Systems Programming: Memory Safety 25th June 2021 Lecture : Dr Karanja

Though almost all Operating systems are coded in C/C++, these two languages belong to a category called Memory unsafe languages . This term was popularized by Mike Hicks.

Memory Safety can be thought of in different views such as

1. A language such as C/C++ is unsafe since executing an erroneous operation causes the entire program to be meaningless, as opposed to just the erroneous operation having an unpredictable result. Memory safe languages such as Rust, Go, C#, Java, Swift, Python, and JavaScript avoid this.

2. Applications written using Memory unsafe language such as C or C++, assembly languages are vulnerable to memory errors such as buffer overflows, dangling pointers, and reads of uninitialized data. Such errors can lead to program crashes, security vulnerabilities, and unpredictable behavior.

3. Microsoft reports that over 70% of reported CVE (Common Vulnerabilities and Exposures) are due to Memory unsafe issues (read the report here https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/

4. Recent papers such Amorim et al (https://link.springer.com/chapter/10.1007/978-3-319-89722-6_4) have delved into descriptions of memory Safety with a look into factors such as 1. noninterference property and 2. Local separation of Logic .Look at them .

5. Memory safety is the property of a program where memory pointers used always point to valid memory, i.e. allocated and of the correct type/size. Memory safety is a correctness issue . A deeper look into this leads us to the concept of Garbage collection in programming languages. The garbage collector attempts to reclaim memory which was allocated by the program, but is no longer referenced . Thus Garbage collector languages such as Java and Are memory safe!

Let's dive into this popular textbook example here in C language on Vector Library

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

// There are at least 7 bugs relating to memory on this snippet.
// Find them all!

// Vec is short for "vector", a common term for a resizable array.
```

```
// For simplicity, our vector type can only hold ints.
typedef struct {
  int* data;     // Pointer to our array on the heap
  int  length;   // How many elements are in our array
  int  capacity; // How many elements our array can hold
} Vec;

Vec* vec_new() {
  Vec vec;
  vec.data = NULL;
  vec.length = 0;
  vec.capacity = 0;
  return &vec;
}

void vec_push(Vec* vec, int n) {
  if (vec->length == vec->capacity) {
    int new_capacity = vec->capacity * 2;
    int* new_data = (int*) malloc(new_capacity);
    assert(new_data != NULL);

    for (int i = 0; i < vec->length; ++i) {
      new_data[i] = vec->data[i];
    }

    vec->data = new_data;
    vec->capacity = new_capacity;
  }

  vec->data[vec->length] = n;
  ++vec->length;
}

void vec_free(Vec* vec) {
  free(vec);
  free(vec->data);
}

void main() {
  Vec* vec = vec_new();
  vec_push(vec, 107);

  int* n = &vec->data[0];
  vec_push(vec, 110);
```

```
  printf("%d\n", *n);

  free(vec->data);
  vec_free(vec);
}
```

The above textbook example though a valid C program , it has 7 issues as listed here under.

1 vec_new: vec is stack-allocated. This is an example of a dangling pointer. The line Vec vec; allocates the struct on the current stack frame and returns a pointer to that struct, however the stack frame is deallocated when the function returns, so any subsequent use of the pointer is invalid. A proper fix is to either heap allocate (malloc(sizeof(Vec))) or change the type signature to return the struct itself, not a pointer.

2. vec_new: initial capacity is 0. When vec_push is called, the capacity will double, but 2 * 0 = 0, resulting in no additional memory being allocated, so space for at least 1 element needs to be allocated up front.

3. vec_push: incorrect call to malloc. The argument to malloc is the size of memory in bytes to allocate, however new_capacity is simply the number of integers. We need to malloc(sizeof(int) * new_capacity).

4. vec_push: missing free on resize. When the resize occurs, we reassign vec->data without freeing the old data pointer, resulting in a memory leak.

5. vec_free: incorrect ordering on the frees. After freeing the vector container, the vec->data pointer is no longer valid. We should free the data pointer and then the container.

6. main: double free of vec->data. We should not be freeing the vector's data twice, instead only letting vec_free do the freeing.

7. main: iterator invalidation of n. This is the most subtle bug of the lot. We start by taking a pointer to the first element in the vector. However, after calling vec_push, this causes a resize to occur, freeing the old data and allocating a new array. Hence, our old n is now a dangling pointer, and dereferencing it in the printf is memory unsafe. This is a special case of a general problem called iterator invalidation, where a pointer to a container is invalidated when the container is modified

Online reading References

On 4 challenges of Memory Safety : read
https://initialcommit.com/blog/memory-safety-programming

On Memory leaks : https://developer.ibm.com/technologies/systems/articles/au-toughgame/

Have a splendid day and Safari rally moments !