Spectre Mitigations in Microsoft's C/C++ Compiler

Paul Kocher
February 13, 2018

Introduction

Microsoft announced support in the Visual C/C++ compiler for mitigating the conditional branch variant of the Spectre attack (aka "variant 1" in Jann Horn's post).

This form of Spectre can be used to attack a broad range of software -- operating systems, device drivers, web APIs, database systems, and almost anything else that receives untrusted input and may run on the same computer as untrusted code. Because large programs can contain literally millions of conditional branches and a lot of legacy software needs to be updated, automated tools to add protections are essential.

The countermeasure approach being used is conceptually straightforward but challenging in practice. Intel has redefined the LFENCE instruction as stopping speculative execution. (Although this post focuses on x86, ARM has defined an instruction named CSDB that works similarly.) If an LFENCE instruction is placed before every vulnerable conditional branch destination, this variant of Spectre is fully addressed.

It's important that no exploitable paths get missed. An attacker needs only a single vulnerable code pattern, which can be anywhere in a process's address space (including libraries and other code that have nothing to do with security). It doesn't help much to lock some doors while leaving others wide open. Likewise, speculation barriers only work if they are inserted in all necessary locations.

Inserting an LFENCE on every path leaving a conditional jump would be effective and conceptually simple, but unfortunately the performance cost would be substantial. For example, I ran an experiment where I took a simple SHA-256 implementation and manually added LFENCEs around the conditional jumps in the main loop. Performance on my Haswell-based laptop fell from 94857 iterations/sec to 38476 iterations/sec., a decrease of 59.4 percent. Some other operations would likely have an even greater performance impact, since in my test the entire compression function was LFENCE-free.

Microsoft's compiler attempts to reduce the performance impact by using the compiler's static analyzer to select where to insert LFENCE instructions. Although Microsoft's post lacks any quantified performance data, I was surprised by the statement that Microsoft has "built all of Windows with /Qspectre enabled and did not notice any performance regressions of concern".

Results

To see how well Microsoft's compiler implementation works, I wrote several Spectre-vulnerable source code examples and compiled them using Microsoft's 64-bit C/C++ compiler version 19.13.26029 with the Spectre mitigation enabled. I then looked at the resulting assembly language listings.

At first, I thought I had the wrong version of the compiler, since I wasn't seeing any LFENCEs. Finally, I tried compiling my example code from the Appendix of the Spectre paper, and saw LFENCEs appearing. Still, even small variations in the source code resulted in unprotected code being emitted, with no warning to developers.

The code examples below include 15 vulnerable functions. The compiler adds LFENCEs to the first two, which closely resemble the example code in the Spectre paper. The remaining 13 examples compile to unsafe output code which, if included in an application where adversaries control the input parameter x, would potentially compromise the entire application. Furthermore, my examples are far from comprehensive -- for example, they all rely on cache modification as a covert channel and they all reside in simple functions that more amenable to static analysis.

Discussion

The strictest security requirement for speculation barrier countermeasures would be to ensure that no unauthorized (e.g. out-of-bounds) memory reads occur during speculative execution. A weaker requirement would be to allow unsafe reads to occur provided that the results are only used in 'safe' operations that are 'guaranteed' to not leak information. Because future processor implementations may add new optimizations, these guarantees should ideally be architecturally defined. Unfortunately, the Microsoft compiler does not do either of these, and simply produces unsafe code when the static analyzer is unable to determine whether a code pattern will be exploitable.

While there is room for improvement, the real issue is one of approach rather than implementation. A compiler cannot reliably determine whether arbitrary sequences of instructions will be exploitable. For example, when compiling a function, the compiler often has no way to infer the properties of the parameters that will be passed when the function is called. A post-compilation analysis tool (e.g. using symbolic execution) could do somewhat a better job, but will still be imperfect since code analysis is an inherently undecidable problem.

The underlying issue is one of security versus performance. Automated tools will inevitably encounter many locations where they are uncertain whether a speculation barrier is required. Inserting LFENCEs in all these places will hurt performance, omitting them is a security risk, and alerts will drown the programmer in a sea of confusing warning messages. Microsoft's current compiler mitigation is designed to minimize the performance overhead.

I've been in touch with the Microsoft compiler team and have had an excellent conversation with them. They understand the trade-offs involved. Given the limitations of static analysis and messiness of the available Spectre mitigations, they are struggling to do what they can without significantly impacting performance. They would welcome feedback – should /Qspectre (or a different option) automatically inserts LFENCEs in all potentially non-safe code patterns, rather than the current approach of protecting known-vulnerable patterns? Would you make use of a /Qspectre variant that provides the most secure mitigation assistance -- at the cost of a significant performance loss? (The Visual Studio team can be reached online or by email.)

In my opinion, the best approach is to address the security issue fully when a developer explicitly passes a compilation flag (e.g. /Qspectre) requesting protection.

Although a there would be a performance impact at first, developers can (relatively) easily rework performance-critical routines as needed. In contrast, manually wading through the compiled code to find missing LFENCEs is entirely impractical. Speculative execution commonly 180+ instructions past a cache miss, and vulnerabilities can involve multiple functions, macros, "?:" operators, etc. It would also be enormously helpful to have a flag in output files (object files, DLLs, executables, etc.) indicating whether comprehensive LFENCE insertion was performed on the components. Static analysis still has an important role to play; instead of identifying known-bad code patterns for LFENCE insertion, its job is to identify safe code patterns where LFENCEs can be omitted. Unreliable defenses should be avoided, since even a single exploitable code pattern in an application or its libraries can leak the entire memory contents of the process to an attacker.

Conclusions

Developers and users cannot rely on Microsoft's current compiler mitigation to protect against the conditional branch variant of Spectre. Speculation barriers are only an effective defense if they are applied to all vulnerable code patterns in a process, so compiler-level mitigations need to instrument all potentially-vulnerable code patterns. Microsoft's blog post states that "there is no guarantee that all possible instances of variant 1 will be instrumented under /Qspectre". In practice, the current implementation misses many (and probably most) vulnerable code patterns, leading to unrealistically optimistic performance results as compared to robust countermeasures while creating a potentially false sense of security.

I gave Microsoft an opportunity to review this post. In addition to edits to how they can receive feedback, they asked me to highlight that the compiler cannot instrument all possible instances of variant 1 without over-inserting barriers, incurring a significant performance cost. I completely agree with this comment and made some edits to reflect this. Still, the opinions expressed here (as well as any errors) are mine.

Code Examples

```
// ---------------------------------------------------------------------------------------
// Define the types used, and specify as extern's the arrays, etc. we will access.
// Note that temp is used so that operations aren't optimized away.
//
```

```
// Compilation flags:  cl /c /d2guardspecload /O2 /Faout.asm
// Note: Per Microsoft's blog post, /d2guardspecload flag will be renamed /Qspectre
//
// This code is free under the MIT license (https://opensource.org/licenses/MIT), but
// is intentionally insecure so is only intended for testing purposes.

#include <stdlib.h>
#include <stdint.h>
extern size_t array1_size, array2_size, array_size_mask;
extern uint8_t array1[], array2[], temp;


// ----------------------------------------------------------------------------------------
// EXAMPLE 1:  This is the sample function from the Spectre paper.
//
// Comments:  The generated assembly (below) includes an LFENCE on the vulnerable code
// path, as expected

void victim_function_v01(size_t x) {
     if (x < array1_size) {
          temp &= array2[array1[x] * 512];
     }
}

//    mov     eax, DWORD PTR array1_size
//    cmp     rcx, rax
//    jae     SHORT $LN2@victim_fun
//    lfence
//    lea     rdx, OFFSET FLAT:__ImageBase
//    movzx   eax, BYTE PTR array1[rdx+rcx]
//    shl     rax, 9
//    movzx   eax, BYTE PTR array2[rax+rdx]
//    and     BYTE PTR temp, al
//  $LN2@victim_fun:
//    ret     0


// ----------------------------------------------------------------------------------------
// EXAMPLE 2:  Moving the leak to a local function that can be inlined.
//
// Comments:  Produces identical assembly to the example above (i.e. LFENCE is included)
// ----------------------------------------------------------------------------------------

void leakByteLocalFunction_v02(uint8_t k) { temp &= array2[(k)* 512]; }
void victim_function_v02(size_t x) {
     if (x < array1_size) {
          leakByteLocalFunction(array1[x]);
     }
}


// ----------------------------------------------------------------------------------------
// EXAMPLE 3:  Moving the leak to a function that cannot be inlined.
//
// Comments: Output is unsafe.  The same results occur if leakByteNoinlineFunction()
// is in another source module.

__declspec(noinline) void leakByteNoinlineFunction(uint8_t k) { temp &= array2[(k)* 512]; }
void victim_function_v03(size_t x) {
     if (x < array1_size)
          leakByteNoinlineFunction(array1[x]);
}

//    mov     eax, DWORD PTR array1_size
//    cmp     rcx, rax
//    jae     SHORT $LN2@victim_fun
//    lea     rax, OFFSET FLAT:array1
//    movzx   ecx, BYTE PTR [rax+rcx]
//    jmp     leakByteNoinlineFunction
//  $LN2@victim_fun:
//    ret     0
//
//  leakByteNoinlineFunction PROC
//    movzx   ecx, cl
//    lea     rax, OFFSET FLAT:array2
```

```
//    shl     ecx, 9
//    movzx   eax, BYTE PTR [rcx+rax]
//    and     BYTE PTR temp, al
//    ret     0
//  leakByteNoinlineFunction ENDP


// ----------------------------------------------------------------------------------------
// EXAMPLE 4:  Add a left shift by one on the index.
//
// Comments: Output is unsafe.

void victim_function_v04(size_t x) {
     if (x < array1_size)
          temp &= array2[array1[x << 1] * 512];
}

//    mov     eax, DWORD PTR array1_size
//    cmp     rcx, rax
//    jae     SHORT $LN2@victim_fun
//    lea     rdx, OFFSET FLAT:__ImageBase
//    movzx   eax, BYTE PTR array1[rdx+rcx*2]
//    shl     rax, 9
//    movzx   eax, BYTE PTR array2[rax+rdx]
//    and     BYTE PTR temp, al
//  $LN2@victim_fun:
//    ret     0


// ----------------------------------------------------------------------------------------
// EXAMPLE 5:  Use x as the initial value in a for() loop.
//
// Comments: Output is unsafe.

void victim_function_v05(size_t x) {
     size_t i;
     if (x < array1_size) {
          for (i = x - 1; i >= 0; i--)
               temp &= array2[array1[i] * 512];
     }
}

//    mov     eax, DWORD PTR array1_size
//    cmp     rcx, rax
//    jae     SHORT $LN3@victim_fun
//    movzx   edx, BYTE PTR temp
//    lea     r8, OFFSET FLAT:__ImageBase
//    lea     rax, QWORD PTR array1[r8-1]
//    add     rax, rcx
//  $LL4@victim_fun:
//    movzx   ecx, BYTE PTR [rax]
//    lea     rax, QWORD PTR [rax-1]
//    shl     rcx, 9
//    and     dl, BYTE PTR array2[rcx+r8]
//    jmp     SHORT $LL4@victim_fun
//  $LN3@victim_fun:
//    ret     0


// ----------------------------------------------------------------------------------------
// EXAMPLE 6:  Check the bounds with an AND mask, rather than "<".
//
// Comments: Output is unsafe.

void victim_function_v06(size_t x) {
     if ((x & array_size_mask) == x)
          temp &= array2[array1[x] * 512];
}

//    mov     eax, DWORD PTR array_size_mask
//    and     rax, rcx
//    cmp     rax, rcx
//    jne     SHORT $LN2@victim_fun
//    lea     rdx, OFFSET FLAT:__ImageBase
//    movzx   eax, BYTE PTR array1[rdx+rcx]
```

```
//    shl     rax, 9
//    movzx   eax, BYTE PTR array2[rax+rdx]
//    and     BYTE PTR temp, al
//  $LN2@victim_fun:
//    ret     0


// ----------------------------------------------------------------------------------------
// EXAMPLE 7:  Compare against the last known-good value.
//
// Comments: Output is unsafe.

void victim_function_v07(size_t x) {
     static size_t last_x = 0;
     if (x == last_x)
          temp &= array2[array1[x] * 512];
     if (x < array1_size)
          last_x = x;
}

//    mov     rdx, QWORD PTR ?last_x@?1??victim_function_v07@@9@9
//    cmp     rcx, rdx
//    jne     SHORT $LN2@victim_fun
//    lea     r8, OFFSET FLAT:__ImageBase
//    movzx   eax, BYTE PTR array1[r8+rcx]
//    shl     rax, 9
//    movzx   eax, BYTE PTR array2[rax+r8]
//    and     BYTE PTR temp, al
//  $LN2@victim_fun:
//    mov     eax, DWORD PTR array1_size
//    cmp     rcx, rax
//    cmovb   rdx, rcx
//    mov     QWORD PTR ?last_x@?1??victim_function_v07@@9@9, rdx
//    ret     0


// ----------------------------------------------------------------------------------------
// EXAMPLE 8:  Use a ?: operator to check bounds.

void victim_function_v08(size_t x) {
     temp &= array2[array1[x < array1_size ? (x + 1) : 0] * 512];
}

//    cmp     rcx, QWORD PTR array1_size
//    jae     SHORT $LN3@victim_fun
//    inc     rcx
//    jmp     SHORT $LN4@victim_fun
//  $LN3@victim_fun:
//    xor     ecx, ecx
//  $LN4@victim_fun:
//    lea     rdx, OFFSET FLAT:__ImageBase
//    movzx   eax, BYTE PTR array1[rcx+rdx]
//    shl     rax, 9
//    movzx   eax, BYTE PTR array2[rax+rdx]
//    and     BYTE PTR temp, al
//    ret     0



// ----------------------------------------------------------------------------------------
// EXAMPLE 9:  Use a separate value to communicate the safety check status.
//
// Comments: Output is unsafe.

void victim_function_v09(size_t x, int *x_is_safe) {
     if (*x_is_safe)
          temp &= array2[array1[x] * 512];
}

//    cmp     DWORD PTR [rdx], 0
//    je      SHORT $LN2@victim_fun
//    lea     rdx, OFFSET FLAT:__ImageBase
//    movzx   eax, BYTE PTR array1[rcx+rdx]
//    shl     rax, 9
//    movzx   eax, BYTE PTR array2[rax+rdx]
```

```
//    and     BYTE PTR temp, al
//  $LN2@victim_fun:
//    ret     0


// ----------------------------------------------------------------------------------------
// EXAMPLE 10:  Leak a comparison result.
//
// Comments: Output is unsafe.  Note that this vulnerability is a little different, namely
// the attacker is assumed to provide both x and k.  The victim code checks whether
// array1[x] == k.  If so, the victim reads from array2[0].  The attacker can try
// values for k until finding the one that causes array2[0] to get brought into the cache.

void victim_function_v10(size_t x, uint8_t k) {
     if (x < array1_size) {
          if (array1[x] == k)
               temp &= array2[0];
     }
}

//    mov     eax, DWORD PTR array1_size
//    cmp     rcx, rax
//    jae     SHORT $LN3@victim_fun
//    lea     rax, OFFSET FLAT:array1
//    cmp     BYTE PTR [rcx+rax], dl
//    jne     SHORT $LN3@victim_fun
//    movzx   eax, BYTE PTR array2
//    and     BYTE PTR temp, al
//  $LN3@victim_fun:
//    ret     0


// ----------------------------------------------------------------------------------------
// EXAMPLE 11:  Use memcmp() to read the memory for the leak.
//
// Comments: Output is unsafe.

void victim_function_v11(size_t x) {
     if (x < array1_size)
          temp = memcmp(&temp, array2 + (array1[x] * 512), 1);
}

//    mov     eax, DWORD PTR array1_size
//    cmp     rcx, rax
//    jae     SHORT $LN2@victim_fun
//    lea     rax, OFFSET FLAT:array1
//    movzx   ecx, BYTE PTR [rax+rcx]
//    lea     rax, OFFSET FLAT:array2
//    shl     rcx, 9
//    add     rcx, rax
//    movzx   eax, BYTE PTR temp
//    cmp     al, BYTE PTR [rcx]
//    jne     SHORT $LN4@victim_fun
//    xor     eax, eax
//    mov     BYTE PTR temp, al
//    ret     0
//  $LN4@victim_fun:
//    sbb     eax, eax
//    or      eax, 1
//    mov     BYTE PTR temp, al
//  $LN2@victim_fun:
//    ret     0

// ----------------------------------------------------------------------------------------
// EXAMPLE 12:  Make the index be the sum of two input parameters.
//
// Comments: Output is unsafe.

void victim_function_v12(size_t x, size_t y) {
     if ((x + y) < array1_size)
          temp &= array2[array1[x + y] * 512];
}

//    mov     eax, DWORD PTR array1_size
//    lea     r8, QWORD PTR [rcx+rdx]
```

```
//    cmp     r8, rax
//    jae     SHORT $LN2@victim_fun
//    lea     rax, QWORD PTR array1[rcx]
//    lea     r8, OFFSET FLAT:__ImageBase
//    add     rax, r8
//    movzx   ecx, BYTE PTR [rax+rdx]
//    shl     rcx, 9
//    movzx   eax, BYTE PTR array2[rcx+r8]
//    and     BYTE PTR temp, al
//  $LN2@victim_fun:
//    ret     0

// ----------------------------------------------------------------------------------------
// EXAMPLE 13:  Do the safety check into an inline function
//
// Comments: Output is unsafe.

__inline int is_x_safe(size_t x) { if (x < array1_size) return 1; return 0; }
void victim_function_v13(size_t x) {
     if (is_x_safe(x))
          temp &= array2[array1[x] * 512];
}

//    mov     eax, DWORD PTR array1_size
//    cmp     rcx, rax
//    jae     SHORT $LN2@victim_fun
//    lea     rdx, OFFSET FLAT:__ImageBase
//    movzx   eax, BYTE PTR array1[rdx+rcx]
//    shl     rax, 9
//    movzx   eax, BYTE PTR array2[rax+rdx]
//    and     BYTE PTR temp, al
//  $LN2@victim_fun:
//    ret     0


// ----------------------------------------------------------------------------------------
// EXAMPLE 14:  Invert the low bits of x
//
// Comments: Output is unsafe.

void victim_function_v14(size_t x) {
     if (x < array1_size)
          temp &= array2[array1[x ^ 255] * 512];
}

//    mov     eax, DWORD PTR array1_size
//    cmp     rcx, rax
//    jae     SHORT $LN2@victim_fun
//    xor     rcx, 255                    ; 000000ffH
//    lea     rdx, OFFSET FLAT:__ImageBase
//    movzx   eax, BYTE PTR array1[rcx+rdx]
//    shl     rax, 9
//    movzx   eax, BYTE PTR array2[rax+rdx]
//    and     BYTE PTR temp, al
//  $LN2@victim_fun:
//    ret     0


// ----------------------------------------------------------------------------------------
// EXAMPLE 15:  Pass a pointer to the length
//
// Comments: Output is unsafe.

void victim_function_v15(size_t *x) {
     if (*x < array1_size)
          temp &= array2[array1[*x] * 512];
}

//    mov     rax, QWORD PTR [rcx]
//    cmp     rax, QWORD PTR array1_size
//    jae     SHORT $LN2@victim_fun
//    lea     rcx, OFFSET FLAT:__ImageBase
//    movzx   eax, BYTE PTR array1[rax+rcx]
//    shl     rax, 9
//    movzx   eax, BYTE PTR array2[rax+rcx]
```

```
//    and     BYTE PTR temp, al
//  $LN2@victim_fun:
//    ret     0
```

[Home page]