

A detailed technical diagram of a telescope mechanism, likely from a historical document. The diagram shows a large circular structure with various components labeled in English. Labels include 'LOUVER', 'UPPER CURTAIN', 'UPPER POSITION OF MOUNT', 'SHUTTERS', 'TRACK', 'CABLES', 'LOWER CURTAIN', 'PARRY PLATFORM', 'SPECTROGRAPH BODY', 'ELEVATING PLATFORMS', 'OBSERVING FLOOR', 'STAIRS', 'TURNING CABLE GUARD', '30 FT. 3 IN. RADIUS OF BAIL', '62" TELESCOPE', and 'LOWER POSITION OF COUNTERWEIGHTS'. The diagram is rendered in a light gray, semi-transparent style, serving as a background for the text.

Computer System Security CS3312

# 计算机系统安全

2024年 春季学期

主讲教师：张媛媛 副教授

上海交通大学 计算机科学与技术系



# 第三章

## 软件安全：函数调用与栈溢出

Software Security: Function Call & Stack Overflow

# 目录/CONTENTS

---

## 01. 函数调用

Procedure call

## 02. 观察“栈”的运行

Look into the Stack

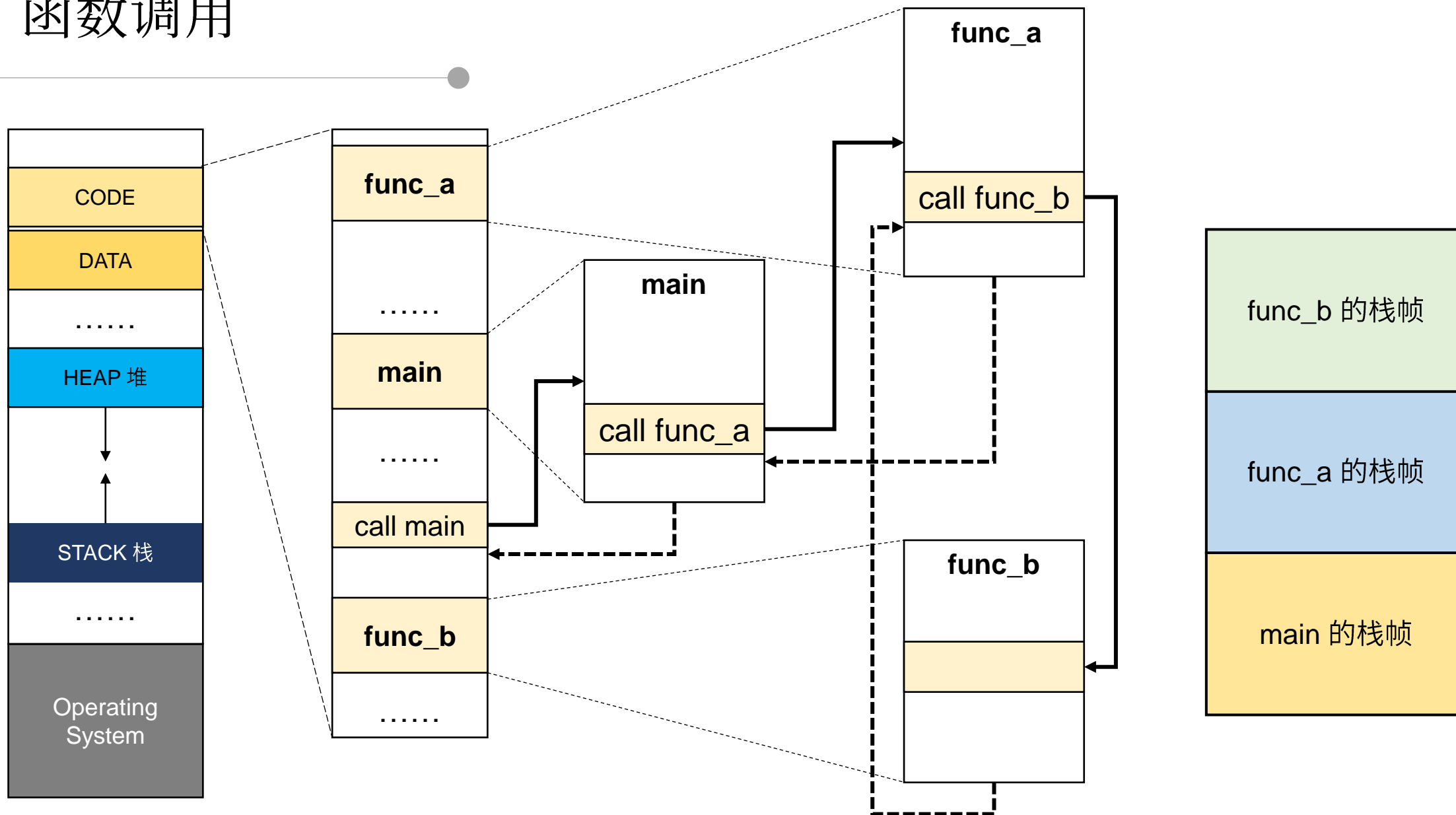
## 03. “栈”溢出

Stack overflow

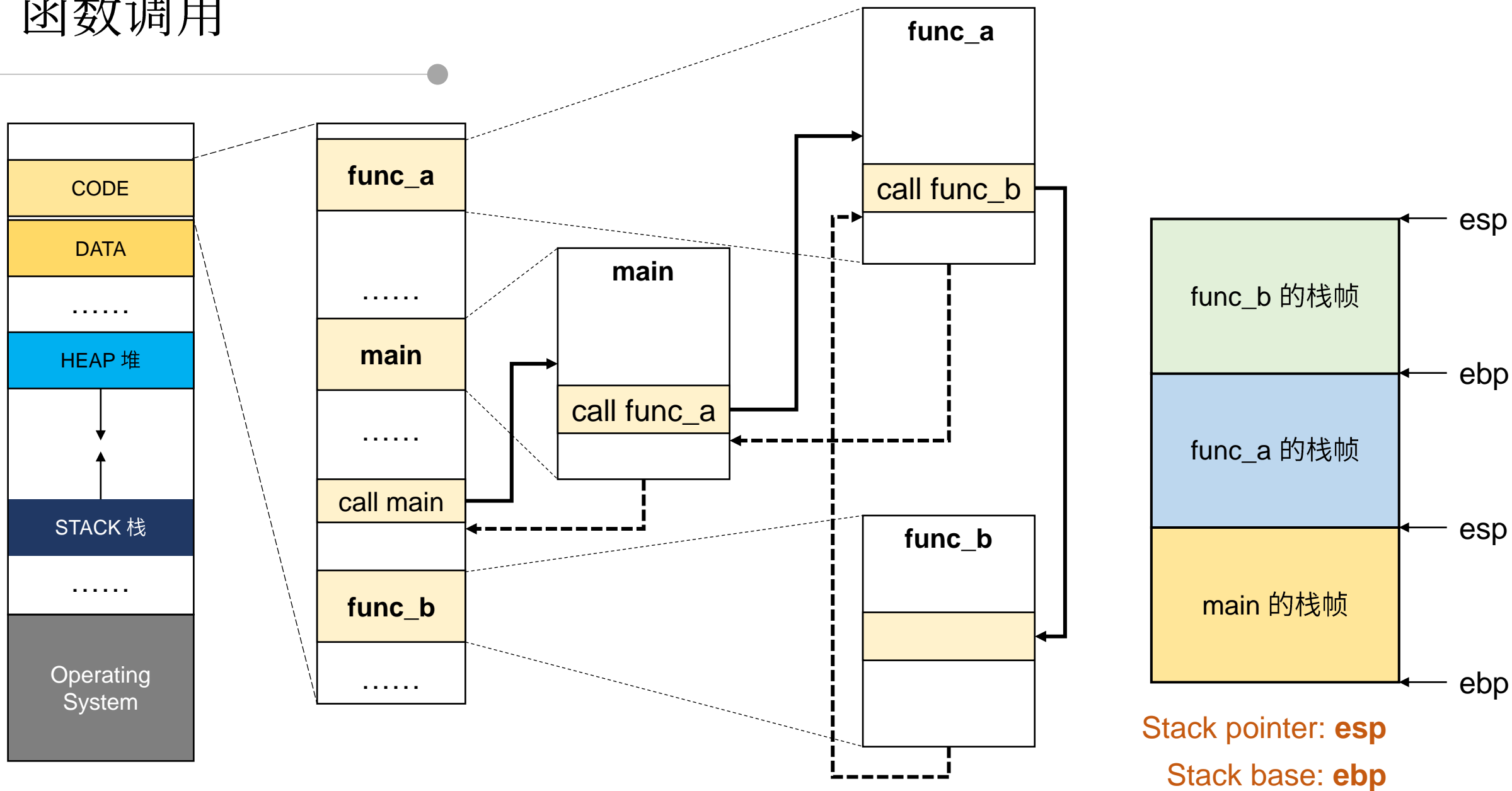
---



# 函数调用



# 函数调用



# 栈 Stack

栈：函数调用发生时，系统在内存空间为当前函数分配的临时存储空间

- 存储内容：
  - 本地变量
  - 函数的返回信息
  - 临时空间
- 栈空间的管理：
  - 当进入函数时被分配
  - 当返回调用函数时被释放

KernelMode - simplestack.exe - [\"G.P.U\" - main thread, module simplest]

Registers (FPU)

EAX 0019FFC0  
ECX 004A12A0 simplest.<ModuleEntryPoint>  
EDX 004A12A0 simplest.<ModuleEntryPoint>  
EBX 0035E000  
ESP 0019FF74  
EBP 0019FF80  
ESI 004A12A0 simplest.<ModuleEntryPoint>  
EDI 004A12A0 simplest.<ModuleEntryPoint>  
EIP 004A12A0 simplest.<ModuleEntryPoint>

C 0 ES 002B 32bit 0(FFFFFFFF)  
P 1 CS 0023 32bit 0(FFFFFFFF)  
A 0 SS 002B 32bit 0(FFFFFFFF)  
Z 1 DS 002B 32bit 0(FFFFFFFF)  
S 0 FS 0053 32bit 361000(FFF)  
T 0 GS 002B 32bit 0(FFFFFFFF)  
D 0  
O 0 LastErrr ERROR\_SUCCESS (00000000)  
EFL 00000246 (NO,NO,E,BE,MS,PE,GE,LE)  
ST0 empty 0.0  
ST1 empty 0.0  
ST2 empty 0.0  
ST3 empty 0.0  
ST4 empty 0.0

Address Value Comment  
004A8010 773F4D90 ntdll.RtlDecodePointer  
004A8014 75CB5C40 kernel132.UnhandledExceptionFilter  
004A8018 75CA1720 kernel132.SetUnhandledExceptionFilter  
004A801C 75CA2000 jmp to KernelBa.IsDebuggerPresent  
004A8020 773F5550 ntdll.RtlEncodePointer  
004A8024 75C99910 kernel132.TerminateProcess  
004A8028 75CA2E80 jmp to KernelBa.GetCurrentProcess  
004A802C 75C9F550 kernel132.GetProcAddress  
004A8030 75CA0E50 kernel132.GetModuleHandleW  
004A8034 75CA4E10 kernel132.ExitProcess  
004A8038 75CA3500 jmp to KernelBa.WriteFile  
004A803C 75CA1700 kernel132.GetStdHandle  
004A8040 75CA0900 kernel132.GetModuleFileNameW

Memory Window Start: 0x4A8010 End: 0x4A8013 Size: 0x4 Value: 0x773F4D90

此处展示栈的内容

# 函数调用案例

```
//simplestack.c
#include <stdio.h>

int add(int a, int b)
{
    int s;
    s = a + b;
    return s;
}

int main()
{
    int a=2, b=4, sum=0;
    sum = add(a, b);
    printf("%d\n", sum);
    return sum;
}
```

(Microsoft Visual Studio的C编译链接命令)

`> cl simplestack.c` 得到可执行文件 simplestack.exe



为了观测进程中的函数调用的过程，  
我们需要使用动态调试工具：

- Windows: OllyDbg、x64dbg
- Linux: GDB



# OllyDbg

The screenshot displays the OllyDbg interface with the following components:

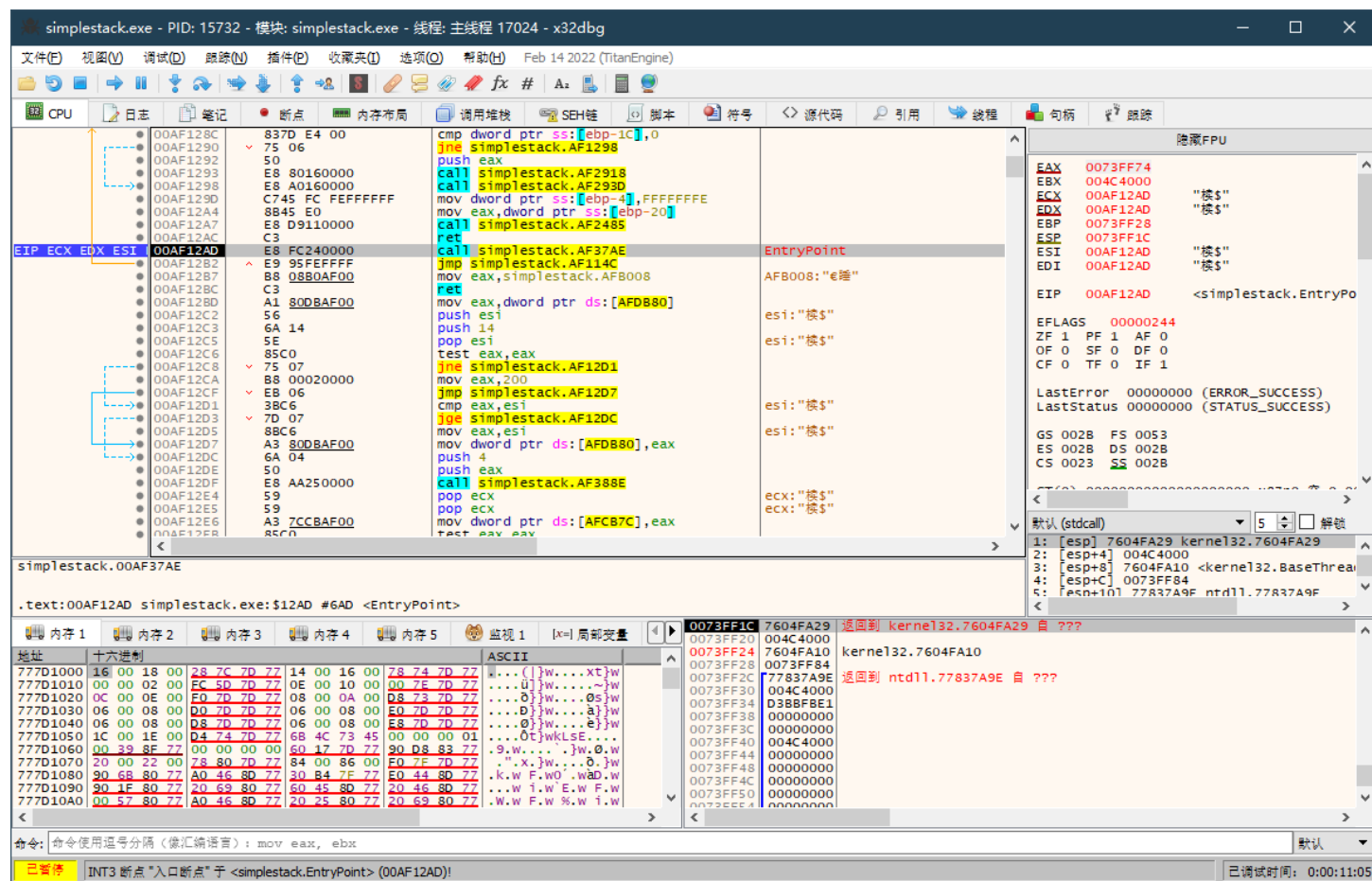
- Assembly Window:** Shows assembly code for `simplest.exe`. The current instruction is `call simplest.004A37AE` at address `004A12AD`. Other instructions include `jmp simplest.004A114C`, `mov eax, simplest.004A0808`, `ret`, `mov eax, dword ptr ds:[0x4AD880]`, `push esi`, `push 0x14`, `pop esi`, `test eax, eax`, `inc short simplest.004A12D1`, `mov eax, 0x200`, `jmp short simplest.004A12D7`, `cmp eax, esi`, `jge short simplest.004A12DC`, `mov eax, esi`, `mov dword ptr ds:[0x4AD880], eax`, `push 0x4`, `push eax`, `call simplest.004A388E`, `pop ecx`, `pop ecx`, `mov dword ptr ds:[0x4ACB7C], eax`, `test eax, eax`, and `inc short simplest.004A1300`.
- Registers (FPU) Window:** Shows the state of CPU registers. `EAX` is `0019FFCC`, `ECX` is `004A12AD`, `EDX` is `004A12AD`, `EBX` is `0035E000`, `ESP` is `0019FF74`, `EBP` is `0019FF80`, `ESI` is `004A12AD`, `EDI` is `004A12AD`, and `EIP` is `004A12AD`. The `LastError` is `ERROR_SUCCESS (00000000)`.
- Memory Window:** Shows a list of memory addresses and their values. The current address is `004A8010` with value `773F4090`. The window is titled `Memory/Window 1 Start: 0x4A8010 End: 0x4A8013 Size: 0x4 Value: 0x773F4090`.

OllyDbg是一款Windows32位平台(only)反汇编动态调试追踪工具，只支持Ring3级别的用户态程序。目前已知最新版本为2013年的2.0.1。由俄国程序员 Oleh Yuschuk (Ollys) 2000年首次发布的共享软件。

- 打开一个新的可执行程序 (F3)
- 重新运行当前调试的程序 (Ctrl+F2)
- 当前调试的程序 (Alt+F2)
- 运行选定的程序进行调试 (F9)
- 暂时停止被调试程序的执行 (F12)
- 单步进入被调试程序的 Call 中 (F7)
- 步过被调试程序的 Call (F8)
- 跟入被调试程序的 Call 中 (Ctrl+F11)
- 跟踪时跳过被调试程序的 Call (Ctrl+F12)
- 执行直到返回 (Ctrl+F9)
- 显示记录窗口 (Alt+L)
- 显示模块窗口 (Alt+E)
- 显示内存窗口 (Alt+M)
- 显示 CPU 窗口 (Alt+C)
- 显示补丁窗口 (Ctrl+P)
- 显示呼叫堆栈 (Alt+K)



## x64dbg



x64dbg是一个开源的面向 Windows32 和 Windows64 的动态调试器。目前由 x64dbg社区维护更新。

**Debugger core by TitanEngine Community Edition**

**Disassembly powered by Zydis**

**Assembly powered by XEDParse and asmjit**

**Import reconstruction powered by Scylla**

**JSON powered by Jansson**

**Database compression powered by lz4**

**Advanced pattern matching powered by yara**

**Decompilation powered by snowman**

**Bug icon by VisualPharm**

**Interface icons by Fugue**

**Website by tr4ceflow**

# GDB

```

student@IS308: ~/Documents
student@IS308:~/Documents$ gdb gdbstep
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from gdbstep...done.
gdb-peda$ list
7      }
8
9      int count(int num) {
10         int i = 0;
11         if(0 > num)
12             return 0;
13         while(i < num) {
14             printf("%d\n", i);
15             i++;
16         }
gdb-peda$

```

```

/*gdbStep.c*/
#include<stdio.h>

int add(int a, int b) {
    int c = a + b;
    return c;
}

int count(int num)
{
    int i = 0;
    if(0 > num)
        return 0;
    while(i < num) {
        printf("%d\n", i);
        i++;
    }
    return i;
}

```

```

int main(void) {
    int a = 3;
    int b = 7;
    printf("it will calc a + b\n");
    int c = add(a,b);
    printf("%d + %d = %d\n", a,b,c);
    count(c);
    return 0;
}

```

```

Linux> gcc -g -o gdbstep gdbstep.c
Linux> gdb gdbstep

```

GDB 全称“GNU symbolic debugger”，诞生于 GNU 计划（同时诞生的还有 GCC、Emacs 等），是 Linux 下常用的程序调试器。实际场景中，GDB 最常用来调试 C 和 C++ 程序。

# 栈 Stack

栈：函数调用发生时，系统在内存空间为当前函数分配的临时存储空间

- 存储内容：
  - 本地变量
  - 函数的返回信息
  - 临时空间
- 栈空间的管理：
  - 当进入函数时被分配
  - 当返回调用函数时被释放
- 实现方式：
  - 全局唯一的栈顶指针（寄存器esp）
    - 指向当前运行函数栈的顶部
  - 全局唯一的栈底指针（寄存器ebp）
    - 指向栈的底部

KernelMode - simplestack.exe - [\"G.P.U\" - main thread, module simplest]

Registers (FPU)

EAX	0019FFC0
ECX	004A12A0 simplest.<ModuleEntryPoint>
EDX	004A12A0 simplest.<ModuleEntryPoint>
EBX	0035E000
ESP	0019FF74
EBP	0019FF80
ESI	004A12A0 simplest.<ModuleEntryPoint>
EDI	004A12A0 simplest.<ModuleEntryPoint>
EIP	004A12A0 simplest.<ModuleEntryPoint>
C 0	ES 002B 32bit 0(FFFFFFFF)
P 1	CS 0023 32bit 0(FFFFFFFF)
A 0	SS 002B 32bit 0(FFFFFFFF)
Z 1	DS 002B 32bit 0(FFFFFFFF)
S 0	FS 0053 32bit 361000(FFF)
T 0	GS 002B 32bit 0(FFFFFFFF)
D 0	
O 0	LastErrr ERROR_SUCCESS (00000000)
EFL	00000246 (NO,NO,E,BE,MS,PE,GE,LE)
ST0	empty 0.0
ST1	empty 0.0
ST2	empty 0.0
ST3	empty 0.0
ST4	empty 0.0
ST5	empty 0.0

Address Value Comment

004A8010	773F4D90	ntdll.RtlDecodePointer
004A8014	75C85C40	kernel32.UnhandledExceptionFilter
004A8018	75C81720	kernel32.SetUnhandledExceptionFilter
004A801C	75C82000	jmp to KernelBa.IsDebuggerPresent
004A8020	773F5750	ntdll.RtlEncodePointer
004A8024	75C89910	kernel32.TerminateProcess
004A8028	75C82E80	jmp to KernelBa.GetCurrentProcess
004A802C	75C9F550	kernel32.GetProcAddress
004A8030	75C8A050	kernel32.GetModuleHandleW
004A8034	75C84E10	kernel32.ExitProcess
004A8038	75C83500	jmp to KernelBa.WriteFile
004A803C	75C81700	kernel32.GetStdHandle
004A8040	75C80900	kernel32.GetModuleFileNameW

Memory Window 1 Start: 0x4A8010 End: 0x4A8013 Size: 0x7F3F4D90

0019FF74 75C9FA29 RETURN to kernel32.75C9FA29

0019FF78 0035E000

0019FF7C 75C9FA10 kernel32.BaseThreadInitThunk

0019FF80 0019FF80

0019FF84 773F5750

0019FF88 0035E000

0019FF8C 75C81720

0019FF90 00000000

0019FF94 00000000

0019FF98 0035E000

0019FF9C 00000000

0019FFA0 00000000

0019FFA4 00000000

0019FFA8 00000000

0019FFAC 00000000

0019FFB0 00000000

0019FFB4 00000000

0019FFB8 00000000

0019FFBC 00000000

0019FFC0 00000000

0019FFC4 00000000

0019FFC8 00000000

0019FFCC 00000000

0019FFD0 00000000

0019FFD4 00000000

0019FFD8 00000000

0019FFDC 00000000

0019FFE0 00000000

0019FFE4 00000000

0019FFE8 00000000

0019FFEC 00000000

0019FFF0 00000000

0019FFF4 00000000

0019FFF8 00000000

0019FFFC 00000000

0019FF00 00000000

0019FF04 00000000

0019FF08 00000000

0019FF0C 00000000

0019FF10 00000000

0019FF14 00000000

0019FF18 00000000

0019FF1C 00000000

0019FF20 00000000

0019FF24 00000000

0019FF28 00000000

0019FF2C 00000000

0019FF30 00000000

0019FF34 00000000

0019FF38 00000000

0019FF3C 00000000

0019FF40 00000000

0019FF44 00000000

0019FF48 00000000

0019FF4C 00000000

0019FF50 00000000

0019FF54 00000000

0019FF58 00000000

0019FF5C 00000000

0019FF60 00000000

0019FF64 00000000

0019FF68 00000000

0019FF6C 00000000

0019FF70 00000000

0019FF74 00000000

0019FF78 00000000

0019FF7C 00000000

0019FF80 00000000

0019FF84 00000000

0019FF88 00000000

0019FF8C 00000000

0019FF90 00000000

0019FF94 00000000

0019FF98 00000000

0019FF9C 00000000

0019FFA0 00000000

0019FFA4 00000000

0019FFA8 00000000

0019FFAC 00000000

0019FFB0 00000000

0019FFB4 00000000

0019FFB8 00000000

0019FFBC 00000000

0019FFC0 00000000

0019FFC4 00000000

0019FFC8 00000000

0019FFCC 00000000

0019FFD0 00000000

0019FFD4 00000000

0019FFD8 00000000

0019FFDC 00000000

0019FFE0 00000000

0019FFE4 00000000

0019FFE8 00000000

0019FFEC 00000000

0019FFF0 00000000

0019FFF4 00000000

0019FFF8 00000000

0019FFFC 00000000

0019FF00 00000000

0019FF04 00000000

0019FF08 00000000

0019FF0C 00000000

0019FF10 00000000

0019FF14 00000000

0019FF18 00000000

0019FF1C 00000000

0019FF20 00000000

0019FF24 00000000

0019FF28 00000000

0019FF2C 00000000

0019FF30 00000000

0019FF34 00000000

0019FF38 00000000

0019FF3C 00000000

0019FF40 00000000

0019FF44 00000000

0019FF48 00000000

0019FF4C 00000000

0019FF50 00000000

0019FF54 00000000

0019FF58 00000000

0019FF5C 00000000

0019FF60 00000000

0019FF64 00000000

0019FF68 00000000

0019FF6C 00000000

0019FF70 00000000

0019FF74 00000000

0019FF78 00000000

0019FF7C 00000000

0019FF80 00000000

0019FF84 00000000

0019FF88 00000000

0019FF8C 00000000

0019FF90 00000000

0019FF94 00000000

0019FF98 00000000

0019FF9C 00000000

0019FFA0 00000000

0019FFA4 00000000

0019FFA8 00000000

0019FFAC 00000000

0019FFB0 00000000

0019FFB4 00000000

0019FFB8 00000000

0019FFBC 00000000

0019FFC0 00000000

0019FFC4 00000000

0019FFC8 00000000

0019FFCC 00000000

0019FFD0 00000000

0019FFD4 00000000

0019FFD8 00000000

0019FFDC 00000000

0019FFE0 00000000

0019FFE4 00000000

0019FFE8 00000000

0019FFEC 00000000

0019FFF0 00000000

0019FFF4 00000000

0019FFF8 00000000

0019FFFC 00000000

0019FF00 00000000

0019FF04 00000000

0019FF08 00000000

0019FF0C 00000000

0019FF10 00000000

0019FF14 00000000

0019FF18 00000000

0019FF1C 00000000

0019FF20 00000000

0019FF24 00000000

0019FF28 00000000

0019FF2C 00000000

0019FF30 00000000

0019FF34 00000000

0019FF38 00000000

0019FF3C 00000000

0019FF40 00000000

0019FF44 00000000

0019FF48 00000000

0019FF4C 00000000

0019FF50 00000000

0019FF54 00000000

0019FF58 00000000

0019FF5C 00000000

0019FF60 00000000

0019FF64 00000000

0019FF68 00000000

0019FF6C 00000000

0019FF70 00000000

0019FF74 00000000

0019FF78 00000000

0019FF7C 00000000

0019FF80 00000000

0019FF84 00000000

0019FF88 00000000

0019FF8C 00000000

0019FF90 00000000

0019FF94 00000000

0019FF98 00000000

0019FF9C 00000000

0019FFA0 00000000

0019FFA4 00000000

0019FFA8 00000000

0019FFAC 00000000

0019FFB0 00000000

0019FFB4 00000000

0019FFB8 00000000

0019FFBC 00000000

0019FFC0 00000000

0019FFC4 00000000

0019FFC8 00000000

0019FFCC 00000000

0019FFD0 00000000

0019FFD4 00000000

0019FFD8 00000000

0019FFDC 00000000

0019FFE0 00000000

0019FFE4 00000000

0019FFE8 00000000

0019FFEC 00000000

0019FFF0 00000000

0019FFF4 00000000

0019FFF8 00000000

0019FFFC 00000000

0019FF00 00000000

0019FF04 00000000

0019FF08 00000000

0019FF0C 00000000

0019FF10 00000000

0019FF14 00000000

0019FF18 00000000

0019FF1C 00000000

0019FF20 00000000

0019FF24 00000000

0019FF28 00000000

0019FF2C 00000000

0019FF30 00000000

0019FF34 00000000

0019FF38 00000000

0019FF3C 00000000

0019FF40 00000000

0019FF44 00000000

0019FF48 00000000

0019FF4C 00000000

0019FF50 00000000

0019FF54 00000000

0019FF58 00000000

0019FF5C 00000000

0019FF60 00000000

0019FF64 00000000

0019FF68 00000000

0019FF6C 00000000

0019FF70 00000000

0019FF74 00000000

0019FF78 00000000

0019FF7C 00000000

0019FF80 00000000

0019FF84 00000000

0019FF88 00000000

0019FF8C 00000000

0019FF90 00000000

0019FF94 00000000

0019FF98 00000000

0019FF9C 00000000

0019FFA0 00000000

0019FFA4 00000000

0019FFA8 00000000

0019FFAC 00000000

0019FFB0 00000000

0019FFB4 00000000

0019FFB8 00000000

0019FFBC 00000000

0019FFC0 00000000

0019FFC4 00000000

0019FFC8 00000000

0019FFCC 00000000

0019FFD0 00000000

0019FFD4 00000000

0019FFD8 00000000

0019FFDC 00000000

0019FFE0 00000000

0019FFE4 00000000

0019FFE8 00000000

0019FFEC 00000000

0019FFF0 00000000

0019FFF4 00000000

0019FFF8 00000000

0019FFFC 00000000

0019FF00 00000000

0019FF04 00000000

0019FF08 00000000

0019FF0C 00000000

0019FF10 00000000

0019FF14 00000000

0019FF18 00000000

0019FF1C 00000000

0019FF20 00000000

0019FF24 00000000

0019FF28 00000000

0019FF2C 00000000

0019FF30 00000000

0019FF34 00000000

0019FF38 00000000

0019FF3C 00000000

0019FF40 00000000

0019FF44 00000000

0019FF48 00000000

0019FF4C 00000000

0019FF50 00000000

0019FF54 00000000

0019FF58 00000000

0019FF5C 00000000

0019FF60 00000000

0019FF64 00000000

0019FF68 00000000

0019FF6C 00000000

0019FF70 00000000

0019FF74 00000000

0019FF78 00000000

0019FF7C 00000000

0019FF80 00000000

0019FF84 00000000

0019FF88 00000000

0019FF8C 00000000

0019FF90 00000000

0019FF94 00000000

0019FF98 00000000

0019FF9C 00000000

0019FFA0 00000000

0019FFA4 00000000

0019FFA8 00000000

0019FFAC 00000000

0019FFB0 00000000

0019FFB4 00000000

0019FFB8 00000000

0019FFBC 00000000

0019FFC0 00000000

0019FFC4 00000000

0019FFC8 00000000

0019FFCC 00000000

0019FFD0 00000000

0019FFD4 00000000

0019FFD8 00000000

0019FFDC 00000000

0019FFE0 00000000

0019FFE4 00000000

0019FFE8 00000000

0019FFEC 00000000

0019FFF0 00000000

0019FFF4 00000000

0019FFF8 00000000

0019FFFC 00000000

0019FF00 00000000

0019FF04 00000000

0019FF08 00000000

0019FF0C 00000000

0019FF10 00000000

0019FF14 00000000

0019FF18 00000000

0019FF1C 00000000

0019FF20 00000000

0019FF24 00000000

0019FF28 00000000

0019FF2C 00000000

0019FF30 00000000

0019FF34 00000000

0019FF38 00000000

0019FF3C 00000000

0019FF40 00000000

0019FF44 00000000

0019FF48 00000000

0019FF4C 00000000

0019FF50 00000000

0019FF54 00000000

0019FF58 00000000

0019FF5C 00000000

0019FF60 00000000

0019FF64 00000000

0019FF68 00000000

0019FF6C 00000000

0019FF70 00000000

0019FF74 00000000

0019FF78 00000000

0019FF7C 00000000

0019FF80 00000000

0019FF84 00000000

0019FF88 00000000

0019FF8C 00000000

0019FF90 00000000

0019FF94 00000000

0019FF98 00000000

0019FF9C 00000000

0019FFA0 00000000

0019FFA4 00000000

0019FFA8 00000000

0019FFAC 00000000

0019FFB0 00000000

0019FFB4 00000000

0019FFB8 00000000

0019FFBC 00000000

0019FFC0 00000000

0019FFC4 00000000

0019FFC8 00000000

0019FFCC 00000000

0019FFD0 00000000

0019FFD4 00000000

0019FFD8 00000000

0019FFDC 00000000

0019FFE0 00000000

0019FFE4 00000000

0019FFE8 00000000

0019FFEC 00000000

0019FFF0 00000000

0019FFF4 00000000

0019FFF8 00000000

0019FFFC 00000000

0019FF00 00000000

0019FF04 00000000

0019FF08 00000000

0019FF0C 00000000

0019FF10 00000000

0019FF14 00000000

0019FF18 00000000

0019FF1C 00000000

0019FF20 00000000

0019FF24 00000000

0019FF28 00000000

0019FF2C 00000000

0019FF30 00000000

0019FF34 00000000

0019FF38 00000000

0019FF3C 00000000

0019FF40 00000000

0019FF44 00000000

0019FF48 00000000

0019FF4C 00000000

0019FF50 00000000

0019FF54 00000000

0019FF58 00000000

0019FF5C 00000000

0019FF60 00000000

0019FF64 00000000

0019FF68 00000000

0019FF6C 00000000

0019FF70 00000000

0019FF74 00000000

0019FF78 00000000

0019FF7C 00000000

0019FF80 00000000

0019FF84 00000000

0019FF88 00000000

0019FF8C 00000000

0019FF90 00000000

0019FF94 00000000

0019FF98 00000000

0019FF9C 00000000

0019FFA0 00000000

0019FFA4 00000000

0019FFA8 00000000

0019FFAC 00000000

0019FFB0 00000000

0019FFB4 00000000

0019FFB8 00000000

0019FFBC 00000000

0019FFC0 00000000

0019FFC4 00000000

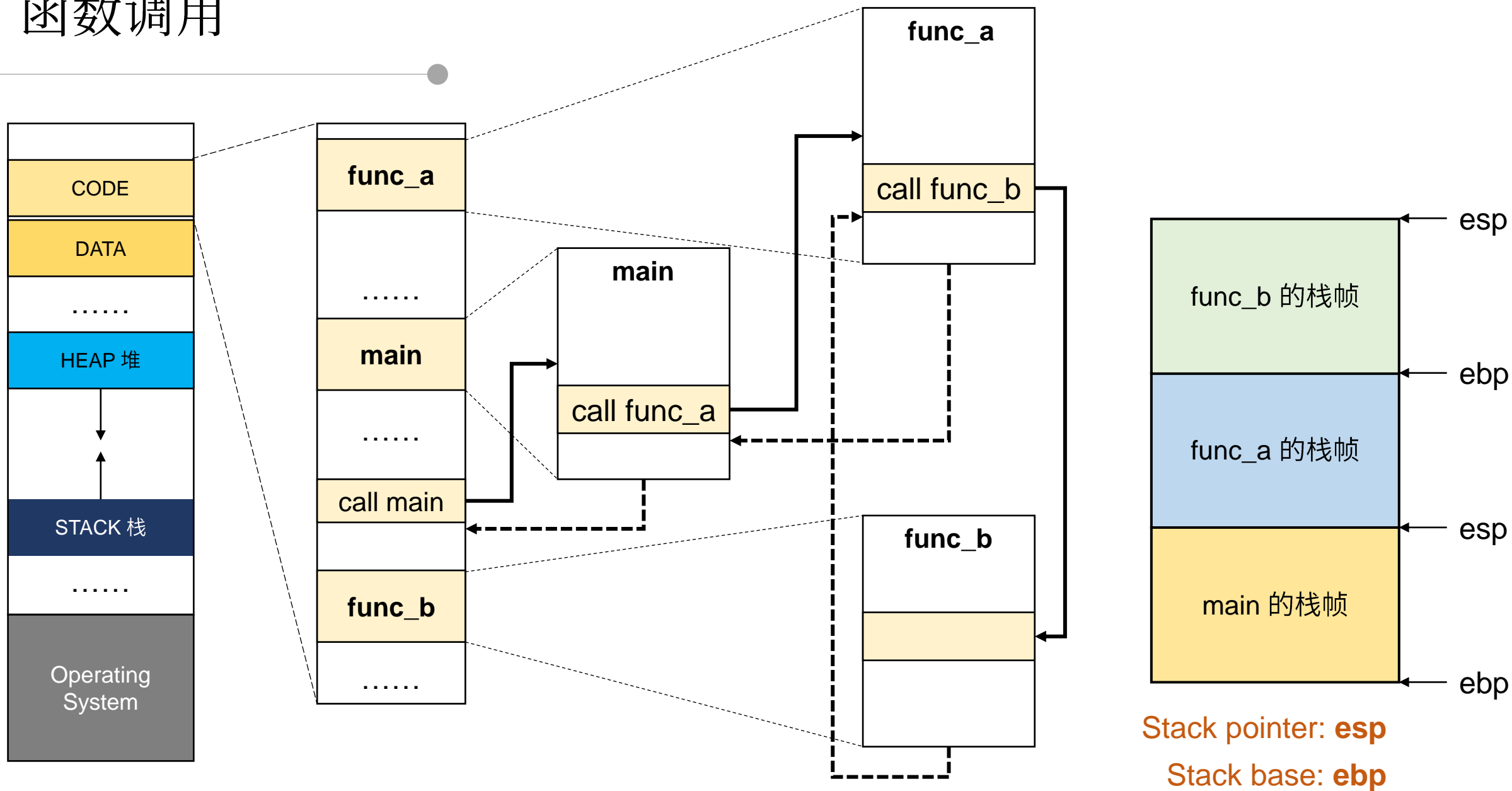
0019FFC8 00000000

0019FFCC 00000000

0019FFD0 00000000

0019FFD4 00000000

0019FFD8 00000000





# 函数调用的prologue与epilogue

retn 是一个复合指令:  
1. esp当前内容发给eip  
2. esp--

```
//simplestack.c
#include <stdio.h>
```

```
int add(int a, int b)
{
    int s;
    s = a + b;
    return s;
}
```

```
int main()
{
    int a=2, b=4, sum=0;
    sum = add(a, b);
    printf("%d\n", sum);
    return sum;
}
```

地址	HEX 数据	反汇编
00401000	55	push ebp
00401001	8BEC	mov ebp, esp
00401003	51	push ecx
00401004	8B45 08	mov eax, [ebp+arg_1]
00401007	0345 0C	add eax, [arg_2]
0040100A	8945 FC	mov [local.1], eax
0040100D	8B45 FC	mov eax, [local.1]
00401010	8BE5	mov esp, ebp
00401012	5D	pop ebp
00401013	C3	retn
00401014	CC	int3
00401015	CC	int3
00401016	CC	int3
00401017	CC	int3
00401018	CC	int3
00401019	CC	int3
0040101A	CC	int3
0040101B	CC	int3
0040101C	CC	int3
0040101D	CC	int3
0040101E	CC	int3
0040101F	CC	int3

地址	HEX 数据	反汇编
00401020	55	push ebp
00401021	8BEC	mov ebp, esp
00401023	83EC 0C	sub esp, 0xC
00401026	C745 FC 02 00	mov [local.1], 0x2
0040102D	C745 F8 04 00	mov [local.2], 0x4
00401034	C745 F4 00 00	mov [local.3], 0x0
0040103B	8B45 F8	mov eax, [local.2]
0040103E	50	push eax
0040103F	8B4D FC	mov ecx, [local.1]
00401042	51	push ecx
00401043	E8 B8FFFFFF	call simplest.00401000
00401048	83C4 08	add esp, 0x8
0040104B	8945 F4	mov [local.3], eax
0040104E	8B55 F4	mov edx, [local.3]
00401051	52	push edx
00401052	68 00B04000	push simplest.0040B000
00401057	E8 0A000000	call simplest.00401066
0040105C	83C4 08	add esp, 0x8
0040105F	8B45 F4	mov eax, [local.3]
00401062	8BE5	mov esp, ebp
00401064	5D	pop ebp
00401065	C3	retn

```
.text:00401000
.text:00401001
.text:00401003
.text:00401004
.text:00401007
.text:0040100A
.text:0040100D
.text:00401010
.text:00401012
.text:00401013
.text:00401013 sub_401000
```

```
push ebp
mov ebp, esp
push ecx
mov eax, [ebp+arg_0]
add eax, [ebp+arg_4]
mov [ebp+var_4], eax
mov eax, [ebp+var_4]
mov esp, ebp
pop ebp
retn
endp
```

prologue

epilogue

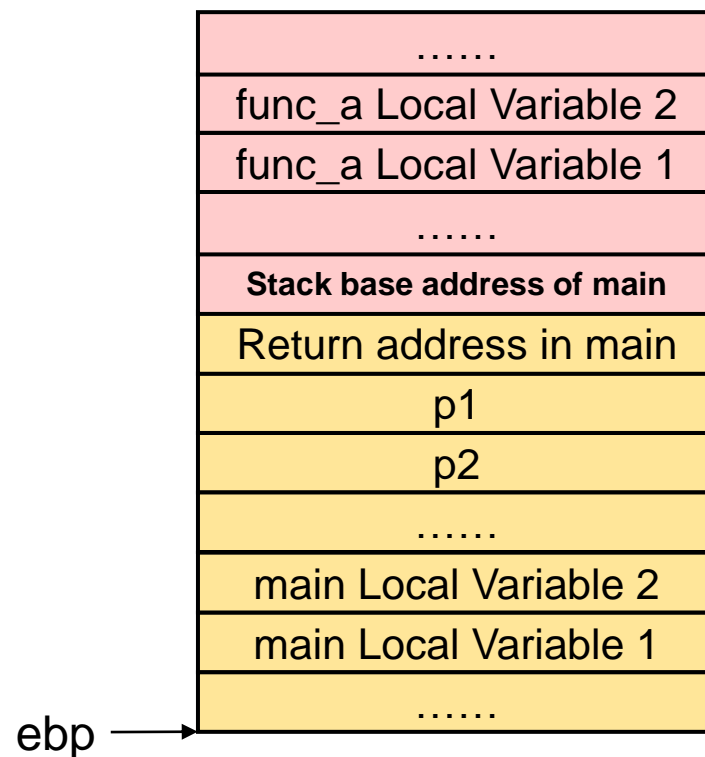
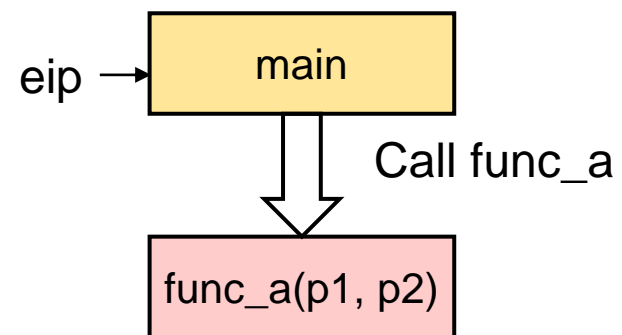
```
.text:00401020
.text:00401021
.text:00401023
.text:00401026
.text:0040102D
.text:00401034
.text:0040103B
.text:0040103E
.text:00401042
.text:00401043
.text:00401048
.text:0040104B
.text:0040104E
.text:00401051
.text:00401057
.text:0040105C
.text:0040105F
.text:00401062
.text:00401064
.text:00401065
.text:00401065 _main
```

```
push ebp
mov ebp, esp
sub esp, 0Ch
mov [ebp+var_4], 2
mov [ebp+var_8], 4
mov [ebp+var_C], 0
mov eax, [ebp+var_8]
push eax
mov ecx, [ebp+var_4]
push ecx
call sub_401000
add esp, 8
mov [ebp+var_C], eax
mov edx, [ebp+var_C]
push edx
push offset aD ; "%d\n"
call _printf
add esp, 8
mov eax, [ebp+var_C]
mov esp, ebp
pop ebp
retn
endp
```

prologue

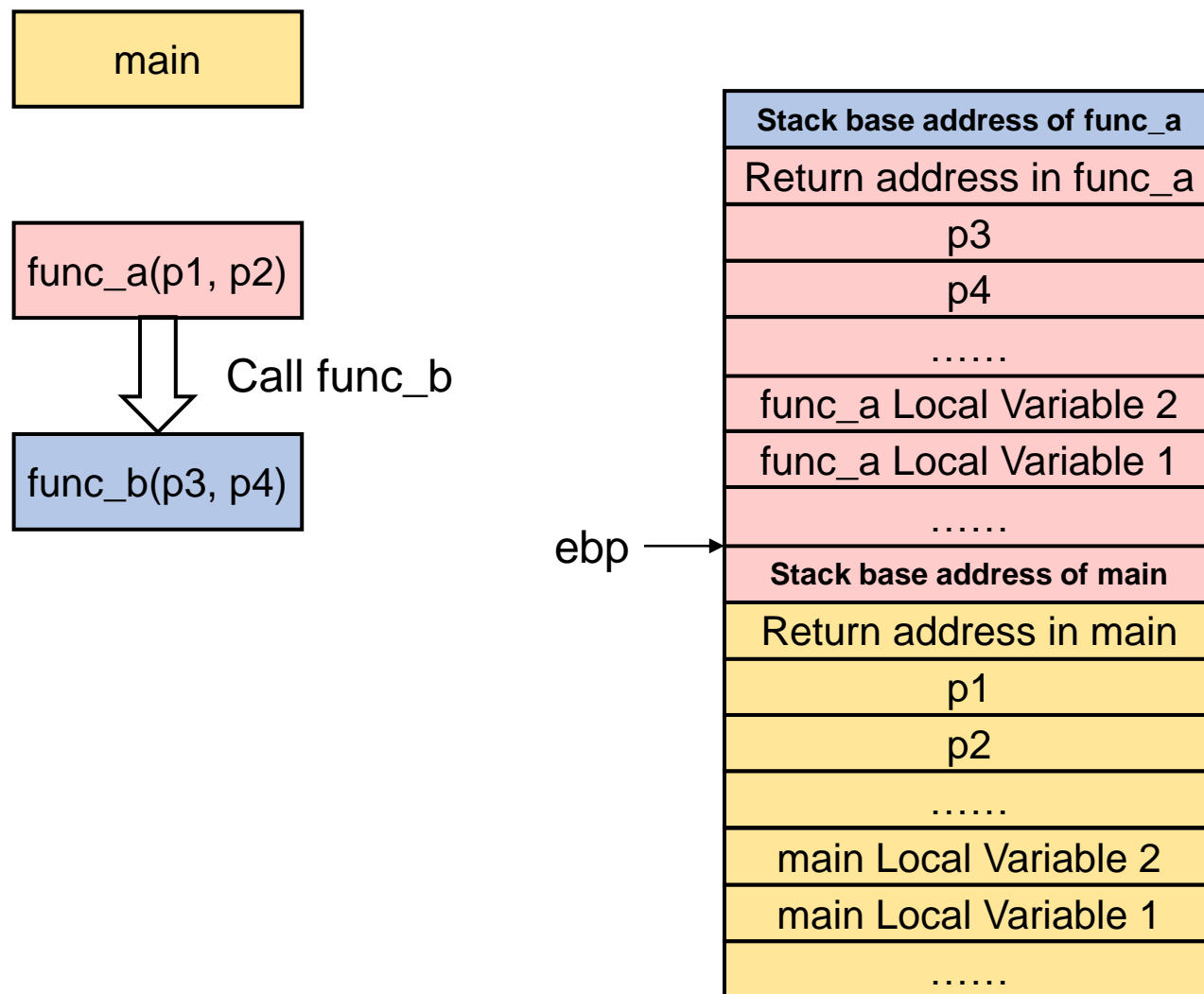
epilogue

# “栈”的运行 Stack



**call stack**

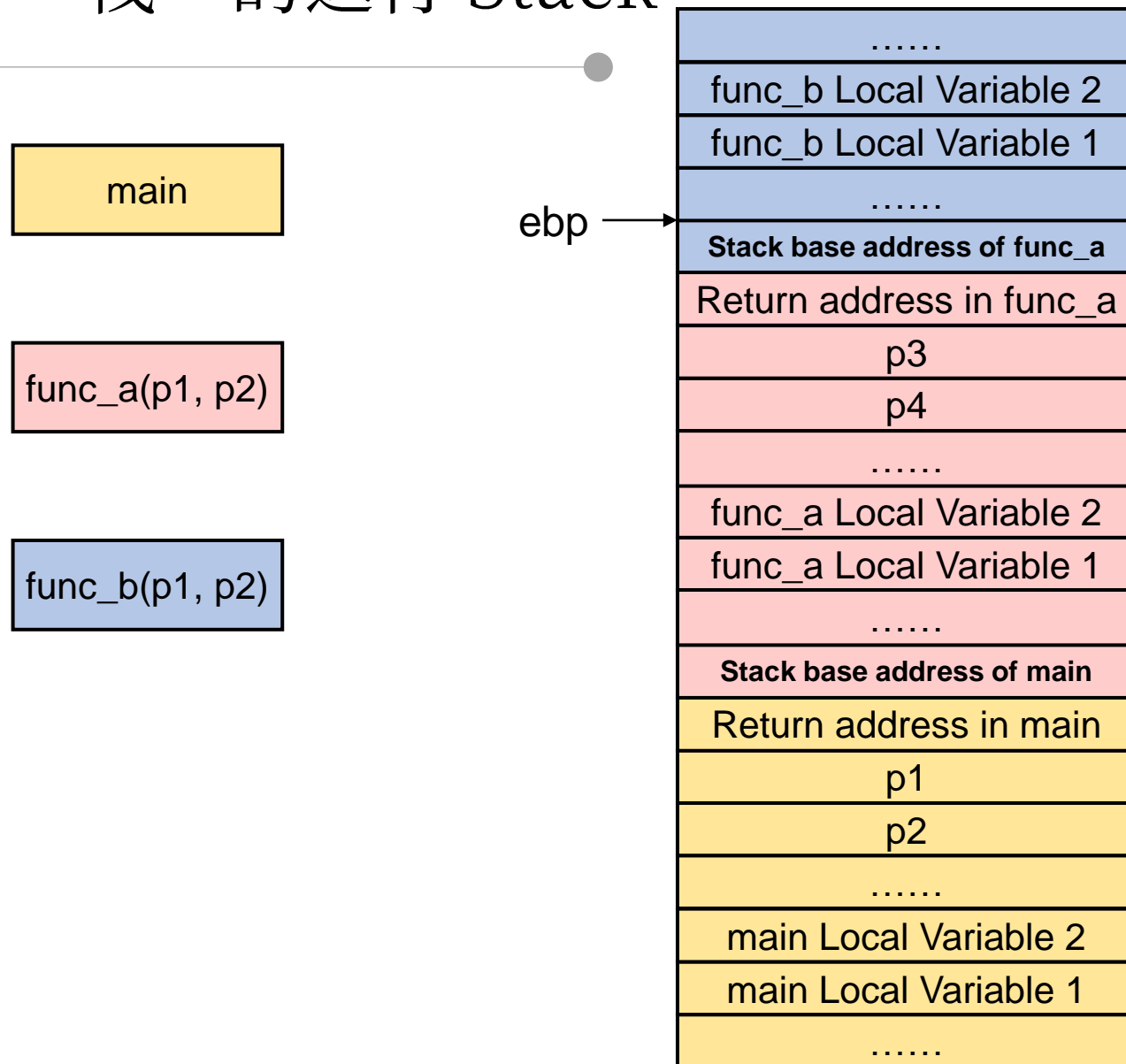
# “栈”的运行 Stack



**call stack**

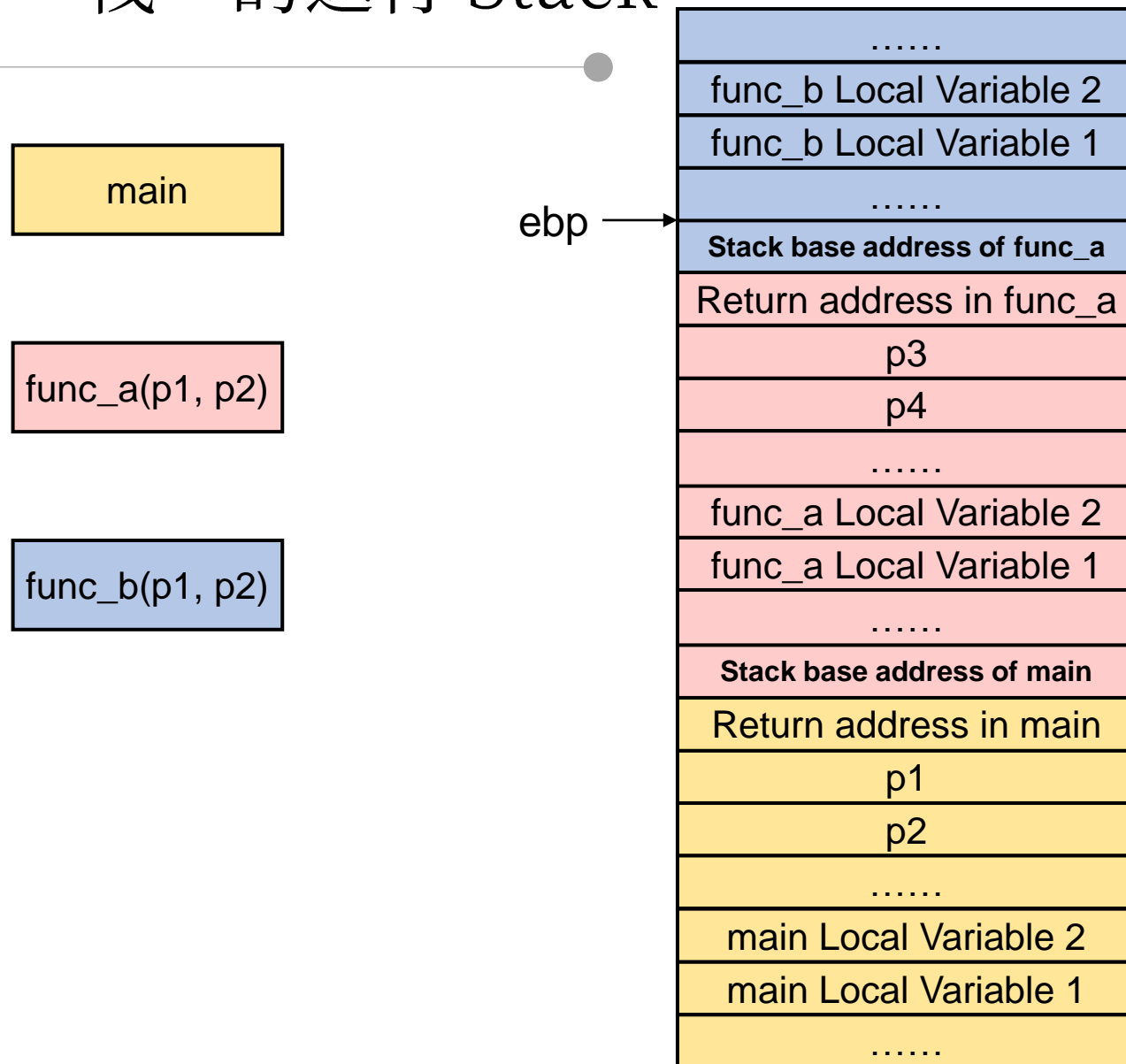


# “栈”的运行 Stack



**call stack**

# “栈”的运行 Stack



**call stack**

# 观察函数调用的汇编代码 Linux下

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void vuln(char *string)
{
    volatile int target;
    char buffer[64];

    target = 0;

    sprintf(buffer, string);

    if(target == 0xdeadbeef) {
        printf("you have hit the target correctly :)\n");
    }
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

leave也是一条复合指令：

```
mov esp, ebp
pop ebp
```

Dump of assembler code for function vuln:

```
0x080483f4 <vuln+0>:    push    ebp
0x080483f5 <vuln+1>:    mov     ebp,esp
0x080483f7 <vuln+3>:    sub     esp,0x68
0x080483fa <vuln+6>:    mov     DWORD PTR [ebp-0xc],0x0
0x08048401 <vuln+13>:   mov     eax,DWORD PTR [ebp+0x8]
0x08048404 <vuln+16>:   mov     DWORD PTR [esp+0x4],eax
0x08048408 <vuln+20>:   lea     eax,[ebp-0x4c]
0x0804840b <vuln+23>:   mov     DWORD PTR [esp],eax
0x0804840e <vuln+26>:   call    0x08048300 <sprintf@plt>
0x08048413 <vuln+31>:   mov     eax,DWORD PTR [ebp-0xc]
0x08048416 <vuln+34>:   cmp     eax,0xdeadbeef
0x0804841b <vuln+39>:   jne     0x08048429 <vuln+53>
0x0804841d <vuln+41>:   mov     DWORD PTR [esp],0x08048510
0x08048424 <vuln+48>:   call    0x08048330 <puts@plt>
0x08048429 <vuln+53>:   leave
0x0804842a <vuln+54>:   ret
```

End of assembler dump.

Dump of assembler code for function main:

```
0x0804842b <main+0>:    push    ebp
0x0804842c <main+1>:    mov     ebp,esp
0x0804842e <main+3>:    and     esp,0xffffffff
0x08048431 <main+6>:    sub     esp,0x10
0x08048434 <main+9>:    mov     eax,DWORD PTR [ebp+0xc]
0x08048437 <main+12>:   add     eax,0x4
0x0804843a <main+15>:   mov     eax,DWORD PTR [eax]
0x0804843c <main+17>:   mov     DWORD PTR [esp],eax
0x0804843f <main+20>:   call    0x080483f4 <vuln>
0x08048444 <main+25>:   leave
0x08048445 <main+26>:   ret
```

End of assembler dump.



```

0x080483f4 <vuln+0>:      push    ebp
0x080483f5 <vuln+1>:      mov     ebp,esp
0x080483f7 <vuln+3>:      sub     esp,0x68
0x080483fa <vuln+6>:      mov     DWORD PTR [ebp-0xc],0x0
0x08048401 <vuln+13>:     mov     eax,DWORD PTR [ebp+0x8]
0x08048404 <vuln+16>:     mov     DWORD PTR [esp+0x4],eax
0x08048408 <vuln+20>:     lea     eax,[ebp-0x4c]
0x0804840b <vuln+23>:     mov     DWORD PTR [esp],eax
0x0804840e <vuln+26>:     call   0x8048300 <sprintf@plt>
0x08048413 <vuln+31>:     mov     eax,DWORD PTR [ebp-0xc]
0x08048416 <vuln+34>:     cmp     eax,0xdeadbeef
0x0804841b <vuln+39>:     jne     0x8048429 <vuln+53>
0x0804841d <vuln+41>:     mov     DWORD PTR [esp],0x8048510
0x08048424 <vuln+48>:     call   0x8048330 <puts@plt>
0x08048429 <vuln+53>:     leave
0x0804842a <vuln+54>:     ret

```

```

0x0804842b <main+0>:      push    ebp
0x0804842c <main+1>:      mov     ebp,esp
0x0804842e <main+3>:      and     esp,0xffffffff
0x08048431 <main+6>:      sub     esp,0x10
0x08048434 <main+9>:      mov     eax,DWORD PTR [ebp+0xc]
0x08048437 <main+12>:     add     eax,0x4
0x0804843a <main+15>:     mov     eax,DWORD PTR [eax]
0x0804843c <main+17>:     mov     DWORD PTR [esp],eax
0x0804843f <main+20>:     call   0x80483f4 <vuln>
0x08048444 <main+25>:     leave
0x08048445 <main+26>:     ret

```

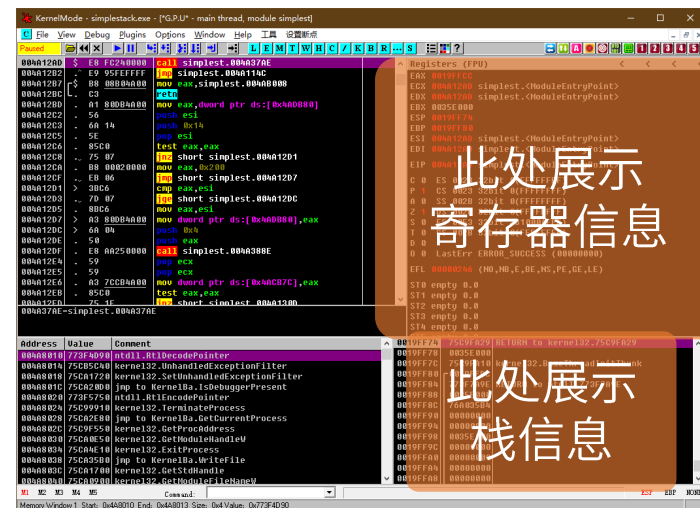
End of assembler dump.

# 观察call main

## 调用程序入口main函数

```
00401252: E8 C9 FD FF FF    call simplest.00401020
00401257: 83 C4 0C          add esp,0xC
```

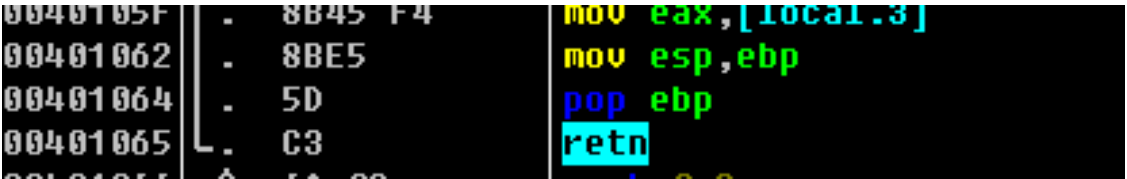
地址	HEX 数据	反汇编
00401230	. 3BC6	cmp eax,esi
00401232	74 07	je Xsimplest.00401238
00401234	50	push eax
00401235	E8 12170000	call simplest.0040294C
0040123A	59	pop ecx
0040123B	> A1 F0BD4000	mov eax,dword ptr ds:[0x40BDF0]
00401240	A3 F4BD4000	mov dword ptr ds:[0x40BDF4],eax
00401245	50	push eax
00401246	FF35 E8BD4000	push dword ptr ds:[0x40BDE8]
0040124C	FF35 E4BD4000	push dword ptr ds:[0x40BDE4]
00401252	E8 C9FDFFFF	call simplest.00401020
00401257	83C4 0C	add esp,0xC
0040125A	8945 E0	mov dword ptr ss:[ebp-0x20],eax
0040125D	3975 E4	cmp dword ptr ss:[ebp-0x1C],esi
00401260	75 06	jnz Xsimplest.00401268
00401262	50	push eax
00401263	E8 9A160000	call simplest.00402902
00401268	> E8 C1160000	call simplest.0040292E
0040126D	EB 2E	jmp Xsimplest.0040129D
0040126F	8B45 EC	mov eax,dword ptr ss:[ebp-0x14]
00401272	8B08	mov ecx,dword ptr ds:[eax]
00401274	8B09	mov ecx,dword ptr ds:[ecx]
00401276	894D DC	mov dword ptr ss:[ebp-0x24],ecx



1. 此时寄存器 eip 的值为 00401252，表示即将执行 00401252 处的指令“call 00401020”
2. 在栈空间的顶部生成一个新的栈，此时该栈为空，栈顶和栈底指针指向同一个地址 0012FF7C
3. 向新栈中push一个地址(4B)，用于返回调用函数，该地址是 call指令（位于00401252）的下一条指令的地址 00401257
4. eip 指向被调用函数 simplestack 的入口：00401020

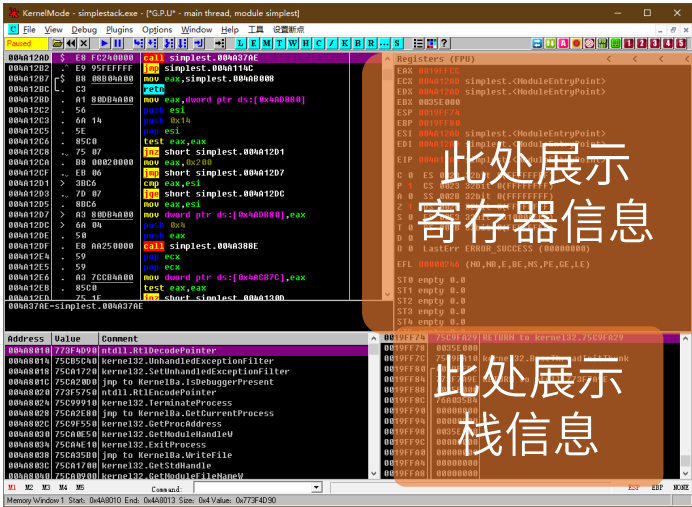
被调用函数返回main函数

00401065: C3      retn



栈上数据:

地址	数值	注释
0012FF78	0012FFC0	
0012FF7C	00401257	返回到 simplest.00401257 来自 simplest.00401020
0012FF80	00000001	
0012FF84	00413088	



- 1. 此时ebp已经回归到调用函数的栈底，eip=00401065 即将执行指令 retn
- 2. 和call一样，retn也包含一系列组合动作：
  - 1. 将esp指向的地址上保存的地址，赋给eip 此时 eip=00401257 (这一步的目的是返回调用函数空)，即指向了call main的下一条指令“add esp,0xC”的地址
  - 2. esp--
- 3. 此时可观察：ebp恢复为调用函数的ebp；esp指向调用函数的栈顶；被调用函数simplestack的栈空间消失。



# 练习：使用Ollydbg观察栈上的数据变化

```
//observestack.c
#include <stdio.h>

void f1()
{
    int a=1, b=2, c=3;
};

void f2()
{
    int a, b, c;
    printf("%d, %d, %d\n", a, b, c);
};

int main()
{
    f1();
    f2();
};
```

↓

```
> cl observestack.c
```

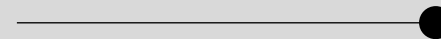
observe.exe

```
> observestack.exe
> 1, 2, 3
```



练习目标：a) 学习Ollydbg；b) 观测栈变化，分析出错原因

1. 编译源代码
2. 在Ollydbg中打开程序文件
3. 开始单步调试
4. 步入（step in）函数 f2 进行单步调试
5. 观察 f2 的栈上数据变化（Ollydbg界面右下角）





# 缓冲区溢出



当程序试图将超出其容量的数据放入缓冲区时，或者当程序试图将数据放入超出缓冲区边界的内存区域时，就会出现缓冲区溢出情况。  
最简单的错误类型，也是导致缓冲区溢出最常见的原因，是“经典”的情况，即程序复制缓冲区而不限限制复制的数量。

# “栈溢出”现象：Stack Smash

```

/*echo.c*/
void echo()
{
    char buf[4]; //Too
    gets(buf);   //small!
    puts(buf);
}

void call_echo()
{
    echo();
}

void main()
{
    call_echo();
}

```

echo:

```

.text:00401000 push    ebp
.text:00401001 mov     ebp, esp
.text:00401003 push    ecx
.text:00401004 lea     eax, [ebp+var_4]
.text:00401007 push    eax ; char *
.text:00401008 call    _gets
.text:0040100D add     esp, 4
.text:00401010 lea     ecx, [ebp+var_4]
.text:00401013 push    ecx ; char *
.text:00401014 call    _puts
.text:00401019 add     esp, 4
.text:0040101C mov     esp, ebp
.text:0040101E pop     ebp
.text:0040101F retn

```

call\_echo:

```

.text:00401020 push    ebp
.text:00401021 mov     ebp, esp
.text:00401023 call    sub_401000
.text:00401028 pop     ebp
.text:00401029 retn

```

# “栈溢出”现象：Stack Smash

在Ollydbg中查看函数echo()栈帧上的变化：

1. 00401000 进入echo()

The screenshot shows the Ollydbg interface with the assembly code for the `echo()` function. The code starts at address `00401000` and ends at `00401028`. The stack frame shows the return address `00401023` and the return value `00401028`.

地址	HEX	数据	反汇编
00401000	55		push ebp
00401001	8BEC		mov ebp,esp
00401003	51		push ecx
00401004	8D45 FC		lea eax,[local.1]
00401007	50		push eax
00401008	E8 79030000		call echo.00401038
0040100D	83C4 04		add esp,0x4
00401010	8D4D FC		lea ecx,[local.1]
00401013	51		push ecx
00401014	E8 23000000		call echo.0040103C
00401019	83C4 04		add esp,0x4
0040101C	8BE5		mov esp,ebp
0040101E	5D		pop ebp
0040101F	C3		ret
00401020	55		push ebp
00401021	8BEC		mov ebp,esp
00401023	E8 D8FFFFFF		call echo.00401000
00401028	5D		pop ebp

寄存器 (FPU)

寄存器	值
EAX	003B30B8
ECX	00000001
EDX	0040C6C8 echo.0040C6C8
EBX	7FFD7000
ESP	0012FF6C
EBP	0012FF70
ESI	00000000
EDI	7C930228 ntdll.7C930228
EIP	00401000 echo.00401000
C 0	ES 0023 32位 0(FFFFFFFF)
P 1	CS 001B 32位 0(FFFFFFFF)
A 0	SS 0023 32位 0(FFFFFFFF)
Z 1	DS 0023 32位 0(FFFFFFFF)
S 0	FS 003B 32位 7FFDF000(FFF)
T 0	GS 0000 NULL
D 0	
O 0	LastErr ERROR_PROC_NOT_FOUND (0000007F)
EFL	00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0	empty -UNORM D0A8 01050104 00000000

地址	HEX	数据	地址	数值	注释
00408000	6D 0C 81 7C	09 95 83 7C	0012FF64	004053B8	echo.004053B8
00408010	37 CD 80 7C	89 0C 81 7C	0012FF68	00402E83	echo.00402E83
00408020	54 1E 80 7C	7A 13 93 7C	0012FF6C	00401028	返回到 echo.00401028 来自 echo.00401000
00408030	EF F6 82 7C	D9 32 93 7C	0012FF70	0012FF78	
00408040	95 DE 80 7C	79 AA 94 7C	0012FF74	00401038	返回到 echo.00401038 来自 echo.00401020
00408050	A2 BF 81 7C	FF 12 81 7C	0012FF78	0012FFC0	
00408060	07 41 81 7C	74 A1 80 7C	0012FF7C	004014D0	返回到 echo.004014D0 来自 echo.00401030
00408070	E0 97 80 7C	65 9C 80 7C	0012FF80	00000001	
00408080	30 FE 92 7C	D0 97 80 7C	0012FF84	003B30B8	
00408090	08 09 81 7C	C7 A4 80 7C	0012FF88	003B30B8	
004080A0	E9 17 80 7C	46 24 80 7C	0012FF8C	F921C578	
004080B0	7B 5F 87 7C	DC A0 81 7C	0012FF90	7C930228	ntdll.7C930228

Command: 起始:408000 结束:407FFF 当前值:7C810C6D

# “栈溢出”现象：Stack Smash

在Ollydbg中查看函数echo()栈帧上的变化：

1. 00401000 进入echo()
2. 运行到 gets(buf)，用户输入 “1234567”

The screenshot shows the Ollydbg interface with the echo() function selected. The main window displays the assembly code and the stack frame. The registers window on the right shows the current state of the CPU registers.

地址	HEX	数据	反汇编
00401030	55		push ebp
00401031	8BEC		mov ebp,esp
00401033	E8 E8FFFFFF		call echo.00401020
00401038	33C0		xor eax,eax
0040103A	5D		pop ebp
0040103B	C3		ret
0040103C	6A 0C		push 0xC
0040103E	68 A0984000		push echo.004098A0
00401043	E8 480F0000		call echo.00401F90
00401048	83CE FF		or esi,0xFFFFFFFF
0040104B	8975 E4		mov [local.7],esi
0040104E	33C0		xor eax,eax
00401050	3945 08		cmp [arg.1],eax
00401053	0F95C0		setne al
00401056	85C0		test eax,eax
00401058	75 17		jnz Xecho.00401071
0040105A	E8 E30E0000		call echo.00401F42
0040105F	C700 16000000		mov dword ptr ds:[eax],0x16

本地调用来自 00401014

地址	HEX	数据	地址	数值	注释
00408000	6D 0C 81 7C	09 95 83 7C	0012FF5C	00401019	返回到 echo.00401019 来自 echo.0040103C
00408010	37 CD 80 7C	89 0C 81 7C	0012FF60	0012FF64	ASCII "1234567"
00408020	54 1E 80 7C	7A 13 93 7C	0012FF64	34333231	
00408030	EF F6 82 7C	D9 32 93 7C	0012FF68	00373635	
00408040	95 DE 80 7C	79 AA 94 7C	0012FF6C	00401028	返回到 echo.00401028 来自 echo.00401000
00408050	A2 BF 81 7C	FF 12 81 7C	0012FF70	0012FF78	
00408060	07 41 81 7C	74 A1 80 7C	0012FF74	00401038	返回到 echo.00401038 来自 echo.00401020
00408070	E0 97 80 7C	65 9C 80 7C	0012FF78	0012FFC0	
00408080	30 FE 92 7C	D0 97 80 7C	0012FF7C	004014D0	返回到 echo.004014D0 来自 echo.00401030
00408090	08 09 81 7C	C7 A4 80 7C	0012FF80	00000001	
004080A0	E9 17 80 7C	46 24 80 7C	0012FF84	003B3088	
004080B0	7B 5F 87 7C	DC A0 81 7C	0012FF88	003B3088	

寄存器 (FPU)

寄存器	值	注释
EAX	0012FF64	ASCII "1234567"
ECX	0012FF64	ASCII "1234567"
EDX	0040C740	echo.0040C740
EBX	7FFD8000	
ESP	0012FF5C	
EBP	0012FF68	ASCII "567"
ESI	00000000	
EDI	7C930228	ntdll.7C930228
EIP	0040103C	echo.0040103C
C 0	ES 0023	32位 0xFFFFFFFF
P 0	CS 001B	32位 0xFFFFFFFF
A 0	SS 0023	32位 0xFFFFFFFF
Z 0	DS 0023	32位 0xFFFFFFFF
S 0	FS 003B	32位 7FFDF000(FFF)
T 0	GS 0000	NULL
D 0		
O 0		
LastErr	ERROR_PROC_NOT_FOUND (0000007F)	
EFL	00000202	(NO,NB,NE,A,NS,PO,GE,G)
ST0	empty	-UNORM D0A8 01050104 00000000

断点位于 echo.0040103C

# “栈溢出”现象：Stack Smash

在Ollydbg中查看函数echo()栈帧上的变化：

1. 00401000 进入echo()
2. 运行到 gets(buf)，用户输入 “1234567”
3. 用户继续输入，直到输入 “123456789”

Ollydbg 窗口显示了函数 echo() 的汇编代码和寄存器状态。主窗口显示了从地址 00401030 到 0040105F 的汇编代码。寄存器窗口显示了 EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI 和 EIP 的值。堆栈窗口显示了从地址 00408000 到 0040800F 的堆栈内容。命令窗口显示了当前的命令：起始:408000 结束:407FFF 当前值:7C810C6D。

地址	HEX 数据	反汇编
00401030	55	push ebp
00401031	8BEC	mov ebp, esp
00401033	E8 E8FFFFFF	call echo.00401020
00401038	33C0	xor eax, eax
0040103A	5D	pop ebp
0040103B	C3	ret
0040103C	6A 0C	push 0xC
0040103E	68 A0984000	push echo.004098A0
00401043	E8 480F0000	call echo.00401F90
00401048	83CE FF	or esi, 0xFFFFFFFF
0040104B	8975 E4	mov [local.7], esi
0040104E	33C0	xor eax, eax
00401050	3945 08	cmp [arg.1], eax
00401053	0F95C0	setne al
00401056	85C0	test eax, eax
00401058	75 17	jnz Xecho.00401071
0040105A	E8 E30E0000	call echo.00401F42
0040105F	C700 16000000	mov dword ptr ds:[eax], 0x16

寄存器 (FPU)	值
EAX	0012FF64 ASCII "123456789"
ECX	0012FF64 ASCII "123456789"
EDX	0040C740 echo.0040C740
EBX	7FFD7000
ESP	0012FF5C
EBP	0012FF68 ASCII "56789"
ESI	00000000
EDI	7C930228 ntdll.7C930228
EIP	0040103C echo.0040103C

地址	HEX 数据	地址	数值	注释
00408000	6D 0C 81 7C	0012FF5C	00401019	返回到 echo.00401019 来自 echo.0040103C
00408010	37 CD 80 7C	0012FF60	0012FF64	ASCII "123456789"
00408020	54 1E 80 7C	0012FF64	34333231	
00408030	EF F6 82 7C	0012FF68	38373635	
00408040	95 DE 80 7C	0012FF6C	00400039	echo.00400039
00408050	A2 BF 81 7C	0012FF70	0012FF78	
00408060	07 41 81 7C	0012FF74	00401038	返回到 echo.00401038 来自 echo.00401020
00408070	E0 97 80 7C	0012FF78	0012FFC0	
00408080	30 FE 92 7C	0012FF7C	004014D0	返回到 echo.004014D0 来自 echo.00401030
00408090	08 09 81 7C	0012FF80	00000001	
004080A0	E9 17 80 7C	0012FF84	003B3088	
004080B0	7B 5F 87 7C	0012FF88	003B3088	

命令: 起始:408000 结束:407FFF 当前值:7C810C6D

# “栈溢出”现象：Stack Smash

当控制台输入：“1234567\0”  
栈帧内容如下：

	.....			
0012FF64	34	33	32	31
0012FF68	00	37	36	35
0012FF6C	00	40	10	28
0012FF70	.....			

# “栈溢出”现象：Stack Smash

当控制台输入：“1234567\0”  
栈帧内容如下：

调用函数的ebp  
返回地址

	.....			
0012FF64	34	33	32	31
0012FF68	00	37	36	35
0012FF6C	00	40	10	28
0012FF70	.....			

当控制台输入：“123456789\0”  
栈帧内容如下：

	.....			
0012FF64	34	33	32	31
0012FF68	38	37	36	35
0012FF6C	00	40	00	39
0012FF70	.....			

0012FF68 原来存放调用函数的栈底地址 ebp (0012FF70) ，已被完全覆盖 (绿色、橙色)  
0012FF6C 原来存放调用函数的返回地址 return address (00401028) ，部分被覆盖(橙色)

# “栈溢出”现象：Stack Smash

当控制台输入：“1234567\0”  
栈帧内容如下：

	.....			
0012FF64	34	33	32	31
0012FF68	00	37	36	35
0012FF6C	00	40	10	28
0012FF70	.....			

当控制台输入一段精心构造的数据：  
“\X31\X32\X33\X34  
\X68\XFF\X12\X00  
\X00\X10\X40\X00”

	.....			
0012FF64	34	33	32	31
0012FF68	00	12	FF	68
0012FF6C	00	40	10	00
0012FF70	.....			

程序继续运行，将会发生什么





# 练习:

---

## Protostar Stack0

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    volatile int modified;
    char buffer[64];

    modified = 0;
    gets(buffer);

    if(modified != 0) {
        printf("you have changed the 'modified' variable\n");
    } else {
        printf("Try again?\n");
    }
}
```

# 本章要点

- 函数调试工具简介
  - 静态调试工具：IDA和objdump
  - 动态调试工具：Ollydbg和GDB
- 函数调用：
  - 函数调用流程
  - 栈帧的生灭
  - ebp和esp的移动
  - 本地变量和参数的摆放
  - 返回地址的摆放
  - 栈上噪音
  - 32位和64位 调用约定的对比