CMCDragonkai / **memory_layout.md**

Last active 2 hours ago • Report abuse

Code Revisions 8 Stars 404 Forks 115

Linux: Understanding the Memory Layout of Linux Executables

`memory_layout.md`

# Understanding the Memory Layout of Linux Executables

Required tools for playing around with memory:

- `hexdump`
- `objdump`
- `readelf`
- `xxd`
- `gcore`
- `strace`
- `diff`
- `cat`

We're going to go through this: https://sploitfun.wordpress.com/2015/02/10/understanding-glibc-malloc/ and http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

There are actually many C memory allocators. And different memory allocators will layout memory in different ways. Currently glibc's memory allocator is `ptmalloc2`. It was forked from dlmalloc. After fork, threading support was added, and was released in 2006. After being integrated, code changes were made directly to glibc's malloc source code itself. So there are many changes with glibc's malloc that is different from the original `ptmalloc2`.

The malloc in glibc, internally invokes either `brk` or `mmap` syscalls to acquire memory from the OS. The `brk` syscall is generally used to increase the size of the heap, while `mmap` will be used to load shared libraries, create new regions for threads, and many other things. It actually switches to using `mmap` instead of `brk` when the amount of memory requested is larger than the `MMAP_THRESHOLD`. We view which calls are being made by using `strace`.

In the old days using `dlmalloc`, when 2 threads call malloc at the same time, only one thread can enter the critical section, the freelist data structure of memory chunks is shared among all available threads. Hence memory allocation is a global locking operation.

However in ptmalloc2, when 2 threads call malloc at the same time, memory is allocated immediately, since each thread maintains a separate heap, and their own freelist chunk data structure

The act of maintain separate heap and freelists for each thread is called "per-thread arena".

In the last session, we identified that a program memory layout is generally in:

```
User Stack
    |
    v
Memory Mapped Region for Shared Libraries or Anything Else
    ^
    |
Heap
Uninitialised Data (.bss)
Initialised Data (.data)
Program Text (.text)
0
```

For the purpose of understanding, most of the tools that investigate memory put the low address at the top and the high address at the bottom.

Therefore, it's easier to think of it like this:

```
0
Program Text (.text)
Initialised Data (.data)
Uninitialised Data (.bss)
Heap
    |
    v
Memory Mapped Region for Shared Libraries or Anything Else
    ^
```

```
     |
User Stack
```

The problem is, we didn't have a proper grasp over what exactly is happening. And the above diagram is too simple to fully understand.

Let's write some C programs and investigate their memory structure.

> Note that neither direct compilation or assembly actually produce an executable. That is done by the linker, which takes the various object code files produced by compilation/assembly, resolves all the names they contain and produces the final executable binary. http://stackoverflow.com/a/845365/582917

This is our first program (compile it using `gcc -pthread memory_layout.c -o memory_layout`:

```c
#include <stdio.h> // standard io
#include <stdlib.h> // C standard library
#include <pthread.h> // threading
#include <unistd.h> // unix standard library
#include <sys/types.h> // system types for linux

// getchar basically is like "read"
// it prompts the user for input
// in this case, the input is thrown away
// which makes similar to a "pause" continuation primitive
// but a pause that is resolved through user input, which we promptly throw away!
void * thread_func (void * arg) {

    printf("Before malloc in thread 1\n");
    getchar();
    char * addr = (char *) malloc(1000);
    printf("After malloc and before free in thread 1\n");
    getchar();
    free(addr);
    printf("After free in thread 1\n");
    getchar();

}

int main () {

    char * addr;
    printf("Welcome to per thread arena example::%d\n", getpid());
    printf("Before malloc in the main thread\n");
    getchar();
    addr = (char *) malloc(1000);
    printf("After malloc and before free in main thread\n");
    getchar();
    free(addr);
    printf("After free in main thread\n");
    getchar();

    // pointer to the thread 1
    pthread_t thread_1;
    // pthread_* functions return 0 upon succeeding, and other numbers upon failing
    int pthread_status;

    pthread_status = pthread_create(&thread_1, NULL, thread_func, NULL);

    if (pthread_status != 0) {
        printf("Thread creation error\n");
        return -1;
    }

    // returned status code from thread_1
    void * thread_1_status;

    pthread_status = pthread_join(thread_1, &thread_1_status);

    if (pthread_status != 0) {
        printf("Thread join error\n");
        return -1;
    }

    return 0;

}
```

The usage of `getchar` above is to basically pause the computation waiting for user input. This allows us to step through the program, when examining its memory layout.

The usage of `pthread` is for creating POSIX threads, which are real kernel threads being scheduled on the Linux OS. The thing si, the usage of threads is interesting for examining how a process memory layout is utilised for many threads. It turns out that each thread requires its own heap and stack.

The `pthread` functions kind of weird, because they return a 0 based status code upon success. This is the success of a `pthread` operation, which does involve a side effect on the underlying operating system.

As we can see above, there are many uses of the reference error pattern, that is, instead of returning multiple values (through tuples), we use reference containers to store extra metadata or just data itself.

Now, we can run the program `./memory_layout` (try using `Ctrl + Z` to suspend the program):

```
$ ./memory_layout
Welcome to per thread arena example::1255
Before malloc in the main thread
```

At this point, the program is paused, we can now inspect the memory contents by looking at `/proc/1255/maps`. This is a kernel supplied virtual file that shows the exact memory layout of the program. It actually summarises each memory section, so it's useful for understanding how memory is layed out without necessarily being able to view a particular byte address.

```
$ cat /proc/1255/maps # you can also use `watch -d cat /proc/1255/maps` to get
updates
00400000-00401000 r-xp 00000000 fc:00 1457150
/home/vagrant/c_tests/memory_layout
00600000-00601000 r--p 00000000 fc:00 1457150
/home/vagrant/c_tests/memory_layout
00601000-00602000 rw-p 00001000 fc:00 1457150
/home/vagrant/c_tests/memory_layout
7f849c31b000-7f849c4d6000 r-xp 00000000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f849c4d6000-7f849c6d6000 ---p 001bb000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f849c6d6000-7f849c6da000 r--p 001bb000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f849c6da000-7f849c6dc000 rw-p 001bf000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f849c6dc000-7f849c6e1000 rw-p 00000000 00:00 0
7f849c6e1000-7f849c6fa000 r-xp 00000000 fc:00 1579084
/lib/x86_64-linux-gnu/libpthread-2.19.so
7f849c6fa000-7f849c8f9000 ---p 00019000 fc:00 1579084
/lib/x86_64-linux-gnu/libpthread-2.19.so
7f849c8f9000-7f849c8fa000 r--p 00018000 fc:00 1579084
/lib/x86_64-linux-gnu/libpthread-2.19.so
7f849c8fa000-7f849c8fb000 rw-p 00019000 fc:00 1579084
/lib/x86_64-linux-gnu/libpthread-2.19.so
7f849c8fb000-7f849c8ff000 rw-p 00000000 00:00 0
7f849c8ff000-7f849c922000 r-xp 00000000 fc:00 1579072
/lib/x86_64-linux-gnu/ld-2.19.so
```

```
7f849cb10000-7f849cb13000 rw-p 00000000 00:00 0
7f849cb1d000-7f849cb21000 rw-p 00000000 00:00 0
7f849cb21000-7f849cb22000 r--p 00022000 fc:00 1579072
/lib/x86_64-linux-gnu/ld-2.19.so
7f849cb22000-7f849cb23000 rw-p 00023000 fc:00 1579072
/lib/x86_64-linux-gnu/ld-2.19.so
7f849cb23000-7f849cb24000 rw-p 00000000 00:00 0
7fffb5d61000-7fffb5d82000 rw-p 00000000 00:00 0                          [stack]
7fffb5dfe000-7fffb5e00000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
```

Each row in /proc/$PID/maps describes a region of contiguous virtual memory in a process. Each row has the following fields:

- address - starting and ending address in the region's process address space
- perms - describes how pages can be accessed `rwxp` or `rwxs`, where `s` meaning private or shared pages, if a process attempts to access memory not allowed by the permissions, a segmentation fault occurs
- offset - if the region was mapped by a file using `mmap`, this is the offset in the file where the mapping begins
- dev - if the region was mapped from a file, this is major and minor device number in hex where the file lives, the major number points to a device driver, and the minor number is interpreted by the device driver, or the minor number is the specific device for a device driver, like multiple floppy drives
- inode - if the region was mapped from a file, this is the file number
- pathname - if the region was mapped from a file, this is the name of the file, there are special regions with names like [heap], [stack] and [vdso], [vdso] stands for virtual dynamic shared object, its used by system calls to switch to kernel mode

Some regions don't have any file path or special names in the pathname field, these are anonymous regions. Anonymous regions are created by mmap, but are not attached to any file, they are used for miscellaneous things like shared memory, buffers not on the heap, and the pthread library uses anonymous mapped regions as stacks for new threads.

There's no 100% guarantee that contiguous virtual memory means contiguous physical memory. To do this, you would have to use an OS with no virtual memory system. But it's a good chance that contiguous virtual memory does equal contiguous physical memory, at least there's no pointer chasing. Still at the hardware level, there's a special device for virtual to physical memory translation. So it's still very fast.

It is quite important to use the `bc` tool because we need to convert between hex and decimal quite often here. We can use it like: `bc <<< 'obase=10; ibase=16; 4010000 - 4000000'`, which essentially does a `4010000 - 4000000` subtraction using hex digits, and then convers the result to base 10 decimal.

Just a side note about the major minor numbers. You can use `ls -l /dev | grep 252` or `lsblk | grep 252` to look up devices corresponding to a `major:minor` number. Where `0d252 ~ 0xfc`.

This lists all the major and minor number allocations for Linux Device Drivers: http://www.lanana.org/docs/device-list/devices-2.6+.txt

It also shows that anything between 240 - 254 is for local/experimental usecase. Also 232 - 239 is unassigned. And 255 is reserved. We can identify that the device in question right now is a device mapper device. So it is using the range reserved for local/experimental use. Major and minor numbers only go up to 255 because it's the largest decimal natural in a single byte. A single byte is: `0b11111111` or `0xFF`. A single hex digit is a nibble. 2 hex digits is a byte.

The first thing to realise, is that the memory addresses starts from low to high, but every time you run this program, many of the regions will have different addresses. So that means for some regions, addresses aren't statically allocated. This is actually due to a security feature, by randomising the address space for certain regions, it makes it more difficult for attackers to acquire a particular piece of memory they are interested in. However there are regions that are always fixed, because you need them to be fixed so you know how to load the program. We can see that program data and executable memory is always fixed along with `vsyscall`. It is actually possible to create what people call a "PIE" (position independent executable), which actually even makes the program data and executable memory randomised as well, however this is not enabled by default, and it also prevents the program from being compiled statically, forcing it to be linked (https://sourceware.org/ml/binutils/2012-02/msg00249.html). Also "PIE" executables incur some performance problems (different kinds of problems on 32 bit vs 64 bit computers). The randomisation of the address for some of the regions is called "PIC" (position independent code), and has been enabled by default on Linux for quite some time. For more information, see: http://blog.fpmurphy.com/2008/06/position-independent-executables.html and http://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries

It is possible to compile the above program using `gcc -fPIE -pie ./hello.c -o hello`, which produces a "PIE" executable. There's some talk on nixpkgs to make compilation to "PIE" by default for 64 bit binaries, but 32 bit binaries remain unPIEd due to serious performance problems. See: NixOS/nixpkgs#7220

BTW: Wouldn't it be great if we had a tool that examined `/proc/$PID/maps` and gave exact human readable byte sizes instead?

So let's go into detail for each region. Remember, this is still at the beginning of the program, where no `malloc` has occurred, so there is no `[heap]` region.

```
     0 - 400000 - 4194304 B - 4096 KiB ~ 4 MiB - NOT ALLOCATED
400000 - 401000 - 4096 B    - 4 KiB
600000 - 601000 - 4096 B    - 4 KiB
601000 - 602000 - 4096 B    - 4 KiB
```

This is our initial range of memory. I added an extra component which starts at `0` address and reaches `40 00 00` address. Addresses appear to be left inclusive, right exclusive. But remember that addresses start at 0. Therefore it is legitimate to use `bc <<< 'obase=10;ibase=16 400000 - 0'` to acquire the actual amount of bytes in that range without adding or subtracing 1. In this case, the first unallocated region is roughly 4 MiB. When I say unallocated, I mean it is not represented in the the `/proc/$PID/maps`. This could mean either of 2 things, either the file doesn't show all the allocated memory, or it doesn't consider such memory to be worth showing, or there is really no memory allocated there.

We can find out whether there really is memory there, by creating a pointer to the memory address somewhere between `0` and `400000`, and try dereferencing it. This can be done by casting an integer into a pointer. I've tried it before, and it results in a segfault, this means there really is no memory allocated between `0-400000`

```c
#include <stdio.h>

int main () {

    // 0x0 is hex literal that defaults to signed integer
    // here we are casting it to a void pointer
    // and then assigning it to a value declared to be a void pointer
    // this is the correct way to create an arbitrary pointer in C
    void * addr = (void *) 0x0;

    // in order to print out what exists at that pointer, we must dereference the pc
    // but C doesn't know how to handle a value of void type
    // which means, we recast the void pointer to a char pointer
    // a char is some arbitrary byte, so hopefully it's a printable ASCII value
    // actually, we don't need to hope, because we have made printf specifically pri
    printf("0x%x\n", ((char *) addr)[0]); // prints 0x0
    printf("0x%x\n", ((char *) addr)[1]); // prints 0x1
    printf("0x%x\n", ((char *) addr)[2]); // prints 0x2
}
```

Running the above gives us a simple `segmentation fault`. Thus proving that `/proc/$PID/maps` is giving us the truth, there really is nothing between `0-400000`.

The question becomes, why is there this roughly 4 MiB gap? Why not start allocating memory from 0? Well this was just an arbitrary choice by the malloc and linker implementors. They just decided that that on 64 bit ELF executables, the entry point of a non-PIE executable should be `0x400000`, whereas for 32 bit ELF executables, the entry point is `0x08048000`. An interesting fact is that if you produce a position independent executable, the starting address instead changes to `0x0`.

See:

- http://stackoverflow.com/questions/7187981/whats-the-memory-before-0x08048000-used-for-in-32-bit-machine
- http://stackoverflow.com/questions/12488010/why-the-entry-point-address-in-my-executable-is-0x8048330-0x330-being-offset-of
- http://stackoverflow.com/questions/14314021/why-linux-gnu-linker-chose-address-0x400000

> The entry address is set by the link editor, at the time when it creates the executable. The loader maps the program file at the address(es) specified by the ELF headers before transferring control to the entry address.

> The load address is arbitrary, but was standardized back with SYSV for x86. It's different for every architecture. What goes above and below is also arbitrary, and is often taken up by linked in libraries and mmap() regions.

What it basically means is that the program executable is loaded into memory before it can start doing things. The entry point of the executable can be acquired by `readelf`. But here's another question, why is the entry point given by `readelf`, not at `0x400000`. It turns out that, that entry point is consider the actual point where the OS should start executing, whereas the position between `0x400000` and entry point is used for the EHDR and PHDR, meaning ELF headers and Program Headers. We'll look into this in detail later.

```
$ readelf --file-header ./memory_layout | grep 'Entry point address'
  Entry point address:               0x400720
```

Next we have the:

```
400000 - 401000 - 4096 B    - 4 KiB
600000 - 601000 - 4096 B    - 4 KiB
601000 - 602000 - 4096 B    - 4 KiB
```

As you can see, we have 3 sections of memory, each of them 4 KiB, and allocated from `/home/vagrant/c_tests/memory_layout`.

What are these sections?

The first segment: "Text Segment".

The second segment: "Data Segment".

The third segment: "BSS Segment".

Text segment stores the binary image of the process. The data segment stores static variables initialised by the programmer, for example `static char * foo = "bar";`. The BSS segment stores uninitialised static variables, which are filled with zeros, for example `static char * username;`.

Our program is so simple right now, that each seemingly fits perfectly into 4 KiB. How can it be so perfect!?

Well, the page size of the Linux OS and many other OSes, is set to 4 KiB by default. This means the minimum addressable memory segment is 4 KiB. See: https://en.wikipedia.org/wiki/Page_%28computer_memory%29

> A page, memory page, or virtual page is a fixed-length contiguous block of virtual memory, described by a single entry in the page table. It is the smallest unit of data for memory management in a virtual memory operating system.

Running `getconf PAGESIZE` shows 4096 bytes.

Therefore, this means each segment is probably far smaller than `4096` bytes, but it gets padded up to 4096 bytes.

As we shown before, it's possible to create an arbitrary pointer, and print out the value stored that byte. We can now do this for the segments shown above.

But hey, we can do better. Rather than just hacking the individual bytes. We can recognise that this data is actually organised as structs.

What kind of structs? We can look at the `readelf` source code to uncover the relevant structs. These structs don't appear to be part of the standard C library, so we cannot just include things to get this working. But the code is simple, so we can just copy and paste. See: [http://rpm5.org/docs/api/readelf_8h-source.html](http://rpm5.org/docs/api/readelf_8h-source.html)

Check this out:

```c
// compile with gcc -std=c99 -o elfheaders ./elfheaders.c
#include <stdio.h>
#include <stdint.h>

// from: http://rpm5.org/docs/api/readelf_8h-source.html
// here we're only concerned about 64 bit executables, the 32 bit executables have d

typedef uint64_t Elf64_Addr;
typedef uint64_t Elf64_Off;
typedef uint64_t Elf64_Xword;
typedef uint32_t Elf64_Word;
typedef uint16_t Elf64_Half;
typedef uint8_t  Elf64_Char;

#define EI_NIDENT 16

// this struct is exactly 64 bytes
// this means it goes from 0x400000 - 0x400040
typedef struct {
    Elf64_Char  e_ident[EI_NIDENT]; // 16 B
    Elf64_Half  e_type;             // 2 B
    Elf64_Half  e_machine;          // 2 B
    Elf64_Word  e_version;          // 4 B
    Elf64_Addr  e_entry;            // 8 B
    Elf64_Off   e_phoff;            // 8 B
    Elf64_Off   e_shoff;            // 8 B
    Elf64_Word  e_flags;            // 4 B
    Elf64_Half  e_ehsize;           // 2 B
    Elf64_Half  e_phentsize;        // 2 B
    Elf64_Half  e_phnum;            // 2 B
    Elf64_Half  e_shentsize;        // 2 B
    Elf64_Half  e_shnum;            // 2 B
    Elf64_Half  e_shstrndx;         // 2 B
} Elf64_Ehdr;

// this struct is exactly 56 bytes
// this means it goes from 0x400040 - 0x400078
typedef struct {
     Elf64_Word  p_type;   // 4 B
     Elf64_Word  p_flags;  // 4 B
     Elf64_Off   p_offset; // 8 B
     Elf64_Addr  p_vaddr;  // 8 B
     Elf64_Addr  p_paddr;  // 8 B
     Elf64_Xword p_filesz; // 8 B
     Elf64_Xword p_memsz;  // 8 B
     Elf64_Xword p_align;  // 8 B
} Elf64_Phdr;

int main(int argc, char *argv[]){

    // from examination of objdump and /proc/ID/maps, we can see that this is the fi
    // earliest in the virtual memory address space, for a 64 bit ELF executable
    // %lx is required for 64 bit hex, while %x is just for 32 bit hex

    Elf64_Ehdr * ehdr_addr = (Elf64_Ehdr *) 0x400000;

    printf("Magic:                      0x");
    for (unsigned int i = 0; i < EI_NIDENT; ++i) {
        printf("%x", ehdr_addr->e_ident[i]);
    }
    printf("\n");
    printf("Type:                       0x%x\n", ehdr_addr->e_type);
    printf("Machine:                    0x%x\n", ehdr_addr->e_machine);
    printf("Version:                    0x%x\n", ehdr_addr->e_version);
    printf("Entry:                      %p\n", (void *) ehdr_addr->e_entry);
    printf("Phdr Offset:                0x%lx\n", ehdr_addr->e_phoff);
    printf("Section Offset:             0x%lx\n", ehdr_addr->e_shoff);
    printf("Flags:                      0x%x\n", ehdr_addr->e_flags);
    printf("ELF Header Size:            0x%x\n", ehdr_addr->e_ehsize);
    printf("Phdr Header Size:           0x%x\n", ehdr_addr->e_phentsize);
    printf("Phdr Entry Count:           0x%x\n", ehdr_addr->e_phnum);
    printf("Section Header Size:        0x%x\n", ehdr_addr->e_shentsize);
    printf("Section Header Count:       0x%x\n", ehdr_addr->e_shnum);
    printf("Section Header Table Index: 0x%x\n", ehdr_addr->e_shstrndx);

    Elf64_Phdr * phdr_addr = (Elf64_Phdr *) 0x400040;

    printf("Type:                     %u\n", phdr_addr->p_type); // 6 - PT_PHDR - se
    printf("Flags:                    %u\n", phdr_addr->p_flags); // 5 - PF_R + PF_X
    printf("Offset:                   0x%lx\n", phdr_addr->p_offset); // 0x40 - byte
    printf("Program Virtual Address:  %p\n", (void *) phdr_addr->p_vaddr); // 0x4000
    printf("Program Physical Address: %p\n", (void *) phdr_addr->p_paddr); // 0x4000
    printf("Loaded file size:         0x%lx\n", phdr_addr->p_filesz); // 504 - bytes
    printf("Loaded mem size:          0x%lx\n", phdr_addr->p_memsz); // 504 - bytes
    printf("Alignment:                %lu\n", phdr_addr->p_align); // 8 - alignment

    return 0;

}
```

Running the above gives:

```
$ ./elfheaders
Magic:                      0x7f454c46211000000000
Type:                       0x2
Machine:                    0x3e
Version:                    0x1
Entry:                      0x400490
Phdr Offset:                0x40
Section Offset:             0x1178
Flags:                      0x0
ELF Header Size:            0x40
Phdr Header Size:           0x38
Phdr Entry Count:           0x9
Section Header Size:        0x40
Section Header Count:       0x1e
Section Header Table Index: 0x1b
Type:                     6
Flags:                    5
Offset:                   0x40
Program Virtual Address:  0x400040
Program Physical Address: 0x400040
Loaded file size:         0x1f8
Loaded mem size:          0x1f8
Alignment:                8
```

Compare the above output with:

```
$ readelf --file-header ./elfheaders
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400490
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4472 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

We've basically just written our own little `readelf` program.

So it's starting to make sense as to what's actually located at the start of `0x400000 - 0x401000`, it's all the ELF executable headers that tell the OS how to use this program, and all the other interesting metadata. Specifically this is about what's located between the program's actual entry point (for `./elfheader`: `0x400490` and for `./memory_layout`: `0x400720`) and the actual start of memory at `0x400000`. There are more program headers to study, but this is enough for now. See: http://www.ouah.org/RevEng/x430.htm

But where does the OS actually get this data from? It has to acquire this data, before it put into memory. Well it turns out the answer is very simple. It's just the file itself.

Let's use `hexdump` to view the actual binary contents of the file, and also later use `objdump` to disassemble it to assembly to make some sense of the machine code.

Obviously the starting memory address, wouldn't be starting file address. So instead of `0x400000`, files should most likely start at `0x0`.

```
$ hexdump -C -s 0x0 ./memory_layout # the -s option is just for offset, it's
actually redundant here, but will be useful later
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
00000010  02 00 3e 00 01 00 00 00  20 07 40 00 00 00 00 00  |..>..... .@.....|
00000020  40 00 00 00 00 00 00 00  a8 11 00 00 00 00 00 00  |@...............|
00000030  00 00 00 00 40 00 38 00  09 00 40 00 1e 00 1b 00  |....@.8...@.....|
00000040  06 00 00 00 05 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000050  40 00 40 00 00 00 00 00  40 00 40 00 00 00 00 00  |@.@.....@.@.....|
00000060  f8 01 00 00 00 00 00 00  f8 01 00 00 00 00 00 00  |................|
00000070  08 00 00 00 00 00 00 00  03 00 00 00 04 00 00 00  |................|
00000080  38 02 00 00 00 00 00 00  38 02 40 00 00 00 00 00  |8.......8.@.....|
00000090  38 02 40 00 00 00 00 00  1c 00 00 00 00 00 00 00  |8.@.............|
000000a0  1c 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
000000b0  01 00 00 00 05 00 00 00  00 00 00 00 00 00 00 00  |................|
000000c0  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
000000d0  34 0c 00 00 00 00 00 00  34 0c 00 00 00 00 00 00  |4.......4.......|
000000e0  00 00 20 00 00 00 00 00  01 00 00 00 06 00 00 00  |.. .............|
000000f0  00 0e 00 00 00 00 00 00  00 0e 60 00 00 00 00 00  |..........`.....|
00000100  00 0e 60 00 00 00 00 00  78 02 00 00 00 00 00 00  |..`.....x.......|
00000110  80 02 00 00 00 00 00 00  00 00 20 00 00 00 00 00  |.......... .....|
00000120  02 00 00 00 06 00 00 00  18 0e 00 00 00 00 00 00  |................|
00000130  18 0e 60 00 00 00 00 00  18 0e 60 00 00 00 00 00  |..`.......`.....|
00000140  e0 01 00 00 00 00 00 00  e0 01 00 00 00 00 00 00  |................|
00000150  08 00 00 00 00 00 00 00  04 00 00 00 04 00 00 00  |................|
00000160  54 02 00 00 00 00 00 00  54 02 40 00 00 00 00 00  |T.......T.@.....|
00000170  54 02 40 00 00 00 00 00  44 00 00 00 00 00 00 00  |T.@.....D.......|
00000180  44 00 00 00 00 00 00 00  04 00 00 00 00 00 00 00  |D...............|
00000190  50 e5 74 64 04 00 00 00  e0 0a 00 00 00 00 00 00  |P.td............|
000001a0  e0 0a 40 00 00 00 00 00  e0 0a 40 00 00 00 00 00  |..@.......@.....|
000001b0  3c 00 00 00 00 00 00 00  3c 00 00 00 00 00 00 00  |<.......<.......|
000001c0  04 00 00 00 00 00 00 00  51 e5 74 64 06 00 00 00  |........Q.td....|
000001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
...
```

It's quite a long piece of text so piping it into `less` is a good idea. Note that `*` means "same as above line".

Firstly checkout the first 16 bytes: `7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00`.

Note how this is the same as the magic bytes shown by `readelf`:

```
$ readelf -h ./memory_layout | grep Magic
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
```

So it turns out that one can say that `0x400000` for a non-PIE 64 bit ELF executable compiled by `gcc` on Linux is the exact the same starting point as the `0x0` for the actual executable file itself.

The file headers are indeed being loaded into memory. But can we tell if the entire file is loaded into memory or not? Let's first check the file size.

```
$ stat memory_layout | grep Size
  Size: 8932            Blocks: 24         IO Block: 4096   regular file
```

Shows that the file is 8932 bytes, which is roughly 8.7 KiB.

Our memory layout showed that at most 4 KiB + 4 KiB + 4 KiB was mapped from the `memory_layout` executable file.

There's plenty of space, and certainly enough to fit the entire contents of the file.

But we can prove this by iterating over the entire memory contents, and check the relevant offsets in memory to see if they match the contents on file.

To do this, we need to investigate `/proc/$PID/mem`. However it's not a normal file that you can cat from, but you must do some interesting syscalls to get some output from it. There's no standard unix tool to read from it, instead we would need to write a C program read it. There's an example program here: http://unix.stackexchange.com/a/251769/56970

Fortunately, there's a thing called `gdb`, and we can use `gcore` to dump a process's memory contents onto disk. It require super user privileges, because we are essentially accessing a process's memory, and memory is usually meant to be isolated!

```
$ sudo gcore 1255

Program received signal SIGTTIN, Stopped (tty input).
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
0x00007f849c407350 in read () from /lib/x86_64-linux-gnu/libc.so.6
Saved corefile core.1255

[1]+  Stopped                 ./memory_layout
```

This produces a file called `core.1255`. This file is the memory dump, so to view it, we must use `hexedit`.

```
$ hexdump -C ./core.1255 | less
```

Now that we have the entire memory contents. Lets try and compare it with the executable file itself. Before we can do that, we must turn our binary file into printable ASCII. Essentially ASCII armoring the binary programs. The `xxd` is better for this purpose because `hexdump` gives us `|` characters which can give confusing output when we use `diff`.

```
$ xxd ./core.1255 > ./memory.hex
$ xxd ./memory_layout > ./file.hex
```

Immediately we can see that 2 sizes are not the same. The `./memory.hex` ~ 1.1 MiB is far larger than the `./file.hex` ~ 37 KiB. This is because the memory dump also contains all the shared libraries and the anonymously mapped regions. But we are not expecting them to be the same, just whether the entire file itself exists in memory.

Both files can now be compared using `diff`.

```
$ diff --side-by-side ./file.hex ./memory.hex # try piping into less
0000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF.......   0000000: 7f45
4c46 0201 0100 0000 0000 0000 0000  .ELF.......
0000010: 0200 3e00 0100 0000 2007 4000 0000 0000  ..>..... .@ | 0000010: 0400
3e00 0100 0000 0000 0000 0000 0000  ..>........
0000020: 4000 0000 0000 0000 a811 0000 0000 0000  @.......... | 0000020: 4000
0000 0000 0000 1022 0400 0000 0000  @........".
0000030: 0000 0000 4000 3800 0900 4000 1e00 1b00  ....@.8...@ | 0000030: 0000
0000 4000 3800 1500 4000 1700 1600  ....@.8...@
0000040: 0600 0000 0500 0000 4000 0000 0000 0000  ........@.. | 0000040: 0400
0000 0400 0000 d804 0000 0000 0000  ...........
0000050: 4000 4000 0000 0000 4000 4000 0000 0000  @.@.....@.@ | 0000050: 0000
0000 0000 0000 0000 0000 0000 0000  ...........
0000060: f801 0000 0000 0000 f801 0000 0000 0000  ........... | 0000060: 200d
0000 0000 0000 0000 0000 0000 0000   ..........
0000070: 0800 0000 0000 0000 0300 0000 0400 0000  ........... | 0000070: 0100
0000 0000 0000 0100 0000 0500 0000  ...........
0000080: 3802 0000 0000 0000 3802 4000 0000 0000  8.......8.@ | 0000080: f811
0000 0000 0000 0000 4000 0000 0000  ..........@
0000090: 3802 4000 0000 0000 1c00 0000 0000 0000  8.@........ | 0000090: 0000
0000 0000 0000 0000 0000 0000 0000  ...........
00000a0: 1c00 0000 0000 0000 0100 0000 0000 0000  ........... | 00000a0: 0010
0000 0000 0000 0100 0000 0000 0000  ...........
00000b0: 0100 0000 0500 0000 0000 0000 0000 0000  ........... | 00000b0: 0100
0000 0400 0000 f811 0000 0000 0000  ...........
00000c0: 0000 4000 0000 0000 0000 4000 0000 0000  ..@.......@ | 00000c0: 0000
6000 0000 0000 0000 0000 0000 0000  ..`........
00000d0: 340c 0000 0000 0000 340c 0000 0000 0000  4.......4.. | 00000d0: 0010
0000 0000 0000 0010 0000 0000 0000  ...........
00000e0: 0000 2000 0000 0000 0100 0000 0600 0000  .. ........ | 00000e0: 0100
0000 0000 0000 0100 0000 0600 0000  ...........
00000f0: 000e 0000 0000 0000 000e 6000 0000 0000  ..........` | 00000f0: f821
0000 0000 0000 0010 6000 0000 0000  .!........`
0000100: 000e 6000 0000 0000 7802 0000 0000 0000  ..`.....x.. | 0000100: 0000
0000 0000 0000 0010 0000 0000 0000  ...........
0000110: 8002 0000 0000 0000 0000 2000 0000 0000  ..........  | 0000110: 0010
0000 0000 0000 0100 0000 0000 0000  ...........
0000120: 0200 0000 0600 0000 180e 0000 0000 0000  ........... | 0000120: 0100
0000 0500 0000 f831 0000 0000 0000  .........1.
0000130: 180e 6000 0000 0000 180e 6000 0000 0000  ..`.......` | 0000130: 00b0
319c 847f 0000 0000 0000 0000 0000  ..1........
0000140: e001 0000 0000 0000 e001 0000 0000 0000  ........... | 0000140: 0000
0000 0000 0000 00b0 1b00 0000 0000  ...........
0000150: 0800 0000 0000 0000 0400 0000 0400 0000  ........... | 0000150: 0100
0000 0000 0000 0100 0000 0400 0000  ...........
0000160: 5402 0000 0000 0000 5402 4000 0000 0000  T.......T.@ | 0000160: f831
0000 0000 0000 0060 6d9c 847f 0000  .1.......`m
0000170: 5402 4000 0000 0000 4400 0000 0000 0000  T.@.....D.. | 0000170: 0000
```

```
0000 0000 0000 0040 0000 0000 0000  .........@.
0000180: 4400 0000 0000 0000 0400 0000 0000 0000  D.......... | 0000180: 0040
0000 0000 0000 0100 0000 0000 0000  .@.........
0000190: 50e5 7464 0400 0000 e00a 0000 0000 0000  P.td....... | 0000190: 0100
0000 0600 0000 f871 0000 0000 0000  .........q.
00001a0: e00a 4000 0000 0000 e00a 4000 0000 0000  ..@.......@ | 00001a0: 00a0
6d9c 847f 0000 0000 0000 0000 0000  ..m........
00001b0: 3c00 0000 0000 0000 3c00 0000 0000 0000  <.......<.. | 00001b0: 0020
0000 0000 0000 0020 0000 0000 0000  . ....... .
00001c0: 0400 0000 0000 0000 51e5 7464 0600 0000  ........Q.t | 00001c0: 0100
0000 0000 0000 0100 0000 0600 0000  ...........
00001d0: 0000 0000 0000 0000 0000 0000 0000 0000  ........... | 00001d0: f891
0000 0000 0000 00c0 6d9c 847f 0000  ..........m
00001e0: 0000 0000 0000 0000 0000 0000 0000 0000  ........... | 00001e0: 0000
0000 0000 0000 0050 0000 0000 0000  .........P.
00001f0: 0000 0000 0000 0000 1000 0000 0000 0000  ........... | 00001f0: 0050
0000 0000 0000 0100 0000 0000 0000  .P.........
0000200: 52e5 7464 0400 0000 000e 0000 0000 0000  R.td....... | 0000200: 0100
0000 0500 0000 f8e1 0000 0000 0000  ...........
0000210: 000e 6000 0000 0000 000e 6000 0000 0000  ..`.......` | 0000210: 0010
6e9c 847f 0000 0000 0000 0000 0000  ..n........
0000220: 0002 0000 0000 0000 0002 0000 0000 0000  ........... | 0000220: 0000
0000 0000 0000 0090 0100 0000 0000  ...........
0000230: 0100 0000 0000 0000 2f6c 6962 3634 2f6c  ......../li | 0000230: 0100
0000 0000 0000 0100 0000 0400 0000  ...........
...
```

What this shows us that even though there are some similiarities between the memory contents and the file contents, they are not exactly the same. In fact we see a deviation between the 2 dumps starting from 17 bytes, which is just pass the ELF magic bytes.

This shows that even though there's a mapping from the file to memory, it isn't exactly the same bytes. Either that, or somewhere in the dumping and hex translation, bytes got changed. It's hard to tell at this point in time.

Anyway moving on, we can alo use `objdump` to disassemble the executable file to see the actual assembly instructions that exist at the file. One thing to note, is that `objdump` uses the program's virtual memory address, as if were to be executed. It's not using the actual address at the file. Since we know the memory regions from `/proc/$PID/maps`, we can inspect the first `400000 - 401000` region.

```
$ objdump --disassemble-all --start-address=0x000000 --stop-address=0x401000
./memory_layout # use less of course

./memory_layout:     file format elf64-x86-64

```

```
Disassembly of section .interp:

0000000000400238 <.interp>:
  400238:       2f                      (bad)
  400239:       6c                      insb   (%dx),%es:(%rdi)
  40023a:       69 62 36 34 2f 6c 64    imul   $0x646c2f34,0x36(%rdx),%esp
  400241:       2d 6c 69 6e 75          sub    $0x756e696c,%eax
  400246:       78 2d                   js     400275 <_init-0x3d3>
  400248:       78 38                   js     400282 <_init-0x3c6>
  40024a:       36                      ss
  40024b:       2d 36 34 2e 73          sub    $0x732e3436,%eax
  400250:       6f                      outsl  %ds:(%rsi),(%dx)
  400251:       2e 32 00                xor    %cs:(%rax),%al

Disassembly of section .note.ABI-tag:

0000000000400254 <.note.ABI-tag>:
  400254:       04 00                   add    $0x0,%al
  400256:       00 00                   add    %al,(%rax)
  400258:       10 00                   adc    %al,(%rax)
  40025a:       00 00                   add    %al,(%rax)
  40025c:       01 00                   add    %eax,(%rax)
  40025e:       00 00                   add    %al,(%rax)
  400260:       47                      rex.RXB
  400261:       4e 55                   rex.WRX push %rbp
  400263:       00 00                   add    %al,(%rax)
  400265:       00 00                   add    %al,(%rax)
  400267:       00 02                   add    %al,(%rdx)
  400269:       00 00                   add    %al,(%rax)
...
```

Unlike `gcore` or manually dereferencing arbitrary pointers, we can see that `objdump` cannot or will not show us memory contents from `400000 - 400238`. Instead it starts showing from `400238`. This is because the stuff from `400000 - 400238` are not assembly instructions, they are just metadata, so `objdump` doesn't bother with them as it's designed to dump assembly code. Another thing to understand is that the elipsis `        ...` (not shown in the above example) (not to be confused with my own `...` meaning the output is an excerpt) mean null bytes. The `objdump` shows a byte by byte machine code with its decompiled equivalent assembly instruction. This is a disassembler, so the output assembly is not exactly what a human would write, because there can be optimisation, and lots of semantic information is thrown away. It's important to note that the hex addresses on the right represent the starting byte address, if there are multiple hex byte digits on right, that means they are joined up as a single assembly instruction. So the gap between `400251 - 400254` is represented by the 3 hex bytes at `2e 32 00`.

Let's jump to somewhere interesting, such as the actual "entry point" `0x400720` as reported by `readelf --file-header ./memory_layout`.

```
$ objdump --disassemble-all --start-address=0x000000 --stop-address=0x401000
./memory_layout | less +/400720
...
Disassembly of section .text:

0000000000400720 <_start>:
  400720:       31 ed                   xor    %ebp,%ebp
  400722:       49 89 d1                mov    %rdx,%r9
  400725:       5e                      pop    %rsi
  400726:       48 89 e2                mov    %rsp,%rdx
  400729:       48 83 e4 f0             and    $0xfffffffffffffff0,%rsp
  40072d:       50                      push   %rax
  40072e:       54                      push   %rsp
  40072f:       49 c7 c0 a0 09 40 00    mov    $0x4009a0,%r8
  400736:       48 c7 c1 30 09 40 00    mov    $0x400930,%rcx
  40073d:       48 c7 c7 62 08 40 00    mov    $0x400862,%rdi
  400744:       e8 87 ff ff ff          callq  4006d0 <__libc_start_main@plt>
  400749:       f4                      hlt
  40074a:       66 0f 1f 44 00 00       nopw   0x0(%rax,%rax,1)
...
```

Scrolling a bit up, we see that `objdump` reports this as the actual `.text` section, and at `400720`, this is the entry point of the program. What we have here, is the real first "procedure" that is executed by the CPU, the function behind the `main` function. And I think you get to play with this directly in C when you eschew the runtime library to produce a stand alone C executable. The assembly here is x86 64 bit assembly ([https://en.wikipedia.org/wiki/X86_assembly_language](https://en.wikipedia.org/wiki/X86_assembly_language)), which I guess is meant to run on backwards compatible Intel/AMD 64 bit processors. I don't know anymore about this particular assembly, so we'll have to study it later in [http://www.cs.virginia.edu/~evans/cs216/guides/x86.html](http://www.cs.virginia.edu/~evans/cs216/guides/x86.html)

What about our other 2 sections (we can see that there's a skip of `401000 - 600000`, this can also just be an arbitrary choice from the linker implementation):

```
600000 - 601000 - 4096 B    - 4 KiB
601000 - 602000 - 4096 B    - 4 KiB
```

```
$ objdump --disassemble-all --start-address=0x600000 --stop-address=0x602000
./memory_layout | less
```

There isn't much to talk about right now. It seems that `0x600000` contains more data and assembly. But the actual addresses for `.data` and `.bss` appears to be:

```
Disassembly of section .data:

0000000000601068 <__data_start>:
        ...

0000000000601070 <__dso_handle>:
        ...

Disassembly of section .bss:

0000000000601078 <__bss_start>:
        ...
```

It turns out we don't have anything in `.data` and `.bss`. This is because we do not have any static variables in our `./memory_layout.c` program!

To recap, our initial understanding of memory layout was:

```
0
Program Text (.text)
Initialised Data (.data)
Uninitialised Data (.bss)
Heap
    |
    v
Memory Mapped Region for Shared Libraries or Anything Else
    ^
    |
User Stack
```

Now we realise that it's actually:

```
0
Nothing here, because it was just an arbitrary choice by the linker
ELF and Program and Section Headers - 0x400000 on 64 bit
Program Text (.text) - Entry Point as Reported by readelf
Nothing Here either
Some unknown assembly and data - 0x600000
Initialised Data (.data) - 0x601068
Uninitialised Data (.bss) - 0x601078
Heap
    |
    v
Memory Mapped Region for Shared Libraries or Anything Else
    ^
    |
User Stack
```

Ok let's move on. After our executable file memory, we have a huge jump from `601000 - 7f849c31b000`.

```
00400000-00401000 r-xp 00000000 fc:00 1457150
/home/vagrant/c_tests/memory_layout
00600000-00601000 r--p 00000000 fc:00 1457150
/home/vagrant/c_tests/memory_layout
00601000-00602000 rw-p 00001000 fc:00 1457150
/home/vagrant/c_tests/memory_layout

... WHAT IS GOING ON HERE? ...

7f849c31b000-7f849c4d6000 r-xp 00000000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f849c4d6000-7f849c6d6000 ---p 001bb000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f849c6d6000-7f849c6da000 r--p 001bb000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f849c6da000-7f849c6dc000 rw-p 001bf000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f849c6dc000-7f849c6e1000 rw-p 00000000 00:00 0
7f849c6e1000-7f849c6fa000 r-xp 00000000 fc:00 1579084
/lib/x86_64-linux-gnu/libpthread-2.19.so
7f849c6fa000-7f849c8f9000 ---p 00019000 fc:00 1579084
/lib/x86_64-linux-gnu/libpthread-2.19.so
7f849c8f9000-7f849c8fa000 r--p 00018000 fc:00 1579084
/lib/x86_64-linux-gnu/libpthread-2.19.so
7f849c8fa000-7f849c8fb000 rw-p 00019000 fc:00 1579084
/lib/x86_64-linux-gnu/libpthread-2.19.so
7f849c8fb000-7f849c8ff000 rw-p 00000000 00:00 0
7f849c8ff000-7f849c922000 r-xp 00000000 fc:00 1579072
/lib/x86_64-linux-gnu/ld-2.19.so
7f849cb10000-7f849cb13000 rw-p 00000000 00:00 0
7f849cb1d000-7f849cb21000 rw-p 00000000 00:00 0
7f849cb21000-7f849cb22000 r--p 00022000 fc:00 1579072
/lib/x86_64-linux-gnu/ld-2.19.so
7f849cb22000-7f849cb23000 rw-p 00023000 fc:00 1579072
/lib/x86_64-linux-gnu/ld-2.19.so
7f849cb23000-7f849cb24000 rw-p 00000000 00:00 0
7fffb5d61000-7fffb5d82000 rw-p 00000000 00:00 0                          [stack]
7fffb5dfe000-7fffb5e00000 r-xp 00000000 00:00 0                          [vdso]
```

```
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
```

That is roughly a massive step of 127 Tebibytes. Why such a large jump in the address space? Well this is where the malloc implementation comes in. This documentation https://github.com/torvalds/linux/blob/master/Documentation/x86/x86_64/mm.txt shows how the memory is structured in this way:

> ```
> Virtual memory map with 4 level page tables:
>
> 0000000000000000 - 00007fffffffffff (=47 bits) user space, different per mm
> hole caused by [48:63] sign extension
> ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for
> hypervisor
> ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys.
> memory
> ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
> ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
> ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
> ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
> ... unused hole ...
> ffffec0000000000 - fffffc0000000000 (=44 bits) kasan shadow memory (16TB)
> ... unused hole ...
> ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
> ... unused hole ...
> ffffffef00000000 - ffffffff00000000 (=64 GB) EFI region mapping space
> ... unused hole ...
> ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from
> phys 0
> ffffffffa0000000 - ffffffffff5fffff (=1525 MB) module mapping space
> ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
> ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
> ```

What we can see is that, is that Linux's memory map reserves the first `0000000000000000 - 00007fffffffffff` as user space memory. It turns out that 47 bits is enough to reserve roughly 128 TiB. http://unix.stackexchange.com/a/64490/56970

Well if we take look at the first and last of these sections of memory:

```
7f849c31b000-7f849c4d6000 r-xp 00000000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
...
7fffb5dfe000-7fffb5e00000 r-xp 00000000 00:00 0                          [vdso]
```

It appears that these regions are almost at the very bottom of the 128 TiB range reserved for user space memory. Considering that there's a 127 TiB gap, this basically means that our malloc uses the user space range `0000000000000000 - 00007fffffffffff` from both ends. From the low end, it grows the heap upwards (upwards in address numbers). While at the high end, it grows the stack downwards (downwards in address numbers).

At the same time, the stack is actually a fixed section of memory, so it cannot actually grow as much as the heap. On the high end, but lower than the stack, we see lots of memory regions assigned for the shared libraries and anonymous buffers which are probably used by the shared libraries.

We can also view the shared libraries that are being used by the executable. This determines which shared libraries will also be loaded into memory at startup. However remember that libraries and code can also be dynamically loaded, and cannot be seen by the linker. By the way `ldd` stands for "list dynamic dependencies".

```
$ ldd ./memory_layout
        linux-vdso.so.1 =>  (0x00007fff1a573000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
(0x00007f361ab4e000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f361a788000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f361ad7a000)
```

You'll notice that if you run `ldd` multiple times, each time it prints a different address for the shared libraries. This corresponds to rerunning the program multiple times and checking that `/proc/$PID/maps` also shows different addresses for the shared libraries. This is due to the "PIE" position independent code discussed above. Basically every time you use `ldd` it does call the linker, and the linker performs address randomisation. For more information about the reasoning behind address space randomisation, see: ASLR. Also you can check whether the kernel has enabled ASLR by running `cat /proc/sys/kernel/randomize_va_space`.

We can see that there are in fact 4 shared libraries. The `vdso` library is not loaded from the filesystem, but provided by the OS.

Also: `/lib64/ld-linux-x86-64.so.2 => /lib/x86_64-linux-gnu/ld-2.19.so`, it's a symlink.

Finally we're on the last few regions:

```
7fffb5d61000-7fffb5d82000 rw-p 00000000 00:00 0                          [stack]
7fffb5dfe000-7fffb5e00000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
```

Here are the relevant sizes for each region:

```
7fffb5d61000     - 7fffb5d82000     - [stack]    - 135168 B - 132 KiB
7fffb5dfe000     - 7fffb5e00000     - [vdso]     - 8192 B   - 8 KiB
ffffffffff600000 - ffffffffff601000 - [vsyscall] - 4096 B   - 4 KiB
```

Our initial stack size is allocated to 132 KiB. I suspect this can be changed with runtime or compiler flags.

So what is `vdso` and `vsyscall`? Both are mechanisms to allow faster syscalls, that is syscalls without context switching between user space and kernel space. The `vsyscall` has now been replaced by `vdso`, but the `vsyscall` is left there for compatibility reasons. The main differences are that:

- `vsyscall` - is always fixed to `ffffffffff600000` and maximum size of 8 MiB, even if PIC or PIE is enabled
- `vdso` - not fixed, but acts like a shared library, thus its address is subject to ASLR (address space layout randomisation)
- `vsyscall` - provides 3 syscalls: `gettimeofday` (`0xffffffffff600000`), `time` (`0xffffffffff600400`), `getcpu` (`0xffffffffff600800`), even though it given the reserved range `ffffffffff600000 - ffffffffffdfffff` of 8 MiB by Linux in 64 bit ELF executables.
- `vdso` - provides 4 syscalls: `__vdso_clock_gettime`, `__vdso_getcpu`, `__vdso_gettimeofday` and `__vdso_time`, however more syscalls can be added to `vdso` in the future.

For more info about `vdso` and `vsyscall`, see: https://0xax.gitbooks.io/linux-insides/content/SysCall/syscall-3.html

It's worth pointing out, that now that we're past the 128 TiB reserved for user space memory, we're now looking at segments of memory provided and managed by the OS. As listed here: https://github.com/torvalds/linux/blob/master/Documentation/x86/x86_64/mm.txt These sections are what we're talking about.

```
ffff800000000000 - ffff87ffffffffff (=43 bits) guard hole, reserved for
hypervisor
ffff880000000000 - ffffc7ffffffffff (=64 TB) direct mapping of all phys. memory
ffffc80000000000 - ffffc8ffffffffff (=40 bits) hole
ffffc90000000000 - ffffe8ffffffffff (=45 bits) vmalloc/ioremap space
ffffe90000000000 - ffffe9ffffffffff (=40 bits) hole
```

```
ffffea0000000000 - ffffeaffffffffff (=40 bits) virtual memory map (1TB)
... unused hole ...
ffffec0000000000 - fffffc0000000000 (=44 bits) kasan shadow memory (16TB)
... unused hole ...
ffffff0000000000 - ffffff7fffffffff (=39 bits) %esp fixup stacks
... unused hole ...
ffffffef00000000 - ffffffff00000000 (=64 GB) EFI region mapping space
... unused hole ...
ffffffff80000000 - ffffffffa0000000 (=512 MB)  kernel text mapping, from phys 0
ffffffffa0000000 - ffffffffff5fffff (=1525 MB) module mapping space
ffffffffff600000 - ffffffffffdfffff (=8 MB) vsyscalls
ffffffffffe00000 - ffffffffffffffff (=2 MB) unused hole
```

Out of the above sections, we currently only see the `vsyscall` region. The rest have not appeared yet.

Let's now move on with the program, and allocate our first heap. Our `/proc/$PID/maps` is now (note that the addresses have changed because I have reran the program):

```
$ ./memory_layout
Welcome to per thread arena example::1546
Before malloc in the main thread

After malloc and before free in main thread
^Z
[1]+  Stopped                 ./memory_layout
$ cat /proc/1546/maps
00400000-00401000 r-xp 00000000 fc:00 1457150
/home/vagrant/c_tests/memory_layout
00600000-00601000 r--p 00000000 fc:00 1457150
/home/vagrant/c_tests/memory_layout
00601000-00602000 rw-p 00001000 fc:00 1457150
/home/vagrant/c_tests/memory_layout
019a3000-019c4000 rw-p 00000000 00:00 0                                  [heap]
...
```

We now see our first `[heap]` region. It exactly: 135168 B - 132 KiB. (Currently the same as our stack size!) Remember we allocated specifically 1000 bytes: `addr = (char *) malloc(1000);` at the beginning. So how did a 1000 Bytes become 132 Kibibytes? Well as we said before, anything smaller than `MMAP_THRESHOLD` uses `brk` syscall. It appears that the `brk` / `sbrk` is called with a padded size in order to reduce the number of syscalls made and the number of context switches. Most programs most likely require more heap than 1000 Bytes, so the system may as well pad the `brk` call to cache some heap memory, and new `brk` or `mmap` calls for heap increase will only occur once you exhaust the padded heap of 132 KiB. The padding calculation is done with:

```
    /* Request enough space for nb + pad + overhead */
    size = nb + mp_.top_pad + MINSIZE;
```

Where `mp_.top_pad` is by default set to 128 * 1024 = 128 KiB. We still have 4 KiB difference. But remember our page size `getconf PAGESIZE` gives us 4096, meaning each page is 4 KiB. This means when allocating 1000 Bytes in our program, a full page is allocated being 4 KiB. And 4 KiB + 128 KiB is 132 KiB, which is the size of our heap. This padding is not a padding to a fixed size, but padding that is always added to the amount allocated via `brk` / `sbrk`. This means that 128 KiB by default is always added to how ever much memory you're trying allocate. However this padding only applies to `brk` / `sbrk`, and not `mmap`, remember that the past the `MMAP_THRESHOLD`, `mmap` takes over from `brk` / `sbrk`. Which means the padding will no longer be applied. However I'm not sure whether the `MMAP_THRESHOLD` is checked prior to the padding, or after the padding. It seems like it should be prior to the padding.

The padding size can be changed with a call like `mallopt(M_TOP_PAD, 1);`, which changes the `M_TOP_PAD` to be 1 Byte. Mallocing 1000 Bytes now will create only a page of 4 KiB.

For more information, see: [http://stackoverflow.com/a/23951267/582917](http://stackoverflow.com/a/23951267/582917)

Why is the older `brk` / `sbrk` getting replaced by newer `mmap`, when an allocation is equal or greater than the `MMAP_THRESHOLD`? Well the `brk` / `sbrk` calls only allow increasing the size of the heap contiguously. If you're just using `malloc` for small things, all of it should be able to be allocated contiguously in the heap, and when it reaches the heap end, the heap can be extended in size without any problems. But for larger allocatiosn, `mmap` is used, and this heap space does need to be contiguously joined with the `brk` / `sbrk` heap space. So it's more flexible. Memory fragmentation is reduced for small objects in this situation. Also the `mmap` call is more flexible, so that `brk` / `sbrk` can be implemented with `mmap`, whereas `mmap` cannot be implemented with `brk` / `sbrk`. One limitation of `brk` / `sbrk`, is that if the top byte in the `brk` / `sbrk` heap space is not freed, the heap size cannot be reduced.

Let's look at a simple program that allocates more than `MMAP_THRESHOLD` (it can also be overridden using `mallopt`):

```
#include <stdlib.h>
#include <stdio.h>

int main () {

    printf("Look at /proc/%d/maps\n", getpid());
    // allocate 200 KiB, forcing a mmap instead of brk
    char * addr = (char *) malloc(204800);

    getchar();

    free(addr);

    return 0;

}
```

Running the above with `strace` gives us:

```
$ strace ./mmap
...
mmap(NULL, 3953344, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0x7f3b113ce000
mprotect(0x7f3b11589000, 2097152, PROT_NONE) = 0
mmap(0x7f3b11789000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1bb000) = 0x7f3b11789000
mmap(0x7f3b1178f000, 17088, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f3b1178f000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f3b119a7000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f3b119a5000
arch_prctl(ARCH_SET_FS, 0x7f3b119a5740) = 0
mprotect(0x7f3b11789000, 16384, PROT_READ) = 0
mprotect(0x600000, 4096, PROT_READ)     = 0
mprotect(0x7f3b119b6000, 4096, PROT_READ) = 0
munmap(0x7f3b119a8000, 45778)           = 0
getpid()                                = 1604
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f3b119b3000
write(1, "Look at /proc/1604/maps\n", 24Look at /proc/1604/maps
) = 24
mmap(NULL, 208896, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f3b11972000
fstat(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f3b119b2000
...
```

There's many `mmap` calls in the above strace. How do we find the `mmap` that our program called, and not the shared libraries or the linker or other stuff? Well the closest call is this one:

```
mmap(NULL, 208896, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7f3b11972000
```

It's actually 204800 B + 4096 B = 208 896 B. I'm not sure why an extra 4 KiB is added, because 200 KiB is exactly divisible by the page size of 4 KiB. It maybe another feature. One thing to note is there isn't any kind of obvious way we can pinpoint our program's syscalls over some other syscall, but we can look at the context of the call, that is the previous and subsequent lines to find the exact control flow. Using things like `getchar` can also pause the `strace`. Consider that immeidately after mmapping 204800 bytes, there's a `fstat` and another `mmap` call, before finally `getchar` is called. I have no idea where these calls are coming from, so in the future, we should look for some easy way to label the syscalls so we can find them quicker. The `strace` tells us that this memory mapped region is mapped to `0x7f3b11972000`. Looking at the process's `/proc/$PID/maps`:

```
$ cat /proc/1604/maps
00400000-00401000 r-xp 00000000 fc:00 1446413
/home/vagrant/c_tests/test
00600000-00601000 r--p 00000000 fc:00 1446413
/home/vagrant/c_tests/test
00601000-00602000 rw-p 00001000 fc:00 1446413
/home/vagrant/c_tests/test
7f3b113ce000-7f3b11589000 r-xp 00000000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f3b11589000-7f3b11789000 ---p 001bb000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f3b11789000-7f3b1178d000 r--p 001bb000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f3b1178d000-7f3b1178f000 rw-p 001bf000 fc:00 1579071
/lib/x86_64-linux-gnu/libc-2.19.so
7f3b1178f000-7f3b11794000 rw-p 00000000 00:00 0
7f3b11794000-7f3b117b7000 r-xp 00000000 fc:00 1579072
/lib/x86_64-linux-gnu/ld-2.19.so
7f3b11972000-7f3b119a8000 rw-p 00000000 00:00 0
7f3b119b2000-7f3b119b6000 rw-p 00000000 00:00 0
7f3b119b6000-7f3b119b7000 r--p 00022000 fc:00 1579072
/lib/x86_64-linux-gnu/ld-2.19.so
7f3b119b7000-7f3b119b8000 rw-p 00023000 fc:00 1579072
/lib/x86_64-linux-gnu/ld-2.19.so
7f3b119b8000-7f3b119b9000 rw-p 00000000 00:00 0
7fff8f747000-7fff8f768000 rw-p 00000000 00:00 0                          [stack]
7fff8f7fe000-7fff8f800000 r-xp 00000000 00:00 0                          [vdso]
```

```
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
[vsyscall]
```

We can find out mmapped heap here:

```
7f3b11972000-7f3b119a8000 rw-p 00000000 00:00 0
```

As you can see, when malloc switches to using `mmap`, the regions acquired are not part of the so called `[heap]` region, which is only provided by the `brk`/`sbrk` calls. It's unlabelled! We can also see that this kind of heap is not put in the same area as the `brk`/`sbrk` heap, which we understood to start at the low end and grow upwards in address space. Instead this mmapped heap is located in the same area as the shared libraries, putting it at the high end of the reserved user space address range. However this region as shown by `/proc/$PID/maps` is actually 221184 B - 216 KiB. It is exactly 12 KiB from 208896. Another mystery! Why do we have different byte sizes, even though `mmap` in the `strace` called exactly `208896`?

Looking at another `mmap` call also shows that the corresponding region in `/proc/$PID/maps` has a difference of 12 KiB. It is possible that 12 KiB here represents some sort of memory mapping overhead, used by malloc to keep track or understand the type of memory that is available here. Or it could just be extra padding as well. So we can say here that something is consistently adding 12 KiB to whatever we're mmapping, and that there is an extra 4 KiB added to my 200 KiB `malloc`.

By the way, there's also a tool called `binwalk` that's really useful for examining firmware images which may include multiple executable files and metadata. Remember you can in fact embed a file into an executable. Which is kind of like how viruses work. I used it to inspect the initrd of NixOS and figure out how it was structured. Combine it with `dd` you can cut and slice and splice binary blobs easily!

At this point, we can continue investigating the heap from our original program, and the thread heaps as well. But I'm stopping here for now.

To be continued...

If you're interested in these sorts of things, you may want to work with me on https://matrix.ai, feel free to send me a message!

**Sarkin** commented on May 19, 2017

Thanks for the write-up, it really helped me understand how process memory is laid out.

**MarcioJales** commented on Aug 4, 2017

Very valuable!
By the way:
"Our initial stack size is allocated to 132 KiB. I suspect this can be changed with runtime or compiler flags."
the *alloca* library call may change the stack frame size.

**guhetier** commented on Oct 26, 2017

Very interesting !
However, I get slightly different result while experimenting.
For example, a heap of 132KiB is always allocated at the start of a program, even for a simple "hello world".
I suspect an optimization to avoid system calls, once again.

**xiaotaoz** commented on Feb 1, 2018

Great!
Thanks for your sharing!

**EMH1899** commented on Aug 25, 2018

Thanks，It is very valuable.BTW，Have you investigated the heap and thread heaps?

**sergiPopescou** commented on Mar 21, 2019

> Very interesting !
> However, I get slightly different result while experimenting.
> For example, a heap of 132KiB is always allocated at the start of a program, even for a simple "hello world".
> I suspect an optimization to avoid system calls, once again.

Me too!

**fengzhuye** commented on Jul 5, 2019

That's a really great post!

**PAX523** commented on Oct 6, 2019

Thanks for your very interesting research! Did you find out why `Elf64_Ehdr.e_type` is set to `ET_EXEC` in file, but is `ET_CORE` in memory (and, respectively, all those other modifications)?

Following is interesting:

```
$ diff --side-by-side ./file.hex ./memory.hex # try piping into less
0000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF....... | 0000000: 7f45 4c46 0201 0100 0000
0000 0000 0000  .ELF.......
0000010: 0200 3e00 0100 0000 2007 4000 0000 0000  ..>..... .@ | 0000010: 0400 3e00 0100 0000 0000
0000 0000 0000  ..>........
```

`2007 4000 0000 0000` vs. `0000 0000 0000 0000`

The very last 8-byte element represents `Elf64_Ehdr.e_entry`. Obviously, in memory it is set to zero. The documentation (https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter6-43405/index.html) says:

> The virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

The next fields ( `e_phoff` and `e_shoff` ) are also weird:

```
0000020: 4000 0000 0000 0000 a811 0000 0000 0000  @.......... | 0000020: 4000 0000 0000 0000 1022
0400 0000 0000  @.......".
```

Each of them consume 8 bytes. `e_phoff` represents the program header table's file offset in bytes, whereas `e_shoff` describes the section header table's file offset in bytes. `e_phoff` matches the value in memory mapping. But `e_shoff` is different. That's strange, because the byte offset in file can't has changed?

**CMCDragonkai** commented on Oct 6, 2019 Author

There's alot of security applied to process memory layouts nowadays. So the memory offsets can all be scrambled once the program is loaded.

**PAX523** commented on Oct 6, 2019

Does it mean that `e_shoff` becomes set to a memory offset even if it originally was a file offset? Since `e_shoff` was a file offset to the Section Header Table, did `0400 1022` really point to its start offset in memory? At least, it looks just like that.

`Elf64_Phdr.p_vaddr` = `0000 0000 4000 4000`

> p_vaddr: The virtual address at which the first byte of the segment resides in memory.

```
0000050: 4000 4000 0000 0000 4000 4000 0000 0000  @.@.....@.@ | 0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ...........
```

**divyanand** commented on Oct 22, 2019

`It's actually 204800 B + 4096 B = 208 896 B. I'm not sure why an extra 4 KiB is added, because 200 KiB is exactly divisible by the page size of 4 KiB. It maybe another feature.`

That is probably for book-keeping.
During malloc, some information related allocation, like the size of memory being allocated, will be stored ahead of the returned pointer.
So if malloc returns address 0x00003, the book-keeping is probably stored in 0x00-0x02.
This will be later used by free() to know exactly how much memory to free.

In your case, you are requesting complete 50 pages (50 * 4096 = 204800) bytes. So there is not enough memory left in the page to store the book-keeping info.
From tiral-and-error I found that if you allocate for 204776 bytes (24 bytes less), you'll be given exactly 204800 bytes.

So I guess malloc needs 24-bytes.

**pulsar17** commented on Nov 1, 2021

> BTW: Wouldn't it be great if we had a tool that examined /proc/$PID/maps and gave exact human readable byte sizes instead?

There is also a corresponding `/proc/$PID/smaps` file that lists sizes of the various sections. For a dummy program, the heap section's details as shown by the `smaps` file are:

```
5571371fb000-55713721c000 rw-p 00000000 00:00 0                          [heap]
Size:                132 kB
KernelPageSize:        4 kB
MMUPageSize:           4 kB
Rss:                   4 kB
Pss:                   4 kB
Shared_Clean:          0 kB
Shared_Dirty:          0 kB
Private_Clean:         0 kB
Private_Dirty:         4 kB
Referenced:            4 kB
Anonymous:             4 kB
LazyFree:              0 kB
AnonHugePages:         0 kB
ShmemPmdMapped:        0 kB
```

```
FilePmdMapped:         0 kB
Shared_Hugetlb:        0 kB
Private_Hugetlb:       0 kB
Swap:                  0 kB
SwapPss:               0 kB
Locked:                0 kB
THPeligible:    0
VmFlags: rd wr mr mw me ac sd
```

**tkchk** commented on Dec 2, 2021

Nice writeup! Kudos

**sheercoding** commented on Dec 8, 2021

Thanks for writing, it's really helpful.
great!

**cthumuluru** commented on Jan 16, 2022

Wow, this is amazing write up. Thanks!

**djanatyn** commented on Aug 17, 2022

> BTW: Wouldn't it be great if we had a tool that examined /proc/$PID/maps and gave exact human readable byte sizes instead?

Does `pmap` accomplish this?

- https://linux.die.net/man/1/pmap

**dnorthup-ums** commented 5 days ago

FYI: The linux on x86_64 memory map documentation has moved to:
https://github.com/torvalds/linux/blob/master/Documentation/arch/x86/x86_64/mm.rst