Is there a problem with CreateRemoteThread on 64-bit systems?

devblogs.microsoft.com/oldnewthing/20180615-00

June 15, 2018

Raymond Chen

Back in the days when it was still fashionable to talk about the Itanium, a customer reported that the `CreateRemoteThread` function didn't work. The customer explained that any attempt to call the `CreateRemoteThread` function results in the target process being terminated. When they attempt to create a remote thread in Explorer, then the Explorer process crashes. When they attempt to create a remote thread in `lsass.exe`, `lsass.exe` process crashes, and the system restarts. They included a sample program that demonstrated the problem.

```
// Code in italics is wrong.  In fact, this is so wrong
// I've intentionally introduced compiler errors so you
// can't possibly use it in production.
struct UsefulInfo {
    int thing1;
    int thing2;
};

DWORD RemoteThreadProc(void* lpParameter)
{
  UsefulInfo* info =
    reinterpret_cast<UsefulInfo*>(lpParameter);

  blah blah blah
  try {
    blah blah blah
  } catch (...) {
    blah blah blah
  }
  return 0;
}

// This symbol lets us find the end of the RemoteThread function.
static void EndOfRemoteThreadProc() { }

// Error checking removed for simplicity of exposition.
void InjectTheThread(
    UsefulInfo* info,
    HANDLE targetProcess)
{
  // Calculate the size of the function.
  DWORD codeSize = (DWORD)EndOfRemoteThreadProc - (DWORD)RemoteThreadProc;

  // Allocate an executable buffer in the target process.
  BYTE* codeBuffer = VirtualAllocEx(targetProcess,
                     codeSize + sizeof(*info),
                     PAGE_EXECUTE_READWRITE);

 // Copy the useful information to the target process
 WriteProcessMemory(targetProcess, codeBuffer,
                    info, sizeof(*info));

 // Followed by the code
 WriteProcessMemory(targetProcess, codeBuffer + sizeof(*info),
                    (void*)RemoteThreadProc, codeSize);

 // Execute it and pass a pointer to the useful information.
 CreateRemoteThread(targetProcess,
                    codeBuffer + sizeof(*info), codeBuffer);
}
```

There is so much wrong with this code it's hard to say where to start.

There's no guarantee that all the code in the `RemoteThreadProc` function is contiguous. The compiler might choose to spread it out into multiple chunks, possibly based on Profile-Guided Optimizations.

Similarly, there is no guarantee that the `EndOfRemoteThreadProc` function will be placed immediately after `RemoteThreadProc` function in memory.

There is no guarantee that the code in the `RemoteThreadProc` function is position-independent.

There is no guarantee that the code in the `RemoteThreadProc` function is self-contained. There may be supporting data in the read-only data segment, such as jump tables for switch statements.

The `RemoteThreadProc` function uses C++ exception handling, but the code didn't inject the C runtime support library or fix up the references to the runtime library.

The code didn't register any exception tables for the dynamically-generated code. x86 is the only architecture that does not require explicit exception vector registration. Everybody else uses table-based exception handling.

Now some ia64-specific remarks.

Function pointers on ia64 don't point to the first byte of code, so subtracting function pointers doesn't give you any information about the size of the function (whatever that means), and copying data starting at the function pointer does not actually copy any code.

Conversely, when you take a pointer to a block of memory that contains code and treat it as a function pointer, you are actually causing the first two 8-byte values at that address to be interpreted as a global pointer and a code address. This results in a garbage global pointer, and code executing from a random location.

The copied code doesn't start at a multiple of 16. Code on ia64 must be 16-byte-aligned.

In general the `CreateRemoteThread` function requires deep knowledge of the machine architecture. Its intended audience was debugggers, which are already well-versed in the details of the machine architecture.

We encouraged the customer to avoid the `CreateRemoteThread` function entirely. In particular, using it with critical system processes like `lsass.exe` is a serious issue for system reliability. Faults in that process can bring the whole system down (as the customer observed), or cause other strange behavior like damaging parts of the security infrastructure, which will lead to hard-to-debug authentication problems at best and full-fledged security vulnerabilities at worst. And the system may in the future take stronger steps to prevent code injection and data tampering in critical system processes, so a design based on `Create-`

`RemoteThread` is living on borrowed time. It's not clear what the customer is trying to do, but they should investigate whether there are supported extensibility mechanisms that give them what they want.

The customer replied that their product contains important functionality that they have constructed out of the `CreateRemoteThread` function, and they cannot afford to abandon it at this point.

Customers like this scare me.

(The customer liaison never revealed the name of the customer, but I did learn that they develop anti-malware software. So now I'm even more scared. Fortunately, fixing this code to work on Itanium became a moot issue, but I still worry about their x64 version, because many of the issues here also apply to x64.)

Raymond Chen

Follow