C++逆向学习(四) 类

ret2nullptr / 2019-05-28 08:39:00 / 浏览数 5044 安全技术 二进制安全 顶(2) 踩(0)

上一篇文章分析了移动构造函数，这篇详细的分析一下C++■的逆向相关内容

已经有很多书和文章分析的比较清楚了，本文尽可能展现一些有新意的内容

测试代码

基类base，派生类derived，分别有成员变量、成员函数、虚函数

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

class base {
public:
   int a;
   double b;
   base() {
       this->a = 1;
       this->b = 2.3;
       printf("base constructor\n");
   }
   void func() {
       printf("%d  %lf\n", a, b);
   }
   virtual void v_func() {
       printf("base v_func()\n");
   }
   ~base() {
       printf("base destructor\n");
   }
};

class derived :public base {
public:
   derived() {
       printf("derived constructor\n");
   }
   virtual void v_func() {
       printf("derived v_func()");
   }
   ~derived() {
       printf("derived destructor\n");
   }
};


int main(int argc, char** argv) {
   base a;
   a.func();
   a.v_func();

   base* b = (base*)new derived();
   b->func();
   b->v_func();
   return 0;
}
```

编译：`g++ test.cpp -o test`

IDA视角

IDA打开，如下：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  derived *v3; // rbx
  char v5; // [rsp+20h] [rbp-30h]
  unsigned __int64 v6; // [rsp+38h] [rbp-18h]

  v6 = __readfsqword(0x28u);
  base::base((base *)&v5);
  base::func((base *)&v5);
  base::v_func((base *)&v5);
  v3 = (derived *)operator new(0x18uLL);
  derived::derived(v3);
  base::func(v3);
  (**(void (__fastcall ***)(derived *))v3)(v3);
  base::~base((base *)&v5);
  return 0;
}
```

this指针

可以看到，base::的每个函数都传入了一个参数(base*)&v5，正是类实例的this■■

以下是普通成员函数func()的调用过程

```
.text:0000000000400915                  xor     eax, eax
.text:0000000000400917                  lea     rax, [rbp+var_30]
.text:000000000040091B                  mov     rdi, rax        ; this
.text:000000000040091E                  call    _ZN4baseC2Ev    ; base::base(void)
.text:0000000000400923                  lea     rax, [rbp+var_30]
.text:0000000000400927                  mov     rdi, rax        ; this
.text:000000000040092A ;   try {
.text:000000000040092A                  call    _ZN4base4funcEv ; base::func(void)
```

rdi作为第一个参数，存放this指针，而windows下是寄存器rcx

this指针是识别类成员函数的一个关键

如果看到C++生成的exe文件中，如果rcx寄存器还没有被初始化就直接使用，很可能是类的成员函数

构造、析构

考虑构造函数时的过程

```
int __fastcall base::base(base *this)
{
  *(_QWORD *)this = &off_400C18;
  *((_DWORD *)this + 2) = 1;
  *((_QWORD *)this + 2) = 4612361558371493478LL;
  return puts("base constructor");
}
```

其中*this = off_400C18，即先把类的虚表地址赋值给类实例的首字段

```
400C08 _ZTV4base        dq 0                    ; offset to this
400C10                  dq offset _ZTI4base     ; `typeinfo for'base
400C18 off_400C18       dq offset _ZN4base6v_funcEv
400C18                                          ; DATA XREF: base::base(void)+C↑o
400C18                                          ; base::~base()+C↑o
400C18                                          ; base::v_func(void)
```

补充一些

注意虚表前还有一个typeinfo，在g++的实现中，真正的typeinfo信息在虚表之后，虚表的前一个字段存放了typeinfo的地址

typeinfo是编译器生成的特殊类型信息，包括对象继承关系、对象本身的描述等

```
Aclass* ptra=new Bclass;
int ** ptrvf=(int**)(ptra);
RTTICompleteObjectLocator str=
*((RTTICompleteObjectLocator*)(*((int*)ptrvf[0]-1)));   //vptr-1
```

这段获取对象RTTI信息相关的代码也显示了这一点

回到构造和析构函数

在构造函数调用中，显然需要将虚表的地址赋值给类实例的虚表指针，从代码上来看也是这样

但是，我们观察base类的析构函数

```
int __fastcall base::~base(base *this)
{
  *(_QWORD *)this = &off_400C18;
  return puts("base destructor");
}
```

析构时也首先重新赋值了虚表指针，看起来可能有点多此一举

但如果析构函数中调用了虚函数，此行为可以保证正确；至于如果不重新赋值会有错误行为的情况就不展开了

虚表指针的赋值是识别的一个关键，排除开发者故意伪造编译器生成的代码来误导分析，基本可以确定是构造函数或者析构函数

同样的，找到了虚表，也就可以根据IDA的交叉引用，找到对应的构造函数和析构函数

构造、析构代理函数

全局对象和静态对象的构造时机相同，可以说是被隐藏了起来，在main函数之前由构造代理函数统一构造

测试代码：

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

using namespace std;

class t {
public:
   char* str;
   t() {
       cout << "constructor" << endl;
       this->str = new char[16];
       memcpy(this->str, "hello", 12);
   }
   ~t() {
       cout << this->str << endl;
```

```
   }
};

t ts[10];

int main(int argc, char** argv) {
   cout << "main" << endl;
   return 0;
}
```

编译：visual studio 2019 x64 release

IDA打开，根据输出，下断点后发现t类全局变量构造函数输出信息调用于initterm_0函数

```
if ( _scrt_current_native_startup_state )
{
  v2 = 1;
}
else
{
  _scrt_current_native_startup_state = 1;
  if ( (unsigned int)initterm_e_0((__int64)&_xi_a, (__int64)&_xi_z) )
    return 255i64;
  initterm_0((__int64)&_xc_a, (__int64)&_xc_z);
  _scrt_current_native_startup_state = 2;
}
```

一段initterm_0的代码实现如下：

```
while (pfbegin < pfend) {
   //pfbegin == __xc_a , pfend == __xc_z
   if (*pfbegin != NULL) {
       (**pfbegin)(); //■■■■■■■■■■■■■■■■
       ++pfbegin();
   }
}
```

执行(**pfbegin)()后并不会进入全局对象的构造函数中，而是进入编译器提供的构造代理函数

最简单的找到全局对象构造函数的方法：因为构造代理函数中会注册析构函数，其注册方式是使用atexit，我们对atexit下断点，调试过程中很容易在附近找到全局对象

如图所示，10即为对象数组的大小，并且最后一个参数传入了构造函数指针t::t()

```
1 int dynamic_initializer_for__ts__()
2 {
3   eh_vector_constructor_iterator_(
4     (__int64)&ts,
5     8i64,
6     10i64,
7     (void (__cdecl *)(__int64, void (__fastcall __noreturn *)()))t::t);
8   return atexit(dynamic_atexit_destructor_for__ts__);
9 }
```

析构代理函数比较类似，就不多分析了，同样以atexit为切入点

t::_t即为t类的析构函数

```
void __cdecl dynamic_atexit_destructor_for__ts__()
{
  eh_vector_destructor_iterator_(
    (__int64)&ts,
    8i64,
    10i64,
    (void (__cdecl *)(__int64, void (__fastcall __noreturn *)()))t::_t);
}
```

虚函数调用

代码中我们用base*指针指向了new derived()，在IDA里如下

```
v3 = (derived *)operator new(0x18uLL);
derived::derived(v3);
base::func(v3);
(**(void (__fastcall ***)(derived *))v3)(v3);
```

v3作为derived类实例的地址，存放的正好是虚表指针，而v_func()正好在虚表的第一个位置，参数v3则是例行传入this指针

已经有很多文章讲过虚函数调用过程了，这里就只是简单说一下

虚基类继承

主要分析一下菱形继承的内存布局，代码如下：

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

//■■■■
class A {
public:
   virtual void function() {
       printf("A virtual function\n");
   }
   int a;
};

//■■■■
class B :virtual public A { //■■■
public:
   virtual void func() {
       printf("B virtual func()\n");
   }
   int b;
};

//■■■■
class C :virtual public A { //■■■
public:
   virtual void func() {
       printf("C virtual func()");
   }
   int c;
};

//■■■
class D :public B, public C {
public:
   virtual void function() {
       printf("D virtual function()");
```

```
    }
    int d;
};


int main(int argc, char** argv) {
    A* A_ptr = (A*)new D();
    A_ptr->function();

    return 0;
}
```

编译：visual studio 2019 x64 release

B、C类都虚继承了A类，然后D类多重继承于B、C类

布局如图：

A_ptr	0x000001ff79255e38 {d=0 }
[D]	{d=0 }
B	{b=0 }
A	{a=0 }
__vfptr	0x00007ff7ff4ead10 {cpp.exe!void(* D::`vftable'[2])()} {0x00007ff7ff4e141a {cpp.exe!B::func(void)}}
b	0
C	{c=0 }
A	{a=0 }
__vfptr	0x00007ff7ff4ead28 {cpp.exe!void(* D::`vftable'[2])()} {0x00007ff7ff4e1181 {cpp.exe!C::func(void)}}
c	0
A	{a=0 }
d	0
__vfptr	0x00007ff7ff4ead40 {cpp.exe!void(* D::`vftable'[2])()} {0x00007ff7ff4e10af {cpp.exe!D::function(void)}}
a	0

具体实现是在B、C类里不再保存A类的内容，而是保存一份偏移地址，然后将A类的数据保存在一个公共位置处，降低数据冗余

为方便说明，使用g++编译并用IDA打开

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  _QWORD *addr; // rbx
  void (__fastcall ***func)(_QWORD); // rax

  addr = (_QWORD *)operator new(0x30uLL);       // 分配内存
  memset(addr, 0, 0x30uLL);
  D::D((D *)addr);                              // 构造函数
  if ( addr )
    func = (void (__fastcall ***)(_QWORD))((char *)addr + *(_QWORD *)(*addr - 24LL));// 找到需要调用的虚函数
  else
    func = 0LL;
  (**func)(func);                               // 虚函数call
  return 0;
}
```

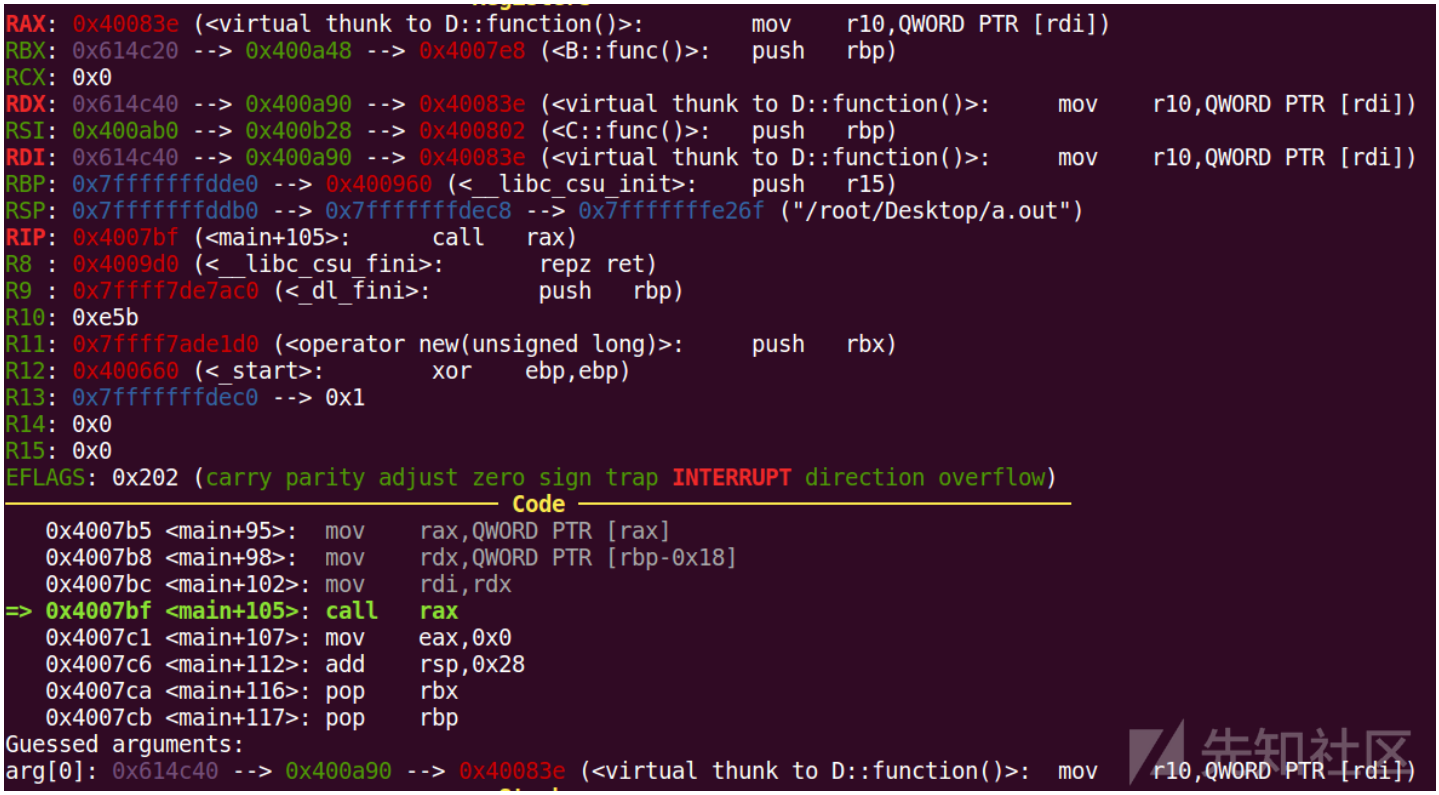
main函数比较清晰，跟进D类的构造函数

```
D *__fastcall D::D(D *this)
{
  D *result; // rax

  A::A((D *)((char *)this + 32));               // 虚基类A的偏移
  B::B(this);                                   // B类偏移(0)
  C::C((D *)((char *)this + 16), off_400AB0);   // C类偏移(16)
  *(_QWORD *)this = off_400A48;                 // B类虚表，相对D类this指针偏移为0
  *((_QWORD *)this + 4) = &off_400A90;          // A类虚表，相对D类this指针偏移为(QWORD)*4 == 32
  result = this;
  *((_QWORD *)this + 2) = &off_400A70;          // A类虚表，相对D类this指针偏移为(QWORD)*2 == 16
  return result;
}
```

虚表占8字节，int占4字节，考虑字节对齐，实际B、C类都占了16字节

接着用gdb跟进一下，断在(**func)(func)上



已经分析过，D类的首字段即存放了B类的虚表，也就是`RBX==0x614c20`是D类实例地址

IDA可以看到`0x400A90==A::vtable`，也就是先找到A类的虚表

而A类虚表实际存放的函数指针值，由于虚函数机制被`D::function()`覆盖，会实际调用到D类对应的函数

补充

关于如何让IDA里的分析更清晰，添加结构体、类的信息来帮助IDA的内容，网上已经有很多，这里不再多说了

推荐一本书《深度探索C++对象模型》，里面有很多类布局的历史实现，以及这些布局设计时对空间、时间效率的权衡