8.04 Get Element

Now let's look at a function that has to do with looking data up in the table. Remember, it doesn't matter which functions we reverse first. I'm choosing based on what I think will be a good order to go in.

Multiple functions might get something from the table: `RtlEnumerateGenericTable`, `RtlGetElementGenericTable`, `RtlLookupElementGenericTable`, and some others. Based on the names, I think `RtlGetElementGenericTable` or `RtlLookupElementGenericTable` will be the easiest. Out of those two, I'd guess `RtlGetElementGenericTable` will be the simplest. If this function works how it sounds, it probably takes two parameters. A pointer to a Generic Table and an index to find.

Disassembly of `RtlGetElementGenericTable`:

```
00007FFD3F0DC1B0 MOV QWORD PTR SS:[RSP+0x8], RBX
00007FFD3F0DC1B5 MOV R10D, DWORD PTR DS:[RCX+0x20]
00007FFD3F0DC1B9 LEA R11D, QWORD PTR DS:[RDX+0x1]
00007FFD3F0DC1BD MOV R8, QWORD PTR DS:[RCX+0x18]
00007FFD3F0DC1C1 OR EBX, 0xFFFFFFFF
00007FFD3F0DC1C4 MOV R9D, R11D
00007FFD3F0DC1C7 CMP EDX, EBX
00007FFD3F0DC1C9 JE ntdll.7FFD3F0DC21C
00007FFD3F0DC1CB MOV EAX, DWORD PTR DS:[RCX+0x24]
00007FFD3F0DC1CE CMP R11D, EAX
00007FFD3F0DC1D1 JA ntdll.7FFD3F0DC21C
00007FFD3F0DC1D3 CMP R11D, R10D
00007FFD3F0DC1D6 JE ntdll.7FFD3F0DC200
00007FFD3F0DC1D8 JB ntdll.7FFD3F126C7A
00007FFD3F0DC1DE SUB EAX, R11D
00007FFD3F0DC1E1 MOV EDX, R11D
00007FFD3F0DC1E4 SUB EDX, R10D
00007FFD3F0DC1E7 INC EAX
00007FFD3F0DC1E9 CMP EDX, EAX
00007FFD3F0DC1EB JA ntdll.7FFD3F0DC20A
00007FFD3F0DC1ED TEST EDX, EDX
00007FFD3F0DC1EF JE ntdll.7FFD3F0DC1F8
00007FFD3F0DC1F1 MOV R8, QWORD PTR DS:[R8]
00007FFD3F0DC1F4 ADD EDX, EBX
00007FFD3F0DC1F6 JNE ntdll.7FFD3F0DC1F1
00007FFD3F0DC1F8 MOV QWORD PTR DS:[RCX+0x18], R8
00007FFD3F0DC1FC MOV DWORD PTR DS:[RCX+0x20], R11
00007FFD3F0DC200 LEA RAX, QWORD PTR DS:[R8+0x10]
00007FFD3F0DC204 MOV RBX, QWORD PTR SS:[RSP+0x8]
00007FFD3F0DC209 RET
00007FFD3F0DC20A LEA R8, QWORD PTR DS:[RCX+0x8]
00007FFD3F0DC20E TEST EAX, EAX
00007FFD3F0DC210 JE ntdll.7FFD3F0DC1F8
00007FFD3F0DC212 MOV R8, QWORD PTR DS:[R8+0x8]
00007FFD3F0DC216 ADD EAX, EBX
00007FFD3F0DC218 JE ntdll.7FFD3F0DC1F8
00007FFD3F0DC21A JMP ntdll.7FFD3F0DC212
```

```
00007FFD3F0DC21C XOR EAX, EAX
00007FFD3F0DC21E JMP ntdll.7FFD3F0DC204
```

Before we get into this function, I'm going to offer a hint, otherwise, you may get very confused. If you don't want the hint, move on. If you're not sure if you want the hint, I'll let you know that it has to do with a computer science topic. More specifically, data structures. Now for the spoiler. This function is going to deal with a splay-tree. For our purposes, we can view it as a linked list. We can figure this out by reversing, but if you don't know these topics you will be very confused. If you need to, go learn these topics before continuing.

General Overview

When a function is pretty involved, I like to start by looking at the obvious stuff, then going back through and analyzing it more closely. This will help me get a general idea of what's going on before I get into the details. The first thing we'll do is check for function parameters.

- RCX is used as a data structure (`MOV EAX, DWORD PTR DS:[RCX+0x24]`), so it's probably the table as usual.
- RDX is also used, but with an offset of +0x1 (`LEA R11D, QWORD PTR DS:[RDX+0x1]`). RDX could be a data structure, but in this case, it looks like RDX is being incremented.
- R8 and R9 are overwritten, so they're probably not parameters. We can guess that this function only takes 2 parameters. The first one being the table, the second one is only a guess, but it's probably the index to look for.

There are two jumps that are interesting, which go to `ntdll.7FFD3F0DC21C`. `ntdll.7FFD3F0DC21C` will set the return value to zero, then jump to `ntdll.7FFD3F0DC204` where RBX is restored. Finally, the function returns. This is likely a fail condition.

> If you don't already, I'd highly recommend writing notes and/or pseudo code as you are reverse engineering.

Part 1

Let's focus on the following portion of code:

```
00007FFD3F0DC1B0 MOV QWORD PTR SS:[RSP+0x8], RBX
00007FFD3F0DC1B5 MOV R10D, DWORD PTR DS:[RCX+0x20]
00007FFD3F0DC1B9 LEA R11D, QWORD PTR DS:[RDX+0x1]
00007FFD3F0DC1BD MOV R8, QWORD PTR DS:[RCX+0x18]
00007FFD3F0DC1C1 OR EBX, 0xFFFFFFFF
00007FFD3F0DC1C4 MOV R9D, R11D
00007FFD3F0DC1C7 CMP EDX, EBX
00007FFD3F0DC1C9 JE ntdll.7FFD3F0DC21C
00007FFD3F0DC1CB MOV EAX, DWORD PTR DS:[RCX+0x24]
00007FFD3F0DC1CE CMP R11D, EAX
00007FFD3F0DC1D1 JA ntdll.7FFD3F0DC21C
```

- `MOV QWORD PTR SS:[RSP+0x8], RBX` saves/preserves RBX on the stack.

- `MOV R10D, DWORD PTR DS:[RCX+0x20]` moves Table->Member5 into R10D.
- `LEA R11D, QWORD PTR DS:[RDX+0x1]` is an interesting instruction. It's essentially `R11D = RDX + 1`. This might be done instead of using `INC` to preserve the value of RDX. **Let's call R11D the AdjustedIndex**. This also tells us another piece of useful information, the index is zero-based. This means it starts from index zero, not 1, similar to a conventional array. Unlike an index, the number of elements is *not* zero-based (0 elements would mean it's empty). An array could have 3 elements, but be access via array[0], array[1], array[2].
- `MOV R8, QWORD PTR DS:[RCX+0x18]` moves Table->Member4 into R8.
- `OR EBX, 0xFFFFFFFF` sets EBX to -1.
- `MOV R9D, R11D` - R11D (AdjustedIndex) is moved into R9D. Again, this is a little weird at first, why not just use R11D? This is probably done to preserve R11D. In other words, have one version that doesn't get changed and one that does.
- `CMP EDX, EBX` compares EDX (the index to look for) to EBX (-1).
  - If the index is -1, then jump to `ntdll.7FFD3F0DC21C` which is the fail condition.
- `MOV EAX, DWORD PTR DS:[RCX+0x24]` moves Table->NumberOfElements into EAX. This reinforces the idea that Table->NumberOfElements is 4 bytes because it's referenced with an offset of +0x24 again, and it's being moved into a 4-byte register.
- `CMP R11D, EAX` - R11D (AdjustedIndex) is then checked if it's greater than EAX (NumberOfElements). This is just some more index validation.
  - If R11D is greater than EAX, then it jumps to the fail condition. `JA` being used *instead of* `JG` tells us that it's unsigned.

Here is some pseudo code of what we just looked at:

```
ULONG adjustedIndex = index + 1;
if(index == -1 || adjustedIndex > Table->NumberOfElements){
    return 0;
}
```

Part 2

Picking up where we were:

```
7FF8BBD1C1D3 CMP R11D, R10D   ; CMP AdjustedIndex, Table->Member5
7FF8BBD1C1D6 JE ntdll.7FFD3F0DC200
7FF8BBD1C1D8 JB ntdll.7FFD3F126C7A
7FF8BBD1C1DE SUB EAX, R11D    ; EAX = Table->NumberOfElements - AdjustedIndex
7FF8BBD1C1E1 MOV EDX, R11D
7FF8BBD1C1E4 SUB EDX, R10D
7FF8BBD1C1E7 INC EAX
7FF8BBD1C1E9 CMP EDX, EAX
```

Here is where things start to get interesting.

- `CMP R11D, R10D` compares AdjustedIndex to the fifth member in the table. This tells us that the fifth member in the table is likely some kind of index. Maybe it stores the last index, first index, the previously found index, or even some kind of special index. Whatever it is, we'll find out.
    - If they are equal, jump to `ntdll.7FFD3F0DC200`.
    - If AdjustedIndex is less than (JB is Jump if Below) the fifth member, jump to `ntdll.7FFD3F126C7A`.
        - We'll worry about where these jumps go later. We'll proceed as if these jumps weren't taken, which means Index+1 is greater than Table->Member5.
- `SUB EAX, R11D` - EAX contains Table->NumberOfElements. This instruction is effectively `EAX = Table->NumberOfElements - AdjustedIndex`.
- `MOV EDX, R11D` copies R11D (AdjustedIndex) into EDX. This another case of preservation.
- `SUB EDX, R10D` is the same as `EDX = AdjustedIndex - Table->Member5`.
- `INC EAX` is the same as `EAX = (Table->NumberOfElements - AdjustedIndex) + 1`.
- `CMP EDX, EAX` compares EDX (AdjustedIndex - Table->Member5) to EAX (Table->NumberOfElements - AdjustedIndex). I should note that Table->NumberOfElements - AdjustedIndex is the difference between the maximum/last index and the requested index.
- There is then a series of conditional jumps. Before we go any further, let's try to make more sense of what just happened.

To help understand what's going on, let's use some test values and see what happens. Let's say that Table->NumberOfElements is 10, and the index is 2. We don't know what Table->Member5 is yet, but you can start to make some good guesses.

```
7FF8BBD1C1D3 CMP 3, Table->Member5
7FF8BBD1C1D6 JE ntdll.7FFD3F0DC200
7FF8BBD1C1D8 JB ntdll.7FFD3F126C7A
7FF8BBD1C1DE EAX = Table->NumberOfElements - 3
7FF8BBD1C1E4 EDX = 3 - Table->Member5
7FF8BBD1C1E7 EAX++
7FF8BBD1C1E9 CMP EDX, EAX    ; CMP (3 - Table->Member5) to (Table->NumberOfElements - 3).
```

Okay, this is where we can start to predict what Member5 is. It's an index of some kind. Let's take a look at some possible scenarios.

If Member5 is the maximum index:

```
CMP (3 - 10) to (10 - 3)
```

If Member5 is the minimum index:

```
CMP (3 - 1) to (1 - 3)
```

If Member5 is the last found index (7 for example):

```
CMP (3 - 7) to (7 - 3)
```

Member5 being the minimum index wouldn't serve any good purpose. The maximum index wouldn't be very helpful either. But if it's the last index found, it could be quite helpful. For example, if the index and the last index found are the same then you can just return the last index found. If the index is greater than the last index found, you know to look above the last index found. If the index is less than the last index found, search below the last index found. This would help improve performance. We still can't be certain, but let's continue with the assumption that Member5 is the last index found.

If you look at the proceeding jumps this scenario seems even more likely.

Part 3

```
7FF8BBD1C1E9 CMP EDX, EAX   ; CMP (AdjustedIndex - Table->Member5) to (Table->NumberOfElements - AdjustedIndex).
7FF8BBD1C1EB JA ntdll.7FFD3F0DC20A
7FF8BBD1C1ED TEST EDX, EDX
7FF8BBD1C1EF JE ntdll.7FFD3F0DC1F8
7FFD3F0DC1F1 MOV R8, QWORD PTR DS:[R8]
7FF8BBD1C1F4 ADD EDX, EBX
7FF8BBD1C1F6 JNE ntdll.7FFD3F0DC1F1
7FF8BBD1C1F8 MOV QWORD PTR DS:[RCX+0x18], R8
7FF8BBD1C1FC MOV DWORD PTR DS:[RCX+0x20], R11D
7FF8BBD1C200 LEA RAX, QWORD PTR DS:[R8+0x10]
7FFD3F0DC204 MOV RBX, QWORD PTR SS:[RSP+0x8]
7FF8BBD1C209 RET
7FF8BBD1C20A LEA R8, QWORD PTR DS:[RCX+0x8]
7FF8BBD1C20E TEST EAX, EAX
7FF8BBD1C210 JE ntdll.7FFD3F0DC1F8
7FFD3F0DC212 MOV R8, QWORD PTR DS:[R8+0x8]
7FF8BBD1C216 ADD EAX, EBX
7FF8BBD1C218 JE ntdll.7FFD3F0DC1F8
7FF8BBD1C21A JMP ntdll.7FFD3F0DC212
7FF8BBD1C21C XOR EAX, EAX
7FF8BBD1C21E JMP ntdll.7FFD3F0DC204
```

- `JA ntdll.7FFD3F0DC20A` - After the comparison, if the result is that EDX is greater than EAX, it will jump to `ntdll.7FFD3F0DC20A`.
    - Let's follow that jump:
        - `LEA R8, QWORD PTR DS:[RCX+0x8]` sets R8 to the address of Table->Member2.
        - `TEST EAX, EAX` will test if EAX is zero. EAX is Table->NumberOfElements - AdjustedIndex. Again, the difference between the last index and the currently requested index.
        - If EAX *is* zero, then jump to `ntdll.7FFD3F0DC1F8`. If EAX is zero, that means Table->NumberOfElements - AdjustedIndex = 0. This means that the requested element was the first element.

    - `MOV QWORD PTR DS:[RCX+0x18], R8` will move R8 (the address of Table->Member2) into Table->Member4.
    - `MOV DWORD PTR DS:[RCX+0x20], R11D` will move AdjustedIndex into Table->Member5. This is setting Member5 to the AdjustedIndex, which is also the index we just found. This tells us that Member5 is almost certainly the last element/index found. **Until we are proven wrong, let's declare Table->Member5 as Table->LastIndexFound.**
    - `LEA RAX, QWORD PTR DS:[R8+0x10]` will load RAX with the address of R8+0x10. Remember, R8 is the address of Table->Member2. Before we continue, let's do some thinking. We can be pretty certain that the three pointers, Table->Member2, Table->Member3, and Table->Member4, are all related somehow. We haven't seen Table->Member3 used, but if Member2 and 4 are related, then Member3 probably is too. With that said, take a look at the offset of +0x10. Since R8 is already Table->Member2, the offset of +0x10 is interesting. If these pointers point to another data structure that also has three pointers that are related, this offset would conveniently skip over them. It's common to see some data structures with a header, and they will often hide or exclude their header from the programmer. Once again, we can't be certain about it, but it's something to note.
    - `MOV RBX, QWORD PTR SS:[RSP+0x8]` restores RBX.
    - `RET`
    - It seems like we're dealing with a series of pointers such as a list or tree of some kind. This function returns what seems to be a pointer to another data structure with a similar set of three pointers. Remember the purpose of this function is to get an index. Maybe this table is part of a linked list of some kind or has a linked list inside of it. If you view the function this way, it seems like that's the case. So that's what we'll go with for now. Also, let's take a look at Member4 and how it's used. It seems to be the last element found, because in this portion of code that what it's set to before it returns. **Let's declare Table->Member4 as Table->LastElementFound.** To be clear, Table->LastElementFound is a pointer to the last element found, and Table->LastIndexFound is the number of the last index found.
- If EAX is *not* zero, then it will continue.
    - `MOV R8, QWORD PTR DS:[R8+0x8]` moves what's pointed to by R8+0x8 into R8. This is very interesting.
    - `ADD EAX, EBX` will add EAX (Table->NumberOfElements - AdjustedIndex) and EBX (-1). This is the same as subtracting 1. If you didn't already know, the ADD instruction will set various flags that can be used by comparison instructions.
    - `JE ntdll.7FFD3F0DC1F8`. We've already reversed the code at `ntdll.7FFD3F0DC1F8` when we followed the `JE` after `TEST EAX, EAX`. `ntdll.7FFD3F0DC1F8` seems to result in a successful return.
    - `JMP ntdll.7FFD3F0DC212` If the previous jump is not taken, there is an unconditional jump to `ntdll.7FFD3F0DC212`. That is the beginning of the little section we are currently reversing. So this is a loop. It's counting down from the difference between the last element and the requested index. This also confirms that we're working with some kind of linked list. Remember the instruction `MOV R8, QWORD PTR DS:[R8+0x8]` which moves what R8+0x8 points to into R8. If this

unconditional jump is taken, that instruction is run again. It's following a chain of pointers however many times is required to reach the requested index.

Let's go back to the beginning of this code chunk, to `CMP EDX, EAX`. We just found out what happens if AdjustedIndex - Table->LastIndexFound is greater than Table->NumberOfElements - AdjustedIndex. In other words, that's what happens if the requested index is closer to the end of the list than the last index found. Now let's find out what happens if the requested index is closer to the last index found.

- `TEST EDX, EDX` will test if EDX (AdjustedIndex - Table->LastIndexFound) is zero. And if it is, jump to `ntdll.7FFD3F0DC1F8`.
- `JE ntdll.7FFD3F0DC1F8` - If they're equal, then just return the last index found. We already reversed this code, so we'll move on.
- `MOV R8, QWORD PTR DS:[R8]` This is a similar situation to before! It's moving what's pointed to by R8 into R8. This will go down a pointer chain. Note that this is a different pointer chain than the one we found before, which was R8+0x8. This is indicative of Flink and Blink in a linked list.
- `ADD EDX, EBX` adds AdjustedIndex - Table->LastIndexFound and -1. This decrements the value.
- `JNE ntdll.7FFD3F0DC1F1` if they aren't equal, keep going!

So if Flink and Blink are being used, which one is being used in this case? If that code is ran then AdjustedIndex - LastFoundIndex < NumberOfElements - AdjustedIndex. In other words, the requested index is closer to LastFoundIndex than the last index. You can use logic or just plug numbers into the equation and you'll find that this must be a Flink.

AdjustedIndex(5) - LastFoundIndex(3) < NumberOfElements(10) - AdjustedIndex(5) 5 - 3 < 10 - 5 2 < 5

To get to 5 you would have to Flink.

So at this point, we know what's going on. The pointers are part of a linked list. This part of the function is just one of many conditions determining what the most efficient method of getting the index is.

Part 4

Now it's time go way back. Earlier we skipped two jumps, so let's revisit them.

```
7FF8BBD1C1D3 CMP R11D, R10D          ; R11D = AdjustedIndex, R10D = Table-
>LastIndexFound
7FF8BBD1C1D6 JE ntdll.7FFD3F0DC200    ; THESE JUMPS
7FF8BBD1C1D8 JB ntdll.7FFD3F126C7A    ; THESE JUMPS
7FF8BBD1C1DE SUB EAX, R11D            ; EAX = Table->NumberOfElements
7FF8BBD1C1E1 MOV EDX, R11D
7FF8BBD1C1E4 SUB EDX, R10D
7FF8BBD1C1E7 INC EAX
7FF8BBD1C1E9 CMP EDX, EAX
```

Let's follow `JE ntdll.7FFD3F0DC200`:

```
7FF8BBD1C200 LEA RAX, QWORD PTR DS:[R8+0x10]
7FFD3F0DC204 MOV RBX, QWORD PTR SS:[RSP+0x8]
7FF8BBD1C209 RET
```

We've already reversed this code. R8+0x10 (R8 is Table->Member2) is probably skipping a header of some kind. After skipping the header, it can return a pointer to whatever data structure it is. It them restores RBX and returns.

Since Table->Member2 is likely a pointer to the head of the linked list, let's call it that. Table->Member2 is now Table->LLHead.

Now follow `JB ntdll.7FFD3F126C7A`:

```
00007FFD3F126C7A MOV EAX, R10D
00007FFD3F126C7D SHR EAX, 0x1
00007FFD3F126C7F CMP R11D, EAX
00007FFD3F126C82 JBE ntdll.7FFD3F126C9B
00007FFD3F126C84 SUB R10D, R11D
00007FFD3F126C87 JE ntdll.7FFD3F0DC1F8
00007FFD3F126C8D MOV R8, QWORD PTR DS:[R8+0x8]
00007FFD3F126C91 ADD R10D, EBX
00007FFD3F126C94 JNE ntdll.7FFD3F126C8D
00007FFD3F126C96 JMP ntdll.7FFD3F0DC1F8
00007FFD3F126C9B LEA R8, QWORD PTR DS:[RCX+0x8]
00007FFD3F126C9F TEST R11D, R11D
00007FFD3F126CA2 JE ntdll.7FFD3F0DC1F8
00007FFD3F126CA8 MOV R8, QWORD PTR DS:[R8]
00007FFD3F126CAB ADD R9D, EBX
00007FFD3F126CAE JNE ntdll.7FFD3F126CA8
00007FFD3F126CB0 JMP ntdll.7FFD3F0DC1F8
```

- `MOV EAX, R10D` moves Table->LastIndexFound into EAX.
- `SHR EAX, 0x1` shifts EAX one byte to the right. This is the same as dividing it by 2. 00110010 becomes 00011001.
- `CMP R11D, EAX` compares AdjustedIndex to Table->LastIndexFound/2.
    - If AdjustedIndex <= Table->LastIndexFound/2 then jump to `ntdll.7FFD3F126C9B`. This is, again, trying to find the most efficient way to get the index. It's checking if it's in the first or second half of the range. **Let's follow that jump.**
        - `LEA R8, QWORD PTR DS:[RCX+0x8]`. Loads the address of Table->LLHead into R8.
        - `TEST R11D, R11D` checks if R11D (AdjustedIndex) is zero.
            - `JE ntdll.7FFD3F0DC1F8` - If R11D is zero, jump to `ntdll.7FFD3F0DC1F8`. This is the portion of code which is the successful return. It sets Table->LastElementFound to the found element, and Table->LastIndexFound to the index number just looked up.
        - If it's not equal, we run into the same scenario as before. It's going to loop following a pointer chain until it gets to the index and then returns that result.

- If AdjustedIndex > Table->LastIndexFound/2 then:
    - `SUB R10D, R11D` will subtract AdjustedIndex from Table->LastIndexFound.
    - `JE ntdll.7FFD3F0DC1F8` - If they are equal, jump to the successful return portion of code.
    - `MOV R8, QWORD PTR DS:[R8+0x8]` - If they aren't equal, loop to get the element.

So now we know what's going on. There are a series of loops that will all get the element at the given index. Which loop is ran is determined based on what is the most efficient method. The function will set both Table->LastIndexFound and Table->LastElementFound. It will return a pointer to the element found.

```
To help make more sense of what's going on, here is a commented version of the
assembly code:
; Save RBX:
00007FFD3F0DC1B0 MOV QWORD PTR SS:[RSP+0x8], RBX
; R10 = LastFoundIndex:
00007FFD3F0DC1B5 MOV R10D, DWORD PTR DS:[RCX+0x20]
; R11 = AdjIndex:
00007FFD3F0DC1B9 LEA R11D, QWORD PTR DS:[RDX+0x1]
; R8 = LastElementFound:
00007FFD3F0DC1BD MOV R8, QWORD PTR DS:[RCX+0x18]
; EBX = -1:
00007FFD3F0DC1C1 OR EBX, 0xFFFFFFFF
; R9 = R11 (AdjIndex):
00007FFD3F0DC1C4 MOV R9D, R11D
00007FFD3F0DC1C7 CMP EDX, EBX
; RET 0 if Index == -1:
00007FFD3F0DC1C9 JE ntdll.7FFD3F0DC21C
; EAX = NumOfElements:
00007FFD3F0DC1CB MOV EAX, DWORD PTR DS:[RCX+0x24]
; CMP AdjIndex, NumOfElements:
00007FFD3F0DC1CE CMP R11D, EAX
; RET 0 if AdjIndex > NumOfElements
00007FFD3F0DC1D1 JA ntdll.7FFD3F0DC21C
; CMP AdjIndex, LastFoundIndex:
00007FFD3F0DC1D3 CMP R11D, R10D
; AdjIndex == LastFoundIndex?:
00007FFD3F0DC1D6 JE ntdll.7FFD3F0DC200
; AdjIndex < LastFoundIndex? Otherwise we know it's above, so find out if it's
closer to last found or end:
00007FFD3F0DC1D8 JB ntdll.7FFD3F126C7A
; EAX = NumOfElements - AdjIndex:
00007FFD3F0DC1DE SUB EAX, R11D
00007FFD3F0DC1E1 MOV EDX, R11D
; EDX = AdjIndex - LastFoundIndex:
00007FFD3F0DC1E4 SUB EDX, R10D
; (NumOfElements-AdjIndex) + 1:
00007FFD3F0DC1E7 INC EAX
; CMP (AdjIndex - LastFoundIndex), (NumberOfElements-AdjIndex):
00007FFD3F0DC1E9 CMP EDX, EAX
; If AdjIndex-LastFoundIndex > NumberOfElements-AdjIndex (Find out if it's closer
to the LastFound than the end):
```

```
00007FFD3F0DC1EB JA ntdll.7FFD3F0DC20A
00007FFD3F0DC1ED TEST EDX, EDX
00007FFD3F0DC1EF JE ntdll.7FFD3F0DC1F8
00007FFD3F0DC1F1 MOV R8, QWORD PTR DS:[R8]
00007FFD3F0DC1F4 ADD EDX, EBX
00007FFD3F0DC1F6 JNE ntdll.7FFD3F0DC1F1
; LastElementFound = R8
00007FFD3F0DC1F8 MOV QWORD PTR DS:[RCX+0x18], R8
; Member5 = (new) AdjIndex
00007FFD3F0DC1FC MOV DWORD PTR DS:[RCX+0x20], R11D
; RAX = &R8+10 (skip pointers)
00007FFD3F0DC200 LEA RAX, QWORD PTR DS:[R8+0x10]
; Restore RBX:
00007FFD3F0DC204 MOV RBX, QWORD PTR SS:[RSP+0x8]
00007FFD3F0DC209 RET
; R8 = &RCX->LLHead:
00007FFD3F0DC20A LEA R8, QWORD PTR DS:[RCX+0x8]
00007FFD3F0DC20E TEST EAX, EAXEAX == 0?
; JE to Success:
00007FFD3F0DC210 JE ntdll.7FFD3F0DC1F8
; Loop to get element:
00007FFD3F0DC212 MOV R8, QWORD PTR DS:[R8+0x8]
00007FFD3F0DC216 ADD EAX, EBX
00007FFD3F0DC218 JE ntdll.7FFD3F0DC1F8
00007FFD3F0DC21A JMP ntdll.7FFD3F0DC212
00007FFD3F0DC21C XOR EAX, EAX
00007FFD3F0DC21E JMP ntdll.7FFD3F0DC204
```

Second section at `ntdll.7FFD3F126C7A`:

```
; EAX = LastFoundIndex:
00007FFD3F126C7A MOV EAX, R10D
; EAX = LastFoundIndex/2:
00007FFD3F126C7D SHR EAX, 0x1
; CMP AdjIndex, LastFoundIndex/2:
00007FFD3F126C7F CMP R11D, EAX
; AdjIndex <= LastFoundIndex/2?:
00007FFD3F126C82 JBE ntdll.7FFD3F126C9B
; LastFoundIndex = LastFoundIndex - AdjIndex:
00007FFD3F126C84 SUB R10D, R11D
; LastFoundIndex - AdjIndex == 0? Then the last element found is result.:
00007FFD3F126C87 JE ntdll.7FFD3F0DC1F8
; AdjIndex > LastFoundIndex/2, loop until found:
00007FFD3F126C8D MOV R8, QWORD PTR DS:[R8+0x8]
00007FFD3F126C91 ADD R10D, EBX
00007FFD3F126C94 JNE ntdll.7FFD3F126C8D
00007FFD3F126C96 JMP ntdll.7FFD3F0DC1F8
; R8 = &Table->LLHead:
00007FFD3F126C9B LEA R8, QWORD PTR DS:[RCX+0x8]
00007FFD3F126C9F TEST R11D, R11D
; Is R11D 0?:
00007FFD3F126CA2 JE ntdll.7FFD3F0DC1F8
```

```
; R8 = [Table->LLHead] (also R8 = [R8]):
00007FFD3F126CA8 MOV R8, QWORD PTR DS:[R8]
; R9 = AdjIndex - 1:
00007FFD3F126CAB ADD R9D, EBX
00007FFD3F126CAE JNE ntdll.7FFD3F126CA8
00007FFD3F126CB0 JMP ntdll.7FFD3F0DC1F8
```

Pseudo-Code:

```
struct Table{
    QWORD Member1;  //Nonzero when the table has elements.
    LIST_ENTRY* LLHead;
    LIST_ENTRY* UnknownEntry;
    LIST_ENTRY* LastElementFound;
    ULONG LastIndexFound;
    ULONG NumberOfElements;
    QWORD Member7;
    QWORD Member8;
    QWORD Member9;
};

void* MyRtlGetElementGenericTable(Table* table, ULONG index)
{
    // Init Vars:
    ULONG numberOfElements = table->NumberOfElements;
    LIST_ENTRY *elementFound = table->LastElementFound;
    ULONG lastIndexFound = table->LastIndexFound;
    ULONG adjustedIndex = index + 1;

    // Validate index:
    if (index == -1 || adjustedIndex > numberOfElements){
        return 0;
    }

    // Don't find element if it's the last element found.
    if (adjustedIndex != lastIndexFound)
    {
        // If our element isn't lastElementFound, find it:
        if (lastIndexFound > adjustedIndex)
        {
            // Find which way to search:
            ULONG halfWay = lastIndexFound / 2;
            if (adjustedIndex > halfWay)
            {
                // Start at lastElementFound and move backwards towards the
beginning:
                ULONG elementsToGo = lastIndexFound - adjustedIndex;
                while(elementsToGo--){
                    elementFound = elementFound->Blink;
                }
            }
            else
            {
                // Start at the beginning and move forward:
                ULONG elementsToGo = adjustedIndex;
                elementFound = (LIST_ENTRY *) &table->LLHead;

                while(elementsToGo--){
                    elementFound = elementFound->Flink;
                }
            }
        }
        // Index is greater than the lastIndexFound:
        else
        {
            ULONG elementsToLastFound = adjustedIndex - lastIndexFound;
            ULONG elementsToEnd = numberOfElements - adjustedIndex + 1;

            // Check if the index is closer to the lastIndexFound or the end of
the list:
            if (elementsToLastFound <= elementsToEnd)
            {
                // The requested index is closer to the last element found.
                // Search the list forward starting at lastElementFound:
                while (elementsToLastFound--){
                    elementFound = elementFound->Flink;
                }
            }
            // The element is close to the end of the list than the last element
found:
            else
            {
                // Start at the list head and search backwards:
                elementFound = (LIST_ENTRY *) &table->LLHead;
                while (elementsToEnd--){
                    elementFound = elementFound->Blink;
                }
            }
        }
        // Save the element and index found into the table:
        table->LastElementFound = elementFound;
        table->LastElementIndex = adjustedIndex;
    }

    // Skip the header and return the found element:
    return elementFound + 0x10;
}
```

Phew... that was not nearly as easy as the others. If you managed to wrap your head around that, you're on your way to greatness! Re-reading might be helpful, and of course, I'm open to questions!

<- Previous Lesson
Next Lesson -> - WIP

Chapter Home