Hack 虚拟内存系列（二）：Python 字节对象

本文翻译自：Hack The Virtual Memory: Python bytes

Hack The Virtual Memory: Python bytes

Hack虚拟内存之第1章：Python 字节对象

在这篇文章中，我们将完成与第0章（C字符串和/proc）几乎相同的事情，但我们将访问正在运行的Python 3脚本的虚拟内存。它不会（像第0章中）那么直接。并在此过程中看看Python 3的一些内部结构！

前提

本文基于我们在前一章中学到的所有内容。在阅读本文之前，请阅读（并理解）第0章：C字符串和/proc。
为了完全理解本文，你需要知道：

- C语言的基础知识
- Python的基础知识
- Linux文件系统和shell的基础知识
- /proc文件系统的基础知识（参见第0章：C字符串和/proc，了解本主题的介绍）

环境

所有脚本和程序都已经在以下系统上进行过测试：

- Ubuntu 14.04 LTS
  - 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
- gcc
  - gcc (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4
- Python 3
  - Python 3.4.3 (default, Nov 17 2016, 01:08:31)
  - [GCC 4.8.4] on linux

Python脚本

一开始我们将使用此脚本（main.py）并尝试在运行它的进程的虚拟内存中修改字符串“Holberton”。

```
#!/usr/bin/env python3
'''
Prints a b"string" (bytes object), reads a char from stdin
and prints the same (or not :)) string again
'''

import sys

s = b"Holberton"
print(s)
sys.stdin.read(1)
print(s)
```

关于字节对象

bytes vs str

如你所见，我们使用一个字节对象（我们在字符串前面使用‘b’前缀）来存储我们的字符串。此类型将以字节的形式存储字符串（与可能的多字节相比 - 你可以阅读unicodeobject.h以了解有关Python 3如何编码字符串的更多信息）。这可确保字符串在运行脚本的进程的虚拟内存中是一连串的ASCII码值。

实际上，s不是Python字符串（但在我们的上下文中并不重要）：

```
julien@holberton:~/holberton/w/hackthevm1$ python3
Python 3.4.3 (default, Nov 17 2016, 01:08:31)
[GCC 4.8.4] on linux
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>> s = "Betty"
>>> type(s)
<class 'str'>
>>> s = b"Betty"
>>> type(s)
<class 'bytes'>
>>> quit()
```

一切都是对象

Python中的所有东西都是一个对象：整数，字符串，字节，函数等等。因此，s = b"Holberton"应该创建一个字节类型的对象，并将字符串b"Holberton"存储在内存中。可能在堆中，因为它必须为对象所引用或存储的字节对象保留空间（此时我们不知道

对Python脚本运行read_write_heap.py

注意：read_write_heap.py是我们在上一章第0章：C字符串和/proc中写过的脚本
让我们运行上面的脚本，然后运行我们的read_write_heap.py脚本：

```
julien@holberton:~/holberton/w/hackthevm1$ ./main.py
b'Holberton'
```

此时，main.py正在等待用户输入。这对应于我们代码中的sys.stdin.read（1）。
运行read_write_heap.py：

```
julien@holberton:~/holberton/w/hackthevm1$ ps aux | grep main.py | grep -v grep
julien     3929  0.0  0.7  31412  7848 pts/0    S+   15:10   0:00 python3 ./main.py
julien@holberton:~/holberton/w/hackthevm1$ sudo ./read_write_heap.py 3929 Holberton "~ Betty ~"
[*] maps: /proc/3929/maps
[*] mem: /proc/3929/mem
[*] Found [heap]:
   pathname = [heap]
   addresses = 022dc000-023c6000
   permisions = rw-p
   offset = 00000000
   inode = 0
   Addr start [22dc000] | end [23c6000]
[*] Found 'Holberton' at 8e192
[*] Writing '~ Betty ~' at 236a192
julien@holberton:~/holberton/w/hackthevm1$
```

正如所料，我们在堆上找到了字符串("Holberton")并替换了它。现在，当我们在main.py脚本中按下Enter键时，它将打印b'~Betty~'：

```
b'Holberton'
julien@holberton:~/holberton/w/hackthevm1$
```

等一下

我们找到字符串"Holberton"并替换它，但并没有输出正确的字符串？
在我们深入思考前，还有一件事需要检查。我们的脚本在找到第一个字符串后停止。让我们运行几次，看看堆中是否有更多相同的字符串。

```
julien@holberton:~/holberton/w/hackthevm1$ ./main.py
b'Holberton'

julien@holberton:~/holberton/w/hackthevm1$ ps aux | grep main.py | grep -v grep
julien     4051  0.1  0.7  31412  7832 pts/0    S+   15:53   0:00 python3 ./main.py
julien@holberton:~/holberton/w/hackthevm1$ sudo ./read_write_heap.py 4051 Holberton "~ Betty ~"
[*] maps: /proc/4051/maps
[*] mem: /proc/4051/mem
[*] Found [heap]:
   pathname = [heap]
   addresses = 00bf4000-00cde000
   permisions = rw-p
   offset = 00000000
   inode = 0
   Addr start [bf4000] | end [cde000]
[*] Found 'Holberton' at 8e162
[*] Writing '~ Betty ~' at c82162
julien@holberton:~/holberton/w/hackthevm1$ sudo ./read_write_heap.py 4051 Holberton "~ Betty ~"
[*] maps: /proc/4051/maps
```

```
[*] mem: /proc/4051/mem
[*] Found [heap]:
   pathname = [heap]
   addresses = 00bf4000-00cde000
   permisions = rw-p
   offset = 00000000
   inode = 0
   Addr start [bf4000] | end [cde000]
Can't find 'Holberton'
julien@holberton:~/holberton/w/hackthevm1$
```

只出现一次。那么脚本中使用的字符串“Holberton”在哪里？Python字节对象在内存中的哪个位置？它可能在栈中吗？在read_write_heap.py脚本中用“[stack]”来替换“[hea
（*）参见上一篇文章，栈在/proc/[pid]/maps文件中被称为“[stack]”

```
#!/usr/bin/env python3
'''
Locates and replaces the first occurrence of a string in the stack
of a process

Usage: ./read_write_stack.py PID search_string replace_by_string
Where:
- PID is the pid of the target process
- search_string is the ASCII string you are looking to overwrite
- replace_by_string is the ASCII string you want to replace
search_string with
'''

import sys

def print_usage_and_exit():
   print('Usage: {} pid search write'.format(sys.argv[0]))
   sys.exit(1)

# check usage
if len(sys.argv) != 4:
   print_usage_and_exit()

# get the pid from args
pid = int(sys.argv[1])
if pid <= 0:
   print_usage_and_exit()
search_string = str(sys.argv[2])
if search_string  == "":
   print_usage_and_exit()
write_string = str(sys.argv[3])
if search_string  == "":
   print_usage_and_exit()

# open the maps and mem files of the process
maps_filename = "/proc/{}/maps".format(pid)
print("[*] maps: {}".format(maps_filename))
mem_filename = "/proc/{}/mem".format(pid)
print("[*] mem: {}".format(mem_filename))

# try opening the maps file
try:
   maps_file = open('/proc/{}/maps'.format(pid), 'r')
except IOError as e:
   print("[ERROR] Can not open file {}:".format(maps_filename))
   print("        I/O error({}): {}".format(e.errno, e.strerror))
   sys.exit(1)

for line in maps_file:
   sline = line.split(' ')
   # check if we found the stack
   if sline[-1][:-1] != "[stack]":
       continue
   print("[*] Found [stack]:")
```

```
    # parse line
    addr = sline[0]
    perm = sline[1]
    offset = sline[2]
    device = sline[3]
    inode = sline[4]
    pathname = sline[-1][:-1]
    print("\tpathname = {}".format(pathname))
    print("\taddresses = {}".format(addr))
    print("\tpermisions = {}".format(perm))
    print("\toffset = {}".format(offset))
    print("\tinode = {}".format(inode))

    # check if there is read and write permission
    if perm[0] != 'r' or perm[1] != 'w':
        print("[*] {} does not have read/write permission".format(pathname))
        maps_file.close()
        exit(0)

    # get start and end of the stack in the virtual memory
    addr = addr.split("-")
    if len(addr) != 2: # never trust anyone, not even your OS :)
        print("[*] Wrong addr format")
        maps_file.close()
        exit(1)
    addr_start = int(addr[0], 16)
    addr_end = int(addr[1], 16)
    print("\tAddr start [{:x}] | end [{:x}]".format(addr_start, addr_end))

    # open and read mem
    try:
        mem_file = open(mem_filename, 'rb+')
    except IOError as e:
        print("[ERROR] Can not open file {}:".format(mem_filename))
        print("        I/O error({}): {}".format(e.errno, e.strerror))
        maps_file.close()
        exit(1)

    # read stack
    mem_file.seek(addr_start)
    stack = mem_file.read(addr_end - addr_start)

    # find string
    try:
        i = stack.index(bytes(search_string, "ASCII"))
    except Exception:
        print("Can't find '{}'".format(search_string))
        maps_file.close()
        mem_file.close()
        exit(0)
    print("[*] Found '{}' at {:x}".format(search_string, i))

    # write the new stringprint("[*] Writing '{}' at {:x}".format(write_string, addr_start + i))
    mem_file.seek(addr_start + i)
    mem_file.write(bytes(write_string, "ASCII"))

    # close filesmaps_file.close()
    mem_file.close()

    # there is only one stack in our example
    break
```

上面的脚本（read_write_stack.py）与前一个脚本（read_write_heap.py）做的事完全相同。除了我们正在查找栈，而不是堆。让我们尝试在栈中找到字符串：

```
julien@holberton:~/holberton/w/hackthevm1$ ./main.py
b'Holberton'

julien@holberton:~/holberton/w/hackthevm1$ ps aux | grep main.py | grep -v grep
julien     4124  0.2  0.7  31412  7848 pts/0    S+   16:10   0:00 python3 ./main.py
julien@holberton:~/holberton/w/hackthevm1$ sudo ./read_write_stack.py 4124 Holberton "~ Betty ~"
```

```
[sudo] password for julien:
[*] maps: /proc/4124/maps
[*] mem: /proc/4124/mem
[*] Found [stack]:
   pathname = [stack]
   addresses = 7fff2997e000-7fff2999f000
   permisions = rw-p
   offset = 00000000
   inode = 0
   Addr start [7fff2997e000] | end [7fff2999f000]
Can't find 'Holberton'
julien@holberton:~/holberton/w/hackthevm1$
```

所以我们的字符串既不在堆中也不在栈中：那么，它在哪里？是时候深入研究Python 3内部结构并使用我们将要学习的知识来找到字符串。打起精神，乐趣才刚刚开始

在虚拟内存中定位字符串

注意：Python 3有许多实现。但在本文中，我们使用原始而最常用的：CPython（用C编码）。下面我们将要讨论的python 3是基于此实现的

id

有一种简单的方法可以知道对象（注意，是对象，不是字符串）在虚拟内存中的位置。
CPython有一个内置id（）的特定实现：id（）将返回对象在内存中的地址。

如果我们在Python脚本中添加一行打印对象id的语句，我们应该能得到它的地址（main_id.py）：

```
#!/usr/bin/env python3
'''
Prints:
- the address of the bytes object
- a b"string" (bytes object)
reads a char from stdin
and prints the same (or not :)) string again
'''

import sys

s = b"Holberton"
print(hex(id(s)))
print(s)
sys.stdin.read(1)
print(s)
```

```
julien@holberton:~/holberton/w/hackthevm1$ ./main_id.py
0x7f343f010210
b'Holberton'
```

->0x7f343f010210。让我们看一下/proc/来知道我们的对象所在的位置。

```
julien@holberton:/usr/include/python3.4$ ps aux | grep main_id.py | grep -v grep
julien     4344  0.0  0.7  31412  7856 pts/0    S+   16:53   0:00 python3 ./main_id.py
julien@holberton:/usr/include/python3.4$ cat /proc/4344/maps
00400000-006fa000 r-xp 00000000 08:01 655561                             /usr/bin/python3.4
008f9000-008fa000 r--p 002f9000 08:01 655561                             /usr/bin/python3.4
008fa000-00986000 rw-p 002fa000 08:01 655561                             /usr/bin/python3.4
00986000-009a2000 rw-p 00000000 00:00 0
021ba000-022a4000 rw-p 00000000 00:00 0                                  [heap]
7f343d797000-7f343de79000 r--p 00000000 08:01 663747                     /usr/lib/locale/locale-archive
7f343de79000-7f343df7e000 r-xp 00000000 08:01 136303                     /lib/x86_64-linux-gnu/libm-2.19.so
7f343df7e000-7f343e17d000 ---p 00105000 08:01 136303                     /lib/x86_64-linux-gnu/libm-2.19.so
7f343e17d000-7f343e17e000 r--p 00104000 08:01 136303                     /lib/x86_64-linux-gnu/libm-2.19.so
7f343e17e000-7f343e17f000 rw-p 00105000 08:01 136303                     /lib/x86_64-linux-gnu/libm-2.19.so
7f343e17f000-7f343e197000 r-xp 00000000 08:01 136416                     /lib/x86_64-linux-gnu/libz.so.1.2.8
7f343e197000-7f343e396000 ---p 00018000 08:01 136416                     /lib/x86_64-linux-gnu/libz.so.1.2.8
7f343e396000-7f343e397000 r--p 00017000 08:01 136416                     /lib/x86_64-linux-gnu/libz.so.1.2.8
7f343e397000-7f343e398000 rw-p 00018000 08:01 136416                     /lib/x86_64-linux-gnu/libz.so.1.2.8
7f343e398000-7f343e3bf000 r-xp 00000000 08:01 136275                     /lib/x86_64-linux-gnu/libexpat.so.1.6.0
7f343e3bf000-7f343e5bf000 ---p 00027000 08:01 136275                     /lib/x86_64-linux-gnu/libexpat.so.1.6.0
7f343e5bf000-7f343e5c1000 r--p 00027000 08:01 136275                     /lib/x86_64-linux-gnu/libexpat.so.1.6.0
```

```
7f343e5c1000-7f343e5c2000 rw-p 00029000 08:01 136275                     /lib/x86_64-linux-gnu/libexpat.so.1.6.0
7f343e5c2000-7f343e5c4000 r-xp 00000000 08:01 136408                     /lib/x86_64-linux-gnu/libutil-2.19.so
7f343e5c4000-7f343e7c3000 ---p 00002000 08:01 136408                     /lib/x86_64-linux-gnu/libutil-2.19.so
7f343e7c3000-7f343e7c4000 r--p 00001000 08:01 136408                     /lib/x86_64-linux-gnu/libutil-2.19.so
7f343e7c4000-7f343e7c5000 rw-p 00002000 08:01 136408                     /lib/x86_64-linux-gnu/libutil-2.19.so
7f343e7c5000-7f343e7c8000 r-xp 00000000 08:01 136270                     /lib/x86_64-linux-gnu/libdl-2.19.so
7f343e7c8000-7f343e9c7000 ---p 00003000 08:01 136270                     /lib/x86_64-linux-gnu/libdl-2.19.so
7f343e9c7000-7f343e9c8000 r--p 00002000 08:01 136270                     /lib/x86_64-linux-gnu/libdl-2.19.so
7f343e9c8000-7f343e9c9000 rw-p 00003000 08:01 136270                     /lib/x86_64-linux-gnu/libdl-2.19.so
7f343e9c9000-7f343eb83000 r-xp 00000000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f343eb83000-7f343ed83000 ---p 001ba000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f343ed83000-7f343ed87000 r--p 001ba000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f343ed87000-7f343ed89000 rw-p 001be000 08:01 136253                     /lib/x86_64-linux-gnu/libc-2.19.so
7f343ed89000-7f343ed8e000 rw-p 00000000 00:00 0
7f343ed8e000-7f343eda7000 r-xp 00000000 08:01 136373                     /lib/x86_64-linux-gnu/libpthread-2.19.so
7f343eda7000-7f343efa6000 ---p 00019000 08:01 136373                     /lib/x86_64-linux-gnu/libpthread-2.19.so
7f343efa6000-7f343efa7000 r--p 00018000 08:01 136373                     /lib/x86_64-linux-gnu/libpthread-2.19.so
7f343efa7000-7f343efa8000 rw-p 00019000 08:01 136373                     /lib/x86_64-linux-gnu/libpthread-2.19.so
7f343efa8000-7f343efac000 rw-p 00000000 00:00 0
7f343efac000-7f343efcf000 r-xp 00000000 08:01 136229                     /lib/x86_64-linux-gnu/ld-2.19.so
7f343f000000-7f343f1b6000 rw-p 00000000 00:00 0
7f343f1c5000-7f343f1cc000 r--s 00000000 08:01 918462                     /usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache
7f343f1cc000-7f343f1ce000 rw-p 00000000 00:00 0
7f343f1ce000-7f343f1cf000 r--p 00022000 08:01 136229                     /lib/x86_64-linux-gnu/ld-2.19.so
7f343f1cf000-7f343f1d0000 rw-p 00023000 08:01 136229                     /lib/x86_64-linux-gnu/ld-2.19.so
7f343f1d0000-7f343f1d1000 rw-p 00000000 00:00 0
7ffccf1fd000-7ffccf21e000 rw-p 00000000 00:00 0                          [stack]
7ffccf23c000-7ffccf23e000 r--p 00000000 00:00 0                          [vvar]
7ffccf23e000-7ffccf240000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
julien@holberton:/usr/include/python3.4$
```

->我们的对象存储在以下内存区域：7f343f000000-7f343f1b6000 rw -p 00000000 00:00
0，这既不是堆也不是栈。这证实了我们之前看到的情况。但这并不意味着字符串本身也存储在同一个内存区域。例如，bytes对象可以存储一个指向字符串的指针，而不是字

bytesobject.h

我们正在使用Python的C实现（CPython），所以让我们看一下字节对象的头文件。

注意：如果你没有Python 3头文件，可以在Ubuntu上使用此命令：sudo apt-get install python3-dev 在你的系统上下载它们。
如果你使用的是与我完全相同的环境（请参阅上面的"环境"部分），那么您应该能够在/usr/include/python3.4/目录中看到Python 3头文件。

来自bytesobject.h：

```
typedef struct {
   PyObject_VAR_HEAD
   Py_hash_t ob_shash;
   char ob_sval[1];

   /* Invariants:
    *     ob_sval contains space for 'ob_size+1' elements.
    *     ob_sval[ob_size] == 0.
    *     ob_shash is the hash of the string or -1 if not computed yet.
    */
} PyBytesObject;
```

从中我们可以了解到：

- Python 3字节对象在内部使用PyBytesObject类型变量表示
- ob_sval包含整个字符串
- 字符串以0结尾
- ob_size存储字符串的长度（查看objects.h中宏PyObject_VAR_HEAD的定义以找到ob_size的含义。稍后我们会看一下该文件）
  所以在我们的例子中，如果我们能够打印字节对象，我们应该会看到：
- ob_sval："Holberton" ->字节值：48 6f 6c 62 65 72 74 6f 6e 00
- ob_size：9
  根据我们之前学到的内容，这意味着字符串在字节对象的"内部"。所以在同一个内存区域内 \o/
  如果我们不知道在CPython中id的实现方式怎么办？实际上我们可以使用另一种方法来查找字符串的位置：查看内存中的实际对象。

查看内存中的字节对象

如果我们想直接查看PyBytesObject变量，我们需要创建一个C函数，并从Python调用这个C函数。有许多方法可以从Python调用C函数。我们将使用最简单的一个：使用动

创建C函数

因此，我们的想法是创建一个从Python调用的C函数，该对象作为参数，然后“探索”该对象以获得字符串的确切地址（以及有关该对象的其他信息）。

函数原型应该是：void print_python_bytes（PyObject * p）;,其中p是指向我们对象的指针（因此p存储该对象的虚拟内存地址）。它不需要返回任何东西。

object.h

你可能已经注意到我们不使用PyBytesObject类型的参数。要了解原因，让我们看一下object.h头文件，看看我们可以从中学到什么：

```
/* Object and type object interface */

/*
Objects are structures allocated on the heap.  Special rules apply to
the use of objects to ensure they are properly garbage-collected.
Objects are never allocated statically or on the stack; they must be
...
*/
```

- “Objects are never allocated statically or on the stack” (“对象永远不会静态分配或在栈上”)->好的，现在我们知道为什么它不在堆栈中。
- “Objects are structures allocated on the heap”（“对象是在堆上分配的结构”）->等一下... WAT？
  我们在堆中搜索了字符串，它不在那里.....我很困惑！我们稍后将在另一篇文章中讨论这个问题

我们还可以了解到什么：

```
/*
...
Objects do not float around in memory; once allocated an object keeps the same size and address.  Objects that must hold varia
...
*/
```

- “Objects do not float around in memory; once allocated an object keeps the same size and address”（“对象不会在内存中移动;
  一个对象一旦被分配了就会保持相同的大小和地址”）。好消息。这意味着如果我们修改该字符串，它将始终被修改，并且地址永远不会改变
- “once allocated”（“一旦分配”）->分配？但不使用堆？困惑！我们稍后将在另一篇文章中讨论这个问题

```
/*
...
Objects are always accessed through pointers of the type 'PyObject *'.
The type 'PyObject' is a structure that only contains the reference count
and the type pointer.  The actual memory allocated for an object
contains other data that can only be accessed after casting the pointer
to a pointer to a longer structure type.  This longer type must start
with the reference count and type fields; the macro PyObject_HEAD should be
used for this (to accommodate for future changes).  The implementation
of a particular object type can cast the object pointer to the proper
type and back.
...
*/
```

- “Objects are always accessed through pointers of the type ‘PyObject '”（“总是通过'PyObject
  '类型的指针访问对象”）->这就是我们必须使用PyObject类型的指针（而不是 PyBytesObject）作为我们函数的参数的原因
- “The actual memory allocated for an object contains other data that can only be accessed after casting the pointer to a pointer to a longer
  structure
  type.”（“为对象分配的实际内存包含其他数据，这些数据只能在将指针转换为指向更长结构类型的指针后才能访问。”）->所以我们必须将我们的函数参数转换为PyByte
  *才能访问它的所有内容。这是可能的，因为PyBytesObject的首部包含PyVarObject，PyVarObject本身首部包含PyObject：

```
/* PyObject_VAR_HEAD defines the initial segment of all variable-size
* container objects.  These end with a declaration of an array with 1
* element, but enough space is malloc'ed so that the array actually
* has room for ob_size elements.  Note that ob_size is an element count,
* not necessarily a byte count.
*/
#define PyObject_VAR_HEAD      PyVarObject ob_base;
#define Py_INVALID_SIZE (Py_ssize_t)-1

/* Nothing is actually declared to be a PyObject, but every pointer to
* a Python object can be cast to a PyObject*.  This is inheritance built
* by hand.  Similarly every pointer to a variable-size Python object can,
```

```
* in addition, be cast to PyVarObject*.
*/
typedef struct _object {
   _PyObject_HEAD_EXTRA
   Py_ssize_t ob_refcnt;
   struct _typeobject *ob_type;
} PyObject;

typedef struct {
   PyObject ob_base;
   Py_ssize_t ob_size; /* Number of items in variable part */
} PyVarObject;
```

->PyVarObject里含有bytesobject.h提到的ob_size。
C函数
基于我们刚刚学到的所有东西，C代码非常简单（bytes.c）：

```
#include "Python.h"

/**
* print_python_bytes - prints info about a Python 3 bytes object
* @p: a pointer to a Python 3 bytes object
*
* Return: Nothing
*/
void print_python_bytes(PyObject *p)
{
    /* The pointer with the correct type.*/
    PyBytesObject *s;
    unsigned int i;

    printf("[.] bytes object info\n");
    /* casting the PyObject pointer to a PyBytesObject pointer */
    s = (PyBytesObject *)p;
    /* never trust anyone, check that this is actually
       a PyBytesObject object. */
    if (s && PyBytes_Check(s))
    {
         /* a pointer holds the memory address of the first byte
        of the data it points to */
         printf("  address of the object: %p\n", (void *)s);
         /* op_size is in the ob_base structure, of type PyVarObject. */
         printf("  size: %ld\n", s->ob_base.ob_size);
         /* ob_sval is the array of bytes, ending with the value 0:
        ob_sval[ob_size] == 0 */
         printf("  trying string: %s\n", s->ob_sval);
         printf("  address of the data: %p\n", (void *)(s->ob_sval));
         printf("  bytes:");
         /* printing each byte at a time, in case this is not
        a "string". bytes doesn't have to be strings.
        ob_sval contains space for 'ob_size+1' elements.
        ob_sval[ob_size] == 0. */
         for (i = 0; i < s->ob_base.ob_size + 1; i++)
         {
              printf(" %02x", s->ob_sval[i] & 0xff);
         }
         printf("\n");
    }
    /* if this is not a PyBytesObject print an error message */
    else
    {
         fprintf(stderr, "  [ERROR] Invalid Bytes Object\n");
    }
}
```

从python脚本调用C函数

创建动态库

正如我们之前所说，我们将使用“动态库方法”从Python 3调用我们的C函数。所以我们只需要用下列命令编译我们的C文件：

```
gcc -Wall -Wextra -pedantic -Werror -std=c99 -shared -Wl,-soname,libPython.so -o libPython.so -fPIC -I/usr/include/python3.4 b
```

不要忘记包含Python 3头文件目录：-I /usr/include/python3.4
这应该创建一个名为libPython.so的动态库。

Python 3中使用动态库

为了使用我们的函数，我们需要在Python脚本中添加下列这几行：

```
import ctypes

lib = ctypes.CDLL('./libPython.so')
lib.print_python_bytes.argtypes = [ctypes.py_object]
```

并以下列方式调用我们的函数：

```
lib.print_python_bytes(s)
```

新的Python脚本

以下是新Python 3脚本（main_bytes.py）的完整源代码：

```
#!/usr/bin/env python3
'''
Prints:
- the address of the bytes object
- a b"string" (bytes object)
- information about the bytes object
And then:
- reads a char from stdin
- prints the same (or not :)) information again
'''

import sys
import ctypes

lib = ctypes.CDLL('./libPython.so')
lib.print_python_bytes.argtypes = [ctypes.py_object]

s = b"Holberton"
print(hex(id(s)))
print(s)
lib.print_python_bytes(s)

sys.stdin.read(1)

print(hex(id(s)))
print(s)
lib.print_python_bytes(s)
```

运行

```
julien@holberton:~/holberton/w/hackthevm1$ ./main_bytes.py
0x7f04d721b210
b'Holberton'
[.] bytes object info
 address of the object: 0x7f04d721b210
 size: 9
 trying string: Holberton
 address of the data: 0x7f04d721b230
 bytes: 48 6f 6c 62 65 72 74 6f 6e 00
```

正如所料：

- id（）返回对象本身的地址（0x7f04d721b210）
- 对象的数据大小（ob_size）为9
- 我们的对象的数据是“Holberton”，48 6f 6c 62 65 72 74 6f 6e 00（它以头文件bytesobject.h中指定的00结尾）

rw_all.py

既然我们对正在发生的事情（指python字节对象结构）有了更多的了解，可以"暴力搜索"映射的内存区域了。让我们更新替换字符串的脚本。不只查看栈或堆，让我们查看这

```
#!/usr/bin/env python3
'''
Locates and replaces (if we have permission) all occurrences of
an ASCII string in the entire virtual memory of a process.

Usage: ./rw_all.py PID search_string replace_by_string
Where:
- PID is the pid of the target process
- search_string is the ASCII string you are looking to overwrite
- replace_by_string is the ASCII string you want to replace
search_string with
'''

import sys

def print_usage_and_exit():
   print('Usage: {} pid search write'.format(sys.argv[0]))
   exit(1)

# check usage
if len(sys.argv) != 4:
   print_usage_and_exit()

# get the pid from args
pid = int(sys.argv[1])
if pid <= 0:
   print_usage_and_exit()
search_string = str(sys.argv[2])
if search_string  == "":
   print_usage_and_exit()
write_string = str(sys.argv[3])
if search_string  == "":
   print_usage_and_exit()

# open the maps and mem files of the process
maps_filename = "/proc/{}/maps".format(pid)
print("[*] maps: {}".format(maps_filename))
mem_filename = "/proc/{}/mem".format(pid)
print("[*] mem: {}".format(mem_filename))

# try opening the file
try:
   maps_file = open('/proc/{}/maps'.format(pid), 'r')
except IOError as e:
   print("[ERROR] Can not open file {}:".format(maps_filename))
   print("        I/O error({}): {}".format(e.errno, e.strerror))
   exit(1)

for line in maps_file:
   # print the name of the memory region
   sline = line.split(' ')
   name = sline[-1][:-1];
   print("[*] Searching in {}:".format(name))

   # parse line
   addr = sline[0]
   perm = sline[1]
   offset = sline[2]
   device = sline[3]
   inode = sline[4]
   pathname = sline[-1][:-1]

   # check if there are read and write permissions
   if perm[0] != 'r' or perm[1] != 'w':
       print("\t[\x1B[31m!\x1B[m] {} does not have read/write permissions ({})".format(pathname, perm))
```

```
        continue

    print("\tpathname = {}".format(pathname))
    print("\taddresses = {}".format(addr))
    print("\tpermisions = {}".format(perm))
    print("\toffset = {}".format(offset))
    print("\tinode = {}".format(inode))

    # get start and end of the memoy region
    addr = addr.split("-")
    if len(addr) != 2: # never trust anyone
        print("[*] Wrong addr format")
        maps_file.close()
        exit(1)
    addr_start = int(addr[0], 16)
    addr_end = int(addr[1], 16)
    print("\tAddr start [{:x}] | end [{:x}]".format(addr_start, addr_end))

    # open and read the memory region
    try:
        mem_file = open(mem_filename, 'rb+')
    except IOError as e:
        print("[ERROR] Can not open file {}:".format(mem_filename))
        print("        I/O error({}): {}".format(e.errno, e.strerror))
        maps_file.close()

    # read the memory region
    mem_file.seek(addr_start)
    region = mem_file.read(addr_end - addr_start)

    # find string
    nb_found = 0;
    try:
        i = region.index(bytes(search_string, "ASCII"))
        while (i):
            print("\t[\x1B[32m:)\x1B[m] Found '{}' at {:x}".format(search_string, i))
            nb_found = nb_found + 1
            # write the new string
        print("\t[:)] Writing '{}' at {:x}".format(write_string, addr_start + i))
            mem_file.seek(addr_start + i)
            mem_file.write(bytes(write_string, "ASCII"))
            mem_file.flush()

            # update our buffer
        region.write(bytes(write_string, "ASCII"), i)

            i = region.index(bytes(search_string, "ASCII"))
    except Exception:
        if nb_found == 0:
            print("\t[\x1B[31m:(\x1B[m] Can't find '{}'".format(search_string))
    mem_file.close()

# close files
maps_file.close()
```

运行

```
julien@holberton:~/holberton/w/hackthevm1$ ./main_bytes.py
0x7f37f1e01210
b'Holberton'
[.] bytes object info
 address of the object: 0x7f37f1e01210
 size: 9
 trying string: Holberton
 address of the data: 0x7f37f1e01230
 bytes: 48 6f 6c 62 65 72 74 6f 6e 00

julien@holberton:~/holberton/w/hackthevm1$ ps aux | grep main_bytes.py | grep -v grep
julien     4713  0.0  0.8  37720  8208 pts/0    S+   18:48   0:00 python3 ./main_bytes.py
julien@holberton:~/holberton/w/hackthevm1$ sudo ./rw_all.py 4713 Holberton "~ Betty ~"
```

```
[*] maps: /proc/4713/maps
[*] mem: /proc/4713/mem
[*] Searching in /usr/bin/python3.4:
   [!] /usr/bin/python3.4 does not have read/write permissions (r-xp)
...
[*] Searching in [heap]:
   pathname = [heap]
   addresses = 00e26000-00f11000
   permisions = rw-p
   offset = 00000000
   inode = 0
   Addr start [e26000] | end [f11000]
   [:)] Found 'Holberton' at 8e422
   [:)] Writing '~ Betty ~' at eb4422
...
[*] Searching in :
   pathname =
   addresses = 7f37f1df1000-7f37f1fa7000
   permisions = rw-p
   offset = 00000000
   inode = 0
   Addr start [7f37f1df1000] | end [7f37f1fa7000]
   [:)] Found 'Holberton' at 10230
   [:)] Writing '~ Betty ~' at 7f37f1e01230
...
[*] Searching in [stack]:
   pathname = [stack]
   addresses = 7ffdc3d0c000-7ffdc3d2d000
   permisions = rw-p
   offset = 00000000
   inode = 0
   Addr start [7ffdc3d0c000] | end [7ffdc3d2d000]
   [:(] Can't find 'Holberton'
...
julien@holberton:~/holberton/w/hackthevm1$
```

如果我们在运行的main_bytes.py中按Enter键...

```
julien@holberton:~/holberton/w/hackthevm1$ ./main_bytes.py
0x7f37f1e01210
b'Holberton'
[.] bytes object info
 address of the object: 0x7f37f1e01210
 size: 9
 trying string: Holberton
 address of the data: 0x7f37f1e01230
 bytes: 48 6f 6c 62 65 72 74 6f 6e 00

0x7f37f1e01210
b'~ Betty ~'
[.] bytes object info
 address of the object: 0x7f37f1e01210
 size: 9
 trying string: ~ Betty ~
 address of the data: 0x7f37f1e01230
 bytes: 7e 20 42 65 74 74 79 20 7e 00
julien@holberton:~/holberton/w/hackthevm1$
```

成功了

结尾

我们设法修改Python 3脚本使用的字符串。非常好！但我们仍然有一些问题要回答：

- [堆]内存区域中的“Holberton”字符串是什么？
- Python 3如何在堆外分配内存？
- 如果Python 3没有使用堆，那么它在object.h中说“对象是在堆上分配的结构”是指什么？

这将是下一次的讨论
与此同时，如果你迫不及待下一篇文章，你可以试着自己找出答案。

文件

这里包含本教程中创建的所有脚本和动态库的源代码：

- main.py: 第一个目标
- main_id.py：第二个目标，打印字节对象的id
- main_bytes.py：最终目标，使用我们的动态库打印有关字节对象的信息
- read_write_heap.py：用于查找和替换进程堆中的字符串的“原始”脚本
- read_write_stack.py：与上一个作用相同，但是在栈中搜索和替换而不是堆
- rw_all.py：与之前作用相同，但是在每个可读写的内存区域中
- bytes.c：用于打印有关Python 3字节对象的信息的C函数

上一篇：x86_64逆向工程简介——其他练习 下一篇：利用DOCX文档远程模板注入执行宏
1. 0 条回复
    - 动动手指，沙发就是你的了！