University of Stuttgart
Institute of
Information Security

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

Homework 6

Submission via ILIAS until June 5, 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: Static Code Analysis

Read the paper "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities" by Wagner et al.[2] Focus on Sections 2 and 3. Afterwards, you should be familiar with the constraint language and generating a constraint system from a C program.

Consider the C program below:

File: code/def–static–wagner–1.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
main(int argc, char *argv[]) {
  char *in;
  if (argc < 2) {
    printf("At least one argument required!\n");
    return -1;
  }
  if (atoi(argv[1]) == 1) {
    in = malloc(64);
    printf("Enter argument: ");
    fgets(in,64,stdin);
  }
  else {
    in = malloc(32);
    printf("Enter argument: ");
    fgets(in,32,stdin);
  }
  /* Do something with in */
  printf("in=%s\n",in);
}
```

1. Analyze the program with the technique by Wagner et al. Show the constraint system.
2. Analyze the program manually (without the technique mentioned above). Can buffer overflows occur? Do you come to a different conclusion than the technique by Wagner et al.? If yes, explain why the results differ.

[2]Available at https://people.eecs.berkeley.edu/~daw/papers/overruns-ndss00.pdf

Problem 2: Return-to-libc

Attack the C program below using *return-to-libc* such that a shell is opened. Show your exploit string and document the attack as usual.

File: code/rtlc–exploit–1–target.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void foo(char *src) {
  char buffer[8];
  strcpy(buffer, src);
}

int main(int argc, char** argv) {
  int i;
  printf("Address of system: %p\n", system);
  for (i = 0; i < argc; i++) {
    printf("Address of argument %d: %p\n", i, argv[i]);
  }
  if (argc == 2) {
    foo(argv[1]);
  }
  return 0;
}
```

Problem 3: Return-oriented Programming

The C program below takes a filename as the first argument and copies the contents of this file to `buffer`.

File: code/rop–exploit–1–target.c

```c
#include <string.h>
#include <stdio.h>
#include <dlfcn.h>

void foo(char* src) {
  char buffer[32];
  static FILE* f;
  static char* ptr;
  f = fopen(src, "r");
  ptr = buffer;
  while ( fread(ptr++, sizeof(char), 1, f) > 0 );
  fclose(f);
}

int main(int argc, char** argv) {
  void* libc = dlopen("/lib/libc-2.7.so", RTLD_NOW);
  void* address = dlsym( libc, "__libc_init_first");
  printf("Address of <__libc_init_first>: %p\n", address);
  if(argc > 1) {
    foo(argv[1]);
  }
  printf("Done.\n");
  return 0;
}
```

Create a file that contains a ROP chain such that a shell is opened when the program above is executed with this file. Document your attack.

Note: You need to add the option `-ldl` to compile the C program above. Hence, the full command to compile this file is as follows:

```
gcc -fno-stack-protector -ggdb3 -O0 -ldl rop-exploit-1-target.c
```

Hints:

- Recall that for dynamic libraries, the contents of the respective file are mapped into memory to a certain address range. Hence, a function of such a library can be found at the address that results from the function's offset within the library file and the address at which this library file begins in the program's memory. You can determine the offset of a function within a library file with the command `objdump -D file` that disassembles the contents of that file and lists (among others) all functions with their respective offset within this file.
- Use `ROPgadget` to analyze the binary file `/lib/libc-2.7.so` that contains the dynamic library `libc`. (Have a look at the command line options of `ROPgadget` with the command `ROPgadget --help`.)
- Given the offset to which the library `libc` is mapped into the program's memory, `ROPgadget` can generate a Python program that prepares a suitable ROP chain (see options `--offset` and `--ropchain`).

- `ROPgadget` is included in the virtual machine.