ECE568 Lab #1: Buffer Overflow, Format String and Double Free Vulnerabilities

Introduction

These are the instructions for Lab 1. This lab consists of 6 programs and each program has a different vulnerability. The first four are mandatory (5 marks each) while you will receive bonus points for the last two parts (also 5 marks each).

- **Stack Overflow (targets 1-4)**: You will write an exploit for the first four programs with a buffer overflow vulnerability. The programs are ordered by difficulty and exploits will become increasingly complex. Nevertheless, once you get the first one working, it should be easier to write exploits for the next three, as these exploits will be very similar to the first one.

- **Bonus (targets 5 & 6):** You will write exploits for a program with a format string vulnerability, and a program with a double-free vulnerability that you will have to use the knowledge gained from doing the first four to figure out.

Lab 1 may be done <u>individually</u> or in groups of <u>two</u>. Final submissions are due at 11:59pm on Friday, Jan 31, 2020.

The Environment

Exploits are very environment dependent. Differences in compilation environment, and kernel version may cause an exploit to work on one platform and not on another. To ensure a consistent environment for testing and grading, your functions make use of a support function that properly configures your environment for you. Each of your targets use a called of `lab_main()`:

```
    int lab_main(int argv, char * argv[])
```

You can treat this as the same as "main" when you are thinking of how the program works – but you will need to remember this, and set your gdb breakpoints in "lab_main" if you want to stop in that part of the target program.

Please note: **you must use `/usr/bin/gcc` as you C compiler**.

The Targets

The `targets/` directory in the assignment tarball contains the source code for the targets, along with a `Makefile` specifying how they are to be built. Your exploits should assume that the compiled target programs are in "`../targets/`". (In other words, please don't rename the directories: leave them as you find them in the lab assignment tar.) Note that the targets for Lab #1 are all included in the file that you will initially download. (The tarball is available on the blackboard web page.)

The Exploits

The `sploits/` directory in the assignment tarball contains skeleton source for the exploits, which you are to complete, along with a `Makefile` for building them and a `stackSetup.c` which will configure your environment for you. (This "setup" code bypasses a few of the things that gcc does to protect against *stack overflow* attacks, and will make life easier when working on these lab.) Also included is `shellcode.h`, which gives Aleph One's shellcode (explained below).

Targets 1 through 4:

These targets have a buffer overflow vulnerability. To exploit such a vulnerability, one takes advantage of the absence of bound checks when strings are copied in the buffer that is vulnerable. By writing a string that is larger than the buffer, one can overwrite adjacent data on the stack. Since the stack grows down on x86 architectures, the return address of a function is stored at a higher address than this function's local variable, and it can therefore be overwritten by a value of the attacker's choosing. This value should be the address of the shellcode the attacker has placed in the attack string. Read Aleph One's seminal paper, "*Smashing the Stack for Fun and Profit*" to learn more about buffer overflow attacks; this paper is included in the Lab #1 tarball, in the `docs/` directory. **For the latter targets, you may need to make use of additional "argv" parameters, or try using the "env" parameter.**

Target 5:

This target has a format string vulnerability. Read scut's "*Exploiting Format String Vulnerabilities*" to learn more about this type of vulnerability and how to exploit them. This paper is included in the tarball, in the `docs/` directory.

Target 6:

This target has a double-free vulnerability. The target allocates two buffers via a call to a simplified implementation of `malloc`, called `tmalloc` ("trivial malloc"). Each buffer is of size 128 bytes. It then frees these two buffers via a call to `tfree`. A third buffer, this time of size 256 bytes, is allocated. The string provided as the first argument to the target on the command line is then copied in this buffer. Finally, the target frees the second buffer again, instead of freeing the third one. An attacker may take advantage of this flaw to once again take control of the target, by carefully crafting the input string and taking advantage of how the dynamic memory allocation library is implemented.
Look at the implementation of `tmalloc` and `tfree`. Study especially carefully how adjacent free buffers are consolidated and take advantage of the fact that `q` is freed twice to overwrite `foo`'s return address.

The Lab

You are to write exploits, one per target. Each exploit, will call the associated target with the attack string to yield a command shell (`/bin/sh`). For marking purposes, make sure the exploits you hand in call the target programs and pass the attack string directly through the command line, via `execve`. (Specifically, the assignments will be tested with scripts, so do not assume that we will call the targets with an environment variable, like the examples in Aleph One's document.)
When you first get the exploit templates, each of the exploit templates will initially just produce an error message:

```
$ ./sploit1
execve failed.
$
```

When your exploits are working properly, they should each call their matching "target" executable, and exploit the corresponding security vulnerability (outlined above) to compromise the target program and open a command shell:

```
$ ./sploit1
sh-2.05b# exit
```

Hints: GDB

To understand what’s going on, it is helpful to run code through `gdb`. In particular, you can use the ”disassemble” and ”stepi” commands. Using the ”x” command will allow you to view arbitrary sections of memory. Help and documentation on gdb is available within gdb by typing the `help` command.

Let's look at the first target. First, you need to compile the targets, and then start the target with “gdb”:

```
cd targets
make
gdb target1
```

We place a breakpoint on the main function, and then run target1, with the command line argument “test” (your output will be slightly different, depending on the environment you run it in):

```
(gdb) break lab_main
Breakpoint 1 at 0x400b21: file target1.c, line 15.
(gdb) run test
Starting program: /root/a1_grad/targets/target1 test
Breakpoint 1, lab_main (argc=2, argv=0x7fffffffe1b8) at target1.c:15
15     int    t = 2;
```

Let's look at the stack with the “info” command (the output from your program will be slightly different):

```
(gdb) info frame
Stack level 0, frame at 0x202dfe90:

 rip = 0x400b21 in lab_main (target1.c:15); saved rip 0x400916
 called by frame at 0x202dfec0
 source language c.
 Arglist at 0x202dfe80, args: argc=2, argv=0x7fffffffe1b8

 Locals at 0x202dfe80, Previous frame's sp is 0x202dfe90
 Saved registers: rbp at 0x202dfe80, rip at 0x202dfe88
```

This tells us that rip (program counter) is saved at 0x202dfe88.

```
(gdb) x 0x202dfe88
0x202dfe88:     0x00400916
```

This means that the return address, once main completes, is 0x00400916. This is the address of an instruction in `__libc_start_main`. You can verify this by typing “x 0x00400916” (substituting the address that you received in your output). Now, let us see where “buf” is located on the stack:

```
(gdb) p &buf
$1 = (char (*)[96]) 0x202dfe10
```

So “buf” is located at addresses 0x202dfe10 to 0x202dfe70 (96=0x60). Now, if we overflow buf, we can see that the return address will be overwritten. The trick (and your task in this lab) is now to overflow buf with an appropriate string.

Hints: Instrumenting the Targets

In addition, note that you have a huge advantage that most hackers do not – you have the source code of the vulnerable programs and you know *exactly* what the environment that you are attacking will be. You may instrument the targets to gather information, so long as the attack also works on the original *unmodified* targets that we'll be marking you against.
While you can gather all information pertaining to memory layout by using gdb, the exact address of the buffer you are overflowing will differ when you are using gdb. This is because gdb uses some memory for itself and as a result shifts the stack of the traced process.

Hints: Stack Layout

Use gdb to understand the stack layout and the way functions are called on the x86 architecture. Place a breakpoint before and after the call to a function and compare the stacks. For architectural reasons, the compiler aligns info in memory. Variables are word-aligned (this means their address can always be divided by 4).
To understand how the stack is organized in the x86_64 ISA specifically:

http://wiki.osdev.org/System_V_ABI#x86-64

Hints: Further Reading

There are many x86 references available on the web; several good starting points are:

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
http://www.sandpile.org/

Read Aleph One's "*Smashing the Stack for Fun and Profit*". Carefully. For the bonus parts, read scut's "*Exploiting Format String Vulnerabilities*". (Both are included in the respective lab tarball and on the course website.)

Warnings

Aleph One gives code that calculates addresses on the target's stack based on addresses on the exploit's stack. The stack address depends highly on how the exploit is called (different shells have different starting stack locations for example). In addition, you will find that the alignment of local variables on the stack is highly dependent on the version of the compiler and compile options used. Do not assume that the numbers Aleph comes up with will work on your system (in fact they will definitely not work). You must therefore hard-code target stack locations in your exploits. You should **not** use a function such as `get_sp()` (as used in his paper) in the exploits you hand in.

Submission Instructions

You will need to submit the source code for your exploits and the explanation of what you did. Note that both the submit command and checking command should be run from the ECF host machines, and your code must properly run there. (This is particularly important if you have been testing this on your personal computer.)
For each target, please explain in at most 100 words, what the vulnerability was and how you exploited it. These explanations must be in the file `explanations_lab1.txt`. These files must also contain the names and student numbers of your group members prefixed by #. Do not prefix other lines by # as this would confuse the automated marking scripts.

```
#first1 last1, studentnum1, e-mail address1
#first2 last2, studentnum2, e-mail address2
```

It is very important that this information is correct as your mark will be assigned by student number, and the email you give here will be how we will get the results of the lab back to you. **This file should be placed in a single directory along with the code for your exploits, the Makefile needed to compile your exploit, and any other files needed to build your exploits.**

Go to the directory where your **exploits** are stored and type the appropriate command. Do not submit until the script indicates that the submission appears to be properly formatted. In addition, check that the script correctly identifies your group members. (Please note that some student numbers were being incorrectly reported as invalid by the ECF "submit" scripts last term: I will attempt to get this resolved…)

Submission is done using the ECF submit command:

```
      submitece568s 1 Makefile shellcode-64.h sploit1.c ... explanations_lab1.txt
```

Please note the number after the submit command, which indicates what lab you are submitting for. A man page on ECF (man submit) contains more information on the ECF submit command. The submission command for a lab will cease to work after the lab is due.