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

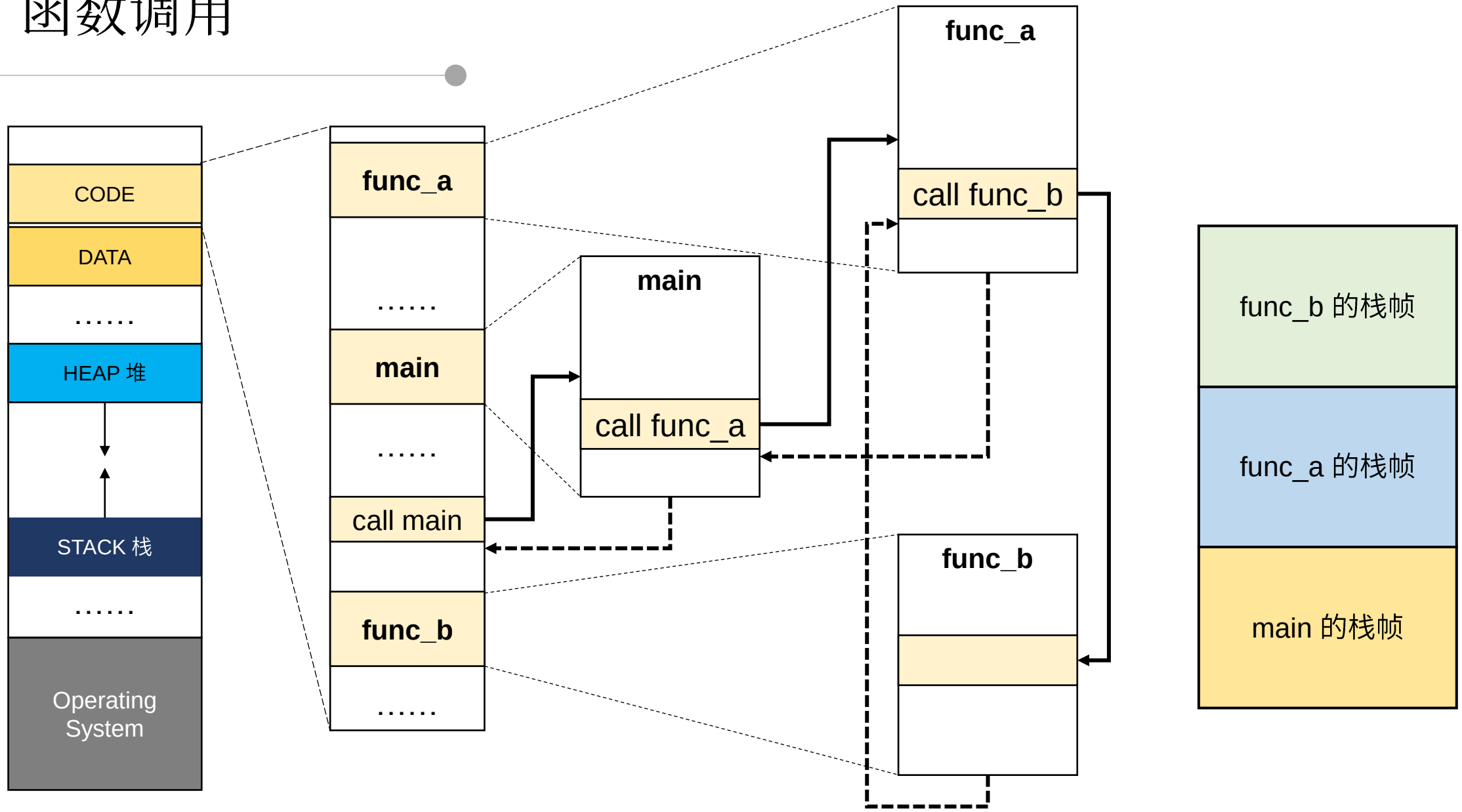
目录/CONTENTS

01

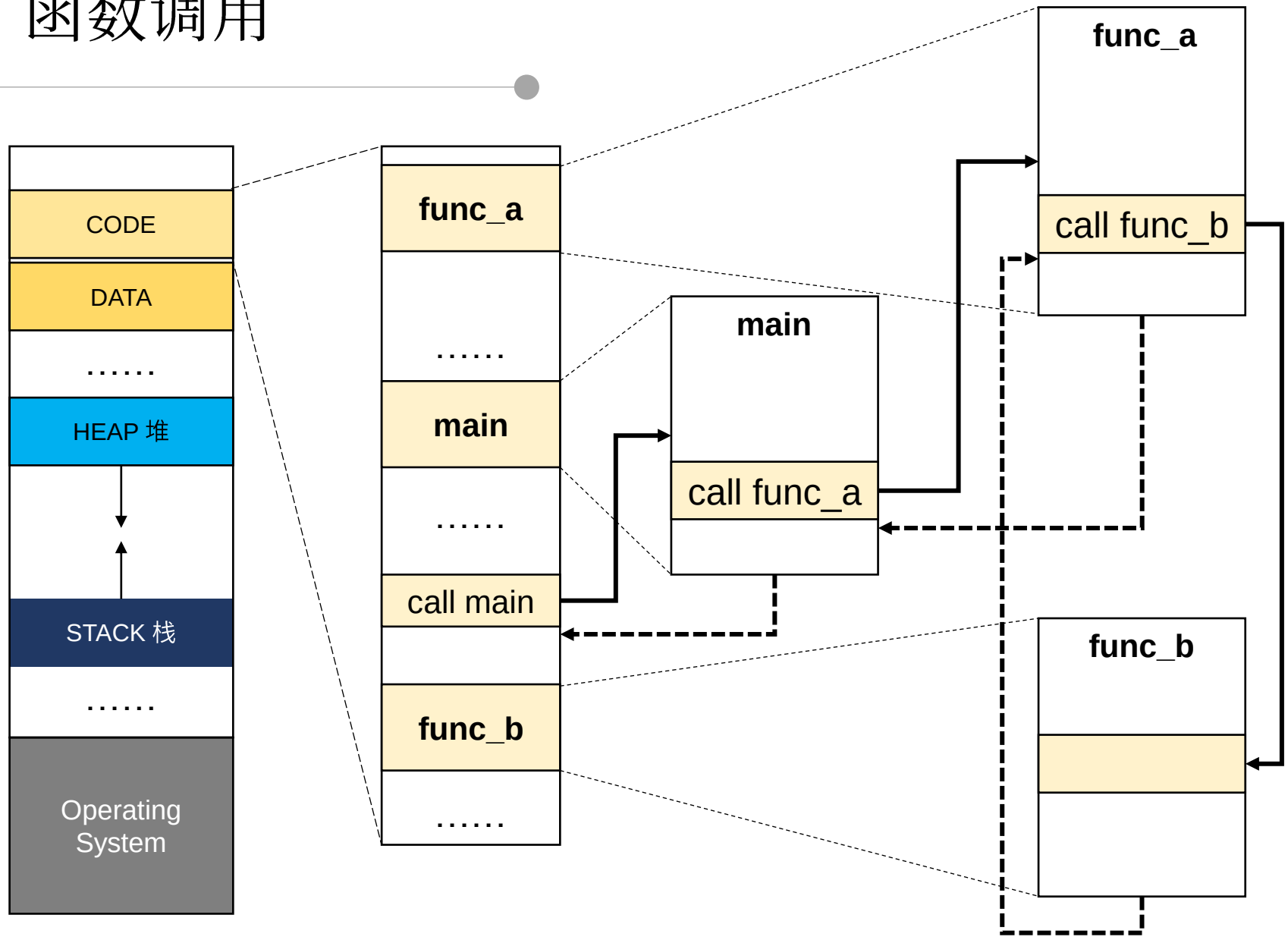
函 数 调 用

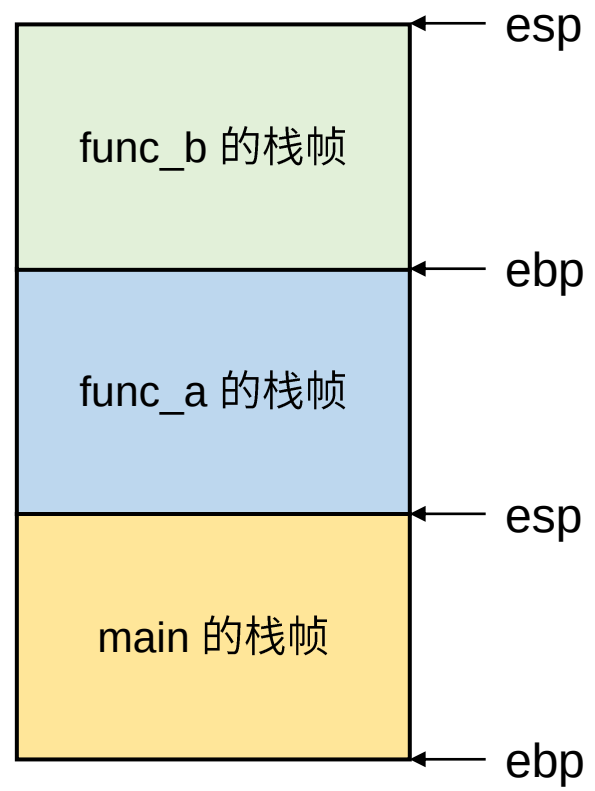
Procedure Call

函数调用


CODE
DATA
……
HEAP 堆
STACK 栈
……
Operating System
func_a
……
main
……
call main
func_b
……
main
call func_a
func_a
call func_b
func_b
func_b 的栈帧
func_a 的栈帧
main 的栈帧

函数调用





Stack pointer: **esp**

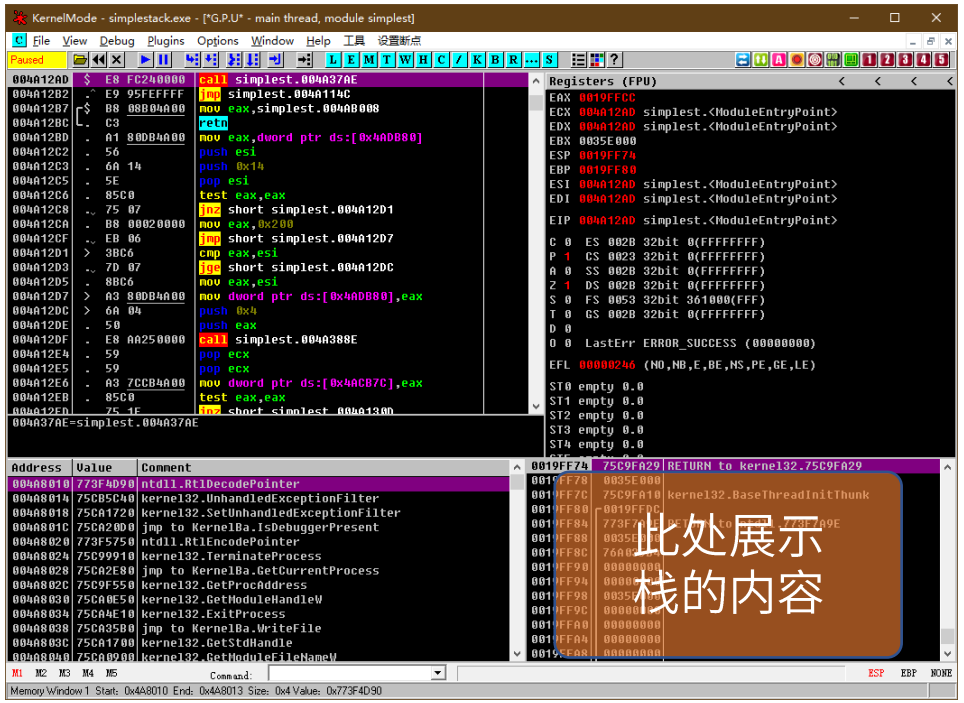
Stack base: **ebp**

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



函数调用案例

```c
//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)

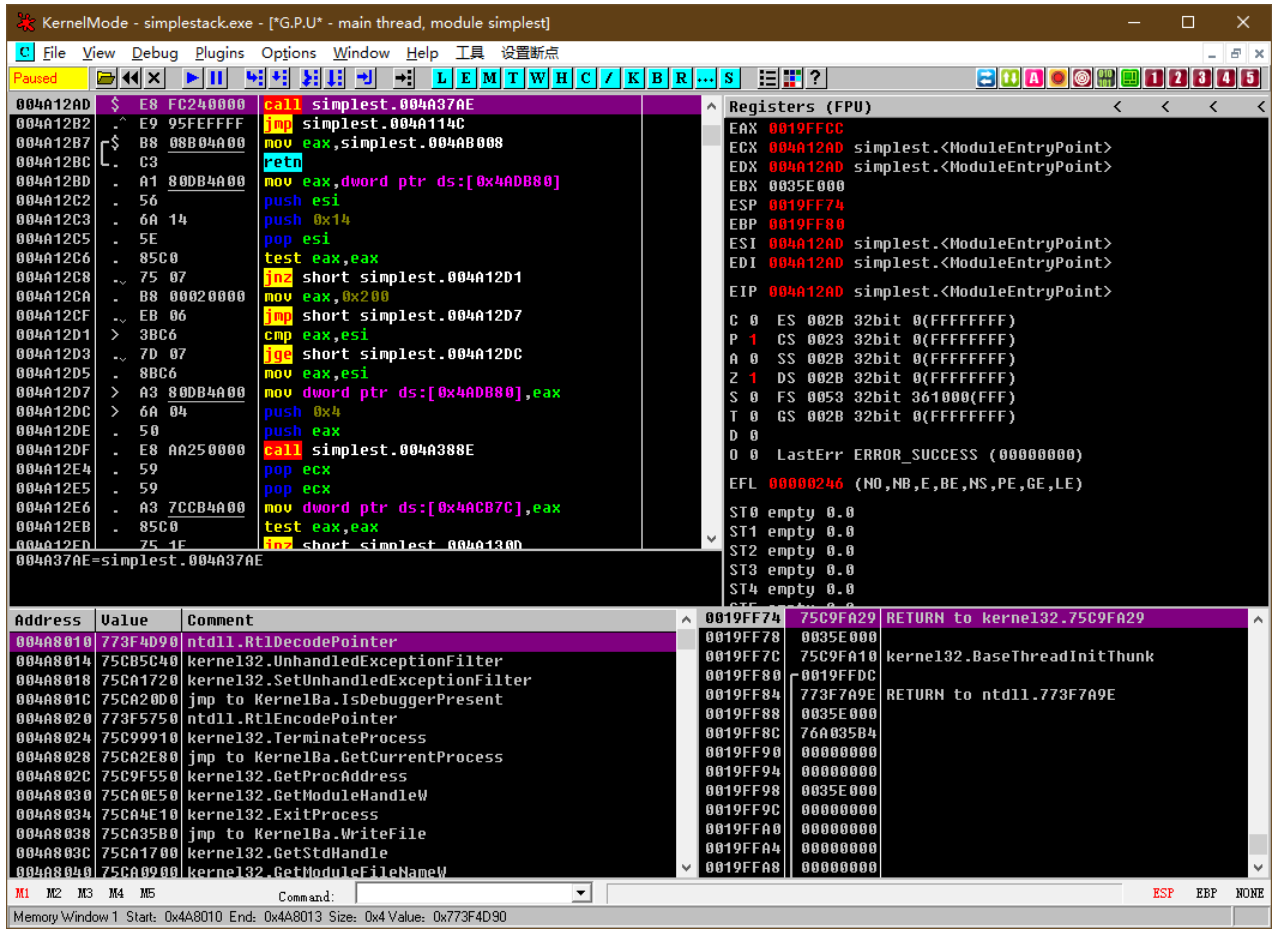
```
> cl simplestack.c
```

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

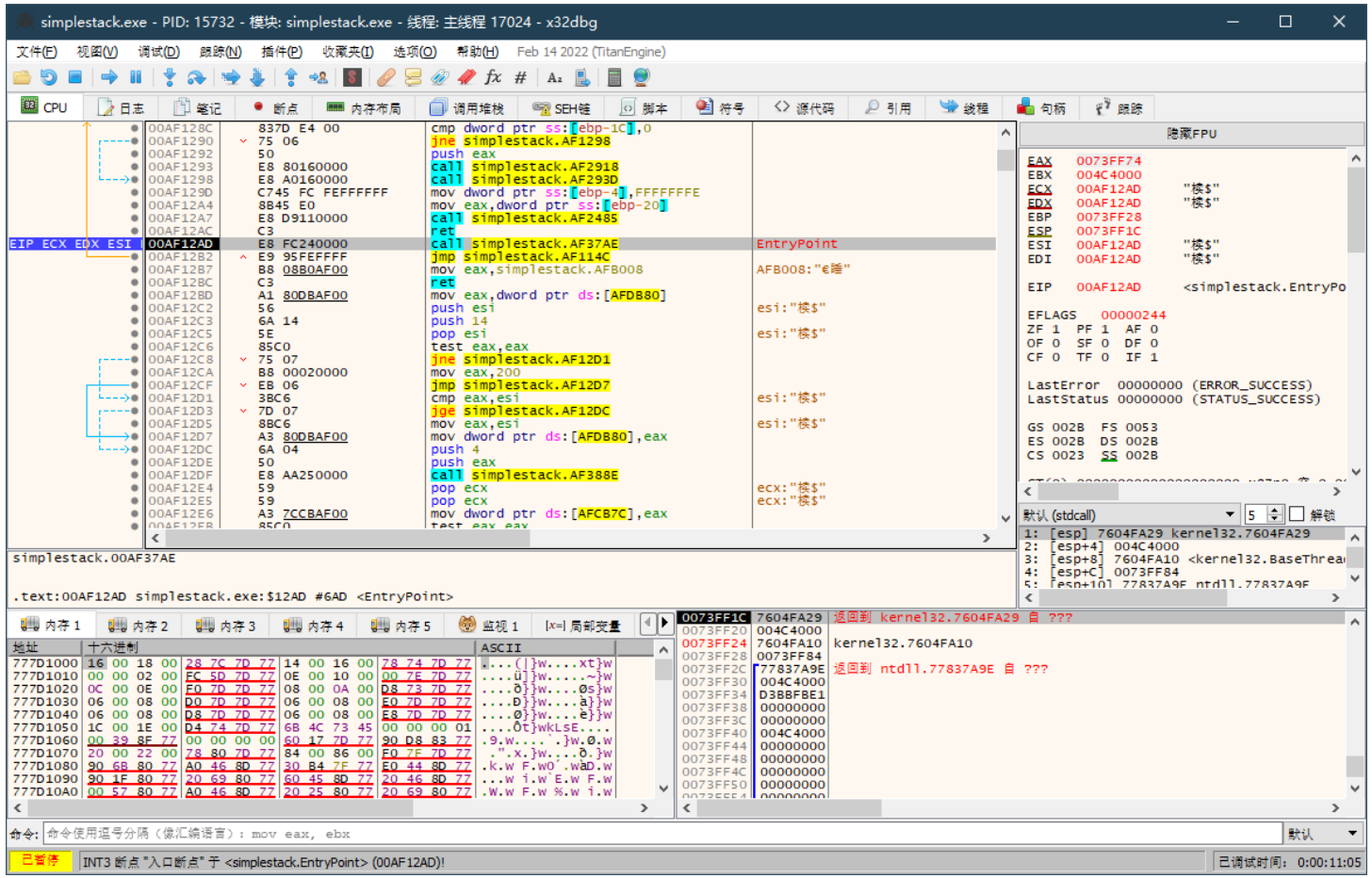
OllyDbg



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$
```

```c
/*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;
}
```

```c
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。

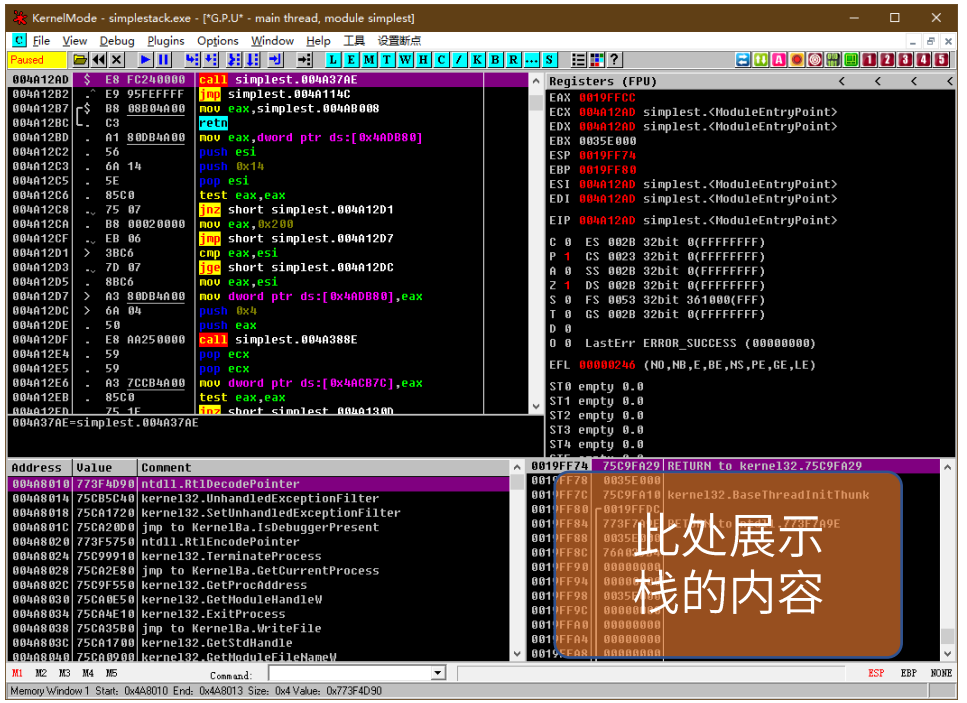
栈 Stack

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

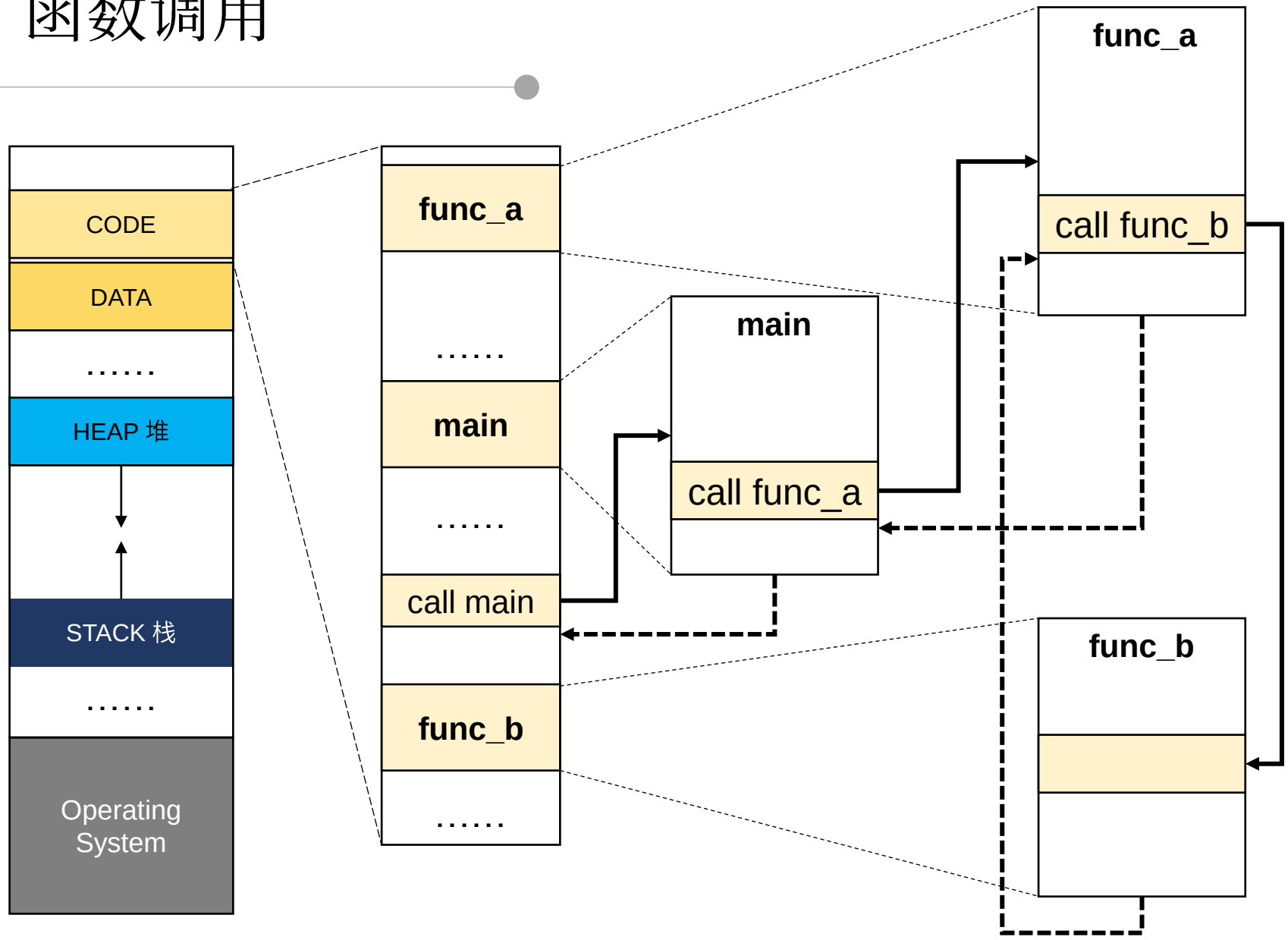
- 存储内容：
  - 本地变量
  - 函数的返回信息
  - 临时空间

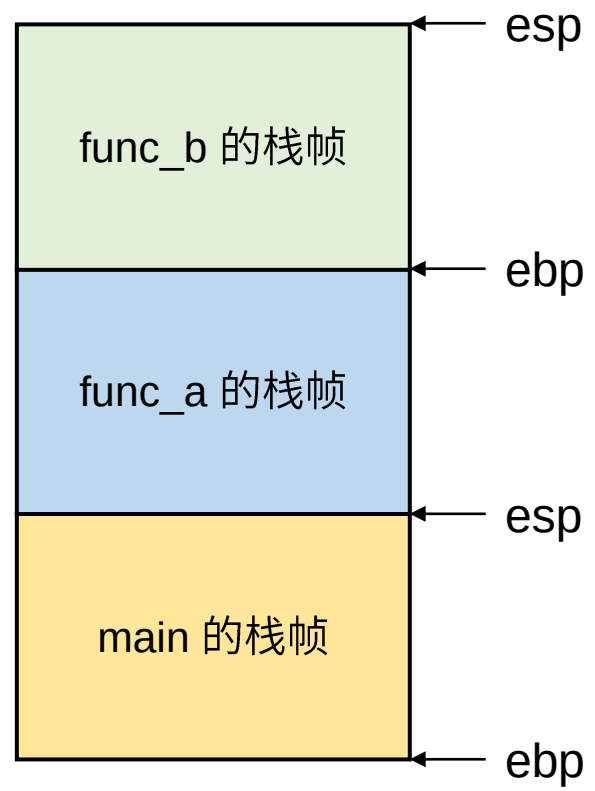
- 栈空间的管理：
  - 当进入函数时被分配
  - 当返回调用函数时被释放

- 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



函数调用

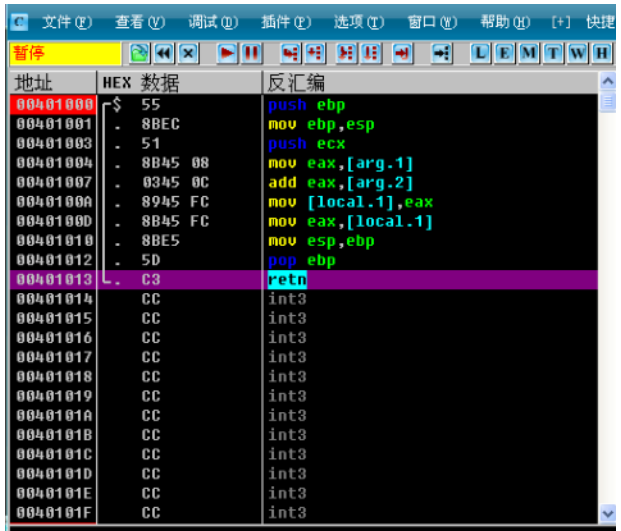


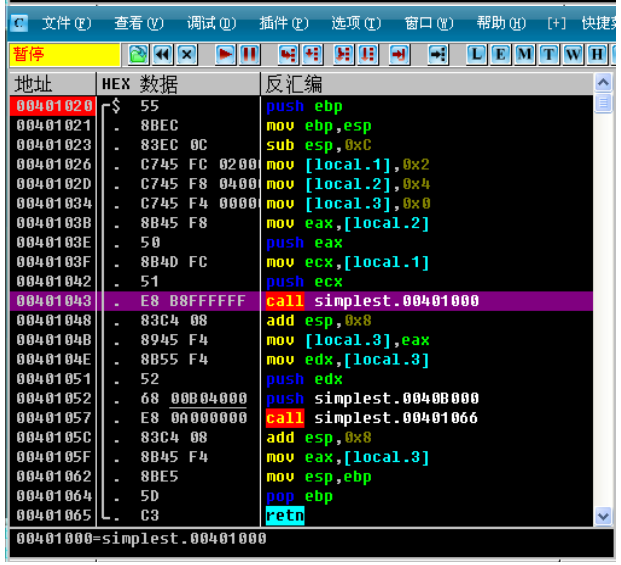


Stack pointer: **esp**

Stack base: **ebp**

函数调用的prologue与epilogue





prologue

```
.text:00401000          push    ebp
.text:00401001          mov     ebp, esp
.text:00401003          push    ecx
.text:00401004          mov     eax, [ebp+arg_0]
.text:00401007          add     eax, [ebp+arg_4]
.text:0040100A          mov     [ebp+var_4], eax
.text:0040100D          mov     eax, [ebp+var_4]
.text:00401010          mov     esp, ebp
.text:00401012          pop     ebp
.text:00401013          retn
.text:00401013 sub_401000 endp
```

epilogue

prologue

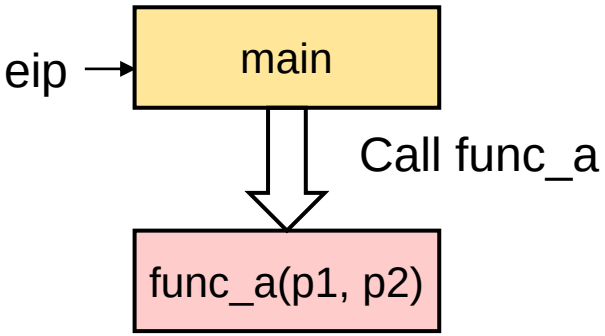
```
.text:00401020          push    ebp
.text:00401021          mov     ebp, esp
.text:00401023          sub     esp, 0Ch
.text:00401026          mov     [ebp+var_4], 2
.text:0040102D          mov     [ebp+var_8], 4
.text:00401034          mov     [ebp+var_C], 0
.text:0040103B          mov     eax, [ebp+var_8]
.text:0040103E          push    eax
.text:0040103F          mov     ecx, [ebp+var_4]
.text:00401042          push    ecx
.text:00401043          call    sub_401000
.text:00401048          add     esp, 8
.text:0040104B          mov     [ebp+var_C], eax
.text:0040104E          mov     edx, [ebp+var_C]
.text:00401051          push    edx
.text:00401052          push    offset aD       ; "%d\n"
.text:00401057          call    _printf
.text:0040105C          add     esp, 8
.text:0040105F          mov     eax, [ebp+var_C]
.text:00401062          mov     esp, ebp
.text:00401064          pop     ebp
.text:00401065          retn
.text:00401065 _main    endp
```

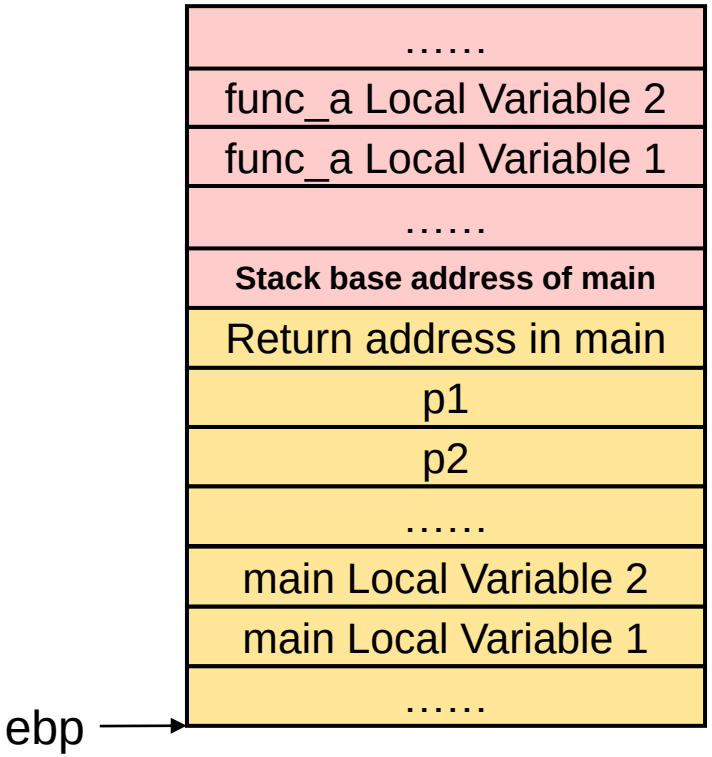
epilogue

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

```c
//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;
}
```

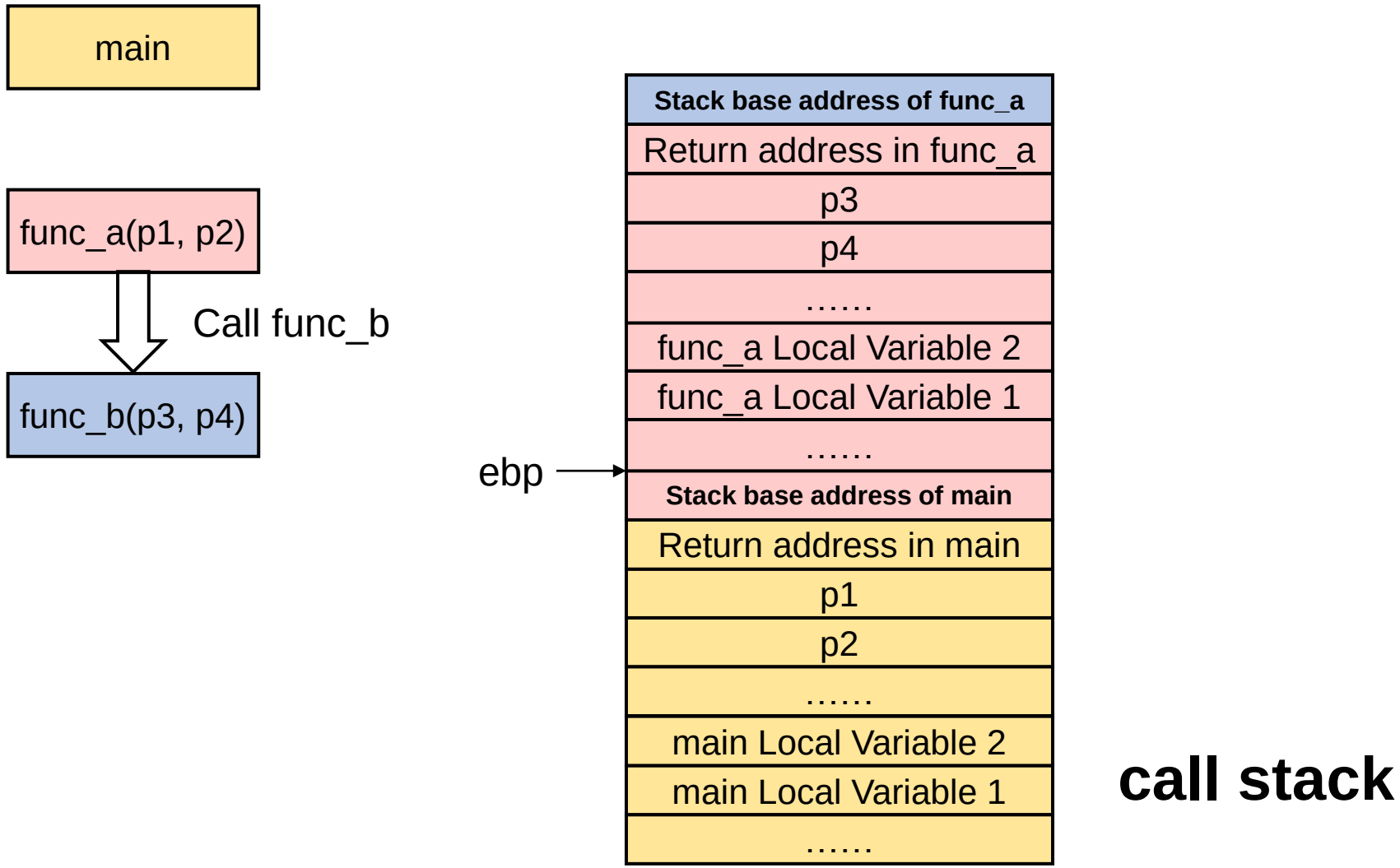
“栈”的运行 Stack





call stack

“栈”的运行 Stack



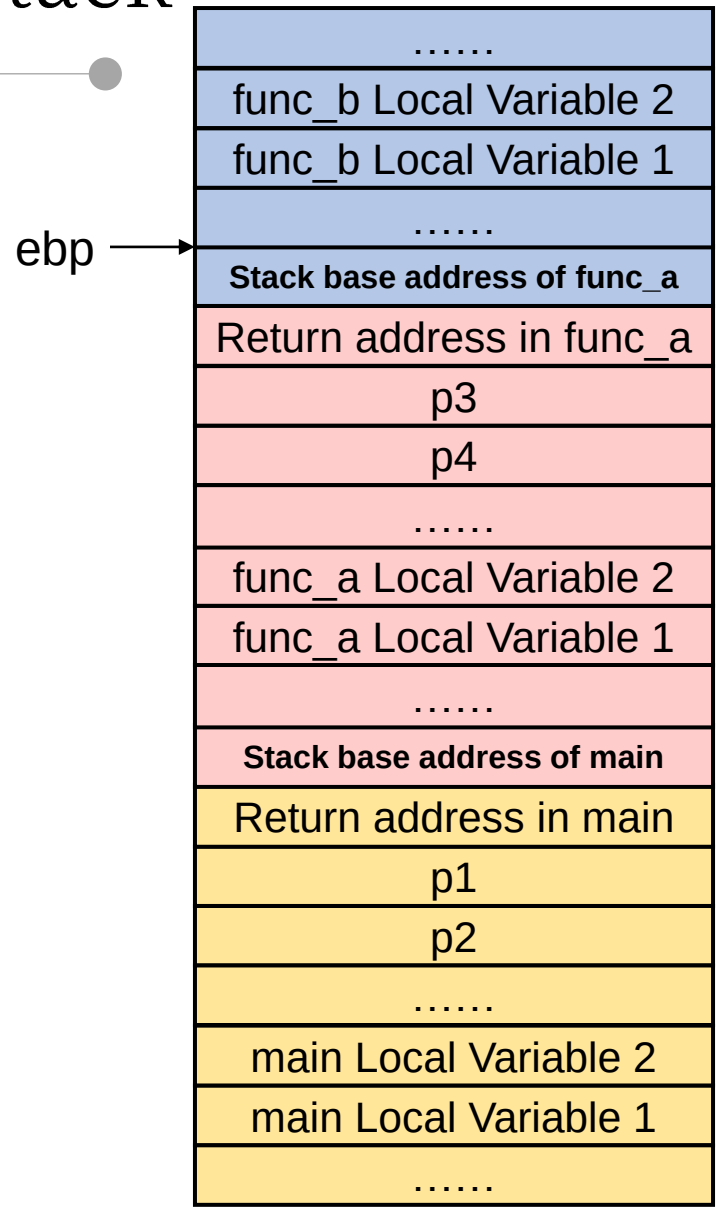
call stack

“栈”的运行 Stack

main

func_a(p1, p2)

func_b(p1, p2)



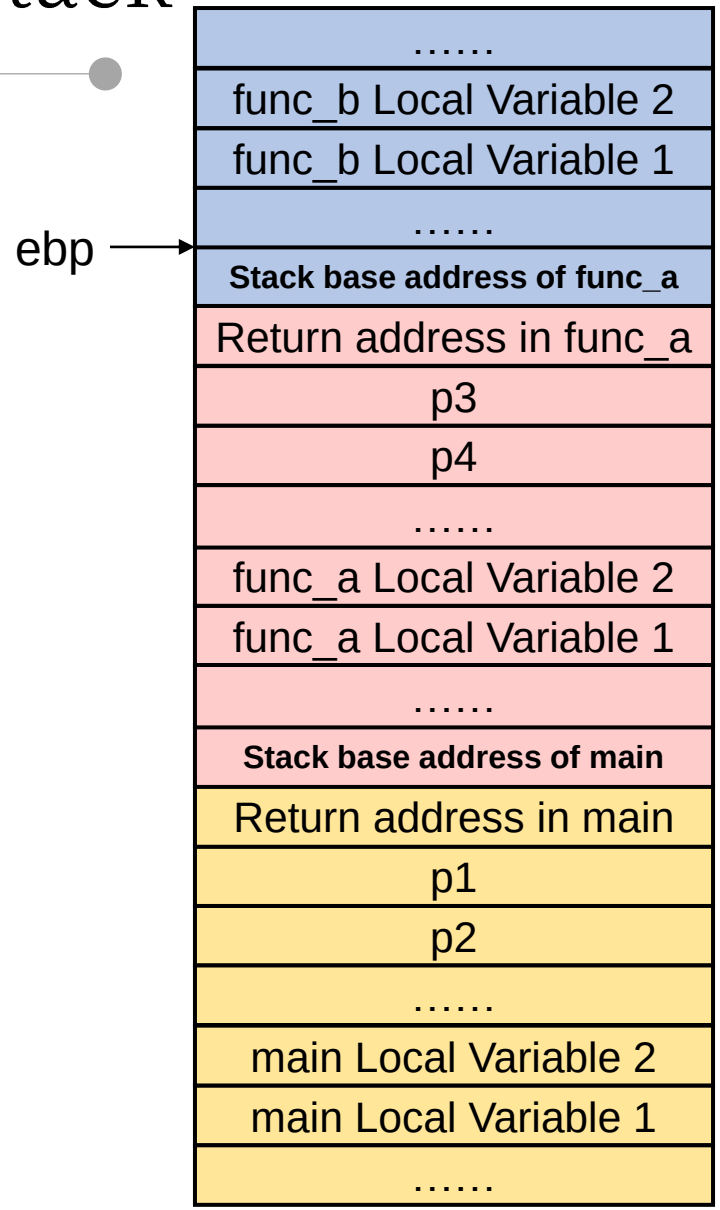
call stack

“栈”的运行 Stack

main

func_a(p1, p2)

func_b(p1, p2)



call stack

观察函数调用的汇编代码 Linux下

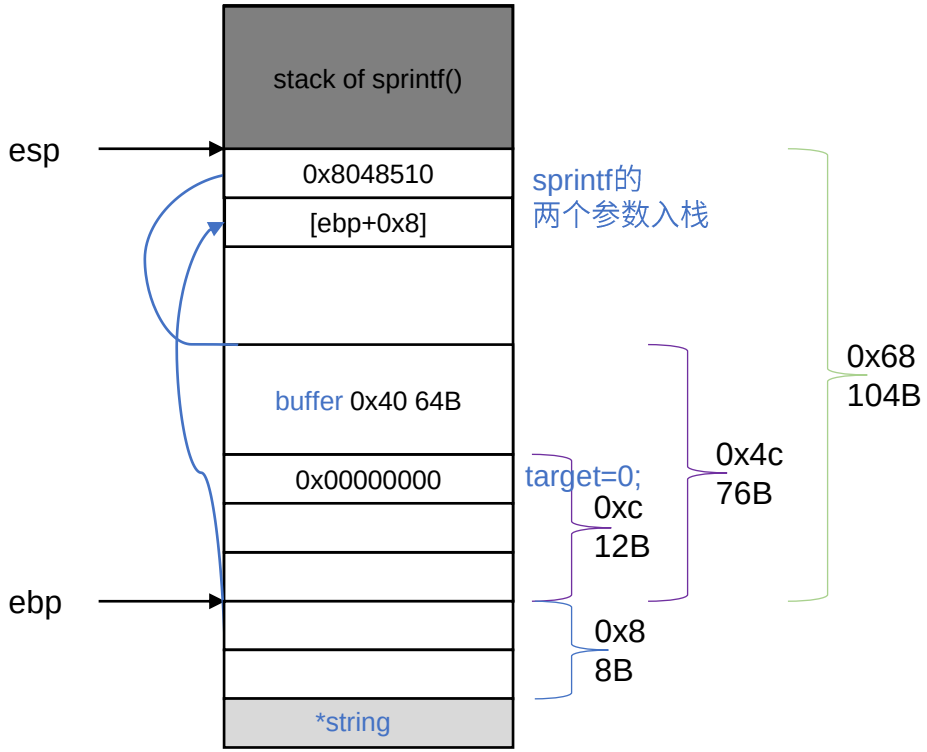
```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

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   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
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,0xfffffff0
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.
```

观察函数调用的汇编代码



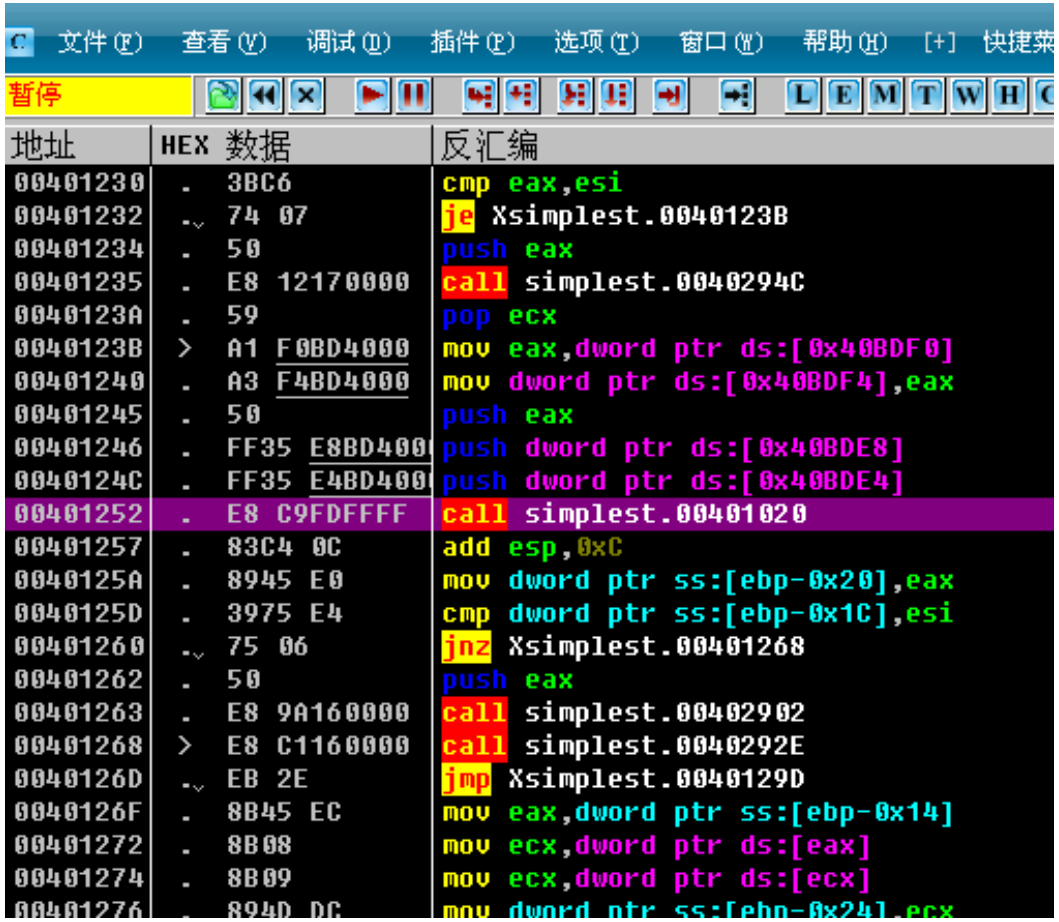

```
(gdb) x/1s 0x08048510
0x8048510:      "you have hit the target correctly :)"
```

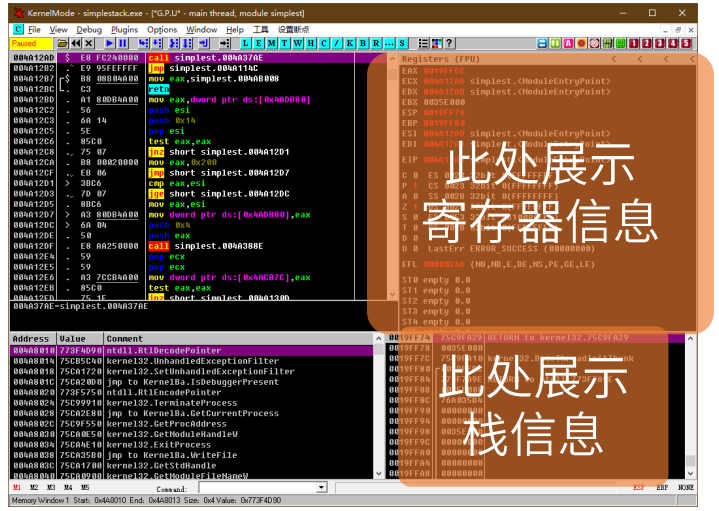
```
Dump of assembler code for function vuln:
0x080483f4 <vuln+0>:     push   ebp
0x080483f5 <vuln+1>:     mov    ebp,esp
0x080483f7 <vuln+3>:     sub    esp,0x68
0x080483fa <vuln+6>:     mov    DWORD PTR [ebp-0xc],0x0
0x08048401 <vuln+13>:    mov    eax,DWORD PTR [ebp+0x8]
0x08048404 <vuln+16>:    mov    DWORD PTR [esp+0x4],eax
0x08048408 <vuln+20>:    lea    eax,[ebp-0x4c]
0x0804840b <vuln+23>:    mov    DWORD PTR [esp],eax
0x0804840e <vuln+26>:    call   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
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,0xfffffff0
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
```





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

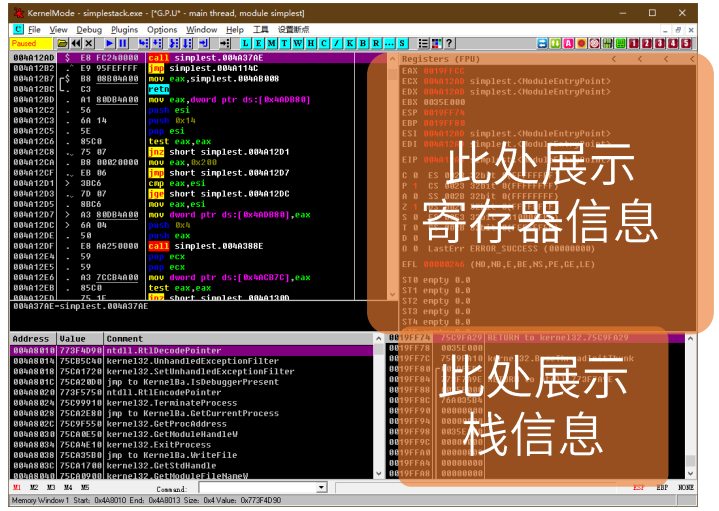
被调用函数返回main函数

```
00401065: C3      retn
```

```
00401062   .  8BE5          mov esp,ebp
00401064   .  5D            pop ebp
00401065   L. 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

```c
//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）

02

“栈”溢出

Stack Overflow

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制复制的数量。

CWE - CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') (4.6) (mitre.org)

“栈溢出”现象：Stack Smash

```c
/*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
```

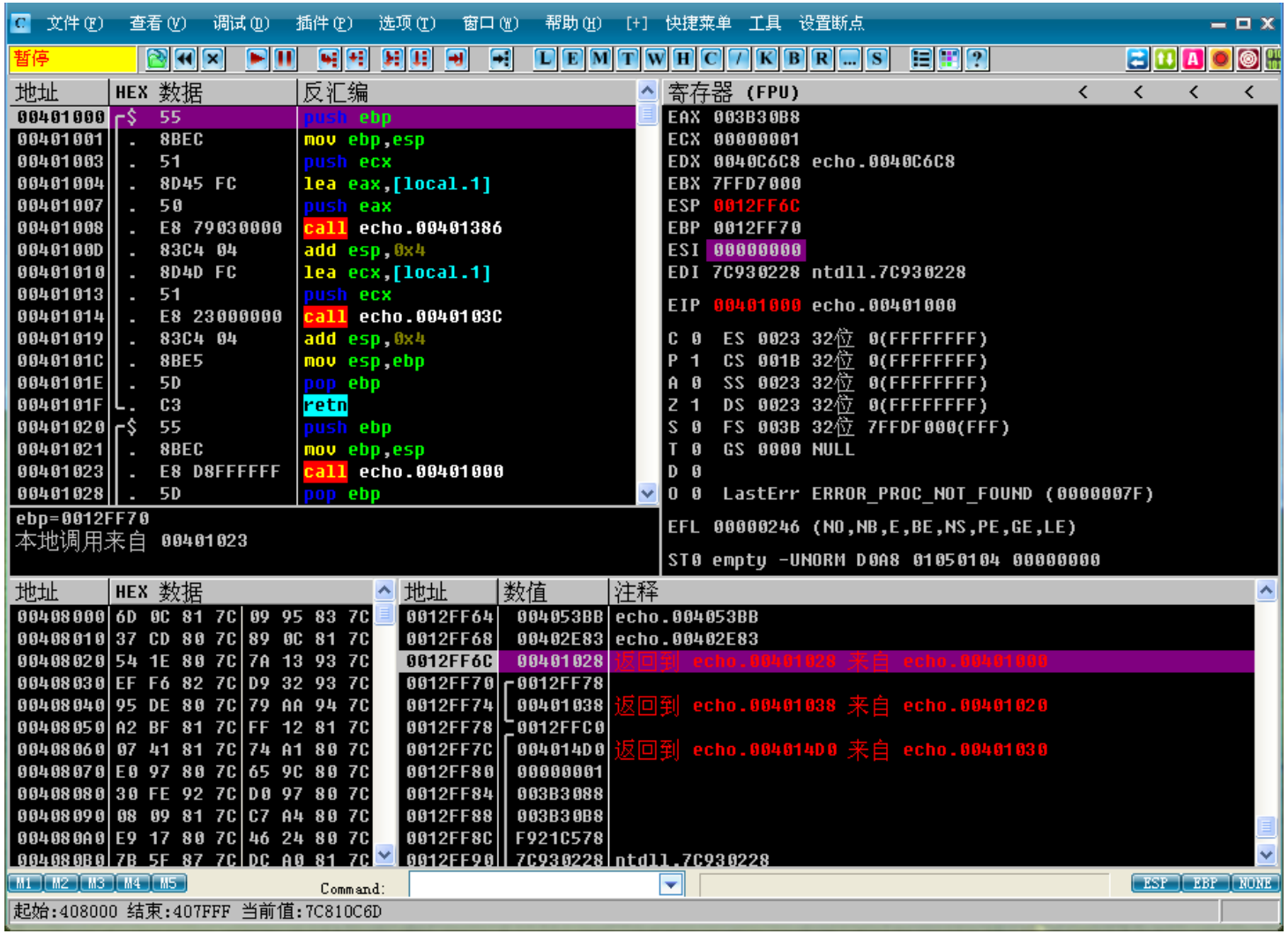
call_echo:

```
.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：

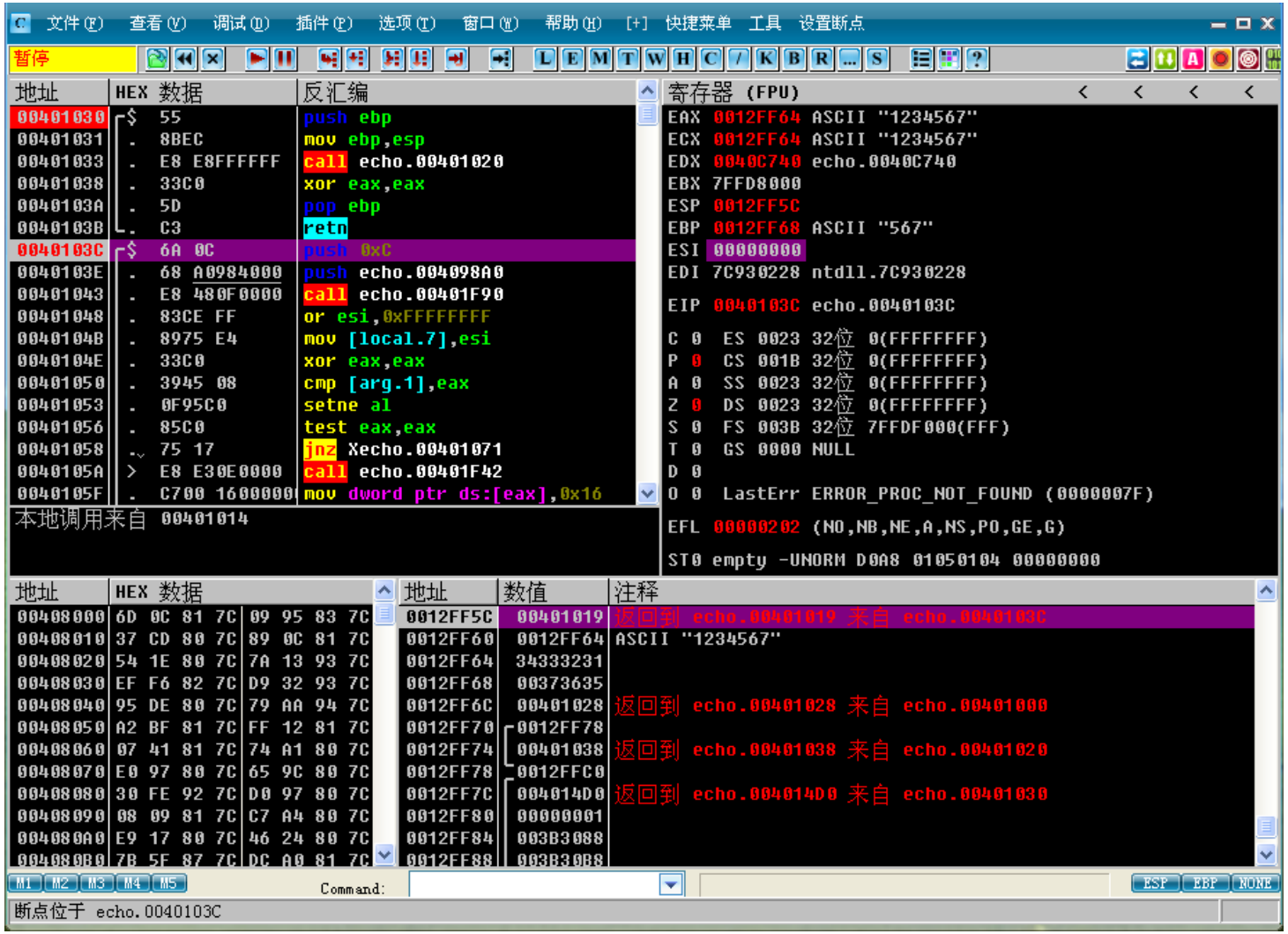
1. 00401000 进入echo()



“栈溢出”现象：Stack Smash

在Ollydbg中查看函数echo()栈帧上的变化：

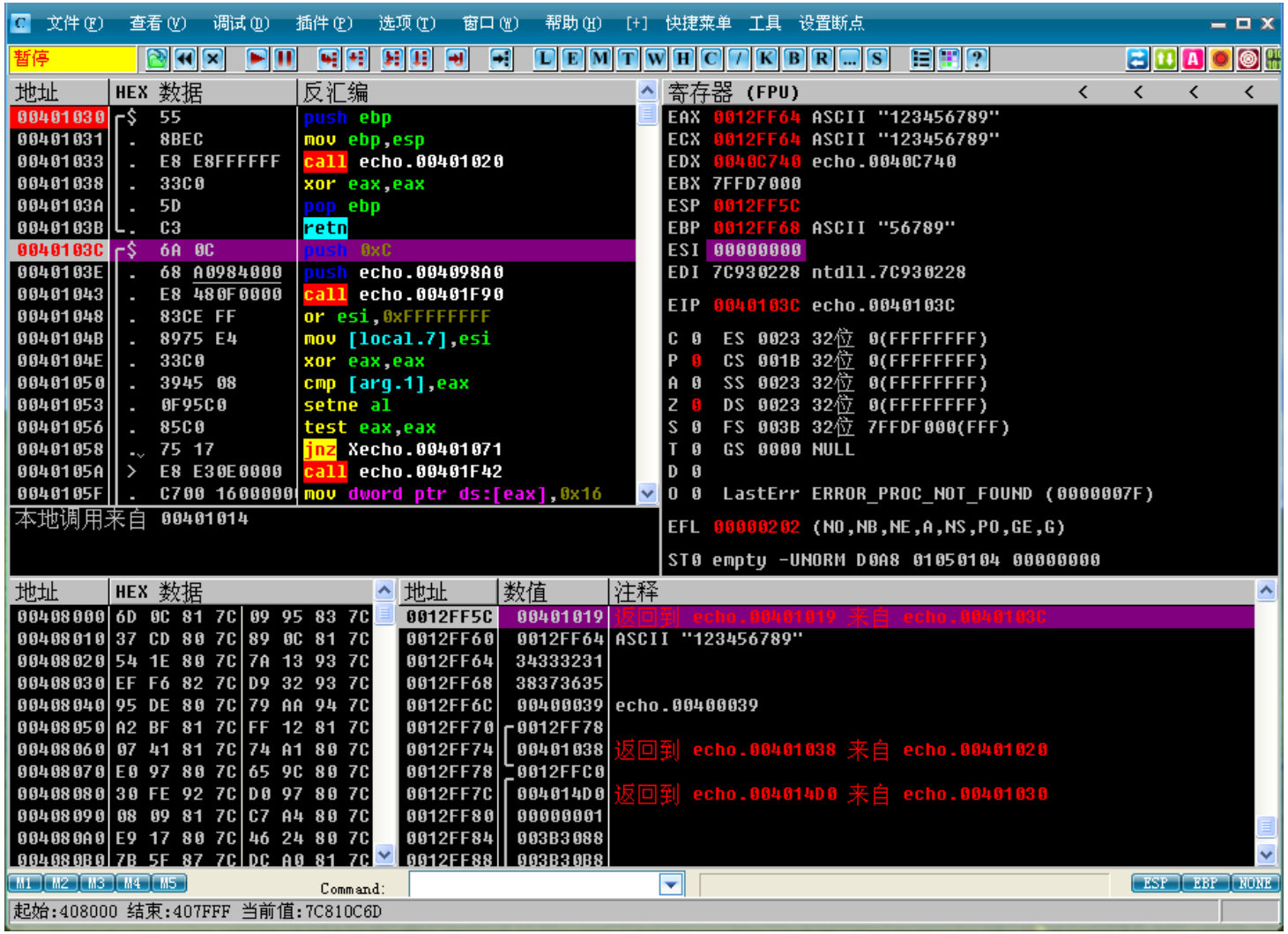
1. 00401000 进入echo()
2. 运行到 gets(buf)，用户输入“1234567”



“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”



“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

```c
#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