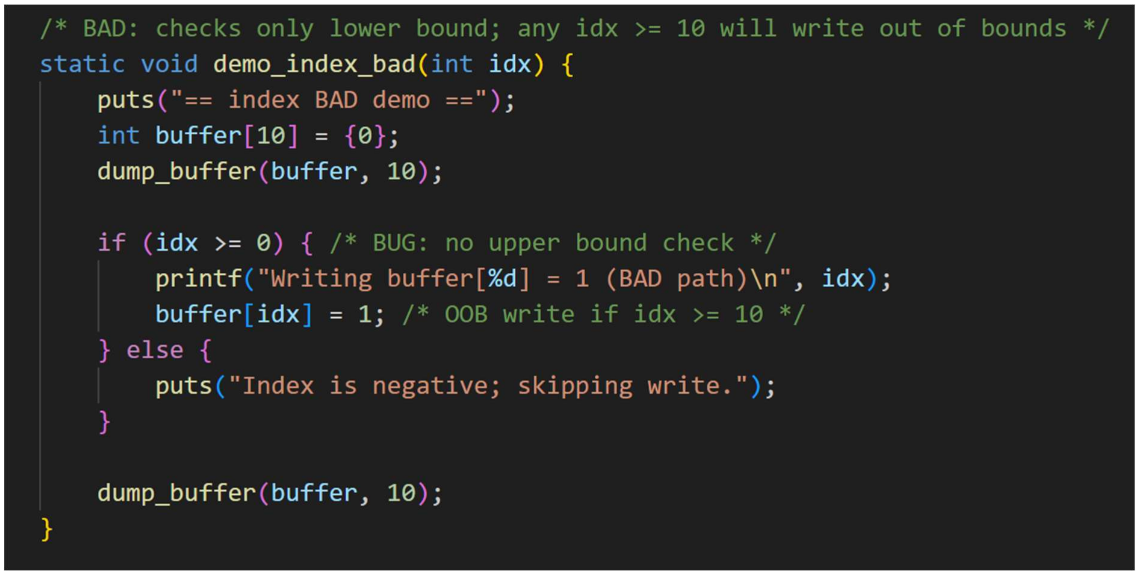
Stack Buffer Over flow testing and fixing vulnerable code snippets

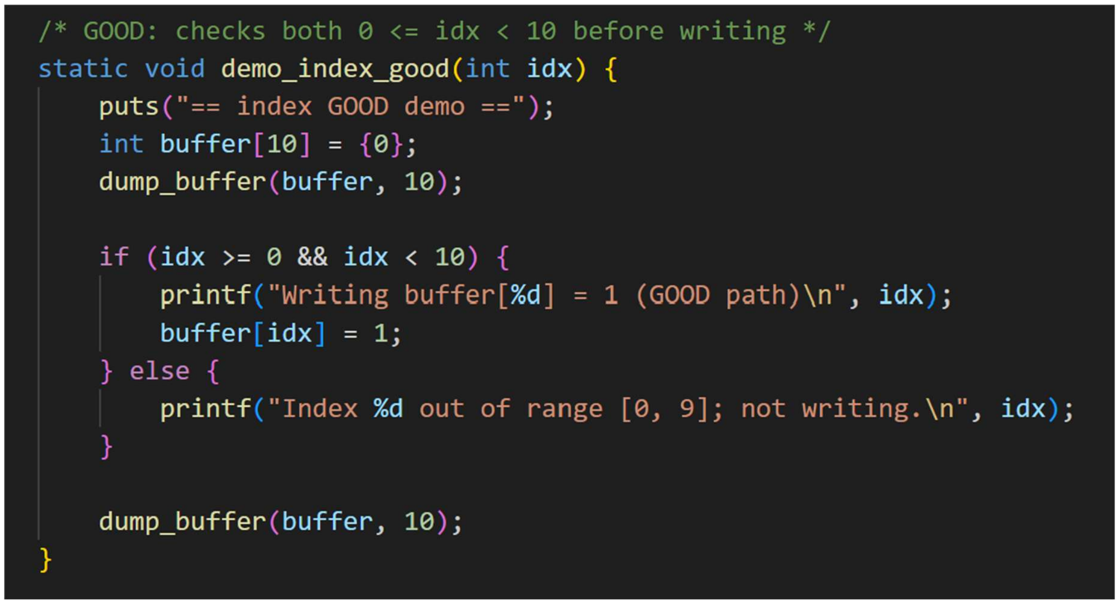
BAD example which is vulnerable towards array out of bound issue.

Broken and vulnerable code:



```c
/* BAD: checks only lower bound; any idx >= 10 will write out of bounds */
static void demo_index_bad(int idx) {
    puts("== index BAD demo ==");
    int buffer[10] = {0};
    dump_buffer(buffer, 10);

    if (idx >= 0) { /* BUG: no upper bound check */
        printf("Writing buffer[%d] = 1 (BAD path)\n", idx);
        buffer[idx] = 1; /* OOB write if idx >= 10 */
    } else {
        puts("Index is negative; skipping write.");
    }

    dump_buffer(buffer, 10);
}
```

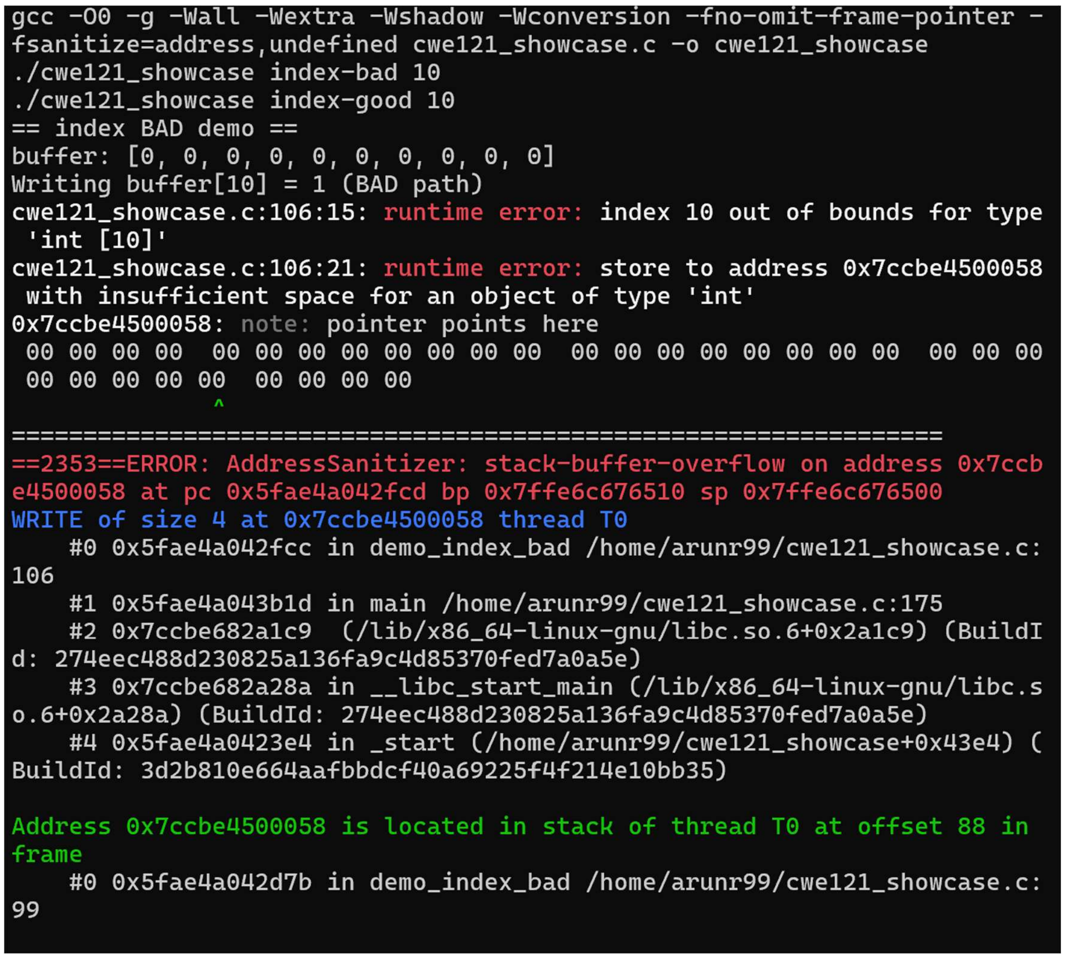
Code Fix done:



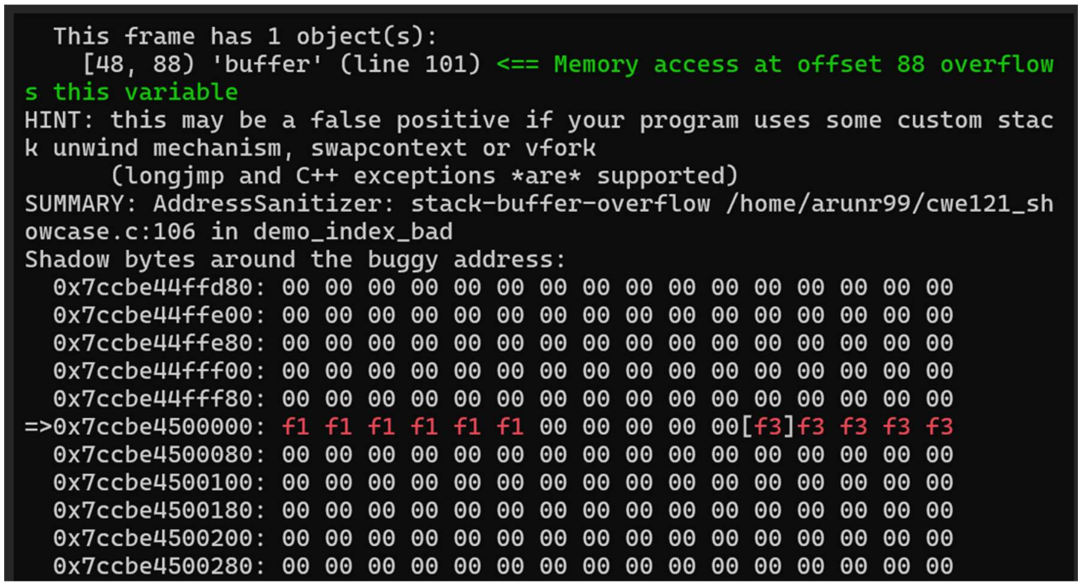
```c
/* GOOD: checks both 0 <= idx < 10 before writing */
static void demo_index_good(int idx) {
    puts("== index GOOD demo ==");
    int buffer[10] = {0};
    dump_buffer(buffer, 10);

    if (idx >= 0 && idx < 10) {
        printf("Writing buffer[%d] = 1 (GOOD path)\n", idx);
        buffer[idx] = 1;
    } else {
        printf("Index %d out of range [0, 9]; not writing.\n", idx);
    }

    dump_buffer(buffer, 10);
}
```

Implementation Results:

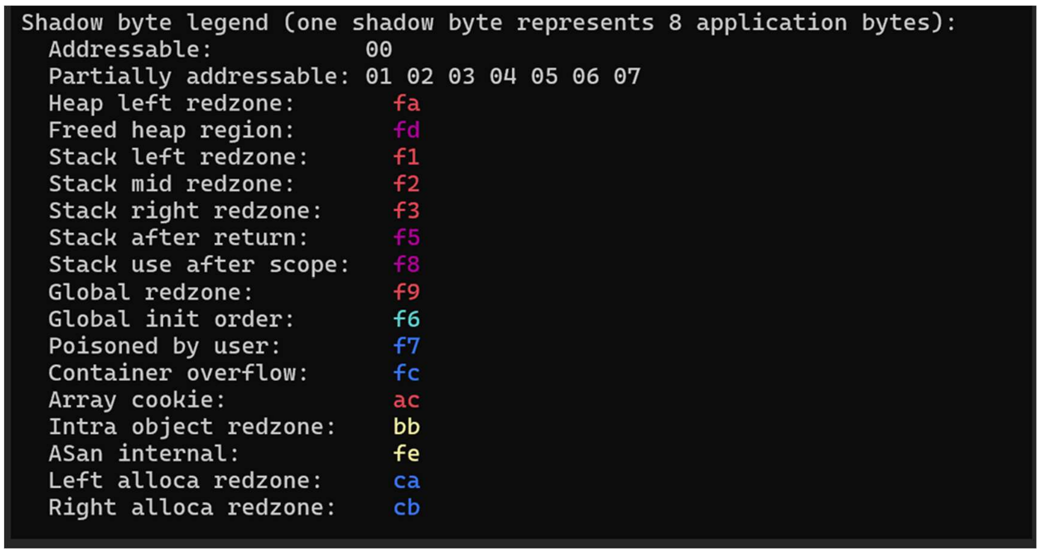
Output from Bad and Vulnerable code, stack overflow happens and it overwrites the memory.



```
gcc -O0 -g -Wall -Wextra -Wshadow -Wconversion -fno-omit-frame-pointer -
fsanitize=address,undefined cwe121_showcase.c -o cwe121_showcase
./cwe121_showcase index-bad 10
./cwe121_showcase index-good 10
== index BAD demo ==
buffer: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Writing buffer[10] = 1 (BAD path)
cwe121_showcase.c:106:15: runtime error: index 10 out of bounds for type
 'int [10]'
cwe121_showcase.c:106:21: runtime error: store to address 0x7ccbe4500058
 with insufficient space for an object of type 'int'
0x7ccbe4500058: note: pointer points here
 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00
 00 00 00 00 00  00 00 00 00
              ^
=================================================================
==2353==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ccb
e4500058 at pc 0x5fae4a042fcd bp 0x7ffe6c676510 sp 0x7ffe6c676500
WRITE of size 4 at 0x7ccbe4500058 thread T0
    #0 0x5fae4a042fcc in demo_index_bad /home/arunr99/cwe121_showcase.c:
106
    #1 0x5fae4a043b1d in main /home/arunr99/cwe121_showcase.c:175
    #2 0x7ccbe682a1c9  (/lib/x86_64-linux-gnu/libc.so.6+0x2a1c9) (BuildI
d: 274eec488d230825a136fa9c4d85370fed7a0a5e)
    #3 0x7ccbe682a28a in __libc_start_main (/lib/x86_64-linux-gnu/libc.s
o.6+0x2a28a) (BuildId: 274eec488d230825a136fa9c4d85370fed7a0a5e)
    #4 0x5fae4a0423e4 in _start (/home/arunr99/cwe121_showcase+0x43e4) (
BuildId: 3d2b810e664aafbbdcf40a69225f4f214e10bb35)

Address 0x7ccbe4500058 is located in stack of thread T0 at offset 88 in
frame
    #0 0x5fae4a042d7b in demo_index_bad /home/arunr99/cwe121_showcase.c:
99
```

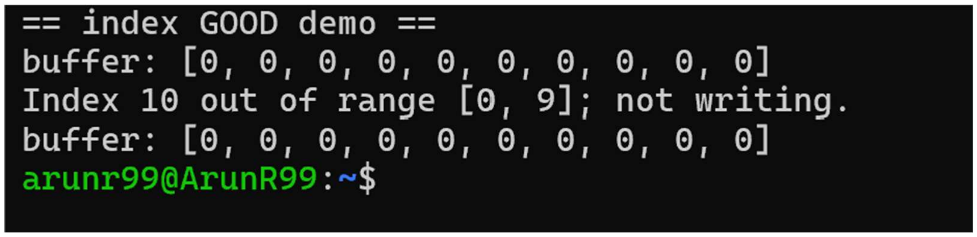


```
  This frame has 1 object(s):
    [48, 88) 'buffer' (line 101) <== Memory access at offset 88 overflow
s this variable
HINT: this may be a false positive if your program uses some custom stac
k unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow /home/arunr99/cwe121_sh
owcase.c:106 in demo_index_bad
Shadow bytes around the buggy address:
  0x7ccbe44ffd80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7ccbe44ffe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7ccbe44ffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7ccbe44fff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7ccbe44fff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x7ccbe4500000: f1 f1 f1 f1 f1 f1 00 00 00 00 00[f3]f3 f3 f3 f3
  0x7ccbe4500080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7ccbe4500100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7ccbe4500180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7ccbe4500200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x7ccbe4500280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```



```
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
```

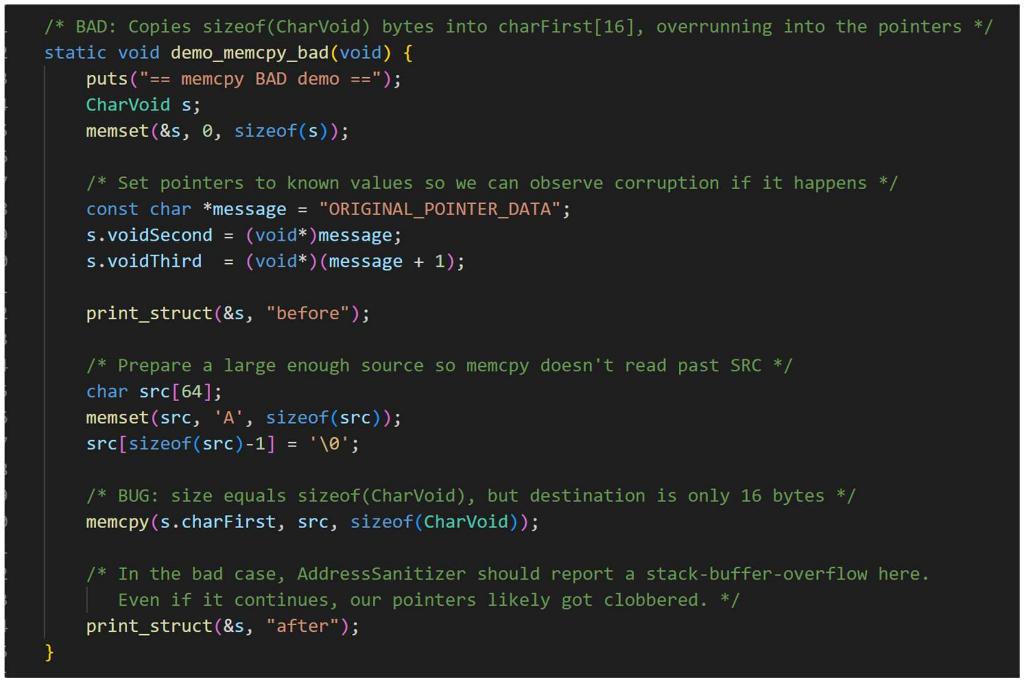
Output from Fixed code, runs successfully without causing an array out of bounds as stack overflow has been fixed here.



```
== index GOOD demo ==
buffer: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Index 10 out of range [0, 9]; not writing.
buffer: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
arunr99@ArunR99:~$
```

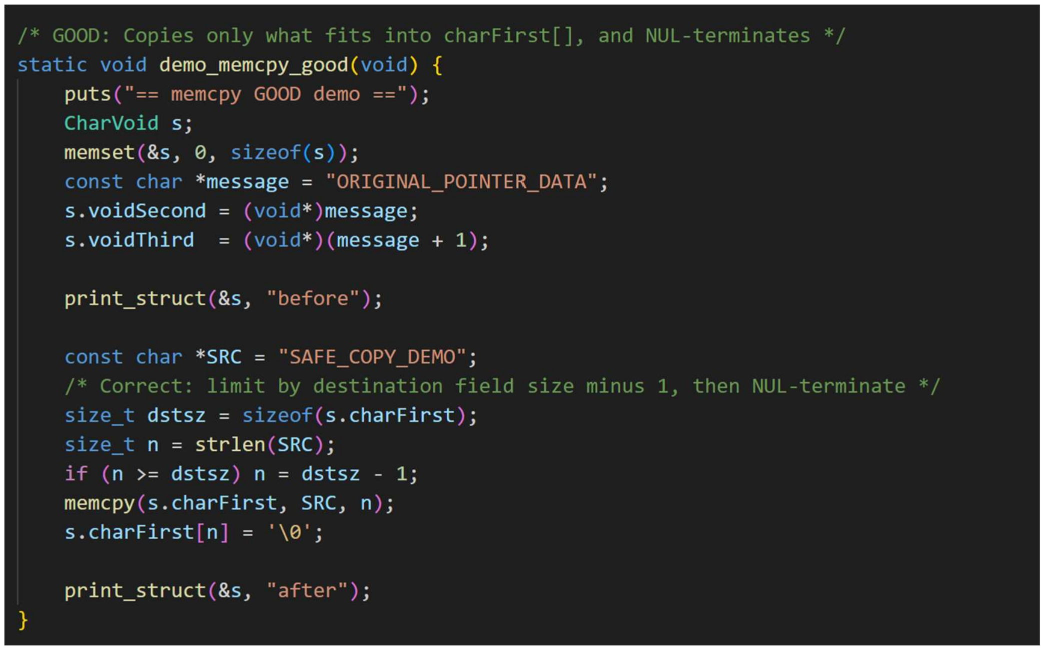
BAD example which is vulnerable towards a Memory copy issue.

Broken and vulnerable code:



```c
/* BAD: Copies sizeof(CharVoid) bytes into charFirst[16], overrunning into the pointers */
static void demo_memcpy_bad(void) {
    puts("== memcpy BAD demo ==");
    CharVoid s;
    memset(&s, 0, sizeof(s));

    /* Set pointers to known values so we can observe corruption if it happens */
    const char *message = "ORIGINAL_POINTER_DATA";
    s.voidSecond = (void*)message;
    s.voidThird  = (void*)(message + 1);

    print_struct(&s, "before");

    /* Prepare a large enough source so memcpy doesn't read past SRC */
    char src[64];
    memset(src, 'A', sizeof(src));
    src[sizeof(src)-1] = '\0';

    /* BUG: size equals sizeof(CharVoid), but destination is only 16 bytes */
    memcpy(s.charFirst, src, sizeof(CharVoid));

    /* In the bad case, AddressSanitizer should report a stack-buffer-overflow here.
       Even if it continues, our pointers likely got clobbered. */
    print_struct(&s, "after");
}
```

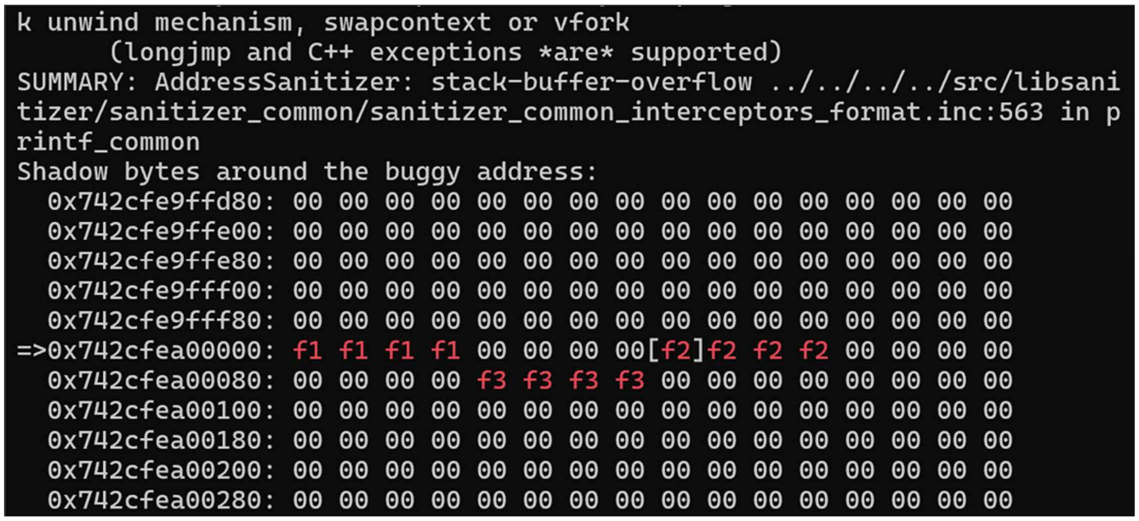
Code Fix done:



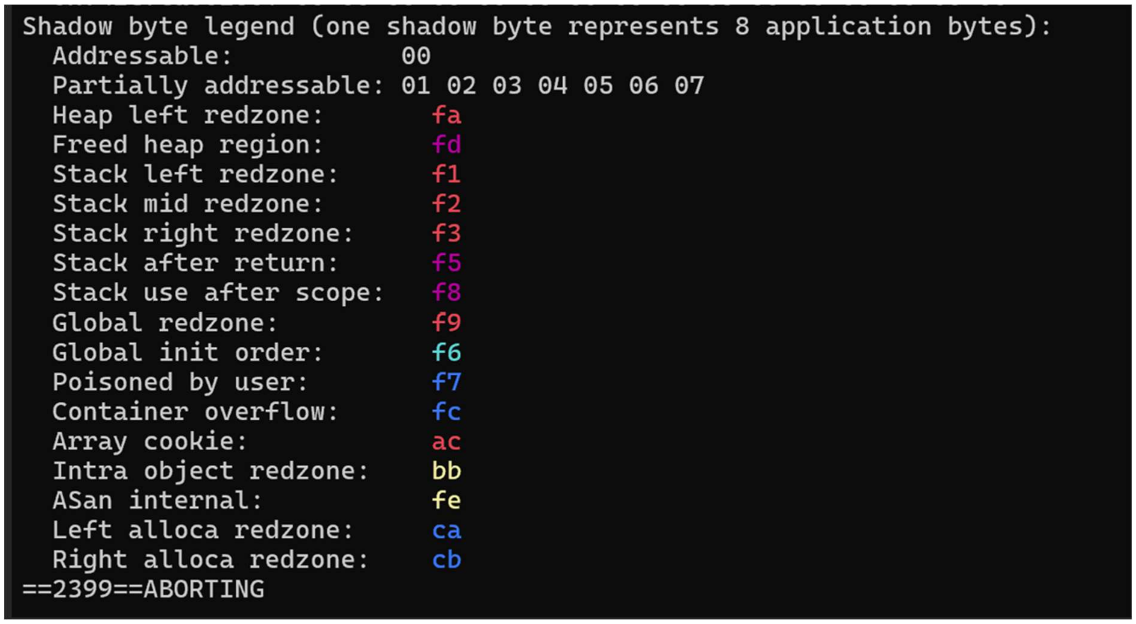
```
/* GOOD: Copies only what fits into charFirst[], and NUL-terminates */
static void demo_memcpy_good(void) {
    puts("== memcpy GOOD demo ==");
    CharVoid s;
    memset(&s, 0, sizeof(s));
    const char *message = "ORIGINAL_POINTER_DATA";
    s.voidSecond = (void*)message;
    s.voidThird  = (void*)(message + 1);

    print_struct(&s, "before");

    const char *SRC = "SAFE_COPY_DEMO";
    /* Correct: limit by destination field size minus 1, then NUL-terminate */
    size_t dstsz = sizeof(s.charFirst);
    size_t n = strlen(SRC);
    if (n >= dstsz) n = dstsz - 1;
    memcpy(s.charFirst, SRC, n);
    s.charFirst[n] = '\0';

    print_struct(&s, "after");
}
```

Implementation Results:

Output from Bad and Vulnerable code, buffer overflow happens dues to the memory copy issue.

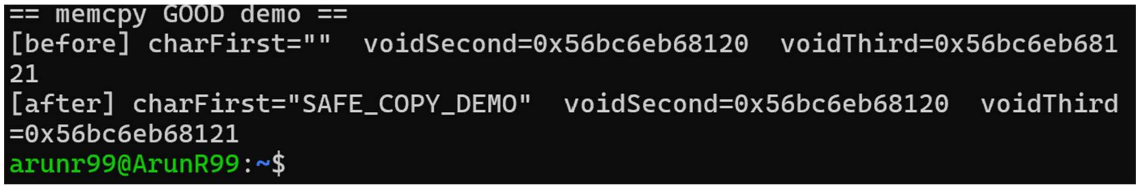


```
k unwind mechanism, swapcontext or vfork
      (longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow ../../../../src/libsani
tizer/sanitizer_common/sanitizer_common_interceptors_format.inc:563 in p
rintf_common
Shadow bytes around the buggy address:
  0x742cfe9ffd80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x742cfe9ffe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x742cfe9ffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x742cfe9fff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x742cfe9fff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x742cfea00000: f1 f1 f1 f1 00 00 00 00[f2]f2 f2 f2 00 00 00 00
  0x742cfea00080: 00 00 00 00 f3 f3 f3 f3 00 00 00 00 00 00 00 00
  0x742cfea00100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x742cfea00180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x742cfea00200: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x742cfea00280: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```



```
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==2399==ABORTING
```

Output from Fixed code, runs successfully without causing any memory copy issue as the buffer overflow has been fixed.



```
== memcpy GOOD demo ==
[before] charFirst=""  voidSecond=0x56bc6eb68120  voidThird=0x56bc6eb681
21
[after] charFirst="SAFE_COPY_DEMO"  voidSecond=0x56bc6eb68120  voidThird
=0x56bc6eb68121
arunr99@ArunR99:~$
```