

最近研究了一下C++类的移动构造函数，同时也进行了一些逆向分析，过程中碰到一个很奇怪的问题，以此记录

相关背景

右值引用

右值引用主要是为了解决C++98/03遇到的两个问题

1. 临时对象非必要的昂贵的拷贝操作
2. 模板函数中如何按照参数的实际类型进行转发

本文主要探讨问题1，一些代码尝试和IDA中逆向的分析

学习链接：[从4行代码看右值引用](#)，这里就不多说了

move语义

比如在`vector.push_back(str)`时，`str(■)`作为实参，会复制一份自身成为形参，进入函数调用

而在这个过程中就会产生临时对象，那么也就会调用拷贝构造函数

而如果`vector.push_back(std::move(str))`，就可以匹配移动构造函数，省去这个拷贝过程以提高效率

链接中已经解释的很详细了，不再赘述，总之就是给将亡值续命，延长它的生命周期（原本很可能是一个临时变量）

代码分析

接下来的部分内容可以作为上一篇文章[C++逆向学习\(二\) vector](#)的补充，在分析移动构造函数时又学到了一些之前没有注意过的vector的细节

Str类源码

```
#include<iostream>
#include<string.h>
#include<vector>

using namespace std;

class Str {
public:
    char* str;
    Str(char value[]) {
        cout << "Ordinary constructor" << endl;
        int len = strlen(value);
        this->str = (char*)malloc(len + 1);
        memset(str, 0, len + 1);
        strcpy(str, value);
    }
    //■■■■■■■
    Str(const Str& s) {
        cout << "copy constructor" << endl;
        int len = strlen(s.str);
        str = (char*)malloc(len + 1);
        memset(str, 0, len + 1);
        strcpy(str, s.str);
    }
    //■■■■■■■
    Str(Str&& s) {
        cout << "move constructor" << endl;
        str = s.str;
        s.str = NULL;
    }
    ~Str() {
```

```

        cout << "destructor" << endl;
        if (str != NULL) {
            free(str);
            str = NULL;
        }
    }
};
//g++ xxx.cpp -std=c++17

```

代码1

main函数中，不使用move语义，会调用拷贝构造函数

```

int main(int argc, char** argv) {
    char value[] = "template";
    Str s(value);
    vector<Str> vs;
    vs.push_back(s);
    return 0;
}

```

IDA打开如下

```

strcpy(value, "template");
Str::Str((Str *)&s, value);
std::vector<Str,std::allocator<Str>>::vector((__int64)&vector);
std::vector<Str,std::allocator<Str>>::push_back(&vector, &s);
std::vector<Str,std::allocator<Str>>::~~vector(&vector);
Str::~Str((Str *)&s);

```

先知社区

简单的流程，甚至Str的高亮都是对称的

最初调用Str的拷贝构造函数，匹配的是Str(char value[])，接着初始化vector，然后一次push_back(s)

跟进push_back

```

__int64 __fastcall std::vector<Str,std::allocator<Str>>::push_back(__int64 vector, __int64 s)
{
    __int64 result; // rax

    if ( *(_QWORD *)(vector + 8) == *(_QWORD *)(vector + 16) )
        return std::vector<Str,std::allocator<Str>>::_M_emplace_back_aux<Str const&>(vector, s);
    std::allocator_traits<std::allocator<Str>>::construct<Str,Str const&>(vector, *(_QWORD *)(vector + 8), s);
    result = vector;
    *(_QWORD *)(vector + 8) += 8LL;
    return result;
}

```

先知社区

一开始仍然是熟悉的判断vector的size & capacity的关系，最终调用的是这里的复制构造函数

```

char *__fastcall Str::Str(Str *this, const Str *a2)
{
    __int64 v2; // rax
    int v3; // ST1C_4

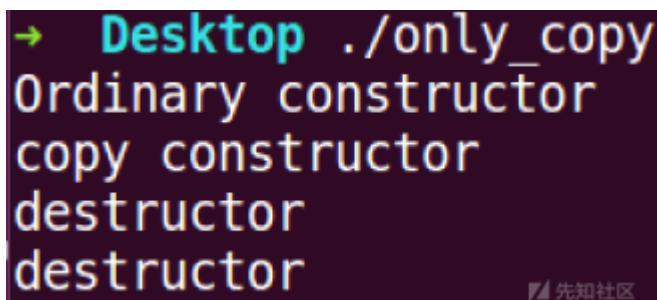
    v2 = std::operator<<<std::char_traits<char>>(&std::cout, "copy constructor");
    std::ostream::operator<<(v2, &std::endl<char,std::char_traits<char>>);
    v3 = strlen(*(const char **)a2);
    *(_QWORD *)this = malloc(v3 + 1);
    memset(*(void **)this, 0, v3 + 1);
    return strcpy(*(char **)this, *(const char **)a2);
}

```

先知社区

注意第一个参数是this，是C++成员函数调用时的第一个参数，类指针

运行结果：



代码2

代码2，只move(s)

```
int main(int argc, char** argv) {
    char value[] = "template";
    Str s(value);
    vector<Str> vs;
    //vs.push_back(s);

    //cout<<"-----"<<endl;
    vs.push_back(move(s));
    return 0;
}
```

IDA打开如下：

```
v9 = __readfsqword(0x28u);
strcpy(value, "template");
Str::Str((Str *)&s, value);
std::vector<Str,std::allocator<Str>>::vector((__int64)&vec);
std::vector<Str,std::allocator<Str>>::push_back(&vec, &s);
v3 = std::operator<<<std::char_traits<char>>(&std::cout, "-----");
std::ostream::operator<<(v3, &std::endl<char,std::char_traits<char>>);
s_move = std::move<Str &>(&s);
std::vector<Str,std::allocator<Str>>::push_back(&vec, s_move);
std::vector<Str,std::allocator<Str>>::~~vector(&vec);
Str::~~Str((Str *)&s);
return 0;
```

注意到其中的std::move，跟进发现其实实现只有一句话

```
__int64 __fastcall std::move<Str &>(__int64 s_addr)
{
    return s_addr;
}
```

也印证了move实际上不移动任何东西，唯一的功能是将一个左值强制转换为一个右值引用

继续跟进

```

__int64 __fastcall std::vector<Str,std::allocator<Str>>::emplace_back<Str>(__int64 vec, __int64 s_move)
{
    __int64 v2; // rax
    __int64 result; // rax
    __int64 v4; // rax

    if ( *(_QWORD *)(vec + 8) == *(_QWORD *)(vec + 16) )
    {
        v4 = std::forward<Str>(s_move);
        result = std::vector<Str,std::allocator<Str>>::_M_emplace_back_aux<Str>(vec, v4);
    }
    else
    {
        v2 = std::forward<Str>(s_move);
        std::allocator_traits<std::allocator<Str>>::construct<Str,Str>(vec, *(_QWORD *)(vec + 8), v2);
        result = vec;
        *(_QWORD *)(vec + 8) += 8LL;
    }
    return result;
}

```

仍然是判断大小和容量的代码，接着调用的是移动构造函数

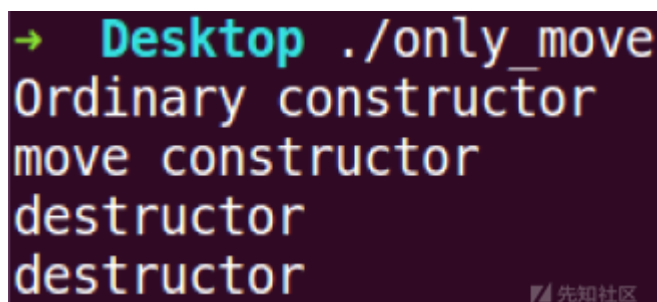
```

_QWORD * __fastcall Str::Str(_QWORD *a1, _QWORD *a2)
{
    __int64 v2; // rax
    _QWORD *result; // rax

    v2 = std::operator<<<std::char_traits<char>>(&std::cout, "move constructor");
    std::ostream::operator<<(v2, &std::endl<char,std::char_traits<char>>);
    *a1 = 0LL;
    *a1 = *a2;
    result = a2;
    *a2 = 0LL;
    return result;
}

```

运行结果：



```

→ Desktop ./only_move
Ordinary constructor
move constructor
destructor
destructor

```

代码3

这段代码实际上只是在`move`之前加上了一句`push_back(s)`，但是运行结果差了很多

作为对vector的补充

"我全都要"写法，同时用拷贝构造和移动构造

```

int main(int argc, char** argv) {
    char value[] = "template";
    Str s(value);
    vector<Str> vs;
    vs.push_back(s);

    cout<<"-----"<<endl;
    vs.push_back(move(s));
}

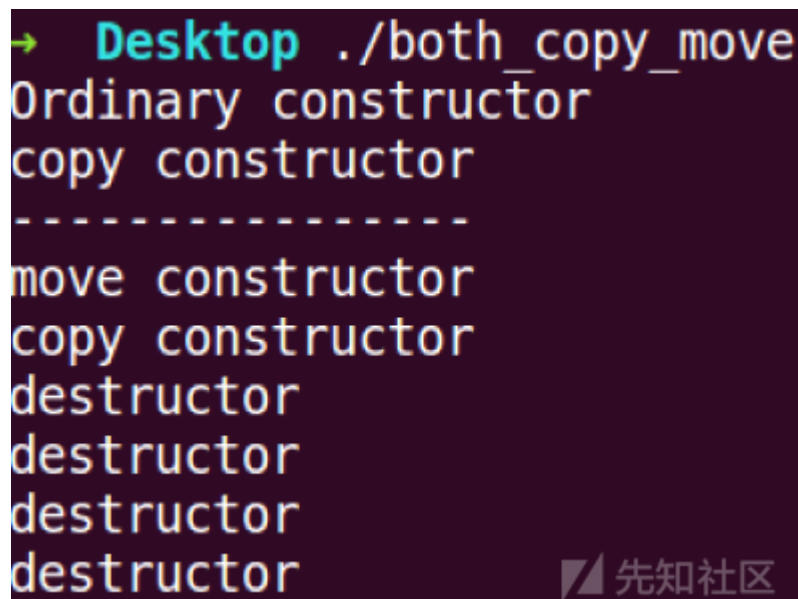
```

```

return 0;
}

```

按理来说，输出结果也应该比代码多一个copy constructor和destructor，但实际上多了很多东西



```

Desktop ./both_copy_move
Ordinary constructor
copy constructor
-----
move constructor
copy constructor
destructor
destructor
destructor
destructor

```

IDA打开并没有出乎意料的结果，仍然是清晰的两次push_back，跟进后也没有什么特别的发现，查看交叉引用也没能找到相关信息

为什么在move之后还会有一次copy，对应的之后又多了一个desctructor？

首先，vector虽然是值语义，但是move过后，既然已经调用了移动构造函数，肯定不会再无聊的拷贝一次

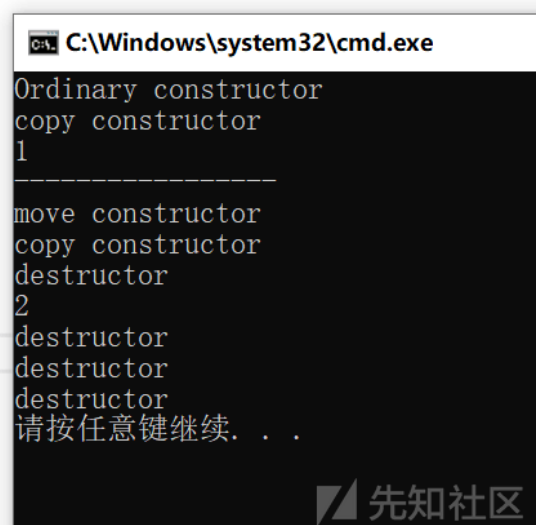
在vs里调试，输出各个时间点的capacity

```

int main(int argc, char** argv) {
    char value[] = "template";
    Str s(value);
    vector<Str> vs;
    vs.push_back(s);
    std::cout << vs.capacity() << std::endl;

    cout << "-----" << endl;
    vs.push_back(move(s));
    std::cout << vs.capacity() << std::endl;
    return 0;
}

```



```

C:\Windows\system32\cmd.exe
Ordinary constructor
copy constructor
1
-----
move constructor
copy constructor
destructor
2
destructor
destructor
destructor
请按任意键继续. . .

```

注意■■■■destructor和■■2的出现时间

跟进源码好久后才发现，多的copy的产生原因，是因为vector内部动态扩容时，在新开辟的空间上调用了复制构造函数

也就是说把原来的一个Str s复制到了新内存空间，这个过程并没有调用移动构造函数

可能这也是写了移动构造函数后，保险起见也要写一个复制构造函数的原因

其他

考虑这个问题

为什么vector内部扩容时，要在新地址调用拷贝构造函数呢？

之前文章已经分析过，vector实际上只存了类型的数据结构

直接memcpy(new_memory,old_memory,size)，再把旧内存空间清零，会造成什么问题？

查了一些资料后发现，扩容是allocator的事情，一个可能的实现是原位new

而如果直接memcpy，会不会出问题取决于vector存的类型是否平凡(POD)
POD是Plain old data structure的缩写

资料提到shared_ptr也可能会受影响，取决于引用计数放在哪里

但无论如何，指针的浅拷贝、深拷贝问题值得注意，否则在vector内部扩容时，可能2个指针指向同一块内存，析构时会产生严重的错误

一个月后的SUCTF会有一道C++底层相关的pwn，欢迎来体验

点击收藏 | 0 关注 | 1

[上一篇：反-反汇编patch学习（四）](#) [下一篇：CVE-2019-0808内核漏洞...](#)

1. 0 条回复
- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)