Lab 3 - Memory Safety

> Secure System Development - Spring 2025

In this lab, we will practice debugging memory-related issues with Valgrind.

- Create a `.md` step-by-step report of the actions you took with screenshots of key results.

Recommended reading: https://en.wikipedia.org/wiki/Memory_safety

Task 1 - Getting Started

> Guide: https://valgrind.org/docs/manual/quick-start.html

1. Install needed tools: `gcc` and `valgrind` using the package tool of your distro.
2. Create `program1.c` with the following content:

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

void program1(int N) {
    int *arr = malloc(N);
    for(int i = 0; i < N; i++) {
        arr[i] = i * i;
        printf("arr[%d] = %d\n", i, arr[i]);
    }
}

int main() {
    program1(4); // Should print the array [0, 1, 4, 9]
}
```

3. Compile with the following command. Briefly explain what each flag does.

```
gcc -Wall -Werror -g -std=c99 -o program1 -O0 program1.c
```

4. Run the program. Check if it works normally or clarify if it doesn't.

```
./program1
```

5. Run the program again with `valgrind`, supplying any needed flags.

> Valgrind runs the program in a sandbox to analyze for memory-related issues.

6. Explain the output, does the program have memory-related issues?
    - Are the issues related to an existing CWEs? If yes, provide a reference to them.
7. Propose a fix for any found issues.
8. Verify that your fix works by re-running `valgrind`. It should no longer report findings.

Task 2 - More Programs

For each of the following programs, do the following:

- Compile the program in a similar way as in Task 1.
- Run the program normally once, and with `valgrind` another time. Explain the outputs.
- Are there memory-related issues? Explain and propose a fix for any found issues.
    - Are the issues related to an existing CWE? If yes, provide a reference to it.
- Verify that your fix works by re-running `valgrind`. It should no longer report findings.

Program 2

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

void work(int* arr, unsigned N) {
    for(int i=1; i<N; i++) {
        arr[i] = arr[i-1] * 2;
    }
    free(arr);
}

void program2(unsigned N) {
    int* arr = (int*)malloc(N * sizeof(*arr));
    memset(arr, 0, sizeof(*arr));
    arr[0] = 1;
    work(arr, N);
    for(int i=0; i<N; i++) {
        printf("arr[%d] = %d\n", i, arr[i]);
    }
}

int main() {
    program2(4); // Should print the array [1, 2, 4, 8]
}
```

Program 3

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

void* program3(unsigned N) {
    void *arr = malloc(N * sizeof(*arr));
    if((N < 1) || (arr = NULL)) {
        printf("%s\n", "Memory allocation falied!");
        return NULL;
    }
    printf("%s\n", "Memory allocation success!");
    return arr;
}

int main() {
    int* arr = (int*)program3(4); // Should typically succeed
    free(arr);
```

```
}
```

Program 4

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

char* getString() {
    char message[100] = "Hello World!";
    char* ret = message;
    return ret;
}

void program4() {
    printf("String: %s\n", getString());
}

int main() {
    program4();
}
```

Task 3 - Vulnerable HashMap Library

Given `hash.c` and `hash.h` for a broken and vulnerable implementation of a HashMap, do the following:

1. Identify, explain, and fix at least 4 potential CWEs in the code.
    - Include in report: CWE number, name, and relevant code snippet.
2. Implement improvements and better practices for any found bad practices (e.g., wrong data types, implicit castings, unchecked return values).
    - You may refer to https://securecoding.cert.org/ for guidelines.
3. Ensure that the code compiles and runs successfully without problem reports from `valgrind`.
    - Include in report: a screenshot from your terminal for the correctly-working code.

Task 4 - Bonus

> Guide: https://ropemporium.com/guide.html

- Create a writeup for the challenge ret2win at ROP Emporium.
- Clearly explain your process, all used tools, and relevant background information.