Inside NT Internals

last updated April 13, 1997

How we figure out how NT works

Introduction

One question we often get is, "how do we know how that algorithm works?" or something similar. Most people are just curious about the techniques we use to analyze NT, but some even think that we have access to the source code. The fact is, we have no access to NT source and the knowledge we gain is the result of some hard work. This short article will describe the basics of the process we follow, and hopefully satisfy both groups of people. Unfortunately, this won't be the cookbook approach to this work that many of you may be looking for. The majority our work is intuitive in nature and very dependent on fluency with x86 assembly language and experience.

Dumpbin

The first step in the analysis process is usually deciding what exported functions are related to the NT subsystem that is being explored. This helps give us an idea of how complex the interfaces to the subsystem are, and the names of the functions can sometimes reveal interesting functionality or configuration capabilites the subsystem supports.

The easiest way to obtain this information is to use the **Dumpbin** utility that ships with Visual C++. Typing:

```
dumpbin /exports ntoskrnl.exe
```

will list all the exported functions and variables in NTOSKRNL.

If we were looking at NT scheduling, for example, we'd look for all exports that might have the words thread, process, boost, or priority in them.

NT-ICE

The next step in the process is one where serious x86 assembly language skill comes into play. First off, Bryce and I both use Soft-ICE/NT (NT-ICE) as our primary "live-step-through" tool. While Windbg would suffice, Soft-ICE is generally more responsive and more reliable for this kind of work.

NT-ICE must first be prepared by making it aware of the symbols for NTOSKRNL and any other modules that you're interested in stepping through. To do this, copy NTOSRKNL.EXE and NTOSKRNL.DBG (or NTKRNLMP.EXE and NTKRNLMP.DBG) to the same directory, load the .EXE into the Soft-ICE symbol loader, direct the loader to translate the symbols into Soft-ICE native form and load them. To have Soft-ICE always load a particular set of symbols, use the Symbols tab in the Setup dialog in the loader and point it to the .NMS files you want.

Now you must figure out what you want to step through. This is where intuition and experience come into play. For example, if you wanted to learn how the object manager works, one place where you could get a hook into its use is to set a breakpoint on **NtCreateFile**. If you're familiar with NT you'll realize that NtCreateFile interacts with the object manager to traverse the Object Manager namespace (see WinObj), create a new file object, and perform security operations on the new object. By stepping through **NtCreateFile** and the functions it calls, one can begin to get an idea of how things work.

A lot of time is typically spent following the control flow through the system multiple times. A better understanding of the way functions are inter-related is obtained iteratively. Also, I often have to go back and follow the flow several times to understand the way parameters are filtered from higher-level functions into the ones they call.

Disassembly

Most of the time stepping through functions can only provide so much information. Truly understanding the nuances of the system requires detailed analysis of specific functions, and a map of internal structure definitions. The choice of functions is usually dictated by the step-through sessions, which will highlight a certain function as being responsible for interesting behaviors.

The disassembler we use is a proprietary private one that we received from a friend. It takes .DBG files and a corresponding executable image and creates a symbolic disassembly of the image. While I don't have experience with V Communications Windows Source disassembly product with PE (Portable Executable) files, I'm sure its capabilities match those of our in-house tool.

I've chosen a documented function in this example so that you can follow along in an easier example where we actually know both the types and data structure definitions for the parameters being passed in. The function we're going to work through is **KeSetEvent**, and has the following prototype:

LONG KeSetEvent(
IN PKEVENT *Event*,
IN KPRIORITY *Increment*,
IN BOOLEAN *Wait*
);

The **KEVENT** data structure is listed in NTDDK.H.

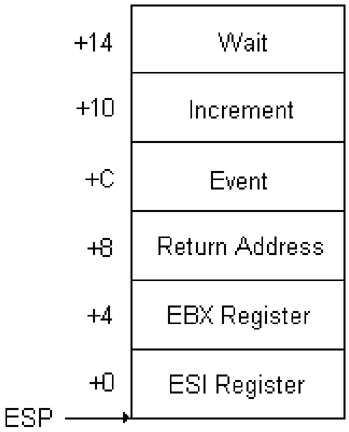
The output of the disassembler for **KeSetEvent** is shown below.

```
_KeSetEvent proc
80116388:    PUSH   EBX
80116389:    MOV    ECX,_KiDispatcherLock (801470C0)
8011638E:    PUSH   ESI
8011638F:    CALL   DWORD PTR [__imp_@KeAcquireSpinLockRaiseToSynch]
80116395:    MOV    BL,AL
80116397:    MOV    ECX,DWORD PTR [ESP+0C]
8011639B:    MOV    ESI,DWORD PTR [ECX+04]
8011639E:    MOV    DWORD PTR [ECX+04],00000001
801163A5:    TEST   ESI,ESI
801163A7:    JNE    L1 801163BC

801163A9:    MOV    EAX,DWORD PTR [ECX+08]
801163AC:    SUB    EAX,ECX
801163AE:    CMP    EAX,08
801163B1:    JE     L1 801163BC

801163B3:    MOV    EDX,DWORD PTR [ESP+10]
801163B7:    CALL   @KiWaitTest

801163BC: L1 MOV    CL,BYTE PTR [ESP+14]
801163C0:    TEST   CL,CL
801163C2:    JE     L2 801163D2

801163C4:    MOV    EAX,FS:[00000124]
801163CA:    MOV    BYTE PTR [EAX+56],CL
801163CD:    MOV    BYTE PTR [EAX+54],BL
801163D0:    JMP    L3 801163D9

801163D2: L2 MOV    ECX,EBX
801163D4:    CALL   @KiUnlockDispatcherDatabase

801163D9: L3 MOV    EAX,ESI
801163DB:    POP    ESI
801163DC:    POP    EBX
801163DD:    RET    000C
```

Now, the code above isn't *exactly* what comes out of the disassembler. You'll notice blank lines after every control transfer instruction as well as labels (L1) associated with each control transfer target. This is the first pass at figuring out the code.

Once the labels have been placed, its time to start decompiling the code. This step requires plenty of experience (and patience) in order to do it correctly and efficiently. Some things that guide procedure include knowing where the input variables are located, and the names and types of any known parameters.

In this case, the compiler did not bother to set up a stack frame (since the local variables did not spill out of the available registers), so the parameters are referenced through the stack pointer (the ESP register). The stack frame looks like the diagram below, so **KeSetEvent** references the *Event* parameter as [ESP+0xC], the *Increment* parameter as [ESP+0x10] and so on.



I decompile code progressing sequentially through the assembly language. As I go I create an intermediate form of the decompiled code. This form I call the "raw" form, and it is annotated with the labels from the assembly, as well as parameter offsets and local variable offsets (where there are some). In many cases the meaning (and names) of parameters and locals are unclear at this stage, so I just use place-holders. I label unknown parameters with *Param1*, *Param2*, etc., and reference local variables with their offset from the stack frame pointer. For example, if I see a reference to a local as [EBP-0x8], I list it as *loc8* in the raw code. Locals that are stored in registers rather than on the stack I indicate with the name of the register being used (e.g. EBX).

As I encounter control-flow statements, I determine what C-construct most likely was the source of the assembly. Lets look at the first few lines of **KeSetEvent** for an example. At address 80116397 in the code you can see the *Event* parameter pointer being moved into register ECX. In the following statement the field at offset 4 in the *Event* structure is being copied into the ESI register. Examining the *Event* structure (in NTDDK.H), you'll find that the field being referenced is *SignalState*. *SignalState* in *Event* is set to 1 (Signaled) in the next line, and the one following that performs a bit-test on the previous state against itself. Here the code is just testing the value to determine if it is 1 or 0, as indicated by the JNE instruction (Jump if Not Equal to zero).

This next part is interesting. Its pretty clear that this is an `if` expression, but the original C expression tests for the opposite of what the compiler spits out in its assembly code! This is the common case where one must reverse the assembly language logic to get back to C. Let's go ahead and look at the entire raw form of **KeSetEvent**, below, so that you can correlate what I've just said with the original disassembly. The `if` statement's test is if(!prevState), where *prevState* is a name that I gave to value held in ESI.

```
                         +c           +10                    +14
KeSetEvent( PKEVENT Event, KPRIORITY Increment, BOOLEAN Wait )
{
```

```
      prevIrql = KeAcquireSpinLockRaiseToSynch(KiDispatcherLock);

      prevState = Event->Header.SignalState; // esi
      Event->Header.SignalState = 1;
      if( !prevState ) {
           waiter = Event->Header.WaitListHead.Flink;
           waiter -= Event;
           if( waiter != 8 ) {

               edx = Increment;
               ecx = Event;
               KiWaitTest( );

           }
       }
L1
       if( Wait ) {

           thread->WaitIrql = prevIrql;
           thread->WaitMode = Wait;
       } else {
L2
           KiUnlockDispatcherDatabase( prevIrql );
       }
L3
       return prevState;
}
```

This tedious process of guessing the appropriate C structure to use, while at the same time keeping track of which register is holding which value is extremely painstaking. **KeSetEvent** is a very simple example, and took about 20 minutes to process. More complex functions where I know few structures or variables can take several hours.

Once the raw form is complete, the fun phase begins (and I'm not being sarcastic). This is the cleanup phase where I review the code I've generated and try to understand what is actually going on. Its in this phase that I'll realize what types of values are stored in parameters and local variables, and even determine their purpose. I often must go back to live step-throughs in Soft-ICE to confirm various hypothesis I have about a particular local, parameter or data structure.

I strive to make the clean code resemble actual C as much possible, and comment it so that I can come back to it later and understand what I've figured out after I've forgotten it. The clean code form of **KeSetEvent** is shown below. For this function I was able to ascertain what all the variables were, but its not unusual for me to end up with code with raw offsets or left-over untranslated names.

```
KeSetEvent( PKEVENT Event, KPRIORITY Increment, BOOLEAN Wait )
{
     KIRQL       prevIrql;
     LONG        prevState;

      //
      // Acquire the scheduler database
      //
      prevIrql = KeAcquireSpinLockRaiseToSynch(KiDispatcherLock);

      //
      // Signal the event, but remember if the event was already
      // in a signaled state
      //
      prevState = Event->Header.SignalState;
      Event->Header.SignalState = 1;

      //
      // If the event is becoming signaled, there may be waiting
      // threads to wake up.
      //
      if( !prevState && !IsListEmpty(&Event->Header.WaitListHead) ) {

            //
```

```
            // List isn't empty, so go wake up threads.
            //
            KiWaitTest( Event, Increment );
        }

        //
        // If the caller said they were going to immediately call
        // KeWaitEvent, we go ahead and set up for that.
        //
        if( Wait ) {

            thread->WaitIrql = prevIrql;
            thread->WaitMode = Wait;
        } else {

            KiUnlockDispatcherDatabase( prevIrql );
        }

        //
        // Return previous state
        //
        return prevState;
}
```

If you look at the difference between the raw form and the cleaned-up form, you see that I've replaced a list check with a macro (**IsListEmpty**) from NTDDK.H. As the number of functions we've processed has grown, so has our knowledge of internal undocumented NT data structures. This in turn has made analyzing new functions easier, as it becomes more likely that we'll know what a particular field in some structure is used for. Of course, our understanding of the way the subsystems are tied together also grows, which helps us to have an idea of what a function is likely to do before we even start looking at it closely. Nevertheless, the analsysis is highly prone to simple errors, so the code that we come up with must always be reviewed for accuracy.

Conclusion

So that's it. You now know the secrets of NT Internals. If you've read this far you can probably appreciate the kind of effort required to understand the intricate workings of code deep in NT's belly, and you can understand why the NT Internals book we're working on is taking so long (we're not analyzing everything though! Please don't send us e-mail asking when it will be done - we'll let you know here on the site).

Home