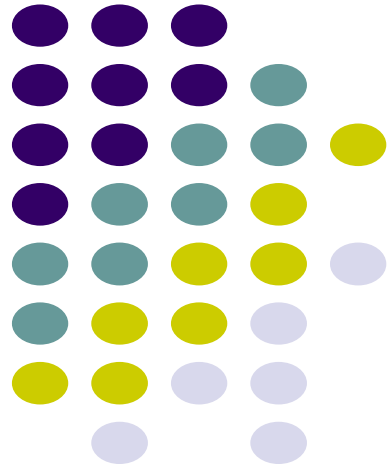
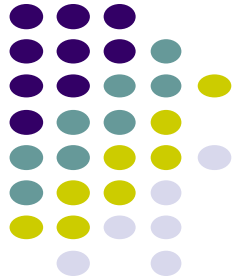
《计算机系统基础（四）：编程与调试实践》

二进制炸弹实验





二进制炸弹实验：概述

实验概述

实验目的

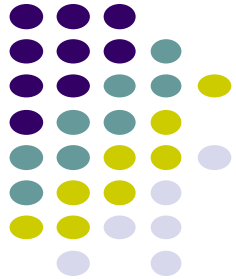
- 通过对一个二进制可执行程序（称为“二进制炸弹”）的理解和逆向工程，加深对对程序的机器级表示、汇编与反汇编、二进制程序调试和逆向工程等方面知识的理解和掌握。

实验内容

- 作为实验目标的二进制炸弹“binary bombs” Linux可执行程序包含了多个阶段（或关卡），在每个阶段程序要求输入一个特定字符串，如果输入满足程序代码所定义的要求，该阶段的炸弹就被拆除了，否则程序输出“炸弹爆炸BOOM!!! ”的提示并转到下一阶段再次等待对应的输入 ——实验的目标是设法得出解除尽可能多阶段的字符串。
- 为完成二进制炸弹拆除任务，需要通过反汇编并分析可执行炸弹文件程序的机器代码或使用gdb调试器跟踪机器代码的执行，从中理解关键机器指令的行为和作用，进而设法推断拆除炸弹所需的目标字符串。

实验环境：Linux 32-bit i386，C/汇编语言

实验数据与文件

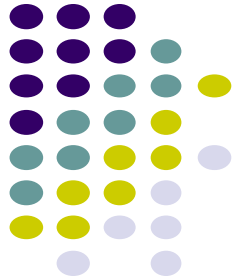


- 实验数据包：**bomblab_学号.tar**
- 解压命令：**tar xvf bomblab_学号.tar**
- 数据包中包含下面文件：
  - bomb：二进制可执行炸弹程序
  - bomb.c：bomb程序的main函数代码框架

实验概述

- 二进制炸弹目标程序
  - 包含了7个阶段以及1个隐藏阶段，分别考察对以下二进制程序表示各方面的理解和掌握：
    - 阶段0：字符串比较
    - 阶段1：浮点数表示
    - 阶段2：循环
    - 阶段3：条件/分支
    - 阶段4：递归调用和栈
    - 阶段5：指针
    - 阶段6：链表/指针/结构
    - 隐藏阶段：只有在阶段4的拆解字符串后再附加一特定字符串后才会出现（作为最后一个阶段）

实验概述



- **二进制炸弹可执行程序bomb**
  - bomb可执行程序接受0或1个命令行参数
    - 如果不指定参数，则bomb程序在输出欢迎信息后，等待用户按行输入每一阶段用来拆除炸弹的字符串，并根据输入字符串决定是通过相应阶段还是引爆该阶段的炸弹（输出“BOOM!!! ”）。
    - 也可将一个文本文件作为程序的唯一参数（例如“./bomb strings.txt”），文件中的每行包含拆除对应炸弹阶段的字符串，程序将依次检查每一阶段拆除字符串的正确性来决定炸弹拆除成败。
  - 在拆除炸弹的过程中，可以选择跳过一些暂时未能拆除的阶段
    - 可在要跳过的阶段中输入任意非空白（即不全是空格、制表、换行字符）字符串，将引爆相应阶段的炸弹，但程序不会中止而是进入下一阶段。

实验内容

- **二进制炸弹程序运行示例1 – 未指定命令行参数：**

  $ linuxer@debian:~/bomblab$ ./bomb

  Welcome to my fiendish little bomb. You have 7 phases with
  which to blow yourself up. Have a nice day!
  （等待输入阶段0的拆解字符串）

- **二进制炸弹程序运行示例2 – 以包含拆解字符串的文件作为参数：**

  $ linuxer@debian:~/bomblab$ ./bomb answers.txt

  Welcome to my fiendish little bomb. You have 7 phases with
  which to blow yourself up. Have a nice day!
  Well done! You seem to have warmed up!
  ……（其余阶段的验证输出）

程序框架bomb.c

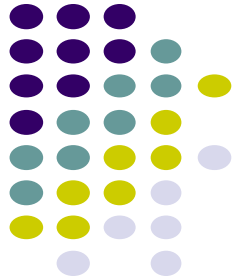
```
int main(int argc, char *argv[])  {
    char *input;
    ……
    initialize_bomb();

    printf("Welcome to my fiendish little bomb. You have 7 phases with\n");
    printf("which to blow yourself up. Have a nice day!\n");

    /* Warm up phase! */
    input = read_line();            /* Get input                */
    if( phase_0(input) ) {          /* Run the phase            */
        phase_defused();
        printf("Well done! You seem to have warmed up!\n");
    }

    /* Hmm...  Six phases must be more secure than one phase! */
    input = read_line();            /* Get input                */
    if( phase_1(input) ) {          /* Run the phase            */
        phase_defused();
        printf("Phase 1 defused. How about the next one?\n");
                }
```

实验工具



- objdump：反汇编二进制炸弹程序，获得其中汇编指令供分析
  - objdump -d bomb　输出bomb程序的反汇编结果
  - objdump -d bomb > bomb.s　获得bomb程序的反汇编结果并保存于文本文件bomb.s中供分析
  - objdump -t bomb　打印bomb程序的符号表，其中包含bomb中所有函数、全局变量的名称和存储地址
- strings：显示二进制程序中的所有可打印字符串

理解反汇编代码

Disassembly of section .init:　　反汇编针对的目标文件中的节

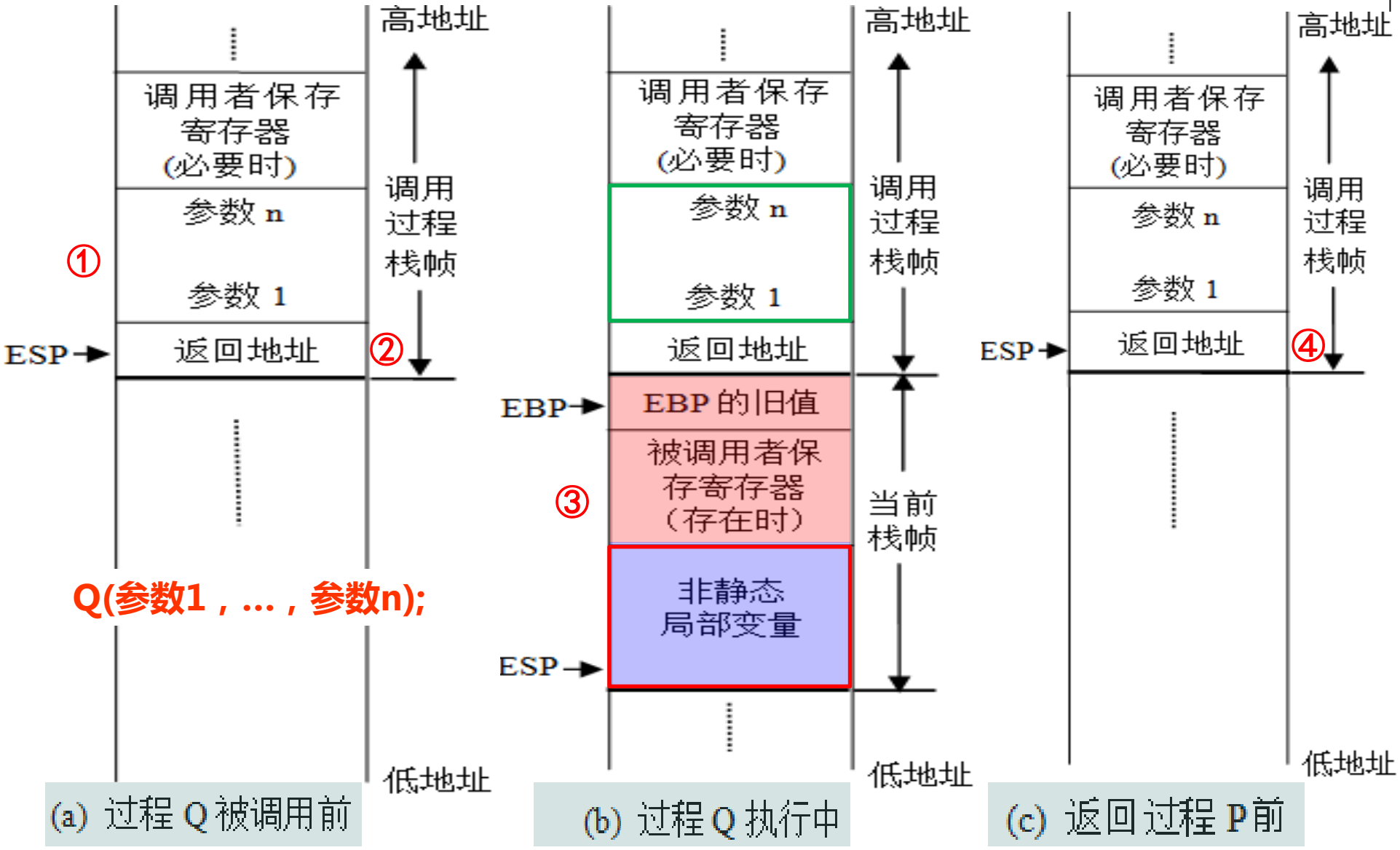
08048932 <Test>:　　函数名

8048932:	55	push %ebp
8048933:	89 e5	mov %esp,%ebp
8048935:	83 ec 08	sub $0x8,%esp
804893f:	8b 45 08	mov 0x8(%ebp),%eax
8048942:	0f af 45 0c	imul 0xc(%ebp),%eax
8048946:	89 45 f8	mov %eax,-0x8(%ebp)
8048949:	db 45 f8	fildl -0x8(%ebp)
804894c:	dd 05 10 96 04 08	fldl 0x8049610
8048952:	de c9	fmulp %st,%st(1)
8048954:	db 45 08	fildl 0x8(%ebp)

指令地址/偏移量　　指令机器码　　汇编语言指令

理解过程调用的机器级表示 – 栈帧

- 过程调用过程中栈和栈帧的变化 (Q为被调用过程)



过程栈帧及入口参数位置

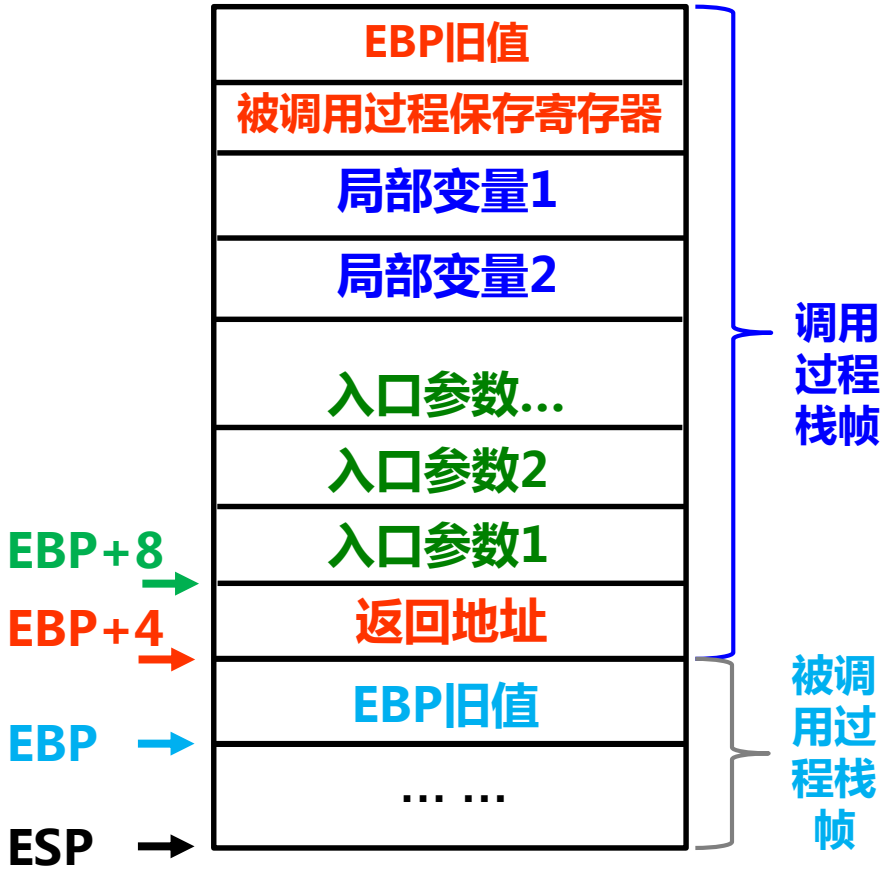
准备入口参数

```
pushl  参数3
.....
pushl  参数1
call   ...
```

或者

```
movl  参数3，8(%esp)
.....
movl  参数1, (%esp)
call   ...
```

返回地址是call指令的下一条指令的地址



i386 System V ABI规范规定，栈中参数按4字节对齐

- IA-32中，若参数类型是unsigned char、char或unsigned short、short，也都分配4个字节
- 因此，在被调用函数中，可使用R[ebp]+8作为有效地址来访问函数的第一个入口参数
- 第二个入口参数的地址 = 第一个入口参数的地址 + 第一个入口参数在栈中所占存储空间大小
- 第三、四、……参数的地址类推

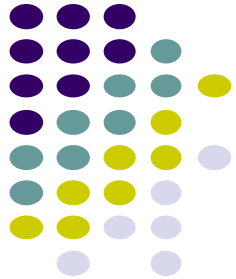
实验工具

- gdb：交互式程序调试工具（详细介绍可参看GDB文档和相关资料），具体具有以下几方面功能：
  - 装载、启动被调试的程序
  - 使被调试的程序在指定的调试断点处中断执行，方便查看程序变量、寄存器、栈内容等运行现场数据
  - 动态改变程序的执行环境，如修改变量的值。

✓ 可使用Gdb跟踪二进制炸弹程序各阶段函数的运行，查看相关数据对象，以帮助从二进制可执行bomb程序中分析、找出触发bomb爆炸的条件

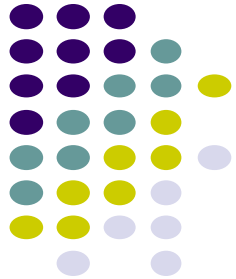
小结

- 本节课介绍了二进制炸弹实验的主要内容、实验所包含的各个阶段、实验数据的组成和实验用到的主要工具，并概要介绍了实验中经常涉及的过程调用的机器级表示方面的相关知识。



二进制炸弹实验：字符串比较

实验阶段0 - 字符串比较



- 阶段说明：该阶段要求输入与程序中内置的某一特定字符串相匹配的字节序列

- 实验步骤：
  1. 对二进制炸弹程序进行反汇编

     $ objdump -d bomb > bomb.s

  2. 对本阶段函数的汇编指令代码进行分析
  3. 定位并获得内置字符串的值，并相应构造输入字符串

实验阶段0 - 字符串比较

- 步骤2：本阶段函数汇编指令代码的分析
  - 定位与函数功能相对应的各控制逻辑
  - 获得主要变量的地址

内置字符串基址

输入字符串基址

字符串匹配比较

```
08048961 <phase_0>:
 8048961:  push   %ebp
 8048962:  mov    %esp,%ebp
 8048964:  sub    $0x8,%esp
 8048967:  sub    $0x8,%esp
 804896a:  push   $0x8049594
 804896f:  pushl  0x8(%ebp)
 8048972:  call   804906e <strings_not_equal>
 8048977:  add    $0x10,%esp
 804897a:  test   %eax,%eax
 804897c:  je     804898a <phase_0+0x29>
 804897e:  call   80492d6 <explode_bomb>
 8048983:  mov    $0x0,%eax
 8048988:  jmp    804898f <phase_0+0x2e>
 804898a:  mov    $0x1,%eax
 804898f:  leave
 8048990:  ret
```

分析红色框中汇编代码可知：为避免执行**0x804897e**处对引爆炸弹函数的调用指令，**je**指令的测试条件应被满足，即**0x804897a**处**test**指令执行之前，寄存器**EAX**的值应为**0**

实验阶段0 - 字符串比较

- **步骤2：本阶段函数汇编指令代码的分析**
  - 确定strings_not_equal函数的控制逻辑
    - 在程序的反汇编结果中搜索、找到strings_not_equal函数的汇编代码
  - 分析汇编代码可知：该函数在输入的两个字符串参数的长度和内容均相同时将返回0，否则返回1，并且返回值保存于EAX寄存器中

```
0804906e <strings_not_equal>:
 ……
8049075:   pushl  0x8(%ebp)
8049078:   call   8049042 <string_length>
804907d:   add    $0x4,%esp
8049080:   mov    %eax,%ebx
8049082:   pushl  0xc(%ebp)
8049085:   call   8049042 <string_length>
804908a:   add    $0x4,%esp
804908d:   cmp    %eax,%ebx
804908f:   je     8049098
8049091:   mov    $0x1,%eax
8049096:   jmp    80490d4
8049098:   mov    0x8(%ebp),%eax
804909b:   mov    %eax,-0x8(%ebp)
804909e:   mov    0xc(%ebp),%eax
80490a1:   mov    %eax,-0xc(%ebp)
80490a4:   jmp    80490c5
80490a6:   mov    -0x8(%ebp),%eax
80490a9:   movzbl (%eax),%edx
80490ac:   mov    -0xc(%ebp),%eax
80490af:   movzbl (%eax),%eax
80490b2:   cmp    %al,%dl
80490b4:   je     80490bd
 ……
```

实验阶段0 - 字符串比较

- 步骤3：定位并获得内置字符串的值，并相应构造输入字符串
  - 前面已推断出：和用户输入字符串相比较的程序内置字符串的存储地址为**0x8049594**，并且两个字符串的内容应相同，因此可使用**gdb**查看地址**0x8049594**中存储的内置字符串的内容：

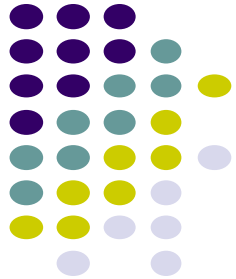
```
linuxer@debian:~/bomblab$ gdb bomb
……
(gdb) b *0x8048972
Breakpoint 1 at 0x8048972
(gdb) r
Starting program:
/home/ics/Course/bomblab/bomb
Welcome to my fiendish little bomb. You have 7
phases with
which to blow yourself up. Have a nice day!
sgjsogjsoigjosjs（此处暂时随意输入一些字符）

Breakpoint 1, 0x08048972 in phase_0 ()
```

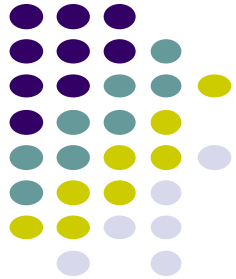
```
(gdb) x/40c 0x8049594
0x8049594:      80 'P'  105 'i' 112 'p' 101 'e' 108 'l'
105 'i' 110 'n' 105 'i'
0x804959c:      110 'n' 103 'g' 32 ' '  105 'i' 109 'm'
112 'p' 114 'r' 111 'o'
0x80495a4:      118 'v' 101 'e' 115 's' 32 ' '  116 't'
104 'h' 114 'r' 111 'o'
0x80495ac:      117 'u' 103 'g' 104 'h' 112 'p' 117 'u'
116 't' 46 '.'  0 '\000'
0x80495b4:      37 '%'  100 'd' 32 ' '  37 '%'  100 'd' 0
'\000'      0 '\000‘
(gdb) x/1s 0x8049594
0x8049594:      "Pipelining improves throughput."
```

  - 在上述**GDB**的**x**命令中，因已知目的地址中存放的是字符串，所以使用**’c’**以可打印字符形式显示地址中存放的**ASCII**编码（也可用命令“**x/1s 0x8049594**”直接打印目标字符串）
  - 从中可看出，内置字符串（到标志其结束的**0x00**字节为止）为**“Pipelining improves throughput.“** —— 此即本阶段期望的输入拆解字节串

小结

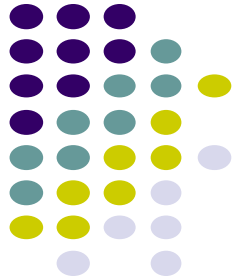


- 本节课介绍了二进制炸弹实验第一阶段（即阶段0-字符串比较）的基本过程和方法，并且演示了其中的主要操作步骤。从中，我们了解了二进制程序的反汇编结果的基本组成、过程之间的参数传递机制和访问方法、通过GDB工具在程序中设置断点并查看数据对象等基本操作。



二进制炸弹实验：浮点数表示

实验阶段1 - 浮点数表示



- 阶段说明：该阶段要求输入对应某浮点（**float**或**double**）数值表示的一对整数（**short**或**int**）

- 实验步骤：
  1. 对本阶段函数的汇编指令代码进行分析
  2. 找到浮点数常量的值，获得其**IEEE-754**表示
  3. 分析程序的输入要求和比较逻辑
  4. 基于上述结果，构造输入字符串

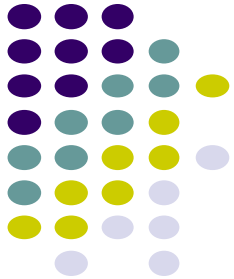
IEEE 754标准

Single / Double Precision：

S	Exponent	Significand
1 bit	8 / 11 bits	23 / 52 bits (SP/DP)

- **Sign bit: 1 表示negative ; 0表示 positive**
- **Exponent（阶码 / 指数编码）：**
  - **bias为127 (single), 1023 (double)**
- **Significand（尾数）：**
  - **规格化**：尾数最高位为1（隐含表示可省1位）
  - 非规格化：尾数最高位为0
  - 有效位数：1 + 23 bits（single），1 + 52 bits（double）

X87浮点指令（MMX和SSE指令）



- IA-32浮点处理架构有两种：
  - 浮点协处理器x87架构（x87 FPU）
    - ✓ 8个80位寄存器ST(0) ~ ST(7)（采用栈结构），栈顶为ST(0)
  - 由MMX发展而来的SSE架构
    - ✓ MMX指令使用8个64位寄存器MM0~MM7，借用8个80位寄存器ST(0)~ST(7)中64位尾数所占的位，可同时处理8个字节，或4个字，或2个双字，或一个64位的数据
    - ✓ SSE指令集采用SIMD（单指令多数据，也称数据级并行）技术，将80位浮点寄存器扩充到128位多媒体扩展通用寄存器XMM0~XMM7，可同时处理16个字节，或8个字，或4个双字（32位整数或单精度浮点数），或两个四字的数据。从SSE2开始，还支持128位整数运算或同时并行处理两个64位双精度浮点数

X87 FPU指令

- **数据传送类**
  - **装入**：FLDx、FILDx
  - **存储**：FSTx（FSTPx）、FISTx（FISTPx）
  - **交换**：FXCH
  - **常数装载到栈顶**：FLD1、FLDZ、FLDPI、FLDL2E、FLDL2T、 FLDLG2、FLDLN2 …

- **算术运算类**
  - **加法**：FADD / FADDP、FIADD
  - **减法**：FSUB / FSUBP、FSUBR / FSUBRP、FISUB、FISUBR
  - **乘法**：FMUL / FMULP、FIMUL
  - **除法**：FDIV / FDIVP、FIDIV、FDIVR / FDIVRP、FIDIVR

- 若指令未带操作数，则默认操作数为ST(0),ST(1)
- 带R后缀指令是指操作数顺序变反，例：
  fsub执行的是x-y，
  fsubr执行的就是y-x

例如： fidivl 0x8(%ebp)将指定存储单元操作数M[R[ebp]+8]中的int型数转换为double型，再将ST(0)除以该数，并将结果存入ST(0)中

X87 FPU指令

- **数据传送类**
  - ✓ FLD从内存将数据装入浮点寄存器栈顶ST(0)，FST则从浮点寄存器栈顶ST(0)将数据存入内存单元
  - ✓ 带P结尾的指令表示操作数会另出栈——ST(1)将变成ST(0)

  - ➢ **装入**

    FLDx: 将（x为s/l分别代表单/双精度浮点）数据装入浮点寄存器栈顶

    FILDx: 将数据从（x为空/l/ll分别代表16/32/64位）int型转换为浮点格式后，装入浮点寄存器栈顶

  - ➢ **存储**

    FSTx: 将栈顶ST(0)转换为（x为s/l分别代表单/双精度）浮点格式后存入存储单元

    FSTPx: 完成与FSTx相同的功能，并弹出栈顶元素

    FISTx: 将栈顶数据从浮点格式转换为int型（x为空/l/ll代表16/32/64位int），存入存储单元

    FISTPx: 完成与FISTx相同的功能，并弹出栈顶元素

实验阶段1 - 浮点数表示

- 步骤1：本阶段函数汇编指令代码分析
  - 定位与函数功能相对应的各部分控制逻辑；获得主要变量的地址

```
08048991 <phase_1>:
 8048991:  push   %ebp
 8048992:  mov    %esp,%ebp
 8048994:  sub    $0x28,%esp
 8048997:  movl   $0x224822d6,-0xc(%ebp)
 804899e:  fildl  -0xc(%ebp)
 80489a1:  fstpl  -0x18(%ebp)
 80489a4:  lea    -0x20(%ebp),%eax
 80489a7:  push   %eax
 80489a8:  lea    -0x1c(%ebp),%eax
 80489ab:  push   %eax
 80489ac:  push   $0x80495b4
 80489b1:  pushl  0x8(%ebp)
 80489b4:  call   80485d0 <__isoc99_sscanf@plt>
 80489b9:  add    $0x10,%esp
 80489bc:  cmp    $0x2,%eax
 80489bf:  je     80489cd <phase_1+0x3c>
 80489c1:  call   80492d6 <explode_bomb>
 80489c6:  mov    $0x0,%eax
 80489cb:  jmp    80489f9 <phase_1+0x68>
```

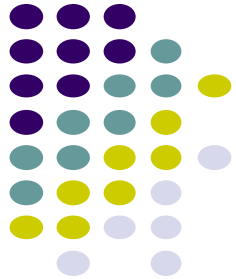
浮点常量初始化

读入数字对

```
 80489cd:  lea    -0x18(%ebp),%eax
 80489d0:  mov    (%eax),%edx
 80489d2:  mov    -0x1c(%ebp),%eax
 80489d5:  cmp    %eax,%edx
 80489d7:  jne    80489e8 <phase_1+0x57>
 80489d9:  lea    -0x18(%ebp),%eax
 80489dc:  add    $0x4,%eax
 80489df:  mov    (%eax),%edx
 80489e1:  mov    -0x20(%ebp),%eax
 80489e4:  cmp    %eax,%edx
 80489e6:  je     80489f4 <phase_1+0x63>
 80489e8:  call   80492d6 <explode_bomb>
 80489ed:  mov    $0x0,%eax
 80489f2:  jmp    80489f9 <phase_1+0x68>
 80489f4:  mov    $0x1,%eax
 80489f9:  leave
 80489fa:  ret
```

比较输入数字与浮点数

实验阶段1 - 浮点数表示



- 步骤2：从该阶段函数的汇编指令代码中，找到浮点数常量的值，获得其IEEE 754表示

```
08048991 <phase_1>:
 8048991:        push   %ebp
 8048992:        mov    %esp,%ebp
 8048994:        sub    $0x28,%esp
 8048997:        movl   $0x224822d6,-0xc(%ebp)
 804899e:        fildl  -0xc(%ebp)
 80489a1:        fstpl  -0x18(%ebp)
 ……
```

  - 从中可看出，整型值0x224822d6通过浮点栈被转为双精度浮点表示，并传送到本阶段函数栈帧中地址为-0x18(%ebp)处开始连续存放
  - 从课程内容可知，该常量值的双精度（double）IEEE 754表示为（十六进制字节串）：41 C1 24 11 6B 0 0 0

实验阶段1 - 浮点数表示

- **步骤3：分析程序的输入要求和比较逻辑**

- 输入拆解字符串的解析调用了sscanf函数，其原型如下：
  - int sscanf (const char * s, const char * format, ...);
  - 返回成功读入的项目个数
- 输入格式字符串起始地址位于0x80495b4，可使用gdb查看存放于该处的值
  - (gdb) x/1s 0x80495b4
  - 0x80495b4: "%d %d"
- 可见，输入的应是空格分隔的两个整数
  - （参考sscanf调用时参数压栈顺序）分别存储于-0x1c(%ebp)、-0x20(%ebp)

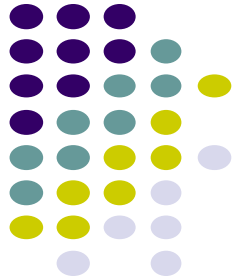
```
80489a4:    lea   -0x20(%ebp),%eax
80489a7:    push  %eax
80489a8:    lea   -0x1c(%ebp),%eax
80489ab:    push  %eax
80489ac:    push  $0x80495b4
80489b1:    pushl 0x8(%ebp)
80489b4:    call  80485d0
            <__isoc99_sscanf@plt>
```

实验阶段1 - 浮点数表示

- 步骤3（续）：分析程序的输入要求和比较逻辑
  - 变量及其存储地址
    1. -0x18(%ebp)：浮点数
    2. -0x1c(%ebp)：第1个输入整数
    3. -0x20(%ebp)：第2个输入整数
  - 0x80489cd-d7处比较了第1个输入整数与浮点数起始存储地址处（小端表示中是其低32位）整数是否相等
  - 0x80489d9-e6处比较了第2个输入整数与浮点数高4个字节存储地址处（小端表示中是其高32位）整数是否相等
  - **因此，输入字符串中应包含分别对应该浮点数表示低32位的整数和高32位的整数**

```
80489cd:        lea    -0x18(%ebp),%eax
80489d0:        mov    (%eax),%edx
80489d2:        mov    -0x1c(%ebp),%eax
80489d5:        cmp    %eax,%edx
80489d7:        jne    80489e8
<phase_1+0x57>
80489d9:        lea    -0x18(%ebp),%eax
80489dc:        add    $0x4,%eax
80489df: mov    (%eax),%edx
80489e1:        mov    -0x20(%ebp),%eax
80489e4:        cmp    %eax,%edx
80489e6:        je     80489f4 <phase_1+0x63>
80489e8:        call   80492d6
<explode_bomb>
80489ed:        mov    $0x0,%eax
80489f2: jmp    80489f9 <phase_1+0x68>
80489f4: mov    $0x1,%eax
80489f9: leave
80489fa: ret
```

实验阶段1 - 浮点数表示



- 步骤4：基于前述分析，构造输入字符串
  - 整型值0x224822d6对应的双精度浮点数的IEEE 754表示为（十六进制字节串，从高位到低位）：

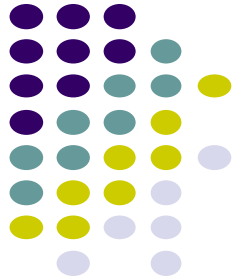
    41 C1 24 11 6B 0 0 0
  - 低32位（从高位到低位）并转为十进制有符号整数：

    6B 0 0 0 = 1795162112
  - 高32位（从高位到低位）并转为十进制有符号整数：

    41 C1 24 11 = 1103176721
  - 因此拆解字节串应为“**1795162112 1103176721**”

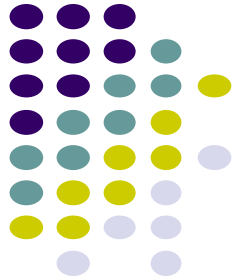
小结

- 本节课介绍了二进制炸弹实验第二阶段（即阶段1-浮点数表示）的基本过程和方法，并且演示了其中的主要操作步骤。从中，我们以实例分析与操作的方式，巩固了对课程中浮点数的IEEE 754表示、基于浮点栈的浮点传送指令、基于栈帧的过程参数传递、过程参数与局部变量的访问等方面知识的掌握。



二进制炸弹实验：课后实验

实验其余阶段



- 二进制炸弹程序的其余阶段作为课后实验自行完成（具体阶段以本学期课程的实验文档为准）
  - 阶段0：字符串比较
  - 阶段1：浮点数表示
  - 阶段2：循环
  - 阶段3：条件/分支
  - 阶段4：递归调用和栈
  - 阶段5：指针
  - 阶段6：链表/指针/结构
  - 隐藏阶段：只有在阶段4的拆解字符串后再附加一特定字符串后才会出现

实验阶段secret

- 分析获得进入secret阶段的条件

1. 在bomb反汇编结果中寻找对secret_phase函数的调用之处——位于phase_defused函数中
2. 分析输入字符串的格式要求

   int sscanf (const char * s, const char * format, ...);

   - 使用GDB获得格式字符串为：

     (gdb) x/1s 0x80496ee

     0x80496ee: "%d %d %s"

   - 对照阶段4所要求的输入两个数字，为进入隐藏阶段，其后应再输入一字符串，并被保存于-0x5c(%ebp)起始的地址中

```
080492ff <phase_defused>:
……
8049312:        lea    -0x5c(%ebp),%eax
8049315:        push   %eax
8049316:        lea    -0x64(%ebp),%eax
8049319:        push   %eax
804931a:        lea    -0x60(%ebp),%eax
804931d:        push   %eax
804931e:        push   $0x80496ee
8049323:        push   $0x804b3c0
8049328:        call   80485d0
                <__isoc99_sscanf@plt>
804932d:        add    $0x20,%esp
8049330:        mov    %eax,-0xc(%ebp)
8049333:        cmpl   $0x3,-0xc(%ebp)
8049337:        jne    8049376
                <phase_defused+0x77>
8049339:        sub    $0x8,%esp

……
```

实验阶段secret

- 分析获得进入secret阶段的条件（续）

3. 为使控制能够到达0x8049371处对secret_phase函数的调用，地址0x804934f处的jne指令应不执行跳转，即eax寄存器的值在strings_not_equal函数返回后应为0

```
080492ff <phase_defused>:
 ……
804933c:        push   $0x80496f7
 8049341:       lea    -0x5c(%ebp),%eax
 8049344:       push   %eax
 8049345:       call   804906e
                  <strings_not_equal>
 804934a:       add    $0x10,%esp
 804934d:       test   %eax,%eax
 804934f:       jne    8049376
                <phase_defused+0x77>
 8049351:       sub    $0xc,%esp
 8049354:       push   $0x8049700
 8049359:       call   8048590
                       <puts@plt>
 804935e:       add    $0x10,%esp
 8049361:       sub    $0xc,%esp
 8049364:       push   $0x8049728
 8049369:       call   8048590
                       <puts@plt>
 804936e:       add    $0x10,%esp
 8049371:       call   8048e49
                  <secret_phase>
 ……
```

实验阶段secret

- 分析获得进入secret阶段的条件（续）

4. 分析strings_not_equal函数的汇编代码，可知该函数在两个输入字符串参数的长度和内容均相同时将返回0，否则返回1，并且返回值保存于EAX寄存器中

```
0804906e <strings_not_equal>:
 ……
8049075:   pushl  0x8(%ebp)
8049078:   call   8049042 <string_length>
804907d:   add    $0x4,%esp
8049080:   mov    %eax,%ebx
8049082:   pushl  0xc(%ebp)
8049085:   call   8049042 <string_length>
804908a:   add    $0x4,%esp
804908d:   cmp    %eax,%ebx
804908f:   je     8049098
8049091:   mov    $0x1,%eax
8049096:   jmp    80490d4
8049098:   mov    0x8(%ebp),%eax
804909b:   mov    %eax,-0x8(%ebp)
804909e:   mov    0xc(%ebp),%eax
80490a1:   mov    %eax,-0xc(%ebp)
80490a4:   jmp    80490c5
80490a6:   mov    -0x8(%ebp),%eax
80490a9:   movzbl (%eax),%edx
80490ac:   mov    -0xc(%ebp),%eax
80490af:   movzbl (%eax),%eax
80490b2:   cmp    %al,%dl
80490b4:   je     80490bd
 ……
```

实验阶段secret

- 分析获得进入secret阶段的条件（续）

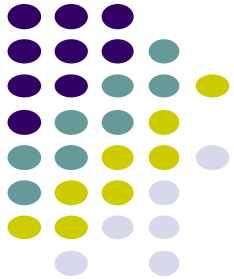
5. 因此，保存于地址-0x5c(%ebp)的输入字符串，与保存于地址0x80496f7的一个字符串，在内容上应相同
6. 使用GDB获得0x80496f7地址处的字符串的内容：

   (gdb) x/1s 0x80496f7

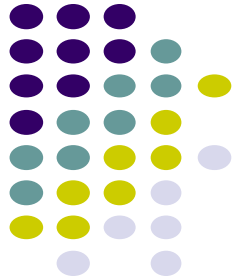
   0x80496f7: "MOclQcP"

   - 这个就是应在阶段4输入最后附加的用以触发secret_phase的字符串

```
080492ff <phase_defused>:
 ……
804933c:        push   $0x80496f7
 8049341:       lea    -0x5c(%ebp),%eax
 8049344:       push   %eax
 8049345:       call   804906e
                  <strings_not_equal>
 804934a:       add    $0x10,%esp
 804934d:       test   %eax,%eax
 804934f:       jne    8049376
                <phase_defused+0x77>
 8049351:       sub    $0xc,%esp
 8049354:       push   $0x8049700
 8049359:       call   8048590
                        <puts@plt>
 804935e:       add    $0x10,%esp
 8049361:       sub    $0xc,%esp
 8049364:       push   $0x8049728
 8049369:       call   8048590
                        <puts@plt>
 804936e:       add    $0x10,%esp
 8049371:       call   8048e49
                  <secret_phase>
 ……
```



实验结果提交



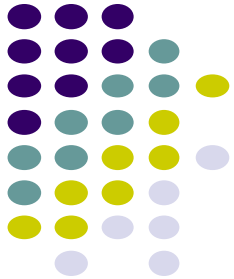
- 将对应二进制炸弹每一阶段的拆解字符串写入一文本文件，命名为“学号.txt”，其中每个拆解字符串独立一行，例如（文件实际包含的行应与本学期实验阶段相对应）：

  string0
  string1
  string2
  string3
  ...

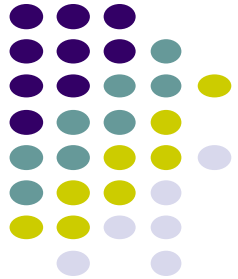
  - 该文本文件必须采用Unix格式，其中的换行字符不同于Windows格式
  - 最后一个字符串后也要进行换行，即所有字符串必须以换行结尾
  - 如果未完成其中某阶段，可用任一非空白字符串置于文本文件中对应该阶段的相应行中（不能放置一空行或直接省略该行）

- 提交该结果文本文件

小结



- 本节课将二进制炸弹实验剩余阶段布置为课后实验自行完成，并介绍了隐藏阶段进入条件的分析过程。此外，说明了实验结果的提交形式。



结 束