# Re: heeeeeeeeeeeeeeeelllllllllllllppppppppppppppppppppp

***Source:*** http://www.tech−archive.net/Archive/VC/microsoft.public.vc.mfc/2006−02/msg00090.html

- *From*: Joseph M. Newcomer <newcomer@xxxxxxxxxxxx>
- *Date*: Wed, 01 Feb 2006 12:52:28 −0500

```
HHHHHHHHHHHHHHHHEEEEEEEEEEEEEEEERRRRRRRRRRRRRRREEEEEEEEEEEEEEEE
IIIIIIIIIIIIIIIIIISSSSSSSSSSSSS AAAAAAAAAAAAAAAAAANNNNNNNNNNNNN
AAAAAAAAAAAAAAAANNNNNNNNNNNNNNNNSSSSSSSSSSWWWWWWWWWWWEEEEEEEEEEERRRRRRRR

First, try to make your subject lines meaningful instead of childish.

There are lots of problems in the code you show, and probably the real problems are in the
doce you didn't bother to show.  See comments below.
                                        joe


On 31 Jan 2006 08:32:30 −0800, "A P" <pinto.albert@xxxxxxxxx> wrote:

>i am trying to revive the file handles of an open process.
*****
Define 'revive'.  Are you suggesting these handles are dead?  If they are dead, why do you
think they have meaning?
****
>after
>reading the lists i found that for XP the class is 28 (which i hard
>coded for th time being).
****
Which lists?  It is hard to guess what you are talking about when there are no references
we can check to see what you are referring to.
****
>when i run this progeam in xp with test
>processes, i get invalid file handle when i try to run a test app that
>opens a file and runs in an infinite loop.
*****
It is worth pointing out that not a single piece of code you have shown in this rambling
and erroneous example has anything to do with how you obtain this handle.  So the problem
is most likely there, but since you don't give any of that code, it is impossible to check
to see if you are making a mistake at that point.  Given you may have a bogus handle
already, everything else you show is pretty meaningless.  Note that having a handle value
is not the same as having a valid handle, which I think may be the key point, but given
that no useful code was shown here, there's no way to tell what you did wrong.
****
>i print the handle in the
>test app also, and they are same in the list i get in my code also. can
>some one help.
****
OK, so suppose the process has handle 0x12345.  You print out, in your app, the numeric
value 0x12345.  Why do you presume that this numeric value represents a valid handle in
the context of the monitoring app?  Perhaps you have confused "numeric value of handle"
with "valid handle".  You will have to make the handle value a valid handle before you can
use it.  DupicateHandle is probably what you are missing.
```

```
****
>
>
>
>u may directly mail me at pinto.albert@xxxxxxxxx also.
>
>
>
>#include <windows.h>
>#include <stdio.h>
>#include <aclapi.h>
>#include <conio.h>
>#include <psapi.h>
>#include <tchar.h>
>#include <string.h>
>
>
>#define NT_SUCCESS(status) ((NTSTATUS)(status)>=0)
>#define STATUS_INFO_LENGTH_MISMATCH ((NTSTATUS)0xC0000004L)
>#define STATUS_ACCESS_DENIED ((NTSTATUS)0xC0000022L)
****
You should be able to get the latter two constants by including ntstatus.h, instead of
duplicating the definitions here
****
>
>#define BUFSIZE 256
>
>
>
>typedef enum _SYSTEM_INFORMATION_CLASS{
> SystemHandleInformation = 16
>} SYSTEM_INFORMATION_CLASS;
>
>/*
> *Information Class 16
>*/
>typedef struct _SYSTEM_HANDLE_INFORMATION{
> ULONG ProcessId;
> UCHAR ObjectTypeNumber;
> UCHAR Flags;
> USHORT Handle;
> PVOID Object;
> ACCESS_MASK GrantedAccess;
>} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;
>
>
>#define InitializeObjectAttributes( p, n, a, r, s ) { (p)->Length =
>sizeof( OBJECT_ATTRIBUTES ); (p)->RootDirectory = r; (p)->Attributes =
>a; (p)->ObjectName = n; (p)->SecurityDescriptor = s;
>(p)->SecurityQualityOfService = NULL; }
****
The odd mishmash of code that uses a blend of 8-bit and Unicode is completely nonsensical
in the light of calling the undocumented API, which is exclusively Unicode.
****
>
>
>typedef ULONG ( __stdcall *RTLNTSTATUSTODOSERROR ) ( IN NTSTATUS Status
>);
>typedef NTSTATUS ( __stdcall *ZWQUERYSYSTEMINFORMATION ) ( IN
>SYSTEM_INFORMATION_CLASS SystemInformationClass, IN OUT PVOID
>SystemInformation, IN ULONG SystemInformationLength, OUT PULONG
>ReturnLength OPTIONAL );
```

```
>
>
>
>
>
>/*************************************************************************
> * *
> * Function Prototype *
> * *
>*************************************************************************/
>
>static DWORD GetEprocessFromPid ( ULONG PID );
>static BOOL LocateNtdllEntry ( void );
>
>
>/*************************************************************************
> * *
> * Static Global Var *
> * *
>*************************************************************************/
>
>static RTLNTSTATUSTODOSERROR RtlNtStatusToDosError = NULL;
>static ZWQUERYSYSTEMINFORMATION ZwQuerySystemInformation = NULL;
>
>static HMODULE hModule = NULL;
>/*************************************************************************/
****
The presence of static global variables tends to introduce nervousness.  There shouldn't
be a need to declare these as pointers anyway (just link with the correct library).
Nebbett's book on the undocumented Windows API shows how to do this.  Just link with
ntdll.lib.
****
>
>
>
>
>
>
>
>const char * GetFileNameFromHandle(HANDLE hFile)
****
As soon as I see char * I question the sanity of the code.  Why are you assuming 8-bit
characters?
****
>{
>
>  BOOL bSuccess = FALSE;
>  TCHAR pszFilename[MAX_PATH+1];
>  HANDLE hFileMap;
>  char *strName;
>  char buff[512];
****
How quaint!  Fixed-size allocations, use of obsolete concepts such as 'char'.
****
>
>
>  // Get the file size.
>  DWORD dwFileSizeHi = 0;
>  DWORD dwFileSizeLo = GetFileSize(hFile, &dwFileSizeHi);
>
>
>  strName=(char *)malloc(1000);
```

```
****
If this is C++, the use of malloc is really quaint.  If it is C, how did you arrive at the
magical value 1000, and why are you using char instead of TCHAR, and why is sizeof(TCHAR)
not part of the allocation size computation?
****
>  if( dwFileSizeLo == 0 && dwFileSizeHi == 0 ){
>     //printf("Cannot map a file with a length of zero.\n");
>     //return FALSE;
>
>  strcpy(strName,"Cannot map a file with a length of zero");
****
In MFC it would make more sense to use CStrings here.  In C, it would make more sense to
defer the allocation until it is known how big it is, and in this case, returning
_tcsdup(_T("Cannot map a file with length of zero")) might make more sense, assuming the
caller has a way to tell an error message text from the filename "Cannot map a file with a
length of zero").  I'd suggest a BOOL−returning frunction and a CString & parameter for
the text.
****
>  return strName;
>  }
>
>
>  // Create a file mapping object.
>  hFileMap = CreateFileMapping(hFile,
>                    NULL,
>                    PAGE_READONLY,
>                    0,
>                    1,
>                    NULL);
>
****
You have not indicated, in this long and rambling message, exactly which of these API
calls failed.and gave the invalid handle value.  That, of course, would have made it a lot
easier to analyze your problem.  I suspect that this is the API call that failed, and it
failed because you truly don't have a valid handle; what you have is a handle value.
****
>  if (hFileMap){
>    // Create a file mapping to get the file name.
>    void* pMem = MapViewOfFile(hFileMap, FILE_MAP_READ, 0, 0, 1);
>
>    if (pMem){
>
>      if (GetMappedFileName (GetCurrentProcess(),
>                             pMem,
>                             pszFilename,
>                             MAX_PATH)){
>
>        // Translate path with device name to drive letters.
>        TCHAR szTemp[BUFSIZE];
>        szTemp[0] = '\0';
****
You are assigning an 8−bit character code to a possible Unicode string.  _T('\0').
Assuming there is any point to setting this value at all.
****
>
>        if (GetLogicalDriveStrings(BUFSIZE−1, szTemp)){
>          TCHAR szName[MAX_PATH];
>          TCHAR szDrive[3] = TEXT(" :");
****
Oh, wow!  A presumtion that a drive string is one letter!  How really retro! (Share names
can be up to MAX_PATH characters).  Did you actually read the documentation for
GetLogicalDriveStrings?
```

```
****
>          BOOL bFound = FALSE;
>          TCHAR* p = szTemp;
>
>          do{
>            // Copy the drive letter to the template string
>            *szDrive = *p;
****
Why do you think this is only one letter?  Just because in one limited test case you never
saw a drive name longer than one characters?
****
>
>            // Look up each device name
>            if (QueryDosDevice(szDrive, szName, BUFSIZE)){
>              UINT uNameLen = _tcslen(szName);
>
>
>              if (uNameLen < MAX_PATH){
>                bFound = _tcsnicmp(pszFilename, szName,
>                    uNameLen) == 0;
>
>                if (bFound){
>                  // Reconstruct pszFilename using szTemp
>                  // Replace device path with DOS path
>                  TCHAR szTempFile[MAX_PATH];
>                  _stprintf(szTempFile,
>                            TEXT("%s%s"),
>                            szDrive,
>                            pszFilename+uNameLen);
****
Now here you understand that you have to deal with Unicode.  Other places you assume that
you are 8-bit only.  Why the odd mix?  But why aren't you considering things like
_tmakepath?
****
>                  _tcsncpy(pszFilename, szTempFile, MAX_PATH);
>                }
>              }
>            }
>
>            // Go to the next NULL character.
>            while (*p++);
*****
Sigh.  So many bugs, so little code.  Read the documentation for GetLogicalDriveStrings
then rewrite this code to be correct.  You can assume that any name is longer than one
character, as is consistent with the documentation.  Or, rather, this particular line DOES
assume the correct interpretation of GetLogicalDriveStrings, and nothing else does
****
>          } while (!bFound && *p); // end of string
*****
Generally, any boolean expression more complex than TRUE leads to code that is hard to
understand and maintain.  Why not just do a 'break' when the correct value is found and
eliminate the bFound variable entirely?  The code becomes simpler.  And as the first test,
if(*p == _T('\0')) break;
****
>        }  //if (GetLogicalDriveStrings(BUFSIZE-1, szTemp)){
>      }  //if (GetMappedFileName
>   else{  // GetMappedFileName() == 0
>    LPSTR lpMsgBuf;
****
The definition is LPTSTR, not LPSTR, so please read the documentation
****
>       FormatMessage(
```

```
>        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
>    NULL,
>    GetLastError(),
****
The FIRST thing you do is call GetLastError() and save the result; if you were to add
anything ahead of this statement, GetLastError() could be erroneous.
****
>    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
>    (LPTSTR) &lpMsgBuf,
>    0,
>    NULL
>    );
>
>    // Display the string.
>    MessageBox( NULL, lpMsgBuf, "GetLastError() for
>GetMappedFileName()", MB_OK|MB_ICONINFORMATION );
*****
Use _T() around all string literals.  This mishmash of Unicode and 8-bit characters is
bizarre.  Using NULL as the parent is almost always a mistake (it means your messagebox
might come up UNDERNEATH your application!).  Use AfxMessageBox.  Or pass in the desired
window handle.  But don't use NULL.  It sometimes gives the illusion that it works.  It is
only an illusion.
****
>
>    // Free the buffer.
>          LocalFree( lpMsgBuf );
>   } //else{  // GetMappedFileName() == 0
>
>
>   bSuccess = TRUE;
>      UnmapViewOfFile(pMem);
>    } //if (pMem)
> else{ //else...if (pMem)
>     LPSTR lpMsgBuf;
>       FormatMessage(
>        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
>    NULL,
>    GetLastError(),
****
Same caveat as above.
****
>    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
>    (LPTSTR) &lpMsgBuf,
****
You cast it as an LPTSTR but declare it as an LPSTR!  Try being consistent
****
>    0,
>    NULL
>    );
>
>    // Display the string.
>    MessageBox( NULL, lpMsgBuf, "GetLastError() for
>GetMappedFileName()", MB_OK|MB_ICONINFORMATION );
****
Same problem as last MessageBox
****
>
>    // Free the buffer.
>          LocalFree( lpMsgBuf );
> }
>      bSuccess = TRUE;
>      UnmapViewOfFile(pMem);
```

```
>    } //if (hFileMap)
> else{
>  LPSTR lpMsgBuf;
****
Ditto
****
>       FormatMessage(
>        FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
>    NULL,
>    GetLastError(),
****
Ditto
****
>    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // Default language
>    (LPTSTR) &lpMsgBuf,
>    0,
>    NULL
>    );
>
>    // Display the string.
>    MessageBox( NULL, lpMsgBuf, "GetLastError() for
>GetMappedFileName()", MB_OK|MB_ICONINFORMATION );
****
Ditto
***
>
>    // Free the buffer.
>          LocalFree( lpMsgBuf );
> }
>
>    CloseHandle(hFileMap);
>
>
>
>  //printf("File name is %s\n", pszFilename);
****
Why do you assume that the name is 8-bit characters?  This code is a random collection of
apparently careful Unicode-aware code and retro 8-bit only code.  Compile it as Unicode
and make sure it still compiles
****
>
>
>  strcpy(strName,pszFilename);
****
_tcscpy
****
>  return strName;
>  //return(bSuccess);
>}
>
>
>
>
>
>static DWORD GetEprocessFromPid ( ULONG PID )
>{
> NTSTATUS status;
> PVOID buf = NULL;
> ULONG size = 1;
> ULONG NumOfHandle = 0;
> ULONG i;
> PSYSTEM_HANDLE_INFORMATION h_info = NULL;
```

```
>
> for ( size = 1; ; size *= 2 ){
>  if ( NULL == ( buf = calloc( size, 1 ) ) ){
>   fprintf( stderr, "calloc( %u, 1 ) failed\n", size );
>   goto GetEprocessFromPid_exit;
****
```

Why fprintf?  Why assume 8-bit characters?  Why calloc (you're about to overwrite the data, so why does it matter if it is zeroed or not?  Why not 'new'?

Isn't 'goto' rather quaint and so-1960s?  Since the pointer is NULL anyway, and there's nothing else to do, why not just return FALSE here, instead of something as bizarre as a goto that tests the value which is now known to be NULL, then returns FALSE?

```
****
>  }
****
```

Writing readable code means making sure that you never, ever end a line with {; that every { is vertically above its }.  This code is hard to read

```
*****
>
>  status = ZwQuerySystemInformation( SystemHandleInformation, buf,
>size, NULL );
>  if ( !NT_SUCCESS( status ) ){
>   if ( STATUS_INFO_LENGTH_MISMATCH == status ){
>    free( buf );
>    buf = NULL;
>   }
>   else{
>    printf( "ZwQuerySystemInformation() failed");
>    goto GetEprocessFromPid_exit;
****
```

Putting two levels of call here means you won't need anything as retro as a goto.  Given that the only purpose is to free the buffer, but you know the buffer exists, you know at this point you can free the buffer and return FALSE, and you don't need something as complex as a goto

```
****
>   }
>  }
>  else{
>   break;
>  }
> } /* end of for */
>
>
> //NumOfHandle = (ULONG)buf;
> NumOfHandle = *(ULONG*)buf;
>
> h_info = ( PSYSTEM_HANDLE_INFORMATION )((ULONG)buf+4);
>
> for(i = 0; i<NumOfHandle ;i++){
>  if( ( h_info[i].ProcessId == PID )  &&( h_info[i].ObjectTypeNumber ==
>28 ))//&&( h_info[i].Handle==0x3d8 ) )
>//  {
>   printf("Handle:0x%x,OBJECT 0x%x, Object Type Number is : %d, Name:
>%s\n\r",h_info[i].Handle,h_info[i].Object,h_info[i].ObjectTypeNumber,GetFileNameFromHandle((HAND
>));
****
```

Another strange blend of 8-bit and Unicode.  And by writing out this massively long line as if it would all work, you have made it hard to debug what is going on.  I notice also that you have a serious memory leak.  Also the number of things that may or may not be commented out makes the code nearly unreadable.  Try to break this down into simpler tasks.  Program vertically.

```
The problem is almost certainly that you have the handle value, not a valid handle.  You
need to use DupicateHandle to convert the handle value to a valid handle.
****
>//   return((DWORD)(h_info[i].Object));
>//  }
> }
> GetEprocessFromPid_exit:
> if ( buf != NULL ){
>  free( buf );
>  buf = NULL;
> }
> return(FALSE);
>}
>
>
>/*
>* ntdll.dll
>*/
>static BOOL LocateNtdllEntry ( void )
>{
> BOOL ret = FALSE;
> char NTDLL_DLL[] = "ntdll.dll";
> HMODULE ntdll_dll = NULL;
>
>
> if ( ( ntdll_dll = GetModuleHandle( NTDLL_DLL ) ) == NULL ){
>  printf( "GetModuleHandle() failed");
>  return( FALSE );
> }
> if ( !( ZwQuerySystemInformation = ( ZWQUERYSYSTEMINFORMATION
>)GetProcAddress( ntdll_dll, "ZwQuerySystemInformation" ) ) ){
>  goto LocateNtdllEntry_exit;
> }
> ret = TRUE;
>
> LocateNtdllEntry_exit:
>
> if ( FALSE == ret ){
>  printf( "GetProcAddress() failed");
> }
*****
It is always sloppy to compare any boolean result to TRUE or FALSE, since a boolean value
ALREADY stands for TRUE or FALSE.  You wouldn't write
        if((value > 0) == TRUE)
so why write a comparison of a boolean to a boolean to get, wow, a boolean?  Write
        if(!ret)
which is the correct way to test booleans
****
> ntdll_dll = NULL;
> return( ret );
>} /* end of LocateNtdllEntry */
>
*****
Actually, Nebbett shows how to link with ntdll.lib so you don't need to do this at
runtime.  I've never had to use GetProcAddress unless I'm looking for an interface which
is OS-specific.
*****
>
>int main(int argc,char **argv)
****
int _tmain(int arcg, TCHAR ** argv)
****
```

```
>{
>
> HANDLE h;
> ULONG inPid;
> LocateNtdllEntry( );
>
>
>
>
> //OpenProcess( PROCESS_ALL_ACCESS,FALSE,GetCurrentProcessId() );
>
> printf("\nEnter the Pid of the proces to be opened : ");
> scanf("%ul",&inPid);
*****
Good grief!  A console application that prompts!  How very 1960s.  Note that there are
serious problems with using MessageBox in a console app.   So now the question arises as
to why you have such an odd blend of printf (should be _tprintf) and MessageBox calls...
****
>
> h = OpenProcess( PROCESS_ALL_ACCESS,FALSE,inPid);
> if(!h){
****
This is weird.  You check a boolean by writing
        if(ret == FALSE)
and you check a handle by treating it as a boolean!  You should write
        if(h == NULL)
I see much serious confusing about programming style here
****
>  LPVOID lpMsgBuf;
****
The specification is LPTSTR, so why do you declare it as LPVOID?
****
>    LPVOID lpDisplayBuf;
>    DWORD dw = GetLastError();
>
>    FormatMessage(
>        FORMAT_MESSAGE_ALLOCATE_BUFFER |
>        FORMAT_MESSAGE_FROM_SYSTEM,
>        NULL,
>        dw,
>        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
>        (LPTSTR) &lpMsgBuf,
>        0, NULL );
>
>    lpDisplayBuf = LocalAlloc(LMEM_ZEROINIT,
>        strlen(lpMsgBuf)+40);
>    wsprintf(lpDisplayBuf,
>        "failed with error %d: %s",
>         dw, lpMsgBuf);
*****
You ;should never, ever use sprintf, wsprintf, etc.  And why do you assume 8-bit
characters?  Particularly because wsprintf wants LPTSTR, which means _T() for a literal
string.  The whole set of ideas about using LocalAlloc and similar techniques, with random
values like +40, is very out-of-date and represents bad programming practice.

Note that the specification of wsprintf is that the first parameter is an LPTSTR, yet you
have an LPVOID.  Is there some reason you choose to disregard the specified interfaces?

Why do you use such odd combinations of allocation?  In some cases, you use malloc; in
others, calloc; in others, LocalAlloc.  Choose one method and use it consistently.  The
only time you're required to use LocalFree is for FormatMessage, and this should be
isolated into a single subroutine so it isn't distributed throughout your code.
```

```
*****
>    MessageBox(NULL, lpDisplayBuf, "Error", MB_OK);
****
See above comment about the fact that MessageBox is hard to use in a console app.  Why
didn't you just do a _tprintf?  Then you could ignore the problems of allocating and
freeing buffers.
*****
>
>    LocalFree(lpMsgBuf);
>    LocalFree(lpDisplayBuf);
>    ExitProcess(dw);
> }
>
> /*DWORD Addr =*/ //GetEprocessFromPid( (DWORD)GetCurrentProcessId() );
> GetEprocessFromPid( inPid);
>
>//printf("result: Current EPROCESS's Address is 0x%x \n\r",Addr);
>
>return TRUE;
****
So you use a retro console app.  Then you return the wrong value.  A console app returns 0
if successful, and a nonzero numeric error code if it fails.
****
>}
Joseph M. Newcomer [MVP]
email: newcomer@xxxxxxxxxxx
Web: http://www.flounder.com
MVP Tips: http://www.flounder.com/mvp_tips.htm
.
```