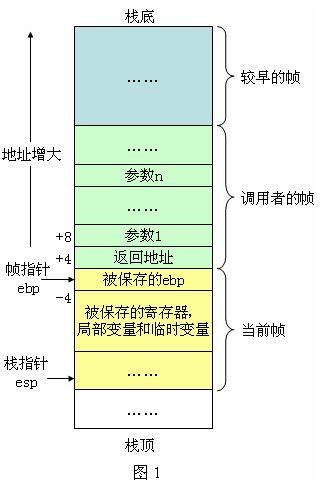
## 一.基本寄存器和反汇编知识。

**EBP:** 扩展基址指针寄存器(extended base pointer)　其内存放一个指针，该指针指向系统栈最上面一个栈帧的底部。

**ESP:** 栈顶指针，用于指向栈的栈顶（下一个压入栈的活动记录的顶部），而EBP为帧指针，指向当前活动记录的底部。

****

**EIP**:指令寄存器，存放当前指令的下一条指令的地址。CPU该执行哪条指令就是通过IP来指示的。

**具体代码：**

**反汇编代码**：

#include "stdafx.h"

int sum(int x, int y)

{

01271000 push ebp

01271001 mov ebp,esp

01271003 push ecx

//\_\_asm mov ebp, 0

int\* p= NULL;

01271004 mov dword ptr [p],0

x = \*p;

0127100B mov eax,dword ptr [p]

0127100E mov ecx,dword ptr [eax]

01271010 mov dword ptr [x],ecx

return (x + y);

01271013 mov eax,dword ptr [x]

01271016 add eax,dword ptr [y]

}

01271019 mov esp,ebp

0127101B pop ebp

0127101C ret

--- 无源文件-----------------------------------------------------------------------

0127101D int 3

0127101E int 3

0127101F int 3

--- c:\users\wujinfeng\documents\visual studio 2008\projects\crashtest\crashtest\crashtest.cpp

int sumstub(int x, int y)

{

01271020 push ebp

01271021 mov ebp,esp

01271023 push ecx

int tmp = 0;

01271024 mov dword ptr [tmp],0

printf("enter fun() ...\n");

0127102B push offset \_\_\_xi\_z+34h (12720F4h)

01271030 call dword ptr [\_\_imp\_\_printf (12720A0h)]

01271036 add esp,4

tmp = sum(x, y);

01271039 mov eax,dword ptr [y]

0127103C push eax

0127103D mov ecx,dword ptr [x]

01271040 push ecx

01271041 call sum (1271000h)

**01271046** add esp,8

01271049 mov dword ptr [tmp],eax

printf("leave fun() ...\n");

0127104C push offset \_\_\_xi\_z+48h (1272108h)

01271051 call dword ptr [\_\_imp\_\_printf (12720A0h)]

01271057 add esp,4

return tmp;

0127105A mov eax,dword ptr [tmp]

}

0127105D mov esp,ebp

0127105F pop ebp

01271060 ret

--- 无源文件-----------------------------------------------------------------------

……

--- c:\users\wujinfeng\documents\visual studio 2008\projects\crashtest\crashtest\crashtest.cpp

int \_tmain(int argc, \_TCHAR\* argv[])

{

01271070 push ebp

01271071 mov ebp,esp

printf("enter main() ...\n");

01271073 push offset \_\_\_xi\_z+5Ch (127211Ch)

01271078 call dword ptr [\_\_imp\_\_printf (12720A0h)]

0127107E add esp,4

printf("sum = %d\n", sumstub(0x1234, 0x5678));

01271081 push 5678h

01271086 push 1234h

0127108B call sumstub (1271020h)

01271090 add esp,8

01271093 push eax

01271094 push offset \_\_\_xi\_z+70h (1272130h)

01271099 call dword ptr [\_\_imp\_\_printf (12720A0h)]

0127109F add esp,8

printf("leave main() ...\n");

012710A2 push offset \_\_\_xi\_z+7Ch (127213Ch)

012710A7 call dword ptr [\_\_imp\_\_printf (12720A0h)]

012710AD add esp,4

return 0;

012710B0 xor eax,eax

}

012710B2 pop ebp

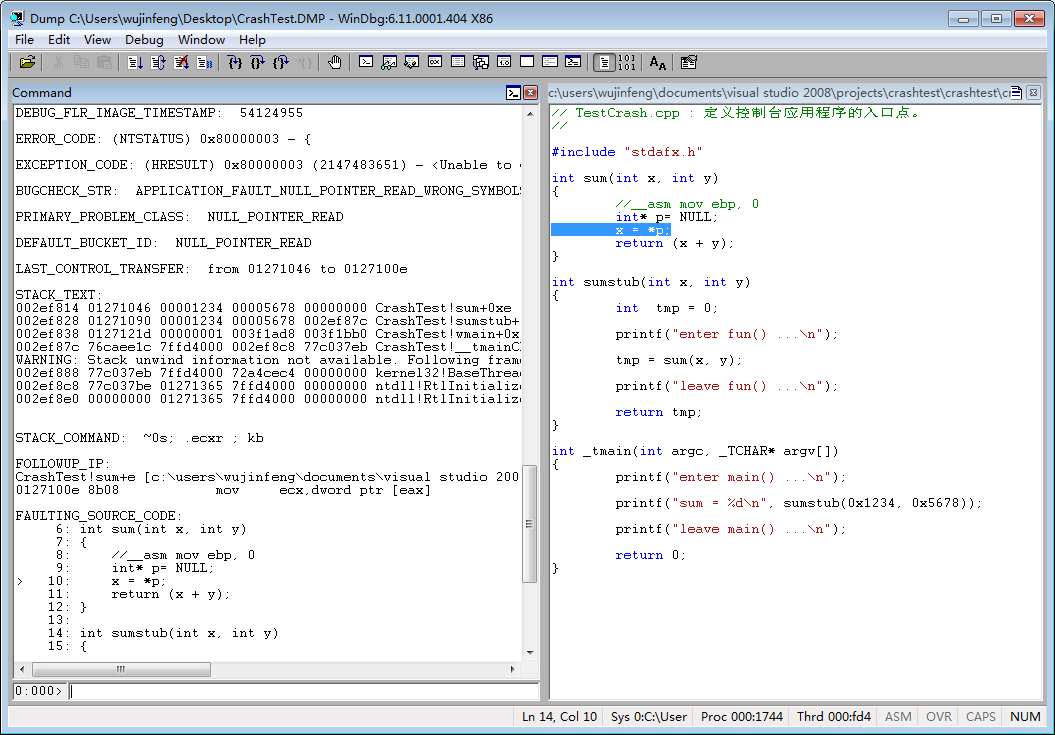
012710B3 ret

**Dump实例分析**

**1.编译器环境下手动抓取dmp**

****

**2. 设置好PDB、代码路径以后，输入 !analyze –v：自动分析命令**

****

**3.堆栈片段分析：**

**. 0（线程序号） Id: 1744（线程ID）.fd4 Suspend: 0 Teb: 7ffdf000 Unfrozen**

**ChildEBP RetAddr Args to Child**

**002ef814 01271046 00001234 00005678 00000000 CrashTest!sum+0xe**

**002ef828 01271090 00001234 00005678 002ef87c CrashTest!sumstub+0x26**

**002ef838 0127121d 00000001 003f1ad8 003f1bb0 CrashTest!wmain+0x20**

**002ef87c 76caee1c 7ffd4000 002ef8c8 77c037eb CrashTest!\_\_tmainCRTStartup+0x10f**

**WARNING: Stack unwind information not available. Following frames may be wrong.**

**002ef888 77c037eb 7ffd4000 72a4cec4 00000000 kernel32!BaseThreadInitThunk+0x12**

**002ef8c8 77c037be 01271365 7ffd4000 00000000 ntdll!RtlInitializeExceptionChain+0xef**

**002ef8e0 00000000 01271365 7ffd4000 00000000 ntdll!RtlInitializeExceptionChain+0xc2**

**几个名词：**

**ChildEBP**: a pointer to a memory location which stores the address of the previous function on the stack ("stack frame").（ebp的值：堆栈基地址）

**RetAddr**: The "return address" where processing will resume once this function returns (finishes what it had to do).（函数执行完以后返回的：计算机指令地址）

**当前寄存器状态：**

0:000> r

eax=00000000 ebx=00000000 ecx=00001234 edx=77be70f4 esi=00000001 edi=0127337c

**eip=0127100e** esp=002ef810 **ebp=002ef814** iopl=0 nv up ei pl nz na pe nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010206

CrashTest!sum+0xe:

0127100e 8b08 mov ecx,dword ptr [eax] ds:0023:00000000=????????

**崩溃原因**：eax为0，所以读取空指针指向内存里的值出现崩溃。

0127100e 是计算机指令地址。

## 二：运行环境下dump能通过自动分析命令分析出结果：（忽略）

## 三.运行环境下dump异常处理（自动分析命令无法分析出结果来）

**1. !analyze -v 命令出现问题怎么办吧。**

如下：

 0:000> !analyze -v  
    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
    \*                                                                             \*  
    \*                        Exception Analysis                                   \*  
    \*                                                                             \*  
    \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
    \*\*\* WARNING: Unable to verify checksum for Dump01.exe  
    \*\*\* ERROR: Symbol file could not be found.  Defaulted to export symbols for lpk.dll -   
    \*\*\* ERROR: Symbol file could not be found.  Defaulted to export symbols for Sysfer.dll -   
    \*\*\* ERROR: Symbol file could not be found.  Defaulted to export symbols for usp10.dll -   
    \*\*\* ERROR: Symbol file could not be found.  Defaulted to export symbols for imm32.dll -   
    \*\*\* ERROR: Symbol file could not be found.  Defaulted to export symbols for apphelp.dll -   
    \*\*\* ERROR: Symbol file could not be found.  Defaulted to export symbols for version.dll -   
    \*\*\* ERROR: Symbol file could not be found.  Defaulted to export symbols for advapi32.dll -   
    \*\*\* ERROR: Symbol file could not be found.  Defaulted to export symbols for shlwapi.dll -   
    FAULTING\_IP:   
    +0  
    00000000 ??              ???  
    EXCEPTION\_RECORD:  ffffffff -- (.exr 0xffffffffffffffff)  
    ExceptionAddress: 00000000  
       ExceptionCode: 80000007 (Wake debugger)  
      ExceptionFlags: 00000000  
    NumberParameters: 0  
……



**2.系统处理异常标识函数：Kernel32! UnhandledExceptionFilter**

0:000> ~\*kv //显示出所有线程，找到对应的函数标识

. 0 Id: 12a0.44c Suspend: 0 Teb: 7ffdf000 Unfrozen

ChildEBP RetAddr Args to Child

WARNING: Stack unwind information not available. Following frames may be wrong.

001df590 7762be2e 001df544 001df5b8 00000000 ntdll!KiFastSystemCallRet

001df5d8 7762be9c 00000002 7ffd5000 00000000 kernel32!WaitForMultipleObjectsEx+0x8e

001df5f4 776406b7 00000002 001df628 00000000 kernel32!WaitForMultipleObjects+0x18

001df660 77640952 001df740 00000001 00000001 kernel32!GetThreadSelectorEntry+0x1ee

001df674 77640900 001df740 00000001 001df710 **kernel32!UnhandledExceptionFilter**+0x249

001df684 7764087b 001df740 00000001 6c699e83 **kernel32!UnhandledExceptionFilter**+0x1f7

001df710 77da7f62 00000000 77d4e31c 00000000 kernel32!UnhandledExceptionFilter+0x172

001dffc0 77d937be 011d107d 7ffd5000 00000000 ntdll!EtwReplyNotification+0x366

001dffd8 00000000 011d107d 7ffd5000 00000000 ntdll!RtlInitializeExceptionChain+0xc2

1. Kernel32! UnhandledExceptionFilter第一个参数： EXCEPTION\_POINTERS结构

2）再根据 KiUserExceptionDispatcher 函数的原型得到本次异常发生时保存的 CONTEXT 结构信息。（windows 系统底层具体处理异常的函数）

VOID KiUserExceptionDispatcher (  
  IN PEXCEPTION\_RECORD ExceptionRecord,  
  IN PCONTEXT ContextRecord  
  )

3）通过命令/显示EXCEPTION\_POINTERS数据结构里各个变量值

0:000> dd 001df740 L2

001df740 001df840 001df85c

4）调用cxr命令解析崩溃瞬间的堆栈

0:000> .cxr 001df85c

eax=00000000 ebx=7ffd5000 ecx=00000000 edx=0fb91408 esi=001dfd68 edi=001dfd58

eip=011d13e8 esp=001dfc80 ebp=001dfd58 iopl=0 nv up ei pl nz ac po nc

cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010212

CrashTest!sum+0x28:

011d13e8 8b08 mov ecx,dword ptr [eax] ds:0023:00000000=????????

**3.公司的程序崩溃时候：（dump太大不放到文档里）**

**处理异常标识函数：**CRashAPI里

#166 Id: 191c.1274 Suspend: 0 Teb: ffe13000 Unfrozen

ChildEBP RetAddr Args to Child

WARNING: Stack unwind information not available. Following frames may be wrong.

274bf4d4 77611194 000001f8 ffffffff 00000000 ntdll+0x1f8c1

274bf4ec 77611148 000001f8 ffffffff 00000000 kernel32+0x11194

274bf500 019b3805 000001f8 ffffffff 038434e8 kernel32+0x11148

274bf518 019b3f79 274bf624 00000000 e260a4d2 CRashAPI+0x3805

274bf54c 1c985597 274bf624 776e030c 77650047 CRashAPI+0x3f79

274bf554 776e030c 77650047 274bf624 e328b74b StreamTransClient!\_\_CxxUnhandledExceptionFilter+0x3e (FPO: [1,0,1])

274bf5e0 1c982e75 274bf624 00000000 274bff78 kernel32+0xe030c

274bf5f8 1c97d85e 00000000 274bf624 1c97d2cf StreamTransClient!\_XcptFilter+0x13a

274bff88 776133ca 1cbb2210 274bffd4 77b59ed2 StreamTransClient!\_beginthreadex+0xf9

274bff94 77b59ed2 1cbb2210 50891f7f 00000000 kernel32+0x133ca

274bffd4 77b59ea5 1c97d7d0 1cbb2210 ffffffff ntdll+0x39ed2

274bffec 00000000 1c97d7d0 1cbb2210 00000000 ntdll+0x39ea5

0:166> dd 274bf624 L2

274bf624 274bf718 274bf768

0:166> .cxr 274bf768

eax=00000043 ebx=00000043 ecx=00000000 edx=00000044 esi=1c995be8 edi=00000000

eip=1c96d249 esp=274bfbcc ebp=1c995be8 iopl=0 nv up ei pl zr na pe nc

cs=0023 ss=002b ds=002b es=002b fs=0053 gs=002b efl=00010246

StreamTransClient!HPR\_Strcasecmp+0x19:

1c96d249 8a0f mov cl,byte ptr [edi] ds:002b:00000000=??

**5.还原出错堆栈**

1）当 !analyze -v 不好使的时候应该怎么得到出错地址和出错堆栈。  
    int sum(int x, int y)  
    {  
        \_\_asm mov ebp, 0  
        return (x + y);  
    }  
    int sumstub(int x, int y)  
    {  
        int  tmp = 0;  
        printf("enter fun() ...\n");  
        tmp = sum(x, y);  
        printf("leave fun() ...\n");  
        return tmp;  
    }  
    int main(int argc, char\* argv[])  
    {  
        printf("enter main() ...\n");  
        printf("sum = %d\n", sumstub(0x1234, 0x5678));  
        printf("leave main() ...\n");  
        return 0;  
    }

同理显示所有的线程

0:000> ~\*kv  
    .  0  Id: 62c.928 Suspend: 1 Teb: 7ffdf000 Unfrozen  
    ChildEBP RetAddr  Args to Child                
    0012f3b8 7c92e9ab 7c86372c 00000002 0012f53c ntdll!KiFastSystemCallRet (FPO: [0,0,0])  
    0012f3bc 7c86372c 00000002 0012f53c 00000001 ntdll!ZwWaitForMultipleObjects+0xc (FPO: [5,0,0])  
    0012fb38 00401dda 0012fb74 0012ffb0 0012ffc0 kernel32!UnhandledExceptionFilter+0x8e4 (FPO: [Non-Fpo])  
    0012fb48 00401198 c0000005 0012fb74 0040261b Dump01!\_XcptFilter+0x13e  
    0012ffc0 7c816fd7 011dd65c 011dd664 7ffd6000 Dump01!mainCRTStartup+0xd1  
    0012fff0 00000000 004010c7 00000000 00000000 kernel32!BaseProcessStart+0x23 (FPO: [Non-Fpo])

1. 先查看所有线程的堆栈信息，然后找出比较像出了问题的线程。

然后显示出错线程的 TEB 信息。

0:000> !teb （显示栈（stack)的起始地址，Tls Storage 的地址， 异常处理的地址，LastError的值）  
    TEB at 7ffdf000  
        ExceptionList:        0012fb28  
        StackBase:            00130000  
        StackLimit:           0012a000  
        SubSystemTib:         00000000  
        FiberData:            00001e00  
        ArbitraryUserPointer: 00000000  
        Self:                 7ffdf000  
        EnvironmentPointer:   00000000  
        ClientId:             0000062c . 00000928  
        RpcHandle:            00000000  
        Tls Storage:          00000000  
        PEB Address:          7ffd6000  
        LastErrorValue:       0  
        LastStatusValue:      103  
        Count Owned Locks:    0  
        HardErrorMode:        0

3）根据堆栈的位置和大小，显示堆栈的所有内容。

 0:000> dps 0x0012a000 0x00130000

根据 Windows 异常处理流程可知，所有没被调试器处理的异常最终都会转到 ntdll!KiUserExceptionDispatcher 函数查找 SEH 异常处理例程来处理异常。

所以在显示的堆栈信息中查找 ntdll!KiUserExceptionDispatcher 字符串。  
    0012fc50  00000000  
    0012fc54  7c92eafa ntdll!KiUserExceptionDispatcher+0xe  
    0012fc58  00000000  
    0012fc5c  0012fc84

第二个参数指向 CONTEXT 结构，利用 WinDbg 的 .cxr 命令显示/切换 CONTEXT 结构。  
    0:000> .cxr 0x0012fc84  
    eax=00005678 ebx=7ffd6000 ecx=00001234 edx=7c92eb94 esi=011dd664 edi=011dd65c  
    eip=0040100b esp=0012ff50 ebp=00000000 iopl=0         nv up ei pl nz na pe nc  
    cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206  
    Dump01!sum+0xb:  
    0040100b 8b4508          mov     eax,dword ptr [ebp+8] ss:0023:00000008=????????

 0:000> kv  
      \*\*\* Stack trace for last set context - .thread/.cxr resets it  
    ChildEBP RetAddr  Args to Child                
    00000000 00000000 00000000 00000000 00000000 Dump01!sum+0xb (CONV: cdecl) [E:\Works\Dump01\Dump01.cpp @ 10]

1. 现在已经找到出错地址为 0x0040100b，下面恢复的出错前的堆栈。

    0:000> ?? sizeof(ntdll!\_CONTEXT)  
    unsigned int 0x2cc

0:000> ? 0x0012fc84 + 0x2cc  
    Evaluate expression: 1245008 = 0012ff50

计算可知，出错前的堆栈位置在 0x0012ff50 处。

1. 反汇编0x0040100b处的堆栈出计算机指令

0:000> ub 0x0040100b L 6  
    Dump01!sum [E:\Works\Dump01\Dump01.cpp @ 7]:  
    00401000 55               push    ebp  
    00401001 8bec           mov     ebp,esp  
    00401003 53               push    ebx  
    00401004 56               push    esi  
    00401005 57               push    edi  
    00401006 bd00000000      mov     ebp,0

反汇编出错地址前的几条指令，可以知道出错原因是 0x00401006 处的指令导致 ebp 被赋零。

1. 返汇编0x0012ff50 前几个堆栈信息

0:000> dps 0x0012ff50 L 0x10  
    0012ff50  011dd65c  
    0012ff54  011dd664  
    0012ff58  7ffd6000  
    0012ff5c  0012ff70  
    0012ff60  0040103b Dump01!sumstub+0x25 [E:\Works\Dump01\Dump01.cpp @ 19]  
    0012ff64  00001234  
    0012ff68  00005678  
    0012ff6c  00000000  
    0012ff70  0012ff80  
    0012ff74  00401074 Dump01!main+0x1f [E:\Works\Dump01\Dump01.cpp @ 30]  
    0012ff78  00001234  
    0012ff7c  00005678  
    0012ff80  0012ffc0  
    0012ff84  0040117b Dump01!mainCRTStartup+0xb4  
    0012ff88  00000001  
    0012ff8c  00520eb0

0:000> r  
    Last set context:  
    eax=00005678 ebx=7ffd6000 ecx=00001234 edx=7c92eb94 esi=011dd664 edi=011dd65c  
    eip=0040100b esp=0012ff50 ebp=00000000 iopl=0         nv up ei pl nz na pe nc  
    cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206  
    Dump01!sum+0xb:  
    0040100b 8b4508          mov     eax,dword ptr [ebp+8] ss:0023:00000008=????????

再根据堆栈信息，出错前往堆栈中压入了 ebx/esi/edi 几个寄存器的值，对比 0x0012ff50 处的堆栈，可知 0x0012ff50 正好是程序出错前的堆栈地址。同时还可以得到保存在堆栈上的ebp的值0012ff70，从而得到正确的出错堆栈。

    0:000> kv L = 0x0012ff5c  
    ChildEBP RetAddr  Args to Child                
    0012ff5c 0040103b 00001234 00005678 00000000 Dump01!sum+0xb (CONV: cdecl)  
    0012ff70 00401074 00001234 00005678 0012ffc0 Dump01!sumstub+0x25 (CONV: cdecl)  
    0012ff80 0040117b 00000001 00520eb0 00520e20 Dump01!main+0x1f (CONV: cdecl)  
    0012ffc0 7c816fd7 011dd65c 011dd664 7ffd6000 Dump01!mainCRTStartup+0xb4  
    0012fff0 00000000 004010c7 00000000 00000000 kernel32!BaseProcessStart+0x23 (FPO: [Non-Fpo])

从这个堆栈来看，起始地址从 kernel32!BaseProcessStart 开始，结束地址也正好在出错地址处，应该是正确的出错堆栈。

## 四：提高班：当上述方法都不行的时候：

## （dmp实例：帮SDK组定位的一个崩溃）