

Docker

Mod233

July 22, 2018

Abstract

Docker 的学习拖拉了很多周了，烦死自己了，处理不好生活中的杂事，对任何事又缺乏持久动力，这样真的不行不行啊，求求自己，接下来的一月多一些的时间都放在 docker 上面吧，学成这个是一劳永逸的事情。

这里开个篇章主要来记录一些基础的命令，我首先要说的是，这些命令需要自己熟练背诵的，所以，整理之后都记下来吧。

Contents

1 Docker 与 Container	2
1.1 Docker 配置	2
1.2 Docker 镜像	8
1.3 Docker 仓库	12
1.4 Docker 数据管理	13
1.5 Dockerfile 编写	14
2 Stack	14
2.1 Stack 与函数调用	14
3 Heap	21
3.1 Heap 函数	21
3.2 Heap 类型	22
3.3 Heap 工作流程	23
3.4 Heap Overflow	27

4	Core	28
4.1	Core 配置	28
4.2	Core 调试	30
5	绘制图表	34
6	幻灯片演示	34
7	从错误中救赎	34
8	问题探骊	34
8.1	动态链接库	34
8.2	Stack 相关	34
8.3	Heap	34

1 Docker 与 Container

这个章节主要讲一些简单 shell 命令组成的 shell 脚本，是很基础的东西，但经过几层嵌套后，就能够实现相对复杂场景。

1.1 Docker 配置

举的第一个例子是字符串的比对，具体例子如下：

```

1  #!/bin/bash
2  # str-test.sh: Testing null strings and unquoted strings,
3  #+ but not strings and sealing wax, not to mention
   cabbages and kings . . .
4
5  # Using  if [ ... ]
6
7  # If a string has not been initialized, it has no defined
   value.
8  # This state is called "null" (not the same as zero!).
9
10 if [ -n $string1 ]    # string1 has not been declared or
   initialized.
11 then

```

```

12     echo "String\|\"string1\|\"_is_not_null.\"
13 else
14     echo "String\|\"string1\|\"_is_null.\"
15 fi                                     # Wrong result.
16 # Shows $string1 as not null, although it was not
    initialized.
17
18 echo
19
20 # Let's try it again.
21
22 if [ -n "$string1" ] # This time, $string1 is quoted.
23 then
24     echo "String\|\"string1\|\"_is_not_null.\"
25 else
26     echo "String\|\"string1\|\"_is_null.\"
27 fi                                     # Quote strings within test brackets
    !
28
29 echo
30
31 if [ $string1 ]      # This time, $string1 stands naked.
32 then
33     echo "String\|\"string1\|\"_is_not_null.\"
34 else
35     echo "String\|\"string1\|\"_is_null.\"
36 fi                                     # This works fine.
37 # The [ ... ] test operator alone detects whether the
    string is null.
38 # However it is good practice to quote it (if [ "$string1"
    ]).
39 #
40 # As Stephane Chazelas points out,
41 #     if [ $string1 ]      has one argument, "]"
42 #     if [ "$string1" ]   has two arguments, the empty "
    $string1" and "]"
43
44
45 echo
46

```

```

47
48 string1=initialized
49
50 if [ $string1 ]           # Again, $string1 stands unquoted.
51 then
52     echo "String_\`string1\`is_not_null."
53 else
54     echo "String_\`string1\`is_null."
55 fi                       # Again, gives correct result.
56 # Still, it is better to quote it ("`string1`"), because .
57     . .
58
59 string1="a_`b"
60
61 if [ $string1 ]           # Again, $string1 stands unquoted.
62 then
63     echo "String_\`string1\`is_not_null."
64 else
65     echo "String_\`string1\`is_null."
66 fi                       # Not quoting "`string1`" now gives
67     wrong result!
68
69 exit 0   # Thank you, also, Florian Wisser, for the "heads
70     -up".

```

给出输出结果：

```

1  sh_script ./exp7-6.sh
2  exp1
3  String "`string1`" is not null.
4  exp2
5  String "`string1`" is null.
6  exp3
7  String "`string1`" is zero.
8  exp4
9  String "`string1`" is not null.
10 exp5
11 String "`string1`" is not null.

```

这个例子就是想说明，如果要想测试一个字符串是否是 null，最好使

用 `if ["$string1"]` 利用引号来把变量包裹起来。

sh 脚本中，数值比较中，字符串和数字采用不同操作符，看下面的例子：

```
1  #!/bin/bash
2
3  a=4
4  b=5
5
6  # Here "a" and "b" can be treated either as integers or
   strings.
7  # There is some blurring between the arithmetic and
   string comparisons,
8  #+ since Bash variables are not strongly typed.
9
10 # Bash permits integer operations and comparisons on
   variables
11 #+ whose value consists of all-integer characters.
12 # Caution advised, however.
13
14 echo
15
16 if [ "$a" -ne "$b" ]
17 then
18     echo "$a_is_not_equal_to_$b"
19     echo "(arithmetic_comparison)"
20 fi
21
22 echo
23
24 if [ "$a" != "$b" ]
25 then
26     echo "$a_is_not_equal_to_$b."
27     echo "(string_comparison)"
28     #      "4" != "5"
29     # ASCII 52 != ASCII 53
30 fi
31
32 # In this particular instance, both "-ne" and "!=" work.
33
```

```

34 echo
35
36 exit 0

```

输出结果：

```

1 exp1
2 ./exp7-5.sh: line 6: [: ne: binary operator expected
3 a is equal to b
4 exp2
5 a is not equal to b

```

这个例子就是想强调，在进行比较操作时，一般整数比较用 `-eq`, `-ns`, `-gt`.etc 等字符，而字符串一般用 `=`, `==`, `>`, `<` 等符号。

部分时候，shell 脚本需要对字符串进行操作，比如下面：

```

1 #!/bin/bash
2 # zmore
3
4 # View gzipped files with 'more' filter.
5
6 E_NOARGS=85
7 E_NOTFOUND=86
8 E_NOTGZIP=87
9
10 if [ $# -eq 0 ] # same effect as: if [ -z "$1" ]
11 # $1 can exist, but be empty: zmore "" arg2 arg3
12 then
13     echo "Usage: `basename $0` filename" >&2
14     # Error message to stderr.
15     exit $E_NOARGS
16     # Returns 85 as exit status of script (error code).
17 fi
18
19 filename=$1
20
21 if [ ! -f "$filename" ] # Quoting $filename allows for
22 # possible spaces.
23 then
24     echo "File $filename not found!" >&2 # Error message
25     to stderr.

```

```

24     exit $E_NOTFOUND
25 fi
26
27 if [ ${filename##*.} != "gz" ]
28 # Using bracket in variable substitution.
29 then
30     echo "File_$1_is_not_a_gzipped_file!"
31     exit $E_NOTGZIP
32 fi
33
34 zcat $1 | more
35
36 # Uses the 'more' filter.
37 # May substitute 'less' if desired.
38
39 exit $? # Script returns exit status of pipe.
40 # Actually "exit $?" is unnecessary, as the script will,
    in any case,
41 #+ return the exit status of the last command executed.

```

3. 脚本中涉及了返回码，这个以后尽可能的有吧，显得代码很规范。

安装完成后，进入 GDB：

```

1  ~ gdb
2  GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
3  Copyright (C) 2016 Free Software Foundation, Inc.
4  ...SNIP...
5  [*] 8 commands could not be loaded, run `gef missing` to
    know why.
6  gef> gef missing
7  [*] Command `ropper` is missing, reason  Missing `
    ropper` package for Python3, install with: `pip3
    install ropper`.
8  [*] Command `unicorn-emulate` is missing, reason  Missing `unicorn` package for Python3. Install with `
    pip3 install unicorn`.

```

```

9  [*] Command `capstone-disassemble` is missing, reason []
    Missing `capstone` package for Python3. Install with `
    pip3 install capstone`.
10 [*] Command `set-permission` is missing, reason []
    Missing `keystone-engine` package for Python3, install
    with: `pip3 install keystone-engine`.
11 [*] Command `assemble` is missing, reason [] Missing `
    keystone-engine` package for Python3, install with: `
    pip3 install keystone-engine`.

```

可能还会出现别的 python3 包缺失的情况，如果利用 pip 能完成安装最好，不行的就需要手动安装了，手动测试了下还是比较顺利的：

```

1  [] .gdbplugins git clone https://github.com/unicorn-engine
    /unicorn.git
2  [] .gdbplugins cd unicorn/bindings/python
3  [] python git:(master) ls
4  build          sample_all.sh          sample_mips.py
    shellcode.py
5  dist           sample_arm64eb.py      sample_network_auditing.py
    unicorn
6  Makefile       sample_arm64.py          sample_sparc.py
    unicorn.egg-info
7  MANIFEST.in   sample_armeb.py          sample_x86.py
8  prebuilt      sample_arm.py            setup.cfg
9  README.TXT    sample_m68k.py          setup.py
10 [] python git:(master) sudo python3 setup.py install

```

1.2 Docker 镜像

:

```

1  [] factorial-test git:(master) [] g++ -g -o factorial
    factorial.c
2  [] factorial-test git:(master) [] ls
3  factorial      factorial.c
4  [] factorial-test git:(master) [] gdb factorial
5  ...SNIP...
6  (gdb) run 1

```



```

7 Starting program: /home/csober/Documents/Github/gdb-test/
  cpp-file/factorial-test/factorial 1
8 factorial 1 = 1
9 [Inferior 1 (process 8533) exited normally]
10 (gdb) run 2
11 Starting program: /home/csober/Documents/Github/gdb-test/
  cpp-file/factorial-test/factorial 2
12 factorial 2 = 2

```

当程序中出现错误后，GDB 中也会提示错误出现的位置，并且可以使用 *backtrace* 回溯函数调用过程，虽然，gdb 没有说为什么程序崩溃，但一般标注在函数头，都是子函数调用引起的错误，可以使用 *backtrace* 来查看活动函数递归调用的栈帧结果。

一般情况下，栈帧由返回地址、函数的参数及局部变量组成。利用 *backtrace* 追踪的时候，从当前停止或暂停的最顶部函数开始，向下直到 *main()* 函数：

```

1 (gdb) run -1
2 Starting program: /home/csober/Documents/Github/gdb-test/
  cpp-file/factorial-test/factorial -1
3
4 Program received signal SIGSEGV, Segmentation fault.
5 0x00000000004005fe in factorial (
6     n=<error reading variable: Cannot access memory at
      address 0x7ffffff7fefec>) at factorial.c:3
7 3   int factorial(int n){
8
9 (gdb) backtrace
10 #0  0x00000000004005fe in factorial (
11     n=<error reading variable: Cannot access memory at
      address 0x7ffffff7fefec>) at factorial.c:3
12 #1  0x0000000000400622 in factorial (n=-174641) at
      factorial.c:6
13 #2  0x0000000000400622 in factorial (n=-174640) at
      factorial.c:6
14 ...SNIP...
15 #174639 0x0000000000400622 in factorial (n=-3) at
      factorial.c:6
16 #174640 0x0000000000400622 in factorial (n=-2) at
      factorial.c:6

```

```

17 #174641 0x0000000000400622 in factorial (n=-1) at
    factorial.c:6
18 #174642 0x0000000000400688 in main (argc=2, argv=0
    x7fffffffda48) at factorial.c:16

```

像上面，如果由于频繁子函数调用引起的栈溢出，可以使用 `backtrace -num` 从内向外打印指定层数的栈结构：

```

1 (gdb) backtrace -5
2 #174638 0x0000000000400622 in factorial (n=-4) at
    factorial.c:6
3 #174639 0x0000000000400622 in factorial (n=-3) at
    factorial.c:6
4 #174640 0x0000000000400622 in factorial (n=-2) at
    factorial.c:6
5 #174641 0x0000000000400622 in factorial (n=-1) at
    factorial.c:6
6 #174642 0x0000000000400688 in main (argc=2, argv=0
    x7fffffffda48) at factorial.c:16

```

如果需要输出执行过程中某个变量的值，可以使用 `print` 命令，如果想长期追踪某个变量的值，可以使用 `display` 命令：

```

1 (gdb) list
2 1  #include <stdio>
3 2  #include <stdlib>
4 3  int factorial(int n){
5 4      int result = 1;
6 5      if(n==0) return 1;
7 6      result=factorial(n-1)*n;
8 7      return result;
9 8  }
10 9  int main(int argc,const char*argv[]){
11 10      int n, result;
12 (gdb) backtrace
13 #0  factorial (n=-1) at factorial.c:4
14 #1  0x0000000000400688 in main (argc=2, argv=0
    x7fffffffda48) at factorial.c:16
15 (gdb) print n
16 $1 = -1
17 (gdb) display n

```

```

18 3: n = -10
19 (gdb) si
20 0x00000000000040061d 6      result=factorial(n-1)*n;
21 3: n = -10
22 (gdb) si
23 factorial (n=0) at factorial.c:3
24 3  int factorial(int n){
25 3: n = 0

```

`display` 命令还有种重要的用法是 `display /i $pc` 显示下一行要执行的汇编代码，这在调试二进制文件中非常有用：

```

1 (gdb) display /i $pc
2 4: x/i $pc
3 => 0x4005f6 <factorial(int)>: push    %rbp
4 (gdb) si
5 0x0000000000004005f7 3  int factorial(int n){
6 3: n = 0
7 4: x/i $pc
8 => 0x4005f7 <factorial(int)+1>: mov    %rsp,%rbp
9 (gdb) si
10 0x0000000000004005fa 3  int factorial(int n){
11 3: n = 0
12 4: x/i $pc
13 => 0x4005fa <factorial(int)+4>: sub    $0x20,%rsp
14 (gdb) si
15 0x0000000000004005fe 3  int factorial(int n){
16 3: n = 0
17 4: x/i $pc
18 => 0x4005fe <factorial(int)+8>: mov    %edi,-0x14(%rbp)
19
20 # 如果需要删除某个 display
21 (gdb) undisplay 3
22 (gdb) undisplay 4

```

还有一个重要的命令是 `x(examine)`，用来检查输出特定内存单位的值，命令格式为 `x/nfu addr`；

`n` 指明需要连续检测多少单位；`f` 指明输出的格式：(‘x’ , ‘d’ , ‘u’ , ‘o’ , ‘t’ , ‘a’ , ‘c’ , ‘f’ , ‘s’)，其中 ‘i’ 代表 machine instruction，‘x’ (hexadecimal) 代表以十六进制输出；

u 指明单位字节大小: b(Bytes), h(Halfwords two bytes), w(Words four bytes), g(Giant words eight bytes)

addr 指明地址位置。

```

1 gef  x/8xb 0x7fffffff990
2 0x7fffffff990: 0x11      0x00      0x00      0x00      0x00      0
   x00      0x00      0x00
3 gef  x/8xb $rbp-0x24
4 0x7fffffff990c: 0x01      0x00      0x00      0x00      0x00      0
   x00      0x00      0x00

```

1.3 Docker 仓库

Docker 仓库类似与 Github 仓库, 可以供用户提交镜像及下载, 地址为: <https://hub.docker.com/>。

下面这个部分, 我主要使用下载的 ubuntu:16.04 的镜像, 制作一个自己用于漏洞调试的镜像, 来方便以后使用:

```

1 $ docker run -t -i ubuntu:16.04 /bin/bash
2 $ docker images
3 REPOSITORY          TAG          IMAGE ID
4 skysider/vulndocker  cve-2018-6789  d9eec37450a4
   4 months ago        364MB
5 nvidia/cuda          latest       af786b192aa5
   6 months ago        2.16GB
6 tensorflow/tensorflow latest-gpu    12cea85b0ad7
   7 months ago        3.36GB
7 tensorflow/tensorflow latest       4f6da7f06480
   7 months ago        1.25GB
8 ubuntu              16.04       20c44cd7596f
   8 months ago        123MB
9 centos              7.4.1708    3afd47092a0e
   8 months ago        197MB

```

run 命令后会自动搜索本地的镜像列表, 如果没有则从 docker hub 下载镜像,

```

1 gef  break m

```

```
2 | gef | break *0x4005a6
3 | Breakpoint 1 at 0x4005a6: file first_fist.cpp, line 3.
```

1.4 Docker 数据管理

程序个别情况下的崩溃在于内存的使用，比如内存泄露、错误访问。内存的剖析工具有多种。最简单的是 `top` 命令，`top` 命令非常易于实现，但当自己需要添加插装代码和做手工分析的时候需要当量工作，特别是当程序中有很多不同模块使用动态内存的时候。

大部分的内存剖析工具，主要通过监视 `malloc()`/`new` 在堆上分配的动态内存，因此大部分也被称为堆剖析工具 (heap profiler)，该类工具会记录动态内存分配的时间、由谁分配、大小以及何时由谁释放。在程序结束后，还会输出图标和日志文件，展示内存使用的细节，并使得内存更容易分配给最大的内存使用者。

常见的工具有：AQtime (Windows)、Massif (Linux)，下面详细介绍 Massif。作为 Valgrind 工具套件的一部分，非常易于使用，并且能够生成内存使用报告，供对应的可视化软件 (Massif-Visualizer) 分析：

```
1 | | malloc-test git:(master) | valgrind --tool=massif ./
  | | malloc i 100000 8
2 | ==9726== Massif, a heap profiler
3 | ==9726== Copyright (C) 2003-2015, and GNU GPL'd, by
  | | Nicholas Nethercote
4 | ==9726== Using Valgrind-3.11.0 and LibVEX; rerun with -h
  | | for copyright info
5 | ==9726== Command: ./malloc i 100000 8
6 | ==9726==
7 | before malloc: hit return to continue
8 | | malloc-test git:(master) | ls
9 | fflush.cpp malloc malloc.cpp massif.out.9726
```

1.5 Dockerfile 编写

2 Stack

Stack 是程序中一段重要的内存空间，在进程运行时就被创建。Stack 主要被用来存储函数的局部变量、环境变量，环境变量主要是能够帮助程序在各个函数间跳转的地址，像 rbp, rsp 等。

Stack 是程序运行过程非常核心的部分，主要是由 rsp(栈顶 TOP) 和 rbp(栈底 BASE) 两个寄存器管理的，涉及到的操作是 PUSH(压栈) 和 POP(弹栈)；每执行一次压栈操作后，rsp 值就减少一定值（在我 64 位的操作系统中是减 8），每执行一次弹栈操作后，rsp 的值就增加一定值。

2.1 Stack 与函数调用

为了尽可能完整的展示函数的调用过程，我使用下面的 c 代码进行分析：

```
1  #include <stdio.h>
2  #include <string.h>
3  void test(int a1,int a2,int a3,int a4,int a5,int a6,int
      a7,int a8,int a9,int a10,int a11,int a12,int a13,int
      a14,int a15,int a16,int a17,int a18){
4      char str[10];
5      memcpy(str,
              "abcdefghijklmnopqrstuvwx\
6 yz1234567890987654321\
7 ABCDEFGHIJKLMNOPQRSTU\
8 VWXYZSUCHANICEDAYYESA\
9 BOYCANDOEVERYTHINGFORAGIRL ",40);
10     int a=2;
11     a++;
12     printf("%d\n",a);
13 }
14 int main(int argc,const char*argv[]){
15     test(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18);
16     return 0;
17 }
```

在函数调用的过程中，一般先将参数压栈，然后将子函数运行完后的返回地址压栈，然后调用 `call` 指令，执行子函数，具体看如下代码：

```

1 |          0x0040067c          pushq $0x12 ; .//
   | first_fist.c:11 ; 18
2 |          0x0040067e          pushq $0x11 ; 17
3 |          0x00400680          pushq $0x10 ; 16
4 |          0x00400682          pushq $0xf ; 15
5 |          0x00400684          pushq $0xe ; 14
6 |          0x00400686          pushq $0xd ; 13
7 |          0x00400688          pushq $0xc ; 12
8 |          0x0040068a          pushq $0xb ; 11
9 |          0x0040068c          pushq $0xa ; 10
10 |         0x0040068e          pushq $9 ; 9
11 |         0x00400690          pushq $8 ; 8
12 |         0x00400692          pushq $7 ; 7
13 |         0x00400694          movl $6, %r9d
14 |         0x0040069a          movl $5, %r8d
15 |         0x004006a0          movl $4, %ecx
16 |         0x004006a5          movl $3, %edx
17 |         0x004006aa          movl $2, %esi
18 |         0x004006af          movl $1, %edi
19 |         0x004006b4          callq sym.test
20 |         0x004006b9          addq $0x60, %rsp ; ``
21 |         0x004006bd          movl $0, %eax ; .//
   | first_fist.c:12
22 |         0x004006c2          leave ; .//first_fist.c
   | :13
23 |         0x004006c3          retq

```

上面的代码有几个细节值得关注：1. 子函数的参数，一般由 `edi`, `esi`, `edx`, `ecx` 等六个寄存器存储，当参数超过六个后，再由 `push` 指令压入栈空间中。2. `callq` 指令是由 `push eip; jmp addr` 两条指令组合在一起的。

下面具体看下进行了 `jump` 跳转后，指令的执行情况：

```

1 | / (fcn) sym.test 119
2 | |   sym.test ();
3 | |       ; var int local_48h @ rbp-0x48
4 | |       ; var int local_44h @ rbp-0x44
5 | |       ; var int local_40h @ rbp-0x40

```

```

6 |             ; var int local_3ch @ rbp-0x3c
7 |             ; var int local_38h @ rbp-0x38
8 |             ; var int local_34h @ rbp-0x34
9 |             ; var int local_24h @ rbp-0x24
10 |            ; var int local_20h @ rbp-0x20
11 |            ; var int local_8h @ rbp-0x8
12 |            0x004005f6          pushq %rbp ; .//
    first_fist.c:3
13 |            0x004005f7          movq %rsp, %rbp
14 |            0x004005fa          subq $0x50, %rsp ; 'P'
15 |            0x004005fe          movl %edi, local_34h
16 |            0x00400601          movl %esi, local_38h
17 |            0x00400604          movl %edx, local_3ch
18 |            0x00400607          movl %ecx, local_40h
19 |            0x0040060a          movl %r8d, local_44h
20 |            0x0040060e          movl %r9d, local_48h
21 |            0x00400612          movq %fs:0x28, %rax ; [0
    x28:8]=-1 ; '(' ; 40
22 |            0x0040061b          movq %rax, local_8h
23 |            0x0040061f          xorl %eax, %eax
24 |            0x00400621          leaq local_20h, %rax ;
    .//first_fist.c:5
25 |            0x00400625          movl $0x28, %edx ; '(' ;
    40
26 |            0x0040062a          movl $str.
    abcdefghijklmnopqrstuvwxyz1234567890987
27 | 654321ABCDEFGHIJKLMNOPQRSTUVWXYZSUCHANICEDAYYESABOYCAND
28 | OEVERYTHINGFORAGIRL, %esi ; 0x400758 ;
29 | "abcdefghijklmnopqrstuvwxyz1234567890987654321ABCDEFGH
30 | IJKLMNOPQRSTUVWXYZSUCHANICEDAYYESABOYCAND OEVERYTHINGFORAGIRL
    "
31 |            0x0040062f          movq %rax, %rdi
32 |            0x00400632          callq sym.imp.memcpy ;
    void *memcpy(void *s1, const void *s2, size_t n)
33 |            0x00400637          movl $2, local_24h ; .//
    first_fist.c:6
34 |            0x0040063e          addl $1, local_24h ; .//
    first_fist.c:7
35 |            0x00400642          movl local_24h, %eax ;
    .//first_fist.c:8

```



```

36 |          0x00400645          movl %eax, %esi
37 |          0x00400647          movl $0x4007ca, %edi
38 |          0x0040064c          movl $0, %eax
39 |          0x00400651          callq sym.imp.printf ;
    |      int printf(const char *format)
40 |          0x00400656          nop ; .//first_fist.c:9
41 |          0x00400657          movq local_8h, %rax
42 |          0x0040065b          xorq %fs:0x28, %rax
43 |          ,=< 0x00400664          je 0x40066b
44 |          | 0x00400666          callq sym.imp.
    |      __stack_chk_fail ; void __stack_chk_fail(void)
45 |          `-> 0x0040066b          leave
46 |          \ 0x0040066c          retq

```

从上面的代码中，也能得到许多重要的提示：

1. 执行完 `call` 命令后，子函数首先要将当前 `rbp` 寄存器值压栈。
2. 将 `rbp` 压栈后，对 `rbp` 赋新值，并将 `rsp` 减去一定值，抬高栈空间，这个数值是依据子函数所需求的内存来定的。
3. 对刚才通过 `edi` 等寄存器存储的参数，会存储到新开辟的栈空间中。
4. `ret` 指令与 `call` 指令对应，实际是 `pop rip`，然后 `rip` 执行下一条指令。

下面检测栈空间的情况：

```

1  0x00007fffffffddac0|+0x00: 0x00007fffffffddb40  0
    x000000000004006d0  <__libc_csu_init+0> push r15
    $rsp, $rbp
2  0x00007fffffffddac8|+0x08: 0x000000000004006b9  <main+76>
    add rsp, 0x60
3  0x00007fffffffddad0|+0x10: 0x00000000000000007
4  0x00007fffffffddad8|+0x18: 0x00000000000000008
5  0x00007fffffffddae0|+0x20: 0x00000000000000009
6  0x00007fffffffddae8|+0x28: 0x0000000000000000a
7  0x00007fffffffddaf0|+0x30: 0x0000000000000000b
8  0x00007fffffffddaf8|+0x38: 0x0000000000000000c

```

可以看到，栈空间中 `0x00007fffffffddad0-0x00007fffffffddaf8` 都是通过 `push` 指令压栈的参数，`0x00007fffffffddac8` 位置存储是返回地址（即 `test` 函数执行完后的下一条指令）

今天在尝试栈溢出来覆盖返回地址的实验时，发现始终不能实现，在执

行了 `test:memcpy` 函数后，指令直接返回了 `main` 函数当中，和学长的讨论中发现是由于 `stack-check` 的原因，可以从子函数的汇编代码中看到这样几条指令：

```

1 |          0x00400612          movq %fs:0x28, %rax ; [0
  |          x28:8]=-1 ; '(' ; 40
2 |          0x0040061b          movq %rax, local_8h
3 |          0x0040061f          xorl %eax, %eax
4 | ...SNIP...
5 |          ,=< 0x00400664          je  0x40066b
6 |          | 0x00400666          callq sym.imp.
  |          __stack_chk_fail ; void __stack_chk_fail(void)
7 |          `-> 0x0040066b          leave

```

`%fs:0x28` 指向的是一个特殊地址，存储 ??????

说明了 `Stack` 与子函数调用后，还想分享下今天尝试写 `shellcode` 遇到一些问题，测试的代码为：

```

1 #include <stdio.h>
2 #include <string.h>
3 void test(){
4     char str[10];
5     scanf("%s",&str);
6     printf("%s\n",str);
7 }
8 int main(int argc,const char*argv[]){
9     test();
10    return 0;
11 }

```

为了方便调试，编译的时候使用 `-g` 加入的调试信息，并关闭了 `Stack-guard`：

```

1 □ first_fist g++ -g scanf.c -fno-stack-protector -o scanf
  -nsp

```

输入数据为：`123456789a123456789b123456789c123456789d`，目的是想通过规则的数据来定位 `rip`，通过内存检查，得到内存的布局为：

```

1 | 0x000007fffffffdd970|+0x00: 0x0000000000000000    □ $rsp,
  | $rsi

```

```

2 0x00007fffffff978|+0x08: 0x0000000000000000
3 0x00007fffffff980|+0x10: 0x00007fffffff9a0  0
    x00000000004005b0  <__libc_csu_init+0> push r15
    $rbp
4 0x00007fffffff988|+0x18: 0x00000000004005a7  <main+20>
    mov eax, 0x0
5 0x00007fffffff990|+0x20: 0x00007fffffff9a8  0
    x00007fffffffdeb9  "/home/csober/Documents/Github/
    how2stack/first_fist[...]"
6
7 ...SNIP...
8
9 0x00007fffffff970|+0x00: "123456789
    a123456789b123456789c123456789d"  $rsp
10 0x00007fffffff978|+0x08: "9
    a123456789b123456789c123456789d"
11 0x00007fffffff980|+0x10: "789b123456789c123456789d"
    $rbp
12 0x00007fffffff988|+0x18: "56789c123456789d"
13 0x00007fffffff990|+0x20: "3456789d"
14 0x00007fffffff998|+0x28: 0x0000000010000000
15 0x00007fffffff9a0|+0x30: 0x00000000004005b0  <
    __libc_csu_init+0> push r15
16 0x00007fffffff9a8|+0x38: 0x00007ffff7a2d830  <
    __libc_start_main+240> mov edi, eax
17
18
19 ...SNIP...
20
21 gef x /10xg 0x00007fffffff960
22 0x7fffffff960: 0x0000000000400470  0x0000000000400590
23 0x7fffffff970: 0x3837363534333231  0x3635343332316139
24 0x7fffffff980: 0x3433323162393837  0x3231633938373635
25 0x7fffffff990: 0x6439383736353433  0x0000000010000000
26 0x7fffffff9a0: 0x00000000004005b0  0x00007ffff7a2d830

```

从中可以看出，栈顶的位置为 0x7fffffff970，当 scanf 读入数据后，返回地址和 rbp 都被覆盖了，但将 0x7fffffff970-0x7fffffff990 地址段转换成 ASCII 码后，难以得到直观的存储结果：

```

1 0x7fffffff970: 8 7 6 5 4 3 2 1 — 6 5 4 3 2 1 a 9
2 0x7fffffff980: 4 3 2 1 b 9 8 7 — 2 1 c 9 8 7 6 5
3 0x7fffffff990: d 9 8 7 6 5 4 3

```

多方询问后，对结果解释如下：

1. 64 位操作系统每次处理 64 位数据，从字节上体现的就是每次处理 8 个字节。

2. x86_64 架构采用的是小端字节序，所以存储起来刚好相反。

在这里要区分，指令 push, pop 对 rsp 进行操作时，与一般情况是反的，但数据存入内存中时，还是从低地址向高地址存储的。

为了直观完成返回地址覆盖的实验，我使用下面程序进行调试：

```

1  #include <stdio.h>
2  #include <string.h>
3  void test(){
4      char str[10];
5      scanf("%s",&str);
6      printf("%s\n",str);
7  }
8  int main(int argc,const char*argv[]){
9      test();
10     printf("finish_test\n");
11     printf("tmp\n");
12     printf("jump_here\n");
13     return 0;
14 }

```

构建 POC 为：

```

1 import os
2
3 def print_unvisible():
4     str = '12345678abcdefgh|xa0|xd9|xff|xff|xff|xf7|x00|
5           x00|xbb|x05|x40|x00|x00|x00|x00|x00'
6     print str
7
8 if __name__ == "__main__":
9     print_unvisible()

```

GDB 调试过程中，重定向了 IO 流：

```
1 gef[] run <input.txt
2 Starting program: /home/csober/Documents/Github/how2stack/
   first_fist/scanf-nsp <input.txt
3 12345678abcdefgh♦♦♦♦♦♦
4 jump here
5
6 Program received signal SIGBUS, Bus error.
```

最后提示 *Bus error*，不知道具体原因，但实现了返回地址的覆盖，就暂时告一段落吧。

针对 GDB 调试重定向的问题，Github 中给出一个有用的插件：<https://github.com/Ovi3/pstdio>，下次遇到必需的时候再来研究下。

3 Heap

Heap 是程序运行过程中一段特殊的内存空间，Heap 主要用于满足进程的动态内存请求。这一章节我会就 Heap 的类型、Linux 下操作函数，及如何利用 GDB 来查看 Heap 空间的分配情况。

3.1 Heap 函数

堆的管理很复杂，但从上层 c 语言来看，主要由 *malloc* 和 *free* 进行管理。

Malloc: void* malloc(size_t size)

1. First time malloc is called in thread:

1.1 Allocates a reasonable amount of memory

1.2 Creates a heap segment or equivalent

1.3 Return the caller a pointer to a memory region within it of suitable for demand.

2 Not the first call to malloc in thread:

2.1 malloc will just return a pointer to a region within the current heap segment (or equivalent) of suitable size for the caller demand.

Free: void free(void *ptr)

1. Set a memory region previously returned from malloc as not in use.

尽管在各个系统架构下都有 *malloc* 和 *free* 两个函数对堆进行管理, 但实现的细节可能很不相同, 最常用的则是 *glibc* 库中提供的堆栈管理方法。整个章节的展开背景也是 *glibc*。

在 *glibc* 中, 涉及到的系统调用有 *brk(void *addr), void *sbrin(void *addr), void* mmap(void *addr, size_t length, int protect, int flags, int filesdes, off_t offset)*。

brk() 和 *sbrk()* 主要用于主线程申请内存空间, *mmap* 主要用在两种情况: 1. 申请的内存空间大于 *DEFAULT_MMAP_THRESHOLD*; 2. 子进程申请内存空间。

3.2 Heap 类型

在对 *glibc* 堆的工作流程进行深入学习前, 需要对基本名词和堆类型有清晰的认识。

Arena: 是所有可被分配的内存, 主要用于减少 *malloc* 调用与系统内核交互的次数, 一般每个线程一个, 并且个数是有限的。

Heap: 是一段连续的, 由多个 *chunk* 组成的内存空间, 每段 *heap* 只属于一个 *arena*。

Chunk: 内存分配的最小单位, 当进行 *malloc* 调用后, 会返回 *chunk* 的可用段的地址, 每个 *chunk* 仅存在于一个 *heap* 并且属于一个 *arena*。

在堆的管理中, 还涉及到三个关键的数据结构:

1. *malloc_chunk*: 管理组成一个 *chunk* 的所有内容
2. *malloc_info*: 管理与进程中与堆交互的所有内容
3. *malloc_state*: 管理组成一个 *arena* 中的所有内容。

具体的数据结构如下:

```
1 struct malloc_chunk {
2   INTERNAL_SIZE_T prev_size; /* Size of previous chunk (if
   free). */
3   INTERNAL_SIZE_T size; /* Size in bytes, including header.
   */
4   struct malloc_chunk* fd; /* double links — used only if
   free. */
5   struct malloc_chunk* bk;
6   /* Only used for large blocks: pointer to next larger size
   . */
7   struct malloc_chunk* fd_nextsize; /* double links — used
   only if free. */
```

```

8 struct malloc_chunk* bk_nextsize;
9 };

```

可以看到，数据结构会依据内存是否占用而有所不同，

3.3 Heap 工作流程

测试的时候使用的代码为：

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int main(){
5     char* a = malloc(512);
6     char* b = malloc(256);
7     char* c;
8     char* d = malloc(8);
9     char* e = malloc(16);
10    char* f = malloc(32);
11    free(d);
12    free(e);
13    free(f);
14    strcpy(a, "this_is_A!");
15    free(a);
16    c = malloc(500);
17    strcpy(c, "this_is_C!");
18 }

```

将断点设置在 `malloc` 函数后，观察几次申请堆空间过程中，chunk 结构的变化：

```

1 gef[] heap chunks
2 [!] No heap section
3 gef[] c

```

```

4 ...SNIP...
5 gef[] heap chunks
6 Chunk(addr=0x602010, size=0x210, flags=PREV_INUSE)
7 [0x000000000000602010 00 00 00 00 00 00 00 00 00 00
8 00 00 00 00 00 00 .....]
9 Chunk(addr=0x602220, size=0x20df0, flags=PREV_INUSE) []
10 top chunk
11 gef[] c
12 ...SNIP...
13 Chunk(addr=0x602010, size=0x210, flags=PREV_INUSE)
14 [0x000000000000602010 00 00 00 00 00 00 00 00 00 00
15 00 00 00 00 00 00 .....]
16 Chunk(addr=0x602220, size=0x110, flags=PREV_INUSE)
17 [0x000000000000602220 00 00 00 00 00 00 00 00 00 00
18 00 00 00 00 00 00 .....]
19 Chunk(addr=0x602330, size=0x20ce0, flags=PREV_INUSE) []
20 top chunk
21 gef[] c
22 ...SNIP...
23 gef[] heap chunks
24 Chunk(addr=0x602010, size=0x210, flags=PREV_INUSE)
25 [0x000000000000602010 00 00 00 00 00 00 00 00 00 00
26 00 00 00 00 00 00 .....]
27 Chunk(addr=0x602220, size=0x110, flags=PREV_INUSE)
28 [0x000000000000602220 00 00 00 00 00 00 00 00 00 00
29 00 00 00 00 00 00 .....]
30 Chunk(addr=0x602330, size=0x20, flags=PREV_INUSE)
31 [0x000000000000602330 00 00 00 00 00 00 00 00 00 00
32 00 00 00 00 00 00 .....]
33 Chunk(addr=0x602350, size=0x20cc0, flags=PREV_INUSE) []
top chunk
gef[] c
...SNIP...
gef[] c
...SNIP...
gef[] heap chunks
Chunk(addr=0x602010, size=0x210, flags=PREV_INUSE)
[0x000000000000602010 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 .....]
Chunk(addr=0x602220, size=0x110, flags=PREV_INUSE)

```



```

34      [0x00000000000602220      00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00      .....]
35  Chunk(addr=0x602330, size=0x20, flags=PREV_INUSE)
36      [0x00000000000602330      00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00      .....]
37  Chunk(addr=0x602350, size=0x20, flags=PREV_INUSE)
38      [0x00000000000602350      00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00      .....]
39  Chunk(addr=0x602370, size=0x30, flags=PREV_INUSE)
40      [0x00000000000602370      00 00 00 00 00 00 00 00 00 00
      00 00 00 00 00 00      .....]
41  Chunk(addr=0x6023a0, size=0x20c70, flags=PREV_INUSE)  []
      top chunk
42  Chunk(addr=0x602370, size=0x30, flags=PREV_INUSE)
43  Chunk size: 48 (0x30)
44  Usable size: 40 (0x28)
45  Previous chunk size: 0 (0x0)
46  PREV_INUSE flag: On
47  IS_MMAPPED flag: Off
48  NON_MAIN_ARENA flag: Off
49  gef[] heap chunk 0x602350
50  Chunk(addr=0x602350, size=0x20, flags=PREV_INUSE)
51  Chunk size: 32 (0x20)
52  Usable size: 24 (0x18)
53  Previous chunk size: 0 (0x0)
54  PREV_INUSE flag: On
55  IS_MMAPPED flag: Off
56  NON_MAIN_ARENA flag: Off
57  gef[] heap chunk 0x602010
58  Chunk(addr=0x602010, size=0x210, flags=PREV_INUSE)
59  Chunk size: 528 (0x210)
60  Usable size: 520 (0x208)
61  Previous chunk size: 0 (0x0)
62  PREV_INUSE flag: On
63  IS_MMAPPED flag: Off
64  NON_MAIN_ARENA flag: Off

```

从中可以发现几点：

1. 申请的内存空间在地址空间上是连续的。
2. 申请的每段空间都存在 16 字节对齐。

3. 申请的每段空间都有 8 字节用于标识。

下面查看具体内存段的载荷情况：

```
1 gef[] x /20xg 0x602328
2 0x602328: 0x00000000000000021 0x0000000000000000
3 0x602338: 0x0000000000000000 0x0000000000000000
4 0x602348: 0x00000000000000021 0x0000000000000000
5 0x602358: 0x0000000000000000 0x0000000000000000
6 0x602368: 0x00000000000000031 0x0000000000000000
7 0x602378: 0x0000000000000000 0x0000000000000000
8 0x602388: 0x0000000000000000 0x0000000000000000
```

可以看到 0x21 和 0x31 分表代表这段地址空间大小，及占用情况。目前都是申请了内存后，chunks 的分配情况，当进行释放后：

```
1 gef[] c
2 ...SNIP...
3 gef[] c
4 ...SNIP...
5 gef[] x /40xg 0x602320
6 0x602320: 0x0000000000000000 0x0000000000000021
7 0x602330: 0x0000000000000000 0x0000000000000000
8 0x602340: 0x0000000000000000 0x0000000000000021
9 0x602350: 0x000000000000602320 0x0000000000000000
10 0x602360: 0x0000000000000000 0x0000000000000031
11 0x602370: 0x0000000000000000 0x0000000000000000
12 0x602380: 0x0000000000000000 0x0000000000000000
13 gef[] c
14 ...SNIP...
```

上面一个重要的点在于，0x602350 地址段，指明了上一个空的段的位置在 0x602320，这样就和上面的知识串起来了。

检查 bin 链表的链接情况：

```
1 gef[] heap bin
2 -----[ Fastbins for arena 0x7ffff7dd1b20
3 ]-----
4 Fastbins[idx=0, size=0x10] [] Chunk(addr=0x602350, size=0
5 x20, flags=PREV_INUSE) [] Chunk(addr=0x602330, size=0
6 x20, flags=PREV_INUSE)
```

```

4 Fastbins[idx=1, size=0x20]  Chunk(addr=0x602370, size=0
    x30, flags=PREV_INUSE)

```

当代码运行到释放了 *a* 指针后：

```

1 -----[ Unsorted Bin for arena 'main_arena'
    ]-----
2 [+] unsorted_bins[0]: fw=0x602000, bk=0x602000
3   Chunk(addr=0x602010, size=0x210, flags=PREV_INUSE)
4 [+] Found 1 chunks in unsorted bin.

```

如果马上申请同样大小的内存段 *c*，会发现 *Unsorted Bin* 区的内存被从新分配，而不是从 *top chunk* 中申请内存：

```

1 -----[ Unsorted Bin for arena 'main_arena'
    ]-----
2 [+] Found 0 chunks in unsorted bin.

```

3.4 Heap Overflow

这个小节介绍一些与堆溢出及利用有关的例子。

```

1 gef  heap chunks
2 Chunk(addr=0x602010, size=0x20, flags=PREV_INUSE)
3   [0x000000000000602010    00 00 00 00 00 00 00 00 65 64
    61 79 6f 00 00 00    .....edayo...]
4 Chunk(addr=0x602030, size=0x410, flags=PREV_INUSE)
5   [0x000000000000602030    63 70 79 32 0a 68 20 6d 65 6d
    63 70 79 31 0a 00    cpy2.h memcpy1..]
6 Chunk(addr=0x602440, size=0x20bd0, flags=PREV_INUSE)  top chunk

```

4 Core

这里的 Core 主要指的是程序运行过程意外退出，Linux 系统自动生成的调试文件，可以供开发者回溯程序崩溃的原因。这个章节就来讲解 Core Dumps 的配置，及利用 Core 文件对堆栈、函数调用做分析。

4.1 Core 配置

默认情况下，Core 配置是关闭的，因为 Core 文件会拖慢程序崩溃后重启的速度，并设计敏感信息。通过 `ulimit c` 命令和 `/proc/sys/kernel/core_pattern` 文件能够得到当前 core 文件的配置情况：

```
1 ❏ first_fist ulimit -c
2 0
3 ❏ first_fist cat /proc/sys/kernel/core_pattern
4 |/usr/share/apport/apport %p %s %c %d %P
```

下面先设置 core 文件大小，且将其以特定的命名规则存放在特定的文件目录中，在配置的时候可能会遇到权限不够，不能修改 `/proc/sys/kernel/core_pattern` 的情况，尤其是在 ubuntu 中，可以用 `sysctl` 命令来实现：

```
1 ❏ ~ ulimit -c unlimited
2 ❏ ~ sudo mkdir -p /var/cores
3 ❏ ~ sudo echo "/var/cores/core.%e.%p.%g.%t" > /proc/sys/
  kernel/core_pattern
4 zsh: permission denied: /proc/sys/kernel/core_pattern
5 ❏ ~ sudo sysctl -w kernel.core_pattern=/var/cores/core.%e
  .%p.%g.%t
6 [sudo] password for csobar:
7 kernel.core_pattern = /var/cores/core.%e.%p.%g.%t
8 ❏ ~ cat /proc/sys/kernel/core_pattern
9 /var/cores/core.%e.%p.%g.%t
```

如果之后想让 `core_pattern` 的设置永久生效，可以在 `/etc/sysctl.conf` 进行配置，具体配置之后再研究。特别说明下，ubuntu 上是依靠 Apport 实现 core dump 信息的存储的，所以对 `emph/proc/sys/kernel/core_pattern` 那步的配置不做，主要不知道具体怎么配置。配置好了 core 的 size 之后，检查是否能得到 core dump 文件：

```

1  [] first_fist file core
2  core: ELF 64-bit LSB core file x86-64, version 1 (SYSV),
   SVR4-style, from './scanf-nsp'
3  [] first_fist gdb scanf-nsp core
4  ...SNIP...
5  Reading symbols from scanf-nsp...done.
6  [New LWP 17776]
7  Core was generated by './scanf-nsp'.
8  Program terminated with signal SIGBUS, Bus error.
9  #0  main (argc=<error reading variable: Cannot access
   memory at address 0xf7fffffd99c>, argv=<error reading
   variable: Cannot access memory at address 0
   xf7fffffd990>) at scanf.c:14
10 14  }

```

最后的两句话给出了两点重要的提示，并且这些提示都是值得 Google 的：

1. 程序崩溃，并且系统发出了 SIGBUS, Bus error 的提示。
2. 程序不能访问位于 0xf7fffffd99c 地址空间内容，并且错误的函数的位置在 scanf.c:14。

有时会遇到和动态库有关的错误提示，可以结合 dpkg 包管理器来定位错误包，如下：

```

1  warning: JITed object file architecture unknown is not
   compatible with target architecture i386:x86-64.
2  Core was generated by `python ./cachetop.py'.
3  Program terminated with signal SIGSEGV, Segmentation fault
   .
4  #0  0x00007f0a37aac40d in doupdate () from /lib/x86_64-
   linux-gnu/libncursesw.so.5
5  # dpkg -l | grep libncursesw
6  ii  libncursesw5:amd64 6.0+20160213-1
   ubuntu1 amd64
7  shared libraries for terminal handling (wide
   character support)

```

4.2 Core 调试

完成了 core 的配置，并通过 GDB 进入后，下一步就是要提取 core 中的信息来查看程序崩溃时堆栈、函数调用的情况了，下面的例子我选用的都是别人的输出，因为别人选择的 core 报警相对复杂，可以看到更复杂的输出，一般步骤为：

1. 利用 `backtrace` 命令回溯系统栈对函数的调用情况。

```
1 (gdb) bt
2 #0  0x00007f0a37aac40d in doupdate () from /lib/x86_64-
   linux-gnu/libncursesw.so.5
3 #1  0x00007f0a37aa07e6 in wrefresh () from /lib/x86_64-
   linux-gnu/libncursesw.so.5
4 #2  0x00007f0a37a99616 in ?? () from /lib/x86_64-linux-gnu
   /libncursesw.so.5
5 #3  0x00007f0a37a9a325 in wgetch () from /lib/x86_64-linux
   -gnu/libncursesw.so.5
6 #4  0x00007f0a37cc6ec3 in ?? () from /usr/lib/python2.7/
   lib-dynload/_curses.x86_64-linux-gnu.so
7 #5  0x000000000004c4d5a in PyEval_EvalFrameEx ()
8 #6  0x000000000004c2e05 in PyEval_EvalCodeEx ()
9 #7  0x000000000004def08 in ?? ()
10 #8  0x000000000004b1153 in PyObject_Call ()
11 #9  0x000000000004c73ec in PyEval_EvalFrameEx ()
12 #10 0x000000000004c2e05 in PyEval_EvalCodeEx ()
13 #11 0x000000000004caf42 in PyEval_EvalFrameEx ()
14 #12 0x000000000004c2e05 in PyEval_EvalCodeEx ()
15 #13 0x000000000004c2ba9 in PyEval_EvalCode ()
16 #14 0x000000000004f20ef in ?? ()
17 #15 0x000000000004eca72 in PyRun_FileExFlags ()
18 #16 0x000000000004eb1f1 in PyRun_SimpleFileExFlags ()
19 #17 0x0000000000049e18a in Py_Main ()
20 #18 0x00007f0a3be10830 in __libc_start_main (main=0x49daf0
   <main>, argc=2, argv=0x7fffd33d94838, init=<optimized
   out>, fini=<optimized out>, rtld_fini=<optimized out>,
21   stack_end=0x7fffd33d94828) at ../csu/libc-start.c:291
22 #19 0x0000000000049da19 in _start ()
```

查看的函数栈的时候从下往上的顺序，如果中途出现“??”，一般是“symbol translation failed”。遇到这种情况时，可以找一些 gdb 的插

件，或者在 gcc 编译的时候，保留符号信息 (*-fno-omit-frame-pointer -g*) 来修复这些问题。

具体看下上面的输出，从 frames 5 to 17 都是与 python 相关的调用，尽管不确定具体的 modules 调用情况，但基本的脉络为：*wgetch()->wrefresh()->doupdate()*，接下来就需要对栈中最顶层的 *doupdate()* 进行分析。

2. 利用 *disas func* 命令反汇编函数栈最上层函数。

```
1 (gdb) disas doupdate
2 Dump of assembler code for function doupdate:
3     0x00007f0a37aac2e0 <+0>:    push    %r15
4     0x00007f0a37aac2e2 <+2>:    push    %r14
5     0x00007f0a37aac2e4 <+4>:    push    %r13
6     0x00007f0a37aac2e6 <+6>:    push    %r12
7     0x00007f0a37aac2e8 <+8>:    push    %rbp
8     0x00007f0a37aac2e9 <+9>:    push    %rbx
9     0x00007f0a37aac2ea <+10>:   sub     $0xc8,%rsp
10    [...]
11    ——Type <return> to continue, or q <return> to quit——
12    [...]
13     0x00007f0a37aac3f7 <+279>:  cmpb    $0x0,0x21(%rcx)
14     0x00007f0a37aac3fb <+283>:  je      0x7f0a37aac3b <
15     doupdate+2395>
16     0x00007f0a37aac401 <+289>:  mov     0x20cb68(%rip),%rax
17     # 0x7f0a37cb8f70
18     0x00007f0a37aac408 <+296>:  mov     (%rax),%rsi
19     0x00007f0a37aac40b <+299>:  xor     %eax,%eax
20 => 0x00007f0a37aac40d <+301>:  mov     0x10(%rsi),%rdi
21     0x00007f0a37aac411 <+305>:  cmpb    $0x0,0x1c(%rdi)
22     0x00007f0a37aac415 <+309>:  jne     0x7f0a37aac6f7 <
23     doupdate+1047>
24     0x00007f0a37aac41b <+315>:  movswl  0x4(%rcx),%ecx
25     0x00007f0a37aac41f <+319>:  movswl  0x74(%rdx),%edi
26     0x00007f0a37aac423 <+323>:  mov     %rax,0x40(%rsp)
27    [...]
```

只输入 *disas* 命令也会默认的反汇编栈帧中最顶层的函数。标示”=>“代表出错执行的指令。根据这条指令，可以将错误定位到寄存器，下面查看寄存器的值即可。3. 利用 *info registers* 命令查看寄存器的值。

```

1 (gdb) i r
2 rax          0x0  0
3 rbx          0x1993060      26816608
4 rcx          0x19902a0      26804896
5 rdx          0x19ce7d0      27060176
6 rsi          0x0  0
7 rdi          0x19ce7d0      27060176
8 rbp          0x7f0a3848eb10   0x7f0a3848eb10 <SP>
9 rsp          0x7ffd33d93c00   0x7ffd33d93c00
10 r8           0x7f0a37cb93e0   139681862489056
11 r9           0x0  0
12 r10          0x8  8
13 r11          0x202      514
14 r12          0x0  0
15 r13          0x0  0
16 r14          0x7f0a3848eb10   139681870703376
17 r15          0x19ce7d0      27060176
18 rip          0x7f0a37aac40d   0x7f0a37aac40d <douupdate
    +301>
19 eflags       0x10246 [ PF ZF IF RF ]
20 cs           0x33  51
21 ss           0x2b  43
22 ds           0x0  0
23 es           0x0  0
24 fs           0x0  0
25 gs           0x0  0

```

可以看到，%rsi 的值为 0，很明显 0x0 不是一个有效的地址空间，出现了一种常见 segfault: *dereferencing an uninitialized or NULL pointer*.

4. 利用 `i proc m` 命令检查内存分配情况。

```

1 (gdb) i proc m
2 Mapped address spaces:
3
4      Start Addr      End Addr      Size      Offset
5      objfile
6      0x400000      0x6e7000      0x2e7000      0x0
   /usr/bin/python2.7
6      0x8e6000      0x8e8000      0x2000      0x2e6000

```


7	/usr/bin/python2.7			
	0x8e8000	0x95f000	0x77000	0x2e8000
	/usr/bin/python2.7			
8	0x7f0a37a8b000	0x7f0a37ab8000	0x2d000	0x0
	/lib/x86_64-linux-gnu/libncursesw.so.5.9			
9	0x7f0a37ab8000	0x7f0a37cb8000	0x200000	0x2d000
	/lib/x86_64-linux-gnu/libncursesw.so.5.9			
10	0x7f0a37cb8000	0x7f0a37cb9000	0x1000	0x2d000
	/lib/x86_64-linux-gnu/libncursesw.so.5.9			
11	0x7f0a37cb9000	0x7f0a37cba000	0x1000	0x2e000
	/lib/x86_64-linux-gnu/libncursesw.so.5.9			
12	0x7f0a37cba000	0x7f0a37ccd000	0x13000	0x0
	/usr/lib/python2.7/lib-dynload/_curses.x86_64-linux-gnu.so			
13	0x7f0a37ccd000	0x7f0a37ecc000	0x1ff000	0x13000
	/usr/lib/python2.7/lib-dynload/_curses.x86_64-linux-gnu.so			
14	0x7f0a37ecc000	0x7f0a37ecd000	0x1000	0x12000
	/usr/lib/python2.7/lib-dynload/_curses.x86_64-linux-gnu.so			
15	0x7f0a37ecd000	0x7f0a37ecf000	0x2000	0x13000
	/usr/lib/python2.7/lib-dynload/_curses.x86_64-linux-gnu.so			
16	0x7f0a38050000	0x7f0a38066000	0x16000	0x0
	/lib/x86_64-linux-gnu/libgcc_s.so.1			
17	0x7f0a38066000	0x7f0a38265000	0x1ff000	0x16000
	/lib/x86_64-linux-gnu/libgcc_s.so.1			
18	0x7f0a38265000	0x7f0a38266000	0x1000	0x15000
	/lib/x86_64-linux-gnu/libgcc_s.so.1			
19	0x7f0a38266000	0x7f0a3828b000	0x25000	0x0
	/lib/x86_64-linux-gnu/libtinfo.so.5.9			
20	0x7f0a3828b000	0x7f0a3848a000	0x1ff000	0x25000
	/lib/x86_64-linux-gnu/libtinfo.so.5.9			
21	[...]			

从地址空间的分配能看到，0x400000-0x6e7000 是第一段有效内存空间，低于这个范围的，都是无效的。上面%rsi 为 0x0，就明显是一个无效的地址空间。

在 ubuntu 上，与 core 相关的说明：<https://wiki.ubuntu.com/Apport>

5 绘制图表

6 幻灯片演示

7 从错误中救赎

8 问题探骊

这里是我在学习软件调试过程遇到的一些问题，有些解决了，有些没有解决，就全部记在这里，供学习完作为思考题。

8.1 动态链接库

1. 程序运行过程中，动态链接库是何时加载到内存空间中？
2. 动态链接库加载后，存储在内存空间中的哪里？
- 3.

8.2 Stack 相关

1. memcpy 引起栈溢出，为什么不会影响 memcpy 函数？
2. memcpy 函数为什么不需要对 rbp 进行保存？

8.3 Heap

1. malloc 申请了堆空间后，如何查看堆的位置？
2. 指针指向一个地址，那么结构体里面的函数代码地址怎么被确定的？