[Sample chapter from the book "Windows® Crash Dump Analysis" by Dmitry Vostokov]

Introduction to WinDbg Scripts for C/C++ Users

All debuggers from Debugging Tools for Windows package use the same engine dbgeng.dll. It contains a script interpreter for a special language we call WinDbg scripting language for convenience and we use WDS file extension for WinDbg script files. Below is the call stack of a WinDbg thread caught while parsing one of the scripts from this chapter:

```
0:000> ~1kL 100
ChildEBP RetAddr
037cd084 6dd28cdc dbgeng!TypedData::ForceU64+0x3
037cd0ec 6dcbd08c dbgeng!GetPseudoOrRegVal+0x11c
037cd134 6dcbceff dbgeng!MasmEvalExpression::GetTerm+0x12c
037cd198 6dcbca23 dbgeng!MasmEvalExpression::GetMterm+0x36f
037cd1d4 6dcbc873 dbgeng!MasmEvalExpression::GetAterm+0x13
037cd220 6dcbc783 dbgeng!MasmEvalExpression::GetShiftTerm+0x13
037cd254 6dcbc523 dbgeng!MasmEvalExpression::GetLterm+0x13
037cd2c0 6dcbc443 dbgeng!MasmEvalExpression::GetLRterm+0x13
037cd2f4 6dcbc424 dbgeng!MasmEvalExpression::StartExpr+0x13
037cd308 6dcbbc2f dbgeng!MasmEvalExpression::GetCommonExpression+0xc4
037cd31c 6dccdca3 dbgeng!MasmEvalExpression::Evaluate+0x4f
037cd390 6dccd83d dbgeng!EvalExpression::EvalNum+0x63
037cd3d0 6dd293cc dbgeng!GetExpression+0x5d
037cd458 6dd2a7e2 dbgeng!ScanRegVal+0xfc
037cd4ec 6dd17502 dbgeng!ParseRegCmd+0x422
037cd52c 6dd194e8 dbgeng!WrapParseRegCmd+0x92
037cd608 6dc8ed19 dbgeng!ProcessCommands+0x1278
037cd644 6dc962af dbgeng!DotFor+0x1d9
037cd658 6dd1872e dbgeng!DotCommand+0x3f
037cd738 6dd19b49 dbgeng!ProcessCommands+0x4be
```

```
037cd77c 6dc5c879 dbgeng!ProcessCommandsAndCatch+0x49
037cdc14 6dd19cc3 dbgeng!Execute+0x2b9
037cdc64 6dc89db0 dbgeng!ProcessCurBraceBlock+0xa3
037cdc74 6dc962af dbgeng!DotBlock+0x10
037cdc88 6dd1872e dbgeng!DotCommand+0x3f
037cdd68 6dd19b49 dbgeng!ProcessCommands+0x4be
037cddac 6dc5c879 dbgeng!ProcessCommandsAndCatch+0x49
037ce244 6dd173ca dbgeng!Execute+0x2b9
037ce2c4 6dd1863c dbgeng!ParseDollar+0x29a
037ce3a0 6dd19b49 dbgeng!ProcessCommands+0x3cc
037ce3e4 6dc5c879 dbgeng!ProcessCommandsAndCatch+0x49
037ce87c 6dc5cada dbgeng!Execute+0x2b9
037ce8ac 00318693 dbgeng!DebugClient::ExecuteWide+0x6a
037ce954 00318b83 windbg!ProcessCommand+0x143
037cf968 0031ae46 windbg!ProcessEngineCommands+0xa3
037cf97c 76fa19f1 windbg!EngineLoop+0x366
037cf988 77c8d109 kernel32!BaseThreadInitThunk+0xe
037cf9c8 00000000 ntdll!_RtlUserThreadStart+0x23
```

In this chapter I assume that you already know C or C++ language or any C-style language like Java or C#. Therefore I omit explanation for language elements that appear to have similar syntax and semantics when we look and compare equivalent C/C++ and WinDbg script code.

Hello World

Let's write our first script that prints the famous message.

```
$$ HelloWorld.wds - Hello World script
.block
{
  .printf "Hello World!\n"
}
```

This script is multiline and it has to be executed using either **$>**< or **$$><** command:

```
0:000> $$><c:\scripts\HelloWorld.wds
Hello World!
```

One line scripts can be executed when we type them in WinDbg command window or you load them from a file using **$<** or **$$<** commands:

```
$$ Hello World script; .block { .printf "Hello World!\n" }
```

We can see that in one line scripts comments and commands must be ended with a semicolon unless the command or comment is final. Semicolons are not required for multiline scripts if commands are on separate lines.

```
0:000> $$<c:\scripts\HelloWorld2.wds
0:000> $$ Hello World script; .block { .printf "Hello World!\n" }
Hello World!
```

From now on we will use only multiline scripts because of their readability. You might have noticed that I deliberately made the first script more complex than necessary by enclosing **.printf** in **.block { }** to show the resemblance to C-style function:

```
// Hello World function
void helloWorld ()
{
  printf ("Hello World!\n");
}
```

Simple arithmetic

Consider the simple C-style function that prints the sum of 2 numbers and uses local variables:

```
void sum ()

{

    unsigned long t1 = 2;

    unsigned long t2 = 3;

    unsigned long t0 = t1 + t2;

    printf("Sum(%x,%x) = %x\n", t1, t2, t0);

}
```

In WinDbg scripts we can use 20 different user-defined variables called pseudo-registers. Their names are **$t0** - **$t19**. If you want to obtain the pseudo-register value then use **@** symbol, for example, **@$t0**. We can use **%p** type field character in **.printf** to interpret the value as a pointer. This is the equivalent WinDbg script and its output:

```
$$ Arithmetic1.wds - Calculate the sum of two predefined variables

.block

{

   r $t1 = 2

   r $t2 = 3

   r $t0 = @$t1 + @$t2

   .printf "Sum(%p, %p) = %p\n", @$t1, @$t2, @$t0

}
```

```
0:000> $$>a<c:\scripts\Arithmetic1.wds

Sum(0000000000000002, 0000000000000003) = 0000000000000005
```

Using hardcoded values is not useful. Let's rewrite the same function to use parameters. The equivalent to function arguments in WinDbg scripts are **$arg1** - **$argN** aliases to character strings. To obtain the alias value enclose it into ${…}, for example, **${$arg1}**. However we don't need to enclose it if you use it in some expression and the type of the argument can be inferred from other participating operands.

```
$$ Arithmetic2.wds - Calculate the sum of two function arguments

.block

{

   r $t0 = $arg1 + $arg2

   .printf "Sum(%p, %p) = %p\n", ${$arg1}, ${$arg2}, @$t0

}
```

Now we can call scripts and specify arguments:

```
0:000> $$>a<c:\scripts\Arithmetic2.wds 2 3

Sum(0000000000000002, 0000000000000003) = 0000000000000005
```

If some arguments are missing we get an error:

```
0:000> $$>a<c:\scripts\Arithmetic2.wds

Couldn't resolve error at '${$arg1} + ${$arg2};   .printf "Sum(%p, %p) = %p\n", ${$arg1}, ${$arg2}, @$t0;'
```

WinDbg allows us to check whether arguments are defined or not. This can be done via a special form of the alias evaluator **${/d:…}**:

```
$$ Arithmetic3.wds - Calculate the sum of two optional function arguments

.block

{

   r $t1 = 0

   .if (${/d:$arg1})

   {

      r $t1 = $arg1

   }

   r $t2 = 0

   .if (${/d:$arg2})

   {

      r $t2 = $arg2

   }

   .printf "Sum(%p, %p) = %p\n", @$t1, @$t2, @$t1+@$t2
```

```
}
```

Here is the script output for some arguments:

```
0:000> $$>a<c:\scripts\Arithmetic3.wds
Sum(0000000000000000, 0000000000000000) = 0000000000000000
0:000> $$>a<c:\scripts\Arithmetic3.wds 2
Sum(0000000000000002, 0000000000000000) = 0000000000000002
0:000> $$>a<c:\scripts\Arithmetic3.wds 2 3
Sum(0000000000000002, 0000000000000003) = 0000000000000005
```

Factorial

Let's write more complicated script that computes the factorial of the given number. Recall the following definition of the factorial function:

n! = 1*2*3*4*…*(n-2)*(n-1)*n

This function can be computed recursively using this code:

```
// C-style factorial function using recursion
unsigned long factorial (unsigned long n)
{
   unsigned long f = 0;
   if (n > 1)
   {
     f = n*factorial(n-1);
   }
   else
   {
     f = 1;
   }
   return f;
```

```
}
```

Alternatively it can be computed using **while** or **for** loops:

```
// C-style factorial function using a “while” loop
unsigned long factorial (unsigned long n)
{
  unsigned long k=1;
  while (n-1)
  {
    k = k * n;
    --n;
  }
  return k;
}
// C-style factorial function using a “for” loop
unsigned long factorial2 (unsigned long n)
{
  unsigned long k=1;
  for (; n-1; --n)
  {
    k = k * n;
  }
  return k;
}
```

WinDbg scripts can be called recursively too. We can map C-style code to WinDbg script where **$t0** pseudo register is used to simulate the function return value:

```
$$ FactorialR.wds - Calculate factorial using recursion
.block
{
  .if (${$arg1} > 1)
  {
    $$>a<c:\scripts\FactorialR.wds ${$arg1}-1
    r $t1 = $arg1
    r $t0 = @$t1 * @$t0
  }
  .else
  {
    r $t0 = 1
  }
  .printf "Factorial(%p) = %p\n", ${$arg1}, @$t0
}
```

The output of the script for some arguments:

```
0:000> $$>a<c:\scripts\FactorialR.wds 1
Factorial(0000000000000001) = 0000000000000001
0:000> $$>a<c:\scripts\FactorialR.wds 2
Factorial(0000000000000001) = 0000000000000001
Factorial(0000000000000002) = 0000000000000002
0:000> $$>a<c:\scripts\FactorialR.wds 3
Factorial(0000000000000001) = 0000000000000001
Factorial(0000000000000002) = 0000000000000002
Factorial(0000000000000003) = 0000000000000006
0:000> $$>a<c:\scripts\FactorialR.wds 4
Factorial(0000000000000001) = 0000000000000001
```

```
Factorial(0000000000000002) = 0000000000000002
Factorial(0000000000000003) = 0000000000000006
Factorial(0000000000000004) = 0000000000000018
0:000> $$>a<c:\scripts\FactorialR.wds 10
Factorial(0000000000000001) = 0000000000000001
Factorial(0000000000000002) = 0000000000000002
Factorial(0000000000000003) = 0000000000000006
Factorial(0000000000000004) = 0000000000000018
Factorial(0000000000000005) = 0000000000000078
Factorial(0000000000000006) = 00000000000002d0
Factorial(0000000000000007) = 00000000000013b0
Factorial(0000000000000008) = 0000000000009d80
Factorial(0000000000000009) = 0000000000058980
Factorial(000000000000000a) = 0000000000375f00
Factorial(000000000000000b) = 0000000002611500
Factorial(000000000000000c) = 000000001c8cfc00
Factorial(000000000000000d) = 000000017328cc00
Factorial(000000000000000e) = 000000144c3b2800
Factorial(000000000000000f) = 0000013077775800
Factorial(0000000000000010) = 0000130777758000
```

Now we are ready to rewrite our script using a **while** loop.

```
 $$ FactorialL.wds - Calculate factorial using a "while" loop
.block
{
    r $t0 = 1
    r $t1 = $arg1
    .while (@$t1-1)
    {
        r $t0 = @$t0 * @$t1
        r $t1 = @$t1 - 1
    }
    .printf "Factorial(%p) = %p\n", ${$arg1}, @$t0
}
```

The output of the script for some arguments:

```
0:000> $$>a<c:\scripts\FactorialL.wds 1
Factorial(0000000000000001) = 0000000000000001
0:000> $$>a<c:\scripts\FactorialL.wds 2
Factorial(0000000000000002) = 0000000000000002
0:000> $$>a<c:\scripts\FactorialL.wds 3
Factorial(0000000000000003) = 0000000000000006
0:000> $$>a<c:\scripts\FactorialL.wds 4
Factorial(0000000000000004) = 0000000000000018
0:000> $$>a<c:\scripts\FactorialL.wds 10
Factorial(0000000000000010) = 0000130777758000
```

We can simplify the script using **.for** loop token:

```
$$ FactorialL2.wds - Calculate factorial using a "for" loop
.block
{
    .for (r $t0 = 1, $t1 = $arg1; @$t1-1; r $t1 = @$t1 - 1)
    {
      r $t0 = @$t0 * @$t1
    }
    .printf "Factorial(%p) = %p\n", ${$arg1}, @$t0
}
```

Its output is the same:

```
0:000> $$>a<c:\scripts\FactorialL2.wds 4
Factorial(0000000000000004) = 0000000000000018
```

[The rest of the sample chapter is available when the book is published]