Development Lifecycle » Debug Tips » General

ASM, VC6, VC7, VC7.1Win2K, WinXP, Win2003, Visual Studio, MFC, Dev, QA

Debug Tutorial Part 2: The Stack

By Toby Opferman

Introduction to the most important ally in the fight against bugs, the stack.

Posted: **20 Mar 2004**
Updated: **29 Mar 2004**
Views: **113,596**
Bookmarked: **167 times**

81 votes for this article.
Popularity: 9.17 Rating: **4.81** out of 5
1 2 3 4 5

Introduction

Welcome to the second installment of these debugging tutorials. In this article, I will investigate the stack and how it plays an integral part in debugging. Anytime you ask the question "What do you do when your program traps?", the most common answer is "Get a stack trace". This is definitely true, it's probably the very first thing you should do anytime you investigate a crash dump.

Sorry if this tutorial is too general and beginner-ish! I should probably set the level as beginner instead of intermediate. I only set it as intermediate since the article is requiring assembly knowledge.

What is the Stack?

This is the first and most obvious question. Unfortunately, I did not cover or really answer this in the first tutorial as I was taking for granted that everyone was familiar with it. In order to explain what the stack is, let me start from where we begin, what a process is.

What is a Process?

A process is basically an instance of an application in memory. The executable and supported libraries are mapped into this address space. A process does not execute, but just rather defines the memory boundaries, resources, and the like which are accessible from anyone operating within that process.

What is a Thread?

A thread is an instance of execution that operates within the boundaries of a process. A process is not scheduled to execute, the threads within a process are. There may be many threads executing in the context of one process. Although a thread may have "thread specific storage", generally all memory and resources created in the context of the process can be used by any executing thread.

Global and Local Resources

Not to be confusing here, but there are exceptions. There are resources that are created globally rather than locally. That means these resources may be used outside the context of the process in which they were created. One such example is a window handle. These resources have their own boundaries outside of a process. Some resources may be system wide, others desktop or session wide. There are also "shared" resources where processes can negotiate sharing of a resource through other means and mechanisms.

What is Virtual Memory?

In general, "Virtual Memory" is generally thought of as fooling the system into thinking there's more physical memory than there really is. This is true and false at the same time. It depends on who "the system" is and there's really more to it than that.

The system is not being fooled into thinking there's more memory than there really is. The hardware already knows there's less memory and is actually the one who implements the necessary mechanisms to support "Virtual Memory". The Operating System is the one which utilizes these capabilities to perform "Virtual Memory", so it is also not being fooled. So, who is being fooled? If anyone is being fooled, it's the processes running on the system.

I don't believe that to be the case either. The application programmer generally knows the system he is programming for already. That means, he knows the Operating System uses "Virtual Memory" or not such as DOS and he programs for that platform. In general, it doesn't mean anything. A simple application really doesn't care as long as it gets to execute. The only time you really run into trouble would be a "cooperative multitasking" system verses a "preemptive multitasking" system. But, then again, the programmer knows his target platform and programs appropriately. The differences with those two types of Operating Systems is beyond the scope of this article and does not apply.

So, back to answering this question. The first thing that "Virtual Memory" does is that it abstracts the physical address space of the machine. This means the application programs do not see or know about the physical address. They know a "Virtual" address. The CPU is then capable of converting a "Virtual" address to a "Physical" address based on certain characteristics setup by the Operating System. The details of that mechanism is beyond the scope of this document. Just understand that the application receives a "Virtual" address and the processor maps it to a physical address.

The next part of the equation is that the "Virtual" address does not need to point to a "Physical" address. The Operating System can use a swap file to keep memory on disk, that way, the entire program does not have to be in physical memory at the same time. This allows many programs to be in memory and execute. If a program attempts to access a memory location that is not in physical memory, the CPU knows this. The CPU will page fault and know that the memory that is being accessed is out on disk. The Operating System will then get told and will pull that memory from disk to Physical memory. Once this is complete, the program is given back execution and will continue where it left off.

There are many algorithms to decide how to pull memory in from disk. Unless you plan to grow the footprint of a process in Physical memory, you usually swap a page out to swap one in. There are many algorithms that the OS can use to grow the process physical foot print and swap pages in and out. A simple one is basically, the least frequently used page in memory. You generally want to avoid writing programs that keep crossing page boundaries frequently, this will eliminate "thrashing" which is swapping in and out pages from memory to disk often. These topics are outside the scope of this tutorial.

The next advantage of "Virtual Memory" is protection. A process cannot directly access another process's memory. That means that at anyone time, the CPU has only the Virtual address mappings for that process. That means, it can't resolve a virtual address in another process. This makes sense because since they are separate mappings, the processes could and will have the same memory address pointed to different locations!

That doesn't mean it's impossible to read another process' memory. If the Operating System has built-in support, such as Windows does, you can access another process' memory. You could also do this if you could gain access to memory locations, and manipulate the CPU registers as they relate to Virtual Memory mapping. Luckily, you can't, as the CPU can check your privilege level before you attempt to execute sensitive assembly instructions, and "Virtual Memory" will keep you away from being a usermode process and manipulating page or descriptor tables (Although, there is a method in Windows 9x to get the LDT in usermode).

What is the stack?

Now that I've described the basics of the system, I can get back to "What is a stack?". In general, a stack is a general-purpose data structure that allows items to be pushed onto it and popped off. Think of it as a stack of plates. You can put items on the top and you can only take items off the top (without cheating). If you followed that strict rule, you have a stack. A stack is generally referred to as "LIFO" or "Last In First Off".

Programs generally use the stack as a means of temporary storage. This is generally unknown to the non-assembly programmer as the language hides these details. However, the generated code produced by your program will use a stack and the CPU has built-in stack support!

On Intel, the assembly instructions to put something on the stack and take something off are `PUSH` and `POP`. Note that some processors use `PUSH`/`PULL`, but in the Intel world, we use `PUSH`/`POP`. This is just a "mnemonic" anyway, which basically means an English word for a machine operation. All assembly boils down to an opcode or number which is processed by the CPU. That means, you can call the instruction anything you want in English as long as you use it correctly and generate the correct "opcode".

Getting back on track, every "thread" executing in a process has its **own** stack. This is because we can't have multiple threads attempting to use the same temporary storage location as we will see in a moment.

How is a function call made?

The function call depends on the "calling convention". The "calling convention" is a basic method that the caller (the function making the call) and callee (the function being called) have agreed on in order to pass parameters to the function and clean up the parameters afterwards. In Windows, we generally support three different calling conventions. These are "this call", "standard call" and "CDECL or C Calling convention".

This Call

This is a C++ calling convention. If you're familiar with C++ internals, member functions of an object require the `this` pointer to be passed into the function. In general, the `this` pointer is the first parameter on the stack. This is not the case with "this call". In "this call", the `this` pointer is in a register, `ECX` to be exact. The parameters are pushed on the stack in reverse order and the callee cleans up the stack.

Standard Call

"Standard Call" is when the parameters are pushed backwards on to the stack and the callee cleans up the stack.

CDECL or C Calling Convention

"C Calling" convention basically means that the parameters are pushed backwards onto the stack and the caller cleans up the stack.

Pascal Calling Convention

If you've seen old programs, you will see "PASCAL" as their calling convention. In WIN32, you actually are not allowed to use `__pascal` anymore. The `PASCAL` macro has actually been redefined as "Standard Call". However, Pascal calling convention parameters are pushed **forward** onto the stack. The callee cleans up the stack.

Cleans up the stack?

The difference in who cleans up the stack is a big deal. The first is saving bytes. If the callee cleans up the stack, that means there doesn't have to be extra instructions generated at every function call to clean up the stack. The disadvantage to this is that you cannot use variable arguments. Variable arguments are used by functions like `printf`. The actual callee does not *REALLY* know how many arguments are pushed onto the stack. It can only *GUESS* by the information provided to it, in say, its format string. If you tell `printf "%i %i %i"`, it will attempt to use 3 more values on the stack to fill those, whether or not you pushed them or not! This may or may not trap. If you push more parameters than you tell `printf`, there's no problem since the caller is cleaning up the stack anyway. They're just there for no reason, but `printf` does not know they're there. Remember that, variable argument functions do not magically know how many parameters are there, they must implement some method for the caller to tell them through their parameter list. `Printf`'s just happen to be the format string, you could even pass a number down if you want, but the compiler isn't going to do it for you.

Also, although it is possible to then clean up the stack, it's not entirely feasible. Since the function does not know at compile time how many parameters are sent to it, it means it has to manipulate the stack and move around the return value in order to clean up. It's easier to just let the caller clean up the stack in this case.

Intel supports an instruction to clean up the stack by the callee. It's `RET <Byte Count>` where `Byte Count` is the size in bytes of the parameters on the stack. A 2 byte instruction.

So, what is the stack?

A stack is a location for temporary storage. Parameters are pushed onto the stack, then the return address is pushed onto the stack. The flow of execution must know where to return to. The CPU is stupid, it just executes one instruction after the other. You have to tell it where to go. In order to tell it how to get back, we need to save the return address, the location after the function call. There is an assembly instruction that does this for us: `CALL`. There is an assembly instruction that uses the current value on the stack and the return address and transfers execution to that location. It's called `RET`. Beyond this, we have local variables and possibly even other values that are pushed onto the stack for temporary storage. This is one of the reasons you can never return an array or an address of any local variable, they disappear when the function returns!

The layout of the stack would be the following:

```
[Parameter n          ]
...
[Parameter 2          ]
[Parameter 1          ]
[Return Address       ]
[Previous Base Pointer]
[Local Variables      ]
```

Before returning, the stack is cleaned up to the return address and then a "return" is issued. If the stack is not kept in proper order, we may get out of sync and return to the wrong address! This can cause a trap, obviously!

What is the "base pointer"?

The "base pointer" in Intel is generally `EBP`. What happens is since `ESP` is dynamic and always changing, you save the old `EBP` from the previous function, then set `EBP` to the current stack location. You can now reference variables directly on the stack from a standard offset. This means that the first parameter will always be `EBP` + `xx`, etc. If you do not save `ESP` and always reference `ESP`, you're going to have to keep track of how much data is on the stack. If you put more and more data on the stack, the offset to the first parameter changes. The assembler **does** generate functions when appropriate to not set `EBP`, so it's not always the case that `EBP` is the base pointer but rather the function could be using `ESP` directly.

Generally, it's `EBP` + `Value` to get to function parameters and `EBP` - `Value` to get to local variables.

Putting it all together

So, you can now see the reason each thread has its own stack. If they shared the same stack, they would overwrite each other's return values and data! Or, could eventually, if they ran out of stack space. That's the next problem we will discuss.

Stack Overflow

A Stack Overflow is when you reached the end of your stack. Windows generally gives the program a fixed amount of user mode stack space. The kernel has its own stack. It generally occurs when you run out of stack space! Recursion is a good way to run out of stack space. If you keep recursively calling a function you may eventually run out of stack and trap.

Windows generally does not allocate all of the stack at once, but instead grows the stack as you need it. This is an optimization obviously.

We can write a small program to perform a stack overflow and then find out how much stack Windows gave us.

```
0:000> g
(928.898): Stack overflow - code c00000fd (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00131ad8 ebx=7ffdf000 ecx=00131ad8 edx=00430df0 esi=00000000 edi=0003347c
eip=00401029 esp=00032ffc ebp=00033230 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010216
*** WARNING: Unable to verify checksum for COMSTRESS.exe
COMSTRESS!func+0x9:
00401029 53               push    ebx
0:000> !teb
TEB at 7ffde000
    ExceptionList:        0012ffb0
    StackBase:            00130000
    StackLimit:           00031000
    SubSystemTib:         00000000
    FiberData:            00001e00
    ArbitraryUserPointer: 00000000
    Self:                 7ffde000
    EnvironmentPointer:   00000000
    ClientId:             00000928 . 00000898
    RpcHandle:            00000000
    Tls Storage:          00000000
    PEB Address:          7ffdf000
    LastErrorValue:       0
    LastStatusValue:      0
    Count Owned Locks:    0
    HardErrorMode:        0
0:000> ? 130000 - 31000
Evaluate expression: 1044480 = 000ff000
0:000>
```

To do this, I simply used !teb, which displays all elements of the TEB or "Thread Environment Block" (found at FS:0 as mentioned in a previous tutorial). If you subtract the base of the stack from the stack limit, you get the size. 1,044,480 bytes is how big of a stack Windows gave us.

Stack Underflow

In general, a Stack Underflow is the opposite of an overflow. You've somehow thought you put more on the stack than you really have and you've popped off too much. You've reached the beginning of the stack and it's empty, but you thought there was more data and kept attempting to pop data off.

Overflows and Underflows

Overflows and Underflows can also be said to occur when your program gets out of sync and crashes thinking the stack is in a different position. The stack could underflow if you clean up too much in a function and then attempt to return. Your stack is out of sync and you return to the wrong address. The reason your stack is out of sync is you thought you had more data on it than you did. You could consider that an underflow.

The opposite can also occur. You've cleaned up too little because you didn't think you had that much data on the stack and you return. You trap when you return because you went to the wrong address. You "could" consider this an overflow as you are out of sync, thinking you have less data on the stack than you really do.

How does the debugger get a stack trace?

This brings me to my next topic, how does a debugger get a stack trace? The first answer is simply by using "Symbols". The symbols can tell the debugger how many parameters are on the stack, how many local variables, etc., so the debugger can then use the symbols to determine how to walk the stack and display the information.

If there are no symbols, it uses the base pointer. Each base pointer points to the previous base pointer. The Base Pointer + 4 also points to the return address. This is how it then walks the stack. If everyone uses `EBP`, the stack trace could be a perfect world. Although the debugger does not know how many parameters there are, it just dumps the location where the parameters **WOULD** be, it's up to you to interpret what the correct parameters are.

Here is a simple table of some function calls. I am going to use the stack trace from the first tutorial.

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012fef4 77c3e68d 77c5aca0 00000000 0012ff44 MSVCRT!_output+0x18
0012ff38 00401044 00000000 77f944a8 00000007 MSVCRT!printf+0x35
0012ff4c 00401147 00000001 00323d70 00322ca8 temp!main+0x44
0012ffc0 77e814c7 77f944a8 00000007 7ffdf000 temp!mainCRTStartup+0xe3
0012fff0 00000000 00401064 00000000 78746341 kernel32!BaseProcessStart+0x23
```

Since `ESP` will point to local variables, etc., we will dump `EBP`. I will use the "DDS" command which means 'Dump Dwords with Symbols'. The debugger will attempt to match the value on the stack with the closest symbol.

Our current `EBP` value is 0012fef4. This is a pointer on the stack, remember? This value points to the previous `EBP`. Remember, `EBP` + 4 == return value, `EBP` + 8 == Parameters. The bold walks the stack to each `EBP` value.

```
[Stack Address |  Value   | Description]
0012fef4  0012ff38
0012fef8  77c3e68d MSVCRT!printf+0x35
0012fefc  77c5aca0 MSVCRT!_iob+0x20
0012ff00  00000000
0012ff04  0012ff44
0012ff08  77c5aca0 MSVCRT!_iob+0x20
0012ff0c  00000000
0012ff10  000007e8
0012ff14  7ffdf000
0012ff18  0012ffb0
0012ff1c  00000001
0012ff20  0012ff0c
0012ff24  0012f8c8
0012ff28  0012ffb0
0012ff2c  77c33eb0 MSVCRT!_except_handler3
0012ff30  77c146e0 MSVCRT!`string'+0x16c
0012ff34  00000000
0012ff38  0012ffc0
0012ff3c  00401044 temp!main+0x44
0012ff40  00000000
0012ff44  77f944a8 ntdll!RtlpAllocateFromHeapLookaside+0x42
0012ff48  00000007
0012ff4c  00000000
0012ff50  00401147 temp!mainCRTStartup+0xe3
0012ff54  00000001
0012ff58  00323d70
0012ff5c  00322ca8
0012ff60  00403000 temp!__xc_a
0012ff64  00403004 temp!__xc_z
0012ff68  0012ffa4
0012ff6c  0012ff94
0012ff70  0012ffa0
0012ff74  00000000
0012ff78  0012ff98
0012ff7c  00403008 temp!__xi_a
0012ff80  0040300c temp!__xi_z
0012ff84  77f944a8 ntdll!RtlpAllocateFromHeapLookaside+0x42
0012ff88  00000007
0012ff8c  7ffdf000
0012ff90  c0000005
0012ff94  00323d70
0012ff98  00000000
0012ff9c  8053476f
0012ffa0  00322ca8
0012ffa4  00000001
0012ffa8  0012ff84
0012ffac  0012f8c8
0012ffb0  0012ffe0
0012ffb4  00401210 temp!except_handler3
0012ffb8  004020d0 temp!&#8962;MSVCRT_NULL_THUNK_DATA+0x80
0012ffbc  00000000
0012ffc0  0012fff0
0012ffc4  77e814c7 kernel32!BaseProcessStart+0x23
0012ffc8  77f944a8 ntdll!RtlpAllocateFromHeapLookaside+0x42
0012ffcc  00000007
0012ffd0  7ffdf000
0012ffd4  c0000005
0012ffd8  0012ffc8
0012ffdc  0012f8c8
0012ffe0  ffffffff
0012ffe4  77e94809 kernel32!_except_handler3
0012ffe8  77e91210 kernel32!`string'+0x98
0012ffec  00000000
0012fff0  00000000
0012fff4  00000000
0012fff8  00401064 temp!mainCRTStartup
```

So, EBP points to (0012fef4) which points to the previous EBP of 0012ff38. EIP == 77c3f10b, which is MSVCRT!_output+0x18. We can then dump EBP + 8 as the parameters. The debugger with "KB" generally dumps the first 3 values of the stack. It doesn't know if those are correct parameters or not, it's just a preview. If you want to know the rest, you simply find the location on the stack and dump.

```
0012fefc  77c5aca0 MSVCRT!_iob+0x20
0012ff00  00000000
0012ff04  0012ff44
```

So, we can assemble the first function:

```
MSVCRT!_output+0x18(77c5aca0, 00000000, 0012ff44);
```

The second function is EBP + 4, the return address. Remember, it doesn't know where the functions start. So, the best it can do is match the return address specifying this as the function.

This is the calling function:

```
0012fef8  77c3e68d MSVCRT!printf+0x35
```

It then goes to the previous EBP, 0012ff38, and adds 8 to get the parameters.

```
0012ff40  00000000
0012ff44  77f944a8 ntdll!RtlpAllocateFromHeapLookaside+0x42
0012ff48  00000007
```

This is the calling function with its parameters.

```
MSVCRT!printf+0x35(00000000, 77f944a8, 00000007);
```

As you can see, if anything is off, this information is wrong. That is why you must use your judgment when interpreting these values.

The next EBP was :0012ffc0. It's the memory location at 0012ff38. The previous return value is:

```
0012ff3c  00401044 temp!main+0x44
```

The previous parameters were at 0012ffc0 + 8. Remember, this also assumes that EBP was the first value pushed onto the stack. If the debugger is smart enough, it could attempt to just walk the stack until it gets the first recognizable symbol and use that as the return value! That's in case something was pushed onto the stack before EBP was saved and set.

These are the parameters:

```
0012ffc8  77f944a8 ntdll!RtlpAllocateFromHeapLookaside+0x42
0012ffcc  00000007
0012ffd0  7ffdf000

temp!main+0x44(77f944a8, 00000007, 7ffdf000)
```

Our next EBP was 0012ffc0, so + 4 is the return value. That's our function now.

```
0012ffc4  77e814c7 kernel32!BaseProcessStart+0x23
```

So, EBP = 0012ffc0, points to previous EBP 0012fff0 and we know that previous EBP + 8 == parameters.

```
0:000> dds 0012fff0
0012fff0  00000000
0012fff4  00000000  <-- Previous return value is NULL so stop here.
0012fff8  00401064 temp!mainCRTStartup  <-- + 8
0012fffc  00000000
00130000  78746341

kernel32!BaseProcessStart+0x23(00401064, 00000000, 78746341)
```

This should be good enough since our previous return value is `NULL`. So, this is our manual generation of the stack:

```
MSVCRT!_output+0x18            (77c5aca0, 00000000, 0012ff44);
MSVCRT!printf+0x35             (00000000, 77f944a8, 00000007);
temp!main+0x44                 (77f944a8, 00000007, 7ffdf000);
kernel32!BaseProcessStart+0x23 (00401064, 00000000, 78746341);
```

This was our stack trace from the debugger:

```
ChildEBP RetAddr  Args to Child
0012fef4 77c3e68d 77c5aca0 00000000 0012ff44 MSVCRT!_output+0x18
0012ff38 00401044 00000000 77f944a8 00000007 MSVCRT!printf+0x35
0012ff4c 00401147 00000001 00323d70 00322ca8 temp!main+0x44
0012ffc0 77e814c7 77f944a8 00000007 7ffdf000 temp!mainCRTStartup+0xe3
0012fff0 00000000 00401064 00000000 78746341 kernel32!BaseProcessStart+0x23
```

What's different and why? Well, we followed a simple rule to walk the stack. `EBP` points to the previous `EBP`. Secondly, we didn't use symbol information to walk the stack. If I delete the symbols for *temp.exe*, I get the following stack trace:

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012fef4 77c3e68d 77c5aca0 00000000 0012ff44 MSVCRT!_output+0x18
0012ff38 00401044 00000000 77f944a8 00000007 MSVCRT!printf+0x35
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ffc0 77e814c7 77f944a8 00000007 7ffdf000 temp+0x1044
0012fff0 00000000 00401064 00000000 78746341 kernel32!BaseProcessStart+0x23
0:000>
```

The same as ours! So, the debugger used symbolic information to walk the stack and display a more accurate picture. However, without symbolic information, there's function calls missing. That means, we cannot always trust the stack trace if symbols are wrong, missing or not complete. If we do not have symbol information for all modules, then we have a problem!

If I continue with these tutorials, one of the next ones will attempt to explain symbols and validating them. However, I will attempt to show you one trick to validating function calls in this tutorial.

As we can see, we notice we are missing a function call. How do you validate function calls? By verifying they were made.

Verifying Function Calls

I ran the program again and got a new stack trace.

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012fef4 77c3e68d 77c5aca0 00000000 0012ff44 MSVCRT!_output+0x18
0012ff38 00401044 00000000 00000000 00000000 MSVCRT!printf+0x35
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ffc0 77e814c7 00000000 00000000 7ffdf000 temp+0x1044
0012fff0 00000000 00401064 00000000 78746341 kernel32!BaseProcessStart+0x23
0:000>
```

Some of the values on the stack are different, but that's what happens when you run programs again. You're not guaranteed the same run every time!

This is your first return value: 77c3e68d

If you un-assemble it, you will get this:

```
0:000> u 77c3e68d
MSVCRT!printf+0x35:
77c3e68d 8945e0          mov     [ebp-0x20],eax
77c3e690 56              push    esi
77c3e691 ff75e4          push    dword ptr [ebp-0x1c]
```

The listing reads like this:

```
<address> <opcode> <assembly instruction in english or mnemonic>

77c3e691 ff75e4  push dword ptr [ebp-0x1c]

77c3e691 == Address
ff75e4   == Opcode or machine code.  This is what the CPU understands
push dword ptr [ebp-0x1c]  == Assembly instruction in english. The mnemonic.
```

This is the return value. What is a return value? It's the next instruction **after** a call is made. Thus, if we keep subtracting from this value, we will eventually un-assemble the call instruction. The trick is to un-assemble enough to make out the call function. Be warned though, Intel opcodes are variable. That means that they are **not** a fixed size and un-assembling in the **middle** of an instruction can generate a completely different instruction and even different instruction list! So, we have to guess. Usually if we go back enough, the instructions eventually get back on track and are unassembled correctly.

```
0:000> u 77c3e68d - 20
MSVCRT!printf+0x15:
77c3e66d bdadffff59      mov     ebp,0x59ffffad
77c3e672 59              pop     ecx
77c3e673 8365fc00        and     dword ptr [ebp-0x4],0x0
77c3e677 56              push    esi
77c3e678 e8c7140000      call    MSVCRT!_stbuf (77c3fb44)
77c3e67d 8945e4          mov     [ebp-0x1c],eax
77c3e680 8d450c          lea     eax,[ebp+0xc]
77c3e683 50              push    eax
0:000> u
MSVCRT!printf+0x2c:
77c3e684 ff7508          push    dword ptr [ebp+0x8]
77c3e687 56              push    esi
77c3e688 e8660a0000      call    MSVCRT!_output (77c3f0f3)
77c3e68d 8945e0          mov     [ebp-0x20],eax
```

As you can see, the return address is 77c3e68d. So, 77c3e688 is the function call. Thus, we are calling `_output`! So, that is a correct function call. Want to try another?

The next return address listed in the stack trace is 00401044. Let's try the same:

```
0:000> u 00401044  - 20
temp+0x1024:
00401024 2408            and     al,0x8
00401026 57              push    edi
00401027 50              push    eax
00401028 6a04            push    0x4
0040102a 6820304000      push    0x403020
0040102f 56              push    esi
00401030 ff1500204000    call    dword ptr [temp+0x2000 (00402000)]
00401036 56              push    esi
0:000> u
temp+0x1037:
00401037 ff1508204000    call    dword ptr [temp+0x2008 (00402008)]
0040103d 57              push    edi
0040103e ff1510204000    call    dword ptr [temp+0x2010 (00402010)]
00401044 59              pop     ecx
```

Unfortunately yes, this is assembly. This is basically a function pointer. It means call the function at address 00402010.

Use "DD" to get the value at the address.

```
0:000> dd 00402010
00402010  77c3e658
```

We will now un-assemble this address since we know it's a function call.

```
0:000> u 77c3e658
MSVCRT!printf:
77c3e658 6a10             push    0x10
```

So, yes, we're calling `printf` here.

The next return value is 77e814c7. Let's see if we're calling `temp`.

```
0:000> u 77e814c7 - 20
kernel32!BaseProcessStart+0x3:
77e814a7 1012             adc     [edx],dl
77e814a9 e977e8288e       jmp     0610fd25
77e814ae ffff             ???
77e814b0 8365fc00         and     dword ptr [ebp-0x4],0x0
77e814b4 6a04             push    0x4
77e814b6 8d4508           lea     eax,[ebp+0x8]
77e814b9 50               push    eax
77e814ba 6a09             push    0x9
0:000> u
kernel32!BaseProcessStart+0x18:
77e814bc 6afe             push    0xfe
77e814be ff159c13e677 call dword ptr [kernel32!_imp__NtSetInformationThread (77e
6139c)]
77e814c4 ff5508           call    dword ptr [ebp+0x8]
77e814c7 50               push    eax
```

It's calling the first parameter. This is the first parameter:

```
0012fff0 00000000 00401064 00000000 78746341 kernel32!BaseProcessStart+0x23

0:000> u 00401064
temp+0x1064:
00401064 55               push    ebp
00401065 8bec             mov     ebp,esp
00401067 6aff             push    0xff
```

So, it is calling something inside of `temp`. Unfortunately, we can't be certain it's the same function. In order to find that out, we need to un-assemble this function and walk through it. Remember, the call to `printf` is at:

```
0040103e ff1510204000     call    dword ptr [temp+0x2010 (00402010)]
```

That means that we have to un-assemble this function from its start all the way to 0401064. Another way to do it would be to use DDS on the stack and find out if there are any other symbols on the stack and verify them.

If we do DDS on `EBP`, we find this:

```
0:000> dds ebp
0012fef4  0012ff38
0012fef8  77c3e68d MSVCRT!printf+0x35
0012fefc  77c5aca0 MSVCRT!_iob+0x20
0012ff00  00000000
0012ff04  0012ff44
0012ff08  77c5aca0 MSVCRT!_iob+0x20
0012ff0c  00000000
0012ff10  000007e8
0012ff14  7ffdf000
0012ff18  0012ffb0
0012ff1c  00000001
0012ff20  0012ff0c
0012ff24  ffffffff
0012ff28  0012ffb0
0012ff2c  77c33eb0 MSVCRT!_except_handler3
0012ff30  77c146e0 MSVCRT!`string'+0x16c
0012ff34  00000000
0012ff38  0012ffc0
0012ff3c  00401044 temp+0x1044
0012ff40  00000000
0012ff44  00000000
0012ff48  00000000
0012ff4c  00000000
0012ff50  00401147 temp+0x1147
0012ff54  00000001
0012ff58  00322470
0012ff5c  00322cf8
0012ff60  00403000 temp+0x3000
0012ff64  00403004 temp+0x3004
0012ff68  0012ffa4
0012ff6c  0012ff94
0012ff70  0012ffa0
0:000> dds
0012ff74  00000000
0012ff78  0012ff98
0012ff7c  00403008 temp+0x3008
0012ff80  0040300c temp+0x300c
0012ff84  00000000
0012ff88  00000000
0012ff8c  7ffdf000
0012ff90  00000001
0012ff94  00322470
0012ff98  00000000
0012ff9c  8053476f
0012ffa0  00322cf8
0012ffa4  00000001
0012ffa8  0012ff84
0012ffac  e3ce0b30
0012ffb0  0012ffe0
0012ffb4  00401210 temp+0x1210
0012ffb8  004020d0 temp+0x20d0
0012ffbc  00000000
0012ffc0  0012fff0
0012ffc4  77e814c7 kernel32!BaseProcessStart+0x23
```

There are a lot of unknown `TEMP` + `xxx` values on the stack! The one in bold is the one we know is the return value for the `printf()`. 00401064, we know is the start address of the function called from `BaseProcessStart()`. What values are close to this one?

This is where guess work comes in. If you think that function does not jump backwards, you could attempt to only look at values that are > than this one. You could attempt to un-assemble every single reference, but you have to start somewhere. I would say, look at the symbols closest to this one first. Here is one:

```
0012ff50  00401147 temp+0x1147

0:000> u 00401147 - 20
temp+0x1127:
00401127 40               inc     eax
00401128 00e8             add     al,ch
0040112a 640000           add     fs:[eax],al
0040112d 00ff             add     bh,bh
0040112f 1520204000       adc     eax,0x402020
00401134 8b4de0           mov     ecx,[ebp-0x20]
00401137 8908             mov     [eax],ecx
00401139 ff75e0           push    dword ptr [ebp-0x20]
0:000> u
temp+0x113c:
0040113c ff75d4           push    dword ptr [ebp-0x2c]
0040113f ff75e4           push    dword ptr [ebp-0x1c]
00401142 e8b9feffff       call    temp+0x1000 (00401000)
00401147 83c430           add     esp,0x30
```

We found this function call and it looks to be a valid address. The way to distinguish an invalid return value on the stack is, the previous instruction is **not** a `CALL` instruction. That is one way to distinguish return values from just values on the stack that may be a symbol, but not return value.

Let's un-assemble this one:

```
0:000> u 00401000
temp+0x1000:
00401000 51               push    ecx
00401001 56               push    esi
00401002 57               push    edi
00401003 33ff             xor     edi,edi
00401005 57               push    edi
00401006 57               push    edi
00401007 6a03             push    0x3
00401009 57               push    edi
0:000> u
temp+0x100a:
0040100a 57               push    edi
0040100b 6800000080       push    0x80000000
00401010 6810304000       push    0x403010
00401015 ff1504204000     call    dword ptr [temp+0x2004 (00402004)]
0040101b 8bf0             mov     esi,eax
0040101d 83feff           cmp     esi,0xffffffff
00401020 741b             jz      temp+0x103d (0040103d)
00401022 8d442408         lea     eax,[esp+0x8]
0:000> u
temp+0x1026:
00401026 57               push    edi
00401027 50               push    eax
00401028 6a04             push    0x4
0040102a 6820304000       push    0x403020
0040102f 56               push    esi
00401030 ff1500204000     call    dword ptr [temp+0x2000 (00402000)]
00401036 56               push    esi
00401037 ff1508204000     call    dword ptr [temp+0x2008 (00402008)]
0:000>
temp+0x103d:
0040103d 57               push    edi
0040103e ff1510204000     call    dword ptr [temp+0x2010 (00402010)]
00401044 59               pop     ecx
00401045 5f               pop     edi
00401046 33c0             xor     eax,eax
00401048 5e               pop     esi
00401049 59               pop     ecx
0040104a c3               ret
0:000>
```

This looks like a valid function call and looks like it calls `printf`. So, we could then disassemble the original function call until we reached this call, to see if it called it, or if there's yet another function call in between.

```
0:000> u 0401064
temp+0x1064:
00401064 55              push    ebp
00401065 8bec            mov     ebp,esp
00401067 6aff            push    0xff
00401069 68d0204000      push    0x4020d0
0040106e 6810124000      push    0x401210
00401073 64a100000000    mov     eax,fs:[00000000]
00401079 50              push    eax
0040107a 648925000000000 mov     fs:[00000000],esp
0:000> u
temp+0x1081:
00401081 83ec20          sub     esp,0x20
00401084 53              push    ebx
00401085 56              push    esi
00401086 57              push    edi
00401087 8965e8          mov     [ebp-0x18],esp
0040108a 8365fc00        and     dword ptr [ebp-0x4],0x0
0040108e 6a01            push    0x1
00401090 ff153c204000    call    dword ptr [temp+0x203c (0040203c)]
0:000>
temp+0x1096:
00401096 59              pop     ecx
00401097 830d40304000ff  or     dword ptr [temp+0x3040 (00403040)],0xffffffff
0040109e 830d44304000ff  or     dword ptr [temp+0x3044 (00403044)],0xffffffff
004010a5 ff1538204000    call    dword ptr [temp+0x2038 (00402038)]
004010ab 8b0d3c304000    mov     ecx,[temp+0x303c (0040303c)]
004010b1 8908            mov     [eax],ecx
004010b3 ff1534204000    call    dword ptr [temp+0x2034 (00402034)]
004010b9 8b0d38304000    mov     ecx,[temp+0x3038 (00403038)]
0:000>
temp+0x10bf:
004010bf 8908            mov     [eax],ecx
004010c1 a130204000      mov     eax,[temp+0x2030 (00402030)]
004010c6 8b00            mov     eax,[eax]
004010c8 a348304000      mov     [temp+0x3048 (00403048)],eax
004010cd e8e1000000      call    temp+0x11b3 (004011b3)
004010d2 833d2830400000  cmp     dword ptr [temp+0x3028 (00403028)],0x0
004010d9 750c            jnz     temp+0x10e7 (004010e7)
004010db 68b0114000      push    0x4011b0
0:000>
temp+0x10e0:
004010e0 ff152c204000    call    dword ptr [temp+0x202c (0040202c)]
004010e6 59              pop     ecx
004010e7 e8ac000000      call    temp+0x1198 (00401198)
004010ec 680c304000      push    0x40300c
004010f1 6808304000      push    0x403008
004010f6 e897000000      call    temp+0x1192 (00401192)
004010fb a134304000      mov     eax,[temp+0x3034 (00403034)]
00401100 8945d8          mov     [ebp-0x28],eax
0:000>
temp+0x1103:
00401103 8d45d8          lea     eax,[ebp-0x28]
00401106 50              push    eax
00401107 ff3530304000    push    dword ptr [temp+0x3030 (00403030)]
0040110d 8d45e0          lea     eax,[ebp-0x20]
00401110 50              push    eax
00401111 8d45d4          lea     eax,[ebp-0x2c]
00401114 50              push    eax
00401115 8d45e4          lea     eax,[ebp-0x1c]
0:000>
temp+0x1118:
00401118 50              push    eax
00401119 ff1524204000    call    dword ptr [temp+0x2024 (00402024)]
0040111f 6804304000      push    0x403004
00401124 6800304000      push    0x403000
00401129 e864000000      call    temp+0x1192 (00401192)
0040112e ff1520204000    call    dword ptr [temp+0x2020 (00402020)]
00401134 8b4de0          mov     ecx,[ebp-0x20]
00401137 8908            mov     [eax],ecx
0:000>
temp+0x1139:
00401139 ff75e0          push    dword ptr [ebp-0x20]
0040113c ff75d4          push    dword ptr [ebp-0x2c]
0040113f ff75e4          push    dword ptr [ebp-0x1c]
00401142 e8b9feffff      call    temp+0x1000 (00401000)
```

If you know assembly, you can simply read through the logic and bet that it could have made it here or if it could not make it here. If it could, then provided two functions do not share the same assembled code base, there are two function calls missing. Now, if you just find their return values on the stack, you can find their parameter list. What can we assume from this? Some of these functions do not use `EBP`, thus we could not get an accurate stack trace. When that happens, we need to verify our trace. As we can see, the previous function did not call `printf`, but another one did that it called.

00401147 is the missing return value. If we find it on the stack, we can update the correct parameters:

```
00000000
00401147 temp+0x1147
00000001
00322470
00322cf8
```

So, here's the one generated from KB:

```
0:000> kb
ChildEBP RetAddr  Args to Child
0012fef4 77c3e68d 77c5aca0 00000000 0012ff44 MSVCRT!_output+0x18
0012ff38 00401044 00000000 00000000 00000000 MSVCRT!printf+0x35
WARNING: Stack unwind information not available. Following frames may be wrong.
0012ffc0 77e814c7 00000000 00000000 7ffdf000 temp+0x1044
0012fff0 00000000 00401064 00000000 78746341 kernel32!BaseProcessStart+0x23
0:000>
```

Here's our modified one:

```
ChildEBP RetAddr  Args to Child
0012fef4 77c3e68d 77c5aca0 00000000 0012ff44 MSVCRT!_output+0x18
0012ff38 00401044 00000000 00000000 00000000 MSVCRT!printf+0x35
WARNING: Stack unwind information not available. Following frames may be wrong.

xxxxxxxx  0401147  00000001 00322470 00322cf8 temp+0x1044
0012ffc0 77e814c7 00000000 00000000 7ffdf000 temp+0x1147
0012fff0 00000000 00401064 00000000 78746341 kernel32!BaseProcessStart+0x23
0:000>
```

We know that the `temp` that calls `printf()` is `main()`. So, `argc` = 1, `*argv[]` = 322470.

`*argv[]` is a pointer to an array of pointers which are ANSI strings.

```
0:000> dd 322470
00322470  00322478 00000000 706d6574 abababab00
00322480  abababab feeefeab 00000000 00000000
00322490  000500c5 feee0400 00325028 00320178
003224a0  feeefeee feeefeee feeefeee feeefeee
003224b0  feeefeee feeefeee feeefeee feeefeee
003224c0  feeefeee feeefeee feeefeee feeefeee
003224d0  feeefeee feeefeee feeefeee feeefeee
003224e0  feeefeee feeefeee feeefeee feeefeee
0:000> da 00322478
00322478  "temp"
```

Dumping the array, which only contains 1 string as per `argc`, we can then use the "da" command to view that string as shown above.

Multiple Return Addresses On The Stack?

Why are there multiple return addresses on the stack? The stack may generally be initialized to zero, but as it's being used, it becomes dirty. You know that local variables aren't always initialized, so if you make a function call, those values aren't reset to zero when the stack moves up. If you pop a value off the stack, the stack may decrement, but the values stay unless they are physically

cleaned up. Sometimes, the stack optimizes things and doesn't clean up variables as well. So, seeing "ghost" values on the stack is very common.

This is not always desirable to leave values on the stack. For example, if your function puts the password on the stack and traps sometime later. A stack dump may still show the password on the stack! So, sometimes when you have sensitive information, you may want to clean up the values on the stack before you return. One way to do this is with the `SecureZeroMemory()` API. This can be used to clear memory securely as calling other APIs may be "optimized" out of the code, for example, if you call them before you return. The compiler knows you're not going to use the variable anymore and may not perform the clearing.

Buffer Overflows

Buffer overflows are a common occurrence on the stack. The stack grows down in memory, but arrays grow up in memory. This is because you usually "increment" a pointer or array when using it to get to the next index, rather than decrementing it. Thus let's say this was your C function:

```
{
      DWORD MyArray[4];
      int Index;
```

That would evaluate to a stack like this:

```
424 [Return Address              ]
420 [ Previous Base Pointer      ]
416 [ Local Array Variable Index 3]
412 [ Local Array Variable Index 2]
408 [ Local Array Variable Index 1]
404 [ Local Array Variable Index 0]
400 [ Local Integer Value        ]
```

As you can see, if you index your array to `MyArray[4]` or `MyArray[5]`, you overwrite critical values that may cause you to trap! Overwriting the previous base pointer may not harm if the calling function does not use it anymore. However, overwriting the return address will most definitely cause a trap! This is why you must always be careful to stay within your array boundaries when dealing with local variables. You could overwrite other local variables, the return address, the parameters, or just about anything!

Windows 2003

Windows 2003 has a new method to attempt to prevent buffer overflows. This can be compiled in VS.NET using the GS flags. A random value is generated as a cookie on startup of the application. The cookie is then XOR'd with the return address of the function and placed on the stack after the base pointer. This is a simple example:

```
[Return Address           ]
[Previous Base Pointer    ]
[Cookie XOR Return Address ]
```

Upon return, the cookie is checked against the return value. If they're unchanged, then the return occurs, if not, then we have a problem. The reason for this security is not to prevent code from trapping without proper handling, but rather to protect code from executing injected code. A security risk is when someone finds out how to overflow a buffer with actual code and an address to that code. This will cause the program to return to and execute that code.

This URL provides the full details of this:

- MSDN

Conclusion

I have confused beginners and probably bored advanced programmers, however, it's hard to portray advanced concepts in a simple manner. I am trying my best though. If you like or dislike

these tutorials, leave me a comment. If you want these to end, let me know too!

I've probably started off too simple then got too advanced too fast. I can't help it though, programmers should study this information and supplement it with other sources to gain full knowledge on the subject. Do not take what you read or posted on message boards as concrete fact. Everyone is human, everyone errors and not one person knows everything. These sites let just about anyone post information, so always be skeptical. Let me know if you found an error. Thanks.

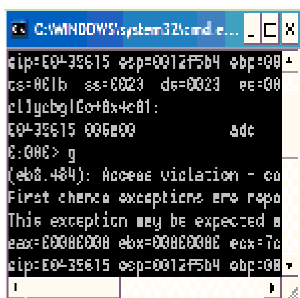
License

This article has no explicit license attached to it but may contain usage terms in the article text or the download files themselves. If in doubt please contact the author via the discussion board below.

A list of licenses authors might use can be found here

About the Author

Toby Opferman



Member

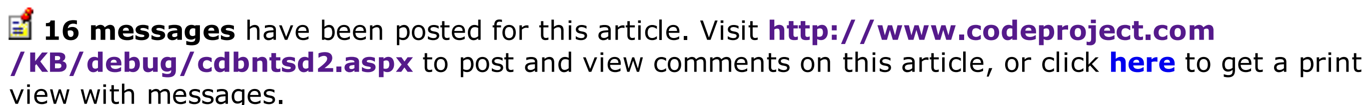
Toby Opferman has worked in just about all aspects of Windows development including applications, services and drivers.

He has also played a variety of roles professionally on a wide range of projects. This has included pure researching roles, architect roles and developer roles. He also was also solely responsible for debugging traps and blue screens for a number of years.

Previously of Citrix Systems he is very experienced in the area of Terminal Services. He currently works on Operating Systems and low level architecture at Intel.

Occupation: Engineer
Company: Intel
Location: United States

Discussions and Feedback



16 messages have been posted for this article. Visit **http://www.codeproject.com/KB/debug/cdbntsd2.aspx** to post and view comments on this article, or click **here** to get a print view with messages.

PermaLink | Privacy | Terms of Use
Last Updated: 29 Mar 2004
Editor: Chris Maunder

Copyright 2004 by Toby Opferman
Everything else Copyright © CodeProject, 1999-2009
Web20 | Advertise on the Code Project