EECS 765 - Introduction to Cryptography and Computer Security

Programming Assignment 3

Amin Mamandipoor

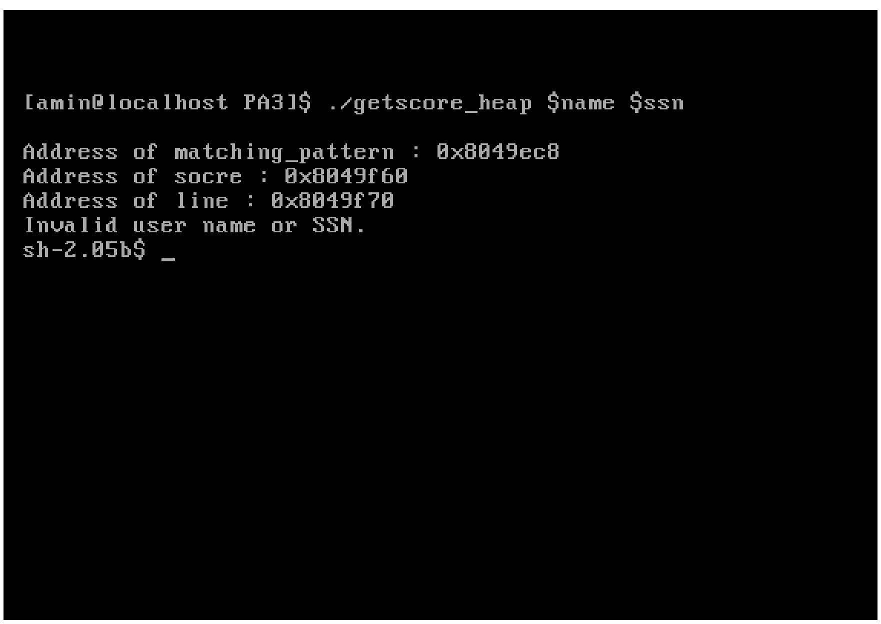
Introduction

This assignment delves into the details of heap memory management as we work with a new version of the *get_score* program. This updated version dynamically allocates memory on the heap rather than the stack. Our main focus is on understanding heap overflow vulnerabilities, similar to stack overflow vulnerabilities, but related to unauthorized shell access. We aim to manipulate heap memory to develop an exploit.

Running the Exploit

The exploit works properly RedHat8 machine using the specified command:

- *./getscore_heap $name $ssn*



[Proof of concept that the exploit works]

Background

Dynamic memory allocation takes place in the heap memory segment, reserved for data that must be allocated and deallocated during the execution of a program. To manage this memory, the *C* programming language provides two fundamental functions: *malloc()* and *free()*. These functions are widely utilized to allocate and release memory in the heap. Below, we present examples of their usage:

```
int* dynamicArray = (int*)malloc(10 * sizeof(int));

free(dynamicArray); // used to avoid memory leaks
```

Each heap in memory contains several crucial components, including a *heap_info* structure, a *malloc_state* structure, and a variable number of *malloc_chunk* structures. The *heap_info* structure serves as an essential management unit for the heap. It defines the size of the heap, points to the memory area allocated for the heap, and also holds a reference to the previous *heap_info* structure, enabling the management of multiple heaps if necessary.

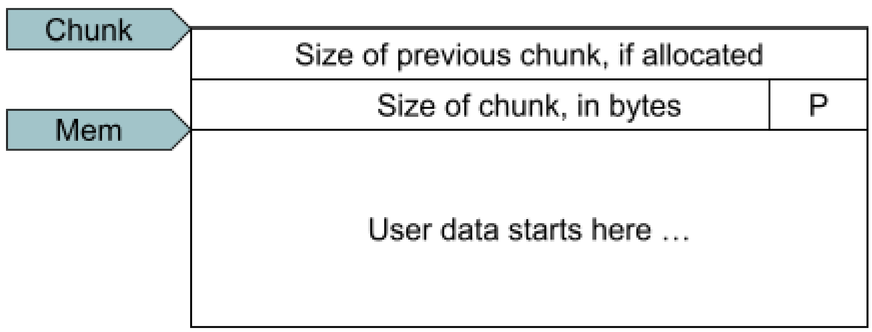
Within these heaps, we find the *malloc_chunk* structure, which plays a key role in memory allocation. The *malloc_chunk* structure is responsible for storing information about allocated memory chunks. It includes details such as the size and status of the chunk, and pointers to the next and previous chunks in the linked list. This structure is essential for memory management within the heap.

- Size of previous adjacent chunk
- Size of current (this) chunk
- If free, pointer to the next *malloc_chunk*
- If free, pointer for the previous *malloc_chunk*

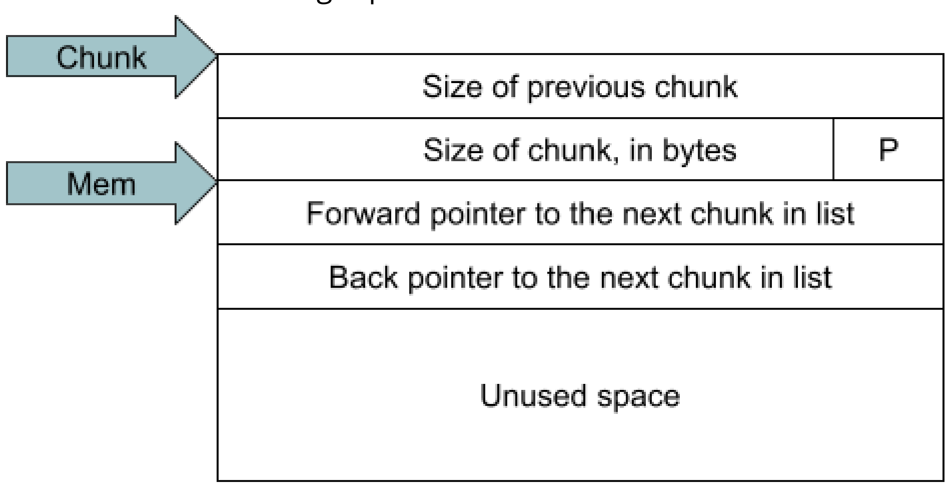
```
Struct malloc_chunk {
```

```
        INTERNAL_SIZE_T       prev_size;
        INTERNAL_SIZE_T            size;
        Struct malloc_chunk*       fd;
        Struct malloc_chunk*       bk;
    }
```

An allocated block of memory is viewed in the following way:



Also, a free chunk has the following representation:



Free chunks are organized into circular *doubly-linked lists*. Each chunk on a free list contains forward and back pointers to the next and previous chunks in the list. These pointers in a free chunk occupy the same four bytes of memory as user data in the allocated chunk. Chunk size is stored in the last four bytes of the free chunk enabling adjacent free chunks to be merged to avoid fragmentation of memory.

Developing the Exploit

This assignment aims to overflow the heap buffer and overwrite the metadata in order to gain full control over the program. In order to accomplish this, we need to follow these steps:

1. Heap Memory Overwrite:

We exploit this vulnerability by writing more data than is allocated within a particular heap memory chunk. This can be achieved by providing input that exceeds the allocated buffer size, causing data to overwrite adjacent memory regions.

2. Create a Fake Heap Structure:

Once the heap memory is overwritten, we create a fake heap structure. This structure is crafted to look like a legitimate free chunk of memory. It's important to make it appear convincing to deceive the program's memory management.

3. Control Over Memory Overwrite:

When we successfully triggered the program to release (free) the corrupted heap memory chunk, the program's memory management system updates its data structures, including linked lists or other metadata, to indicate that this chunk is now free and available for reuse. Now, we have control over this fake free chunk and can manipulate its content.

- **Controlling the "*Where*":**

  We aim to control where the program's execution will jump to after the attack. By modifying these memory locations, we hope to point them to our malicious code and ultimately the shellcode.

- **Controlling the "*What*":**

This represents the address of the malicious code that we have injected into the corrupted heap. Also, we need to be careful and make sure that the shellcode can "jump over" any holes or areas of memory that could interfere with its execution.

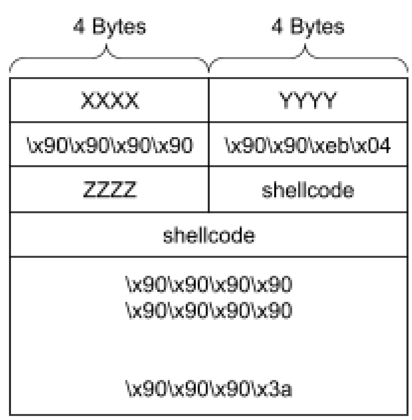
I carefully examined the *getscore_heap.c* file. There are two input arguments to the program, one for the *name*, and the other for the *SSN*. There are dynamically allocated buffers called *matching_pattern* with a size of *strlen(name)+17*. The *score* buffer has a size of 10. Finally, the *line* has a size of 127. The following statements summarize the text:

```
matching_pattern = (char *)malloc(strlen(name)+17);
score = (char *)malloc(10);
line = (char *)malloc(128);
```

The *name* is an input argument to the *getscore_heap* program. The following statements copy data to *matching_pattern*:

```
strcpy(matching_pattern, name);
strcat(matching_pattern, ":");
strcat(matching_pattern, ssn);
```

In the initial step, the *name* was copied to the beginning of the *matching_pattern* buffer. Following this, a colon *: (\x3a)* and the *SSN* were appended to the end of the *matching_pattern*. Both the *name* and *SSN* are input arguments to the program, making it possible for anyone with execute privileges to manipulate these input arguments and potentially send malicious input to the program. The *name* and *SSN* are used in constructing the exploit or the malicious input. Let's focus on the *name* argument first. The size of the *name* was chosen to be 127 bytes, and it was constructed as illustrated in the figure below:



The forward and backward links are represented by XXXX and YYYY, respectively. Afterwards, 6 Nop sleds are appended to the name. A JMP 0x6 instruction is then placed to direct the code to jump to the shellcode's beginning. In the next step, the shellcode is appended to the *name*, with its size exactly 45 bytes. The remaining bytes are appended with NOPs in order to keep the name size at 127 bytes. *name* is sent as the first argument to the *getscore_heap* program, where it is stored in the *matching_pattern* buffer. Also, a colon (*0x3a*) is then appended to the buffer, resulting in a total of 128 bytes in the buffer.

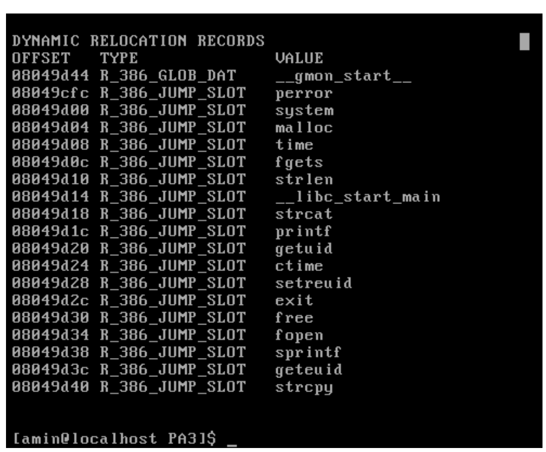
Following this, the *SSN* input argument is appended to the buffer. As a part of the sent input, a fake heap structure was created for the *SSN*, and the second input argument was constructed as follows:

4 Bytes	4 Bytes
\x90\x90\x90\x90	\x90\x90\x90\x90
\x90\x90\x90\x90	\x90\x90\x90\x90
0xffffffff (-1)	0xffffffff (-1)
GOT_entry - 12	Buffer + 8

The initial part of the input consists of NOPs padding. NOPs were chosen for their value *(\x90)* because it is an even value, which leads the adjacent heap chunk to interpret it as a free chunk.

The GOT entry address was found using the following command:

$ objdump -R getscore_heap

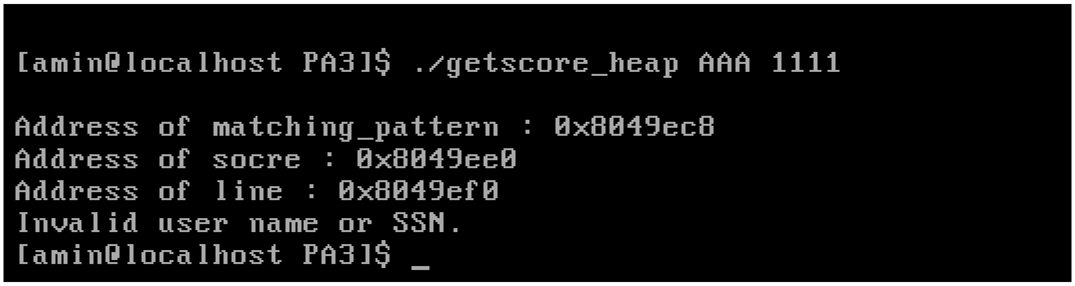


```
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049d44 R_386_GLOB_DAT    __gmon_start__
08049cfc R_386_JUMP_SLOT   perror
08049d00 R_386_JUMP_SLOT   system
08049d04 R_386_JUMP_SLOT   malloc
08049d08 R_386_JUMP_SLOT   time
08049d0c R_386_JUMP_SLOT   fgets
08049d10 R_386_JUMP_SLOT   strlen
08049d14 R_386_JUMP_SLOT   __libc_start_main
08049d18 R_386_JUMP_SLOT   strcat
08049d1c R_386_JUMP_SLOT   printf
08049d20 R_386_JUMP_SLOT   getuid
08049d24 R_386_JUMP_SLOT   ctime
08049d28 R_386_JUMP_SLOT   setreuid
08049d2c R_386_JUMP_SLOT   exit
08049d30 R_386_JUMP_SLOT   free
08049d34 R_386_JUMP_SLOT   fopen
08049d38 R_386_JUMP_SLOT   sprintf
08049d3c R_386_JUMP_SLOT   geteuid
08049d40 R_386_JUMP_SLOT   strcpy

[amin@localhost PA3]$ _
```

[*output of* `objdump` *command*]

$ getscore_heap AAA 1111

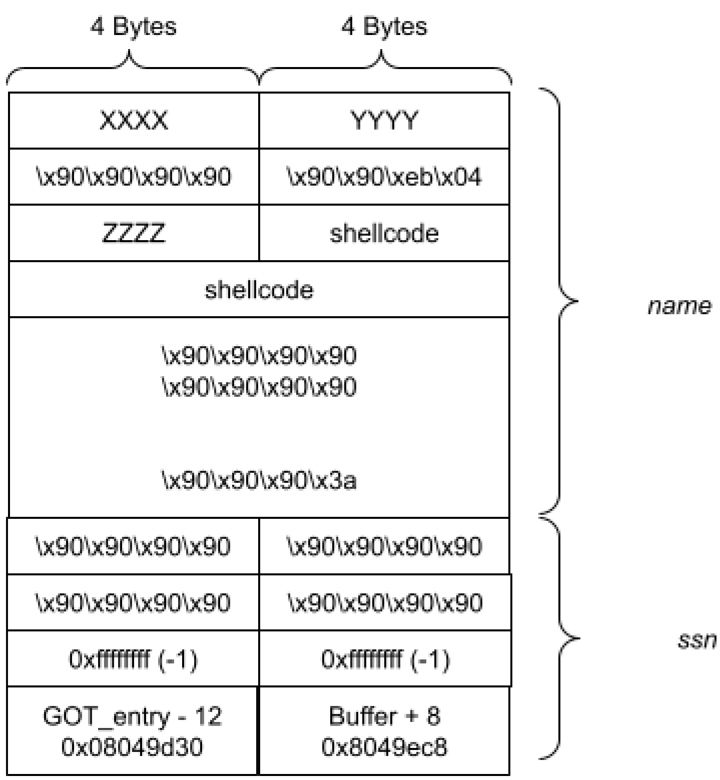


```
[amin@localhost PA3]$ ./getscore_heap AAA 1111

Address of matching_pattern : 0x8049ec8
Address of socre : 0x8049ee0
Address of line : 0x8049ef0
Invalid user name or SSN.
[amin@localhost PA3]$ _
```

[output of getscore_heap for getting buffer's address]

Structure of the Malicious Input:



References and Collaborations

Watched the lecture’s video multiple times and online sources for understanding heap memory management better.