Software and Web-Security
Assignment 5b, Monday, March 9, 2015

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

Deadline: Monday, March 23, 24:00 (midnight)

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 this 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 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\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
```

1. For part 1 of this assignment you will analyse the shellcode and determine what it does, by hand. You should not need to use any computer programs, but you will probably need online resources such as a system call reference table.

   (a) The shellcode translates to AT&T syntax assembly as follows:

   ```
   #  "\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
   ```

   What all of these instructions mean is explained on https://www.cs.uaf.edu/2007/fall/cs301/support/x86_64/index.html, but note there are some syntactic differences: in our translation, the source is the first argument, the destination is the second argument. Registers are prefixed with %, and values with $. Also note that the "smaller" registers simply use the lower bytes of their larger counterparts, i.e. in the code above `%al` is the lower 8 bits of `%rax`.

   The `syscall` at the end indicates that we are doing a system call. System calls are requests to an operating system's kernel to provide a service, such as opening a file, reading from an open file descriptor, exit a program, etc.

   Using the above explanation of what the assembly instructions do, and a reference table for system calls on 64-bit Linux, such as the one at http://blog.rchapman.org/post/36801038863/linux-system-call-table-for-x86-64, determine which system call we are trying to use here. Write your answer to a file called `exercise1`.

(b) Examine the manpage for that system call with `man 2 <system call>`. Look at its arguments. Which arguments are in which registers at the moment the `syscall` is run? (You can use the reference table from the previous exercise.) For each register, also give the lines of assembly code which put the argument there. Add your answer to `exercise1`.

(c) Considering this, what do you expect this shellcode to do when run? Add your answer to `exercise1`.

(d) Explain why `%rbx` is shifted right by 8 bits. (Hint: What does the value in `%rbx` represent? What is it missing before the shift? What should shellcode usually not contain, considering one of the most used ways of exploiting buffer overflows?) Add your answer to `exercise1`.

2. For part 2 of this exercise you will 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.

   Note: if you want to use a NOP-sled, on the x86_64-architecture you may need to use `0x0f 0x1f` instead of `0x90`.

   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 `exercise1`, `exercise2`, `exercise2e.{sh,c}`, and the directory `exercise2f` and all its contents 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.