Hacking in C

Assignment 5, Thursday, March 15, 2018

Handing in your answers: Submission via Blackboard (`http://blackboard.ru.nl`)

Deadline: **Wednesday, March 28**, 23:59:59 (midnight)

1. In assignment 3, you had to figure out the amount of stack space a called function uses. In assignment 4, you had to alter the control flow of a program with a single malicious input so that you gained "root access". In this exercise, you will first analyze, then (optionally) manipulate the stack to influence the flow of a program. Consider the following code:

```
#include <stdio.h>

void function_b(void) {
    char buffer[4];

    // ... insert code here

    fprintf(stdout, "Executing function_b\n");
  }

  void function_a(void) {
    int beacon = 0xa0b1c2d3;
    fprintf(stdout, "Executing function_a\n");
    function_b();
    fprintf(stdout, "Executed function_b\n");
  }

  int main(void) {
    function_a();
    fprintf(stdout, "Finished!\n");
    return 0;
  }
```

Place this code in a file. Create a Makefile so that the program compiles with debug symbols turned on (`-g`), no optimizations enabled (omit any `-O` flags), and make sure the frame pointer is present (`-fno-omit-frame-pointer`): `gcc -g -fno-omit-frame-pointer exercise4.c`

Depending on your solution and the system you run on you may also have to include `-fno-stack-protector`.

When run it should print

```
Executing function_a
Executing function_b
Executed function_b
Finished!
```

(a) For this exercise you must run the program in `gdb`. `gdb`'s TUI mode (`gdb --tui a.out`) might be helpful.

Run the program until somewhere in `function_b()`. Inspect the stack frames for both the current stack frame (for `function_b()`) and the stack frame of its caller (`function_a()`). You can switch frames with the command `frame X`, where X is the number of the frame you want. For each frame, write the following information, and the steps you took to get it, to a file called `exercise1a`:

- which frame it is,
- which function the frame belongs to,
- the location (address in memory) of the frame,

- the location of the frame it was called by,
- (if applicable) the location of the frame it called,
- the value of the instruction pointer (`%eip` on 32-bit, `%rip` on 64-bit systems),
- the location of the return address,
- the value of the return address,
- the value of the base pointer (`%ebp`, `%rbp`),
- the location of the saved base pointer,
- the value of the saved base pointer,
- the value of the stack pointer, and
- the address of the local variable (try e.g. `print/x &beacon`).

You can print memory from addresses in `gdb` using `x`. To print 8 bytes at memory address 0x12345678 you would enter `x/xg 0x12345678`. See `help x` from within `gdb` for more information.

NOTE: It is easy to get this information; if you find you are using more than a few commands for each frame, ask for a hint.

(b) This exercise also has two variants, "normal" and "hard". The normal variant assumes the program will be run within gdb (without TUI mode, which breaks printing), which results in a predictable address layout. For the hard variant, run the program on the command line as normal.

Copy the program to a file called `exercise1b.c`. Change the designated section of `function_b` so that it changes the stack in such a way that the program prints

```
Executing function_a
Executing function_b
Finished!
```

Note that `Executed function_b` is not in the output. Once again, you are only allowed to insert code *before* the call to `fprintf`, and are not allowed to change the other functions. You are also not allowed to use additional print statements to generate the desired output.

Hint: The location of the return addresses, base pointers, their values and their location relative to the local variables, from exercise 2b, should help you determine what to do. However, after changing the code you will need to repeat some steps from exercise 2b to get the correct values for the new program.

2. In this assignment you will perform a remote exploit on an echo server we provide. The echo server echoes input, and as attackers you can repeatedly supply format strings or long inputs to reveal or corrupt data on the stack. Please be considerate of your fellow students: If you intentionally break the server, they cannot complete the exercise until we fix it. However, do not worry too much about this: we have taken steps to make it hard to accidentally break the server.

   Remember that these exercises do not count to your final grade, however, you should make an effort to at least partially answer all questions, and if you get stuck on a question, clearly explain why you cannot continue.

   The hostname of the echo server is hackme.cs.ru.nl, the port is 2266. If you connect to this address with e.g. `netcat` aka `nc` (`nc hackme.cs.ru.nl 2266`, `man netcat`) it will simply print everything you send back to you. For the purpose of this exercise, we went out of our way to do this insecurely. We have manually disabled several security mechanisms on the program you connect with. There is no reason to ever do this on modern systems, so you are unlikely to encounter programs in the wild which are exploited as easily as this one.

   Since we consider it relatively easy to exploit, we will not give you the source code or the binary executable of the program. Rather, you must use the knowledge gained from the previous lectures and exercises to figure out how to exploit this program with the shellcode we provide.

   The hexadecimal string representation of the (64-bit) shellcode, in an easy format to use in a C program or pass to `echo -e "<`*shellcode*`>"`, is:

```
\x48\x31\xc0\x48\x31\xd2\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68
\x48\xc1\xeb\x08\x53\x48\x89\xe7\x52\x57\x48\x89\xe6\xb0\x3b\x0f\x05
```

This shellcode translates to AT&T syntax assembly as follows:

```
#  "\x48\x31\xc0"                              // xor %rax, %rax
#  "\x48\x31\xd2"                              // xor %rdx, %rdx
#  "\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68" // mov $0x68732f6e69622f2f, %rbx
#  "\x48\xc1\xeb\x08"                          // shr $0x8, %rbx
#  "\x53"                                      // push %rbx
#  "\x48\x89\xe7"                              // mov %rsp, %rdi
#  "\x52"                                      // push %rdx
#  "\x57"                                      // push %rdi
#  "\x48\x89\xe6"                              // mov %rsp, %rsi
#  "\xb0\x3b"                                  // mov $0x3b, %al
#  "\x0f\x05"                                  // syscall
```

The target of this exercise is to write and execute an exploit based on the shellcode. Note that the entire exercise can be done without the use of C, although you may need to write some simple one- or two-line bash scripts. You can get an actual byte representation of the shellcode in a file using the command `echo -e "<shellcode>" > file`

The command `netcat` takes input from files, commands or scripts if you simply redirect the standard input, and send them over a network connection. For example, `nc hackme.cs.ru.nl 2266 < file` will establish a connection to the echo server and send the contents of `file` to it. As another example, `./exploit.sh | nc hackme.cs.ru.nl 2266` will send the output of the script `exploit.sh` to the echo server. Since you are not giving the input on the command line, all you see is what the server echoes back.

(a) First you will need to gather information on the target you are attacking. Since it is an insecure echo server, a buffer overflow is likely to work. Determine the size of the buffer the echo server is using. Write your answer, and how you obtained it, to a file `exercise2`.

(b) Since you now know the size of the buffer, you can try to use a format string attack to read the contents of the memory. Try to do so. Add the memory contents you see, and the format string you used to obtain them, to `exercise2`.

(c) Identify the buffer in the memory dump from your answer to part (b). Then, identify the return address and saved frame pointer. (Hint: the return address will point to the *code* section of memory, not the stack.) Add your answer to `exercise2`.

(d) Try to estimate the start address of the buffer from the information you now have. Also try to explain what any values between the buffer and the return address might be. Add your answers to `exercise2`.

(e) Write a shell script or C program that first sends some stored string to standard output, then allows you to input text by hand which is sent to standard output. The shell script can consist of just one or two commands. (Hint: the commands `cat` or `tee` may be useful.) Save your script to `exercise2e.sh`, or your program to `exercise2e.c`. Include a Makefile if necessary.

(f) Combine all the information you have into a working exploit using `netcat`. Remember that you will need to align your input correctly. (Hint: the shellcode is not a multiple of 8 bytes.) Run the exploit using the shell script or program from (e).

You will not immediately notice it if your shellcode works, because the shell will not recognize the network connection as an interactive terminal and therefore will not echo a prompt. You will have to give a command such as `ls` to confirm you were successful. Alternatively, edit the shellcode so that instead of starting a shell, it will start `ls` directly.

After confirming that your exploit was successful, run the command `proof`, and follow the instructions on-screen. Don't forget to send the e-mail. Please do not attempt to continue exploiting the server afterwards.

Save all the files you used to exploit the server, and an explanation of how they should be used, to the *directory* `exercise2f`.

3. Place the files `exercise1a`, and `exercise1b.c`, `exercise2`, `exercise2e.sh` (or `exercise2e.c`), and the directory `exercise2f` in a directory called

`sws1-assignment5-STUDENTNUMBER1-STUDENTNUMBER2` (again, replace `STUDENTNUMBER1` and `STUDENTNUMBER2` by your respective student numbers). Make a `tar.gz` archive of the whole `sws1-assignment5-STUDENTNUMBER1-STUDENTNUMBER2` directory and submit this archive in Blackboard.