哈爾濱工業大學

实验报告

实 验（四）

题　　目　LinkLab

链接

专　　业　计算机类

学　　号　1170300821

班　　级　1703008

学　　生　罗瑞欣

指 导 教 师　郑贵滨

实 验 地 点　G712

实 验 日 期　2018.11.12

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（5 分）

只读代码段 (.init, .text, .rodata)	show_pointer 0x5629f84ba81a 94738259355674 useless 0x5629f84ba84d 94738259355725 main 0x5629f84ba858 94738259355736
读写数据段 （.data,.bss）	huge array 0x5629f86bc040 94738261459008 global 0x5629f86bc02c 94738261458988 big array 0x562a386bc040 94739335200832 p2 0x562a3b168670 94739379947120
运行时堆	p1 0x7fcad27e9010 140509091631120 p3 0x7fcae2dc7010 140509366218768 p4 0x7fca927e8010 140508017885200 p5 0x7fca127e7010 140505870397456
共享库内存映射区	exit 0x7fcae282d120 140509360345376 printf 0x7fcae284ee80 140509360483968 malloc 0x7fcae2881070 140509360689264 free 0x7fcae2881950 140509360691536
用户栈	local 0x7ffd098ca220 140724763664928 argc 0x7ffd098ca21c 140724763664924 argv 0x7ffd098ca348 140724763665224 argv[0] 7ffd098cb2d5 argv[1] 7ffd098cb2e3 argv[2] 7ffd098cb2e6 argv[3] 7ffd098cb2f1 argv[0] 0x7ffd098cb2d5 140724763669205 ./linkaddress argv[1] 0x7ffd098cb2e3 140724763669219 -u argv[2] 0x7ffd098cb2e6 140724763669222

1170300821
argv[3]　0x7ffd098cb2f1　140724763669233
罗瑞欣

env　0x7ffd098ca370 140724763665264
env[0]　*env 0x7ffd098cb2fb 140724763669243
CLUTTER_IM_MODULE=xim
env[1]　*env 0x7ffd098cb311 140724763669265
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:do=01;35:bd=40;
33;01:cd=40;33;01:or=40;31;01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;
42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;
31:*.lha=01;31:*.lz4=01;31:*.lzh=01;31:*.lzma=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;
31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;
31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tzst=01;31:*.bz2=01;31:*.bz=01;
31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;
31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;
31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;
31:*.esd=01;31:*.jpg=01;35:*.jpeg=01;35:*.mjpg=01;35:*.mjpeg=01;35:*.gif=01;
35:*.bmp=01;35:*.pbm=01;35:*.pgm=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;
35:*.xpm=01;35:*.tif=01;35:*.tiff=01;35:*.png=01;35:*.svg=01;35:*.svgz=01;
35:*.mng=01;35:*.pcx=01;35:*.mov=01;35:*.mpg=01;35:*.mpeg=01;35:*.m2v=01;
35:*.mkv=01;35:*.webm=01;35:*.ogm=01;35:*.mp4=01;35:*.m4v=01;35:*.mp4v=01;
35:*.vob=01;35:*.qt=01;35:*.nuv=01;35:*.wmv=01;35:*.asf=01;35:*.rm=01;35:*.rmvb=01;
35:*.flc=01;35:*.avi=01;35:*.fli=01;35:*.flv=01;35:*.gl=01;35:*.dl=01;35:*.xcf=01;
35:*.xwd=01;35:*.yuv=01;35:*.cgm=01;35:*.emf=01;35:*.ogv=01;35:*.ogx=01;
35:*.aac=00;36:*.au=00;36:*.flac=00;36:*.m4a=00;36:*.mid=00;36:*.midi=00;
36:*.mka=00;36:*.mp3=00;36:*.mpc=00;36:*.ogg=00;36:*.ra=00;36:*.wav=00;
36:*.oga=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
env[2]　*env 0x7ffd098cb8fd 140724763670781
LC_MEASUREMENT=zh_CN.UTF-8
env[3]　*env 0x7ffd098cb918 140724763670808
LESSCLOSE=/usr/bin/lesspipe %s %s
env[4]　*env 0x7ffd098cb93a 140724763670842
LC_PAPER=zh_CN.UTF-8
env[5]　*env 0x7ffd098cb94f 140724763670863
LC_MONETARY=zh_CN.UTF-8
env[6]　*env 0x7ffd098cb967 140724763670887
XDG_MENU_PREFIX=gnome-
env[7]　*env 0x7ffd098cb97e 140724763670910
LANG=zh_CN.UTF-8
env[8]　*env 0x7ffd098cb98f 140724763670927
DISPLAY=:0
env[9]　*env 0x7ffd098cb99a 140724763670938
GNOME_SHELL_SESSION_MODE=ubuntu
env[10]　*env 0x7ffd098cb9ba 140724763670970
COLORTERM=truecolor
env[11]　*env 0x7ffd098cb9ce 140724763670990
USERNAME=1170300821
env[12]　*env 0x7ffd098cb9e2 140724763671010
XDG_VTNR=2
env[13]　*env 0x7ffd098cb9ed 140724763671021
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh

	env[14] *env 0x7ffd098cba16 140724763671062 LC_NAME=zh_CN.UTF-8 env[15] *env 0x7ffd098cba2a 140724763671082 XDG_SESSION_ID=2 env[16] *env 0x7ffd098cba3b 140724763671099 USER=1170300821 env[17] *env 0x7ffd098cba4b 140724763671115 DESKTOP_SESSION=ubuntu env[18] *env 0x7ffd098cba62 140724763671138 QT4_IM_MODULE=xim env[19] *env 0x7ffd098cba74 140724763671156 TEXTDOMAINDIR=/usr/share/locale/ env[20] *env 0x7ffd098cba95 140724763671189 GNOME_TERMINAL_SCREEN=/org/gnome/Terminal/screen/457fb1f6_5a7b_4173_a894_d999d705e2d4 env[21] *env 0x7ffd098cbaeb 140724763671275 PWD=/home/luoruixin/hitics/lab5 env[22] *env 0x7ffd098cbb0b 140724763671307 HOME=/home/luoruixin env[23] *env 0x7ffd098cbb20 140724763671328 TEXTDOMAIN=im-config env[24] *env 0x7ffd098cbb35 140724763671349 SSH_AGENT_PID=1518 env[25] *env 0x7ffd098cbb48 140724763671368 QT_ACCESSIBILITY=1 env[26] *env 0x7ffd098cbb5b 140724763671387 XDG_SESSION_TYPE=x11 env[27] *env 0x7ffd098cbb70 140724763671408 XDG_DATA_DIRS=/usr/share/ubuntu:/usr/local/share:/usr/share:/var/lib/snapd/desktop env[28] *env 0x7ffd098cbbc3 140724763671491 XDG_SESSION_DESKTOP=ubuntu env[29] *env 0x7ffd098cbbde 140724763671518 LC_ADDRESS=zh_CN.UTF-8 env[30] *env 0x7ffd098cbbf5 140724763671541 GJS_DEBUG_OUTPUT=stderr env[31] *env 0x7ffd098cbc0d 140724763671565 LC_NUMERIC=zh_CN.UTF-8 env[32] *env 0x7ffd098cbc24 140724763671588 GTK_MODULES=gail:atk-bridge env[33] *env 0x7ffd098cbc40 140724763671616 PAPERSIZE=a4 env[34] *env 0x7ffd098cbc4d 140724763671629 WINDOWPATH=2 env[35] *env 0x7ffd098cbc5a 140724763671642 TERM=xterm-256color env[36] *env 0x7ffd098cbc6e 140724763671662 SHELL=/bin/bash env[37] *env 0x7ffd098cbc7e 140724763671678 VTE_VERSION=5202 env[38] *env 0x7ffd098cbc8f 140724763671695 QT_IM_MODULE=ibus env[39] *env 0x7ffd098cbca1 140724763671713 XMODIFIERS=@im=ibus

env[40] *env 0x7ffd098cbcb5 140724763671733
IM_CONFIG_PHASE=2
env[41] *env 0x7ffd098cbcc7 140724763671751
XDG_CURRENT_DESKTOP=ubuntu:GNOME
env[42] *env 0x7ffd098cbce8 140724763671784
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
env[43] *env 0x7ffd098cbd1c 140724763671836
GNOME_TERMINAL_SERVICE=:1.73
env[44] *env 0x7ffd098cbd39 140724763671865
XDG_SEAT=seat0
env[45] *env 0x7ffd098cbd48 140724763671880
SHLVL=1
env[46] *env 0x7ffd098cbd50 140724763671888
LANGUAGE=zh_CN:en_US:en
env[47] *env 0x7ffd098cbd68 140724763671912
LC_TELEPHONE=zh_CN.UTF-8
env[48] *env 0x7ffd098cbd81 140724763671937
GDMSESSION=ubuntu
env[49] *env 0x7ffd098cbd93 140724763671955
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
env[50] *env 0x7ffd098cbdbf 140724763671999
LOGNAME=1170300821
env[51] *env 0x7ffd098cbdd2 140724763672018
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
env[52] *env 0x7ffd098cbe08 140724763672072
XDG_RUNTIME_DIR=/run/user/1000
env[53] *env 0x7ffd098cbe27 140724763672103
XAUTHORITY=/run/user/1000/gdm/Xauthority
env[54] *env 0x7ffd098cbe50 140724763672144
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
env[55] *env 0x7ffd098cbe7d 140724763672189
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
env[56] *env 0x7ffd098cbee5 140724763672293
LC_IDENTIFICATION=zh_CN.UTF-8
env[57] *env 0x7ffd098cbf03 140724763672323
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
env[58] *env 0x7ffd098cbf24 140724763672356
SESSION_MANAGER=local/luoruixin:@/tmp/.ICE-unix/1441,unix/luoruixin:/tmp/.ICE-unix/1441
env[59] *env 0x7ffd098cbf7c 140724763672444
LESSOPEN=| /usr/bin/lesspipe %s
env[60] *env 0x7ffd098cbf9c 140724763672476
GTK_IM_MODULE=ibus
env[61] *env 0x7ffd098cbfaf 140724763672495
LC_TIME=zh_CN.UTF-8
env[62] *env 0x7ffd098cbfc3 140724763672515
OLDPWD=/home/luoruixin
env[63] *env 0x7ffd098cbfda 140724763672538

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(使用 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

3.1 阶段 1 的分析

程序运行结果截图：

```
1170300821@luoruixin:~/hitics/lab5$ gcc -m32 -o linkbomb1 main.o phase1.o
1170300821@luoruixin:~/hitics/lab5$ ./linkbomb1
1170300821
```

分析与设计的过程：

（1）先试运行 main.o，得到输出为欢迎字符

```
1170300821@luoruixin:~/hitics/lab5$ gcc -m32 -o linkbomb main.o
1170300821@luoruixin:~/hitics/lab5$ ./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。

```
00000000 <main>:
   0:   8d 4c 24 04             lea    0x4(%esp),%ecx
   4:   83 e4 f0                and    $0xfffffff0,%esp
   7:   ff 71 fc                pushl  -0x4(%ecx)
   a:   55                      push   %ebp
   b:   89 e5                   mov    %esp,%ebp
   d:   53                      push   %ebx
   e:   51                      push   %ecx
   f:   e8 fc ff ff ff          call   10 <main+0x10>
  14:   05 01 00 00 00          add    $0x1,%eax
  19:   8b 90 00 00 00 00       mov    0x0(%eax),%edx
  1f:   8b 12                   mov    (%edx),%edx
  21:   85 d2                   test   %edx,%edx
  23:   74 0c                   je     31 <main+0x31>
  25:   8b 80 00 00 00 00       mov    0x0(%eax),%eax
  2b:   8b 00                   mov    (%eax),%eax
  2d:   ff d0                   call   *%eax
  2f:   eb 14                   jmp    45 <main+0x45>
  31:   83 ec 0c                sub    $0xc,%esp
  34:   8d 90 00 00 00 00       lea    0x0(%eax),%edx
  3a:   52                      push   %edx
  3b:   89 c3                   mov    %eax,%ebx
  3d:   e8 fc ff ff ff          call   3e <main+0x3e>
```

（2）将 main.o 和 phase1.o 链接运行，得到一行无意义的字符串，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。

```
1170300821@luoruixin:~/hitics/lab5$ gcc -m32 -o linkbomb main.o phase1.o
1170300821@luoruixin:~/hitics/lab5$ ./linkbomb
RmZmn36YgS31Xy8 hm3EviftCEvyID8ps5jdERRgP8ZQK   MWVwzqWdwiTl    8X1hOPP01lzh3
KS8cIxRxcu5RQHOnYS9JKqYz4E1wd
```

printf(“%s\n”,s); 会优化为 puts(s)，注意 s 为字符串常数，应该在数据段，可知输出字符串起始地址在.data 节中偏移量为 xx 的位置。只需要在 phase1 中查找 到相应的字符串更改为我的学号就行了。

（3）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 1170300821 插入，并将学号后四个字节改为 0000，表示字符串结束。

	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	F403	0000	0000	0000	3400	0000	0000	2800	?.......4.....(.
0x030	1000	0F00	0100	0000	0800	0000	5589	E553	............U??S
0x040	83EC	04E8	FCFF	FFFF	0501	0000	008D	9064	??.?? .....??d
0x050	0000	0083	EC0C	5289	C3E8	FCFF	FFFF	83C4	...??.R???? ??
0x060	1090	8B5D	FCC9	C300	0000	0000	0000	0000	.??]???........
0x070	0000	0000	0000	0000	0000	0000	0000	0000	....
0x080	6971	734A	6C49	595A	7972	6409	6F48	6838	iqsJlIYZyrd.oHh8
0x090	6C42	6C64	6753	6E51	6C4D	3474	4967	484E	lBldgSnQlM4tIgHN
0x0A0	736A	5370	666B	4945	5A70	657A	4E51	4646	sjSpfkIEZpezNQFF
0x0B0	4A44	5A37	724A	6A64	6F66	204B	4447	0955	JDZ7rJjdof KDG.U
0x0C0	6147	6769	4148	7131	4D4D	6977	3876	414C	aGgiAHq1MMiw8vAL
0x0D0	4D4C	724E	6437	3971	6C74	3968	7A69	3076	MLrNd79qlt9hzi0v
0x0E0	6748	524D	3131	3730	3330	3038	3231	0000	gHRM1170300821..
0x0F0	3079	3809	686D	3345	7669	6674	4345	7679	0y8.hm3EviftCEvy
0x100	4944	3870	7335	6A64	4552	5267	5038	5A51	ID8ps5jdERRgP8ZQ
0x110	4B09	4D57	5677	7A71	5764	7769	546C	0938	K.MWVwzqWdwiTl.8
0x120	5831	684F	5050	3031	6C7A	6833	4B53	3863	X1hOPP01lzh3KS8c
0x130	4978	5278	6375	3552	5148	4F6E	5953	2020	IxRxcu5RQHOnYS
0x140	2020	2020	2020	2020	2000	0000	0000	0000	....
0x150	8B04	24C3	0047	4343	3A20	2855	6275	6E74	?.$?.GCC: (Ubunt
0x160	7520	372E	332E	302D	3136	7562	756E	7475	u 7.3.0-16ubuntu
0x170	3329	2037	2E33	2E30	0000	0000	1400	0000	3) 7.3.0........

（4）保存修改并运行

```
1170300821@luoruixin:~/hitics/lab5$ gcc -m32 -o linkbomb1 main.o phase1.o
1170300821@luoruixin:~/hitics/lab5$ ./linkbomb1
1170300821
```

3.2 阶段 2 的分析

程序运行结果截图：

```
1170300821@luoruixin:~/hitics/lab5$ gcc -m32 -o linkbomb2  main.o phase2.o
1170300821@luoruixin:~/hitics/lab5$ ./linkbomb2
1170300821
```

分析与设计的过程：

（1）解决 phase2 的两个基本要素，一是 ELF 表结构

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

另一个是 phase2 的函数结构

```
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\t..." );
}
```

（2）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，对应命令：

gcc –m32 –o linkbomb2 main.o phase2.o ， objdump –s –d linkbomb2 > linkbomb2.txt 。

```
000005b5 <hVFlxYLt>:
 5b5:   55                      push   %ebp
 5b6:   89 e5                   mov    %esp,%ebp
 5b8:   53                      push   %ebx
 5b9:   83 ec 04                sub    $0x4,%esp
 5bc:   e8 9f fe ff ff          call   460 <__x86.get_pc_thunk.bx>
 5c1:   81 c3 13 1a 00 00       add    $0x1a13,%ebx
 5c7:   83 ec 08                sub    $0x8,%esp
 5ca:   8d 83 50 e7 ff ff       lea    -0x18b0(%ebx),%eax
 5d0:   50                      push   %eax
 5d1:   ff 75 08                pushl  0x8(%ebp)
 5d4:   e8 07 fe ff ff          call   3e0 <strcmp@plt>
 5d9:   83 c4 10                add    $0x10,%esp
 5dc:   85 c0                   test   %eax,%eax
 5de:   75 10                   jne    5f0 <hVFlxYLt+0x3b>
 5e0:   83 ec 0c                sub    $0xc,%esp
 5e3:   ff 75 08                pushl  0x8(%ebp)
 5e6:   e8 05 fe ff ff          call   3f0 <puts@plt>
 5eb:   83 c4 10                add    $0x10,%esp
 5ee:   eb 01                   jmp    5f1 <hVFlxYLt+0x3c>
 5f0:   90                      nop
 5f1:   8b 5d fc                mov    -0x4(%ebp),%ebx
 5f4:   c9                      leave
 5f5:   c3                      ret
```

（3）找到 strcmp 函数，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。

```
 5d0:   50                      push   %eax
 5d1:   ff 75 08                pushl  0x8(%ebp)
 5d4:   e8 07 fe ff ff          call   3e0 <strcmp@plt>
```

因此目标是在 do_phase 函数的 nop 中写入执行压栈和相对位置跳转（call）到输出函数 hVFlxYLt 的逻辑。困难在于这里的 MYID 地址是变化的，我们是无法直接获得压栈的绝对地址值。

（4）解决方案：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，进行修改 PC 值从而完成跳转。其中 PC 值指的是 call 这一条完整指令的下一条指令的地址值。

（5）观察到在函数 hVFlxYLt 中调用了函数 x86.get_pc_thunk.bx 。其功能为执行完这个函数之后，可以通过被写的寄存器来通过偏移量访问 global 类型。

```
000005b1 <__x86.get_pc_thunk.ax>:
 5b1:   8b 04 24                mov    (%esp),%eax
 5b4:   c3                      ret
```

在 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，进入函数 x86.get_PC_thunk.ax 之后，将栈顶的值赋给寄存器%eax，此时寄存器%eax 中就放着可以用来相对寻址的下一条指令的位置了。

（6）输出函数中调用了 x86.get_PC_thunk.bx 使 ebx 指向了下一条 add 指令的位置，地址为 0x5b4，随后经过两个计算+0x1a20-0x18c0 得出 eax，即 MYID 的实际地址。

```
5b9:   83 ec 04                sub    $0x4,%esp
5bc:   e8 9f fe ff ff          call   460 <__x86.get_pc_thunk.bx>
5c1:   81 c3 13 1a 00 00       add    $0x1a13,%ebx
5c7:   83 ec 08                sub    $0x8,%esp
5ca:   8d 83 50 e7 ff ff       lea    -0x18b0(%ebx),%eax
5d0:   50                      push   %eax
```

（7）观察到在函数 do_phase 中，调用 x86.get_PC_thunk.ax 并使%eax 指向下一条 add 指令。所以在 do_phase 中利用 eax 的值计算字串运行时保存位置的公式：%eax-(0x5f1-0x5b4)+0x1a20-0x18c0=%eax+0x123

```
000005f6 <do_phase>:
 5f6:   55                      push   %ebp
 5f7:   89 e5                   mov    %esp,%ebp
 5f9:   e8 b3 ff ff ff          call   5b1 <__x86.get_pc_thunk.ax>
 5fe:   05 d6 19 00 00          add    $0x19d6,%eax
 603:   90                      nop
```

（8）构造汇编代码，编写汇编代码入 getcode.s 。gcc –m32 –c getcode.s 获得 getcode.o 。 objdump –d getcode.o > getcode.txt 将反汇编代码放入 getcode.txt 中，可以得到反汇编代码。

（9）使用 HexEdit 修改 phase2.o

然鹅

真实的操作过程，是更简单的一种方法。。。。。。。。。。。。。。。。。。。。。。。。。。。。

还是先说一下思路吧

（1）解决 phase2 的两个基本要素，一是 ELF 表结构

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

另一个是 phase2 的函数结构

```
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\t...” );
}
```

（2）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，对应命令：gcc－m32 －o linkbomb2 main.o phase2.o ， objdump －s －d linkbomb2 > linkbomb2.txt 。

```
000005b5 <hVFlxYLt>:
 5b5:   55                      push   %ebp
 5b6:   89 e5                   mov    %esp,%ebp
 5b8:   53                      push   %ebx
 5b9:   83 ec 04                sub    $0x4,%esp
 5bc:   e8 9f fe ff ff          call   460 <__x86.get_pc_thunk.bx>
 5c1:   81 c3 13 1a 00 00       add    $0x1a13,%ebx
 5c7:   83 ec 08                sub    $0x8,%esp
 5ca:   8d 83 50 e7 ff ff       lea    -0x18b0(%ebx),%eax
 5d0:   50                      push   %eax
 5d1:   ff 75 08                pushl  0x8(%ebp)
 5d4:   e8 07 fe ff ff          call   3e0 <strcmp@plt>
 5d9:   83 c4 10                add    $0x10,%esp
 5dc:   85 c0                   test   %eax,%eax
 5de:   75 10                   jne    5f0 <hVFlxYLt+0x3b>
 5e0:   83 ec 0c                sub    $0xc,%esp
 5e3:   ff 75 08                pushl  0x8(%ebp)
 5e6:   e8 05 fe ff ff          call   3f0 <puts@plt>
 5eb:   83 c4 10                add    $0x10,%esp
 5ee:   eb 01                   jmp    5f1 <hVFlxYLt+0x3c>
 5f0:   90                      nop
 5f1:   8b 5d fc                mov    -0x4(%ebp),%ebx
 5f4:   c9                      leave
 5f5:   c3                      ret
```

（3）找到 strcmp 函数，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。

```
 5d0:   50                      push   %eax
 5d1:   ff 75 08                pushl  0x8(%ebp)
 5d4:   e8 07 fe ff ff          call   3e0 <strcmp@plt>
```

因此目标是在 do_phase 函数的 nop 中写入执行压栈和相对位置跳转（call）到输出函数 hVFlxYLt 的逻辑。困难在于这里的 MYID 地址是变化的，我们是无法直接获得压栈的绝对地址值。

（4）解决方案：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，进行修改 PC 值从而完成跳转。其中 PC 值指的是 call 这一条完整指令的下一条指令的地址值。

（5）下图代码实现了将%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。

```
 5bc:   e8 9f fe ff ff          call   460 <__x86.get_pc_thunk.bx>
 5c1:   81 c3 13 1a 00 00       add    $0x1a13,%ebx
```

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。

```
0x050  81C3 0200 0000 83EC 088D 8300 0000 0050  ??....??.??....P
0x060  FF75 08E8 FCFF FFFF 83C4 1085 C075 1083   u.??    ??.??u.?
0x070  EC0C FF75 08E8 FCFF FFFF 83C4 10EB 0190  ?. u.??    ??.?.?
0x080  8B5D FCC9 C355 89E5 E8FC FFFF FF05 0100  ?]???U????    ...
0x090  0000 9090 9090 9090 9090 9090 9090 9090  ..????????????????
0x0A0  9090 9090 9090 9090 9090 9090 9090 9090  ????????????????
0x0B0  9090 905D C331 3137 3033 3030 3832 3100  ???]?1170300821.
0x0C0  0000 0000 8B04 24C3 8B1C 24C3 0047 4343  ....?.$??.$?.GCC
0x0D0  3A20 2855 6275 6E74 7520 372E 332E 302D  : (Ubuntu 7.3.0-
```

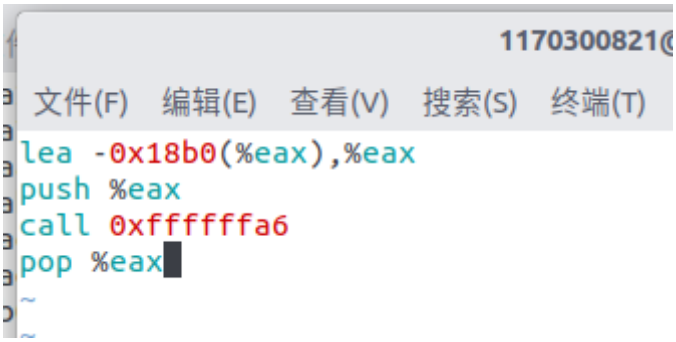
（6）在函数 hVFlxYLt 中

```
5bc:   e8 9f fe ff ff          call   460 <__x86.get_pc_thunk.bx>
5c1:   81 c3 13 1a 00 00       add    $0x1a13,%ebx
5c7:   83 ec 08                sub    $0x8,%esp
5ca:   8d 83 50 e7 ff ff       lea    -0x18b0(%ebx),%eax
```

将%ebx 指向_GLOBAL_OFFSET_TABLE_，后一步也是一个重定向之后确定的值，重定向之后%eax 指向了.rodata，就是 MYID。

（7）只需要将%eax 也指向.rodata。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 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，最后使 %eax 出栈恢复栈。

（8）写汇编+编译+反汇编+找 16 进制机器码+在 hexedit 里找到 nop+改成目标机器码



```
1170300821@luoruixin: ~/hitics/lab5
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)

getcode.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   8d 80 50 e7 ff ff       lea    -0x18b0(%eax),%eax
   6:   50                      push   %eax
   7:   e8 a2 ff ff ff          call   0xffffffae
   c:   58                      pop    %eax
~
```

3.3 阶段 3 的分析

程序运行结果截图：

```
1170300821@luoruixin:~/hitics/lab5$ gcc -m32 -c -o phase3_patch.o phase3_patch.c
1170300821@luoruixin:~/hitics/lab5$ gcc -m32 -o linkbomb3 main.o phase3.o phase3
_patch.o
1170300821@luoruixin:~/hitics/lab5$ ./linkbomb3
117000821
1170300821@luoruixin:~/hitics/lab5$
```

分析与设计的过程：

（1）do_phase 函数构造

■ **phase3.c程序框架**

```
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" );
}
```

（2）为了方便以后查看链接过程信息，首先将一个定义了任意名称的全局字符串的 phase3_patch.o 与其他两个.o 文件进行链接。

（3）分析 do_phase 函数反汇编指令，获知 COOKIE 字符串（保存于栈帧中的局部字符数组中）的组成内容和起始地址。使用 edb（简化了找到目标变量地址的过程）查找到对应循环输出的位置如下：

```
565f:8641 eb 2b                 jmp 0x565f866e
565f:8643 8d 55 e9              lea edx, [ebp-0x17]
565f:8646 8b 45 e4              mov eax, [ebp-0x1c]
565f:8649 01 d0                 add eax, edx
565f:864b 0f b6 00              movzx eax, byte [eax]
565f:864e 0f b6 c0              movzx eax, al
565f:8651 8d 93 70 00 00 00     lea edx, [ebx+0x70]
565f:8657 0f b6 04 02           movzx eax, byte [edx+eax]
565f:865b 0f be c0              movsx eax, al
565f:865e 83 ec 0c              sub esp, 0xc
565f:8661 50                    push eax
565f:8662 e8 e9 fd ff ff        call linkbomb3!putchar@plt
565f:8667 83 c4 10              add esp, 0x10
565f:866a 83 45 e4 01           add dword [ebp-0x1c], 1
565f:866e 8b 45 e4              mov eax, [ebp-0x1c]
565f:8671 83 f8 09              cmp eax, 9
565f:8674 76 cd                 jbe 0x565f8643
```

可以得到，COOKIE 字符串保存在栈帧中的字符数组是：

```
ff97:b670 67637872717566fc  fuqrxcg
ff97:b678 7be64d0000656273  sbe..M¥{
```

（4）查找指定 PHASE3_CODEBOOK 字符串的真实名称，通过 readelf －a 命令结合 ppt 中给定的该数字的大小特征，查得该字符串的真实名称为：

```
0000004e  00000b2b R_386_GOT32X      00000020   MvQfIErYyd
```

（5）在字符数组 cookie 的字符所指向的 MvQfIErYyd 数组的 指定位置处按顺序填上自己的学号即可。

```
1170300821@luoruixin: ~/hitics/lab5
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
char MvQfIErYyd[256]={0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,'2','0',0,
        '1','1','0',0,0,0,0,0,0,0,
        0,0,'7','0','8',0,'1',0,
        0,0,0,0,0,0,0,0,'3',
        0};
```

3.4 阶段 4 的分析

程序运行结果截图：

分析与设计的过程：

3.5 阶段 5 的分析

程序运行结果截图：

分析与设计的过程：

第 4 章 总结

4.1 请总结本次实验的收获

4.2 请给出对本次实验内容的建议

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.