Assignment 0x02

Frank Kaiser – 1742945, Jan Martin – 1796943

December 3, 2017

Contents

1 Task 1: Understanding Assembly

First we tried out the binary and came to the conclusion that it computes the greatest common divisor of two numbers. We then inspected the assembly code of the dostuff function line by line and wrote comments to each line to understand its meaning. This is how it looked like:

```
dostuff():

//1 Basepointer auf stack
0x080484b6 <+0>:    push   ebp
//2 Stackpointer auf Basepointer
0x080484b7 <+1>:    mov    ebp,esp
//3 Vergleiche Zahl in Adresse 0x080484be mit 0x0 (also mit 0)
0x080484b9 <+3>:    cmp    DWORD PTR [ebp+0x8],0x0
// 4 Wenn ungleich 0, Springe zu Zeile +36
0x080484bd <+7>:    jne    0x80484da <dostuff+36>
// 5 Kopiert jmp-Befehl von Zeile +12 ins Register eax
// ebp+0xc = 0x80484c2
0x080484bf <+9>:    mov    eax,DWORD PTR [ebp+0xc]
// 6 Springt zu Zeile + 45
0x080484c2 <+12>:   jmp    0x80484e3 <dostuff+45>
// 7 Schreibt 1. Zahl ins Register eax
0x080484c4 <+14>:   mov    eax,DWORD PTR [ebp+0x8]
// 8 Vergleicht 1.Zahl in Register mit 2. Zahl
0x080484c7 <+17>:   cmp    eax,DWORD PTR [ebp+0xc]
// 9 Wenn Zahl1 <= Zahl2 Springe zu +30
0x080484ca <+20>:   jle    0x80484d4 <dostuff+30>
//10 Sonst: Schreibe 2. Zahl in eax
0x080484cc <+22>:   mov    eax,DWORD PTR [ebp+0xc]
//11 Subtrahierte eax von 1. Zahl
0x080484cf <+25>:   sub    DWORD PTR [ebp+0x8],eax
//12 Springe zu Zeile +36
0x080484d2 <+28>:   jmp    0x80484da <dostuff+36>
//13 Schreibt 1. Zahl in Register
0x080484d4 <+30>:   mov    eax,DWORD PTR [ebp+0x8]
//14 Zieht 1. Zahl (die kleinere) von 2. Zahl ab.
0x080484d7 <+33>:   sub    DWORD PTR [ebp+0xc],eax
// 15 Vergleich, ob 2. Zahl Null ist.
0x080484da <+36>:   cmp    DWORD PTR [ebp+0xc],0x0
// 16 Springt zu +14, wenn 2. Zahl ungleich Null
0x080484de <+40>:   jne    0x80484c4 <dostuff+14>
// 17 Schreibt die 1. Zahl ins Register
0x080484e0 <+42>:   mov    eax,DWORD PTR [ebp+0x8]
// 18 Nimmt oberstes Element von Stack und schreibt es in ebp
// register
```

```
0x080484e3 <+45>:    pop    ebp
// 19 Return
0x080484e4 <+46>:    ret
```

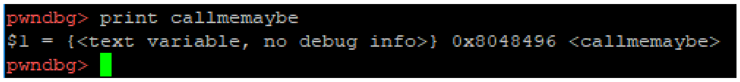
Then we did the same with the main function but no so detailled. By using gdb with break main and nexti we found the address where the result is stored in is: 0x804857d In the corresponding file we found the source code for the main function.

The reversed source Code is in the following file: reverseme.c

Below the source code readable:

```
#include <stdio.h>

int dostuff(int a, int b){

    while (a != 0) {
        if (a > b) {
            a = a - b;
        } else {
            b = b - a;
        }
        if (b == 0){
            return a;
        }
    }
    return b;
}

int main (int argc, char *argv[]) {
    int a, b, c;

    printf("Enter␣two␣positive␣numbers␣seperated␣by␣a␣space:␣");
    if (scanf("%d␣%d",&a, &b) != 2) {
        printf("Numbers,␣separated␣by␣space,␣I␣said!\n");
        return 1;
    }

    if (a < 0 || b < 0){
        printf("Positive,␣I␣said!\n");
        return -1;
    }

    c = dostuff(a, b);
    printf("dostuff␣says:␣%d\n", c);

    return 0;
}
```

2 Task 2: Exploiting a Simple Buffer Overflow on the Stack



```
pwndbg> print callmemaybe
$1 = {<text variable, no debug info>} 0x8048496 <callmemaybe>
pwndbg> 
```

Figure 1: hexcode

0x8048496 = HI

Buffer located at: 0xffffd640

Address it crashes on in EIP: 0x5c363978

python -c ”print(bin(0xffffd640-0x5c363978))”

0b10100011110010011001110011001000

python -c ”print(len(’10100011110010011001110011001000’))”

32

We think the buffer has size 32, however we didn’t manage to call the callmemaybe function. H is the 8th letter in the alphabet, so after HHHH the buffer should be full. This was a example input:

AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH\x96\x84\x04\x08\x0x

We also tried this:

AAAABBBBCCCC\x96\x84\x04\x08\x0x

Unfortunately to no avail..