University of Stuttgart
Institute of
Information Security

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

Homework 5

Submission via ILIAS until May 29, 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: Reading Data with Format Strings

Consider the C program below, which implements a simple user authentication.

File: code/fstr–read–2.c

```c
#include <stdio.h>
#include <string.h>
int try_login(char * user, char * pass,
              char * validuser, char * validpass) {
  int logged_in = 0;
  if (strcmp(user, validuser) == 0 &&
      strcmp(pass, validpass) == 0)
    logged_in = 1;
  else {
    printf("Username or password invalid. Username: ");
    printf(user);
    printf("\n");
  }
  return logged_in;
}
main(int argc, char * argv[]) {
  if (argc != 3) {
    printf("No username and password!\n");
    return -1;
  }
  if (try_login(argv[1], argv[2], "root", "passwd") == 1)
    printf("Login successful. Have fun!\n");
  else
    printf("ACCESS DENIED!\n");
}
```

Provide a string `str` such that the command `./fstr-read-2 'str'` outputs the (valid) login credentials. Also retrieve the username and password in the same manner from the program `fstr-read-2-extra`, which uses the same code as above.

Hint: The program `fstr-read-2-extra` must be marked as executable after being copied into the virtual machine. You can do this with the following command: `chmod +x fstr-read-2-extra`

Problem 2: Writing Data with Format Strings

Consider the C program below:

File: code/fstr–write–2.c

```c
#include <stdio.h>
#include <string.h>
int try_login(char * user, char * pass,
              char * validuser, char * validpass) {
  int status = 0x42; // Logged out
  if (strcmp(user, validuser) == 0 &&
      strcmp(pass, validpass) == 0)
    status = 0x17; // Logged in
  else {
    printf("Username or password invalid. Username: ");
    printf(user);
    printf("\n\n&status=%08x, status=%08x\n\n",
           &status, status);
  }
  return status;
}
main(int argc, char * argv[]) {
  if (argc != 3) {
    printf("Provide username and password!\n");
    return -1;
  }
  if (0x17 == try_login(argv[1], argv[2], "root", "geheim"))
    printf("Login successful. Have fun!\n");
  else
    printf("ACCESS DENIED.\n");
}
```

Circumvent the password protection by chosing the username `user` such that the variable status is set to 0x17. Use the following console command as a template for your exploit (line breaks have been added for readability—they are not part of the command):

```
./fstr-write-2 $(echo -e "
AAAA\x??\x??\x??\x??
BBBB\x??\x??\x??\x??
CCCC\x??\x??\x??\x??
DDDD\x??\x??\x??\x??
%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x%08x
%??x%n%??x%n%??x%n%??x%n") password
```

In particular, you have to

- determine the correct values for the target addresses (`\x??`),
- determine the correct length of the stack-pop sequence (`%08x`),
- determine the correct output length for the output of the dummies (`%??x`),
- and—if needed—add dummy characters to align the string.

Hint: The stack-pop sequence can be very long. You can easily determine the position of the dummy values `AAAA`, `BBBB`, ... on the stack by looking for output snippets of 41414141, 42424242, ... when adjusting the stack-pop sequence. Also, it might be helpful to replace `%n` with `%x` during testing.

Problem 3: Effectiveness of Address Space Layout Randomization (ASLR)

In stack smashing attacks, the attacker needs to guess an address of the NOP sled in the exploit string that is placed on the stack. ASLR, however, is supposed to make this guess hard. In this task, we will analyze ASLR's effectiveness from a theoretical point of view.

Let s be the number of bits of ASLR's random shift, i.e., an offset o is chosen equally from $\{0, \ldots, 2^s - 1\}$ and o is added to each address. Let l be the length of the exploit string's NOP sled. The attacker guesses an address of this NOP sled (leaving out that for a successful attack, he may also guess the first byte of the shellcode). Assume that the NOP sled starts at address x when ASLR is disabled and that the attacker knows x, but not o. What is the probability for the attacker to guess an address of the NOP sled? *Hint:* The probability distribution consists of three parts.

Consider a system that allows the attacker to infinitely try to guess an address, but each time the offset o is chosen freshly. On average, how many guesses does an attacker need who follows the best[2] guessing strategy? Calculate this value for the table below.

$s \setminus l$	128	512	1024	2048	4096
16					
20					
24					

What is your verdict on the effectiveness of ASLR?

[2]Strictly speaking, there are multiple best strategies.