Vulnerability in Tiancore EDK2 UEFI reference implementation

Lack of validation in TCG2 module leading to LPE

Introduction

A vulnerability was identified in the implementation of the TCG2 module in Tianocore's EDK2. This bug is due to a lack of validation in an SMM handler that could allow an attacker to write arbitrary data in SMM. Due to the limitation mentioned in the next section, one needs to be able to execute code in the DXE phase to reach the bug. Therefore, we estimate that the impact of such vulnerability is rather reduced as it can only be used to escalate privileges to SMM.

Considering the git history of the impacted module, it seems this bug was introduced two year ago by commit 3c2dc30 which separated Tcg2Smm into 2 modules. This report is based on the latest version of EDK2 at the time it was written, tagged edk2-stable202302.

Constrains

The faulty handler is actually removed at the end of the DXE phase. In order to trigger the bug, one needs to send a message before `EndOfDxe` protocol is published which means that it can only be used by an attacker already having control over a DXE module (whether it be via another pre-boot vulnerability or a backdoor). This limitation implies that at most, it can be used to gain a better footing over the UEFI environment by escalating to ring -2.

Vulnerability Description

The Tcg2 module is used to implement TPM 2.0 definition block in ACPI table. It registers several methods of communication:

- two SMI callback functions (Tcg2PhysicalPresence and MemoryClear) used to handle the requests from ACPI method;
- a root handler to communicate the NVS region and SMI channel between SMM and DXE.

The root handler, implemented in `Tcg2Smm.c`, takes a `TPM_NVS_MM_COMM_BUFFER` structure as data in the communication buffer. This structure has the following definition:

```
typedef struct {
  UINT64                  Function;
  UINT64                  ReturnStatus;
  EFI_PHYSICAL_ADDRESS    TargetAddress;
  UINT64                  RegisteredPpSwiValue;
  UINT64                  RegisteredMcSwiValue;
} TPM_NVS_MM_COMM_BUFFER;
```

While the communication buffer is correctly validated to ensure that it is outside of SMRAM, the structure includes an address (`TargetAddress`) that is directly used, without any check, to fill the global variable named `mTcgNvs`.

```
// Tcg2Smm.c L. 90

//
// Farm out the job to individual functions based on what was requested.
//
CommParams = (TPM_NVS_MM_COMM_BUFFER *)CommBuffer;
Status     = EFI_SUCCESS;
switch (CommParams->Function) {
    case TpmNvsMmExchangeInfo:
        DEBUG ((DEBUG_VERBOSE, "[%a] - Function requested: MM_EXCHANGE_NVS_INFO\n", __FUNCTI
        CommParams->RegisteredPpSwiValue = mPpSoftwareSmi;
        CommParams->RegisteredMcSwiValue = mMcSoftwareSmi;
        mTcgNvs                          = (TCG_NVS *)(UINTN)CommParams->TargetAddress;
        break;
    default:
        DEBUG ((DEBUG_INFO, "[%a] - Unknown function %d!\n", __FUNCTION__, CommParams->Funct
        Status = EFI_UNSUPPORTED;
        break;
}
```

Unfortunately, this global variable holds a pointer to a `TCG_NVS` structure which is later used to retrieve the NVS region when handling the SMI callbacks. The following snippet shows an example of `mTcgNvs` manipulation via the Tcg2PhysicalPresence callback:

```
// Tcg2Smm.c L. 131

EFI_STATUS
EFIAPI
PhysicalPresenceCallback (
  IN EFI_HANDLE  DispatchHandle,
  IN CONST VOID  *Context,
  IN OUT VOID    *CommBuffer,
  IN OUT UINTN   *CommBufferSize
  )
{
  UINT32  MostRecentRequest;
  UINT32  Response;
  UINT32  OperationRequest;
  UINT32  RequestParameter;
if (mTcgNvs->PhysicalPresence.Parameter == TCG_ACPI_FUNCTION_RETURN_REQUEST_RESPONSE_TO_OS)
    mTcgNvs->PhysicalPresence.ReturnCode = Tcg2PhysicalPresenceLibReturnOperationResponseTo(
                                              &MostRecentRequest,
                                              &Response
                                              );
    mTcgNvs->PhysicalPresence.LastRequest = MostRecentRequest;
    mTcgNvs->PhysicalPresence.Response    = Response;
    return EFI_SUCCESS;
  }

  // ...
}
```

For the record, the TCG_NVS has the following definition:

```
#pragma pack(1)
typedef struct {
  PHYSICAL_PRESENCE_NVS    PhysicalPresence;
  MEMORY_CLEAR_NVS         MemoryClear;
  UINT32                   PPRequestUserConfirm;
  UINT32                   TpmIrqNum;
  BOOLEAN                  IsShortFormPkgLength;
} TCG_NVS;

typedef struct {
  UINT8     SoftwareSmi;
  UINT32    Parameter;
  UINT32    Response;
  UINT32    Request;
  UINT32    RequestParameter;
  UINT32    LastRequest;
  UINT32    ReturnCode;
} PHYSICAL_PRESENCE_NVS;

typedef struct {
  UINT8     SoftwareSmi;
  UINT32    Parameter;
  UINT32    Request;
  UINT32    ReturnCode;
} MEMORY_CLEAR_NVS;
```

Depending on the SMI callback used and the value of the `Parameter` field in the structure, it is therefore possible to arbitrary write values anywhere in SMRAM. The following list synthesizes the possible outcomes:

- PhysicalPresence callback:
  - PHYSICAL_PRESENCE_NVS.Parameter == 2:
    - PHYSICAL_PRESENCE_NVS.Request = 0x000000XX;

        * PHYSICAL_PRESENCE_NVS.ReturnCode = 0x00000001;
        * Leak several bytes of SMRAM in `Tcg2PhysicalPresence` nvs variable.
    - PHYSICAL_PRESENCE_NVS.Parameter == 5:
        * PHYSICAL_PRESENCE_NVS.Response = 0xXXXXXXXX;
        * PHYSICAL_PRESENCE_NVS.LastRequest = 0x000000XX;
        * PHYSICAL_PRESENCE_NVS.ReturnCode = 0x00000000.
    - PHYSICAL_PRESENCE_NVS.Parameter == 7:
        * PHYSICAL_PRESENCE_NVS.Request = 0x000000XX;
        * PHYSICAL_PRESENCE_NVS.ReturnCode = 0x00000001;
        * Leak several bytes of SMRAM in `Tcg2PhysicalPresence` nvs variable.
    - PHYSICAL_PRESENCE_NVS.Parameter == 8: PHYSICAL_PRESENCE_NVS.ReturnCode = 0x00000000.
- MemoryClear callback:
    - MEMORY_CLEAR_NVS.Parameter == 1: MEMORY_CLEAR_NVS.ReturnCode = 0x00000000.
    - MEMORY_CLEAR_NVS.Parameter == 2: MEMORY_CLEAR_NVS.ReturnCode = 0x00000000.
    - default value: MEMORY_CLEAR_NVS.ReturnCode = 0x00000001.

Where XX indicates that the value is retrieved from a non-volatile variable (`Tcg2PhysicalPresence`).

Exploitation

As it was described in the previous section, the write primitive induced by the manipulation of `mTcgNvs` is quite limited. This section illustrates a way to transform it to actual code execution. For the purpose of demonstrating the actual feasibility of an attack using this vulnerability, we put ourselves in the scenario where we already have code execution in DXE.

A PoC is provided in annexes. However, it should be noted that the following modifications were applied in the target module to ensure its functioning in OVMF (for which we developed the exploit):

- changing both SMI callbacks into root MMI handlers in `Tcg2Smm.c`;
- hardcoding value of `mMcSoftwareSmi` and `mPpSoftwareSmi` in `Tcg2Smm.c` to reflect the previous modification;
- adding `gEfiMmCpuIoProtocolGuid` as dependancy in the `.inf` file;
- removing the `gEfiSmmSwDispatch2ProtocolGuid` depex in the `.inf` file.

The diff files are given in annexes.

4-byte Write Primitive to Arbitrary Read-Write Primitive

Considering the conditions and outcomes that goes with the flaw, the easiest manipulation we can gain is a fixed 4-byte write primitive of 0x00000001 anywhere in SMRAM (using the MemoryClear SMI callback). It is possible to transform this rather weak primitive into something more powerful by corrupting global variables used in other SMI handlers.

For the sake of the reproducibility of the exploit, we decided to only focus on SMI handlers that are provided "as is" in EDK2. In such context, one possible module that could be worth corrupting is `SMMLockBox`. This module provides a SMI handler that allows to save and restore data in SMRAM which can be useful from an exploitation perspective.

Note: the LockBox SMI handler is also locked at the end of the DXE phase. However, considering the scenario in which we are (i.e., executing code before `EndOfDxe` protocol publication), we are not impacted by this limitation. Should we handle this issue, it would not cause too much difficulty as it would simply add an extra step in the exploit chain (changing the value of the `mLocked` variable to "unlock" the handler).

In order to transform the SmmLockBox SMI handler into a proper Read/Write SMRAM primitive, one needs to change the value of the `mSmmMemLibInternalSmramCount` variable. This variable is used by `SmmIsBufferOutsideSmmValid` to ensure that the provided buffer does not overlap with the SMRAM.

```
// SmmMemLib.c L. 112

BOOLEAN
EFIAPI
SmmIsBufferOutsideSmmValid (
  IN EFI_PHYSICAL_ADDRESS  Buffer,
  IN UINT64                Length
  )
{
// [...]

  for (Index = 0; Index < mSmmMemLibInternalSmramCount; Index++) {
    if (((Buffer >= mSmmMemLibInternalSmramRanges[Index].CpuStart) && (Buffer < mSmmMemLibIn
        ((mSmmMemLibInternalSmramRanges[Index].CpuStart >= Buffer) && (mSmmMemLibInternalSmr
    {
      DEBUG ((
        DEBUG_ERROR,
        "SmmIsBufferOutsideSmmValid: Overlap: Buffer (0x%lx) - Length (0x%lx), ",
        Buffer,
        Length
        ));
      DEBUG ((
        DEBUG_ERROR,
        "CpuStart (0x%lx) - PhysicalSize (0x%lx)\n",
        mSmmMemLibInternalSmramRanges[Index].CpuStart,
        mSmmMemLibInternalSmramRanges[Index].PhysicalSize
        ));
      return FALSE;
    }
  }

  return TRUE;
}
```

Since the current primitive allows to write 0x00000001, it is possible to overwrite the variable with the higher bytes of the primitive to set it to 0. This prevents iterating over the SMRAM memory ranges and ensuring that the buffer is outside the SMRAM. However, we can note that even if we set it to 1 by aligning the corruption with the address of the variable, the exploit would still work as the first range referenced in the `mSmmMemLibInternalSmramRanges` list is `0x7000000 - 0x7001000`. Most of the buffers used in the PoC are located at higher addresses.

With this check removed, it becomes very straightforward to craft a read/write primitive in SMRAM. The steps are:

1. Create a LockBox entry with the LockBox command `EFI_SMM_LOCK_BOX_COMMAND_SAVE`;
2. To read in SMRAM:
   - Update the entry with an address in SMRAM (`EFI_SMM_LOCK_BOX_COMMAND_UPDATE`). This will copy the content into the LockBox buffer;
   - Restore the Lockbox with `EFI_SMM_LOCK_BOX_COMMAND_RESTORE` to retrieve the data.
3. To write in SMRAM:
   - Update the entry with a controlled buffer outside of SMRAM (`EFI_SMM_LOCK_BOX_COMMAND_UPDATE`);
   - Send a `EFI_SMM_LOCK_BOX_COMMAND_RESTORE` request with an address in SMRAM. This will overwrite the data at this address with the content that has been previously sent.

These steps are detailed in the functions `CorruptLockBox` and `CreateLockBoxPrimitive` in the provided PoC.

Arbitrary Read-Write Primitive to Code Execution

Once again, we'll leverage the use of the SmmLockBox module for the sake of the exploitation. Adding a LockBox entry via `EFI_SMM_LOCK_BOX_COMMAND_SAVE` provides a handy way to arbitrarily allocate pool buffers in SMRAM. Using this feature, it is possible to send and store a shellcode in SMRAM.

As LockBox entries are linked together in a `LIST_ENTRY`, finding the location of the buffer can be achieved by retrieving the last entry added to the list head (stored in the global variable `mLockBoxQueue`).

The only drawback of this technique is that the buffer is allocated as `EfiRuntimeServicesData` memory, meaning that the page in which the shellcode is stored is non-executable.

The `CraftPayload` function, defined in the PoC, describes a ROPchain that respectively removes the write protect and NX bits of the CR0 register and page entry in the PTE.

The execution is done by inserting a fake SMI handler in the double-linked list stored in `PiSmmCore`. As the purpose of the PoC is not to demonstrate the implementation of a proper backdoor, we didn't put much effort in crafting the handler. Thus, while the PoC currently makes the firmware crash at the end of its execution, we consider that a real attacker will be able to make it cleaner if needed be.

Side note: During the development of the exploit, we noticed that the `/ALIGN:4096` build option is missing in the Ovmf package for the MSFT toolchain. Therefore, the page-level protections are currently not enforced on this target. Adding the following lines fix the issue:

```
[BuildOptions.common.EDKII.DXE_RUNTIME_DRIVER]
  MSFT:*_*_*_DLINK_FLAGS         = /ALIGN:4096


[BuildOptions.common.EDKII.DXE_SMM_DRIVER, BuildOptions.common.EDKII.SMM_CORE]
  MSFT:*_*_*_DLINK_FLAGS         = /ALIGN:4096
```

Remediation

The remediation is straightforward as one simply needs to verify that the location of the structure pointed by `CommParams->TargetAddress` does not overlap with the SMRAM. The following piece of code illustrates such validation:

```
  // Tcg2Smm.c L.90

  if (!IsBufferOutsideMmValid (CommParams->TargetAddress, sizeof(TCG_NVS))) {
    return EFI_ACCESS_DENIED;
  }
```