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, since there is no better way to learn than to experience it first hand, 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

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/

3 Your Task

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

```c
#include<stdio.h>
int main(int argc, char *argv[]) {
    char buffer[600];
    int i;
    setuid(0);
    for (i = 1; i < argc; ++i) {
       strcpy(buffer, argv[i]);
       printf(buffer);
    }
    return 0;
}
```

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 rootech)
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 the following questions:

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 this buffer?
3. Does the code have any buffer overflow and format string vulnerabilities?
4. As we 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?

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? Why (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?

3.2 Design

The questions in this section will help you better understand how the attacks work.

3.2.1 Buffer Overflow Questions

1. Utilizing gdb or by simply altering rootecho. Get the address of the buffer. Please note that the address will change when the length of the input arguments (argv) change.
2. Draw a memory map (like the ones in the supplement to lecture 8) of how the rootecho process stack looks. Notice that only the stack is sufficient. Please be detailed, include labels, such as (Saved IP).
3. Using the map from 2 and your answers to the above questions, design an egg that will overflow the buffer and overwrite the saved IP.
4. From the egg in part 2, what should the return address of the egg be?
3.2.2 Format String Attacks

1. Complete questions 1 and 2 from the previous subsection, finding address and building a memory map. This time, use an attack string to do a partial stack dump and therefore get a more accurate look at memory. One thing to notice is that, by doing a partial stack dump with an attack string, you will start dumping the stack at the point when printf() is called. This means that you should have two “saved ips” to choose from.

2. From the map created in 1, identify the saved IP you wish to overwrite and its offset from the start of “buffer”.

3. Using the results from the previous questions, build an attack string that will overwrite the saved IP of choice with the address of the NOP region in your attack string.

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\n\x0b\x0b\x87\x3f\x8d\x8d\x08\x8d\x53\x0c\xcd\x80\x29\xc0\x40\xcd\x80\xe8\xe6\xff\xff\xff/bin/sh"
```

Now, using the following format for an attacking program,

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

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

    //execute rootecho and pass in the egg as parameter
    exec1("/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.

```c
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)