C/C++

- 函数重载的底层实现，虚函数的底层实现
- C++的STL库你了解吗
- 一个类，写了一个构造函数，还写了一个虚构造函数，可不可以，会发生什么
- 如何实现一个不可以被继承的类
  - 方法一 使用final关键字 C++11新特性
  - 方法二 手动实现
- C++实现多态需要几个条件 在汇编层是如何实现虚函数的
- C++虚函数的工作原理是什么？？
- C语言的关键字 "volatile" 有何作用？？
- 什么函数不能是虚函数 为什么(重点)
- C和C++中的struct有什么区别
- C++中的struct和class有什么区别
- C++函数中值的传递方式有几种
- 引用和指针有什么区别
- C++中virtual与inline的含义分别是什么
- const与#define比较 const有什么优点
- 有了malloc/free为什么还要new/delete
- 多态类中的虚函数表是Complie-Time还是Run-Time时建立的
- 内存的分配有几种方式
- 全局变量和局部变量有什么区别 是怎么实现的 操作系统和编译器是怎么知道的
- 输入一个字符串 将其逆序后输出
- 编写strcpy函数
- 写一个函数判断系统是大端还是小端
- 简述一下cdecl,fastcall,stdcall和thiscall的区别
- 如果不使用API 使用库函数来获取文件大小
- 有下面的结构体声明 分别指出其在32位和64位平台下的sizeof大小 该结构成员的声明顺序是否有问题 如果有 是优点还是缺点 请说明具体原因

Windows编程

- 请写出windows32位平台上下面数据类型的表示范围
- 简述临界区，互斥量，信号量，事件的区别
- GastLastError函数中是如何实现各线程间互不干扰的 请说明其原理
- 简述一下如何截取键盘的响应 让所有的'a'变成'b'
- 简述一下常用的进程通信方法有哪些
- 简述一下PostMessage与SendMessage有何区别

数据结构和算法

- 二叉树的遍历
- 快速排序
- 冒泡排序
- 你所了解的字符串匹配的算法有哪些?

PE文件

- PE文件的加载流程
- PE的常见区段，一个全局变量在哪个区段
- 为什么PE文件格式要用到RVA呢
- 区段结束默认用0补齐 可以用其他内容补齐吗
- PE头可不可以放在最后一个区段？PE文件最多能有几个区段?
- 如何判断一个文件是PE文件
- PE文件大体结构是什么样子的
- 怎么找到NT头
- PE文件默认加载基址保存在哪里
- 扩展头后面跟的是什么?怎么找到这个位置?

- 区段表中的VirtualAddress字段保存的是什么,PointToRawData呢?
- 区段表的属性都有什么属性
- 区段的名字有什么作用？区段的名字都是固定的吗?
- VirtualAddress和PointToRawData为什么会不一样?
- 什么是RVA？什么是Offset?转换关系是什么样的?
- 区段表中，区段文件大小和内存中的大小，哪个是对齐过的?
- 导出表的作用是什么? 没有它exe能运行吗?
- 导入表的作用是什么? 没有它exe能运行吗?
- 怎么才知道导出函数是仅以序号导出还是以名称导出?
- 如何判断导入函数是以序号导入还是以名称导入?
- 已知函数名，如果我们想要使用代码找到某一个函数的地址，该怎么做?
- 什么是IAT？里面存储的什么?
- 什么是INT？里面存储的什么?
- 怎么才能知道一个exe都使用了哪些API?
- 重定位有什么作用?
- 哪些PE文件需要重定位?
- 怎么判断PE是DLL,还是EXE
- 怎么判断PE文件是32位还是64位
- 如何去掉一个程序的重定位
- 为什么要修复重定位 如何修复重定位?
- 说出你熟知的windows资源
- windows资源表有几层结构
- 资源表每一层结构分别代表什么含义?

软件逆向与汇编

- 调试器是如何实现栈回溯的
- Exeinfo PE 和 PEiD两个工具的 原理
- OD用过什么插件 StrongOD常用选项 跳过哪些异常
- 如何调试没有源代码的驱动 SXE命令 驱动断点位置
- IDA F5不能正确识别函数，怎么解决(花指令 混淆的处理方法)
- IDA重设基址的方法
- 已经执行的程序 如何从头重新调试
- 如何调试服务
- DLL文件如何调试
- CALL和FF15 FF25有什么区别
- 反虚拟机技术都有哪些
- 各种反调试技术
- 分析过什么程序
- 32位windows上函数调用 参数压栈方式 堆栈平衡方式有几种？为什么要那么多种？简单说说每种的特点
- 描述一下反汇编后的switch是怎么进行的
- IDA和OD在逆向的区别 其中如何在OD中定位win32程序的main函数
- 如果分析一个杀毒软件的杀毒引擎 有什么思路
- 如果写一个远控软件 有什么思路
- 常见加解密算法了解多少
- od和ida的反汇编引擎有什么区别?
- 反汇编引擎工作原理?
- 怎么写一个注册机？如果是网络验证如何破解?
- IDA脚本是否用过
- 如何确定汇编指令的长度
- 给你一个外挂，如何逆向获得里面的功能?
- 假如调用了一个API，那么在调用API处，OPCODE与汇编指令的形式是什么样的?
- 汇编指令call和ret的具体操作是什么
- PE文件是什么 属于PE文件的扩展名有哪些
- OD中常规断点的快捷键是什么 并简要说明常规断点与内存断点的实现原理

- 简述一个OFFSET与LEA的区别
- 简述一下Intel CPU体系架构是如何实现硬件断点 且相对于软件断点的优势
- 80x86 16位CPU有四个段寄存器 它们有什么作用

病毒分析

- 病毒分析流程
  - 静态分析
  - 动态分析
- 假设有1000个样本，怎么分析，识别黑白?
- 常见的病毒类型有哪些
- 傀儡进程的实现原理
- BadRabbit执行流程
- API Monitor什么功能
- 手工查杀病毒一般步骤有哪些
- DES密钥长度有多少位 多少个字节 多少个有效位
- MD5和SHA的长度
- 对僵尸，蠕虫，木马有什么了解，说下理解，给你一个病毒你是怎么判断是这三种病毒
- 常见的病毒行为或是特征有哪些
- 勒索病毒 是用的什么加密方式?
- 杀毒软件源码看过没有?
- DLL劫持原理
- Windows病毒常用自启动方式 注册表有哪些启动项? 注册表启动项由谁来启动? 计划任务由哪个进程启动?
- 简单谈一谈主动防御
- yara规则的优点与缺点?
- 反病毒引擎扫描的工作方式

调试与异常

- 没有调试器如何处理异常 VEH VCH区别 VEH位置 SEH栈中结构 成员字段
- 异常分发流程
  - 用户层异常处理流程
  - 内核层异常处理流程

HOOK和注入

- 输入法注入的原理
- Inline Hook的实现流程
- 简述ssdt hook
- 什么是远程注入? 还有哪些注入方式?
- 零环的HOOK都有哪些，如何实现?
- 什么是DLL注入技术?
- APC注入流程
- 远程线程注入流程
- 远程线程注入的原理?
- 消息钩子注入的原理?
- 什么是Hook技术?
- inline-Hook的原理 如何检测?
- IAT-Hook的原理 如何检测?

漏洞相关

- PCManFTP是什么类型漏洞，利用方法
- Office0158漏洞原理
- 漏洞缓解技术，漏洞防范技术
- Windows下常见的漏洞缓解技术有哪些

内核相关

- 进程和线程数据结构
- 内核重载方法 有什么注意事项
- 简述DPL,CPL,RPL的含义 谈一谈你的理解
- 什么是段选择子 请详细描述段选择子的结构

C/C++

函数重载的底层实现，虚函数的底层实现

名称粉碎机制

虚函数表+虚表指针

C++的STL库你了解吗

用过vector string map

一个类，写了一个构造函数，还写了一个虚构造函数，可不可以，会发生什么

不可以 构造函数不能是虚函数 原因有以下几点

1. 虚函数的作用是实现部分或默认的功能 而且该功能可以被子类所修改 如果父类构造函数设置成虚函数 那么子类的构造函数会直接覆盖掉父类的构造函数 而父类的构造函数就失去了一些初始化的功能 这与子类的构造需要先

完成父类的构造的流程相违背 而这个后果相当严重

2. 虚函数的调用是通过虚函数表来进行的 而虚函数表也需要在对象实例化之后才能够进行调用 在构造对象的过程中 还没有为虚函数表分配内存 所以这个调用也是违背先实例化后调用的准则
3. 虚函数的调用是由父类指针进行完成的 而对象的构造则是由编译器完成的 由于在创建一个对象的过程中 涉及到资源的创建 类型的确定 而这些是无法在运行过程中确定的 需要在编译的过程中就确定下来 而多态是运行过程中体现出来的 所以是不能够通过虚函数来创建构造函数的 与实例化的次序不同也有关系

如何实现一个不可以被继承的类

方法一 使用final关键字 C++11新特性

```
class Base final
{
};

// 错误，Derive不能从Base派生。
class Derive: public Base
{
};
```

方法二 手动实现

```
#include<iostream>
using  namespace std;

class A
{
public:
    static A * Construct(int  n)
    {
        A *pa = new A;
        pa->num = n;
        cout << "num  is:" << pa->num << endl;
        return pa;
    }
    static void Destruct(A * pIntance)
    {
        delete  pIntance;
        pIntance = NULL;
    }

private:
    A(){}
    ~A(){}

public:
    int num;
};

void  main()
{
    A *f = A::Construct(9);
```

```
    cout << f->num << endl;
    A::Destruct(f);
}
```

```
#include<iostream>
using namespace std;

template <typename T>
class Base
{
    friend T;
private:
    Base() {}
    ~Base() {}
};

class Finalclass : virtual public Base<Finalclass>
{
public:
    Finalclass() {}
    ~Finalclass() {}
};

class TestClass : public Finalclass
{
};

void  main()
{
    Finalclass* p = new Finalclass;  // 堆上对象
    Finalclass fs;                // 栈上对象
//  TestClass tc;  // 基类构造函数私有，不可以被继承。因此不可以创建栈上对象。

    system("pause");
}
```

C++实现多态需要几个条件 在汇编层是如何实现虚函数的

C++实现多态需要实现虚函数 继承 父类指针指向子类对象

在汇编层实现虚函数 有两种情况

1. 直接调用虚函数 和普通函数一模一样 因为没有父类指针指向子类对象 是直接调用的
2. 父类指针指向子类对象 会调用虚函数表中的虚函数

C++虚函数的工作原理是什么？？

虚函数需要虚函数表才能实现。如果一个类有函数声明成虚拟的，就会生成一个vtable，存放这个类的虚函数地址。此外，编译器还会在类里加入隐藏的vptr变量。若子类没有覆写虚函数，该子类的vtable就会存放父类的函数地址。调用 这个虚函数时，就会通过vtable解析函数的地址。在C++里，动态绑定就是通过vtable机制实现的。 由此，将子类对象赋值给基类指针时，vptr变量就会指向子类的vtable。这样一来，就能确保继承关系最末端的子类虚函数会被

调用到。在C++里，非虚函数的调用是在编译期通过静态绑定确定的，而虚函数的调用则是在运行期通过动态绑 定确定的。

C语言的关键字 ”volatile“ 有何作用？？

关键字volatile的作用是指示编译器，即使代码不对变量做任何改动，该变量的值仍可能会被外界修改。操作系统、硬件或其它线程都有可能修改该变量。该变量的值有可能遭受意料之外的修改，因此，每一次使用时，编译器都会重新从内存中获取这个值。 volatile变量在多线程程序里也很有用，对于全局变量，任意线程都可能修改这些共享的变量。我们可能不希望编译器对这些变量进行优化。

什么函数不能是虚函数 为什么(重点)

一 不能被继承的函数 二 不能被重写的函数

1. 普通函数 普通函数不属于成员函数 是不能被继承的 普通函数只能被重载 不能被重写 因此声明为虚函数没有意义 因为编译器会在编译时绑定函数 而多态体现在运行时绑定 通常通过基类指针指向子类对象实现多态
2. 友元函数 友元函数不属于类的成员函数 不能被继承 对于没有继承特性的函数没有虚函数的说法
3. 构造函数 构造函数是用来初始化对象的 假如子类可以继承基类构造函数 那么子类对象的构造将使用基类的构造函数 而基类构造函数并不知道子类有什么成员 显然是不符合语义的 从另外一个角度讲 多态是通过基类指针指向子类对象来实现多态的 在对象构造之前并没有对象产生 因此无法使用多态特性 这是矛盾的 因此构造函数不允许继承
4. 内联成员函数 内联函数就是为了在代码中直接展开 减少函数调用花费的代价 也就是说内联函数是在编译时展开的 而虚函数是为了实现多态 是在运行时绑定的 因此内联函数和多态的特性相违背
5. 静态成员函数 首先静态成员函数理论是可继承的 但是静态成员函数是编译时确定的 无法动态绑定 不支持多态 因此不能被重写

C和C++中的struct有什么区别

C语言中结构体不能定义函数 没有访问权限之分 C++的结构体有访问权限 默认是public 可以定义函数

C++中的struct和class有什么区别

1. 是通过struct继承下来的默认权限是public 而class是private
2. 定义一个struct 默认权限是public 而class是private

C++函数中值的传递方式有几种

C++函数的三种传递方式为：值传递 指针传递和引用传递

引用和指针有什么区别

1. 引用必须被初始化 指针不需要
2. 引用初始化以后不能被改变 指针可以改变所指的对象
3. 不存在指向空值的引用 但是存在指向空值的指针

C++中virtual与inline的含义分别是什么

在基类成员函数的声明前加上virtual关键字 意味着将该成员函数声明为虚函数 inline与函数的定义体放在一起 使该函数称为内联 inline是一种用于函数实现的关键字 而不是用于函数声明的关键字

const与#define比较 const有什么优点

1. const常量有数据类型 而宏常量没有数据类型 编译器可以对const常量进行类型安全检查 而对宏常量只进行字符替换 没有安全检查 并且在字符替换可能产生意料不到的错误
2. 有些集成化的调试工具可以对const常量进行调试 但是不能对宏常量进行调试

有了malloc/free为什么还要new/delete

malloc/free是C/C++语言的标准库函数 new/delete是C++的运算符 它们都用于申请动态内存和释放 对于非内部数据类型的对象而言 光用malloc/free无法满足动态申请对象的要求 对象在创建的同时要自动执行构造函数 由于malloc/free是库函数而不是运算符 不在编译器控制权限之内 不能把执行构造函数和析构函数的任务强加与malloc/free

因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new 以及一个能完成清理与释放内存工作的运算符delete

多态类中的虚函数表是Complie-Time还是Run-Time时建立的

虚函数表是在编译时就建立了 各个虚函数 这时被组织成了一个虚函数的入口地址的数组 而对象的隐藏成员 虚函数指针是在运行期——也就是构造函数被调用时进行初始化的 这是实现多态的关键

内存的分配有几种方式

1. 从静态存储区域分配 内存在程序编译的时候就已经分配好了 这块内存在程序的整个运行期间都存在 例如全局变量
2. 在栈上创建 在执行函数的时候 函数内局部变量的存储单元都可以在栈上创建 函数执行结束时这些存储单元自动被释放 栈内存分配运算内置于处理器的指令集中 效率很高 但是分配内存的容量有限
3. 从堆上分配 也叫动态分配内存 程序在运行的时候用malloc或者new申请任意多少的内存 程序员自己负责在何时用free或delete释放内存 动态内存的生存期由我们决定 使用非常灵活 但问题也最多

全局变量和局部变量有什么区别 是怎么实现的 操作系统和编译器是怎么知道的

1. 生存周期不同 全局变量随主程序创建而创建 随主程序销毁而销毁 局部变量在局部函数内部 甚至局部循环体等内部存在 退出就不存在 内存中分配在全局数据区
2. 作用域不同 通过声明后全局变量在程序的各个部分都可以用到 局部变量只能在局部使用 分配在栈区 操作系统和编译器通过内存分配的位置来知道的 全局变量分配在全局数据段并且在程序开始运行的时候被加载 局部变量则被分配在堆栈

输入一个字符串 将其逆序后输出

```
int main()
{
    char a[50] = { 0 };
    int i = 0, j;
    char t;
    cin.getline(a, 50, '\n');
    for (i = 0,j=strlen(a)-1;i<strlen(a)/2; i++,j--)
    {
        t = a[i];        //把第一个存起来
        a[i] = a[j];     //把最后一个赋给第一个
        a[j] = t;        //把第一个赋给最后一个
    }
```

```
    cout << a << endl;
    system("pause");
}
```

编写strcpy函数

```
#include "pch.h"
#include <stdio.h>
#include <assert.h>
#include <windows.h>

char* MyStrcpy(char* strDest, const char* strSrc)
{
    char* address = strDest;
    assert((NULL != strDest) && (NULL != strSrc));
    while ((*strDest++ = *strSrc++) != '\0');
    return address;
}

int main()
{
    char arr[20] = { 0 };
    MyStrcpy(arr,"hello world");
    system("pause");
}
```

写一个函数判断系统是大端还是小端

```
int CheckCPU()
{
    union w
    {
        int a;
        char b;
    }c;
    c.a = 1;
    return (c.b == 1);
}
```

简述一下cdecl,fastcall,stdcall和thiscall的区别

- cdecl是C Declaration的缩写 表示C语言默认的函数调用方法 所有参数从右到左依次入栈 这些参数由调用者清除 称为手动清栈(由调用者把参数弹出栈).对于传送参数的内存栈是由调用者来维护的(正因为如此 实现可变参数的函数只能使用该调用约定) 被调用函数无需要求调用者传递多少参数 调用者传递过多或者过少的参数 甚至完全不同的参数都不会产生编译阶段的错误 cdcel调用约定仅在输出函数名前加上一个下划线前缀 格式为 _functionname
- stdcall是Standard Call的缩写 是C++的标准调用方式 所有参数从右到左依次入栈,如果是调用类成员的话 最后一个入栈的是this指针 这些堆栈中的所有参数由被调用的函数在返回后清除 使用的指令是retn x.x表示参数占用的字节数 cpu在ret之后自动弹出x字节的堆栈空间 称为自动清栈 函数在编译的时候就必须确定参数个数 并且调

用者必须严格的控制参数的生成 不能多 不能少 否则返回后悔报错 stdcall调用约定在输出函数名前加上一个下划线前缀 后面加上一个@符号和其参数的字节数 格式为_functionname@number

- fastcall调用约定是人如其名 它的主要特点就是快 因为它是用过寄存器来传参数的(实际上 它用ecx和edx传送前两个双字(DWORD)或更小的参数(或若干个)) 剩下的参数仍旧自右向左压栈传送 被调用的函数在返回前清理传送参数的内存栈 在函数名修饰约定方面 它和前两个均不同 VC将函数编译后会在函数名前面加上@前缀 在函数名后面加上@和参数字节数 fastcall调用约定在输出函数名前加一个@符号 后面也是一个@符号和其参数的字节数 格式为@funcionname@number
- thiscall仅仅应用于C++成员函数 this指针存放于ECX寄存器 参数从右往左压 thiscall不是关键词 因此不能被程序员指定 thiscall是为了解决类成员调用this指针传递而规定的 thiscall要求把this指针放在特定寄存器中 该寄存器由编译器决定 VC使用ecx Borland的C++编译器使用eax 返回值和stdcall相当

如果不使用API 使用库函数来获取文件大小

```
int GetFileSize(char* filename)
{
    FILE* fp = fopen(filename, "r");
    if (!fp)
    {
        return -1;
    }
    fseek(fp, 0L, SEEK_END);
    int size = ftell(fp);
    fclose(fp);
    return size;
}
```

有下面的结构体声明 分别指出其在32位和64位平台下的sizeof大小 该结构成员的声明顺序是否有问题 如果有 是优点还是缺点 请说明具体原因

```
struct AAA
    {
        char* a;
        int b;
        char c;
        void* d;
        short e;
        int f;
    };
//32位下的内存布局
//a a a a b b b b
//c 0 0 0 d d d d
//e e 0 0 f f f f
//64位下指针占8个字节 其他不变
```

在默认对齐的情况下

32位 sizeof(AAA)=24

64位 sizeof(AAA)=32

缺点：数组元素声明一般按照从大到小的顺序声明 避免产生空隙 上面的定义违反了此规定 一定程度上会影响读写效率

Windows编程

请写出windows32位平台上下面数据类型的表示范围

```
USHORT      0-0xFFFF;
LONG        -0x80000000-0x7FFFFFFF;
DWORD       0-0xFFFFFFFF;
```

简述临界区，互斥量，信号量，事件的区别

临界区：同一个进程内，实现互斥 速度快 用的人最多

互斥量：可以跨进程，实现互斥 主要用于防双开

信号量：主要是实现同步，可以跨进程

事件：实现同步，可以跨进程 最自由

GastLastError函数中是如何实现各线程间互不干扰的 请说明其原理

采用的是操作系统提供的TLS线程局部存储基址 TLS相关的API有

```
TlsAlloc
TlsGetValue
TlsSetValue
TlsFree
```

简述一下如何截取键盘的响应 让所有的'a'变成'b'

使用键盘钩子:`SetWindowsHookEx`

简述一下常用的进程通信方法有哪些

文件映射 共享内存 管道 邮槽 剪贴板 动态链接库 NetBios函数 WM_COPYDATA消息 文件 网络

简述一下PostMessage与SendMessage有何区别

PostMessage只是把消息放入队列 不管其他程序是否处理都返回 然后继续执行 返回值表示函数执行是否正确

而SendMessage必须等待其他程序处理消息后才返回 继续执行 返回值表示其他程序处理消息后的返回值

数据结构和算法

二叉树的遍历

前序 中序 后序 层序

快速排序

原理：

1. 在待排序的元素任取一个元素作为基准(通常选取第一个元素 但最好的选择方法是从待排序元素中随机选取一个作为基准) 称为基准元素
2. 将待排序的元素进行分区 比基准元素大的元素在它右边 比其小的在它左边
3. 对左右两个分区重复以上步骤直到所有元素都是有序的

推荐文章：`https://www.cnblogs.com/MOBIN/p/4681369.html`

代码实现：

```
//递归对数组进行排序 L表示最左边的下标 R表示最右边的下标
void QuickSort(int arr[], int L, int R)
{
    //设置左右指针
    int i = L;
    int j = R;
    //设置基准值(数组最中间的数)
    int pivot = arr[(L + R) / 2];

    //当左指针小于或等于右指针时 一直循环
    while (i <= j)
    {
        //当左指针小于基准值 指针继续右移
        while (arr[i] < pivot)
        {
            i++;
        }

        //当右指针大于基准值 指针继续左移
        while (arr[j] > pivot)
        {
            j--;
        }

        //当左指针小于或等于右指针时 对数值进行交换
        if (i<=j)
        {
            //如果找到符合条件的数 就交换两个数 并且左右指针移动
            swap(arr[i], arr[j]);
            i++;
            j--;
        }
    }

    //再次对两边的数组做快速排序
    if (L<j)
    {

        QuickSort(arr, L, j);
    }
    if (i<R)
    {
        QuickSort(arr, i, R);
```

```
    }

}

int main()
{
    int arr[] = { 4,7,2,6,5,1,3,9,0 };
    QuickSort(arr,0,8);
    for (int i = 0; i < 9; i++)
    {
        printf("%d", arr[i]);
    }
    system("pause");
}
```

冒泡排序

```
void BubbleSort(int*a,int len)
{
    for (int i = 0; i < len-1; i++)     //外层循环表示要进行几轮排序 循环的次数为n-1次
    {
        //j表示最后一个元素 由j开始往前排序
        //j>i为循环的终止条件   当i=0时 对整个数组进行循环判断
        //当i=1时 不对第一个元素进行循环比较 因为第一个已经是最小的了
        for (int j = len-1; j>i ; j--)  //内层循环表示每一轮的排序
        {
            //如果前一个元素大于后一个元素 则交换两个数
            if (a[j-1]>a[j])
            {
                swap(a[j - 1], a[j]);
            }
        }
    }

}
int main()
{
    int arr[] = { 4,7,2,6,5,1,3,9,0 };
    BubbleSort(arr,9);
    for (int i = 0; i < 9; i++)
    {
        printf("%d", arr[i]);
    }
    system("pause");
}
```

你所了解的字符串匹配的算法有哪些?

- 暴力匹配算法
- KPM算法
- SUNDAY算法
- Horspool算法

KMP算法推荐视频：
`https://www.bilibili.com/video/av11866460from=search&seid=18245460411724293019`

PE文件

PE文件的加载流程

1. 检测Dos头首字节是否是5A4D，PE头的首字节是否是4550
2. 读入PE文件的DOS头，PE头和Section头
3. 根据映像大小SizeOfImage申请内存，如果已被其他模块占用，则重新分配一块空间。
4. 根据Section头部的信息，把文件的各个Section映射到分配的空间,并根据各个Section定义的数据来修改所映射的页的属性。
5. 根据PE文件的导入表加载所需要的DLL到进程空间
6. 修复IAT。
7. 根据PE头内的数据生成初始化的堆和栈
8. 创建初始化线程，执行TLS回调函数，开始运行进程。

PE的常见区段，一个全局变量在哪个区段

data

为什么PE文件格式要用到RVA呢

为了减少PE加载器的负担 因为每个模块都可能被重载到任何虚拟地址空间，如果让PE加载器修正每个重定位项，这肯定是个巨大的工程量。相反，如果所有重定位项都使用RVA，那么PE加载器就不必操心那些东西，他只要将整个模块重定位到新的起始VA。

区段结束默认用0补齐 可以用其他内容补齐吗

可以 因为那段数据本身没有任何用处，只是为了占满文件空间

PE头可不可以放在最后一个区段？PE文件最多能有几个区段？

当然不行，PE加载器会从最开始判断文件是否是PE文件，如果将PE头放在最后一个区段，则Signature不会被识别。

256个 文件头中有一个NumberOfSections字段是区段数量

如何判断一个文件是PE文件

1. 看Dos头的第一个字段e_magic是否是5A4D
2. 看Nt头的第一个字段Signature是否是4550

PE文件大体结构是什么样子的

Dos头+Nt头+PE头+区段

怎么找到NT头

Dos头的e_magic+e_elfanew

PE文件默认加载基址保存在哪里

扩展头的ImageBase

扩展头后面跟的是什么?怎么找到这个位置?

区段头表 IMAGE_FIRST_SECTION

区段表中的VirtualAddress字段保存的是什么,PointToRawData呢?

区段装载到内存中的RVA 区段在磁盘文件的偏移

区段表的属性都有什么属性

1. 可读
2. 可写
3. 可执行
4. 包含可执行代码
5. 包含已初始化数据
6. 包含未初始化数据
7. 不能隐藏
8. 不可分页

区段的名字有什么作用？区段的名字都是固定的吗?

区段名方便我们识别编译器 名字不固定 由编译器决定 也可以修改

VirtualAddress和PointToRawData为什么会不一样?

一个是内存中的RVA 一个是文件偏移 因为内存对齐粒度和文件对齐粒度不一样

什么是RVA？什么是Offset?转换关系是什么样的?

RVA:相对于加载基址的偏移

Offset:文件偏移

FOA=需要转换的RVA-所在区段的RVA+所在区段的FOA

区段表中，区段文件大小和内存中的大小，哪个是对齐过的?

内存中的大小是经过对齐的 VirtualSize是实际的 没有经过对齐的实际大小

导出表的作用是什么? 没有它exe能运行吗?

作用:记录了导出符号的地址,名称,与序号

给其他PE文件提供导出函数 没有导出表exe依旧可以运行

导入表的作用是什么? 没有它exe能运行吗?

记录了一个exe或者一个dll所用到的其他模块导出的函数

所记录的信息有:用了哪些模块(用了哪些dll),用了dll的哪些函数

没有导入表程序无法运行

怎么才知道导出函数是仅以序号导出还是以名称导出?

遍历序号表 判断地址表的下标有没有存在于序号表中 存在就说明是以名称导出,不存在就说明是以序号导出

如何判断导入函数是以序号导入还是以名称导入?

在IMAGE_THUNK_DATA32这个结构体(结构体里面就是一个联合体,大小为32位)里面,判断结构体字段中的最高位, 最高位为1:以序号导入 最高位为0:以名称导入

已知函数名，如果我们想要使用代码找到某一个函数的地址，该怎么做?

使用LoadLibrary + GetProcAddress

什么是IAT？里面存储的什么?

IAT导入函数地址表 在文件中存储的是函数名 在内存中存储的是函数地址

什么是INT？里面存储的什么?

INT导入函数名称表 在文件中存储的是函数名 在内存中存储的也是函数名

怎么才能知道一个exe都使用了哪些API?

查看exe的导入表

重定位有什么作用?

生成程序的时候,很多涉及到地址的代码,都使用一个绝对的虚拟内存地址 但是当程序的加载基址产生变化的时候,新的加载基址和默认的加载基址就不一样了 那些涉及到地址的代码就不能运行了,此时就需要将那些涉及到地址的代码,把他们的操作数修改(去掉默认加载基址,再加上新的加载基址)才能够使程序运行起来.

哪些PE文件需要重定位?

1. 当代码段使用了其他区段的数据,所生成的代码就会产生重定位的需求,其他段数据的首地址,就是产生重定位需求的根源
2. 凡是出现全局变量的地方,都会导致重定位的产生
3. 调用外部模块的函数都会产生重定位

怎么判断PE是DLL,还是EXE

文件头里面有一个文件值属性Characteristics ,dll和exe是不同的!

怎么判断PE文件是32位还是64位

通过文件头里面有一个值:扩展头大小:32位程序是E0,64位程序一般是F0

如何去掉一个程序的重定位

去掉随机基址 可选头的DllCharacteristics

为什么要修复重定位 如何修复重定位?

在生成程序的时候,很多涉及到地址的代码,都使用一个绝对的虚拟内存地址(这个虚拟内存地址是假设程序加载到0x400000的地方时才能够使用的),但是当程序的加载基址产生变化的时候,新的加载基址和默认的加载基址就不一样了,那些涉及到地址的代码就不能运行了,此时就需要将那些涉及到地址的代码,把他们的操作数修改(去掉默认加载基址,再加上新的加载基址)才能够使程序运行起来.

重定位公式：需要重定位的地址-默认加载基址+实际加载基址

需要重定位的数据所在位置=实际加载基址+VirtualAddress+重定位数据的偏移

说出你熟知的windows资源

- 对话框
- 图标
- 光标
- Bitmap
- 菜单

windows资源表有几层结构

三层

资源表每一层结构分别代表什么含义?

一层：有多少种资源；二层：每种资源有多少个，名称； 三层：资源数据

软件逆向与汇编

调试器是如何实现栈回溯的

目前的主流CPU架构都是使用栈来进行函数调用的，栈上记录了函数的返回地址，因此，通过递归是寻找放在栈上的函数返回地址，便可以追溯出当前线程的函数调用序列，这便是栈回溯的基本原理。

Exeinfo PE 和 PEiD两个工具的 原理

特征码识别

OD用过什么插件 StrongOD常用选项 跳过哪些异常

- API Break
- 异常计数器
- StrongOD
- 中文搜索引擎
- LoadMapEx

如何调试没有源代码的驱动 SXE命令 驱动断点位置

使用PE工具查看驱动程序的入口点偏移 然后在入口函数处下断点

IDA F5不能正确识别函数，怎么解决(花指令 混淆的处理方法)

- 利用OD的去除花指令插件

- 在IDA中找到流氓指令 手动去除，去除方式，将数据修改为代码或者将代码修改为数据

IDA重设基址的方法

菜单->Edit->Segments->ReBase Program

已经执行的程序 如何从头重新调试

找到头部，右键 此处为新的EIP

如何调试服务

1. 在调试配置中生成服务
2. 安装服务
3. 启动服务
4. 使用管理员权限启动VS
5. 附加服务
6. 在代码中设置断点
7. 访问服务管理器并操作服务

DLL文件如何调试

1. 使用OD的LoadDll.exe
2. 自己写一个程序，用LoadLibrary+GetProcAddress加载目标DLL，然后调试自己的程序

CALL和FF15 FF25有什么区别

FF15是call FF25是jmp E8也是call E9是jmp

FF15是通过跳板过去，假设在0x450000的位置存储着0x500000的地址，那么FF15 0x450000就会直接跳到0x500000那个地址，

call是通过两个地址间的偏移来跳转的。E8 目标地址-当前地址-5

反虚拟机技术都有哪些

- 检测特定的文件夹或文件信息
- 检测当前进程信息
- 检测特定的服务名
- 检测特定的注册表信息
- 通过特权指令检测虚拟机
- 利用IDT基址检测虚拟机
- 利用LDT和GDT
- 基于时间差检测虚拟机
- 利用IO端口检测虚拟机

各种反调试技术

- BeingDebugg反调试
- NtQueryInformationProcess反调试
- 进程状态检测

- SedebugPrivilege
- 调试环境检测
- 注册表检测
- 窗口检测
- 父进程检测
- 进程扫描
- INT3 扫描
- MD5校验
- 时钟检测
- TLS反调试
- 利用SEH清除硬件断点

分析过什么程序

分析过一些CrackMe 还有扫雷游戏 做过扫雷外挂

分析过010Edit的注册算法 先是暴力破解 然后对其算法进行了分析 算出了正确的序列号

分析过勒索病毒 远控木马 office宏病毒

分析过android的APK 反编译smali代码

32位windows上函数调用 参数压栈方式 堆栈平衡方式有几种？为什么要那么多种？简单说说每种的特点

- API使用的方式 _stdcall,参数压栈的方式是从右到左 堆栈平衡的方式是call内平衡
- _cdcel,参数压栈的方式是从右到左 堆栈平衡的方式是call外平衡
- X64 delphi使用方式,fastcall 参数压栈方式是x64前四个参数使用寄存器 其余使用push 从右到左 delphi前2个参数使用寄存器 其余使用push 从右到左
- C++使用的方式 thiscall 参数压栈方式是从右到左 ECX传递对象this指针 其他和stdcall一样

产生这么多方式的原因有两个：

1. 语言不同
2. 编译器不同 程序编译的兼容性 编译器为了提供程序兼容性 会继承不同的调用约定方式

描述一下反汇编后的switch是怎么进行的

1. 当case语句较少时，反汇编和if else相同 都是通过cmp和JCC指令实现跳转 这个时候switch和if else的效率相同，这个是普通结构
2. 当case语句较多的时候 会采用跳转表来编译switch case语句 此时switch效率要比if else高，这个是大表结构
3. 当间隔较多且case语句较多时，会采用大表+小表结构
4. 树形结构

IDA和OD在逆向的区别 其中如何在OD中定位win32程序的main函数

IDA:一般作为静态分析工具 可以反编译成C代码 看代码逻辑更清晰 并且支持各种平台的文件格式 Android C++程序可以使用IDA动态调试分析

OD：一般作为动态分析工具 可以动态调试汇编代码 做运行时分析 只能在windows平台进行操作

如果分析一个杀毒软件的杀毒引擎 有什么思路

如果分析杀毒引擎 应用层的话从扫描器开始分析 也就是从文件扫描这块入手 在文件操作的API下断 CreateFileA/W ReadFile,跟踪扫描流程 驱动层的话 先使用ARK工具PCHunter定位杀毒引擎驱动做了哪些事 看看有没有SSDT HOOK IDT HOOK inline HOOK等 确定杀毒引擎的驱动文件 用IDA静态分析所需要的功能 用windbg双机调试驱动程序 动态观察驱动的运行

如果写一个远控软件 有什么思路

远程控制软件其实就是管理木马的软件 所以编写远控 我会根据实现木马的功能 然后去编写远程控制软件 常见的功能就是建立网络连接 发送cmd命令 上传下载文件 如果时间有限 我会根据网络中比较流行的远控 比如大灰狼远控 Gh0st远控 根据他们的架构进行设计编写客户端 服务端也就是木马 我会根据需求来编写 功能少的比如只有网络连接 下载功能 我会用汇编语言编写 要是功能多 我会使用C语言来编写框架 用汇编语言实现具体功能 尽可能使用shellcode实现每一个功能 方便做免杀

常见加解密算法了解多少

基本的编码算法有base64,http/https加密传输的时候都会使用base64编码加密数据 转为字符串进行传输 base64的特征是字母和数字结合的一串字符串 末尾一般都等号 常见的加密算法分为两种 一种是对称加密 一种是非对称加密

对称加密一般都是通过异或 循环加密 位移 置换等这些操作进行加密的 非对称加密使用的是数学难题 正向求解容易 逆向推理难

常见的对称加密算法有DES,3DES,TEA.AES 常见的非对称加密算法有RSA，椭圆曲线

逆向分析时一般分析算法是先使用插件识别程序中的编译常量来确定算法 插件有PEID的算法插件 yara规则算法特征库 IDA的算法插件 然后定位代码 再动态调试分析算法 然后根据具体的算法实现逻辑找算法的key或是其他关键的部件 进行深入分析

od和ida的反汇编引擎有什么区别?

OD的反汇编引擎算法使用的是线性扫描，Ctrl+A功能使用的算法是递归下降

IDA反汇编引擎算法使用的是递归下降

反汇编引擎工作原理?

OpCode查表

怎么写一个注册机？如果是网络验证如何破解?

首先将程序的注册算法分析出来 然后根据注册的算法推出注册机

找到网络验证的地方 直接暴力破解

IDA脚本是否用过

在破解反虚拟机的时候用过一个搜索反虚拟机指令的python脚本

如何确定汇编指令的长度

1. 没有操作数的指令，指令长度为1个字节
2. 操作数只涉及寄存器的的指令，指令长度为2个字节 如：mov bx,ax
3. 操作数涉及内存地址的指令，指令长度为3个字节 如：mov ax,ds:[bx+si+idata]

4. 操作数涉及立即数的指令，指令长度为： 寄存器类型+1 8位寄存器，寄存器类型=1，如：mov al,8；指令长度为2个字节 16位寄存器，寄存器类型=2，如：mov ax,8；指令长度为3个字节
5. 跳转指令，分为2种情况：
    1. 段内跳转（指令长度为2个字节或3个字节）
        - jmp指令本身占1个字节
        - 段内短转移，8位位移量占一个字节，加上jmp指令一个字节，整条指令占2个字节
        - 如：jmp short opr
        - 段内近转移，16位位移量占两个字节，加上jmp指令一个字节，整条指令占3个字节
        - 如：jmp near ptr opr
    2. 段间跳转，指令长度为5个字节
        - 如：jmp dword ptr table[bx][di]
        - 或 jmp far ptr opr
        - 或 jmp dword ptr opr

给你一个外挂，如何逆向获得里面的功能?

根据外挂的快捷键和按钮一个一个的去定位到所有的响应事件 然后开始逆向 比如说F1是秒杀 那么可以从F1的响应事件开始入手 逐个逆向分析

假如调用了一个API，那么在调用API处，OPCODE与汇编指令的形式是什么样的?

首先在调用API之前会有几个push用于传递参数 然后会有一个E8 也就是call用于调用API

汇编指令call和ret的具体操作是什么

RET等价于将栈顶地址pop出来 然后再jmp到这个地址

CALL等价于将CALL指令下一条地址push到栈中 然后再jmp到CALL后面要跳转的地址中

PE文件是什么 属于PE文件的扩展名有哪些

PE文件是windows下的可执行文件格式 扩展名exe,dll,sys,ocx,com都是PE文件格式

OD中常规断点的快捷键是什么 并简要说明常规断点与内存断点的实现原理

OD常规断点的快捷键是F2 类似VS中的F9

常规断点 即是int3断点 机器码为CC 通过简单指令临时替换实现 中断下来后再恢复原有指令 API函数DebugBreak内部即是int3 要特别说明的是int3指令是专为调试而设计的中断指令 只有1字节大小 可以替换任意指令

内存断点 在访问内存时进行中断 通过修改内存属性来实现 当对指定内存访问时会引起访问违规 系统发送调试事件给调试器 实现断点机制

简述一个OFFSET与LEA的区别

- lea是机器指令 offset是伪指令
- lea是在运行时执行 而ofset是在编译时执行
- lea可以获取由寄存器组成的表达式地址 可以进行比较复杂的运算
- offset只能取得用"数据定义伪指令"定义的变量的有效地址 不能取得一般操作数的有效地址

简述一下Intel CPU体系架构是如何实现硬件断点 且相对于软件断点的优势

硬件断点原理是使用4个调试寄存器(DR0 DR1 DR2 DR3)来设定地址 以及DR7设定状态 硬件断点的优点是速度快 在INT3断点容易被发现的地方 使用硬件断点来替代会有很好的效果 缺点就是最多只能设置4个断点

80x86 16位CPU有四个段寄存器 它们有什么作用

四个段寄存器分别为：CS,DS,ES,SS

CS:代码段寄存器， 装代码段的起始地址

DS:数据段寄存器，装数据段的起始地址

ES:附加段寄存器，装附加段的起始地址

SS:堆栈段寄存器，装堆栈段的起始地址

病毒分析

病毒分析流程

静态分析

- 查壳
- 查看字符串
- 查看导入表
- 查看资源断
- 使用PEiD插件扫描加密和压缩算法

动态分析

- 使用火绒剑监控病毒行为，文件操作 网络操作 注册表操作 进程操作
- 使用regshot监控注册表
- OD动态跟踪

假设有1000个样本，怎么分析，识别黑白?

如果可以上传的话，我可以写一个工具，批量上传到查毒软件，然后分析其结果

不让上传的话，自己搭建沙箱环境，在写自动分析工具

常见的病毒类型有哪些

木马 后门 蠕虫 感染性病毒 勒索病毒 APT木马 RootKit

傀儡进程的实现原理

傀儡进程的创建原理就是修改某一进程的内存数据，向内存数据写入Shellcode代码，并修改该进程的执行流程，使其执行Shellcode代码。这样，进程还是原来的进程，但执行的操作却替换了。

BadRabbit执行流程

1. 释放infub.dat通过rundll32.exe运行

2. 提取权限 检测杀软
3. 保存代码到堆空间更改执行流程 删除自身 重新执行
4. 检测cscc.dat是否存在
5. 根据平台释放7/8号资源cscc.dat
6. 释放9号资源dispci.exe 添加计划任务运行
    1. 重启后连接驱动层
    2. 创建计划任务
    3. 加密MBR
    4. 关机
7. 创建关机计划任务
8. 创建线程
9. 根据平台释放1/2号资源 tmp文件
10. 执行tmp文件
11. 创建线程 局域网弱口令传播
12. 创建线程 加密文件
13. Sleep关机等待

API Monitor什么功能

- 监视和控制应用程序和服务，能让你看到应用程序是如何工作的
- 能显示API的调用信息和调用层次结构，API的定义，能让你通过API调用设置断点控制目标应用程序，解码和显示2000种不同的结构体和联合体
- 能查看输入和输出缓冲区

手工查杀病毒一般步骤有哪些

1. 使用ARK工具检查各种启动项有无可疑文件
2. 使用ARK工具检查运行的进程有无可疑进程
3. 检查敏感目录看是否有可疑文件 尤其是隐藏的系统文件
4. 基本上以上三步可以找出目标 如果不行 需要抓包查看是否有可疑网络连接和流量
5. 当发现之后 先将样本提取出来 备份 然后清理文件 进程 注册表等

DES密钥长度有多少位 多少个字节 多少个有效位

64位 8个字节 每个7位会设置一个用于错误检查的位 所以有效位是56位

MD5和SHA的长度

MD5输出128bit

SHA1(常用):输出160bit,对于长度小于2^64位的消息,SHA1会产生一个160位消息摘要,在传输过程中,数据如果产生变化, 就会产生不同的消息摘要!

对僵尸，蠕虫，木马有什么了解，说下理解，给你一个病毒你是怎么判断是这三种病毒

- **蠕虫**——这种程序具有自我复制的功能，唯一的目的就是为了扩大传播范围。通过互联网或者存储介质扩散到另一台计算机。不会对计算机造成损害，自我复制的特性会耗用硬盘空间，从而减慢计算机的运行速度
- **病毒**——他们也具有自我复制的能力，不过确实会破坏它们攻击的计算机上的文件，他们的主要弱点在于，只有得到宿主程序的支持，它们才能起作用
- **特洛伊木马**——基本上来讲，特洛伊木马不是病毒，并非旨在破坏或删除系统上。他们的唯一任务就是开一道后门，以便恶意程序在你不知情 未经允许的情况下，偷偷进入你的系统，窃取你的数据
- **广告软件**——广告软件用来显示程序中的广告。 他们一般依附在免费使用的软件程序上，因为广告软件是开发那些软件程序的人员唯一的收入来源。
- **间谍软件**——这种程序也依附在其他免费软件上，跟踪你的浏览及其他个人信息，并将其发送给远程用户。它们还便于安装来自互联网的用户不需要的软件。不像广告软件，间谍软件作为独立程序来运行，悄然行事
- **僵尸程序**——僵尸程序或机器人程序是自动化进程，旨在通过互联网来进行联系，根本不需要人的干预。它们可以用于正道，也可以用于歪路。不法分子可以构建一个恶意的僵尸程序，能够独自感染宿主。僵尸程序传播到宿主设备后与中央服务器建立起连接；在连接到该网络（名为僵尸网络）的被感染宿主看来，中央服务器充当控制中心。
- **勒索软件**：——这种类型的恶意软件改变计算机的正常运行，因而让你无法正常使用。之后，这种程序会显示警示信息，索要钱财，如数交钱后你的设备才会恢复到正常的工作状态

常见的病毒行为或是特征有哪些

1. 注册表增加启动项 开始菜单增加启动项
2. 注册系统服务
3. 修改释放的exe为隐藏文件 系统文件
4. 加密文件 敲诈勒索
5. 感染exe
6. 释放文件
7. 修改MBR扇区
8. 伪装成系统进程

勒索病毒 是用的什么加密方式?

一般都是用AES+RSA混合加密，也叫数字信封

杀毒软件源码看过没有?

看过一小部分百度雪狼杀毒引擎的源码

推荐文章：`https://www.freebuf.com/articles/system/53021.html`

DLL劫持原理

如果进程尝试加载一个DLL的时候，没有指定DLL的绝对路径，那么Windows会尝试去指定目录下查找这个DLL，如果攻击者能够控制其中一个目录，并放一个恶意的DLL到这个目录下，那么这个恶意的DLL便会由进程加载，进程会执行DLL 中的代码，这就是所谓的DLL劫持

Windows病毒常用自启动方式 注册表有哪些启动项？注册表启动项由谁来启动？计划任务由哪个进程启动?

1. 注册表
2. 快速启动目录
3. 计划任务

4. 系统服务

注册表启动进程——Winlogon.exe

简单谈一谈主动防御

主动防御技术是指对未知病毒的防范 在没有获得病毒样本之前阻止病毒的入侵

传统的杀毒流程：病毒产生——研究病毒源码——升级或者产生反病毒工具——抵御病毒

主动防御：通过HOOK系统建立进程的API 杀毒软件就在一个进程建立前对进程的代码进行扫描 如果发现敏感操作就提示 如果用户放行 就让程序继续运行 接下来监视进程调用API的情况 如果发现一些敏感的API就发出警告 比如以读写的方式打开一个EXE文件 可能进程的线程想感染文件 再比如CreateRemoteThread

yara规则的优点与缺点?

如果有足够的亚规则 可以匹配很多东西 可以匹配病毒 漏洞 PE文件等等 只要是二进制或者十六进制的都行

支持的细节不到位而且很麻烦 假设说我有三个病毒 那么我就不能用一条亚规则匹配三个病毒 而且通过亚规则并不能够知道病毒的细节 只能够知道这是一个病毒

反病毒引擎扫描的工作方式

主要依靠一个已知恶意代码可识别片段的特征数据库（病毒文件特征库），以及基于行为与模式匹配的分析（启发式检测），来识别可疑文件

调试与异常

没有调试器如何处理异常 VEH VCH区别 VEH位置 SEH栈中结构 成员字段

VEH是向量化异常 属于全局，是进程相关

SEH是结构化异常属于局部 是线程相关 结构是链表

异常分发流程

用户层异常处理流程

1. KeContextFromKframs将Trap_frame备份到context为返回3环做准备
2. 判断先前模式 0是内核层调用 1是用户层调用
3. 是否是第一次调用
4. 是否有内核调试器
5. 发送调试信息给三环调试器 通过调试子系统 调试子系统通过SendApiMessage发送给三环调试器
6. 如果三环调试器没有处理这个异常 那么准备返回三环 开始修改Trap_frame结构体里的值
7. 最关键的修改 修改EIP为KiUserExceptionDispatcher 即回到三环什么地方
8. KiUserExceptionDispatcher函数执行结束 返回调用的位置CPU异常和模拟异常返回的地点不同
9. 无论哪种方式 当线程再次回到三环时 将执行KiUserExceptionDispatcher函数

内核层异常处理流程

1. 调用_KeContextFromKframes 将Trap_fram备份到context为返回3环做准备
2. 判断先前模式 PreViousMode 0是内核调用 1是用户层调用
3. 判断是是否是第一次调用 不为1则跳转 为1则继续往下走 通过函数的最后一个参数

4. 判断一个全局变量_**KiDebugRoutine** 看内核调试器是否存在 有的话 这个值是非零的 如果没有内核调试器直接跳转
5. 有内核调试器会先调用内核调试器 如果调用返回1说明调用成功 说明内核调试器已经处理了 就将Context再转成Trap_Frame直接返回 如果调试器没有处理 就接着让3环处理 如果内核调试器没有处理 也就是返回失败 直接跳转
6. 如果没有内核调试器 或者内核调试器没有处理 会调用RtlDispatchException函数 这个函数专门负责处理异常 功能是负责调用异常处理函数
7. 再次判断是否有内核调试器 有调用 没有直接蓝屏

HOOK和注入

输入法注入的原理

输入法注入:利用Windows系统中在切换输入字符时,系统就会把这个输入法所需要的ime文件装载到当前进程中,而由于这个ime文件本质上就是存放在C:\windows\System32目录下的特殊的dll文件,因此我们可以利用这个特性,在IME文件中使用Loadlibrary()函数替代注入的dll文件

Inline Hook的实现流程

用MessageBoxA做例子

- 开启HOOK

1. 获取MessageBoxA在user32.dll里的函数地址
2. 保存MessageBoxA函数的前五个字节
3. 获得进程ID
4. 修改进程MessageBoxA的前五个字节的属性为可写
5. 修改进程MessageBoxA的前五个字节为JMP到MyMessageBoxA
6. 修改进程MessageBoxA的前五个字节的属性为原来的属性

- 卸载HOOK

1. 修改进程MessageBoxA的前五个字节的属性为可写
2. 修改进程MessageBoxA的前五个字节为原来的五个字节
3. 修改进程MessageBoxA的前五个字节的属性为原来的属性

简述ssdt hook

SSDT是最常见的内核层 HOOK。SSDT的全称是系统服务描述符表(System Services Descriptor Table)。SSDT是关联ring3的 Win32 API和ring0的内核API的重要数据结构，它存储着Windows把需要调用的所有内核API地址。 SSDT HOOK原理是将内核层 API地址修改为指向其位于Ring0层的驱动入口，这样每次系统执行到这个函数时，都会通过SSDT表将原始调用引向修改后的模块中

什么是远程注入？还有哪些注入方式?

远程注入是将一个dll注入到目标进程的技术 除此之外还有

- APC注入
- 注册表注入
- 全局钩子注入
- 输入法注入

零环的HOOK都有哪些，如何实现?

1. SSDT HOOK
    - SSDT是最常见的内核层 HOOK。SSDT的全称是系统服务描述符表(System Services Descriptor Table)。SSDT是关联ring3的 Win32 API和ring0的内核API的重要数据结构，它存储着Windows把需要调用的所有内核API地址。 SSDT HOOK原理是将内核层 API地址修改为指向其位于Ring0层的驱动入口，这样每次系统执行到这个函数时，都会通过SSDT表将原始调用引向修改后的模块中
2. IDT HOOK
3. SYSENTRY HOOK
    - SYSENTRY是Windows XP之后的操作系统进入ring0的函数。Win2000中通过 int2e系统调用机制，涉及到的 Interrupt/Exception Handler的调用都是通过 call/trap/task这一类的gate来实现的，这种方式会进行栈切换，并且系统 栈的地址等信息由TSS提供，可能会引起多次内存访问 （来获取这些切换信息），系统开销较大。SYSENTER通过汇编指令实现快速系统调用机制。 SYSENTER HOOK的原理是首先Ntdll 加载相应的请求服务号到EAX 寄存器中，同时EDX 寄存器存贮当前的栈指针ESP，然后Ntdll发出SYSENTER 指令，该指令转移控制权到寄存器IA32_SYSENTER_EIP 存贮的地址中[21]，通过修改这个地址，可实现相应的挂接
4. Inline HOOK
    - inline hook是直接在以前的函数体内修改指令，用一个跳转或者其他指令来实现挂钩的目的。 而普通的hook只是修改函数的调用地址，而不是在原来的函数体里面做修改。 Inline hook原理是解析函数开头的几条指令，把他们Copy到数组保存起来，然后用一个调用我们的函数的几条指令来替换，如果要执行原函数，则在我们函数处理完毕，再执行我们保存起来的开头几条指令，然后调回我们取指令之后的地址执行。它需要在程序中嵌入汇编代码（Inline Assembly）以操作堆栈和执行内核API对应的部分汇编指令。
5. OBJECT HOOK
    - OBJECT HOOK是相对于IAT HOOK之类的 API HOOK而言，API HOOK是挂钩应用层函数，而OBJECT HOOK是挂钩内核层函数。其原理与API HOOK类似。
6. IRP HOOK
    - IRP是 I/O request packets，在Windows中几乎所有的I/O都是通过包（packet）驱动的，每个单独的I/O由一个工作命令描述，此命令将会告诉驱动程序需要一些什么操作，并通过I/O子系统跟踪处理过程。这些工作命令就表现为一个个被称为IRP的数据结构。 IRP HOOK原理是拦截管理器发向文件或网络系统等驱动程序的IRP。一般通过创建一个上层过滤器设备对象并将之加入系统 设备所在的设备堆栈中。也有部分IRP HOOK通过拦截传递IRP请求包的函数IofCallDriver或MajorFunction函数表来实现的

什么是DLL注入技术?

DLL注入简单来说就是将代码插入/注入到正在运行的进程中的过程 注入的代码是动态链接库（DLL）的形式

APC注入流程

1. 通过`OpenProcess`函数打开目标进程 获取目标进程句柄
2. 遍历线程 获取目标进程的所有线程ID
3. 使用`VirtualAllocEx`在目标进程中申请内存，并通过`WriteProcessMemory`函数向内存中写入DLL的绝对路径
4. 最后 遍历获取到的线程ID，并调用`OpenThread`函数打开线程拿到线程句柄，并调用`QueueUserAPC`函数向线程插入APC函数

远程线程注入流程

1. OpenProcess打开注入进程 获取进程句柄
2. 在注入进程中申请内存 向申请的内存中写入DLL的绝对路径
3. 获取LoadLibrary的函数地址
4. 创建远程线程，调用LoadLibrary，实现dll注入
5. 关闭句柄

远程线程注入的原理?

远程线程注入是使用关键函数CreateRemoteThread+LoadLibrary在其他进程空间中创建一个线程 并且执行DLL中的代码 因为CreateRemoteThread需要传一个函数地址和多线程参数 而LoadLibrary正好只有一个参数 所以只要将CreateRemoteThread+LoadLibrary结合 就能在目标进程注入DLL了

关键在于获取目标进程中某个DLL路径字符串的地址和LoadLibrary的地址

关于LoadLibrary的地址 虽然windows引入了随机基址的安全机制 但是系统的DLL的加载基址是固定的 所以Kernel32.dll里的LoadLibrary和自己程序空间的地址是一样的

关于DLL路径字符串的地址 可以直接调用VirtualAllocEx在目标进程空间申请内存 再调用WriteProcessMemory函数将指定的DLL路径写入目标进程空间就可以了

消息钩子注入的原理?

使用Windows提供的SetWindowsHookEx可以设置消息钩子，系统维护了一个钩子链。 设置完成之后，操作系统会自动帮你安装钩子到钩子链头部，当对应的事件发生的时候 系统就会先启动对应的回调函数，把DLL加载到发生事件的进程中，这样就实现了DLL注入。

什么是Hook技术?

Hook 技术又叫做钩子函数，在系统没有调用该函数之前，钩子程序就先捕获该消息，钩子函数先得到控制权，这时钩子函数既可以加工处理（改变）该函数的执行行为，还可以强制结束消息的传递。简单来说，就是把系统的程序拉出来变成我们自己执行代码片段

inline-Hook的原理 如何检测?

原理：修改函数的前五个字节为jmp XXX 跳转到自己的函数地址

检测函数的前五个字节是否为EB E9 EA

对比函数所在的PE文件在本地和内存中的数据

校验代码段的HASH值

IAT-Hook的原理 如何检测?

修改导入函数地址表中的地址 使调用该函数时跳转到自己的函数地址

检测方法：

1. 在程序运行之前枚举导入表 然后使程序运行 读取文件再次枚举导入表 比较两个表中的FOA和RVA
2. 查找导入表中INT的函数名称 使用LoadLibrary+GetProcAddress手动加载函数地址得到的是真正的函数地址 然后跟IAT表的函数地址做比较

漏洞相关

PCManFTP是什么类型漏洞，利用方法

缓冲区溢出漏洞 借助USER命令中的长字符串利用该漏洞执行任意代码

Office0158漏洞原理

Cve-2012-0158漏洞出现在MSCOMCTL.OCX，是微软Office系列办公软件在处理MSCOMCTL的ListView控件的时候由于检查失误，缓冲区的长度和验证的长度存在文件里，导致攻击者可以通过构造恶意的文档执行任意代码。

漏洞缓解技术，漏洞防范技术

GS:针对缓冲区溢出时覆盖函数返回地址这一特点,微软使用了GS这一编译选项

在所有函数发生调用的时候,向栈内压入一个额外的随机DWORD,如果使用IDA反汇编的话,IDA会将其标注为"SecurityCookie", Security Cookie位于EBP之前,,系统还将在.data的内存区域内存放一副本,当栈发生溢出的时候,SecurityCookie将会被首先淹没,之后才是EBP和返回地址,在函数返回之前,系统将执行一个额外的验证操作,被称作secruitycheck,在执行secruity check的过程中,系统将比较栈中原先存放的Security Cookie和Security Cookie副本,如果不吻合,说明栈帧中的Security Cookie已经被破坏,即栈中发生了溢出,系统立即进入异常处理流程,函数不会被正常返回,ret指令也不会被执行

Windows下常见的漏洞缓解技术有哪些

- GS(Control Stack Checking Calls)
- SafeSEH(Safe Structured Exception Handler)
- Heap Protection
- DEP(Data Execution Prevention)
- ASLR(Address Space Layout Randomization)

内核相关

进程和线程数据结构

进程和线程在内核中的结构分别叫做KPROCESS和KTHREAD

进程和线程在执行体层中的结构分别叫做EPROCESS* 和 ETHREAD

内核重载方法 有什么注意事项

1. 将内核文件加载到内存
2. 进行基址重定位
3. 重定位ssdt结构
4. Hook KiFastCallEntry，让RING3进程调用走新内核

简述DPL,CPL,RPL的含义 谈一谈你的理解

DPL是目标代码段的访问权限

CPL当前执行权限

RPL代码段权限

当段选择子切换的时候会对特权级别做检测

什么是段选择子 请详细描述段选择子的结构

是一个16位的数值 高13位为全局描述符表的索引 第三位为索引GDT LDT 低两位为RPL

在windbg中如何查看IDT 如何查看GDT

IDT:r idtr

GDT:r gdtr

设备通讯有三种方式 是哪三种 有什么区别

1. 缓冲区设备读写方式 将用户态缓冲区拷贝到内核态 在内核态使用完毕再拷贝回用户态
2. 直接读取方式 对用户态内存地址进程重新映射 映射到内核空间
3. 其他方式 具体方式取决于创建完设备对象后 设置其flags的值DO_BUFFERED_IO,DO_DIRECT_IO,0

请简述驱动对象 设备对象 IRP之间的关系

类似于程序 窗口 消息

每个驱动程序只有一个驱动对象(程序实例句柄) 每个对象可以对应一个或多个设备对象(窗口) 每个设备对象可以处理不同的IRP(消息)

什么是IRQL?

中断请求级别 用于不同的中断执行代码时 不会被低级别的中断代码所打断 一般情况下 非中断代码均运行在较低的级别

学习内核编程中你了解了哪些数据结构

- **EPROCESS:**执行体进程块 位于内核层 侧重于提供各种管理策略 同时为上层应用程序提供基本的功能接口
- **PEB：**进程环境块 位于进程地址空间(3环)的内存块 其中包含了有关进程地址空间的堆和模块等信息
- **KPROCESS:**内核进程块 EPROCESS结构体的第一个成员
- **ETHREAD:**执行体线程块 位于内核层上 包含了某个进程中的其中一个线程信息 可以通过EPROCESS的ThreadListHead域遍历该进程所有线程的ETHREAD信息
- **TEB:**线程环境块 位于3环的内存块 用于描述线程的相关信息
- **KPCR:**由于windows需要支持多个CPU 因为windows内核为此定义了一套以处理器控制区即KPCR为枢纽的数据结构 使每个CPU都有个KPCR，其中KPCR这个结构中有一个域KPRCB结构 这个结构扩展了KPCR 这两个结构用来保存与线程切换相关的全局信息 通常FS段寄存器在内核模式下指向KPCR 用户模式下指向TEB

3环函数进入0环我们有两种方式 请叙述是哪两种方式

1. 指令SYSENTERY 通过快速系统调用进入内核层
2. int 2E 通过中断门进入内核层

windows中,API调用通过syenter系统进入内核层后 eax中存放着什么 edx中存放着什么

eax为ssdt或shadowssdt的调用号 edx存放着当前的esp栈帧

请简述分页机制

分页机制在CPU看来是可选的 但是主流操作系统均使用了分页机制

在32位系统下 有两种分页机制

1. 未开启PAE 在未开启PAE的情况下 32位线性地址被分为三个部分：10-10-12 分别被称为页目录索引 页表索引 页内偏移 通过当前环境的CR3寄存器 能够访问到页目录表 按照三级索引能够一级一级的找到物理内存
2. 开启PAE：在开启了PAE的情况下 32位线性地址被分为四个部分 2-9-9-12 分别被称为页目录指针表索引 页目录索引 页表索引 按照四级索引能够一级一级的找到物理内存

在64位系统下 也有与之类似的模式

在各自向上索引的过程中 每个项的后12位为页的属性 包括：可读 可写 是否访问过 是否被修改过 是否只能是超级用户访问等属性 从而实现对页内存的保护

将CR0寄存器的第17位置0 可以关闭页保护

网络相关

Tcp和udp的区别，要实现一个简单的聊天程序，选那个?

1. 基于连接与无连接
2. 对系统资源的要求（TCP较多，UDP少）
3. UDP程序结构较简单；
4. 流模式与数据报模式
5. TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证

实现聊天程序选TCP

加壳与脱壳

###脱壳之后为什么要修复导入表

PE加载器的原理就是根据INT表的函数名字换成函数地址 当PE文件被壳压缩或加密之后 会破坏原本的导入表 所以脱壳之后要把导入名称表的函数名填回去 这样程序才能正常运行

介绍下自己写的壳的执行流程

1. 获取必要的API函数地址
2. 解密代码段
3. 恢复数据目录表
4. 修复重定位
5. 修复IAT
6. 增加反调试
7. 加密IAT
8. 调用TLS回调函数

手工加壳步骤

1. 修改文件头的NumberOfSections
2. 设置区段名(可选)
3. 设置 区段数据的实际字节数Misc.VirtualSize(可以和SizeOfRawData一样)
4. 设置区段的VirtualAddress(区段数据在内存中的RVA, 必选)

5. 设置 区段的PointerToRawData(区段数据在文件中的偏移,必选)
6. 设置区段以文件对齐粒度对齐后的大小SizeOfRawData(必选)
7. 设置区段属性
8. 插入区段数据
9. 修改扩展头中的映像大小

脱壳流程

1. 判断壳的类型
2. 寻找程序入口点（OEP） A. 根据跨段指令寻找 IDA Pro中有静态的跨段指令的视图（Segment），这里是动态的，基本的办法是单步跟踪。 B. 设置内存断点 在代码段释放完毕后，在代码段首部设置内存断点，中断后就是OEP C. 根据堆栈平衡原理 D. 根据编译语言的特点（常见的初始化函数）找OEP
3. Dump 内存
4. 修复 PE 文件头和输入表 IAT

安卓逆向

Android的四大组件是哪几个

活动 广播提供者 服务 内容提供者

Android系统架构有几层

1. 应用层
2. framework(应用框架层)
3. C++/虚拟机层
4. 硬件层
5. 内核驱动层

抓包工具用过吗?

用过wireshark(tcp/udp/http都支持) fiddler(http/https) Charles(http/https)

hook有哪些?

1. Android Java层hook是xposed修改app_process进程 注入到zygote 修改了执行流程 在调用java方法时执行自己的方法 完成Hook
2. Android C++层Hook有got hook inline hook got hook是hookso库加载时的got表(类似PE文件的导入表) inline hook修改函数内部的arm指令完成hook

dexhunter的原理

dexhunter是android二代壳的通用脱壳机

dexhunter原理 就是hook类加载的函数 defineClass defineClassNative 在dex文件内存加载时 在内存中完成对dex文件的重构

apktool的源码有看过吗?

使用git下载过 写dex解析的时候看过一小部分 APKtool源码有很多值得借鉴的地方 比如资源的解析 dex文件的解析 当遇到反编译出错时改源码才是王道

linker源码有阅读过吗?

了解过,和linux下的ld的源码,类似windows下的PE加载器 linker对so文件进行了加载和解释 linker源码非常值得阅读 因为要想写so加固和分析 so加固之后的程序 就需要对linker有一定的了解 就像windows写壳 如果对PE加载器不了解 不理解 很难写出壳

linux下遍历进程的函数是什么

linux下遍历进程使用的文件操作的函数 遍历了proc文件目录下的所有数字即可 如果想遍历详细的信息需要再读取数字目录下的文件 命令行信息是cmdline 状态信息是status 如果要读取进程内存需要使用ptrace附加进程 读取mem

android加固的原理?

最早的加固就是自己的dex替换源程序的 然后程序运行时动态解密加载源dex 一般是在自己实现的application中的attachBaseContext或者是onCreate方法中执行

android最早的入口函数是什么

一般是入口类中的onCreate方法 如果加固了是自己实现的application中的attachBaseContext方法 之后是application中的onCreate方法

android加固的特征是什么

不同厂商加固之后都会有对应的so文件 比如360加固 里面有libjiagu.so文件

so加固的特征是什么

会自己实现一个Loader 对so文件加壳 抹去导入信息 elf文件格式畸形化

制作一个手机助手 有什么思路

因为手机助手的功能基本上用adb都可以实现 所以猜测手机助手背后就是使用adb 从这一点开始分析adb 抓包分析 看看有没有执行一些adb命令 然后再下断点 CreateProcessA/W或者send 看看有没有创建cmd 执行命令或者是网络连接发送数据 然后再分析手机助手的安装文件夹 看看有什么动态库以及库的导出函数 根据这些再去进一步分析 等了解了手机助手的原理之后 再动手写代码

如果分析wifi共享精灵 有什么思路

如果分析WIFI共享精灵 我会从开启WIFI热点这个功能入手 先补充一下开启热点的知识 然后根据开启热点的流程中用到的API 然后下断开始分析 感觉基本的开启热点会访问网卡 建立服务器 网络相关的socket api肯定会用到 可以从这方面入手

其他

火绒剑的原理是什么?

Hook一些敏感API

是否了解powershell?

相当于cmd的升级版 既支持原本的cmd命令 也支持linux命令

操作系统启动过程

1. BIOS自检
2. 初始化阶段 执行MBR分区的引导代码
3. BOOT加载阶段，从启动分区加载ntldr
4. 检测和配置硬件阶段
5. 内核加载阶段
6. Windows的会话管理启动
7. 登陆阶段，启动服务子系统，启动本地授权，显示登陆界面