Load Value Injection in the Line Fill Buffers: How to Hijack Control Flow without Spectre

Andrei LUȚAȘ (vlutas@bitdefender.com), Dan LUȚAȘ (dlutas@bitdefender.com)

Abstract

In 2018, two new types of microarchitectural side-channel attacks were disclosed: Meltdown and Spectre. Meltdown allows an attacker to speculatively access memory that is inaccessible, while Spectre allows an attacker to alter the branch prediction structures in order to gain speculative arbitrary code execution. In 2019, another class of microarchitectural side-channel attacks was disclosed: Microarchitectural Data Sampling, or MDS. This attack allows an attacker to pick-up in-flight data from various microarchitectural data structures (line fill buffers or LFBs - MFBDS, load ports - MLPDS or store buffers - MSBDS). Until now, MDS and MFBDS in particular have been viewed in only one direction; the victim loads the secret data inside the LFBs, while the attacker leaks the secret data by issuing a load instruction which requires a microcode assist (for example, by accessing an invalid address). In this whitepaper, we present a different view of MFBDS whereby the attacker stores rogue data inside the LFBs, and the victim unwillingly uses this data during speculative execution.

This new technique is called Load Value Injection, and the CVE assigned to it is CVE-2020-0551. The most straightforward example that we've come-up with for this scenario is LVI based control flow hijacking. The attacker sprays the LFBs with the address of a malicious function, and when the victim issues an indirect branch through memory which requires a microcode assist (for example, the page containing the destination branch address is swapped out), the address of the malicious function is loaded from the LFBs, thus leading to the attacker function being speculatively executed.

Introduction

In recent years, several researchers have discovered and disclosed a series of vulnerabilities named microarchitectural side channel attacks. A side channel attack relies on careful measurements made by an attacker to determine the value of a secret located inside the victim memory (which is normally inaccessible to the attacker). The initial "wave" of side-channel attacks includes Meltdown [1] and Spectre [2]. Meltdown abuses the fact that the CPU continues to speculatively execute instructions beyond an instruction that triggered a fault (for example, a page-fault). Instructions following such a faulting instructions may be fed by the out-of-order core with data that would otherwise be inaccessible (for example, because it is located inside kernel memory). Spectre abuses the branch-prediction unit of modern CPUs to fool it into either accessing buffer data beyond a limit or speculatively executing a malicious function. L1TF [3] exploits the fact that the CPU may speculatively access a virtual address even if a valid translation does not exist for it. MDS and TAA [4] [5] [6] allow an attacker to access in flight data (from line fill buffers, store buffers or load ports) being used by a victim, and SWAPGS [7] is a particular example of Spectre, where a critical system instruction gets executed speculatively when not needed.

In this whitepaper, we disclose a new variant of MDS: Load Value Injection in the Line Fill Buffers, or LVI-LFB for short. Researchers have previously looked at MDS from one direction only; the victim accesses the secret, which gets loaded in the MDS buffers, while the attacker leaks the contents of the MDS buffers by issuing a load instruction which requires microcode assists (for example, by reading an invalid address). However, the MDS buffers can also be abused the other way around; if an attacker sprays the MDS buffers with a particular value, a victim may speculatively load that particular value when a load instruction triggers such a microcode assist (for example, the load instruction triggers a fault). By carefully analyzing what Spectre is and what MDS is, a keen eye will quickly identify the root cause of the new vulnerability; an indirect memory branch which requires a microcode assist being fed stale values from the MDS buffers which can be controlled by an attacker, thus leading to speculative arbitrary code execution.

Recap – Meltdown, Spectre & MDS

Meltdown

Memory accesses are subject to CPU access rights checks. Whenever an invalid access is encountered, the CPU would signal this using a fault which will be handled by the operating system. The CPU is capable of executing instructions out of order, meaning that a younger instruction may finish execution earlier than an older instruction that was fetched & decoded first. However, the results of the instructions are committed in program order (instruction retirement). On CPUs which are vulnerable to Meltdown, a load instruction that triggers a memory fault (for example, by accessing a kernel address from user-mode) may forward the result of the load to a younger dependent instruction, even if this instruction will never retire – the architectural state is not affected, but the caches are – the attacker can make fine access measurements to various memory locations to determine if they have been cached or not. Consider the following example:

```
movzx   eax, byte [rdx]     ; [1]
shl     eax, 12             ; [2]
mov     al, [rcx + rax]     ; [3]
```

Assume that **rdx** contains a kernel address, and that the code is executed in user-mode. Instruction [1] will try to access that kernel address, which will result in a page-fault being generated. On CPUs affected by Meltdown, instruction [1] would also load the actual kernel value into the **eax** register – note that normally, this does not matter, as any modification made by instructions following [1] will be discarded (they will not be committed to the architectural state) due to the fault being generated, but the address cached by instruction [3] will remain, allowing an attacker to obtain the kernel value by measuring which offset inside the **rcx** buffer has been cached.

Spectre

Generally, indirect branches use a structure called the Branch Target Buffer to predict the destination address of an indirect branch. If the branch target address is mispredicted, the pipeline is flushed, and the CPU waits for the actual target address to be computed. If, for example, the target address lies in memory, the CPU would wait for that memory load to finish, and then it would start executing the code located at the address indicated by the loaded value. Consider the following example:

```
fnptr   dq      TargetFunction  ; [1]

    ...
    call    [rel fnptr]         ; [2]
    ...

TargetFunction:                 ; [3]
    ...
```

In this example, line [1] defines a function pointer which points to the **TargetFunction**. Instruction at line [2] issues an indirect **call** to the value located at address **fnptr**. The CPU attempts to predict the address instruction [2] tries to branch at, but eventually it has to load the actual memory value and check if the target was predicted correctly or not. If the target was not predicted correctly, the pipeline will be flushed, and the CPU starts executing the correct function, **TargetFunction**, located at line 3.

With Spectre v2 (Branch Target Injection), an attacker may pollute the Branch Target Buffer with the address of a malicious function. This way, the CPU encounters instruction [2] and predicts that the target address of the branch is the malicious function the attacker inserted inside the BTB. It starts executing the malicious function speculatively, and at some point, when the value of the **fnptr** has been loaded from memory and the branch misprediction is detected, the CPU discards the modifications made by the malicious function and will start executing the correct function. However, not all modifications made by the malicious function are properly discarded. Memory addresses that have been cached by the CPU remain cached, and the attacker can measure the access time to these memory locations to determine, for example, the value of a secret.

Microarchitectural Data Sampling

Microarchitectural Data Sampling or MDS for short is a group of vulnerabilities which allow an attacker to obtain the value of in-flight data from several internal CPU buffers (line fill buffers, store buffers and load ports). Consider the instruction **mov rax, [0]**. This instruction accesses an address which is usually invalid. When accessing such an address, the CPU detects that it is invalid, and a fault is generated. In some cases, however, the CPU may forward stale data from within the MDS buffers, which can then be obtained by the attacker. The following code sequence is a general way of obtaining such data:

```
movzx   eax, byte [0]       ; [1]
shl     eax, 12             ; [2]
mov     al, [rcx + rax]     ; [3]
```

Instruction at line [1] accesses an invalid address, which leads to a fault being generated. However, the CPU continues executing instructions speculatively even after instruction [1], until this instruction is retired, and the fault delivered. Up until this point, this looks like Meltdown, except instruction [1] does not access an address which is valid, but inaccessible to the attacker. Instead, the instruction accesses a completely invalid address. Due to the MDS bug, the CPU forwards stale data from the line fill buffers into the **rax** register, and instruction [2] shifts this byte value 12 positions left, whereas instruction [3] would access the obtained offset into the buffer pointed by **rcx**. Once this gadget finishes executing speculatively, the attacker can measure which offset inside the **rcx** buffer was cached. If offset 0x0 was cached, it means that instruction [1] loaded an in-flight value of 0, whereas if offset 0xBD000 was cached, it means that an in-flight value of 0xBD was loaded.

Load Value Injection in the Line Fill Buffers

Now that we understand MDS, it is clear that the classic use-case involved an active victim which accesses secret data that passes through the MDS buffers, and a passive attacker, which patiently sniffs data from the MDS buffers until obtaining the desired secret value. It is possible, however, to employ the opposite scenario: the victim is actively trying to execute some code, and the attacker actively fills the MDS buffers with carefully chosen values to influence the execution of the victim thread. To confirm this scenario, we have created a very simple PoC - a single attacker thread which continuously fills the line fill buffers with the address of a test variable. The test variable is not accessed directly by the program anywhere. After some time, the test variable is cached nonetheless, implying that it most likely was accessed by some other code which was fed with the test variable address from the line fill buffers following a faulting memory load. Once we stop the attacker thread which sprays the LFBs with the address of the test variable, we no longer observe it being cached, thus confirming that the source of the accesses is within code that was most likely executing with stale data loaded from the LFBs.

We have identified several scenarios that might be used to abuse the LVI-LFB problem. All have one important thing in common: a memory load instruction which requires a microcode assist. The easiest example for such a microcode assists is the hardware page-walker which is performed – as an example - for addresses which are not mapped (they are swapped out) or lack the accessed and/or the dirty bit:

1. **Influence an address that is accessed**. Consider the following (victim) code example: the attacker may control the value loaded by instruction [1], which in-turn leads to instruction [2] further accessing that address.
```
mov     rsi, [rax]          ; [1]
mov     rdi, [rsi]          ; [2]
```

2. **Influence the offset within an accessed buffer**. Consider the following (victim) code example: the attacker may control an offset which is loaded by instruction [1], which is then used by instruction [2] when accessing the memory addresses pointed by **rcx**.
```
mov     edx, [rax]          ; [1]
mov     ecx, [rcx + rdx]    ; [2]
```

3. **Influence the result of a conditional branch**. Consider the following (victim) code example: if instruction [1] requires an assist, it may execute with stale data fetched from the LFBs; if the attacker controls the contents of the LFBs, it may trick instruction [2] to branch in the desired direction.
```
    cmp     [rax], 1            ; [1]
    jne     target              ; [2]
    ...
target:
    ...
```

4. **Influence the destination of an indirect branch**. Consider the following (victim) code example: the attacker may control the destination instruction [1] will branch at.
```
call    [rax]               ; [1]
```

In all of these cases, we call instruction [1], which must induce a microcode assist, the **pivot** instruction. Other similar instances where the data read by the victim could influence exist as well.

Exploiting LVI-LFB: scenarios 1 & 2

Exploiting LVI-LFB in a real life scenario requires several conditions to be met. First, a victim gadget that can be used by the attacker to leak secret values from the victim memory is required. Second, the gadget must contain a memory load instruction that requires microcode assist (for example, it accesses a swapped out page). Consider the following hypothetical gadget:

```
mov         rax, [rcx]          ; [1]
mov         rsi, [rax + 0]      ; [2]
mov         rdi, [rax + 8]      ; [3]
movzx       eax, byte [rsi]     ; [4]
shl         eax, 12             ; [5]
mov         rax, [rdi + rax]    ; [6]
```

Looking at the instructions inside this gadget, we observe that it looks extremely convenient from an attacker perspective, but it still has several prerequisites:

1. The address pointed by **rcx** in the pivot instruction [1] must induce a microcode assist;
2. A malicious address **X** sprayed by the attacker in the LFBs must be loaded by [1] instead;
3. **X** must point to a victim accessible, valid, memory address which ideally contains the address of a sensor array (used to determine the secret) and the address of the secret to be leaked;
4. The gadget (or a similar gadget) must exist inside the victim memory

Exploiting LVI-LFB: scenarios 3 & 4

We have explored the scenario where we influence the destination of indirect branches, which we also call **LVI based control flow hijacking**. Although this looks like Spectre (since it speculatively hijacks the control flow), it is not a true Spectre, as it is data-speculation based (as is Meltdown or MDS) instead of being control-flow speculation based like true Spectre. LVI based control flow hijacking allows an attacker to trick the victim into speculatively executing a function of his choosing. This works, theoretically, across all security boundaries: process to process, user-mode to kernel-mode, guest-mode to root-mode, and perhaps even user-mode to enclave.

Our PoC consists of two processes: the victim and the attacker. In this particular scenario, although the processes are different, they are backed by the same executable file. The attacker sprays the LFBs with the address of a function located inside the victim process. This function accesses a test variable, a function that is not otherwise executed. The victim process will execute an indirect branch through an invalid memory address, and after this branch, checks if the test variable is cached. Running only the victim process yields no results – the test variable is not cached since the target function is not executed. As soon as the attacker process - which sprays the LFBs with the address of the function - starts, the test variable is cached, confirming our scenario: the indirect branch was taken to a stale address loaded from the LFBs.

The victim process consists of the following code gadgets:

```
VictimFunctionFault PROC
    mfence
    mov         rax, 0000000000000000h      ; [1]
    jmp         qword ptr [rax]             ; [2]
    mfence
    ret
VictimFunctionFault ENDP

PoisonFunction PROC
    mov             rcx, 0BDBD0000h         ; [3]
    mov             rax, [rcx]              ; [4]
    mfence
    ret
PoisonFunction ENDP
```

The first function, **VictimFunctionFault** is executed in a loop by the victim. All it does is zero out a register [1] and then jump to whatever lies at address 0 [2]. This code will crash by triggering a page-fault; however, due to MFBDS, stale data from the LFBs is used. On the sibling thread, the attacker sprays the address of **PoisonFunction**. This causes the CPU to speculatively start executing the sprayed function address, which stores the address of our test variable in a register [3] and then accesses it [4], causing it to be cached and allowing us to measure the access time to it and confirm the success of the attack.

Real-life exploit

Creating a real-life exploit poses some significant challenges:

1. Identifying a suitable gadget for one of the scenarios; this depends a lot on the victim and what code it contains; certain gadgets may not be available at all;
2. Making sure the pivot instruction incurs a microcode assist so it loads attacker-controlled data from the LFBs. This is quite challenging to do, but there are some options – one could simply wait for that page to have the accessed bit cleared, and when it is accessed again, induce the microcode assist. Another way to achieve this is by forcing that address to be swapped out by applying high memory pressure over the system; this has, however, a severe disadvantage, because data that is useful for the attacker may be swapped out as well, thus rendering this method inefficient;
3. Finding a way to speculatively transmit the secret from the victim to the process. While transmitting the secret from kernel to user can be done rather easily, doing so from one process to another is more complicated. On Windows operating systems, dynamic libraries are loaded inside each process at the same address, and the backing physical pages are the same, as long as they are not written (in which case the copy-on-write mechanism will create a local copy). Therefore, we can use such a shared region of memory – for example, the resources section inside the ntdll module is read-only, and is accessed rarely enough that significant noise is not generated when used as a sensor for leaking the secret.

We reported the issue to Intel on the 10th of February 2020. Their response on 25th of February 2020 acknowledged the issue, and they stated that the embargo is applicable until 10th of March 2020. Unfortunately, the very short time between the reporting date and the public disclosure date did not allow us to further research this issue or finish the real-life exploit PoC. Only a synthetic PoC was finished for the LVI-LFB control flow hijacking scenario (available on the Bitdefender website). Other scenarios have been described by academia in [8], which have independently discovered and reported the issue to Intel in April 2019. For more info about the advisory, please check out the official Intel advisory [9].

Mitigations

Existing mitigations for Meltdown, Spectre, and MDS are not sufficient. First, although it hijacks the control flow, LVI based control flow hijacking is not a true Spectre vulnerability since it does not rely on bringing the branch prediction unit to a known state; therefore, existing Spectre mitigations do not help with this new class of vulnerabilities. Second, MDS mitigations are currently not sufficient as operating systems flush the MDS buffers only when transitioning from a more privileged mode to a less privileged mode, in order to evict any secret that might have remained in the MDS buffers. To properly mitigate LVI-LFB, the operating system must also flush the MDS buffers (LFBs in particular) when transitioning from a less privileged mode into a more privileged mode, in order to avoid microcode assisted memory accesses from executing speculatively with attacker controlled data. In addition, just like classical MDS, disabling HT is a good idea on systems where security is critical, as would serializing all critical load operations using the **lfence** instruction. Other mitigations could involve modifications to the compilers, in order to generate code that is not vulnerable to such type of attacks. Intel will probably address this new type of issue in-silicon in future CPU generations. For more info regarding Intel's description of the issue and mitigations, see [10].

In order to avoid process to process leaks via shared memory, we also propose **horizontal KPTI**. Currently, KPTI works in a vertical manner by isolating the more privileged kernel-mode from the less privileged user-mode – this way, a process does not have access to the kernel memory in any way. However, a malicious process could potentially leak sensitive data from a (potentially more privileged) victim process by using the memory that is shared between them (generally represented by shared libraries, such as ntdll or kernelbase on Windows systems – since they contain the same code & data, these modules are mapped to the same physical pages). With horizontal KPTI, processes lying in different security domains would have their own physical copy of the shared libraries, thus mitigating leaks via the shared memory channel (which can be used to transmit secrets from the victim to the attacker).

Conclusions

We have disclosed a new perspective on MDS (MFBDS in particular) which is called LVI-LFB. Instead of leaking sensitive data from the LFBs, spray them with known values which get used speculatively by the victim. In addition, we have elaborated a particular example of LVI-LFB - which we call **LVI based control flow hijacking** - which allows an attacker to feed certain indirect branches with addresses to malicious code, and thus gain speculative arbitrary code execution. We have also discussed other possibilities for LVI-LFB as we think that this technique can be used to initiate a kernel-to-user leak, or even across other security boundaries, such as enclaves or a hypervisor. A synthetic PoC has been published on the main Bitdefender site.

Credits

We would like to credit the researchers who first reported this issue to Intel: **Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens** and we would like to thank them for their cooperation on this issue.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," *Proceedings of the 27th USENIX Conference on Security Symposium, https://meltdownattack.com/meltdown.pdf,* 2018.

[2] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," *CoRR, https://spectreattack.com/spectre.pdf,* 2018.

[3] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch and Y. Yarom, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient," *https://foreshadowattack.eu/foreshadow-NG.pdf,* 2018.

[4] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar and Y. Yarom, "Fallout: Reading Kernel Writes From User Space," *https://mdsattacks.com/files/fallout.pdf,* 2019.

[5] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," *https://zombieloadattack.com/zombieload.pdf,* 2019.

[6] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos and C. Giuffrida, "RIDL: Rogue In-Flight Data Load," *https://mdsattacks.com/files/ridl.pdf,* 2019.

[7] A. Lutas and D. Lutas, "Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction," Bitdefender, 2019. [Online]. Available: https://businessresources.bitdefender.com/hubfs/noindex/Bitdefender-WhitePaper-SWAPGS.pdf. [Accessed 27 02 2020].

[8] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41st IEEE Symposium on Security and Privacy*, San Francisco, 2020.

[9] Intel, "Intel SA 00334," Intel, 10 03 2020. [Online]. Available: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00334.html. [Accessed 10 03 2020].

[10] Intel, "Deep Dive Load Value Injection," Intel, 10 03 2020. [Online]. Available: https://software.intel.com/security-software-guidance/insights/deep-dive-load-value-injection. [Accessed 10 03 2020].

[11] D. Lutas and A. Lutas, "Security implications of speculatively executing segmentation related instructions on Intel CPUs," Bitdefender, 2019. [Online]. Available: https://businessresources.bitdefender.com/hubfs/noindex/Bitdefender-WhitePaper-INTEL-CPUs.pdf?utm_campaign=swapgs&utm_source=web&adobe_mc=MCMID%3D75629865980736119172972677486032326210%7CMCORGID%3D0E920C0F53DA9E9B0A490D45%2540AdobeOrg%7CTS%3D1582795014. [Accessed 27 02 2020].