VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Secure Hardware Devices
Microarchitectural Data Sampling

30. března 2020 Jiří Matějka (xmatej52)

Obsah

1 Introduction

Microarchitectural Data Sampling, also known as MDS, is a hardware vulnerability allowing unprivileged speculative access to data which are stored in various CPU buffers [1]. There is set of possible attacks on MDS vulnerability – ZombieLoad, RIDL and Fallout. Using these attacks, the attacker can gain access to potentially sensitive data, such as passwords or private keys.

This vulnerability affects a wide range of Intel processors and is not present in processors from non-Intel vendors, Older processors (where CPU family is ¡ 6), some Atoms processors (Bonnel, Saltwell, Goldmont and GoldmontPlus) and in intel processors which have the ARCH_CAP_MDS_NO bit set in the IA32_ARCH_CAPABILITIES MSR [1].

Sections with basic informations about RIDL (section 1.1), Fallout (section 1.2) and ZombieLoad (section 1.3) were inspired by the Graz University of Technology web pages [2, 7, 4].

1.1 RIDL

Rogue In-Flight Data Load is an exploit that allows the attacker to mount practical attacks and leak sensitive data in real-world settings. Data are leaked in-flight from different internal CPU buffers, that are used by the CPU while loading or storing data from memory.

Attacker, who can run unprivileged code on machines with impacted CPUs, can access potentially sensitive data (such as passwords or encryption keys) using side channels. Malicious code can be run on shared cloud computing resources and since code can be written in JavaScript, the attacker can use websites or advertisement banners to execute an attack.

1.2 Fallout

The Fallout enables the attacker to leak data from store buffers, which are used every time a CPU pipeline need to store any data. The attacker can then pick later which data to leak from the CPU's store buffer.

The Fallout attack can be also used to break kernel address space layout randomization, as well as to leak sensitive data written to memory by the operating system kernel.

1.3 ZombieLoad

The ZombieLoad attack can exploit internal CPU buffers to get hold of data currently processed by other processes. These data can be user-level secrets, such as user keys, passwords, browser history and website content, or it can be system-level secrets, such as encryption keys. The ZombieLoad attack can be exploited on personal computers as well as in the cloud.

2 RIDL

This section will describe, how to attack victim virtual machine in the cloud using RIDL vulnerability. This tutorial is based on presentation and documentation provided by the Graz University of Technology [5, 6].

2.1 Preparing the attack

Our victim virtual machine is located in the cloud, so we need to get our virtual machine on the same server and make sure, that it is co-located via line fill buffers. Our victim virtual machine then runs the SSH server. Now we only need to:

- Get data in flight,
- leak data,
- filter data.

2.2 Getting data in flight

The easiest part of mounting the attack is to get data in flight. To do so, we just need to connect to the SSH server. SSH server then will need to load data through line fill buffers. So after connecting to SSH server, file */etc/shadow* will be in flight.

2.3 Leaking data

Since data are now in flight, it is possible to leak them. We start RIDL application on our virtual machine that will leak data from the line fill buffers.

The first step, the RIDL application should do, is to flush the cache. Since we will later detect cache hits, this step is necessary in order to correctly detect leaked data.

Next step is to leak data. This is how line fill buffers work:

1. Send out memory request,
2. allocate line fill buffers entry,
3. store address in line fill buffers,
4. serve other loads and stores,
5. pending request eventually completes.

RIDL exploits problem, where there are data from a previous load operation in line fill buffers entry in the second step (allocation of line fill buffers entry). So now, we are ready to leak in-flight data from an invalid or unmapped page.

To retrieve data, we will have to access the probe array and measure how long we waited for data. If we get data from specific index faster then from other indexes, the used index is leaked byte of data. Follows an example of possible RIDL application in C language:

```c
// Flush buffer entries
for ( int i = 0; i < 256; i++ ) {
    flush( probeArray + ( i * 4096 ) );
}

// Storing potentially leaked byte into variable
char byte =  *( new_page );

// Using leaked byte as an index into probeArray
char *p   = probeArray + ( byte * 4096 )

// Loading leaked byte into cache
*( p );

// Getting leaked byte from cache
for ( int i = 0; i < 256; i++ ) {

    t0 = cycles();
    *( volatile char * ) ( probeArray + ( i * 4096 ) );
    dt = cycles() - t0;

    // If we accessed data quickly enought, current index is leaked byte
    if ( dt < 100 ) {
        results[ i ]++;
    }
}
```

2.4 Filtering data

The last step is to filter leaked data – we leak data that are in flight and not all of the leaked information is useful for us. But filtering data is easy – we can use prefix matching (we know, that first line in /etc/shadow starts as root:) or a different method of filtering data based on the mask. Example of prefix matching:

```
Iteration 1
Known prefix: "root:"
Current data: https:// [No Match]
Current data: root:Sp/ [Match, so we add "S" into result]

Iteration 2
Known prefix: root:S
Current data: someText [No Match]
Current data: root:Sp/ [Match, so we add "p" into result]

Iteration 3
Known prefix: root:Sp
Current data: FILE.TXT [No Match]
Current data: root:Sp/ [Match, so we add "/" into result]
...
```

3 Fallout

In this section, we will describe a simple fallout attack using Write Transient Forwarding, also known as WTF. Information about this attack was gathered from the Graz University of Technology documentation about Fallout attack [3].

3.1 Store Buffer

When the CPU needs to write data to memory, it writes them into store buffer. Thanks to this, CPU does not have to wait for a write to finish and can continue executing instructions from the current instruction stream. Meanwhile, the store buffer asynchronously ensures, that the results are written to memory.

3.2 Write Transient Forwarding

Write Transient Forwarding incorrectly passes values from memory writes to subsequent faulting load operations. When a program attempts to read from an address, the CPU must first check the store buffers for write to the same address and perform store-to-ride forwarding if the address matches. Note, that the faulting load is transient and will not retire. However, the microarchitectural side effects of transient execution following the faulting load may result in an information leak.

In this example, we will use non-canonical addresses to cause a fault since these addresses work reliably across all CPU generations. Note, that other exception causes are also possible. We refer to the Graz University of Technology documentation [3] to the detailed analysis of different exception types and how they may trigger WTF.

First, we need to allocate a page, where the user can write and read data. Then we define pointer, which points to non-canonical address. Dereferencing such pointer will result in a general protection fault, faulting the dereferencing load. Then we store secret into the allocated page. CPU then allocates store buffer entry for holding the secret information to be written into memory.

To read previous stores, we can dereference the defined pointer to the non-canonical address. We then suppress the general protection fault that results from this address using a TXS transaction. In the meantime, the CPU transiently forwards the value of the previous store at the same page offset. So, the memory access picks-up the value of the store to the allocated page. Using the cache-based covert channel, we transmit the incorrectly forwarded value.

Using the same mechanism as in RIDL attack (section 2), we can retrieve data – we access an array if the executing of the memory access instruction is faster then for other indexes, the current index is leaked byte.

```
// Allocates new page in user space
char * allocatedPage = mmao(...);

// Defining poiter to non-canonical address
char* invalidAdress = 0x1478953214578952ull;

// Writing secret information into allocated page
int index = 12;
```

```
allocatedPage[ index ] = 111;

// Accessing invalid adres using TSX, so exception is suppressed
if ( tsx_begin() == 0 ) {
    memory_access( probeArray + ( 4096 * invalidAdress[ index ] ) );
    tsx_end();
}

// Getting leaked byte
for ( int i = 0; i < 256; i++ ) {

    t0 = cycles();
    *( volatile char * ) ( probeArray + ( i * 4096 ) );
    dt = cycles() - t0;

    // If we accessed data quickly enought, current index is leaked byte
    // It will be fastest for i == 111
    if ( dt < 100 ) {
        results[ i ]++;
    }
}
```

This code shows, that it is possible to leak data without directly accessing allocatedPage array, so leak of potentially sensitive information is possible.

4 ZombieLoad

Due to limited size of this article,, ZombieLoad will be described briefly and we will not provide any source code of possible attack. All the information was gained from documentation provided by the Graz University of Technology [8].

4.1 About ZombieLoad

ZombieLoad is a transient-execution attack that observes the values of memory loads and stores on the CPU core. This attack exploits, that the fill buffer is used by all logical CPUs of a CPU core and that it does not distinguish between processes or privileges.

If there is memory load during execution, CPU reserves an entry in the load buffer. If the load was not an L1 cache hit, it requires a fill-buffer entry. If store operations miss the L1 cache, data are temporarily stored in the fill-buffer entry as well.

But under certain conditions (as a fault for example), where the load requires microcode assist, it may first read state values before being re-issued. This opens a transient-execution window, where the value can be used for subsequent calculation. Thanks to this, an attacker is able to store the leaked value into the cache.

With the ZombieLoad attack, the attacker is not able to select the value to leak based on an attacker-specified address. The ZombieLoad attack leaks any value which is currently loaded or stored by the physical CPU core.

4.2 ZombieLoad attack via Intel TSX Transaction

To execute this attack, it is necessary to have allocated physical page that is user-accessible via virtual address v (any page allocated in user space fulfil this requirement). Inside a TSX transaction, encode the value of v into the cache using convert-channel – the same way as in Fallout scenario (section 3). Although this is a legitimate load to the user-accessible address v and should not itself cause the TSX transaction fail, the TSX transaction fails and does not commit due to included conflicts in the read set. There is an only transient fault which results into a zombie load (a load, that is either architectural or microarchitectural fault and thus cannot complete, requiring a re-issue of the load at a later point).

Note, that there two other attack scenarios in the documentation provided by the Graz University of Technology [8]. This approach was chosen because it works even with hardware fixes for Meltdown.

5 Conclusion

The goal of this project was to create a tutorial on how to execute an attack using Microarchitectural Data Sampling vulnerability in 4 – 6 pages of text. In this article, we described RIDL attack (section 2), Fallout attack (section 3) and ZombieLoad attack (section 4). In all of these attacks, we briefly described possible scenarios and for RIDL and Fallout attacks, we also provided source codes for a better understanding of the attacks.

Provided information about these attack should be sufficient to mount an attack on vulnerable devices if certain conditions for each attack are fulfilled. Provided source code should ease the development of such applications. Note, that this article is an academic study and abuse of this vulnerability may be illegal and is at least immoral.

Reference

[1] COMMUNITY The kernel development. *MDS – Microarchitectural Data Sampling* [online]. [vid. 2020-03-22]. Dostupné z: <`https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/mds.html`>.

[2] TECHNOLOGY Graz University of. *CPU.fail* [online]. 2019 [vid. 2020-03-22]. Dostupné z: <`https://cpu.fail/`>.

[3] TECHNOLOGY Graz University of. *Fallout: Leaking Data on Meltdown-resistant CPUs* [online]. 2020 [vid. 2020-03-28]. Dostupné z: <`https://mdsattacks.com/files/fallout.pdf`>.

[4] TECHNOLOGY Graz University of. *MDS: Microarchitectural Data Sampling* [online]. 2020 [vid. 2020-03-22]. Dostupné z: <`https://mdsattacks.com/`>.

[5] TECHNOLOGY Graz University of. *RIDL* [online]. 2020 [vid. 2020-03-22]. Dostupné z: <`https://mdsattacks.com/slides/slides.html`>.

[6] TECHNOLOGY Graz University of. *RIDL: Rogue In-Flight Data Load* [online]. 2020 [vid. 2020-03-28]. Dostupné z: <`https://mdsattacks.com/files/ridl.pdf`>.

[7] TECHNOLOGY Graz University of. *ZombieLoad Attack* [online]. 2020 [vid. 2020-03-22]. Dostupné z: <`https://zombieloadattack.com/`>.

[8] TECHNOLOGY Graz University of. *ZombieLoad: Cross-Privilege-Boundary Data Sampling* [online]. 2020 [vid. 2020-03-31]. Dostupné z: <`https://mdsattacks.com/files/fallout.pdf`>.