Buffer Overflow
Deadine: Midnight 29/3/2017

1 Lab Tasks

1.1 Initial setup

You need to setup an Ubuntu x86_64 virtual machine. It is common practice for security researchers to study malicious code and infected systems under a virtualized environment in order to isolate the effects and keep the host system safe. You can use any virtualization software you like, depending on the host system. Most common choices include VMWare and VirtualBox, both available for Windows, Linux, Mac OS X. Another Linux only choice is QEMU/KVM. The latest official Ubuntu installation image is available for download at: http://releases.ubuntu.com/16.04/ubuntu-16.04.2-desktop-amd64.iso. Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simplify our attacks, we need to disable them first.

Address Space Randomization. Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable these features using the following commands:

```
$ su root
  Password: (enter root password)
#sysctl -w kernel.randomize_va_space=0
```

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called ”Stack Guard” to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection if you compile the program using the *-fno-stack-protector* switch. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

```
$ gcc -fno-stack-protector example.c
```

Non-Executable Stack. Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, the stack is set to be non-executable. To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -z execstack  -o test test.c

For non-executable stack:
$ gcc -z noexecstack  -o test test.c
```

1.2 Shellcode

Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

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

int main(int argc, char *argv[]) {
   char *name[2];

   name[0] = ``/bin/sh'';
   name[1] = NULL;
   execve(name[0], name, NULL);

   exit(0);
   return(0);
}
```

The shellcode that we use is just the assembly version of the above program. You can "spy" the assembly created for the above program by gcc. However this will not take you a long way, since gcc invokes a call to the execve function implemented by glibc (which in turn calls the execve system call), instead of invoking the execve system call directly. This is the case for exit as well.

As a hint, in order to generate the equivalent assembly code program, look at what we did in the class for the more innocent payload which just prints out messages. The execve and exit glibc implementations use exactly the same arguments as the respective system calls. The system call numbers for execve and exit on x86_64 linux systems are 59 and 60 respectively. You can find an always up-to-date quick reference of the system call numbers at `https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl`. Another useful piece of information is the x86_64 ABI, namely how functions and system calls are called in x86_64 systems, which registers are used for parameter passing etc. You can find the respective document at `https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI` (x86-64 psABI), in Appendix A. Alternatively, although not suggested, you can just mimic the examples we discussed in class.

I am sure you will be able to find an assembly shellcode with a quick search in the net. PLEASE try to resist the temptation!

The following program shows you how to launch a shell by executing a shellcode stored in a buffer (code[]) for testing purposes. Of course the shellcode is incomplete (you will have to fill it yourselves). Please compile and run the following code, and see whether a shell is invoked.

```
/* call_shellcode.c  */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x48\x31\xc0"            /* 1:  xor    %rax, %rax                  */
  "\x48\x31\xd2"            /* 2:  xor    %rdx, %rdx                  */
  ...
  ...
  ...
 ;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Please use the following command to compile the code (don’t forget the `execstack` option):

```
  $ gcc -fno-stack-protector -z execstack -o call_shellcode call_shellcode.c
```

1.3 The Vulnerable Program

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[48];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

Compile the above vulnerable program and make it set-root-uid. You can achieve this by compiling it in the `root` account, and `chmod` the executable to `4755` (don't forget to include the `execstack` and `-fno-stack-protector` options to turn off the non-executable stack and StackGuard protections):

```
$ su root
  Password (enter root password)
# gcc -o stack -z execstack -fno-stack-protector stack.c
# chmod 4755 stack
# exit
```

The above program has a buffer overflow vulnerability. It first reads an input from a file called "badfile", and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of 517 bytes, but the buffer in `bof()` has only 12 bytes long. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called "badfile". This file is under users' control. Now, our objective is to create the contents for "badfile", such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

1.4 Task 1: Exploiting the Vulnerability

Try to create a "badfile" that has the opcodes of the shellcode payload you created earlier, plus anything else needed in order to exploit the vulnerability. You can create a C program that "prints" the opcodes to a file, or use a script (perl, python or whatever you feel comfortable with).

Then run the vulnerable program `stack`. If your exploit is implemented correctly, you should be able to get a root shell

```
$./stack          // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

It should be noted that although you have obtained the "#" prompt, your real user id is still yourself (the effective user id is now root). You can check this by typing the following:

```
# id
uid=(500) euid=0(root)
```

Many commands will behave differently if they are executed as `Set-UID root` processes, instead of just as `root` processes, because they recognize that the real user id is not `root`. To solve this problem, you can run the following program to turn the real user id to `root`. This way, you will have a real `root` process, which is more powerful.

```
void main()
{
  setuid(0);  system("/bin/sh");
}
```

1.5 Task 2: Address Randomization

Now, we turn on the address randomization. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult? You should describe your observation and explanation in your report. You can use the following instructions to turn on the address randomization:

```
$ su root
  Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=2
```

If running the vulnerable code once does not get you the root shell, how about running it for many times? You can run `./stack` in the following loop , and see what will happen. If your exploit program is designed properly, you should be able to get the root shell after a while. You can modify your exploit program to increase the probability of success (i.e., reduce the time that you have to wait).

```
$ sh -c "while [ 1 ]; do ./stack; done;"
```

1.6 Task 3: Stack Guard

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we disabled the “Stack Guard” protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of Stack Guard. To do that, you should compile the program without the *-fno-stack-protector’* option. For this task, you will recompile the vulnerable program, stack.c, to use GCC’s Stack Guard, execute task 1 again, and report your observations. You may report any error messages you observe.

1.7 Task 4: Non-executable Stack

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the `noexecstack` option, and repeat the attack in Task 1. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult. You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the non-executable stack protection.

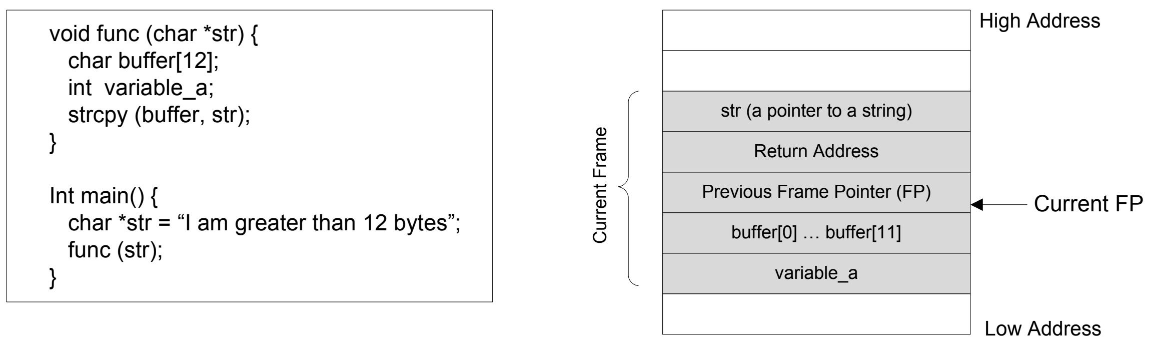
```
# gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

It should be noted that non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. The *return-to-libc* attack is an example.

Whether the non-executable stack protection works or not depends on the CPU and the setting of your virtual machine, because this protection depends on the hardware feature that is provided by CPU. If you find that the non-executable stack protection does not work, check our document (“Notes on Non-Executable Stack”) that is linked to the lab’s web page, and see whether the instruction in the document can help solve your problem. If not, then you may need to figure out the problem yourself.

2 Guidelines

We can load the shellcode into “badfile”, but it will not be executed because our instruction pointer will not be pointing to it. One thing we can do is to change the return address to point to the shellcode. But we have two problems: (1) we do not know where the return address is stored, and (2) we do not know where the shellcode is stored. To answer these questions, we need to understand the stack layout the execution enters a function. The following figure gives an example.



```
void func (char *str) {
  char buffer[12];
  int  variable_a;
  strcpy (buffer, str);
}

Int main() {
  char *str = “I am greater than 12 bytes”;
  func (str);
}
```

(a) A code example

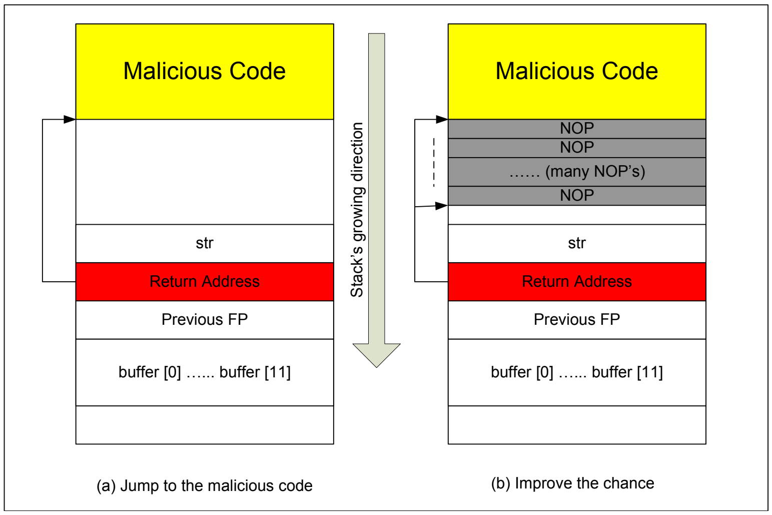
(b) Active Stack Frame in func()

Finding the address of the memory that stores the return address. From the figure, we know, if we can find out the address of `buffer[]` array, we can calculate where the return address is stored. Since the vulnerable program is a `Set-UID` program, you can make a copy of this program, and run it with your own privilege; this way you can debug the program (note that you cannot debug a `Set-UID` program). In the debugger, you can figure out the address of `buffer[]`, and thus calculate the starting point of the malicious code. You can even modify the copied program, and ask the program to directly print out the address of `buffer[]`. The address of `buffer[]` may be slightly different when you run the `Set-UID` copy, instead of of your copy, but you should be quite close. Also the address maybe different when executing within gdb and when executing natively.

If the target program is running remotely, you may not be able to rely on the debugger to find out the address. However, you can always *guess*. The following facts make guessing a quite feasible approach:

- Stack usually starts at the same address.
- Stack is usually not very deep: most programs do not push more than a few hundred or a few thousand bytes into the stack at any one time.
- Therefore the range of addresses that we need to guess is actually quite small.

Finding the starting point of the malicious code. If you can accurately calculate the address of `buffer[]`, you should be able to accurately calcuate the starting point of the malicious code. Even if you cannot accurately calculate the address (for example, for remote programs), you can still guess. To improve the chance of success, we can add a number of NOPs (NOP shled) to the beginning of the malcious code; therefore, if we can jump to any of these NOPs, we can eventually get to the malicious code. The following figure depicts the attack.



(a) Jump to the malicious code

(b) Improve the chance

Storing an long integer in a buffer: In your exploit program, you might need to store an `long` integer (8 bytes) into an buffer starting at buffer[i]. Since each buffer space is one byte long, the integer will actually occupy eight bytes starting at buffer[i] (i.e., buffer[i] to buffer[i+4]). Because buffer and long are of different types, you cannot directly assign the integer to buffer; instead you can cast the buffer+i into an `long` pointer, and then assign the integer. The following code shows how to assign an `long` integer to a buffer starting at buffer[i]:

```
char buffer[20];
long addr = 0xFFEEDD88DEADBEEF;

long *ptr = (long *) (buffer + i);
*ptr = addr;
```

Alternatively, and probably preferably, assign the address as we discussed in class. Remember to take into account the endianess issues.

References

[1] Aleph One. Smashing The Stack For Fun And Profit. *Phrack 49*, Volume 7, Issue 49. Available at http://www.cs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html