Strategies for Stronger Software SMI Security in UEFI Firmware

Fall 2017 UEFI Seminar and Plugfest
October 30 – November 3, 2017
Presented by Tim Lewis (Insyde Software)
Agenda

• Management Mode Overview
• Implementing Software MMI Handlers Securely
• Call To Action & Resources
Management Mode* (MM) Overview

- UEFI PI-standard for creating a protected execution environment using hardware resources
  - Dedicated, protected memory space, entry point and hardware resources, such as timers and interrupt controllers
  - Implemented using SMM (on x86) or TrustZone (Arm)
  - Highest-privilege operating mode with greatest access to system memory and hardware resources

*Formerly known as SMM in the PI specification.
Software Management Mode Interrupts* (Software MMIs)

• Management Mode Interrupts generated by software synchronously are called Software MMIs
  – Generated using I/O resources or CPU instructions

• Used to provide firmware services to the
  – OS (ACPI, TPM)
  – OS drivers (device handoff, CPU management)
  – UEFI runtime support (variables, capsule, etc.)
  – BIOS vendor applications (flash utilities, setup access)
  – OEM/ODM applications

*Formerly known as SMI s in the PI specification.
Why Are Software MMI Vulnerabilities Dangerous?

• Software MMIs can be asked to perform privileged operations
  – Flash BIOS, flash EC, write to MMIO, write to MMRAM, etc.

• Software MMIs can be asked to overwrite OS code/data

• Software MMIs can be asked to copy protected OS data to another unprotected location

• Software MMIs can be asked to copy protected firmware data to another unprotected location

• Software MMIs can be asked to overwrite BIOS code/data
Assumptions For This Presentation...

- Memory protected by the OS cannot be snooped while in use by the OS application or OS driver
  - No protection from MM, VMs or hardware snooping

- Flash protected by hardware cannot be modified outside of MM after the end of DXE
  - Not worried about snooping since no secrets are stored in BIOS
  - Not worried about flash-altering hardware attacks

- Software MMIs cause CPUs to enter SMM in SMRAM at a fixed location

- MMRAM cannot be altered from outside SMM
Implementing Software MMI Handlers Securely
Implementing Software MMI Handlers Securely Overview

• #1: Allocate The Buffer In PEI/DXE

• #2: Never Trust That Pointers Point To The Buffer

• #3: Prohibit Input/Output Buffer Overlap

• #4: Don’t Trust Structure Sizes

• #5: Verify Variable-Length Data
#1: Allocate The Buffer In PEI/DXE

- Don’t use a buffer provided by the OS application or OS driver
  - Might point to SMRAM, MMIO, OS data, firmware data or generate an exception
- Allocate the buffer during DXE and pass the pointer to the buffer by a table (ACPI, System Configuration) or some other tamper resistant method
- Provide a library function that verifies if a range of bytes exists within the command buffer. Example:
  - `BOOLEAN BufferInCmdBuffer(VOID *Ptr, UINTN Size);`
#2: Never Trust Pointers Point To The Buffer

• Provide a library function that verifies if a range of bytes exists within the buffer
• Must also test pointers to pointers
• Example:
  ```c
  BOOLEAN
  BufferInCmdBuffer(
      IN CONST VOID *Ptr,
      IN UINTN Size
  );
  ```
#3: Prohibit Input/Output Buffer Overlap

- If the pointers of input and output buffers overlap, then output data may overwrite input data after it has been validated, but before it has been used.

- Example:
  - Verify Input Parameter 1
  - Verify Input Parameter 2
  - Read Input Parameter 1
  - Write Output Parameter 1
    - Oops! Changes Input Parameter 2!
  - Read Input Parameter 2
  - Write Output Parameter 2
#3: Prohibit Input/Output Buffer Overlap

Example

- Check for buffer-overlap when two buffers are passed in

```c
// StructurePtr = pointer to 1st buffer.
// Structure2Ptr = pointer to 2nd buffer.
// StructureSize = size of 1st buffer.
// Structure2Size = size of 2nd buffer.

UINT8 *StructurePtrOffset = (UINT8 *) StructurePtr;
UINT8 *StructurePtr2Offset = (UINT8 *) StructurePtr2;

if (StructurePtrOffset+StructureSize >= StructurePtr2Offset &&
    StructurePtrOffset < StructurePtr2Offset+Structure2Size) {
    return SECURITY_ERROR;
}
```
#4: Don’t Trust Structure Sizes

- Verify that StructureSize member is actually in the Buffer!
  - Even if the start of the structure is in the Command Buffer, the Structure Size member might not be in the Buffer

```c
StructurePtr = (STRUCTURE_NAME *)Register;
StructureSizeOffset = OFFSET_OF(STRUCTURE_NAME, StructureSize);
StructureSizeSize = sizeof(StructurePtr->StructureSize);
if (!BufferInCmdBuffer((VOID *)StructurePtr,
                      StructureSizeOffset + StructureSizeSize - 1)){
    return SECURITY_ERROR;
}
```
#4: Don’t Trust Structure Sizes

• Verify that StructureSize is at least the minimum size of the structure that contains it
  – Later code may assume that they are working on a specific structure, but need to verify the buffer can actually hold that structure

```c
StructureSize = StructurePtr->StructureSize;
if (StructureSize < sizeof(STRUCTURE_NAME)) {
    return SECURITY_ERROR;
}
```
#5: Verify Variable-Length Data

- While parsing variable length data, the software MMI handler must not go past the end of the input buffer or output buffer
  - When parsing variable-length structures
  - When handling null-terminated strings
  - When handling arrays with fixed or variable-sized entries
#5: Verify Null-Terminated Strings

- Missing null-terminators on strings can cause many functions (StrLen, StrCpy, StrCmp, AsciiStrToUnicodeS, etc.) to access data outside of the command buffer. Example:

```c
CHAR16 Password[32];
StrCpy(Password, ptr);
```

If `ptr` points to a 40 byte string, then bytes 37-40 will be copied over the return address on the stack, causing the SMM function to return somewhere unplanned.

- For strings, use `StrnLenS()` or `AsciiStrnLenS()` in `MdePkg\Include\BaseLib.h` to verify that the string does not extend past the end of the command buffer.

```c
str = pointer to string
Length= StrnLenS (ptr, (UINT8*)end-of-buffer-(UINT8*) str);
```
#5: Verify Variable-Length Arrays

• With variable length arrays, it is easy to accidentally read/write bytes outside of the buffer
  – Especially if each entry is also variable-length
• Verify that each entry does not extend past the end of the buffer
• Verify each entry header is in buffer before reading entry size

```c
end-of-buffer = start-of-buffer + size-of-buffer.
do {
    if (!BufferInCmdBuffer(ptr, sizeof(header-struct) ||
        !BufferInCmdBuffer(ptr, ptr->StructureSize) { 
        return SECURITY_ERROR;
    }
    switch(ptr->Type) {
        ..process structure..
    }
    ptr = (header-struct*)((UINT8*)ptr + ptr->StructureSize)
} while ((UINT8 *)ptr < end-of-buffer && ptr->Type!=0x00);
```
Call To Action
Call To Action

• Revise APIs to remove trust of the calling application
• Handle multi-stage operations with good security
• Revise handler buffer code to safely process variable-length data
Thanks for attending the Fall 2017 UEFI Plugfest

For more information on the UEFI Forum and UEFI Specifications, visit http://www.uefi.org

presented by

insyde®