哈爾濱工業大學

实验报告

实 验（五）

题　　目 LinkLab

链接

专　　业 计算机

学　　号 1173000825

班　　级 1703008

学　　生 李大鑫

指 导 教 师 郑贵滨

实 验 地 点

实 验 日 期

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

- 理解链接的作用与工作步骤
- 掌握 ELF 结构、符号解析与重定位的工作过程
- 熟练使用 Linux 工具完成 ELF 分析与修改

1.2 实验环境与工具

1.2.1 硬件环境

- X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

- Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；

1.2.3 开发工具

- Visual Studio 2010 64 位以上；GDB/OBJDUMP；DDD/EDB 等

1.3 实验预习

■ **上实验课前，必须认真预习实验指导书（PPT 或 PDF）**

■ **了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。**

■ **请按顺序写出 ELF 格式的可执行目标文件的各类信息。**

■ 请按照内存地址从低到高的顺序，写出 **Linux** 下 **X64** 内存映像。

■ 请运行**“LinkAddress -u** 学号 姓名**”** 按地址顺序写出各符号的地址、空间。并按照 **Linux** 下 **X64** 内存映像标出其所属各区。

■ 请按顺序写出 **LinkAddress** 从开始执行到 **main** 前/后执行的子程序的名字。**(gcc** 与 **objdump/GDB/EDB)**

第 2 章 实验预习

2.1 请按顺序写出 ELF 格式的可执行目标文件的各类信息（5 分）

ELF 头
段头部表
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
节头部表

2.2 请按照内存地址从低到高的顺序，写出 Linux 下 X64 内存映像。（5 分）

内核内存
用户栈 （运行时 创建）
（栈-向下） . . . （映射区域-向上）
共享库的内存映射区域
. . . （堆-向上）
运行时堆 （由 malloc 创建）

读/写段 （.data,.bss）
只读代码段 （.init,.text,.rodata）
. . .

2.3 请运行“LinkAddress -u 学号 姓名” 按地址循序写出各符号的地址、空间。并按照 Linux 下 X64 内存映像标出其所属各区。

（5 分）

所属	符号、地址、空间（从小到大）
0（NULL）	p5 (nil) 0
只读代码段 （.init , .text , .rodata）	show_pointer 0x55890bd0681a 94047097088026 useless 0x55890bd0684d 94047097088077 main 0x55890bd06858 94047097088088
读/写段 （.data , .bss）	global 0x55890bf0802c 94047099191340 huge array 0x55890bf08040 94047099191360 big array 0x55894bf08040 94048172933184 p2 0x55894ead4670 94048218859120
运行时堆	p1 0x7f442c23c010 139930775044112 p3 0x7f443c819010 139931049627664 p4 0x7f43ec23b010 139929701298192
共享库的内存映射区域	exit 0x7f443c280120 139931043758368 printf 0x7f443c2a1e80 139931043896960 malloc 0x7f443c2d4070 139931044102256 free 0x7f443c2d4950 139931044104528
用户栈 （运行时创建）	. . . argc 0x7ffebe3ea1ac 140732090196396 local 0x7ffebe3ea1b0 140732090196400 argv 0x7ffebe3ea2d8 140732090196696 argv[0] 7ffebe3ec231 argv[1] 7ffebe3ec23f argv[2] 7ffebe3ec242 argv[3] 7ffebe3ec24d argv[0] 0x7ffebe3ec231 140732090204721 ./linkaddress argv[1] 0x7ffebe3ec23f 140732090204735 -u argv[2] 0x7ffebe3ec242 140732090204738 1170300825 argv[3] 0x7ffebe3ec24d 140732090204749 李大鑫

```
------------------------------------------------------------------------

env  0x7ffebe3ea300  140732090196736
env[0]    *env  0x7ffebe3ec257  140732090204759
CLUTTER_IM_MODULE=xim
env[1]    *env  0x7ffebe3ec26d  140732090204781
LS_COLORS=rs=0:d...
env[2]    *env  0x7ffebe3ec859  140732090206297
LC_MEASUREMENT=zh_CN.UTF-8
env[3]    *env  0x7ffebe3ec874  140732090206324
LESSCLOSE=/usr/bin/lesspipe %s %s
env[4]    *env  0x7ffebe3ec896  140732090206358
LC_PAPER=zh_CN.UTF-8
env[5]    *env  0x7ffebe3ec8ab  140732090206379
LC_MONETARY=zh_CN.UTF-8
env[6]    *env  0x7ffebe3ec8c3  140732090206403
XDG_MENU_PREFIX=gnome-
env[7]    *env  0x7ffebe3ec8da  140732090206426
LANG=en_US.UTF-8
env[8]    *env  0x7ffebe3ec8eb  140732090206443
MANAGERPID=1211
env[9]    *env  0x7ffebe3ec8fb  140732090206459
DISPLAY=:0
env[10]   *env  0x7ffebe3ec906  140732090206470
INVOCATION_ID=62acd4e0ef664dabbcc8a0be8a64da03
env[11]   *env  0x7ffebe3ec935  140732090206517
GNOME_SHELL_SESSION_MODE=ubuntu
env[12]   *env  0x7ffebe3ec955  140732090206549
COLORTERM=truecolor
env[13]   *env  0x7ffebe3ec969  140732090206569
USERNAME=linda
env[14]   *env  0x7ffebe3ec978  140732090206584
XDG_VTNR=2
env[15]   *env  0x7ffebe3ec983  140732090206595
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
env[16]   *env  0x7ffebe3ec9ac  140732090206636
LC_NAME=zh_CN.UTF-8
env[17]   *env  0x7ffebe3ec9c0  140732090206656
XDG_SESSION_ID=2
env[18]   *env  0x7ffebe3ec9d1  140732090206673
USER=linda
env[19]   *env  0x7ffebe3ec9dc  140732090206684
DESKTOP_SESSION=ubuntu
env[20]   *env  0x7ffebe3ec9f3  140732090206707
QT4_IM_MODULE=xim
env[21]   *env  0x7ffebe3eca05  140732090206725
TEXTDOMAINDIR=/usr/share/locale/
env[22]   *env  0x7ffebe3eca26  140732090206758
GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/4ff065af_522e_4d38_a63f_4b7bf92d7661
env[23]   *env  0x7ffebe3eca7c  140732090206844
PWD=/home/linda/Desktop/VmShare
env[24]   *env  0x7ffebe3eca9c  140732090206876
HOME=/home/linda
env[25]   *env  0x7ffebe3ecaad  140732090206893
JOURNAL_STREAM=9:41219
env[26]   *env  0x7ffebe3ecac4  140732090206916
TEXTDOMAIN=im-config
env[27]   *env  0x7ffebe3ecad9  140732090206937
SSH_AGENT_PID=1324
env[28]   *env  0x7ffebe3ecaec  140732090206956
QT_ACCESSIBILITY=1
env[29]   *env  0x7ffebe3ecaff  140732090206975
XDG_SESSION_TYPE=x11
env[30]   *env  0x7ffebe3ecb14  140732090206996
XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share:/usr/share:/var/lib/snapd/desktop
env[31]   *env  0x7ffebe3ecb67  140732090207079
XDG_SESSION_DESKTOP=ubuntu
env[32]   *env  0x7ffebe3ecb82  140732090207106
LC_ADDRESS=zh_CN.UTF-8
env[33]   *env  0x7ffebe3ecb99  140732090207129
DBUS_STARTER_ADDRESS=unix:path=/run/user/1000/bus,guid=e141dacb1ecb888eea4b95b85bf7478c
env[34]   *env  0x7ffebe3ecbf1  140732090207217
LC_NUMERIC=zh_CN.UTF-8
env[35]   *env  0x7ffebe3ecc08  140732090207240
GTK_MODULES=gail:atk-bridge
env[36]   *env  0x7ffebe3ecc24  140732090207268
PAPERSIZE=a4
env[37]   *env  0x7ffebe3ecc31  140732090207281
WINDOWPATH=2
env[38]   *env  0x7ffebe3ecc3e  140732090207294
TERM=xterm-256color
env[39]   *env  0x7ffebe3ecc52  140732090207314
VTE_VERSION=5202
env[40]   *env  0x7ffebe3ecc63  140732090207331
SHELL=/bin/bash
env[41]   *env  0x7ffebe3ecc73  140732090207347
QT_IM_MODULE=ibus
```

```
env[42]    *env  0x7ffebe3ecc85  140732090207365
XMODIFIERS=@im=ibus
env[43]    *env  0x7ffebe3ecc99  140732090207385
IM_CONFIG_PHASE=2
env[44]    *env  0x7ffebe3eccab  140732090207403
DBUS_STARTER_BUS_TYPE=session
env[45]    *env  0x7ffebe3eccc9  140732090207433
XDG_CURRENT_DESKTOP=ubuntu:GNOME
env[46]    *env  0x7ffebe3eccea  140732090207466
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
env[47]    *env  0x7ffebe3ecd1e  140732090207518
GNOME_TERMINAL_SERVICE=:1.72
env[48]    *env  0x7ffebe3ecd3b  140732090207547
SHLVL=1
env[49]    *env  0x7ffebe3ecd43  140732090207555
XDG_SEAT=seat0
env[50]    *env  0x7ffebe3ecd52  140732090207570
LANGUAGE=en_US
env[51]    *env  0x7ffebe3ecd61  140732090207585
LC_TELEPHONE=zh_CN.UTF-8
env[52]    *env  0x7ffebe3ecd7a  140732090207610
GDMSESSION=ubuntu
env[53]    *env  0x7ffebe3ecd8c  140732090207628
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
env[54]    *env  0x7ffebe3ecdb8  140732090207672
LOGNAME=linda
env[55]    *env  0x7ffebe3ecdc6  140732090207686
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus,guid=e141dacb1ecb888eea4b95b85bf7478c
env[56]    *env  0x7ffebe3ece22  140732090207778
XDG_RUNTIME_DIR=/run/user/1000
env[57]    *env  0x7ffebe3ece41  140732090207809
XAUTHORITY=/run/user/1000/gdm/Xauthority
env[58]    *env  0x7ffebe3ece6a  140732090207850
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
env[59]    *env  0x7ffebe3ece97  140732090207895
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
env[60]    *env  0x7ffebe3eceff  140732090207999
LC_IDENTIFICATION=zh_CN.UTF-8
env[61]    *env  0x7ffebe3ecf1d  140732090208029
PS1=\[\033[0;32m\]\u:\W>\[\033[0m\]
env[62]    *env  0x7ffebe3ecf41  140732090208065
SESSION_MANAGER=local/ubuntu:@/tmp/.ICE-unix/1240,unix/ubuntu:/tmp/.ICE-unix/1240
env[63]    *env  0x7ffebe3ecf93  140732090208147
LESSOPEN=| /usr/bin/lesspipe %s
env[64]    *env  0x7ffebe3ecfb3  140732090208179
GTK_IM_MODULE=ibus
env[65]    *env  0x7ffebe3ecfc6  140732090208198
LC_TIME=zh_CN.UTF-8
env[66]    *env  0x7ffebe3ecfda  140732090208218
_=./linkaddress
```

2.4 请按顺序写出 LinkAddress 从开始执行到 main 前/后执行的子程序的名字。（gcc 与 objdump/GDB/EDB）（5 分）

时间段	程序
Main 函数执行前	Ld-2.27.so!_dl_start Ld-2.27.so!_dl_init Libc-2.27.so!_cxa_atexit Linkaddress!_init Linkaddress!_register_tm_clones Libc-2.27.so!_setjmp Libc2.27.so!__sigsetjmp Libc2.27.so!__sigjmpsave
Main 函数执行之后	Linkaddress!puts@plt

	Linkaddress!useless@plt Linkaddress!showpointer@plt malloc Linkaddress!.plt Libc-2.27.so!exit

第 3 章 各阶段的原理与方法

每阶段 40 分，phasex.o 20 分，分析 20 分，总分不超过 80 分

（请先移步阅读）
* edb 操作基础知识
* readelf 使用
* 在 EditHex 中定位节 Section

3.1 阶段 1 的分析

程序运行结果截图：

```
linda:linklab-1170300825>gcc -m32 -o linkbomb main.o phase
1.o
linda:linklab-1170300825>ls
linkbomb      main.o    phase1.o  phase3.o  phase5.o
linkbomb.txt  main.txt  phase2.o  phase4.o
linda:linklab-1170300825>./linkbomb
1170300825
```

分析与设计的过程：

首先

直接将 main.o 编译，得到 linkbomb，运行之后屏幕显示：

```
linda:linklab-1170300825>gcc -m32 -o linkbomb main.o
linda:linklab-1170300825>./linkbomb
Welcome to this small lab of linking. To begin lab, please
 link the relevant object module(s) with the main module.
```

通过查看 main 的反汇编可以得出 main 函数的主要逻辑：判断 phase 是否为空，如果为空则打印上述输出字串，如果不为空则调用 phase 函数。

尝试将 main.o 与 phase1.o 进行链接，运行 linkbomb 后屏幕输出为：

```
linda:linklab-1170300825>./linkbomb
GZxxO5hnLEAFo5Pi5dNf5usR2cLZoMCi1WinGvJ 9UnW7tRuAFnqroXx
02nluanC6 axfj86v1K0XkmnXXE75APVs tVDg  GiehghRaNI
```

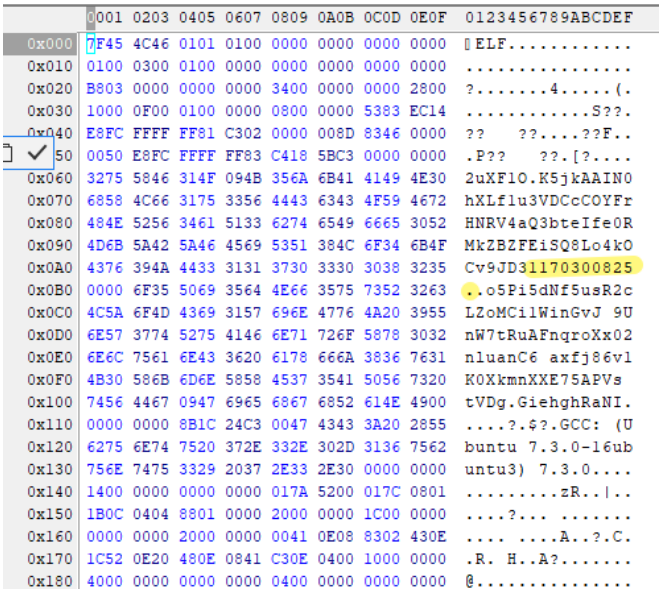
可以看到这是一串没有什么特殊意义字符串。我们的目标就是将该字符串的前部替换为我们的学号，最终使屏幕输出我们的学号。

分析一下

printf(“%s\n”,s)输出函数最终会被优化为 puts(s)，s 为字符串常数因此被保存在.data 节数据段中。因此我们只需要在 phase1 中查找到相应的字符串更改为学号就行了

更改代码

利用 HexEdit 打开 phase1（HexEdit 可以直接看到字符串的内容，简化了定位的操作），将学号“1170300825\0”填入到指定位置，如下截图：

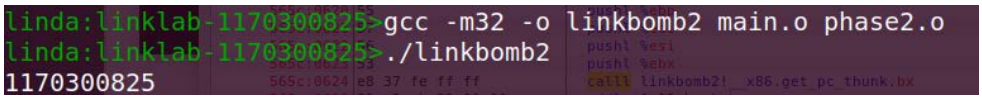


	0001	0203	0405	0607	0809	0A0B	0C0D	0E0F	0123456789ABCDEF
0x000	7F45	4C46	0101	0100	0000	0000	0000	0000	▯ELF............
0x010	0100	0300	0100	0000	0000	0000	0000	0000	....
0x020	B803	0000	0000	0000	3400	0000	0000	2800	?.......4.....(.
0x030	1000	0F00	0100	0000	0800	0000	5383	EC14	..........S??.
0x040	E8FC	FFFF	FF81	C302	0000	008D	8346	0000	?? ??....??F..
0x050	0050	E8FC	FFFF	FF83	C418	5BC3	0000	0000	.P?? ??.[?....
0x060	3275	5846	314F	094B	356A	6B41	4149	4E30	2uXF1O.K5jkAAIN0
0x070	6858	4C66	3175	3356	4443	6343	4F59	4672	hXLf1u3VDCcCOYFr
0x080	484E	5256	3461	5133	6274	6549	6665	3052	HNRV4aQ3bteIfe0R
0x090	4D6B	5A42	5A46	4569	5351	384C	6F34	6B4F	MkZBZFEiSQ8Lo4kO
0x0A0	4376	394A	4433	3131	3730	3330	3038	3235	Cv9JD31170300825
0x0B0	0000	6F35	5069	3564	4E66	3575	7352	3263	..o5Pi5dNf5usR2c
0x0C0	4C5A	6F4D	4369	3157	696E	4776	4A20	3955	LZoMCi1WinGvJ 9U
0x0D0	6E57	3774	5275	4146	6E71	726F	5878	3032	nW7tRuAFnqroXx02
0x0E0	6E6C	7561	6E43	3620	6178	666A	3836	7631	nluanC6 axfj86v1
0x0F0	4B30	586B	6D6E	5858	4537	3541	5056	7320	K0XkmnXXE75APVs
0x100	7456	4467	0947	6965	6867	6852	614E	4900	tVDg.GiehghRaNI.
0x110	0000	0000	8B1C	24C3	0047	4343	3A20	2855	....?.$?.GCC: (U
0x120	6275	6E74	7520	372E	332E	302D	3136	7562	buntu 7.3.0-16ub
0x130	756E	7475	3329	2037	2E33	2E30	0000	0000	untu3) 7.3.0....
0x140	1400	0000	0000	0000	017A	5200	017C	0801	.........zR..|..
0x150	1B0C	0404	8801	0000	2000	0000	1C00	0000	....?... ....
0x160	0000	0000	2000	0000	0041	0E08	8302	430E	....A..?.C.
0x170	1C52	0E20	480E	0841	C30E	0400	1000	0000	.R. H..A?.......
0x180	4000	0000	0000	0000	0400	0000	0000	0000	@...............

\0 是全 0 的一个字节。

3.2 阶段 2 的分析

程序运行结果截图：



分析与设计的过程：

（不想看我多哔哔，可以直接到这里）

分析 ELF 文件结构

如下：

ELF 头
段头部表
.init
.text
.rodata
.data
.bss
.symtab
.debug
.line
.strtab
节头部表

do_phase 函数结构如下：

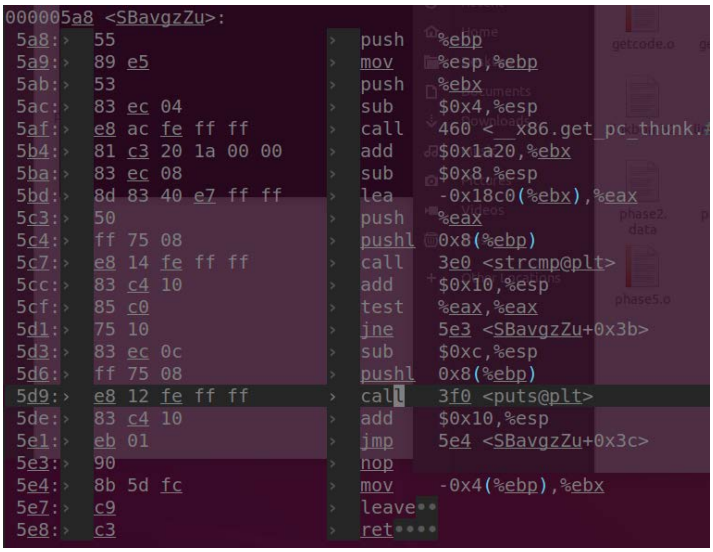
```
static void OUTPUT_FUNC_NAME( const char *id )    // 该函数名对每名学生均不同
{
    if( strcmp(id,MYID) != 0 ) return;
    printf("%s\n", id);
}
void do_phase() {
    // 在代码节中预留存储位置供学生插入完成功能的必要指令
    asm( “nop\n\tnop\n\tnop\n\tnop\n\tnop\n\tnop\n\tnop\n\tnop\n\tnop\n\t…” );
}
```

代码分析

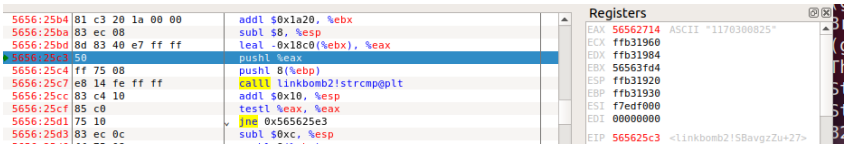
对 ELF 文件的 Section 部分进行解析

objdump –s –d phase2.o > phase2.txt

在.text 节中找到指定的输出函数位置（注意，puts，strcmp 等函数是要在链接重定向完成之后才能确定地址，在反汇编代码中显示出来。这一步对应命令：gcc –m32 –o linkbomb2 main.o phase2.o ， objdump –s –d linkbomb2 > linkbomb2.txt）：



通过 edb 在代码中定位 strcmp 函数的位置，发现此时 eax 指向学号的字符串，在执行 strcmp 之前向栈中压入了两个参数，显然一个是 MYID 一个则是函数传入的参数，因此我们目标就是在 do_phase 的 nop 中写入执行压栈和相对位置跳转（call）到输出函数 SBavgzZu 的逻辑，我们需要压栈的值是 MYID 的地址。同时因为这里的 MYID 地址是变化的，我们是无法直接获得压栈的地址值的。



解决问题

a. 如何进行相对位置调用：在汇编指令中 call 指令就是根据 PC 与跳转目标指令的相对差来进行修改 PC 值从而完成跳转的，注意，这里的 PC 值指的是 call 这一条完整指令的下一条指令的地址值。

b. 如何获得目标字符串 MYID 的地址：通过观察 do_phase 函数，我们发现一个名为 x86.get_PC_thunk.ax 的 call 调用，跟进发现其中的执行逻辑是：

询问搜索引擎后得知，from StackOverFlow caf：

This call is used in position-independent code on x86. It loads the position of the code into the `%ebx` register, which allows global objects (which have a fixed offset from the code) to be accessed as an offset from that register.

也就是说执行完这个函数之后我们就可以通过被写的寄存器来通过偏移量访问 global 类型，这也恰好解决了我们的问题。这是如何实现的呢？call 调用之后此时 PC 指向下一条指令，同时将这条指令的地址压入栈中，进入 x86.get_PC_thunk.ax 之后，将栈顶的值赋值给指定的寄存器（后缀 ax 代表是%eax），这时候指定寄存器中就放着我们可以用来相对寻址的下一条指令的位置了。

```
eax, edx ▶ 5664:75f6 55              pushl %ebp
           5664:75f7 89 e5           movl %esp, %ebp
           5664:75f9 e8 b3 ff ff ff  calll linkbomb2!__x86.get_pc_thunk.ax
           5664:75fe 05 d6 19 00 00  addl $0x19d6, %eax
           5664:7603 90              nop
           5664:7604 90              nop
```

其实，

call linkbomb2!__x86.get_pc_thunk.ax

addl $0x19d6, %eax

实现了将 %eax 指向 _GLOBAL_OFFSET_TABLE_ 的功能，_GLOBAL_OFFSET_TABLE_用来定位 global 变量的真实（运行时）地址，对于上图的

b3 ff ff ff 和 d6 19 00 00

都是在链接过程中经过重定位确定了值，没有链接之前是这样滴：

```
e8 fc ff ff ff      call   45 <do_phase+0x4>
05 01 00 00 00      add    $0x1,%eax
```

链接之后才是这样滴：

```
5ec: e8 28 00 00 00      call   619 <__x86.get_pc_thunk.ax>
5f1: 05 e3 19 00 00      add    $0x19e3,%eax
```

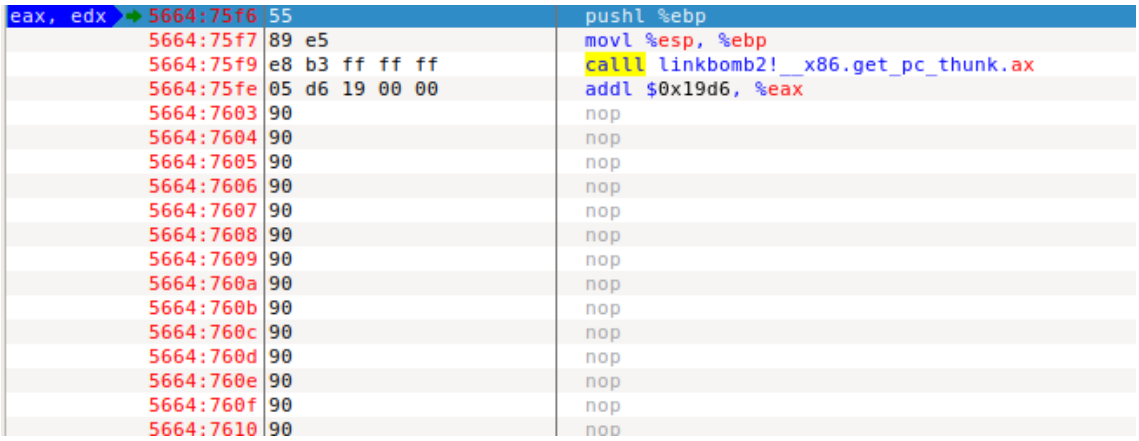
。

得到_GLOBAL_OFFSET_TABLE_地址之后，加上指定偏移量就可以得到特定的 global 变量。

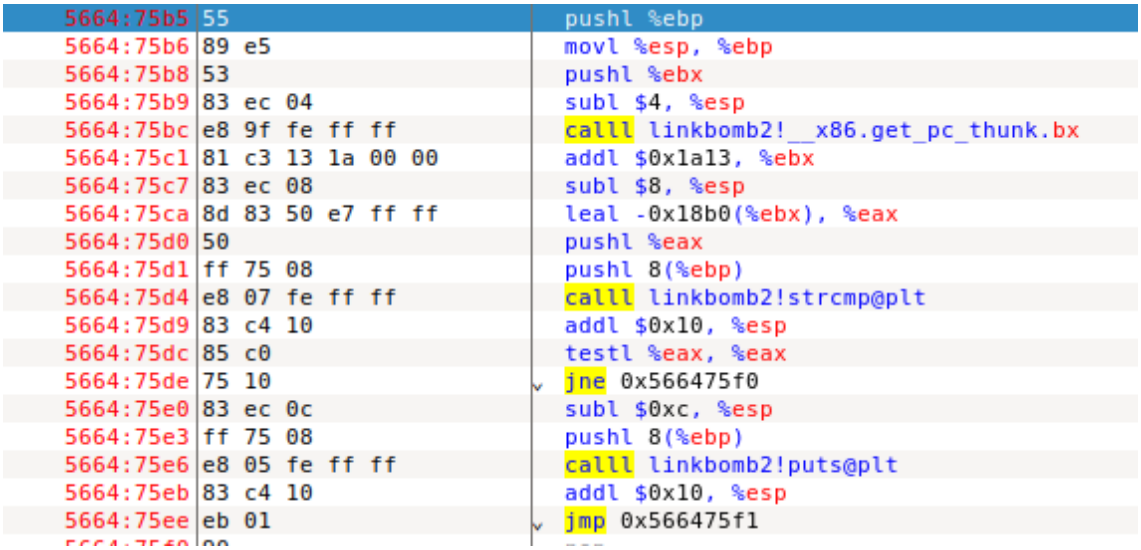
c. 相对寻址：

我们先来观察 do_phase 函数和输出函数 SbavgzZu 的反汇编代码，将没有修改的 phase2.o 链接，对 linkbomb2 反汇编，如下：

Do_phase:



SbavgzZu 函数：



在 SbavgzZu 函数中，经过：

Calll linkbomb2!__x86.get_pc_thunk.bx

addl $0x1a13,%ebx

leal -0x18b0(%ebx),%eax

前两步将%ebx 指向_GLOBAL_OFFSET_TABLE_，后一步也是一个重定向之后确定的值，没有重定向之前是这样滴：

重定向之后%eax 指向了.rodata，就是 MYID。

其实这里也是一个重定向，是链接之后确定的，目的地址是.rodata。

所以我们只需要将%eax 也指向.rodata 就可以了。在 do_phase 的 nop 之前 eax 也已经指向了_GLOBAL_OFFSET_TABLE_，所以只需要

leal -0x18b0(%eax),%eax

就在 do_phase 中也使%eax 指向了.rodata，将之作为参数压栈，然后 call 指令执行相对跳转，最后不要忘记使 eax 出栈“恢复现场”。

修改后汇编代码如下：

```
565c:d5f6 55                    pushl %ebp
565c:d5f7 89 e5                 movl %esp, %ebp
565c:d5f9 e8 b3 ff ff ff        calll linkbomb2!__x86.get_pc_thunk.ax
565c:d5fe 05 d6 19 00 00        addl $0x19d6, %eax
565c:d603 8d 80 50 e7 ff ff     leal -0x18b0(%eax), %eax
565c:d609 50                    pushl %eax
565c:d60a e8 a6 ff ff ff        calll linkbomb2!SBavgzZu
565c:d60f 58                    popl %eax
```

插入代码

操作总结

输出函数 SBavgzZu 的反汇编：

```
5664:75b5 55                    pushl %ebp
5664:75b6 89 e5                 movl %esp, %ebp
5664:75b8 53                    pushl %ebx
5664:75b9 83 ec 04              subl $4, %esp
5664:75bc e8 9f fe ff ff        calll linkbomb2!__x86.get_pc_thunk.bx
5664:75c1 81 c3 13 1a 00 00     addl $0x1a13, %ebx
5664:75c7 83 ec 08              subl $8, %esp
5664:75ca 8d 83 50 e7 ff ff     leal -0x18b0(%ebx), %eax
5664:75d0 50                    pushl %eax
5664:75d1 ff 75 08              pushl 8(%ebp)
5664:75d4 e8 07 fe ff ff        calll linkbomb2!strcmp@plt
5664:75d9 83 c4 10              addl $0x10, %esp
5664:75dc 85 c0                 testl %eax, %eax
5664:75de 75 10                 jne 0x566475f0
5664:75e0 83 ec 0c              subl $0xc, %esp
5664:75e3 ff 75 08              pushl 8(%ebp)
5664:75e6 e8 05 fe ff ff        calll linkbomb2!puts@plt
5664:75eb 83 c4 10              addl $0x10, %esp
5664:75ee eb 01                 jmp 0x566475f1
```

看一下自己输出函数中 leal [数字 x](%ebx),%eax 是什么样的，（在我的里面数字 x 是-0x18b0,看看上面那张图标绿的那句），然后用数字 x 来替换

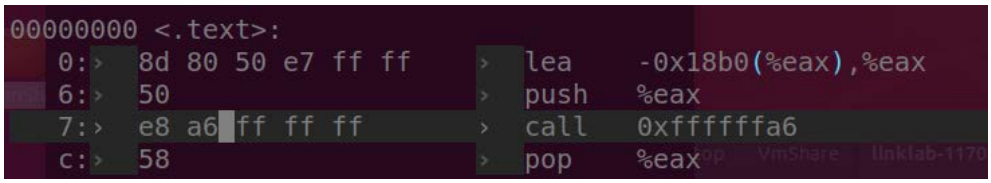
```
1 lea -0x18b0(%eax),%eax
2 push %eax
3 call 0xffffffa6
4 pop %eax
```

的第一句中的-0x18b0（正常人应该都能看得懂吧。。。），得到自己应该插入的汇编代码，保存在 getcode.s。

```
gcc -m32 -c getcode.s

objdump -d getcode.o > getcode.txt
```

得到：



（反编译之后 call 后面的值变化的话，自己修改回来）

剩下的就是用 HexEdit 更改二进制了，二进制在上图左边。

完事儿，(´ゕゃゕ)σ。

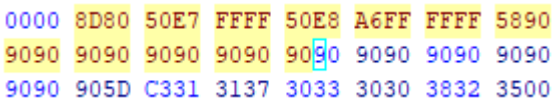
获得 16 进制代码的方法：

1） 编写汇编代码入 getcode.s

2） gcc –m32 –c getcode.s 获得 getcode.o

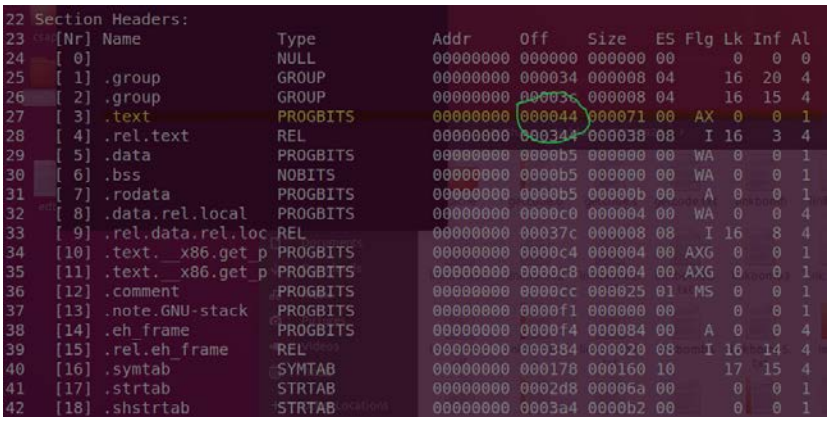
3） objdump –d getcode.o > getcode.txt 将反汇编代码放入 getcode.txt 中，就可以得到上图的反汇编代码了，左侧就是相对应的 16 进制代码。

使用 HexEdit 修改 phase2.o:

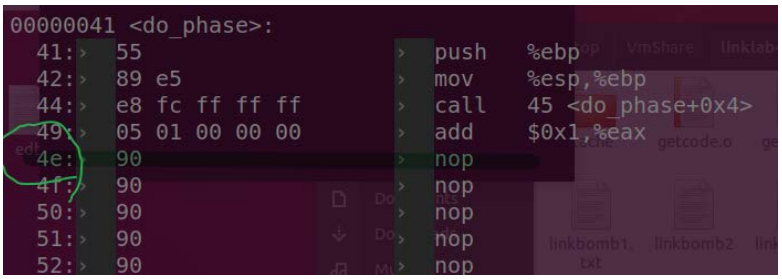


找到插入位置的方法（其实只要看见一片 90 无脑覆盖就行了）：

1） readelf -a phase2.o > phase2.elf 得到.text 节相对于 elf 文件头的偏移地址 0x44：



2） objdump -s -d phase2.o > phase2.txt 得到第一个 nop 相对于.text 节的偏移量 0x4e：

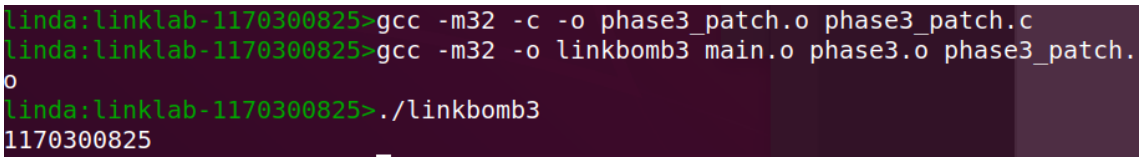


3） 计算第一个 nop 在 HexEdit 中的地址 0x44+0x4e=0x92

4） 在 HexEdit 中寻找其他位置的操作类似。

3.3 阶段 3 的分析

程序运行结果截图：



分析与设计的过程：

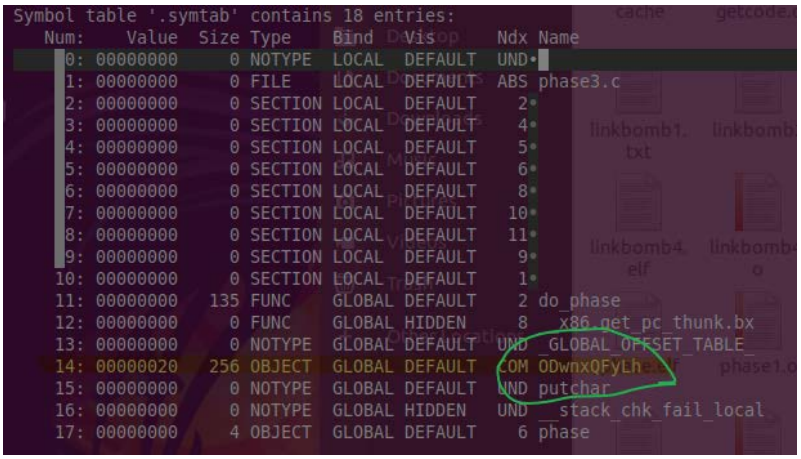
获得数组名称

使用 readelf 确定 PPT 中所谓的 PHASE3_CODEBOOK 数组对应的名

称。

readelf -a phase3.o > phase3.txt

得到：



```
Symbol table '.symtab' contains 18 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS phase3.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    2
     3: 00000000     0 SECTION LOCAL  DEFAULT    4
     4: 00000000     0 SECTION LOCAL  DEFAULT    5
     5: 00000000     0 SECTION LOCAL  DEFAULT    6
     6: 00000000     0 SECTION LOCAL  DEFAULT    8
     7: 00000000     0 SECTION LOCAL  DEFAULT   10
     8: 00000000     0 SECTION LOCAL  DEFAULT   11
     9: 00000000     0 SECTION LOCAL  DEFAULT    9
    10: 00000000     0 SECTION LOCAL  DEFAULT    1
    11: 00000000   135 FUNC    GLOBAL DEFAULT    2 do_phase
    12: 00000000     0 FUNC    GLOBAL HIDDEN     8 __x86.get_pc_thunk.bx
    13: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
    14: 00000020   256 OBJECT  GLOBAL DEFAULT  COM QDwnxQFyLh
    15: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND putchar
    16: 00000000     0 NOTYPE  GLOBAL HIDDEN   UND __stack_chk_fail_local
    17: 00000000     4 OBJECT  GLOBAL DEFAULT    6 phase
```

可见，PHASE3_CODEBOOK 的实际名称是 QDwnxQFyLh。

这是因为 QDwnxQFyLh 已经在 phase3.o 中有了全局弱定义，所以必然存在于符号表.symtab 中。

分析 do_phase 结构

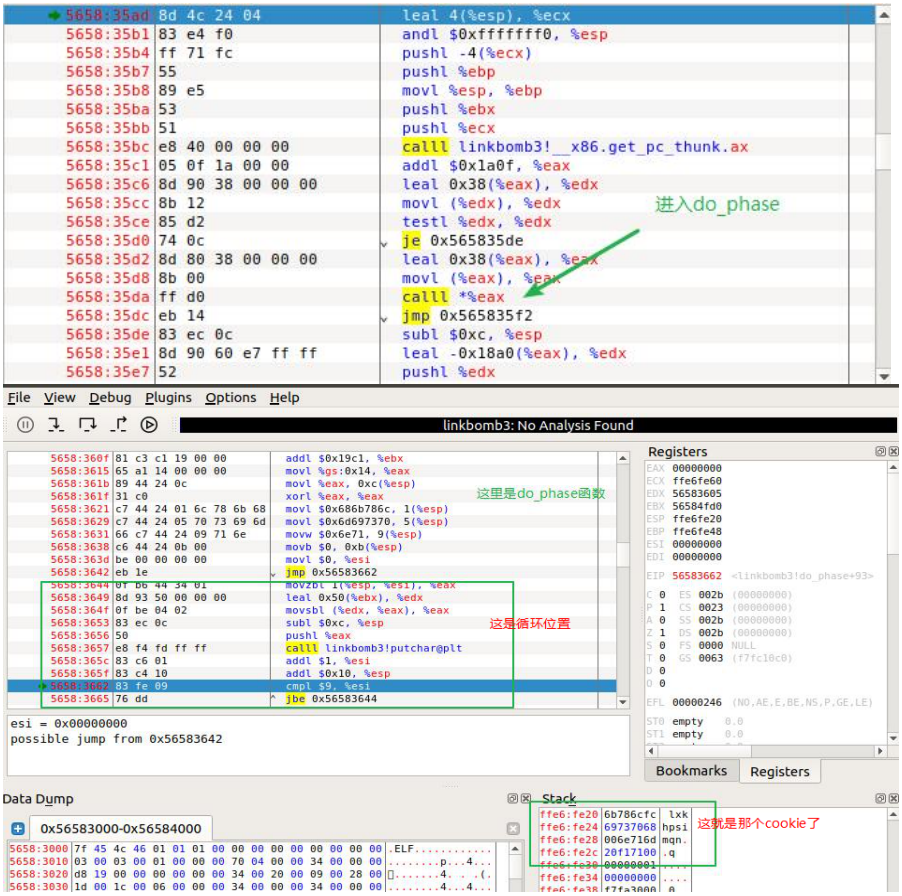
```
char PHASE3_CODEBOOK[256];
void do_phase(){
    const char char cookie[] = PHASE3_COOKIE;
    for( int i=0; i<sizeof(cookie)-1; i++ )
        printf( "%c", PHASE3_CODEBOOK[ (unsigned char)(cookie[i]) ] );
    printf( "\n" );
}
```

可见最终输出的字符存储在 QDwnxQFyLh 数组中，最终输出是使用 cookie 这个字符数组来进行寻址，cookie 是已知不变的，所以我们的工作是：得到 cookie 数组，根据 cookie 数组构造 QDwnxQFyLh 数组，要求按照 cookie 的索引顺序在 QDwnxQFyLh 中依次填入自己的学号。

获得 cookie 数组

gcc -o linkbomb3 main.o phase3.o

得到 linkbomb3，使用 edb 运行 linkbomb3，点击运行，运行到 main 函数，然后单步运行到 do_phase 中的循环位置，如下：



得到我的 cookie 数组是 lxkhpsimqn。

构造字符串

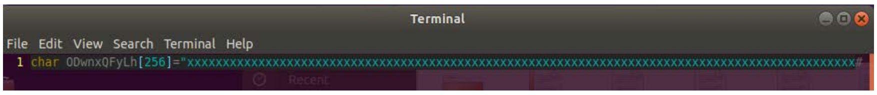
因此我们只需要在字符数组 cookie 的字符所指向的 OdwnxQFyLh 数组的指定位置处 按顺序 填上自己的学号即可。

解释一下：比如我的 cookie 数组是 lxkhpsimqn，正好 10 个对应我的 10 位学号，比如我要填第一个，‘1’对应 ASCII 码的 108，所以我要使 OdwnxQFyLh[108]=’1’。剩下的类似。其他地方随便填，我就用 x 填上了。

字符串构造如下：

char OdwnxQFyLh[256]=xxx...xxxxxxxxxoxxxxxxx00x7185x32x0xxx1xxxxxxoxxxxxxxxx...xxx

剩下的工作：在 phase3_patch.c 中定义 OdwnxQFyLh 数组，



~~（如果不爽 换种字符数组初始化方法就是了）~~

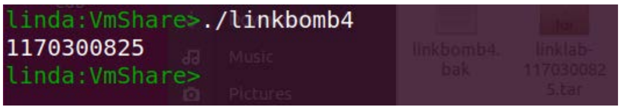
然后：

gcc -m32 -c phase3_patch.c

gcc -m32 -o linkbomb3 main.o phase3.o phase3_patch.o

3.4 阶段 4 的分析

程序运行结果截图：



（注：因为这个题个人感觉 PPT 上叙述的做法有点冲突，所以我是直接修改的 linkbomb4，所以在测试的时候直接运行 linkbomb4 即可）

分析与设计的过程：

了解框架：

■ **phase4.c程序框架**

```
void do_phase()
{
    const char cookie[] = PHASE4_COOKIE;
    char c;
    for ( int i = 0; i < sizeof(cookie)-1; i++ )
   {
        c = cookie[i];
        switch (c)
        {
            // 每个学生的映射关系和case顺序建议不一样
            case 'A': { c = 48; break; }
            case 'B': { c = 121; break; }
             ...
            case 'Z': { c = 93; break; }
        }
       printf("%c", c);
   }
}
```

实验步骤

1）通过分析do_phase函数的反汇编程序获知COOKIE字符串（保存于栈帧中的局部字符数组中）的组成内容

2）确定switch跳转表在.rodata节中的偏移量

3）定位COOKIE中每一字符'c'在switch跳转表中的对应表项（索引为'c'-0x41），将其值设为输出目标学号中对应字符的case首指令的偏移量

分析反汇编代码：

a) 通过 edb 获得 cookie 的值，可以得到：

```
ffc1:8ce0 554b58fc  XKU
ffc1:8ce4 4654594a JYTF
ffc1:8ce8 00484d57 WMH.
```

b) 定位 switch 主体代码：

i.

```
565e:9639 8d 55 e9          leal -0x17(%ebp), %edx
565e:963c 8b 45 e4          movl -0x1c(%ebp), %eax
565e:963f 01 d0             addl %edx, %eax
565e:9641 0f b6 00          movzbl (%eax), %eax
565e:9644 88 45 e3          movb %al, -0x1d(%ebp)
565e:9647 0f be 45 e3       movsbl -0x1d(%ebp), %eax
565e:964b 83 e8 41          subl $0x41, %eax
565e:964e 83 f8 19          cmpl $0x19, %eax
565e:9651 0f 87 b5 00 00 00 ja 0x565e970c
565e:9657 c1 e0 02          shll $2, %eax
565e:965a 8b 84 18 94 e8 ff ff movl -0x176c(%eax, %ebx), %eax
565e:9661 01 d8             addl %ebx, %eax
565e:9663 ff e0             jmpl *%eax
```

c）分析

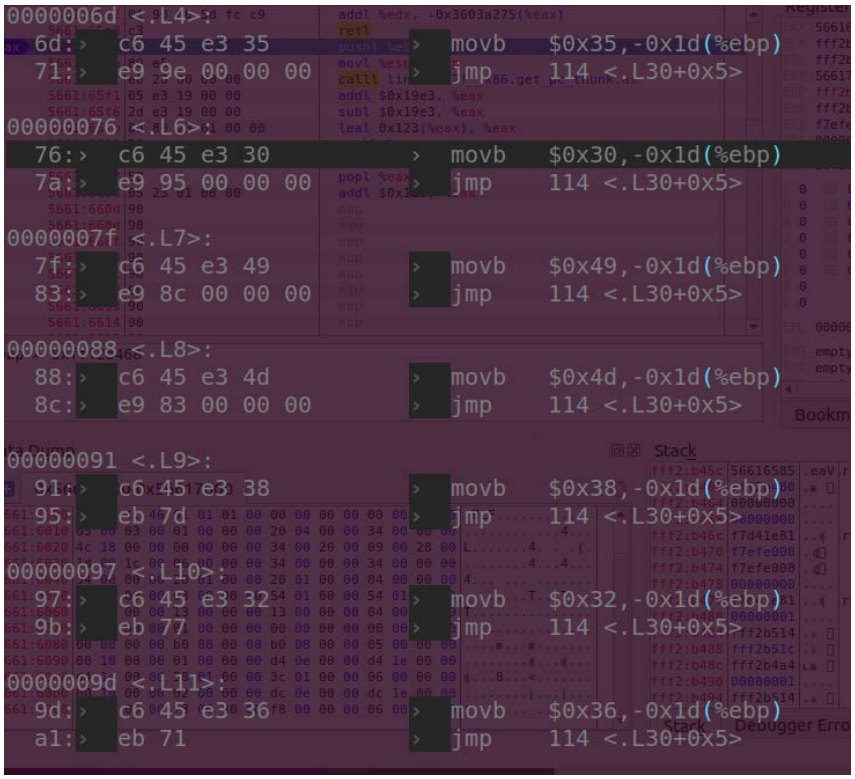
a) 可以看出这里代码块的主要功能就是计算地址值到 eax，然后跳转到 eax 所指向的地址。Eax 的计算公式为（（%eax-0x41）<<2+%ebx-0x176c），外面的一个括号代表寻址，这里是将 cookie 值作为索引映射到 switch 跳转表，需要注意的是 switch 跳转表保存在.roddata 中，同时，当只有 phase4.o 的时候 switch 的跳转表是不能确定的，只有当将 main.o 和 phase4.o 链接之后才能确定跳转表的值。当程序运行时，程序先拿到跳转表的值，跳转表中存放着相对偏移位置，通过 %eax=%ebx+偏移量 获得存储在.text 段中共的 case 代码段地址，之后跳转执行。

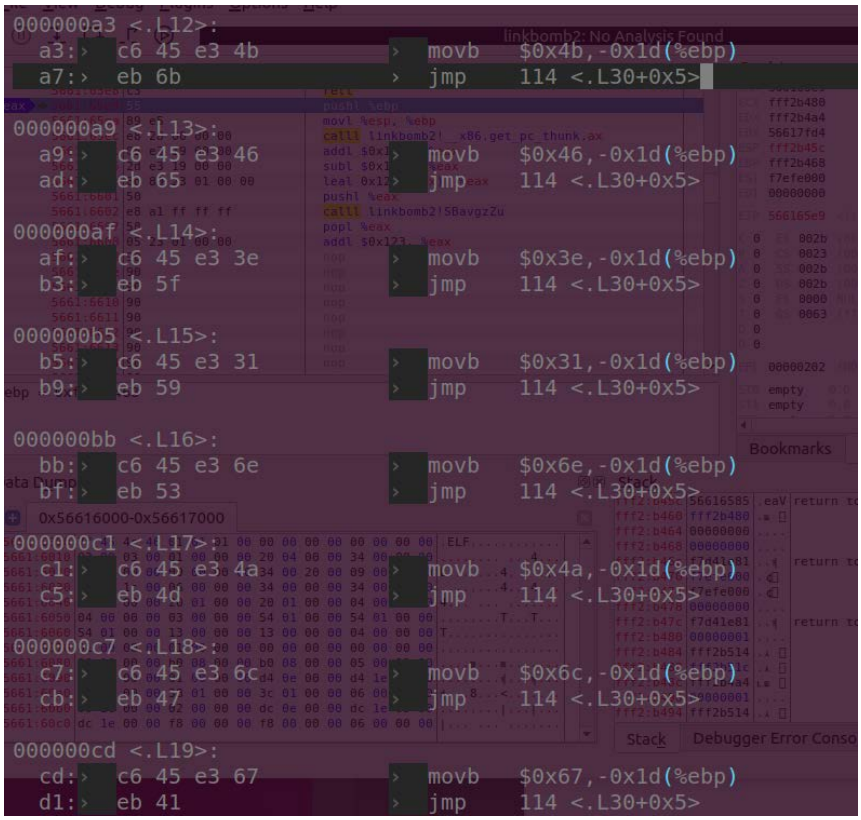
b) 根据.rodata 段的性质我们可以确定我们的破解策略：将 main.o 和 phase4.o 进行链接成为 linkbomb4，通过 HexEdit

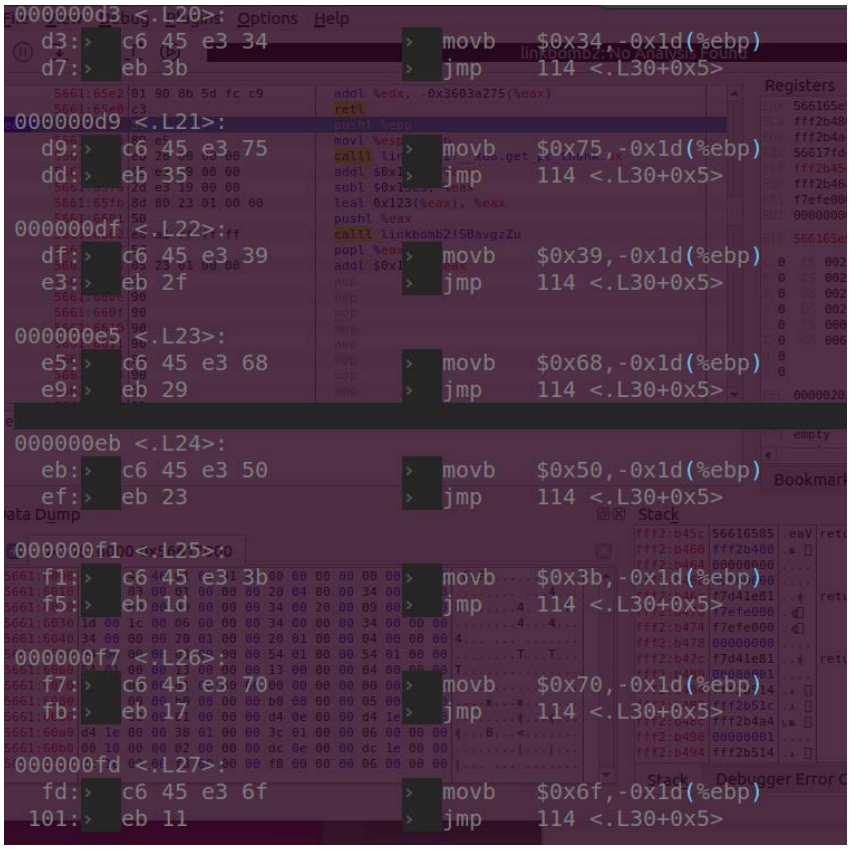
程序更改 linkbomb4 程序的 rodata 段中 switch 的跳转表为满足 cookie 映射先后顺序。（注意，每次连接过程可能会产生不同的跳转表）。PS:注意到 PPT 中是让修改 phase4.o 的 rodata 节，但是这里的 rodata 是在链接之后才能确定的，如果修改 phase4.o 的 rodata 链接之后会被覆盖，因此只能修改 linkbomb4。

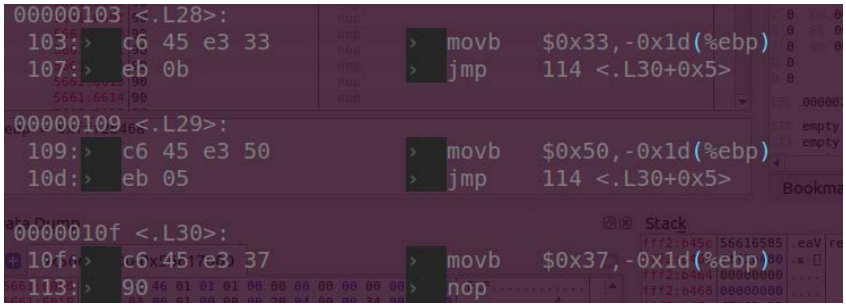
构造答案：

c) 首先通过 edb 查看没有修改过的 linkbomb4，发现第一个 cookie 值“X”对应的跳转表中的偏移量为 0xffffe738，对应到 case 代码块执行输出 0x33，查看 phase4 的反汇编代码：

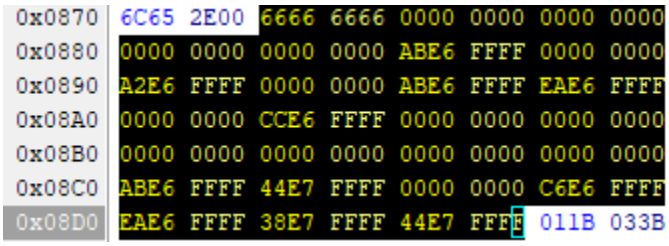



000000a3 <.L12>:
a3: c6 45 e3 4b movb $0x4b,-0x1d(%ebp)
a7: eb 6b jmp 114 <.L30+0x5>
000000a9 <.L13>:
a9: c6 45 e3 46 movb $0x46,-0x1d(%ebp)
ad: eb 65 jmp 114 <.L30+0x5>
000000af <.L14>:
af: c6 45 e3 3e movb $0x3e,-0x1d(%ebp)
b3: eb 5f jmp 114 <.L30+0x5>
000000b5 <.L15>:
b5: c6 45 e3 31 movb $0x31,-0x1d(%ebp)
b9: eb 59 jmp 114 <.L30+0x5>
000000bb <.L16>:
bb: c6 45 e3 6e movb $0x6e,-0x1d(%ebp)
bf: eb 53 jmp 114 <.L30+0x5>
000000c1 <.L17>:
c1: c6 45 e3 4a movb $0x4a,-0x1d(%ebp)
c5: eb 4d jmp 114 <.L30+0x5>
000000c7 <.L18>:
c7: c6 45 e3 6c movb $0x6c,-0x1d(%ebp)
cb: eb 47 jmp 114 <.L30+0x5>
000000cd <.L19>:
cd: c6 45 e3 67 movb $0x67,-0x1d(%ebp)
d1: eb 41 jmp 114 <.L30+0x5>
Bookmarks
Stack
Debugger Error Conso


000000d3 <.L20>:
d3: c6 45 e3 34 movb $0x34,-0x1d(%ebp)
d7: eb 3b jmp 114 <.L30+0x5>
000000d9 <.L21>:
d9: c6 45 e3 75 movb $0x75,-0x1d(%ebp)
dd: eb 35 jmp 114 <.L30+0x5>
000000df <.L22>:
df: c6 45 e3 39 movb $0x39,-0x1d(%ebp)
e3: eb 2f jmp 114 <.L30+0x5>
000000e5 <.L23>:
e5: c6 45 e3 68 movb $0x68,-0x1d(%ebp)
e9: eb 29 jmp 114 <.L30+0x5>
000000eb <.L24>:
eb: c6 45 e3 50 movb $0x50,-0x1d(%ebp)
ef: eb 23 jmp 114 <.L30+0x5>
000000f1 <.L25>:
f1: c6 45 e3 3b movb $0x3b,-0x1d(%ebp)
f5: eb 1d jmp 114 <.L30+0x5>
000000f7 <.L26>:
f7: c6 45 e3 70 movb $0x70,-0x1d(%ebp)
fb: eb 17 jmp 114 <.L30+0x5>
000000fd <.L27>:
fd: c6 45 e3 6f movb $0x6f,-0x1d(%ebp)
101: eb 11 jmp 114 <.L30+0x5>
Registers
Stack



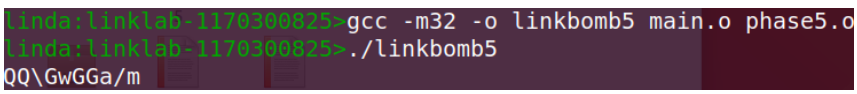
b）通过 phase4 反汇编代码我们可以确定 case 代码块之间相对位置，通过输出 0x33 的代码块的跳转表偏移量，我们可以得到所有的 case 类在跳转表里面对应的偏移量，我们选择输出指定字母串为学号的 case 块的跳转表偏移量 **按 cookie 映射顺序与位置** 填入到.rodata 跳转表之中。说起来有点儿绕，我们不妨看一下截图：



截图说明：前面的 6 只是为了标识开始而已；其中填入 0 的都是 cookie 映射不到的；以 0xffffe738 为例，偏移量对应 cookie-“X”（填入到了跳转表中‘x’映射到的位置），而 0xffffe738 是我们填入跳转表中的值，凭借跳转表程序跳转到 switch 的对应 case 语句，通过反汇编代码我们知道 case 之间的相对位置，在 0xffffe738 基础上进行加减就可以获得其他 case 块对应的跳转表值。

3.5 阶段 5 的分析

程序运行结果截图：



分析与设计的过程：

操作姿势

a) **EDB** 操作基础知识：首先点击运行，这时程序会运行前面的初始化函数到 main，此时可以开始单步调试。

   i. step into：执行代码，如果是函数则进入。

   ii. step over：执行代码，如果是函数会执行然后跳过

   iii. step out：如果没有断点会直接跳到函数的 ret 指令处。

   iv. F2 调用右键的 Toggle BreakPoint 设置断点， 当然右键也有 Conditionnal BreakPoint 条件断点的选择，对应 Shift+F2。

   v. Rigisters 和 Stack 窗口可以分别查看运行处寄存器和栈的值。Data Dump 可以查看内存区域的值，这一块内存区域应该是伴随程序分配的内存区域。三个窗口呈现的形式都是左边是二进制，右边是字符串，如果真的存储的是字符串类型，那么我们可以通过右边直接看到，比较方便。

   vi. 如果想要查看程序不同段的内存，可以通过 View->Memory Regions 查看。

   vii. 无论是 Registers 还是 Stack 都可以右键直接设定二进制的值,这个特性可以用来进行程序的调试工作。

   viii. Ctrl+F 调用 Plugin 中的 BinarySearcher 可以直接搜索内存区域中的字符串。

   ix. 选定一行，Ctrl+*调用邮件的 set EIP to this instruction，跳过前面的代码直接运行到当前行。

   了解以上知识之后，简单使用 edb 运行程序应该不成问题。

b) **Readelf**：将所有的 elf 信息输出重定向到 phase5.elf 文件中，操作命令为 readelf –a phase5.o > phase5.elf。在 phase5.elf 文件中，我们可以观察到 Section Headers，rel.text，.rel.rodata 和.symtab 等各个 Section 信息，.rel.重定向节包括

offset	需要进行重定向的代码在.text 或.data 节中的偏移位置，4 个字节。
Info	包括 symbol 和 type 两部分，其中 symbol 占前 3 个字节，type 占后 1 个字节，symbol 代表重定位到的目标在.symtab 中的偏移量，type 代表重定位的类型
Type	重定位到的目标的类型
Name	重定向到的目标的名称

其中我们需要补充的就是每个重定位目标的 offset 和 info，一共 8 个字节，这里注意因为小端序的原因在 hexedit 中作为数的两者都是反的。

.symtab 节包括：

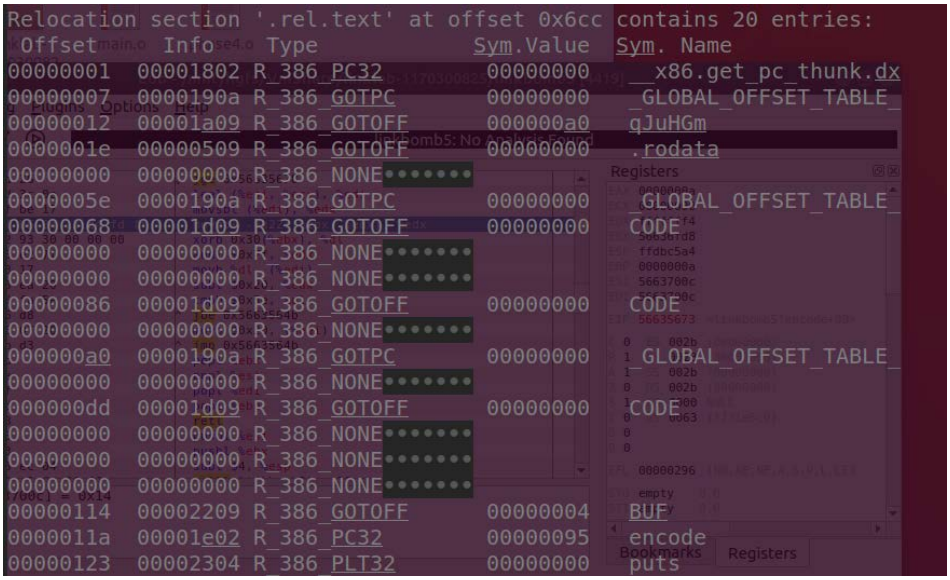
Num	symbol 十进制的偏移量
Name	Symbol 的名字

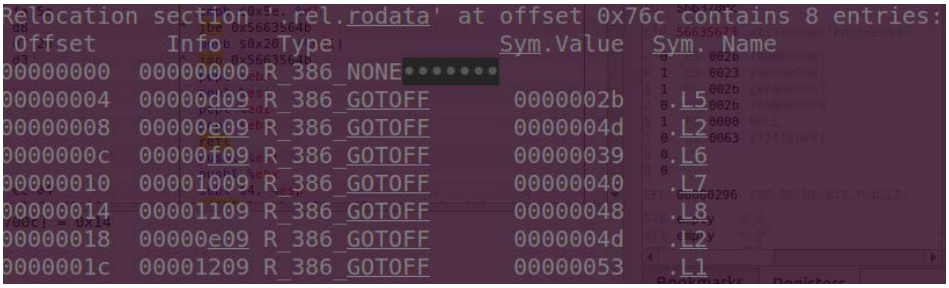
Section Headers 中我们可以看到所有节偏移量 off 和大小 size，这个偏移量是相对于整个 elf 文件而言的，通过这个偏移量和 hexedit 我们可以找到对应的节在 elf 二进制文件中的位置，进而进行观察和修改。

c）**HexEdit**：hexedit 可以用来修改.o 文件或 elf 可执行文件的二进制信息。我们可以查看 elf 文件已经有的重定位二进制信息。

查看已有的重定位信息

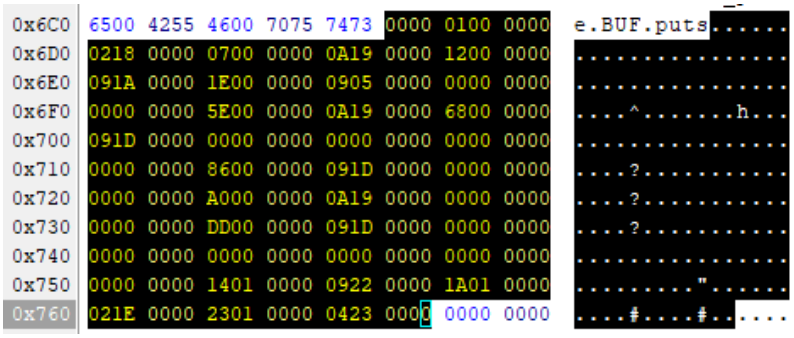
readelf –a phase5.o > phase5.elf 获得 phase5.elf 文件，截图如下：



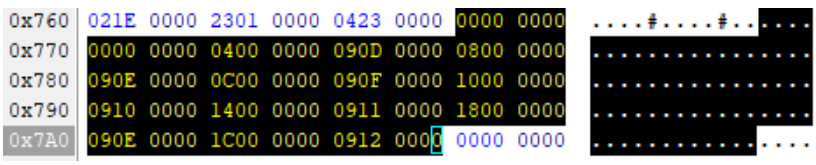


通过 hexedit 查看重定位部分对应的二进制信息（通过 phase5.elf 中 Section Headers 节给出的信息定位各个节的位置）：

.rel.text



.rel.rodata



仔细查看两种显示下重定位信息的对应关系，我们发现一个重定位信息占 8 个字节，前 4 个字节代表 offset，后四个字节代表 info，其中 info 的高 3 个字节代表 symbol 是该 symbol 在.symtab 中的 Num，info 的低 1 个字节代表 type 对应上面 phase5.elf 截图的 Type，不同的值代表不同的 type。

将.rel.text 的**已有**重定位信息整理如下，

info.type	含义	已知重定位目标
02	R_386_PC32	__X86.get_pc_thunk.dx ,encode
0a	R_386_GOTPC	_GLOBAL_OFFSET_TABLE_
09	R_386_GOTOFF	qJuHGm , .rodata , CODE,BUF
04	R_386_PLT32	puts

其中.rel.rodata **没有**的重定位信息是.L3。

我们需要加上的就是没有的重定位信息。

补充重定位信息

代码框架：

■ phase5.c程序框架

```
const int TRAN_ARRAY[] = {... ...};
const char FDICT[] = FDICTDAT;
char BUF[] = MYID;
char CODE = PHASE5_COOKIE;

int transform_code( int code, int mode ) {
  switch( TRAN_ARRAY [mode] & 0x00000007 ) {
    case 0:
      code = code & (~ TRAN_ARRAY[mode]);
      break;
    case 1:
      code = code ^ TRAN_ARRAY[mode];
      break;
    ... ...
  }
  return code;
}

void generate_code( int cookie ) {
  int i;
  CODE = cookie;
  for( i=0; i<sizeof(TRAN_ARRAY)/sizeof(int); i++ )
      CODE = transform_code( CODE, i );
}
```

```
int encode( char* str ) {
  int i, n = strlen(str);
  for( i=0; i<n; i++ ) {
    str[i] = (FDICT[str[i]] ^ CODE) & 0x7F;
    if( str[i]<0x20 || str[i]>0x7E ) str[i] = ' ';
  }
  return n;
}

void do_phase() {
  generate_code(PHASE5_COOKIE);
  encode(BUF);
  printf("%s\n", BUF);
}
```

- 上列绿色标出（以及如switch的跳转表等）的符号引用的对应重定位记录中随机选择若干个被置为全零。
- 涉及的重定位记录可能位于.text, .rodata等不同重定位节中

21

通过 phase5.elf 我们可以得到已经重定位的代码 offset（相对于.text 节），就可以推断出重定位代码的大致位置。通过给出的代码框架和 phase5.o 的反汇编代码，我们可以推出在哪里插入，以及插入什么重定位信息。这里不再详述，将我的反

汇编代码中作出的重定位信息补充列在下面（绿的是已有的，红的是补充的）：

```
Disassembly of section .text:

00000000 <transform_code>:
   0:   e8 fc ff ff ff        call   1 <transform_code+0x1>    //0218 0000
    //映射__X86.get_pc_thunk.dx
   5:   81 c2 02 00 00 00     add    $0x2,%edx                 //0A19 0000
    //映射_GLOBAL_OFFSET_TABLE_
   b:   8b 44 24 08           mov    0x8(%esp),%eax
   f:   8b 84 82 00 00 00 00  mov    0x0(%edx,%eax,4),%eax     //091A 0000
    //映射qJuHGm TRAN_ARRAY
  16:   89 c1                 mov    %eax,%ecx
  18:   83 e1 07              and    $0x7,%ecx
  1b:   03 94 8a 00 00 00 00  add    0x0(%edx,%ecx,4),%edx     //0905 0000
    //映射Num 8
  22:   ff e2                 jmp    *%edx

00000055 <generate_code>:
  55:   56                    push   %esi
  56:   53                    push   %ebx
  57:   e8 fc ff ff ff        call   58 <generate_code+0x3>  |58000000 021c0000
    //缺少__X86.get_pc_thunk.si
  5c:   81 c6 02 00 00 00     add    $0x2,%esi                 //0A19 0000
    //映射_GLOBAL_OFFSET_TABLE_
  62:   8b 44 24 0c           mov    0xc(%esp),%eax
  66:   88 86 00 00 00 00     mov    %al,0x0(%esi)             //091D 0000
    //映射CODE
  6c:   bb 00 00 00 00        mov    $0x0,%ebx
  71:   eb 1a                 jmp    8d <generate_code+0x38>
  73:   53                    push   %ebx
  74:   0f be 86 00 00 00 00  movsbl 0x0(%esi),%eax   |77000000 091D000
    //缺少CODE
  7b:   50                    push   %eax
  7c:   e8 fc ff ff ff        call   7d <generate_code+0x28>|7d000000 02170000
    //缺少tranform_code
  81:   83 c4 08              add    $0x8,%esp
  84:   88 86 00 00 00 00     mov    %al,0x0(%esi)             //091D 0000
    //映射CODE
  8a:   83 c3 01              add    $0x1,%ebx
  8d:   83 fb 0d              cmp    $0xd,%ebx
  90:   76 e1                 jbe    73 <generate_code+0x1e>
  92:   5b                    pop    %ebx
  93:   5e                    pop    %esi
  94:   c3                    ret
```

```
00000095 <encode>:
  95:   55                      push   %ebp
  96:   57                      push   %edi
  97:   56                      push   %esi
  98:   53                      push   %ebx
  99:   e8 fc ff ff ff          call   9a <encode+0x5>      |9a000000 021f0000
    //缺少__X86.get_pc_thunk.bx
  9e:   81 c3 02 00 00 00       add    $0x2,%ebx                //0A19 0000
    //映射_GLOBAL_OFFSET_TABLE_
  a4:   8b 74 24 14             mov    0x14(%esp),%esi
  a8:   b9 ff ff ff ff          mov    $0xffffffff,%ecx
  ad:   b8 00 00 00 00          mov    $0x0,%eax
  b2:   89 f7                   mov    %esi,%edi
  b4:   f2 ae                   repnz scas %es:(%edi),%al
  b6:   89 c8                   mov    %ecx,%eax
  b8:   f7 d0                   not    %eax

  ba:   83 e8 01                sub    $0x1,%eax
  bd:   89 c5                   mov    %eax,%ebp
  bf:   b9 00 00 00 00          mov    $0x0,%ecx
  c4:   eb 03                   jmp    c9 <encode+0x34>
  c6:   83 c1 01                add    $0x1,%ecx
  c9:   39 e9                   cmp    %ebp,%ecx
  cb:   7d 26                   jge    f3 <encode+0x5e>
  cd:   8d 3c 0e                lea    (%esi,%ecx,1),%edi
  d0:   0f be 17                movsbl (%edi),%edx
  d3:   0f b6 94 13 00 00 00    movzbl 0x0(%ebx,%edx,1),%edx|d7000000 0920000
    //缺少FDICT
  da:   00
  db:   32 93 00 00 00 00       xor    0x0(%ebx),%dl            //091D 0000
    //映射CODE
  e1:   83 e2 7f                and    $0x7f,%edx
  e4:   88 17                   mov    %dl,(%edi)
  e6:   83 ea 20                sub    $0x20,%edx
  e9:   80 fa 5e                cmp    $0x5e,%dl
  ec:   76 d8                   jbe    c6 <encode+0x31>
```

```
000000f8 <do_phase>:
  f8:   56                      push   %esi
  f9:   53                      push   %ebx
  fa:   83 ec 04                sub    $0x4,%esp
  fd:   e8 fc ff ff ff          call   fe <do_phase+0x6>  |fe000000 021f0000
    //缺少__X86.get_pc_thunk.bx
 102:   81 c3 02 00 00 00       add    $0x2,%ebx  /04010000 0A190000
    //缺少_GLOBAL_OFFSET_TABLE_

 108:   68 c0 00 00 00          push   $0xc0
 10d:   e8 fc ff ff ff          call   10e <do_phase+0x16>    |0e010000 021b0000
    //缺少generate_code
 112:   8d b3 00 00 00 00       lea    0x0(%ebx),%esi     //0922 0000
    //映射BUF
 118:   56                      push   %esi
 119:   e8 fc ff ff ff          call   11a <do_phase+0x22>   //021E 0000
    //映射encode
 11e:   83 ec 04                sub    $0x4,%esp
 121:   56                      push   %esi
 122:   e8 fc ff ff ff          call   123 <do_phase+0x2b>   //0423 0000
    //映射puts

 127:   83 c4 14                add    $0x14,%esp
 12a:   5b                      pop    %ebx
 12b:   5e                      pop    %esi
 12c:   c3                      ret
```

通过 hexedit 修改之后的 elf 文件的重定位信息部分为：

```
6500 4255 4600 7075 7473 0000 0100 0000  e.BUF.puts.....
0218 0000 0700 0000 0A19 0000 1200 0000  ................
091A 0000 1E00 0000 0905 0000 5800 0000  ............X...
021C 0000 5E00 0000 0A19 0000 6800 0000  ....^.......h...
091D 0000 7700 0000 091D 0000 7D00 0000  ....w.......}...
0217 0000 8600 0000 091D 0000 9A00 0000  ....?.......?...
021F 0000 A000 0000 0A19 0000 D700 0000  ....?.......?...
021F 0000 DD00 0000 091D 0000 FE00 0000  ....?.......?...
021F 0000 0401 0000 0A19 0000 0E01 0000  ................
021B 0000 1401 0000 0922 0000 1A01 0000  .........".......
021E 0000 2301 0000 0423 0000 0000 0000  ....#....#......
090C 0000 0400 0000 090D 0000 0800 0000  ................
```

第 5 章 总结

4.1 请总结本次实验的收获

* ELF 文件结构
* 更加深入理解链接的原理细节
* 利用 edb 的可视化可以更快更直接的完成 gdb 的工作。

4.2 请给出对本次实验内容的建议

注：本章为酌情加分项。

[1]16 进制计算网站 http://www.99cankao.com/digital-computation/hex-calculator.php