Challenges of Debugging Optimized x64 Code - Ntdebugging Blog - Site Home - MSDN Blogs

Notebook:	Windbg
Created:	6/10/2014 2:46 PM
URL:	http://blogs.msdn.com/b/ntdebugging/archive/2009/01/09/challenges-of-debuggi...

Challenges of Debugging Optimized x64 Code

ntdebug 9 Jan 2009 6:50 PM | 7

RATE THIS

If you have not had the luxury of debugging optimized x64 code as of yet, don't wait much longer and fall behind the times! Due to the x64 fastcall-like calling convention coupled with the abundance of general purpose registers, finding variable values at arbitrary points in a call stack can be very tricky indeed.

In this article, I'd like to detail some of my favorite techniques for debugging optimized x64 code. But before digging into these techniques, let's first have a quick overview of the x64 calling convention.

The x64 Calling Convention

Those of you familiar with the fastcall calling convention on x86 platforms will recognize the similarities to the x64 calling convention. Whereas you typically have to maintain knowledge of multiple calling conventions on x86 platforms, on x64 platforms there is currently just one. (Of course, I'm excluding the case of no calling convention which one can achieve with __declspec(naked) or by coding in straight assembly.)

I won't go into all of the various nuances of the x64 calling convention, therefore I recommend you check out the following link (**http://msdn.microsoft.com/en-us/library/ms794533.aspx**). But commonly, the first four parameters into a function are passed via the registers rcx, rdx, r8, and r9. If the function accepts more than four parameters, those parameters are passed on the stack. (Those of you familiar with the x86 fastcall calling convention where the first two parameters are passed in ecx and edx will recognize the similarities).

To help illustrate how the x64 calling convention works, I have created some simple example code. Although the code is contrived and far from real-world code, it demonstrates some scenarios that are likely to encounter in the real word. The code is shown below.

```
#include <stdlib.h>

#include <stdio.h>

#include <windows.h>

__declspec(noinline)
```

```
void
FunctionWith4Params( int param1, int param2, int param3,
                     int param4 )
{
    size_t lotsOfLocalVariables1 = rand();
    size_t lotsOfLocalVariables2 = rand();
    size_t lotsOfLocalVariables3 = rand();
    size_t lotsOfLocalVariables4 = rand();
    size_t lotsOfLocalVariables5 = rand();
    size_t lotsOfLocalVariables6 = rand();
    DebugBreak();
    printf( "Entering FunctionWith4Params( %X, %X, %X, %X )\n",
            param1, param2, param3, param4 );
    printf( "Local variables: %X, %X, %X, %X, %X, %X \n",
            lotsOfLocalVariables1, lotsOfLocalVariables2,
            lotsOfLocalVariables3, lotsOfLocalVariables4,
            lotsOfLocalVariables5, lotsOfLocalVariables6 );
}
__declspec(noinline)
void
FunctionWith5Params( int param1, int param2, int param3,
                     int param4, int param5 )
{
    FunctionWith4Params( param5, param4, param3, param2 );
    FunctionWith4Params( rand(), rand(), rand(), rand() );
}
__declspec(noinline)
void
FunctionWith6Params( int param1, int param2, int param3,
                     int param4, int param5, int param6 )
{
    size_t someLocalVariable1 = rand();
    size_t someLocalVariable2 = rand();
    printf( "Entering %s( %X, %X, %X, %X, %X, %X )\n",
            "FunctionWith6Params",
            param1, param2, param3, param4, param5, param6 );
    FunctionWith5Params( rand(), rand(), rand(),
                         param1, rand() );
    printf( "someLocalVariable1 = %X, someLocalVariable2 = %X\n",
            someLocalVariable1, someLocalVariable2 );
}
int
main( int /*argc*/, TCHAR** /*argv*/ )
{
    // I use the rand() function throughout this code to keep
    // the compiler from optimizing too much.  If I had used
    // constant values, the compiler would have optimized all
    // of these away.
    int params[] = { rand(), rand(), rand(),
                     rand(), rand(), rand() };
    FunctionWith6Params( params[0], params[1], params[2],
                         params[3], params[4], params[5] );
    return 0;
}
```

Cut and paste this code into a cpp file (such as example.cpp). I used the Windows SDK (specifically the Windows SDK CMD Shell) to compile this code as C++ code by using the following command line:

```
cl /EHa /Zi /Od /favor:INTEL64 example.cpp /link /debug
```

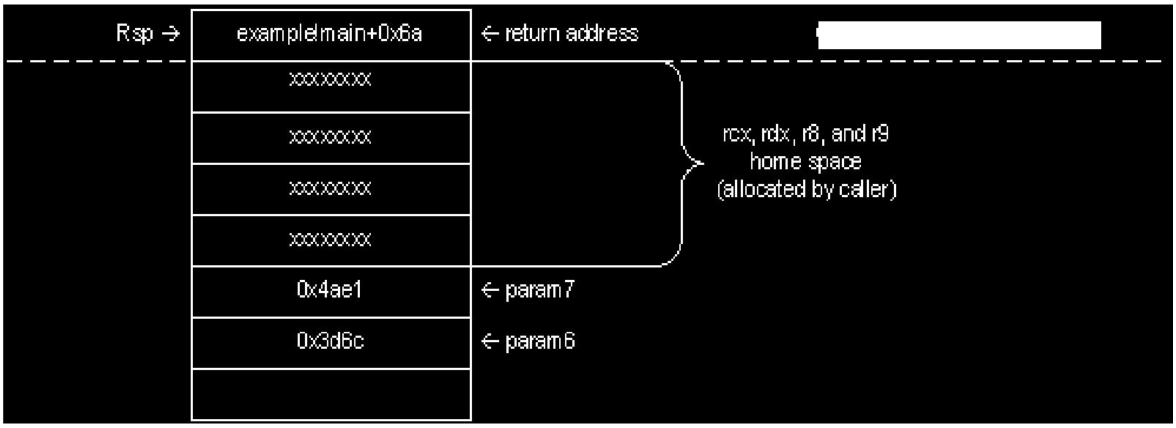
Notice the /Od switch. This disables all optimizations. Later on, I'll enable maximum optimization and that's when the fun begins!

Once you have the executable module built (mine is named example.exe), then you can fire it up in the debugger as follows:

```
windbg -Q -c "bu example!main;g;" example.exe
```

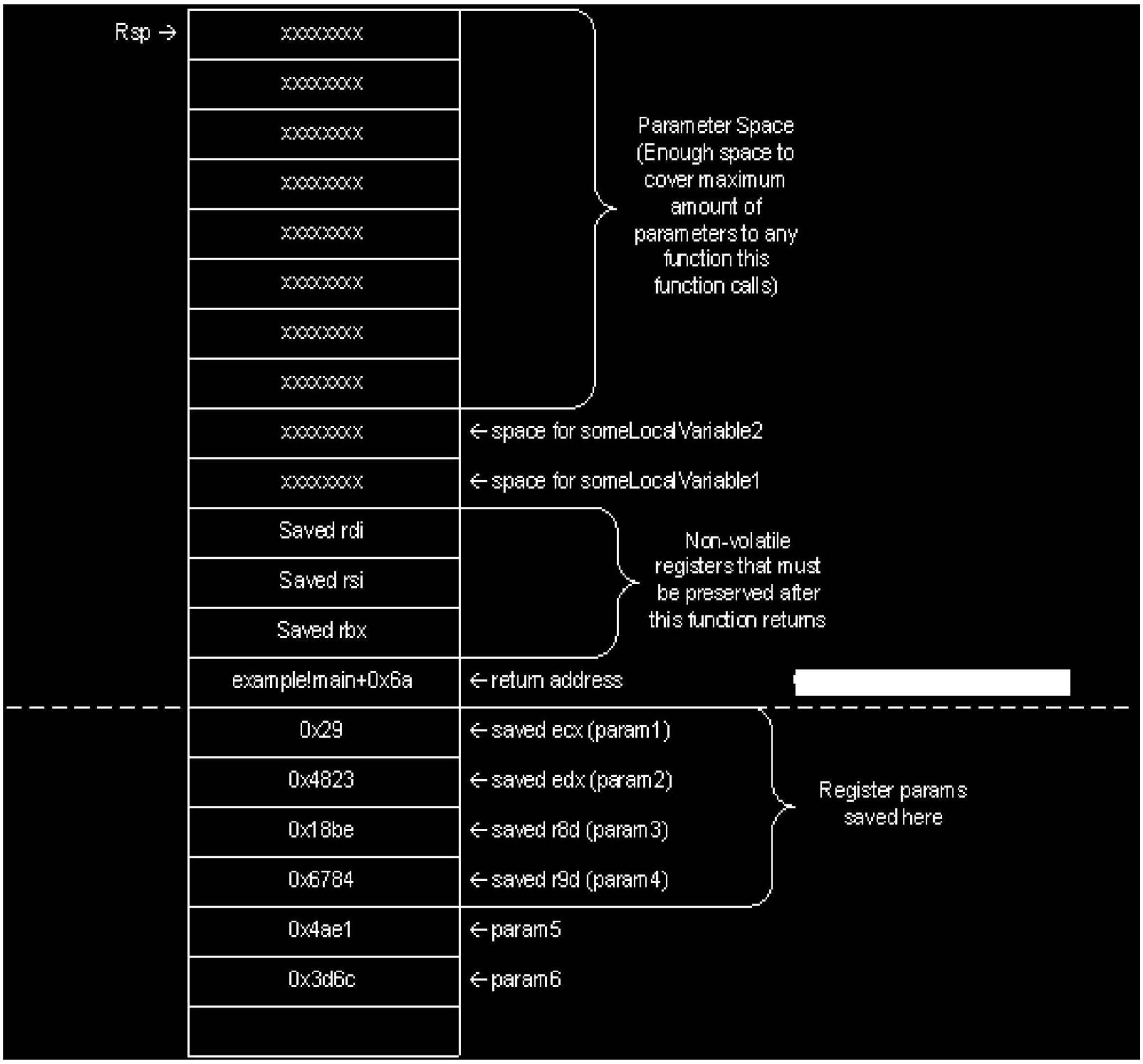
The command above will launch the application in windbg, set a breakpoint on the main() routine, and then go to that breakpoint.

Now, let’s have a look at a diagram of what the stack looks like when FunctionWith6Params() gets called. The diagram shown below illustrates the stack when the instruction pointer is at the beginning of the code for FunctionWith6Params() but before the prolog code has executed:



Notice that the caller, in this case main(), allocated enough space on the stack for all six parameters to FunctionWith6Params() even though the first four parameters are passed in via registers. The extra space on the stack is commonly referred to as the “home space” for the register parameters. In the previous diagram, I have shown those slots filled with xxxxxxxx to indicate that the values within there are virtually random at this point. That’s because the caller, main(), does not initialize these slots. The called function, at its discretion, may store the first four parameters in this space for safe keeping. This is exactly what happens in non-optimized builds and is a huge debugging convenience since you can easily find the contents of the first four parameters on the stack if you need to. Additionally, windbg stack commands such as kb and kv which show these first few parameters will report true results.

With all of that said, here is what the stack looks like after the prolog code in FunctionWith6Params() executes:



The prolog assembly code for FunctionWith6Params() is shown below:

0:000> uf .

example!FunctionWith6Params [c:\temp\blog_entry\sample_code\example.cpp @ 28]:

```
   41 00000001`40015900 mov     dword ptr [rsp+20h],r9d
   41 00000001`40015905 mov     dword ptr [rsp+18h],r8d
   41 00000001`4001590a mov     dword ptr [rsp+10h],edx
   41 00000001`4001590e mov     dword ptr [rsp+8],ecx
   41 00000001`40015912 push    rbx
   41 00000001`40015913 push    rsi
   41 00000001`40015914 push    rdi
   41 00000001`40015915 sub     rsp,50h
```

You can see that the first four instructions save the first four parameters on the stack in the home space allocated by main(). Then, the prolog code saves any non-volatile registers that FunctionWith6Params() plans to use during its execution. The saved registers' states are restored in the function epilog code prior to returning to the caller. Finally the prolog code reserves some space on the stack, in this case, for 0x50 bytes.

What is this space reserved on the top of the stack for? First, space is created for any local variables. In this case, FunctionWith6Params() has two. However, those two local variables only account for 0x10 bytes. What's the deal with the rest of the space created on the top of the stack?

On the x64 platform, when code prepares the stack for calling another function, it does not use push instructions to put the parameters on the stack as is commonly the case in x86 code. Instead, the stack pointer typically remains fixed for a particular function. The compiler looks at all of the functions the code in the current function calls, it finds the one with the maximum number of parameters, and then creates enough space on the stack to accommodate those parameters. In this example, FunctionWith6Params() calls printf() passing it 8 parameters. Since that is the called function with the maximum number of parameters, the compiler creates 8 slots on the stack. The top four slots on the stack will then be the home space used by any functions FunctionWith6Params() calls.

A handy side effect of the x64 calling convention is that once you are inside the bracket of the prolog and epilog of a function, the stack pointer does not change while the instruction pointer is in that function. This eliminates the need for a base pointer which is common in x86 calling conventions. When the code in FunctionWith6Params() prepares to call a child function, it simply puts the first four parameters into the required registers and, if there are more than 4 parameters, it uses mov instructions to place the remaining parameters in the allocated stack space but making sure to skip the first four parameter slots on the stack.

Debugging Optimized x64 code (The Nightmare Begins)

Why is debugging x64 optimized code so tricky? Well, remember that home space that the caller creates on the stack for the callee to save the first four parameters? It turns out that the calling convention **does not require** the callee to use that space! And you can certainly bet that optimized x64 code will not use that space unless it is necessary and convenient for its optimization purposes. Moreover, when optimized code does use the home space, it could use it to store non-volatile registers rather than the first four parameters to the function.

Go ahead and recompile the example code using the following command line:

```
cl /EHa /Zi /Ox /favor:INTEL64 example.cpp /link /debug
```

Notice the use of the /Ox switch. This turns on maximum optimization. Debug symbols are still turned on so we can debug the optimized code easily. **Always build your release product with debug information turned on so you can debug your release builds!**

Let's look at how the prolog assembly code for FunctionWith6Params() has changed:

```
41 00000001`400158e0 mov     qword ptr [rsp+8],rbx
41 00000001`400158e5 mov     qword ptr [rsp+10h],rbp
41 00000001`400158ea mov     qword ptr [rsp+18h],rsi
41 00000001`400158ef push    rdi
41 00000001`400158f0 push    r12
41 00000001`400158f2 push    r13
41 00000001`400158f4 sub     rsp,40h
41 00000001`400158f8 mov     ebx,r9d
41 00000001`400158fb mov     edi,r8d
41 00000001`400158fe mov     esi,edx
41 00000001`40015900 mov     r12d,ecx
```

The optimized code is significantly different! Let's itemize the changes below:

- The function uses the home space on the stack, however, it does not store the first four parameters there. Instead it uses the space to store some non-volatile registers it must restore later in the epilog code. This optimized code is going to make use of more processor registers, therefore it must save more of the non-volatile registers.
- It still pushes three non-volatile registers onto the stack for safe keeping along with the other three it stored in the home space.
- It then creates space on the stack. However, it's less space than in the non-optimized code, and is only 0x40 bytes. That's because the optimized code uses registers to represent the local variables someLocalVariable1 and someLocalVariable2. Therefore, it only has to create space for the 8 slots needed to call the function with the maximum number of parameters, printf().
- It then stores the first four parameters into non-volatile registers rather than in the home space. (Don't count on this behavior. An optimized function may make no copies of the contents of rcx, rdx, r8, and r9. It all depends on the structure of the code)

Now step through FunctionWith6Params() to the source line just after the first printf() call. The output generated from the printf() call on my machine is as follows:

```
Entering FunctionWith6Params( 29, 4823, 18BE, 6784, 4AE1, 3D6C )
```

A common version of the stack command in windbg is kb, which also displays the first few parameters to each function in the frame. In reality, it is displaying the first few positions of the stack. The output for the kb command is as follows:

```
0:000> kb
RetAddr           : Args to
Child                                                             : Call Site
00000001`4001593b : 00000000`00004ae1 00000000`00004823 00000000`000018be
00000000`007e3570 : example!FunctionWith6Params+0x6a
[c:\temp\blog_entry\sample_code\example.cpp @ 37]
00000001`40001667 : 00000000`00000000 00000000`00000000 00000000`00000000
00000000`00000001 : example!main+0x5b
[c:\temp\blog_entry\sample_code\example.cpp @ 57]
00000000`76d7495d : 00000000`00000000 00000000`00000000 00000000`00000000
00000000`00000000 : example!__tmainCRTStartup+0x15b
00000000`76f78791 : 00000000`00000000 00000000`00000000 00000000`00000000
00000000`00000000 : kernel32!BaseThreadInitThunk+0xd
00000000`00000000 : 00000000`00000000 00000000`00000000 00000000`00000000
00000000`00000000 : ntdll!RtlUserThreadStart+0x1d
```

Notice that not all of the first four parameters of FunctionWith6Params() match what is shown by the kb command! Of course, this is a side effect of the optimization. **You simply cannot trust the output displayed by kb and kv in optimized code**. This is the biggest reason why optimized x64 code is so difficult to debug. Trust me when I say that it's just pure luck that the second and third slots in the kb output above match the actual parameter values to FunctionWith6Params(). It's because FunctionWith6Params() stores non-volatile registers in those slots and it just so happens that main() put those values in those non-volatile registers prior to calling FunctionWith6Params().

Parameter Sleuthing -- Technique 1 (Down the Call Graph)

Now, let’s look at some techniques for finding elusive function parameters to functions in the call stack while running x64 code. I have placed a DebugBreak() call in FunctionWith4Params() to illustrate. Go ahead and let the code run in windbg until it hits this breakpoint. Now, imagine what you are looking at is actually not a live debugging scenario but rather a dump file from a customer of yours and this is the point where your application has crashed. So, you take a look at the stack and it looks like the following:

```
0:000> kL
Child-SP          RetAddr           Call Site
00000000`0012fdc8 00000001`40015816 ntdll!DbgBreakPoint
00000000`0012fdd0 00000001`400158a0 example!FunctionWith4Params+0x66
00000000`0012fe50 00000001`40015977 example!FunctionWith5Params+0x20
00000000`0012fe80 00000001`40015a0b example!FunctionWith6Params+0x97
00000000`0012fee0 00000001`4000168b example!main+0x5b
00000000`0012ff20 00000000`7733495d example!__tmainCRTStartup+0x15b
00000000`0012ff60 00000000`77538791 kernel32!BaseThreadInitThunk+0xd
00000000`0012ff90 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

Now, let’s say that in order for you to figure out what went wrong, you need to know the first parameter to FunctionWith6Params(). **Assume you have not seen the first parameter in the console output. No fair cheating!**

The first technique I would like to illustrate involves digging downward into the call graph to find out what has happened to the contents of rcx (the first parameter) after entering FunctionWith6Params(). In this case, since the parameters are 32bit integers, we’ll be attempting to follow the contents of ecx, which is the lower half of rcx.

Let’s start by looking at the assembly code within FunctionWith6Params() starting from the beginning up to the call into FunctionWith5Params()::

```
0:000> u example!FunctionWith6Params example!FunctionWith6Params+0x97
example!FunctionWith6Params [c:\temp\blog_entry\sample_code\example.cpp @ 41]:
00000001`400158e0 mov     qword ptr [rsp+8],rbx
00000001`400158e5 mov     qword ptr [rsp+10h],rbp
00000001`400158ea mov     qword ptr [rsp+18h],rsi
00000001`400158ef push    rdi
00000001`400158f0 push    r12
00000001`400158f2 push    r13
00000001`400158f4 sub     rsp,40h
00000001`400158f8 mov     ebx,r9d
00000001`400158fb mov     edi,r8d
00000001`400158fe mov     esi,edx
00000001`40015900 mov     r12d,ecx
00000001`40015903 call    example!rand (00000001`4000148c)
00000001`40015908 movsxd  r13,eax
00000001`4001590b call    example!rand (00000001`4000148c)
00000001`40015910 lea     rdx,[example!`string'+0x68 (00000001`40020d40)]
00000001`40015917 movsxd  rbp,eax
```

```
00000001`4001591a mov     eax,dword ptr [rsp+88h]
00000001`40015921 lea     rcx,[example!`string'+0x80 (00000001`40020d58)]
00000001`40015928 mov     dword ptr [rsp+38h],eax
00000001`4001592c mov     eax,dword ptr [rsp+80h]
00000001`40015933 mov     r9d,esi
00000001`40015936 mov     dword ptr [rsp+30h],eax
00000001`4001593a mov     r8d,r12d
00000001`4001593d mov     dword ptr [rsp+28h],ebx
00000001`40015941 mov     dword ptr [rsp+20h],edi
00000001`40015945 call    example!printf (00000001`400012bc)
00000001`4001594a call    example!rand (00000001`4000148c)
00000001`4001594f mov     edi,eax
00000001`40015951 call    example!rand (00000001`4000148c)
00000001`40015956 mov     esi,eax
00000001`40015958 call    example!rand (00000001`4000148c)
00000001`4001595d mov     ebx,eax
00000001`4001595f call    example!rand (00000001`4000148c)
00000001`40015964 mov     r9d,r12d
00000001`40015967 mov     r8d,esi
00000001`4001596a mov     edx,ebx
00000001`4001596c mov     ecx,eax
00000001`4001596e mov     dword ptr [rsp+20h],edi
00000001`40015972 call    example!ILT+5(?FunctionWith5ParamsYAXHHHHHZ)
(00000001`4000100a)
```

FunctionWith6Params() copies ecx into r12d to preserve it for later use since the contents must be passed to multiple functions within the body of FunctionWith6Params(). Notice at the point where FunctionWith5Params() is called, the contents of ecx have been copied into both r12d and r9d, however, r9d is volatile so we must be careful with it since it could get overwritten prior to the next function call when FunctionWith5Params() calls FunctionWith4Params(). Armed with this information, let's dig into the assembly code for FunctionWith5Params() that has executed up to this point:

```
0:000> u example!FunctionWith5Params example!FunctionWith5Params+0x20
example!FunctionWith5Params [c:\temp\blog_entry\sample_code\example.cpp @ 32]:
00000001`40015880 mov     qword ptr [rsp+8],rbx
00000001`40015885 mov     qword ptr [rsp+10h],rsi
00000001`4001588a push    rdi
00000001`4001588b sub     rsp,20h
00000001`4001588f mov     ecx,dword ptr [rsp+50h]
00000001`40015893 mov     eax,r9d
00000001`40015896 mov     r9d,edx
00000001`40015899 mov     edx,eax
00000001`4001589b call    example!ILT+10(?FunctionWith4ParamsYAXHHHHZ)
(00000001`4000100f)
```

At the point where FunctionWith4Params() is called, the value we are after is now in eax, edx, and r12d. Again, be careful with eax and edx as they are volatile. However, since FunctionWith5Params() did not touch r12d, the contents of the parameter we are still after are still in r12d

Now, let's look at the code in FunctionWith4Params() that has executed so far:

```
0:000> u example!FunctionWith4Params example!FunctionWith4Params+0x66
example!FunctionWith4Params [c:\temp\blog_entry\sample_code\example.cpp @ 9]:
00000001`400157b0 48895c2408      mov     qword ptr [rsp+8],rbx
00000001`400157b5 48896c2410      mov     qword ptr [rsp+10h],rbp
00000001`400157ba 4889742418      mov     qword ptr [rsp+18h],rsi
00000001`400157bf 57              push    rdi
00000001`400157c0 4154            push    r12
00000001`400157c2 4155            push    r13
00000001`400157c4 4156            push    r14
00000001`400157c6 4157            push    r15
00000001`400157c8 4883ec50        sub     rsp,50h
00000001`400157cc 458be1          mov     r12d,r9d
00000001`400157cf 458be8          mov     r13d,r8d
00000001`400157d2 448bf2          mov     r14d,edx
00000001`400157d5 448bf9          mov     r15d,ecx
00000001`400157d8 e8afbcfeff      call    example!rand (00000001`4000148c)
00000001`400157dd 4898            cdqe
00000001`400157df 4889442448      mov     qword ptr [rsp+48h],rax
00000001`400157e4 e8a3bcfeff      call    example!rand (00000001`4000148c)
00000001`400157e9 4898            cdqe
00000001`400157eb 4889442440      mov     qword ptr [rsp+40h],rax
00000001`400157f0 e897bcfeff      call    example!rand (00000001`4000148c)
00000001`400157f5 4863e8          movsxd  rbp,eax
00000001`400157f8 e88fbcfeff      call    example!rand (00000001`4000148c)
00000001`400157fd 4863f0          movsxd  rsi,eax
00000001`40015800 e887bcfeff      call    example!rand (00000001`4000148c)
00000001`40015805 4863f8          movsxd  rdi,eax
00000001`40015808 e87fbcfeff      call    example!rand (00000001`4000148c)
00000001`4001580d 4863d8          movsxd  rbx,eax
00000001`40015810 ff15a24b0100    call    qword ptr [example!_imp_DebugBreak
(00000001`4002a3b8)]
```

We just found what we are looking for! The red highlighted line shows r12 being saved on the stack because FunctionWith4Params() wants to reuse r12. Since r12 is a non-volatile register, it must save the contents somewhere so it can restore the contents before the function exits. All we have to do is locate that slot on the stack, and assuming that the stack has not been corrupted, we'll have our prize.

One technique for finding the slot is to start with the Child-SP value associated with the FunctionWith4Params() frame in the stack dump shown previously, which is 00000000`0012fdd0 in my build. Using that value, let's dump the stack content using the dps command:

```
0:000> dps 00000000`0012fdd0 L10
00000000`0012fdd0  00000001`00000001
00000000`0012fdd8  00000001`40024040 example!_iob+0x30
```

```
00000000`0012fde0  00000000`00000000
00000000`0012fde8  00000001`40002f9e example!_getptd_noexit+0x76
00000000`0012fdf0  00000000`00261310
00000000`0012fdf8  00000001`40001a92 example!_unlock_file2+0x16
00000000`0012fe00  00000000`00000001
00000000`0012fe08  00000000`00004823
00000000`0012fe10  00000000`000041bb
00000000`0012fe18  00000000`00005af1
00000000`0012fe20  00000000`00000000
00000000`0012fe28  00000000`00000000
00000000`0012fe30  00000000`00002cd6
00000000`0012fe38  00000000`00000029
00000000`0012fe40  00000000`00006952
00000000`0012fe48  00000001`400158a0 example!FunctionWith5Params+0x20
[c:\temp\blog_entry\sample_code\example.cpp @ 34]
```

I have highlighted the position that rsp points to when we enter FunctionWith4Params() in red. Based on the prolog code shown for FunctionWith4Params() above, we can find the slot where our prize is stored. I have highlighted it in green above and you can see the value on my machine is 0x29, which matches the value printf() sent to the console. Additionally, I highlighted r14d in green in the assembly code for FunctionWith4Params() to indicate where the contents of edx (the second parameter) were copied to. Since FunctionWith4Params() is virtually the top function on the stack (due to the fact that DebugBreak() takes no parameters), then r14d should also contain the value we are after. Dumping the contents of r14 proves this as shown below:

```
0:000> r r14
r14=0000000000000029
```

To sum up, when you are chasing register-passed parameter values down through a call graph, look for places where the value is copied into. Specifically, if the value is copied into a non-volatile register, that can be a good thing. If a downstream function wants to reuse that non-volatile register, it must first save the contents (usually on the stack) so it can restore it when it is done. If you're not that lucky, you may be able to trace a register it was copied into which has not been changed at the breakpoint. Both conditions were shown above.

Parameter Sleuthing -- Technique 2 (Up the Call Graph)

The second technique I would like to demonstrate is very similar to the first technique except that we walk the stack/call-graph in the opposite direction as before, that is, up the call graph. Unfortunately, none of these techniques are fool proof and guaranteed to bear fruit. So, it's nice to have multiple techniques to employ even though all of them may strike out.

We know that when FunctionWith6Params() gets called, ecx contains the value we are after. Therefore, if we look at the code for main(), maybe we can find the source from which the ecx register was filled prior to the function call. Let's have a look as the assembly code in main():

```
0:000> u example!main example!main+0x5b
example!main [c:\temp\blog_entry\sample_code\example.cpp @ 58]:
00000001`400159b0 48895c2408      mov     qword ptr [rsp+8],rbx
00000001`400159b5 48896c2410      mov     qword ptr [rsp+10h],rbp
```

```
00000001`400159ba 4889742418      mov     qword ptr [rsp+18h],rsi
00000001`400159bf 48897c2420      mov     qword ptr [rsp+20h],rdi
00000001`400159c4 4154            push    r12
00000001`400159c6 4883ec30        sub     rsp,30h
00000001`400159ca e8bdbafeff      call    example!rand (00000001`4000148c)
00000001`400159cf 448be0          mov     r12d,eax
00000001`400159d2 e8b5bafeff      call    example!rand (00000001`4000148c)
00000001`400159d7 8be8            mov     ebp,eax
00000001`400159d9 e8aebafeff      call    example!rand (00000001`4000148c)
00000001`400159de 8bf0            mov     esi,eax
00000001`400159e0 e8a7bafeff      call    example!rand (00000001`4000148c)
00000001`400159e5 8bf8            mov     edi,eax
00000001`400159e7 e8a0bafeff      call    example!rand (00000001`4000148c)
00000001`400159ec 8bd8            mov     ebx,eax
00000001`400159ee e899bafeff      call    example!rand (00000001`4000148c)
00000001`400159f3 448bcf          mov     r9d,edi
00000001`400159f6 89442428        mov     dword ptr [rsp+28h],eax
00000001`400159fa 448bc6          mov     r8d,esi
00000001`400159fd 8bd5            mov     edx,ebp
00000001`400159ff 418bcc          mov     ecx,r12d
00000001`40015a02 895c2420        mov     dword ptr [rsp+20h],ebx
00000001`40015a06 e8fab5feff      call    example!ILT+0(?
FunctionWith6ParamsYAXHHHHHHZ) (00000001`40001005)
```

We see that ecx was copied from the contents of r12d. This is helpful since r12d is a non-volatile register, and if it is reused by a function further down the call stack, it must be preserved and that preservation usually means putting a copy on the stack. It would have been nice if ecx were filled with a value from the stack, at which point we would be virtually done. But in this case, we just need to start our journey back downwards again.

We don't have to look very far. Let's have another look at the prolog code for FunctionWith6Params():

```
example!FunctionWith6Params [c:\temp\blog_entry\sample_code\example.cpp @ 41]:
   41 00000001`400158e0 mov     qword ptr [rsp+8],rbx
   41 00000001`400158e5 mov     qword ptr [rsp+10h],rbp
   41 00000001`400158ea mov     qword ptr [rsp+18h],rsi
   41 00000001`400158ef push    rdi
   41 00000001`400158f0 push    r12
   41 00000001`400158f2 push    r13
   41 00000001`400158f4 sub     rsp,40h
   41 00000001`400158f8 mov     ebx,r9d
   41 00000001`400158fb mov     edi,r8d
   41 00000001`400158fe mov     esi,edx
   41 00000001`40015900 mov     r12d,ecx
```

r12 is reused in FunctionWith6Params(), which means that our prize will be on the stack. Let's start by looking at the Child-SP for this frame which is at 00000000`0012fe80 by using the dps command:

```
0:000> dps 00000000`0012fe80 L10
00000000`0012fe80  00000000`00001649
00000000`0012fe88  00000000`00005f90
00000000`0012fe90  00000000`00000029
00000000`0012fe98  00000000`00004823
00000000`0012fea0  00000000`00006952
00000000`0012fea8  00000001`00006784
00000000`0012feb0  00000000`00004ae1
00000000`0012feb8  00000001`00003d6c
00000000`0012fec0  00000000`00000000
00000000`0012fec8  00000000`00000029
00000000`0012fed0  00000000`00006784
00000000`0012fed8  00000001`4000128b example!main+0x5b
[c:\temp\blog_entry\sample_code\example.cpp @ 72]
```

I have highlighted in red the slot rsp points to when we enter FunctionWith6Params(). At this point, it is a simple matter to walk the assembly code and find the slot where the value is stored. I have highlighted it in green above.

Parameter Sleuthing -- Technique 3 (Inspecting Dead Space)

The final technique I would like to demonstrate involves a little more trickery and involves looking at “dead” or previously used slots on the stack that are not used by the current function call. To demonstrate, let’s say that after the DebugBreak() is hit, we need to know what the contents of param4 that were passed to FunctionWith6Params(). Let’s have another look at the assembly that has executed for FunctionWith6Params() and this time, let’s follow r9d, the fourth parameter:

```
0:000> u example!FunctionWith6Params example!FunctionWith6Params+0x97
example!FunctionWith6Params [c:\temp\blog_entry\sample_code\example.cpp @ 41]:
00000001`400158e0 mov     qword ptr [rsp+8],rbx
00000001`400158e5 mov     qword ptr [rsp+10h],rbp
00000001`400158ea mov     qword ptr [rsp+18h],rsi
00000001`400158ef push    rdi
00000001`400158f0 push    r12
00000001`400158f2 push    r13
00000001`400158f4 sub     rsp,40h
00000001`400158f8 mov     ebx,r9d
00000001`400158fb mov     edi,r8d
00000001`400158fe mov     esi,edx
00000001`40015900 mov     r12d,ecx
00000001`40015903 call    example!rand (00000001`4000148c)
00000001`40015908 movsxd  r13,eax
00000001`4001590b call    example!rand (00000001`4000148c)
00000001`40015910 lea     rdx,[example!`string'+0x68 (00000001`40020d40)]
00000001`40015917 movsxd  rbp,eax
00000001`4001591a mov     eax,dword ptr [rsp+88h]
00000001`40015921 lea     rcx,[example!`string'+0x80 (00000001`40020d58)]
00000001`40015928 mov     dword ptr [rsp+38h],eax
00000001`4001592c mov     eax,dword ptr [rsp+80h]
```

```
00000001`40015933 mov     r9d,esi
00000001`40015936 mov     dword ptr [rsp+30h],eax
00000001`4001593a mov     r8d,r12d
00000001`4001593d mov     dword ptr [rsp+28h],ebx
00000001`40015941 mov     dword ptr [rsp+20h],edi
00000001`40015945 call    example!printf (00000001`400012bc)
00000001`4001594a call    example!rand (00000001`4000148c)
00000001`4001594f mov     edi,eax
00000001`40015951 call    example!rand (00000001`4000148c)
00000001`40015956 mov     esi,eax
00000001`40015958 call    example!rand (00000001`4000148c)
00000001`4001595d mov     ebx,eax
00000001`4001595f call    example!rand (00000001`4000148c)
00000001`40015964 mov     r9d,r12d
00000001`40015967 mov     r8d,esi
00000001`4001596a mov     edx,ebx
00000001`4001596c mov     ecx,eax
00000001`4001596e mov     dword ptr [rsp+20h],edi
00000001`40015972 call    example!ILT+5(?FunctionWith5ParamsYAXHHHHHZ)
(00000001`4000100a)
```

Notice that r9d is first moved into ebx. But also, notice that it copied the contents into a slot on the stack at rsp+0x28. What is this slot? It's the sixth parameter to the following printf() call. Remember that the compiler looks at all of the functions the code calls and finds the function with the maximum number of parameters and then allocates enough space for that function. As the code prepares to call printf(), it is moving the value we are after into the sixth parameter slot in that reserved stack space. But what use is this information?

If you examine FunctionWith6Params(), you see that every function called after printf() takes less than six parameters. Specifically, the call to FunctionWith5Params() only uses five of those slots and just leaves the remaining three with junk in them. This junk is actually our treasure! From examining the code, it's guaranteed that nobody has overwritten the slot represented by rsp+28.

To find this slot, let's again start by getting the Child-SP value for the frame we're talking about as shown below:

```
0:000> kL
Child-SP          RetAddr           Call Site
00000000`0012fdc8 00000001`40015816 ntdll!DbgBreakPoint
00000000`0012fdd0 00000001`400158a0 example!FunctionWith4Params+0x66
00000000`0012fe50 00000001`40015977 example!FunctionWith5Params+0x20
00000000`0012fe80 00000001`40015a0b example!FunctionWith6Params+0x97
00000000`0012fee0 00000001`4000168b example!main+0x5b
00000000`0012ff20 00000000`7733495d example!__tmainCRTStartup+0x15b
00000000`0012ff60 00000000`77538791 kernel32!BaseThreadInitThunk+0xd
00000000`0012ff90 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

We can then take the highlighted value above and use the same offset in the code to find our value:

```
0:000> dd 000000000012fe80+28 L1
```

```
00000000`0012fea8  00006784
```

As expected, the “dead” slot on the stack contains the value we are after. You can compare the value to the output shown on the console to verify.

A Non-volatile Register Shortcut

Now that I have shown you the theory behind finding these elusive values passed around in registers, let me show you a shortcut that will make life a little bit easier. The shortcut relies upon the /r option of the .frame command. When using .frame /r, the debugger has the smarts to track non-volatile registers. But as with any technique, always have multiple tools in your pocket in case you need to use all of them to verify a result.

To demonstrate, let’s consider Technique 2 described previously where we look up the call graph and we want to know what r12 was prior to main() calling FunctionWith6Params(). Go ahead and re-launch the application in windbg and let it run until it hits the DebugBreak(). Now, let’s take a look at the stack including the frame numbers:

```
0:000> knL
 # Child-SP          RetAddr           Call Site
00 00000000`0012fdc8 00000001`40015816 ntdll!DbgBreakPoint
01 00000000`0012fdd0 00000001`400158a0 example!FunctionWith4Params+0x66
02 00000000`0012fe50 00000001`40015977 example!FunctionWith5Params+0x20
03 00000000`0012fe80 00000001`40015a0b example!FunctionWith6Params+0x97
04 00000000`0012fee0 00000001`4000168b example!main+0x5b
05 00000000`0012ff20 00000000`7748495d example!__tmainCRTStartup+0x15b
06 00000000`0012ff60 00000000`775b8791 kernel32!BaseThreadInitThunk+0xd
07 00000000`0012ff90 00000000`00000000 ntdll!RtlUserThreadStart+0x1d
```

Based on our previous analysis of the assembly in main(), we know that the first parameter to FunctionWith6Params() was also stored in the non-volatile register r12 in main() prior to calling FunctionWith6Params(). Now, check out what we get when we use the .frame /r command to set the current frame to 4.

```
0:000> .frame /r 4
04 00000000`0012fee0 00000001`4000168b example!main+0x5b
[c:\temp\blog_entry\sample_code\example.cpp @ 70]
rax=0000000000002ea6 rbx=0000000000004ae1 rcx=0000000000002ea6
rdx=0000000000145460 rsi=00000000000018be rdi=0000000000006784
rip=0000000140015a0b rsp=000000000012fee0 rbp=0000000000004823
 r8=000007fffffdc000  r9=0000000000001649 r10=0000000000000000
r11=0000000000000246 r12=0000000000000029 r13=0000000000000000
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl nz na pe nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202
example!main+0x5b:
00000001`40015a0b 488b5c2440      mov     rbx,qword ptr [rsp+40h]
ss:00000000`0012ff20=0000000000000000
```

As you can see, .frame /r shows the register contents as they were in main() prior to calling

FunctionWith6Params(). **Beware! You can only trust the non-volatile registers when using this command!** Be sure to check out the following link to see which registers are considered volatile: **Register Usage for x64 64-Bit**.

.frame /r can spare you the time spent manually digging around on the stack to find saved volatile registers. In my experiments, .frame /r even works where there is no symbol information available. However, it never hurts to know how to do it manually in case you're faced with a situation where .frame /r breaks down.

Conclusion

The x64 calling convention and the abundance of general purpose registers in the processor bring many opportunities for optimization to the table. However, when all of those optimizations are in play, they can certainly make debugging difficult. After giving a brief overview of the x64 calling convention, I demonstrated three techniques one can use to find parameter values to various functions in the call stack. I also showed you a shortcut you can use to see non-volatile registers for a particular frame in the call stack. I hope you find these techniques useful in your debugging adventures. Additionally, I urge you to become more familiar with all of the nuances of **the x64 calling convention**.