Stupid debugger tricks: Calling functions and methods

devblogs.microsoft.com/oldnewthing/20070427-00

April 27, 2007

Raymond Chen

Back in the old days, if you wanted to call a function from inside the debugger, you had to do it by hand: Save the registers, push the parameters onto the stack (or into registers if the function uses `fastcall` or `thiscall`) push the address of the `ntdll!DbgBreakPoint` function, move the instruction pointer to the start of the function you want to call, then hit "g" to resume execution. The function runs then returns to the `ntdll!DbgBreakPoint`, where the debugger regains control and you can look at the results. Then restore the registers (including the original instruction pointer) and resume debugging. (That paragraph was just a quick recap; I'm assuming you already knew that.)

The Windows symbolic debugger engine (the debugging engine behind `ntsd`, `cdb` and `windbg`) can now automate this process. Suppose you want to call this function:

```
int DoSomething(int i, int j);
```

You can ask the debugger to do all the heavy lifting:

```
0:001> .call ABC!DoSomething(1,2)
Thread is set up for call, 'g' will execute.
WARNING: This can have serious side-effects,
including deadlocks and corruption of the debuggee.
0:001> r
eax=7ffde000 ebx=00000001 ecx=00000001 edx=00000003 esi=00000004 edi=00000005
eip=10250132 esp=00a7ffbc ebp=00a7fff4 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ABC!DoSomething:
10250132 55               push    ebp
0:001> dd esp
00a7ffbc  00a7ffc8 00000001 00000002 ccfdebcc
```

Notice that the debugger nicely pushed the parameters onto the stack and set the `eip` register for you. All you have to do is hit "g" and the `DoSomething` function will run. Once it returns, the debugger will restore the original state.

This technique even works with C++ methods:

```
// pretend that we know that 0x00131320 is an IStream pointer
0:001> .dvalloc 1000
Allocated 1000 bytes starting at 00a80000
0:001> .call ABC!CAlphaStream::Read(0x00131320, 0xa80000, 0x1000, 0)
Thread is set up for call, 'g' will execute.
WARNING: This can have serious side-effects,
including deadlocks and corruption of the debuggee.
```

Notice that when calling a nonstatic C++ method, you have to pass the “this” parameter as an explicit first parameter. The debugger knows what calling convention to use and puts the registers in the correct location. In this case, it knew that `CAlphaStream::Read` uses the `stdcall` calling convention, so the parameters have all been pushed onto the stack.

And what’s with that `.dvalloc` command? That’s another debugger helper function that allocates some memory in the debugged process’s address space. Here, we used it to allocate a buffer that we want to read into.

But what if you want to call a method on an interface, and you don’t have the source code to the implementation? For example, you want to read from a stream that was passed to you from some external component. Well, you can play a little trick. You can pretend to call a function that you **do** have the source code to, one that has the same function signature, and then move the `eip` register to the desired entry point.

```
// pretend that we know that 0x00131320 is an IStream pointer
0:000>  dp 131320 l1
00131320  77f6b5e8 // vtable
0:000> dps 77f6b5e8 l4
77f6b5e8  77fbff0e SHLWAPI!CFileStream::QueryInterface
77f6b5ec  77fb34ed SHLWAPI!CAssocW2k::AddRef
77f6b5f0  77f6b670 SHLWAPI!CFileStream::Release
77f6b5f4  77f77474 SHLWAPI!CFileStream::Read
0:000> .call SHLWAPI!CFileStream::Read(0x00131320, 0xa80000, 0x1000, 0)
                ^ Symbol not a function in '.call SHLWAPI!CFileStream::Read'
```

That error message is the debugger’s somewhat confusing way of saying, “I don’t have enough information available to make that function call.” But that’s okay, because we have a function that’s “close enough”, namely `CAlphaStream::Read`:

```
0:001> .call ABC!CAlphaStream::Read(0x00131320, 0xa80000, 0x1000, 0)
Thread is set up for call, 'g' will execute.
WARNING: This can have serious side-effects,
including deadlocks and corruption of the debuggee.
0:000> r eip=SHLWAPI!CFileStream::Read
0:000> r
eax=00131320 ebx=0007d628 ecx=00130000 edx=0013239e esi=00000000 edi=00000003
eip=77f77474 esp=0007d384 ebp=0007d3b0 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
SHLWAPI!CFileStream::Read:
77f77474 8bff             mov     edi,edi
```

Woo-hoo! We got `ABC!CAlphaStream::Read` to push all the parameters for us, and then **whoosh** we swap out that function and slip `CFileStream::Read` in its place. Now you can hit “g” to execute the `CFileStream::Read` call.

This just skims the surface of what you can do with the `.call` command. Mix in some C++ expression evaluation and you’ve got yourself a pretty nifty “pseudo-immediate mode” expression evaluator.

Raymond Chen

Follow