Homework 02

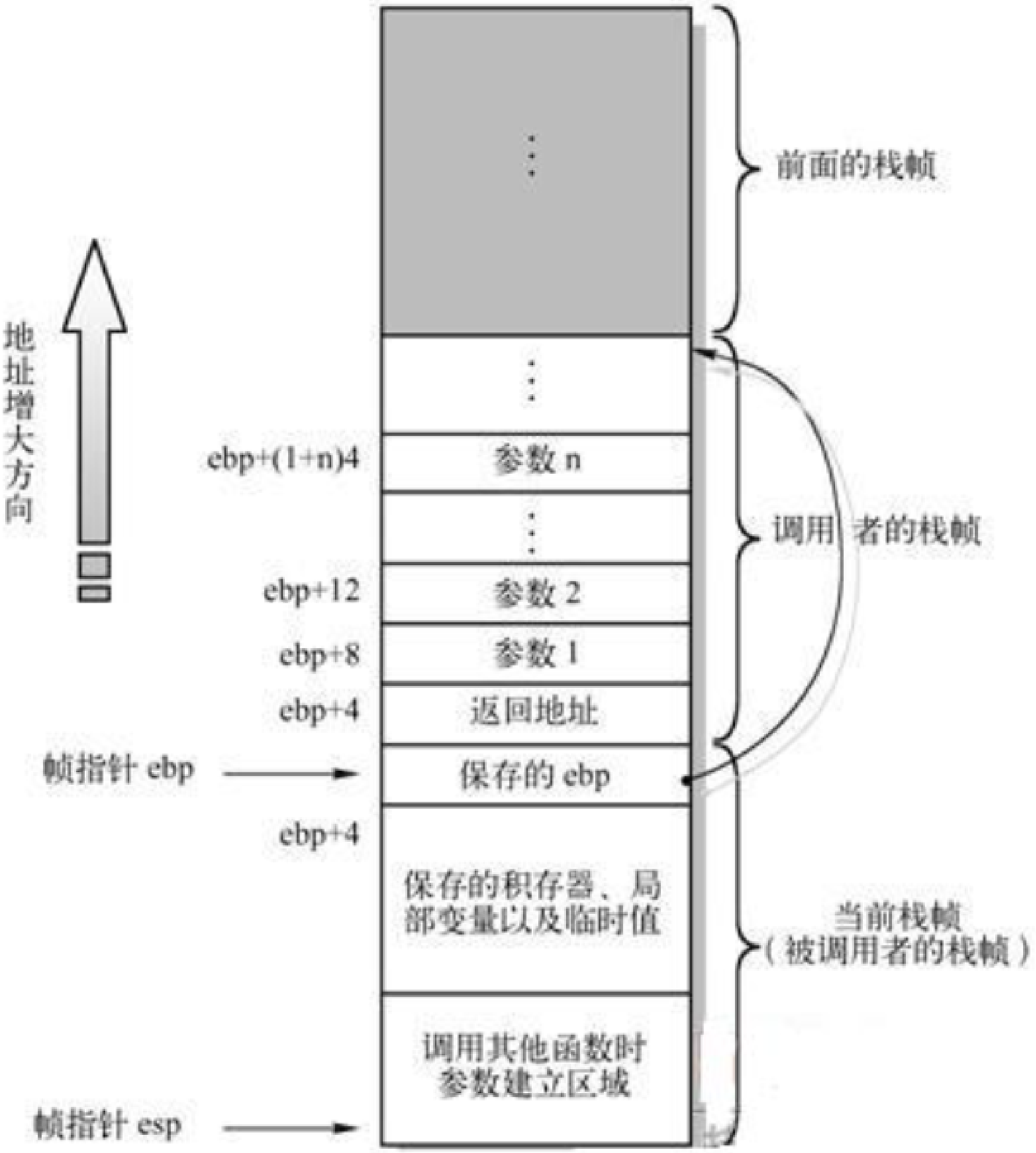
- Introduction
  - I used the method of simulating **buffer overflow** to carry out an attack. In information security and programming, a buffer overflow, is an error where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. So, the hackers may use this way to overload or obtain some important information, which may result in crucial loss.
  - In my algorithm, the simulation is running **under Windows XP environment**. Due to the good protection in Windows 10, which is my computer origin environment, It's better to finish this task under XP. Under Windows XP, some C debug tools have better environment to obtain the value in memory and register, and that is convenient for my task.

- Programming Environment
  - Under Windows XP
  - Using debug tools: VC6
  - Including lib: `<windows.h>`
  - The address in `code.c` should be changed according to user's actual condition. And the length of invalid str like `11111111` should be adjusted as well. (After test, I found that under Windows XP OS, the length is always 8).

- Design Way

  First, let me introduce the principle of this algorithm.

  When calling subroutine in C, the stack will be processed as the following:



So, if the buffer flow is serious that the string overload the **返回地址**, then we can return to some function or information intentionally.

Then is my programming details.

There are a `main` and two sub functions `func` and `warnBox` in my program.

```
void func(const char* inputMsg);
void warnBox();
```

The attack process will be finished in two runs:

- First run

  In the `main`, I obtain the address of `func` and `warnBox`, and define a malicious message to input into `func`. In `warnBox`, the windows will jump out a messageBox, which is a warning of overflow.

  ```
  printf("Address of function = %p\n", func);
  printf("Address of Important message example = %p\n", warnBox);
  char inputMsg[];
  ```

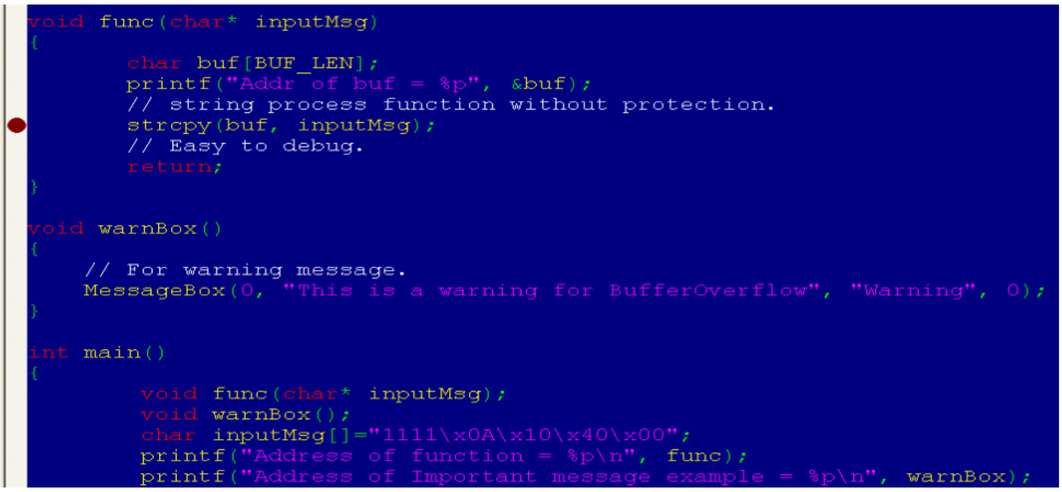
```
void warnBox()
{
    // For warning message.
    // In practice, the malicious code could point to a more important 
area.
    MessageBox(0, "This is a warning for BufferOverflow", "Warning", 
0);
}
```

And in `func`, I defined a buffer `buf` with length of 8. Here I use the system function `strcpy` to copy the value in `inputMsg` to buffer. This way is dangerous. We also output the address of `buf`.

```
void func(const char* inputMsg)
{
        char buf[BUF_LEN];
        printf("Addr of buf = %p", &buf);
        // string process function without protection.
        strcpy(buf, inputMsg);
        return;
}
```

Now, we start the debug program.

First we add a breakpoint in `func`.

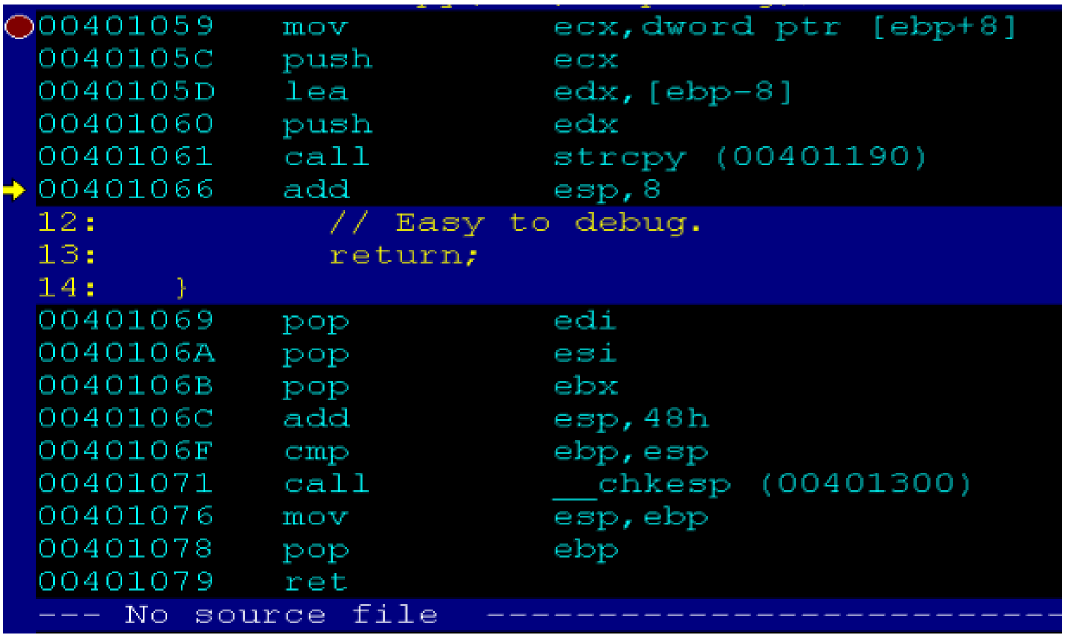


Then step into this step.

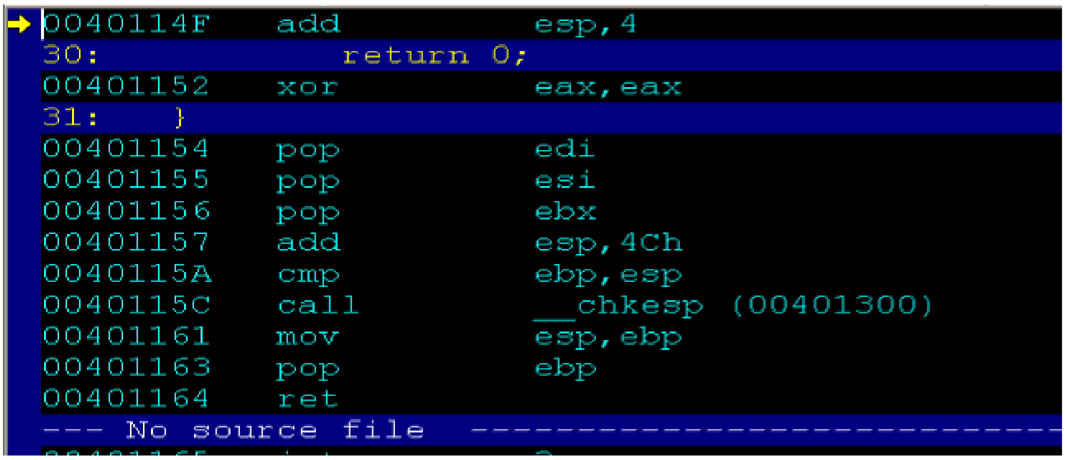
Here we get the address of three function. So to overload the return address, we can define the `inputMsg` as `########\x0A\x10\40\x00`, where the hex string is the address of `warnBox`, then the program can return to `warnBox`.

Next, we should confirm the length between `buf` and `ret ip`. Using disassembly tool in VC6, we can find the corresponding assembly of `return` in `func` .

Disassembly code in `func`:

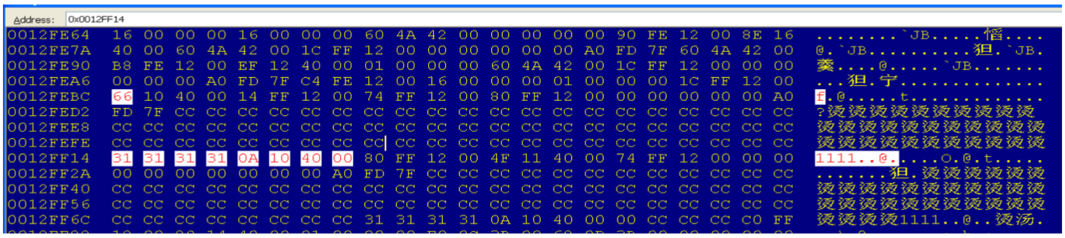


Disassembly code in `main`:



We can find that the return address of `func` is `0x0040114F`.

We debug the program again, in the memory block, we can find that when CPU call the `func` , this return ip is pushed into the stack, as well as the buffer we provide.



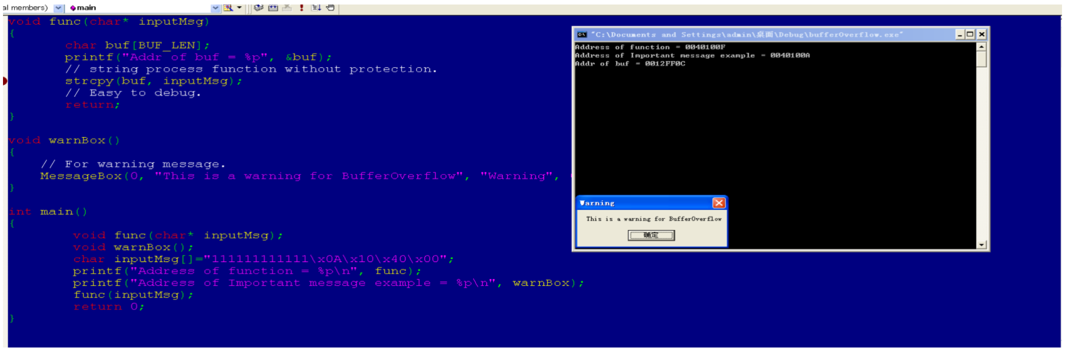
So we can find that there are 8 intervals between the end of buffer and the return ip (Now the buffer is `1111\x0A\x10\x40\x00`). So we can say, when the buffer is filled as `111111111111\x0A\x10\x40\x00`, the return ip will be replaced by the start ip of `warnBox`.

- Next Run

  This time we defined the `inputMsg` as

  ```
  char inputMsg[]="111111111111\x0A\x10\x40\x00";
  ```

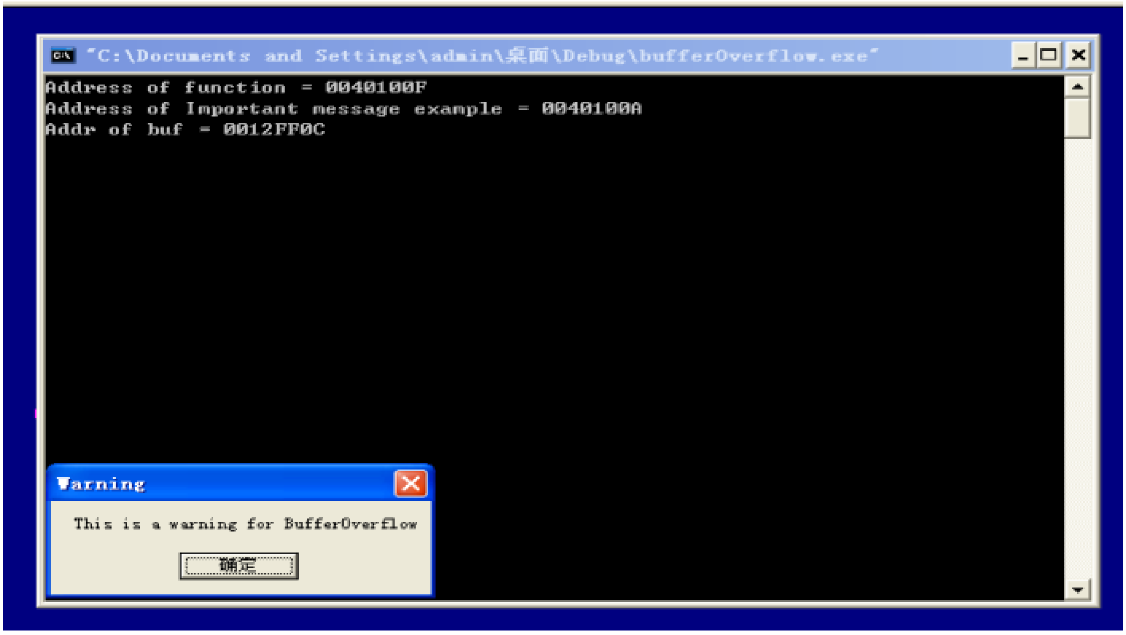
  And run the program and get the result.



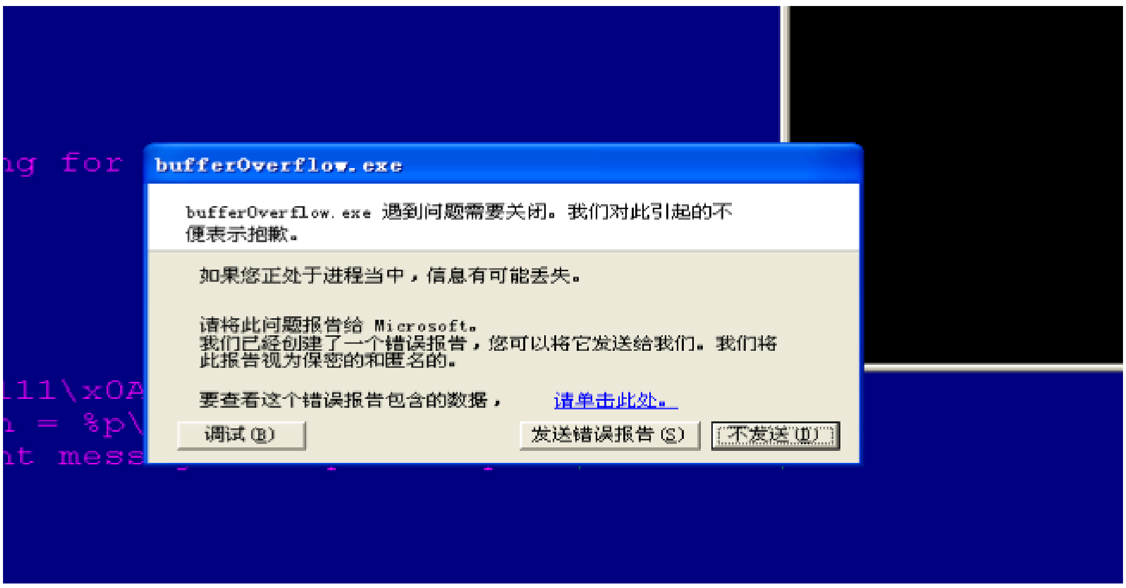
We find that the messageBox is called, which means we successfully overwrite the return ip and go into the `warnBox` module. Because in this module we just simply call a warning box, the program is harmless. However, hacker may do something harmful if there is such a weakness in industry programs.

- Results

  The overflow attack is successfully finished.



And CPU give us a warning as well:



We can say that this weakness is dangerous in industry, which should be took good care of.

- How to defend?
  - The C compiler trusts the programmer enough to think that when performing string operations like `strcpy()`, the length of the arguments should be checked by the programmer. If the programmer ignores this, the program will be vulnerable and the attacker can take advantage of it, so be careful when writing C. So the best way to defend this is to check the length of buffer. Like this:

```
void func(const char* inputMsg)
{
        char buf[BUF_LEN];
        printf("Addr of buf = %p", &buf);
        // string process function with protection.
        if(strlen(inputMsg) > (sizeof(buf) / sizeof(char)){
            printf("Warning of bufferflow").
            return;
        }
        strcpy(buf, inputMsg);
}
```

  - There are many buffer monitoring softwares on Internet. They can judge if the buffer is overflow. We can adopt some of these softwares to help with monitoring, which will be helpful for our program safety.

- Summary and Experience

  In this process, I learned the basics of buffer overflow and how to design a successful buffer overflow attack. At the same time, I also mastered the relevant defense measures. The downside is that this experiment was done on a personal computer, and hopefully next time it will be done on a real server. This is a very meaningful computer security experiment.