Mitigation Overview for Potential Side-Channel Cache Exploits in Linux*

White Paper

January, 2018

Intel is collaborating with the Linux* kernel community and industry partners to help mitigate potential side-channel cache exploits. This document can help those in the industry make informed decisions regarding various Linux kernel mitigations if they are not already consuming them from other sources like Linux distributions and cloud providers.

To find the latest security updates on these issues, including the latest status on microcode, see the security advisory.

Document Number: 337034-001

Contents

Revision History

Date	Revision	Description
January, 2018	1.0	Initial release.

Note: Releases in the table are listed in reverse order so that the latest/newest is in the top row.

§

1.0 Background

As detailed by Google Project Zero, three new side-channel analysis methods were discovered that potentially facilitate access to unauthorized information. These methods rely on a property of modern processors called speculative execution. Despite this commonality, the methods for mitigating these three "variants" are very different. As a result, we discuss the mitigations separately.

Anyone consuming security mitigations must be prepared to consume mitigations that will continually be morphed and improved over time as opposed to consuming them at any single point in time. Linux distributions and stable kernel trees both provide these streams of mitigations.

Because Linux is continually improving, the latest upstream kernel is expected to be the most hardened against all security vulnerabilities.

§

2.0 Bounds Check Bypass (Spectre Variant 1)

This bounds check bypass (Spectre variant 1) exploit targets a processor's "conditional branch instructions." The operating system uses conditional branches for processing and validating virtually all data, including untrusted data processed by the kernel. Even code appearing "correct" can potentially be exploited to gain unauthorized access to information.

To mitigate this exploit technique in the Linux kernel, we first identify instruction sequences that can be tricked into exploitable behavior. Since a sequence must fit a specific pattern and operate on untrusted data before any potential exploit, not all conditional branches are exploitable. Techniques such as static analysis or manual inspection can identify these sequences.

The Google Project Zero exploit for the bounds check bypass mitigation (Spectre variant 1) targets a very specific kernel feature called eBPF (enhanced Berkeley Packet Filter). When targeting this feature, an attacker actively inserts vulnerable instruction sequences into the kernel, which can then be leveraged into a full exploit. The mitigations for eBPF are essential for mitigating the bounds check bypass (Spectre variant 1) since this exploit technique is publicly documented extensively.

Pointers supplied by applications are common sources of untrusted kernel data and have always been validated to help ensure security and stability. Mitigations that target the paths validating pointers are currently under discussion in the community and are expected to provide significant additional mitigation against bounds check bypass exploits.

As additional manual inspection and static analysis is performed, the list of vulnerable code will be refined and patched if possible.

3.0 Branch Target Injection (Spectre Variant 2)

The branch target injection (Spectre variant 2) exploit targets a processor's "indirect branch predictor." Indirect branches are used very differently than the conditional branches that the first exploit may target. In Linux, indirect branches are used relatively rarely compared to conditional branches, but they are used in critical locations. In addition, the compiler may insert indirect branches without the programmer ever being aware.

Since the compiler generates these branches, mitigation against this exploit is the most straightforward when the compiler can simply avoid generating vulnerable branch sequences. A software construct called retpoline can be deployed to help ensure that a given indirect branch is resistant to exploitation. The compiler can insert retpoline sequences, or the programmer can insert them manually. Retpoline sequences deliberately steer the processor's branch prediction logic to a trusted location, preventing a potential exploit from steering them elsewhere.

> *As an example:* It's like the gutter on a bowling lane: either the ball hits the pins or it goes to a controlled place (the gutter). It will not end up in the next lane.

Mitigation with retpoline sequences requires obtaining a compiler with retpoline support, as well as a kernel that has been updated to support retpoline sequences. The kernel build then enables special compiler options instructing the compiler to automatically generate retpoline sequences instead of vulnerable indirect branches. In addition to retpoline sequences inserted by the compiler, assembly code making indirect calls has also been manually updated to avoid exploitable indirect calls. This is generally accomplished using retpoline sequences directly from assembly.

Processors based on Broadwell and later microarchitectures may consult the indirect branch predictor when returning from function calls. On these processors, maximum mitigation with retpoline sequences requires following Intel's microcode guidance.

See the security advisory for more details. Additional mitigations avoiding use of the indirect branch predictor by "ret" are described in detail in a white paper and include the use of Supervisor Mode Execution Protection (SMEP) and manipulation of the Return Stack Buffer (RSB).

In addition to the software-based retpoline mitigation, Intel is providing a microcode-based solution using indirect branch control mitigations described in the Speculative Execution Side Channel Mitigations white paper. Although some vendors are pursuing mitigations entirely based on these techniques, the Linux community has chosen to proceed with the retpoline approach.

4.0 Rogue Data Cache Load (Meltdown Variant 3)

The rogue data cache load (Meltdown variant 3) exploit targets a processor's speculative data loading mechanisms. Even though the processor's access control may protect a piece of data, it may still be read speculatively before an illegal access is determined to exist. The first two exploits require manipulating the kernel to do something *for* an attacker, while this exploit occurs entirely within code that is under the control of an attacker. This means we cannot modify the kernel to mitigate this exploit--we must fundamentally change where kernel data is available.

The mitigation for this is conceptually very simple: instead of relying on a processor's access-control mechanisms to protect data, simply remove the data instead.

> *As an example:* If you want to protect your laptop from thieves, you can either place it in your car's trunk (rely on access-control) or you can take it with you (remove the data). The latter is more effective.

In Linux, this mitigation is referred to as kernel Page Table Isolation (PTI). This mitigation removes the data from the reach of exploits by having the kernel maintain two independent copies of the hardware page tables. One copy contains the minimal set of data and code needed to run an application and enter or exit the kernel, but it does not contain valuable kernel data. This helps ensure no valuable data can be *leaked* by an exploit running outside the kernel. The other set of page tables, active only while the kernel is running, contains everything needed for the kernel to function normally, including its private data.

§

5.0 Mitigation Availability

- Check our website for the latest status on both kernel and compiler mitigations.
- See details on kernels with mitigations.
- See also distribution compilers containing retpoline support.

§