Hacking in C

Assignment 3, Thursday, Februari 14, 2019

Handing in your answers: Submission via Brightspace (http://brightspace.ru.nl)

Deadline: Thursday, Februari 21, 23:59 (midnight)

1. Recall from the lecture that there is no default initialization on the stack. There is also no cleanup, so by reading memory below the current stack frame directly before and after a function call, you can learn things about that function.

   Consider the following code snippet:

   ```
   extern void magic_function();
   int main (void)
   {
     ...
     magic_function();
     ...
   }
   ```

   Write this snippet to a file called `exercise1.c` and create a `magic_function()` in `magic_function.c`. Complete the program such that it prints the amount of bytes of stack space used by `magic_function`. When grading, we will use your program with our own implementations of `magic_function`.
   Hint 1: The program in `exercise1.c` should not assume anything about `magic_function`, except that it does not receive any arguments and you do not use its return value.
   Hint 2: You should try with some own implementations of `magic_function`. However, compilers are smart. Due to optimizations your function might end up using no stack space at all. To prevent this, make sure your `magic_function` does something meaningful with its local variables (e.g. add them).
   Hint 3: You may assume that `magic_function` does not use more than 4 MB (4194304 bytes) of stack space.
   Hint 4: You may need to compile with compiler option `-fno-stack-protector`.
   Hint 5: Implement `magic_function()` in a separate file and compile and *link* them as follows:

   ```
   $ gcc -c -o magic_function.o magic_function.c
   $ gcc -o exercise1 exercise1.c magic_function.o
   ```

2. This exercise is about the size of heap space available to a program.

   (a) Write a program (in a file called `exercise2.c`), which determines the maximal amount of heap space that can be allocated in one call to `malloc`. The output of the program should be of the following form (where `XXX` is replaced by the correct number):

   ```
   One malloc can allocate at most XXX bytes.
   ```

   (b) Is the output of the program always the same? Explain why. Write your answer to a file called `exercise2b`.

   (c) If you would repeat your experiment with `calloc`, would you get the same result? What is the difference? Would it be better to use `calloc` or `malloc` in normal use?

3. Consider the following program:

   ```
   int main() {
     int32_t x[4];
     x[0] = 23;
     x[1] = 42;
     x[2] = 5;
     x[3] = (1<<7);
   ```

```
  printf("%p\n", x);          // prints 0x7ffb3cc3b20
  printf("%p\n", &x);         // (a)
  printf("%p\n", x+1);        // (b)
  printf("%p\n", &x+1);       // (c)
  printf("%d\n", *x);         // (d)
  printf("%d\n", *x+x[2]);    // (e)
  printf("%d\n", *x+*(x+3));  // (f)
  return 0;
}
```

Assume that the first call to `printf` prints `0x7fffb3cc3b20`. What do the other 6 calls to `printf` print? **Explain your answers.** Write your answer to a file called `exercise3`.

4. Consider the following C code:

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

int main(void)
{
  int i,j;
  unsigned long long **m;
  unsigned long long **mt;

  while(1)
  {
    // allocate matrix m
    m = malloc(1000*sizeof(unsigned long long*));
    if(m == NULL) return -1;
    for(i=0;i<1000;i++)
    {
      m[i] = malloc(1000*sizeof(unsigned long long));
      if(m[i] == NULL) return -1;
    }

    // allocate matrix mt
    mt = malloc(1000*sizeof(unsigned long long*));
    if(mt == NULL) return -1;
    for(i=0;i<1000;i++)
    {
      mt[i] = malloc(1000*sizeof(unsigned long long));
      if(mt[i] == NULL) return -1;
    }

    for(i=0;i<1000;i++)
      for(j=0;j<1000;j++)
        m[i][j] = 1000*i+j;

    // transpose matrix m, write to mt
    for(i=0;i<1000;i++)
      for(j=0;j<1000;j++)
        mt[i][j] = m[j][i];

    // free matrices m and mt
    free(m);
    free(mt);
  }
```

```
  return 0;
}
```

(a) Write the code to a file called `exercise4.c` or download it from `https://git.io/fhQ66`.

(b) Compile and run the code; describe what happens and explain why. Write your answer to a file called `exercise4`.

(c) Explain how to fix the problem in this code and add that description to your answer file `exercise4`.

5. Consider the following code snippet:

```
int main(void) {
  char *s1 = malloc(9);
  if (s1 == NULL) return 1;
  char *s2 = malloc(9);
  if (s2 == NULL) return 1;

  strcpy(s1, "s0123456");
  strcpy(s2, "s0123456");

  // do your attack

  printf("student 1: %s\n", s1);
  printf("student 2: %s\n", s2);
  return 0;
}
```

(a) Copy this snippet to a file called `exercise5.c`.

(b) Replace the student numbers in the `main` function. Why do we use `strcpy` and do we not write this as `s1 = "s0123456"`? Write your answer to a file called `exercise5`.

(c) Why do we need to allocate 9 bytes to host the 8-character strings? Add your answer to the file `exercise5`.

(d) By modifying something in memory, you can make the first `printf` print out *both* student numbers. Modify the program where the comment `//do your attack` is. Hand in your modified program. **Hint:** It's very likely that these two are close in the memory.

6. Place the files

- `exercise1.c`,
- `exercise2.c`,
- `exercise2b`, and
- `exercise3`
- `exercise4`
- `exercise5`
- `exercise5.c`

in a directory called `hic-assignment3-STUDENTNUMBER1-STUDENTNUMBER2` directory (as in the previous assignments, replace `STUDENTNUMBER1` and `STUDENTNUMBER2` by your respective student numbers). Make a `tar.gz` archive of this directory and submit the archive in Brightspace.