函数调用分析

本文所有程序基于GCC-7.5.0编译完成，需要分析的程序源代码为：

```
#include <stdio.h>
int sumall(int a, int b, int c);
void main()
{
    int x,y,z;
    int s;
    x=1;
    y=2;
    z=3;
    s = sumall(x,y,z);
    printf("s=%d\n", s);
}
int sumall(int a, int b, int c)
{
    int res;
    res=a+b+c;
    return res;
}
```

分析的函数调用过程是`sumall`函数

需要注意的一点是，如果需要编译出32位使用`push`传参的程序，`gcc`需要添加`-fno-stack-protector`参数，关闭栈保护措施，完整的编译指令

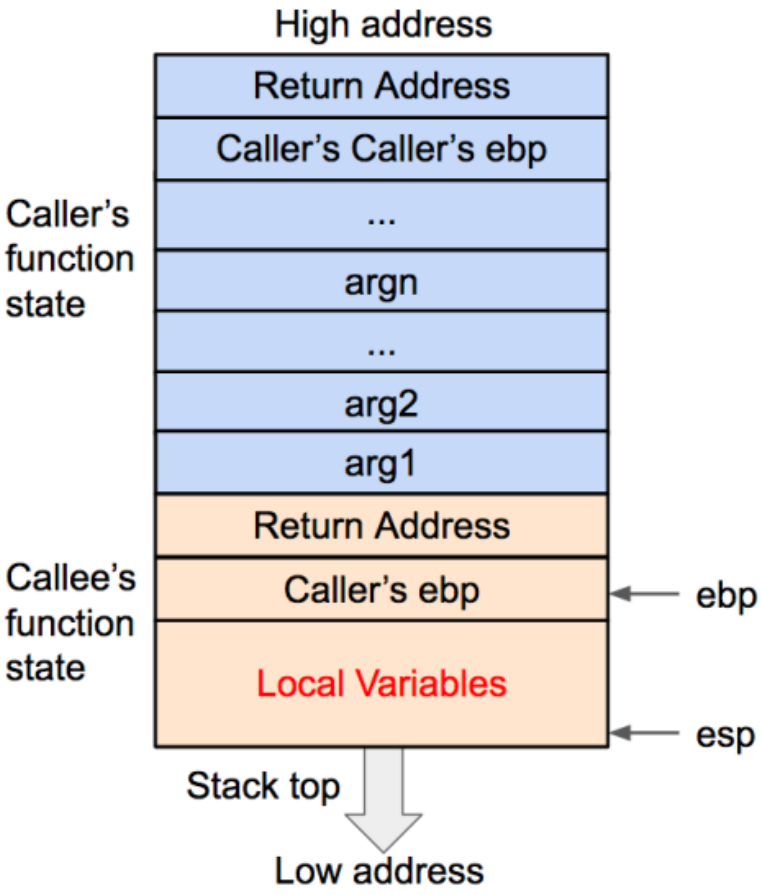
```
gcc -m32 -fno-stack-protector -g 01.c -o 1
```

然后因为操作系统均开启了动态偏移加载保护，所有静态分析的地址都是相对地址

Linux_X86函数调用分析

理论分析

函数参数在函数返回地址的上方，一个完整的Linux_x86的函数调用栈帧如图所示



经查资料Linux的GCC采用的是cdecl的调用约定，参数从右向左入栈，主调函数负责栈平衡。

静态分析

函数状态主要涉及三个寄存器esp，ebp，eip。esp 用来存储函数调用栈的栈顶地址，在压栈和退栈时发生变化。ebp 用来存储当前函数状态的基地址，在函数运行时不变，可以用来索引确定函数参数或局部变量的位置。eip 用来存储即将执行的程序指令的地址，cpu 依照 eip 的存储内容读取指令并执行，eip 随之指向相邻的下一条指令，如此反复，程序就得以连续执行指令。

首先将被调用函数的参数按照逆序依次压入栈内。如果被调用函数不需要参数，则没有这一步骤。这些参数仍会保存在调用函数的函数状态内，之后压入栈内的数据都会作为被调用函数的函数状态来保存。

使用Ghidra反汇编引擎获得如下汇编代码：

```
.text:0000051D main             proc near               ; DATA XREF:
.got:main_ptr↓o
.text:0000051D
.text:0000051D s                = dword ptr -18h
.text:0000051D z                = dword ptr -14h
.text:0000051D y                = dword ptr -10h
.text:0000051D x                = dword ptr -0Ch
.text:0000051D argc             = dword ptr  8
.text:0000051D argv             = dword ptr  0Ch
.text:0000051D envp             = dword ptr  10h
.text:0000051D
.text:0000051D ; __unwind {
.text:0000051D                  lea     ecx, [esp+4]
.text:00000521                  and     esp, 0FFFFFFF0h
.text:00000524                  push    dword ptr [ecx-4]
.text:00000527                  push    ebp
.text:00000528                  mov     ebp, esp
.text:0000052A                  push    ebx
```

```
.text:0000052B                 push    ecx
.text:0000052C                 sub     esp, 10h
.text:0000052F                 call    __x86_get_pc_thunk_bx
.text:00000534                 add     ebx, 1AA4h
.text:0000053A                 mov     [ebp+x], 1
.text:00000541                 mov     [ebp+y], 2
.text:00000548                 mov     [ebp+z], 3
.text:0000054F                 sub     esp, 4
.text:00000552                 push    [ebp+z]         ; c
.text:00000555                 push    [ebp+y]         ; b
.text:00000558                 push    [ebp+x]         ; a
.text:0000055B                 call    sumall
.text:00000560                 add     esp, 10h
.text:00000563                 mov     [ebp+s], eax
.text:00000566                 sub     esp, 8
.text:00000569                 push    [ebp+s]
.text:0000056C                 lea     eax, (aSD - 1FD8h)[ebx] ; "s=%d\n"
.text:00000572                 push    eax             ; format
.text:00000573                 call    _printf
.text:00000578                 add     esp, 10h
.text:0000057B                 nop
.text:0000057C                 lea     esp, [ebp-8]
.text:0000057F                 pop     ecx
.text:00000580                 pop     ebx
.text:00000581                 pop     ebp
.text:00000582                 lea     esp, [ecx-4]
.text:00000585                 retn
.text:00000585 ; } // starts at 51D
.text:00000585 main            endp

.text:00000586 ; int __cdecl sumall(int a, int b, int c)
.text:00000586                 public sumall
.text:00000586 sumall          proc near               ; CODE XREF: main+3E↑p
.text:00000586
.text:00000586 res             = dword ptr -4
.text:00000586 a               = dword ptr  8
.text:00000586 b               = dword ptr  0Ch
.text:00000586 c               = dword ptr  10h
.text:00000586
.text:00000586 ; __unwind {
.text:00000586                 push    ebp
.text:00000587                 mov     ebp, esp
.text:00000589                 sub     esp, 10h
.text:0000058C                 call    __x86_get_pc_thunk_ax
.text:00000591                 add     eax, 1A47h
.text:00000596                 mov     edx, [ebp+a]
.text:00000599                 mov     eax, [ebp+b]
.text:0000059C                 add     edx, eax
.text:0000059E                 mov     eax, [ebp+c]
.text:000005A1                 add     eax, edx
.text:000005A3                 mov     [ebp+res], eax
.text:000005A6                 mov     eax, [ebp+res]
.text:000005A9                 leave
.text:000005AA                 retn
.text:000005AA ; } // starts at 586
.text:000005AA sumall          endp
```

这里主要分析`call sumall`前后的汇编代码，

```
//main函数
.text:00000552                 push    [ebp+z]         ; c
.text:00000555                 push    [ebp+y]         ; b
.text:00000558                 push    [ebp+x]         ; a
.text:0000055B                 call    sumall
.text:00000560                 add     esp, 10h

//sumall函数里
.text:00000586                 push    ebp
.text:00000587                 mov     ebp, esp
.text:00000589                 sub     esp, 10h
```

这里开始栈帧分析

首先是main函数的栈帧

Low address	
Caller's Caller's ebp(main ebp)	**esp/ebp**
Return Address	
High address	

`push    [ebp+z]         ; c`

Low address	
C	**esp**
Caller's Caller's ebp(main ebp)	**ebp**
Return Address	
High address	

`push    [ebp+y]         ; b`

Low address	
B	**esp**
C	
Caller's Caller's ebp(main ebp)	**ebp**
Return Address	
High address	

`push    [ebp+x]         ; a`

Low address	
A	esp
B	
C	
Caller's Caller's ebp(main ebp)	ebp
Return Address	
High address	

`call sumall`后开始分析，这里系统会隐式的执行`push Return Address`，就是会把执行call指令后下一条指令地址压入栈，这里就是`00000560 add esp, 10h`，也就是用来清理栈帧的命令

Low address	
Return Address(0x00000560)	esp
A	
B	
C	ebp
Caller's Caller's ebp(main ebp)	
Return Address	
High address	

`push ebp`

Low address	
Caller's ebp	esp
Return Address(0x00000560)	
A	
B	
C	
Caller's Caller's ebp(main ebp)	ebp
Return Address	
High address	

`mov ebp, esp`再将当前的ebp寄存器的值压入栈内，并将 ebp 寄存器的值更新为当前栈顶的地址。这样调用函数的 ebp信息得以保存。同时，ebp 被更新为被调用函数的基地址。

Low address	
Caller's ebp	ebp / esp
Return Address(0x00000560)	
A	
B	
C	
Caller's Caller's ebp(main ebp)	
Return Address	
High address	

此时就完成了对sumall函数的调用工作，当sumll函数结束之后，需要平衡栈帧，具体的工作是：

`leave`指令，其实是`mov esp,ebp`,`pop ebp`,假如sumll函数内有临时变量，则先恢复栈顶指针esp至ebp位置，之后将**Caller's ebp**的值弹出至`ebp`，恢复**main**函数的栈基地址

Low address	
Return Address(0x00000560)	esp
A	
B	
C	
Caller's Caller's ebp(main ebp)	ebp
Return Address	
High address	

`retn`指令其实就是`pop eip`，相当于恢复执行顺序，`eip`指向`call sumll`之后的地址

Low address	
A	esp
B	
C	
Caller's Caller's ebp(main ebp)	ebp
Return Address	
High address	

`add esp, 10h`，相当于抬高esp恢复栈顶，为什么是10h，因为是32位程序 4 * 4 = 10h，此时栈帧恢复完全

Low address	
A	
B	
C	
Caller's Caller's ebp(main ebp)	esp/ebp
Return Address	
High address	

动态验证

动态验证使用GDB + gdbdbg插件完成分析

```
▶ 0x56555552 <main+53>    push   dword ptr [ebp - 0x14]
```

```
pwndbg> stack 5
00:0000| esp  0xffffd08c —▸ 0x56555534 (main+23) ◂— add    ebx, 0x1aa4
01:0004|      0xffffd090 ◂— 0x1
02:0008|      0xffffd094 ◂— 0x3
03:000c|      0xffffd098 ◂— 0x2
04:0010|      0xffffd09c ◂— 0x1
```

执行完毕后

```
pwndbg> stack 5
00:0000| esp  0xffffd088 ◂— 0x3
01:0004|      0xffffd08c —▸ 0x56555534 (main+23) ◂— add    ebx, 0x1aa4
02:0008|      0xffffd090 ◂— 0x1
03:000c|      0xffffd094 ◂— 0x3
04:0010|      0xffffd098 ◂— 0x2
```

一路执行至

```
▶ 0x5655555b <main+62>    call   sumall <0x56555586>
        arg[0]: 0x1
        arg[1]: 0x2
        arg[2]: 0x3
```

此时栈结构

```
pwndbg> stack 5
00:0000| esp  0xffffd080 ◂— 0x1
01:0004|      0xffffd084 ◂— 0x2
02:0008|      0xffffd088 ◂— 0x3
```

步入操作，执行至`▶ 0x56555586 <sumall>       push   ebp`

此时观察寄存器的值

```
 EAX  0xf7fb4dd8 (environ) —▸ 0xffffd15c —▸ 0xffffd310 ◂—
'CLUTTER_IM_MODULE=xim'
 EBX  0x56556fd8 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ee0
 ECX  0xffffd0c0 ◂— 0x1
 EDX  0xffffd0e4 ◂— 0x0
 EDI  0x0
 ESI  0xf7fb3000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d7d6c
 EBP  0xffffd0a8 ◂— 0x0
 ESP  0xffffd07c —▸ 0x56555560 (main+67) ◂— add    esp, 0x10
 EIP  0x56555586 (sumall) ◂— push   ebp
```

执行完后观察栈，ebp被压入栈

```
pwndbg> stack 5
00:0000| esp  0xffffd078 —▸ 0xffffd0a8 ◂— 0x0
01:0004|      0xffffd07c —▸ 0x56555560 (main+67) ◂— add    esp, 0x10
02:0008|      0xffffd080 ◂— 0x1
03:000c|      0xffffd084 ◂— 0x2
04:0010|      0xffffd088 ◂— 0x3
```

执行完`0x56555587 <sumall+1>     mov    ebp, esp`，观察寄存器，ebp已经被修改完现在的值

```
EAX  0xf7fb4dd8 (environ) —▸ 0xffffd15c —▸ 0xffffd310 ◂— 'CLUTTER_IM_MODULE=xim'
 EBX  0x56556fd8 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ee0
 ECX  0xffffd0c0 ◂— 0x1
 EDX  0xffffd0e4 ◂— 0x0
 EDI  0x0
 ESI  0xf7fb3000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d7d6c
 EBP  0xffffd078 —▸ 0xffffd0a8 ◂— 0x0
 ESP  0xffffd078 —▸ 0xffffd0a8 ◂— 0x0
 EIP  0x56555589 (sumall+3) ◂— sub    esp, 0x10
```

现在开始验证函数调用完毕的操作

执行至`▶ 0x565555a9 <sumall+35>    leave`，先观察一下寄存器的值

```
 EAX  0x6
 EBX  0x56556fd8 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ee0
 ECX  0xffffd0c0 ◂— 0x1
 EDX  0x3
 EDI  0x0
 ESI  0xf7fb3000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d7d6c
 EBP  0xffffd078 —▸ 0xffffd0a8 ◂— 0x0
 ESP  0xffffd068 —▸ 0xf7e0b4a9 (__new_exitfn+9) ◂— add    ebx, 0x1a7b57
 EIP  0x565555a9 (sumall+35) ◂— leave
```

执行完毕后，esp和ebp的值均恢复

```
EAX  0x6
EBX  0x56556fd8 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ee0
ECX  0xffffd0c0 ◂— 0x1
EDX  0x3
EDI  0x0
ESI  0xf7fb3000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d7d6c
EBP  0xffffd0a8 ◂— 0x0
ESP  0xffffd07c —▸ 0x56555560 (main+67) ◂— add    esp, 0x10
EIP  0x565555aa (sumall+36) ◂— ret
```

然后执行完`► 0x565555aa <sumall+36>    ret             <0x56555560; main+67>`，eip值已经恢复

```
EAX  0x6
EBX  0x56556fd8 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1ee0
ECX  0xffffd0c0 ◂— 0x1
EDX  0x3
EDI  0x0
ESI  0xf7fb3000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d7d6c
EBP  0xffffd0a8 ◂— 0x0
ESP  0xffffd080 ◂— 0x1
EIP  0x56555560 (main+67) ◂— add    esp, 0x10
```

执行完`0x56555560 <main+67>      add    esp, 0x10`后，栈帧恢复

```
pwndbg> stack 5
00:0000| esp  0xffffd090 ◂— 0x1
01:0004|      0xffffd094 ◂— 0x3
02:0008|      0xffffd098 ◂— 0x2
03:000c|      0xffffd09c ◂— 0x1
04:0010|      0xffffd0a0 —▸ 0xffffd0c0 ◂— 0x1
```

此时函数调用全过程分析完毕

Linux_X64函数调用分析

理论分析

在 64 位程序中，函数的前 6 个参数是通过寄存器传递的，System V AMD64 ABI (Linux、FreeBSD、macOS 等采用) 中前六个整型或指针参数依次保存在 **RDI, RSI, RDX, RCX, R8 和 R9 寄存器**中，如果还有更多的参数的话才会保存在栈上。

静态分析

```
.text:000000000000064A main            proc near               ; DATA XREF:
_start+1D↑o
.text:000000000000064A
.text:000000000000064A x               = dword ptr -10h
.text:000000000000064A y               = dword ptr -0Ch
.text:000000000000064A z               = dword ptr -8
.text:000000000000064A s               = dword ptr -4
.text:000000000000064A
.text:000000000000064A ; __unwind {
.text:000000000000064A                 push    rbp
```

```
.text:000000000000064B                 mov     rbp, rsp
.text:000000000000064E                 sub     rsp, 10h
.text:0000000000000652                 mov     [rbp+x], 1
.text:0000000000000659                 mov     [rbp+y], 2
.text:0000000000000660                 mov     [rbp+z], 3
.text:0000000000000667                 mov     edx, [rbp+z]    ; c
.text:000000000000066A                 mov     ecx, [rbp+y]
.text:000000000000066D                 mov     eax, [rbp+x]
.text:0000000000000670                 mov     esi, ecx        ; b
.text:0000000000000672                 mov     edi, eax        ; a
.text:0000000000000674                 call    sumall
.text:0000000000000679                 mov     [rbp+s], eax
.text:000000000000067C                 mov     eax, [rbp+s]
.text:000000000000067F                 mov     esi, eax
.text:0000000000000681                 lea     rdi, format     ; "s=%d\n"
.text:0000000000000688                 mov     eax, 0
.text:000000000000068D                 call    _printf
.text:0000000000000692                 nop
.text:0000000000000693                 leave
.text:0000000000000694                 retn
.text:0000000000000694 ; } // starts at 64A
.text:0000000000000694 main            endp

.text:0000000000000695 sumall          proc near               ; CODE XREF:
main+2A↑p
.text:0000000000000695
.text:0000000000000695 c               = dword ptr -1Ch
.text:0000000000000695 b               = dword ptr -18h
.text:0000000000000695 a               = dword ptr -14h
.text:0000000000000695 res             = dword ptr -4
.text:0000000000000695
.text:0000000000000695 ; __unwind {
.text:0000000000000695                 push    rbp
.text:0000000000000696                 mov     rbp, rsp
.text:0000000000000699                 mov     [rbp+a], edi
.text:000000000000069C                 mov     [rbp+b], esi
.text:000000000000069F                 mov     [rbp+c], edx
.text:00000000000006A2                 mov     edx, [rbp+a]
.text:00000000000006A5                 mov     eax, [rbp+b]
.text:00000000000006A8                 add     edx, eax
.text:00000000000006AA                 mov     eax, [rbp+c]
.text:00000000000006AD                 add     eax, edx
.text:00000000000006AF                 mov     [rbp+res], eax
.text:00000000000006B2                 mov     eax, [rbp+res]
.text:00000000000006B5                 pop     rbp
.text:00000000000006B6                 retn
.text:00000000000006B6 ; } // starts at 695
.text:00000000000006B6 sumall          endp
```

这里主要分析`call    sumall`前后的汇编代码，

```
.text:0000000000000667                mov     edx, [rbp+z]    ; c
.text:000000000000066A                mov     ecx, [rbp+y]
.text:000000000000066D                mov     eax, [rbp+x]
.text:0000000000000670                mov     esi, ecx        ; b
.text:0000000000000672                mov     edi, eax        ; a
.text:0000000000000674                call    sumall

.text:0000000000000695                push    rbp
.text:0000000000000696                mov     rbp, rsp
```

就可以很清晰的看到，利用的是**RDI, RSI, RDX**传参，而不再是利用栈结构传参，只有参数大于一定数目的时候才会利用栈结构传参，若利用栈结构传参，则和32位环境下一致，在此不做过多赘述

动态验证

执行至`► 0x555555554667 <main+29>    mov    edx, dword ptr [rbp - 8]`，执行完毕后观察寄存器的值，RDX = 3

```
RAX  0x55555555464a (main) ◂— push   rbp
 RBX  0x0
 RCX  0x5555555546c0 (__libc_csu_init) ◂— push   r15
 RDX  0x3
 RDI  0x1
 RSI  0x7fffffffdfd8 —▸ 0x7fffffffe304 ◂— '/home/syc/2'
 R8   0x7ffff7dd0d80 (initial) ◂— 0x0
 R9   0x7ffff7dd0d80 (initial) ◂— 0x0
 R10  0x1
 R11  0x0
 R12  0x555555554540 (_start) ◂— xor    ebp, ebp
 R13  0x7fffffffdfd0 ◂— 0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffdef0 —▸ 0x5555555546c0 (__libc_csu_init) ◂— push   r15
 RSP  0x7fffffffdee0 ◂— 0x200000001
 RIP  0x55555555466a (main+32) ◂— mov    ecx, dword ptr [rbp - 0xc]
```

继续执行完相关，直至

```
   0x55555555466a <main+32>    mov    ecx, dword ptr [rbp - 0xc]
   0x55555555466d <main+35>    mov    eax, dword ptr [rbp - 0x10]
   0x555555554670 <main+38>    mov    esi, ecx
   0x555555554672 <main+40>    mov    edi, eax
 ► 0x555555554674 <main+42>    call   sumall <0x555555554695>
        rdi: 0x1
        rsi: 0x2
        rdx: 0x3
```

观察寄存器的值

```
 RAX  0x1
 RBX  0x0
 RCX  0x2
 RDX  0x3
 RDI  0x1
 RSI  0x2
```

```
R8   0x7ffff7dd0d80 (initial) ◂— 0x0
R9   0x7ffff7dd0d80 (initial) ◂— 0x0
R10  0x1
R11  0x0
R12  0x555555554540 (_start) ◂— xor    ebp, ebp
R13  0x7fffffffdfd0 ◂— 0x1
R14  0x0
R15  0x0
RBP  0x7fffffffdef0 —▸ 0x5555555546c0 (__libc_csu_init) ◂— push   r15
RSP  0x7fffffffdee0 ◂— 0x200000001
RIP  0x555555554674 (main+42) ◂— call   0x555555554695
```

RDI = 0x1 , RSI = 0x2 , RDX = 0x3 ， 所有的参数布置完毕，之后调用结束的过程和32位大体一致，EIP还是需要被压栈

```
pwndbg> stack 5
00:0000| rsp  0x7fffffffded8 —▸ 0x555555554679 (main+47) ◂— mov    dword ptr
[rbp - 4], eax
01:0008|      0x7fffffffdee0 ◂— 0x200000001
02:0010|      0x7fffffffdee8 ◂— 0x3
03:0018| rbp  0x7fffffffdef0 —▸ 0x5555555546c0 (__libc_csu_init) ◂— push   r15
04:0020|      0x7fffffffdef8 —▸ 0x7ffff7a05b97 (__libc_start_main+231) ◂— mov
edi, eax
```

其他大体一致

Windows函数调用分析

其实和Linux差距不大，也就是默认为stdcall，参数从右向左入栈，被调函数负责栈平衡。