You can’t copy code with memcpy; code is more complicated than that

devblogs.microsoft.com/oldnewthing/20211229-00

December 29, 2021

Raymond Chen

Back in the day, a customer reported that their program crashed on Itanium.

Wait, come back!

Itanium is where the customer recognized the problem, but it applies to all other architectures, so stick with me.

Their code went roughly like this:

```
struct REMOTE_THREAD_INFO
{
    int data1;
    int data2;
    int data3;
};

static DWORD CALLBACK RemoteThreadProc(REMOTE_THREAD_INFO* info)
{
    try {
        ... use the info to do something ...
    } catch (...) {
        ... ignore all exceptions ...
    }
    return 0;
}
static void EndOfRemoteThreadProc()
{
}

// Error checking elided for expository purposes
void DoSomethingCrazy()
{
    // Calculate the number of code bytes.
    SIZE_T functionSize = (BYTE*)EndOfRemoteThreadProc - (BYTE*)RemoteThreadProc;

    // Allocate memory in the remote process
    SIZE_T allocSize = sizeof(REMOTE_THREAD_INFO) + functionSize;
    REMOTE_THREAD_INFO* buffer = (REMOTE_THREAD_INFO*)
      VirtualAllocEx(targetProcess, NULL, allocSize, MEM_COMMIT,
        PAGE_EXECUTE_READWRITE);

    // Write data to the remote process
    REMOTE_THREAD_INFO localInfo = { ... };
    WriteProcessMemory(targetProcess, buffer,
                       &localInfo, sizeof(localInfo));

    // Write code to the remote process
    WriteProcessMemory(targetProcess, buffer + 1,
                       (void*)RemoteThreadProc, functionSize);

    // Execute it!
    CreateRemoteThread(targetProcess, NULL, 0,
                       (LPTHREAD_START_ROUTINE)(buffer + 1),
                       buffer);
}
```

This code is such a bad idea, I’ve intentionally introduced errors so it won’t even compile.

The idea is that they want to inject some code into a target process, so they use `Virtual-Alloc` to allocate some memory in that process. The first part of the memory block contains some data that they want to pass. The second part of the memory block contains the code bytes that they want to execute, and they tell `CreateRemoteThread` execution at those code bytes.

I'm just going to say it right now: The entire idea that went into this code is fundamentally flawed.

The customer reported that this code "worked just fine on 32-bit x86 and 64-bit x86", but it doesn't work on Itanium.

Actually, I'm surprised that it worked even on x86!

The design assumes that all of the code in `RemoteThreadProc` is position-independent. There is no requirement that generated code be position-independent. For example, one code generation option for `switch` statements is to use a jump table, and that jump table consists of absolute addresses on x86.

In fact, it's clear that the code *isn't* position-independent, because it's using C++ exception handling, and the Microsoft compiler's implementation of exception handling involves a table that maps points of execution to `catch` statements, so that it knows which `catch` statement to use. And if they had used a filtered `catch`, then there would be additional tables for deciding whether the `catch` filter applies to the exception that was thrown.

The design also assumes that the code contains no references to anything outside the function body itself. All of the jump tables and lookup tables used by the function need to be copied to the target process, and the code assumes that those tables are also between the labels `EndOfRemoteThreadProc` `RemoteThreadProc`.

Indeed, we know that there will be references to content outside the function body itself, because the C++ try/catch block will call into functions in the C runtime support library.

Both x86-64 and Itanium use unwind codes for exception handling, and there was no attempt to register those unwind codes in the target process.

My guess is that they were lucky and no exceptions were thrown, or at least they were thrown infrequently enough that it eluded their testing.

There is also no guarantee that `EndOfRemoteThreadProc` will be placed directly after `RemoteThreadProc` in memory. Indeed, there's not even a guarantee that `EndOfRemoteThreadProc` will have an independent existent. The linker may perform COMDAT folding, which causes multiple identical functions to be combined into one. Even if you disable COMDAT folding, Profile-Guided Optimization will move the functions independently, and they are unlikely to end up in the same place.

Indeed, there's no requirement that the code bytes for the `RemoteThreadProc` function be contiguous at all! Profile-Guided Optimization will rearrange basic blocks, and the code for a single function may end up scattered across different parts of the program, depending on their usage patterns.

Even without Profile-Guided Optimization, compile-time optimization may inline some or all of a function, so a single function might have multiple copies in memory, each of which has been optimized for its specific call site.

There are also some Itanium-specific rules that ensure abject failure on Itanium.

On Itanium, all instructions must be aligned on 16-byte boundaries, but the above code does not respect that. Also, on Itanium, function pointers point not to the first code byte, but to a descriptor structure that contains a pair of pointers, one to the functions `gp`, and the other to the first byte of code. (This is the same pattern used by PowerPC.)

I pointed out to the customer liaison that what the customer is trying to do is very suspicious and looks like a virus. The customer liaison explained that it's quite the opposite: The customer is a major anti-virus software vendor! The customer has important functionality in their product that that they have built based on this technique of remote code injection, and they cannot afford to give it up at this point.

Okay, now I'm scared.

A safer[1] way to inject code into a process is to load the code as a library, via `LoadLibrary`. This invokes the loader, which will do the work of applying fixups as necessary, allocating all the memory in the appropriate way, with the correct alignment, registering control flow guard and exception unwind tables, loading dependent libraries, and generally getting the execution environment set up properly to run the desired code.

We never heard back from the customer.

[1] I didn't say it was a *safe* way to inject code. Just that it was *safer*.

Raymond Chen

Follow