Hack 虚拟内存系列（四）：malloc，堆和程序间断点

lm0963 / 2018-08-06 12:08:12 / 浏览数 2663 技术文章 技术文章 顶(0) 踩(0)

本文翻译自：https://blog.holbertonschool.com/hack-the-virtual-memory-malloc-the-heap-the-program-break

Hack the Virtual Memory: malloc, the heap & the program break

堆

在本章中，我们将查看堆和malloc，以回答我们在上一章末尾遗留的一些问题：

- 为什么我们分配的内存不在堆的最开始处（0x2050010 vs 02050000）？前16个字节被用来干什么？
- 堆真的在向上增长吗？

前提

为了完全理解本文，你需要知道：

- C语言的基础知识（特别是指针）
- Linux文件系统和shell的基础知识
- 我们将用到/proc/[pid]/maps文件（查阅man proc或阅读本系列的第一篇文章： 第0章：C字符串和/proc ）

环境

所有脚本和程序都已经在以下系统上进行过测试：

- Ubuntu
  - Linux ubuntu 4.4.0-31-generic #50~14.04.1-Ubuntu SMP Wed Jul 13 01:07:32 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux

使用的工具：

- gcc
  - gcc（Ubuntu 4.8.4-2ubuntu1~14.04.3）4.8.4
- glibc 2.19（如果你需要检查你的glibc版本，请参见version.c）
- strace的
  - strace —— version 4.8

下文描述均基于此系统/环境，在其他系统上可能会不一样
我们还将查看Linux源代码。如果你使用的是Ubuntu，则可以通过以下命令下载当前内核的源代码：

```
apt-get source linux-image-$(uname -r)
```

malloc

malloc是用于动态分配内存的常用函数。该内存分配在“堆”上。
注意：malloc不是系统调用。

来自man malloc：

```
[...] allocate dynamic memory[...]
void *malloc(size_t size);
[...]
The malloc() function allocates size bytes and returns a pointer to the allocated memory.
```

No malloc, no [heap]

让我们看一下并没有调用malloc函数的进程的内存区域（0-main.c）。

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

/**
* main - do nothing
*
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
```

```
*/
int main(void)
{
   getchar();
   return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 0-main.c -o 0
julien@holberton:~/holberton/w/hackthevm3$ ./0
```

*小提示（1/3）：/proc/[pid]/maps文件中会列出进程的内存区域。因此，我们首先需要知道进程的PID。可使用ps命令查看PID；ps aux打印的第二列是进程的PID。请阅读第0章以了解更多信息。

```
julien@holberton:/tmp$ ps aux | grep \ \./0$
julien     3638  0.0  0.0   4200   648 pts/9    S+   12:01   0:00 ./0
```

*小提示（2/3）：从上面的输出中，可以看到我们要查看的进程的PID是3638.因此，maps文件可以在目录/proc/3638中找到。

```
julien@holberton:/tmp$ cd /proc/3638
```

*小提示（3/3）：maps文件包含进程的内存区域。此文件中每行的格式为：
地址 权限 偏移 dev inode 路径名 （dev：文件的主设备号和次设备号；inode：设备的节点号，0表示没有节点与内存相对应）

```
julien@holberton:/proc/3638$ cat maps
00400000-00401000 r-xp 00000000 08:01 174583                             /home/julien/holberton/w/hack_the_virtual_memory/03.
00600000-00601000 r--p 00000000 08:01 174583                             /home/julien/holberton/w/hack_the_virtual_memory/03.
00601000-00602000 rw-p 00001000 08:01 174583                             /home/julien/holberton/w/hack_the_virtual_memory/03.
7f38f87d7000-7f38f8991000 r-xp 00000000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f38f8991000-7f38f8b91000 ---p 001ba000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f38f8b91000-7f38f8b95000 r--p 001ba000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f38f8b95000-7f38f8b97000 rw-p 001be000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f38f8b97000-7f38f8b9c000 rw-p 00000000 00:00 0
7f38f8b9c000-7f38f8bbf000 r-xp 00000000 08:01 136229                     /lib/x86_64-linux-gnu/ld-2.19.so
7f38f8da3000-7f38f8da6000 rw-p 00000000 00:00 0
7f38f8dbb000-7f38f8dbe000 rw-p 00000000 00:00 0
7f38f8dbe000-7f38f8dbf000 r--p 00022000 08:01 136229                     /lib/x86_64-linux-gnu/ld-2.19.so
7f38f8dbf000-7f38f8dc0000 rw-p 00023000 08:01 136229                     /lib/x86_64-linux-gnu/ld-2.19.so
7f38f8dc0000-7f38f8dc1000 rw-p 00000000 00:00 0
7ffdd85c5000-7ffdd85e6000 rw-p 00000000 00:00 0                          [stack]
7ffdd85f2000-7ffdd85f4000 r--p 00000000 00:00 0                          [vvar]
7ffdd85f4000-7ffdd85f6000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
julien@holberton:/proc/3638$
```

*注意：hackthevm3是hack_the_virtual_memory/03. The Heap/的符号链接。

->正如我们从上面的maps文件中看到的那样，没有[heap]区域。

malloc(x)

运行一个调用malloc函数的程序并重复上面的操作（1-main.c）：

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

/**
* main - 1 call to malloc
*
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
*/
int main(void)
{
   malloc(1);
   getchar();
   return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 1-main.c -o 1
julien@holberton:~/holberton/w/hackthevm3$ ./1
```

```
julien@holberton:/proc/3638$ ps aux | grep \ \./1$
julien     3718  0.0  0.0   4332   660 pts/9    S+   12:09   0:00 ./1
julien@holberton:/proc/3638$ cd /proc/3718
julien@holberton:/proc/3718$ cat maps
00400000-00401000 r-xp 00000000 08:01 176964                             /home/julien/holberton/w/hack_the_virtual_memory/03.
00600000-00601000 r--p 00000000 08:01 176964                             /home/julien/holberton/w/hack_the_virtual_memory/03.
00601000-00602000 rw-p 00001000 08:01 176964                             /home/julien/holberton/w/hack_the_virtual_memory/03.
01195000-011b6000 rw-p 00000000 00:00 0                                  [heap]
...
julien@holberton:/proc/3718$
```

-> 存在[heap]
检查malloc的返回值，以确保返回的地址在堆区域中（2-main.c）：

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

/**
* main - prints the malloc returned address
*
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
*/
int main(void)
{
   void *p;

   p = malloc(1);
   printf("%p\n", p);
   getchar();
   return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 2-main.c -o 2
julien@holberton:~/holberton/w/hackthevm3$ ./2
0x24d6010
```

```
julien@holberton:/proc/3718$ ps aux | grep \ \./2$
julien     3834  0.0  0.0   4336   676 pts/9    S+   12:48   0:00 ./2
julien@holberton:/proc/3718$ cd /proc/3834
julien@holberton:/proc/3834$ cat maps
00400000-00401000 r-xp 00000000 08:01 176966                             /home/julien/holberton/w/hack_the_virtual_memory/03.
00600000-00601000 r--p 00000000 08:01 176966                             /home/julien/holberton/w/hack_the_virtual_memory/03.
00601000-00602000 rw-p 00001000 08:01 176966                             /home/julien/holberton/w/hack_the_virtual_memory/03.
024d6000-024f7000 rw-p 00000000 00:00 0                                  [heap]
...
julien@holberton:/proc/3834$
```

-> 024d6000 <0x24d6010 < 024f7000
malloc返回的地址在堆区域内。并且正如我们在前一章中看到的那样，返回的地址并不在堆区域的最开始处；我们稍后会看到原因。

strace, brk 和 sbrk

malloc是一个“常规”函数（与系统调用不同），因此它必须调用某种类型的系统调用才能操作堆。让我们用strace来查看一下。
strace是一个用于跟踪系统调用和信号的程序。在执行main函数之前，所有程序都会调用一些系统调用。为了知道malloc使用了哪些系统调用，我们将在调用malloc前和调

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

/**
* main - let's find out which syscall malloc is using
*
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
*/
int main(void)
{
   void *p;

   write(1, "BEFORE MALLOC\n", 14);
   p = malloc(1);
   write(1, "AFTER MALLOC\n", 13);
   printf("%p\n", p);
   getchar();
   return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 3-main.c -o 3
julien@holberton:~/holberton/w/hackthevm3$ strace ./3
execve("./3", ["./3"], [/* 61 vars */]) = 0
...
write(1, "BEFORE MALLOC\n", 14BEFORE MALLOC
)          = 14
brk(0)                                  = 0xe70000
brk(0xe91000)                           = 0xe91000
write(1, "AFTER MALLOC\n", 13AFTER MALLOC
)           = 13
...
read(0,
```

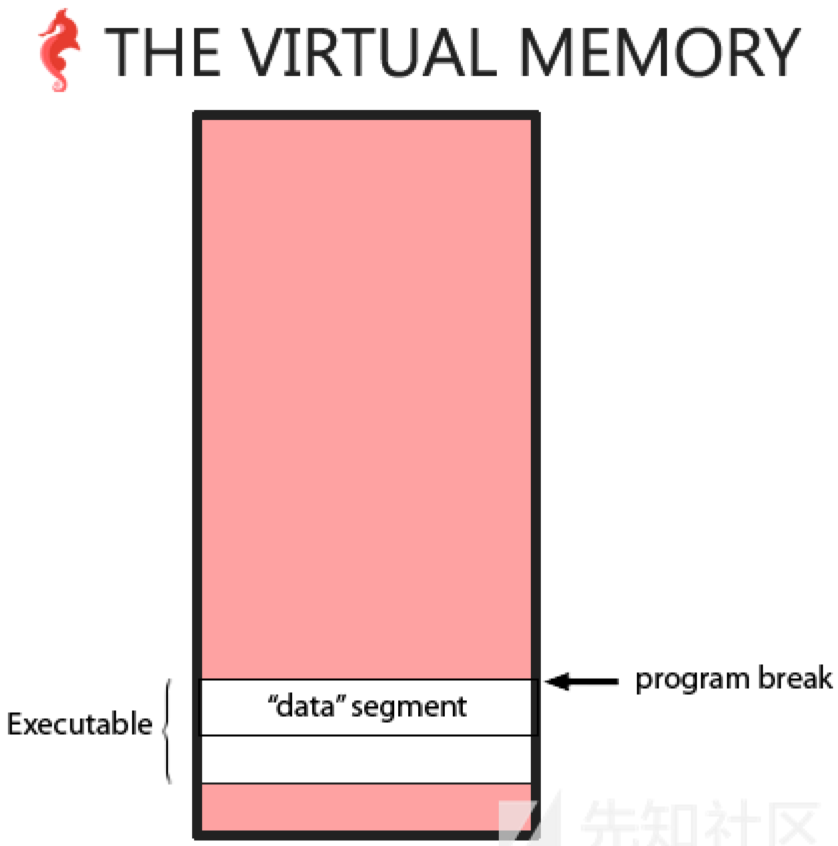
在上面的列表中我们关注这个：

```
brk(0)                                  = 0xe70000
brk(0xe91000)                           = 0xe91000
```

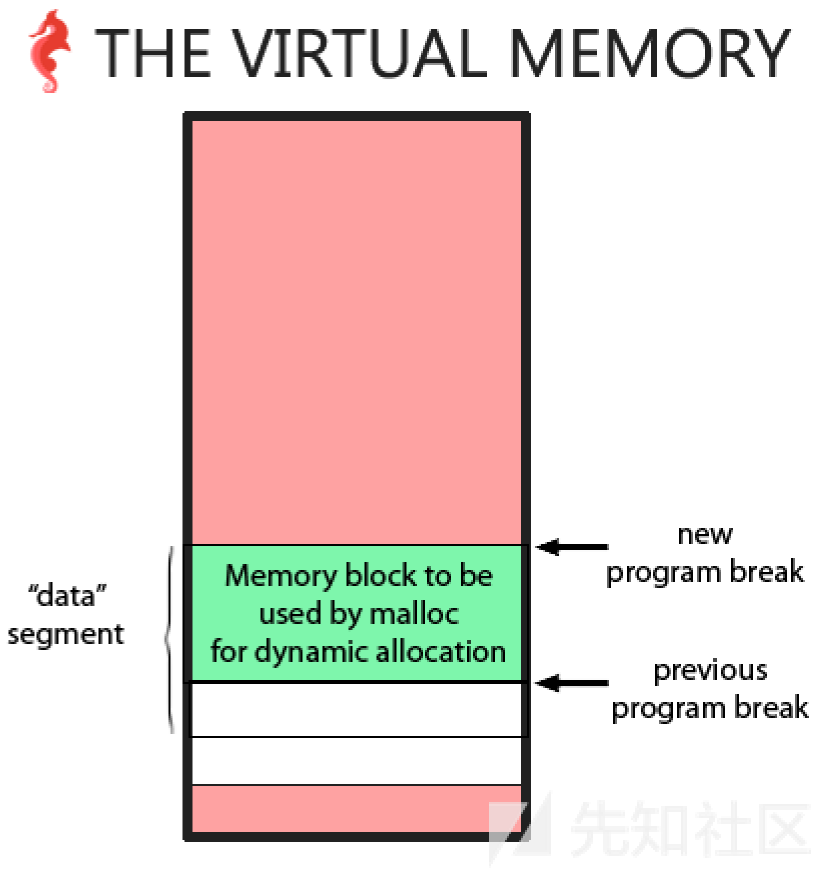
-> malloc使用brk系统调用来操纵堆。从brk手册上（man brk），我们可以看到这个系统调用做了什么：

```
...
      int brk(void *addr);
      void *sbrk(intptr_t increment);
...
DESCRIPTION
      brk() and sbrk() change the location of the program  break,  which  defines the end of the process's data segment (i.e.,
      brk() sets the end of the data segment to the value specified by addr, when that  value  is  reasonable,  the system has
      sbrk() increments the program's data space  by  increment  bytes.   Calling sbrk()  with  an increment of 0 can be used
```

程序间断点是程序的数据段结束后的第一个位置的虚拟内存地址。



通过brk或sbrk，增加程序间断点的值，malloc函数预留新的内存空间之后动态分配给进程使用（使用malloc）。



所以堆实际上是程序数据段的扩展。

第一次调用brk（brk（0））会将程序间断点的当前地址返回给malloc。第二次调用brk才真正通过增加程序间断点的值来创建新内存（因为0xe91000> 0xe70000）。在上面的示例中，堆开始于0xe70000并结束于0xe91000。让我们仔细查看一下/proc/[PID]/maps文件：

```
julien@holberton:/proc/3855$ ps aux | grep \ \./3$
julien     4011  0.0  0.0   4748   708 pts/9    S+   13:04   0:00 strace ./3
julien     4014  0.0  0.0   4336   644 pts/9    S+   13:04   0:00 ./3
julien@holberton:/proc/3855$ cd /proc/4014
julien@holberton:/proc/4014$ cat maps
00400000-00401000 r-xp 00000000 08:01 176967                             /home/julien/holberton/w/hack_the_virtual_memory/03.
00600000-00601000 r--p 00000000 08:01 176967                             /home/julien/holberton/w/hack_the_virtual_memory/03.
00601000-00602000 rw-p 00001000 08:01 176967                             /home/julien/holberton/w/hack_the_virtual_memory/03.
00e70000-00e91000 rw-p 00000000 00:00 0                                  [heap]
...
julien@holberton:/proc/4014$
```

-> 00e70000-00e91000 rw -p 00000000 00:00 0 [heap]符合brk返回给malloc的指针值。

为什么当我们只要求1个字节时，malloc将堆增大了0x00e91000 - 0x00e70000 = 0x21000（135168）个字节？

多次调用malloc

当我们多次调用malloc时，会发生什么？（4-main.c）

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

/**
* main - many calls to malloc
*
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
*/
int main(void)
{
   void *p;

   write(1, "BEFORE MALLOC #0\n", 17);
   p = malloc(1024);
```

```
    write(1, "AFTER MALLOC #0\n", 16);
    printf("%p\n", p);

    write(1, "BEFORE MALLOC #1\n", 17);
    p = malloc(1024);
    write(1, "AFTER MALLOC #1\n", 16);
    printf("%p\n", p);

    write(1, "BEFORE MALLOC #2\n", 17);
    p = malloc(1024);
    write(1, "AFTER MALLOC #2\n", 16);
    printf("%p\n", p);

    write(1, "BEFORE MALLOC #3\n", 17);
    p = malloc(1024);
    write(1, "AFTER MALLOC #3\n", 16);
    printf("%p\n", p);

    getchar();
    return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 4-main.c -o 4
julien@holberton:~/holberton/w/hackthevm3$ strace ./4
execve("./4", ["./4"], [/* 61 vars */]) = 0
...
write(1, "BEFORE MALLOC #0\n", 17BEFORE MALLOC #0
)      = 17
brk(0)                                  = 0x1314000
brk(0x1335000)                          = 0x1335000
write(1, "AFTER MALLOC #0\n", 16AFTER MALLOC #0
)       = 16
...
write(1, "0x1314010\n", 100x1314010
)             = 10
write(1, "BEFORE MALLOC #1\n", 17BEFORE MALLOC #1
)      = 17
write(1, "AFTER MALLOC #1\n", 16AFTER MALLOC #1
)       = 16
write(1, "0x1314420\n", 100x1314420
)             = 10
write(1, "BEFORE MALLOC #2\n", 17BEFORE MALLOC #2
)      = 17
write(1, "AFTER MALLOC #2\n", 16AFTER MALLOC #2
)       = 16
write(1, "0x1314830\n", 100x1314830
)             = 10
write(1, "BEFORE MALLOC #3\n", 17BEFORE MALLOC #3
)      = 17
write(1, "AFTER MALLOC #3\n", 16AFTER MALLOC #3
)       = 16
write(1, "0x1314c40\n", 100x1314c40
)             = 10
...
read(0,
```

->不是每次调用malloc，都会调用到brk。

首次调用malloc时，malloc会为程序创建一个新空间（堆）（通过增加程序间断点的位置）。之后调用malloc，malloc会使用相同的空间为我们的程序提供"新的"内存块。那些"新的"内存块是以前使用brk分配的内存的一部分。这样，malloc就不必每次调用它时都使用系统调用（brk），运行速度更快。它还允许malloc和free优化内存的使用。

让我们确保只有唯一的一个堆，由首次调用brk分配：

```
julien@holberton:/proc/4014$ ps aux | grep \ \./4$
julien     4169  0.0  0.0   4748   688 pts/9    S+   13:33   0:00 strace ./4
julien     4172  0.0  0.0   4336   656 pts/9    S+   13:33   0:00 ./4
julien@holberton:/proc/4014$ cd /proc/4172
julien@holberton:/proc/4172$ cat maps
00400000-00401000 r-xp 00000000 08:01 176973                             /home/julien/holberton/w/hack_the_virtual_memory/03.
00600000-00601000 r--p 00000000 08:01 176973                             /home/julien/holberton/w/hack_the_virtual_memory/03.
```

```
00601000-00602000 rw-p 00001000 08:01 176973                             /home/julien/holberton/w/hack_the_virtual_memory/03.
01314000-01335000 rw-p 00000000 00:00 0                                  [heap]
7f4a3f2c4000-7f4a3f47e000 r-xp 00000000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f4a3f47e000-7f4a3f67e000 ---p 001ba000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f4a3f67e000-7f4a3f682000 r--p 001ba000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f4a3f682000-7f4a3f684000 rw-p 001be000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f4a3f684000-7f4a3f689000 rw-p 00000000 00:00 0
7f4a3f689000-7f4a3f6ac000 r-xp 00000000 08:01 136229                     /lib/x86_64-linux-gnu/ld-2.19.so
7f4a3f890000-7f4a3f893000 rw-p 00000000 00:00 0
7f4a3f8a7000-7f4a3f8ab000 rw-p 00000000 00:00 0
7f4a3f8ab000-7f4a3f8ac000 r--p 00022000 08:01 136229                     /lib/x86_64-linux-gnu/ld-2.19.so
7f4a3f8ac000-7f4a3f8ad000 rw-p 00023000 08:01 136229                     /lib/x86_64-linux-gnu/ld-2.19.so
7f4a3f8ad000-7f4a3f8ae000 rw-p 00000000 00:00 0
7ffd1ba73000-7ffd1ba94000 rw-p 00000000 00:00 0                          [stack]
7ffd1bbed000-7ffd1bbef000 r--p 00000000 00:00 0                          [vvar]
7ffd1bbef000-7ffd1bbf1000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
julien@holberton:/proc/4172$
```

->我们只有一个[heap]，地址符合sbrk返回的地址：0x1314000和0x1335000

简易malloc实现

基于前面所述，并假设我们不需要释放任何内存，现在可以编写一个简易的malloc，它将在每次调用时移动程序间断点。

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

/**
* malloc - naive version of malloc: dynamically allocates memory on the heap using sbrk
* @size: number of bytes to allocate
*
* Return: the memory address newly allocated, or NULL on error
*
* Note: don't do this at home :)
*/
void *malloc(size_t size)
{
   void *previous_break;

   previous_break = sbrk(size);
   /* check for error */
   if (previous_break == (void *) -1)
   {
       /* on error malloc returns NULL */
       return (NULL);
   }
   return (previous_break);
}
```

消失的0x10字节

如果我们查看前一个程序（4-main.c）的输出，将会发现首次调用malloc返回的地址不在堆的最开始处，而是在0x10字节之后：0x1314010 VS 0x1314000。另外，当我们第二次调用malloc（1024）时，返回地址应该是0x1314010（首次调用malloc的返回值）+ 1024（或十六进制的0x400，因为首次调用malloc请求分配1024字节）=0x1318010。 但第二次调用malloc的返回值是0x1314420。
我们又丢了0x10字节！后续调用malloc也是如此。
让我们看一下那些"丢失的"0x10字节内存空间中的内容以及其是否保持不变（5-main.c）：

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

/**
* pmem - print mem
* @p: memory address to start printing from
* @bytes: number of bytes to print
*
* Return: nothing
*/
```

```c
void pmem(void *p, unsigned int bytes)
{
   unsigned char *ptr;
   unsigned int i;

   ptr = (unsigned char *)p;
   for (i = 0; i < bytes; i++)
   {
       if (i != 0)
       {
           printf(" ");
       }
       printf("%02x", *(ptr + i));
   }
   printf("\n");
}

/**
* main - the 0x10 lost bytes
*
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
*/
int main(void)
{
   void *p;
   int i;

   for (i = 0; i < 10; i++)
   {
       p = malloc(1024 * (i + 1));
       printf("%p\n", p);
       printf("bytes at %p:\n", (void *)((char *)p - 0x10));
       pmem((char *)p - 0x10, 0x10);
   }
   return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 5-main.c -o 5
julien@holberton:~/holberton/w/hackthevm3$ ./5
0x1fa8010
bytes at 0x1fa8000:
00 00 00 00 00 00 00 00 11 04 00 00 00 00 00 00
0x1fa8420
bytes at 0x1fa8410:
00 00 00 00 00 00 00 00 11 08 00 00 00 00 00 00
0x1fa8c30
bytes at 0x1fa8c20:
00 00 00 00 00 00 00 00 11 0c 00 00 00 00 00 00
0x1fa9840
bytes at 0x1fa9830:
00 00 00 00 00 00 00 00 11 10 00 00 00 00 00 00
0x1faa850
bytes at 0x1faa840:
00 00 00 00 00 00 00 00 11 14 00 00 00 00 00 00
0x1fabc60
bytes at 0x1fabc50:
00 00 00 00 00 00 00 00 11 18 00 00 00 00 00 00
0x1fad470
bytes at 0x1fad460:
00 00 00 00 00 00 00 00 11 1c 00 00 00 00 00 00
0x1faf080
bytes at 0x1faf070:
00 00 00 00 00 00 00 00 11 20 00 00 00 00 00 00
0x1fb1090
bytes at 0x1fb1080:
00 00 00 00 00 00 00 00 11 24 00 00 00 00 00 00
0x1fb34a0
bytes at 0x1fb3490:
00 00 00 00 00 00 00 00 11 28 00 00 00 00 00 00
```

```
julien@holberton:~/holberton/w/hackthevm3$
```

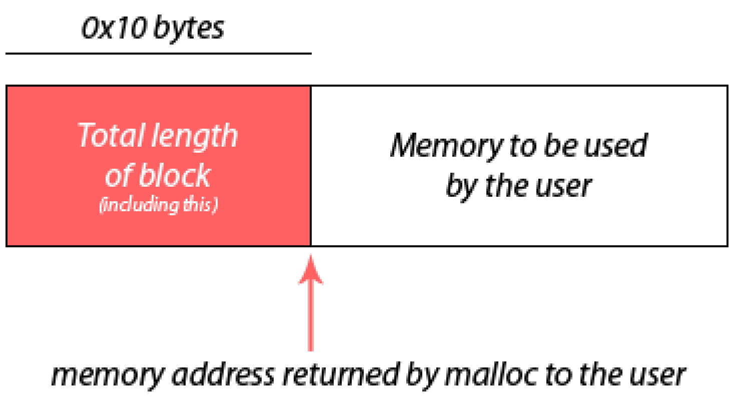
一个明显的特点：前面的0x10字节包含malloc分配的内存块大小。例如，首个malloc调用是分配1024（0x0400）个字节，我们可以在前面的0x10字节中找到11 04 00 00 00 00 00 00。后8个字节代表数字0x 00 00 00 00 00 00 04 11 = 0x400（1024）+ 0x10（1024字节之前的块大小+ 1（我们将在本章后面讨论这个“+1”）。如果查看malloc每个返回地址前面的0x10字节，会发现它们都包含请求malloc分配的内存大小 + 0x10 + 1。

在这一点上，结合我们之前所说的，我们可以猜测那些0x10字节是malloc（和free）用来处理堆的一种数据结构。实际上，即使我们还未了解所有内容，只要我们有堆的起

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

/**
* pmem - print mem
* @p: memory address to start printing from
* @bytes: number of bytes to print
*
* Return: nothing
*/
void pmem(void *p, unsigned int bytes)
{
   unsigned char *ptr;
   unsigned int i;

   ptr = (unsigned char *)p;
   for (i = 0; i < bytes; i++)
   {
       if (i != 0)
       {
           printf(" ");
       }
       printf("%02x", *(ptr + i));
   }
   printf("\n");
}

/**
* main - using the 0x10 bytes to jump to next malloc'ed chunks
*
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
*/
int main(void)
{
   void *p;
   int i;
   void *heap_start;
   size_t size_of_the_block;

   heap_start = sbrk(0);
   write(1, "START\n", 6);
   for (i = 0; i < 10; i++)
   {
       p = malloc(1024 * (i + 1));
       *((int *)p) = i;
       printf("%p: [%i]\n", p, i);
   }
   p = heap_start;
   for (i = 0; i < 10; i++)
   {
       pmem(p, 0x10);
       size_of_the_block = *((size_t *)((char *)p + 8)) - 1;
       printf("%p: [%i] - size = %lu\n",
             (void *)((char *)p + 0x10),
             *((int *)((char *)p + 0x10)),
             size_of_the_block);
       p = (void *)((char *)p + size_of_the_block);
   }
   write(1, "END\n", 4);
   return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 6-main.c -o 6
julien@holberton:~/holberton/w/hackthevm3$ ./6
START
0x9e6010: [0]
0x9e6420: [1]
0x9e6c30: [2]
0x9e7840: [3]
0x9e8850: [4]
0x9e9c60: [5]
0x9eb470: [6]
0x9ed080: [7]
0x9ef090: [8]
0x9f14a0: [9]
00 00 00 00 00 00 00 00 11 04 00 00 00 00 00 00
0x9e6010: [0] - size = 1040
00 00 00 00 00 00 00 00 11 08 00 00 00 00 00 00
0x9e6420: [1] - size = 2064
00 00 00 00 00 00 00 00 11 0c 00 00 00 00 00 00
0x9e6c30: [2] - size = 3088
00 00 00 00 00 00 00 00 11 10 00 00 00 00 00 00
0x9e7840: [3] - size = 4112
00 00 00 00 00 00 00 00 11 14 00 00 00 00 00 00
0x9e8850: [4] - size = 5136
00 00 00 00 00 00 00 00 11 18 00 00 00 00 00 00
0x9e9c60: [5] - size = 6160
00 00 00 00 00 00 00 00 11 1c 00 00 00 00 00 00
0x9eb470: [6] - size = 7184
00 00 00 00 00 00 00 00 11 20 00 00 00 00 00 00
0x9ed080: [7] - size = 8208
00 00 00 00 00 00 00 00 11 24 00 00 00 00 00 00
0x9ef090: [8] - size = 9232
00 00 00 00 00 00 00 00 11 28 00 00 00 00 00 00
0x9f14a0: [9] - size = 10256
END
julien@holberton:~/holberton/w/hackthevm3$
```

现在回答前一章中的一个开放性问题：malloc使用0x10个额外的字节来为每个分配的内存块存储块的大小。


0x10 bytes
Total length of block (including this)
Memory to be used by the user
memory address returned by malloc to the user

实际上，这些数据将被free用于将其保存到可用块列表中，以便将来调用malloc时使用。

但是我们的研究也提出了一个新问题：16（0x10）个字节的前8个字节有什么用？它似乎总是零。它只是填充？

RTFSC

在这个阶段，我们可能想查看malloc的源代码以确认我们刚刚找到的东西（malloc.c from the glibc）。

```
/*
  malloc_chunk details:

   (The following includes lightly edited explanations by Colin Plumb.)
```

```

    Chunks of memory are maintained using a `boundary tag' method as
    described in e.g., Knuth or Standish.  (See the paper by Paul
    Wilson ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps for a
    survey of such techniques.)  Sizes of free chunks are stored both
    in the front of each chunk and at the end.  This makes
    consolidating fragmented chunks into bigger chunks very fast.  The
    size fields also hold bits representing whether chunks are free or
    in use.

    An allocated chunk looks like this:


    chunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of previous chunk, if unallocated (P clear)  |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of chunk, in bytes                     |A|M|P|
      mem-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             User data starts here...                          .
            .                                                               .
            .             (malloc_usable_size() bytes)                      .
            .                                                               |
nextchunk-> +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             (size of chunk, but used for application data)    |
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
            |             Size of next chunk, in bytes                |A|0|1|
            +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

    Where "chunk" is the front of the chunk for the purpose of most of
    the malloc code, but "mem" is the pointer that is returned to the
    user.  "Nextchunk" is the beginning of the next contiguous chunk.
```

->我们是正确的\o/。malloc返回给用户的地址之前的16个字节中，有两个变量：

- 上一个块的大小（如果上一个块未分配）：我们并没有释放任何块，所以这就是它始终为0的原因
- 块的大小，以字节为单位

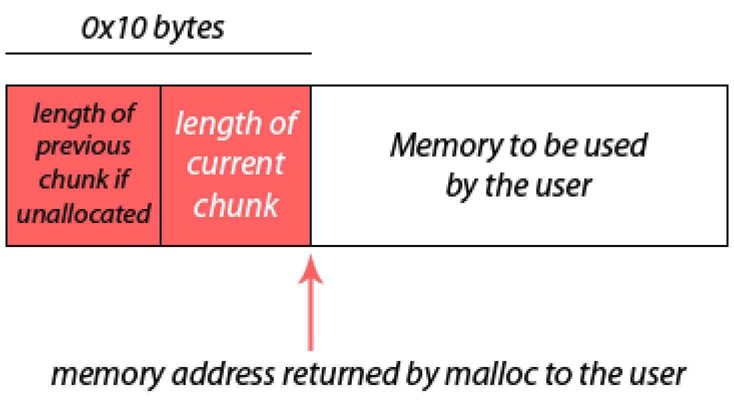
让我们释放一些块以确认前8个字节的使用方式与源代码中描述的方式相同（7-main.c）：

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

/**
* pmem - print mem
* @p: memory address to start printing from
* @bytes: number of bytes to print
*
* Return: nothing
*/
void pmem(void *p, unsigned int bytes)
{
   unsigned char *ptr;
   unsigned int i;

   ptr = (unsigned char *)p;
   for (i = 0; i < bytes; i++)
   {
       if (i != 0)
       {
           printf(" ");
       }
       printf("%02x", *(ptr + i));
   }
   printf("\n");
}

/**
* main - confirm the source code
*
```

```
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
*/
int main(void)
{
   void *p;
   int i;
   size_t size_of_the_chunk;
   size_t size_of_the_previous_chunk;
   void *chunks[10];

   for (i = 0; i < 10; i++)
   {
       p = malloc(1024 * (i + 1));
       chunks[i] = (void *)((char *)p - 0x10);
       printf("%p\n", p);
   }
   free((char *)(chunks[3]) + 0x10);
   free((char *)(chunks[7]) + 0x10);
   for (i = 0; i < 10; i++)
   {
       p = chunks[i];
       printf("chunks[%d]: ", i);
       pmem(p, 0x10);
       size_of_the_chunk = *((size_t *)((char *)p + 8)) - 1;
       size_of_the_previous_chunk = *((size_t *)((char *)p));
       printf("chunks[%d]: %p, size = %li, prev = %li\n",
             i, p, size_of_the_chunk, size_of_the_previous_chunk);
   }
   return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 7-main.c -o 7
julien@holberton:~/holberton/w/hackthevm3$ ./7
0x1536010
0x1536420
0x1536c30
0x1537840
0x1538850
0x1539c60
0x153b470
0x153d080
0x153f090
0x15414a0
chunks[0]: 00 00 00 00 00 00 00 00 11 04 00 00 00 00 00 00
chunks[0]: 0x1536000, size = 1040, prev = 0
chunks[1]: 00 00 00 00 00 00 00 00 11 08 00 00 00 00 00 00
chunks[1]: 0x1536410, size = 2064, prev = 0
chunks[2]: 00 00 00 00 00 00 00 00 11 0c 00 00 00 00 00 00
chunks[2]: 0x1536c20, size = 3088, prev = 0
chunks[3]: 00 00 00 00 00 00 00 00 11 10 00 00 00 00 00 00
chunks[3]: 0x1537830, size = 4112, prev = 0
chunks[4]: 10 10 00 00 00 00 00 00 10 14 00 00 00 00 00 00
chunks[4]: 0x1538840, size = 5135, prev = 4112
chunks[5]: 00 00 00 00 00 00 00 00 11 18 00 00 00 00 00 00
chunks[5]: 0x1539c50, size = 6160, prev = 0
chunks[6]: 00 00 00 00 00 00 00 00 11 1c 00 00 00 00 00 00
chunks[6]: 0x153b460, size = 7184, prev = 0
chunks[7]: 00 00 00 00 00 00 00 00 11 20 00 00 00 00 00 00
chunks[7]: 0x153d070, size = 8208, prev = 0
chunks[8]: 10 20 00 00 00 00 00 00 10 24 00 00 00 00 00 00
chunks[8]: 0x153f080, size = 9231, prev = 8208
chunks[9]: 00 00 00 00 00 00 00 00 11 28 00 00 00 00 00 00
chunks[9]: 0x1541490, size = 10256, prev = 0
julien@holberton:~/holberton/w/hackthevm3$
```

正如我们从上面的列表中看到的，当前一个块被释放时，已分配块的前8个字节包含前一个未分配块的大小。所以malloc块的正确表示如下：


0x10 bytes
length of previous chunk if unallocated
length of current chunk
Memory to be used by the user
memory address returned by malloc to the user

此外，似乎后8个字节（包含当前块的大小）的第一位用作检查前一个块是否被使用（1）或未被使用（0）的标志。因此，我们程序正确更新后的版本（8-main.c）：

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

/**
* pmem - print mem
* @p: memory address to start printing from
* @bytes: number of bytes to print
*
* Return: nothing
*/
void pmem(void *p, unsigned int bytes)
{
   unsigned char *ptr;
   unsigned int i;

   ptr = (unsigned char *)p;
   for (i = 0; i < bytes; i++)
   {
       if (i != 0)
       {
           printf(" ");
       }
       printf("%02x", *(ptr + i));
   }
   printf("\n");
}

/**
* main - updating with correct checks
*
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
*/
int main(void)
{
   void *p;
   int i;
   size_t size_of_the_chunk;
   size_t size_of_the_previous_chunk;
   void *chunks[10];
   char prev_used;

   for (i = 0; i < 10; i++)
   {
       p = malloc(1024 * (i + 1));
       chunks[i] = (void *)((char *)p - 0x10);
   }
   free((char *)(chunks[3]) + 0x10);
   free((char *)(chunks[7]) + 0x10);
   for (i = 0; i < 10; i++)
   {
       p = chunks[i];
       printf("chunks[%d]: ", i);
       pmem(p, 0x10);
       size_of_the_chunk = *((size_t *)((char *)p + 8));
       prev_used = size_of_the_chunk & 1;
       size_of_the_chunk -= prev_used;
       size_of_the_previous_chunk = *((size_t *)((char *)p));
       printf("chunks[%d]: %p, size = %li, prev (%s) = %li\n",
             i, p, size_of_the_chunk,
             (prev_used? "allocated": "unallocated"), size_of_the_previous_chunk);
   }
   return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 8-main.c -o 8
julien@holberton:~/holberton/w/hackthevm3$ ./8
chunks[0]: 00 00 00 00 00 00 00 00 11 04 00 00 00 00 00 00
chunks[0]: 0x1031000, size = 1040, prev (allocated) = 0
chunks[1]: 00 00 00 00 00 00 00 00 11 08 00 00 00 00 00 00
chunks[1]: 0x1031410, size = 2064, prev (allocated) = 0
chunks[2]: 00 00 00 00 00 00 00 00 11 0c 00 00 00 00 00 00
chunks[2]: 0x1031c20, size = 3088, prev (allocated) = 0
chunks[3]: 00 00 00 00 00 00 00 00 11 10 00 00 00 00 00 00
chunks[3]: 0x1032830, size = 4112, prev (allocated) = 0
chunks[4]: 10 10 00 00 00 00 00 00 10 14 00 00 00 00 00 00
chunks[4]: 0x1033840, size = 5136, prev (unallocated) = 4112
chunks[5]: 00 00 00 00 00 00 00 00 11 18 00 00 00 00 00 00
chunks[5]: 0x1034c50, size = 6160, prev (allocated) = 0
chunks[6]: 00 00 00 00 00 00 00 00 11 1c 00 00 00 00 00 00
chunks[6]: 0x1036460, size = 7184, prev (allocated) = 0
chunks[7]: 00 00 00 00 00 00 00 00 11 20 00 00 00 00 00 00
chunks[7]: 0x1038070, size = 8208, prev (allocated) = 0
chunks[8]: 10 20 00 00 00 00 00 00 10 24 00 00 00 00 00 00
chunks[8]: 0x103a080, size = 9232, prev (unallocated) = 8208
chunks[9]: 00 00 00 00 00 00 00 00 11 28 00 00 00 00 00 00
chunks[9]: 0x103c490, size = 10256, prev (allocated) = 0
julien@holberton:~/holberton/w/hackthevm3$
```

堆真的在向上增长吗？

没有回答的最后一个问题是：“堆真的在向上增长吗？”。 从brk手册看来，似乎如此：

```
DESCRIPTION
      brk() and sbrk() change the location of the program break, which defines the end  of  the
      process's  data  segment  (i.e., the program break is the first location after the end of
      the uninitialized data segment).  Increasing the program break has the effect of allocat■
      ing memory to the process; decreasing the break deallocates memory.
```

检查一下！(9-main.c)

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

/**
* main - moving the program break
*
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
*/
int main(void)
{
   int i;

   write(1, "START\n", 6);
   malloc(1);
   getchar();
   write(1, "LOOP\n", 5);
   for (i = 0; i < 0x25000 / 1024; i++)
   {
       malloc(1024);
   }
   write(1, "END\n", 4);
   getchar();
   return (EXIT_SUCCESS);
}
```

现在使用strace确认这个假设：

```
julien@holberton:~/holberton/w/hackthevm3$ strace ./9
execve("./9", ["./9"], [/* 61 vars */]) = 0
...
write(1, "START\n", 6START
)                  = 6
brk(0)                                  = 0x1fd8000
brk(0x1ff9000)                          = 0x1ff9000
...
write(1, "LOOP\n", 5LOOP
)                   = 5
brk(0x201a000)                          = 0x201a000
write(1, "END\n", 4END
)                    = 4
...
julien@holberton:~/holberton/w/hackthevm3$
```

显然，malloc只调用了brk两次，以增加堆上的分配空间。第二次调用使用了更高的内存地址参数（0x201a000>0x1ff9000）。第二次brk系统调用发生于堆上的空间太小而

让我们用/proc再次确认。

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 9-main.c -o 9
julien@holberton:~/holberton/w/hackthevm3$ ./9
START

julien@holberton:/proc/7855$ ps aux | grep \ \./9$
julien     7972  0.0  0.0   4332   684 pts/9    S+   19:08   0:00 ./9
julien@holberton:/proc/7855$ cd /proc/7972
julien@holberton:/proc/7972$ cat maps
...
00901000-00922000 rw-p 00000000 00:00 0                                  [heap]
...
julien@holberton:/proc/7972$
```

-> 00901000-00922000 rw-p 00000000 00:00 0 [heap]
点击回车并再次查看[heap]：

```
LOOP
END

julien@holberton:/proc/7972$ cat maps
...
00901000-00943000 rw-p 00000000 00:00 0                                  [heap]
...
julien@holberton:/proc/7972$
```

-> 00901000-00943000 rw-p 00000000 00:00 0 [heap]
堆的起始地址仍然相同，但是大小从0x00922000向上增加到0x00943000。

地址空间布局随机化（ASLR）

你可能已经注意到上面/proc/pid/maps中的一些我们想要研究的“奇怪”的东西：

程序间断点是超出数据段当前末尾的第一个位置的地址 ——
也就是虚拟内存中可执行文件之外的第一个位置的地址。因此，堆的起始处应该紧挨着内存中可执行文件的末尾。但正如你在上面所看到的那样，情况并非如此。唯一正确的

*下面几行的格式：[maps文件的PID]：[heap]开头的地址 - 可执行文件结束的地址=内存间隙大小

- [3718]: 01195000 – 00602000 = b93000

- [3834]: 024d6000 – 00602000 = 1ed4000
- [4014]: 00e70000 – 00602000 = 86e000
- [4172]: 01314000 – 00602000 = d12000
- [7972]: 00901000 – 00602000 = 2ff000

似乎这个间隙大小是随机的，事实上，确实如此。如果我们查看ELF二进制加载器源代码（fs/binfmt_elf.c），我们可以找到这个：

```
if ((current->flags & PF_RANDOMIZE) && (randomize_va_space > 1)) {
               current->mm->brk = current->mm->start_brk =
                       arch_randomize_brk(current->mm);
#ifdef compat_brk_randomized
               current->brk_randomized = 1;
#endif
       }
```

其中current->mm->brk是程序间断点的地址。arch_randomize_brk函数可以在arch/x86/kernel/process.c文件中找到：

```
unsigned long arch_randomize_brk(struct mm_struct *mm)
{
       unsigned long range_end = mm->brk + 0x02000000;
       return randomize_range(mm->brk, range_end, 0) ? : mm->brk;
}
```

randomize_range返回一个起始地址，像这样：

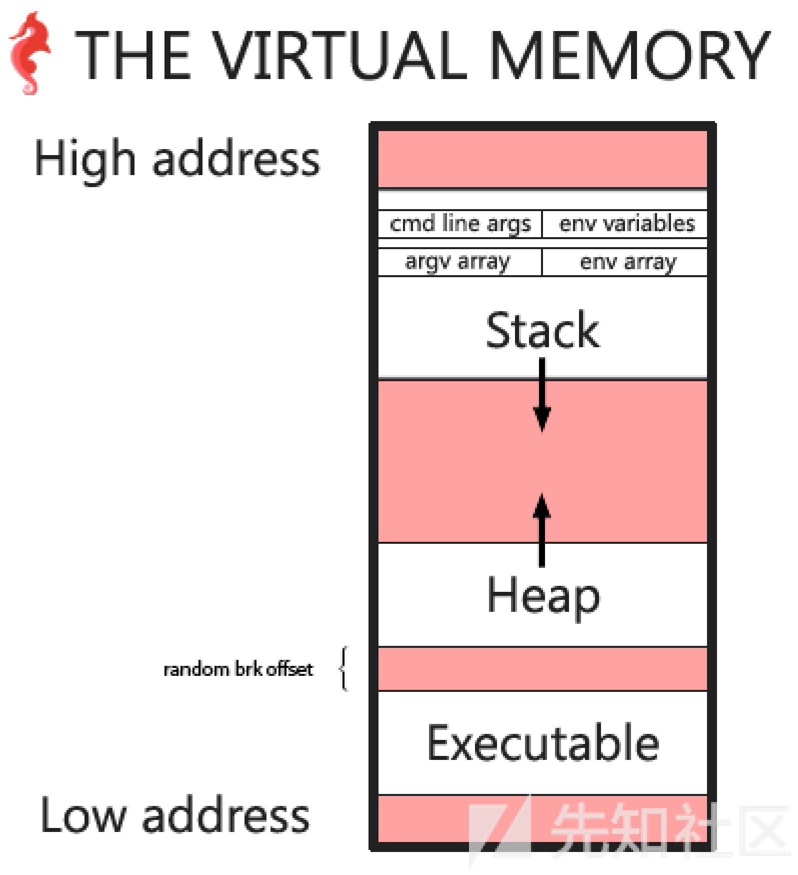
```
[...... <range> .....]
 start                  end
```

randomize_range函数的源代码（drivers/char/random.c）：

```
/*
* randomize_range() returns a start address such that
*
*    [...... <range> .....]
*  start                  end
*
* a <range> with size "len" starting at the return value is inside in the
* area defined by [start, end], but is otherwise randomized.
*/
unsigned long
randomize_range(unsigned long start, unsigned long end, unsigned long len)
{
       unsigned long range = end - len - start;

       if (end <= start + len)
               return 0;
       return PAGE_ALIGN(get_random_int() % range + start);
}
```

因此，当进程运行时，可执行文件的数据部分与程序间断点初始位置之间的偏移量可以是0到0x02000000之间的任何值。这种随机化称为地址空间布局随机化（ASLR）。A

更新后的内存图



malloc(0)

你有没有想过当我们调用malloc(0)时会发生什么？（10-main.c）

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

/**
* pmem - print mem
* @p: memory address to start printing from
* @bytes: number of bytes to print
*
* Return: nothing
*/
void pmem(void *p, unsigned int bytes)
{
   unsigned char *ptr;
   unsigned int i;

   ptr = (unsigned char *)p;
   for (i = 0; i < bytes; i++)
   {
       if (i != 0)
       {
           printf(" ");
       }
       printf("%02x", *(ptr + i));
   }
   printf("\n");
}

/**
* main - moving the program break
*
* Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
*/
int main(void)
```

```
{
   void *p;
   size_t size_of_the_chunk;
   char prev_used;

   p = malloc(0);
   printf("%p\n", p);
   pmem((char *)p - 0x10, 0x10);
   size_of_the_chunk = *((size_t *)((char *)p - 8));
   prev_used = size_of_the_chunk & 1;
   size_of_the_chunk -= prev_used;
   printf("chunk size = %li bytes\n", size_of_the_chunk);
   return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 10-main.c -o 10
julien@holberton:~/holberton/w/hackthevm3$ ./10
0xd08010
00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
chunk size = 32 bytes
julien@holberton:~/holberton/w/hackthevm3$
```

-> malloc(0)实际上分配32个字节，包括之前的0x10字节。
注意，情况并非总是如此。来自手册页（man malloc）：

```
NULL may also be returned by a successful call to malloc() with a size of zero
```