POSITIVE TECHNOLOGIES

HOW TO HACK A TURNED-OFF COMPUTER, OR RUNNING UNSIGNED CODE IN INTEL ME

Contents

1. Introduction

Intel Management Engine (Intel ME) is a proprietary technology that consists of a microcontroller integrated into the Platform Controller Hub (PCH) chip and a set of built-in peripherals. The PCH carries almost all communication between the processor and external devices. Therefore, Intel ME has access to almost all data on the computer. The ability to execute third-party code on Intel ME would allow for a complete compromise of the platform.

We see increasing interest in Intel ME internals from researchers all over the world. One of the reasons is the transition of this subsystem to new hardware (x86) and software (modified MINIX as an operating system [1]). The x86 platform allows researchers to make use of the full power of binary code analysis tools. Previously, firmware analysis was difficult because earlier versions of ME were based on an ARCompact microcontroller with an unfamiliar set of instructions.

Analysis of Intel ME 11 was previously impossible because the executable modules are compressed by Huffman codes with unknown tables. However, our research team has succeeded in recovering these tables and created a utility for unpacking images [2].

After unpacking the executable modules, we proceeded to examine the software and hardware internals of Intel ME. Our efforts to understand the workings of ME were rewarded: ME was ultimately not so unapproachable as it had seemed.

1.1. Intel Management Engine 11 overview

A detailed description of Intel ME internals and components can be found in several papers: [1], [3], [4]. It should be noted that starting in 2015, the LMT processor core with the x86 instruction set has been integrated into the PCH. Such a core is used in the Quark SOC.

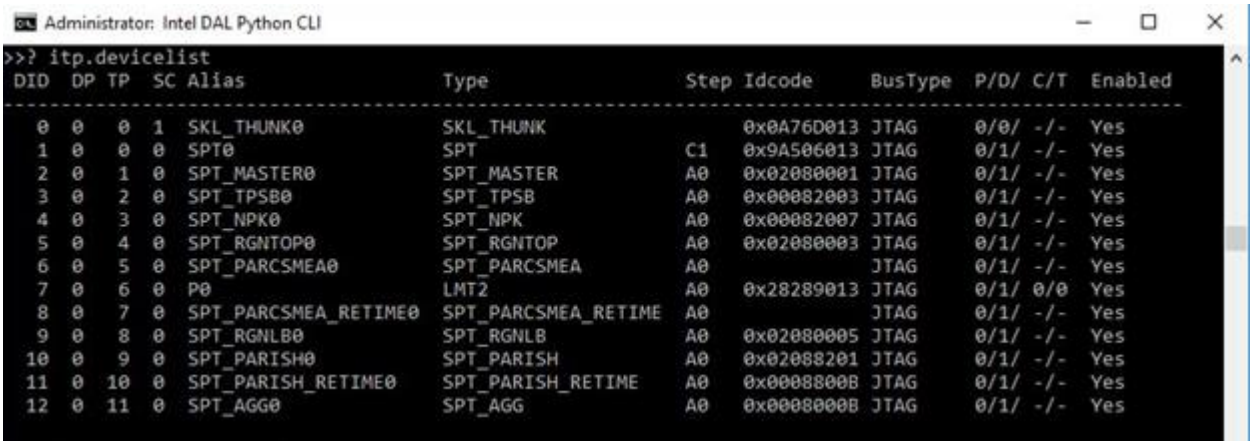

Figure 1. LMT2 IdCode of ME core

Many modern technologies by Intel are built around Intel Management Engine: Intel Active Management Technology, Intel Platform Trust Technology (fTPM), Intel Software Guard Extensions, and Intel Protected Audio Video Path. ME is also a root of trust for Intel Boot Guard, which prevents attackers from injecting their code into UEFI. The main purpose of ME is to initialize the platform and start the main processor. ME also has virtually unlimited access to data processed on the computer. ME can intercept and modify network packets as well as images on graphics cards; it has full access to USB devices. Such capabilities mean that if an attacker finds an opportunity to execute arbitrary code inside ME, this will spawn a new generation of malware that cannot be detected using current protection tools. Fortunately, only three (publicly known) vulnerabilities have been detected in the 17-year history of this technology.

1.2. Published vulnerabilities in Intel ME

1.2.1. Ring-3 rootkits

The first publicly known vulnerability was discovered in Intel ME in 2009. At Black Hat, Alexander Tereshkin and Rafal Wojtczuk gave a talk entitled "Introducing Ring-3 Rootkits". The attack involved injecting code into a special region of UMA memory into which ME unloads currently unused memory pages.

After the research was made public, Intel introduced UMA protection. Now this region is encrypted with AES and ME stores the checksum for each page, which is checked when the page is returned to the main memory of ME.

1.2.2. Zero-Touch Provisioning

In 2010, Vassilios Ververis presented an attack on the implementation of ME in GM45[10]. By using "zero-touch" provisioning mode (ZTC), he was able to bypass AMT authorization.

1.2.3. Silent Bob is Silent

In May 2017, a vulnerability in the AMT authorization system (CVE-2017-5689) was published [11]. It allowed an unauthorized user to obtain full access to the main system on motherboards supporting the vPro technology.

Thus, to date only one vulnerability (Ring-3 rootkits) allowing execution of arbitrary code inside Intel ME has been found.

2. Potential attack vectors

Virtually all data used by ME is either explicitly or implicitly signed by Intel. However, ME still allows some interaction with the user:

- Local communication interface (HECI)
- Network (vPro only)
- Host memory (UMA)
- Firmware SPI layout
- Internal file system

2.1. HECI

HECI is a separate PCI device serving as a circular buffer to exchange messages between the main system and ME.

Applications located inside ME can register their HECI handlers. This increases the number of potential security issues (CVE-2017-5711). On Apple computers, HECI is disabled by default.

2.2. Network (vPro only)

AMT is a large module with a huge number of different network protocols of various levels integrated into it. This module contains a great deal of legacy code but can only be found in business systems.

2.3. Hardware attack on SPI interface

While we were studying ME, it occurred to us to attempt bypassing signature verification with the help of an SPI flash emulator. This specialized device would look like regular SPI flash to the PCH, but can send different data each time it is accessed. This means that if the data signature is checked in the beginning and then the data is reread, one can conduct an attack and inject code into ME. We did not find such errors in the firmware: first, data is read, and then the signature is verified. When accessed

again, data is checked to make sure it is identical to the data obtained during the first read.

2.4. Internal file system

Intel ME uses SPI flash as primary file storage with its own file system. While the file system has a rather complicated structure [6], many privileged processes store their configuration files in it. Therefore the file system seemed a very promising place for acting on ME.

The next step in searching for vulnerabilities was to choose a binary module.

2.5. Selecting a module for analysis

The ME operating system implements a Unix-like access control model, the difference being that controls are on a per-process basis. The user-id, group-id, list of accessible hardware, and allowed system calls are set statically for each process.

```
Ext#5 Process:
  flags: permanent_process, single_instance
  main_thread_id: 0xC
  priv_code_base_address: 0x00040000
  uncompressed_priv_code_size: 0x29C6
  cm0_heap_size: 0x0
  bss_size: 0x7004
  default_heap_size: 0x1000
  main_thread_entry: 0x0004020A
  allowed_sys_calls: e000c783f804000000000000
  user_id: 0x005C
  group_ids[1]: [0x0121]

Ext#8 MmioRanges[41]:
  sel=  7, base:F5022000, size:00000C00, flags:00000003 :: SUSRAM_S
```

Figure 2. Example of static rules for a process

The result is that only some system processes are able to load and run modules. A parent process is responsible for verifying integrity and setting privileges for its

child process. One risk, of course, is that a process can set high privileges for its child in order to bypass restrictions.

One process with the ability to spawn child processes is BUP (BringUP). In the process of reverse engineering the BUP module, we discovered a stack buffer overflow vulnerability in the function for Trace Hub device initialization. The file `/home/bup/ct` was unsigned, enabling us to slip a modified version into the ME firmware with the help of Flash Image Tool. Now we were able to cause a buffer overflow inside the BUP process with the help of a large BUP initialization file. But exploiting this required bypassing the mechanism for protection against stack buffer overflows.

```
if ( !(reg & 0x1000000) && !bup_get_si_features(si_features) &&
!bup_get_file_size("/home/bup/ct", &ct_file_size) )  {

    if ( ct_file_size ) {
              LOBYTE(err) = bup_dfs_read_file("/home/bup/ct", 0, ct_file_data, ct_file_size,
                                                                                &read_size);
```

Figure 3. Stack buffer overflow vulnerability

2.6. Bypassing stack buffer overflow protection

ME implements a classic method for protection from a buffer overflow in the stack—a stack cookie. The implementation is as follows:

1. When a process is created, a 32-bit value is copied from the hardware random number generator to a special region (read-only for process).
2. In the function prologue, this value is copied above the return address in the stack, thus protecting it.
3. In the function epilogue, the saved value is compared with the known good value. If they do not match, a software interrupt (`int 81h`) terminates the process.

So exploitation requires either predicting the cookie value or taking control before cookie integrity is checked. Further research showed that any error in the random number generator is regarded by ME as fatal, causing it to fail.

Looking at the functions that are called after an overflow and before the integrity check, we found that the function we named `bup_dfs_read_file` indirectly calls `memcpy`**.** It, in turn, gets the destination address from the structure we named Tread Local Storage (TLS). Notably, BUP functions for file read/write use system library services for accessing shared memory. In other words, read and write functions obtain and record data via a shared memory mechanism. But this data is not used anywhere other than BUP, so use of this mechanism may raise eyebrows. In our view, memory is shared likely because the portion of BUP code responsible for MFS interaction was copied from another module (file system driver), where use of shared memory is justified.

```
signed int __cdecl sys_write_shared_mem(...)
{
...
    sm_block_desc = sys_get_shared_mem_block(block_idx);
...
    memcpy_s((sm_block_desc->start_addr_linked_block_idx + offset),
            sm_block_size - offset, src_data, write_size);
...
}
```

Figure 4. Calling the `memcpy` function

```
int __cdecl sys_get_ctx_struct_addr(SYS_LIB_CTX_STRUCT_ID struct_id)
{
...
  sys_ctx_start_ptr = sys_get_tls_data_ptr(SYSLIB_GLB_SYS_CTX);
  switch ( struct_id ) {
     case SYS_CTX_SHARED_MEM:
        addr = *sys_ctx_start_ptr + 0x68;
        break;
...
    }
  return addr;
}
```

Figure 5. Getting address from the TLS

As we discovered later, in case of a buffer overflow this region of the TLS can be overwritten by a file read function, which could be used to bypass buffer overflow protection.

2.7. Tread Local Storage

Access to the TLS is mediated by the `gs` segment registry. The structure looks as follows:

```
typedef struct
{
        uint32_t         reservied;
        SYSLIB_CTX_PTR * syslib_ptr;
        int32_t          last_error;
        uint32_t         thread_id;
        void *           self_pointer;
} T_TLS;
```

Figure 6. TLS structure

```
sys_get_tls_data_ptr proc near
tls_idx = dword ptr  8
        push    ebp
        mov     ebp, esp
        mov     eax, large gs:0
        mov     ecx, [ebp+tls_idx]
        pop     ebp
        lea     edx, ds:0[ecx*4]
        sub     eax, edx
        retn
sys_get_tls_data_ptr endp
```

Figure 7. Getting TLS fields

The segment to which `gs` points is not write-accessible, but the TLS structure itself is at the bottom of the stack (!!!), which allows modifying it in spite of the restrictions. So in the case of a buffer overflow, we can overwrite the pointer to the `SYSLIB_CTX` in the TLS and generate new such structure. Because of how the `bup_dfs_read_file` function works, this trick gives us arbitrary write abilities.

2.8. Using implementation of read function to get an arbitrary write primitive

The `bup_dfs_read_file` function reads from SPI-flash in 64-byte blocks, due to which it is possible to overwrite the pointer to `SYSLIB_CTX` in a one iteration and during the next iteration, the `sys_write_shared_mem` function extracts the address that we created and passes it to `memcpy` as the destination address. With this done, we can get an arbitrary write primitive.

```
int __cdecl bup_read_mfs_file(BUP_MFS_DESC *mfs_desc, int file_number, unsigned int offset,
                                        unsigned int *size, int sm_block_idx,
                                        __int16 proc_thread_id)
{
...
      while ( 1 ) {
        if ( cur_offset >= read_size ) break;
...
          err = bup_mfs_read_data_chunks(mfs_desc, buffer,
                                  mfs_desc->data_chunks_offset + ((read_start_chunk_id -
mfs_desc->total_files) << 6),
                                  block_chunks_count);
...
          err = sys_write_shared_mem(proc_thread_id, sm_block_idx, cur_offset,
                                 &buffer[chunk_offset], copy_size, copy_size);
...
        }
      }
...
}
```

Figure 8. Iterative reading of file inside `bup_dfs_read_file`

The absence of ASLR enables us to overwrite a return address using the arbitrary write primitive and hijack the program control flow. But here lies an unpleasant surprise for the attacker—the stack is not executable. Remember, however, that BUP can spawn new processes and is responsible for checking module signatures. So with Return-Oriented Programming (ROP), we can create a new process with the rights we need.

2.9. Possible exploitation vectors

To successfully exploit the vulnerability, we need write access to the MFS or entire Intel ME region. Vendors are supposed to block access to the ME region, but many fail to do so [8]. Such a configuration error makes the system vulnerable.

By design, Intel ME allows for write access to the ME region via special HMR-FPO messages sent over HECI from the BIOS [9]. An attacker can send such a message by exploiting a BIOS vulnerability, or directly from OS if ME is in manufacture-mode, or via a DMA attack.

Attackers with physical access can always overwrite with their own image (via SPI programmer or Security Descriptor Override jumper), resulting in a complete compromise of the platform.

One of the most common questions regards the possibility of remote exploitation. We think that remote exploitation is possible if the following conditions are true:

1. The target platform has AMT activated.
2. The attacker knows the AMT administrator password or can use a vulnerability to bypass authorization.
3. The BIOS is not password-protected (or the attacker knows the password).
4. The BIOS can be configured to open up write access to the ME region.

If all these conditions are met, there is no reason why an attacker would not be able to obtain access to the ME region remotely.

Also note that during startup, the ROM does not check the version of firmware, leaving the possibility that an attacker targeting an up-to-date system could maliciously downgrade ME to a vulnerable version.

2.10. CVE-2017-5705,6,7 overview

The vulnerability was assigned number INTEL-SA-00086 (CVE-2017-5705, CVE-2017-5706, CVE-2017-5707). The description of the vulnerability includes the following information:

CVSSv3 Vectors:

8.2 High AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H

Affected products [12]:

- 6th, 7th & 8th Generation Intel® Core™ Processor Family
- Intel® Xeon® Processor E3-1200 v5 & v6 Product Family
- Intel® Xeon® Processor Scalable Family
- Intel® Xeon® Processor W Family
- Intel® Atom® C3000 Processor Family
- Apollo Lake Intel® Atom Processor E3900 series
- Apollo Lake Intel® Pentium™
- Celeron™ N and J series Processors

2.11. Disclosure Timeline

- June 27, 2017 - Bug reported to Intel PSIRT
- June 28, 2017 - Intel started initial investigation
- July 5, 2017 - Intel requested proof-of-concept
- July 6, 2017 - Additional information sent to Intel PSIRT
- July 17, 2017 - Intel acknowledged the vulnerability
- July 28, 2017 - Bounty payment received
- November 20, 2017 - Intel published SA-00086 security advisory

3. Conclusion

The most important finding of our research was a vulnerability that allows running arbitrary code in Intel ME. Such a vulnerability has the potential to jeopardize a number of technologies, including Intel Protected Audio Video Path (PAVP), Intel Platform Trust Technology (PTT / fTPM), Intel Boot Guard, and Intel Software Guard Extension (SGX).

By exploiting the vulnerability that we found in the bup module, we were able to turn on a mechanism, PCH red unlock, that opens full access to all PCH devices for their use via the DFx chain—in other words, using JTAG. One such device is the x86 ME processor itself, and so we obtained access to its internal JTAG interface. With such access, we could debug code executed on ME, read memory of all processes and the kernel, and manage all devices inside the PCH. We found a total of about 50 internal devices to which only ME has full access, while the main processor has access only to a very limited subset of them.

Our research does not claim to be the final word on ME security or free of errors. Nonetheless, we hope that this work will be of benefit to other researchers interested in platform security and ME internals.

References

[1] Dmitry Sklyarov, Intel ME: The Way of the Static Analysis, Troopers, 2017.

[2] Intel ME 11.x Firmware Images Unpacker (https://github.com/ptresearch/unME11).

[3] Xiaoyu Ruan, Platform Embedded Security Technology Revealed. Safeguarding the Future of Computing with Intel Embedded Security and Management Engine, Apress, ISBN 978-1-4302-6572-6, 2014

[4] Igor Skochinsky, Intel ME Secrets. Hidden code in your chipset and how to discover what exactly it does, RECON 2014.

[5] Alexander Tereshkin, Rafal Wojtczuk, Introducing Ring-3 Rootkits, Black Hat USA, 2009.

[6] Dmitry Sklyarov, Intel ME: flash file system explained, Black Hat Europe, London, 2017.

[8] Alex Matrosov, Who Watch BIOS Watchers? (https://medium.com/@matrosov/bypass-intel-boot-guard-cc05edfca3a9).

[9] Mark Ermolov, Maxim Goryachy, How to Become the Sole Owner of Your PC, PHDays VI, 2016 (http://2016.phdays.com/program/51879/).

[10] Vassilios Ververis, Security Evaluation of Intel's Active Management Technology, Sweden, TRITA-ICT-EX-2010:37, 2010.

[11] Dmitriy Evdokimov, Alexander Ermolov, Maksim Malyutin, Intel AMT Stealth Breakthrough, Black Hat USA, 2017 Las Vegas, NV.

[12] Intel Management Engine Critical Firmware Update (Intel-SA-00086) (http://www.intel.com/sa-00086-support).

Contacts

Email: pr@ptsecurity.com
Web: www.ptsecurity.com

Twitter: @PTsecurity_UK
Blog: blog.ptsecurity.com

361 King Street,
London, W6 9NA
United Kingdom

+44 203 769 3606