University of Stuttgart
Institute of
Information Security

System and Web Security
Summer Term 2018
Prof. Dr. Ralf Küsters

Homework 1

Submission via ILIAS until April 24, 2018, 07:59.

General Notes

- If you encounter difficulties, you SHOULD[1] ask the teaching assistants in the consultation hour.
- To solve the homework, you SHOULD form teams of 3 people.
- Your team size MUST NOT exceed 3 people.
- You MUST submit your homework in ILIAS. (In the ILIAS course, you can find the *Homework Submissions* item. There, you can register your team and submit your solution.)
- You are free to choose whether you write your solutions in German or in English.
- Your submission MUST include all team member's names and matriculation numbers.
- You MUST use the System Security Virtual Machine to work on all problems (except if stated otherwise).
- You MUST submit your solution as a PDF or TXT document. If your solution contains source code, you MUST submit your source code in its native file format separately. In your main document, it is sufficient to point to these source code files.
- When you carry out attacks, you SHOULD take a screenshot to document that your exploit worked (and include this screenshot in your submission).
- Your team gains one point for each problem if solved reasonably.
- If you do not adhere to these rules, you risk loosing points.

Information for Solving the Problems

- Many problems of the homework depend on the computer architecture you use. In ILIAS, you can download the *System Security Virtual Machine (VM)*. This VM serves as our reference system. Other systems may (or most likely will) produce different results that we cannot grade as correct. **You MUST use this VM to work on all problems (except if stated otherwise).**

- Please note that you can find further information on how to install the VM in ILIAS.

- When you compile source code, you MUST use `gcc` with the following options:

  -**fno-stack-protector** disables several defence mechanisms

  -**O0** disables optimizations

  -**ggdb3** adds meta information for the debugger to the binary

  Although we have configured the shell of the VM such that the `gcc` command uses these options automatically, you should not omit these (especially when writing your own make files, etc.). For example, you compile a C program (contained in file `myprog.c`) like this:

  ```
  gcc -fno-stack-protector -O0 -ggdb3 -o myprog myprog.c
  ```

[1]SHOULD, MUST, and MUST NOT are used as defined in RFC2119.

Problem 1: Disassembling with gdb

Consider the C program below with one modification: Replace `X` in Line 5 with the sum of the ASCII codes of the first letter of each team member's given name.

Compile this C program with the command below in the system security virtual machine (please have a look at the advice on the first page of this homework sheet). This command creates the executable binary file `a.out`.

```
gcc -fno-stack-protector -O0 -ggdb3 c-simple-c-example2.c
```

Open the compiled program with the command `gdb a.out`. Retrieve the assembly code of the main function with the gdb command `disassemble main`.[2]

Show the assembly code and state a hypothesis which lines of the assembly code correspond to Lines 5, 6, and 7 in the C program, respectively.

```
int main() {
  int a;
  int b;
  int c;
  a = X;
  b = 3;
  c = a+b;
  return 0;
}
```

Problem 2: x86 Assembly Code (I)

Compile the C program below. Disassemble the `main` and the `myfunction` functions with the `disassemble` command in `gdb`. Give a brief explanation of all occurring assembly instructions in their respective context.

```
#include <stdio.h>

void myfunction(int a) {
        long some_long;
        char * some_strings[2];
        some_long = 42;
        some_strings[0] = "foo";
        some_strings[1] = "bar";
        some_long -= 19;
        printf("%ld\n%d\n%s\n", some_long, a, some_strings[0]);
}
int main () {
        int x = 0;
        myfunction(x);
}
```

[2]The syntax of gdb is often called AT&T or GAS syntax. In the literature in ILIAS, you find a link to the *Wikibook: x86 Assembly*, which explains many instructions and notations.

Problem 3: x86 Assembly Code (II)

Below, you find the assembly code of the function `blub`. What is the functionality of this function? Translate the assembly code of this function into C code. Complete your C code with a `main` function that calls `blub`. (Note that, if you disassemble your compiled C program, you likely get different assembly code. This is fine as long as your code provides the same functionality.)

```
Dump of assembler code for function blub:
0x080483c4 <blub+0>:    push   %ebp
0x080483c5 <blub+1>:    mov    %esp,%ebp
0x080483c7 <blub+3>:    sub    $0x18,%esp
0x080483ca <blub+6>:    movl   $0x17,-0x4(%ebp)
0x080483d1 <blub+13>:   mov    -0x4(%ebp),%eax
0x080483d4 <blub+16>:   sub    $0xf,%eax
0x080483d7 <blub+19>:   mov    %eax,-0x8(%ebp)
0x080483da <blub+22>:   mov    -0x8(%ebp),%eax
0x080483dd <blub+25>:   mov    %eax,0x4(%esp)
0x080483e1 <blub+29>:   movl   $0x80484d0,(%esp)
0x080483e8 <blub+36>:   call   0x80482f8 <printf@plt>
0x080483ed <blub+41>:   leave
0x080483ee <blub+42>:   ret
End of assembler dump.
```

Problem 4: Stack Frames

Compile the C program below and analyze a program run. Have a look at the stack immediately before Line 5 is executed. Draw the structure of the stack frames of `main` and `do_something`. This figure has to show all memory cells of these stack frames (i.e., byte by byte) and has to include all local variables, arguments, stack pointer, frame pointer, return address, and the old frame pointer. (Note that the stack frame of `main` does not have a return address or an old frame pointer. The gdb command `info frame` can be useful to solve this problem.)

```c
void do_something(int a, int * b) {
  int c;
  int d;

  c = a + *b;
}

int main() {
  int e;
  int f;
  int g;

  e = 42;

  do_something(23, &e);
}
```