西北工業大學

计 算 机 系 统 基 础 实 验 心 得 总 结

姓名 韩喻泷

班级 02

学号 2020303181

时间 2021 年 12 月 29 日

实验 1　数据表示

实验收获与心得

在实验 1 的伪编程实验中，我被要求使用有限的操作符和数据大小实现一系列函数。由于大部分题目均不允许使用循环结构，而仅仅能够使用逻辑运算和移位、取反等基础位运算操作，这使得函数的实现出现了种种困难。

`byteSwap` 函数是我的第一个障碍，我的思路是首先把要换的字节取出来，之后重新摆放，但是后者却出了问题——我希望在原数字中减去这两个字节，但却禁止使用减法操作，这里我想起了减法的补码运算：`a-b=a+((~b)+1)`，最后成功完成。

```
205  int byteSwap(int x, int n, int m) {
206    int N = n << 3;
207    int M = m << 3;
208    int xn = x >> N & 0xff;
209    int xm = x >> M & 0xff;
210    int y = xn << M | xm << N;
211    int y1 = xn << N | xm << M;
212    int z = x + (~y1 + 1) + y;
213    return z;
214  }
```

在实现 `logicalShift` 函数时，我想到了一种利用 `bitMask` 来清除前导 1 的方法，于是我直接放入了修改的 `bitMask` 的代码实现了逻辑位移函数（规则要求不允许调用其他函数）。

最后的几道题目难度很大，经过我数个小时的研究和探索，才找到了解决的办法。例如 `bitCount` 函数中巧妙利用分治思想进行的统计（实际上他的本质是取出每一位进行累积，但是题目的操作数限制了这样的“暴力”解法）。

```
387  int leftBitCount(int x) {
388    int t, a, s;
389    t = x;
390    a = !(~(t >> 16)) << 4;
391    t = t << a;
392    s = !(~(t >> 24)) << 3;
393    t = t << s;
394    a = a | s;
395    s = !(~(t >> 28)) << 2;
396    t = t << s;
397    a = a | s;
398    s = !(~(t >> 30)) << 1;
399    t = t << s;
400    a = a | s;
401    return (a | ((t >> 31) & 1)) + !(~x);
402  }
```

在 `leftBitCount` 函数中，我最终实现了一种将逻辑判断蕴含在位运算中的方法，其思路是按 2 的整次方从大到小倍增考虑，如果前面连续的一段均为 1，则移动到后面继续判断，若不是则在该位置直接考虑更小的 2 的整次幂长度，最后实际上是对 1 的前缀的一个二进制分解。

经过了本次实验，我了解了很多位运算的操作技巧，也对 IEEE 浮点数的格式以及其特殊的舍入规则更加熟悉，在实验中我在 StackOverflow 网站上学到了很多关于位运算的技巧，也同时锻炼了我搜集信息进行学习的能力。

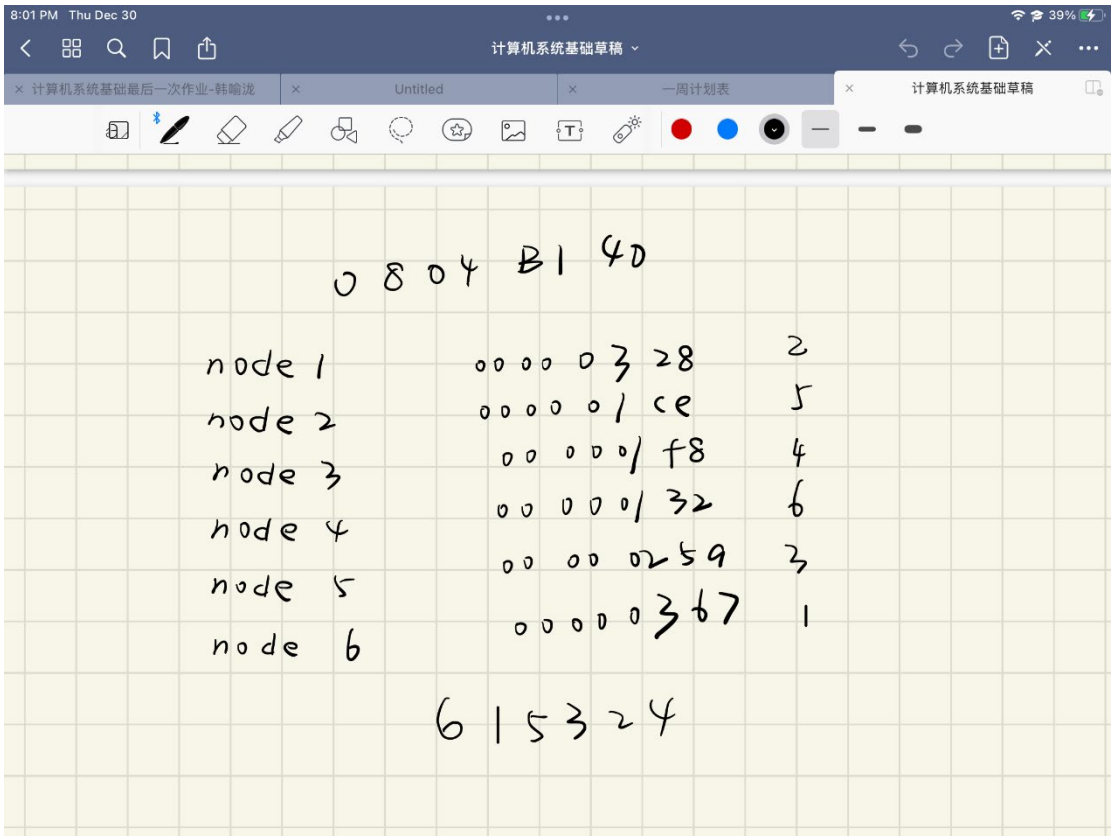
实验二 二进制炸弹

实验收获与心得

在实验二中，我需要通关 6 个 phase 以及一个隐藏 phase，通关的条件每个 phase 的“炸弹”不被引爆。在本次实验中，我遇到的一大问题是：汇编语言。由于每个阶段的“密钥”字符串都必须在二进制文件中寻找，并且还需要结合 objdump 和 IDA Pro 软件对可执行文件进行反汇编，并且有时需要使用 gdb 实时调试来确定某些寄存器和内存的存储数据。阶段涵盖了众多的汇编结构：函数栈帧结构、对数据区的数据结构分析、对分支和跳转结构的分析等等，每个阶段各有侧重。

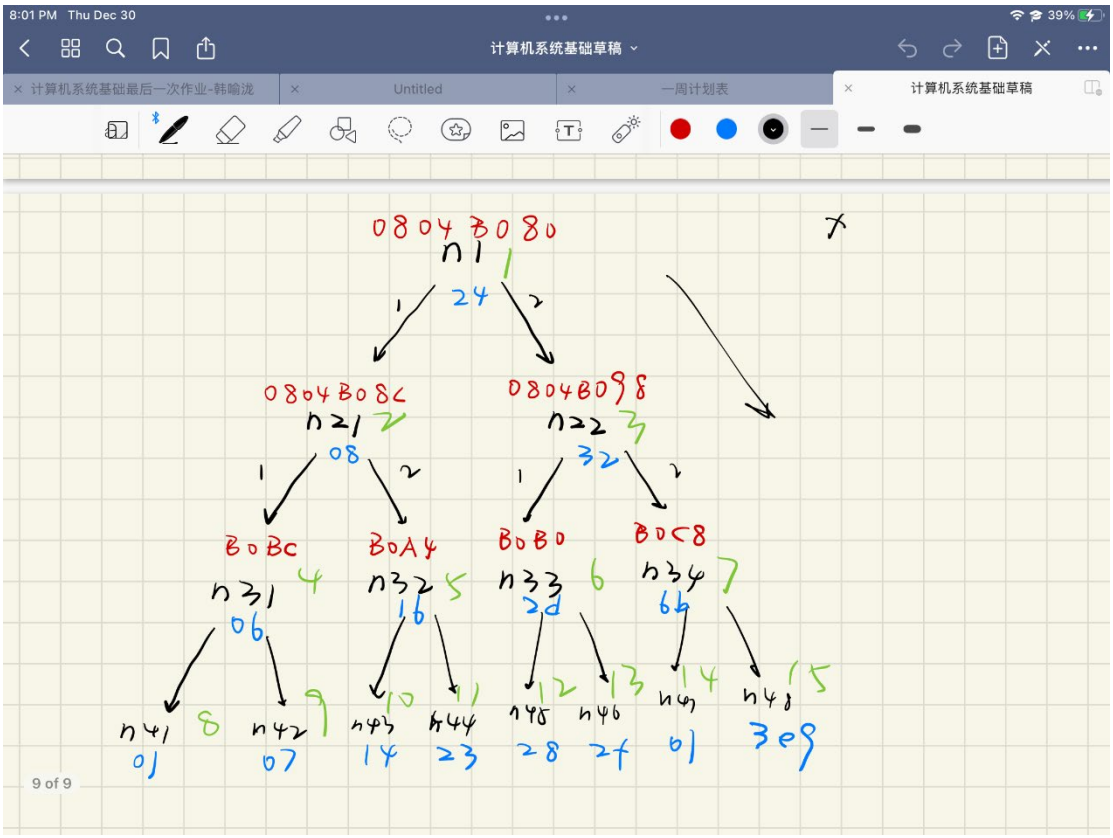
在 phase1 和 phase2 中，分别是在程序的.rodata 段寻找一个只读字符串和识别一个较为简单的循环结构。在 phase3 中是一个 switch 的跳转表结构，phase4 中调用了一个递归函数，这个函数包含判断结构与递归结构，也是让我分析了很久的一个阶段。

我遇到的最棘手的是 phase6 与 phase7 的隐藏关，对于 phase6 的链表结构，我手动画出了其较为直观的结构：



最后通过分析汇编，我得到了答案：6,1,5,3,2,4，是对各个 node 的大小进行从大到小的选择后依次得到的其 node 序号，这让我学到的是：对于给定的二进制数据不好观察，实际上可以通过自己在纸上组织结构来更清晰地观察。

最后的隐藏关需要输入一个特殊字符串附加在之前 phase 的结尾即可进入，对于最后一关，我对其结构体数据的二叉树结构也进行了手动的可视化：



图中红色的是其虚拟地址，蓝色的则是其数据值，绿色的则是其在数据段的偏移（下标），由于隐藏关的递归函数是一个类似树上查找的结构，我最后自己用 C++实现了这个函数，并将数据输入到我自己编写的程序里，得到了最后的答案：

```cpp
#include <bits/stdc++.h>
using namespace std;

struct node {
    int x, a, b;
    node() {}
    node(int _x, int _a, int _b) {
        x = _x;
        a = _a;
        b = _b;
    }
};
node S[16] = {
    node(-1, 0, 0),
    node(0x24, 2, 3),
    node(0x08, 4, 5),
    node(0x32, 6, 7),
    node(0x06, 8, 9),
    node(0x16, 10, 11),
    node(0x2d, 12, 13),
    node(0x6b, 14, 15),
    node(0x01, 0, 0),
    node(0x07, 0, 0),
    node(0x14, 0, 0),
    node(0x23, 0, 0),
    node(0x28, 0, 0),
    node(0x2f, 0, 0),
    node(0x01, 0, 0),
    node(0x3e9, 0, 0)
};
```

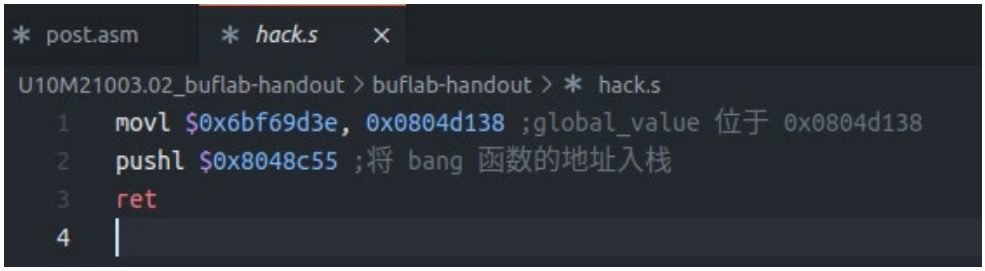
```cpp
int f(int p, int x) {
  int r; // eax
  if (!p)
    return -1;
  if (S[p].x > x )
    return 2 * f(S[p].a, x);
  r = 0;
  if (S[p].x != x)
    r = 2 * f(S[p].b, x) + 1;
  return r;
}

int main() {
    for(int i = 1; i <= 0x3e9; i++) {
        if(!f(1, i)) {
            printf("%d\n", i);
        }
    }
    return 0;
}
```

实验三 缓冲区溢出攻击

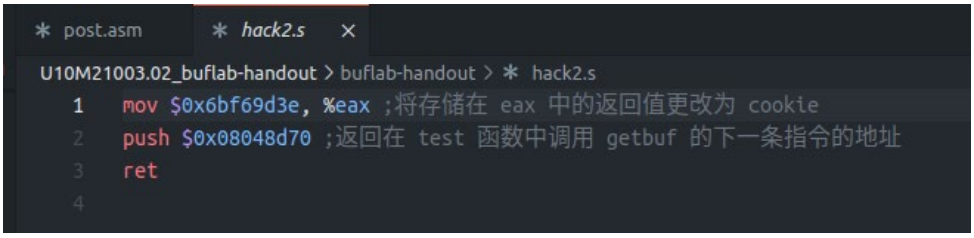
实验收获与心得

在实验三中，我需要利用 gets 函数的缓冲区溢出漏洞对程序进行攻击，前几个阶段中对返回地址的改写以及对函数参数的改写都没有太大的问题，但是当进行到后几个阶段时，我需要真正地在输入字符串中编写自己的机器指令，这使得难度有了不小的提升。

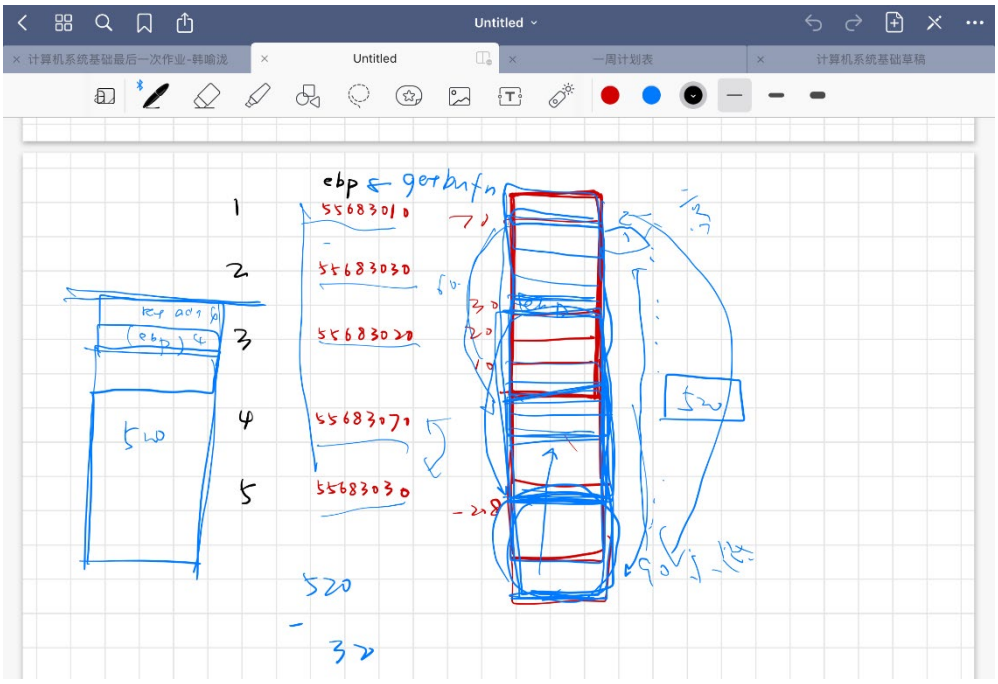
在 bang 阶段，我的恶意代码需要修改全局变量，我的思路还是较为清晰：找到全局变量的地址，然后直接 movl 更改其内容，由于还需要调用 bang 函数，我采用了 ret 的跳转方式，将函数地址 push 入栈，ret 取出绝对地址进行跳转：



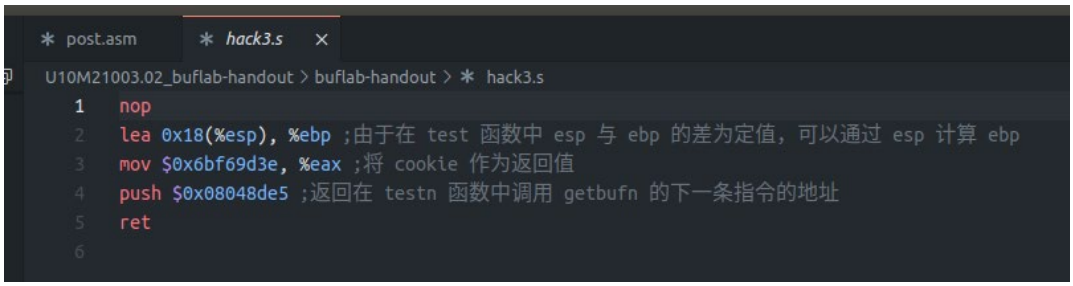
在 boom 阶段，我需要更改函数的返回值，由于单个双字的返回值会存放在 eax 寄存器中，于是我直接修改了 eax 寄存器的值：



到了最后的 nitro 阶段，我遇到了一个大难题——这一次调用的多次 getbufn 函数的每个地址均不同，根据 gdb 的内容，我画出了一张表：

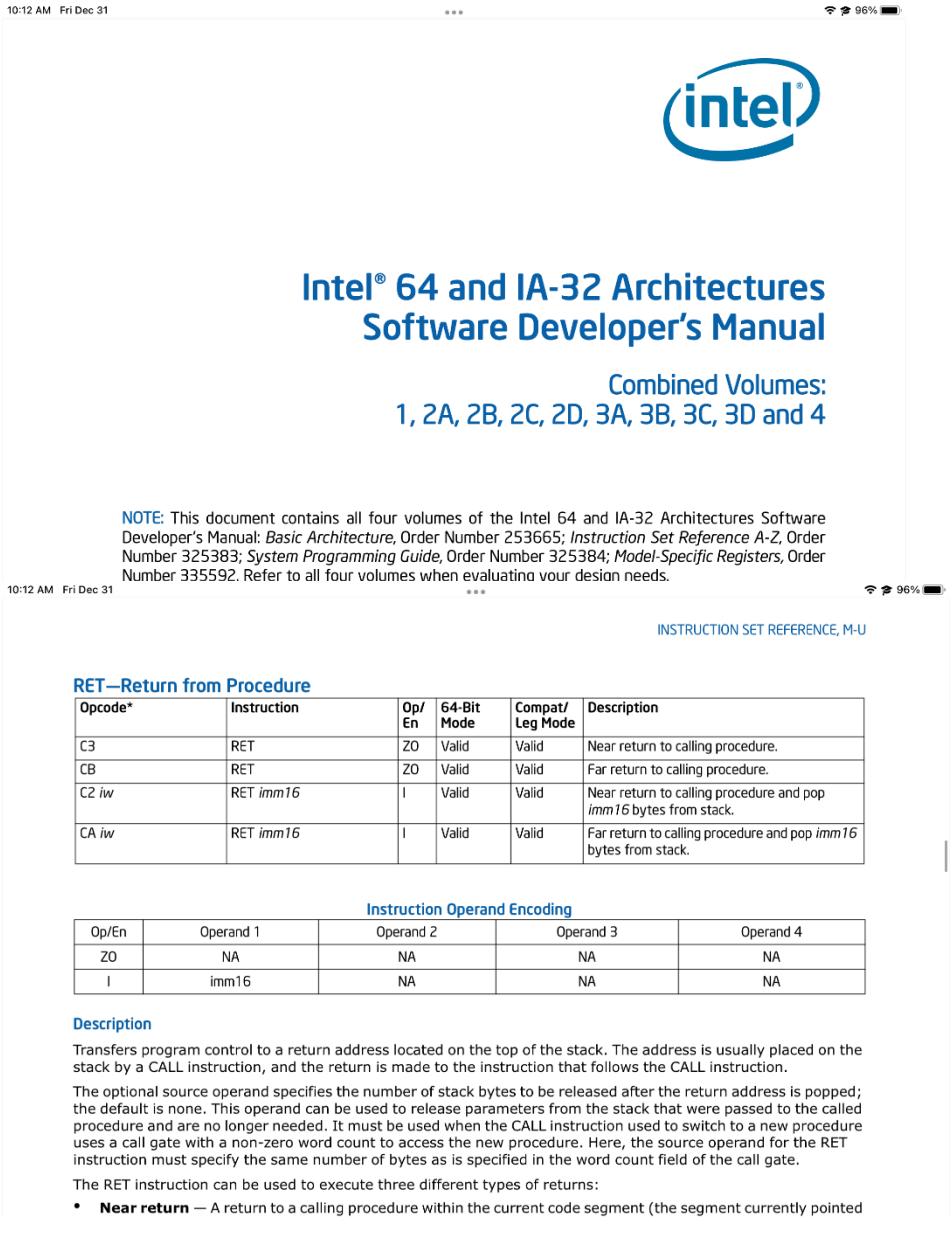


由于 ebp 不再固定，我们可以换一个思路：由于我们要保存的旧 ebp 是调用 getbufn 的 testn 函数，我们通过阅读 testn 的汇编可以发现，testn 的 ebp 值与调用 getbufn 的 esp 值之前差的是一个常数，这样我们便可以用 esp 间接得到 esp 的正确的值了：（在我的程序中 ebp=esp+0x18）



在最后的攻击字符串中，由于我在机器指令前加入了大量的 nop 指令，并且将最后的跳转地址指向了最低的 ebp 下拉一段距离的位置，以保证每一次都能跳到 nop，再通过“滑行”一步步运行到恶意代码部分。

经过实验三的练习，我真正尝试了编写汇编代码，以及将汇编代码与其机器码之间建立关联，在练习过程中，对于不太熟悉的汇编写法，我去查阅了 Intel 公司的官方 IA-32 指令集手册（这个手册有接近 5000 页的内容，并且涉及到机器底层的控制信号），在这个过程中我对 IA-32 的指令集更加熟悉了，也体会到了编写汇编代码的乐趣。

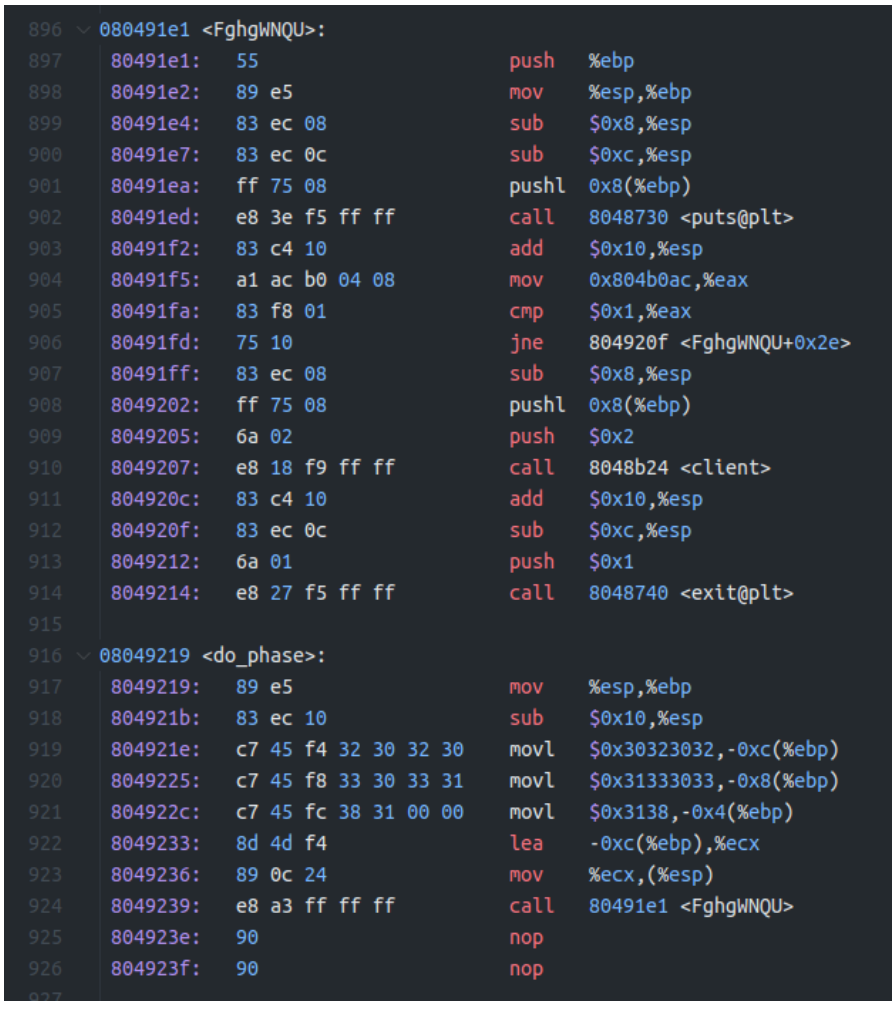


实验四 ELF 与链接

实验收获与心得

实验四主要是对 ELF 文件格式的各自操作，我在 phase2 阶段就遇到了第一个大难题——我需要补全一整个函数的汇编代码，包括建立栈帧，创建临时变量，传地址参数，调用其他函数，经过数个小时的奋战，最终成功运行的代码如下：（由于 ecx 寄存器空闲，所以我在传参数的时候使用 ecx 临时转存，并且由于担心空间不够，我精简了部分不需要的栈帧结构指令，因为调用函数后并不需要继续返回到该函数）

在链接之后，我手动修改了 linkbomb2 的二进制数据，使得最后的 call 指令指向正确的地址（924 行：`call 80491e1 <FghgWNQU>`）：



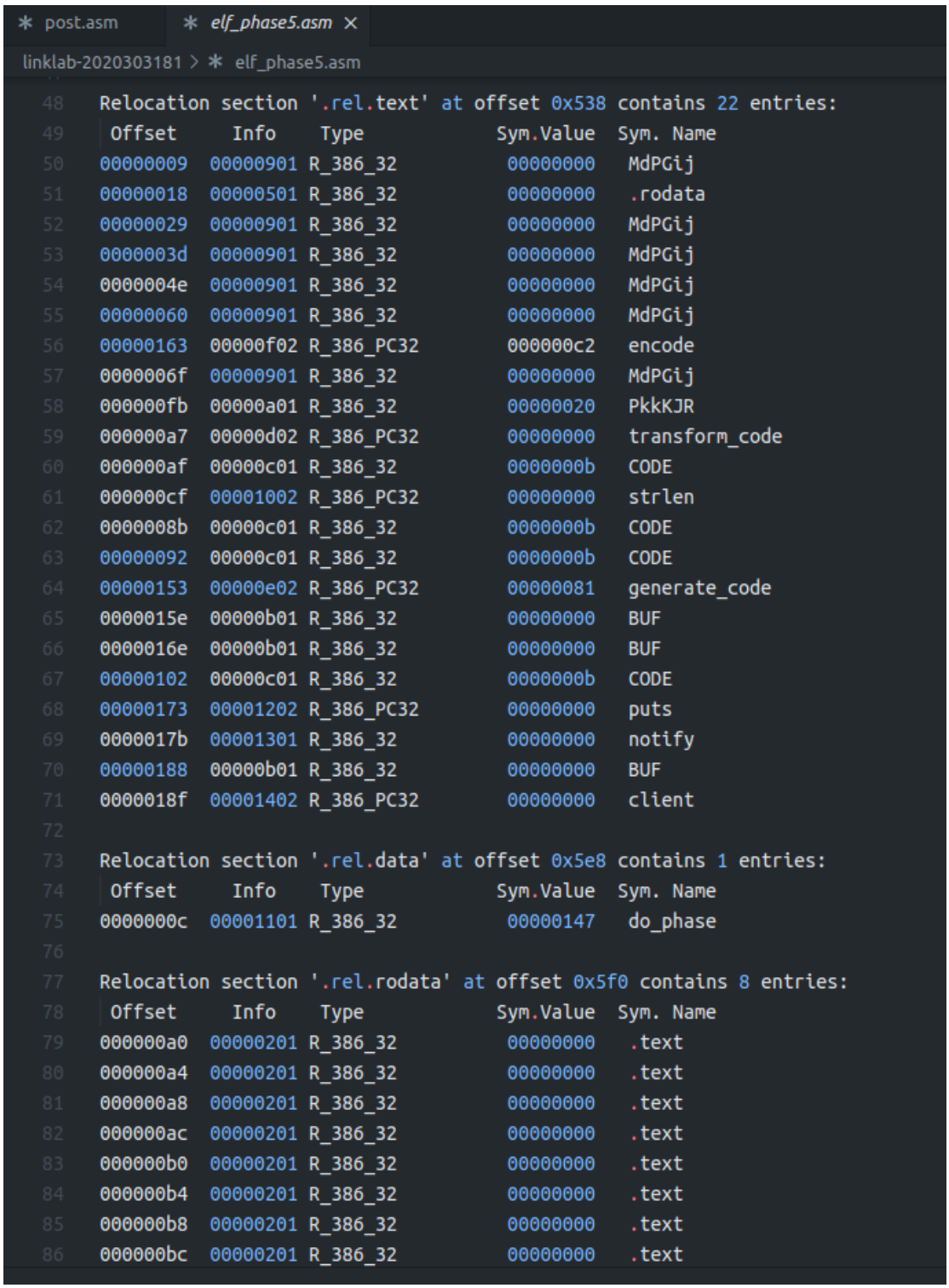
在之后的 phase3 强符号解析阶段，我自己写了一个 C++程序，用来生成 C 代码：

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    freopen("p3h.c", "w", stdout);
    char tmp[256] = {0};
    char s[] = "fungvlrcxm";
    char id[] = "2020303181";
    for(int i = 0; i < 10; i++) {
        tmp[s[i]] = id[i];
    }
    printf("char KPTEPZAOEW[256] = {\n");
    for(int i = 0; i < 16; i++) {
        printf("\t");
        for(int j = 0; j < 16; j++) {
            int idx = i * 16 + j;
            printf("%d,\t", tmp[idx]);
        }
        printf("\n");
    }
    printf("};\n");
    return 0;
}
```

运行输出的文件（p3h.c）：

```c
char KPTEPZAOEW[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,	0,	0,	0,	0,	0,	0,	0,	0,
	0,	0,	0,	49,	0,	0,	50,	48,	0,	0,	0,	0,	48,	49,	50,	0,
	0,	0,	51,	0,	0,	48,	51,	0,	56,	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,	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,
};
```

“生成代码”使我的 phase3 光速完成，在 phase4 的修改 switch 结构跳转表的分支地址也没有太大的问题，但是在最后的 phase5 阶段，我整整研究了一天多的时间才最终解决：一个是全面学习了 ELF 的各项结构与规范，以及对 readelf 中各种表的研究，之后是比对原汇编中缺失的重定向位置，并猜测重定向的内容——这个猜测的部分需要反复修改二进制，反复调试程序，是一个极其漫长的过程。最后我补全了的`.rel.text` 表以及`.rel.rodata` 如下所示：



post.asm elf_phase5.asm

linklab-2020303181 > elf_phase5.asm

```
Relocation section '.rel.text' at offset 0x538 contains 22 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000009  00000901 R_386_32          00000000   MdPGij
00000018  00000501 R_386_32          00000000   .rodata
00000029  00000901 R_386_32          00000000   MdPGij
0000003d  00000901 R_386_32          00000000   MdPGij
0000004e  00000901 R_386_32          00000000   MdPGij
00000060  00000901 R_386_32          00000000   MdPGij
00000163  00000f02 R_386_PC32        000000c2   encode
0000006f  00000901 R_386_32          00000000   MdPGij
000000fb  00000a01 R_386_32          00000020   PkkKJR
000000a7  00000d02 R_386_PC32        00000000   transform_code
000000af  00000c01 R_386_32          0000000b   CODE
000000cf  00001002 R_386_PC32        00000000   strlen
0000008b  00000c01 R_386_32          0000000b   CODE
00000092  00000c01 R_386_32          0000000b   CODE
00000153  00000e02 R_386_PC32        00000081   generate_code
0000015e  00000b01 R_386_32          00000000   BUF
0000016e  00000b01 R_386_32          00000000   BUF
00000102  00000c01 R_386_32          0000000b   CODE
00000173  00001202 R_386_PC32        00000000   puts
0000017b  00001301 R_386_32          00000000   notify
00000188  00000b01 R_386_32          00000000   BUF
0000018f  00001402 R_386_PC32        00000000   client

Relocation section '.rel.data' at offset 0x5e8 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0000000c  00001101 R_386_32          00000147   do_phase

Relocation section '.rel.rodata' at offset 0x5f0 contains 8 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
000000a0  00000201 R_386_32          00000000   .text
000000a4  00000201 R_386_32          00000000   .text
000000a8  00000201 R_386_32          00000000   .text
000000ac  00000201 R_386_32          00000000   .text
000000b0  00000201 R_386_32          00000000   .text
000000b4  00000201 R_386_32          00000000   .text
000000b8  00000201 R_386_32          00000000   .text
000000bc  00000201 R_386_32          00000000   .text
```

这个 phase5 是我整个实验课中耗时最长的一个实验，也是最有成就感的一个实验。

学习总结

几周下来，我通过本次实验课实地操作了很多汇编结构，对程序的结构以及汇编指令、机器指令有了更加深刻的理解与认识，我从中也收获了许多的乐趣，看着自己这几周写的代码和最后的成果，我也深深体会到了计算机程序设计的精密与环环相扣，这也是当今全世界计算机高速、高效、稳定运行的最坚实的基础。在之后的学习中，我会一步一步学习更多的专业课知识——数字逻辑设计、计算机组成原理、……在那一天，我希望我们的计算机能够更加地强大，改变我们每一个人的生活。

2021 年 12 月 31 日