

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	- 3 -
1.1 实验目的.....	- 3 -
1.2 实验环境与工具.....	- 3 -
1.2.1 硬件环境.....	- 3 -
1.2.2 软件环境.....	- 3 -
1.2.3 开发工具.....	- 3 -
1.3 实验预习.....	- 3 -
第 2 章 实验预习	- 5 -
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）	- 5 -
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）	- 5 -
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。	- 6 -
（5 分）	- 6 -
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）	- 6 -
第 3 章 各阶段的原理与方法	- 7 -
3.1 阶段 1 的分析.....	- 7 -
3.2 阶段 2 的分析	- 8 -
3.3 阶段 3 的分析	- 11 -
3.4 阶段 4 的分析	- 12 -
3.5 阶段 5 的分析	- 15 -
第 4 章 总结	- 19 -
4.1 请总结本次实验的收获.....	- 19 -
4.2 请给出对本次实验内容的建议.....	- 19 -
参考文献	- 20 -

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)

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)

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：

```
linda:linklab-1170300825>gcc -m32 -o linkbomb main.o phase1.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：

- 1) 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符串，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
GZxx05hnLEAFo5Pi5dNf5usR2cLZoMCi1WinGvJ 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。

- 2) 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 “1170300825\0” 填入到指定位置, 如下截图:

	001	0203	0405	0607	0809	0A0B	0C0D	0E0F	0123456789ABCDEF
0x000	F45	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	EC14S??.
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	hXLflu3VDCcCOYFr
0x080	484E	5256	3461	5133	6274	6549	6665	3052	HNRV4aQ3bteIfe0R
0x090	4DEB	5A42	5A46	4569	5351	384C	6F34	6B4F	Mk2BZFEiSQ8Lo4k0
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	LZoMCilWinGvJ 9U
0x0D0	6E57	3774	5275	4146	6E71	726F	5878	3032	nW7tRuAFngroXx02
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	0801zR.. ..
0x150	1B0C	0404	8801	0000	2000	0000	1C00	0000?....
0x160	0000	0000	2000	0000	0041	0E08	8302	430EA..?.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:

```
linda:linklab-1170300825>gcc -m32 -o linkbomb2 main.o phase2.o
linda:linklab-1170300825>./linkbomb2
1170300825
```

分析与设计的过程:

- 1) 分析 ELF 文件结构, 如下:

ELF Header
.text
.data
.bss
...
other sections
Section header table
String Tables
Symbol Tables
...

do_phase 函数结构如下:

```
static void OUTPUT_FUNC_NAME( const char *id ) // 该函数名对每个学生均不同
{
    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..." );
}
```

2) 对 ELF 文件的 Section 部分进行解析在 .text 节中找到指定的输出函数位置:

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的逻辑。同时因为这里的 MYID 地址是变化的,我们不能是无法直接获得压栈的地址值的。

4) 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跳指令

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：

565c:0619 8b 04 24	movl (%esp), %eax
565c:061c c3	retl

经过询问搜索引擎后得知，from StackOverFlow ca:

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，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。

- c. 相对寻址：

- a) 首先，我们发现输出函数中调用了 getPcThunc.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。

在 do_phase 中 eax 指向下一条 add 指令，地址为 0x5f1。

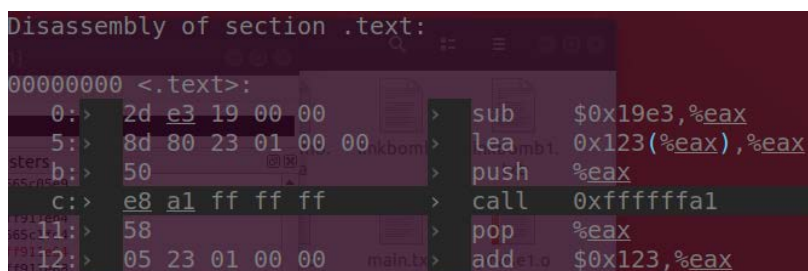
则在 do_phase 中利用 eax 的值计算字符串保存位置的公式：

$$\%eax-(0x5f1-0x5b4)+0x1a20-0x18c0=\%eax+0x123$$

- b) 根据构造出来的 call 指令下一条指令的起始地址来决定 call 指令相对寻址数值的大小，需要注意的是，将减法转化为补码相加形式。

- d. 构造插入代码：

- a) 只需注意遵守“恢复现场”原则即可。



```

Disassembly of section .text:
00000000 <.text>:
0: 2d e3 19 00 00 > sub $0x19e3,%eax
5: 8d 80 23 01 00 00 > lea mb1, 0x123(%eax),%eax
b: 50 > push %eax
c: e8 a1 ff ff ff > call 0xffffffa1
11: 58 > pop %eax
12: 05 23 01 00 00 > add $0x123,%eax

```

使用 HexEdit 修改 phase2.o:

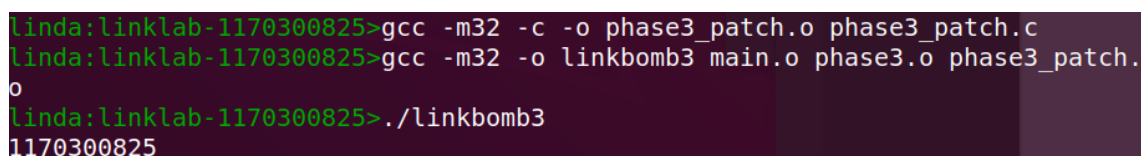
```

0000 2DE3 1900 008D 8023 0100 0050 E8A1 ..-?...?E#...P??
FFFF FF58 0523 0100 0090 9090 9090 9090 X.#...???????
9090 905D C331 3137 3033 3030 3832 3500 ???]?1170300825.

```

3.3 阶段 3 的分析

程序运行结果截图:



```

linda:linklab-1170300825>gcc -m32 -c -o phase3_patch.o phase3_patch.c
linda:linklab-1170300825>gcc -m32 -o linkbomb3 main.o phase3.o phase3_patch.o
linda:linklab-1170300825>./linkbomb3
1170300825

```

分析与设计的过程:

- 1) 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。
- 2) 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。这里直接使用 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:4637	0f b6 44 34 01	movzbl 1(%esp, %esi), %eax
565f:463c	8d 93 70 01 00 00	leal 0x170(%ebx), %edx
565f:4642	0f be 04 02	movsbl (%edx, %eax), %eax
565f:4646	83 ec 0c	subl \$0xc, %esp
565f:4649	50	pushl %eax
565f:464a	e8 01 fe ff ff	call linkbomb3!putchar@plt
565f:464f	83 c6 01	addl \$1, %esi
565f:4652	83 c4 10	addl \$0x10, %esp
565f:4655	83 fe 09	cmpl \$9, %esi
565f:4658	76 dd	jbe 0x565f4637

可以得到的信息有: COOKIE 字符串保存在栈帧中的字符数组是:

fffb:1fd0	6b786cfc	lzk
fffb:1fd4	69737068	hpsi
fffb:1fd8	006e716d	mqn.

题目中的所谓的 PHASE3_CODEBOOK 别保存在地址 0x565be140 开头的一段内存区域中。

至此我们已经得到了想要的信息。

3) 查找指定 PHASE3_CODEBOOK 字符串的真实名称, 通过 readelf -s 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:

```
46: 00002140 256 OBJECT GLOBAL DEFAULT 24 OdwnxQFyLh
```

4) 构造字符串: do_phase 函数构造如 PPT

```
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" );
}
```

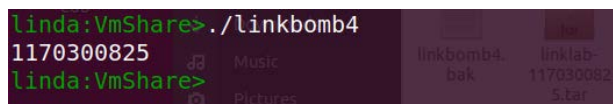
因此我们只需要在字符数组的字符所指向的 OdwnxQFyLh 数组的指定位置处填上自己的学号即可。

字符串构造如下:

```
char OdwnxQFyLh[256]=xxx...xxxxxxxxxxxxxxxx00x7185x32x0xxxx1xxxxxxxxxxxxxxxx...xxx
```

3.4 阶段 4 的分析

程序运行结果截图:



分析与设计的过程:

1) 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);
    }
}
```

a)

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

b) case首指令的偏移量

2) 分析反汇编代码:

a) 通过 edb 获得 cookie 的值，可以得到:

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

b) 定位 switch 主体代码:

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

i.

c) 分析

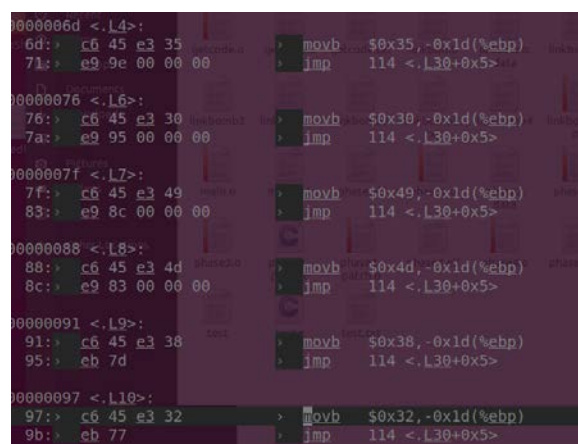
- a) 可以看出这里代码块的主要功能就是计算地址值到 `eax`，然后跳转到 `eax` 所指向的地址。`Eax` 的计算公式为 $((\%eax-0x41) \ll 2 + \%ebx-0x176c)$ ，外面的一个括号代

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.rodata 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 $\%eax = \%ebx + \text{偏移量}$ 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。

3) 构造答案:

- a) 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：



```

0000006d <.L4>:
6d:  c6 45 e3 35      movb $0x35, -0x1d(%ebp)
71:  e9 9e 00 00 00    jmp 114 <0x30+0x5>

00000076 <.L6>:
76:  c6 45 e3 30      movb $0x30, -0x1d(%ebp)
7a:  e9 95 00 00 00    jmp 114 <0x30+0x5>

0000007f <.L7>:
7f:  c6 45 e3 49      movb $0x49, -0x1d(%ebp)
83:  e9 8c 00 00 00    jmp 114 <0x30+0x5>

00000088 <.L8>:
88:  c6 45 e3 4d      movb $0x4d, -0x1d(%ebp)
8c:  e9 83 00 00 00    jmp 114 <0x30+0x5>

00000091 <.L9>:
91:  c6 45 e3 38      movb $0x38, -0x1d(%ebp)
95:  eb 7d            jmp 114 <0x30+0x5>

00000097 <.L10>:
97:  c6 45 e3 32      movb $0x32, -0x1d(%ebp)
9b:  eb 77            jmp 114 <0x30+0x5>

```

i.

.....

- 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：

```

6C65 2E00 6666 6666 0000 0000 0000 0000 1e...ffff.....
0000 0000 0000 0000 ABE6 FFFF 0000 0000 .....?? .....
A2E6 FFFF 0000 0000 ABE6 FFFF EAE6 FFFF ?? .....?? ??
0000 0000 CCE6 FFFF 0000 0000 0000 0000 ....?? .....
0000 0000 0000 0000 0000 0000 0000 0000 .....
ABE6 FFFF 44E7 FFFF 0000 0000 C6E6 FFFF ?? D? ....??
EAE6 FFFF 38E7 FFFF 44E7 FFFF 011B 033B ?? 8? D? ...;

```

（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，0xffffe738 偏移量对应 cookie- “X”）

3.5 阶段 5 的分析

程序运行结果截图：

分析与设计的过程：

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

```
# define ELF32_R_SYM(INFO)      ((INFO) >> 8)
# define ELF32_R_TYPE(INFO)    ((uint8_t) (INFO))

enum Rtl_Types {
    R_386_NONE                = 0, // No relocation
    R_386_32                  = 1, // Symbol + Offset
    R_386_PC32                = 2  // Symbol + Offset - Section Offset
};
```

```
// Symbol value
int symval = 0;
if(ELF32_R_SYM(rel->r_info) != SHN_UNDEF) {
    symval = elf_get_symval(hdr, reldtab->sh_link, ELF32_R_SYM(rel->r_info));
    if(symval == ELF_RELOC_ERR) return ELF_RELOC_ERR;
}
```

```
static int elf_get_symval(Elf32_Ehdr *hdr, int table, uint idx) {
    if(table == SHN_UNDEF || idx == SHN_UNDEF) return 0;
    Elf32_Shdr *symtab = elf_section(hdr, table);

    uint32_t symtab_entries = symtab->sh_size / symtab->sh_entsize;
    if(idx >= symtab_entries) {
        ERROR("Symbol Index out of Range (%d:%u). \n", table, idx);
        return ELF_RELOC_ERR;
    }

    int symaddr = (int)hdr + symtab->sh_offset;
    Elf32_Sym *symbol = &((Elf32_Sym *)symaddr)[idx];
```

565e:f6/d	5b	pushl %esi
565e:f67e	53	pushl %ebx
565e:f67f	83 ec 04	subl \$4, %esp
565e:f682	e8 fc ff ff ff	calll 0x565ef683
565e:f687	81 c3 02 00 00 00	addl \$2, %ebx
565e:f68d	68 c0 00 00 00	pushl \$0xc0
565e:f692	e8 fc ff ff ff	calll 0x565ef693
565e:f697	8d b3 34 00 00 00	leal 0x34(%ebx), %esi
565e:f69d	56	pushl %esi
565e:f69e	e8 77 ff ff ff	calll linkbomb5!encode
565e:f6a3	83 ec 04	subl \$4, %esp
565e:f6a6	56	pushl %esi
565e:f6a7	e8 14 fd ff ff	calll linkbomb5!puts@plt
565e:f6ac	83 c4 14	addl \$0x14, %esp
565e:f6af	5b	popl %ebx
565e:f6b0	5e	popl %esi
565e:f6b1	c3	retl

Symbol table '.symtab' contains 37 entries:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
0:	00000000	0	NOTYPE	LOCAL	DEFAULT	UND	linklab-117030082
1:	00000000	0	FILE	LOCAL	DEFAULT	ABS	phase5.c
2:	00000000	0	SECTION	LOCAL	DEFAULT	4	
3:	00000000	0	SECTION	LOCAL	DEFAULT	6	
4:	00000000	0	SECTION	LOCAL	DEFAULT	linklab7	main.o
5:	00000000	0	SECTION	LOCAL	DEFAULT	1703008	phase4.o
6:	00000000	0	SECTION	LOCAL	DEFAULT	10	
7:	00000000	0	SECTION	LOCAL	DEFAULT	12	
8:	00000000	0	SECTION	LOCAL	DEFAULT	13	
9:	00000000	0	SECTION	LOCAL	DEFAULT	14	
10:	00000000	0	SECTION	LOCAL	DEFAULT	16	
11:	00000000	0	SECTION	LOCAL	DEFAULT	17	
12:	00000024	0	NOTYPE	LOCAL	DEFAULT	4	.L3
13:	0000002b	0	NOTYPE	LOCAL	DEFAULT	4	.L5
14:	0000004d	0	NOTYPE	LOCAL	DEFAULT	4	.L2
15:	00000039	0	NOTYPE	LOCAL	DEFAULT	4	.L6
16:	00000040	0	NOTYPE	LOCAL	DEFAULT	4	.L7
17:	00000048	0	NOTYPE	LOCAL	DEFAULT	4	.L8
18:	00000053	0	NOTYPE	LOCAL	DEFAULT	4	.L1
19:	00000000	0	SECTION	LOCAL	DEFAULT	15	
20:	00000000	0	SECTION	LOCAL	DEFAULT	1	
21:	00000000	0	SECTION	LOCAL	DEFAULT	2	
22:	00000000	0	SECTION	LOCAL	DEFAULT	3	
23:	00000000	85	FUNC	GLOBAL	DEFAULT	4	transform_code
24:	00000000	0	FUNC	GLOBAL	HIDDEN	12	__x86.get_pc_thunk.dx
25:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	__GLOBAL_OFFSET_TABLE__
26:	000000a0	56	OBJECT	GLOBAL	DEFAULT	8	qJuHGm
27:	00000055	64	FUNC	GLOBAL	DEFAULT	4	generate code
28:	00000000	0	FUNC	GLOBAL	HIDDEN	14	__x86.get_pc_thunk.si
29:	00000000	1	OBJECT	GLOBAL	DEFAULT	6	CODE
30:	00000095	99	FUNC	GLOBAL	DEFAULT	4	encode
31:	00000000	0	FUNC	GLOBAL	HIDDEN	13	__x86.get_pc_thunk.bx
32:	00000020	128	OBJECT	GLOBAL	DEFAULT	8	smILsX
33:	000000f8	53	FUNC	GLOBAL	DEFAULT	4	do_phase
34:	00000004	11	OBJECT	GLOBAL	DEFAULT	6	BUF
35:	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	puts
36:	00000000	4	OBJECT	GLOBAL	DEFAULT	10	phase

Relocation section '.rel.text' at offset 0x6cc contains 20 entries

Offset	Info	Type	Sym.Value	Sym. Name
00000001	00001802	R_386_PC32	00000000	_x86.get_pc_thun#
00000007	0000190a	R_386_GOTPC	00000000	_GLOBAL_OFFSET_T#
00000012	00001a09	R_386_GOTOFF	000000a0	linkermain.o phase4.o
0000001e	00000509	R_386_GOTOFF	00000000	rodata
00000000	00000000	R_386_NONE		
0000005e	0000190a	R_386_GOTPC	00000000	_GLOBAL_OFFSET_T#
00000068	00001d09	R_386_GOTOFF	00000000	CODE
00000000	00000000	R_386_NONE		
00000000	00000000	R_386_NONE		
00000086	00001d09	R_386_GOTOFF	00000000	CODE
00000000	00000000	R_386_NONE		
000000a0	0000190a	R_386_GOTPC	00000000	_GLOBAL_OFFSET_T#
00000000	00000000	R_386_NONE		
000000dd	00001d09	R_386_GOTOFF	00000000	CODE
00000000	00000000	R_386_NONE		
00000000	00000000	R_386_NONE		
00000000	00000000	R_386_NONE		
00000114	00002209	R_386_GOTOFF	00000004	BUF
0000011a	00001e02	R_386_PC32	00000095	encode
00000123	00002304	R_386_PLT32	00000000	puts

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	0000	0000
0000	0000	5E00	0000	0A19	0000	6800	0000	...^.....h...
091D	0000	0000	0000	0000	0000	0000	0000
0000	0000	8600	0000	091D	0000	0000	0000	...?.....
0000	0000	A000	0000	0A19	0000	0000	0000	...?.....
0000	0000	DD00	0000	091D	0000	0000	0000	...?.....
0000	0000	0000	0000	0000	0000	0000	0000
0000	0000	1401	0000	0922	0000	1A01	0000"
021E	0000	2301	0000	0423	0000	0000	0000	...#...#...

第 4 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 的可视化可以通过类似于 debug 的方式得到寄存器的值，寄存器指向位置的值。
- * 利用 edb 可以修改寄存器、内存中的值，可以单步执行，发现 bug，更改 bug 比较高效。

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>.