Hacking in C

Assignment 2, Tueday, April 21, 2020

Handing in your answers: Submission via Brightspace (https://brightspace.ru.nl). Make sure you are enrolled in a group.

Deadline: Tuesday, May 12, 12:30:00

1. You are given the following code fragment:

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

int main() {
  int32_t a = 1;
  int16_t b = 2;
  unsigned char c = 3;
  int8_t d = 4;
  unsigned long long e = 5;
  short f[3] = {6, 6, 6};
  long g = 7;
  unsigned long int h = 8;
  uint8_t i = 9;

  fprintf(stderr, "address\t\tvariable\tvalue\tsizeof\tnext_addr\n");

  return 0;
}
```

   (a) Write this code snippet to a file called `exercise1.c`.

   (b) Create a `Makefile` that compiles this to an executable called `exercise1`. Make sure you compile with `-Wall -Wextra -pedantic`.

   (c) Add code to the `main` function such that you print out the following information for each variable:

```
address           variable        value    sizeof   next_addr
0x7fff........    a               1        4        0x7ff.........
```

   - Make sure you use the most specific format string specifier for each type. For example, print an `int` with `%d`, and a long with `%ld`.
   - To compute the next address correctly, you will need to cast the pointers to the special `uintptr_t` before you add the size of the type. Arithmetic operations on pointers have special semantics otherwise: they are implicitly multiplied with the size of the type. You will see this in the third lecture and in exercise 3. You can print an `uintptr_t` using the `PRIxPTR` format string.

   (d) What happens when you take `sizeof(f)`?

   (e) Use the `sort` shell program to sort the output of the `exercise1` program based on the memory addresses. Also, write the output to `exercise1.txt`. Note that because the `address` line is printed to `stderr`, it will not be affected by pipes.

   (f) The compiler re-ordered the variables. Can you make a guess why this happened? Add your answers to `exercise1.txt`

   (g) Are there any gaps between variables in memory? Can you again guess why this happened? Add your answers to `exercise1.txt`.

2. Since the C99 standard, the C programming language has a `bool` data type. Programs that use this data type have to include the file `stdbool.h`. Write a program (in a file called `exercise2.c`), which finds out about the internal representation of bool. Specifically, your program shall print the following:

- How many bytes does a `bool` use?
- What hexadecimal representation does a `bool` have, if you set it to `true`?
- What hexadecimal representation does a `bool` have, if you set it to `false`?
- Can you assign other hexadecimal values than these two to a `bool` variable? Are those interpreted as `true` or as `false` or do they cause an error?

Add the appropriate compilation instructions to your `Makefile`.

3. Write a program in `exercise3.c`. It should print out `This computer uses two-complement representation` if the computer is being run on uses two-complement representation for signed integers. If the computer is not using two-complement representation, print `This computer doesn't use two-complement representation`. Do not assume that *all* computers use two-complement representation, when writing this solution.

   Hint:

   - The ~ operator flips all bits.

4. This exercise is about pointer arithmetic. Pointer arithmetic will be further discussed in lecture 3, but you can also for example watch https://www.youtube.com/watch?v=gv-y9hIhvq4.

   You are given the following code fragment:

```c
int main (void)
{
  short i = 0x1234;
  char x = -127;
  long sn1 = <STUDENT NUMBER OF TEAM MEMBER 1, WITHOUT LEADING S>;
  long sn2 = <STUDENT NUMBER OF TEAM MEMBER 2, WITHOUT LEADING S>;
  int y[2] = {0x11223344,0x44332211};

}
```

   (a) Write this code snippet to a file called `exercise4.c`.

   (b) Set the values of `sn1` and `sn2` to your student numbers.

   (c) Extend the functionality of the program to print the memory layout of all of the local variables, in a byte-by-byte fashion, so a four-byte integer becomes four lines. More specifically, your program should print a table of the following form (addresses and data are fictional):

```
address     content (hex)     content (dec)
-------------------------------------------
0x...00     0xFF              255
0x...01     0x12               18
0x...02     ...               ...
```

   Note that you will have to rely on undefined behaviour to get this done. You do not have to sort the output.

   Hint: To walk through memory byte-by-byte, you will want to use a pointer of type `char*` or `uint8_t*`.

   (d) Compile your program with `gcc -O3 -Wall` and run the program. Write the output of the program to a file called `exercise4.out`. Explain which variable is stored at which location in memory and write this explanation to a file called `exercise4.txt`

5. Write all parts of this exercise into a file called `exercise5.c`. For testing you should write a suitable `main` function and perhaps some other test functions. If you want an extra challenge, you can put this test code in separate C files. Add compiling `exercise5.c` and your test code to your `Makefile`.

   (a) Consider the following function `addvector`, which is adding elements of two vectors of length `len`:

```c
void addvector(int *r, const int *a, const int *b, unsigned int len)
{
  unsigned int i;
  for(i=0;i<len;i++)
  {
    r[i] = a[i] + b[i];
  }
}
```

Rewrite this function to use pointer arithmetic instead of array indexing with bracket notation.

(b) Write your own version of the `memcmp` standard C library function. Don't use any array indexing with bracket notation but only pointer arithmetic.
For documentation of this function, see `man memcmp` or http://pubs.opengroup.org/onlinepubs/009695399/functions/memcmp.html.

(c) Now write a function called `memcmp_backwards` with the same signature as memcmp. This function shall compare the two input byte arrays backwards, i.e., the sign of a non-zero return value shall be determined by the sign of the difference between the values of the *last* pair of bytes that differ in the objects being compared.
Again, don't use any array indexing with bracket notation but only pointer arithmetic.

(d) **(optional)** For an additional challenge, think about how to make the `memcmp` function faster for long input arrays. As a hint, consider that in C it takes 1 operation to compare two values of a basic type (e.g. `char`, `int`) regardless of that type. If you decide to submit a solution to this part, write it into a function `memcmp_fast`, also in the file `exercise3.c`. Again, don't use any array indexing with bracket notation but only pointer arithmetic.

(e) **(optional)** For yet another challenge, think about how to ensure that the time taken by the `memcmp` function only depends on the length of the inputs, not on the values in the input arrays.
This "constant time" property is often important for security-sensitive code. If you decide to submit a solution to this part, write it into a function `memcmp_consttime`, also in the file `exercise3.c`. Feel free to use array indexing for this part.

6. Place the files you created for this assignment in a directory called `hic-assignment2-STUDENTNUMBER1-STUDENTNUMBER2` (again, replace `STUDENTNUMBER1` and `STUDENTNUMBER2` by your respective student numbers). Do not include binaries. Make sure to include the `Makefile` that (with a single invocation of `make` in the `hic-assignment2-STUDENTNUMBER1-STUDENTNUMBER2` directory) compiles all the C programs in this exercise. Make sure that this `Makefile` is also in the `hic-assignment2-STUDENTNUMBER1-STUDENTNUMBER2` directory.
Make a `tar.gz` archive of the whole `hic-assignment2-STUDENTNUMBER1-STUDENTNUMBER2` directory and submit this archive in Brightspace.