

函数

通常使用 `ebp` 来进行参数寻址，优化后，函数功能很少的情况下，低版本编译器可能会直接使用 `esp` 寻址，而 `ebp` 另作他用

在高版本中一直使用 `ebp`，内联汇编中一定使用的是 `ebp`，调用外部函数一定使用的是 `ebp`

识别调用约定

1. `__cdecl`

- 在函数内部有参数的访问，在 `ret` 没有自带平栈
- 编译器对其函数做符号名称是加前缀 `_`，例如函数 `func` 的符号名称为 `_func`

2. `__stdcall`

- 在函数内部有参数的访问，在 `ret` 有自带平栈，即 `ret i8`
- 寄存器没有参与信息传递的工作
- 编译器对其函数做符号名称是加前缀 `_`，并补充后缀 `@参数总字节数`，例如函数 `func(char, int)` 的符号名称为 `_func@5`

3. `__fastcall`

- 寄存器 `ecx` 和 `edx` 分别参与参数1和参数2的信息传递工作，更多的参数通过栈传递
- 在函数内部有参数的访问，在 `ret` 有自带平栈，即 `ret i8`
- 编译器对其函数做符号名称是加前缀 `@`，并补充后缀 `@参数总字节数`，例如函数 `func(char, int)` 的符号名称为 `@func@5`
- 该约定非标准，实际分析中应结合具体编译环境

优化

传统优化是以函数为单位

全程序优化可以跨函数优化

作用域

参数与局部变量

- 参数使用 `ebp + xxx` 寻址
 - `mov eax, [ebp + 8]`
- 局部变量使用 `ebp - xxx`
 - `mov eax, [ebp - 4]`

全局和静态全局变量

静态全局和全局都使用立即数间接寻址 [xxx]，且在底层上是没有区别的，作用域是在编译期间由编译器和链接器控制，例如 `mov eax, [0x40100000]`

当全局变量以变量（或函数）给初值的时候，静态分析中初值为0，且会程序运行时赋初值

```
int func()
{
    return 3;
}

int n = func();

int main()
{
    return 0;
}
```

在main函数的调用方的 `_cinit` 中最第二个 `initterm` 初始化回调函数中（在高版本中可能被内联优化了，但还是在第二个 `initterm` 中），访问了一个函数指针数组，其中代理函数调用了 `func`，然后把返回值给了 `n` 变量，即类似C++全局对象的构造

VS 2017，在Release版中，`initterm` 初始化函数在 `__scrt_common_main_seh` 里



```
static __declspec(noinline) int __cdecl __scrt_common_main_seh()
{
    if (!__scrt_initialize_crt(__scrt_module_type::exe))
        __scrt_fastfail(FAST_FAIL_FATAL_APP_EXIT);

    bool has_ctor = false;
    __try
    {
        bool const is_nested = __scrt_acquire_startup_lock();

        if (__scrt_current_native_startup_state == __scrt_native_startup_state::initializing)
        {
            __scrt_fastfail(FAST_FAIL_FATAL_APP_EXIT);
        }
        else if (__scrt_current_native_startup_state == __scrt_native_startup_state::uninitialized)
        {
            __scrt_current_native_startup_state = __scrt_native_startup_state::initializing;

            if (_initterm_e(__xi_a, __xi_z) != 0)
                return 255;

            _initterm(__xc_a, __xc_z); // [_xc_a, _xc_z]范围
            __scrt_current_native_startup_state = __scrt_native_startup_state::initialized;
        }
        else
        {
            has_ctor = true;
        }

        __scrt_release_startup_lock(is_nested);

        // If this module has any dynamically initialized __declspec(thread)
        // variables, then we invoke their initialization for the primary thread
    }
    __finally
    {
        __scrt_release_startup_lock(is_nested);
    }
}
```

静态局部和堆变量

- 静态局部
 - 存放位置，基本同全局一致
 - 以常量初始化时放在以初始化区
 - 但是，以变量（或函数）初始化时放在未初始化区
 - 初始化行为
 - 以常量初始化时不产生代码

- 但是，以变量（或函数）初始化时一定会产生代码，通过标志来判断是否进行赋值操作

1. 找一个能访问到的全局空间（高版本在TLS中，考虑了多线程问题）用来记录初始化状态
2. 判断初始化状态
3. 若为未初始化状态，将未初始化改为已初始化状态，执行初始化操作

- 堆变量

- 通过函数签名识别、通过地址识别、通过内存中的一些特征值（调试状态下大部分是 0xfeef 或 0xdddd 或 0x0、非调试状态下是随机值）识别