基于 Windows 平台调试器的设计与实现

作者单位: 武汉科锐

作者姓名: 黄　奇

指导老师: 钱 林 松

摘要:

科锐第三阶段项目之一

目录

前言

依然记得刚来科锐学习的时候，钱林松老师说: “程序不是写出来的，而是调出来的！“

回忆学生时代写程序的时候，遇到 bug 往往很无奈，常常是把程序从头到尾看一遍，或者是在一些关键的地方，觉得会出错的地方，printf 一下，来判断是不是这里出错了。曾经还觉得这个方法不错，沾沾自喜呢。

后来工作后，或多或少要调一些程序，但是总觉得不得要领。

所以我觉得无论是软件逆向工程的研究者还是开发人员 ,都有必要自己实现一个调试器 ,当你的调试器完成的时候 ,或许你对软件调试会有一种顿悟的感觉。

而且可以根据自己的需要，写一个适合自己的调试器，对于逆向或者调试程序来说，都会得心应手，而且还可以过针对某些调试器的反调试哦！

因鄙人没有天马行空的创意，亦没有巧夺天工的技术!所以调试器设计得也有很多不足，但也略有心得，故拿出来，与大家分享之，希望能帮助到有需要的人。

由于本人能力有限，文中必然有错漏之处，恳请读者不吝赐教。

第一章 调试器框架

1.1 框架的搭建

有写过 Windows 程序的人都知道 Windows 是基于消息的，当程序创建，窗口移动，键盘按下等，窗口过程函数都会收到消息，然后根据消息做相应的处理，其实调试器也是这样的,在 MSDN 中已经给出了一个现成的框架了,如下，我已经去掉部分注释了，如果想获得祥细信息的话可以查下 MSDN。

```
DEBUG_EVENT DebugEv;       // debugging event information
DWORD dwContinueStatus = DBG_CONTINUE; // exception continuation
for(;;)
{
// Wait for a debugging event to occur. The second parameter indicates
// that the function does not return until a debugging event occurs.
    WaitForDebugEvent(&DebugEv, INFINITE);
// Process the debugging event code.
    switch (DebugEv.dwDebugEventCode)
    {
        case EXCEPTION_DEBUG_EVENT:
        // Process the exception code. When handling
        // exceptions, remember to set the continuation
        // status parameter (dwContinueStatus). This value
```

```
// is used by the ContinueDebugEvent function.
    switch (DebugEv.u.Exception.ExceptionRecord.ExceptionCode)
    {
        case EXCEPTION_ACCESS_VIOLATION:
        case EXCEPTION_BREAKPOINT:
        case EXCEPTION_DATATYPE_MISALIGNMENT:
        case EXCEPTION_SINGLE_STEP:
        case DBG_CONTROL_C:
    }
case CREATE_THREAD_DEBUG_EVENT:
case CREATE_PROCESS_DEBUG_EVENT:
case EXIT_THREAD_DEBUG_EVENT:
// Display the thread's exit code.
case EXIT_PROCESS_DEBUG_EVENT:
// Display the process's exit code.
case LOAD_DLL_DEBUG_EVENT:
// Read the debugging information included in the newly
// loaded DLL.
case UNLOAD_DLL_DEBUG_EVENT:
// Display a message that the DLL has been unloaded.
case OUTPUT_DEBUG_STRING_EVENT:
// Display the output debugging string.
```

```
        }
    // Resume executing the thread that reported the debugging event.
    ContinueDebugEvent(DebugEv.dwProcessId,
        DebugEv.dwThreadId, dwContinueStatus);
    }
```

Windows 提供了一些 Debug API,他们已经实现了一般调试器所需的大部分功能，通过现有的框架和 windows 提供的 API 我们可以很容易的搭建起一个调试器的框架出来。

1.2 调试事件种类

调试事件	含义
EXCEPTION_ACCESS_VIOLATION	访问异常(目标内存分页权限不足 ,或者目标地址不存在)
EXCEPTION_BREAKPOINT	断点(Int3 断点)
EXCEPTION_SINGLE_STEP	单步(硬件断点或单步)
CREATE_THREAD_DEBUG_EVENT	线程创建
CREATE_PROCESS_DEBUG_EVENT	进程创建
EXIT_THREAD_DEBUG_EVENT	线程退出
EXIT_PROCESS_DEBUG_EVENT	进程退出
LOAD_DLL_DEBUG_EVENT	加载 DLL
UNLOAD_DLL_DEBUG_EVENT	卸载 DLL
OUTPUT_DEBUG_STRING_EVENT	输出调试字符串

1.3 调试事件处理

通过对上面几个事件的处理，基本可以实现我们的调试器了。当我们创建一个调试进程后，调试器会收到 CREATE_PROCESS_DEBUG_EVENT 事件，在这里我们做一些 PE 解析工作，比如取得 OEP 等等，然后在 OEP 处下一个 Int3 断点，处理完之后,将调用 ContinueDebugEvent 函数来继续线程的运行，这里要注意，这个函数的 dwContinueStatus 有二个取值，DBG_CONTINUE 和 DBG_EXCEPTION_NOT_HANDLED，如果这个异常我们可以处理则取 DBG_CONTINUE，否则取 DBG_EXCEPTION_NOT_HANDLED。DBG_EXCEPTION_NOT_HANDLED 则告诉操作系统，我的程序不能处理这个异常，给你处理，然后操作系统再问下调试进程能不能处理，如果不能的话，直接给干掉。

同时还有一点要注意，当创建一个调试进程的时候，系统会先调用一次 DebugBreak 函数，我们只要判断一下是不是系统调用的，是的话无视掉就行了。

接着调试器再调用 WaitForDebugEvent 函数等待调试事件的发生，直到接收到 EXIT_PROCESS_DEBUG_EVENT 事件时结束。

第二章 内存断点

2.1 问题的提出与分析

2.1.1 问题提出

因为内存断点是调试器的一个难点,所以我们这里放在最前面讲,这样也可以避免在后面的设计中一些组合问题的产生。

一个内存分页可能会存在多个断点，一个断点可能会跨多个内存分页。某个内存分页可能不具有用户要设的断点属性。某内存分页具有读与写的权限，但是用户只下了写的断点，如果将读的权限也一起拿掉，程序在读那块内存分页也要断下来，这样将大大的降低程序的效率，如何优化，以及断点信息与内存信息的存储。

2.2 问题的解决

2.2.1 内断断点与内存分页的关系

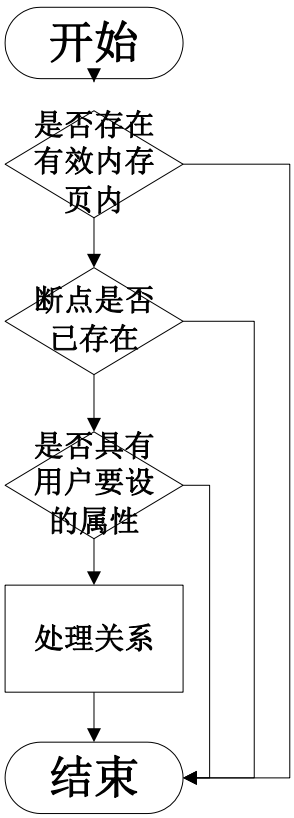
一个内存分页可能会存在多个断点，一个断点可能会跨多个内存分页。这个不由的让我们想起第一阶段的学生管理系统(一个学生可以选任意门课程，一门课程可能有任意个学生选)。其实二者的关系是一样的，因为是多对多的关系，所以我们要构造出第三张表来管理他们的关系。

为了检测用户下断点有效性等问题，在下断点的时候取得那个内页分页信

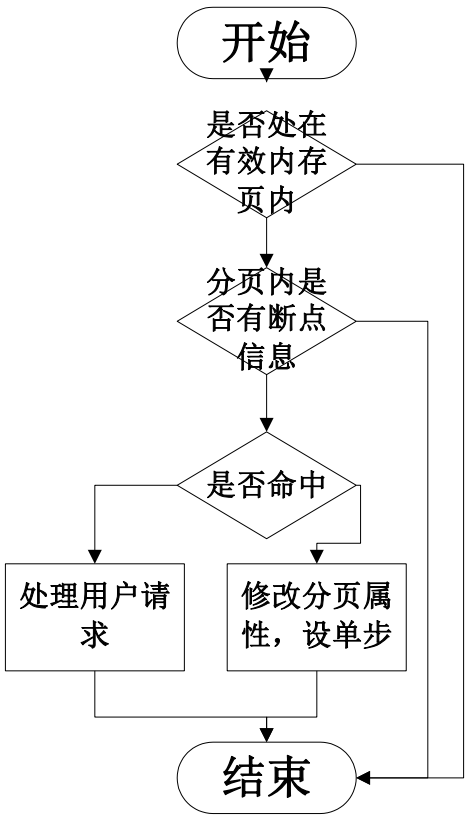
息，判断地址是否有效，然后再判断是否具有要断下来的属性。对一个没有读属性的内存分页下读断点是无效的。

为了提高效率，中间表用了二个，一个是以内存页序号为主键，一个是以断点序号为主键，使用动态邻接表(长度自动增长)存储，这样可以提高程序的处理速度。当调试器收到异常信息，首先取得产生异常的地址，以及错误编码，再通过异常地址取得内存页的序号，再通过内存页的序号去邻接表里查他有多少个断点，如果当前有内存断点的话，恢复内存页原属性，然后再判断当前位置是否处于用户设的断点地址内，是的话，处理用户的请求，最后设单步，在单步中再将内存页的属性设成新的属性。

添加内存断点流程图



处理异常流程图

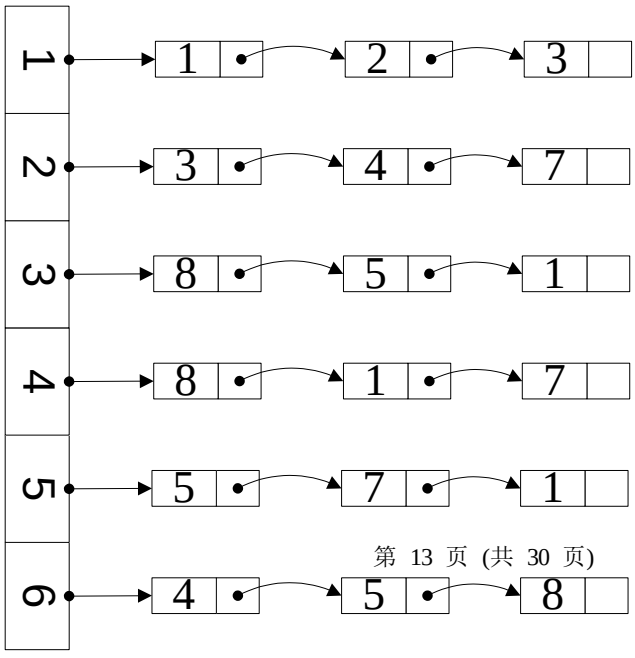


内存断点本来就是一件很耗资源的，为了最大的提升效率，当处理某内存分页具有读写的权限，而用户只下读或写的权限的这类情况，我们可以通过对原属性的分析，然后再减去用户新的属性，再根据结果生成新的属性,因为移除某个属性,并不能单单的通过去除某个位而完成。

相关记录项的结构:

```
// 记录内存断点
typedef struct _MemBreakNode
{
        _MemBreakNode   *pNext ;        // 指向下一个内存断点信息的地址
        int             nID ;
        LPVOID          lpBreakBase ;   // 内存断点的起码地址
        SIZE_T          BreakLen    ;   // 内存断点的长度
        DWORD           Protect     ;   // 权限(1:读 2:写)
        int             isActive    ;   // 是否激活
} MemBreakNode, *PMemBreakNode ;
// 记内存分页属性
typedef struct _MemNode
{
public:
        _MemNode       *pNext ;                 // 指向下一个内存分页地址
        int             nID    ;                // 唯一标识
        PVOID           BaseAddress ;           // 分页开始基地址
        PVOID           AllocationBase ;        // 开始基址四舍五入后的结果
        DWORD           AllocationProtect ;
        SIZE_T          RegionSize ;
        DWORD           State ;                 // 状态
        DWORD           Protect ;               // 属性
        DWORD           newProtect ;            // 新属性
        DWORD           Type ;
} MemNode, *PMemNode;
```

内存页与断点关系表(邻接表):



第三章 int3 断点与硬件断点

3.1 问题的提出与解决

3.1.1 问题提出

如何实现软硬件断点?如何判断用户断点的正确性?如何提高效率?

3.2 问题的解决

3.2.1 Int3 断点的实现

8086/8088 提供断点指令 Int3 和单步标志 TF，调试工具利用它们可以设置断点和实现单步。从 80386 开始，在片上集成了调试寄存器。利用这些调试寄存器不仅可以设置代码执行断点，而且还可以设置数据访问断点，不仅可以把断点设置在 RAM 中，也可以把断点设置在 ROM 中。

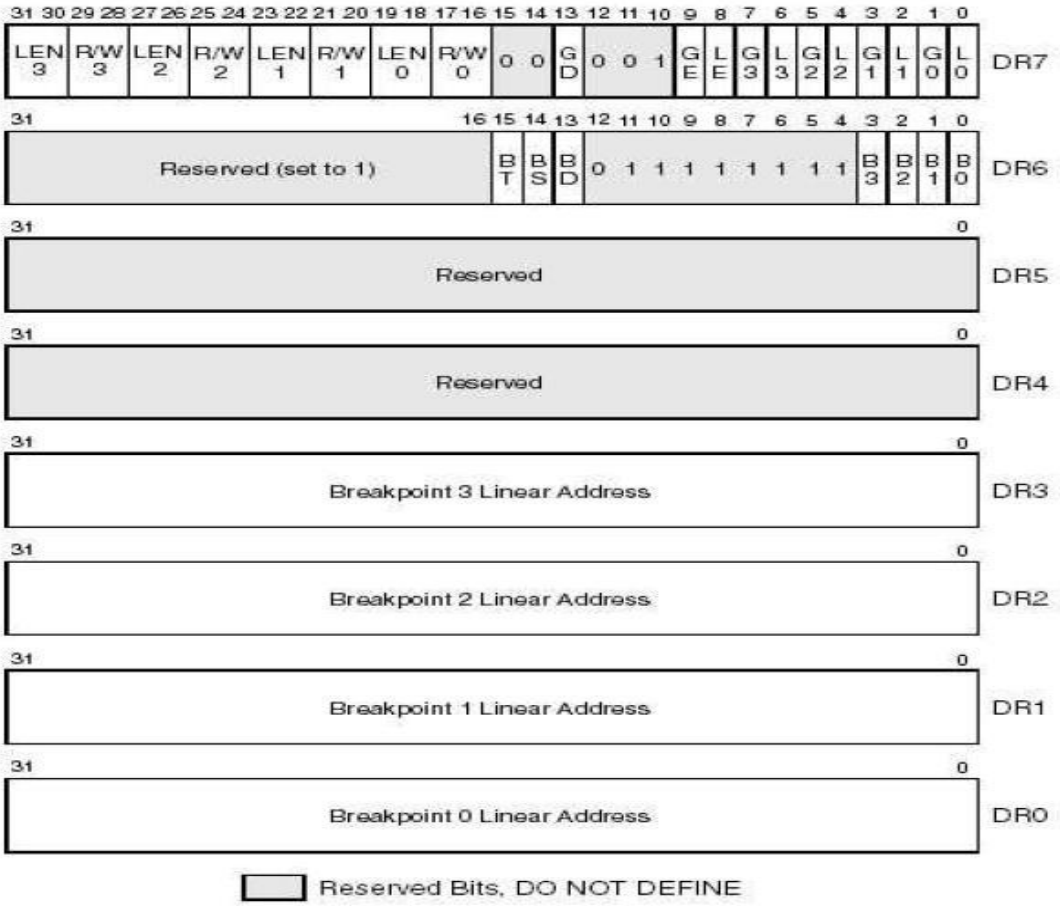
Int3 是 Intel 系列 CPU 专门引入的用于表示中断的指令，对于 Int3 断点,常见的做法是在用户要下断点的地方,将指令替换成 CC，当调试程序运行到 CC 处,调试器将会收到 EXCEPTION_BREAKPOINT 消息,然后在这里将原 CC 处的代码还原及 EIP 减一再处理用户的请求。用 Int3 的好处很明显，可以设任意多个断点，缺点是改变了程序的原指令，容易被软件检测到，而且这里可能会遇到一个问题,当用户在某个内存分页设了一个不可读,不可写的断点,这时调试器是无法将 CC 写进去的,也无法将原来的指令读出来!所以在设之前,我们先将目标内存页的属性设为可读可写的,设完之后再将内存页置为新的属性(移除了用户断点权限的新属性)，这个开销是非常大的。

本程序中,对于用户断点的正确性检测只做了如下判断,首先是判断用户的断点是否是处于有效的内存分页内,然后再判断是否重复,同一个内存地址,只能设一个断点,而对于 Int3 断点和硬件执行断点并没有判断是否处于指令起始处。

当初的设想是被调试程序跑起来的时候,将被调试程序的指令全部解析一下,然后取得每条指令的正确起始位置,添加到树中,然后用户在下 Int3 断点的时候,再判断一下下的断点是否处于指令的首地址处!后来请教了一下钱老师以及查看 windbg 等同类软件,发现也没有做这类的检测，同时分析觉得此弊大于利,所以放弃了这种做法,程序中并没有检测。

3.2.2 硬件断点的实现

80386 和 80486 都支持 6 个调试寄存器，如图:



他们分别是 DR0、DR1、DR2、DR3，调试状态寄存器 DR6 和调试控制寄存器 DR7。这些断点寄存器都是 32 位寄存器。

Dr0~3 用于设置硬件断点的线性地址，也就是断点位置，产生的异常是 STATUS_SINGLE_STEP 。这些寄存器长 32 位，与 32 位线性地址长度相符，处理器硬件把执行指令所涉及的线性地址和断点地址寄存内的线性地址进行比较，判断执行指令是否触及断点。与 Int3 不同的是，硬件断点不仅可以设置在 RAM 中，也可以设置在 ROM 中，且速度比 Int3 更快。

Dr7 是一些控制位，用于控制断点的方式。其中有四个 GiLi，分别对应四个断点，Gi、Li 控制 DRi 所指示的断点 i 在断点条件满足是，是否引起异常。

Gi 和 Li 分别称为全局断点允许位和局部断点允许位。在任务切换时，处理器清各 Li 位，所以 Li 位只支持一个任务范围内的断点，任务切换并不影响 Gi 位，所以 Gi 支持系统内各任务的断点。

R/Wi 分别对应四个 DRi ,R/Wi 字段指示 DRi 断点信息的类型。每一个 R/W 占两位，所表示的类型为:

RWE 字段取值	断点类型	RWE 字段取值	断点类型
0 0	只执行	1 0	未定义
0 1	只写入	1 1	只读或写数据

LENi 字段指示 DRi 断点信息的长度，每一个 LENi 占两位。所表示的长度如下:

LEN 字段取值	断点长度(范围)	断点地址寄存器指示的断点地址
0 0	1 字节	全部 32 位指示一个单字节的断点
0 1	2 字节	最低 1 位被忽略，确定字对齐开始的 2

		字节断点
1 0	未定义(不能取该值)	
1 1	4 字节	最低 2 位被忽略，确定双字对齐地址开始的 4 字节断点

注意: 指令执行断点的断点长度必须为 1 字节，数据访问的断点长度可以是 1、2、4 字节。

Dr6 用于显示是哪些引起断点的原因，如果是 Dr0～3 或单步（EFLAGS 的 TF）的话，则相应设置对应的位。

Bi 表示由 DRi 所指示的断点引起的调试陷阱。

对于 DR7 的设置，原本程序中是使用位移来完成的，但是调试的时候非常之不方便，所以后来改用结构体了。结构体的定义如下:

```
typedef union _Tag_DR7
{
        struct __DRFlag
        {
                unsigned int L0:  1;
                unsigned int G0:  1;
                unsigned int L1:  1;
                unsigned int G1:  1;
                unsigned int L2:  1;
                unsigned int G2:  1;
                unsigned int L3:  1;
                unsigned int G3:  1;
                unsigned int Le:  1;
                unsigned int Ge:  1;
                unsigned int b:   3;
                unsigned int gd:  1;
                unsigned int a:   2;
                unsigned int rw0: 2;
                unsigned int len0:2;
                unsigned int rw1: 2;
                unsigned int len1:2;
                unsigned int rw2: 2;
                unsigned int len2:2;
                unsigned int rw3: 2;
                unsigned int len3:2;
        } DRFlag;
        DWORD dwDr7 ;
}DR7 ;
```

3.2.3 对于 int3 断点的优化

当设 int3 断点的时候，判断一下当前调试寄存器有没有空闲的，有的话则优先使用调试寄存器，因为 int3 断点涉及内存分页属性，读目标进程内存，写目标进程，而硬件断点只需设调试寄存器，这样可以大大的提高效率。

第四章 函数名的解析

4.1 问题的提出与分析

4.1.1 问题的分析

最初对于这个问题处理的想法就是分析被调试程序的导入表，后来和苏玉海讨论，发现这样不能解决问题，因为 API 里可能还调用了别的 API,再跟进 API 的话，程序就不能正确解析函数名了！

据说 Windows 有上万个 API,如何组织这些数据，如果提高查询的效率、以及最大化的合理使用空间。

4.2 问题的解决

4.2.1 函数名与函数地址的组织

用一个链表用来存储已经加载的 DLL 的名字已及当前状态。为了提高查询的效率，同时记录 DLL 名字的 CRC 值，当查找时，算出要查找的 DLL 名的 CRC 值，再直接查找通过 CRC 值查找，当 CRC 值相同时，再用字符串比较是否完全相等，当加载 DLL 比较多的时候，或者经常要查询的话，这样可以大大的提升的效率。

同时用红黑树记录函数名地址(函数名在被调试进程里面的地址)以及函数的绝对地址(基址加偏移后的真实地址)。

红黑树并不追求“完全平衡”——它只要求部分地达到平衡要求，降低了对

旋转的要求，从而提高了性能。红黑树能够以 O(log2 n) 的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。

4.2.2 函数名的显示

在我的设计中，使用的是 OD 的反汇编引擎，结构体中有记录命令的类型，首先判断他是不是 call、jmp 这类的指令，然后再根据相应的值，找出 API 的地址，再判断目标地址是否是 API 函数的地址，是的话则从本地加载 DLL 链表中取得 DLL 名字，然后再通过函数名地址去调试进程中取得函数名(在程序中，并没有申请空间保存函数名，因为导出函数可能非常之多，不停的申请空间，然后字符串拷备，这个开销是非常之大的，所以在程序中我们只记录函数名在调试进程中的地址，同时 dll 名链表中有个状态值，当状态值为真时，表明那个 dll 当前已经加载，这样可以直接去内存拿函数名)。而对于 call ebp 之类的指令，而需要在运行时才解析，因为在没有执行到那条语句的时候，寄存器的值是不确定的。

效果如下:

```
-u
00401000  6A 00           PUSH 0
00401002  68 0E304000     PUSH 40300E
00401007  68 00304000     PUSH 403000
0040100C  6A 00           PUSH 0
0040100E  E8 0D000000     CALL 00401020 <USER32.dll::MessageBoxA>
00401013  6A 00           PUSH 0
00401015  E8 00000000     CALL 0040101A <KERNEL32.dll::ExitProcess>
0040101A  FF25 00204000   JMP [402000] <KERNEL32.dll::ExitProcess>
-
```

第五章 指令记录

5.1 问题的提出与分析

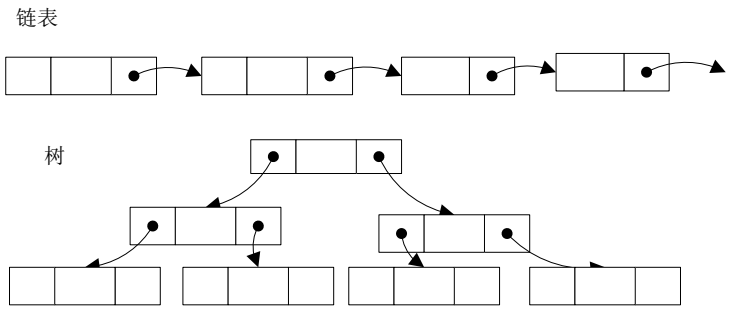
5.1.1 问题的分析

当初听到这个的时候,觉得无从下手,而且也想到了很多极端的问题,比如运行的时候不停的修改原来的指令来达到往下运行的效果。然后进到 API 里面去了,那个记录量是非常之大的,而且当时 A1Pass 的测试结果是每秒只能跑 7 条指令.于是花了很多时间在想着效率提升这块.后来和岳磊讨论一下,他叫我学习一下 OD 的这个功能.

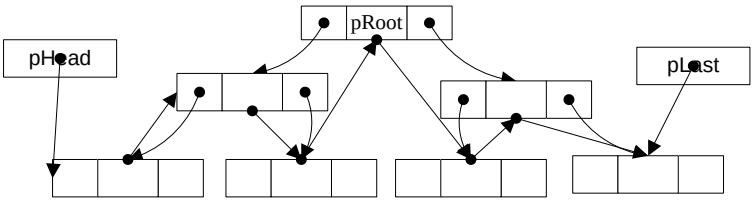
5.2 问题的解决

5.2.1 数据的组织

因为可能记录的数据量非常之大,所以要采用一种合理的数据结构,比如用户可以按指令跑的顺序记下来,也可以按地址的高低来记录,程序在运行中，可能某个地址的指令要跑很多次，但是我们不能重复记录很多次，所以这需要一种特定的数据结构,在程序中,我使用的是带头尾指针的链表与红黑树的结合,首先是将指令加到链表尾部,然后再插到红黑树里面去,这样查找起来的话,速度非常快的(因为我们每跑一条指令，都要先判断一下树中有没有，没有的话才插到树中去),而且体积只是比单单的红黑树多一个指针域。



结合后的数据结构:这样就可以把链表与树的优点结合起来了，而补了链表的查找效率的不足。



5.2.2 指令记录实现

在第一个解决方案中,采用的是置单步记录，但是经过测试这个效率不高，而且处理起来也很复杂，比如对于每个 call 之类的指令，判断是不是调用 API 之类的，而且经常会出现一些小问题，兼容性也不太好！

在第二个解决方案中，采用的是内存断点的方法，这个实现起来方便快捷，也不用判断是不是调用 API,而且可以指定要记录的部分 ,不过内存断点的开销还是挺大的。

5.2.3 效率的优化

假定计算机访问寄存器需要花费一个单位时间片 ,访问 cache 可能要花费几

个单位时间片，或许现代计算机已经可以像访问寄存器那样高速了，但是访问内存却要花费一百多的单位时间，则访问磁盘则更慢了！

为了充分的利用计算机时间，让写入磁盘与处理同时进行，在树节点中，我们直接记录指令的十六进制的值，而不直接记录反汇编结果，这样最大的节省空间，如果对于调用 API 的话，在节点 JmpConst 中记录，如果没有的话，这个值为零。然后在写入磁盘的时间，调用反汇编引擎将 Dump 的内容解析出来，如果 JmpConst 有值的话，再转成相应的 DLL 名和函数名。这样就可以并发处理了，从而充分的利用了 CPU 的时间以及节省了内存空间。

结点内存图:

左儿子	右儿子	父节点	**pNext**	颜色	地址	指令	**jmpConst**

第六章 反反调试

6.1 问题的提出与分析

一些恶意软件常常使用一些反调技手段用以阻碍对其进行逆向工程。反调试技术是一种常见的反检测技术,因为恶意软件总是企图监视自己的代码以检测是否自己正在被调试。为做到这一点,恶意软件可以检查自己代码是否被设置了断点,或者直接通过系统调用来检测调试器。

虽然在本设计中还没有来得及添加这些功能,但是想通过本章的内容,达到一个抛砖引玉的目的。

下面介绍二种常见的反调试方法:

一、检测 CC 断点。

二、检测调试器。

6.2 问题的解决

对于检测断点的反调试方法,我们可以动态的生成断点,比如 CC 断点,CD 断点等等,然后在处理函数做相应的处理就可以了。

检测调试器的存在,一般是使用 IsDebuggerPresent 函数,跟进 API 我们发现他的实现很简单。

```
7C8130A3 >   64:A1 18000000   MOV EAX,DWORD PTR FS:[18]
7C8130A9     8B40 30          MOV EAX,DWORD PTR DS:[EAX+30]
7C8130AC     0FB640 02        MOVZX EAX,BYTE PTR DS:[EAX+2]
```

7C8130B0 C3 RETN

FS 指向的当前活动线程的 TEB 结构，TEB 的第一个结构是 TIB，TIB 偏移 0x18 处是一个回指指针，指向自身，这里虽然是指向自身，但是我觉得这可能是微软以后为了扩展功能用的，而且从相关的代码中也可以看出来，他们都是先在 TEB 偏 0x18 处取值，然后再做相关的操作。 之后再取得 PEB 的地址，

通过 Windbg 我们可以到 PEB 的结构信息：

0:000> dt _PEB 0x7ffde000

ntdll!_PEB

+0x000 InheritedAddressSpace : 0 ''

+0x001 ReadImageFileExecOptions : 0 ''

+0x002 BeingDebugged : 0x1 ''

+0x003 SpareBool : 0 ''

+0x004 Mutant : 0xffffffff

+0x008 ImageBaseAddress : 0x01000000

+0x00c Ldr : 0x00181ea0 _PEB_LDR_DATA

+0x010 ProcessParameters : 0x00020000 _RTL_USER_PROCESS_PARAMETERS

.....

这样我们就可以看到，IsDebuggerPresent 只是简单的通过判断 PEB 结构的 BeingDebugged 值来判断是否被调试器加载。

这样很简单的反反调试就出来了，我们只需将 PEB 结构的 BeingDebugged 的值改成 0 就可以了。

```
__asm
{
        mov eax, fs:[0x18]
        mov eax, [eax + 0x30]
        mov byte ptr [eax+2], 0
}
```

只要将这段代码写到目标进程里跑一下就可以了！当然，更好的话当然是自己实现 CreateProcess 这类函数。

结束语

本系统的功能基本实现，因为个人努力不够的关系，很多地方不尽人意，系统原定的很多功能也被减去了，对于部分反调试功能等等,这个将作为后期的开发计划。

致谢

非常感谢钱林松和方志强老师，本设计是在他们的悉心指导下完成的。深深感谢钱林松老师和方志强老师对我们学业的关心，对设计进度的监督和指导，以及对相关工作的指导和帮助。老师严谨的治学作风、认真的工作态度、勤劳的工作精神和渊博的知识深深的影响着我，使我受益终身。在此我对他表示由衷的感谢。同时还要多谢赵海旭和岳磊二位辅导老师，多谢他们对我的栽培，是他们传授给我方方面面的知识，拓宽了我的知识面，培养了我的功底。

感谢我所引用的文献资料的作者，没有你们的辛勤劳动，我的论文就不能顺利诞生。同时也要衷心感谢一起战斗的兄弟们:孙年忠、薛亮亮、许晓明、易新、任晓晖、柳德春、吴彬、彭燕青等各位科锐的同学，感谢他们对我的支持与鼓励，使设计能够顺利完成。

参考文献

[1] 段钢 著 《加密与解密(第三版)》[M]电子工业出版社 2008 年 7 月

[2] 罗云彬 著 Windows 环境下 32 位汇编语言程序设计（第 2 版） [M]电子工业出版社 2006 年 3 月

[3] 毛德操 著 《Windows 内核情景分析--采用开源代码 ReactOS》 [M]电子工业出版社 2009 年 5 月

[4] 看雪学院 著《软件加密技术内幕》[M]电子工业出版社 2004 年 8 月

[5] 张静盛 著《Windows 编程循序渐进》[M]机械工业出版社 2008 年 5 月

[6] （美）Thomas H.Cormen, Charles E.Leiserson 著 潘金贵[同译者作品] 顾铁成 李成法 叶懋译 《Introduction to Algorithms, Second Edition(算法导论)》 机械工业出版社 2006 年 9 月

[7] （美）Mark Allen Weiss 著 冯舜玺[同译者作品] 译 《数据结构与算法分析——C 语言描述（原书第 2 版）》 机械工业出版社 2004 年 1 月

[8] 《Intel 64 and IA-32 Architectures Software Developer's Manuals Volume 3A》 2008 年

[9] 杨季文 著《80X86 汇编语言程序设计教程》 [M] 清华大学出版社 1999 年 3 月

附录

开发工具

模　　块	工具
平　　台	Windows XP sp2、VC6.0
建　　模	Microsoft Visio2003
文　　档	Microsoft Word2003

运行环境

角色	硬件要求	软件要求
客户端	处理器:Inter Pertium(Celeron)2 或更高 内存: 128M 或者更高	操作系统: WinNT 平台