Secure and Dependable Systems
Jacobs University Bremen
Dr. Jürgen Schönwälder

Course: CO21-320203
Date: 2020-03-12
Due: 2020-03-19

SADS 2020 Problem Sheet #4

Problem 4.1: *Intel function disassembly and stack frame layout* (6+2+2 = 10 points)

Consider the C functions `foo()` shown below.

```
1  int foo(int a)
2  {
3      int i, r;
4
5      r = 0;
6      for (i = 1; i <= a; i++) {
7          r += i;
8      }
9      return r;
10 }
```

The source code has been compiled using gcc 8.3.00 on Debian 10.3 for an Intel x86_64 processor. Note that an `int` is 32 bits long on this system.

a) Explain the disassembled x86_64 machine code shown below by writing a meaningful comment for each assembler statement and how it relates to the C code shown above.

```
(gdb) disassemble
Dump of assembler code for function foo:
   0x0000555555555125 <+0>:     push   %rbp
   0x0000555555555126 <+1>:     mov    %rsp,%rbp
   0x0000555555555129 <+4>:     mov    %edi,-0x14(%rbp)
   0x000055555555512c <+7>:     movl   $0x0,-0x8(%rbp)
   0x0000555555555133 <+14>:    movl   $0x1,-0x4(%rbp)
   0x000055555555513a <+21>:    jmp    0x555555555146 <foo+33>
   0x000055555555513c <+23>:    mov    -0x4(%rbp),%eax
   0x000055555555513f <+26>:    add    %eax,-0x8(%rbp)
   0x0000555555555142 <+29>:    addl   $0x1,-0x4(%rbp)
   0x0000555555555146 <+33>:    mov    -0x4(%rbp),%eax
   0x0000555555555149 <+36>:    cmp    -0x14(%rbp),%eax
   0x000055555555514c <+39>:    jle    0x55555555513c <foo+23>
   0x000055555555514e <+41>:    mov    -0x8(%rbp),%eax
   0x0000555555555151 <+44>:    pop    %rbp
   0x0000555555555152 <+45>:    retq
End of assembler dump.
```

b) Explain the layout of the stack frame when `foo()` is called. Create a diagram showing where the variables, parameters, old base pointer, and return address are located in the stack frame.

c) The stack pointer (rsp) and the base pointer (rbp) are the same during the function execution while data is stored “on top of the top of the stack”. Why is this possible in this case?