|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **2021信息安全02** | | | **姓名** | **于璐** |
| **实验题目** | 实验三 PE文件结构分析 | | | | | |
| **实验时间** | **2023/11/11** | | **实验地点** | **DS3305** | | |
| **实验成绩** |  | | **实验性质** | **☑验证性 □设计性 □综合性** | | |
| 教师评价：  ☑算法/实验过程正确； ☑源程序/实验内容提交； ☑程序结构/实验步骤合理；  □实验结果正确； □语法、语义正确； ☑报告规范；  其他：  评价教师签名： 杨吉云 | | | | | | |
| 1. 实验目的 2. 验证局部变量的机制 3. 验证PE文件头结构 4. 验证PE文件目录结构 5. 验证PE文件节表结构 | | | | | | |
| 1. 实验项目内容 2. 编写程序，要求：1）在数据区定义10个单元的word数组，初始化为确定的数值；2）含1个子程序，在子程序中定义10个单元的word数组，将数据区定义的数组赋给子程序定义的数组；在主程序中调用子程序，记录堆栈变化情况。 3. 在1编写程序的基础上，加入调用MessageBox函数生成exe文件，利用Winhex、OD分析该exe文件的结构； | | | | | | |
| 1. 实验过程或算法（源程序）   1.编写源程序  1  构建并运行:  messagebox弹窗加载出来,程序运行成功：  弹窗  2.在OllyDBG中观察堆栈变化情况  2  子程序执行后数组被写入到堆栈区域  3.在lordPE下查看pe文件头关键信息  1  4.在winhex下观察pe文件结构  加载之前：  3  加载之后：  4 | | | | | | |
| 1. 实验结果及分析和（或）源程序调试过程   **1.在主程序中调用子程序，记录堆栈变化情况。**  用OD打开lab3.exe 按F7调试  1.1初始化的堆栈,和寄存器  1 2  1.2 按下F7键程序从主函数开始执行  执行到invoke subProc语句后:  3  观察到esi即数组的偏移,esi的值为403000h，首先被压入栈。  然后ecx(数组的长度)被压入栈，ecx的值为10,即0Ah。  执行invoke指令，因为我们写的子函数没有参数，所以这里直接将当前指令的下一个指令的地址压入栈。（如果有参数，应该是先将参数从右至左压入栈，然后将指令地址的下一条地址压入栈。）所以下一条地址401031被压入栈，作用是子程序执行完返回到lab3的模块入口点。  4  return的地址  1.3执行子程序，跳转到401000处：  5  在子程序中，我们要用edi指向子程序的数组地址，所以首先将edi压入栈，然后将ebp的值压入栈，将esp的值赋值给ebp。  mov esi,[ebp+16] ;将栈中保存的array10的地址赋值给esi。  mov ecx,[ebp+12] ;将栈中保存的array10的长度赋值给ecx。  然后执行sub ebp,10\*type word;申请10个word长度的地址，将数据区的数据复制到申请的空间里。  将ebp的值赋值给edi，此时edi指向的新数组的低地址（因为栈是向低地址增长的）。  6  edi指向新数组的低地址（观察寄存器区）  1.4 执行子程序的循环，从第一个数字开始拷贝  循环的逻辑:  ecx的值为数组的长度，即循环的次数  将esi指向的地址中的值赋值给ax  将ax中的值赋值给edi所在的地址。  esi的值加上一个word的长度,即esi+2,使esi指向原数组中的下一个数的地址  edi的值加上一个word的长度,即edi+2,使edi指向新数组中的下一个数的地址  循环  ；1000h,2000h,3000h,4000h,5000h,6000h,7000h,8000h,9000h,0A000h  1.4.1 执行5个循环之后:  7  执行5次循环，前五个数字被成功拷贝  1.4.2 程序跳出循环之后：  8  数组被拷贝成功  1.5 pop ebp ，pop edi ret  因为我们在子程序中分别将edi和ebp入栈,所以按照栈后进先出的规则,我们将ebp和edi弹栈。  因为我们没有定义局部变量，所以ret之后没有参数。  9  pop ebp;栈顶指针下下移，esp的值变为0019ff64h  ebp的值恢复为0019ff80h  11  pop edi;栈顶指针下下移，esp的值变为0019ff68h(esp+4)  1.6 按下F7执行ret，返回到主程序中  12  此时栈的状态恢复到调用子程序之前的状态  **2.调用MessageBox时的堆栈变化情况**  MessageBox和我自己写的子函数不同，他有四个参数要传入，所以invoke messagebox指令被解析为四条push指令和一条call指令，也就是说，五条数据会被压入堆栈：从又至左4个实参和当前指令的下一条指令的地址。  2.1观察代码区：  13  红框中即invoke messagebox指令的代码区，从401031开始  2.1执行观察堆栈（按5次F7）  2.1.1按下四次F7将参数压入栈  14  首先入栈的是第四个参数,MB\_OK,通过push 0来实现  然后是标题内容，在数据段定义lpCaption字符串，传入字符串的地址  然后是文本内容，在数据段定义lpText字符串，传入字符串的地址  第一个参数，对实验无意义，传入null即可，体现到代码里就是push 0  2.1.2执行call指令  15  堆栈区  16  代码区  由代码区的提示得出，程序要跳转到user32.MessageBoxA  2.1.3 执行jump,程序来到user32.dll所在的内存空间  17  代码段：可以观察到程序来到76A8000基地址处  2.1.4调试user32.MessageBoxA  调试观察代码区，堆栈区，EIP  18  堆栈区，按照user32中的代码压栈弹栈  19  寄存器区，EIP的值变为76A80CB5，不再是lab3而是user32  调用过程比较复杂，观察堆栈区以及EIP指针后面的注释，可以看到user32还调用了ntdll，kernellba等程序。  在堆栈区可以观察到我自己定义的文本，比如I am text逐步被拷贝到别的地址处：  403014h处(我们自己写的程序的数据段）的文本一直是完整的，他们被拷贝到6743A8h（猜测这里应该是该动态链接库的数据段区域）处  20  不断按F7,观察堆栈的增长/减少  21  展示一个弹出要完成的汇编代码行数太多了，底层很复杂，因为要将框画出来并展示在屏幕上，不再叙述这部分。  **3.观察invoke ExitProcess,0指令的执行**  将EIP指针的值指向401044h处并调试  22  代码区：观察到程序会跳转到kernel32处  栈里面是push 0,以及下一条指令的地址40104Bh  23  堆栈区  **4.使用lordPE打开lab3.exe,观察PE文件头信息：**  24  由lordPE所展示的信息可以得到：  EntryPoint:程序入口点为1020h，即程序开始执行EIP的值为401020h  ImageBase:即(建议)加载到内存中的基地址  SizeofImage:内存中空间占用，为4000h  BaseOfCode:代码节的起始RVA，为1000h  BaseOfData:数据节的起始RVA，为2000h  SectionAlignment:加载到内存中节的对齐粒度为1000h,即4KB  FileAlignment:文件中节的对齐粒度为200h,即512B  NumberOfSections:区段的数量为3  TimeDataStamp:时间戳，可以转换为文件生成的日期值  SizeOfHeaders:所有头+节表的大小为400h  点击sections查看节的信息（由NumberOfSections，节数量为3）：  25  节表  点击L查看PE文件详情：  111  DOS头  112  File Header  113  Optional Header  114  DataDirectory  **5.用winhex打开lab3.exe文件查看pe文件信息：**  在文件中：  11  根据MZ头中[3Ch]处的值得知PE头是从B0h开始的  PE头中包括FileHeader和OptionalHeader以及数据目录项Datadirectory，在上图中标注了一些重要信息。  PE头之后是区块  在三个节表中，可以得到他们在文件中对齐的尺寸以及在文件中的偏移，当然也可以直接从lordPE中直接获取。  前八个字节是节的名字  26  三个节在文件中对齐后的尺寸都为200h（在文件中的对齐粒度）,即512B  名为.text节在文件中的偏移为400h:  31  .text  名为.rdata节在文件中的偏移为600h:  32  .rdata  名为.data节在文件中的偏移为800h:  这里存放的是我们在数据段自己定义的数据，然后用0补齐200h的对齐粒度  33  .data  加载到内存后：  根据ImageBase中的值，加载到内存中的基地址为40000h。节在内存中的对齐粒度为1000h。  777  13  根据计算，此时.data在内存中的偏移应该是403000h  42  内存中的.data  **6.查看导入表信息**  导入表是从动态链接库引入的函数，所以导入函数的地址位于被加载的进程地址空间空间中的相应的动态链接库模块内。系统在执行用户程序对导入函数的调用语句时，会跳转到改地址处执行的导入函数代码。  已经在OD中了解到了执行（jmp指令）导入函数后EIP的跳转。  观察一下动态链接库user32.dll中messagebox的调用情况：  用lordPE工具：  导入表  导入表信息  可以得到：  导入表数据所在RVA=2010h  导入表数据大小=3ch(分为3组,前两组代表两个动态链接库,最后一组为全0结构,表示导入表描述符已经结束)，本程度调用的动态链接库有user32.dll和kernel32.dll。  11111  选中部分为调用messagebox的user32.dll有关的导入表数据。  最后一个双字为FirstThunk=2080h  在内存402008h处的值为76A80CA0h,该值为messagebox函数在动态链接库中的地址，而且76A80CA0h刚好位于user32.dll的代码段中  666  invoke指令被分解为3部分   1. 将参数压栈部分 2. call指令部分 3. jmp指令部分   首先根据OriginalFirstThunk的值2054h(该值是一个RVA)，计算出FOA为654h,然后去文件中的654h处，从654h处开始取双字，取到的依次是hint name,然后加载器会根据函数的Hint/Name从内存地址空间中查找到函数的VA,然后将找到的函数地址覆盖掉内存中的地址，当程序被加载到内存之后，jmp指令跳转到的地址的值就是调用的动态链接库的函数的代码段的地址，为上面内存中的[00402008] = 76A80CA0h。程序跳转到user32.dll的代码区继续执行。  222  654h处为019dH+字符串（黄色阴影部分为函数名name）  **7.实验总结**  通过本次实验，对PE文件结构和OD等工具的使用都有了更加深刻的了解，突然可以理解老师上课讲的内容了。通过看书更好的了解了PE文件的结构，了解了invoke指令的底层实现，并在虚拟机中验证了。更充分的理解了EntryPoint，ImageBase，FileAlignment等的意义。同时对寄存器的意义，对EIP,ESP,EBP等指针寄存器的理解更加充分。  通过对逆向分析工具的使用学到了更多的知识,通过学习对齐粒度更好的理解了操作系统课上学习到的内存管理。从以下截图中可以知道，我自己的程序lab3.exe在内存的基地址为40000h（根据ImageBase中的值），与动态链接库的基地址不同，比如调用的两个动态链接库，user32.dll的基地址为76A00000h，kernel32.dll在内存中的基地址为75320000h：  41 | | | | | | |