2021/01/19_32位汇编_第7课_内联汇编、裸函数、调试器的使用

笔记本: 32位汇编
创建时间: 2021/1/19 星期二 10:54
作者: ileemi

内联汇编

在高级代码中直接编写汇编代码（不是标准），使用的汇编代码不会被编译器优化，同时编写内敛汇编时需要保存寄存器环境、恢复寄存器环境（pushad、popad）以及保存标志寄存器和恢复标志寄存器（pushf、popf）代码实例：

```
//内联汇编  inlineAsm
_asm
__asm  int  3
//__asm  mov  ebx,  1;
__asm  mov  eax,  eax;  //  汇编代码前添加  "__asm"

if (argc == 1){
    __asm{
    pushad
    pushf
    // 多行汇编代码
    // 可以使用标号，但是不能使用伪指令（.if .while local invoke db dw）
    popf
    popad
    }
}
```

裸函数

> 使用汇编代码编写的函数不希望编译器帮助自动生成一些额外的汇编代码（保存寄存器等），在函数的返回类型前添加 "__declspec(naked)"，**裸函数中使用汇编代码需要自己保存寄存器环境，使用局部变量时需要自己抬栈（debug版抬栈时会多生成一些空间（32个字节），Release不会多抬空间）。**

手动抬栈可以使用伪指令（__LOCAL_SIZE），可以自动计算需要抬栈的字节数（局部变量需要使用的前提下）。

裸函数中可以使用高级语言的语法，直接使用库函数也不会生成额外的代码，推荐自己使用 "call" 调用库函数。

裸函数中不能使用 "db"，可以需要伪指令 "_ emit" 替换，但是一次只能定义一个字符，代码示例：

```
__declspec(naked)  int  MyAdd(int  nNum1,  int  nNum2)  {
    int  local1;
    int  local2;
    int  local3;
    Point  pt;
    __asm
    {
        push  ebp
        mov  ebp,  esp
        sub  esp,  __LOCAL_SIZE  //自动计算局部变量的大小
    }
    __asm  {
        mov  local1,  1h
        mov  local2,  2
        mov  local3,  3
        mov  eax,  nNum1
        add  eax,  nNum2
        push  offset  MY_MSG
        call  printf // 调用C库函数
        add  esp,  4
    }
    __asm  {
        leave
        retn
        MY_MSG:
                _emit  'M'
                _emit  'y'
                _emit  0dh
                _emit  0ah
                _emit  0
    }
}
```

数组操作

汇编代码中访问数组元素同样可以使用伪指令（TYPE ary（数组首地址，编译器会自动判断数组元素的类型）），判断数组元素的个数可以使用伪指令："SIZE ary / TYPE ary" 或者 "LENGTH ary"。

结构体操作

高级语言定义结构体，汇编代码访问结构体中的元素，代码示例：

```
#include <iostream>

//裸函数  不额外增加汇编代码
const char* p = "MyAdd\n";
struct Point {
    int x;
    int y;
};
struct Point g_pt;
__declspec(naked) int MyAdd(int nNum1, int nNum2) {
    int ary[3];
    Point pt;
    __asm
    {
        push ebp
        mov ebp, esp
        sub esp, __LOCAL_SIZE //自动计算局部变量的大小
    }
    //pt.x = 2;                          // 类似C语言直接访问
    //pt.y = 3;
    //g_pt.x = 4;
    //g_pt.y = 5;
    __asm {
        //结构体操作
        /*mov ecx, offset g_pt
        mov [ecx]Point.x, 1
        mov [ecx]Point.y, 2
        lea ebx, pt
        mov [ebx]Point.x, 0
        mov [ebx]Point.y, 0*/
        cmp n1, 0
        //数组操作
        //mov ary[0 * TYPE int], 1
```

```
        mov  eax,  LENGTH  ary                    //  计算数组中元素
的个数
        mov  eax,  SIZE  ary  /  TYPE  ary    //  计算数组中元素的个数
        mov  ary[1  *  TYPE  ary],  2          //  给数组第二个元素进
行赋值
        mov  ary[2  *  TYPE  ary],  3          //  给数组第三个元素进
行赋值
        mov  eax,  nNum1
        add  eax,  nNum2
        mov  eax,  n1
        add  eax,  n2
        push  offset  MY_MSG
        call  printf
        add  esp,  4
    }
    __asm  {
        leave
        retn
        MY_MSG:
              _emit  'M'
              _emit  'y'
              _emit  0dh
              _emit  0ah
              _emit  0
    }
}
int  main(int  argc)  {
    printf("Hello  World!:%d\n",  MyAdd(1,  2));
    return  0;
}
```

调试器

- OD（32位）
- x64Dbg（32位、64位）
- WinDbg（32位、64位）

异常设置可能会导致调式程序入口断点不会断下来。

ollydbg

查看菜单：

- 调用堆栈：通过调用堆栈窗口可以分析出函数间的调用关系

- 源码：通过 ".pdb" 文件可以在ollydbg进行源码调试（需要.cpp文件和 ".pdb" 文件同时存在，现在这个功能没有用）

调试菜单：

- 快捷键：
  F2：添加断点
  F4：执行到所选代码
  F7：单步跟踪（步入），一条代码一条代码地执行，遇到call 语句时会跟入执行该语句调用地址处地代码或者调用函数的代码。
  F8：单步跟踪（步过），遇到 call 语句时不会跟入。
  F9：加载程序后，按F9运行程序
  F12：暂停（停到系统的.dll中）
  Ctrl+F2：重新开始调试
  Ctrl+F7：自动步入
  Ctrl+F8：自动步过
  Ctrl+F9：执行到返回
  Alt+F2：关闭调试
  Alt+F9：执行到用户代码（如果进入到引用的dll模块领空，则可以使用该快捷键快速回到程序领空）。
  Shift+F9：忽略异常继续运行

主界面内存窗口下面的 "命令"，支持命令下断点：bp CreateFile g

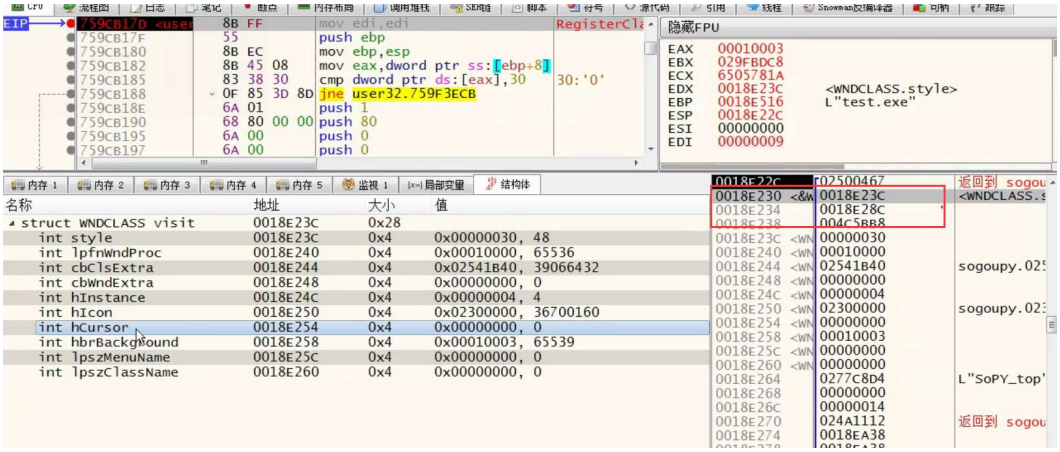
断点：

- **条件断点**：调试窗口程序时，窗口过程函数处下普通断点，运行程序消息会不停的来到这个断点处，当只要 "WM_CREATE" 消息时，只需要添加对应的条件断点就行（[esp + 8] == 1 或者 [exp + 8] == WM_CREATE）。
- **硬件断点**：利用CPU硬件下断点，硬件断点并不会将程序的代码改为 "int3（0xCC）" 指令，如果有些程序有自效验功能，就可以使用硬件断点，下中断的方法和下内存断点的方法相同，共有三种方式：硬件访问、硬件写入、硬件执行。最多一共可以设置4个硬件断点。
- **内存断点**：。分为 **内存访问断点** 和 **内存写入断点**，ollydbg 只允许一个内存断点存在。
  （1）内存访问断点：在程序运行时调用被选择的内存数据就会被OD中断。
  （2）内存写入断点：在程序运行时向被选择的内存地址写入数据就会被OD中断。
- **RUN跟踪**：记录跑过的所有汇编代码，常用来分析汇编代码的差异性。

x64Dbg

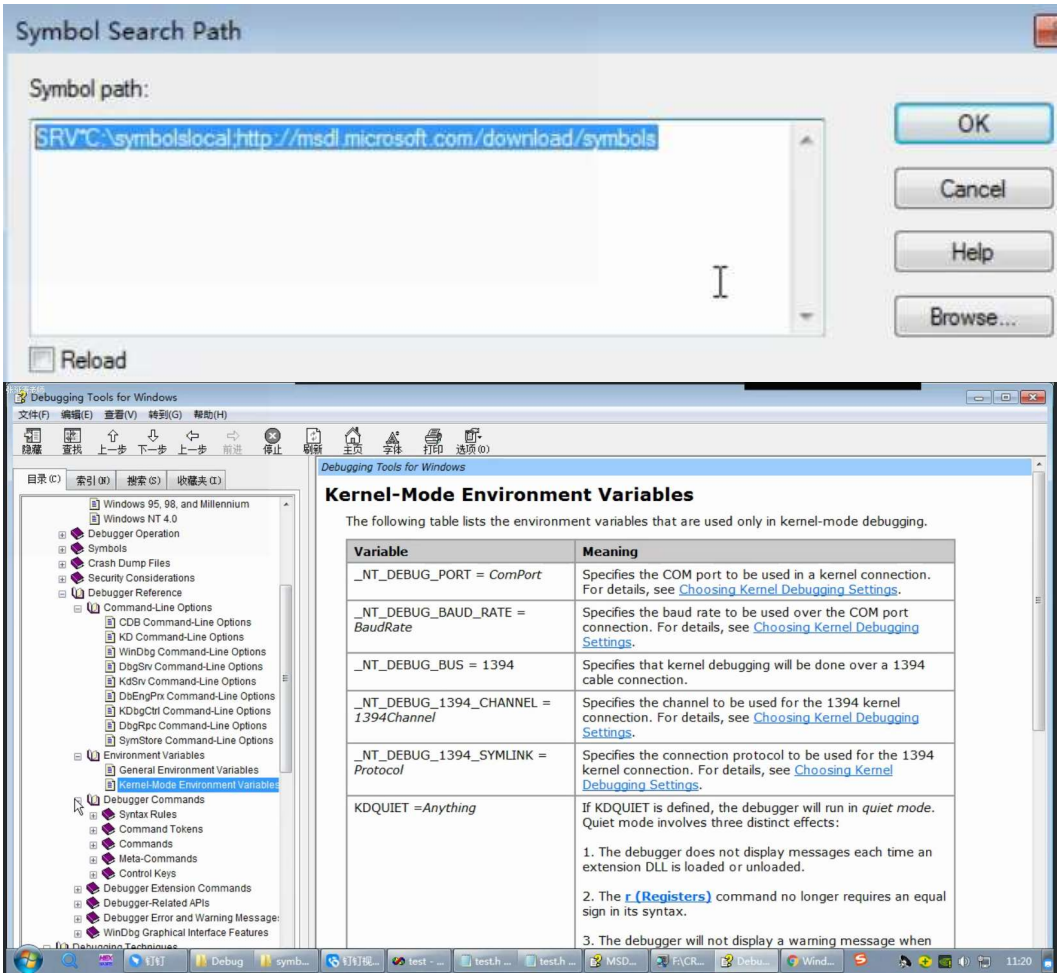
支持64位程序的调试

支持查看结构体成员的数值：再对应的API处下断点，经目标结构体做一个头文件后，分析结构体：



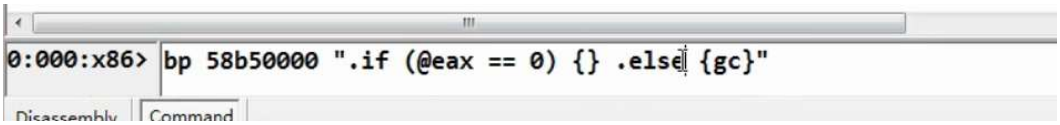
Windbg

支持API符号（.pdb）下载（可以在反汇编窗口显示API的名字）
支持UI调试，命令调试



bp $entry

条件断点（支持多种命令）：



扫雷分析雷区

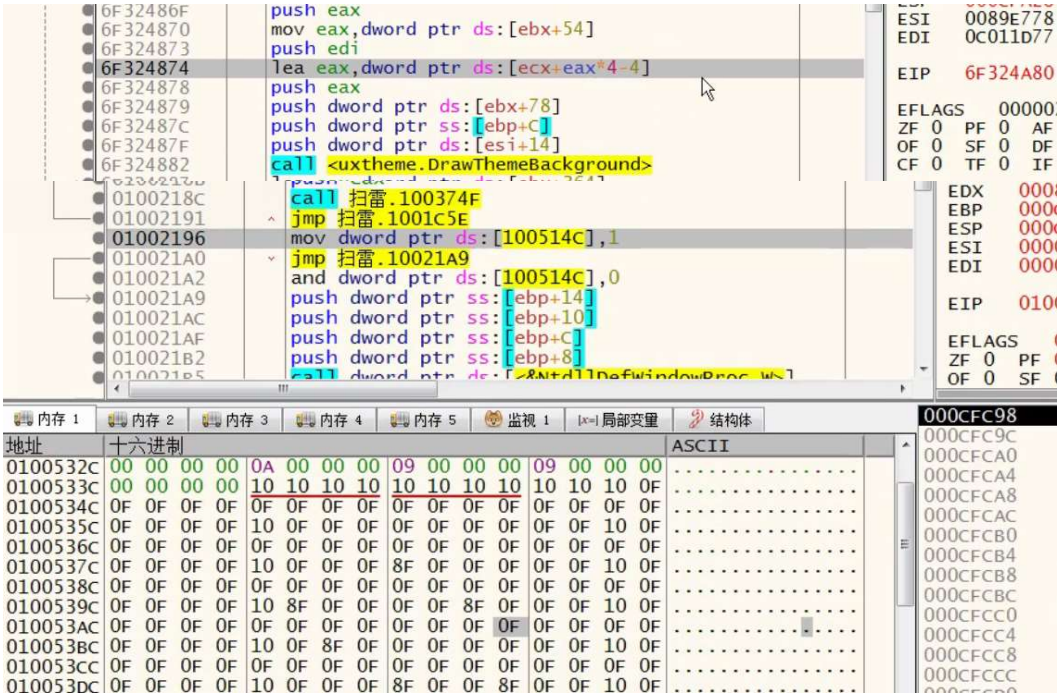
通过分析业务逻辑（程序功能角度） 获取代码（窗口最大化最小化会访问数组进行重新绘制），

双缓冲（防止窗口闪烁）

gdi32.dll -- Bitblt

找调用 Bitblt 的位置后，执行到用户代码后，往上寻找数组的寻址公式

或者找全局变量的地址



通过分析代码定位数组

不知道调用什么API的情况下，通过数据（程序界面上的可视数据）定位：

> CE工具

内存断点，通过数据

筛选器异常处理

> 当程序运行出现异常时，操作系统可以监控带程序出现异常，一般情况下程序发生异常都是系统来处理的，但是Windows操作系统也提供了一些异常处理的基址：允许让程序自己来处理异常，使用 **筛选器处理异常**（操作系统去调用程序中处理异常的代码，并运行，会将程序的异常信息发送给程序的开发者，以便修复程序的bug），主要用来定位程序中未知的bug。

API：**SetUnhandledExceptionFilter（注册筛选器异常）**，允许应用程序取代每个线程和进程的顶级异常处理程序。参数需要给一个函数指针（**UnhandledExceptionFilter**），执行处理程序异常的函数（类似回调函数）

> 程序运行时就需要注册筛选器异常处理：函数有返回值（操作系统根据返回值决定谁去处理异常）。**筛选器异常再程序中只能注册一次（第二次注册后的地址会将第一次注册的地址进行覆盖）。**

当程序出现异常的时候，操作系统最先检测到（操作系统内部对这个异常弹一个信息框，不关闭程序的情况下，当前程序并没有退出），操作系统会判断程序内存是否注册了筛选器处理异常，注册了，执行程序内部筛选器处理异常的回调函数，没有注册，由操作系统去处理这个异常。

注册筛选器异常:

> 回调函数的参数结构：

```
typedef struct _EXCEPTION_POINTERS {
   // 指向EXCEPTION_RECORD结构体的指针，该结构体包含与机器无关的异常描述
  PEXCEPTION_RECORD ExceptionRecord; // 可以显示具体的异常信息（出现异常的地址，异常代码）
  // 该结构包含发生异常时处理器状态的处理器特定描述，
  // 上下文结构包含处理器特定的寄存器数据（寄存器环境）。系统使用上下文结构来执行各种内部操作。
  PCONTEXT ContextRecord; // 可以尝试修改寄存器修改异常bug
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

- 说明：EXCEPTION_POINTERS 结构包含一个异常记录，该记录具有与机器无关的异常描述，以及一个上下文记录，该记录具有发生异常时处理器上下文的与机器相关的描述。

> 异常回调函数 "UnhandledExceptionFilter" 的返回值：

- EXCEPTION_CONTINUE_SEARCH（告诉操作系统无法处理异常）程正在被调试，因此异常应该(作为第二次机会)传递给应用程序的调试器。
- EXCEPTION_EXECUTE_HANDLER（告诉操作系统程序自己处理异常）：如果在之前对SetErrorMode的调用中指定了SEM_NOGPFAULTERRORBOX标志，则不会显示应用程序错误消息框。该函数将控制权返回给异常处理程序，异常处理程序可以自由地采取任何适当的操作。
- EXCEPTION_CONTINUE_EXECUTION：异常已经处理，程序继续执行

根据异常描述bug在可控之内，就进行异常处理，如果程序异常未知，回调函数的返回值应该为EXCEPTION_EXECUTE_HANDLER，保守做法。也可以将出现异常的详细信息通过程序发送给程序的服务器，或者写入到日志文件，等程序下次正常运行再发送。

程序出现异常时，操作系统的API会将异常信息发送给调试器，当调试器运行的时候，即便时调试器不处理（Shift+F9不处理异常）这个异常，程序也收不到异常相关的信息（操作系统API有一个bug）。

如何获取操作系统API调用程序异常处理函数地址的代码位置：操作系统调用程序异常处理的函数时，栈顶存放的是调用处理异常函数前操作系统执行代码的地址（返回地址），通过返回地址就可以找到操作系统调用异常函数的位置。

在程序不调试的情况下，当操作系统调用处理异常函数时，将返回地址打印出来（显示异常详细信息时，将操作系统调用异常函数前的返回地址进行打印），就可以找到操作系统调用异常处理时的API地址，找到地址就可以分析操作系统API对应的Bug。

获取返回地址，并记录，同时程序不能退出，使用ollydbg附加程序进行调试，直接访问刚才记录的 "返回地址"。可以确定其在哪个模块中进行的调用。在函数首地址下断点调试分析。