bomb lab 实验

2022.9
lirx20@fudan.edu.cn

概述

lrx 创造的 bomb 是一款全新的代码世界冒险游戏，相比 csapp 的经典款，它更加易玩。你将假扮一位精通汇编语言/ linux / gdb / 数据结构的同学，在国庆小长假发现隐藏在混乱的 x86 汇编背后的真相...

- 二进制炸弹（bomb）是一个可执行文件，包含若干关卡
- 输入正确的口令，可以通过一个关卡；各关卡口令互不相同
- 口令错误，炸弹就会爆炸 o(T⌓To)
- 实验的目标是拆弹成功，也就是无伤通过所有关卡

实验内容

1. 字符串匹配
2. 循环和数组
3. switch
4. 递归调用
5. 虚函数
6. gdb call 和全局变量和树（完全不重要）
7. secret phase is a secret.

拆弹成功和失败

下面第一张图是成功时的命令行输出，第二张是炸弹爆炸的命令行输出

```
You have 6 phases with which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
You've enter the float point world! It's not hard o(*^ _ ^*)m
Congratulations!
```

```
BOOM!!!
The bomb has blown up.
```

实验步骤

- 下载 bomb.zip
- 反汇编：得到bomb的汇编代码，通过阅读和分析代码，推测
  - 花式使用 objdump (-x -C -D)
- 调试：设置断点、单步运行、查看寄存器和内存状态，推测
  - 使用 gdb 及其插件

- 验证你的推测 × N
- 记录所有的口令
    - 试试输入重定向（不用重复输入多次）
- 准备好实验报告和提交包

提交事项

内容

你需要提交两份文件，即

- 实验报告，命名为`report.pdf`。只允许提交 pdf 文件。
- 为每个关卡提供一个口令，口令之间以换行分开，写入一个文本文件并命名为`password.txt`。

> 助教の温馨提示：
>
> 实验报告应包含以下内容：
>
> 1. 姓名和学号
> 2. 每个关卡的推演过程（重要，如函数说明，实现方式等）
> 3. 拆弹成功的截图（随便弄下）
> 4. 如果有，请列出引用的内容以及参考的资料
> 5. 意见 + 建议（可选）

> 助教の温馨提示（2022秋季限定）：
>
> 实验报告是用来判断你是否是**独立思考**解决本次 lab 的。
>
> 助教们都很讨厌冗长的或表意混乱的报告。

评分

- 提交格式正确（含`password.txt`内部格式） 5%
- 解开每个关卡 $6 * 10\% = 60\%$
- 解开隐藏关卡 10%
- 实验报告 25%
- 抄袭倒扣分！！勿谓言之不预

> 反卷斗士の承诺：
>
> 实验过程非抄袭且提交格式正确且通过所有关卡的同学，实验报告部分赋满分。

格式要求

和上次提交相似。可按如下流程操作：

1. 新建文件夹，以你的学号命名（例如`20307130163`）。将你的`report.pdf`和`password.txt`放入文件夹中。
2. 在终端中进入到目录`20307130163/`，执行`cd ..`返回到上级目录，然后

    ```
    tar -cf 20307130163.tar 20307130163/
    ```

    这将得到一个名为`20307130163.tar`的文件。（该学号仅为示例，请使用你的学号）
3. 在 elearning 上对应作业处提交打包完成的`tar`文件。

deadline

2022-10-20 23:59:59 前在 elearning 提交。

bomblab 实验指导

> 我们在这里讲一遍，理论课再讲一遍，辅以 lab，希望大家能对这部分的知识掌握的更加牢固。当然，更重要的是善用搜索，学会自学。

寄存器相关

Figure 3.2 **Integer registers.** The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

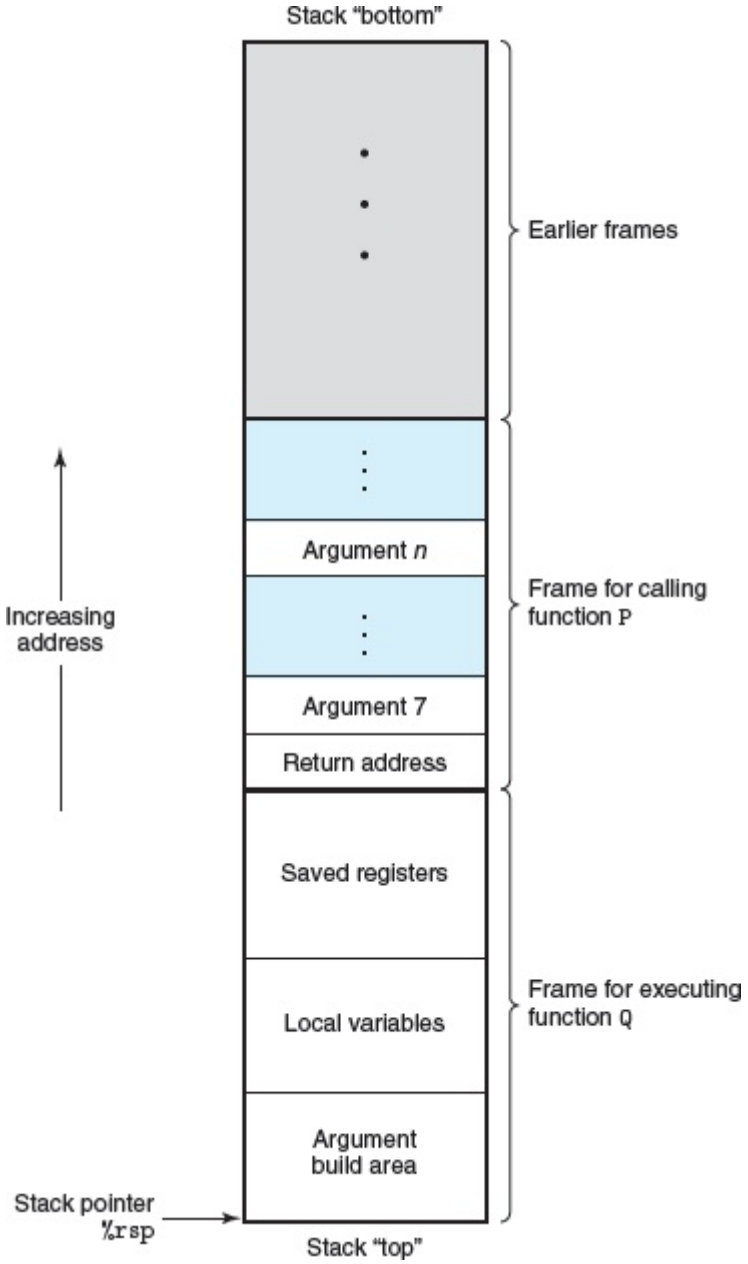
函数调用规则

函数调用时，被调用函数可能也会用到寄存器，覆盖了当前函数需要的数据，为了避免数据覆盖，x86 人为规定了通用寄存器的保存方式

- caller save（调用者保存）：在进行函数调用前，调用者需要提前保存好这些寄存器的值（如果后续还需要的话），因为被调用者可能会修改它们的值
- callee save（被调用者保存）：调用者不必保存，被调用者若要使用这些寄存器必须先保存

其它寄存器

- %rip（instruction pointer）== PC（program counter）：指向当前程序运行的地方
- %rsp（stack pointer）：栈指针，用来存放临时变量、函数返回值等等（从高地址向低地址减少，本次实验对程序栈有个印象即可，不需要深入了解）



常见汇编指令

> x86指令集是CISC（**complex** instruction set computer，复杂指令集架构），特点是指令长度不定，指令数量众多，因此汇编指令一般都是现查现用。

一般每条汇编指令有1到2个操作数，每个操作数的格式为寄存器、立即数或者地址等

以下列举本次实验会遇到的一些比较常见的汇编指令，还有很多其它的指令请自行查阅

指令	作用	示例
endbr64	在每个函数开头有一个，所有间接跳转指令的目标必须是该指令（即函数开头），否则报错（防止攻击的指令，本次实验可不管）	
push/pop	寄存器的值压栈/出栈，栈指针（rsp）对应减小/增大	push $rbp
callq/retq/call/ret	函数调用和返回指令（返回地址会对应压栈或出栈）	callq main
lea(Load effective address)	将地址加载到指定的寄存器（可能会用来做一些神奇的操作）	lea 0x0(,%rax,4),%rdx
add/sub	加减等运算，结果放在第二个操作数中	add %eax,-0x8(%rbp)
xor/and/or/sar	位运算，结果放在第二个操作数中	xor %rax %rax
test	将两个结果进行与运算，然后设置标志位，一般用来配合跳转指令	test %rdi %rdi
cmp	将第二个操作数与第一个操作数进行比较，然后设置标志位，一般用来配合跳转指令	cmpq $0x0 %rax
j{n/ne/a}	条件跳转指令，根据标识位的结果进行条件跳转	jn 2333
mov/movslq/movblz, etc.	移动指令，将第一个操作数的数据放在第二个操作数中	mov -0x8(%rbp),%rax

地址格式

形似 `<offset>(<reg1>, <reg2>, <imme>)` 的地址（其中各个字段可能省略），一般是表示 `reg1+reg2*imme+offset` 该地址对应的值（其中 lea 指令比较特殊，是直接加载地址）举例：

- `lea  0x0(,%rax,4),%rdx`：将 %rax 乘 4 的**值**（也就是地址本身）加载到 %rdx 中，常见于数组索引当中（为什么？）
- `mov  -0x8(%rbp),%rax`：将 %rbp 加上 -0x8 的**地址存放的值**加载到 %rax 中
- `mov  (%rdx,%rax,1),%eax`：将 %rdx 加上 %rax 乘 1的结果作为**地址**，取出这个地址存放的值，然后加载到 %eax 中
- `add  %eax,-0x8(%rbp)`：将 %eax 和 (%rbp 加上 -0x8的**地址存放的值**) 求和，然后再写回%rbp 加上 -0x8的地址处

一些常见形式的编译表示

条件判断

```
if(array == 0)return;

endbr64
test   %rdi,%rdi                        #让函数第一个参数和自己作与运算
je     11cb <for_sum(int*, int)+0x22> #若结果为0，则跳转；相当于判断参数是否为0，为0则跳转
```

循环结构

```
s = 0
for(int i = 0; i < size; ++i)s += array[i];

mov    $0x0,%eax  #初始化自增条件变量 i = 0
mov    $0x0,%edx  #初始化总和  s = 0
cmp    %esi,%eax  #判断条件变量是否满足条件 i < size ?
jge    11d0 <for_sum(int*, int)+0x27>  #大于或等于则跳出循环
add    (%rdi,%eax,4),%edx    #用i进行数组索引 base+i*4 => array[i]; s += array[i]
add    $0x1,%eax             #自增 i++
jmp    11bc <for_sum(int*, int)+0x13>  #回到比较的地方
```

switch case结构

1. 打表：如果case的不同条件比较紧凑规律（比如0、1、2、3.....），则将case的条件作为索引、对应的地址（可能是地址偏移量）作为值，创建一个表格存放在内存中。

case	addr
0	0x1000
1	0x1004
2	0x1008
3	0x1010

2. 类似if else嵌套：如果case比较稀疏，则通过采用连续的比较和条件跳转指令

```
cmp    $0x61,%dil
je     1224 <switch_case2(char)+0x1c>
cmp    $0x7a,%dil
je     1227 <switch_case2(char)+0x1f>
cmp    $0xa,%dil
je     122b <switch_case2(char)+0x23>
lea    0x1(%rdi),%eax
retq
```

递归

函数内部调用了函数本身，需要注意对callee-save寄存器的保存和使用

```
int recursion(int x){  #求斐波那契数
    if(x < 2)return x;
    return recursion(x-1)+recursion(x-2);
}

000000000000122c <recursion(int)>:
push   %rbp                     #保存 rbp
push   %rbx                     #保存 ebx
mov    %edi,%ebx                #保存第一个参数x，因为x和rdi寄存器后续函数都要使用，因此将x保存在新的寄存器中（这里是ebx），防止后续的函数将rdi中存放的x覆盖掉，而ebx因为是callee-save，因此提前保存了它
cmp    $0x1,%edi
jle    1258 <recursion(int)+0x2c> # 小于等于 1 则跳转，相当于 x < 2
lea    -0x1(%rdi),%edi          # x-1
callq  122c <recursion(int)>    # recursion(x-1)
mov    %eax,%ebp                # 将得到的返回值放在rbp中，因为rbp是callee-save，要提前保存
lea    -0x2(%rbx),%edi          # x-2，这里也用到了x，因此需要提前保存x
callq  122c <recursion(int)>
...
```

struct

内部变量按顺序排列，需要对齐

```
typedef struct {
    char a = 'y';      //root
    unsigned b = 2333; //root+0x4 虽然之前的char只占了1个字节，但是unsigned需要4字节对齐
    float c = 123.123; //root+0x8  0xb-0xf 之间的数值为0
    double d = 1930.1231313;  //root+0x10 虽然之前的float只占了4个字节，但是double需要8字节对齐
} node;
```

浮点运算

```
pxor   %xmm0,%xmm0              #浮点数的异或，这里相当于清零
#cvtsi2ss：源存储器1个32位有符号(补码)整数转为1个单精度浮点数,放入目的寄存器低32位
cvtsi2ss %edi,%xmm0             #这里相当于将一个 int 转换成 float
addss  0xd93(%rip),%xmm0        # a+=10
addss  %xmm0,%xmm0              # a*=2
subss  0xd8b(%rip),%xmm0        # a-=20
```

C++虚函数

类的非静态函数调用时，默认第一个参数是指向该对象的指针 **this**

每一个类有唯一的一个虚函数表（vtable，其中只包含虚函数的指针），不是每个对象都有一个vtable，恰恰是每个同一个类的对象都有一个指针，这个指针指向该类的vtable（当然，前提是这个类包含虚函数）。那么，每个对象只额外增加了一个指针的大小。

```
mov    $0x8,%edi
callq  1090 <operator new(unsigned long)@plt>  #申请了8个字节的空间
mov    %rax,%rbx                    #保存该空闲空间的起始地址，即指向对象的指针
mov    %rax,%rdi
callq  158c <B::B()>                #调用初始化函数，进行初始化，第一个参数为空闲空间的起始地址，即指向对象的指针this
#对象已经创建好
mov    (%rbx),%rax                  # 对象的内容只包含一个指针，这条指令通过this指针获取对象的内容，即类的vtable的起始地址
mov    (%rax),%rdx                  # 根据类的vtable起始地址获取要调用的函数的地址
mov    $0x2,%esi                    # 函数的第二个参数为 2
mov    %rax,%rdi
callq  *%rdx                        # 调用 rdx 位置处的函数（*号只是做标识用，并非取地址），参数分别为对象的地址（this）和2
```

objdump

objdump 是查看目标文件或者可执行的目标文件的汇编代码的常用反汇编工具。

输入 `objdump -d ./bomb > bomb.s` 以获得反汇编文件 `bomb.s`。

> 助教の温馨提示：
>
> `objdump --help` 会打印出 objdump 的所有用法，并且会给出精简的解释。`man objdump`会打印出详尽的解释。这些东西是自带的，不看白不看 o(￣▽￣*))
>
> 这些东西叫文档，他们是最全也是最权威的教会你如何使用 objdump 的资料。我们看文档的习惯，毕竟上网搜索得到的东西不一定全，甚至都不一定对。另外一些情况是，你无法组织语言表达你心中的疑问，或者你的疑问在哪都搜不到（这在随后几年的学习中会变得很常见！）。
>
> 文档也不都是像你`man objdump`得到的一样无聊——有些文档是基于网页的，做的很漂亮，也支持关键字搜索。
>
> 通常文档的内容**十分**详细，毕竟它包含了**一切**问题的答案。你需要懂得如何使用目录/小标题来定位你所关心的问题。不过如果你希望十分钟入门， 还是先尝试搜索引擎吧。

> 助教の温馨提示（2022秋季限定）：
>
> 你可以尝试在`objdump`后加上 `-D` 和/或 `-x` 和/或 `-C`，就像上面出现的`-d`一样。你可以通过搜索或读文档的方式了解这会导致什么后果；TA相信这几个参数会对你的实验有很大帮助！

gdb常用命令

> gdb（GNU symbolic debugger），是linux下常用的程序调试器，是本次lab的主力之一。
>
> 你同样可以通过 `man gdb` 或者上网搜索来获得更详尽的用法。

命令（缩写）	作用	示例
tab	命令或关键字补全	
run(r)	运行程序	r
quit(q)	退出 gdb	q
break(b)	打断点，可指定函数名或具体地址	b *0x400540
continue(c)	继续运行程序，一般是从断点处开始	c
print(p)	打印数据，可以指定格式，也可以指定寄存器、内存、变量等	p /x *($rsp+8)
display(disp)	指定的内容和print一样，不过可以重复展示	display /5i $pc
info(i)	获取信息，可指定寄存器、函数、断点信息等	i r{egister} pc
delete	删除gdb指定的内容，可指定断点等（默认是断点）	d {break} 1
stepi(si)	执行一条指令，后可跟数字表示执行多条	si 10
examine(x)	代码检查，后跟具体地址，可以指定输出的方式和格式	x /10i $pc
call	在gdb中进行函数调用（很神奇的功能）	call printf("\n")
backtrace(bt)	查看调用栈，即函数调用的情况	bt

正所谓熟能生巧，就让我们先用gdb一览一个程序的全貌吧。

如何获得帮助

lrx觉得你会遇到大量问题；而且计算机是一门博大精深的~~玄学~~科学，有的问题助教也不一定能给出令你满意的解答。lrx决定在这里给你一些额外的帮助和提示。

搜索

https://dir.scmor.com/ 含有谷歌镜像。当你发现百度搜出来的东西乱七八糟的时候，不妨一试。

https://stackoverflow.com 是程序领域的问答网站。可能需要特殊的技巧才能访问。当你在 csdn 和 cnblog 找到了完全错误的、甚至需要付费的答案的时候不妨一试。

手册

你想在网页里看`xxx`的手册？尝试搜索`xxx manual`或者`xxx doc`或者`xxx reference`。

你喜欢在命令行里看的话事情就非常简单——`man xxx`和`xxx --help`即可。看，简单而优雅，不是吗！

更牢固的基础

十一假期想充充电？强烈推荐这个，也就是著名的“ MIT 缺失一课 ”。B站有带字幕的视频。

你也可以尝试这个，本文档的部分内容做了参考。

更好的提问

我们，以及其他程序员不愿意回答的提问如下：

```
我的程序错误了/命令执行不了了/输出的东西乱七八糟的，怎么办?
```

因为这是一次课程作业！！！

我们希望看到**你的**努力。例如：

```
* 描述出错的现象
* 首先我做了aaa，现象是AAA，我得到的结论是XXX
* 然后我做了bbb，现象是BBB，我得到的结论是YYY
* 我搜索了ccc，搜到了CCC，我运行后现象是DDD/我还没来得及按上面的步骤做。

我觉得问题可能出在ddd，但我接下来没有思路了，我的分析和理解是否正确? 或者我忽略了什么吗?
我的linux系统版本是eee。
```

我觉得这个 lab 有的设计不合理

欢迎联系助教（即使是国庆放假期间也没有关系！你可以联系lrx。他的邮箱是 lirx20@fudan.edu.cn）。