

6.05 Print Array

There are two functions called `PrintArray`. Here are the symbols for both functions:

1. `?PrintArray@@YAXQEADH@Z`

- `void __cdecl PrintArray(char * __ptr64 const,int)`
- Parameters:
 1. `const char*`
 2. `int`

2. `?PrintArray@@YAXQEAHH@Z`

- `void __cdecl PrintArray(int * __ptr64 const,int)`
- Parameters:
 1. `const int*`
 2. `int`

It's good to know that they both take pointers because this lines up with their names. If they are supposed to print an array, they will need the address of the start of the array. We aren't sure what the second parameter is. The only thing I could think of is that it's either the size of the array or how many elements of the array to print. One piece of reversing advice you should know is to always remember the big picture. If the name of this function is accurate, then it should print out the elements in an array. Remember this because it can help when problem-solving or making educated guesses.

Lies and Deception

When looking at the exports, the `PrintArray` functions are declared as `cdecl` with `__cdecl`. So apparently these functions use the `cdecl` calling convention. If you look at the disassembly of the functions you can see that they use `EDX` right at the beginning. This is extremely suspicious. Why would a register be used if the calling convention is `cdecl`? This is `cdecl`... right? Nope. This is actually still `fastcall`. How can we identify that this is `fastcall`?

- `Test EDX, EDX` - This makes sure that `EDX` is not zero. `EDX` is the second parameter passed to a function when using `fastcall`.
- `MOV RSI, RCX` - `RCX` is moved into `RSI` but `RCX` hasn't been initialized in our current scope. `RCX` is the first parameter in `fastcall`.

`x64dbg` does actually tell you that it's `fastcall`. `x64dbg` allows you to view the parameters passed to a function and it allows you to view the parameters with different calling conventions. Typically you can choose what calling convention is being used, however, `x64dbg` is only allowing us to use `fastcall` which is another strong indicator (although not a guarantee) that we are dealing with `fastcall`. You can see the parameters and calling convention under the registers window.

```

Hide FPU
RAX 00007FFA467D8EF4 <dll.EntryPoint>
RBX 000000007FFE0385
RCX 00007FFA467D0000 dll.00007FFA467D0000
RDX 0000000000000001
RBP 00000000004FF448
RSP 00000000004FF148
RSI 0000000000000001
RDI 000000007FFE0384

R8 0000000000000000
R9 00007FFA58D9FA50 <ntdll._guard_dispatch_icall_nop>
R10 000000007FFE0384
R11 0000000000000246 L'g'
R12 00007FFA467D8EF4 <dll.EntryPoint>
R13 0000000000000001
R14 00007FFA467D0000 dll.00007FFA467D0000
R15 0000000000000000

RIP 00007FFA467D8EF4 <dll.EntryPoint>

RFLAGS 0000000000000244
ZF 1 PF 1 AF 0
OF 0 SF 0 DF 0
CF 0 TF 0 IF 1

LastError 00000000 (ERROR_SUCCESS)
LastStatus C0150008 (STATUS_SXS_KEY_NOT_FOUND)

GS 002B FS 0053
ES 002B DS 002B
CS 0033 SS 002B

ST(0) 0000000000000000 x87r0 Empty 0.0000000000000000
ST(1) 0000000000000000 x87r1 Empty 0.0000000000000000
ST(2) 0000000000000000 x87r2 Empty 0.0000000000000000
ST(3) 0000000000000000 x87r3 Empty 0.0000000000000000
ST(4) 0000000000000000 x87r4 Empty 0.0000000000000000
ST(5) 0000000000000000 x87r5 Empty 0.0000000000000000
ST(6) 0000000000000000 x87r6 Empty 0.0000000000000000
ST(7) 0000000000000000 x87r7 Empty 0.0000000000000000

x87TagWord FFFF
x87TW 0 3 (Empty) x87TW 1 3 (Empty)

Default (x64 fastcall) | 5 | Unlocked
1: rcx 00007FFA467D0000 dll.00007FFA467D0000
2: rdx 0000000000000001
3: r8 0000000000000000
4: r9 00007FFA58D9FA50 <ntdll._guard_dispatch_icall_nop>
5: [rsp+28] 0000000000000000

```

You can also change how many parameters this panel is showing.

We can be extremely confident that it's using fastcall. Other than the `__cdecl` prefix, there is nothing hinting this is `cdecl`.

PrintArray (Int)

We now know that the `PrintArray` functions are using fastcall. We also know what types of parameters they take. Now we just need to figure out how it works, what it does, what the parameters are, and what does it return (if anything). According to the undecorated names, these functions shouldn't return anything (void). Let's start with the `PrintArray` function that takes nothing but integers as parameters. I'm choosing to start with this one because integers are typically easier to work with.

00007FFA467D1E80	85D2	TEST EDX, EDX	?PrintArray@@YAXQEAAHH@Z
00007FFA467D1E82	7E 43	JLE d11.7FFA467D1EC7	
00007FFA467D1E84	48:897424	MOV QWORD PTR SS:[RSP + 0x10], RSI	
00007FFA467D1E89	57	PUSH RDI	
00007FFA467D1E8A	48:83EC 20	SUB RSP, 0x20	
00007FFA467D1E8E	48:895C24	MOV QWORD PTR SS:[RSP + 0x30], RBX	
00007FFA467D1E93	48:8BF1	MOV RSI, RCX	
00007FFA467D1E96	33DB	XOR EBX, EBX	
00007FFA467D1E98	48:63FA	MOVSXD RDI, EDX	
00007FFA467D1E9B	0F1F4400	NOP DWORD PTR DS:[RAX + RAX], EAX	
00007FFA467D1EA0	8B149E	MOV EDX, DWORD PTR DS:[RSI + RBX * 4]	
00007FFA467D1EA3	E8 08020000	CALL <d11.sub_7FFA467D20B0>	
00007FFA467D1EA8	48:8BC8	MOV RCX, RAX	[Arg1 = rax:EntryPoint
00007FFA467D1EAB	E8 202F0000	CALL <d11.sub_7FFA467D4DD0>	sub_7FFA467D4DD0
00007FFA467D1EB0	48:FFC3	INC RBX	
00007FFA467D1EB3	48:3BDF	CMP RBX, RDI	
00007FFA467D1EB6	7C E8	JL d11.7FFA467D1EA0	
00007FFA467D1EB8	48:8B5C24	MOV RBX, QWORD PTR SS:[RSP + 0x30]	
00007FFA467D1EBD	48:8B7424	MOV RSI, QWORD PTR SS:[RSP + 0x38]	
00007FFA467D1EC2	48:83C4 20	ADD RSP, 0x20	
00007FFA467D1EC6	5F	POP RDI	
00007FFA467D1EC7	C3	RET	

- The first two lines are making sure EDX is not zero with `TEST EDX, EDX`. If EDX is less than or equal to zero then it jumps to `d11.7FFA467D1EC7` which just returns. The jump location is represented by the orange arrow on the far left.
- `RSI` is then moved into `RSP+0x10`. This is probably done to preserve `RSI`. `RDI` is then pushed onto the stack. This is probably done to preserve `RDI`.
- `SUB RSP, 0x20` - This is part of the function prologue which sets up the stack. I'm pointing this out because this pretty much confirms that `RSI` and `RDI` were being preserved. Preservation is almost always done in or around the prologue.
- `RBX` is probably just being saved. Just like `RSI` and `RDI`, it's considered nonvolatile in fastcall and therefore needs to be saved.
- `MOV RSI, RCX` - This seems to be the start of the "real" part of the function we are interested in. `RCX` is the first parameter. `RSI` now holds the first parameter passed to the function. If you remember from the registers chapter I mentioned that `RSI` is often used as the source pointer. This seems to be what's happening here.
- `XOR EBX, EBX` - Zeros out `EBX`.
- `MOVSXD RDI, EDX` - Moves `EDX` into `RDI`. `MOVSXD` is short for "Move double-word to quad-word with sign-extension." Sign extension refers to increasing the number of bits while also preserving the sign (positive or negative). All we really care about here is that `RDI` is now equal to whatever was in `EDX`. Remember that `EDX` is the second parameter.

Quick interjection. Here is where experience helps. The fact that `RDI` and `RSI` are being set makes me think that they will be used in some sort of loop. More likely, a loop that iterates over whatever is in `RSI`. There is no guarantee, but it seems likely considering the purpose of this function is to print everything in an array.

- `MOV EDX, DWORD PTR DS:[RSI + RBX * 4]` - Whatever is in `RSI + RBX * 4` is moved into `EDX`. `RSI` is our first parameter, which we know is an array (well, the address to the *start* of an array). `RBX` is zero right now. `RBX` is multiplied by 4 which just results in zero. For now, this seems pointless, but if you look you'll notice that this is a loop. We'll touch back on this again. Right now it's just putting the first element in the array (`RSI`) into `EDX`.
- `CALL <d11.sub_7FFA467D20B0>` - Digging into this function shows, based on our previous research, that this is most likely `std::cout`. Okay, but isn't the first parameter passed through `RCX`, not `EDX` (or `RDX`)? If we go into the function at `d11.sub_7FFA467D20B0` we can see that it doesn't actually use `RCX` as if it was passed as a parameter and just overrides it. It does, however, treat `EDX` like a parameter. I'm assuming that this is some compiler optimization. `std::cout` calls a function with `RCX` so maybe the

compiler decided to leave RCX alone when calling `std::cout` so it doesn't have to save RCX into another register. This would save a few instructions.

- `CALL <d11.sub_7FFA467D4DD0>` - Based on previous research, we can see that this is `std::endl`.
- `INC RBX` - This increments RBX by one. Remember that RBX was zero and was multiplied by 4. The result of this operation was then used as an offset into our array. Now it makes sense why RBX was zero, RBX holds the iteration of the loop. RBX is then used to access the element in the array at the index of the loop's iteration count multiplied by 4. This should make sense if you've ever written a program that printed an array. For every iteration of the loop, you access the element in the array at the loop index (iteration count) (Ex. `array[2]`, where 2 is the current iteration of the loop). The iteration of the loop is multiplied by 4 to account for the fact that an integer is 4 bytes.
- `CMP RBX, RDI` - This compares RBX to RDI. RBX holds the current iteration of the loop and RDI was set to hold the second parameter in the function. So the second parameter is compared with the loop counter. This information, combined with previous information and assumptions, makes me think that the second parameter passed is the maximum number of elements to print or the size of the array.
- `JL d11.7FFA467D1EA0` - Jump to the start of the loop.

The loop continues. It prints whatever is in `array[RSI + RBX * 4]`. I want to touch on `RBX*4`. This is actually pretty important. This is done because in Assembly you can access individual bytes. This is an array of integers, each integer is 4 bytes. The instruction `MOV EDX, DWORD PTR DS:[RSI + RBX * 4]` is moving the element in the array corresponding with the iteration of the loop into EDX. If this is the second iteration, then the iteration is 1 (this loop starts at zero). $1 * 4$ is 4. So it's access whatever is at `array+4` which, because integers are 4 bytes, is the second element. When the loop is finished the number of elements specified by the second parameter will be printed.

Finally, we have the function epilogue.

```
MOV RBX, QWORD PTR SS:[RSP + 0x30]
MOV RSI, QWORD PTR SS:[RSP + 0x38]
ADD RSP, 0x20
POP RDI
RET
```

This function doesn't appear to have any return value because RAX is never set.

PrintArray Conclusion

We've determined that `PrintArray` will print the elements of an array up to the number given in the second parameter. The function takes two parameters. The first parameter is an array, and the second parameter is how many elements to print. The second parameter could also be the size of the array, but in the end, it works the same so it doesn't really matter.

PrintArray (Char)

So now let's take a look at the `PrintArray` function that takes a `char*` as a parameter.

<pre> 00007FFA467D1ED0 85D2 TEST EAX, EAX 00007FFA467D1ED2 7E 44 JLE d11.7FFA467D1F18 00007FFA467D1ED4 48:897424 MOV QWORD PTR SS:[RSP + 0x10], RSI 00007FFA467D1ED9 57 PUSH RDI 00007FFA467D1EDA 48:83EC 20 SUB RSP, 0x20 00007FFA467D1EDE 48:895C24 MOV QWORD PTR SS:[RSP + 0x30], RBX 00007FFA467D1EE3 48:8BF1 MOV RSI, RCX 00007FFA467D1EE6 33DB XOR EBX, EBX 00007FFA467D1EE8 48:63FA MOVSDX RDI, EDI 00007FFA467D1EEB 0F1F4400 NOP DWORD PTR DS:[RAX + RAX], EAX 00007FFA467D1EF0 > 0FB61433 MOVZX EDI, BYTE PTR DS:[RBX + RSI] 00007FFA467D1EF4 E8 D72B0000 CALL <d11.sub_7FFA467D4AD0> 00007FFA467D1EF9 48:8BC8 MOV RCX, RAX 00007FFA467D1EFC E8 CF2E0000 CALL <d11.sub_7FFA467D4DD0> 00007FFA467D1F01 48:FFC3 INC RBX 00007FFA467D1F04 48:3BDF CMP RBX, RDI 00007FFA467D1F07 ^ 7C E7 JL d11.7FFA467D1EF0 00007FFA467D1F09 48:8B5C24 MOV RBX, QWORD PTR SS:[RSP + 0x30] 00007FFA467D1F0E 48:8B7424 MOV RSI, QWORD PTR SS:[RSP + 0x38] 00007FFA467D1F13 48:83C4 20 ADD RSP, 0x20 00007FFA467D1F17 5F POP RDI 00007FFA467D1F18 > C3 RET </pre>	<pre> ?PrintArray@@YAXQEADH@Z [Arg1 = rax:_D11MainCRTStartup std::endl<char,std::char_traits<char </pre>
--	---

This function is almost exactly the same as the other one. The only difference I want to point out is how the elements in the array are accessed. Because a char is only one byte, it doesn't need to access elements in the array using `[RBX + ESI*4]` (4 bytes at a time) like the integer version of `PrintArray` did. Instead, it can access elements with just `[RBX + ESI]`. `RBX` is the loop counter.

Implementing `PrintArray` In Our Own Program

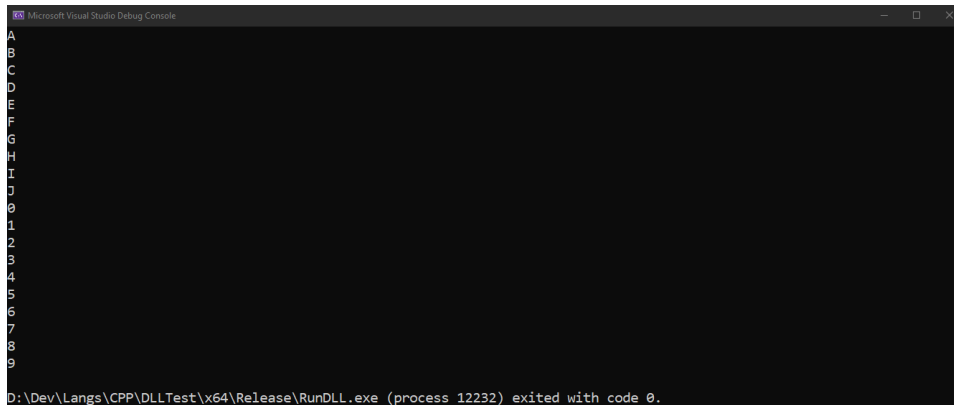
The code is pretty much the same as it was for `SayHello` just different data types.

```

1  #include <iostream>
2  #include <Windows.h>
3
4  //void PrintArray(char/int array[], int sizeofArray/ElementsToPrint);
5  typedef void(WINAPI* IPrintArray_char)(char[], int);    ///?PrintArray@@YAXQEADH@Z
6  typedef void(WINAPI* IPrintArray_int)(int[], int);      ///?PrintArray@@YAXQEADH@Z
7
8  int main()
9  {
10     HMODULE dll = LoadLibraryA("DLL.DLL"); //Load our DLL.
11
12     if (dll != NULL)
13     {
14         //Char PrintArray
15         IPrintArray_char cPrintArray = (IPrintArray_char)GetProcAddress(dll, "?PrintArray@@YAXQEADH@Z");
16         if (cPrintArray != NULL) {
17             char myArray[10] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J' };
18             cPrintArray(myArray, 10);
19         }
20         else { printf("Can't load the function."); }
21
22         //Int PrintArray
23         IPrintArray_int iPrintArray = (IPrintArray_int)GetProcAddress(dll, "?PrintArray@@YAXQEADH@Z");
24         if (iPrintArray != NULL) {
25             int myArray[10] = { 0,1,2,3,4,5,6,7,8,9 };
26             iPrintArray(myArray, 10);
27         }
28         else { printf("Can't load the function."); }
29     }
30 }
31

```

Here is the output of the function:



Final Notes

That was a good amount of work, I hope you enjoyed it. This lesson dabbled in some problem solving and I can assure you we have much more of that coming soon. One of the thrills of reversing is figuring it all out and putting the puzzle together with all the information you have gathered. You can go take a well-earned break now.

Copy/Paste Code

```
#include <iostream>
#include <Windows.h>

//void PrintArray(char/int array[], int sizeofArray/ElementsToPrint);
typedef void(WINAPI* IPrintArray_char)(char[], int);    //?PrintArray@@YAXQEADH@Z
typedef void(WINAPI* IPrintArray_int)(int[], int);      //?PrintArray@@YAXQEAHH@Z

int main()
{
    HMODULE dll = LoadLibraryA("DLL.DLL"); //Load our DLL.

    if (dll != NULL)
    {
        //Char PrintArray
        IPrintArray_char cPrintArray = (IPrintArray_char)GetProcAddress(dll, "?PrintArray@@YAXQEADH@Z");
        if (cPrintArray != NULL) {
            char myArray[10] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J' };
        };
        cPrintArray(myArray, 10);
    }
    else { printf("Can't load the function."); }

    //Int PrintArray
    IPrintArray_int iPrintArray = (IPrintArray_int)GetProcAddress(dll, "?PrintArray@@YAXQEAHH@Z");
    if (iPrintArray != NULL) {
        int myArray[10] = { 0,1,2,3,4,5,6,7,8,9 };
        iPrintArray(myArray, 10);
    }
}
```

```
    }  
    else { printf("Can't load the function."); }  
}  
}
```

[<- Previous Lesson](#)[Next Lesson ->](#)[Chapter Home](#)