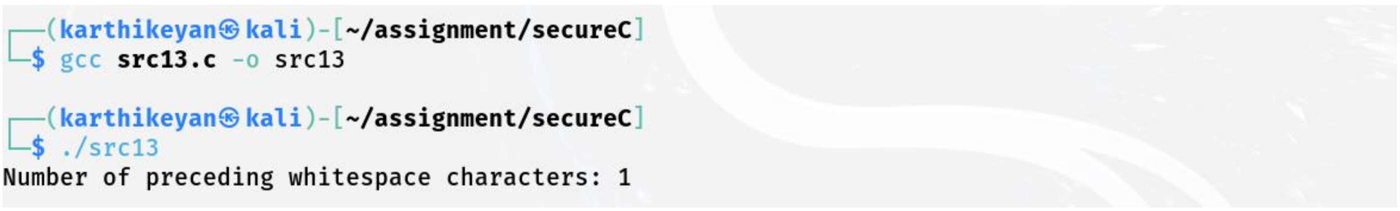
STR37-C. Arguments to character-handling functions must be representable as an unsigned char

This rule is applicable only to code that runs on platforms where the char data type is defined to have the same range, representation, and behavior as signed char.

Noncompliant Code Example

On implementations where plain char is signed, this code example is noncompliant because the parameter to isspace(), *t, is defined as a const char *, and this value might not be representable as an unsigned char:

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

size_t count_preceding_whitespace(const char *s) {
  const char *t = s;
  size_t length = strlen(s) + 1;
  while (isspace(*t) && (t - s < length)) {
    ++t;
  }
  return t - s;
}

int main() {
    const char *input = "  Hello, World!";
    size_t whitespace_count = count_preceding_whitespace(input);
    printf("Number of preceding whitespace characters: %zu\n", whitespace_count);
    return 0;
}
```



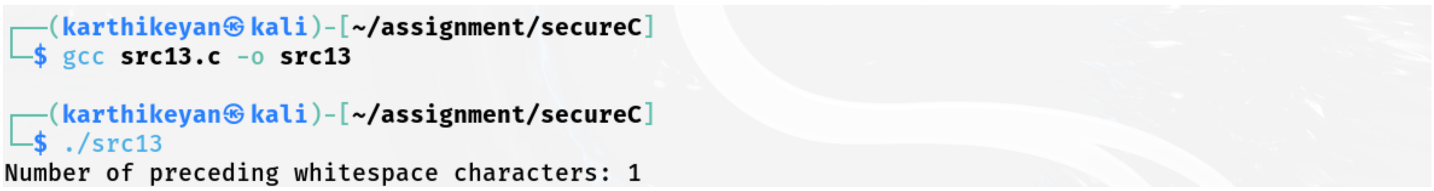
```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src13.c -o src13

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src13
Number of preceding whitespace characters: 1
```

The argument to is space () must be EOF or representable as an unsigned char; otherwise, the result is undefined.

Compliant Solution

This compliant solution casts the character to unsigned char before passing it as an argument to the isspace() function:

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

size_t count_preceding_whitespace(const char *s) {
  const char *t = s;
  size_t length = strlen(s) + 1;
  while (isspace((unsigned char)*t) && (t - s < length)) {
    ++t;
  }
  return t - s;
}

int main() {
    const char *input = "  Hello, World!";
    size_t whitespace_count = count_preceding_whitespace(input);
    printf("Number of preceding whitespace characters: %zu\n", whitespace_count);
    return 0;
}
```



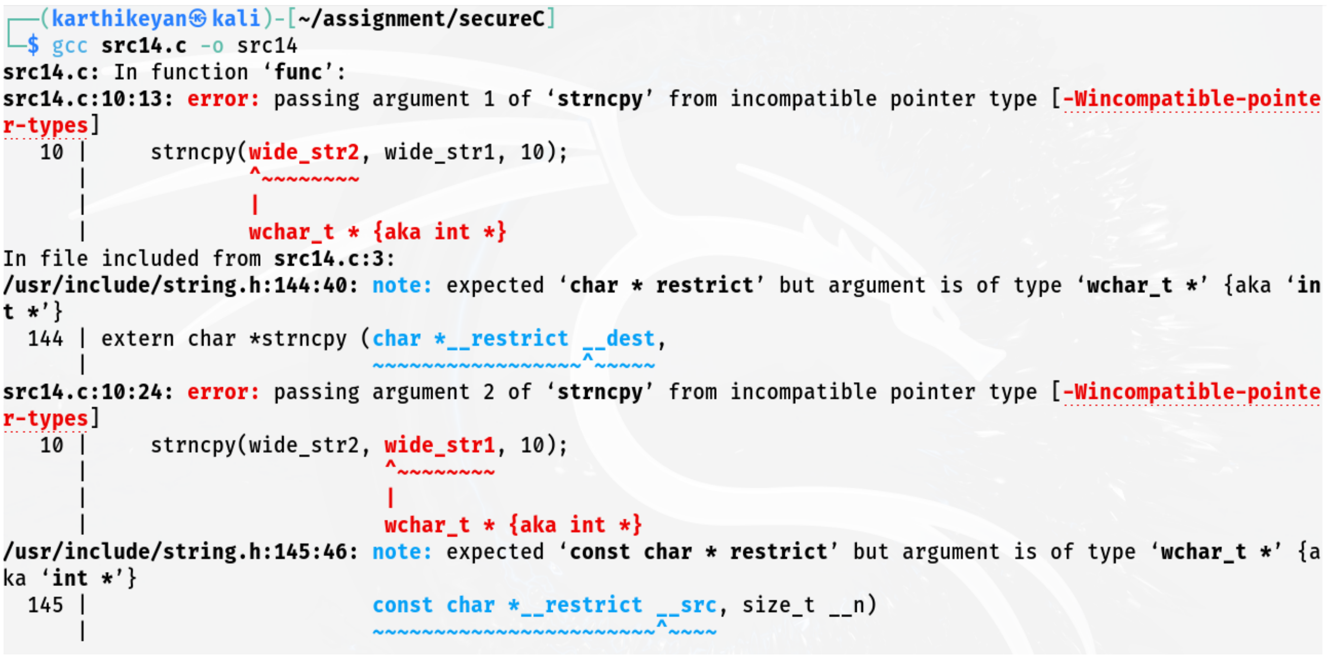
```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src13.c -o src13

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src13
Number of preceding whitespace characters: 1
```

STR38-C. Do not confuse narrow and wide character strings and functions

- Passing narrow string arguments to wide string functions, or vice versa, causes unexpected and undefined behavior.
- This can lead to scaling problems due to the size difference between wide and narrow characters.
- Wide strings are terminated by a null wide character and may contain null bytes, making their length determination problematic.
- According to ARR39-C, do not add or subtract a scaled integer to a pointer, as it may cause issues when dealing with different character sizes.
- Since wchar_t and char are distinct types, many compilers provide warning diagnostics when the wrong type of function is used.

Noncompliant Code Example (Wide Strings with Narrow String Functions)

```c
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <wchar.h>

void func(void) {
  wchar_t wide_str1[]  = L"0123456789";
  wchar_t wide_str2[] =  L"0000000000";

  strncpy(wide_str2, wide_str1, 10);
}
int main() {
  func(); // Call the function
  return 0;
}
```



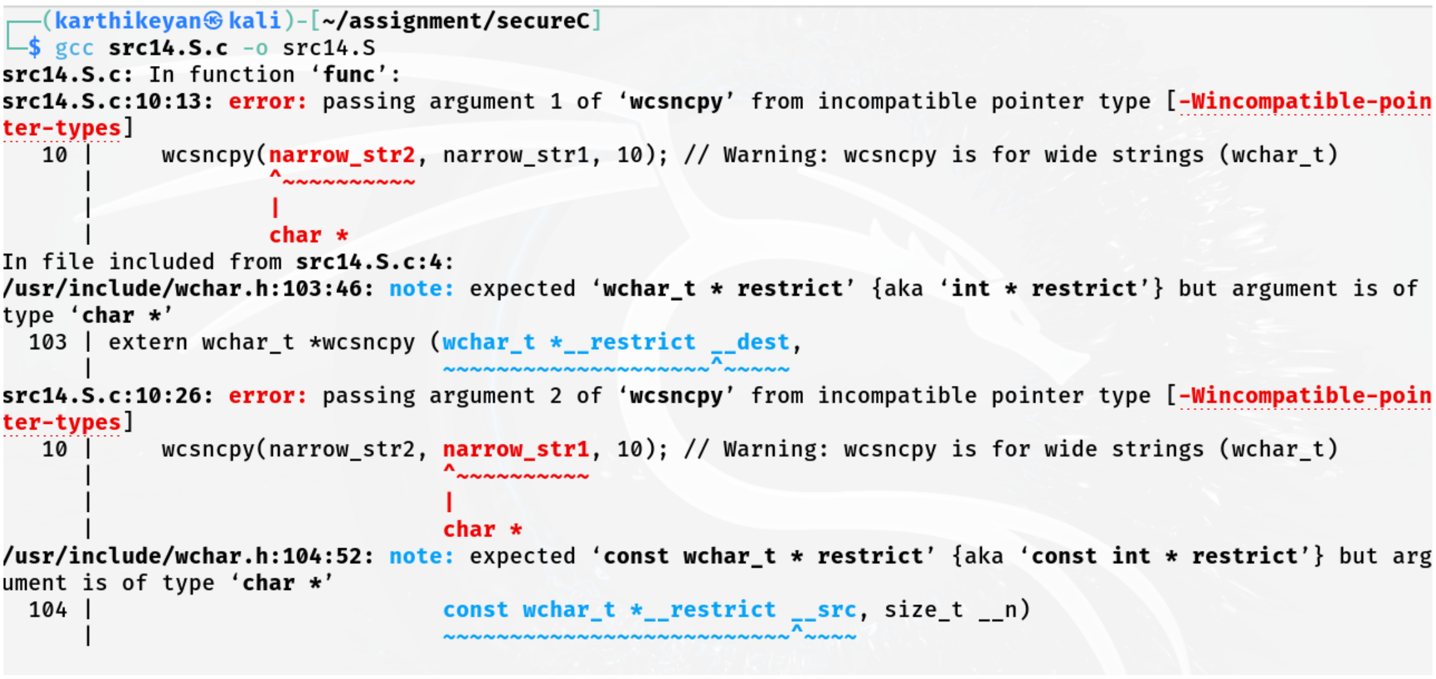
```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src14.c -o src14
src14.c: In function ‘func’:
src14.c:10:13: error: passing argument 1 of ‘strncpy’ from incompatible pointer type [-Wincompatible-pointe
r-types]
   10 |     strncpy(wide_str2, wide_str1, 10);
      |             ^~~~~~~~~
      |             |
      |             wchar_t * {aka int *}
In file included from src14.c:3:
/usr/include/string.h:144:40: note: expected ‘char * restrict’ but argument is of type ‘wchar_t *’ {aka ‘in
t *’}
  144 | extern char *strncpy (char *__restrict __dest,
      |                       ~~~~~~~~~~~~~~~~~^~~~~~
src14.c:10:24: error: passing argument 2 of ‘strncpy’ from incompatible pointer type [-Wincompatible-pointe
r-types]
   10 |     strncpy(wide_str2, wide_str1, 10);
      |                        ^~~~~~~~~
      |                        |
      |                        wchar_t * {aka int *}
/usr/include/string.h:145:46: note: expected ‘const char * restrict’ but argument is of type ‘wchar_t *’ {a
ka ‘int *’}
  145 |                       const char *__restrict __src, size_t __n)
      |                       ~~~~~~~~~~~~~~~~~~~~~~~^~~~~
```

This noncompliant code example incorrectly uses the strncpy() function in an attempt to copy up to 10 wide characters. However, because wide characters can contain null bytes, the copy operation may end earlier than anticipated, resulting in the truncation of the wide string.

Noncompliant Code Example (Narrow Strings with Wide String Functions)

This noncompliant code example incorrectly invokes the wcsncpy() function to copy up to 10 wide characters from narrow_str1 to narrow_str2. Because narrow_str2 is a narrow string, it has insufficient memory to store the result of the copy and the copy will result in a buffer overflow.

```c
#include <stdio.h>
#include <stddef.h>
#include <string.h>
#include <wchar.h>

void func(void) {
  char narrow_str1[] = "01234567890123456789";
  char narrow_str2[] = "0000000000";

  wcsncpy(narrow_str2, narrow_str1, 10);
}
int main() {
  func(); // Call the function
  return 0;
}
```

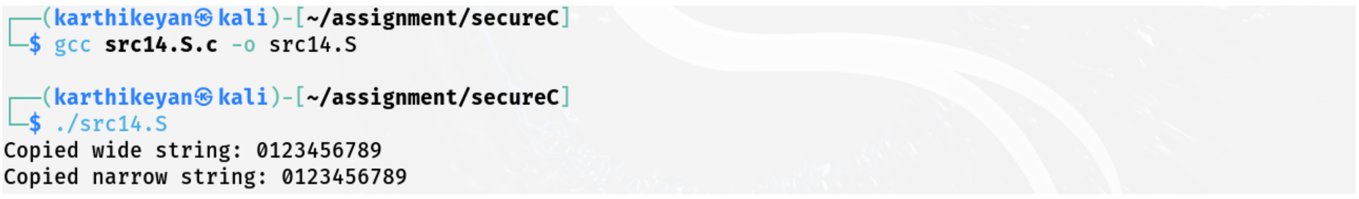


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src14.S.c -o src14.S
src14.S.c: In function ‘func’:
src14.S.c:10:13: error: passing argument 1 of ‘wcsncpy’ from incompatible pointer type [-Wincompatible-poin
ter-types]
   10 |     wcsncpy(narrow_str2, narrow_str1, 10); // Warning: wcsncpy is for wide strings (wchar_t)
      |             ^~~~~~~~~~~
      |             |
      |             char *
In file included from src14.S.c:4:
/usr/include/wchar.h:103:46: note: expected ‘wchar_t * restrict’ {aka ‘int * restrict’} but argument is of
type ‘char *’
  103 | extern wchar_t *wcsncpy (wchar_t *__restrict __dest,
      |                          ~~~~~~~~~~~~~~~~~~~~^~~~~~
src14.S.c:10:26: error: passing argument 2 of ‘wcsncpy’ from incompatible pointer type [-Wincompatible-poin
ter-types]
   10 |     wcsncpy(narrow_str2, narrow_str1, 10); // Warning: wcsncpy is for wide strings (wchar_t)
      |                          ^~~~~~~~~~~
      |                          |
      |                          char *
/usr/include/wchar.h:104:52: note: expected ‘const wchar_t * restrict’ {aka ‘const int * restrict’} but arg
ument is of type ‘char *’
  104 |                          const wchar_t *__restrict __src, size_t __n)
      |                          ~~~~~~~~~~~~~~~~~~~~~~~~~~^~~~~
```

Compliant Solution

This compliant solution uses the proper-width functions. Using wcsncpy() for wide character strings and strncpy() for narrow character strings ensures that data is not truncated and buffer overflow does not occur.

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

void func(void) {
  wchar_t wide_str1[] = L"0123456789";
  wchar_t wide_str2[] = L"0000000000";
  /* Use of proper-width function */
  wcsncpy(wide_str2, wide_str1, 10);
  printf("Copied wide string: %ls\n", wide_str2);
  char narrow_str1[] = "0123456789";
  char narrow_str2[] = "0000000000";
  /* Use of proper-width function */
  strncpy(narrow_str2, narrow_str1, 10);
  printf("Copied narrow string: %s\n", narrow_str2);
}
int main() {
  func(); // Call the function
  return 0;
}
```

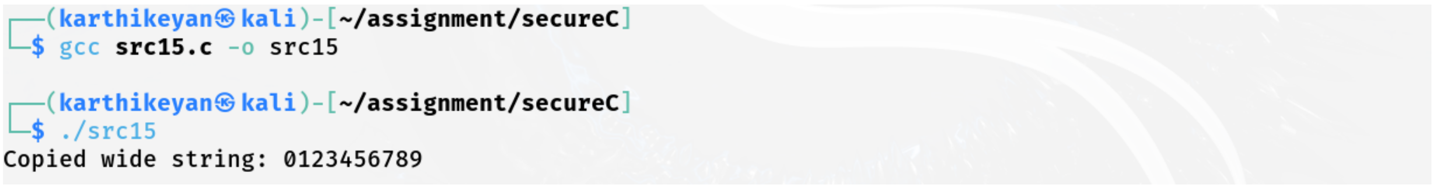


```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src14.S.c -o src14.S

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src14.S
Copied wide string: 0123456789
Copied narrow string: 0123456789
```

Noncompliant Code Example (strlen())

In this noncompliant code example, the strlen() function is used to determine the size of a wide character string:

```
#include  <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wchar.h>
void func(void) {
  wchar_t wide_str1[] = L"0123456789";
  wchar_t *wide_str2 = (wchar_t*)malloc((wcslen(wide_str1) + 1) * sizeof(wchar_t));
  if (wide_str2 == NULL) {
  wprintf(L"Memory allocation failed.\n");
  return;
  }
  wcsncpy(wide_str2, wide_str1, wcslen(wide_str1) + 1);
  wprintf(L"Copied wide string: %ls\n", wide_str2);
  free(wide_str2);
  wide_str2 = NULL; // Set pointer to NULL after freeing
}
int main() {
  func(); // Call the function
  return 0;
}
```



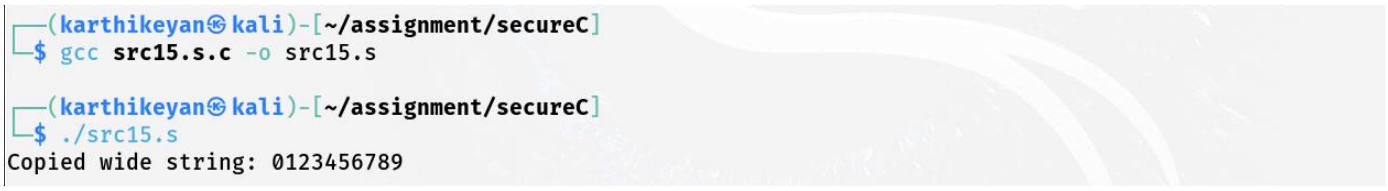
```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src15.c -o src15

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src15
Copied wide string: 0123456789
```

The strlen() function determines the number of characters that precede the terminating null character. However, wide characters can contain null bytes, particularly when expressing characters from the ASCII character set, as in this example. As a result, the strlen() function will return the number of bytes preceding the first null byte in the wide string.

Compliant Solution

This compliant solution correctly calculates the number of bytes required to contain a copy of the wide string, including the terminating null wide character:

```
#include   <stdio.h>
#include  <stdlib.h>
#include  <string.h>
#include <wchar.h>
void func(void) {
   wchar_t wide_str1[] = L"0123456789";
   wchar_t *wide_str2 = (wchar_t *)malloc((wcslen(wide_str1) + 1) * sizeof(wchar_t));
   if (wide_str2 == NULL)
   {
     wprintf(L"Memory allocation failed.\n");
     return;
   }
   wcscpy(wide_str2, wide_str1);
   wprintf(L"Copied wide string: %ls\n", wide_str2);
   free(wide_str2);
   wide_str2 = NULL;
}
int main() {
   func(); // Call the function
   return 0;
}
```



```
┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ gcc src15.s.c -o src15.s

┌──(karthikeyan㉿kali)-[~/assignment/secureC]
└─$ ./src15.s
Copied wide string: 0123456789
```