X64 Deep Dive

codemachine.com/articles/x64_deep_dive.html

This tutorial discusses some of the key aspects of code execution on the X64 CPU like compiler optimizations, exception handling, parameter passing and parameter retrieval and shows how these topics are closely related to each other. It covers the important debugger commands related to the above topics and provides the necessary background required to interpret and understand the output of these commands. It also highlights how the X64 CPU does things differently from the X86 CPU and how it affects debugging on X64. And finally it ties everything together and illustrates how this knowledge can be applied to retrieve register based parameters from X64 call stacks, something that always poses a challenge when debugging X64 code. This tutorial takes a step by steps approach to present the content and makes use of diagrams, disassembly listings and debugger output extensive to drive home the key points. Readers are expected to have a good understand of how things work on the X86 CPU in terms of register usage, stack usage and function layout to make most of this tutorial.

Compiler Optimizations

This section discusses some of the compiler optimization that affects the way X64 code is generated. It starts with a description of the X64 registers and then focusses on optimizations like function in-lining, tail call elimination, frame pointer optimization and stack pointer based local variable access.

Register Changes

All registers on the X64 CPU, with the exception of the segment registers and the EFlags register, are 64-bits which implies that all fetches from memory are 64-bit wide. Also X64 instructions are capable of processing 64-bits at a time which makes x64 a native 64 bit processor. Eight new registers have been added i.e. r8 - r15 which are labeled with numbers as opposed to the other registers that are labeled with alphabets. The following debugger output shows the registers on X64.

```
1: kd> r
rax=fffffa60005f1b70 rbx=fffffa60017161b0 rcx=000000000000007f
rdx=0000000000000008 rsi=fffffa60017161d0 rdi=0000000000000000
rip=fffff80001ab7350 rsp=fffffa60005f1a68 rbp=fffffa60005f1c30
 r8=0000000080050033  r9=00000000000006f8 r10=fffff80001b1876c
r11=0000000000000000 r12=000000000000007b r13=0000000000000002
r14=0000000000000006 r15=0000000000000004
iopl=0         nv up ei ng nz na pe nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000282
nt!KeBugCheckEx:
fffff800`01ab7350 48894c2408      mov     qword ptr [rsp+8],rcx
ss:0018:fffffa60`005f1a70=000000000000007f
```

The usage of some of these register have changed, from X86, as well. The changes can be grouped as follows:

- Non-volatile Registers are registers that are saved across function calls. X64 has an expanded non-volatile register set in which all the old X86 non-volatile registers are also included. New ones in this set are R12 through R15. These are important from the perspective of retrieving register based function parameters.
- Fastcall registers are used to pass parameters to functions. Fastcall is the default calling convention on X64 where in the first 4 parameters are passed via the registers RCX, RDX, R8, R9.
- RBP is no longer used as frame pointer. It is now a general purpose register like any of the other registers like RBX, RCX etc. The debugger can no longer use the RBP register to walk the call stack.
- On the X86 CPU, the FS segment register points to Thread Environment Block (TEB) and the Processor Control Region (KPCR) but on the X64, it is the GS register that points to the TEB while in user mode and the KPCR while in kernel mode. However when running WOW64 applications (i.e. 32 bit applications on X64 systems), the FS register continues to point to the 32-bit version of TEB.

The trap frame data structure (nt!_KTRAP_FRAME) on the X64 does not contain valid contents of non-volatile registers. The prolog of X64 functions save the values of non-volatile registers if they intend to overwrite them. The debugger can always pull the saved values of these non-volatile registers from stack instead of having to retrieve them from the trap frame. During kernel mode debugging on X64, the output of the ".trap" command prints a note highlighting the fact that the values of all the registers retrieved from the trap may not be accurate, as shown below. There are exceptions to this rule e.g., trap frames generated for user to kernel mode transitions do contain the correct values of all the registers.

```
1: kd> kv
Child-SP          RetAddr           : Args to Child
.
.
.
nt!KiDoubleFaultAbort+0xb8 (TrapFrame @ fffffa60`005f1bb0)
.
.
.

1: kd> .trap  fffffa60`005f1bb0
NOTE: The trap frame does not contain all registers.
Some register values may be zeroed or incorrect
```

Function in-lining

The X64 compiler performs inline expansion of functions by which if certain criteria is met, it replaces the call to a function with the body of the callee. Although in-lining is not unique to X64, the X64 compiler is over zealous about in-lining functions. The advantages of inlining are that it avoids the overhead of setting up the stack, branching to the callee and then returning back to the caller. The downside of in-lining is that due to the code duplication, the size of the executable file bloats up and the functions expand resulting in cache miss and increased number of page faults. Function in-lining also impedes debugging in that when one tries to set a breakpoint on a function that the compiler has chosen to inline, the debugger is unable to find the symbol of the in-lined function. In-lining at a source file level is controlled by compiler's /Ob flag and in-lining can be disabled on a per function basis by __declspec(noinline). Figure 1 shows function2 and Function3 being inlined inside Function1.

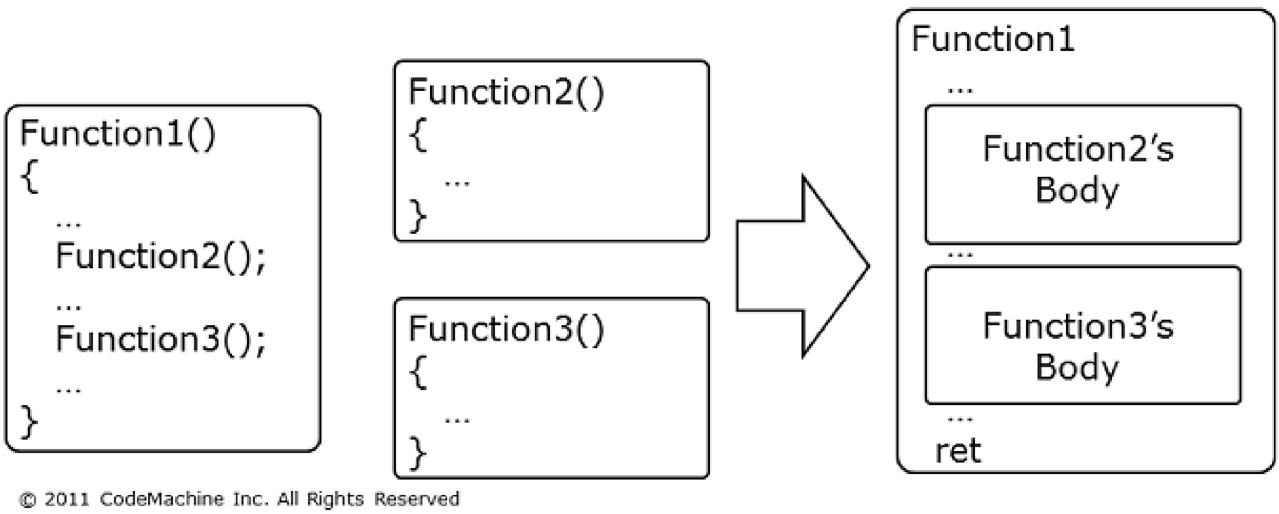

Figure 1 : Function Inlining

Tail Call Elimination

X64 compiler can optimize the last call made from a function by replacing it with a jump to the callee. This avoids the overhead of setting up the stack frame for the callee. The caller and the callee share the same stack frame and the callee returns directly to the caller's caller. This is especially beneficial when the caller and the callee have the same parameters, since, if the relevant parameters are already in the required registers and those registers haven't changed, they don't have to be reloaded. Figure 2 shows tail call elimination in Function1 when calling Function4. Function1 jumps to Function4 and when Function4 finishes execution, it returns directly to the caller of Function1.

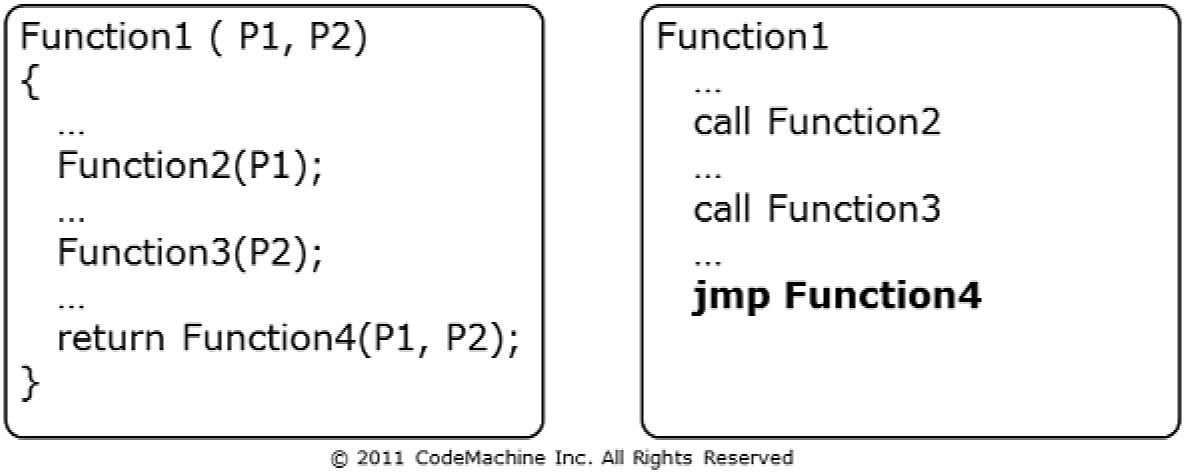

Figure 2 : Tail Call Elimination

Frame Pointer Omission

Unlike the X86 CPU where the EBP register is used to access parameters and local variables on the stack, X64 functions do not make use of the RBP register for this purpose i.e. do not use the EBP register as a frame pointer. Instead, it uses the RSP register both as a stack pointer and a frame pointer, more on how this works in the next topic. So, on X64 the RBP register is now freed up from its stack duties and can be used as a general purpose register. An exception to this rule are functions that use alloca() to dynamically allocate space on the stack. Such functions will use the RBP register as a frame pointer, as they did with EBP on the X86.

The following assembler code snippet shows the X86 function KERNELBASE!Sleep. References to the EBP register show that it is being used as the frame pointer. While calling the function SleepEx(), the parameters are being pushed on to the stack and SleepEx() is called through a call instruction.

```
0:009> uf KERNELBASE!Sleep
KERNELBASE!Sleep:
75ed3511 8bff            mov     edi,edi
75ed3513 55              push    ebp
75ed3514 8bec            mov     ebp,esp
75ed3516 6a00            push    0
75ed3518 ff7508          push    dword ptr [ebp+8]
75ed351b e8cbf6ffff      call    KERNELBASE!SleepEx (75ed2beb)
75ed3520 5d              pop     ebp
75ed3521 c20400          ret     4.
```

The next code snippet shows the same function i.e. kernelbase!Sleep() on X64. There are some striking differences - the X64 version is much more compact due to the fact that there is no saving/restoring/setup of the RBP register i.e. the usage of the frame pointer is omitted and neither is there any setup for the stack frame for the callee i.e. SleepEx(). In fact Sleep() and SleepEx() end up using the same stack frame, an example of tail call optimization in action.

```
0:000> uf KERNELBASE!Sleep
KERNELBASE!Sleep:
000007fe`fdd21140 xor     edx,edx
000007fe`fdd21142 jmp     KERNELBASE!SleepEx (000007fe`fdd21150)
```

Stack Pointer based local variable access

On the X86 CPU, the most important function of the frame pointer (EBP) register is to provide access to stack based parameters and local variables. As discussed earlier, on the X64 CPU, the RBP register does not point to the stack frame of the current function. So on X64, it is the RSP register that has to serve both as a stack pointer as well as a frame pointer. So all stack references on X64 are performed based on RSP. Due to this, functions on X64 depend on the RSP register being static throughout the function body, serving as a frame of reference for accessing locals and parameters. Since push and pop instructions alter the stack pointer, X64 functions restrict push and pop instructions to the function prolog and epilog respectively. The fact that the stack pointer does not change at all between the prolog and the epilog is a characteristic feature of X64 functions, as shown in Figure 3.

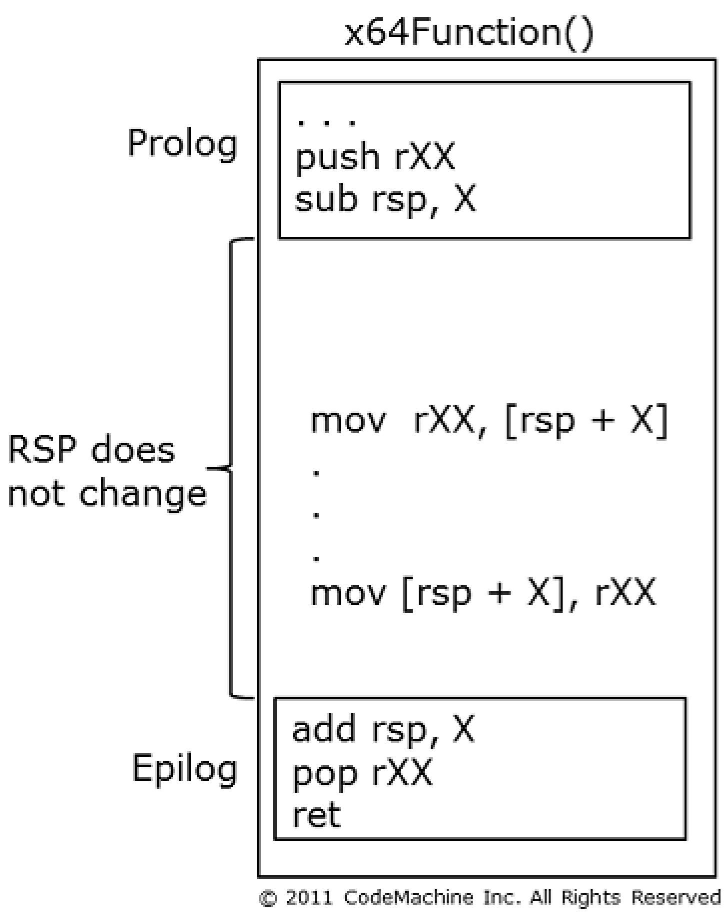

Figure 3 : Static Stack Pointer

The following code snippet shows the complete listing of the function user32!DrawTestExW. This function's prolog ends with the instruction "sub rsp, 48h" and it's epilog starts with the instruction "add rsp, 48h". Since instructions between prolog and epilog access stack contents using the RSP as a reference, there are no intervening push or pop instructions in the function body.

```
0:000> uf user32!DrawTextExW
user32!DrawTextExW:
00000000`779c9c64 sub     rsp,48h
00000000`779c9c68 mov     rax,qword ptr [rsp+78h]
00000000`779c9c6d or      dword ptr [rsp+30h],0FFFFFFFFh
00000000`779c9c72 mov     qword ptr [rsp+28h],rax
00000000`779c9c77 mov     eax,dword ptr [rsp+70h]
00000000`779c9c7b mov     dword ptr [rsp+20h],eax
00000000`779c9c7f call    user32!DrawTextExWorker (00000000`779ca944)
00000000`779c9c84 add     rsp,48h
00000000`779c9c88 ret
```

Exception Handling

This section discusses the underlying mechanism and data structures that X64 functions use for exception handling and also how the debugger leverages these structures to walk the call stack. It also points to some of the unique aspects of X64 call stacks.

RUNTIME_FUNCTION

X64 executable files use a file format that is a variant of the PE file format, used for X86, called PE32+. Such files have an extra section called ".pdata" or Exception Directory that contains information used for handling exceptions. This "Exception Directory" contains a RUNTIME_FUNCTION structure for every non-leaf function in the executable. Non-leaf functions are those that call other functions. Each RUNTIME_FUNCTION structure contains the offset of the first and the last instruction in the function (i.e. the function extents) and a pointer to the unwind information structure that describes how the function's call stack is to be unwound in the event of an exception. Figure 4 shows RUNTIME_FUNCTION structure for a module containing offsets to the beginning and the end of the functions in that module.

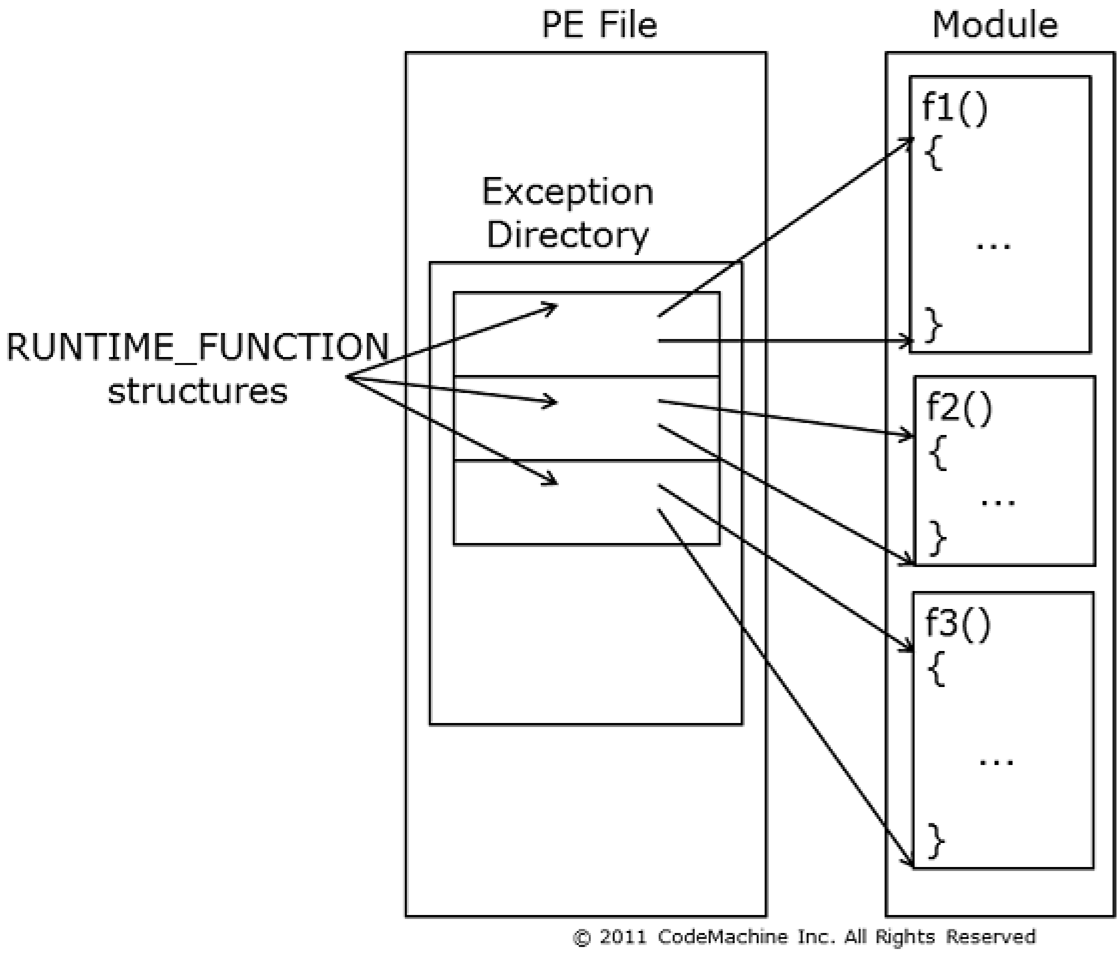

Figure 4 : RUNTIME_FUNCTION

The following assembler code snippets show some of the differences in code generation related to exception handling on the X86 and X64. On x86, when the high level language (C/C++) code contains structured exception handling constructs like __try/__except, the compiler generates special code in the prolog and epilog of the function that builds the exception frame on the stack at runtime. This can be observed in the code snippet below in the calls to ntdll!_SEH_prolog4 and ntdll!_SEH_epilog4.

```
0:009> uf ntdll!__RtlUserThreadStart
ntdll!__RtlUserThreadStart:
77009d4b push    14h
77009d4d push    offset ntdll! ?? ::FNODOBFM::`string'+0xb5e (76ffc3d0)
77009d52 call    ntdll!_SEH_prolog4 (76ffdd64)
77009d57 and     dword ptr [ebp-4],0
77009d5b mov     eax,dword ptr [ntdll!Kernel32ThreadInitThunkFunction (770d4224)]
77009d60 push    dword ptr [ebp+0Ch]
77009d63 test    eax,eax
77009d65 je      ntdll!__RtlUserThreadStart+0x25 (77057075)

ntdll!__RtlUserThreadStart+0x1c:
77009d6b mov     edx,dword ptr [ebp+8]
77009d6e xor     ecx,ecx
77009d70 call    eax
77009d72 mov     dword ptr [ebp-4],0FFFFFFFEh
77009d79 call    ntdll!_SEH_epilog4 (76ffdda9)
77009d7e ret     8
```

In the x64 version of the function, however, there is no indication that the function uses structured exception handling, since no stack based exception frames are built at runtime. The RUNTIME_FUNCTION structures along with the current value of the instruction pointer register (RIP) are used to locate the exception handling information from the executable file itself.

```
0:000> uf ntdll!RtlUserThreadStart
Flow analysis was incomplete, some code may be missing
ntdll!RtlUserThreadStart:
00000000`77c03260 sub     rsp,48h
00000000`77c03264 mov     r9,rcx
00000000`77c03267 mov     rax,qword ptr [ntdll!Kernel32ThreadInitThunkFunction
(00000000`77d08e20)]
00000000`77c0326e test    rax,rax
00000000`77c03271 je      ntdll!RtlUserThreadStart+0x1f (00000000`77c339c5)

ntdll!RtlUserThreadStart+0x13:
00000000`77c03277 mov     r8,rdx
00000000`77c0327a mov     rdx,rcx
00000000`77c0327d xor     ecx,ecx
00000000`77c0327f call    rax
00000000`77c03281 jmp     ntdll!RtlUserThreadStart+0x39 (00000000`77c03283)

ntdll!RtlUserThreadStart+0x39:
00000000`77c03283 add     rsp,48h
00000000`77c03287 ret

ntdll!RtlUserThreadStart+0x1f:
00000000`77c339c5 mov     rcx,rdx
00000000`77c339c8 call    r9
00000000`77c339cb mov     ecx,eax
00000000`77c339cd call    ntdll!RtlExitUserThread (00000000`77bf7130)
00000000`77c339d2 nop
00000000`77c339d3 jmp     ntdll!RtlUserThreadStart+0x2c (00000000`77c53923)
```

UNWIND_INFO and UNWIND_CODE

The BeginAddress and EndAddress fields of the RUNTIME_FUNCTION structure contain the offset of the start and end of the function's code in the virtual memory respectively, from the start of the module. When the function generates an exception, the OS scans the memory mapped copy of the PE file looking for a RUNTIME_FUNCTION structure whose extents include the current instruction address. The UnwindData field of the RUNTIME_FUNCTION structure contains the offset of another structure that tells the OS runtime as to how it should go about unwinding the stack, this is the UNWIND_INFO structure. The UNWIND_INFO structure contains a variable number of UNWIND_CODE structures, each one of which reverses the effect of a single stack related operation performed by the function's prolog.

For dynamically generated code, the OS support functions RtlAddFunctionTable() and RtlInstallFunctionTableCallback() are used to create the RUNTIME_FUNCTION information at runtime.

Figure 5 shows the relationship between the RUNTIME_FUNCTION and the UNWIND_INFO structures and the location of the function in memory.

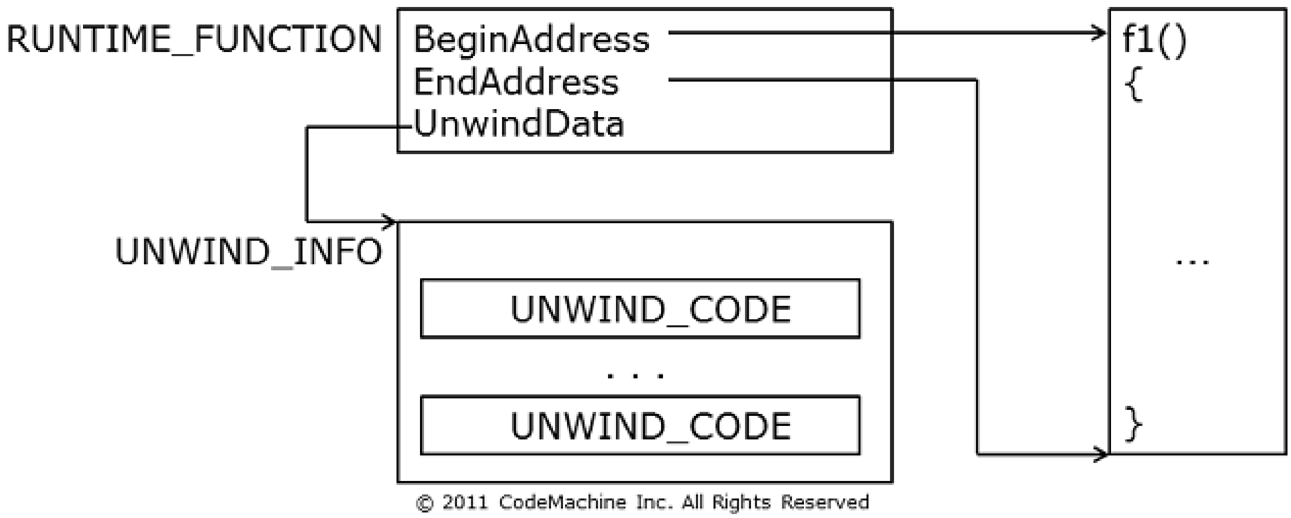

Figure 5 : Unwind Information

The debugger's ".fnent" command displays information about the RUNTIME_FUNCTION structure for a given function. The following example shows the output of the ".fnent" command for the function ntdll!RtlUserThreadStart.

```
0:000> .fnent ntdll!RtlUserThreadStart
Debugger function entry 00000000`03be6580 for:
(00000000`77c03260)   ntdll!RtlUserThreadStart   |  (00000000`77c03290)
ntdll!RtlRunOnceExecuteOnce
Exact matches:
    ntdll!RtlUserThreadStart = <no type information>

BeginAddress      = 00000000`00033260
EndAddress        = 00000000`00033290
UnwindInfoAddress = 00000000`00128654

Unwind info at 00000000`77cf8654, 10 bytes
  version 1, flags 1, prolog 4, codes 1
  frame reg 0, frame offs 0
  handler routine: ntdll!_C_specific_handler (00000000`77be50ac), data 3
  00: offs 4, unwind op 2, op info 8    UWOP_ALLOC_SMALL
```

If BeginAddress shown above is added to the base of the module i.e. ntdll.dll which contains the function RtlUserThreadStart, the resultant address 0x0000000077c03260 is the start of the function RtlUserThreadStart as shown below.

```
0:000> ?ntdll+00000000`00033260
Evaluate expression: 2009084512 = 00000000`77c03260

0:000> u ntdll+00000000`00033260
ntdll!RtlUserThreadStart:
00000000`77c03260 sub     rsp,48h
00000000`77c03264 mov     r9,rcx
00000000`77c03267 mov     rax,qword ptr [ntdll!Kernel32ThreadInitThunkFunction
(00000000`77d08e20)]
00000000`77c0326e test    rax,rax
00000000`77c03271 je      ntdll!RtlUserThreadStart+0x1f (00000000`77c339c5)
00000000`77c03277 mov     r8,rdx
00000000`77c0327a mov     rdx,rcx
00000000`77c0327d xor     ecx,ecx
```

If EndAddress is used the same way, the resultant address points just past the end of the function as shown in the example below.

```
0:000> ?ntdll+00000000`00033290
Evaluate expression: 2009084560 = 00000000`77c03290

0:000> ub 00000000`77c03290 L10
ntdll!RtlUserThreadStart+0x11:
00000000`77c03271 je      ntdll!RtlUserThreadStart+0x1f (00000000`77c339c5)
00000000`77c03277 mov     r8,rdx
00000000`77c0327a mov     rdx,rcx
00000000`77c0327d xor     ecx,ecx
00000000`77c0327f call    rax
00000000`77c03281 jmp     ntdll!RtlUserThreadStart+0x39 (00000000`77c03283)
00000000`77c03283 add     rsp,48h
00000000`77c03287 ret
00000000`77c03288 nop
00000000`77c03289 nop
00000000`77c0328a nop
00000000`77c0328b nop
00000000`77c0328c nop
00000000`77c0328d nop
00000000`77c0328e nop
00000000`77c0328f nop
```

So the BeginAddress and EndAddress fields of the RUNTIME_FUNCTION structure describe where the corresponding function resides in memory. There is, however, an optimization, that may be applied to the module after it has been linked, that can potentially alter the above observations; more on this later.

Although the main purpose of the UNWIND_INFO and UNWIND_CODE structures is to describe how the stack is unwound during an exception, the debugger uses this information to walk the call stack without having access to the symbols for the module. Each UNWIND_CODE structure can describe one of the following operations performed by a function's prolog:

- SAVE_NONVOL - Save a non-volatile register on the stack.
- PUSH_NONVOL - Push a non-volatile register on the stack.
- ALLOC_SMALL - Allocate space (up to 128 bytes) on the stack.
- ALLOC_LARGE - Allocate space (up to 4GB) on the stack.

So, in essence, the UNWIND_CODEs are a meta-data representation of the functions prolog.

Figure 6 shows the relationship between stack related operations performed by the function prolog and the description of these operations in the UNWIND_CODE structures. The UNWIND_CODE structures appear in the reverse order of the instructions they represent, such that during an exception, the stack can be unwound in the opposite direction in which it was created.

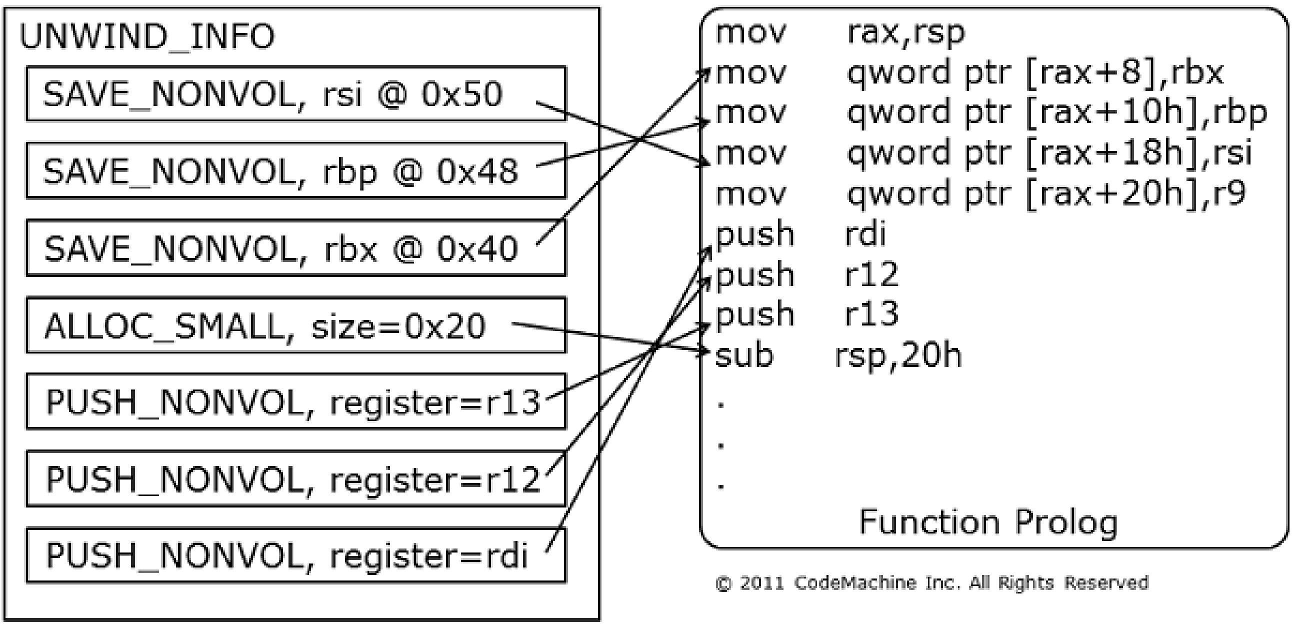

Figure 6 : Unwind Code

The following example displays the ".pdata" section header from the PE file for the native version of notepad.exe on an X64 system. The "virtual address" field indicates that the .pdata section is located at an offset of 0x13000 from the beginning of the executable file.

```
T:\link -dump -headers c:\windows\system32\notepad.exe
.
.
.
SECTION HEADER #4
  .pdata name
     6B4 virtual size
   13000 virtual address (0000000100013000 to 00000001000136B3)
     800 size of raw data
    F800 file pointer to raw data (0000F800 to 0000FFFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read Only
.
.
.
```

The next example shows the UNWIND_INFO and the UNWIND_CODE structures from the same executable file i.e. notepad.exe. Each UNWIND_CODE structure describes an operation like PUSH_NONVOL or ALLOC_SMALL that the function's prolog performs and must be undone when the stack is unwound, as shown below. The debugger's ".fnent" command also shows the contents of these two structures. However, the output of "link -dump -unwindinfo" decodes the entire contents of the UNWIND_CODE structures which ".fnent" does not.

```
T:\link -dump -unwindinfo c:\windows\system32\notepad.exe
.
.
.
  00000018 00001234 0000129F 0000EF68
    Unwind version: 1
    Unwind flags: None
    Size of prologue: 0x12
    Count of codes: 5
    Unwind codes:
      12: ALLOC_SMALL, size=0x28
      0E: PUSH_NONVOL, register=rdi
      0D: PUSH_NONVOL, register=rsi
      0C: PUSH_NONVOL, register=rbp
      0B: PUSH_NONVOL, register=rbx.
.
.
.
```

The ALLOC_SMALL in the above output represents the "sub" instruction in the function's prolog that allocates 0x28 bytes of stack space. Each PUSH_NONVOL corresponds to a "push" instruction in the function's prolog which saves a non-volatile register on the stack and is restored by the "pop" instruction in the function's epilog. These instructions can be seen in the disassembly of the function at offset 0x1234 shown below:

```
0:000> ln notepad+1234
(00000000`ff971234)   notepad!StringCchPrintfW   |  (00000000`ff971364)
notepad!CheckSave
Exact matches:
    notepad!StringCchPrintfW = <no type information>
    notepad!StringCchPrintfW = <no type information>

0:000> uf notepad!StringCchPrintfW
notepad!StringCchPrintfW:
00000001`00001234 mov     qword ptr [rsp+18h],r8
00000001`00001239 mov     qword ptr [rsp+20h],r9
00000001`0000123e push    rbx
00000001`0000123f push    rbp
00000001`00001240 push    rsi
00000001`00001241 push    rdi
00000001`00001242 sub     rsp,28h
00000001`00001246 xor     ebp,ebp
00000001`00001248 mov     rsi,rcx
00000001`0000124b mov     ebx,ebp
00000001`0000124d cmp     rdx,rbp
00000001`00001250 je      notepad!StringCchPrintfW+0x27 (00000001`000077b5)
...
notepad!StringCchPrintfW+0x5c:
00000001`00001294 mov     eax,ebx
00000001`00001296 add     rsp,28h
00000001`0000129a pop     rdi
00000001`0000129b pop     rsi
00000001`0000129c pop     rbp
00000001`0000129d pop     rbx
00000001`0000129e ret
```

Performance Optimization

Windows operating system binaries are subject to a profile guided optimization called Basic Block Tools (BBT), which increases the spatial locality of code. Parts of a function that are executed frequently are kept together, potentially in the same page, and infrequently used parts are moved to other locations. This reduces the number of pages that are required to be kept in memory for the most commonly executed code paths, ultimately resulting in overall working set reduction. In order to apply this optimization, the binary is linked, executed, profiled and then the profile data is used to rearrange parts of a function based on execution frequency.

In the resultant function, some of the function's code blocks are moved outside the function's main body which was originally defined by the extents of the RUNTIME_FUNCTION structure. Due to the code block movement the function body gets broken up into multiple discontiguous parts and hence the RUNTIME_FUNCTION structure, that was originally generated by the linker, is no longer able to accurately identify the extents of such functions. In order to address this problem, the BBT process adds multiple new RUNTIME_FUNCTION structures each defining one contiguous code block with the optimized function. These RUNTIME_FUNCTION structures are chained together with the chain terminating at the original RUNTIME_FUNCTION structure whose BeginAddress always points to the start of the function.

Figure 7 shows a function made from three basic blocks. After applying the BBT process block #2 gets moved outside the function body causing the information in the original RUNTIME_FUNCTION to become invalid. So the BBT process creates a second RUNTIME_FUNCTION structure and chains it to the first one, thus describing the entire function.

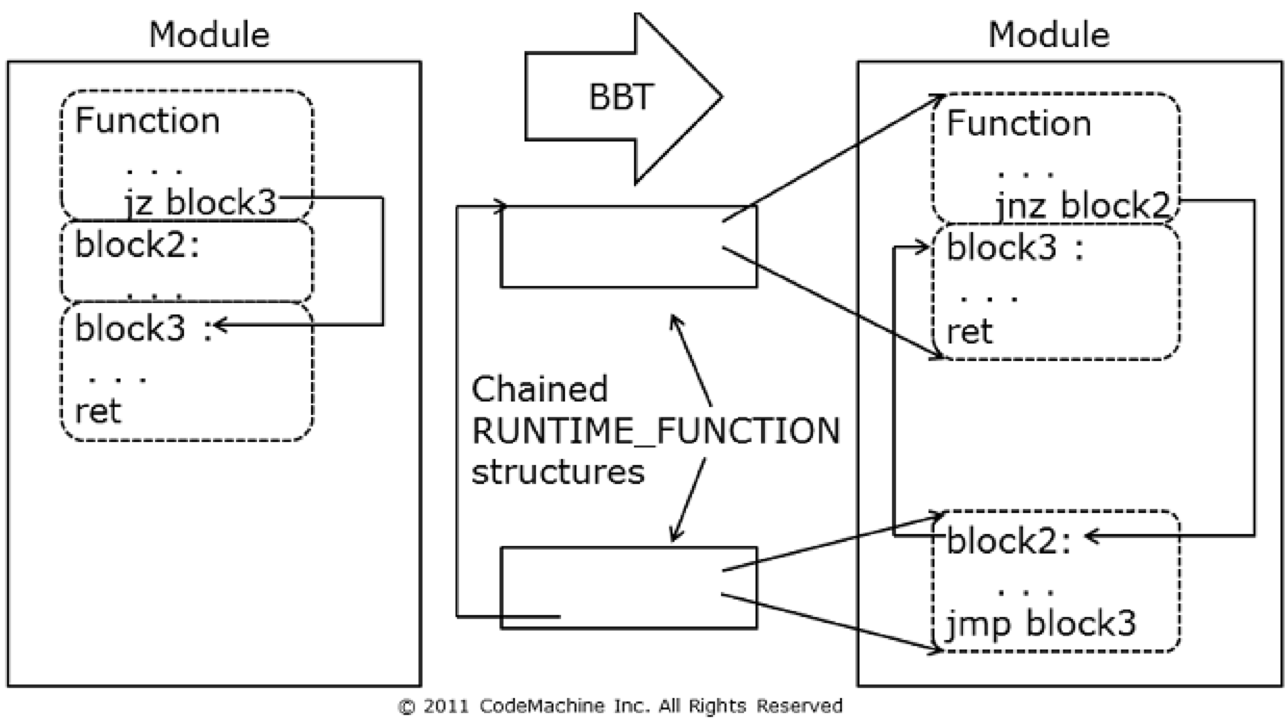

Figure 7 : Performance Optimization : Basic Block Tools

The current public version of the debugger does not walk the complete chain of RUNTIME_FUNCTION structures. So the debugger is unable to show correct names of optimized functions in which the return address maps to a code block that has been moved outside the main function body.

The following example shows functions in the call stack whose names are not displayed correctly. Instead the names are displayed in the form of "ntdll! ?? ::FNODOBFM::`string'. The debugger incorrectly translates the return address 0x0000000077c17623 in frame 0x0c to the name "ntdll! ?? ::FNODOBFM::`string'+0x2bea0".

```
0:000> kn
 # Child-SP          RetAddr           Call Site
00 00000000`0029e4b8 000007fe`fdd21726 ntdll! ?? ::FNODOBFM::`string'+0x6474
01 00000000`0029e4c0 000007fe`fdd2dab6 KERNELBASE!BaseSetLastNTError+0x16
02 00000000`0029e4f0 00000000`77ad108f KERNELBASE!AccessCheck+0x64
03 00000000`0029e550 00000000`77ad0d46 kernel32!BasepIsServiceSidBlocked+0x24f
04 00000000`0029e670 00000000`779cd161 kernel32!LoadAppInitDlls+0x36
05 00000000`0029e6e0 00000000`779cd42d user32!ClientThreadSetup+0x22e
06 00000000`0029e950 00000000`77c1fdf5 user32!_ClientThreadSetup+0x9
07 00000000`0029e980 000007fe`ffe7527a ntdll!KiUserCallbackDispatcherContinue
08 00000000`0029e9d8 000007fe`ffe75139 gdi32!ZwGdiInit+0xa
09 00000000`0029e9e0 00000000`779ccd1f gdi32!GdiDllInitialize+0x11b
0a 00000000`0029eb40 00000000`77c0c3b8 user32!UserClientDllInitialize+0x465
0b 00000000`0029f270 00000000`77c18368 ntdll!LdrpRunInitializeRoutines+0x1fe
0c 00000000`0029f440 00000000`77c17623 ntdll!LdrpInitializeProcess+0x1c9b
0d 00000000`0029f940 00000000`77c0308e ntdll! ?? ::FNODOBFM::`string'+0x2bea0
0e 00000000`0029f9b0 00000000`00000000 ntdll!LdrInitializeThunk+0xe
```

The next example uses the return address 0x0000000077c17623, from above, to display the RUNTIME_FUNCTION, UNWIND_INFO and UNWIND_CODEs for the function with the incorrect name. The displayed information contains a section titled "Chained Info:", which indicates that some of this function's code blocks are outside the function's main body.

```
0:000> .fnent 00000000`77c17623
Debugger function entry 00000000`03b35da0 for:
(00000000`77c55420)   ntdll! ?? ::FNODOBFM::`string'+0x2bea0   |  (00000000`77c55440)
ntdll! ?? ::FNODOBFM::`string'

BeginAddress      = 00000000`000475d3
EndAddress        = 00000000`00047650
UnwindInfoAddress = 00000000`0012eac0

Unwind info at 00000000`77cfeac0, 10 bytes
  version 1, flags 4, prolog 0, codes 0
  frame reg 0, frame offs 0

Chained info:
BeginAddress      = 00000000`000330f0
EndAddress        = 00000000`000331c0
UnwindInfoAddress = 00000000`0011d08c

Unwind info at 00000000`77ced08c, 20 bytes
  version 1, flags 1, prolog 17, codes a
  frame reg 0, frame offs 0
  handler routine: 00000000`79a2e560, data 0
  00: offs f0, unwind op 0, op info 3   UWOP_PUSH_NONVOL
  01: offs 3, unwind op 0, op info 0    UWOP_PUSH_NONVOL
  02: offs c0, unwind op 1, op info 3   UWOP_ALLOC_LARGE FrameOffset: d08c0003
  04: offs 8c, unwind op 0, op info d   UWOP_PUSH_NONVOL
  05: offs 11, unwind op 0, op info 0   UWOP_PUSH_NONVOL
  06: offs 28, unwind op 0, op info 0   UWOP_PUSH_NONVOL
  07: offs 0, unwind op 0, op info 0    UWOP_PUSH_NONVOL
  08: offs 0, unwind op 0, op info 0    UWOP_PUSH_NONVOL
  09: offs 0, unwind op 0, op info 0    UWOP_PUSH_NONVOL
```

The BeginAddress displayed after the "Chained Info" above points to the beginning of the original function. The output of the "ln" command below shows that the scrambled function name is actually ntdll!LdrpInitialize.

```
0:000> ln ntdll+000330f0
(00000000`77c030f0)   ntdll!LdrpInitialize   |  (00000000`77c031c0)
ntdll!LdrpAllocateTls
Exact matches:
    ntdll!LdrpInitialize = <no type information>
```

The debugger's "uf" command displays the assembler code of the entire function, given any address within the function. It does so by visiting all the different code blocks in the function by following the jmp/jCC instructions in each code block. The following output shows the complete assembler listing for the function ntdll!LdrpInitialize. The main body of the function starts at address 00000000`77c030f0 and ends at address 00000000`77c031b3. There is, however, a code block that belongs to the function at address 00000000`77bfd1a4.

This code movement is a result of the BBT process. The debugger attempts to map this address to the nearest symbol and comes up with the incorrect symbol "ntdll! ?? ::FNODOBFM::`string'+0x2c01c", seen in the stack trace earlier.

```
0:000> uf 00000000`77c030f0
ntdll! ?? ::FNODOBFM::`string'+0x2c01c:
00000000`77bfd1a4 48c7842488000000206cfbff mov qword ptr [rsp+88h],0FFFFFFFFFFFB6C20h
00000000`77bfd1b0 443935655e1000  cmp     dword ptr [ntdll!LdrpProcessInitialized
(00000000`77d0301c)],r14d
00000000`77bfd1b7 0f856c5f0000    jne     ntdll!LdrpInitialize+0x39
(00000000`77c03129)
.
.
.
ntdll!LdrpInitialize:
00000000`77c030f0 48895c2408      mov     qword ptr [rsp+8],rbx
00000000`77c030f5 4889742410      mov     qword ptr [rsp+10h],rsi
00000000`77c030fa 57              push    rdi
00000000`77c030fb 4154            push    r12
00000000`77c030fd 4155            push    r13
00000000`77c030ff 4156            push    r14
00000000`77c03101 4157            push    r15
00000000`77c03103 4883ec40        sub     rsp,40h
00000000`77c03107 4c8bea          mov     r13,rdx
00000000`77c0310a 4c8be1          mov     r12,rcx
.
.
.
ntdll!LdrpInitialize+0xac:
00000000`77c0319c 488b5c2470      mov     rbx,qword ptr [rsp+70h]
00000000`77c031a1 488b742478      mov     rsi,qword ptr [rsp+78h]
00000000`77c031a6 4883c440        add     rsp,40h
00000000`77c031aa 415f            pop     r15
00000000`77c031ac 415e            pop     r14
00000000`77c031ae 415d            pop     r13
00000000`77c031b0 415c            pop     r12
00000000`77c031b2 5f              pop     rdi
00000000`77c031b3 c3              ret
```

Modules which have been subjected to BBT optimization can be identified by the word "perf" in the "Characteristics" field in the output of the debuggers "!lmi" command, as shown below.

```
0:000> !lmi notepad
Loaded Module Info: [notepad]
         Module: notepad
   Base Address: 00000000ff4f0000
     Image Name: notepad.exe
   Machine Type: 34404 (X64)
     Time Stamp: 4a5bc9b3 Mon Jul 13 16:56:35 2009
           Size: 35000
       CheckSum: 3e749
Characteristics: 22  perf
Debug Data Dirs: Type  Size     VA  Pointer
             CODEVIEW    24,  b74c,    ad4c RSDS - GUID: {36CFD5F9-888C-4483-B522-
B9DB242D8478}
               Age: 2, Pdb: notepad.pdb
                CLSID     4,  b748,    ad48 [Data not mapped]
     Image Type: MEMORY   - Image read successfully from loaded memory.
    Symbol Type: PDB      - Symbols loaded successfully from symbol server.
                 c:\symsrv\notepad.pdb\36CFD5F9888C4483B522B9DB242D84782\notepad.pdb
    Load Report: public symbols , not source indexed
                 c:\symsrv\notepad.pdb\36CFD5F9888C4483B522B9DB242D84782\notepad.pdb
```

Parameter Passing

This section discusses how parameters are passed to X64 functions, how the function stack frames are constructed and how the debugger uses this information to walk the call stack.

Register based parameter passing

On X64, the first 4 parameters are always passed in registers and the rest of the parameters are passed via the stack. This is one of main causes of grief during debugging since register values tend to change as functions execute and it becomes difficult to determine the original parameter values that were passed to a function, half-way into its execution. Other than this one issue with retrieving parameters, x64 debugging is not that different from x86 debugging.

Figure 8 shows X64 assembler code depicting how parameters are passed by the caller to the callee.

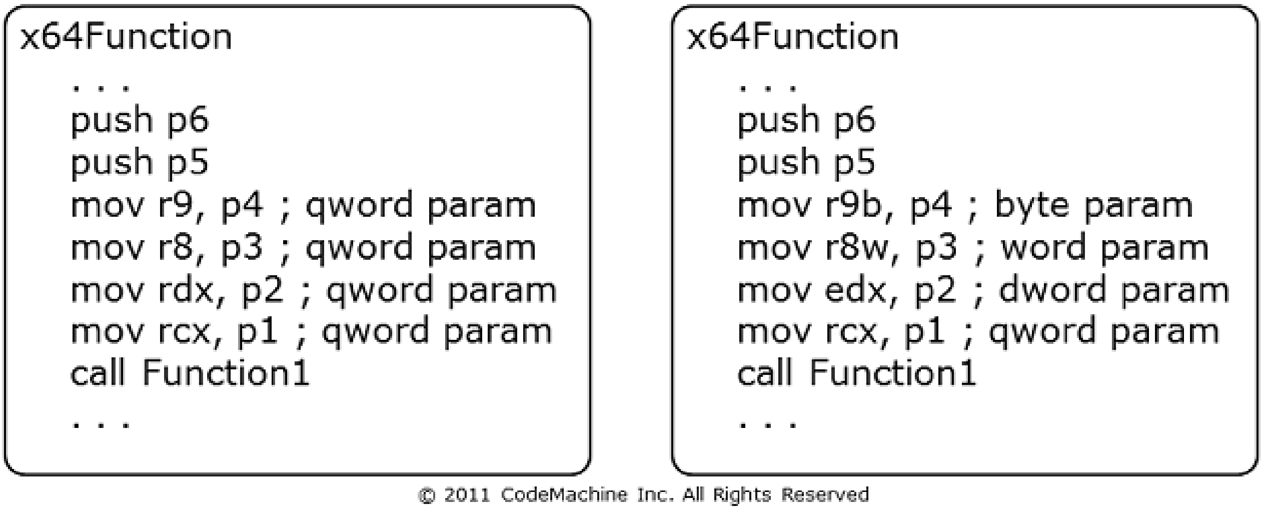

Figure 8 : Parameter Passing on X64

The following call stack shows the function kernel32!CreateFileWImplementation calling KERNELBASE!CreateFileW.

```
0:000> kn
 # Child-SP          RetAddr           Call Site
00 00000000`0029bbf8 000007fe`fdd24d76 ntdll!NtCreateFile
01 00000000`0029bc00 00000000`77ac2aad KERNELBASE!CreateFileW+0x2cd
02 00000000`0029bd60 000007fe`fe5b9ebd kernel32!CreateFileWImplementation+0x7d
.
.
.
```

From the MSDN documentation, the function CreateFileW() takes seven parameters and it's prototype is as follows:

```
HANDLE WINAPI
CreateFile(
  __in      LPCTSTR lpFileName,
  __in      DWORD dwDesiredAccess,
  __in      DWORD dwShareMode,
  __in_opt  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  __in      DWORD dwCreationDisposition,
  __in      DWORD dwFlagsAndAttributes,
  __in_opt  HANDLE hTemplateFile );
```

From the call stack, shown earlier, the return address for the frame containing the function KERNELBASE!CreateFileW is 00000000`77ac2aad. Disassembling backwards from this return address shows the instructions in kernel32!CreateFileWImplementation just before the call to kernel32!CreateFileW. The instructions "mov rcx,rdi", "mov edx,ebx", "mov r8d,ebp", "mov r9,rsi" show the first 4 parameters being moved to registers in preparation for the call to kernel32!CreateFileW. Similarly the instructions "mov dword ptr [rsp+20h],eax", "mov dword ptr [rsp+28h],eax" and "mov qword ptr [rsp+30h],rax" show the rest of parameters, i.e. 5 through 7, being moved to the stack.

```
0:000> ub  00000000`77ac2aad L10
kernel32!CreateFileWImplementation+0x35:
00000000`77ac2a65 lea     rcx,[rsp+40h]
00000000`77ac2a6a mov     edx,ebx
00000000`77ac2a6c call    kernel32!BaseIsThisAConsoleName (00000000`77ad2ca0)
00000000`77ac2a71 test    rax,rax
00000000`77ac2a74 jne     kernel32!zzz_AsmCodeRange_End+0x54fc (00000000`77ae7bd0)
00000000`77ac2a7a mov     rax,qword ptr [rsp+90h]
00000000`77ac2a82 mov     r9,rsi
00000000`77ac2a85 mov     r8d,ebp
00000000`77ac2a88 mov     qword ptr [rsp+30h],rax
00000000`77ac2a8d mov     eax,dword ptr [rsp+88h]
00000000`77ac2a94 mov     edx,ebx
00000000`77ac2a96 mov     dword ptr [rsp+28h],eax
00000000`77ac2a9a mov     eax,dword ptr [rsp+80h]
00000000`77ac2aa1 mov     rcx,rdi
00000000`77ac2aa4 mov     dword ptr [rsp+20h],eax
00000000`77ac2aa8 call    kernel32!CreateFileW (00000000`77ad2c88)
```

Homing Space

Although the first four parameters are passed via registers, there is still space allocated on the stack for these four parameters. This is called the parameter homing space and is used to store parameter values if either the function accesses the parameters by address instead of by value or if the function is compiled with the /homeparams flag. The minimum size of this homing space is 0x20 bytes or four 64-bit slots, even if the function takes less than 4 parameters. When the homing space is not used to store parameter values, the compiler uses it to save non-volatile registers.

Figure 9 shows homing space on the stack for register based parameters and how the function prolog stores non-volatile registers in this parameter homing space.

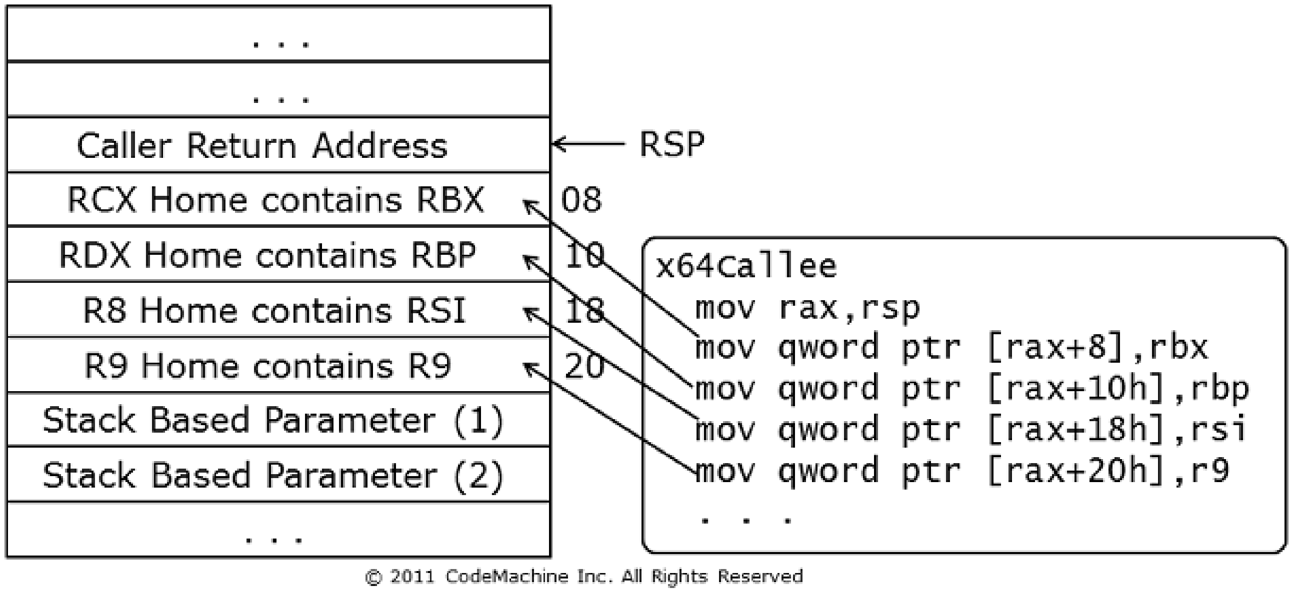

Figure 9 : Parameter Homing Space

In the example below, the "sub rsp, 20h" instruction shows the prolog of a function allocating 0x20 bytes on the stack, which is enough homing space for four 64-bit values. The next part of the example shows that the function msvcrt!malloc() is a non-leaf function in that it calls a bunch of other functions.

```
0:000> uf msvcrt!malloc
msvcrt!malloc:
000007fe`fe6612dc mov     qword ptr [rsp+8],rbx
000007fe`fe6612e1 mov     qword ptr [rsp+10h],rsi
000007fe`fe6612e6 push    rdi
000007fe`fe6612e7 sub     rsp,20h
000007fe`fe6612eb cmp     qword ptr [msvcrt!crtheap (000007fe`fe6f1100)],0
000007fe`fe6612f3 mov     rbx,rcx
000007fe`fe6612f6 je      msvcrt!malloc+0x1c (000007fe`fe677f74)
.
.
.

0:000> uf /c msvcrt!malloc
msvcrt!malloc (000007fe`fe6612dc)
  msvcrt!malloc+0x6a (000007fe`fe66132c):
    call to ntdll!RtlAllocateHeap (00000000`77c21b70)
  msvcrt!malloc+0x1c (000007fe`fe677f74):
    call to msvcrt!core_crt_dll_init (000007fe`fe66a0ec)
  msvcrt!malloc+0x45 (000007fe`fe677f83):
    call to msvcrt!FF_MSGBANNER (000007fe`fe6ace0c)
  msvcrt!malloc+0x4f (000007fe`fe677f8d):
    call to msvcrt!NMSG_WRITE (000007fe`fe6acc10)
  msvcrt!malloc+0x59 (000007fe`fe677f97):
    call to msvcrt!_crtExitProcess (000007fe`fe6ac030)
  msvcrt!malloc+0x83 (000007fe`fe677fad):
    call to msvcrt!callnewh (000007fe`fe696ad0)
  msvcrt!malloc+0x8e (000007fe`fe677fbb):
    call to msvcrt!errno (000007fe`fe661918)
.
.
.
```

The following assembler code snippet of WinMain's prolog shows four non-volatile registers being saved in locations on the stack designated as parameter homing area.

```
0:000> u notepad!WinMain
notepad!WinMain:
00000000`ff4f34b8 mov     rax,rsp
00000000`ff4f34bb mov     qword ptr [rax+8],rbx
00000000`ff4f34bf mov     qword ptr [rax+10h],rbp
00000000`ff4f34c3 mov     qword ptr [rax+18h],rsi
00000000`ff4f34c7 mov     qword ptr [rax+20h],rdi
00000000`ff4f34cb push    r12
00000000`ff4f34cd sub     rsp,70h
00000000`ff4f34d1 xor     r12d,r12d
```

Parameter Homing

As described in the previous section, all X64 non-leaf functions have parameter homing area allocated in their stack frames. As per X64 calling convention, a caller will always use registers to pass the first 4 parameters to the callee. When parameter homing is enabled using the compiler's /homeparams flag, only the callee's code gets affected. This flags is always enabled in checked/debug builds of binaries built using the Windows Driver Kit (WDK) build environment. The callee's prolog reads the parameter values from the registers and stores those values on the stack in to the parameter homing area.

Figure 10 shows the assembler code for the caller where in it moves parameter values into the respective registers. It also shows the prolog of the callee that has been compiled with the /homeparams flag, which causes it to home the parameter values onto the stack. The callee's prolog reads the parameter values from the registers and stores those values on the stack in the parameter homing area.

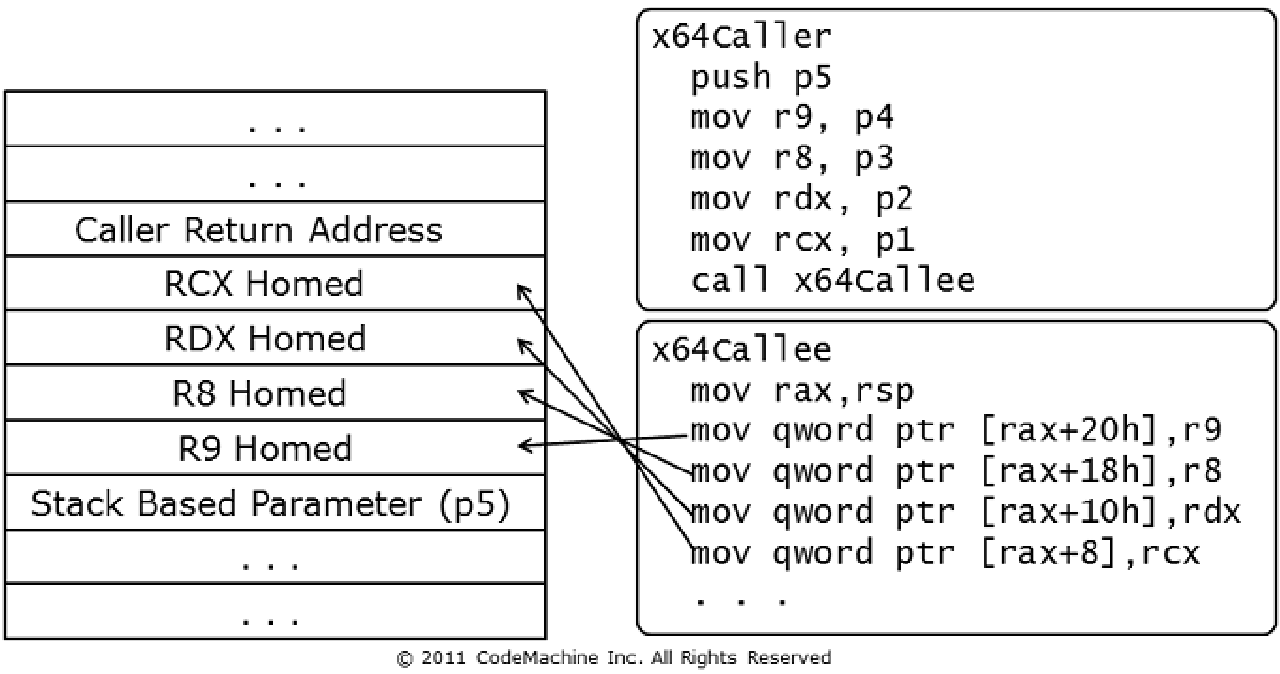

Figure 10 : Parameter Homing

The following code snippet shows register values being moved to homing area on the stack allocated by printf's caller.

```
0:000> uf msvcrt!printf
msvcrt!printf:
000007fe`fe667e28 mov     rax,rsp
000007fe`fe667e2b mov     qword ptr [rax+8],rcx
000007fe`fe667e2f mov     qword ptr [rax+10h],rdx
000007fe`fe667e33 mov     qword ptr [rax+18h],r8
000007fe`fe667e37 mov     qword ptr [rax+20h],r9
000007fe`fe667e3b push    rbx
000007fe`fe667e3c push    rsi
000007fe`fe667e3d sub     rsp,38h
000007fe`fe667e41 xor     eax,eax
000007fe`fe667e43 test    rcx,rcx
000007fe`fe667e46 setne   al
000007fe`fe667e49 test    eax,eax
000007fe`fe667e4b je      msvcrt!printf+0x25 (000007fe`fe67d74b)
.
.
.
```

Stack Usage

The stack frame of an X64 function contains the following items:

- Caller Return Address.
- Non-Volatile registers pushed onto the stack by the function prolog.
- Local variables used by the function.
- Stack based parameters passed to callees.
- Homing space for register based parameters passed to callees.

Other than the return address, all the items on the stack are put there by the function's prolog. The stack space occupied by the locals, stack based parameters to the callees and the homing space for the parameters are all allocated in a single "sub rsp, xxx" instruction. The space reserved for the stack based parameters caters to the callee with the most number of parameters. The register based parameter homing space exists only for non-leaf functions. It contains space for four parameters even if there isn't a single callee that takes that many parameters.

Figure 11 shows the layout of the function stack frame on the X64 CPU. The RSP registers points to location shown in the picture right after the function prolog completes execution.

Figure 11 : Stack Usage

The debugger's "knf" command displays the call stack along with the amount of stack space utilized by every frame in the stack. This stack space utilization is listed under the "Memory" column.

```
0:000> knf
 #   Memory  Child-SP          RetAddr           Call Site
00           00000000`0029bbf8 000007fe`fdd24d76 ntdll!NtCreateFile
01         8 00000000`0029bc00 00000000`77ac2aad KERNELBASE!CreateFileW+0x2cd
02       160 00000000`0029bd60 000007fe`fe5b9ebd
kernel32!CreateFileWImplementation+0x7d
03        60 00000000`0029bdc0 000007fe`fe55dc08 usp10!UniStorInit+0xdd
04        a0 00000000`0029be60 000007fe`fe5534af usp10!InitUnistor+0x1d8
```

The following assembler code snippet shows the prolog of the function CreateFileW, which saves the non-volatile registers r8d and edx to the parameter homing area, pushes rbx, rbp, esi, edi on the stack and allocates 0x138 bytes worth of stack space for local variables and parameters to be passed to the callees.

```
0:000> uf KERNELBASE!CreateFileW
KERNELBASE!CreateFileW:
000007fe`fdd24ac0 mov     dword ptr [rsp+18h],r8d
000007fe`fdd24ac5 mov     dword ptr [rsp+10h],edx
000007fe`fdd24ac9 push    rbx
000007fe`fdd24aca push    rbp
000007fe`fdd24acb push    rsi
000007fe`fdd24acc push    rdi
000007fe`fdd24acd sub     rsp,138h
000007fe`fdd24ad4 mov     edi,dword ptr [rsp+180h]
000007fe`fdd24adb mov     rsi,r9
000007fe`fdd24ade mov     rbx,rcx
000007fe`fdd24ae1 mov     ebp,2
000007fe`fdd24ae6 cmp     edi,3
000007fe`fdd24ae9 jne     KERNELBASE!CreateFileW+0x449 (000007fe`fdd255ff)
```

Child-SP

The value of the Child-SP register displayed by the debugger's "k" command represents the address at which the stack pointer (RSP) points to, as the point where the function displayed in that frame, has finished executing its prolog. The next item that would be pushed on the stack would be the return address of the function as it invokes its callees. Since X64 functions do not modify the value of RSP after the function prolog, any stack accesses performed by the rest of the function are done relative to this position of the stack pointer. This includes access to stack based parameters and local variables.

Figure 12 shows the stack frame of function f2 and its relationship with the RSP register displayed in the output of the stack "k" command. The return address RA1 points to the instruction in function f2 right after the "call f1" instruction. This return address appears on the call stack right next to the location that the RSP2 points to.

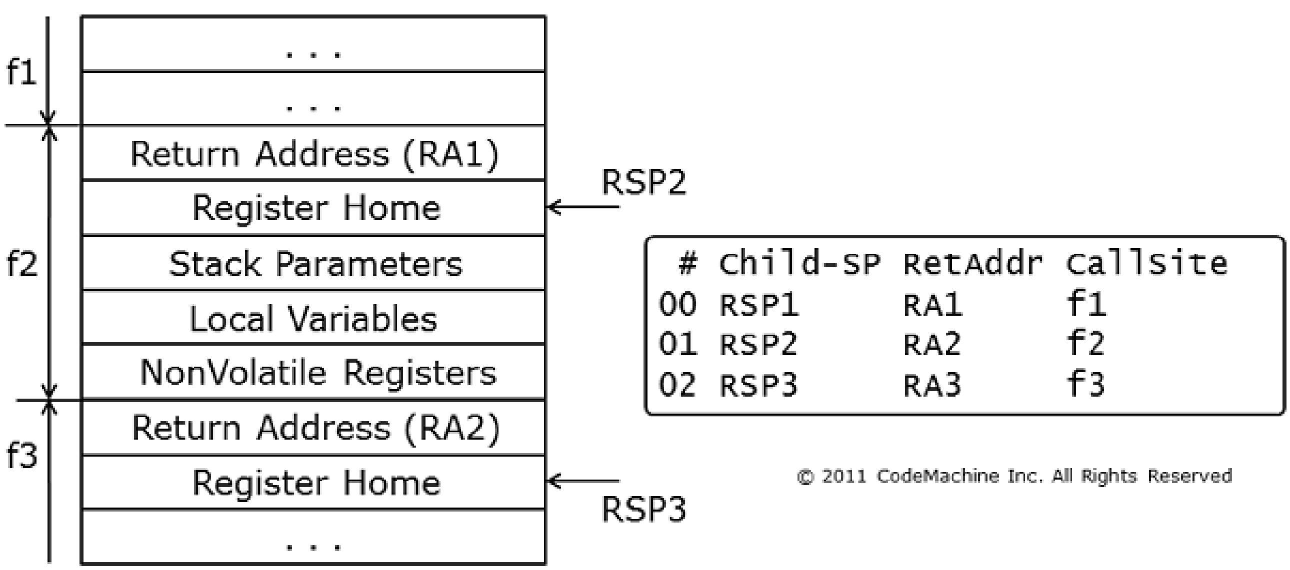

Figure 12 : Relationship between Child-SP and function frames

In the following call stack, the value of Child-SP for frame #01 is 00000000`0029bc00. This is the value of the RSP register at the point of execution in CreateFileW() when its prolog has just completed.

```
0:000> knf
 #   Memory  Child-SP          RetAddr           Call Site
00           00000000`0029bbf8 000007fe`fdd24d76 ntdll!NtCreateFile
01         8 00000000`0029bc00 00000000`77ac2aad KERNELBASE!CreateFileW+0x2cd
02       160 00000000`0029bd60 000007fe`fe5b9ebd
kernel32!CreateFileWImplementation+0x7d
03        60 00000000`0029bdc0 000007fe`fe55dc08 usp10!UniStorInit+0xdd
04        a0 00000000`0029be60 000007fe`fe5534af usp10!InitUnistor+0x1d8
.
.
.
```

As discussed above, the contents of the stack right before the address 00000000`0029bc00 is the return address 000007fe`fdd24d76 which corresponds to KERNELBASE!CreateFileW+0x2cd and is pushed there by the call to ntdll!NtCreateFile.

```
0:000> dps 00000000`0029bc00-8 L1
00000000`0029bbf8  000007fe`fdd24d76 KERNELBASE!CreateFileW+0x2cd
```

Walking the call stack

On the X86 CPU, the debugger follows the frame pointer (EBP) chain to walk the call stack from the most recent function frame to the least recent one. The debugger can typically do this without having access to the symbols of the module whose functions appear on the stack. However this frame pointer chain can be broken under certain circumstances, like when functions have their frame pointer omitted (FPO). In these cases, the debugger needs the symbols of the module to be able to accurately walk the call stack.

X64 functions, on the other hand, don't use the RBP register as a frame pointer and hence, the debugger has no frame pointer chain to follow. Instead, the debugger uses the stack pointer and the size of the stack frame to walk the stack. The debugger locates the RUNTIME_FUNCTION, UNWIND_INFO and UNWIND_CODE structures to compute the stack space utilization for every function in the call stack and adds these values to the Child-SPs to compute the value of subsequent Child-SPs.

Figure 13 shows the layout of a function's stack frame. The total size of the stack frame (or stack space utilization) can be calculated by adding the size of the return address (8 bytes) and the amount of stack space taken up by the non-volatile registers, the local variables, the stack based parameters to callees and the homing space allocated for the four register based parameters (0x20 bytes). The UNWIND_CODE structures indicate the number of non-volatile registers that are pushed on the stack and the amount of space allocated for the locals and the parameters.

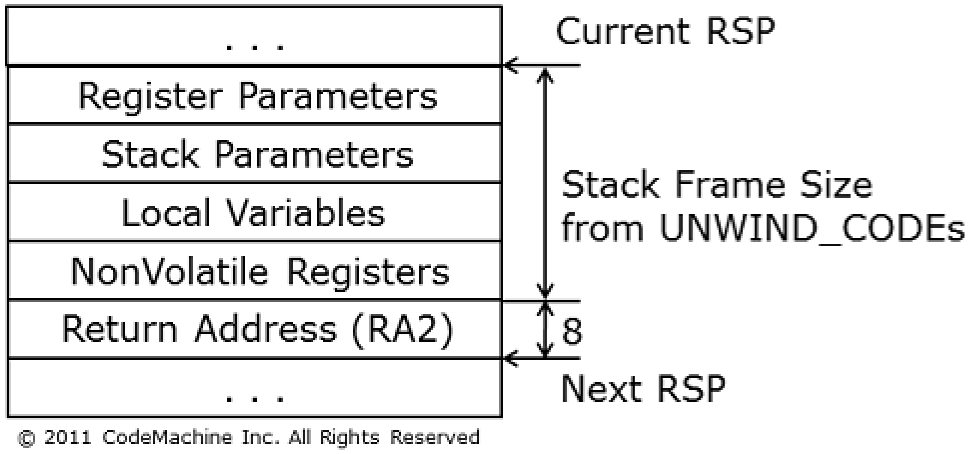

Figure 13 : Walking the x64 call stack

In the following stack trace, the amount of stack space consumed by the function in frame #1 i.e. CreateFileW is 0x160 bytes. The next section shows how this number is computed and how the debugger uses this to compute the value of Child-SP for frame #2. Note that the stack space consumed by the function listed in frame #1 is shown under the "Memory" column for frame #2.

```
0:000> knf
 #   Memory  Child-SP          RetAddr           Call Site
00           00000000`0029bbf8 000007fe`fdd24d76 ntdll!NtCreateFile
01         8 00000000`0029bc00 00000000`77ac2aad KERNELBASE!CreateFileW+0x2cd
02       160 00000000`0029bd60 000007fe`fe5b9ebd
kernel32!CreateFileWImplementation+0x7d
03        60 00000000`0029bdc0 000007fe`fe55dc08 usp10!UniStorInit+0xdd
04        a0 00000000`0029be60 000007fe`fe5534af usp10!InitUnistor+0x1d8
.
.
.
```

The following output shows the operations described by the UNWIND_CODE structures. There are a total of 4 non-volatile registers being pushed on the stack and an allocation of 0x138 bytes for locals and parameters. Non-volatile registers that are moved (UWOP_SAVE_NONVOL), as opposed to pushed (UWOP_PUSH_NONVOL) on to the stack, don't contribute towards consumption of stack space.

```
0:000> .fnent kernelbase!CreateFileW
Debugger function entry 00000000`03be6580 for:
(000007fe`fdd24ac0)   KERNELBASE!CreateFileW   |  (000007fe`fdd24e2c)
KERNELBASE!SbSelectProcedure
Exact matches:
    KERNELBASE!CreateFileW = <no type information>

BeginAddress      = 00000000`00004ac0
EndAddress        = 00000000`00004b18
UnwindInfoAddress = 00000000`00059a48

Unwind info at 000007fe`fdd79a48, 10 bytes
  version 1, flags 0, prolog 14, codes 6
  frame reg 0, frame offs 0
  00: offs 14, unwind op 1, op info 0   UWOP_ALLOC_LARGE FrameOffset: 138
  02: offs d, unwind op 0, op info 7    UWOP_PUSH_NONVOL
  03: offs c, unwind op 0, op info 6    UWOP_PUSH_NONVOL
  04: offs b, unwind op 0, op info 5    UWOP_PUSH_NONVOL
  05: offs a, unwind op 0, op info 3    UWOP_PUSH_NONVOL
```

Adding up the sizes listed above yields a stack space consumption of 0x138 + (8*4) = 0x158 bytes.

```
0:000> ?138+(8*4)
Evaluate expression: 344 = 00000000`00000158
```

Adding the size of the return address (8 bytes) to the above number gives a total stack frame size of 0x160 bytes. This is the same number shown by the debugger's "knf" command, shown earlier.

```
0:000> ?158+8
Evaluate expression: 352 = 00000000`00000160
```

Referring to the output of the "knf" command, the debugger adds the frame size (0x160) to the value of the Child-SP value in frame #01 i.e. 00000000`0029bc00 to get the Child-SP value in frame #02 i.e. 00000000`0029bd60.

```
0:000> ?00000000`0029bc00+160
Evaluate expression: 2735456 = 00000000`0029bd60
```

So the space allocated on the stack for each frame can be computed from information in the PE file itself using the RUNTIME_FUNCTION, UNWIND_INFO and UNWIND_CODE structures. Due to this, the debugger can walk the call stack without requiring symbols (public or private) for the modules present on the stack. The following call stack shows the module "vmswitch" for which symbols are not available on Microsoft's public symbol server but that does not stop the debugger from walking and displaying the call stack accurately, an example of the fact that the X64 call stack can be walked without symbols.

```
1: kd> kn
 # Child-SP          RetAddr           Call Site
00 fffffa60`005f1a68 fffff800`01ab70ee nt!KeBugCheckEx
01 fffffa60`005f1a70 fffff800`01ab5938 nt!KiBugCheckDispatch+0x6e
.
.
.
21 fffffa60`01718840 fffffa60`0340b69e vmswitch+0x5fba
22 fffffa60`017188f0 fffffa60`0340d5cc vmswitch+0x769e
23 fffffa60`01718ae0 fffffa60`0340e615 vmswitch+0x95cc
24 fffffa60`01718d10 fffffa60`009ae31a vmswitch+0xa615
.
.
.
44 fffffa60`0171aed0 fffffa60`0340b69e vmswitch+0x1d286
45 fffffa60`0171af60 fffffa60`0340d4af vmswitch+0x769e
46 fffffa60`0171b150 fffffa60`034255a0 vmswitch+0x94af
47 fffffa60`0171b380 fffffa60`009ac33c vmswitch+0x215a0
.
.
.
```

Parameter Retrieval

In the previous section, the inner workings of the X64 stack was explained along with information on how to interpret every detail from the output of the stack trace displayed by the debugger. In this section, the theory would be applied to demonstrate techniques to retrieve register based parameters passed to X64 functions. Unfortunately, there is no silver bullet to finding parameters. All the techniques here depend heavily on the X64 assembler instructions generated by the compiler. If the parameters are not in "reachable memory", there is simply no way to get them. Having private symbols for modules and functions that appear in the call stack doesn't help too much either. Private symbols do tell the number and types of parameters a function takes, but that's about it. It does not tell what those parameter values are.

Summary of Techniques

The discussions in this section assume that the X64 functions have been compiled without the /homeparams flag. When compiled with the /homeparams flag, it is trivial to retrieve register based parameters as they are guaranteed to be homed on to the stack by the callee. Also the fifth and higher numbered parameters are always passed via the stack, irrespective of whether the function is compiled with /homeparams, so retrieving these parameters should not be an issue in any case.

During live debugging, setting a breakpoint on the beginning of the function is the easiest way to retrieve parameters that were passed in by the caller, since during the function's prolog, the first 4 parameters are guaranteed to be available in the registers RCX, RDX, R8 and R9 respectively.

However, as execution progresses within the function body, the contents of the parameter registers change and the initial parameter value gets overwritten. So, to determine the value of these register based parameters at any point during function execution, one needs to find out - where is the value of the parameter being read from and where is the value of the parameter being written to? Answers to these questions can be found by performing a sequence of steps in the debugger which can be grouped as follows:

- Determine if the parameters are loaded into the registers from memory. If so, the memory location can be examined to determine the parameter values.
- Determine if the parameters are loaded from non-volatile registers and if those registers are saved by the callee. If so, the saved non-volatile register values can be examined to determine the parameter values.
- Determine if the parameters are saved from the registers into memory. If so, the memory location can be examined to determine the parameter values.
- Determine if the parameters are saved into non-volatile registers and if those registers are saved by the callee. If so, the saved non-volatile register values can be examined to determine the parameter values.

In the next few sections, each one of the above techniques is described in detail with examples on how to use them. Each one of the techniques requires disassembling the caller and the callee functions involved in the parameter passing. In Figure 14, if the intention is to find parameters passed to function f2(), frame 2 must be disassembled to find parameter from sources and frame 0 must be disassembled to find them from their destinations.

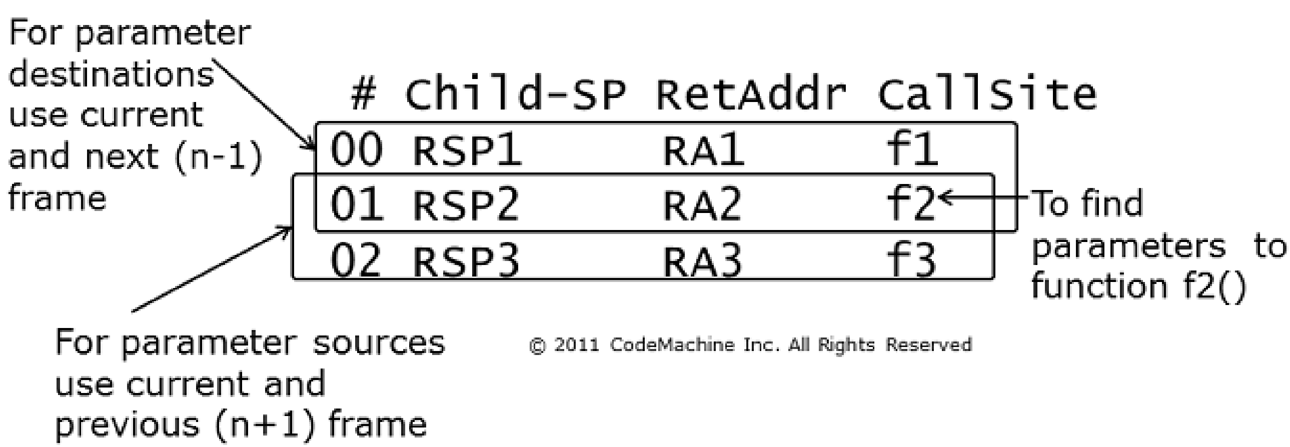

Figure 14 : Finding Register Based Parameters

Identifying Parameter Sources

This technique involves determining the source of the values being loaded into parameter registers. It works for sources like constant values, global data structures, stack addresses, values stored on the stack etc.

As illustrated in Figure 15, disassembling the caller (X64caller) shows that the values being loaded into RCX, RDX, R8 and R9 to be passed as parameters to the function X64callee are being loaded from sources that can be examined in the debugger as long as the values haven't changed.

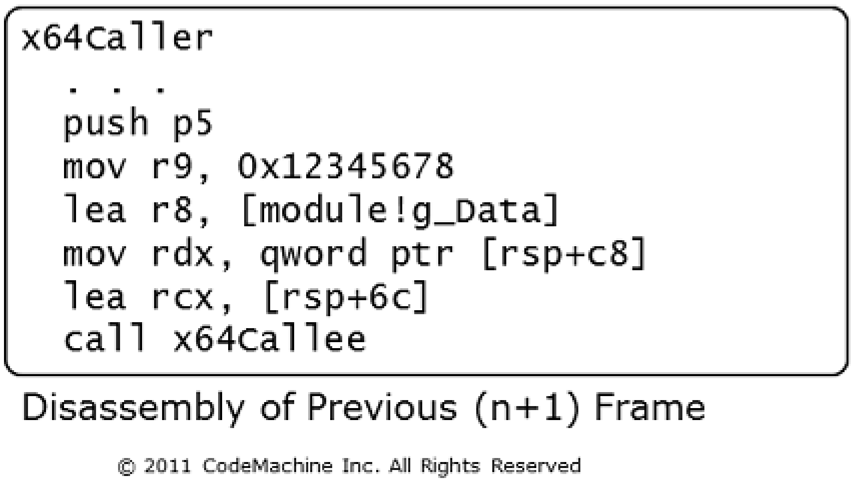

Figure 15 : Identifying parameter sources

The following example applies this technique to find the value of the third parameter to the function NtCreateFile() as show in the call stack below.

```
0:000> kn
 # Child-SP          RetAddr           Call Site
00 00000000`0029bbf8 000007fe`fdd24d76 ntdll!NtCreateFile
01 00000000`0029bc00 00000000`77ac2aad KERNELBASE!CreateFileW+0x2cd
02 00000000`0029bd60 000007fe`fe5b9ebd kernel32!CreateFileWImplementation+0x7d
.
.
.
```

As shown below, from the prototype of the function NtCreateFile(), the parameter type for the third parameter is POBJECT_ATTRIBUTES.

```
NTSTATUS NtCreateFile(
  __out     PHANDLE FileHandle,
  __in      ACCESS_MASK DesiredAccess,
  __in      POBJECT_ATTRIBUTES ObjectAttributes,
  __out     PIO_STATUS_BLOCK IoStatusBlock,
.
.
. );
```

Disassembling the caller using the return address in frame #0 shows the following instructions. The value being loaded into the R8 i.e. the register assigned for parameter 3 is rsp+0xc8. The output of the "kn" command above shows that the value of the RSP register at the time the caller i.e. KERNELBASE!CreateFileW was executing, was 00000000`0029bc00.

```
0:000> ub 000007fe`fdd24d76
KERNELBASE!CreateFileW+0x29d:
000007fe`fdd24d46 and     ebx,7FA7h
000007fe`fdd24d4c lea     r9,[rsp+88h]
000007fe`fdd24d54 lea     r8,[rsp+0C8h]
000007fe`fdd24d5c lea     rcx,[rsp+78h]
000007fe`fdd24d61 mov     edx,ebp
000007fe`fdd24d63 mov     dword ptr [rsp+28h],ebx
000007fe`fdd24d67 mov     qword ptr [rsp+20h],0
000007fe`fdd24d70 call    qword ptr [KERNELBASE!_imp_NtCreateFile]
```

Manually reconstructing the value that was loaded into the R8 register from the information above yields a value that can be type-casted to the OBJECT_ATTRIBUTE structure.

```
0:000> dt ntdll!_OBJECT_ATTRIBUTES 00000000`0029bc00+c8
   +0x000 Length           : 0x30
   +0x008 RootDirectory    : (null)
   +0x010 ObjectName       : 0x00000000`0029bcb0 _UNICODE_STRING "\??
\C:\Windows\Fonts\staticcache.dat"
   +0x018 Attributes       : 0x40
   +0x020 SecurityDescriptor : (null)
   +0x028 SecurityQualityOfService : 0x00000000`0029bc68
```

Non-Volatile Registers as parameter sources

This technique involves finding if the values being loaded into parameter registers are being read out of the non-volatile registers and if the non-volatile registers are being saved on the stack.

Figure 16 shows the disassembly of the caller (X64caller) and the callee (X64Callee). The instructions just before the caller calls the callee (on the left hand side) shows that the values being loaded into the parameter registers (RCX, RDX, R8 and R9) are being read from the non-volatile registers (RDI, R12, RBX, R9). The instructions in the callee's prolog (on the right hand side) show that these non-volatile registers are being saved to the stack. These saved values can be retrieved, which indirectly yield the values that were loaded into the parameter registers earlier.

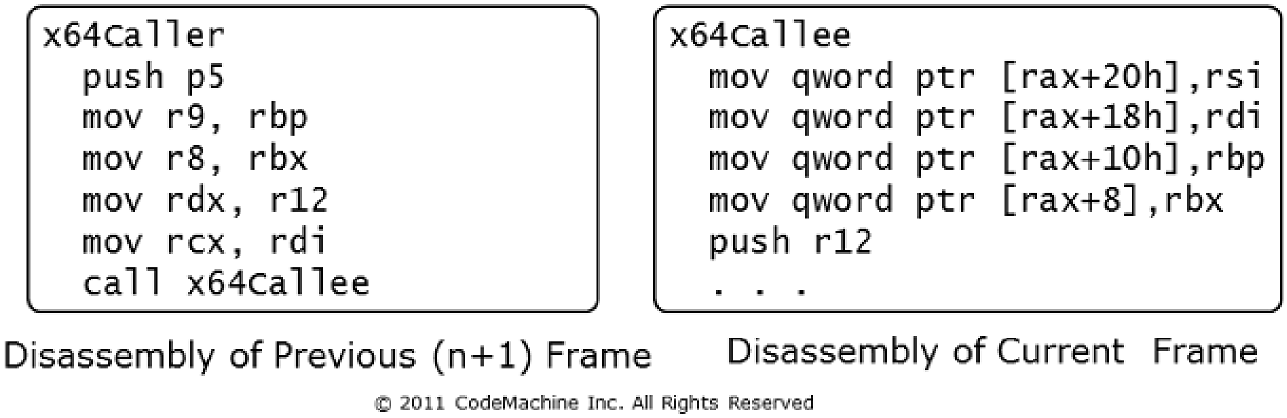

Figure 16 : Non-Volatile Registers as parameter sources

The following example applies this technique to find the value of the first parameter to the function CreateFileW() as shown in the call stack below.

```
0:000> kn
 # Child-SP          RetAddr           Call Site
00 00000000`0029bbf8 000007fe`fdd24d76 ntdll!NtCreateFile
01 00000000`0029bc00 00000000`77ac2aad KERNELBASE!CreateFileW+0x2cd
02 00000000`0029bd60 000007fe`fe5b9ebd kernel32!CreateFileWImplementation+0x7d
.
.
.
```

As shown below, from the prototype of the function CreateFile(), the type for the first parameter is LPCTSTR.

```
HANDLE WINAPI
CreateFile(
  __in      LPCTSTR lpFileName,
  __in      DWORD dwDesiredAccess,
  __in      DWORD dwShareMode,
  __in_opt  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
.
.
. );
```

Disassembling the caller using the return address in frame 1 shows the instructions below. The value being loaded into the RCX i.e. the register assigned for parameter 1 is being read from RDI, a non-volatile register. The next step is to find if the callee CreateFileW() saves EDI.

```
0:000> ub 00000000`77ac2aad L B
kernel32!CreateFileWImplementation+0x4a:
00000000`77ac2a7a mov     rax,qword ptr [rsp+90h]
00000000`77ac2a82 mov     r9,rsi
00000000`77ac2a85 mov     r8d,ebp
00000000`77ac2a88 mov     qword ptr [rsp+30h],rax
00000000`77ac2a8d mov     eax,dword ptr [rsp+88h]
00000000`77ac2a94 mov     edx,ebx
00000000`77ac2a96 mov     dword ptr [rsp+28h],eax
00000000`77ac2a9a mov     eax,dword ptr [rsp+80h]
00000000`77ac2aa1 mov     rcx,rdi
00000000`77ac2aa4 mov     dword ptr [rsp+20h],eax
00000000`77ac2aa8 call    kernel32!CreateFileW (00000000`77ad2c88)
```

Disassembling the callee shows the following instructions in the function's prolog. The RDI register is being saved on the stack by the instruction "push rdi". The value being saved would be the same value that was loaded into the RCX. The next step is to find the saved contents of EDI.

```
0:000> u KERNELBASE!CreateFileW
KERNELBASE!CreateFileW:
000007fe`fdd24ac0 mov     dword ptr [rsp+18h],r8d
000007fe`fdd24ac5 mov     dword ptr [rsp+10h],edx
000007fe`fdd24ac9 push    rbx
000007fe`fdd24aca push    rbp
000007fe`fdd24acb push    rsi
000007fe`fdd24acc push    rdi
000007fe`fdd24acd sub     rsp,138h
000007fe`fdd24ad4 mov     edi,dword ptr [rsp+180h]
```

The debugger's ".frame /r" command displays the values of non-volatile registers when a particular function was executing. It does so by retrieving the non-volatile register values saved by the callee's prolog as discussed earlier. The following command shows the value of EDI as 000000000029beb0 when CreateFileWImplementation() called the CreateFileW(). This value can be used to display the file name parameter that was passed to CreateFile().

```
0:000> .frame /r 2
02 00000000`0029bd60 000007fe`fe5b9ebd kernel32!CreateFileWImplementation+0x7d
rax=0000000000000005 rbx=0000000080000000 rcx=000000000029bc78
rdx=0000000080100080 rsi=0000000000000000 rdi=000000000029beb0
rip=0000000077ac2aad rsp=000000000029bd60 rbp=0000000000000005
 r8=000000000029bcc8  r9=000000000029bc88 r10=0057005c003a0043
r11=00000000003ab0d8 r12=0000000000000000 r13=fffffffffb6011c12
r14=0000000000000000 r15=0000000000000000

0:000> du /c 100 000000000029beb0
00000000`0029beb0  "C:\Windows\Fonts\staticcache.dat"
```

Identifying parameter destinations

This technique involves finding if the values in parameter registers are written to memory within a function. When a function is compiled with /homeparams, the function's prolog will always save the contents of the parameter registers to the parameter homing area on the stack. However, for functions that are not compiled with /homeparams, the parameter register contents may be written to memory anywhere within the function body.

Figure 17 shows the disassembly of a function body wherein the parameter values in registers RCX, RDX, R8 and R9 are being written to the stack. The parameters can be determined by displaying the contents of the memory location using the value of the stack pointer for the current frame.

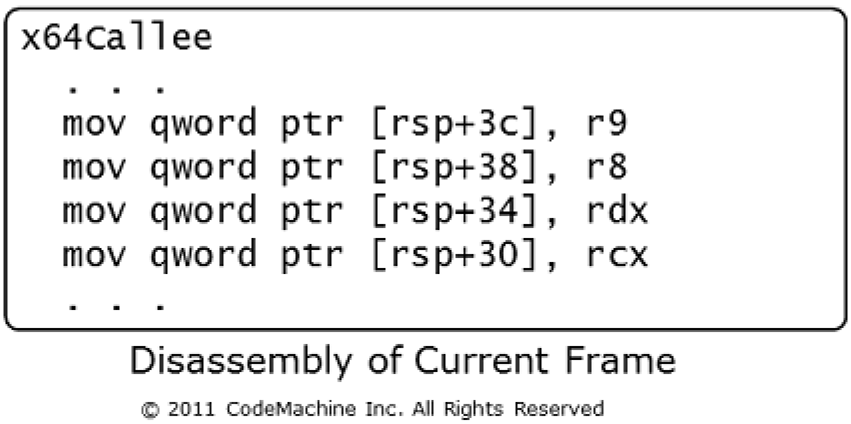

Figure 17 : Identifying parameter destinations

The following example applies this technique to find the value of the third and fourth parameter to the function DispatchClientMessage() as shown in the call stack below.

```
0:000> kn
 # Child-SP          RetAddr           Call Site
.
.
.
26 00000000`0029dc70 00000000`779ca01b user32!UserCallWinProcCheckWow+0x1ad
27 00000000`0029dd30 00000000`779c2b0c user32!DispatchClientMessage+0xc3
28 00000000`0029dd90 00000000`77c1fdf5 user32!_fnINOUTNCCALCSIZE+0x3c
29 00000000`0029ddf0 00000000`779c255a ntdll!KiUserCallbackDispatcherContinue
.
.
.
```

The third and fourth parameters to a function are in the R8 and R9 register respectively. Disassembling the function DispatchClientMessage() and looking for any writes from R8 or R9 to memory, leads to the instructions "mov qword ptr [rsp+28h], r9" and "mov qword ptr [rsp+20h], r8" indicating that the third and fourth parameters are being written to the stack. These instructions are not a part of the function prolog but rather a part of the larger function body. It is important to note this, since the values of the R8 and R9 registers may have been modified before they were written to the stack. Although that does not happen in the case of DispatchClientMessage(), it is important to always verify parameter register overwrites when using this technique.

```
0:000> uf user32!DispatchClientMessage
user32!DispatchClientMessage:
00000000`779c9fbc sub     rsp,58h
00000000`779c9fc0 mov     rax,qword ptr gs:[30h]
00000000`779c9fc9 mov     r10,qword ptr [rax+840h]
00000000`779c9fd0 mov     r11,qword ptr [rax+850h]
00000000`779c9fd7 xor     eax,eax
00000000`779c9fd9 mov     qword ptr [rsp+40h],rax
00000000`779c9fde cmp     edx,113h
00000000`779c9fe4 je      user32!DispatchClientMessage+0x2a (00000000`779d7fe3)

user32!DispatchClientMessage+0x92:
00000000`779c9fea lea     rax,[rcx+28h]
00000000`779c9fee mov     dword ptr [rsp+38h],1
00000000`779c9ff6 mov     qword ptr [rsp+30h],rax
00000000`779c9ffb mov     qword ptr [rsp+28h],r9
00000000`779ca000 mov     qword ptr [rsp+20h],r8
00000000`779ca005 mov     r9d,edx
00000000`779ca008 mov     r8,r10
00000000`779ca00b mov     rdx,qword ptr [rsp+80h]
00000000`779ca013 mov     rcx,r11
00000000`779ca016 call    user32!UserCallWinProcCheckWow (00000000`779cc2a4)
.
.
.
```

Using the value of the stack pointer (RSP) for the frame #27 i.e. 00000000`0029dd30, from the output of the "kn" command above, and adding the offset at which R8 register is stored show 00000000`00000000 which is the value of the third parameter passed to DispatchClientMessage().

```
0:000> dp 00000000`0029dd30+20 L1
00000000`0029dd50  00000000`00000000
```

Similarly adding the offset at which the R9 register is stored shows 00000000`0029de70 which is the value of the fourth parameter passed to DispatchClientMessage().

```
0:000> dp 00000000`0029dd30+28 L1
00000000`0029dd58  00000000`0029de70
```

Non-Volatile Registers as Parameter Destinations

This technique involves finding if the contents of the parameter registers are saved into non-volatile registers by the function in question and then if these non-volatile registers are saved on the stack by the callee.

Figure 18 shows the disassembly of the caller (X64Caller) and the callee (X64Callee). The intention is to find the values of the register based parameters that were passed to the function X64Caller. The body of the function X64Caller (shown on the left hand side) contains instructions that save the parameter registers (RCX, RDX, R8 and R9) into non-volatile registers (RDI, RSI, RBX, RBP). The prolog of the function X64Callee contains instructions (shown on the right hand side) that save these non-volatile registers on to the stack making it feasible to retrieve their values which would indirectly yield the values of the parameter registers.

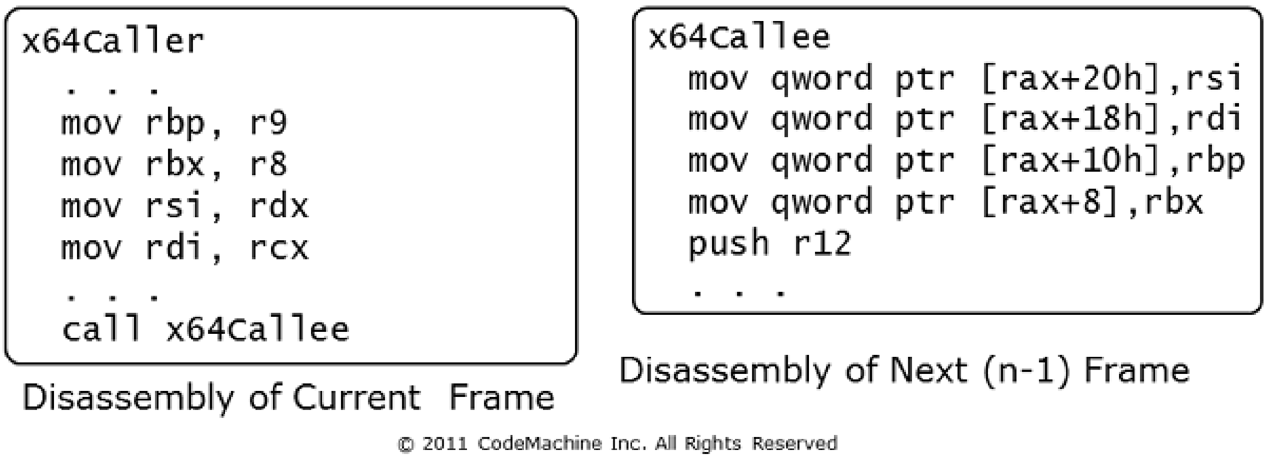

Figure 18 : Non-Volatile Registers as Parameter Destinations

The following example applies this technique to find the value of all the four register based parameters to the function CreateFileWImplementation().

```
0:000> kn
 # Child-SP          RetAddr           Call Site
00 00000000`0029bbf8 000007fe`fdd24d76 ntdll!NtCreateFile
01 00000000`0029bc00 00000000`77ac2aad KERNELBASE!CreateFileW+0x2cd
02 00000000`0029bd60 000007fe`fe5b9ebd kernel32!CreateFileWImplementation+0x7d
03 00000000`0029bdc0 000007fe`fe55dc08 usp10!UniStorInit+0xdd
```

The complete disassembly of the function CreateFileWImplementation() reveals that, right after the function prolog, the parameter registers are being saved to non-volatile registers by the instructions "mov ebx,edx", "mov rdi,rcx", mov rsi,r9" and "mov ebp,r8d". It is important to examine the instructions up to the call to the next function i.e. CreateFileW() to ascertain that these non-volatile registers are not being overwritten. Although not explicitly shown here, this verification has been performed by examining all the code paths in CreateFileWImplementation() that lead to the call to CreateFileW(). The next step is to disassemble the prolog of the function CreateFileW() to find out if it saves these non-volatile registers containing the register based parameters on the stack.

```
0:000> uf kernel32!CreateFileWImplementation
kernel32!CreateFileWImplementation:
00000000`77ac2a30 mov     qword ptr [rsp+8],rbx
00000000`77ac2a35 mov     qword ptr [rsp+10h],rbp
00000000`77ac2a3a mov     qword ptr [rsp+18h],rsi
00000000`77ac2a3f push    rdi
00000000`77ac2a40 sub     rsp,50h
00000000`77ac2a44 mov     ebx,edx
00000000`77ac2a46 mov     rdi,rcx
00000000`77ac2a49 mov     rdx,rcx
00000000`77ac2a4c lea     rcx,[rsp+40h]
00000000`77ac2a51 mov     rsi,r9
00000000`77ac2a54 mov     ebp,r8d
00000000`77ac2a57 call    qword ptr [kernel32!_imp_RtlInitUnicodeStringEx
(00000000`77b4cb90)]
00000000`77ac2a5d test    eax,eax
00000000`77ac2a5f js      kernel32!zzz_AsmCodeRange_End+0x54ec (00000000`77ae7bc0)
.
.
.
```

The following output shows that the function CreateFileW() saves the no-volatile registers (rbx, rbp, rsi and edi) onto the stack, which enables the debugger's ".frame /r" command to display their values.

```
0:000> u KERNELBASE!CreateFileW
KERNELBASE!CreateFileW:
000007fe`fdd24ac0 mov     dword ptr [rsp+18h],r8d
000007fe`fdd24ac5 mov     dword ptr [rsp+10h],edx
000007fe`fdd24ac9 push    rbx
000007fe`fdd24aca push    rbp
000007fe`fdd24acb push    rsi
000007fe`fdd24acc push    rdi
000007fe`fdd24acd sub     rsp,138h
000007fe`fdd24ad4 mov     edi,dword ptr [rsp+180h]
```

Running the command ".frame /r" on frame 2 containing the function CreateFileWImplementation() displays the values of these non-volatile registers at the time that the frame was active.

```
0:000> .frame /r 02
02 00000000`0029bd60 000007fe`fe5b9ebd kernel32!CreateFileWImplementation+0x7d
rax=0000000000000005 rbx=0000000080000000 rcx=000000000029bc78
rdx=0000000080100080 rsi=0000000000000000 rdi=000000000029beb0
rip=0000000077ac2aad rsp=000000000029bd60 rbp=0000000000000005
 r8=000000000029bcc8  r9=000000000029bc88 r10=0057005c003a0043
r11=00000000003ab0d8 r12=0000000000000000 r13=ffffffffb6011c12
r14=0000000000000000 r15=0000000000000000
iopl=0         nv up ei pl zr na po nc
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000244
kernel32!CreateFileWImplementation+0x7d:
00000000`77ac2aad mov     rbx,qword ptr [rsp+60h] ss:00000000`0029bdc0=
{usp10!UspFreeForUniStore (000007fe`fe55d8a0)}
```

Mapping the non-volatile registers with the parameters registers based on the "mov" instructions shown earlier yields the following results.

- P1 = RCX = RDI = 000000000029beb0
- P2 = EDX = EBX = 0000000080000000
- P3 = R8D = EBP = 0000000000000005
- P4 = R9 = RSI = 0000000000000000

It may be time consuming and cumbersome to apply the four steps discussed in this section when attempting to retrieve parameters from X64 call stack. CodeMachine provides a debugger extension command **!cmkd.stack -p** that automates this whole process. This command attempts to retrieve and display parameters to all the functions that appear on the X64 call stack of a thread. In order to use the command to retrieve parameters for any thread during user mode debugging, use the "~s" command to switch to that particular thread. Similarly during kernel mode debugging use the ".thread" command.

This article covered some of the optimizations that the compiler performs on X64 that make the code generated very different from that on X86. It discussed exception handling mechanism on X64 and showed how the executable file format and data structures were modified to support this feature. It then discussed how the X64 stack frame are built at run time and how this knowledge can be applied to retrieve registers based function parameters passed to X64 functions, and thus overcome this painful hurdle on X64.