The case of a program that crashed on its first instruction

devblogs.microsoft.com/oldnewthing/20241108-00

November 8, 2024

A customer was baffled by crash reports that indicated that their program was failing on its very first instruction.

I opened one of the crash dumps, and it was so weird, the debugger couldn’t even say what went wrong.

```
ERROR: Unable to find system thread FFFFFFFF
ERROR: The thread being debugged has either exited or cannot be accessed
ERROR: Many commands will not work properly
This dump file has an exception of interest stored in it.
The stored exception information can be accessed via .ecxr.
ERROR: Exception C0000005 occurred on unknown thread FFFFFFFF
(61c.ffffffff): Access violation - code c0000005 (first/second chance not available)
0:???> r
WARNING: The debugger does not have a current process or thread
WARNING: Many commands will not work
       ^ Illegal thread error in 'r'
0:???> .ecxr
WARNING: The debugger does not have a current process or thread
WARNING: Many commands will not work
0:???>
```

Let’s see what threads we have.

```
0:???> ~
WARNING: The debugger does not have a current process or thread
WARNING: Many commands will not work
   0  Id: 61c.12b4 Suspend: 1 Teb: 000000c7`9604d000 Unfrozen
   1  Id: 61c.22d4 Suspend: 1 Teb: 000000c7`9604f000 Unfrozen
   2  Id: 61c.1ab0 Suspend: 1 Teb: 000000c7`96051000 Unfrozen
   3  Id: 61c.3308 Suspend: 1 Teb: 000000c7`96053000 Unfrozen
   4  Id: 61c.2af0 Suspend: 1 Teb: 000000c7`96055000 Unfrozen
   5  Id: 61c.2054 Suspend: 1 Teb: 000000c7`96059000 Unfrozen
0:???>
```

I wonder what they are doing.

We’ll switch to each thread just to see what instruction they are at

```
0:???> ~0s
WARNING: The debugger does not have a current process or thread
WARNING: Many commands will not work
ntdll!RtlUserThreadStart:
00007ffa`bb16df50 4883ec78        sub     rsp,78h
0:000> ~*s
         ^ Illegal thread error in '~*s'
0:000> ~1s
00000293`42074058 66894340        mov     word ptr [rbx+40h],ax
ds:00007ff6`e4600040=1f0e
0:001> ~2s
ntdll!ZwWaitForWorkViaWorkerFactory+0x14:
00007ffa`bb1b29c4 c3              ret
0:002> ~3s
ntdll!ZwWaitForWorkViaWorkerFactory+0x14:
00007ffa`bb1b29c4 c3              ret
0:003> ~4s
ntdll!ZwWaitForWorkViaWorkerFactory+0x14:
00007ffa`bb1b29c4 c3              ret
0:004> ~5s
ntdll!ZwDelayExecution+0x14:
00007ffa`bb1af3f4 c3              ret
```

The ostensible reason for the crash was an invalid write instruction, and only thread 1 is doing a write. Let’s take a closer look at what it’s trying to write to.

```
0:001> !address @rbx

Usage:                  Image
Base Address:           00007ff6`e4600000
End Address:            00007ff6`e4601000
Region Size:            00000000`00001000 (   4.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000002          PAGE_READONLY
Type:                   01000000          MEM_IMAGE
Allocation Base:        00007ff6`e4600000
Allocation Protect:     00000080          PAGE_EXECUTE_WRITECOPY
Image Path:             C:\Program Files\Contoso\ContosoDeluxe.exe
Module Name:            ContosoDeluxe
Loaded Image Name:      ContosoDeluxe.exe
Mapped Image Name:      C:\Program Files\Contoso\ContosoDeluxe.exe
More info:              lmv m ContosoDeluxe
More info:              !lmi ContosoDeluxe
More info:              ln 0x7ff6e4600000
More info:              !dh 0x7ff6e4600000

Content source: 2 (mapped), length: 400
0:001> ln @rbx
(00000000`00000000)   ContosoDeluxe!__ImageBase
```

Okay, so we are writing to the mapped image header for ContosoDeluxe itself. This is a read-only page (`PAGE_READONLY`), which is why we take a write access violation.

In fact, we’re writing into the image header, which is not something anybody normally does. This looks quite suspicious.

If we ask for stacks, we get this:

```
0:001> ~*k

   0  Id: 61c.12b4 Suspend: 1 Teb: 000000c7`9604d000 Unfrozen
Child-SP          RetAddr               Call Site
000000c7`962ffd48 00000000`00000000     ntdll!RtlUserThreadStart

   1  Id: 61c.22d4 Suspend: 1 Teb: 000000c7`9604f000 Unfrozen
Child-SP          RetAddr               Call Site
000000c7`963ff900 00007ff6`e4600000     0x00000293`42074058

   2  Id: 61c.1ab0 Suspend: 1 Teb: 000000c7`96051000 Unfrozen
Child-SP          RetAddr               Call Site
000000c7`964ff718 00007ffa`bb145a0e     ntdll!ZwWaitForWorkViaWorkerFactory+0x14
000000c7`964ff720 00007ffa`ba25244d     ntdll!TppWorkerThread+0x2ee
000000c7`964ffa00 00007ffa`bb16df78     kernel32!BaseThreadInitThunk+0x1d
000000c7`964ffa30 00000000`00000000     ntdll!RtlUserThreadStart+0x28

   3  Id: 61c.3308 Suspend: 1 Teb: 000000c7`96053000 Unfrozen
Child-SP          RetAddr               Call Site
000000c7`965ff6a8 00007ffa`bb145a0e     ntdll!ZwWaitForWorkViaWorkerFactory+0x14
000000c7`965ff6b0 00007ffa`ba25244d     ntdll!TppWorkerThread+0x2ee
000000c7`965ff990 00007ffa`bb16df78     kernel32!BaseThreadInitThunk+0x1d
000000c7`965ff9c0 00000000`00000000     ntdll!RtlUserThreadStart+0x28

   4  Id: 61c.2af0 Suspend: 1 Teb: 000000c7`96055000 Unfrozen
Child-SP          RetAddr               Call Site
000000c7`966ffad8 00007ffa`bb145a0e     ntdll!ZwWaitForWorkViaWorkerFactory+0x14
000000c7`966ffae0 00007ffa`ba25244d     ntdll!TppWorkerThread+0x2ee
000000c7`966ffdc0 00007ffa`bb16df78     kernel32!BaseThreadInitThunk+0x1d
000000c7`966ffdf0 00000000`00000000     ntdll!RtlUserThreadStart+0x28

   5  Id: 61c.2054 Suspend: 1 Teb: 000000c7`96059000 Unfrozen
Child-SP          RetAddr               Call Site
000000c7`968ffcb8 00007ffa`bb165833     ntdll!ZwDelayExecution+0x14
000000c7`968ffcc0 00007ffa`b88f9fcd     ntdll!RtlDelayExecution+0x43
000000c7`968ffcf0 00000293`420a1efd     KERNELBASE!SleepEx+0x7d
000000c7`968ffd70 00000000`00000000     0x00000293`420a1efd
```

Thread 1 is the suspicious thread that committed the access violation.

There's another suspicious thread, thread 5, which is in a `SleepEx` call called from the same suspicious source `0x00000293`420xxxxx`. This other thread is probably waiting for something to happen, so let's take a look at it.

First, let's see what kind of memory we are executing from.

```
0:001> !address 00000293`420a1ee0

Usage:                  <unknown>
Base Address:           00000293`420a0000
End Address:            00000293`420ca000
Region Size:            00000000`0002a000 ( 168.000 kB)
State:                  00001000          MEM_COMMIT
Protect:                00000040          PAGE_EXECUTE_READWRITE
Type:                   00020000          MEM_PRIVATE
Allocation Base:        00000293`420a0000
Allocation Protect:     00000040          PAGE_EXECUTE_READWRITE
```

Yikes, `PAGE_EXECUTE_READWRITE`. That's not a good sign. That smells like malicious code injection, because it is highly unusual for normal code to be read-write. But let's hold out hope that maybe there's a legitimate explanation for all of this, and it's just a matter of finding it.

Let's see what code we are executing.

```
00000293`420a1ed9 add     rsp,30h
00000293`420a1edd pop     rdi
00000293`420a1ede ret
00000293`420a1edf int     3
00000293`420a1ee0 push    rbx
00000293`420a1ee2 sub     rsp,20h
00000293`420a1ee6 call    00000293`420a13e0
00000293`420a1eeb mov     qword ptr [00000293`420c0c78],rax
00000293`420a1ef2 mov     ecx,3E8h
00000293`420a1ef7 call    qword ptr [00000293`420b4028]
                  ^^^^^^^^ YOU ARE HERE
00000293`420a1efd call    00000293`420a13e0 // do it again
00000293`420a1f02 mov     rdx,rax
00000293`420a1f05 mov     rbx,rax
00000293`420a1f08 call    00000293`420a19d0
00000293`420a1f0d test    eax,eax
00000293`420a1f0f jne     00000293`420a1f22
00000293`420a1f11 mov     rax,qword ptr [00000293`420c0c78]
00000293`420a1f18 mov     qword ptr [00000293`420c0c78],rbx
00000293`420a1f1f mov     rbx,rax
00000293`420a1f22 mov     rcx,rbx
00000293`420a1f25 call    00000293`420a17f0
00000293`420a1f2a jmp     00000293`420a1ef2
```

The first few instructions, up to the `int 3` appear to be the end of the previous function, so we can start our analysis at the `push rbx`.

```
    push rbx                          ; preserve register
    sub rsp, 20h                      ; stack frame
    call 00000293`420a13e0            ; mystery function 1
    mov  [00000293`420c0c78],rax      ; save answer in global

00000293`420a1ef2:
    mov  ecx, 3E8h                    ; decimal 1000
    call [00000293`420b4028]          ; mystery function 2
    ^^^^^^^^ YOU ARE HERE

    call 00000293`420a13e0            ; mystery function 1
    mov  rdx, rax                     ; return value becomes param1
    mov  rbx, rax                     ; save return value in rbx
    call 00000293`420a19d0            ; mystery function 3
    test eax,eax                      ; Q: did it succeed?
    jne  00000293`420a1f22            ; N: Skip
    mov  rax, [00000293`420c0c78]     ; get previous value
    mov  [00000293`420c0c78], rbx     ; replace with new value
    mov  rbx, rax                     ; save previous value in rbx

00000293`420a1f22:
    mov   rcx, rbx                    ; rcx = updated value in rbx
    call    00000293`420a17f0         ; mystery function 3
    jmp     00000293`420a1ef2         ; loop back forever
```

One thing that's apparent here is that this thread never exits. It's an infinite loop.

First, let's see if we can identify the mystery functions.

The easiest is probably mystery function 2, since it looks like a call to an imported function.

```
0:001> dps 00000293`420b4028 L1
00000293`420b4028  00007ffa`ba258370 kernel32!SleepStub
```

Aha, mystery function 2 is `Sleep`, and the call is a `Sleep(1000)`. Which we sort of knew from the stack trace but it's nice to see confirmation.

But let's look around near that address, since that may be part of a larger table of function pointers.

```
00000293`420b4000  00007ffa`baa59810 advapi32!RegCloseKeyStub
00000293`420b4008  00007ffa`baa596e0 advapi32!RegQueryInfoKeyWStub
00000293`420b4010  00007ffa`baa595a0 advapi32!RegOpenKeyExWStub
00000293`420b4018  00007ffa`baa5ab30 advapi32!RegEnumValueWStub
00000293`420b4020  00000000`00000000
00000293`420b4028  00007ffa`ba258370 kernel32!SleepStub
00000293`420b4030  00007ffa`ba250cc0 kernel32!GetLastErrorStub
00000293`420b4038  00007ffa`ba266b60 kernel32!lstrcatW
00000293`420b4040  00007ffa`ba25ff00 kernel32!CloseHandle
00000293`420b4048  00007ffa`ba254380 kernel32!CreateThreadStub
```

Bingo, this appears to be a table of imported function pointers.

Mystery function 1 seems to be called to start things off, and then again in a loop, so it seems kind of important. Let’s see what it is.

```
00000293`420a13e0 mov     qword ptr [rsp+8],rbx
00000293`420a13e5 mov     qword ptr [rsp+10h],rsi
00000293`420a13ea mov     qword ptr [rsp+18h],rdi
00000293`420a13ef push    rbp
00000293`420a13f0 mov     rbp,rsp
00000293`420a13f3 sub     rsp,80h
00000293`420a13fa mov     rax,qword ptr [00000293`420bf010]
00000293`420a1401 xor     rax,rsp
00000293`420a1404 mov     qword ptr [rbp-8],rax
00000293`420a1408 mov     ecx,40h
00000293`420a140d call    00000293`420a8478 // mystery function 3
```

This looks like a typical C function, not hand-coded assembly. After saving non-volatile registers, it builds a stack frame, and the `mov rax, [global]` followed by a `xor rax, rsp` looks a lot like a `/GS` stack canary.

So at least it’s nice that this rogue code was compiled with stack buffer overflow protection. Can’t be too careful.

Let’s look at mystery function 3.

```
00000293`420a8478
    push rbx
    sub  rsp, 20h
    mov  rbx, rcx
    jmp  00000293`420a8492

00000293`420a8483
    mov  rcx, rbx
    call 00000293`420aad50
    test eax, eax
    je   00000293`420a84a2
    mov  rcx, rbx

00000293`420a8492
    call 00000293`420aadb4
    test rax, rax
    je   00000293`420a8483
    add  rsp, 20h
    pop  rbx
    ret

00000293`420a84a2
    cmp  rbx, 0FFFFFFFFFFFFFFFFh
    je   00000293`420a84ae

    call 00000293`420a8c80
    int  3

00000293`420a84ae
    call 00000293`420a8ca0
    int  3

00000293`420a84b4
    jmp  00000293`420a8478
```

This reverse-compiles to

```
uint64_t something(uint64_t value)
{
    uint64_t p;
    while (uint64_t p = func00000293420aadb4(value); !p) {
        if (!func00000293420aad50(value)) {
            if (value == ~0ULL) {
                func00000293420a8c80();
            } else {
                func00000293420a8c80();
            }
            // NOTREACHED
        }
    }
    return p;
}
```

This seems to call a function at `func00000293420aadb4` repeatedly.

```
00000293`420aadb4 jmp     00000293`420acf8c
```

This appears to be an incremental linking thunk. So whatever this is, it looks like it was compiled in debug mode.

```
00000293`420acf8c
    push rbx
    sub  rsp, 20h
    mov  rbx,rcx
    cmp  rcx, 0FFFFFFFFFFFFFFE0h
    ja   00000293`420acfd7
    test rcx, rcx
    mov  eax, 1
    cmove rbx, rax
    jmp  00000293`420acfbe

00000293`420acfa9
    call 00000293`420b02c0
    test eax, eax
    je   00000293`420acfd7
    mov  rcx, rbx
    call 00000293`420aad50
    test eax, eax
    je   00000293`420acfd7

00000293`420acfbe
    mov  rcx, [00000293`420c07f8]
    mov  r8, rbx
    xor  edx, edx
    call [00000293`420b4298]
    test rax, rax
    je   00000293`420acfa9
    jmp  00000293`420acfe4

00000293`420acfd7
    call  00000293`420ac71c
    mov   [rax], 0Ch
    xor   eax, eax
    add   rsp, 20h
    pop   rbx
    ret
```

The initial comparison against `0xFFFFFFFF`FFFFFFFE` makes me suspect that this is `malloc()` or `operator new` because those functions begin with a check for an excessive allocation size, to avoid integer overflow.

And indeed, that's basically what this function is, as revealed by the indirect function call:

```
0:005> dps 00000293`420b4298 L1
00000293`420b4298  00007ffa`bb14cca0 ntdll!RtlAllocateHeap
```

Okay, so we found `malloc()` or `operator new`.

This will help us understand mystery function 1 a lot better.

```
00000293`420a13e0
    mov     [rsp+8], rbx
    mov     [rsp+10h], rsi
    mov     [rsp+18h], rdi
    push    rbp
    mov     rbp, rsp
    sub     rsp, 80h
    mov     rax, [00000293`420bf010]
    xor     rax, rsp
    mov     [rbp-8], rax      ; /GS canary
    mov     ecx, 40h
    call    00000293`420a8478 ; allocate 64 bytes
    xorps   xmm0, xmm0
    mov     ecx, 18h
    mov     rdi,rax           ; save first allocation
    movups  [rax],xmm0        ; zero out first allocation
    movups  [rax+10h],xmm0
    movups  [rax+20h],xmm0
    movups  [rax+30h],xmm0
    call    00000293`420a8478 ; allocate 24 bytes
    xor     esi,esi
    mov     ecx, 80h
    mov     rbx,rax           ; save second allocation
    mov     [rax+0Ch], rsi    ; zero out second allocation
    mov     [rax+14h], esi
    mov     [rax], esi
    mov     [rax+4], 10h
    mov     [rax+8], 1
    call    00000293`420a84b4 ; mystery function 4
    mov     [rbx+10h], rax    ; save result
    lea     ecx, [rsi+10h]    ; ecx = 0x10
    mov     [rdi], rbx
    call    00000293`420a8478 ; third allocation
    lea     ecx, [rsi+40h]    ; ecx = 0x40
    mov     rbx, rax
    mov     [rax+8], rsi      ; initialize third allocation
    mov     [rax], esi
    mov     [rax+4], 10h
    call    00000293`420a84b4 ; mystery function 4
    mov     [rbx+8], rax
    lea     ecx, [rsi+18h]    ; ecx = 0x18
```

Okay, so this function starts by allocating many memory blocks and initializing them.

Let's skip ahead to where it finally does something interesting.

```
    lea     rdx, [00000293`420bba90] ; LR"(SOFTWARE\systemconfig)"
    lea     rax, [rbp-50h]
    mov     [rdi+38h], rbx
    mov     r9d, 20119h       ; KEY_READ
    mov     [rsp+20h], rax
    xor     r8d, r8d
    mov     rcx,0FFFFFFFF80000002h ; HKEY_LOCAL_MACHINE
    call    qword ptr [00000293`420b4010] ; RegOpenKeyExW
    test    eax, eax
```

A `dps 00000293`420b4010` reveals that the function pointer is `RegOpenKeyExW`, so the entire function call must have been

```
RegOpenKeyExW(HKEY_LOCAL_MACHINE,
    L"SOFTWARE\\systemconfig", 0, KEY_READ, &key);
```

Further disassembly shows that if the code successfully opens the key, it tries to read some values from it. My guess is that `systemconfig` is where the code stores its state.

Okay, so maybe I can speed things up by dumping strings and seeing if there's anything that will give me a clue about the identity of this code. Recall that the `!address` command told us that the memory block was

```
0:001> !address 00000293`420a1ee0
Base Address:           00000293`420a0000
End Address:            00000293`420ca000
```

We'll ask the `!mex` debugger extension to find any strings in the memory block.

```
0:005> !mex.strings 00000293`420a0000 00000293`420ca000
...
00000293420bbd10 system
00000293420bc1d4 H:\rootkit\r77-rootkit-master\vs\x64\Release\r77-x64.pdb
```

Okay, so I guess it's malware, or at least self-identifies as a rootkit. And, hey, an Internet search for this rootkit name shows that its source code is public.

The good news for the developer is that the problem is not their fault. The bad news is that since the crash dumps are submitted anonymously, they have no way of contacting the users to tell them that they have been infected with malware.