Function Call

Before We Begin

- The name of the program in the following examples is "Testing.exe". This is why you will see "testing.*" prepended to some things.
- Parameters do not have to be set in any order. RCX can be set after EDX with no issue.
- Parameters don't have to be passed through the 64-bit registers. For example, you can pass a parameter through RCX, ECX, CX, etc.
- You'll see new instructions that are variants on other instructions you know. For example, MOVAPS is new, but it's just a variant on the MOV instruction. MOVAPS is used for moving single precision values around. **Single-precision values are floats, and double-precision values are doubles.** Unless I'm trying to figure out exactly what the data type of some value is, I usually ignore the variant and just focus on the main idea. In other words, I don't really care that MOVAPS works with single precision, for now I just care that it moves some data.

You should be able to follow this section without the binary, but if you want to compile your own version of the code we will be reversing here is the code:

```
#include <iostream>
int main() {
        printf("Printing with printf(). Here is some data: %d, %d, %f, %p, %d, %f,
%f\n", 4, 2, 1.0f, (void*)0xFFF7865, 1498, 87.003f, 99123.34);
        return 0;
}
```

Analyzing The Function Call

Let's take a look at a function call. In this case the function being called is `printf()`. Look back at the calling convention section if you need to.

```
00007FF773AF1070 L$ SUB RSP, 0x48
00007FF773AF1074    MOVAPS XMM0, XMMWORD PTR DS:[<__xmm@40f8333570a3d70a4055c03120000000>]
00007FF773AF107B    LEA RCX, QWORD PTR DS:[0x7FF773B09D80]
00007FF773AF1082    MOVSD XMM3, QWORD PTR DS:[<__real@3ff0000000000000>]
00007FF773AF108A    MOV EDX, 0x4
00007FF773AF108F    MOVUPS XMMWORD PTR SS:[RSP + 0x30], XMM0
00007FF773AF1094    MOV DWORD PTR SS:[RSP + 0x28], 0x5DA
00007FF773AF109C    MOVQ R9, XMM3
00007FF773AF10A1    MOV QWORD PTR SS:[RSP + 0x20], 0xFFF7865
00007FF773AF10AA    LEA R8D, QWORD PTR DS:[RDX - 0x2]
00007FF773AF10AE    CALL <testing.printf>
00007FF773AF10B3    XOR EAX, EAX
00007FF773AF10B5    ADD RSP, 0x48
00007FF773AF10B9    RET
```

When this function is ran, it prints the following to the console:
`Printing with printf(). Here is some data: 4, 2, 1.000000, 000000000FFF7865, 1498, 87.002998, 99123.340000`

Parameters Passed

We know these parameters are being passed to `printf()` because the parameters are set before `CALL <testing.printf>`.

1. The **first** parameter passed (RCX) is the memory address 0x7FF773B09D80. This is the address of the string `Printing with printf(). Here is some data: %d, %d, %f, %p, %d, %f, %f\n`. This is our format string for `printf()`. Don't forget that this format string *is* a parameter to `printf()`.
2. The **second** parameter passed (EDX) is the value of 0x4.
3. The **third** parameter (R8D) is set to `RDX - 0x2` which is 0x2.
4. The **fourth** parameter (R9) is set to XMM3. XMM3 contains the value at memory address real@3ff000... This is not a memory address, this is just a "label" that x64dbg is giving the memory address. The real address is 0x7FF773B09DC8. This address contains 1 (as a double).

At this point all other parameters are going to be passed on the stack. Remember that parameters are passed in reverse, so I will start talking about parameters that are passed towards the bottom of the Assembly code given in the example.

5. The **fifth** parameter is put onto the stack with `MOV QWORD PTR SS:[RSP + 0x20], 0xFFF7865`. This is the value of "000000000FFF7865" that gets printed.
6. The **sixth** parameter is put onto the stack with `MOV DWORD PTR SS:[RSP + 0x28], 0x5DA`. 0x5DA is hexadecimal for 1498.
7. The **seventh** and **eighth** parameters are put onto the stack at the same time with `MOVUPS XMMWORD PTR SS:[RSP + 0x30], XMM0`. Remember that the XMM registers can hold two values. XMM0 contains both 87.002998 and 99123.34.

At this point the `printf()` function is called. `printf()` will now take these parameters and use them as it needs to.

There you go, that's a function call. If you can understand what was just covered then you shouldn't have much issue understanding any other function call you encounter. Remember, if you aren't reversing 64-bit Windows the calling convention might be different.