贪吃蛇
反汇编代码分析报告

THINCT

December 10, 2023

SnakeGame

EBX 代替当前的函数栈底

```
004079C0    push    ebx
004079C1    mov     ebx, esp
004079C3    sub     esp, 8
004079C6    and     esp, -8
004079C9    add     esp, 4
004079CC    push    ebp
004079CD    mov     ebp, [ebx+4]
004079D0    mov     [esp+4], ebp
004079D4    mov     ebp, esp
```

1. 当 eip 在.text:004079C0 处，esp 所指向的是 ret addr.
2. 当 eip 在.text:004079C1 处，ebx 所指向的是 esp-4. 此时:
   - ebx+4 指向的是 ret addr
   - ebx+8 指向的是第一个参数

EBX 代替当前的函数栈底

```
004079C3    sub     esp, 8
004079C6    and     esp, 0FFFFFFF8h
004079C9    add     esp, 4
004079CC    push    ebp
```

esp 实现了向下最近的 8 的倍数取证。比如 12 取整就是 8，16 取整就是 16，18 取整就是 16. 因为是针对栈结构地址取整，所以越是往小的方向越安全，因为对于栈结构来讲，越小的地址是没有用过的地址。所以后面的 ebp,esp, ebp 只能作为局部变量的索引，而对于参数的索引，用 ebx 比较合适。

总结:

对于这个函数来讲，并不是按照套路 *ebp* 作为局部变量和函数参数的唯一参考.

operator += 传参

```
004079FF    mov     eax, [ebx+8]
00407A02    mov     ecx, [eax+4]
00407A05    push    ecx
00407A06    mov     edx, [eax]
00407A08    push    edx
00407A09    mov     eax, [ebp-2Ch]
00407A0C    add     eax, 28h ; '('
00407A0F    push    eax
00407A10    call    sf::operator+=(sf::Time
    &,sf::Time)
```

- 从 0x00407A09 到 0x00407A0F 是第一个参数, 已知 [ebp-2Ch] 为 this, 所以第一个参数为 this->offset28h, 并且为 sf::Time 引用类型. 所以 ***sf::Time* this->offset28h***.

operator += 传参

```
004079FF    mov     eax, [ebx+8]
00407A02    mov     ecx, [eax+4]
00407A05    push    ecx
00407A06    mov     edx, [eax]
00407A08    push    edx
00407A09    mov     eax, [ebp-2Ch]
00407A0C    add     eax, 28h ; '('
00407A0F    push    eax
00407A10    call    sf::operator+=(sf::Time
    &,sf::Time)
```

- 0x004079FF 已推导出为当前函数的第一个参数, 而 0x00407A02 到 0x00407A08 是连续的内存, 从 call 得知这个连续的内存是 sf::Time 类型, 所以推导出 [ebx+8] 是 sf::Time* 类型, 即 *sf::Time* [ebx+8]*

operator += 传参

总结:

operator += 第一个参数是传地址, 第二个参数是传值, 只不过 sf::Time 的内存是 8 个字节, 所以从起始地址连续压栈 2 次. 本重载函数主要需要掌握的是: **不能根据 *push* 来判断函数的参数个数**.

函数的返回值才是第一个参数

```
00407A19 mov       ecx, [ebp-2Ch]
00407A1C movss     xmm0, ds:__real@3f800000
00407A24 divss     xmm0, dword ptr [ecx+24h]
00407A29 push      ecx
00407A2A movss     dword ptr [esp], xmm0
00407A2F lea       edx, [ebp-28h]
00407A32 push      edx
00407A33 call
    ds:__imp_?seconds@sf@@YA?AVTime@1@M@Z ;
    sf::seconds(float)
```

0x00407A2F 处压栈的是第一个参数, 为局部变量 (**暂存临时返回值**),0x00407A29 和 0x00407A2A 压栈第二个参数, 其中 push 只是占位作用,0x00407A2A 才是第二个参数的值, 也就是计算出来的浮点数.

函数的返回值才是第一个参数

```
7AC94C90     movss     xmm0,dword ptr [esp+8]
7AC94C96     mulss     xmm0,dword ptr
   ds:[7AC982B8h]
7AC94C9E     call      7AC9600E
7AC94CA3     mov         ecx,dword ptr [esp+4]
7AC94CA7     mov         dword ptr [ecx],eax
7AC94CA9     mov         eax,ecx
7AC94CAB     mov         dword ptr [ecx+4],edx
7AC94CAE     ret
```

观察调用的函数, 分别从 0x7AC94CA3 和 0x7AC94CA9 可知: 第一个参数也是该函数的返回值, 所以可以推断出: **函数的返回值才是第一个参数, 并且该函数其实只有一个参数**, 即 0x00407A2A 处压栈的参数.

参数与函数可能隔了几个 call

```
00407B1A    push        1 ; includesHead
00407B1C    mov         ecx, [ebp-2Ch]
00407B1F    add         ecx, 1408h
00407B25    call        sf::Transformable::getPosit
00407B2B    push        eax ; location
00407B2C    mov         ecx, [ebp-2Ch]
00407B2F    add         ecx, 30h ; this
00407B32    call        Snake::collidesWith(sf::Vec
    const &,bool)
```

0x00407B1A 处压栈参数，在 0x00407B25 call 之后并没有平栈.
通过动态调试得知 **0x00407B25 前后 esp 没有变化, 说明该 call 是没有参数的**, 跟进 0x00407B25 call 直接将以地址给 eax 了, 直接说明了 **0x00407B1A push 不是 0x00407B25 call 使用的**.

参数与函数可能隔了几个 call

```
00407B1A    push        1 ; includesHead
00407B1C    mov         ecx, [ebp-2Ch]
00407B1F    add         ecx, 1408h
00407B25    call        sf::Transformable::getPosit
00407B2B    push        eax ; location
00407B2C    mov         ecx, [ebp-2Ch]
00407B2F    add         ecx, 30h ; this
00407B32    call        Snake::collidesWith(sf::Vec
    const &,bool)
```

分析 0x00407B32 call, 得出 0x00407B2B 是第一个参数, 上面的 0x00407B1A 是第二个参数. 是因为动态调试发现经过 0x00407B32 call 前后,esp 变化为 8, 所以直接找最近的栈的两个 push 即可.

参数与函数可能隔了几个 call

总结:

1. 通过动态调试观察 esp 变化来判断参数的个数
2. 经过编译器的优化, 函数的 push 可能在其他 call 之前进行压栈的

思考:

在函数被调用前后观察 esp 的变化, 是不是就能确定函数的调用约定了呢?

SnakeGameApp

虚函数简单识别

```
0040906B    mov     edx,dword ptr [eax+4]
0040906E    call    edx
```

0040906E call 调用目标是寄存器, 说明 edx 存放的是函数指针, 从 0040906B 处 edx 通过 eax+4 解引用得到的。初步确认 edx 存放的是虚函数指针, 而 eax 是虚表的首地址.eax 进一步在内存中确认:

0x0040B9AC f0 78 40 00

0x0040B9B0 c0 79 40 00

0x0040B9B4 d0 7c 40 00

0x0040B9B8 40 74 40 00

0x0040B9BC 60 78 40 00

从上面的存放内容, 基本就可以确认是虚表内容了. 如果某个对象存放了这个 eax, 那么那个引用地址就是虚表的所属的对象了。

虚函数简单识别

总结:
call 后面的寄存器是由一个地址加偏移的结果解引用得到的, 那么就大胆的猜测它就是虚函数吧!