Multiple TOCTOU conditions in SMIFlash SW SMI handlers

Researchers

- Enrique Nissim (enrique.nissim@ioactive.com)
- Krzysztof Okupski (krzysztof.okupski@ioactive.com)
- 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:

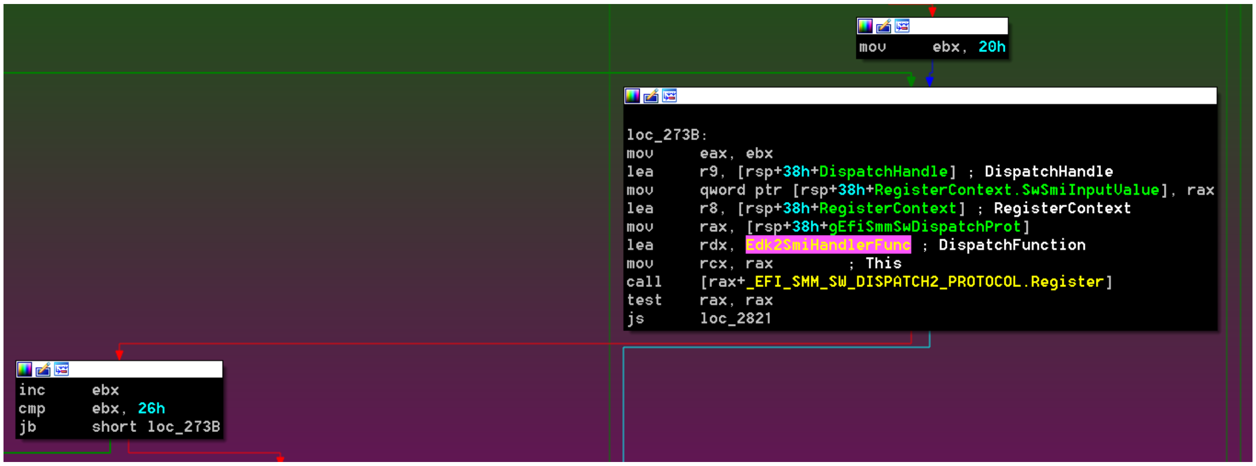
- CVE-2017-3753 - https://nvd.nist.gov/vuln/detail/CVE-2017-3753
- CVE-2017-11316

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 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.

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:



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

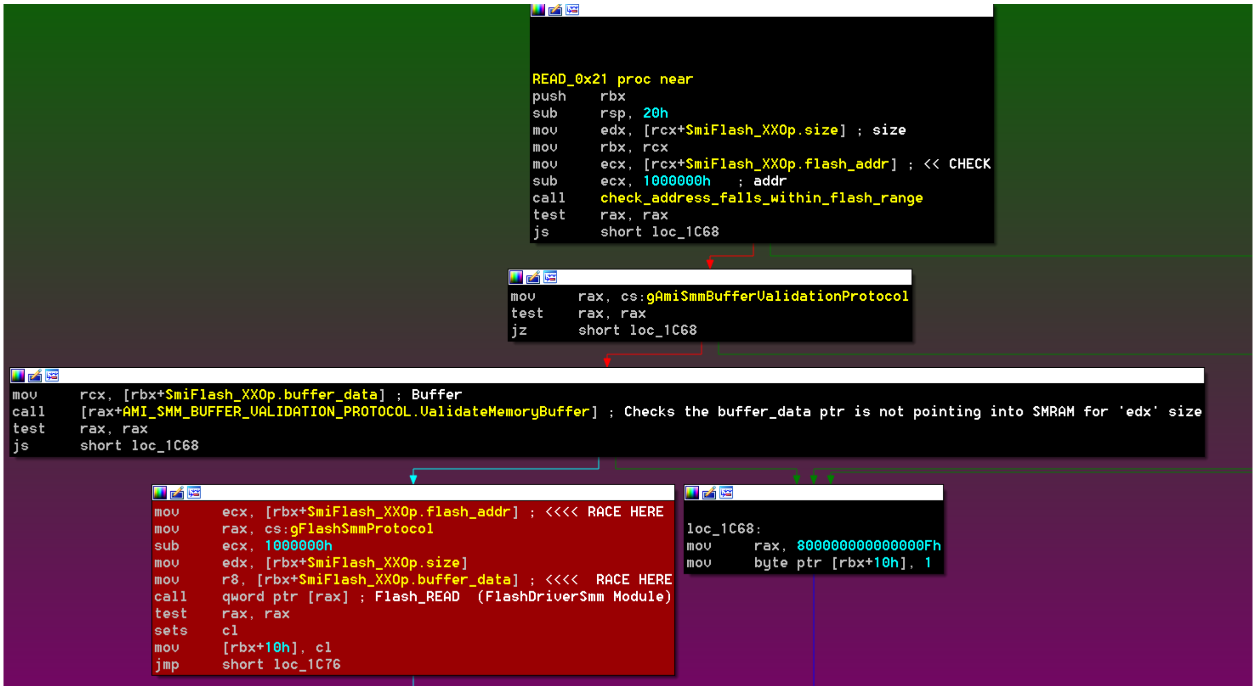
- 0x20 - ENABLE
- 0x21 - READ
- 0x22 - ERASE
- 0x23 - WRITE
- 0x24 - DISABLE
- 0x25 - GET_INFO

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:



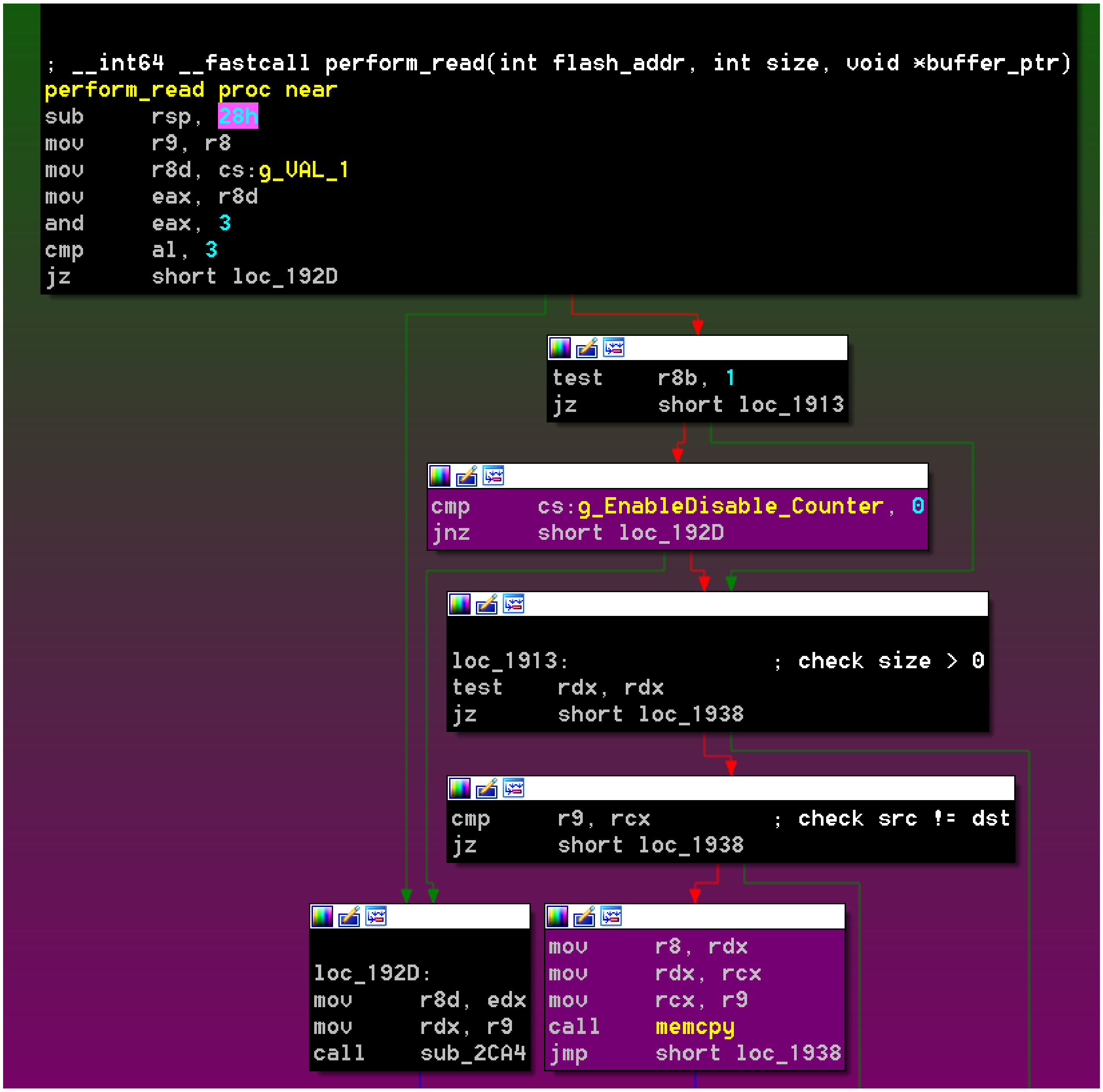
RCX holds the controlled pointer and is copied into RBX. The function starts by checking that the `flash_addr` value falls within the intended flash range mmio (0xFF000000-0xFFFFFFFF). It continues by using the AMI_SMM_BUFFER_VALIDATION_PROTOCOL to ensure that the buffer_data ptr resides outside SMRAM. 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).

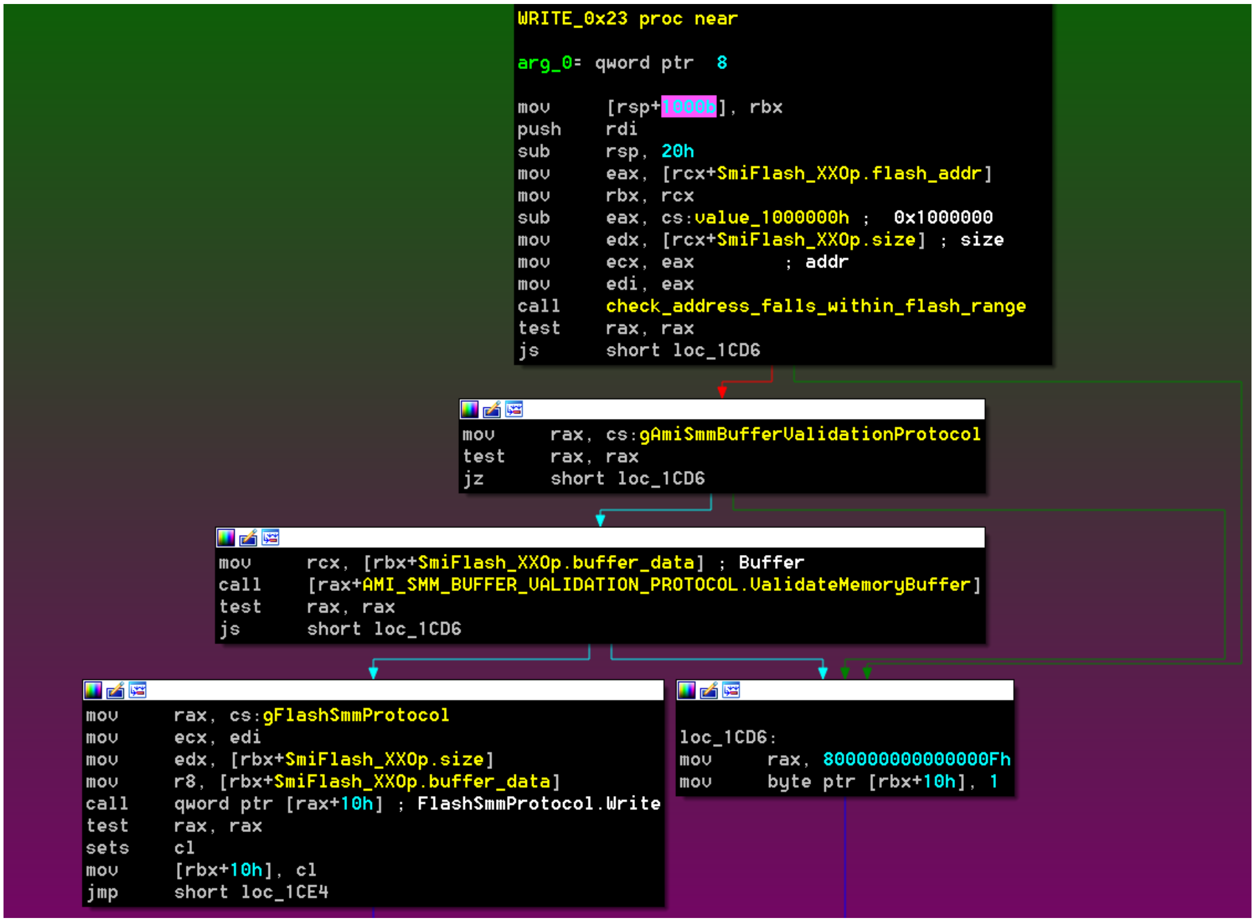
Winning the race for the Read operation leads to writing to SMRAM with values retrieved from Flash. However, by disabling the Flash with the DISABLE operation first, the underlying implementation of the FLASH_SMM_PROTOCOL (which resides in the FlashDriverSmm module), will use a simple memcpy to fulfill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

```
; __int64 __fastcall Flash_Read(int flash_addr, int size, void *buffer_ptr)
Flash_Read proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h

mov     [rsp+arg_0], rbx
mov     [rsp+arg_8], rbp
mov     [rsp+arg_10], rsi
push    rdi
sub     rsp, 20h
mov     rdi, r8
mov     rsi, rdx
mov     rbp, rcx
call    Disable_SoftwareSMIs
xor     ebx, ebx
test    rax, rax
js      short loc_1FFC
```

```
mov     r8, rdi            ; buffer_ptr
mov     rdx, rsi           ; size
mov     rcx, rbp           ; flash_addr
call    perform_read
cmp     cs:g_EnableDisable_Counter, ebx
mov     r9, rax
jz      short loc_1FF4
```

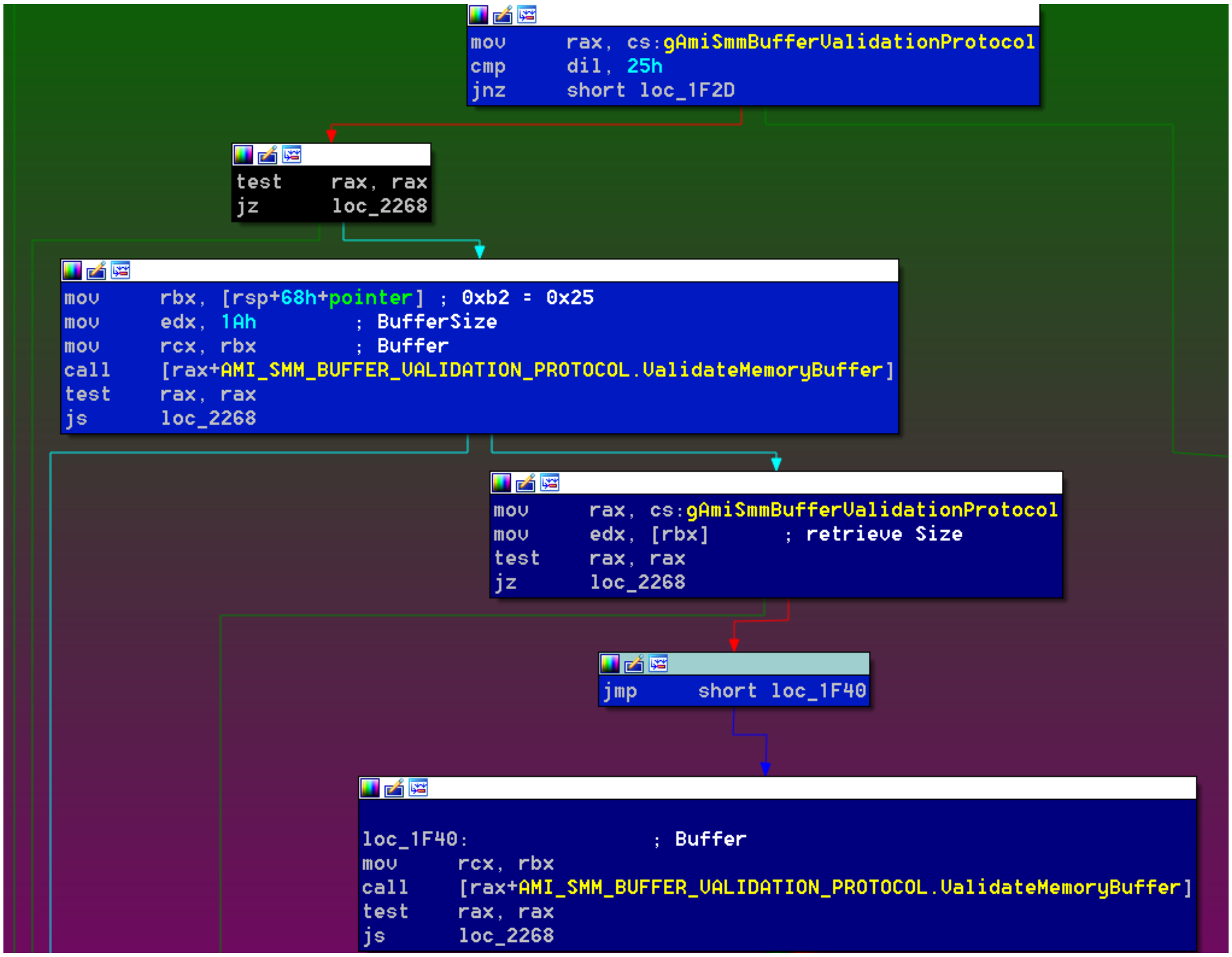


The Write operation has the exact same condition, although in this case, winning the race means leaking content from SMRAM into the Flash:



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.

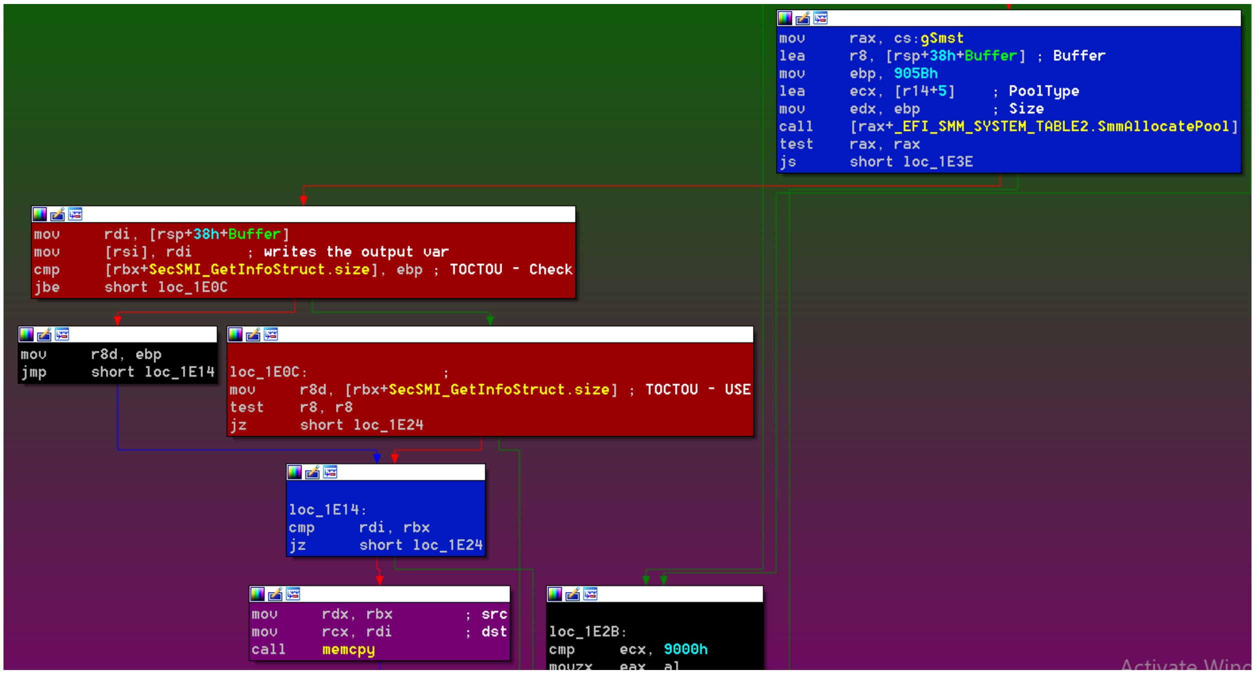
The operation GET_INFO (0x25) is affected by the same condition, although in a different way. In this case, as soon as the user pointer is constructed, the code verifies it to be outside of SMRAM for at least 1Ah bytes. Then, it retrieves the value of the first dword and uses it to further check the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



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:



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.