Stepping Insyde System Management Mode

research.nccgroup.com/2023/04/11/stepping-insyde-system-management-mode

April 11, 2023

In October of 2022, Intel's Alder Lake BIOS source code was leaked online. The leaked code was comprised of firmware components that originated from three sources:

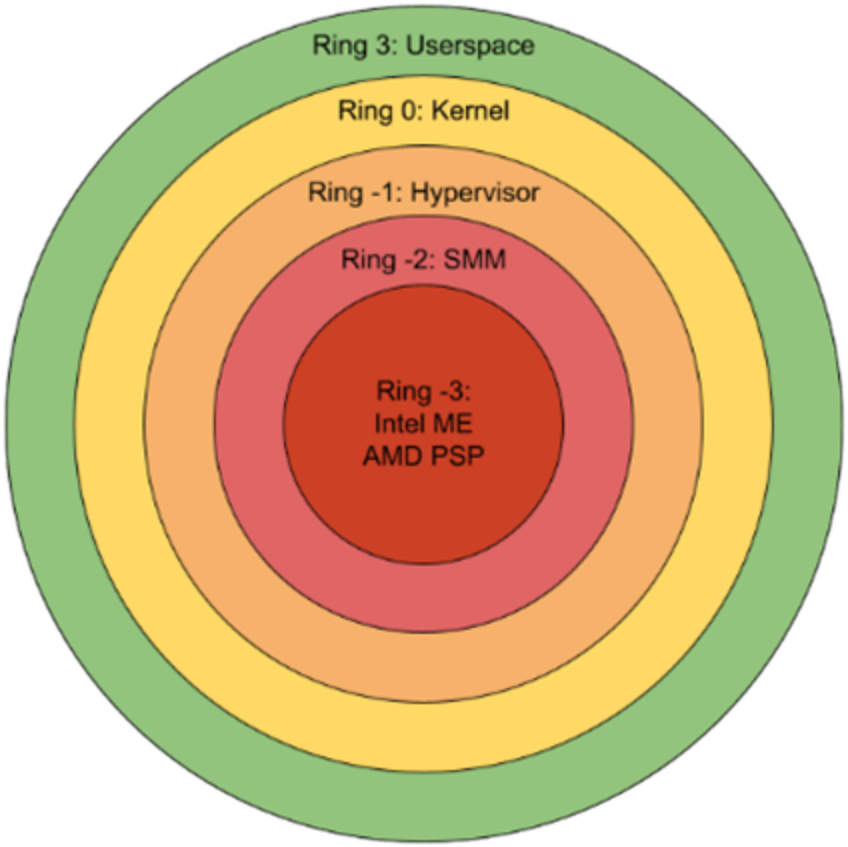
- The independent BIOS vendor (IBV) named Insyde Software,
- Intel's proprietary Alder Lake BIOS reference code,
- The Tianocore EDK2 open-source UEFI reference implementation.

I obtained a copy of the leaked code and began to hunt for vulnerabilities. This writeup focuses on the vulnerabilities that I found and reported to Insyde Software. These bugs span various System Management Mode (SMM) modules, including:

- Insyde H2O Internal Soft-SMI Interface (IHISI) dispatcher
- Flash BIOS Through SMI (FTBS) handlers
- BIOS Guard SMI handlers

What is System Management Mode and Why is it Interesting?

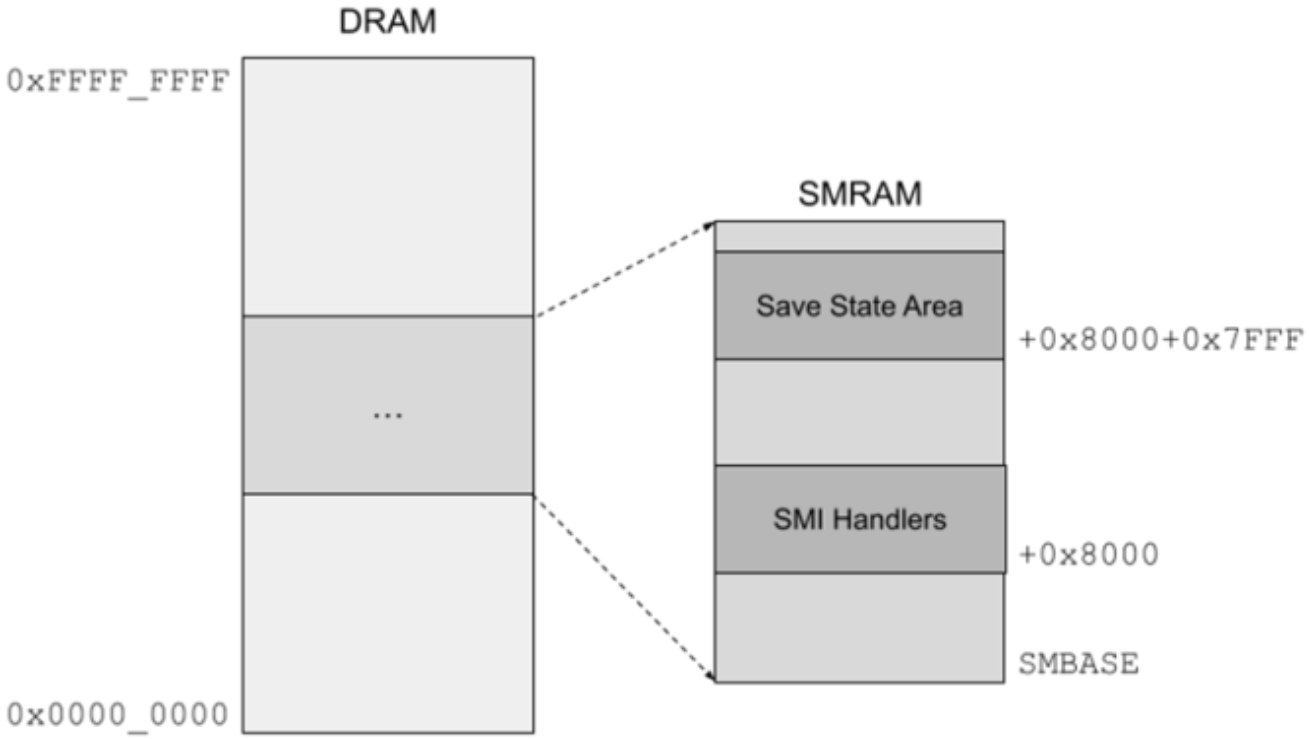
Before diving into the bug details, let's first take a brief detour to talk about System Management Mode. SMM is a highly privileged x86 operating mode. It has a variety of purposes, including control of hardware and peripherals, handling hardware interrupts, power management, and more. SMM is sometimes referred to as "Ring -2" using the protection ring nomenclature.



x86 Protection Levels

A CPU transitions to System Management Mode when a System Management Interrupt (SMI) is issued. A SMI can be generated from hardware or from software, such as by writing to an IO port. These interrupts are high priority and are unmaskable (e.g., they can't be ignored).

SMM executes from a protected region of memory known as System Management RAM (SMRAM). The System-Management Range Register (SMRR) can be (***ahem** should be*) programmed to restrict access to the SMRAM region, preventing external agents from accessing SMRAM. In other words, the OS should not be able to read or write SMRAM to directly influence SMM execution.



SMRAM Layout

SMM execution is transparent to the operating system. While a SMI handler is executing, the so-called SMI Rendezvous procedure will cause the other CPU cores to also enter SMM and wait. The OS can't see or inspect what SMM is doing.

But on the other hand, SMM can influence OS execution. SMM has (*nearly*) full access to the platform's DRAM. I say *nearly* here, because there are a few exceptions, such as certain DRAM carveouts that are owned by the even-more-highly-privileged firmware IPs, like AMD's PSP or Intel's CSME.

Beyond near-complete access to physical memory, SMM possesses additional powerful capabilities: It has full access to the platform's SPI flash, and it can read/write all MSRs.

For these reasons, SMM is a desirable location for attackers to implant a bootkit. Such a bootkit will be simultaneously invisible to most anti-virus software and will also be highly privileged. If you want to read more on the topic of bootkits, Alex Matrosov has done an excellent job of documenting some examples. You might also be curious to check out the SmmBackdoor project.

One of the most essential security requirements for preventing runtime exploitation of SMM is that the integrity of SMRAM must be upheld. In other words: *Simply don't do memory corruption*. But as we know, this is a tall order, especially because SMM firmware is written in C, where undefined behavior runs rampant and upholding memory safety is akin to the delicate circus act of balancing several spinning plates.

So unsurprisingly, over the years there have been countless examples of memory corruption vulnerabilities in SMM. For further reading, I encourage you to check out Xeno Kovah's catalogue of Low Level PC/Server Attacks for an impressive timeline of SMM vulnerability research (among other cool firmware security topics!).

SMM Attack Surfaces

Within SMM, individual SMI handlers are registered using the `gSmst->SmiHandlerRegister()` function. Each handler has a unique GUID, which is used to select the appropriate handler when the OS invokes a SMI.

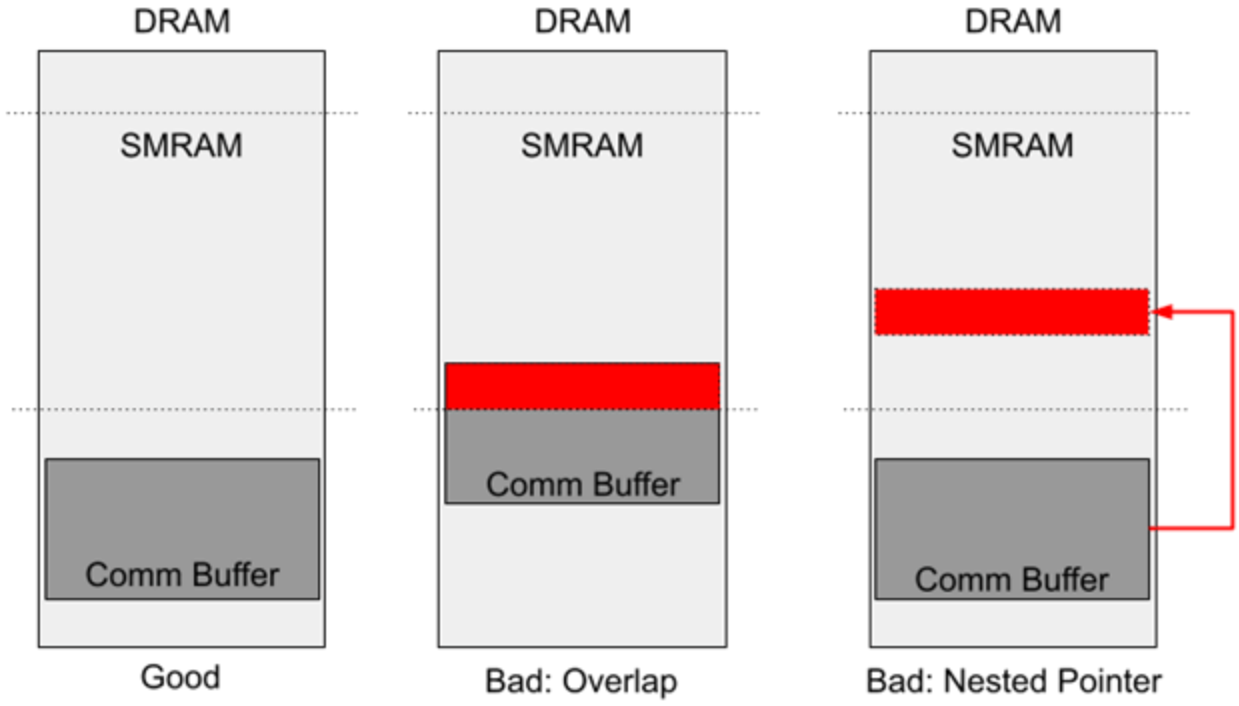
Arguments can be passed to the SMI handlers via a Communication Buffer in shared memory. Strict **input validation** of all arguments passed to a SMI handler is paramount to preserve the property of memory safety.

Another attack surface relates to various platform resources that are shared between SMM and other agents such as the host OS, peripherals, and firmware IPs. Here, **race conditions** such as time-of-check-time-of-use (TOCTOU) problems are also a significant concern. Some typical examples of shared resources that are consumed by SMM include the following:

- SPI flash (e.g., EFI variables)
- Memory-Mapped I/O (e.g., PCIe BARs)
- Shared physical memory regions (e.g., the SMI Comm Buffer)
- Model Specific Registers (MSRs)

Because these resources can be shared between multiple agents of differing privilege levels, a malicious low-privilege agent could tamper with the shared data while SMM is in the midst of processing it.

Another notable vulnerability class in SMM is the **confused deputy**. Confused deputy problems can occur when an attacker passes a pointer argument to SMM (e.g., the Comm Buffer) but forces the buffer to overlap with SMRAM. If the SMI handler fails to validate the pointer (don't forget nested pointers too!), it may mistakenly read or write its own address space, believing it is reading SMI input or writing SMI output. This, of course, would have the undesirable result of corrupting SMRAM.



Communication Buffer Overlap

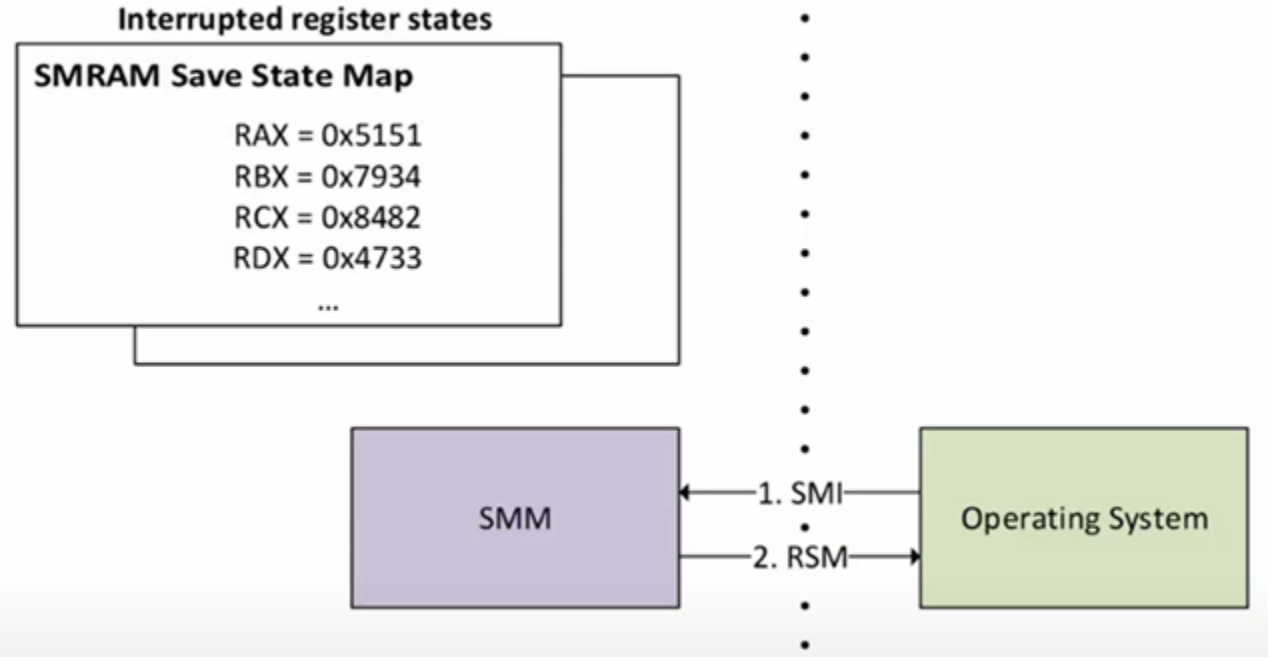
If you want to read more on these topics, check out the "A Tour Beyond BIOS: SMM Communication" whitepaper for an in depth description of these and other vulnerability classes that relate to SMM.

Finally, I want to add that Microsoft's "Secured-Core PC" initiative is beginning to push the industry towards stronger SMM hardening through the use of an SMM Supervisor, which effectively deprivileges and isolates SMI handlers. Though, like most defensive technologies, creative people will find ways to break it. For example, last year Ilja van Sprundel of IOActive presented some excellent research that reveals several critical vulnerabilities in Microsoft's MM Supervisor which is part of Project Mu.

The Focus of My Research

SMI handlers typically receive input arguments via the Communication Buffer, which resides in a region of shared memory that may be statically or dynamically defined. As mentioned above, the Comm Buffer must be positioned outside of SMRAM, and it is the duty of SMM to enforce this every time a SMI is handled.

However, SMI handlers may also receive arguments through general purpose registers. So how does that work? Well, when an SMI is issued by the OS, the processor state is saved, and execution context is switched to SMM. The saved general purpose registers reside inside SMRAM within the State Save Area. All of this is necessary because when a SMI handler completes, CPU state must be restored so that execution control can be returned to the caller.



High Level SMI Flow (from ABC to XYZ of SMM Drivers)

Of course, a malicious or compromised host OS could place any values in these registers prior to invoking the SMI. Per SMM's threat model, the OS is completely untrusted, so the SMI handlers must be extremely cautious to validate all data that is read from the Save State Area.

For my research, I focused on the Insyde H2O (Hardware-2-Operating System) UEFI BIOS, which exposes an SMI interface named IHISI (Insyde H2O Internal Soft-SMI Interface). This interface is made up of many sub-commands which read and write these saved state registers, treating them as arguments to the sub-command handlers.

Let's dive into the bug details!

Vulnerability Details

All these vulnerabilities share a common root cause (insufficient input validation) and a common impact (SMRAM corruption). Their details are summarized in the following table:

#	Title	CVE	Insyde	CVSS
1	IhisiServicesSmm: Save State Register Not Checked Before Use	CVE-2023-22616	SA-2023022	6.4
2	IhisiServicesSmm: Memory Corruption in FTBS SMI Handler	CVE-2023-22612	SA-2023019	8.1
3	IhisiServicesSmm: IHISI Subfunction Execution May Corrupt SMRAM.	CVE-2023-22615	SA-2023021	6.4
4	IhisiServicesSmm: Write To Attacker Controlled Address	CVE-2023-22613	SA-2023023	7.3
5	ChipsetSvcSmm: Insufficient Input Validation In BIOS Guard Updates	CVE-2023-22614	SA-2023020	7.9

These issues were fixed in the Insyde release which occurred on April 10th 2023. They impact several different Insyde platforms, spanning Intel and AMD mobile and server devices. The specific platforms and versions can be found in the Insyde advisories, linked above.

Bug 1. IhisiServicesSmm: Save State Register Not Checked Before Use

The following SMI handler is an IHISI sub-function that is associated with Insyde's Flash BIOS Through SMI (FTBS) functionality. The handler reads a structure pointer named `BiosRomMap` from RDI in the Save State Area.

```
EFI_STATUS EFIAPI FbtsGetWholeBiosRomMap ( VOID )
{
  UINTN                                 RomMapSize;
  UINTN                                 NumberOfRegions;
  FBTS_INTERNAL_BIOS_ROM_MAP           *BiosRomMap;
  UINTN                                 Indxe;

  NumberOfRegions = 0;
  BiosRomMap  = (FBTS_INTERNAL_BIOS_ROM_MAP *) (UINTN)
                  mH2OIhisi->ReadCpuReg32 (EFI_SMM_SAVE_STATE_REGISTER_RDI);
  ...
```

This pointer is not validated before it is dereferenced for both read and write operations. A confused deputy vulnerability arises when the caller forces RDI to point to SMRAM. This effectively coerces SMM into mistakenly accessing its own private memory space.

Next, the `BiosRomMap` array is walked to count the `NumberOfRegions`, which influences the for-loop sentinel condition, potentially allowing `Indxe` (sic) to accumulate to a large integer value. Together, these missing input validation problems may allow an attacker to corrupt SMRAM on the lines below:

```
  ...
  while (BiosRomMap[NumberOfRegions].Type != FbtsRomMapEos) {
    NumberOfRegions++;
  }
  NumberOfRegions++;

  RomMapSize = NumberOfRegions * sizeof (FBTS_INTERNAL_BIOS_ROM_MAP);
  for (Indxe = 0; Indxe < (NumberOfRegions - 1); Indxe++) {
    BiosRomMap[Indxe].Address =  BiosRomMap[Indxe].Address
                                 - PcdGet32(PcdFlashAreaBaseAddress)
                                 + PcdGet32(PcdFlashPartBaseAddress);
  }
  ...
```

Finally, before returning, the saved RDI register is used to copy the updated `BiosRomMap` back to the caller who invoked the SMI handler.

```
  ...
  CopyMem ((VOID *)(UINTN)mH2OIhisi->ReadCpuReg32 (EFI_SMM_SAVE_STATE_REGISTER_RDI),
           (VOID *)BiosRomMap,
           RomMapSize);
  return IHISI_SUCCESS;
}
```

But once again, because RDI was not previously checked to prevent overlap with SMRAM, this `CopyMem` operation could overwrite SMRAM.

Bug 2. IhisiServicesSmm: Memory Corruption in FTBS SMI Handler

The Insyde IHISI exposes a sub-command (AH=0x48) which is handled by the following function.

The SMI handler receives attacker-controlled input through the save-state register, RSI. Below, `ImageBlkPtr` is tainted by the caller, and is dereferenced without checking whether it overlaps SMRAM. Additionally, the nested pointer, `ImageBlock`, is also dereferenced without checking for SMRAM overlap.

```
EFI_STATUS SecureFlashFunction ( VOID )
{
  ...
    ImageBlkPtr = (FBTS_SECURE_FLASH_IMAGE_BLOCK_STRUCTURE*)(UINTN)
                        IhisiProtReadCpuReg32 (EFI_SMM_SAVE_STATE_REGISTER_RSI);

    ImageBlock = ImageBlkPtr->BlockDataItem;
    ImageBase = (UINT8*)(UINTN)(ImageBlock->ImageBlockAddress);
  ...
```

Next, the inner-most pointer named `ImageBase` is finally checked to ensure it doesn't overlap SMRAM. But when checking for overlap, the call to `IhisiProtBufferInCmdBuffer()` uses the `ImageBlock->ImageBlockSize` value, which also happens to be attacker controlled. This effectively allows this sanity check to be easily circumvented.

```
  ...
    if (!IhisiProtBufferInCmdBuffer ((VOID *)ImageBase, (UINTN)(ImageBlock-
>ImageBlockSize)))
    {
      mFlashImageInfo.RemainingImageSize = 0;
      return IHISI_BUFFER_RANGE_ERROR;
    }
  ...
```

Later in the SMI handler, `MergeImageBlockWithoutCompress()` is called. This function also reads the RSI save-stage register to get the `ImageBlkPtr` pointer. This time, the function **does** check whether the pointer overlaps SMRAM, but it does so only after dereferencing it. This dereference-then-validate pattern is most likely only an uninteresting denial of service.

```
EFI_STATUS MergeImageBlockWithoutCompress (
  IN EFI_PHYSICAL_ADDRESS       TargetImageAddress
  )
{
  ...
  TotalImageSize = mFlashImageInfo.TotalImageSize -
mFlashImageInfo.RemainingImageSize;
  ImageBlkPtr = (FBTS_SECURE_FLASH_IMAGE_BLOCK_STRUCTURE*)(UINTN)
                 IhisiProtReadCpuReg32 (EFI_SMM_SAVE_STATE_REGISTER_RSI);
  ...
  NumberOfImageBlk = ImageBlkPtr->BlockNum;
  if (!IhisiProtBufferInCmdBuffer ((VOID *) ImageBlkPtr, NumberOfImageBlk)) {
    return IHISI_BUFFER_RANGE_ERROR;
  }
  ...
```

However, what is more interesting is the usage of the `ImageBlock` pointer because we know from earlier analysis that this pointer is attacker controlled. If it points into attacker-controlled memory, it is subject to TOCTOU vulnerabilities. As a result, `ImageBlock->ImageBlockSize` can change between the several dereferences, shown below.

```
...
ImageBlock = ImageBlkPtr->BlockDataItem;
...
Destination = (UINT8 *) (UINTN) (TargetImageAddress + TotalImageSize);
for (Index = 0; Index < NumberOfImageBlk; Index++) {
  if (!FeaturePcdGet(PcdH2OIhisiCmdBufferSupported)
      ImageBlock->ImageBlockSize > UTILITY_ALLOCATE_BLOCK_SIZE)
  {
    // The max block size need co-operate with utility
    return EFI_INVALID_PARAMETER;
  }

  CopyMem ((VOID *) Destination,
           (UINT8 *)(UINTN) ImageBlock->ImageBlockAddress,
           (UINTN) ImageBlock->ImageBlockSize);
  ...
}
```

If a DMA-capable attacker wins this race condition, they can modify `ImageBlock->ImageBlockSize` after it has been validated but before it is used in the `CopyMem()` call. This results in corruption of memory beyond the end of the `Destination` memory region.

Curiously, the `Destination` pointer was originally obtained from the "SecureFlashInfo" EFI variable (not shown for the sake of brevity), which is stored with the BS+RT+NV attributes, indicating that its value is also controllable by a malicious host OS.

In conclusion, this means that the attacker controls the destination address, source address and size parameters that are passed to `CopyMem()`. This is a powerful write-what-where memory corruption primitive.

Bug 3. IhisiServicesSmm: IHISI Subfunction Execution May Corrupt SMRAM

The following code block shows the main IHISI subfunction dispatcher. It walks a table of subfunctions, finds a registered subfunction that matches the command code, and then invokes the handler function, as shown below:

```
EFI_STATUS EFIAPI IhisiProtExecuteCommandByPriority (
  IN UINT32         CmdCode,
  IN UINT8          FromPriority,
  IN UINT8          ToPriority
  )
{
  EFI_STATUS                    Status;
  LIST_ENTRY                   *Link;
  IHISI_COMMAND_ENTRY          *CmdNode;
  IHISI_FUNCTION_ENTRY         *FunctionNode;

  CmdNode = IhisiFindCommandEntry (CmdCode);
  ...
  for (Link = GetFirstNode ( CmdNode->FunctionChain);
       !IsNull ( CmdNode->FunctionChain, Link);
       Link = GetNextNode ( CmdNode->FunctionChain, Link))
  {
    FunctionNode = IHISI_FUNCTION_ENTRY_FROM_LINK (Link);
    if (FunctionNode->Priority > ToPriority || FunctionNode->Priority < FromPriority)
{
      continue;
    }
    Status = FunctionNode->Function();
    ...
```

After the subfunction returns, and if the `CmdCode` is equal to `OEMSFOEMExCommunication`, the contents of the communication buffer will be copied back to the caller as the SMI output. The destination address for this `CopyMem()` operation is decided by the caller of the SMI handler because it was passed in the RCX save state register.

```
    ...
    if (CmdCode == OEMSFOEMExCommunication) {
      CopyMem( (AP_COMMUNICATION_DATA_TABLE*) (UINTN)
                  IhisiProtReadCpuReg32 (EFI_SMM_SAVE_STATE_REGISTER_RCX),
               mApCommDataBuffer,
              sizeof (AP_COMMUNICATION_DATA_TABLE) );
    }
    ...
```

The problem here is that when an attacker controls the contents of RCX, they can coerce the above code to write the `mApCommDataBuffer` to an attacker-controlled location in SMRAM.

In evaluating the impact of this, we must check whether each and every IHISI subfunction properly validates RCX before returning to the dispatcher. The relevant subfunctions that are associated with the `OEMSFOEMExCommunication` command code are listed below:

```
STATIC IHISI_REGISTER_TABLE OEM_EXT_COMMON_REGISTER_TABLE[] = {
  { OEMSFOEMExCommunication, "S41Kn_CommuSaveRegs", KernelCommunicationSaveRegs
},
  { OEMSFOEMExCommunication, "S41Cs_ExtDataCommun", ChipsetOemExtraDataCommunication
},
  { OEMSFOEMExCommunication, "S41OemT01Vbios00000", OemIhisiS41T1Vbios
},
  { OEMSFOEMExCommunication, "S41OemT54LogoUpdate", OemIhisiS41T54LogoUpdate
},
  { OEMSFOEMExCommunication, "S41OemT55CheckSignB",
OemIhisiS41T55CheckBiosSignBySystemBios },
  { OEMSFOEMExCommunication, "S41OemReservedFun00", OemIhisiS41ReservedFunction
},
  { OEMSFOEMExCommunication, "S41Kn_T51EcIdelTrue", KernelT51EcIdelTrue
},
  { OEMSFOEMExCommunication, "S41Kn_ExtDataCommun", KernelOemExtraDataCommunication
},
  { OEMSFOEMExCommunication, "S41Kn_T51EcIdelFals", KernelT51EcIdelFalse
},
  { OEMSFOEMExCommunication, "S41OemT50Oa30RWFun0", OemIhisiS41T50a30ReadWrite
},
  ...
```

After careful inspection, we determined that most of these IHISI subfunctions perform strict validation of the pointer stored in RCX. For example, the first handler, `KernelCommunicationSaveRegs()` is shown below. Here, we can observe that `ApCommDataBuffer` (the pointer that was read from RCX) is checked to ensure that it correctly resides inside the Comm Buffer.

```
EFI_STATUS EFIAPI KernelCommunicationSaveRegs ( VOID )
{
  AP_COMMUNICATION_DATA_TABLE      *ApCommDataBuffer;
  UINTN                            BufferSize;

  mRomBaseAddress = 0;
  mRomSize        = 0;
  ApCommDataBuffer = (AP_COMMUNICATION_DATA_TABLE*) (UINTN)
                       IhisiProtReadCpuReg32 (EFI_SMM_SAVE_STATE_REGISTER_RCX);

  if (!IhisiProtBufferInCmdBuffer ((VOID *) ApCommDataBuffer,
                                   sizeof(AP_COMMUNICATION_DATA_TABLE)))
  {
    return IHISI_BUFFER_RANGE_ERROR;
  }
  ...
  BufferSize = ApCommDataBuffer->StructureSize;
  if (BufferSize < sizeof(AP_COMMUNICATION_DATA_TABLE)) {
    BufferSize = sizeof(AP_COMMUNICATION_DATA_TABLE);
  }
  if (!IhisiProtBufferInCmdBuffer ((VOID *) ApCommDataBuffer, BufferSize)) {
    return IHISI_BUFFER_RANGE_ERROR;
  }
  ...
}
```

However, there are two subfunctions that do not validate RCX:

- `KernelT51EcIdelTrue()`
- `KernelT51EcIdelFalse()`

This oversight is most likely a consequence of the fact that these subfunctions do not use RCX, so perhaps the developer assumed it was not necessary to validate RCX. However, even though these subfunctions never use RCX, the `IhisiProtExecuteCommandByPriority()` dispatcher will still use RCX as the destination address for a `CopyMem()` operation.

Therefore, if an attacker set an address in RCX that overlapped SMRAM before invoking the `S41Kn_T51EcIdelTrue` or `S41Kn_T51EcIdelFalse` subfunctions, they could corrupt SMRAM with the contents of the AP communication buffer.

Bug 4. IhisiServicesSmm: Write To Attacker Controlled Address

The following SMI handler reads a structure pointer named `OutputData` from the RCX save state register, as shown below:

```
STATIC EFI_STATUS ReadDefaultSettingsToFactoryCopy ( VOID )
{
  OUTPUT_DATA_STRUCTURE          *OutputData;
  UINT64                         FactoryCopySize;

  OutputData = (OUTPUT_DATA_STRUCTURE *) (UINTN)
                  IhisiProtReadCpuReg32 (EFI_SMM_SAVE_STATE_REGISTER_RCX);
  ...
```

The SMI handler then performs writes to fields in this structure without validating `OutputData` for overlap with SMRAM.

```
  ...
  OutputData->BlockSize = COMMON_REGION_BLOCK_SIZE_4K;

  FactoryCopySize =  FdmGetSizeById (...);
  ...
  if (FactoryCopySize == 0x10000) {
    OutputData->DataSize = COMMON_REGION_SIZE_64K;
  } else {
    OutputData->DataSize = COMMON_REGION_REPORT_READ_SIZE;
    OutputData->PhysicalDataSize = (UINT32) FactoryCopySize;
  }
  ...
```

At the risk of sounding like a broken record: Once again, this is a straightforward SMM memory corruption vulnerability.

Bug 5. ChipsetSvcSmm: Insufficient Input Validation In BIOS Guard Updates

BIOS Guard is a security feature under the Intel's "Hardware Shield" marketing umbrella. It hardens the BIOS flash update process by restricting access to SPI flash via the BIOS Guard ACM, which authenticates BIOS updates. There's little public documentation on BIOS Guard, but this talk reveals some design aspects that Alex recovered by reverse engineering. The following vulnerability affects Insyde's SMM module which parses the BIOS Guard Update Header, whose layout is shown below:

Below, the `InputDataBuffer` is read from RSI, and points to the above BIOS Guard update structure. This pointer is dereferenced to calculate the BIOS Guard certificate offset (`BgupcOffset`) without first checking whether the pointer overlaps SMRAM. Because `ScriptSectionSize` and `DataSectionSize` (both `UINT32` types) are tainted, `BgupcOffset` should also be considered tainted, and can take on any 32-bit integer value.

```
EFI_STATUS BiosGuardUpdateWrite (
VOID )
{
  ...
  UINT32
BgupcSize;
  UINT32
BgupcOffset;
  UINT32
BufferSize;
  EFI_PHYSICAL_ADDRESS
BgupCertificate;
  UINT8
*InputDataBuffer;
  UINT32
DataSize;
  ...

  InputDataBuffer = (UINT8*)
(UINTN)mH2OIhisi->ReadCpuReg32
(EFI_SMM_SAVE_STATE_REGISTER_RSI);
  BgupcOffset = sizeof(BGUP_HEADER)
                  + ((BGUP *)
InputDataBuffer)-
>BgupHeader.ScriptSectionSize
                  + ((BGUP *)
InputDataBuffer)-
>BgupHeader.DataSectionSize;
  ...
```

BIOS Guard Update Structure

Next, `BufferSize` is read from RDI,
and it is used to check whether the input buffer resides within the command buffer. However, this code is lacking strict checks to ensure that `BufferSize` is sufficiently large. If `BufferSize` happened to be smaller than the size of the `BGUP_HEADER` structure, then the earlier pointer dereferences (when reading members from `BgupHeader`) might access memory beyond the bounds of the input buffer, leading to an out-of-bounds read.

```
  ...
  BufferSize      =  mH2OIhisi->ReadCpuReg32 (EFI_SMM_SAVE_STATE_REGISTER_RDI);
  BgupCertificate = (EFI_PHYSICAL_ADDRESS) (mBiosGuardMemAddress
                       + mBiosGuardMemSize
                       - BGUPC_MEMORY_OFFSET);

  if (!mH2OIhisi->BufferInCmdBuffer ((VOID *) InputDataBuffer, BufferSize)) {
    return EFI_INVALID_PARAMETER;
  }
  ...
```

Then, `BgupcSize` is checked to ensure it is consistent with `BufferSize`. However, this sanity check can also be bypassed because the attacker controls both sides of the conditional expression — both `BgupcOffset` and `BufferSize`.

```
...
if ((BgupcOffset + BgupcSize) != BufferSize) {
  return EFI_INVALID_PARAMETER;
}
...
```

The last step taken before triggering the BIOS Guard ACM is to use the attacker-controlled `BgupcOffset` (which can be very large) to copy the certificate and update data. This is shown below:

```
...
ZeroMem ((VOID *)(UINT64) mBiosGuardMemAddress, mBiosGuardMemSize);
CopyMem ((VOID *)(UINT64) mBiosGuardMemAddress, InputDataBuffer, BgupcOffset);
CopyMem ((VOID *) BgupCertificate, InputDataBuffer + BgupcOffset, BgupcSize);
...
```

The above `CopyMem()` calls can lead to corruption of SMRAM when memory beyond the end of the `mBiosGuardMemAddress` region is overwritten.

Thanks

I would like to thank Insyde PSIRT, and in particular, Kevin Davis, for being a pleasure to work with during this disclosure period.