Lab II

Software Security, DV2546

Edgar Lopez-Rojas*
Blekinge Institute of Technology
School of Computing

November 1, 2015

Contents

1 Introduction

This is the second lab in the Software Security course and your task this time is to examine and exploit one of the most common security defects in software systems - the buffer overflow.

1.1 Time Approximation

We estimate that a student on average will use 40 hours to solve this exercise.

2 Recommended Reading Before the Lab

We recommend that you read the following articles and documents as a part of the lab:

- Chapter 11 in the course book 'GRAY HAT HACKING'

*Originally created by: Per Mellstrand and Martin Boldt

- The Intel®64 and IA-32 Architectures Software Developer's Manual, Volume 2 (*Use this for reference only*) http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- `gdb` Quick Reference Card http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf
- Debugging with GDB http://sources.redhat.com/gdb/documentation/ (*Use this for reference only*)
- Aleph One, Smashing The Stack For Fun And Profit http://inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf

3 The Buffer Overflow

Your task in this laboration is to implement a full-blown buffer overflow attack against a small program.[1] In the additional file provided you will find the archive 'oflow.zip' which contains source code for a simple c program that you will use to exploit its vulnerabilities. You are free to chose in which operative system do you want to perform the exploit. You should have in mind that modern operative system have already built-in protection for the execution of your exploits so you should learn how to "disable" it.

3.1 Task 1 (Warming up)

Your task is to analyse and discuss the web sites provided below.

- Common vulnerabilities and exposures list (http://cve.mitre.org/)
- Common weakness enumeration (http://cwe.mitre.org/)

Answer the following questions in prose in no more than one page:

- What are those websites for?
- What are the difference between them?
- How do they relate with each other?

3.2 Task 2

Your first task is to set up the operative system for testing buffer overflow by disabling its security mechanism. Compile the c code with appropriate flags to disable stack protection and other security features of the compiler. Explain clearly all steps and the flags learned from this task with proper references.

3.3 Task 3

Your task is to construct an input string to the program that makes the program execute the function `notcalled`.

[1] Yes, a real buffer overflow.

3.4 Task 4

Your task is to construct an input string to the program that makes the program process print its current hostname[2] to `STDOUT`[3].

3.5 Task 5 (Optional)

Run the following command on the oflow file using root access: *chmod 4755 oflow*. Your task is to construct an input string to the program that makes the program replace its execution (call `execv`) with `/bin/sh`. To test this, run the command: *whoami* from the new shell. If you get as answer '*root*' your task is completed.

3.6 Hints and Tips

In this section we provide you with some tips and tricks that might be helpful when solving this exercise.

- You may use different input strings for each task and that only tasks 0, 1, 2 and 3 are mandatory.
- To find the address to a function in a non-stripped binary you can use the command `nm <filename> | grep <functionname>`
- The original lab was made for FREE BSD, but you might find plenty information of how to test buffer overflow in many Linux distributions.
- It is somewhat tricky to enter binary data in a command line, so you might be helped by writing a small tool that does this for you (e.x. c, perl, ruby).
- For task 2, 3 and 4 you might want to look closer at the appendix 'Finding the Right Address' or Chapter 11 in the course book 'GRAY HAT HACKING'
- If at first you don't succeed, then try and try again :-), and remember gdb is your best friend!

4 Examination

You are encourage to work in groups of two (2) students or alone after approval from the lab staff. *Please note that the time estimations done for this exercise are based on the assumption that students work pair-wise in groups.* The group will be examined as a whole (i.e. both students must be able to explain the methods and techniques used by the group to pass examination). *You only need to write one report for the whole group, not one per student.*

[2] You can see the hostname by using the command `hostname` or by using the system call `gethostname`.

[3] File descriptor 1.

4.1 Report

This exercise is examined with both a written and an oral exam. You should write a short report (typically about one or two pages per task) where you (for each task) describe;

1. Briefly explain the *methods, techniques and tools* you used to solve (or attempted to solve) the task (max 1 page). Please include references to any tool you have used that you have found yourself (i.e not provided by us in any image) and please also include source code to any tools you have developed.

2. A detailed description of *BUGS* found, *VULNERABILITIES* discovered and *EXPLOITS*.

3. Source code of exploit and proper reference if taken from another source. Screen shots or output of the exploit to show completion of the task.

4. A discussion of *COUNTERMEASURES* for these vulnerabilities, one or several suggestions of *how to fix* the vulnerable system if it was in use. Notice that in most of the cases there is an obvious solution, but a good report should contain more elaborated solutions that address different aspects of how to protect against an specific or similar vulnerabilities. We expect a discussion of methods tools and techniques that you feel could have prevented, detected or in any other way helped in the development of this software. This is important factor for the quality of the report and will affect directly the grade.

5. A general *REFLEXION* of what you learnt during this lab (between 100-200 words) explaining How long time (approximation in whole hours) you spent on each task and How important, hard or easy you found each of the task.

This report should be sent to the examiner and if the written report is of acceptable quality you can continue with the oral examination. Several time slots for the oral exam will be provided by the examiner. The report should be in clear and understandable English.

4.2 Plagiarism

With exception of your lab partner you are not allowed to cooperate with other students or groups in any way that compromise the specific solution of each part of the lab. We check for plagiarism with a platform that compare your report with other web resources and other students reports from this term and past terms. We will not accept any report that contains plagiarism. In certain cases we will be force to inform the students that incurred in plagiarism.

Saying this, we encourage all the students to properly reference any source of information used to write the report and also to use their own words in the reflections and explanations of the steps to obtain the results.

4.3 Oral Examination

During the oral exam the group of students should explain and defend their report. After the oral exam you will be notified by the examiner if you passed or failed the examination. If you fail you must use another examination time and send in the report for that as if you were submitting the report for the first time.

During the oral exam you might also be asked to explain the source code of your solution, i.e. go through the source code and explain for the examiner what each line does. It is therefore very important that you *store all* source code, byte-code or binary code that you have produced or used when solving the task on the secured server. Note: you *must* be able to describe all code you have used to solve a lab or part of a lab.

4.3.1 Important issues regarding oral examination

The oral examination is an important part of the Software Security course. Please read this section carefully and be sure to follow the rules to avoid any problems during the examination.

- We are running oral examinations on a very tight schedule and cannot accept students arriving late for their examination. Please try to arrive at least ten minutes before your examination time.
- If you are working in a group both student must be present at the examination. Please remember that the group is examined as a whole and that the examiner freely can choose which student to direct a question to.
- You might be asked to demonstrate a part of, or the complete, solution to the exercise. *You must be prepared for this and have tools (scripts) developed to automate any step required.* Such tools must be available on your *seclab* account and should typically execute in less than 1 (one) minute.

4.4 Grading

For this lab you can get grade A-F, where:

1. **Grade A** requires that you finish all task including the optional tasks, AND have a really good report AND that you can describe it well during the oral examination.
2. **Grade B** requires that you finish all mandatory tasks AND make a strong and good attempt of the optional tasks AND can describe it during the oral examination.
3. **Grade C** requires no optional tasks AND that you have a good report AND can describe it well during the oral examination.
4. **Grade D** requires (no optional tasks) that you have either a good report OR can describe it well during the oral examination.
5. **Grade E** requires (no optional tasks) that you have a sufficient report AND can describe it during the oral examination.

6. **Grade F** when you do not fulfil the tasks required at least for grade E.

In all the cases you must fulfil tasks 0-3.

... And Finally

Good Luck!

Appendix A: Finding The Right Address

Appendix A: Finding the Right Address

Finding the address to the string buffer `buf` can be a bit tricky. There are many ways to do this, but most involve a bit of guessing in the end. In this appendix we provide you with some hints on how this can be done.

Step 1: Finding the Approximate Address

The first step is to find the address of the stack buffer just before the call to the vulnerable function (`gets`) is made. There are several ways of doing this, and we will describe only two; one requiring a bit of programming and the other requiring a bit of debugging skills. These methods may be less suitable for larger programs, but as we are working with a rather simple program in this exercise they will do fine.

Using ELF Injection

This method requires a bit of programming skills, but is very easy when working with small (dummy) programs. The key concept is that we create a function with the same name as the one that we are interested in (`gets`) which instead of providing the normal functionality just dump the values of the parameters that we are interested in. We then inject this file into the host binary such that it will replace the normal library function. For more information on the injection part, please refer to the man page for the dynamic ELF linker, rtld. Say, for example, that we wanted to override the function `puts()` in the C library we would create a function with the same prototype as `puts()` that prints the value :

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

int puts(const char * str) {
  printf("The address is: 0x%x\n" , str);
  exit(1);
}
```

This function prints the address to `str` and terminates the process. The next step is to compile this source code to a dynamic linkable object:

```
gcc -shared -o ourlib.so <sourcefile>
```

The third and final step is to execute the host process with the new module injected. In `sh` or `bash` write:

```
( export LD_PRELOAD=./ourlib.so ; ./host_process )
```

The first call the host program make to `puts` will be directed to the function we wrote which will print the address to the buffer and terminate execution of the program. Once you have this address you can move on to Step 2.

Using a Debugger

This is another method that might be easier to use on larger programs and also require no coding. Instead of injecting code with a dynamic module we examine the process using a normal debugger. In this section we provide examples using the `gdb` debugger. To start the process you want to investigate use a command line similar to this:

```
gdb <program name>
```

One in the debugger you will get a prompt, typically `(gdb)`. From this prompt you can enter commands to the debugger, enter `quit` to quit the debugger. Typically you want to investigate the function that makes the call as this is where the address of the variable is stored. If you had had access to the source code for the program you could have placed a breakpoint relative a source code line, but now we have to examine the machine code and place a breakpoint based on this instead.

```
(gdb) break main
Breakpoint 1 at 0x804849c
(gdb) run
Starting program: /usr/home/per/illuminati/a.out
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x804849c in main ()
(gdb)
```

Now we can examine the function we want (to make things easier we will examine the function `main` here). You examine data with the `x` command in `gdb`. To examine the first 20 assembly instructions in the `main` function type:

```
(gdb) x/20i main
0x804849c <main>:       push   %ebp
0x804849d <main+1>:     mov    %esp,%ebp
0x804849f <main+3>:     sub    $0x78,%esp
0x80484a2 <main+6>:     add    $0xfffffff4,%esp
0x80484a5 <main+9>:     lea    0xffffff9c(%ebp),%eax
0x80484a8 <main+12>:    push   %eax
0x80484a9 <main+13>:    call   0x804834c <puts>
0x80484ae <main+18>:    add    $0x10,%esp
0x80484b1 <main+21>:    leave
0x80484b2 <main+22>:    ret
0x80484b3 <main+23>:    nop
0x80484b4 <__do_global_ctors_aux>:      push   %ebp
0x80484b5 <__do_global_ctors_aux+1>:    mov    %esp,%ebp
0x80484b7 <__do_global_ctors_aux+3>:    sub    $0x14,%esp
0x80484ba <__do_global_ctors_aux+6>:    push   %ebx
0x80484bb <__do_global_ctors_aux+7>:    mov    $0x8049594,%ebx
0x80484c0 <__do_global_ctors_aux+12>:   cmpl   $0xffffffff,0x8049594
0x80484c7 <__do_global_ctors_aux+19>:   je     0x80484d8 <__do_global_ctors_aux+36>
0x80484c9 <__do_global_ctors_aux+21>:   lea    0x0(%esi),%esi
0x80484cc <__do_global_ctors_aux+24>:   mov    (%ebx),%eax
(gdb)
```

The first 11 lines output from `gdb` is from the function main (as you can see in the second column as they all say `<main+X>` and hence we can safely ignore all instructions past this point. Now, when we examine this output, we can easily see that the call to `puts` is done from the line that begins with `0x80484a9`. As we are interested in the parameter sent to this function and the function uses normal good old-fashioned C calling convention, we place a breakpoint at this instruction and examine the stack:

```
(gdb) break *0x80484a9
Breakpoint 1 at 0x80484a9
(gdb) cont
Continuing.
Breakpoint 3, 0x80484a9 in main ()
```

When the program has executed to the instruction where the breakpoint is placed the debugger will suspend the execution and wait for commands from the user (that means you). As the interception is done just before the call instruction is executed the value stored on top of (or bottom of, depending on your view) the stack we just examine the stack to see what's there. We begin by having a look at the processor registers to see which value the stack pointer (called esp on X86) has.

```
(gdb) info registers
eax            0xbfbffa6c       -1077937556
ecx            0xbfbffc53       -1077937069
edx            0x80484e4 134513892
ebx            0x1      1
esp            0xbfbffa48       0xbfbffa48
ebp            0xbfbffad0       0xbfbffad0
esi            0xbfbffb2c       -1077937364
edi            0xbfbffb34       -1077937356
eip            0x80484a9 0x80484a9
eflags         0x283    643
cs             0x1f     31
ss             0x2f     47
ds             0x2f     47
es             0x2f     47
fs             0x2f     47
gs             0x2f     47
(gdb)
```

As you can see from this output the current value of the stack pointer is `0xbfbffa48` (The line beginning with esp). So the final step is to investigate which value is stored in the memory where esp points (as this is the top of the stack).

```
(gdb) x/x 0xbfbffa48
0xbfbffa48:     0x23230244
```

And now we have the value for the first parameter (`0x23230244`).

Step 2: Finding the Exact Address

When we have the approximate address it's time to see if we need to adjust this a few bytes to compensate for other data placed on the stack. As usual, there are several ways this can be done, but we will explain only one here. As we are exploring a program that has a buffer overflow we begin by providing a buffer that is too long (which will make the program crash) and then we examine the memory around the address that we found in the previous example. To examine the memory we use a debugger and examine the core file that will be created when the program crashed.

```
# ./buggyprogram
Please enter your name: warning: this program uses gets(), which is unsafe.
AAAAAAA<more characters>AAA
Segmentation fault (core dumped)
#
```

Now we have a core file (typically named `<name of program>.core`) which we can examine in everyones favourite debugger, `gdb`!.

```
# gdb buggyprogram buggyprogram.core
(Some copyrigt text removed for readability)
#0  0x41414141 in ?? ()
(gdb)
```

As you might already have guessed from the address (`0x41414141`) all the A characters that we used for input were interpreted as the return address (A is 65 in ASCII and 65 is 41 in hexadecimal, hence 0x41414141). This is of lesser interest, however, so let's move on to something a bit more interesting; examining the memory as it were just before the process terminated. To do this we use the `x` command again and examine the memory at the address around the buffer (the approximate address). In this printout all real addresses have been modified to not provide too much help.

```
(gdb) x/40xb 0x23230264
0x23230264:     0xc9    0xe6    0x04    0x28    0x92    0xc1    0x04    0x28
0x2323026c:     0x68    0xcc    0x05    0x28    0x00    0x00    0x06    0x28
0x23230274:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x2323027c:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x23230284:     0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
```

Now we see a clear pattern; at address `0x23230274` begins a large number of `0x41` chars which we know is the ASCII code for 'A'. Based on this we can be rather sure that this is the address of the buffer. You might want to try the process again, but with a different character as input to see if the memory is modified to reflect the changes in input. For a good list of ASCII values in decimal hexadecimal and octal see `man ascii`.

Using the INT3 Hack

Now that you know at which address the buffer is and how to modify execution by creating a buffer overflow, you might want to try to execute your buffer (this is needed to execute shellcode which you must do to fulfill task 1 and 2). A simple trick which we recommend is to use a debugging feature of the processor.

As you know assembly mnemonics are coded into machine code which the CPU executes. If there is a problem with your program (such as a memory violation) the CPU will notify the operating system which will halt the execution of the program and print a diagnostic message. This was what happened before when we entered a too long string in the buffer:

```
# ./buggyprogram
Please enter your name: warning: this program uses gets(), which is unsafe.
AAAAAAA<more characters>AAA
Segmentation fault (core dumped)
#
```

In this case we got a 'Segmentation fault' which is a common reason for a program to crash. If you want to verify that you really are executing data that you provided in the buffer you can use a special instruction, `int 3`, that will make the program crash in a different way, which will typically look like this:

```
Please enter your name: warning: this program uses gets(), which is unsafe.
Hello "ÌÌÌÌÌÌÌ<more characters>ÌÌÌxxxx"
Trace/BPT trap (core dumped)
```

To use this trick you must make the CPU execute the `int 3` instruction which is only a single byte long and encoded as `0xCC`.