PROCESS MEMORY AND MEMORY CORRUPTIONS

The prerequisite for this part of the tutorial is a basic understanding of ARM assembly (covered in the first tutorial series "ARM Assembly Basics"). In this chapter you will get an introduction into the memory layout of a process in a 32-bit Linux environment. After that you will learn the fundamentals of Stack and Heap related memory corruptions and how they look like in a debugger.

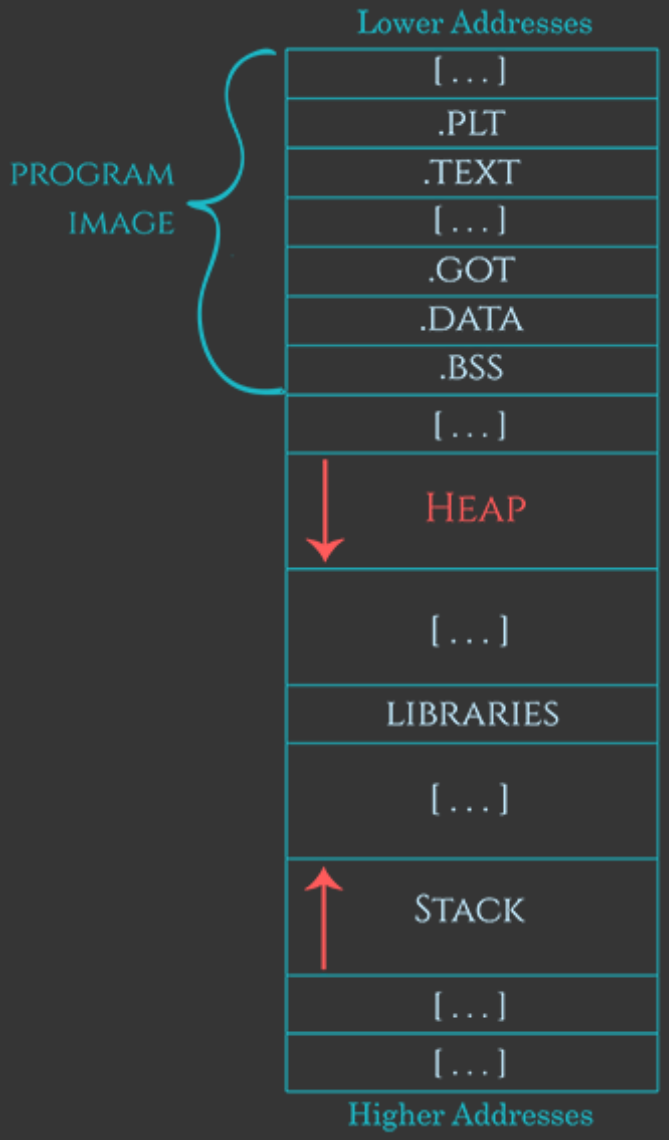
The examples used in this tutorial are compiled on an ARMv6 32-bit processor. If you don't have access to an ARM device, you can create your own lab and emulate a Raspberry Pi distro in a VM by following this tutorial: Emulate Raspberry Pi with QEMU. The debugger used here is GDB with GEF (GDB Enhanced Features). If you aren't familiar with these tools, you can check out this tutorial: Debugging with GDB and GEF.

MEMORY LAYOUT OF A PROCESS

Every time we start a program, a memory area for that program is reserved. This area is then split into multiple regions. Those regions are then split into even more regions (segments), but we will stick with the general overview. So, the parts we are interested are:

1. Program Image
2. Heap
3. Stack

In the picture below we can see a general representation of how those parts are laid out within the process memory. The addresses used to specify memory regions are just for the sake of an example, because they will differ from environment to environment, especially when ASLR is used.



Program Image region basically holds the program's executable file which got loaded into the memory. This memory region can be split into various segments: .plt, .text, .got, .data, .bss and so on. These are the most relevant. For example, .text contains the executable part of the program with all the Assembly instructions, .data and .bss holds the variables or pointers to variables used in the application, .plt and .got stores specific pointers to various imported functions from, for example, shared libraries. From a security standpoint, if an attacker could affect the integrity (rewrite) of the .text section, he could execute arbitrary code. Similarly, corruption of Procedure Linkage Table (.plt) and Global Offsets Table (.got) could under specific circumstances lead to execution of arbitrary code.

The **Stack** and **Heap** regions are used by the application to store and operate on temporary data (variables) that are used during the execution of the program. These regions are commonly exploited by attackers, because data in the Stack and Heap regions can often be modified by the user's input, which, if

not handled properly, can cause a memory corruption. We will look into such cases later in this chapter.

In addition to the mapping of the memory, we need to be aware of the attributes associated with different memory regions. A memory region can have one or a combination of the following attributes: **R**ead, **W**rite, e**X**ecute. The **Read** attribute allows the program to read data from a specific region. Similarly, **Write** allows the program to write data into a specific memory region, and **Execute** – execute instructions in that memory region. We can see the process memory regions in GEF (a highly recommended extension for GDB) as shown below:

```
azeria@labs:~/exp $ gdb program
...
gef> gef config context.layout "code"
gef> break main
Breakpoint 1 at 0x104c4: file program.c, line 6.
gef> run
...
gef> nexti 2
------------------------------------------------------------------------------------------[ co
...
      0x104c4 <main+20>        mov    r0,  #8
      0x104c8 <main+24>        bl     0x1034c <malloc@plt>
->    0x104cc <main+28>        mov    r3,  r0
      0x104d0 <main+32>        str    r3,  [r11,  #-8]
...
gef> vmmap
Start      End        Offset     Perm Path
0x00010000 0x00011000 0x00000000 r-x /home/azeria/exp/program <---- Program Image
0x00020000 0x00021000 0x00000000 rw- /home/azeria/exp/program <---- Program Image continues..
```

```
0x00021000 0x00042000 0x00000000 rw- [heap] <---- HEAP
0xb6e74000 0xb6f9f000 0x00000000 r-x /lib/arm-linux-gnueabihf/libc-2.19.so <---- Shared libra
0xb6f9f000 0xb6faf000 0x0012b000 --- /lib/arm-linux-gnueabihf/libc-2.19.so <---- libc continu
0xb6faf000 0xb6fb1000 0x0012b000 r-- /lib/arm-linux-gnueabihf/libc-2.19.so <---- libc continu
0xb6fb1000 0xb6fb2000 0x0012d000 rw- /lib/arm-linux-gnueabihf/libc-2.19.so <---- libc continu
0xb6fb2000 0xb6fb5000 0x00000000 rw-
0xb6fcc000 0xb6fec000 0x00000000 r-x /lib/arm-linux-gnueabihf/ld-2.19.so <---- Shared library
0xb6ffa000 0xb6ffb000 0x00000000 rw-
0xb6ffb000 0xb6ffc000 0x0001f000 r-- /lib/arm-linux-gnueabihf/ld-2.19.so <---- ld continues..
0xb6ffc000 0xb6ffd000 0x00020000 rw- /lib/arm-linux-gnueabihf/ld-2.19.so <---- ld continues..
0xb6ffd000 0xb6fff000 0x00000000 rw-
0xb6fff000 0xb7000000 0x00000000 r-x [sigpage]
0xbefdf000 0xbf000000 0x00000000 rw- [stack] <---- STACK
0xffff0000 0xffff1000 0x00000000 r-x [vectors]
```

The **Heap section** in the vmmap command output **appears only after some Heap related function was used**. In this case we see the **malloc** function being used to create a buffer in the Heap region. So if you want to try this out, you would need to debug a program that makes a **malloc** call (you can find some examples in this page, scroll down or use find function).

Additionally, in Linux we can inspect the process' memory layout by accessing a process-specific "file":

```
azeria@labs:~/exp $ ps aux | grep program
azeria   31661 12.3 12.1  38680 30756 pts/0    S+   23:04   0:10 gdb program
azeria   31665  0.1  0.2   1712   748 pts/0    t    23:04   0:00 /home/azeria/exp/program
azeria   31670  0.0  0.7   4180  1876 pts/1    S+   23:05   0:00 grep --color=auto program
```

```
azeria@labs:~/exp $ cat /proc/31665/maps
00010000-00011000 r-xp 00000000 08:02 274721     /home/azeria/exp/program
00020000-00021000 rw-p 00000000 08:02 274721     /home/azeria/exp/program
00021000-00042000 rw-p 00000000 00:00 0          [heap]
b6e74000-b6f9f000 r-xp 00000000 08:02 132394     /lib/arm-linux-gnueabihf/libc-2.19.so
b6f9f000-b6faf000 ---p 0012b000 08:02 132394     /lib/arm-linux-gnueabihf/libc-2.19.so
b6faf000-b6fb1000 r--p 0012b000 08:02 132394     /lib/arm-linux-gnueabihf/libc-2.19.so
b6fb1000-b6fb2000 rw-p 0012d000 08:02 132394     /lib/arm-linux-gnueabihf/libc-2.19.so
b6fb2000-b6fb5000 rw-p 00000000 00:00 0
b6fcc000-b6fec000 r-xp 00000000 08:02 132358     /lib/arm-linux-gnueabihf/ld-2.19.so
b6ffa000-b6ffb000 rw-p 00000000 00:00 0
b6ffb000-b6ffc000 r--p 0001f000 08:02 132358     /lib/arm-linux-gnueabihf/ld-2.19.so
b6ffc000-b6ffd000 rw-p 00020000 08:02 132358     /lib/arm-linux-gnueabihf/ld-2.19.so
b6ffd000-b6fff000 rw-p 00000000 00:00 0
b6fff000-b7000000 r-xp 00000000 00:00 0          [sigpage]
befdf000-bf000000 rw-p 00000000 00:00 0          [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0          [vectors]
```

Most programs are compiled in a way that they use shared libraries. Those libraries are not part of the program image (even though it is possible to include them via static linking) and therefore have to be referenced (included) dynamically. As a result, we see the libraries (libc, ld, etc.) being loaded in the memory layout of a process. Roughly speaking, the shared libraries are loaded somewhere in the memory (outside of process' control) and our program just creates virtual "links" to that memory region. This way we save memory without the need to load the same library in every instance of a program.

INTRODUCTION INTO MEMORY CORRUPTIONS

A memory corruption is a software bug type that allows to modify the memory in a way that was not intended by the programmer. In most cases, this condition can be exploited to execute arbitrary code, disable security mechanisms, etc. This is done by crafting and injecting a payload which alters certain memory sections of a running program. The following list contains the most common memory corruption types/vulnerabilities:

1. Buffer Overflows
    - Stack Overflow
    - Heap Overflow
2. Dangling Pointer (Use-after-free)
3. Format String

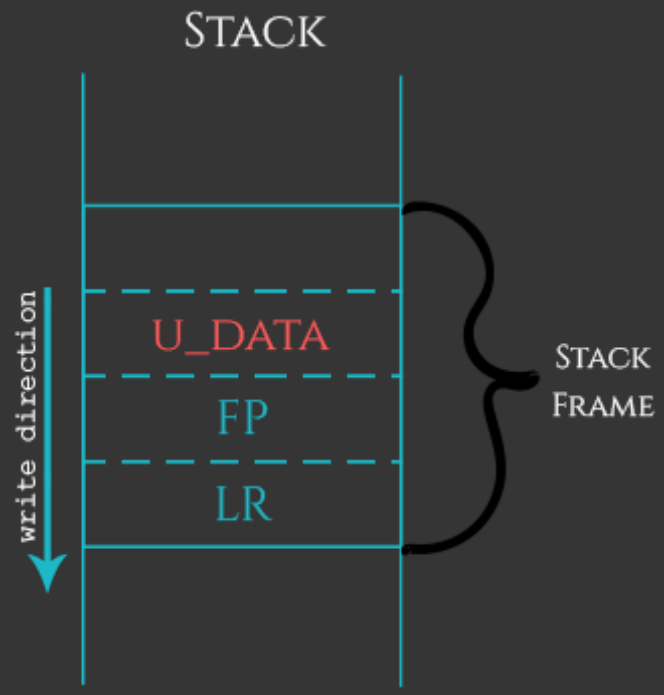
In this chapter we will try to get familiar with the basics of Buffer Overflow memory corruption vulnerabilities (the remaining ones will be covered in the next chapter). In the examples we are about to cover we will see that the main cause of memory corruption vulnerabilities is an improper user input validation, sometimes combined with a logical flaw. For a program, the input (or a malicious payload) might come in a form of a username, file to be opened, network packet, etc. and can often be influenced by the user. If a programmer did not put safety measures for potentially harmful user input it is often the case that the target program will be subject to some kind of memory related issue.

BUFFER OVERFLOWS

Buffer overflows are one of the most widespread memory corruption classes and are usually caused by a programming mistake which allows the user to supply more data than there is available for the destination variable (buffer). This happens, for example, when vulnerable functions, such as **gets**, **strcpy**, **memcpy** or others are used along with data supplied by the user. These functions do not check the length of the user's data which can result into writing past (overflowing) the allocated buffer. To get a better understanding, we will look into **basics** of Stack and Heap based buffer overflows.

Stack Overflow

Stack overflow, as the name suggests, is a memory corruption affecting the Stack. While in most cases arbitrary corruption of the Stack would most likely result in a program's crash, a carefully crafted Stack buffer overflow can lead to arbitrary code execution. The following picture shows an abstract overview of how the Stack can get corrupted.



As you can see in the picture above, the Stack frame (a small part of the whole Stack dedicated for a specific function) can have various components: user data, previous Frame Pointer, previous Link Register, etc. In case the user provides too much of data for a controlled variable, the FP and LR fields might get overwritten. This breaks the execution of the program, because the user corrupts the address where the application will return/jump after the current function is finished.

To check how it looks like in practice we can use this example:

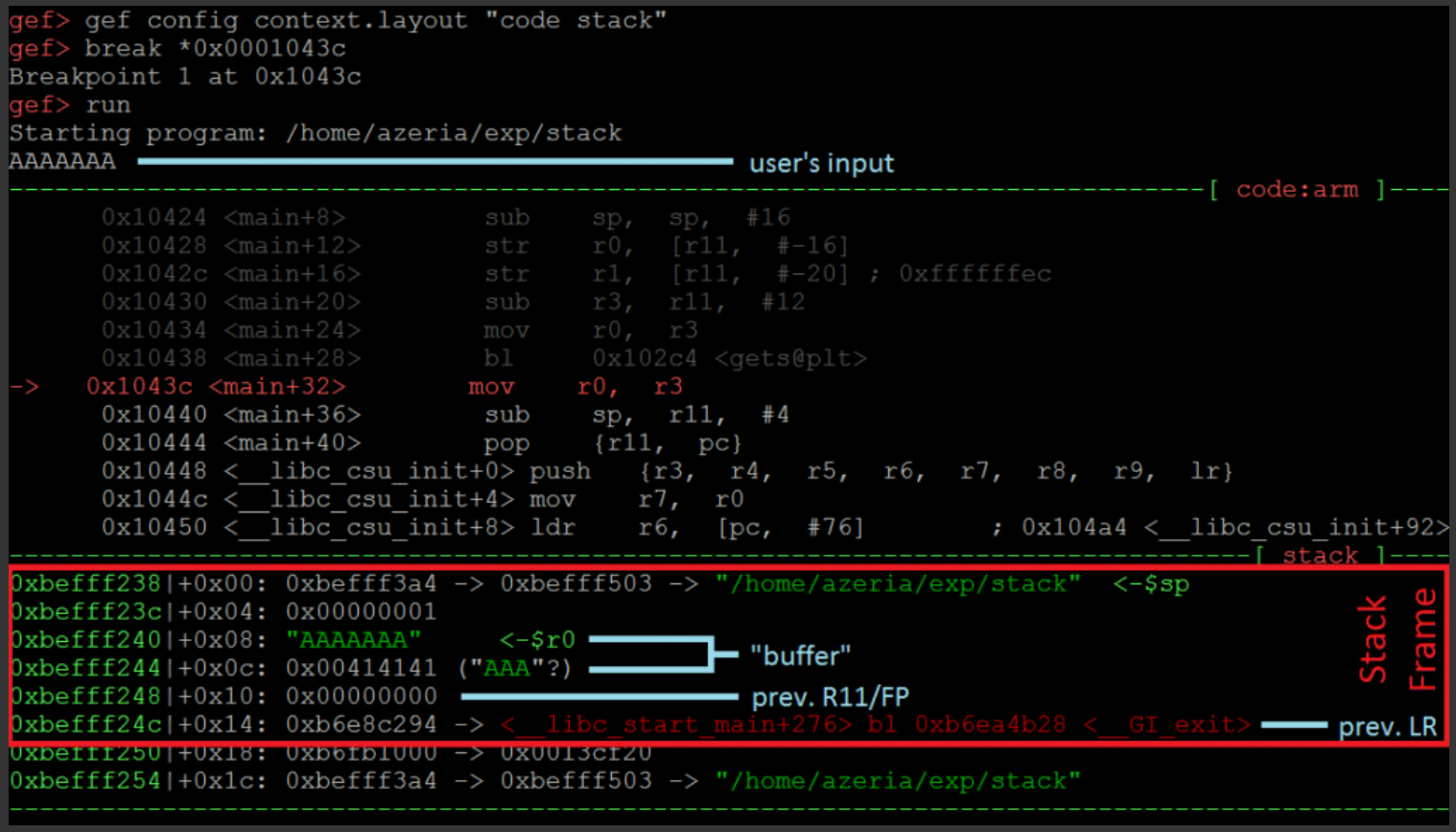
```
/*azeria@labs:~/exp $ gcc stack.c -o stack*/
#include "stdio.h"

int main(int argc, char **argv)
{
char buffer[8];
gets(buffer);
}
```

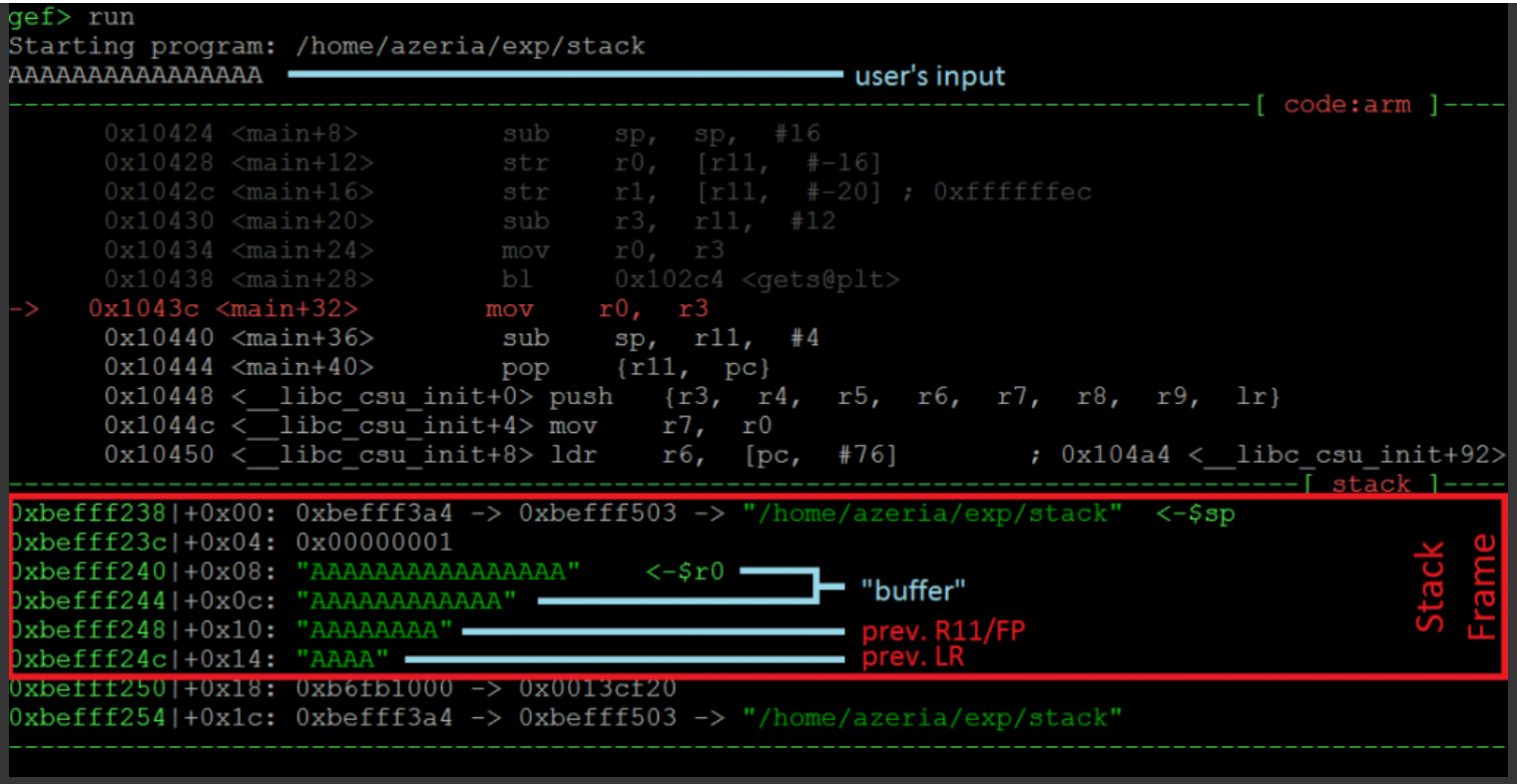
Our sample program uses the variable “buffer”, with the length of 8 characters, and a function “gets” for user’s input, which simply sets the value of the variable “buffer” to whatever input the user provides. The disassembled code of this program looks like the following:

```
azeria@labs:~/exp $ gdb -q stack
GEF for linux ready, type `gef' to start, `gef config' to configure
50 commands loaded for GDB 7.12 using Python engine 2.7
[*] 3 commands could not be loaded, run `gef missing` to know why.
Reading symbols from stack...(no debugging symbols found)...done.
gef> disassemble main
Dump of assembler code for function main:
   0x0001041c <+0>:     push    {r11, lr}
   0x00010420 <+4>:     add     r11, sp, #4
   0x00010424 <+8>:     sub     sp, sp, #16
   0x00010428 <+12>:    str     r0, [r11, #-16]
   0x0001042c <+16>:    str     r1, [r11, #-20] ; 0xffffffec
   0x00010430 <+20>:    sub     r3, r11, #12
   0x00010434 <+24>:    mov     r0, r3
   0x00010438 <+28>:    bl      0x102c4 <gets@plt>
   0x0001043c <+32>:    mov     r0, r3
   0x00010440 <+36>:    sub     sp, r11, #4
   0x00010444 <+40>:    pop     {r11, pc}
End of assembler dump.
```

Here we suspect that a memory corruption could happen right after the function “gets” is completed. To investigate this, we place a break-point right after the branch instruction that calls the “gets” function – in our case, at address 0x0001043c. To reduce the noise we configure GEF’s layout to show us only the code and the Stack (see the command in the picture below). Once the break-point is set, we proceed with the program and provide 7 A’s as the user’s input (we use 7 A’s, because a null-byte will be automatically appended by function “gets”).



When we investigate the Stack of our example we see (image above) that the Stack frame is not corrupted. This is because the input supplied by the user fits in the expected 8 byte buffer and the previous FP and LR values within the Stack frame are not corrupted. Now let’s provide 16 A’s and see what happens.



In the second example we see (image above) that when we provide too much of data for the function “gets”, it does not stop at the boundaries of the target buffer and keeps writing “down the Stack”. This causes our previous FP and LR values to be corrupted. When we continue running the program, the program crashes (causes a “Segmentation fault”), because during the epilogue of the current function the previous values of FP and LR are “poped” off the Stack into R11 and PC registers forcing the program to jump to address 0x41414140 (last byte gets automatically converted to 0x40 because of the switch to Thumb mode), which in this case is an illegal address. The picture below shows us the values of the registers (take a look at $pc) at the time of the crash.

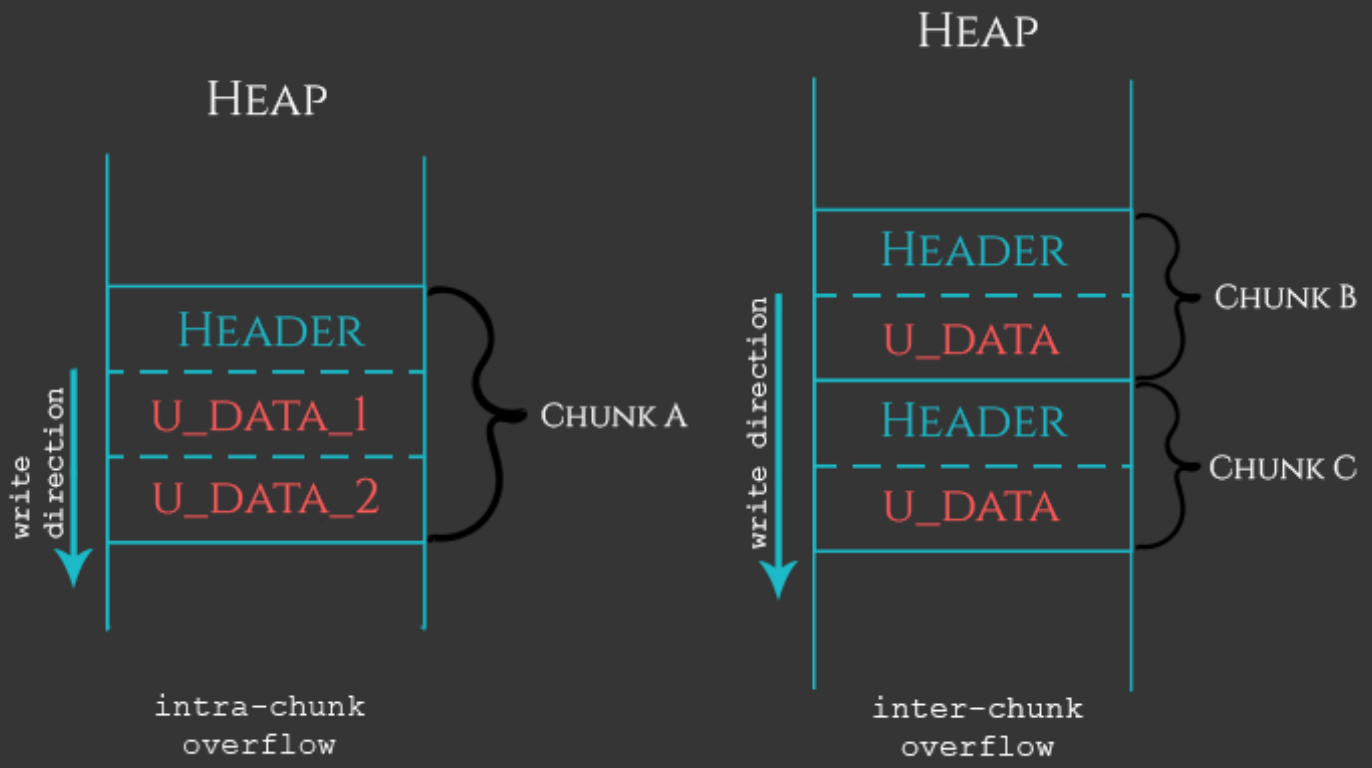
```
gef> gef config context.layout "regs"
gef> continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
[ registers ]
$r0   : 0x00000000
$r1   : 0x00000000
$r2   : 0x00000001
$r3   : 0x00000000
$r4   : 0x00000000
$r5   : 0x00000000
$r6   : 0x000102f4 -> <_start+0> mov r11,  #0
$r7   : 0x00000000
$r8   : 0x00000000
$r9   : 0x00000000
$r10  : 0xb6ffc000 -> 0x0002ff44
$r11  : 0x41414141 ("AAAA"?)
$r12  : 0x00000014
$sp   : 0xbefff250 -> 0xb6fb1000 -> 0x0013cf20
$lr   : 0x41414141 ("AAAA"?)
$pc   : 0x41414140 ("@AAA"?)
$cpsr : [THUMB fast interrupt overflow CARRY ZERO negative]
0x41414140 in ?? ()
```

Heap Overflow

(If you want to learn how to heap works under the hood, read my post on "Understanding the GLibc Heap Implementation")

First of all, Heap is a more complicated memory location, mainly because of the way it is managed. To keep things simple, we stick with the fact that every object placed in the Heap memory section is "packed" into a "chunk" having two parts: header and user data (which sometimes the user controls fully). In the Heap's case, the memory corruption happens when the user is able to write more data than is expected. In that case, the corruption might happen within the chunk's boundaries (intra-chunk Heap overflow), or across the boundaries of two (or more) chunks (inter-chunk Heap overflow). To put things in perspective, let's take a look at the following illustration.



As shown in the illustration above, the intra-chunk heap overflow happens when the user has the ability to supply more data to **u_data_1** and cross the boundary between **u_data_1** and **u_data_2**. In this way the fields/properties of the current object get corrupted. If the user supplies more data than the current Heap chunk can accommodate, then the overflow becomes inter-chunk and results into a corruption of the adjacent chunk(s).

Intra-chunk Heap overflow

To illustrate how an intra-chunk Heap overflow looks like in practice we can use the following example and compile it with "-O" (optimization flag) to have a smaller (binary) program (easier to look through).

```
/*azeria@labs:~/exp $ gcc intra_chunk.c -o intra_chunk -O*/
#include "stdlib.h"
#include "stdio.h"

struct u_data                                          //object model: 8 bytes for name, 4 by
{
 char name[8];
 int number;
};

int main ( int argc, char* argv[] )
{
 struct u_data* objA = malloc(sizeof(struct u_data)); //create object in Heap

 objA->number = 1234;                                  //set the number of our object to a sta
 gets(objA->name);                                     //set name of our object according to u

 if(objA->number == 1234)                              //check if static value is intact
 {
  puts("Memory valid");
 }
 else                                                  //proceed here in case the static value
 {
  puts("Memory corrupted");
 }
}
```

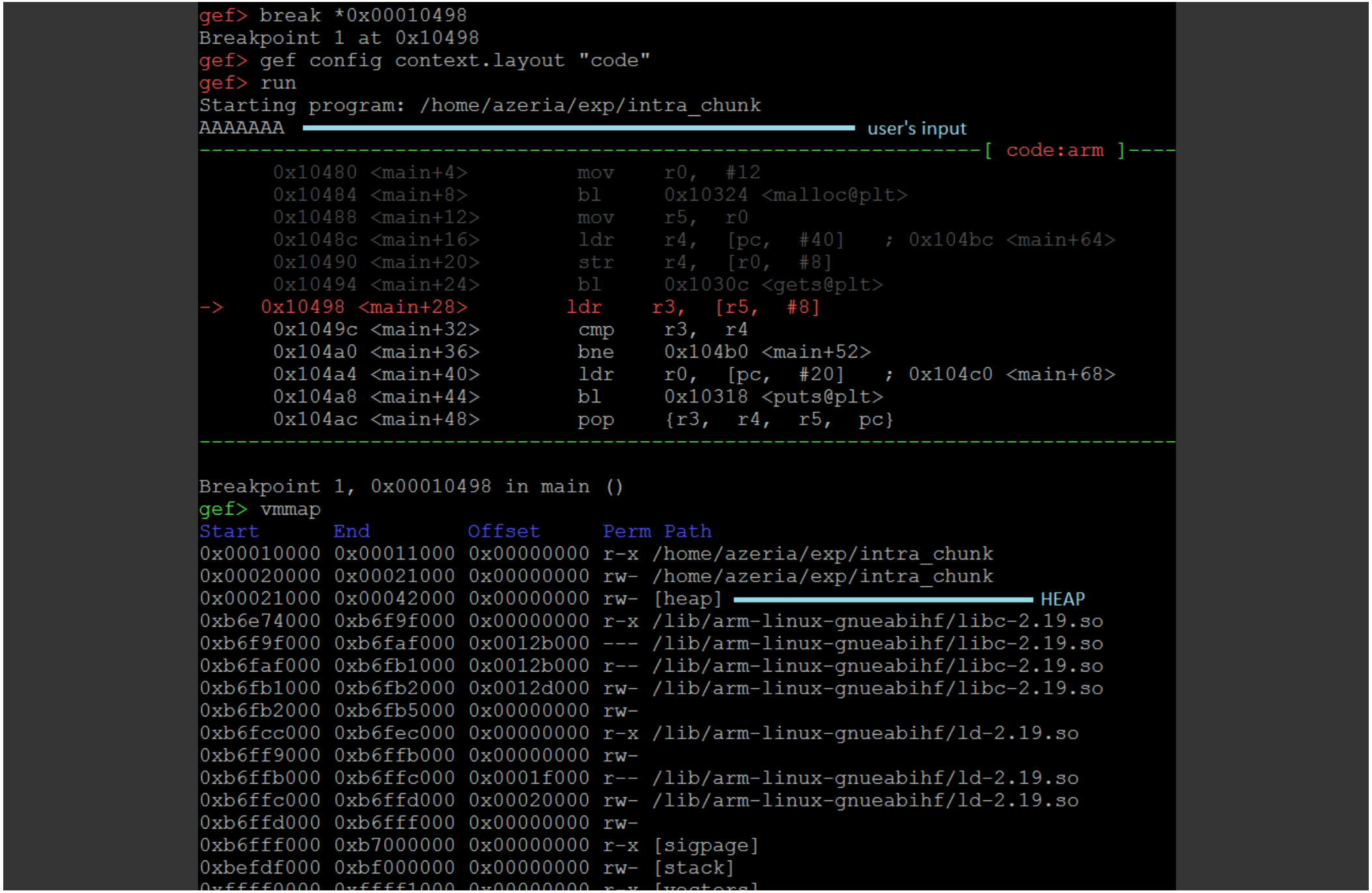
The program above does the following:

1. Defines a data structure (u_data) with two fields
2. Creates an object (in the Heap memory region) of type u_data
3. Assigns a static value to the number's field of the object
4. Prompts user to supply a value for the name's field of the object
5. Prints a string depending on the value of the number's field

So in this case we also suspect that the corruption might happen after the function "gets". We disassemble the target program's main function to get the address for a break-point.

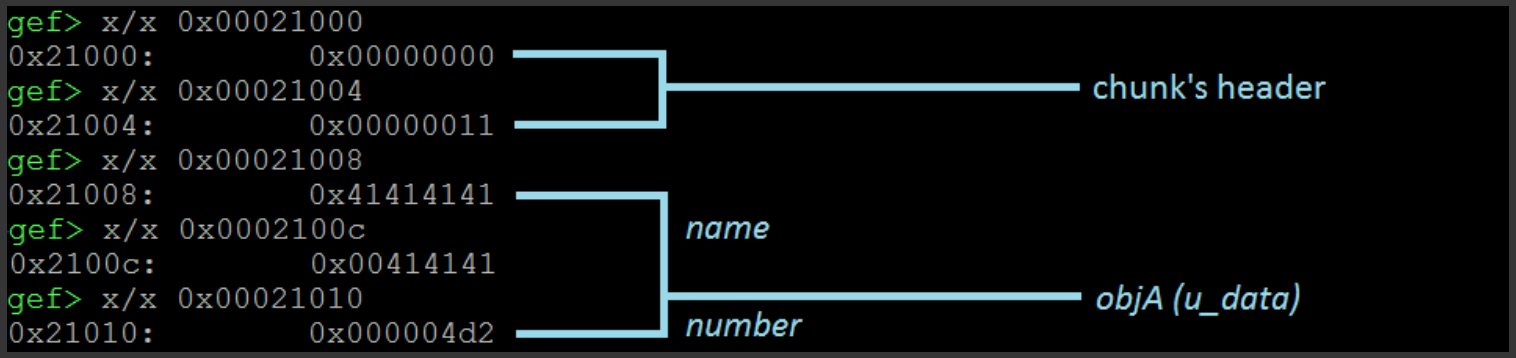
```
azeria@labs:~/exp $ gdb -q intra_chunk
GEF for linux ready, type `gef' to start, `gef config' to configure
50 commands loaded for GDB 7.12 using Python engine 2.7
[*] 3 commands could not be loaded, run `gef missing` to know why.
Reading symbols from intra_chunk...(no debugging symbols found)...done.
gef> disassemble main
Dump of assembler code for function main:
   0x0001047c <+0>:     push    {r3, r4, r5, lr}
   0x00010480 <+4>:     mov     r0, #12
   0x00010484 <+8>:     bl      0x10324 <malloc@plt>
   0x00010488 <+12>:    mov     r5, r0
   0x0001048c <+16>:    ldr     r4, [pc, #40]   ; 0x104bc <main+64>
   0x00010490 <+20>:    str     r4, [r0, #8]
   0x00010494 <+24>:    bl      0x1030c <gets@plt>
   0x00010498 <+28>:    ldr     r3, [r5, #8]
   0x0001049c <+32>:    cmp     r3, r4
   0x000104a0 <+36>:    bne     0x104b0 <main+52>
   0x000104a4 <+40>:    ldr     r0, [pc, #20]   ; 0x104c0 <main+68>
   0x000104a8 <+44>:    bl      0x10318 <puts@plt>
   0x000104ac <+48>:    pop     {r3, r4, r5, pc}
   0x000104b0 <+52>:    ldr     r0, [pc, #12]   ; 0x104c4 <main+72>
   0x000104b4 <+56>:    bl      0x10318 <puts@plt>
   0x000104b8 <+60>:    pop     {r3, r4, r5, pc}
   0x000104bc <+64>:    ldrdeq  r0, [r0], -r2
   0x000104c0 <+68>:    andeq   r0, r1, r12, lsr r5
   0x000104c4 <+72>:    andeq   r0, r1, r12, asr #10
End of assembler dump.
```

In this case we set the break-point at address 0x00010498 – right after the function “gets” is completed. We configure GEF to show us the code only. We then run the program and provide 7 A’s as a user input.

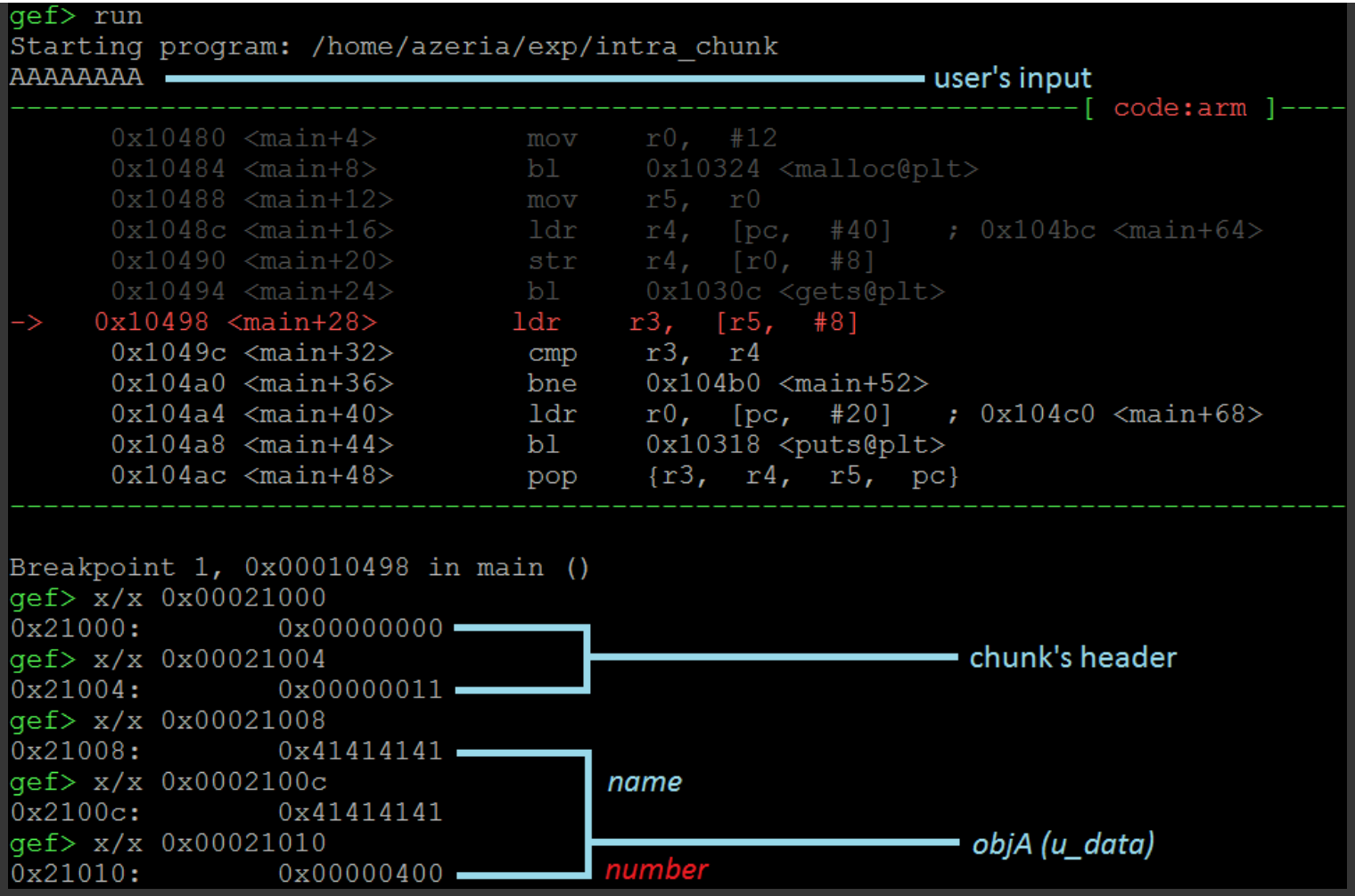


```
0x11110000 0x11111000 0x00000000 r-x [vectors]
```

Once the break-point is hit, we quickly lookup the memory layout of our program to find where our Heap is. We use vmmap command and see that our Heap starts at the address 0x00021000. Given the fact that our object (objA) is the first and the only one created by the program, we start analyzing the Heap right from the beginning.



The picture above shows us a detailed break down of the Heap's chunk associated with our object. The chunk has a header (8 bytes) and the user's data section (12 bytes) storing our object. We see that the name field properly stores the supplied string of 7 A's, terminated by a null-byte. The number field, stores 0x4d2 (1234 in decimal). So far so good. Let's repeat these steps, but in this case enter 8 A's.



While examining the Heap this time we see that the number's field got corrupted (it's now equal to 0x400 instead of 0x4d2). The null-byte terminator overwrote a portion (last byte) of the number's field. This results in an intra-chunk Heap memory corruption. Effects of such a corruption in this case are not devastating, but visible. Logically, the else statement in the code should never be reached as the number's field is intended to be static. However, the memory corruption we just observed makes it possible to reach that part of the code. This can be easily confirmed by the example below.

```
azeria@labs:~/exp $ ./intra_chunk
AAAAAAA
Memory valid
azeria@labs:~/exp $ ./intra_chunk
AAAAAAAA
Memory corrupted
```

Inter-chunk Heap overflow

To illustrate how an inter-chunk Heap overflow looks like in practice we can use the following example, which we now compile **without** optimization flag.

```
/*azeria@labs:~/exp $ gcc inter_chunk.c -o inter_chunk*/
#include "stdlib.h"
#include "stdio.h"

int main ( int argc, char* argv[] )
{
 char *some_string = malloc(8);  //create some_string "object" in Heap
 int *some_number = malloc(4);   //create some_number "object" in Heap

 *some_number = 1234;            //assign some_number a static value
 gets(some_string);              //ask user for input for some_string

 if(*some_number == 1234)        //check if static value (of some_number) is in tact
 {
 puts("Memory valid");
 }
 else                            //proceed here in case the static some_number gets corrupted
```

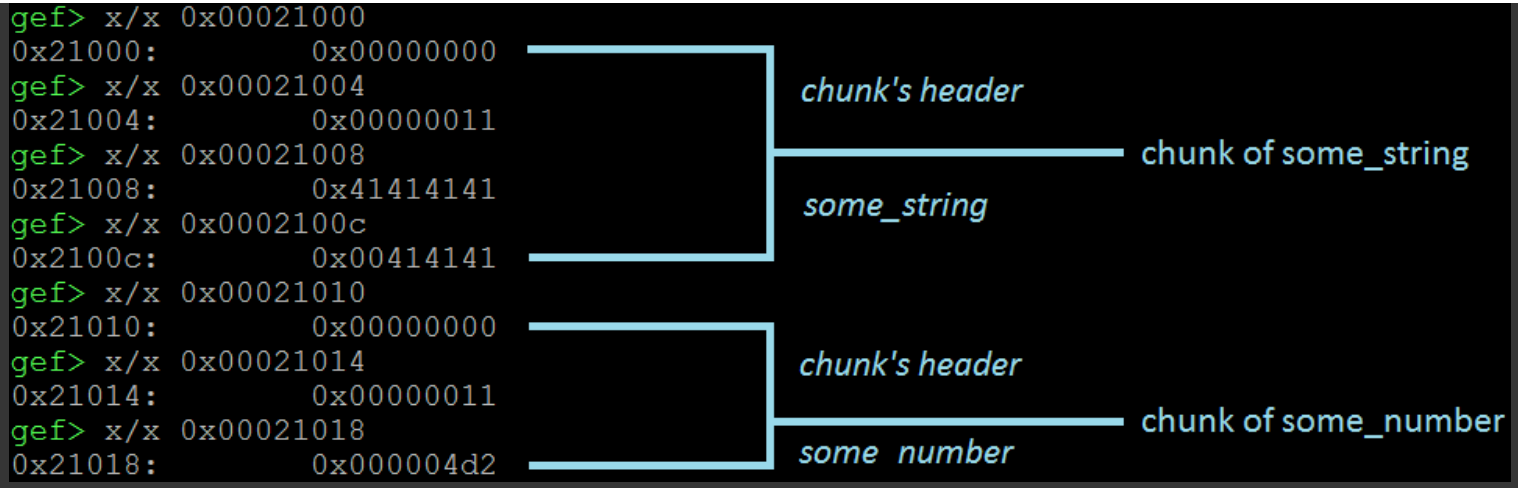
```
  {
  puts("Memory corrupted");
  }
 }
```

The process here is similar to the previous ones: set a break-point after function “gets”, run the program, supply 7 A’s, investigate Heap.

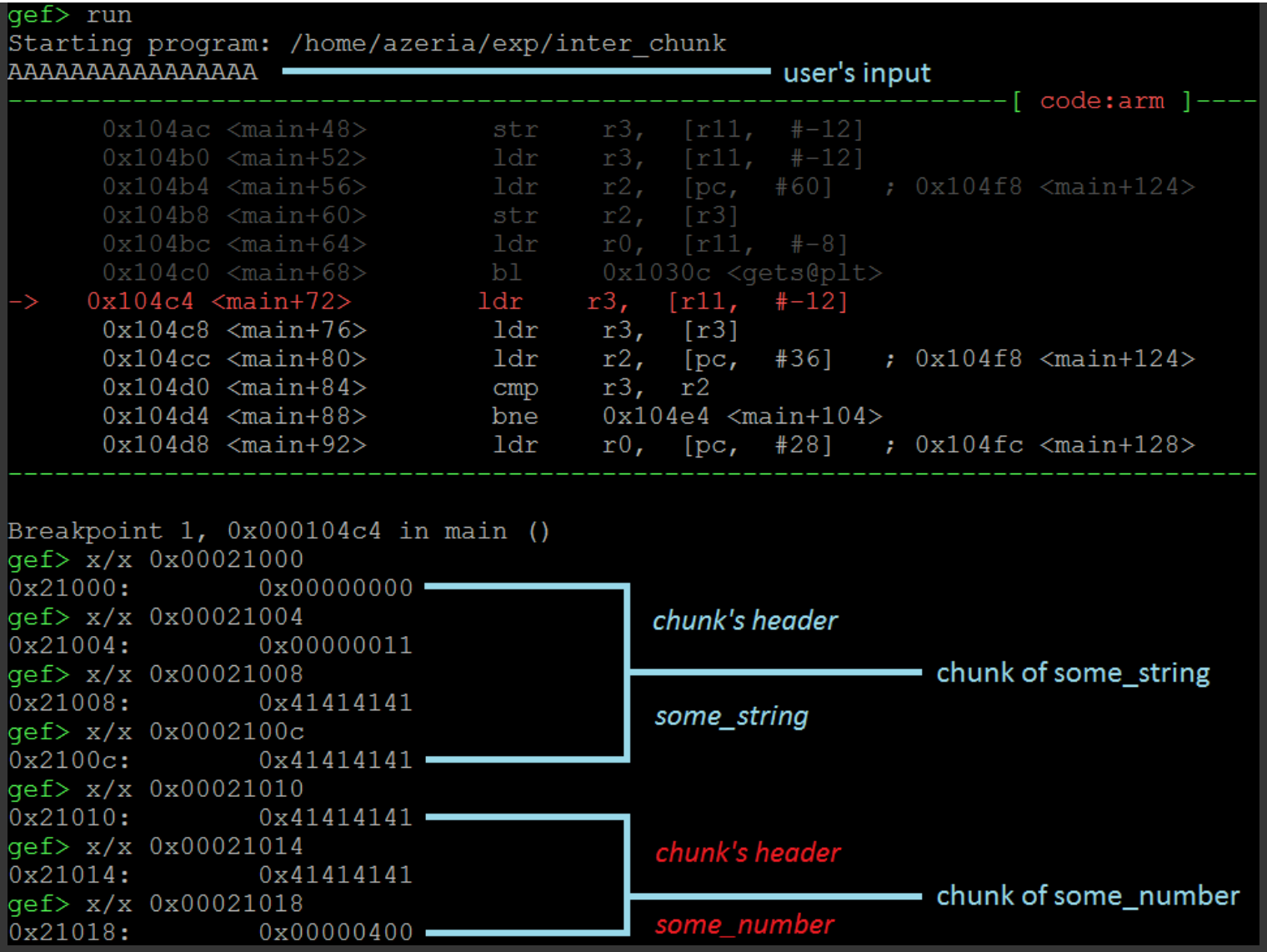
```
azeria@labs:~/exp $ gdb -q inter_chunk
GEF for linux ready, type `gef' to start, `gef config' to configure
50 commands loaded for GDB 7.12 using Python engine 2.7
[*] 3 commands could not be loaded, run `gef missing` to know why.
Reading symbols from inter_chunk...(no debugging symbols found)...done.
gef> disassemble main
Dump of assembler code for function main:
   0x0001047c <+0>:     push    {r11, lr}
   0x00010480 <+4>:     add     r11, sp, #4
   0x00010484 <+8>:     sub     sp, sp, #16
   0x00010488 <+12>:    str     r0, [r11, #-16]
   0x0001048c <+16>:    str     r1, [r11, #-20] ; 0xffffffec
   0x00010490 <+20>:    mov     r0, #8
   0x00010494 <+24>:    bl      0x10324 <malloc@plt>
   0x00010498 <+28>:    mov     r3, r0
   0x0001049c <+32>:    str     r3, [r11, #-8]
   0x000104a0 <+36>:    mov     r0, #4
   0x000104a4 <+40>:    bl      0x10324 <malloc@plt>
   0x000104a8 <+44>:    mov     r3, r0
   0x000104ac <+48>:    str     r3, [r11, #-12]
   0x000104b0 <+52>:    ldr     r3, [r11, #-12]
   0x000104b4 <+56>:    ldr     r2, [pc, #60]   ; 0x104f8 <main+124>
   0x000104b8 <+60>:    str     r2, [r3]
   0x000104bc <+64>:    ldr     r0, [r11, #-8]
   0x000104c0 <+68>:    bl      0x1030c <gets@plt>
   0x000104c4 <+72>:    ldr     r3, [r11, #-12]
   0x000104c8 <+76>:    ldr     r3, [r3]
   0x000104cc <+80>:    ldr     r2, [pc, #36]   ; 0x104f8 <main+124>
   0x000104d0 <+84>:    cmp     r3, r2
   0x000104d4 <+88>:    bne     0x104e4 <main+104>
   0x000104d8 <+92>:    ldr     r0, [pc, #28]   ; 0x104fc <main+128>
   0x000104dc <+96>:    bl      0x10318 <puts@plt>
   0x000104e0 <+100>:   b       0x104ec <main+112>
   0x000104e4 <+104>:   ldr     r0, [pc, #20]   ; 0x10500 <main+132>
   0x000104e8 <+108>:   bl      0x10318 <puts@plt>
   0x000104ec <+112>:   mov     r0, r3
   0x000104f0 <+116>:   sub     sp, r11, #4
   0x000104f4 <+120>:   pop     {r11, pc}
   0x000104f8 <+124>:   ldrdeq  r0, [r0], -r2
   0x000104fc <+128>:   andeq   r0, r1, r8, ror r5
```

```
   0x00010500 <+132>:   andeq   r0, r1, r8, lsl #11
End of assembler dump.
gef> break *0x000104c4
Breakpoint 1 at 0x104c4
gef> gef config context.layout "code"
gef> run
Starting program: /home/azeria/exp/inter_chunk
AAAAAAA                                                    user's input
-----------------------------------------------------------------[ code:arm ]----
      0x104ac <main+48>        str    r3, [r11, #-12]
      0x104b0 <main+52>        ldr    r3, [r11, #-12]
      0x104b4 <main+56>        ldr    r2, [pc, #60]   ; 0x104f8 <main+124>
      0x104b8 <main+60>        str    r2, [r3]
      0x104bc <main+64>        ldr    r0, [r11, #-8]
      0x104c0 <main+68>        bl     0x1030c <gets@plt>
->    0x104c4 <main+72>        ldr    r3, [r11, #-12]
      0x104c8 <main+76>        ldr    r3, [r3]
      0x104cc <main+80>        ldr    r2, [pc, #36]   ; 0x104f8 <main+124>
      0x104d0 <main+84>        cmp    r3, r2
      0x104d4 <main+88>        bne    0x104e4 <main+104>
      0x104d8 <main+92>        ldr    r0, [pc, #28]   ; 0x104fc <main+128>
------------------------------------------------------------------------------
```

Once the break-point is hit, we examine the Heap. In this case, we have two chunks. We see (image below) that their structure is in tact: the some_string is within its boundaries, the some_number is equal to 0x4d2.



Now, let's supply 16 A's and see what happens.



As you might have guessed, providing too much of input causes the overflow resulting into corruption of the adjacent chunk. In this case we see that our user input corrupted the header and the first byte of the some_number's field. Here again, by corrupting the some_number we manage to reach the code section which logically should never be reached.

```
azeria@labs:~/exp $ ./inter_chunk
AAAAAAA
Memory valid
azeria@labs:~/exp $ ./inter_chunk
AAAAAAAAAAAAAAAA
Memory corrupted
```

SUMMARY

In this part of the tutorial we got familiar with the process memory layout and the basics of Stack and Heap related memory corruptions. In the next part of this tutorial series we will cover other memory corruptions: Dangling pointer and Format String. Once we cover the most common types of memory corruptions, we will be ready for learning how to write working exploits.

ARM Exploit Development

Writing ARM Shellcode

TCP Bind Shell (ARM 32-bit)

TCP Reverse Shell (ARM 32-bit)

Process Memory and Memory Corruption

Stack Overflow Challenges

Process Continuation Shellcode

Introduction to Glibc Heap (malloc)

Introduction to Glibc Heap (free, bins)

Part 1: Heap Exploit Development

Part 2 Heap Overflows and iOS Kernel

Twitter: @Fox0x01 and @azeria_labs

ARM Assembly Cheat Sheet

POSTER DIGITAL