1 Motivation

Buffer overflows and format string vulnerabilities are widespread and as security professionals we should have a good understanding of what they are and how they can be exploited. Also, there is no better way to learn than to experience it first hand. In this lab you will learn how and what it takes to implement a local attack for stack based buffer overflow and format string vulnerabilities. You will first analyze problematic code, design an attack and finally follow through with writing an attack program and finally launch a successful attack.

2 References

"Writing buffer overflow exploits - a tutorial for beginners” Mixter.
http://mixter.void.ru/exploit.html

"Smashing the Stack for Fun and Profit” Aleph One.
http://www.shmoo.com/phrack/Phrack49/p49-14

"Exploiting Format String Vulnerabilities” Teso Security Group
http://teso.scene.at/articles/formatstring/

"Small Buffer Format String Attack” SecuriTeam
http://www.securiteam.com/securityreviews/5TP032KAAQ.html

3 Your Task

You will use buffer overflow and format string attacks to exploit the following code:

```c
\rootecho.c
1. #include<stdio.h>
2. int main(int argc, char *argv[]) {
3.   char buffer[600];
4.   int i;
5.   setuid(0);
6.   for (i = 1; i < argc; ++i) {
7.     strcpy(buffer, argv[i]);
8.     printf(buffer);
9.   }
10.  return 0;
11. }
```

This program is known as "rootecho." It works the same way as regular echo except it runs as root. Before we go into how to exploit the code please install the rootecho binary first. Follow these instructions:
1. Compile and build the code above to `rootecho`. (gcc rootecho.c -o rootecho)
2. su to root, if you are not already root. (su -)
3. Copy the rootecho binary to `/usr/bin/`. (cp rootecho /usr/bin/)
4. Change `rootecho` to be world executable. (chmod a+x /usr/bin/rootecho)
5. Change `rootecho` to have setuid. (chmod +s /usr/bin/rootecho)
6. If everything worked out then, `ls -la /usr/bin/rootecho` should output something like:

   `-rwsr-xr-x root root rootecho`

For the rest of the lab, you will be working under a non-root account, such as guest. Therefore make sure you have one of those available.

### 3.1 Analysis

The following questions are designed to help you get a grasp of the situation, please answer them:

1. Does the code have any internal buffers? If so, name them and their sizes.
2. Does the code do any input validation or check the input length before copying it into the buffer?
3. Does the code have any buffer overflow and/or format string vulnerabilities?
4. As stated earlier, this code runs as root. What can one possibly do with rootecho if he/she were able to find a bug and exploit it?

#### 3.1.1 Buffer Overflow Questions

Given the answers to the questions in the previous part and the references, answer these follow up questions:

1. What is an egg and how is it organized?
2. Given the source code what is the minimum size of the egg used in an attack? Given this information decide on a length for the egg.

#### 3.1.2 Format String Questions

Please answer these questions also:

1. If we were to use a format string to attack rootecho, what is the format of the attack string? What is the purpose of each part of the string?
2. Is the attack string going to be on the stack? If yes, what difference would it make if the string was on the heap instead? If no, where is it located?

### 3.2 Design

The questions in this section will help you better understand how the attacks work. Before we continue on with more questions, we will have a short gdb lesson.
3.2.1 Using gdb to Retrieve an Address

In both buffer overflows and format string attacks we need to get the address of something. In buffer overflows we need the address of somewhere in the nop area. Since the nop area will lie within the internal buffer if we know the address of the internal buffer we know the address of the nop area. In the case of format string attacks, we need to know what is the target address we want to overwrite, address of saved IP. This mini-lesson will help you understand how gdb can be used to get the address of a variable. The following steps should give you a good idea of where the target address is and can even be used in conjunction with other tools, such as a stack dump from a format string attack (You will do this later).

Note: I found these steps to work, if you know of better or easier steps use them instead.

As mentioned in class the address of variables are relative. For example if the environment changes then the addresses of all the variables will change accordingly. This means that if we want to get accurate addresses, we need to keep the environment as close to, if not exactly the same as, the environment that the attack will be running in. If the environment is not the same then we want to know if there is a constant difference between the debugging environment and the attacking environment. (All this talk will make more sense shortly.)

In general if we want to find the address of a variable in memory we can follow these steps. We will use the following program (test.c) as an exercise. Lets get the address of message:

```c
#include<stdio.h>

int main(int argc, char *argv[])
{
    char message[100] = "Hello ";
    printf("%08x\n",&message);
    return(0);
}
```

1. Compile the program so we can debug it, (gcc -g test.c -o test.gdb)
2. Run the debugger on the compiled file, (gdb test.gdb)
3. At this point you should see something like:
   ```
   (gdb)
   ```
4. Find a break point so we get a chance to look at the address. To print out the source code use list. The output of list will show something like:
   ```
   (gdb) list
   1 #include<stdio.h>
   2
   3 int main(int argc, char *argv[])
   4 {
   5     char message[100] = "Hello ";
   6     printf("%08x\n",&message);
   7     return(0);
   8 }
   ```
If one list is not enough to print out enough source code, then either press <enter> (which means repeat last command) or enter the list command again.

5. From the output, we see that if we choose a break point at 6 then it main would have started. So, to set the breakpoint at line 6 we enter (break 6)

6. Once the breakpoint is set, we use the run command to start the program (run)

7. The program will run and stop at the breakpoint we specified. Now we can get the address of the message by using the print command (you can use any C operator with print). (print &message). The output will look something like:

   $1 = (char (*)[100]) 0xbffffa70

Now we have the address of message (enter 'q' to get out of gdb).

At this point we should know how to get the address of a variable, we now need to get the address offset due to the different environments of running the program with run in gdb and running the program using execl. To do this we need to have 2 dummy programs, the first one prints out the address of a variable, such as the test.c above. The second represents the attack code (attack.c) that we will use (the source is given below.) The steps that follow will tell us what, if any, the offset in addresses from gdb is.

\attack.c
#include <stdio.h>

int main()
{
    char temp[4096] ="01234567890123456789";

    execl("./killme2.bin","killme2.bin",temp,0);
    return(0);
}

In the attack code above, the string "0.....9" represents the attack string we will use. Notice that when we call execl() we need to pass in the attack string. Depending on the length of the attack string the Arguments section of memory will grow. Also notice that the first parameter should go into the Environment Variables section of memory. What this means is that the length of these parameters need to be exactly the same as the length of the respective parameters in the final attacking program. If they are not the same, the addresses and offsets may not be correct.

1. Compile the target program (gcc test.c -o test.bin)

2. Compile the attacking program (gcc attack.c -o attack.bin)

3. Run attack.bin to get the address of message (.attack.bin) We should get something that looks like: bfffffa40

4. Now we want to get the address of the same variable using gdb. The steps are exactly the same as the ones above, except this time we use the run command with an extra parameter, a string of the same length as the attackstring (or just the attack string itself). (run "01234567890123456789") This time, for the output of print &message, we get 0xbfffffa50.
From the difference of the two numbers we can conclude that for any program that we are running with the format of `execl` and `gdb` described above, with parameters of the same string lengths (*this is important*), the address given by `gdb` will be 0x10 more than the actual address.

Now that we know how to get the address of a variable we can continue with more questions.

### 3.2.2 Buffer Overflow Questions

1. Alter the `rootecho` program to print out the address of the buffer. (Don’t make this permanent. You are doing this step so it shows that the address should be the same as `gdb`)

2. Utilizing `gdb`, get the address of the buffer (*Don’t forget to find the offset also*). Please note that the address will change when the length of the input arguments to `execl` change (so make sure the length of the parameter passed to `execl` and to run in `gdb`’s `run` are the same as the length you decided on in the Analysis section). The address you get should be the same if the offset found in the above mentioned test is accounted for.

3. Draw a memory map (like the ones in the supplement to lecture 8) of how the `rootecho` process stack looks (*Just the stack nothing more*). Please be detailed, include labels, such as (Saved IP).

4. From the results of 1 and 2, what should the return address of the egg be?

5. Using the map from 3 and your answers to the above questions, design an egg that will overflow the buffer and overwrite the saved IP. (This should be just basic filling in the different parts)

### 3.2.3 Format String Attacks

1. Use a stack dumping format string to get a stack dump up to the beginning of buffer.

2. Using ‘x’ as the location of buffer and the output from question 1, build a stack map up to the point where the buffer is. Label all the addresses with respect to x (i.e. x-0xff — 0x08000221 (saved IP)) Also, include the address of where a saved IP should be located. (You have 2 to choose from, the saved IP from main and the one from printf())

3. From question 1 and your lecture notes, what is the attack string going to look like? (This must be exact in terms of the number of %x’s needed and the length of this string should be the same as the length of the final attack string. You should use the %8x and etc to be place holders for %hn as shown in the slides) Remember you want to overwrite the saved IP with the address of somewhere in the nop region.

4. At this point all we need is the address of the nop area, and the address of a saved IP. In question 2 you determined the offset of one of the saved IP’s from the beginning of buffer. Since the nop area is going to be copied into the buffer, from your attack string format find the offset from the beginning of the attack string to the nop area of your attack string. Given these two offsets, once we have the "real" address of the buffer we will know the address of both.

5. Using the original `rootecho`, (without the addition of the printf to print the address of the buffer) and `gdb` find the address of the buffer.

6. Using the answers to the above questions, build a detailed stack map. Using real addresses.

7. Assuming you have an attack string format of "targetAdd dummyword targetAdd-2", find the decimal representation of the upper 2 bytes to be written (the upper 2 bytes of the address that is somewhere in the nop area) and the lower 2 bytes. What is the value of the first and second %—–x’s that will ensure that by the time %hn is processed, the number of characters printed out will be exactly the ones calculated above. (Fill in the —–’s)

8. Using the results from the previous questions, fill in the values in the attack string determined in question 3.
3.3 Deployment

Remember that you should be compiling and running the attack programs under a non-root account. Please use the following shellcode for your attack programs:

```
"\xeb\x1d\x5e\x29\xc0\x88\x46\x07\x89\x46\x0c\x89\x76\x08\xb0\x0b\x87\xf3\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\x80\x80\x29\x40\xcd\x80\xe8\xde\xff\xff\xff/bin/sh"
```

Now, using the following format for an attacking program,

```
int main()
{
    char *egg;

    ..... FILL IN ..... 

    //execute rootecho and pass in the egg as parameter
    execl("/usr/bin/rootecho", "rootecho", egg, 0);
    return 0;
}
```

attack rootecho using both a buffer overflow attack and a format string attack.

3.3.1 Suggestions

For the buffer overflow problem you may use the following function to get the return address to use. It is up to you to understand why it works. Also, you may have to subtract \( x, 0 < x < 220 \) from the number returned by ret_address.

```
unsigned long ret_address(void)
{
    __asm__("movl %esp, %eax");
}
```

It might be a good idea to build the egg or format string using variables, such as return address, instead of hardcoding everything as in the supplementary lecture notes.

4 Hand In

Hand in the source code for both the format string attack and for the buffer overflow attack.

Hand in a report that includes, the answers to the above questions. Screenshots that show you have succeeded. These screenshots should have a before and after picture showing the output from the "whoami" command (The first should show something that is not root, and the after should show root) Zip (.zip) up your work and email it to: sraio01@utopia.poly.edu with the subject CS392/681 Lab 6. If you do not have this subject you will be graded harshly.