Multiple TOCTOU conditions in SMIFlash SW SMI handlers

Researchers

- Enrique Nissim (enrique.nissim@ioactive.com)
- Krzysztof Okupski (<u>krzysztof.okupski@ioactive.com</u>)
- Joseph Tartaro (joseph.tartaro@ioactive.com)

Impact

BIOS Image: G513QR.330 (2022/10/28) - Latest available as on Feb 2023

The SMIFlash module installs 6 SW-SMI handlers that are prone to double fetches (TOCTOU) that, if successfully exploited, could be leveraged to execute arbitrary code in System Management Mode (ring-2).

There is a public CVE related to this module from 2017:

- CVE-2017-3753 https://nvd.nist.gov/vuln/detail/CVE-2017-3753
- CVE-2017-11316

Based on the presentation from Alexander Matrosov

(https://www.blackhat.com/docs/us-17/wednesday/us-17-Matrosov-Betraying-The-BIOS-Where-The-Guardians-Of-The-BIOS-Are-Failing.pdf), this module was not using SmmlsBufferOutsideSmmValid(). The findings presented here show that this validation was indeed added (through AMI_SMM_BUFFER_VALIDATION_PROTOCOL) but mistakes were made in the process.

This document describes three vulnerabilities in three different code paths of the SW-SMI handler deployed by the SMIFlash module.

Description

The SMIFlash module (GUID BC327DBD-B982-4F55-9F79-056AD7E987C5) is one of the SMM Modules that are included in the BIOS Image G513QR.330 for the ASUS Rog Strix G513QR. This module registers a single SW SMI handler for six different SwSmiInputs (0x20, 0x21, 0x22, 0x23, 0x24, 0x25) via the EFI_SMM_SW_DISPATCH2_PROTOCOL protocol:

```
loc_273B:
mov ebx, 20h

loc_273B:
mov eax, ebx
lea r9, [rsp+38h+RegisterContext.5w5milnputUalue], rax
lea r8, [rsp+38h+RegisterContext]; RegisterContext
mov rax, [rsp+38h+RegisterContext]; DispatchHandle
nov quord ptr [rsp+38h+RegisterContext]; PegisterContext
mov rax, [rsp+38h+RegisterContext]; DispatchEprot]
lea rdx, [rsp+38h+RegisterContext]; DispatchEprot]
tea rdx, [rsp+38h+RegisterContext]; DispatchEprot]
tea rdx, [rsp+38h-RegisterContext]; DispatchEprot]
tea rdx, [rsp+38h-RegisterContext]; DispatchEprot]
tea rdx, [rsp+38h-RegisterContext]; DispatchHandle
nov quord ptr [rsp+38h+RegisterContext]; RegisterContext
mov rax, [rsp+38h-RegisterContext]; DispatchHandle
nov quord ptr [rsp+38h-RegisterContext]; RegisterContext
nov rax, [rsp+38h-RegisterContext]; DispatchHandle
nov quord ptr [rsp+38h-RegisterContext]; RegisterContext
nov rax, [rsp+38h-RegisterContext]; DispatchHandle
nov quord ptr [rsp+38h-RegisterContext]; RegisterContext
nov rax, [rsp+38h-RegisterContext]; DispatchHandle
nov quord ptr [rsp+38h-RegisterContext]; RegisterContext
nov rax, [rsp+38h-RegisterContext]; DispatchHandle
nov quord ptr [rsp+38h-RegisterContext]; RegisterContext
nov rax, [rsp+38h-RegisterContext]; DispatchHandle
nov quord ptr [rsp+38h-RegisterContext]; RegisterContext
nov rax, [rsp+38h-RegisterContext]; DispatchHandle
nov quord ptr [rsp+38h-RegisterContext]; RegisterContext
nov rax, [rsp+38h-RegisterContext]; Regist
```

Based on public documentation of this module, the above operations map to:

```
- 0x20 - ENABLE

- 0x21 - READ

- 0x22 - ERASE

- 0x23 - WRITE

- 0x24 - DISABLE

- 0x25 - GET INFO
```

The handler uses the EFI_MM_CPU_PROTOCOL to read the content of the saved ECX and EBX registers to create a 64-bit pointer. For all operations except for GET_INFO, this constructed address is verified to be outside SMRAM using the

AMI_SMM_BUFFER_VALIDATION_PROTOCOL for exactly 18h bytes long. 18h is therefore the size of the basic input record this module needs to work with. Reverse engineering the structure led to the following layout:

```
struct SmiFlash_XXOp

{
  void *buffer_data;
  UINT32 flash_addr;
  UINT32 size;
  UINT32 status;
};
```

Depending on the value written on the SW-SMI trigger port (the SwSmiInput), the execution continues in one of the previously listed operations (ENABLE, READ, WRITE.. etc).

The operations READ and WRITE receive the pointer to the record as argument and both are prone to the same Time-of-Check to Time-of-Use vulnerability (TOCTOU). Let's look at the read implementation:

```
READ_0x21 proc near
push rbx
sub rsp. 26h
nov edx. [rcx+5miFlash_XXOp.size]; size
nov rbx. rcx
nov edx. [rcx+5miFlash_XXOp.flash_addr]; << CHECK
sub ecx. 10808080h : addr
call check_address_falls_within_flash_range
test rax, rax
js short loc_1C68

**Total Control Contro
```

RCX holds the controlled pointer and is copied into RBX. The function starts by checking that the 'flash_addr' value falls within the intended flash range mmio (0xFF000000-0xFFFFFFF). It continues by using the AMI_SMM_BUFFER_VALIDATION_PROTOCOL to ensure that the buffer_data ptr resides outside SMRAM. This is interesting because the reported issues that ended up in the CVE-2017-3753 and CVE-2017-11316 suggest to be related to the lack of validation over the input parameters If the input pointers are not properly verified using SmmIsBufferOutsideSmmValid() (in this case ValidateMemoryBuffer()), an attacker can pass a pointer with an SMRAM address value and have the ability to read and/or write to SMRAM. In our current version, this is not the case anymore and we can see verification is there.

Nevertheless, the code is retrieving the values from memory twice for all three members (flash_addr, size, and buffer_data), This means that the values checked do not necessarily correspond to the ones being passed to FlashDriverSmm module. This is a race condition that an attacker can exploit through a DMA attack. Such an attack can be easily performed with a custom PCI device (e.g. PciLeech - https://github.com/ufrisk/pcileech).

Winning the race for the Read operation leads to writing to SMRAM with values retrieved from Flash. However, by disabling the Flash with the DISABLE operation first, the underlying implementation of the FLASH_SMM_PROTOCOL (which resides in the FlashDriverSmm module), will use a simple memcpy to fulfill the request:

```
; __int64 __fastcall Flash_Read(int flash_addr, int size, void \times buffer_ptr)
Flash_Read proc near
arg_0= qword ptr 8
arg_8= qword ptr 10h
arg_10= qword ptr 18h
         [rsp+arg_0], rbx
[rsp+arg_8], rbp
[rsp+arg_10], rsi
mov
mov
mov
          rdi
push
         rsp, 20h
sub
         rdi, r8
mov
          rsi, rdx
mov
         rbp, rcx
Disable_SoftwareSMIs
mov
call
          ebx, ebx
xor
test
          rax, rax
          short loc_1FFC
js
                        =
                                  r8, rdi
                        mov
                                                      ; buffer_ptr
                        mov
                                  rdx, rsi
                                                      ; size
                                                      ; flash_addr
                        mov
                                  rcx, rbp
                        call
                                  cs:g_EnableDisable_Counter, ebx
                        стр
                        mov
                                  r9, rax
                                  short loc_1FF4
                        jz
```

```
; __int64 __fastcall perform_read(int flash_addr, int size, void *buffer_ptr)
perform_read proc near
sub    rsp, 201
         r9, r8
mov
         r8d, cs:g_UAL_1
mov
         eax, r8d
mov
         eax, 3
and
         al, 3
cmp
jz
         short loc_192D
                                          test
                                                    r8b, 1
                                                    short loc_1913
                                          jz
                                *
                                         cs:g_EnableDisable_Counter,
                                cmp
                                         short loc_192D
                                jnz
                                  1
                                  loc_1913:
test r
                                                             ; check size > 0
                                           rdx, rdx
                                           short loc_1938
                                  jz
                                  <u></u>
                                                             ; check src != dst
                                           r9, rcx
                                  cmp
                                  jz
                                           short loc_1938
                      <u></u>
                                          4
                                          mov
                                                   r8, rdx
                      loc_192D:
                                          mov
                                                   rdx, rcx
                      mov
                               r8d, edx
                                                   rcx, r9
                                          mov
                      mov
                               rdx, r9
                                          call
                                                   memcpy
                      call
                               sub_2CA4
                                          jmp
                                                   short loc_1938
```

The Write operation has the exact same condition, although in this case, winning the race means leaking content from SMRAM into the Flash:

```
WRITE_0x23 proc near
                                              arg_0= gword ptr 8
                                                        [rsp+1000b], rbx
                                              mov
                                              push
                                                        rdi
                                                        rsp, <mark>20</mark>h
                                              sub
                                                        eax, [rcx+SmiFlash_XXOp.flash_addr]
                                              mov
                                              mov
                                                        rbx, rcx
                                                       eax, cs:value_1000000h; 0x10000000
edx, [rcx+SmiFlash_XX0p.size]; size
ecx, eax; addr
                                              sub
                                              mov
                                              mou
                                              mov
                                                        edi, eax
                                              call
                                                        check_address_falls_within_flash_range
                                              test
                                                        rax, rax
                                                        short loc_1CD6
                                        <u></u>
                                        mov
                                        test
                                                  rax, rax
                                                  short loc_1CD6
              <u></u>
              mov
                        [rax+AMI_SMM_BUFFER_VALIDATION]
              call
              test
                        rax, rax
                        short loc_1CD6
                                                               II 🚄
🔟 🚄 🚾
mov
                                                               loc_1CD6:
mov
         ecx, edi
         edx, [rbx+SmiFlash_XXOp.size] r8, [rbx+SmiFlash_XXOp.buffer
                                                                         rax, 8000000000000000Fh
mov
                                                               mov
                                                                         byte ptr [rbx+10h], 1
                                             data1
                                                               mov
mov
         qword ptr [rax+10h] ; FlashSmmProtocol.Write
call
test
         rax, rax
sets
mov
         [rbx+10h], cl
         short loc_1CE4
jmp
```

As a summary, in both cases, the block of code performing the checks did not make local copies of the values into SMRAM. The values are being retrieved again from user controlled memory when they are about to be used, which means they could have changed.

The operation GET_INFO (0x25) is affected by the same condition, although in a different way. In this case, as soon as the user pointer is constructed, the code verifies it to be outside of SMRAM for at least 1Ah bytes. Then, it retrieves the value of the first dword and uses it to further check the length of the provided region:

```
rax, cs:gAmiSmmBufferValidationProtocol
                                     mov
                                      стр
                                               dil, 25h
                                              short loc_1F2D
                                      jnz
                test
                         rax, rax
                         loc 2268
                jz
rbx, [rsp+68h+pointer] ; 0xb2 = 0x25
edx, 1Ah ; BufferSize
rcx, rbx ; Buffer
mov
mov
mov
call
         [rax+AMI_SMM_BUFFER_UALIDATION_PROTOCOL.UalidateMemoryBuffer]
        rax, rax
loc_2268
test
js
                                       🔟 🚄 🖼
                                                 rax, cs:gAmiSmmBufferValidationProtocol
                                       mov
                                                edx, [rbx]
rax, rax
loc_2268
                                        mov
                                                                   ; retrieve Size
                                        test
                                        jz
                                                  short loc_1F40
                                                  jmp
                           <u></u>
                           loc_1F40:
                                                      ; Buffer
                           mov
                                    rcx, rbx
                           call
                                    [rax+AMI_SMM_BUFFER_UALIDATION_PROTOCOL.UalidateMemoryBuffer]
                                    rax, rax
                           test
                                    loc_2268
                           js
```

The reversed engineered structure looks like follows:

```
struct SecSMI_GetInfoStruct {
    /* 0x00 */ DWORD size;
    /* 0x04 */ BYTE getinfo_type_status;
    /* 0x05 */ BYTE operation_mb;
    /* 0x06 */ WORD writtenItems;
    /* 0x08 */ DWORD Code_mb;
    /* 0x0C */ DWORD xxx1;
    /* 0x10 */ DWORD val_not_zero_to_copy;
    /* 0x14 */ DWORD status;
    /* 0x18 */ BYTE unk3;
    /* 0x19 */ BYTE unk4; // end of Header portion

// PAYLOAD STARTS HERE
    /* 0x1A */
}
```

The code continues by calling into a function that allocates 905Bh bytes of SMRAM memory and attempts to copy the data into it. RBX is the pointer to the record, and the double-fetch is clear:

```
| Fig. | nov | rax, cs.gSmst | lea | r8, [rsp+38h+8uffer] | Buffer | nov | ebp. 9098h | nov | edx, ebp | size | size | rax, rax | size | r
```

The code is trying to enforce 905Bh bytes as an upper limit for the copy but because the memory is fetched twice, the value could have changed after the check passed. As a result, SMRAM memory will be corrupted.

Recommendations

- Copy the provided arguments into SMRAM and then perform the corresponding validation in order to avoid double-fetch issues.
- Call AMI_SMM_BUFFER_VALIDATION_PROTOCOL.ValidateMemoryBuffer on provided buffers before reading or writing to them.