Software and Web-Security

Assignment 3, Monday, February 23, 2015

Handing in your answers: Submission via Blackboard (http://blackboard.ru.nl)

Deadline: Monday, March 2, 24:00 (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:

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

   Write this snippet to a file called `exercise1.c`. Complete the program such that it prints the amount of bytes of stack space used by `magic_function`.
   Hint 1: You do not know 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, then return the result), and
   - implement your `magic_function` in a separate source file, and compile with separate compilation and linking steps. E.g:

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

   When grading, we will use your program with our own implementations of `magic_function`.
   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`.

2. This exercise is about pointer arithmetic. Write all parts of this exercise into a file called `exercise2.c`. For testing you probably need to write a `main` function; however, this should be in a separate file, which is not part of the submission.

   (a) Consider the function `addvector.c` shown in the first werkcollege (the code is online on the lecture's website). 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 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 fast for long input arrays. If you decide to submit a solution to this part, write it into a function `memcmp_fast`, also in the file `exercise2.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.
If you decide to submit a solution to this part, write it into a function `memcmp_consttime`, also in the file `exercise2.c`. Feel free to use array indexing for this part.

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);
  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. Place the files

- `exercise1.c`,
- `exercise2.c`, and
- `exercise3`

in a directory called `sws1-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 Blackboard.