3.5 Windows x64 Calling Convention

When a function is called you could, theoretically, pass parameters via registers, the stack, or even on disk. You just need to be sure that the function you are calling knows how you are calling it. This isn't too big of a problem if you are using your own functions, but things would get messy when you start using libraries. To solve this problem we have **calling conventions** that define how parameters are passed to a function, who allocates space for local variables, and who cleans up the stack.

> **Callee** refers to the function being called, and the **caller** is the function making the call.

There are several different calling conventions including cdecl, syscall, stdcall, fastcall, and many more. Because we are going to be reverse engineering on x64 Windows we will mostly focus on x64 fastcall. When we get into the DLL chapter we will also have to know cdecl. If you do plan on reversing on other platforms, be sure to learn the calling convention(s).

> You will usually see a double underscore prefix (__) before a calling convention's name. For example: __fastcall. I won't be doing this because it's not necessary.

Fastcall

Fastcall is *the* calling convention for x64 Windows. Windows uses a four-register fastcall calling convention by default. Fastcall pushes function parameters backward (right to left). When talking about calling conventions you will hear about something called the "Application Binary Interface" (ABI). The ABI defines various rules for programs such as calling conventions, parameter handling, and more.

How does the x64 Windows calling convention work?

- The first four arguments/parameters are passed in registers. Parameters that are *not* floating point values (floats or doubles) will be passed via RCX, RDX, R8, and R9 (in that order). Non-floating point values include pointers, integers, booleans, chars, etc. Floating-point parameters will be passed via XMM0, XMM1, XMM2, and XMM3 (in that order). Floating-point values include floats and doubles. If the parameter being passed is too big to fit in a register then it is passed by reference. Parameters are never spread across multiple registers. Any other parameters are put on the stack.

> All parameters, even ones passed through registers, have space reserved on the stack for them. Also, there is always space made for 4 parameters on the stack even if there are no parameters passed. This space isn't completely wasted because the compiler can, and often will, use it.

- The base pointer (RBP) is saved so it can be restored.
- A function's return value is passed via RAX if it's an integer, bool, char, etc., or XMM0 if it's a float or double.
- Member functions (functions that are part of a class/struct) have an implicit first parameter for the "this" pointer. Because it's a pointer it will be passed via RCX.[1]

- The *caller* is responsible for allocating space for parameters for the *callee*. The caller must always allocate space for 4 parameters even if no parameters are passed.
- The registers RAX, RCX, RDX, R8, R9, R10, R11 are considered volatile and must be considered destroyed on function calls (unless otherwise safety-provable by analysis such as whole program optimization). The registers RBX, RBP, RDI, RSI, RSP, R12, R13, R14, and R15 are considered nonvolatile and must be saved and restored by a function that uses them.[2]

Stack Access

Data on the stack such as local variables and function parameters are often accessed with RBP or RSP. On x64 it's extremely common to see RSP used instead of RBP for parameters. Remember that the first four parameters, even though they are passed via registers, still have space reserved for them on the stack. This space is going to be 32 bytes (0x20), 8 bytes for each of the 4 registers.

- 1-4 Parameters:
  - Arguments will be pushed via their respective registers. The compiler will likely use RSP+0x0 to RSP+0x18 for other purposes.
- More Than 4 Parameters:
  - The first four arguments are passed via registers, the rest are pushed onto the stack starting at offset RSP+0x20. This makes RSP+0x20 the fifth argument and RSP+0x28 the sixth.
  - Note: Arguments 1-4 are not pushed onto the stack by default, only the space for them is allocated. Sometimes when more than four arguments are passed, the first four arguments *are* put onto the stack by the callee. Be sure to look out for this.

Here is a very simple example where the numbers 1 to 8 are passed from one function to another function. Notice the order they are put in.

```
MOV DWORD PTR SS:[RSP + 0x38], 0x8
MOV DWORD PTR SS:[RSP + 0x30], 0x7
MOV DWORD PTR SS:[RSP + 0x28], 0x6
MOV DWORD PTR SS:[RSP + 0x20], 0x5
MOV R9D, 0x4
MOV R8D, 0x3
MOV EDX, 0x2
MOV ECX, 0x1
CALL function
```

As you can see, the first few parameters are passed via ECX (because it doesn't need the full RCX register), EDX, R8D, and R9D as usual. It passes the rest of the parameters through RSP+0x20, RSP+0x28, etc.

Further Exploration

That's the x64 Windows fastcall calling convention for you. Learning your first calling convention is like learning your first programming language. It seems complex and daunting at first, but it's really quite simple. It's typically harder to learn your first calling convention than it is your second or third.

If you want to learn more about this calling convention you can here:
https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019
https://docs.microsoft.com/en-us/cpp/build/x64-software-conventions?view=vs-2019

> If this lesson was confusing, read through 3.3 Instructions.md then re-read this lesson. I apologize for this but there really isn't a good order to teach this stuff in since it all goes together.

cdecl (C Declaration)

- The parameters are passed on the *stack* backward (right to left).
- The base pointer (RBP) is saved so it can be restored.
- The return value is passed via EAX.
- The caller cleans the stack. This is what makes cdecl special. Because the caller cleans the stack, cdecl allows for a variable number of parameters.

That wasn't too hard, was it? Cdecl is pretty easy and this is your second calling convention so I can exclude *many* details.

<- Previous Lesson
Next Lesson ->

Chapter Home

Sources

- https://docs.microsoft.com/en-us/cpp/build/x64-software-conventions?view=vs-2019
- https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019
- https://docs.microsoft.com/en-us/cpp/build/prolog-and-epilog?view=vs-2019
- https://www.gamasutra.com/view/news/171088/x64_ABI_Intro_to_the_Windows_x64_calling_convention.php