今晚 217 答疑 (19:00 – 20:45)

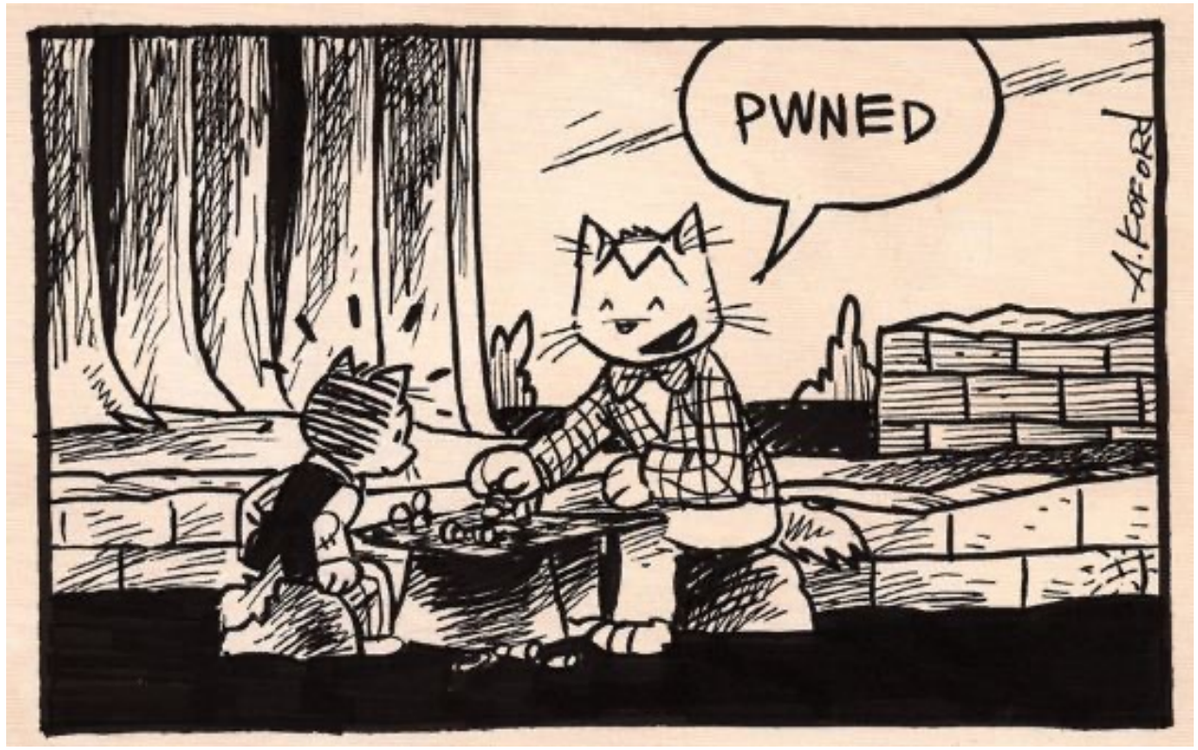
如果前几次作业遇到问题的可以来
不一定能解决
但可以一起想办法 ☺

@所有人

下午的课程讲基础pwn，可能用到的示例代码已经传在学在浙大了，请查收

P.S. 会用到 linux 环境所以导论作业还没完成的可以迅速完成一下

pwn 基础导论



Acknowledgement: Some slides are based on Zhi Wang (FSU)'s slides.

Outline

- 一些基础部分
- What is PWN?
- stack buffer overflow
- ~~shellcode 基础~~
- ~~常见的保护方式与绕过~~

基础恶补

目前了解到的几个 linux 环境

- 虚拟机
- WSL
- docker
- 双系统
- 远程服务器

再谈 ssh

计算机内存架构

- 寄存器 (register) – 内存 (memory) – 存储 (storage)

Quick Review

前几节课学的咋样，这些词还在脑海里么？

堆栈布局、静态链接和动态链接、共享库

GOT、PLT 与 lazy binding

调用约定

dynamic debug in Linux 手把手

正好学学汇编

特定命令

结合动态调试理解一些之前的概念

remote shell + gdb + helloworld / or attachment

```
sudo apt install gdb
```

- 32 bit arch
  - sudo dpkg --add-architecture i386
  - sudo apt install libc6:i386 libstdc++6:i386
  - sudo apt update
  - sudo apt install libncurses5-dev lib32z1
  - sudo apt install gcc-multilib

- 64 bit arch
  - cool

并且配置 gdb 插件 (peda, pwndbg, gef)

- 直接 gdb 启动以及 attach 已有进程

系统调用

- 课程末尾了解 shellcode 的时候需要学习
- hello world + strace demo

调试重看 GOT 和 PLT

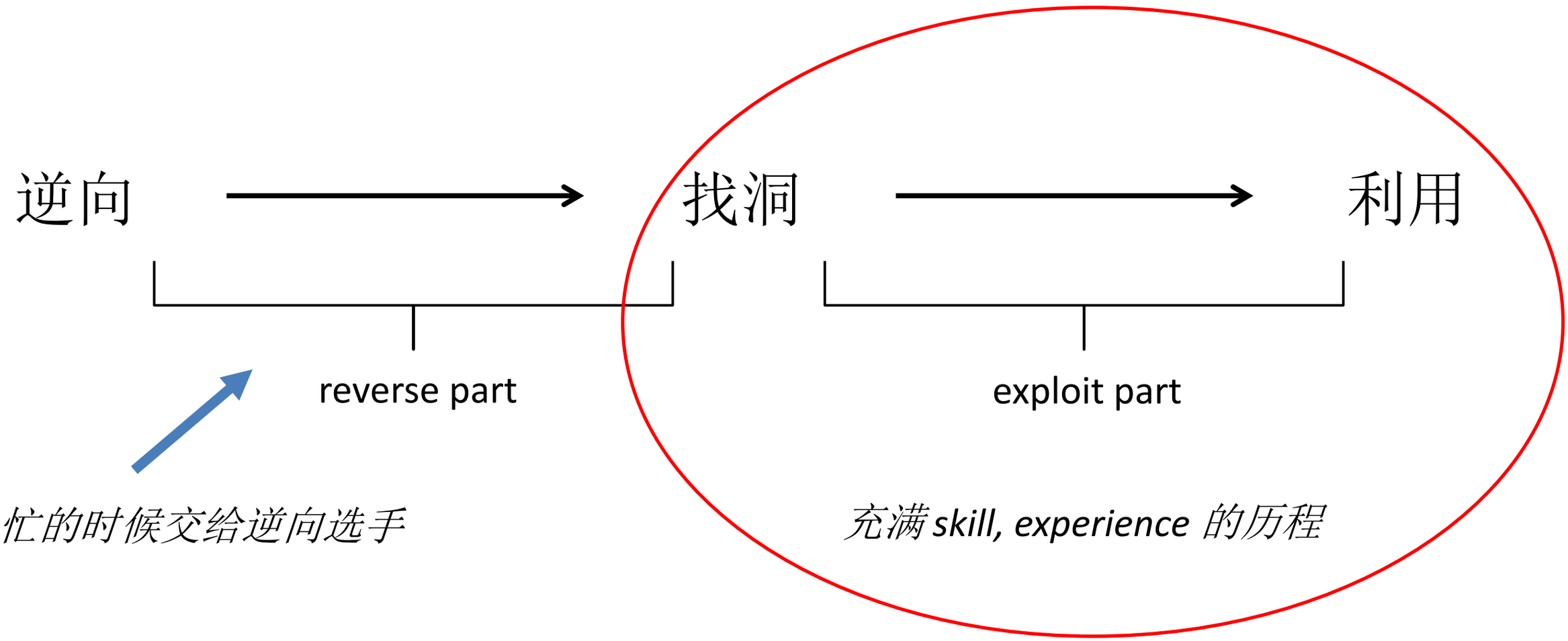
跟进 printf 看看发生了啥

- 注意 libc 加载的基地址
- 结合 readelf
- 查看 proc maps
- 实际思考动态链接和静态链接

休息一下咯

思考题：为啥 gdb 启动和 gdb attach 启动存在一些奇怪的区别

What is **PWN**


逆向
找洞
利用
reverse part
exploit part
忙的时候交给逆向选手
充满 skill, experience 的历程

所以，「洞」是什么

首先，洞肯定是 programmer 不期待在程序中出现的**问题**

如下程序有啥问题?

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
  int divdend;
  int divisor;
  float result;
  scanf("%d", &divdend);
  scanf("%d", &divisor);
  result = divdend / divisor;
  printf("%d / %d = %f\n", divdend, divisor, result);
  return 0;
}
```

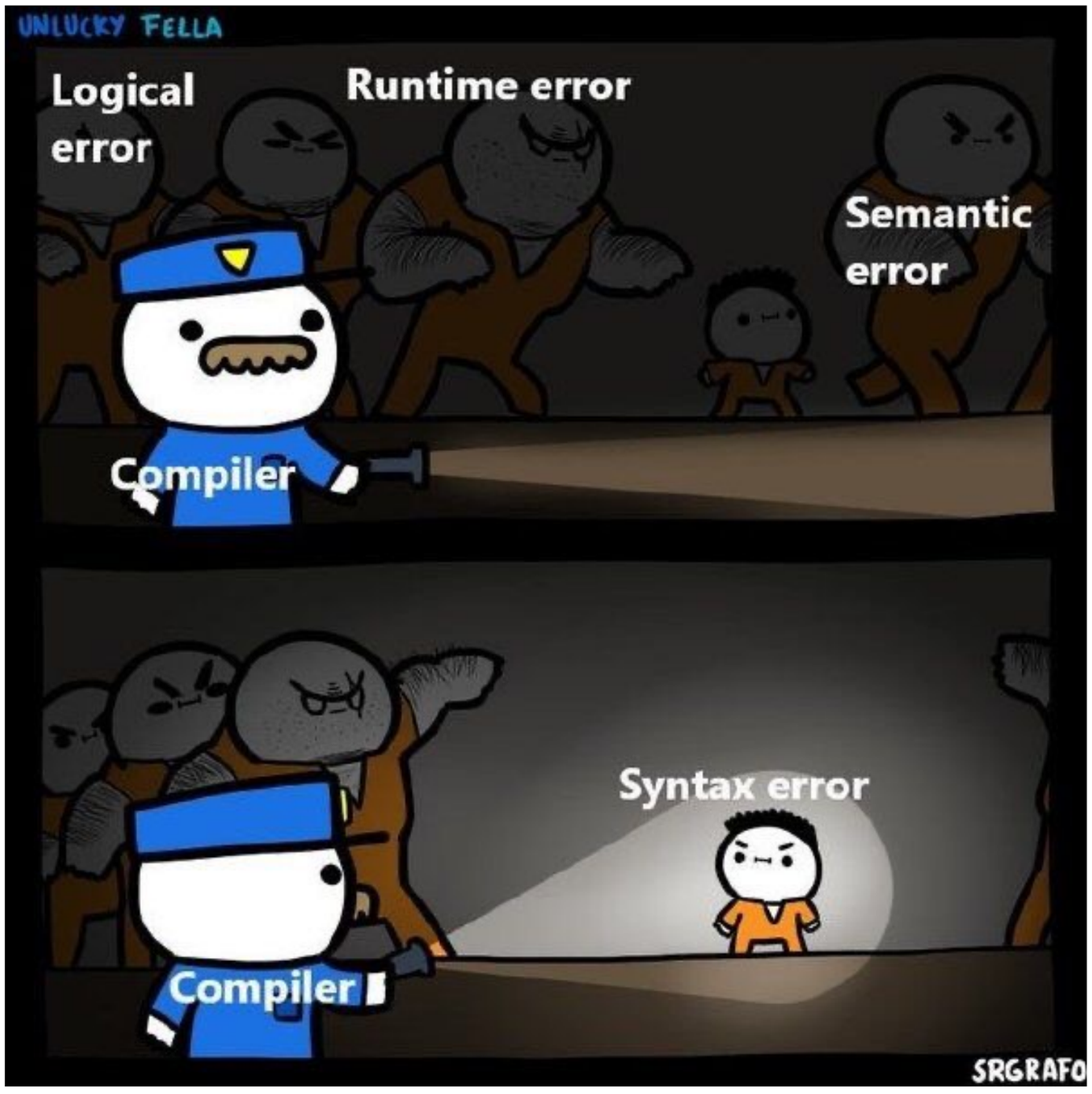
值得感恩的是，许多问题编译器已经发现

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    int buffer[5] = {0,1,2,3,4};
    buffer[5] = 5;
    return 0;
}
```

```
clang -Wall overflow.c
overflow.c:6:5: warning: array index 5 is past the end of the array (which contains 5
elements) [-Warray-bounds]
    buffer[5] = 5;
    ^      ~
overflow.c:5:5: note: array 'buffer' declared here
    int buffer[5] = {0,1,2,3,4};
    ^
1 warning generated.
```

但是




```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    int buffer[5] = {0,1,2,3,4};
    int index = getchar();
    buffer[index] = 5;
    return 0;
}
```

Generally speaking

A Software Bug is a failure or flaw in a program that produces undesired or incorrect results.

It’s an error that prevents the application from functioning as it should.

毕竟，不会有人编程的目的就是写 BUG 吧

Bug的成因1 – 没学明白

```c
#include <stdio.h>


int main(int argc, char* argv[])
{
    long d = 0x12345678;
    printf("value of d: %s\n", d);

    return 0;
}
```

Bug的成因2 – 大意了，没有闪

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    char buffer[16] = {0};

    for (int i = 0; i <= 16; i++) {
        scanf("%c", buffer + i);
    }

    return 0;
}
```

believe it or not

有洞很正常，没洞才罕见

because

1. corner cases造成的漏洞本身就是反人类编程思维的

2. 复杂大型应用如操作系统浏览器就有那么多洞啊
   (每千行代码漏洞比例）

3. 更新迭代过程中可能发生的“顾此失彼”

.....

When Bug becomes Vulnerability

minor security bugs:

- local Denial of Service
- local information leak

major security bugs:

- local privilege escalation
- remote Denial of Service
- remote information leak
- remote code execution

PWN

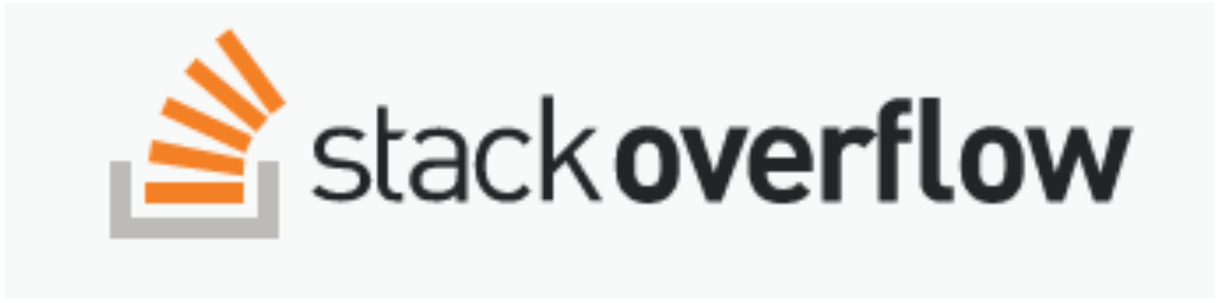
在 CTF 赛题中，pwn 类型赛题即要求 player 找到 target 程序中出题人 intended “隐藏”的 BUG，并通过特定的利用方式最终构建代码执行能力，并读取 flag 内容

非预期：发现出题人都没注意到引入的 BUG，并利用且完成题解

Example – ACTF

kkk: overflow in encryption / decryption

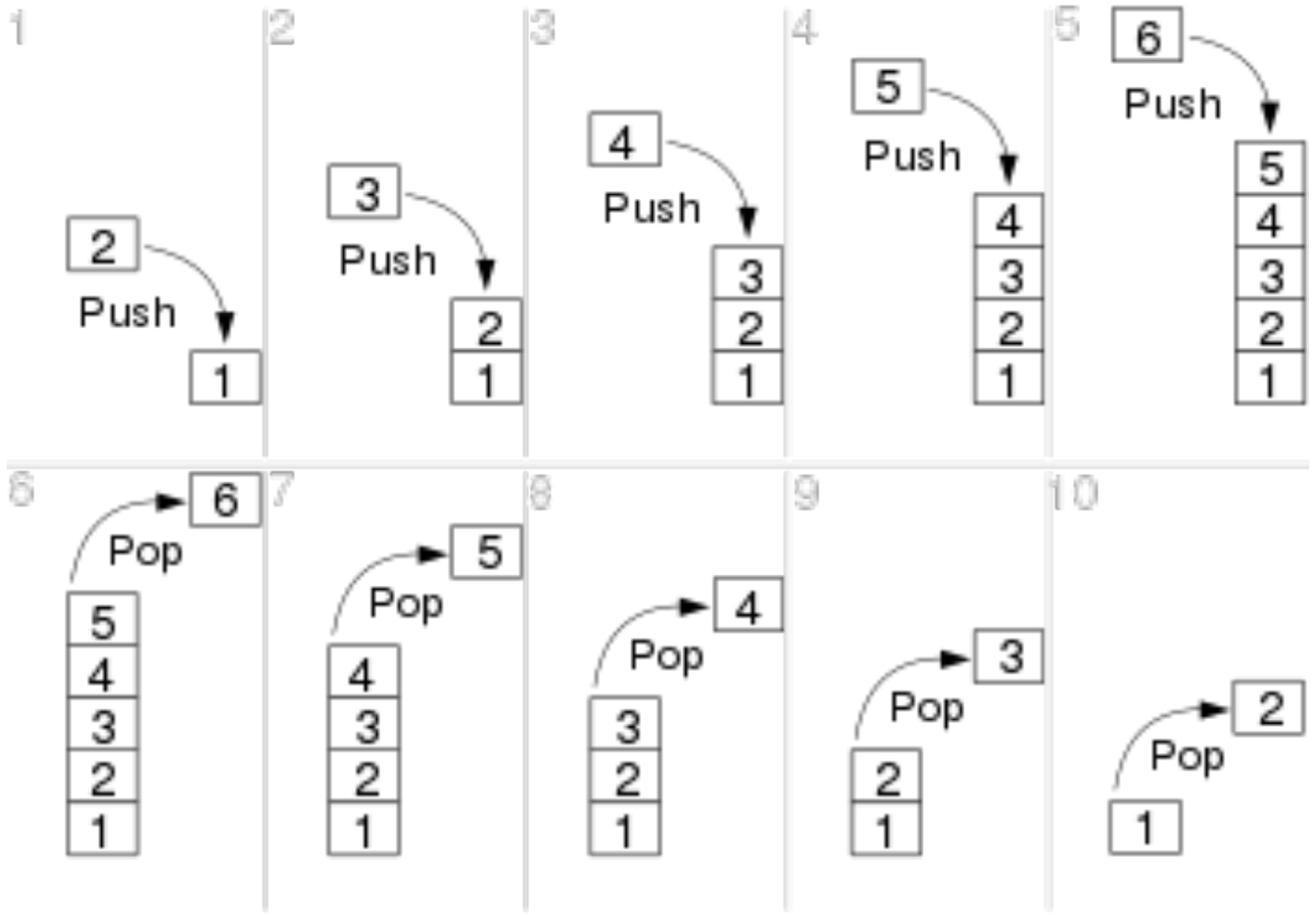
stack buffer overflow



stack buffer overflow

stack data structure　　先进后出即为栈

抽象一个 *PUSH* 操作，一个 *POP* 操作

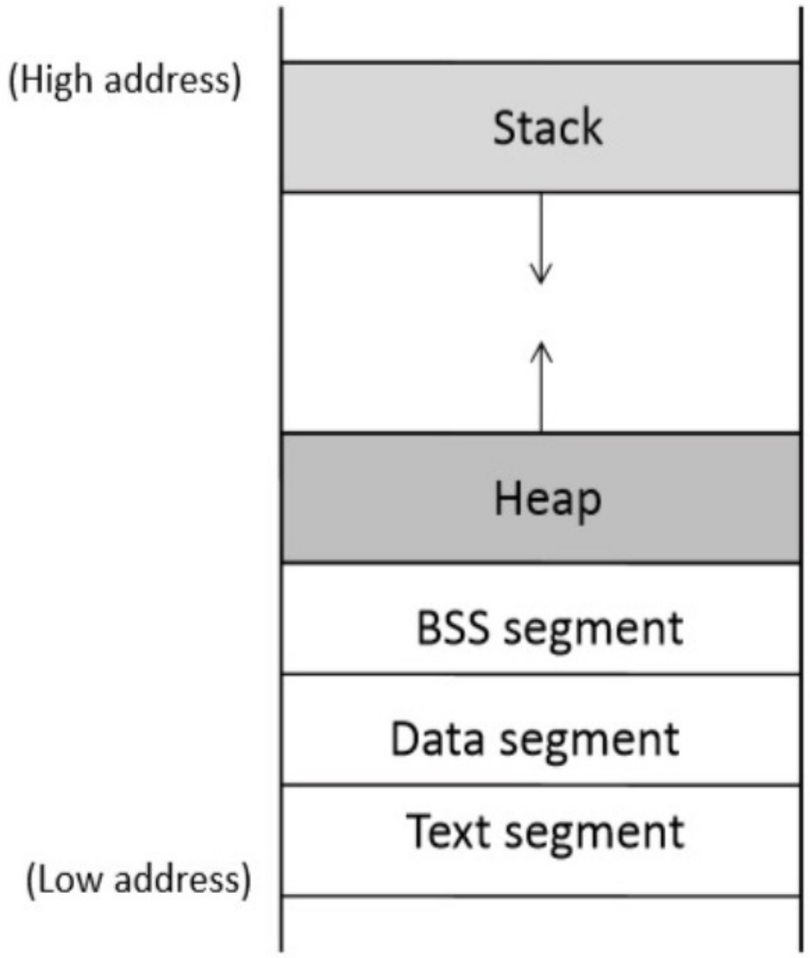


pictures from wikipedia

从内存的布局角度说

已知：

- 代码段，放可执行代码
- 数据段，放初始化完成的静态以及的全局变量
- BSS段，放未初始化的静态以及全局变量
- 堆：动态内存管理
- 栈：存放运行时的**局部变量**，返回地址，调用参数等



记住就好

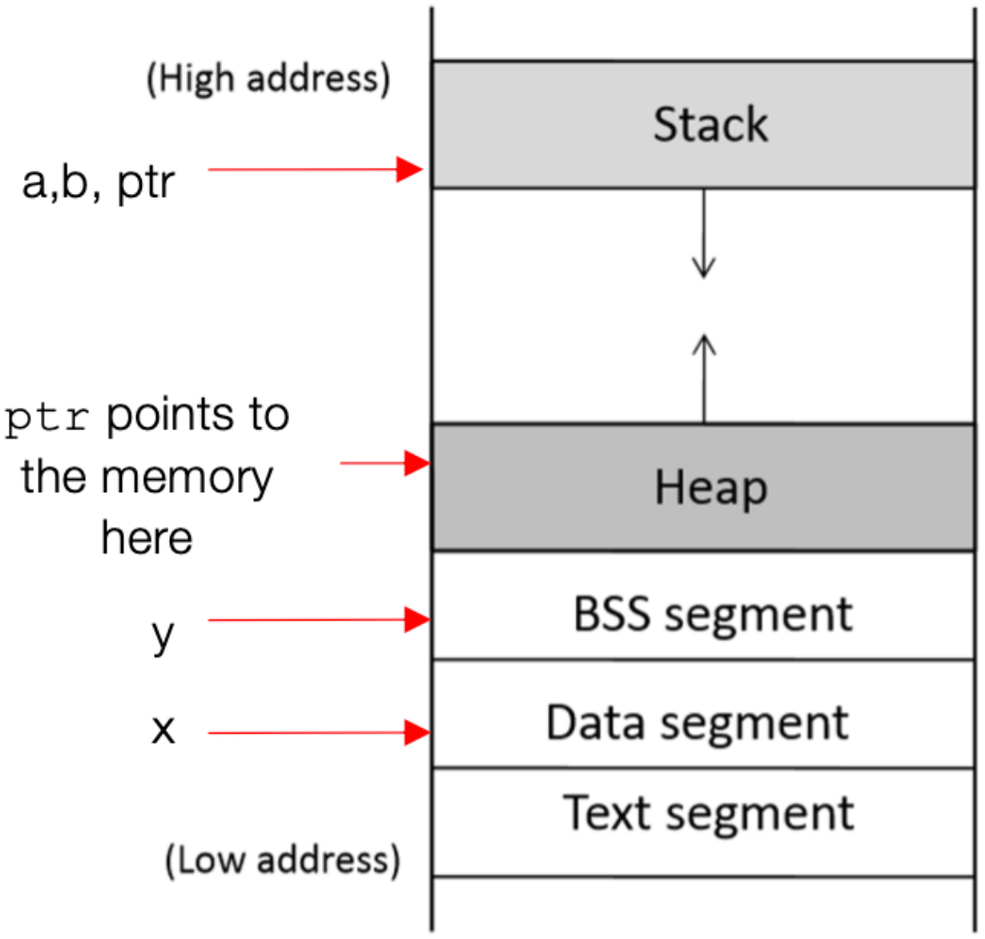
```c
int x = 100;
int main()
{
  // data stored on stack
  int   a=2;
  float b=2.5;
  static int y;

  // allocate memory on heap
  int *ptr = (int *) malloc(2*sizeof(int));

  // values 5 and 6 stored on heap
  ptr[0]=5;
  ptr[1]=6;

  // deallocate memory on heap
  free(ptr);

  return 1;
}
```



栈布局

源代码

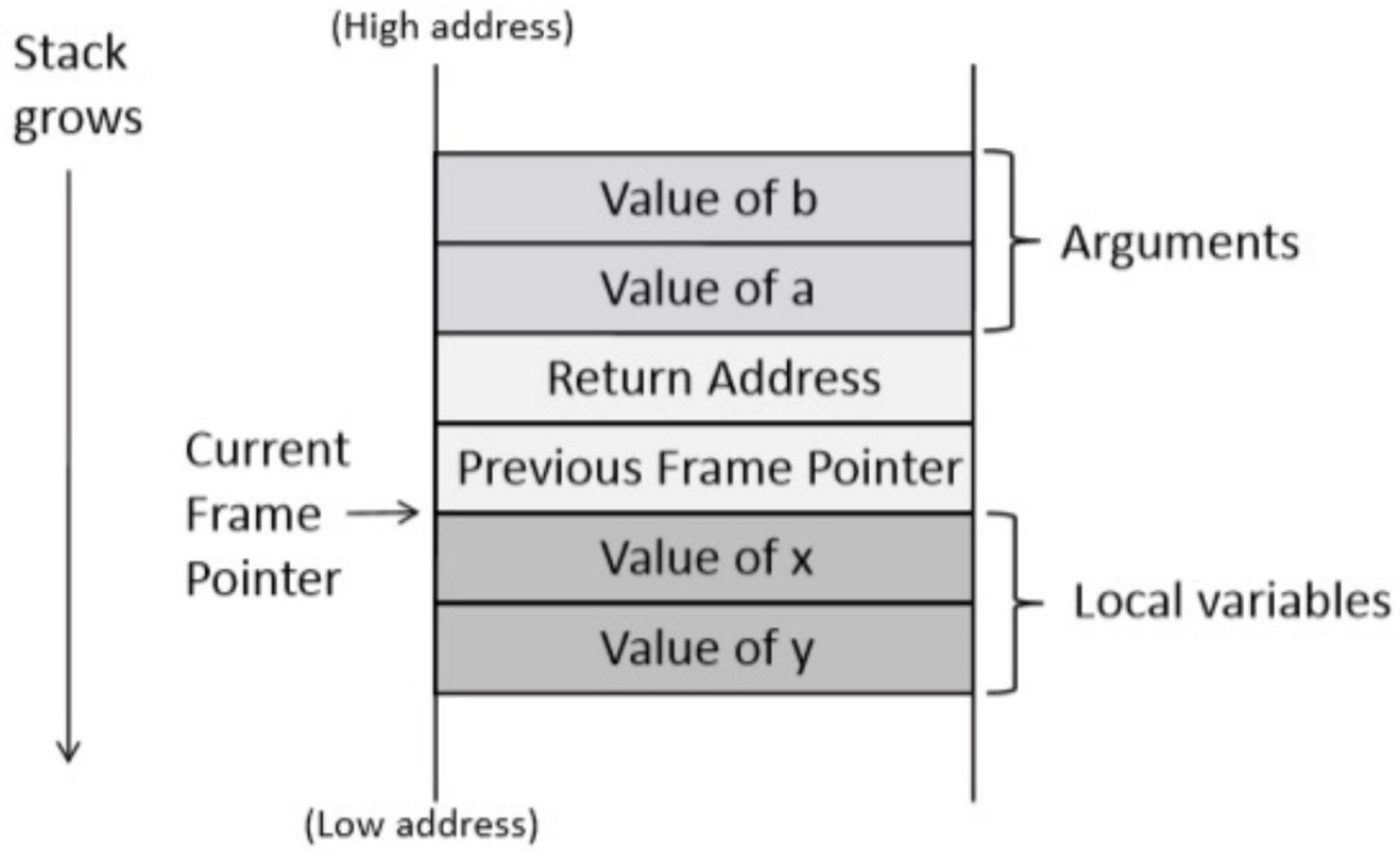
```
void func(int a, int b)
{
    int x, y;

    x = a + b;
    y = a - b;

}
```

栈布局

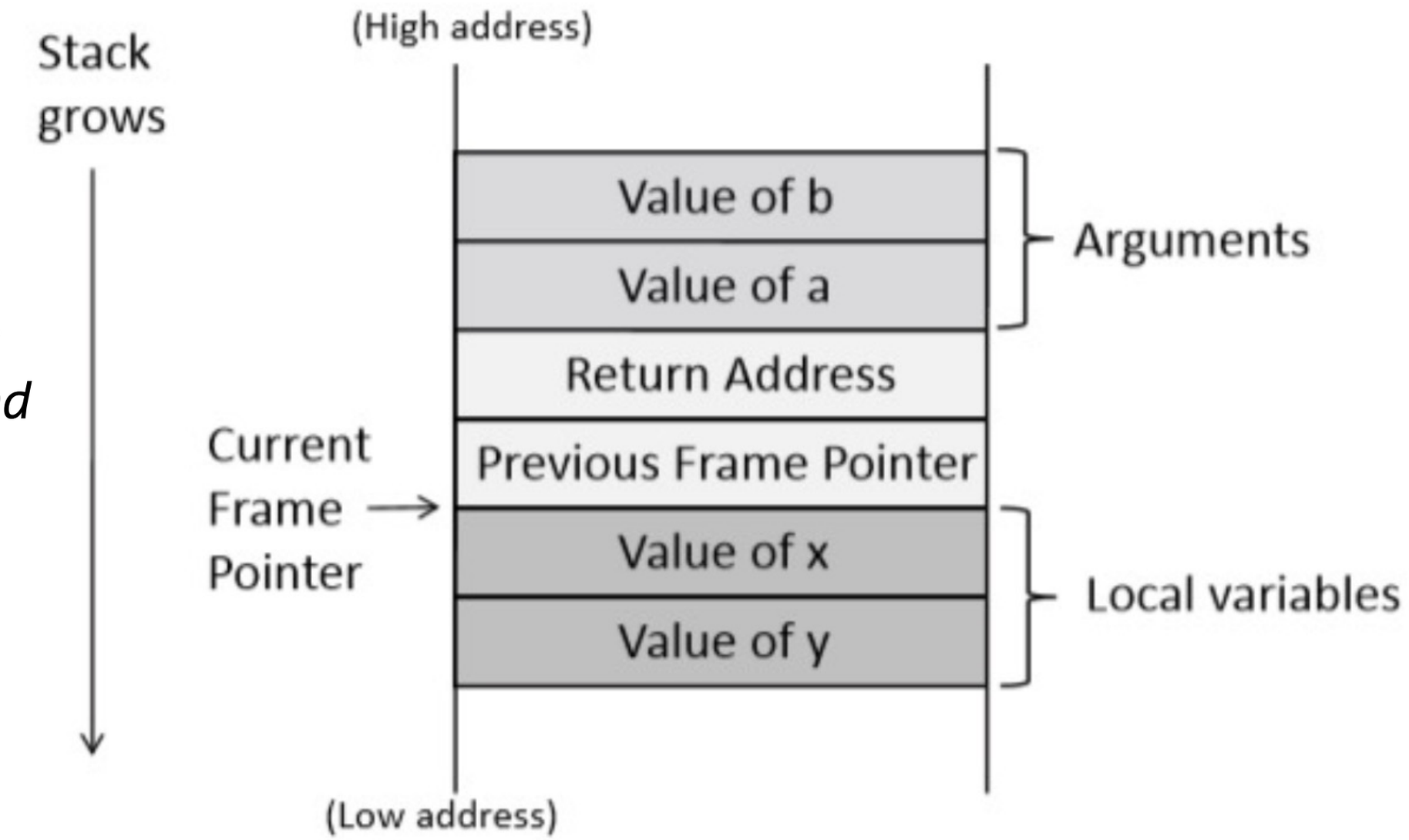
静态栈结构（理论）


(High address)
Stack
grows
Value of b
Value of a
Arguments
Return Address
Current
Frame
Pointer
Previous Frame Pointer
Value of x
Value of y
Local variables
(Low address)

栈布局

静态栈结构（理论）

stack frame
activation frame
activation record



frame pointer

stack frame pointer is used to access local variables

- stack frame pointer + offset = local variables
- stack frame pointer is set during runtime

```
movl    12(%ebp), %eax      ; b is stored in %ebp + 12
movl    8(%ebp), %edx       ; a is stored in %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x is stored in %ebp - 8
```

$$x = a + b$$

敲桌：一定要理解并会画栈的图（func.c）

- 现文字看然后调试看
- 一定要分清楚 grow 的方向

actually - 32 (func.c)

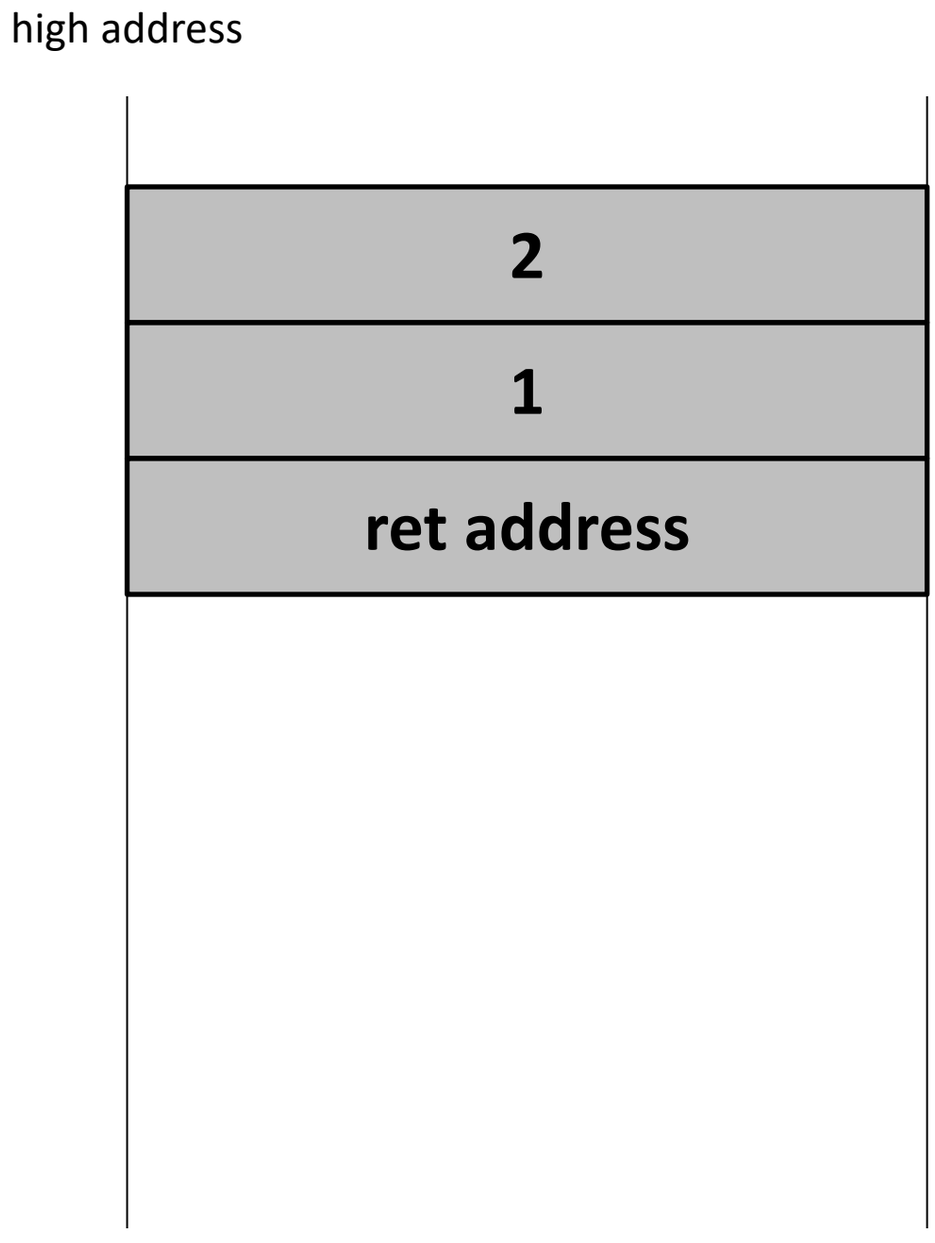
```
00000514 <main>:
 514:   55                              push   %ebp
 515:   89 e5                           mov    %esp,%ebp
 517:   e8 18 00 00 00                  call   534 <__x86.get_pc_thun
51c:   05 c0 1a 00 00                  add    $0x1ac0,%eax
521:   6a 02                           push   $0x2
523:   6a 01                           push   $0x1
525:   e8 c3 ff ff ff                  call   4ed <func>
52a:   83 c4 08                        add    $0x8,%esp
52d:   b8 00 00 00 00                  mov    $0x0,%eax
532:   c9                              leave
533:   c3                              ret
```

actually - 32 (func.c)

```
000004ed <func>:
 4ed:   55                      push   %ebp
 4ee:   89 e5                   mov    %esp,%ebp
 4f0:   83 ec 10                sub    $0x10,%esp
 4f3:   e8 3c 00 00 00          call   534 <__x86.get_pc_thunk.ax>
 4f8:   05 e4 1a 00 00          add    $0x1ae4,%eax
 4fd:   8b 55 08                mov    0x8(%ebp),%edx
 500:   8b 45 0c                mov    0xc(%ebp),%eax
 503:   01 d0                   add    %edx,%eax
 505:   89 45 f8                mov    %eax,-0x8(%ebp)
 508:   8b 45 08                mov    0x8(%ebp),%eax
 50b:   2b 45 0c                sub    0xc(%ebp),%eax
 50e:   89 45 fc                mov    %eax,-0x4(%ebp)
 511:   90                      nop
 512:   c9                      leave
 513:   c3                      ret
```

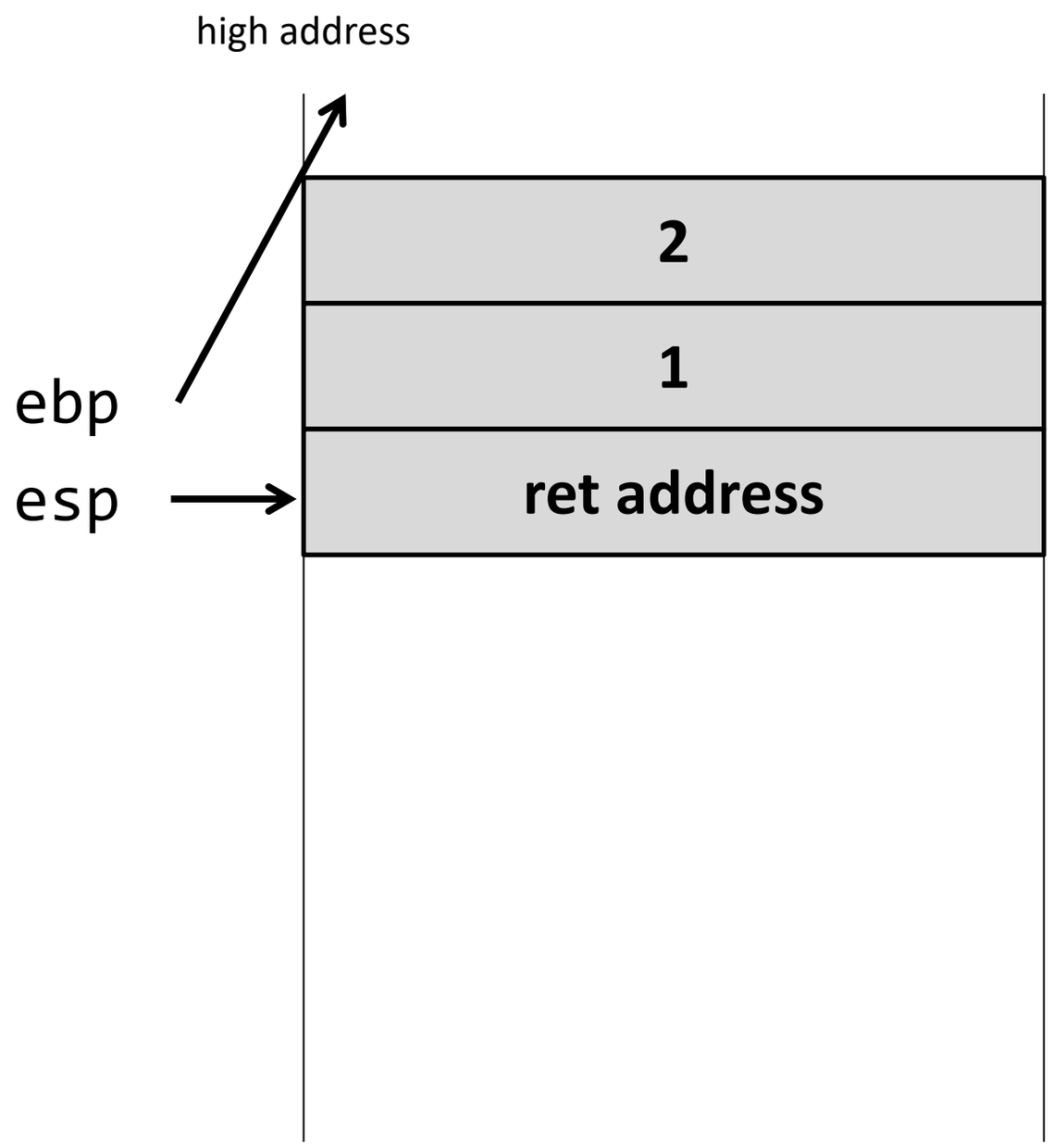
actually – 32位 x86

```
1. push $0x2
2. push $0x1
3. call func
```

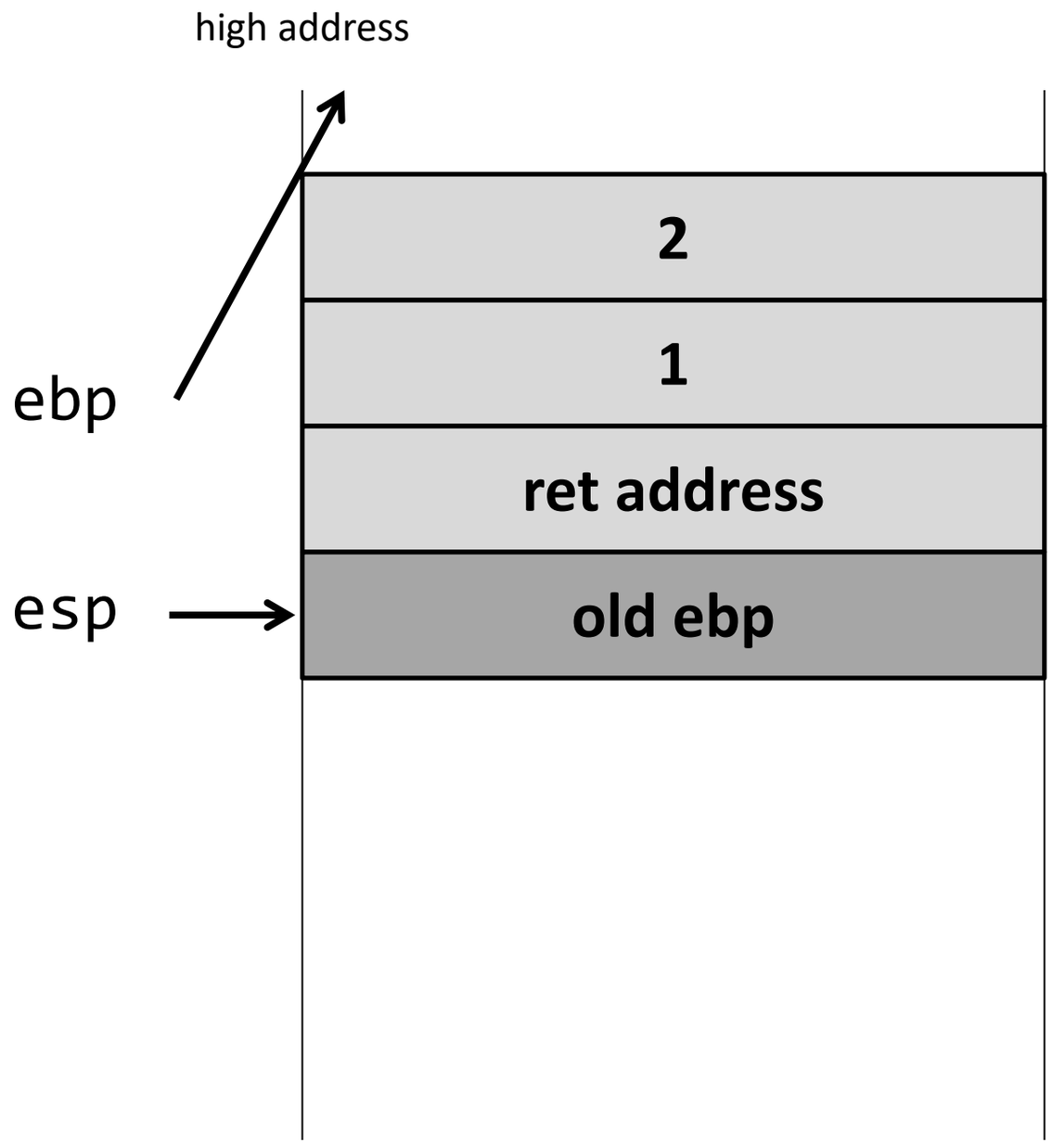


actually – 32位 x86

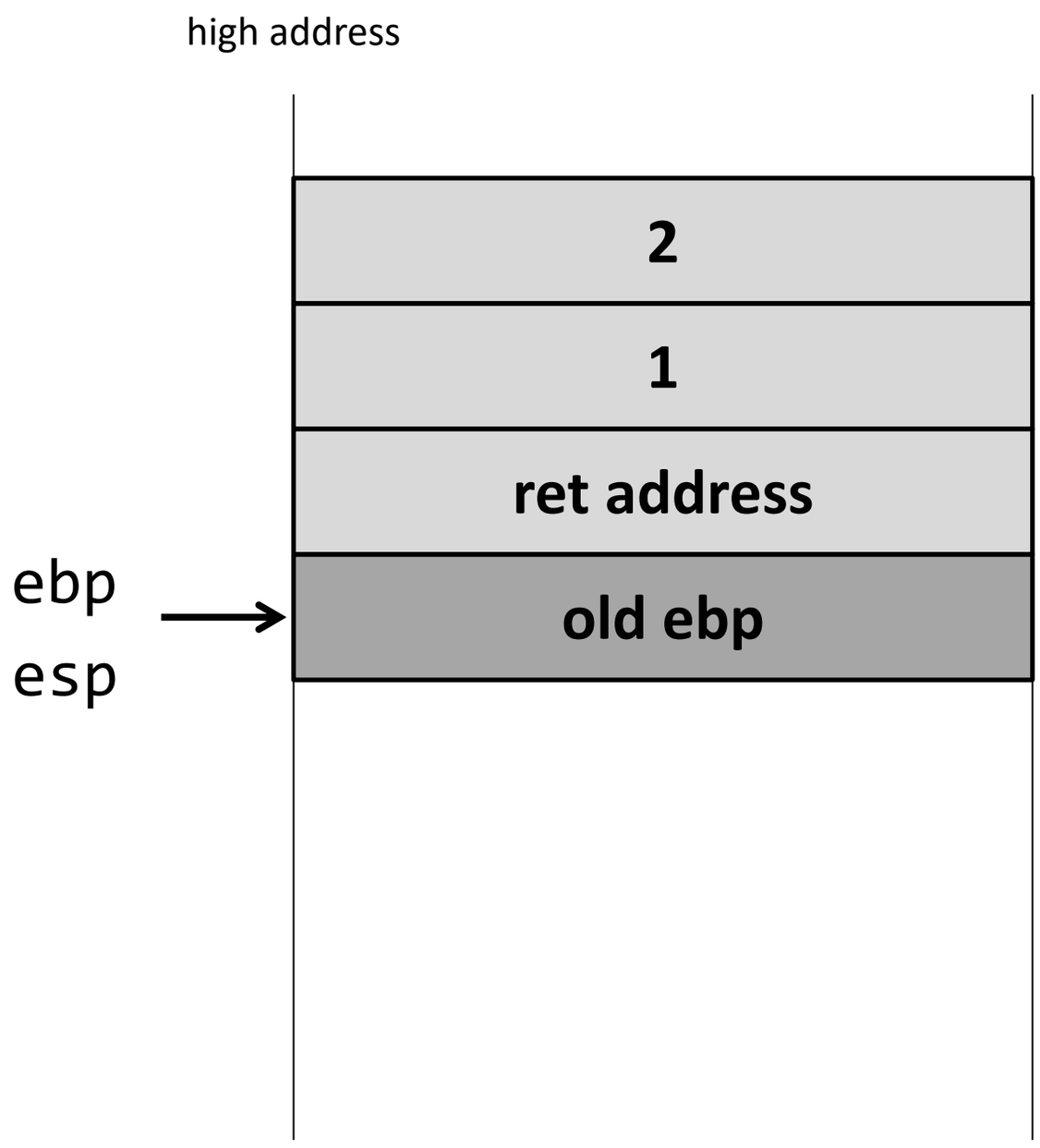
目前的stack寄存器和
frame pointer 寄存器



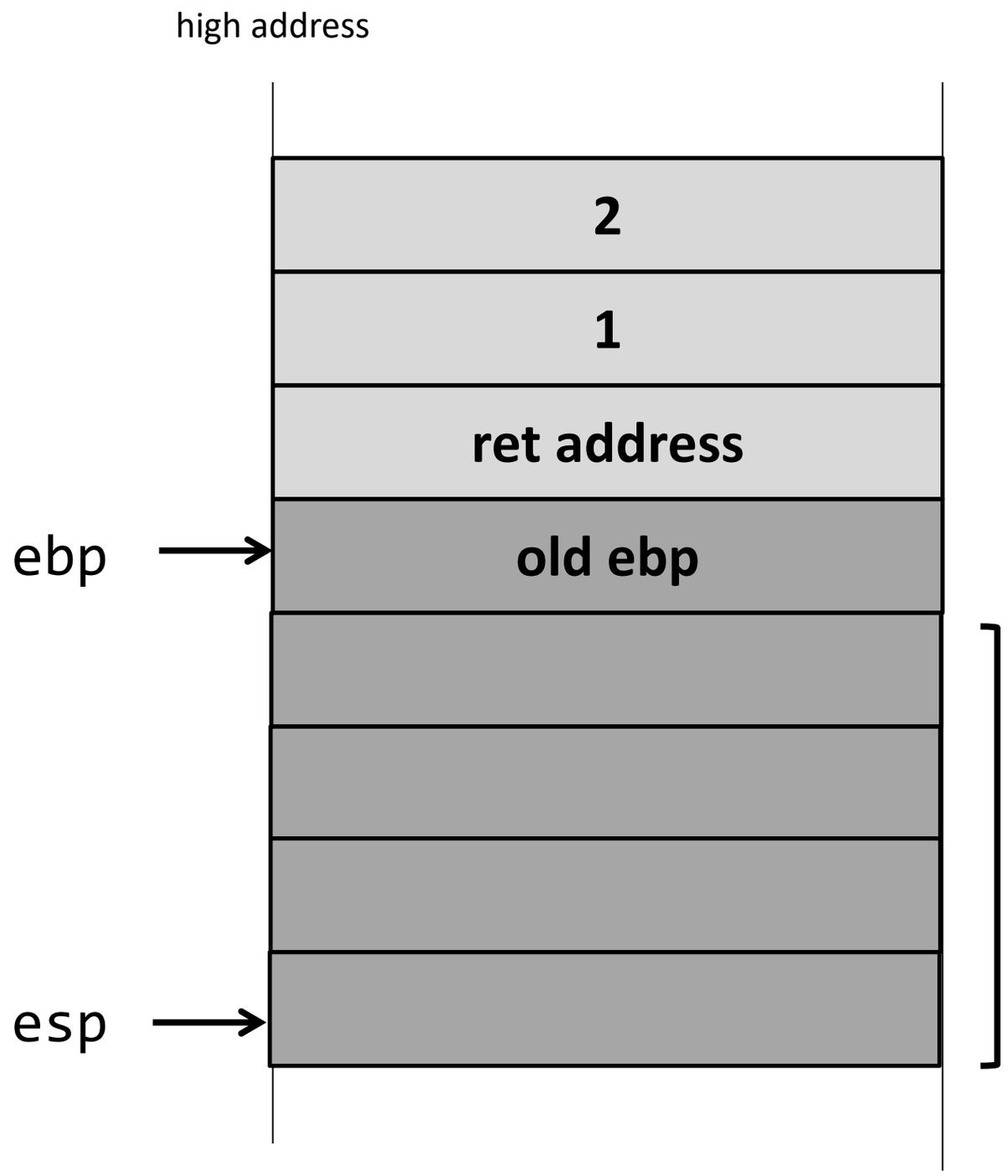
```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
call   534 <__x86.get_pc_thunk.ax>
add    $0x1ae4,%eax
mov    0x8(%ebp),%edx
mov    0xc(%ebp),%eax
add    %edx,%eax
mov    %eax,-0x8(%ebp)
mov    0x8(%ebp),%eax
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
nop
leave
ret
```



```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
call   534 <__x86.get_pc_thunk.ax>
add    $0x1ae4,%eax
mov    0x8(%ebp),%edx
mov    0xc(%ebp),%eax
add    %edx,%eax
mov    %eax,-0x8(%ebp)
mov    0x8(%ebp),%eax
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
nop
leave
ret
```

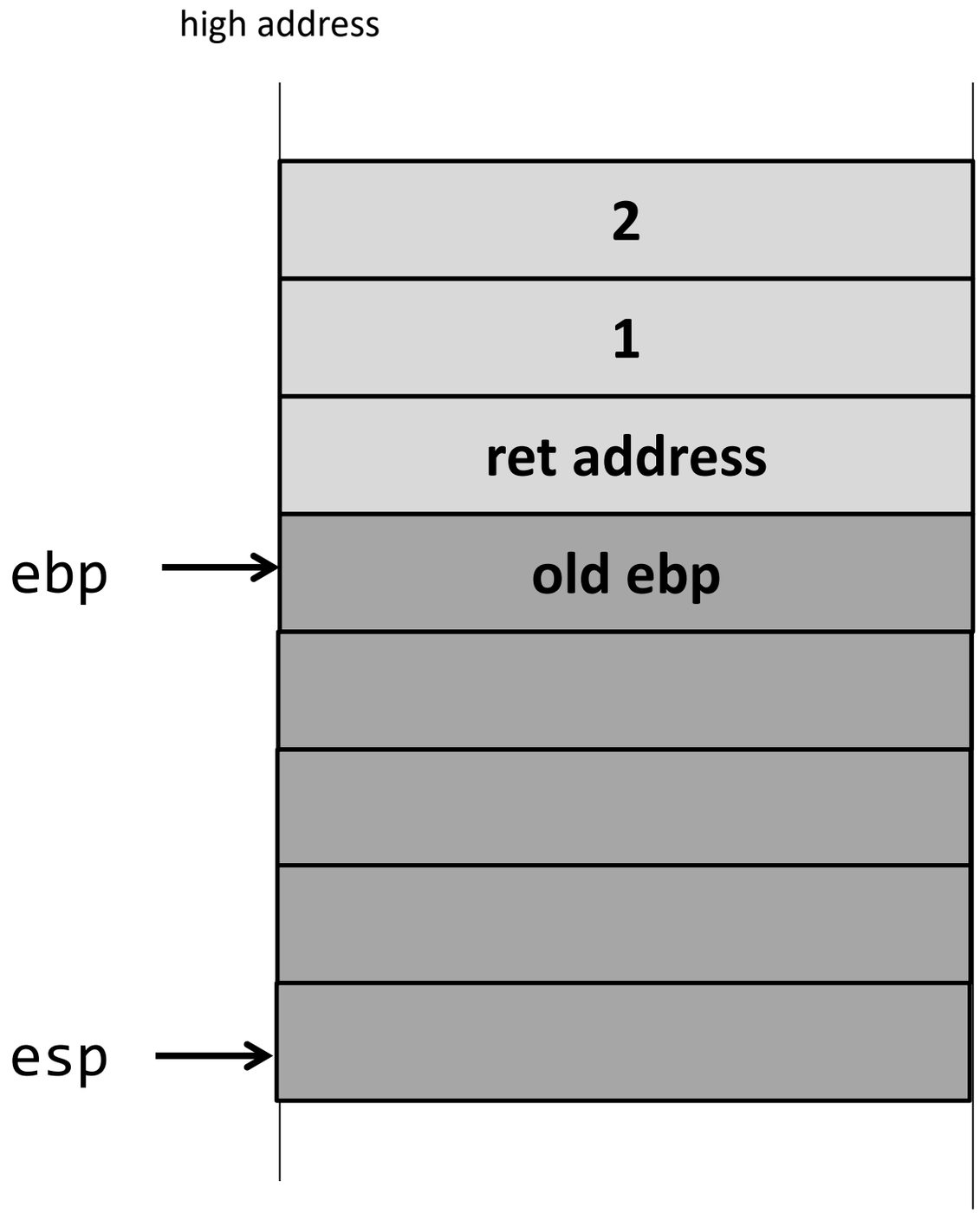


```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
call   534 <__x86.get_pc_thunk.ax>
add    $0x1ae4,%eax
mov    0x8(%ebp),%edx
mov    0xc(%ebp),%eax
add    %edx,%eax
mov    %eax,-0x8(%ebp)
mov    0x8(%ebp),%eax
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
nop
leave
ret
```



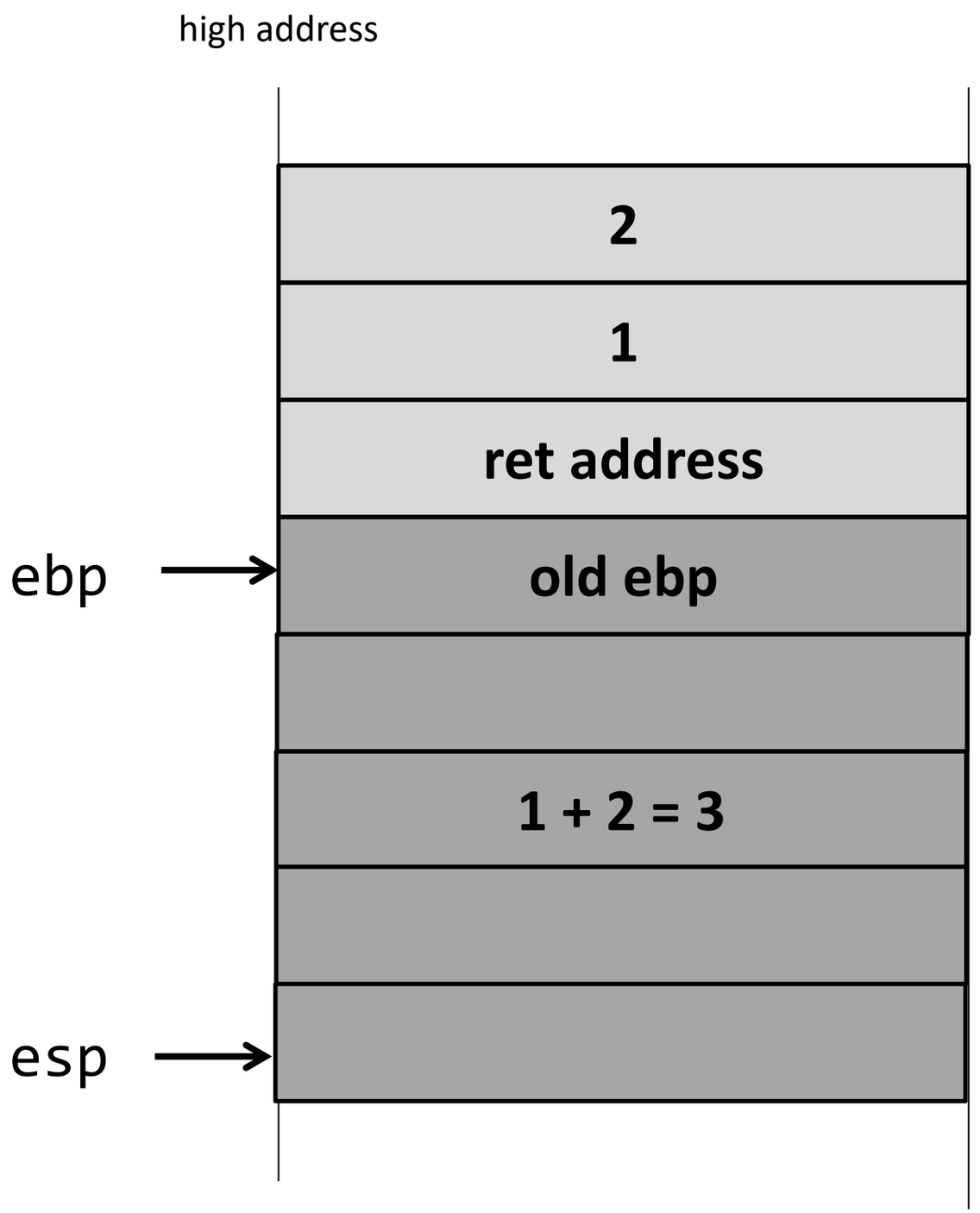
```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
call   534 <__x86.get_pc_thunk.ax>
add    $0x1ae4,%eax
mov    0x8(%ebp),%edx
mov    0xc(%ebp),%eax
add    %edx,%eax
mov    %eax,-0x8(%ebp)
mov    0x8(%ebp),%eax
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
nop
leave
ret
```

edx: 1
eax: 2

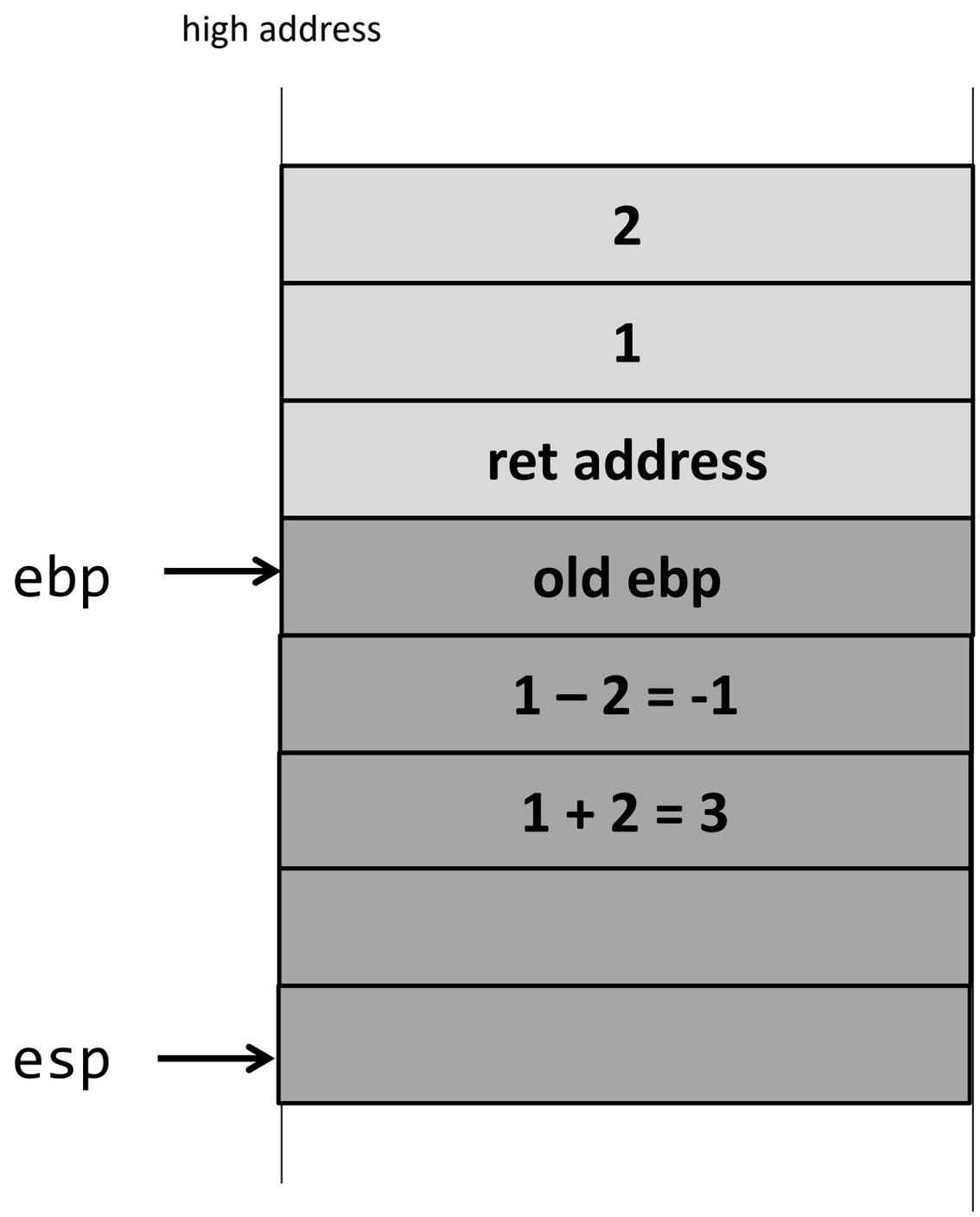


```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
call   534 <__x86.get_pc_thunk.ax>
add    $0x1ae4,%eax
mov    0x8(%ebp),%edx
mov    0xc(%ebp),%eax
add    %edx,%eax
mov    %eax,-0x8(%ebp)
mov    0x8(%ebp),%eax
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
nop
leave
ret
```

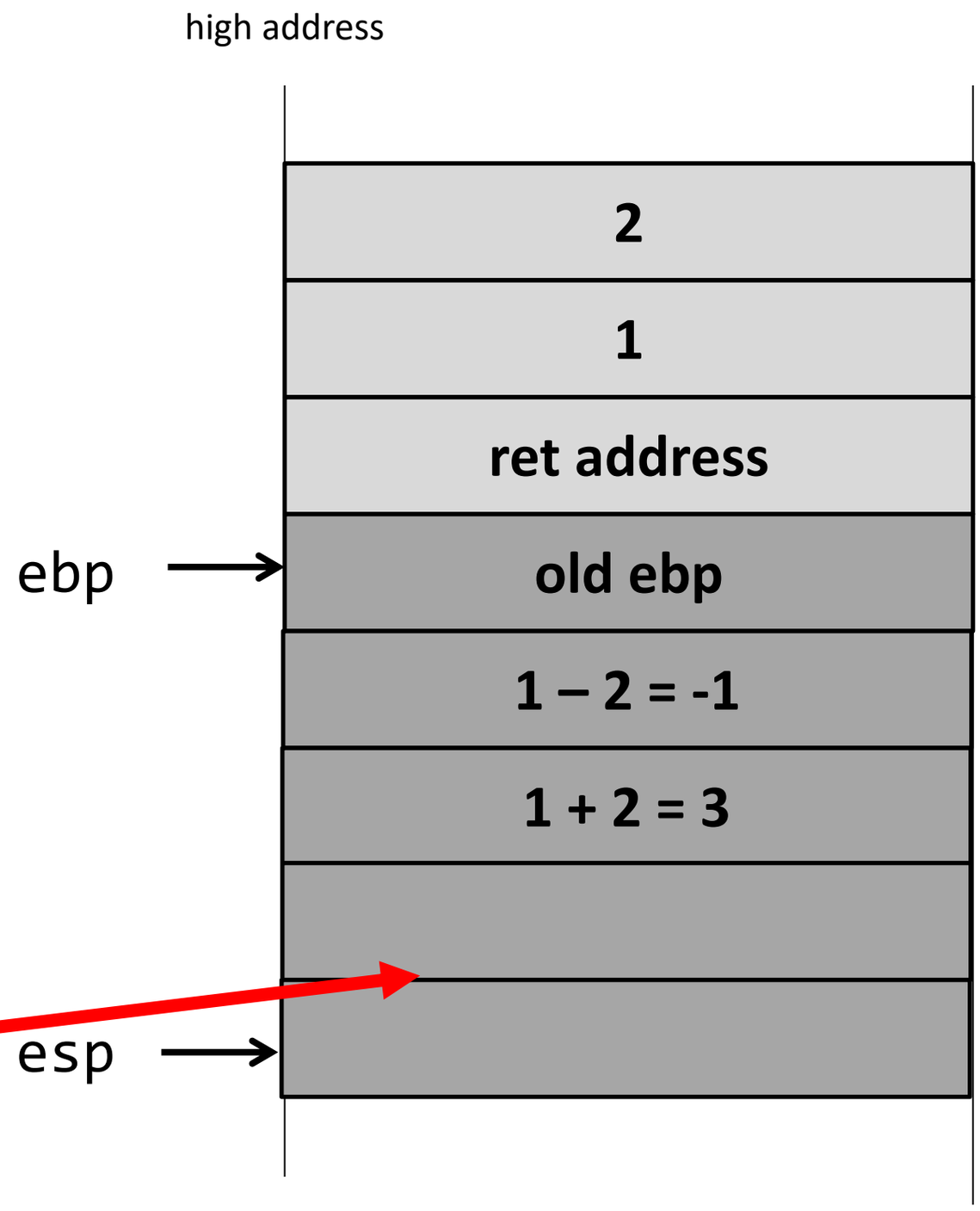
edx + eax = 3



```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
call   534 <__x86.get_pc_thunk.ax>
add    $0x1ae4,%eax
mov    0x8(%ebp),%edx
mov    0xc(%ebp),%eax
add    %edx,%eax
mov    %eax,-0x8(%ebp)
mov    0x8(%ebp),%eax
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
nop
leave
ret
```

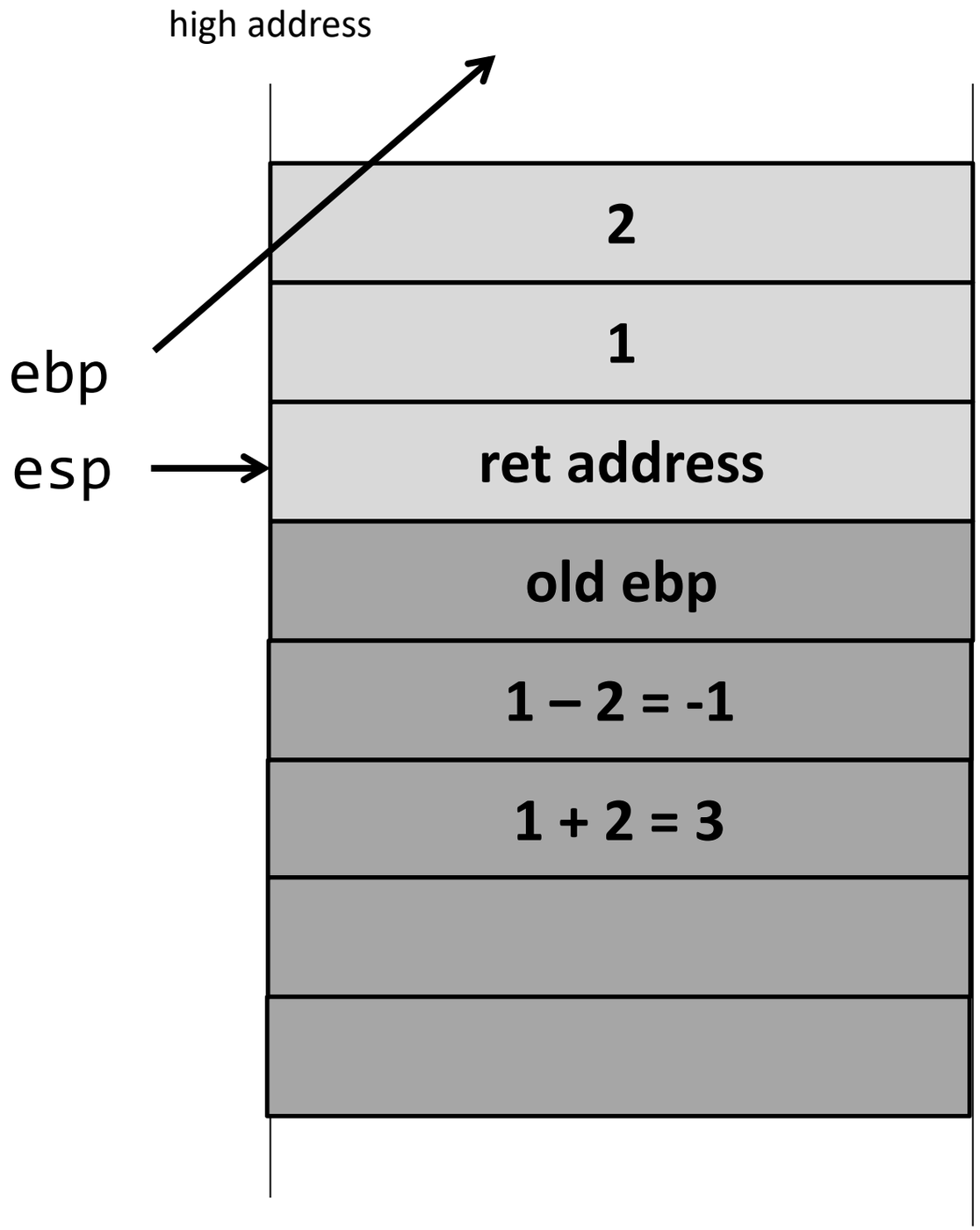


```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
call   534 <__x86.get_pc_thunk.ax>
add    $0x1ae4,%eax
mov    0x8(%ebp),%edx
mov    0xc(%ebp),%eax
add    %edx,%eax
mov    %eax,-0x8(%ebp)
mov    0x8(%ebp),%eax
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
nop
leave
ret
```



```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
call   534 <__x86.get_pc_thunk.ax>
add    $0x1ae4,%eax
mov    0x8(%ebp),%edx
mov    0xc(%ebp),%eax
add    %edx,%eax
mov    %eax,-0x8(%ebp)
mov    0x8(%ebp),%eax
sub    0xc(%ebp),%eax
mov    %eax,-0x4(%ebp)
nop
leave
ret
```

```
leave:
mov $ebp, $esp
pop $ebp
```



别人家的动画做的好一点，看一下别人家的

cool, what about x86_64

```
0000000000005fa <func>:
 5fa:   55                              push   %rbp
 5fb:   48 89 e5                        mov    %rsp,%rbp
 5fe:   89 7d ec                        mov    %edi,-0x14(%rbp)
 601:   89 75 e8                        mov    %esi,-0x18(%rbp)
 604:   8b 55 ec                        mov    -0x14(%rbp),%edx
 607:   8b 45 e8                        mov    -0x18(%rbp),%eax
 60a:   01 d0                           add    %edx,%eax
 60c:   89 45 f8                        mov    %eax,-0x8(%rbp)
 60f:   8b 45 ec                        mov    -0x14(%rbp),%eax
 612:   2b 45 e8                        sub    -0x18(%rbp),%eax
 615:   89 45 fc                        mov    %eax,-0x4(%rbp)
 618:   90                              nop
 619:   5d                              pop    %rbp
 61a:   c3                              retq
```

cool, what about arm32

```
000103c8 <func>:
   103c8:       e52db004        push    {fp}            ; (str fp, [sp, #-4]!)
   103cc:       e28db000        add     fp, sp, #0
   103d0:       e24dd014        sub     sp, sp, #20
   103d4:       e50b0010        str     r0, [fp, #-16]
   103d8:       e50b1014        str     r1, [fp, #-20]  ; 0xffffffec
   103dc:       e51b2010        ldr     r2, [fp, #-16]
   103e0:       e51b3014        ldr     r3, [fp, #-20]  ; 0xffffffec
   103e4:       e0823003        add     r3, r2, r3
   103e8:       e50b300c        str     r3, [fp, #-12]
   103ec:       e51b2010        ldr     r2, [fp, #-16]
   103f0:       e51b3014        ldr     r3, [fp, #-20]  ; 0xffffffec
   103f4:       e0423003        sub     r3, r2, r3
   103f8:       e50b3008        str     r3, [fp, #-8]
   103fc:       e1a00000        nop                     ; (mov r0, r0)
   10400:       e28bd000        add     sp, fp, #0
   10404:       e49db004        pop     {fp}            ; (ldr fp, [sp], #4)
   10408:       e12fff1e        bx      lr
```

cool, what about aarch64

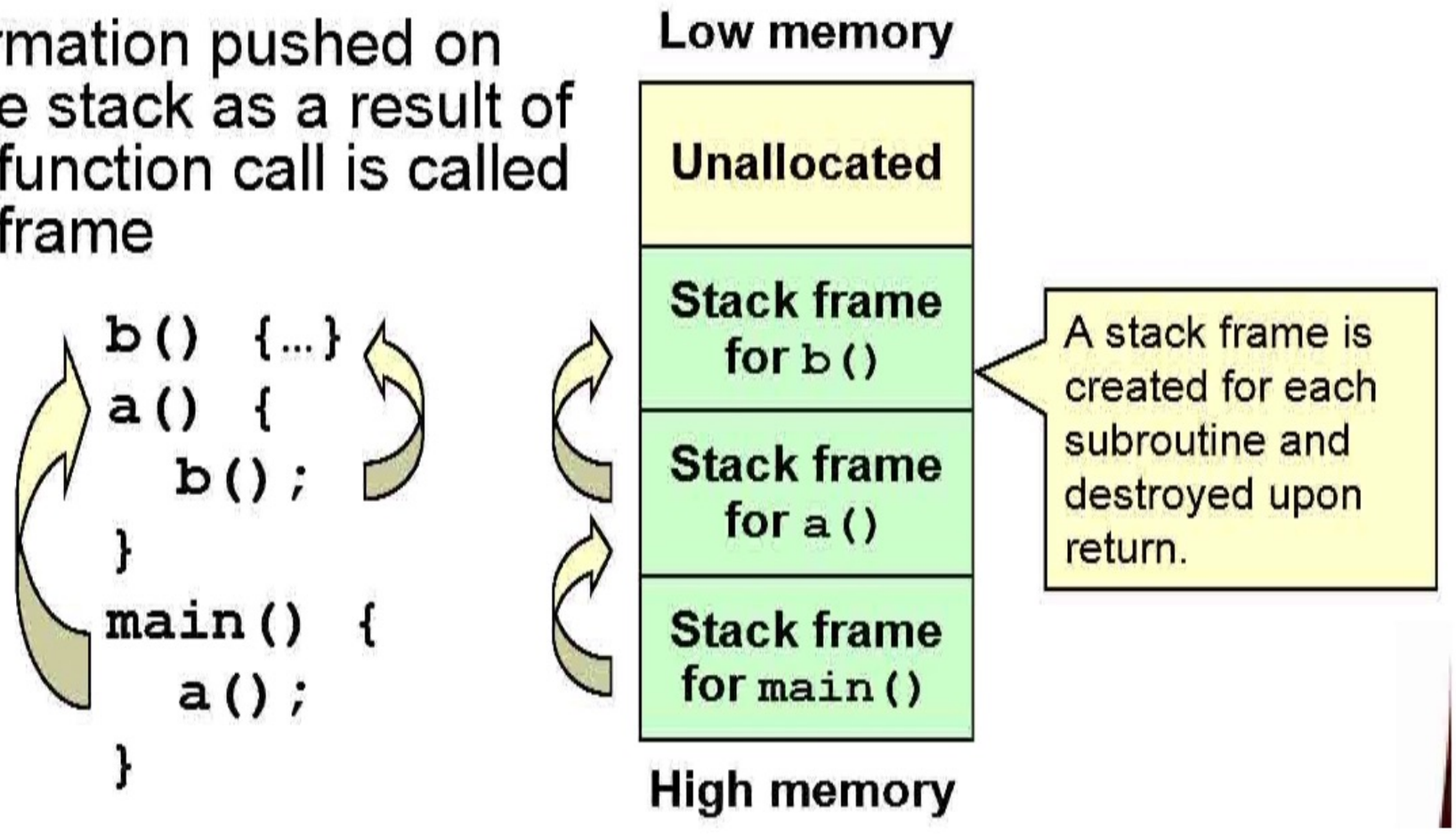
```
0000000000006e4 <func>:
 6e4:   d10083ff        sub     sp, sp, #0x20
 6e8:   b9000fe0        str     w0, [sp, #12]
 6ec:   b9000be1        str     w1, [sp, #8]
 6f0:   b9400fe1        ldr     w1, [sp, #12]
 6f4:   b9400be0        ldr     w0, [sp, #8]
 6f8:   0b000020        add     w0, w1, w0
 6fc:   b9001be0        str     w0, [sp, #24]
 700:   b9400fe1        ldr     w1, [sp, #12]
 704:   b9400be0        ldr     w0, [sp, #8]
 708:   4b000020        sub     w0, w1, w0
 70c:   b9001fe0        str     w0, [sp, #28]
 710:   d503201f        nop
 714:   910083ff        add     sp, sp, #0x20
 718:   d65f03c0        ret
```

emmmm, no frame pointer?

从 多个函数的角度来看，理解局部变量

The stack supports nested invocation calls

Information pushed on the stack as a result of a function call is called a frame



*Slide by Robert Seacord

stack buffer overflow **summary**

- stack grows from high memory to low memory

- stack pointer remove when push and pop
  - `esp (x86)`
  - `rsp (x64)`
  - `sp (arm32, aarch64)`

- frame pointer is used to pinpoint stack frame (activation record)
  - also called as base pointer
  - `ebp (x86)`
  - `rbp (x64)`
  - `fp (arm32, aarch64)`

stack buffer overflow summary

cdecl 调用约定：

- x86: 参数从“右”往“左”依次压栈，返回值放在eax
- x64:
  寄存器 rdi, rsi, rdx, rcx, r8, r9 会优先承担传参功能，返回值放 rax

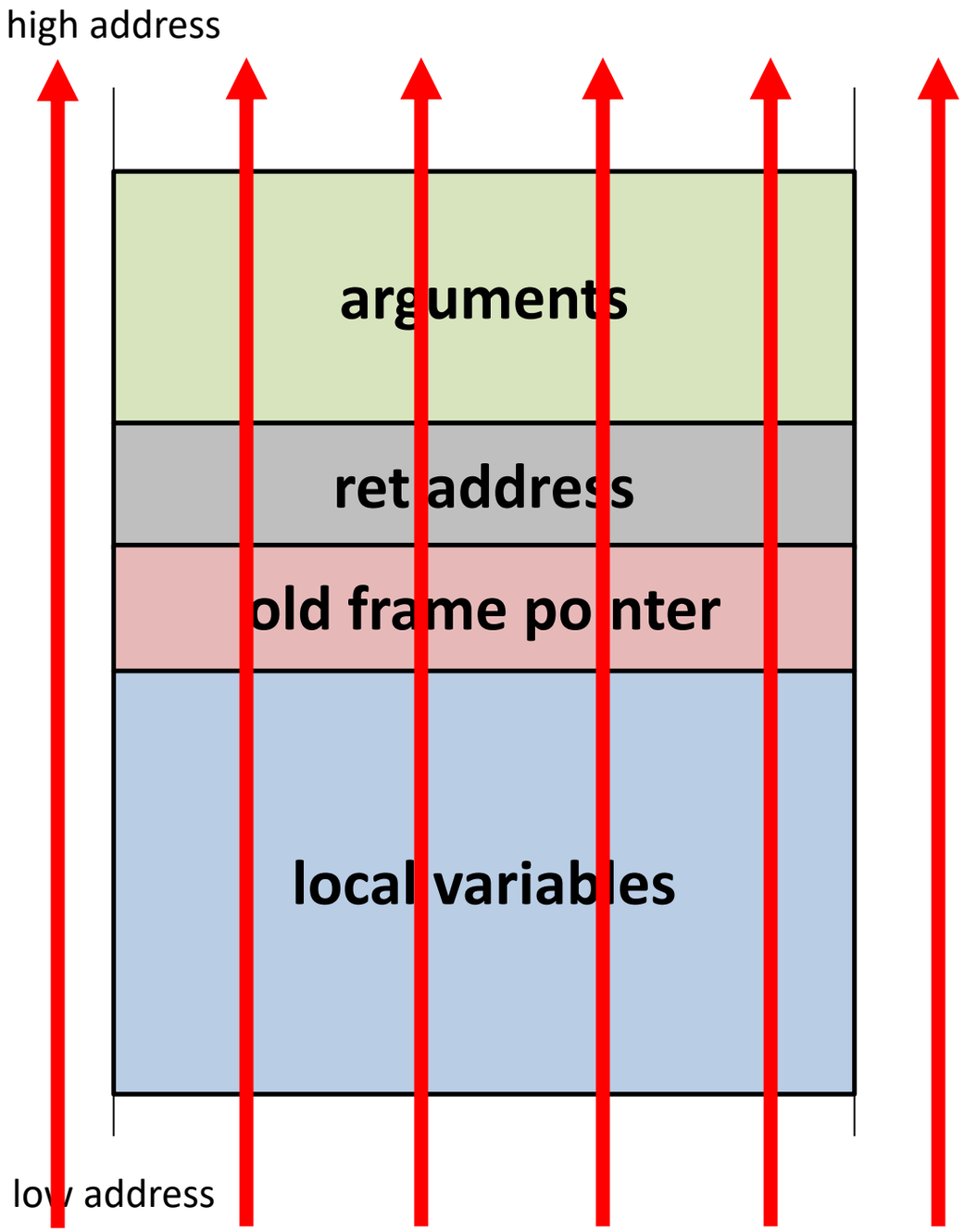
- arm:
  寄存器 r0 – r3 (又称 a1 – a4) 会优先承担传参功能，返回值放在 r0
- aarch64:
  寄存器 r0 – r7 优先承担传参功能，返回值放在r0

栈布局

Question: 不是有寄存器么，为啥还需要栈来放临时变量

或许该休息一下

stack **buffer overflow**



stack **buffer overflow**

赛题中常见的 stack overflow 情形

1. gets()

2. 整型符号问题

warning: never use `gets()`

```
GETS(3)                    Linux Programmer's Manual                    GETS(3)

NAME
       gets - get a string from standard input (DEPRECATED)

SYNOPSIS
       #include <stdio.h>

       char *gets(char *s);

DESCRIPTION
       Never use this function.

       gets()  reads a line from stdin into the buffer pointed to by s until either a
       terminating newline or EOF, which it replaces with a  null  byte  ('\0').   No
       check for buffer overrun is performed (see BUGS below).
```

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and **because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use.** It has been used to break computer security. Use fgets() instead.

符号问题的例子

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    char buffer[32];
    int inputsize;
    scanf("%d", &inputsize);
    if (inputsize >= 32) {
      printf("size too large\n");
      exit(1);
    }
    fgets(buffer, inputsize, stdin);
    puts(buffer);

    return 0;
}
```

stack **buffer overflow** 危害

首先，前文已知，栈上包含:

- 临时变量
- frame pointer
- return address
- arguments
  - *嗷，如果是64位，参数不一定在栈上哦*

然后，要分析目前的栈溢出漏洞**能力**

- *能溢出多少字节*
- *溢出部分的可控性*

如何交互

Time for pwntools: (demo.py)

1. 安装 python 和 pwntools
   ```
   sudo apt install python3 python3-pip
   pip3 install pwntools
   ```
2. python interactive 与 python script
3. 通过 process 和本地 binary 交互 (echo.c)
   1. 读
   2. 写
4. 通过 remote 和远端 binary 交互
   1. sbus calculator

overflow examples 之旅

1. 覆盖栈上临时变量
2. 覆盖栈上返回地址
    1. crash程序
    2. 控制流劫持
3. 覆盖 frame pointer

example 1: 覆盖其他临时变量改变程序逻辑

example 2: 覆盖返回地址 crash 整个程序

example 3: 覆盖返回地址完成控制流劫持

example 4: 覆盖frame pointer值完成“栈迁移”

stack **buffer overflow summary**

利用能力上

- 分析最大溢出长度
  - 分析溢出范围内的 targets
    - 分析overwrite这些targets的字节可控性

交互方式上

- 使用 pwntools

shellcode

想要实现更强破坏？
真的会有 backdoor 函数么？

however
攻击者可以通过 payload 向 target 的运行时中“注入”这样一个有着后门功能，用来 launch shell 的代码

即 ***shellcode***

shellcode example x86

```
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"              /* xorl    %eax,%eax      */
    "\x50"                  /* pushl   %eax           */
    "\x68""//sh"            /* pushl   $0x68732f2f    */
    "\x68""/bin"            /* pushl   $0x6e69622f    */
    "\x89\xe3"              /* movl    %esp,%ebx      */
    "\x50"                  /* pushl   %eax           */
    "\x53"                  /* pushl   %ebx           */
    "\x89\xe1"              /* movl    %esp,%ecx      */
    "\x99"                  /* cdq                    */
    "\xb0\x0b"              /* movb    $0x0b,%al      */
    "\xcd\x80"              /* int     $0x80          */
;
```

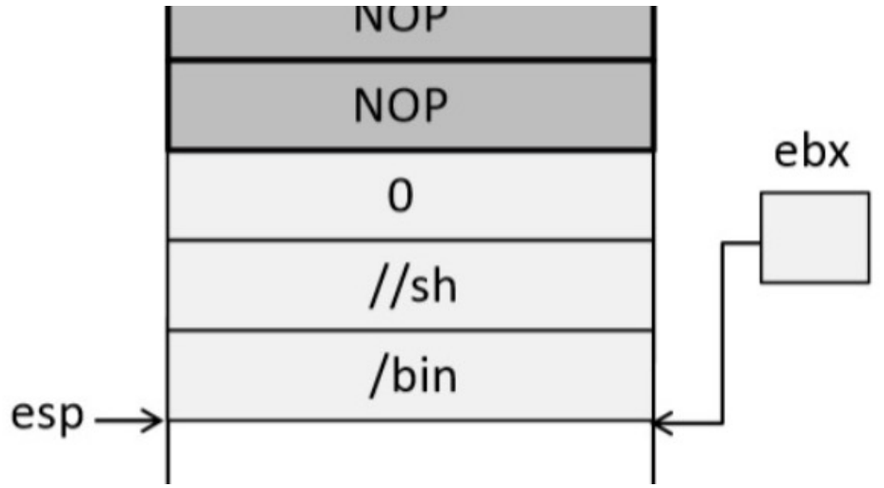
或者找 alternatives

- shell-storm: https://shell-storm.org/shellcode/
- pwntools: shell

line by line

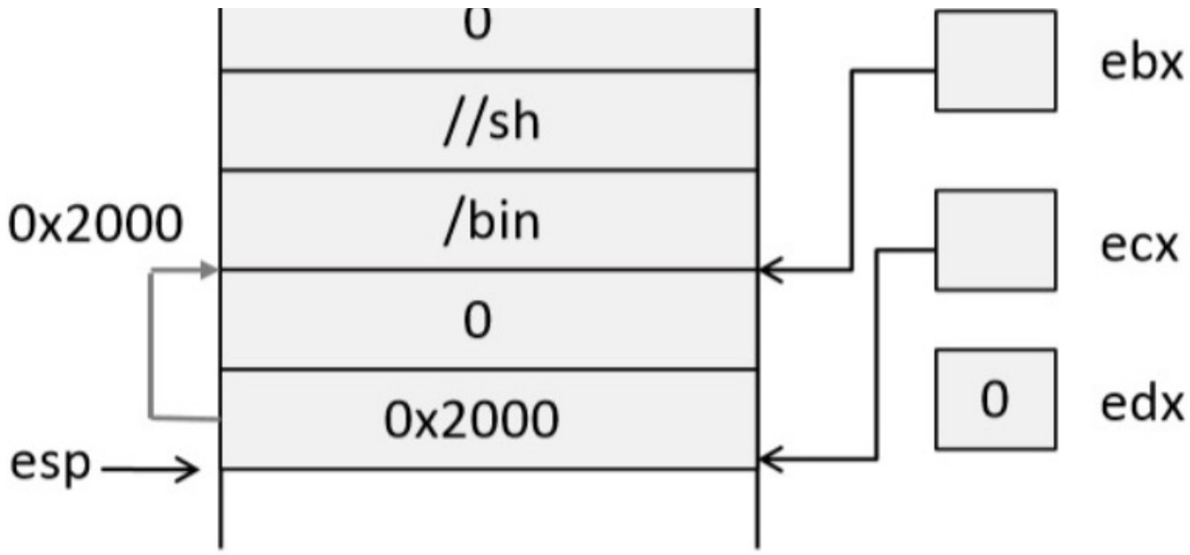
```
/* xorl    %eax,%eax      */
/* pushl   %eax           */
/* pushl   $0x68732f2f    */
/* pushl   $0x6e69622f    */
/* movl    %esp,%ebx      */
/* pushl   %eax           */
/* pushl   %ebx           */
/* movl    %esp,%ecx      */
/* cdq                    */
/* movb    $0x0b,%al      */
/* int     $0x80          */
```

0x68732f2f – “//sh”
0x6e69622f – “/bin”



line by line

```
/* xorl    %eax,%eax      */
/* pushl   %eax           */
/* pushl   $0x68732f2f    */
/* pushl   $0x6e69622f    */
/* movl    %esp,%ebx      */
/* pushl   %eax           */
/* pushl   %ebx           */
/* movl    %esp,%ecx      */
/* cdq                    */
/* movb    $0x0b,%al      */
/* int     $0x80          */
```



line by line

```
/* xorl    %eax,%eax      */
/* pushl   %eax           */
/* pushl   $0x68732f2f    */
/* pushl   $0x6e69622f    */
/* movl    %esp,%ebx      */
/* pushl   %eax           */
/* pushl   %ebx           */
/* movl    %esp,%ecx      */
/* cdq                    */
/* movb    $0x0b,%al      */
/* int     $0x80          */
```

```
#include <unistd.h>

int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

系统调用参数准备：

al：系统调用号（execve）

参数通过寄存器传递（因为系统调用严格限制参数个数，不用考虑变长参的情况）

1-th: ebx

2-th: ecx

3-th: edx

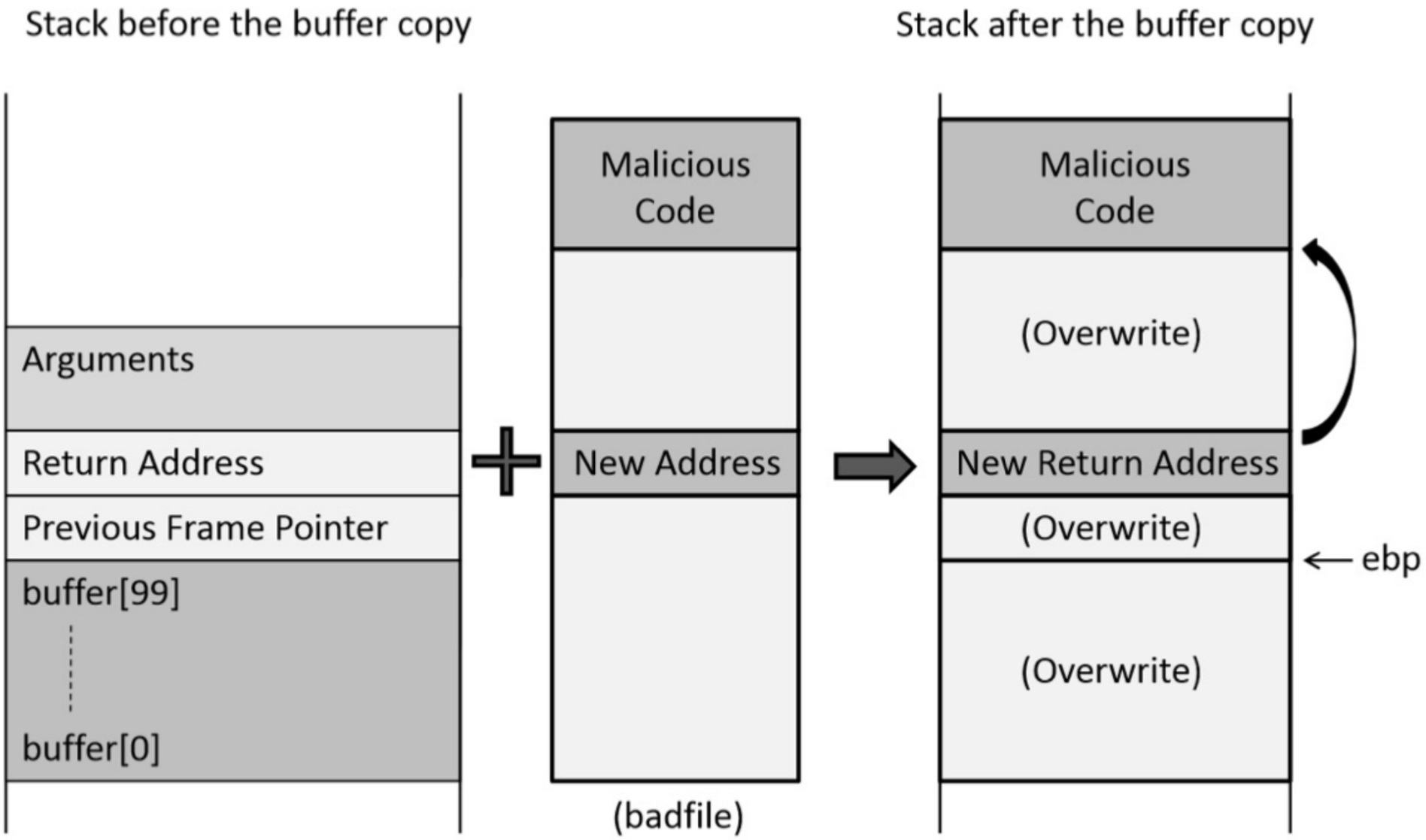
能讲到这里么

shellcode example x64

Will be in your homework ☺

shellcode + stack overflow

设想，如果在 overflow 的 payload 中加入 shellcode



example 5: 覆盖返回地址跳 shellcode

1. 确保栈可执行
2. 确保栈的随机化已经被 leak
3. 通过 pid 进行 trace 调试

stack overflow 的保护与绕过

首先

- never use gets()
- strcpy -> strncpy
- %s -> %Ns in scanf
- ……

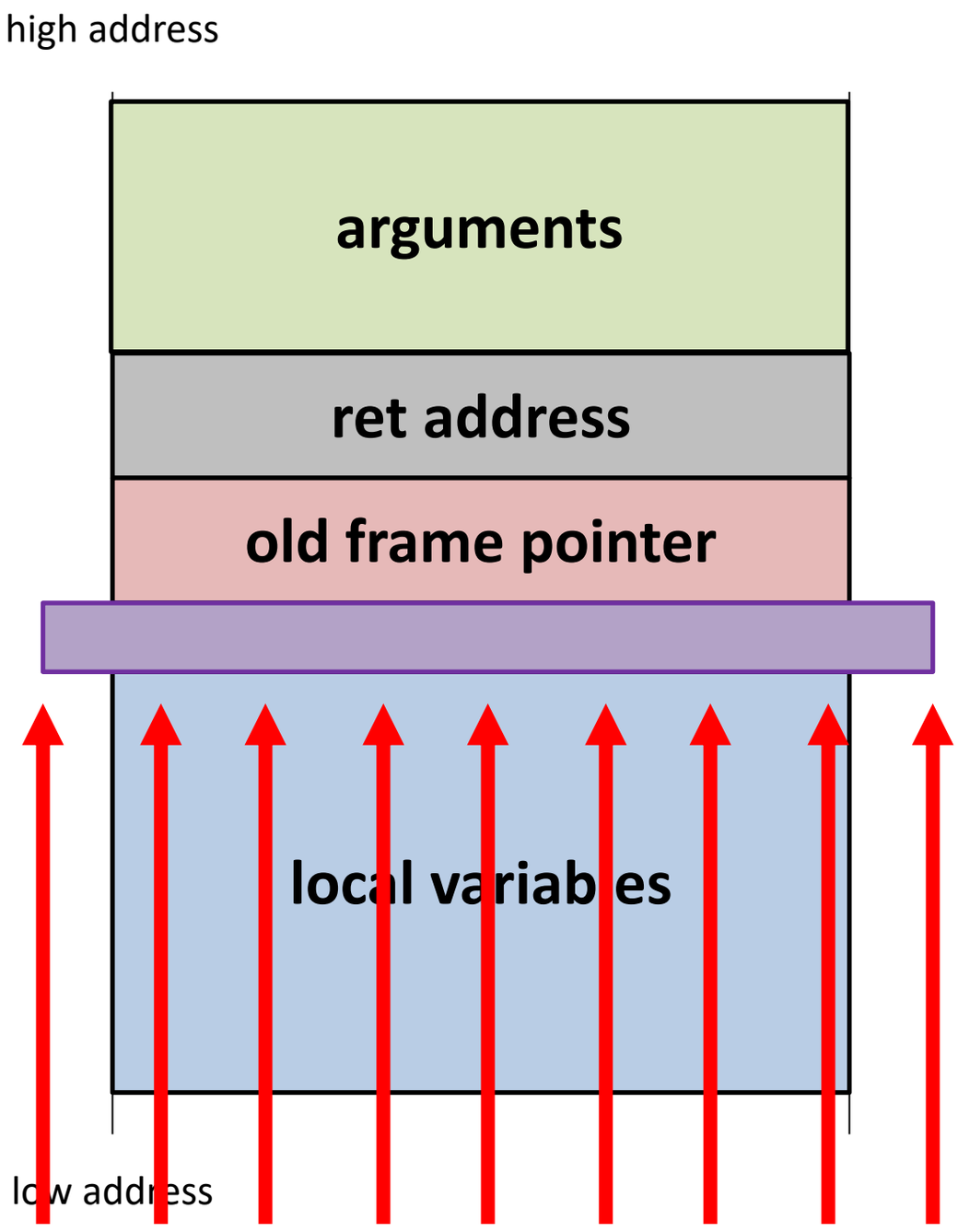
此外

- stack canary (stack cookie, stack guardian)
- DEP (Data Execution Prevention)
- ASLR (address space layout randomization)

stack canary

通过对随机数来完成对溢出进行检测！

demo 一下



还可以稍微调整一下栈的布局