How do I pass a lot of data to a process when it starts up?

devblogs.microsoft.com/oldnewthing/20031211-00

December 11, 2003

Raymond Chen

As we discussed yesterday, if you need to pass more than 32767 characters of information to a child process, you'll have to use something other than the command line.

One method is to wait for the child process to go input idle, then `FindWindow` for some agreed-upon window and send it a `WM_COPYDATA` message. This method has a few problems:

- You have to come up with some way of knowing that the child process has created its window so you can start looking for it. (`WaitForInputIdle` is helpful here.)
- You have to make sure the window you found belongs to the child process and isn't just some other window which happens to have the same name by coincidence. Or, perhaps, not by coincidence: If there is more than once instance of the child process running, you will need to make sure you're talking to the right one. (`GetWindowThreadProcessId` is helpful here.)
- You have to hope that nobody else manages to find the window and send it the `WM_COPYDATA` before you do. (If they do, then they have effectively taken over your child process.)
- The child process needs to be alert for the possibility of a rogue process sending bogus `WM_COPYDATA` messages in an attempt to confuse it.

The method I prefer is to use anonymous shared memory. The idea is to create a shared memory block and fill it with goodies. Mark the handle as inheritable, then spawn the child process, passing the numeric value of the handle on the command line. The child process parses the handle out of its command line and maps the shared memory block to see what's in it.

Remarks about this method:

- You need to be careful to validate the handle, in case somebody tries to be sneaky and pass you something bogus on your command line.

- In order to mess with your command line, a rogue process needs PROCESS_VM_WRITE permission, and in order to mess with your handle table, it needs PROCESS_DUP_HANDLE permission. These are securable access masks, so proper choice of ACLs will protect you. (And the default ACLs are usually what you want anyway.)
- There are no names that can be squatted or values that can be spoofed (assuming you've protected the process against PROCESS_VM_WRITE and PROCESS_DUP_HANDLE).
- Since you're using a shared memory block, nothing actually is copied between the two processes; it is just remapped. This is more efficient for large blocks of data.

Here's a sample program to illustrate the shared memory technique.

```
#include <windows.h>
#include <shlwapi.h>
#include <strsafe.h>
struct STARTUPPARAMS {
    int iMagic;     // just one thing
};
```

In principle, the `STARTUPPARAMS` can be arbitrarily complicated, but for illustrative purposes, I'm just going to pass a single integer.

```
STARTUPPARAMS *CreateStartupParams(HANDLE *phMapping)
{
    STARTUPPARAMS *psp = NULL;
    SECURITY_ATTRIBUTES sa;
    sa.nLength = sizeof(sa);
    sa.lpSecurityDescriptor = NULL;
    sa.bInheritHandle = TRUE;
    HANDLE hMapping = CreateFileMapping(INVALID_HANDLE_VALUE,
                &sa, PAGE_READWRITE, 0,
                sizeof(STARTUPPARAMS), NULL);
    if (hMapping) {
        psp = (STARTUPPARAMS *)
                MapViewOfFile(hMapping, FILE_MAP_WRITE,
                    0, 0, 0);
        if (!psp) {
            CloseHandle(hMapping);
            hMapping = NULL;
        }
    }
    *phMapping = hMapping;
    return psp;
}
```

The `CreateStartupParams` function creates a `STARTUPPARAMS` structure in an inheritable shared memory block. First, we fill out a `SECURITY_ATTRIBUTES` structure so we can mark the handle as inheritable by child processes. Setting the `lpSecurityDescriptor` to NULL means that we will use the default security descriptor, which is fine for us. We then create a shared memory object of the appropriate size, map it into memory, and return both the handle and the mapped address.

```
STARTUPPARAMS *GetStartupParams(LPSTR pszCmdLine, HANDLE *phMapping)
{
    STARTUPPARAMS *psp = NULL;
    LONGLONG llHandle;
    if (StrToInt64ExA(pszCmdLine, STIF_DEFAULT, &llHandle)) {
        *phMapping = (HANDLE)(INT_PTR)llHandle;
        psp = (STARTUPPARAMS *)
                MapViewOfFile(*phMapping, FILE_MAP_READ, 0, 0, 0);
        if (psp) {
            //  Now that we've mapped it, do some validation
            MEMORY_BASIC_INFORMATION mbi;
            if (VirtualQuery(psp, &mbi, sizeof(mbi)) >= sizeof(mbi) &&
                mbi.State == MEM_COMMIT &&
                mbi.BaseAddress == psp &&
                mbi.RegionSize >= sizeof(STARTUPPARAMS)) {
                // Success!
            } else {
                // Memory block was invalid - toss it
                UnmapViewOfFile(psp);
                psp = NULL;
            }
        }
    }
    return psp;
}
```

The `GetStartupParams` function is the counterpart to `CreateStartupParams`. It parses a handle from the command line and attempts to map a view. If the handle isn't a file mapping handle, the call to `MapViewOfFile` will fail, so we get that part of the parameter validation for free. We use `VirtualQuery` to validate the size of the memory block. (We can't use a strict equality test since the value we get back will be rounded up to the nearest page multiple.)

```
void FreeStartupParams(STARTUPPARAMS *psp, HANDLE hMapping)
{
    UnmapViewOfFile(psp);
    CloseHandle(hMapping);
}
```

After we're done with the startup parameters (either on the creation side or the consumption side), we need to free them to avoid a memory leak. That's what `FreeStartupParams` is for.

```
void PassNumberViaSharedMemory(HANDLE hMapping)
{
    TCHAR szModule[MAX_PATH];
    TCHAR szCommand[MAX_PATH * 2];
    DWORD cch = GetModuleFileName(NULL, szModule, MAX_PATH);
    if (cch > 0 && cch < MAX_PATH &&
        SUCCEEDED(StringCchPrintf(szCommand, MAX_PATH * 2,
                  TEXT("\"%s\" %I64d"), szModule,
                  (INT64)(INT_PTR)hMapping))) {
        STARTUPINFO si = { sizeof(si) };
        PROCESS_INFORMATION pi;
        if (CreateProcess(szModule, szCommand, NULL, NULL,
                          TRUE,
                          0, NULL, NULL, &si, &pi)) {
            CloseHandle(pi.hProcess);
            CloseHandle(pi.hThread);
        }
    }
}
```

Most of the work here is just building the command line. We run ourselves (using the `GetModuleFileName(NULL)` trick), passing the numerical value of the handle on the command line, and passing `TRUE` to `CreateProcess` to indicate that we want inheritable handles to be inherited. Note the extra quotation marks in case our program's name contains a space.

```
int CALLBACK
WinMain(HINSTANCE hinst, HINSTANCE hinstPrev,
        LPSTR pszCmdLine, int nShowCmd)
{
    HANDLE hMapping;
    STARTUPPARAMS *psp;
    if (pszCmdLine[0]) {
        psp = GetStartupParams(pszCmdLine, &hMapping);
        if (psp) {
            TCHAR sz[64];
            StringCchPrintf(sz, 64, TEXT("%d"), psp->iMagic);
            MessageBox(NULL, sz, TEXT("The Value"), MB_OK);
            FreeStartupParams(psp, hMapping);
        }
    } else {
        psp = CreateStartupParams(&hMapping);
        if (psp) {
            psp->iMagic = 42;
            PassNumberViaSharedMemory(hMapping);
            FreeStartupParams(psp, hMapping);
        }
    }
    return 0;
}
```

At last we put it all together. If we have a command line parameter, then this means that we are the child process, so we convert it into a `STARTUPPARAMS` and display the number that was passed. If we don't have a command line parameter, then this means that we are the parent process, so we create a `STARTUPPARAMS`, stuff the magic number into it (42, of course), and pass it to the child process.

So there you have it: Passing a "large" (well, okay, small in this example, but it could have been megabytes if you wanted) amount of data securely to a child process.

Raymond Chen

Follow