

ЛАБОРАТОРНАЯ РАБОТА № 2

«Реверс-инжиниринг SMM модулей UEFI BIOS»

по дисциплине «Принципы построения, проектирования и эксплуатации
автоматизированных информационных систем»

Выполнил
студент гр. 4851003/70801

Гасанов Э.А.

<подпись>

Преподаватель
старший преподаватель

Чернов А.Ю.

<подпись>

Цель

Получить базовые навыки реверс-инжиниринга встроенного ПО UEFI BIOS на примере SMM-модулей.

Задачи

1. Изучить процесс инициализации SMM-режима в рамках функционирования фаз загрузки ЭВМ, построенной на базе UEFI BIOS (глава в отчет)
2. Выявить программные компоненты прошивки UEFI BIOS, участвующие в ходе инициализации SMM-режима. (глава в отчет)
3. Описать возможные способы перехода CPU в SMM-режим, соответствующие точки входа и обработчики (глава в отчет)
4. Произвести реверс-инжиниринг 3-х произвольных обработчиков #SWSMI в соответствии с примером (по 3 на каждого студента, если работа ведется в команде их 2-х человек). По каждому модулю нужно приложить IDB файл IDAPro + описать вектора воздействия на обработчик прерываний.

Исходные данные

Материнская плата ASUS UX310UAK-AS.312, AMI

- Intel Core i5 7200U (Kaby Lake – chipset C422)
- Версия BIOS 312; 4/18/2019
- Размер – 6 МБ, так как это урезанный файл обновления (по умолчанию должно быть 8 (8 390 656 байт))
- 8 Гб оперативной памяти; 2 ядра и 4 логических процессора
- Используется Ida Pro v.7.6
- Используется плагин для Ida Pro – efiXplorer и efiXloader

Ход работы

После того, как был инициализирован DXE Foundation (передачей ему NOB-list, и инициализацией таблиц UEFI Boot Services Table, UEFI Runtime Services Table и DXE Services Table), контроль передаётся в DXE Dispatcher.

SMM фаза запускается в DXE и продолжает работать параллельно с другими фазами инициализации (вплоть до выключения компьютера).

Способ, согласно которому DXE Dispatcher загружает и запускает DXE драйвера, это смесь строгого и слабого порядка запуска (mix of strong and weak orderings). Строгий порядок запуска определяется с помощью а priori file (рисунок 1).

Action	Type	Subtype	Text
capsule	Capsule	Aptio signed	
age	Image	UEFI	
FC-AF1D-4E5D-BDC5-DACD6D278AEC	Padding	Non-empty	
9D26F-1A11-4988-B91F-858745CF824	Volume	FFSV2	NVAR store
9D26F-1A11-4988-B91F-858745CF824	File	Raw	StdDefaults
9D26F-1A11-4988-B91F-858745CF824	NVAR entry	Full	
Free space	Free space		
Padding	Padding	Non-empty	
Volume	Volume	Empty (0xFF)	
File	File	FFSV2	
Section	Section	Freeform	
File	File	Freeform subtype GUID	
Section	Section	Freeform	
File	File	Freeform subtype GUID	
Volume image	Volume image	GUID defined	
Section	Section	Raw	
Volume image	Volume image	FFSV2	
File	File	Freeform	DXE a priori file

Рисунок 1 – Содержимое а priori файла

Как видно из рисунка 1 перечисленные драйвера запускаются перво-наперво и именно в указанном порядке. Слабый порядок запуска определяется dependency expressions (DepEx) в DXE драйвере (рисунок 2). Это означает, что драйвер имеет зависимости. DepEx реализован через упрощенный стек, поэтому в нём используются команды PUSH, AND, BEFORE, END и тд. А если зависимостей нет, то и стек состоит из TRUE и END.

Action	Type	Subtype	Text
79F086B3-D74F-4D6F-9577-F8E0629770...	File	DXE driver	SerialDebugInitDxe
2F0868D0-1A47-43F7-8D68-D0C1F25142...	File	DXE driver	SmbusDebugDxe
614778F8-885F-438E-9674-52DE7E32EA...	File	DXE driver	ChargeLcdDebugDxe
5C403B01-8C30-4E68-96A0-55612F9C32...	File	DXE driver	AsusEcDxeS
89F639B0-092D-4952-8EC8-F1F05FE340...	File	DXE driver	OemActivation
4D883B45-9C48-41ED-9FC7-C3FC00267B...	File	DXE driver	CountryCode
B51AF856-7196-433C-885E-C7DEC83842...	File	DXE driver	Manufacture
36AD18AA-241A-4316-811F-220CEDCF15...	File	DXE driver	AutoTpm
FC740D58-59BA-4298-99EF-6270517378...	File	DXE driver	AsusImageDecoder
PEFirmwareUpdateEfi	File	DXE driver	PEFirmwareUpdateDXE
3B749932-2DE6-4021-A639-5753536A4F...	File	DXE driver	AsusSensor
98857AE4-2029-4EC0-97F9-C9E7C4BF3D...	File	DXE driver	AsusSerialIO
A2F03EEA-9304-40BE-87B2-7F45780123...	File	DXE driver	AsusSmbios
ResLoader	File	DXE driver	ResLoader
91A44023-5DCC-4425-91E6-42DA616408...	File	DXE driver	AsusBootDeviceSkip
0C805087-D2F0-42F8-8528-D1DEB2F2E9...	File	DXE driver	SetupIoDetect
CapsuleRuntimeDxe	File	DXE driver	CapsuleRuntimeDxe
RuntimeDxe	File	DXE driver	RuntimeDxe
PiSmmIpl	File	DXE driver	PiSmmIpl
DXE dependency section	Section	DXE dependency	
PE32 image section	Section	PE32 image	
UI section	Section	UI	
Version section	Section	Version	

Рисунок 2 – DepEx Initial Program Load (IPL) SMM

Считаем, что PiSmmIpl является драйвером в DXE, который начинает инициализацию режима SMM.

SMM фаза состоит из двух частей:

1. SMRAM инициализация – начинается в фазе DXE (в DXE-драйвере PiSmmIpl), она “открывает” SMRAM и создаёт SMRAM карту памяти и предоставляет необходимые сервисы для запуска уже SMM-драйверов. Для этого необходимо использовать EFI_SMM_CONFIGURATION_PROTOCOL. Он указывает, какие области в SMRAM зарезервированы для использования ЦП для целей: стек, сохранение состояния процессора, точка входа SMM.
2. Управление SMI – при генерации SMI создается среда выполнения драйвера, затем обнаруживаются источники SMI и вызываются обработчики SMI.

Таким образом, чтобы попасть в SMM используются прерывания SMI двух типов:

- Software System Management Interrupt (SwSMI)
- Hardware System Management Interrupt (HwSMI)

EFI_SMM_ACCESS2_PROTOCOL — описывает различные SMRAM регионы доступные в системе.

EFI_SMM_CONTROL2_PROTOCOL — используется для инициации синхронных SMI прерываний.

EFI_SMM_BASE2_PROTOCOL — используется, чтобы обнаружить System Management Services Table (SMST).

EFI_SMM_CONFIGURATION_PROTOCOL — индикация, какие области в SMRAM будут использоваться под стек, сохранения состояния или точка входа в SMM.

EFI_SMM_COMMUNICATION_PROTOCOL — предоставляет средства связи между драйверами вне SMM и обработчиками SMI внутри SMM.

EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL — указание, что SMM вот-вот заблокируется. Для регистрации этого протокола вызывается функция EFI_SMM_ACCESS2_PROTOCOL.Lock () для блокировки SMRAM.

Порядок вызовов протоколов происходит согласно стеку DepEx из рисунка 2.

Таблица 1 – Точка входа в драйвер

Дизассемблер	TianoCore - PiSmmIpl.c
<pre> unsigned __int64 __fastcall sub_1EB4(__int64 a1) { qword_4138 = a1; gBS_43E0->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID_40A0, 0i64, (void **)&::Interface); gBS_43E0- >LocateProtocol(&EFI_SMM_CONTROL2_PROTOCOL_GUID_40C0, 0i64, &qword_4480); Buffer = (void *)sub_1AE0(&qword_4140); // функция описана в таблице 2 Тогда a1= ImageHandle Тогда Interface= mSmmAccess Тогда qword_4140= gSmmCorePrivate->SmramRangeCount Тогда Buffer = gSmmCorePrivate->SmramRanges </pre>	<pre> EFI_STATUS EFIAPI SmmIplEntry (IN EFI_HANDLE ImageHandle, IN EFI_SYSTEM_TABLE *SystemTable) { // // Fill in the image handle of the SMM IPL so the SMM Core can use this as the ParentImageHandle field of the Load Image Protocol for all SMM Drivers loaded by the SMM Core // mSmmCorePrivateData.SmmIplImageHandle = ImageHandle; // // Get SMM Access Protocol // Status = gBS->LocateProtocol (&gEfiSmmAccess2ProtocolGuid, NULL, (VOID ***)&mSmmAccess); ASSERT_EFI_ERROR (Status); // // Get SMM Control2 Protocol // Status = gBS->LocateProtocol (&gEfiSmmControl2ProtocolGuid, NULL, (VOID ***)&mSmmControl2); ASSERT_EFI_ERROR (Status); gSmmCorePrivate->SmramRanges = GetFullSmramRanges (&gSmmCorePrivate->SmramRangeCount); // // Open all SMRAM ranges // Status = mSmmAccess->Open (mSmmAccess); // SMRAM window is now open. </pre>
<pre> (*(void (__fastcall **)(EFI_TPL))mSmmAccess)(mSmmAccess); </pre>	<pre> // // Find the largest SMRAM range between 1MB and 4GB that is at least 256KB - 4K in size // mCurrentSmramRange = NULL; for (Index = 0, MaxSize = SIZE_256KB - EFI_PAGE_SIZE; Index < gSmmCorePrivate->SmramRangeCount; Index++) { // // Skip any SMRAM region that is already allocated, needs testing, or needs ECC initialization // if ((gSmmCorePrivate->SmramRanges[Index].RegionState & (EFI_ALLOCATED EFI_NEEDS_TESTING EFI_NEEDS_ECC_INITIALIZATION)) != 0) { continue; } if (gSmmCorePrivate->SmramRanges[Index].CpuStart >= BASE_1MB) { </pre>
<p>Пытаемся найти самое большое “окно” в процессоре для SMRAM</p> <pre> v2 = 258048i64; if (!qword_4140) goto LABEL_33; v3 = qword_4140; v4 = (char *)Buffer + 16; do { if ((v4[8] & 0x70) == 0 && *((_QWORD *)v4 - 1) >= 0x100000ui64 && *(_QWORD *)v4 >= v2) { v2 = *(_QWORD *)v4; v1 = (__int64)(v4 - 16); } v4 += 32; } while (v3 < v2); </pre>	<pre> // // Find the largest SMRAM range between 1MB and 4GB that is at least 256KB - 4K in size // mCurrentSmramRange = NULL; for (Index = 0, MaxSize = SIZE_256KB - EFI_PAGE_SIZE; Index < gSmmCorePrivate->SmramRangeCount; Index++) { // // Skip any SMRAM region that is already allocated, needs testing, or needs ECC initialization // if ((gSmmCorePrivate->SmramRanges[Index].RegionState & (EFI_ALLOCATED EFI_NEEDS_TESTING EFI_NEEDS_ECC_INITIALIZATION)) != 0) { continue; } if (gSmmCorePrivate->SmramRanges[Index].CpuStart >= BASE_1MB) { </pre>

<pre>while (v3);</pre> <p>Таким образом находится место в памяти процессора под SMRAM, которое будет использоваться SMM IPL и SMM Core</p>	<pre>if ((gSmmCorePrivate->SmramRanges[Index].CpuStart + gSmmCorePrivate->SmramRanges[Index].PhysicalSize - 1) <= MAX_ADDRESS) { if (gSmmCorePrivate->SmramRanges[Index].PhysicalSize >= MaxSize) { MaxSize = gSmmCorePrivate->SmramRanges[Index].PhysicalSize; mCurrentSmramRange = &gSmmCorePrivate->SmramRanges[Index]; } } } }</pre>
<pre>qword_4498 является аналогом mCurrentSmramRange и имеет такой же тип данных - EFI_SMRAM_DESCRIPTOR v4 = (char *)Buffer + 16; // Buffer = gSmmCorePrivate->SmramRanges v1 = (__int64)(v4 - 16); qword_4498 = v1; Interface = 0i64; gBS_43E0->LocateProtocol(&EFI_CPU_ARCH_PROTOCOL_GUID_40E0, 0i64, &Interface)</pre> <pre>v12 = sub_15EC(qword_4498, (char *)Buffer + 32 * qword_4140 - 32, aSmmc);</pre> <pre>// qword_4140= gSmmCorePrivate->SmramRangeCount</pre> <p>Продолжение в таблице 4</p>	<pre>CpuArch = NULL; Status = gBS->LocateProtocol (&gEfiCpuArchProtocolGuid, NULL, (VOID **)&CpuArch);</pre> <pre>// // Load SMM Core into SMRAM and execute it from SMRAM // Эта функция описана в таблице 3 Status = ExecuteSmmCoreFromSmram (mCurrentSmramRange, &gSmmCorePrivate->SmramRanges[gSmmCorePrivate- >SmramRangeCount - 1], gSmmCorePrivate);</pre>

Функция GetFullSmramRanges заполучает диапазоны SMRAM с помощью SmmAccess и зарезервированные диапазоны SMRAM из протокола SmmConfiguration, разделяя записи, если между ними есть перекрытие. Также будет зарезервирована точка входа SMM core.

Таблица 2 – Функция GetFullSmramRanges

Дизассемблер	TianoCore - PiSmmIpl.c
<pre>__int64 __fastcall sub_1AE0(unsigned __int64 *a1) { v43 = 0i64; gBS_43E0- >LocateProtocol(&EFI_SMM_CONFIGURATION_PROTOCOL_GUID_40B0, 0i64, (void **)&v43); Тогда v43 – это SmmConfiguration</pre>	<pre>EFI_SMRAM_DESCRIPTOR * GetFullSmramRanges (OUT UINTN *FullSmramRangeCount) { // // Get SMM Configuration Protocol if it is present. // SmmConfiguration = NULL; Status = gBS->LocateProtocol (&gEfiSmmConfigurationProtocolGuid, NULL, (VOID **)&SmmConfiguration);</pre>
<pre>v3 = 0i64; if (!SmmConfiguration) // наоборот SmmConfig == NULL обратная логика goto LABEL_50; v4 = (_QWORD *)(*SmmConfiguration + 8i64); if (!*v4) goto LABEL_50; do { v4 += 2; ++v3; } while (*v4); Тогда v3 - SmramReservedCount</pre>	<pre>// // Get SMRAM reserved region count. // SmramReservedCount = 0; if (SmmConfiguration != NULL) { while (SmmConfiguration- >SmramReservedRegions[SmramReservedCount].SmramReservedSize != 0) { SmramReservedCount++; } }</pre>
<p>У меня декомпилировалось сразу в “иначе”</p> <p>То есть SMM Configuration Protocol даёт зарезервированную точку “вхождения” в SMRAM</p> <pre>if (SmramReservedCount) { v41 = 16 * (v2 + 2 * SmramReservedCount); Buffer = (void *)sub_221C(4i64); // sub_221C имеет AllocatePool v5 = 0i64; v6 = (char *)Buffer; v7 = SmramReservedCount; do</pre>	<pre>if (SmramReservedCount == 0) { // // No reserved SMRAM entry from SMM Configuration Protocol. // //...// return FullSmramRanges; } MaxCount = SmramRangeCount + 2 * SmramReservedCount; Size= MaxCount * sizeof (EFI_SMM_RESERVED_SMRAM_REGION); SmramReservedRanges = (EFI_SMM_RESERVED_SMRAM_REGION *)AllocatePool (Size); ASSERT (SmramReservedRanges != NULL);</pre>

```

{
    v8 = (char *) (v5 + *SmmConfiguration);
    if ( v6 != v8 )
        sub_3700( // содержит qтeмcпy == aнaлoг CoпyMeм
            v6,
            v8,
            16i64);

    v5 += 16i64;
    v6 += 16;
    --v7;
}
while ( v7 );

```

Тoгдa v2 - SmramRangeCount

Тoгдa v41 - Size

Buffer – SmramReservedRanges

v5= SmramRangeCount + 2 * SmramReservedCount

```

Size = 32 * v5;
v10 = sub_221C(4i64, 32 * v5); // sub_221C имeeт AllocatePool
v11 = 0i64;
v43 = 0i64;
v12 = v10;
v13 = sub_221C(4i64, Size); // sub_221C имeeт AllocatePool
v39 = v13;

```

Тoгдa v39=v13= SmramRanges

```

do
{
    v14 = 0i64;
    v15 = 0;
    v38 = 0i64;
    if ( !SmramRangeCount )
        break; // - Skip zero size entry.

```

oпyckaeм пoдpoбнocти пpoвepoк в циклe

```

for (Index = 0; Index < SmramReservedCount; Index++)
{
    CopyMem
    (
        &SmramReservedRanges[Index],
        &SmmConfiguration->SmramReservedRegions[Index],
        sizeof(EFI_SMM_RESERVED_SMRAM_REGION)
    );
}

```

```

Size = MaxCount * sizeof (EFI_SMRAM_DESCRIPTOR);
TempSmramRanges = (EFI_SMRAM_DESCRIPTOR *) AllocatePool
(Size);
ASSERT (TempSmramRanges != NULL);
TempSmramRangeCount = 0;

SmramRanges = (EFI_SMRAM_DESCRIPTOR *) AllocatePool (Size);

```

```

// split the entries if there is overlap between them
// разделяем точку входа в SMM если есть перекрытие
do
{
    Rescan = FALSE;
    for (Index = 0; (Index < SmramRangeCount) && !Rescan; Index++) {
        //
        // Skip zero size entry.
        //
        if (SmramRanges[Index].PhysicalSize != 0)
        {
            for (Index2 = 0; (Index2 < SmramReservedCount)
                && !Rescan; Index2++)

```


<pre> sub_1894((_DWORD)v16, (_DWORD)v19, (_DWORD)SmramRanges, (unsigned int)&v37, (__int64)v18, (__int64)v41, (__int64)v12, (__int64)&v43); SmramReservedCount = v41[0]; v18 = SmramReservedRanges; v15 = 1; } } } v23 = v11++; v43 = v11; v24 = (char *)&v12[4 * v23]; if (v24 != v16) sub_3700(v24, v16, 0x20ui64); *((_QWORD *)v16 + 2) = 0i64; LABEL_30: v14 = v38; } ++v14; v16 += 32; v38 = v14; } while (v14 < SmramRangeCount); v13 = SmramRanges; } while (v15); Тогда v15 = Rescan v12 = TempSmramRanges </pre>	<pre> { // // Skip zero size entry. // if (SmramReservedRanges[Index2].SmramReservedSize != 0) { if (SmmlsSmramOverlap (&SmramRanges[Index], &SmramReservedRanges[Index2])) { // // There is overlap, need to split entry and then rescan. // SmmSplitSmramEntry (&SmramRanges[Index], &SmramReservedRanges[Index2], SmramRanges, &SmramRangeCount, SmramReservedRanges, &SmramReservedCount, TempSmramRanges, &TempSmramRangeCount); Rescan = TRUE; } } } if (!Rescan) { // // No any overlap, // copy the entry to the temp SMRAM ranges. // Zero SmramRanges[Index].PhysicalSize = 0; // CopyMem (&TempSmramRanges[TempSmramRangeCount++], &SmramRanges[Index], sizeof (EFI_SMRAM_DESCRIPTOR)); SmramRanges[Index].PhysicalSize = 0; } } while (Rescan); </pre>
<pre> v25 = sub_224C(32 * (v11 + 1)); *a1 = 0i64; v26 = v25; do { v27 = 0i64; if (v11) { v28 = TempSmramRanges + 2; do { if (*v28) break; ++v27; v28 += 4; } while (v27 < v11); } } </pre>	<pre> // // Sort the entries // FullSmramRanges = AllocateZeroPool ((TempSmramRangeCount + AdditionSmramRangeCount) * sizeof (EFI_SMRAM_DESCRIPTOR)); ASSERT (FullSmramRanges != NULL); *FullSmramRangeCount = 0; do { for (Index = 0; Index < TempSmramRangeCount; Index++) { if (TempSmramRanges[Index].PhysicalSize != 0) { break; } } } ASSERT (Index < TempSmramRangeCount); for (Index2 = 0; Index2 < TempSmramRangeCount; Index2++) { </pre>

<pre> } v29 = 0i64; if (v11) { v30 = TempSmramRanges + 1; v31 = 32 * v27; do { if (v29 != v27 && v30[1] && *v30 < *(_QWORD *)((char *)TempSmramRanges + v31 + 8)) { v27 = v29; v31 = (__int64)v30 - 8 - (_QWORD)TempSmramRanges; } ++v29; v30 += 4; } while (v29 < v11); } v32 = (char *)&TempSmramRanges[4 * v27]; v33 = (char *) (v26 + 32 * *a1); if (v33 != v32) sub_3700(v33, v32, 0x20ui64); ++*a1; *(_QWORD *)v32 + 2 = 0i64; } while (*a1 < v11); v34 = gBS_43E0; ++*a1; v34->FreePool(v13); gBS_43E0->FreePool(SmramReservedRanges); gBS_43E0->FreePool(TempSmramRanges); return v26; </pre>	<pre> if ((Index2 != Index) && (TempSmramRanges[Index2].PhysicalSize != 0) && (TempSmramRanges[Index2].CpuStart < TempSmramRanges[Index].CpuStart)) { Index = Index2; } } CopyMem (&FullSmramRanges[*FullSmramRangeCount], &TempSmramRanges[Index], sizeof (EFI_SMRAM_DESCRIPTOR)); *FullSmramRangeCount += 1; TempSmramRanges[Index].PhysicalSize = 0; } while (*FullSmramRangeCount < TempSmramRangeCount); ASSERT (*FullSmramRangeCount == TempSmramRangeCount); *FullSmramRangeCount += AdditionSmramRangeCount; FreePool (SmramRanges); FreePool (SmramReservedRanges); FreePool (TempSmramRanges); return FullSmramRanges; </pre>
---	---

Таблица 3 – Запуск образа ядра SMM

Дизассемблер	PiSmmIpl.c
<pre> __int64 __fastcall sub_15EC (__int64 a1, _QWORD *a2, __int64 a3) { Каждый том прошивки драйвера должен создавать экземпляры Firmware Volume Protocol, если том микропрограммы должен быть видим для системы во время фазы DXE. v6 = gBS_43E0->LocateHandleBuffer(ByProtocol, &EFI_FIRMWARE_VOLUME2_PROTOCOL_GUID_40F0, 0i64, &NoHandles, &Buffer); v6 = gBS_43E0->HandleProtocol(Buffer[v8], &EFI_FIRMWARE_VOLUME2_PROTOCOL_GUID_40F0, &Interface); result = sub_28E8(&v24); if (result < 0) return result; v12 = v29 - 1; v13 = (((unsigned __int64)v29 + v25) >> 12) + (((v29 + v25) & 0xFFF) != 0) << 12; </pre>	<pre> // Load the SMM Core image into SMRAM and executes the SMM Core from SMRAM. EFI_STATUS ExecuteSmmCoreFromSmram (IN OUT EFI_SMRAM_DESCRIPTOR *SmramRange, // дескриптор //диапазона SMRAM для SMM Core IN OUT EFI_SMRAM_DESCRIPTOR *SmramRangeSmmCore, // Дескриптор диапазона SMRAM, содержащий SMM Core IN VOID *Context // контекст для передачи в SMM Core) Status = GetPeCoffImageFixLoadingAssignedAddress (&ImageContext); // Allocate memory for the image being loaded from the EFI_SRAM_DESCRIPTOR specified by SmramRange PageCount = (UINTN)EFI_SIZE_TO_PAGES ((UINTN)ImageContext.ImageSize + ImageContext.SectionAlignment); </pre>

<pre> typedef struct { EFI_PHYSICAL_ADDRESS PhysicalStart; // физический in DRAM EFI_PHYSICAL_ADDRESS CpuStart; // Адрес, который ЦП //использует для доступа к обработчику SMI UINT64 PhysicalSize; UINT64 RegionState; } EFI_SMRAM_DESCRIPTOR; v3[2] -= v13; v14 = v3[1] + v3[2]; a2[1] = v14; *a2 = *v3 + v3[2]; v15 = v3[3] 0x10i64; a2[2] = v13; a2[3] = v15; v24 = ~v12 & (v12 + v14); Тогда v24 – ImageContext Тогда v3 – SmramRange Тогда a2 - SmramRangeSmmCore </pre>	<pre> SmramRange->PhysicalSize -= EFI_PAGES_TO_SIZE (PageCount); SmramRangeSmmCore->CpuStart = SmramRange->CpuStart + SmramRange->PhysicalSize; SmramRangeSmmCore->PhysicalStart = SmramRange->PhysicalStart + SmramRange->PhysicalSize; SmramRangeSmmCore->PhysicalSize = EFI_PAGES_TO_SIZE (PageCount); SmramRangeSmmCore->RegionState = SmramRange->RegionState EFI_ALLOCATED; ImageContext.ImageAddress += ImageContext.SectionAlignment - 1; ImageContext.ImageAddress &= ~((EFI_PHYSICAL_ADDRESS)ImageContext.SectionAlignment - 1); </pre>
<pre> __int64 (__fastcall *v26)(__int64, EFI_SYSTEM_TABLE *); v6 = sub_2F6C(&ImageContext); if (v6 >= 0) { v6 = sub_2C9C(&ImageContext); if (v6 >= 0) { qword_4190 = (__int64)v26; v6 = v26(a3, gST_43D8); } } gBS_43E0->FreePool(v17); </pre>	<pre> // // Load the image to our new buffer // Status = PeCoffLoaderLoadImage (&ImageContext); if (!EFI_ERROR (Status)) { // // Relocate the image in our new buffer // Status = PeCoffLoaderRelocateImage (&ImageContext); if (!EFI_ERROR (Status)) { // // Execute image // EntryPoint = (EFI_IMAGE_ENTRY_POINT)(UINTN)ImageContext.EntryPoint; Status = EntryPoint ((EFI_HANDLE)Context, gST); } </pre>

Таблица 4 – Установка протоколов

Дизассемблер	TianoCore; PiSmmIpl.c
<pre> gBS->InstallMultipleProtocolInterfaces(&Handle, &EFI_SMM_BASE2_PROTOCOL_GUID_4080, off_4110, &EFI_SMM_COMMUNICATION_PROTOCOL_GUID_4090, &off_4120, 0i64); off_4110: off_4110 dq offset sub_1254 dq offset sub_1270 unsigned __int64 __fastcall sub_1254(__int64 a1, _BYTE *a2) { if (!a2) return 0x8000000000000002ui64; *a2 = byte_4159; return 0i64; } unsigned __int64 __fastcall sub_1270(__int64 a1, _QWORD *a2) { if (!a1 !a2) return 0x8000000000000002ui64; if (!byte_4159) return 0x8000000000000003ui64; *a2 = qword_4160; return 0i64; } </pre>	<pre> // // Install SMM Base2 Protocol and SMM Communication Protocol // Status = gBS->InstallMultipleProtocolInterfaces (&mSmmIplHandle, &gEfiSmmBase2ProtocolGuid, &mSmmBase2, &gEfiSmmCommunicationProtocolGuid, &mSmmCommunication, NULL); mSmmBase2: // // SMM Base 2 Protocol instance // EFI_SMM_BASE2_PROTOCOL mSmmBase2 = { SmmBase2InSmram, SmmBase2GetSmstLocation }; SmmBase2InSmram (// индикатор запуска драйвера в фазе SMM Initialization IN CONST EFI_SMM_BASE2_PROTOCOL *This, OUT BOOLEAN *InSmram) { if (InSmram == NULL) { return EFI_INVALID_PARAMETER; } *InSmram = gSmmCorePrivate->InSmm; return EFI_SUCCESS; } Находит местоположение System Management System Table (SMST) SmmBase2GetSmstLocation (IN CONST EFI_SMM_BASE2_PROTOCOL *This, OUT EFI_SMM_SYSTEM_TABLE2 **Smst) { if ((This == NULL) (Smst == NULL)) { return EFI_INVALID_PARAMETER; } if (!gSmmCorePrivate->InSmm) { return EFI_UNSUPPORTED; } *Smst = gSmmCorePrivate->Smst; return EFI_SUCCESS; } </pre>

<pre> off_4120: off_4120 dq offset sub_12A8 unsigned __int64 __fastcall sub_12A8(__int64 a1, __int64 a2, __int64 *a3) { if ((*(__int64 (__fastcall **)(void *, _QWORD, _QWORD, _QWORD, _QWORD))qword_4480)(qword_4480,0i64,0i64,0i64,0i64) < 0) return 0x8000000000000003ui64; return qword_4178; } </pre>	<pre> mSmmCommunication: EFI_SMM_COMMUNICATION_PROTOCOL mSmmCommunication = { SmmCommunicationCommunicate }; SmmCommunicationCommunicate (IN CONST EFI_SMM_COMMUNICATION_PROTOCOL *This, IN OUT VOID *CommBuffer, IN OUT UINTN *CommSize OPTIONAL) { // // Generate Software SMI // Status = mSmmControl12->Trigger (mSmmControl12,NULL,NULL,FALSE,0); if (EFI_ERROR (Status)) { return EFI_UNSUPPORTED; } // // Return status from software SMI // if (CommSize != NULL) { *CommSize = gSmmCorePrivate->BufferSize; } return gSmmCorePrivate->ReturnStatus; } </pre>
<pre> typedef struct { BOOLEAN Protocol; BOOLEAN CloseOnLock; EFI_GUID *Guid; +8 EFI_EVENT_NOTIFY NotifyFunction; VOID *NotifyContext; +24 EFI_TPL NotifyTpl; +32 EFI_EVENT Event; +40 } SMM_IPL_EVENT_NOTIFICATION; v14 = (void (__cdecl **)(EFI_EVENT, void *))&off_41B0; //LocateProtocol v15 = 0i64; do { v16 = *v14; v17 = *(void **)&byte_41A0[v15 + 24]; // NotifyFunction v18 = *(_QWORD *)&byte_41A0[v15 + 32]; // NotifyTpl if (byte_41A0[v15]) { v19 = *(EFI_GUID **)&byte_41A0[v15 + 8]; gBS_43E0->CreateEvent(0x200u, v18, v16, v17, &Event); gBS_43E0->RegisterProtocolNotify(v19, Event, &Registration); gBS_43E0->SignalEvent(Event); *(_QWORD *)&byte_41A0[v15 + 40] = Event; } else </pre>	<p>Создаётся массив типа SMM_IPL_EVENT_NOTIFICATION и заполняется функциями типа SmmIplSmmConfigurationEventNotify или SmmIplReadyToLockEventNotify. Первая ищет точку входа, вторая – закрывает SMRAM READY_TO_LOCK_PROTOCOL и так далее.</p> <pre> // // Create the set of protocol and event notifications that the SMM IPL requires // for (Index = 0; mSmmIplEvents[Index].NotifyFunction != NULL; Index++) { if (mSmmIplEvents[Index].Protocol) { mSmmIplEvents[Index].Event = EfiCreateProtocolNotifyEvent (mSmmIplEvents[Index].Guid, mSmmIplEvents[Index].NotifyTpl, mSmmIplEvents[Index].NotifyFunction, mSmmIplEvents[Index].NotifyContext, &Registration); } else { Status = gBS->CreateEventEx (EVT_NOTIFY_SIGNAL, mSmmIplEvents[Index].NotifyTpl, mSmmIplEvents[Index].NotifyFunction, mSmmIplEvents[Index].NotifyContext, </pre>

<pre> { gBS_43E0->CreateEventEx(0x200u, v18, v16, v17, *(const EFI_GUID **)&byte_41A0[v15 + 8], (EFI_EVENT *)&byte_41A0[v15 + 40]); } ++v13; v15 = 48 * v13; v14 = (void (__cdecl **)(EFI_EVENT, void *))&byte_41A0[48 * v13 + 16]; } while (*v14); </pre>	<pre> mSmmIplEvents[Index].Guid, &mSmmIplEvents[Index].Event); ASSERT_EFI_ERROR (Status); } } </pre>
--	---

Отдельно рассмотрим протокол READY_TO_LOCK_PROTOCOL (таблица 5).

Таблица 5 – Использование READY_TO_LOCK_PROTOCOL

Дизассемблер	TianoCore; PiSmmIpl.c
<pre> void __fastcall sub_14F8(__int64 a1, _QWORD *a2) { if (!byte_43C4 && (*a2 != *(_QWORD*) &EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL_GUID_40D0.Data1 a2[1] != *(_QWORD *) EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL_GUID_40D0.Data4 (gBS_43E0-> LocateProtocol(&EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL_GUID_40D0, 0i64, &Interface) & 0x8000000000000000ui64) == 0i64)) (*(void (__fastcall **)(EFI_TPL))(mSmmAccess + 16))(mSmmAccess); v4 - SMM_IPL_EVENT_NOTIFICATION. v4 = 0i64; do { if (byte_41A0[v4 + 1]) gBS_43E0->CloseEvent(*(EFI_EVENT *)&byte_41A0[v4 + 40]); ++v3; v4 = 48 * v3; // 48 – размер одного элемента массива(одна структура) } while (*(_QWORD *)&byte_41A0[48 * v3 + 16]); </pre>	<pre> SmmIplReadyToLockEventNotify (IN EFI_EVENT Event, IN VOID *Context) { if (mSmmLocked) { return; } // // Make sure this notification is for this handler // if (CompareGuid ((EFI_GUID *)Context, &gEfiDxeSmmReadyToLockProtocolGuid)) { Status = gBS->LocateProtocol (&gEfiDxeSmmReadyToLockProtocolGuid, NULL, &Interface); if (EFI_ERROR (Status)) { return; } } // // Lock the SMRAM (Note: Locking SMRAM may not be supported on all platforms) // mSmmAccess->Lock (mSmmAccess); // Close protocol and event notification events that do not apply after the // DXE SMM Ready To Lock Protocol has been installed or the Ready To Boot // event has been signalled. for (Index = 0; mSmmIplEvents[Index].NotifyFunction != NULL; Index++) { if (mSmmIplEvents[Index].CloseOnLock) { gBS->CloseEvent (mSmmIplEvents[Index].Event); } } } </pre>

<pre> sub_13B8(a1, (__int64 *)&EFI_DXE_SMM_READY_TO_LOCK_PROTOCOL_GUID_40D0); byte_43C4 = 1; </pre>	<pre> // // Inform SMM Core that the DxeSmmReadyToLock // protocol was installed // SmmIplGuidedEventNotify (Event, (VOID *)&gEfiDxeSmmReadyToLockProtocolGuid); // // Set flag so this operation will not be // performed again // mSmmLocked = TRUE; </pre>
--	--

Обозначим функцию, которая находит точку входа в SMM (таблица 6).

Таблица 6 – Нахождение точки входа в SMM

Дизассемблер	TianoCore; PiSmmIpl.c
<pre> struct _EFI_SMM_CONFIGURATION_PROTOCOL { /// /// A pointer to an array SMRAM ranges used by the initial SMM entry code. /// EFI_SMM_RESERVED_SMRAM_REGION *SmramReservedRegions; EFI_SMM_REGISTER_SMM_ENTRY RegisterSmmEntry; }; </pre> <p>EFI_SMM_RESERVED_SMRAM_REGION - 64 бита весит, поэтому +64 приведет на следующее поле структуры</p> <pre> __int64 sub_21F4() { __int64 result; // rax if (qword_43F0) return (*(__int64 (__fastcall **)(__QWORD, __int64 *))(qword_4400 + 64))(0i64, &qword_43F0); } </pre> <p>В силу того, что у нас нет вызова функции LocateProtocole, второй параметр 0</p>	<pre> SmmIplSmmConfigurationEventNotify (IN EFI_EVENT Event, IN VOID *Context) // Make sure this notification is for this handler // Status = gBS->LocateProtocol (Context, NULL, (VOID ***)&SmmConfiguration); // // Register the SMM Entry Point provided by the SMM // Core with the SMM Configuration protocol // Status = SmmConfiguration->RegisterSmmEntry (SmmConfiguration, gSmmCorePrivate->SmmEntryPoint); ASSERT_EFI_ERROR (Status); </pre>

Обобщим происходящее в таблицах 1,2,3,4,5.

Инициализируем SMM IPL хэндл, чтобы SMM Core мог использовать родительский его в качестве родительского для загрузки всех SMM Drivers. Заполучаем протоколы SMM Access Protocol и SMM Control2 Protocol. Вызываем функцию GetFullSmramRanges. Предназначение этой функции – найти все диапазоны SMRAM, если между ними есть пересечение – разделить его.

Далее открываем все SMRAM диапазоны. Появляется так называемое SMRAM-окно, и оно теперь открыто. Затем пытаемся найти наибольшее SMRAM-окно (где-то в 256Kб). Таким образом, находится место в памяти процессора под SMRAM, которое будет использоваться SMM IPL и SMM Core. Потом загружаем SMM Core в SMRAM и запускаем его из SMRAM. Этот функционал реализуется процедурой ExecuteSmmCoreFromSmram. В ней определяется, что каждый том прошивки драйвера должен создавать экземпляр Firmware Volume Protocol, если том микропрограммы должен быть видим для системы во время фазы DXE. Загружают образ в новый буфер и запускают образ через функцию EntryPoint.

Затем происходит установка протоколов SMM Base2 Protocol и SMM Communication Protocol. Устанавливается индикация, что запуск драйвера происходит уже в SMM Initialization(функция SmmBase2InSmram). Находится местоположение System Management System Table (SMST) и функция, генерирующая Software SMI прерывание.

Далее создаётся массив типа SMM_IPL_EVENT_NOTIFICATION и заполняется функциями типа SmmIplSmmConfigurationEventNotify или SmmIplReadyToLockEventNotify, которые по очереди вызываются. Первая ищет точку входа, то есть регистрирует SMM Entry Point, предоставленную SMM Core вместе с SMM Configuration протоколом. Вторая – закрывает SMRAM через READY_TO_LOCK_PROTOCOL, то есть проверяется, не закрыта ли уже Smm, а если нет, то закрывает SMRAM и отменяет протоколы нотификации, так как применился протокол DXE SMM Ready To Lock Protocol. Информировываем, что протокол DxeSmmReadyToLock установлен и устанавливаем флаг, что SMM закрыт.

Модуль SmmHddSecurity

Находим точку входа в SmmHddSecurity драйвер (рисунок 3).

```
EFI_STATUS __fastcall ModuleEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    __int64 v4; // rax
    EFI_STATUS v5; // rdi
    __int64 v6; // rbx

    sub_330();
    qword_2358 = 0x8000000000000001ui64;
    if ( !sub_1900(&unk_2260) )
    {
        v4 = sub_107C((__int64)ImageHandle, (__int64)SystemTable);
        if ( v4 >= 0 || qword_2358 < 0 )
            qword_2358 = v4;
        sub_1990(&unk_2260, -1i64);
    }
    v5 = qword_2358;
    if ( qword_2358 < 0 )
    {
        v6 = qword_2430;
        if ( (unsigned __int8)sub_1594(qword_2430) )
            *(void (__fastcall **)(__int64))(qword_23A8 + 88)(v6);
        else
            *(void (__fastcall **)(__int64))(qword_23A0 + 72)(v6);
    }
    return v5;
}
```

Рисунок 3 – Экспортируемая точка входа

Рассмотрим первую функцию. Используя плагин efiXplorer, GUID'ы выставляются автоматически (рисунок 4).

```
__int64 __fastcall sub_330(__int64 a1, EFI_SYSTEM_TABLE *a2)
{
    EFI_BOOT_SERVICES *BootServices; // rax
    __int64 v3; // rax
    void *v4; // rcx
    unsigned __int64 v6; // [rsp+30h] [rbp+8h] BYREF
    void *Interface; // [rsp+38h] [rbp+10h] BYREF
    EFI_TPL NewTpl; // [rsp+40h] [rbp+18h] BYREF

    BootServices = a2->BootServices;
    qword_2390 = a1;
    gST_2398 = a2;
    gBS_23A0 = BootServices;
    Interface = 0i64;
    BootServices->LocateProtocol(&EFI_SMM_BASE2_PROTOCOL_GUID_1BF0, 0i64, &Interface);
    *((void (__fastcall **)(void *, _EFI_SMM_SYSTEM_TABLE2 **))Interface + 1)(Interface, &gSmst_23A8);
    byte_2440 = 0;
    gBS_23A0->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID_1C40, 0i64, (void **)&NewTpl);
    v6 = 0i64;
    (*(void (__cdecl **)(EFI_TPL))(NewTpl + offsetof(EFI_BOOT_SERVICES, RaiseTPL)))(NewTpl);
    Buffer = (void *)sub_15D8(6i64, v6);
    (*(void (__fastcall **)(EFI_TPL, unsigned __int64 *, void **))(NewTpl + 24))(NewTpl, &v6, Buffer);
    qword_2438 = v6 >> 5;
    v3 = gSmst_23A8->SmmLocateProtocol(&gAmiSmmBufferValidationProtocolGuid_1C60, 0i64, &qword_23F0);
    v4 = qword_23F0;
    if ( v3 < 0 )
        v4 = 0i64;
    qword_23F0 = v4;
    return ((__int64 (__fastcall *) (EFI_GUID *, _QWORD, __int64 *))gSmst_23A8->SmmLocateProtocol)(
        &EFI_S3_SMM_SAVE_STATE_PROTOCOL_GUID_1C70,
        0i64,
        &qword_23F8);
}
```

Рисунок 4 – Функция sub_330

Произведем переименования фактических параметров функции sub_330 (рисунок 5)

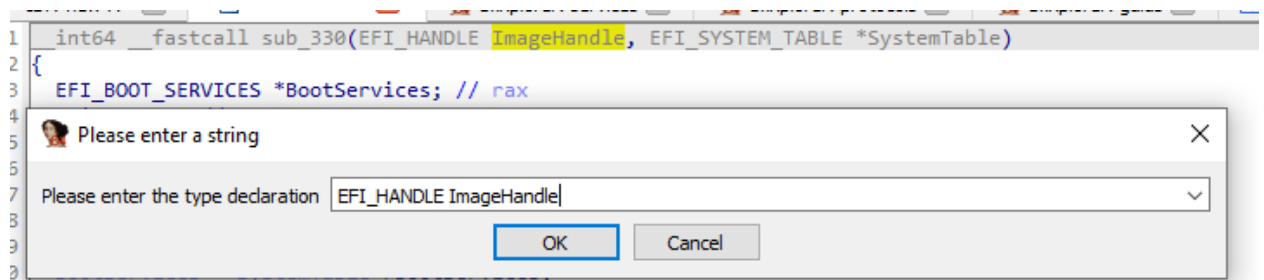


Рисунок 5 - Переименования аргументов

Затем зададим глобальной переменной верное название, следуя из контекста (рисунок 6).

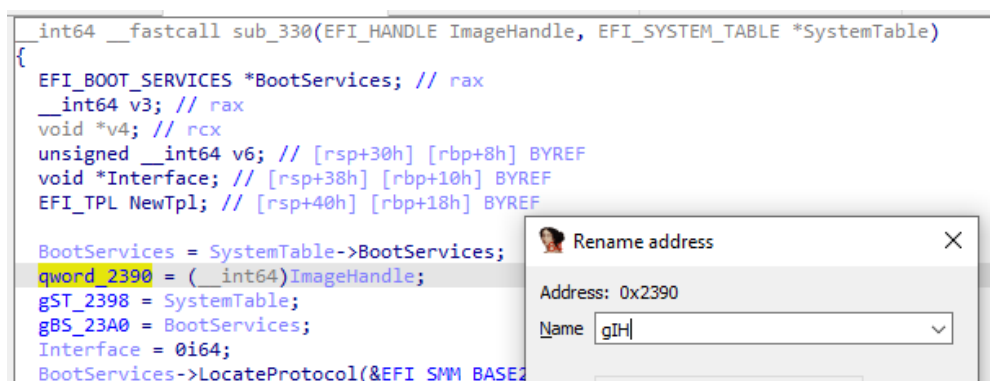


Рисунок 6 – Переименование переменной-копии ImageHandle

Мы видим использование LocateProtocol, а значит понимаем, что последняя переменная будет такого же типа, как и GUID (рисунки 7 и 8).

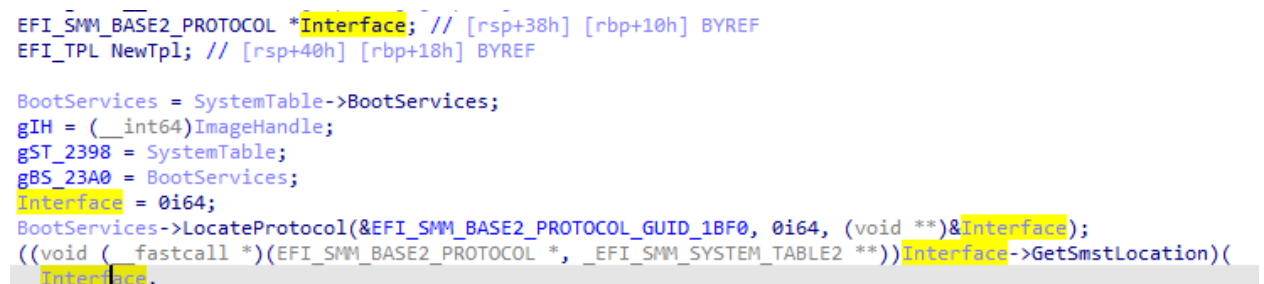


Рисунок 7 – Задание нового типа данных в соответствии с GUID и определению функции LocateProtocol

```

2 | gST_2398 = SystemTable;
3 | gBS_23A0 = BootServices;
4 | Interface = 0i64;
5 | BootServices->LocateProtocol(&EFI_SMM_BASE2_PROTOCOL_GUID_1BF0, 0i64, (void **)&Interface);
6 | ((void (__fastcall *)(EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))Interface->GetSmstLocation)
7 |   Interface,
8 |   &gSmst_23A8);
9 | byte_2440 = 0;
10 | gBS_23A0->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID_1C40, 0i64, (void **)&NewTpl);

```

Рисунок 8 – Установлен верный тип данных

Аналогично поступаем с остальными параметрами (рисунки 8 и 9)

```

EFI_TPL NewTpl; // [rsp+40h] [rbp+18h] BYREF

BootServices = SystemTable->BootServices;
gIH = (__int64)ImageHandle;
gST_2398 = SystemTable;
gBS_23A0 = BootServices;
Interface = 0i64;
BootServices->LocateProtocol(&EFI_SMM_BASE2_PROTOCOL_GUID_1BF0, 0i64, (void **)&Interface);
((void (__fastcall *)(EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))Interface->GetSmstLoc
  Interface,
  &gSmst_23A8);
byte_2440 = 0;
gBS_23A0->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID_1C40, 0i64, (void **)&NewTpl);
v6 = 0i64;
*(void (__cdecl **)(EFI_TPL))(NewTpl + offsetof(EFI_BOOT_SERVICES, RaiseTPL))(NewTpl);
Buffer = (void *)sub_15D8(6i64, v6);

```

Рисунок 8 – Перед изменением типа данных на верный

```

DA View-A x Pseudocode-A x efXplorer: services x efXplorer: protocols x efXplorer: guides x Hex View-1 x Structures
__int64 __fastcall sub_330(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
  EFI_BOOT_SERVICES *BootServices; // rax
  __int64 v3; // rax
  void *v4; // rcx
  unsigned __int64 v6; // [rsp+30h] [rbp+8h] BYREF
  EFI_SMM_BASE2_PROTOCOL *Interface; // [rsp+38h] [rbp+10h] BYREF
  EFI_SMM_ACCESS2_PROTOCOL *NewTpl; // [rsp+40h] [rbp+18h] BYREF

  BootServices = SystemTable->BootServices;
  gIH = (__int64)ImageHandle;
  gST_2398 = SystemTable;
  gBS_23A0 = BootServices;
  Interface = 0i64;
  BootServices->LocateProtocol(&EFI_SMM_BASE2_PROTOCOL_GUID_1BF0, 0i64, (void **)&Interface);
  ((void (__fastcall *)(EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))Interface->GetSmstLocation)(
    Interface,
    &gSmst_23A8);
  byte_2440 = 0;
  gBS_23A0->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID_1C40, 0i64, (void **)&NewTpl);
  v6 = 0i64;
  ((void (__cdecl **)(EFI_TPL))(NewTpl->GetCapabilities))((EFI_TPL)NewTpl);
  Buffer = (void *)sub_15D8(6i64, v6);
  ((void (__fastcall *)(EFI_SMM_ACCESS2_PROTOCOL *, unsigned __int64 *))(NewTpl->Access2Protocol))(NewTpl,
    EFI_SMM_ACCESS2_PROTOCOL *NewTpl; // [rsp+40h] [rbp+18h] BYREF

```

Рисунок 9 – Верный тип данных

Проверено в EDK2, что последний параметр GetCapabilities является типом данных EFI_SMRAM_DESCRIPTION (рисунки 10 и 11)

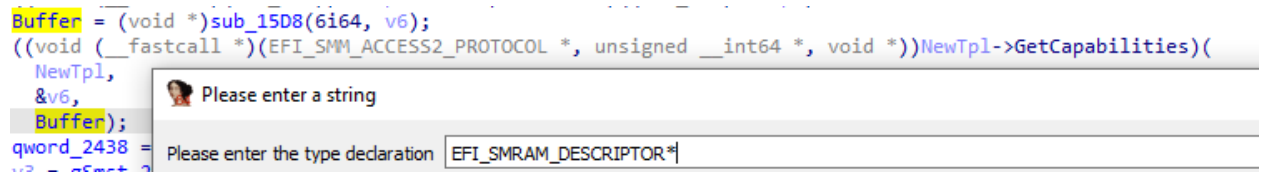


Рисунок 10 – GetCapabilities и его последний параметр

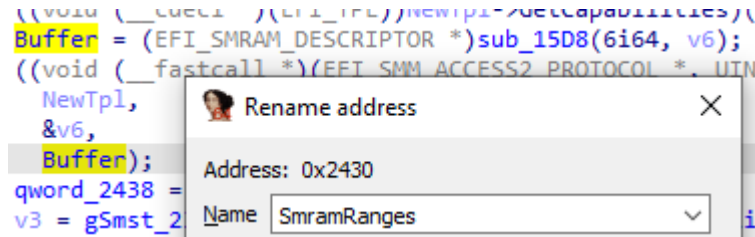


Рисунок 11 – Верное имя

Продолжаем проделывать изменения типов данных на верные и выставлять верные имена (рисунки 12 и 13)

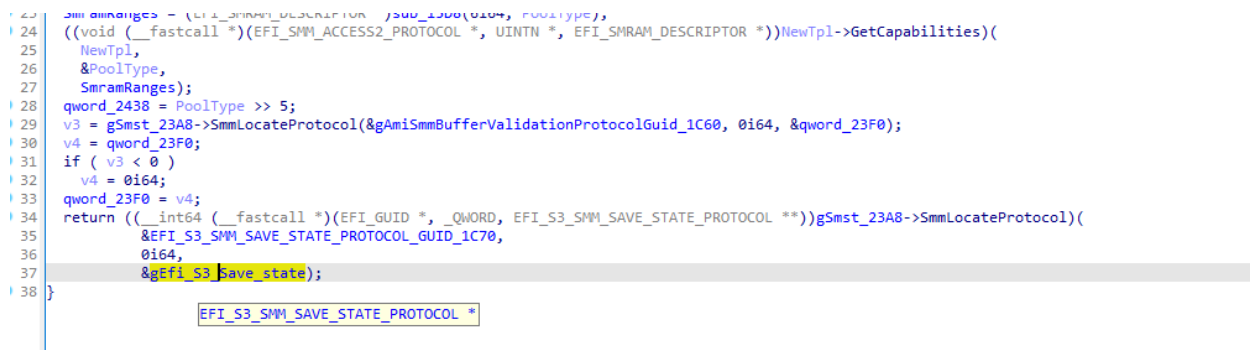


Рисунок 12 – Верный тип данных

Итоговый результат (рисунок 13)

```

__int64 __fastcall InitAmi(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    EFI_BOOT_SERVICES *BootServices; // rax
    __int64 v3; // rax
    void *v4; // rcx
    UINTN PoolType; // [rsp+30h] [rbp+8h] BYREF
    EFI_SMM_BASE2_PROTOCOL *SmmBase2Proto; // [rsp+38h] [rbp+10h] BYREF
    EFI_SMM_ACCESS2_PROTOCOL *NewTpl; // [rsp+40h] [rbp+18h] BYREF

    BootServices = SystemTable->BootServices;
    gIH = (__int64)ImageHandle;
    gST_2398 = SystemTable;
    gBS_23A0 = BootServices;
    smmBase2Proto = 0i64;
    BootServices->LocateProtocol(&gEfiSmmBase2ProtocolGuid_18F0, 0i64, (void **)&smmBase2Proto);
    ((void (__fastcall *) (EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))smmBase2Proto->GetSmstLocation)(
        smmBase2Proto,
        &gSmst_23A8);
    byte_2440 = 0;
    gBS_23A0->LocateProtocol(&gEfiSmmAccess2ProtocolGuid_1C40, 0i64, (void **)&NewTpl);
    PoolType = 0i64;
    ((void (__cdecl *) (EFI_TPL))NewTpl->GetCapabilities)((EFI_TPL)NewTpl);
    SmmramRanges = (EFI_SMRAM_DESCRIPTOR *)sub_15D8(6i64, PoolType);
    ((void (__fastcall *) (EFI_SMM_ACCESS2_PROTOCOL *, UINTN *, EFI_SMRAM_DESCRIPTOR *))NewTpl->GetCapabilities)(
        NewTpl,
        &PoolType,
        SmmramRanges);
    qword_2438 = PoolType >> 5;
    v3 = gSmst_23A8->SmmLocateProtocol(&gAmiSmmBufferValidationProtocolGuid_1C60, 0i64, &qword_23F0);
    v4 = qword_23F0;
    if (v3 < 0)
        v4 = 0i64;
    qword_23F0 = v4;
    return ((__int64 (__fastcall *) (EFI_GUID *, _QWORD, EFI_S3_SMM_SAVE_STATE_PROTOCOL **))gSmst_23A8->SmmLocateProtocol)(
        &gEfiS3SmmSaveStateProtocolGuid_1C70,
        0i64,
        &gEfi_S3_Save_state);
}

```

Рисунок 13 – Итоговая функция InitAmi

Выставим в точке входа переменную статуса (рисунок 14)

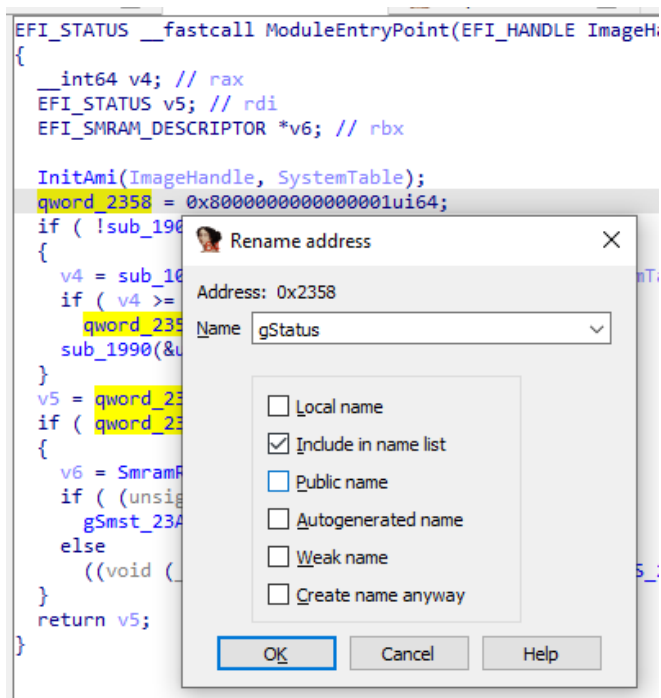


Рисунок 14 – Верное по контексту имя переменной

Узнаём из контекста перехват исключений setjmp и longjmp (рисунок 15)

```
if ( !setjmp((__int64)&unk_2260) )
{
    v4 = sub_107C((__int64)ImageHandle, (__int64)SystemTable);
    if ( v4 >= 0 || gStatus < 0 )
        gStatus = v4;
    longjmp((__int64)&unk_2260);
}
```

Рисунок 15 - Перехват исключений

Заходим в функцию sub_107C (рисунок 16)

```
1  __int64 __fastcall sub_107C(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
2  {
3      EFI_BOOT_SERVICES *gBootServices; // r9
4      EFI_RUNTIME_SERVICES *gRuntimeServices; // rax
5      EFI_SMM_BASE2_PROTOCOL *v6; // rax
6      __int64 v7; // rax
7      EFI_RUNTIME_SERVICES *v8; // rcx
8      __int64 result; // rax
9      char v10; // [rsp+40h] [rbp+18h] BYREF
10     void *Interface; // [rsp+48h] [rbp+20h] BYREF
11
12     if ( gST_2400 )
13     {
14         gBootServices = gBS_2408;
15     }
16     else
17     {
18         gST_2400 = SystemTable;
19         gBootServices = SystemTable->BootServices;
20         gRuntimeServices = SystemTable->RuntimeServices;
21         gBS_2408 = gBootServices;
22         gRT_2410 = gRuntimeServices;
23         gIH_ = ImageHandle;
24     }
25     v6 = qword_2418;
26     if ( !qword_2418 )
27     {
28         if ( ((__int64 (__fastcall *) (EFI_GUID *, _QWORD, EFI_SMM_BASE2_PROTOCOL **))gBootServices->LocateProtocol)(
29             &EFI_SMM_BASE2_PROTOCOL_GUID_2158,
30             0i64,
31             &qword_2418) < 0 )
32             goto LABEL_12;
33         v6 = qword_2418;
34     }
35 }
```

Рисунок 16 – Новое имя переменной

Изменяем тип данных gSMST_ на _EFI_SMM_SYSTEM_TABLE2* (рисунок 17)

```
((void (__fastcall *) (EFI_SMM_BASE2_PROTOCOL *, char *))efiSmmBase2->InSmm)(efiSmmBase2, &in_smram);
if ( in_smram
    && ((__int64 (__fastcall *) (EFI_SMM_BASE2_PROTOCOL *, EFI_SMM_SYSTEM_TABLE **))efiSmmBase2Proto->GetSmstLocation)(
        efiSmmBase2Proto,
        &gSMST_) >= 0 )
{
    v7 = SmmHddSecRuntime
    v8 = gRT_2410;
    DebugFlag = 1;
    byte_2381 = 0;
    if ( v7 )
```

Рисунок 17 – Новый тип данных

Далее рассмотрим функции из рисунка 18.

```
{
    v7 = SmmHddSecRuntimeServ();
    v8 = gRT_2410;
    DebugFlag = 1;
    byte_2381 = 0;
    if ( v7 )
        v8 = (EFI_RUNTIME_SERVICES *)v7;
    gRT_2410 = v8;
    HddSecInitSmmStatusCodeProtocol();
    HddSecInitSmmSomeProtocol();           // установка протоколов
    byte_2381 = 1;
}
```

Рисунок 18 – Некоторые функции

Происходит подмена RUNTIME_SERVICES таблицы на SMM'ную версию, чтобы разделить режимы RING 0 и SMM (рисунок 19).

```
UINT64 SmmHddSecRuntimeServ()
{
    EFI_CONFIGURATION_TABLE *SmmConfigurationTable; // rbx
    __int64 v1; // r11

    if ( !gSMST_ )
        return 0i64;
    SmmConfigurationTable = gSMST_->SmmConfigurationTable;
    if ( !gSMST_->NumberOfTableEntries )
        return 0i64;
    while ( CmpGuid((unsigned __int64)SmmConfigurationTable, &EFI_SMM_RUNTIME_SERVICES_TABLE_GUID_2238) )
    {
        ++SmmConfigurationTable;
        if ( v1 == 1 )
            return 0i64;
    }
    return (UINT64)SmmConfigurationTable->VendorTable;
}
```

Рисунок 19 – Подмена таблицы на SMM'ные

Две функции HddSecInitSmmStatusCodeProtocol и HddSecInitSmmSomeProtocol представлены на рисунках 20 и 21.

```
__int64 HddSecInitSmmStatusCodeProtocol()
{
    __int64 result; // rax

    result = 0i64;
    if ( !gEfiSmmStatusCodeProto )
    {
        if ( gSMST_ )
            return ((__int64 (__fastcall *) (EFI_GUID *, _QWORD, EFI_SMM_STATUS_CODE_PROTOCOL **))gSMST_->SmmLocateProtocol)(
                &gEfiSmmStatusCodeProtocolGuid_2178,
                0i64,
                &gEfiSmmStatusCodeProto);
        else
            return 0x8000000000000003ui64;
    }
    return result;
}
```

Рисунок 20 – Функция HddSecInitSmmStatusCodeProtocol

```

__int64 HddSecInitSmmSomeProtocol()
{
    __int64 v1; // r11
    EFI_TPL v2; // rbx
    void *v3; // rax

    if ( DebugFlag )
    {
        if ( gAmiSmmDebugServProto )
            return 0i64;
        if ( !gSMST_ )
            return 0x8000000000000003ui64;
        v1 = ((__int64 (__fastcall *) (EFI_GUID *, _QWORD, void **))gSMST_->SmmLocateProtocol)(
            &gAmiSmmDebugServiceProtocolGuid_1C30,
            0i64,
            &gAmiSmmDebugServProto);
        if ( v1 < 0 )
            gAmiSmmDebugServProto = 0i64;
    }
    else
    {
        if ( gAmiDebugServicePpi )
            return 0i64;
        if ( byte_2381 == 1 )
            return 0x8000000000000003ui64;
        v2 = gBS_2408->RaiseTPL(0x1Fui64);
        ((void (__fastcall *) (EFI_TPL))gBS_2408->RestoreTPL)(v2);
        if ( v2 > 0x10 )
            return 0x8000000000000003ui64;
        v1 = gBS_2408->LocateProtocol(&gAmiDebugServiceProtocolGuid_1C50, 0i64, &gAmiDebugServicePpi);
        v3 = gAmiDebugServicePpi;
        if ( v1 < 0 )
            v3 = 0i64;
        gAmiDebugServicePpi = v3;
    }
    return v1;
}

```

Рисунок 21 – Функция HddSecInitSmmSomeProtocol

Далее находим hddSecMain, в которой сосредоточена основная функциональность (рисунок 22)

```

else
{
    qword_2468 = 0i64;
    qword_2470 = 0i64;
    qword_2460 = 0i64;
    qword_2388 = 4026531840i64;
    return hddSecMain(ImageHandle, SystemTable); // Main?
}

```

Рисунок 22 – Вход в hddSecMain

Определение функции hddSecMain представлено на рисунке 23 и процесс реверс-инжиниринга совпадает с вышеописанными действиями.


```

int64 __fastcall hddSecMain(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    EFI_BOOT_SERVICES *BootServices; // r9
    EFI_RUNTIME_SERVICES *RuntimeServices; // rax
    __int64 v4; // rdi
    __int64 result; // rax
    UINT64 v6; // rax
    EFI_RUNTIME_SERVICES *v7; // rcx
    char v8[24]; // [rsp+20h] [rbp-18h] BYREF

    v8[0] = 0;
    if ( gST_2400 )
    {
        BootServices = gBS_2408;
    }
    else
    {
        gST_2400 = SystemTable;
        BootServices = SystemTable->BootServices;
        RuntimeServices = SystemTable->RuntimeServices;
        gBS_2408 = BootServices;
        gRT_2410 = RuntimeServices;
        gIH_ = ImageHandle;
    }
    v4 = ((__int64 (__fastcall *))(EFI_GUID *, _QWORD, EFI_SMM_BASE2_PROTOCOL **))BootServices->LocateProtocol(
        &gEfiSmmBase2ProtocolGuid_2158,
        0i64,
        &efiSmmBase2Proto);
    if ( v4 < 0 )
        return v4;
    ((void (__fastcall *))(EFI_SMM_BASE2_PROTOCOL *, char *))efiSmmBase2Proto->InSmm(efiSmmBase2Proto, v8);
    if ( !v8[0] )
        return v4;
    result = ((__int64 (__fastcall *))(EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))efiSmmBase2Proto->GetSmstLocation(
        efiSmmBase2Proto,
        &gSMST_);
    if ( result >= 0 )
    {
        v6 = SmmHddSecRuntimeServ();
        v7 = gRT_2410;
        DebugFlag = 1;
        byte_2381 = 0;
        if ( v6 )
            v7 = (EFI_RUNTIME_SERVICES *)v6;
        gRT_2410 = v7;
        HddSecInitSmmStatusCodeProtocol();
        HddSecInitSmmSomeProtocol();
        byte_2381 = 1;
        return Handlers(); // тут обработки прерываний
    }
    return result;
}

```

Рисунок 23 – Определение hddSecMain

В функции Handlers находятся обработчики прерываний (SWSMI) (рисунок 24)

```
__int64 Handlers()
{
    __int64 result; // rax
    __int64 v1[3]; // [rsp+20h] [rbp-18h] BYREF
    EFI_HANDLE DispatchHandle; // [rsp+50h] [rbp+18h] BYREF
    __int64 v3; // [rsp+58h] [rbp+20h] BYREF

    v3 = 209164;
    v1[0] = 211164;
    result = gSmst_2480->SmiHandlerRegister((EFI_SMM_HANDLER_ENTRY_POINT2)SmiHandler_97C, &HandlerType, &DispatchHandle); // Первый обработчик
    if ( result >= 0 )
    {
        result = gSmst_2480->SmiHandlerRegister(SmiHandler_D24, &stru_21F8, &DispatchHandle); // Второй обработчик
        if ( result >= 0 )
        {
            result = ((__int64 (__fastcall *) (EFI_SMM_SW_DISPATCH2_PROTOCOL *, __int64 (__fastcall *) (EFI_HANDLE, void *, void *, UINTN *), __int64 *, EFI_HANDLE *, __int64)))Interface->Register(
                Interface,
                hddSecSwSmiDispatch1_Handler, // Возможно это обработчик, сделал им callback. Нет вектора атаки
                &v3,
                &DispatchHandle,
                v1[0]);
            if ( result >= 0 )
            {
                result = ((__int64 (__fastcall *) (EFI_SMM_SW_DISPATCH2_PROTOCOL *, __QWORD, __int64 *, EFI_HANDLE *)))Interface->Register(
                    Interface,
                    hddSecSwSmiDispatch2_Handler, // Возможно это обработчик, сделал им callback. Нет вектора атаки
                    v1,
                    &DispatchHandle);
                if ( result >= 0 )
                {
                    result = gSmst_2480->SmiHandlerRegister(// третий
                        (EFI_SMM_HANDLER_ENTRY_POINT2)SmiHandler_E64,
                        &stru_2218,
                        &DispatchHandle);
                    if ( result >= 0 )
                    {
                        result = gSmst_2480->SmiHandlerRegister(// четвертый
                            (EFI_SMM_HANDLER_ENTRY_POINT2)SmiHandler_EA4,
                            &stru_2228,
                            &DispatchHandle);
                        if ( result >= 0 )
                        {
                            gSmst_2480->SmiHandlerRegister((EFI_SMM_HANDLER_ENTRY_POINT2)SmiHandler_F3C, &stru_2208, &DispatchHandle); // пятый
                            return 0164;
                        }
                    }
                }
            }
        }
    }
    return result;
}

EFI_STATUS __cdecl(EFI_HANDLE DispatchHandle, const void *Context, void *CommBuffer, UINTN *CommBufferSize)
0: 0008 rcx     EFI_HANDLE DispatchHandle;
1: 0008 rdx     const void *Context;
2: 0008 r8      void *CommBuffer;
3: 0008 r9      UINTN *CommBufferSize;
RET 0008 rax     EFI_STATUS;
TOTAL STKARGS SIZE: 32
```

00000F50 Handlers:6 (F50)

Рисунок 24 – Обработчики прерываний с callback-функциями

Функциям hddSecSwSmiDispatch1_Handler и hddSecSwSmiDispatch2_Handler были добавлены аргументы обработчика прерывания, так как их указатели находились в качестве аргументов в функции Register. Однако при детальном рассмотрении они не имеют интересного содержимого (рисунок 25).

```
__int64 __fastcall hddSecSwSmiDispatch2_Handler(
    EFI_HANDLE DispatchHandle,
    void *Context,
    void *CommBuffer,
    UINTN *CommBufferSize)
{
    __QWORD *v4; // rbx
    __int64 result; // rax
    int v6; // [rsp+20h] [rbp-38h]
    int v7; // [rsp+28h] [rbp-30h]
    __int64 v8; // [rsp+40h] [rbp-18h]

    v4 = (__QWORD *)qword_2468;
    result = v8;
    while ( v4 )
    {
        if ( *((_DWORD *)v4 - 24) == 608715844 )
        {
            LOBYTE(v7) = 0;
            LOBYTE(v6) = 0;
            sub_82C((__int64)(v4 - 12), 0, 0i64, 0i64, v6, v7, 0xF5u);
            result = sub_550((__int64)(v4 - 12));
            if ( result < 0 )
                break;
        }
        v4 = (__QWORD *)*v4;
    }
    return result;
}
```

Рисунок 25 – Определение hddSecSwSmiDispatch2_Handler

Таким образом, имеем 5 обработчиков на рассмотрение:

- SmiHandler_97C
- SmiHandler_D24
- SmiHandler_E64
- SmiHandler_EA4
- SmiHandler_F3C

Для автоматизации поиска уязвимостей (и соответственно векторов атаки злоумышленника) применим скрипт под названием brick.py от Sentinel-One [1].

Обратимся к результатам проверок (рисунок 26)

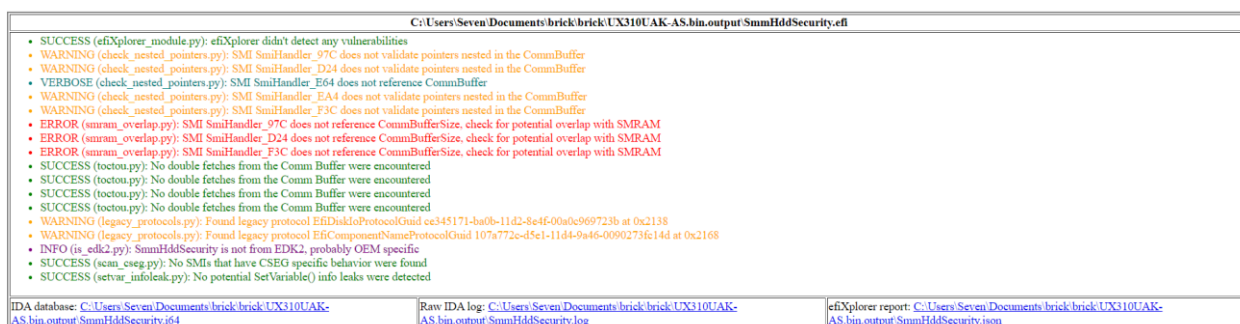


Рисунок 26 – Результат оценки SmmHddSecurity

Первым делом опишем ERROR-результаты и вектор атаки, связанный с этой ошибкой. Автоматизированное средство проверки посчитало опасным отсутствие использования проверки размера буфера, так как потенциально коммуникационный буфер (CommBuffer) может перекрыть SMRAM (рисунок 27). Другими словами, чтобы SMRAM был в безопасности, коммуникационный буфер не должен перекрывать SMRAM. В ином случае, обработчик, который пишет результаты в CommBuffer и не проверяет размер (с помощью CommBufferSize) потенциально может модифицировать содержимое SMRAM.

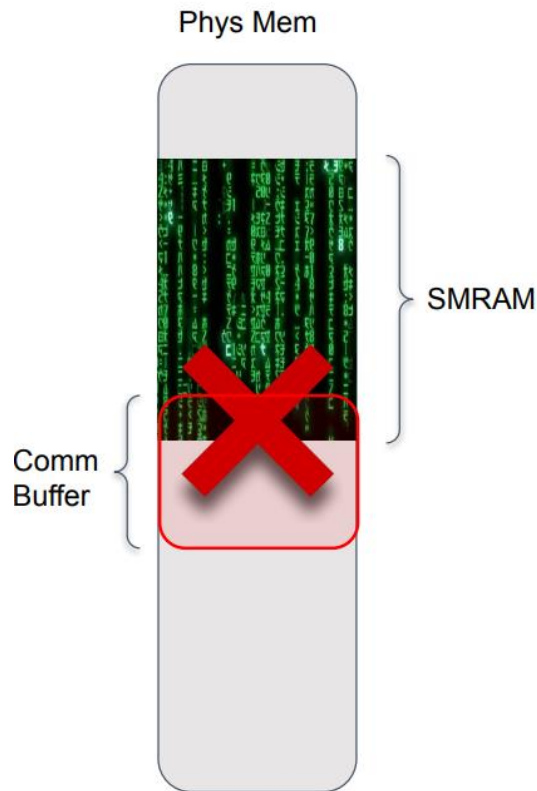


Рисунок 27 – Перекрытие SMRAM CommBuffer'ом [2]

В связи с вышеуказанными доводами опишем вектор атаки. Но сделаем оговорку, что атакующий уже имеет привилегии уровня ядра (ring0). Атакующий должен поместить CommBuffer прямо под SMRAM (SMRAM_BASE-1) с размером коммуникационного буфера равного 1 (CommBufferSize=1). Затем атакующий триггерит (например, командой chipsec'ка – smi smmc) SmmEntryPoint, который проверит, что (пока что) CommBuffer не перекрывается с SMRAM (рисунок 28).

```

if (gSmmCorePrivate->CommunicationBuffer != NULL) {
    //
    // Synchronous SMI for SMM Core or request from Communicate protocol
    //
    IsOverlapped = InternalIsBufferOverlapped (
        (UINT8 *) gSmmCorePrivate->CommunicationBuffer,
        gSmmCorePrivate->BufferSize,
        (UINT8 *) gSmmCorePrivate,
        sizeof (*gSmmCorePrivate)
    );
    if (!SmmIsBufferOutsideSmmValid ((UINTN)gSmmCorePrivate->CommunicationBuffer, gSmmCorePrivate->BufferSize) || IsOverlapped) {
        //
        // If CommunicationBuffer is not in valid address scope,
        // or there is overlap between gSmmCorePrivate and CommunicationBuffer,
        // return EFI_INVALID_PARAMETER
        //
        gSmmCorePrivate->CommunicationBuffer = NULL;
    }
}

```

Рисунок 28 – Проверка перекрытия в SmmEntryPoint

Если всё успешно, то запускается SMI обработчик, который запишет в CommBuffer данные, которые перезапишут содержимое SMRAM, так как нет проверки на CommBufferSize (рисунок 29).

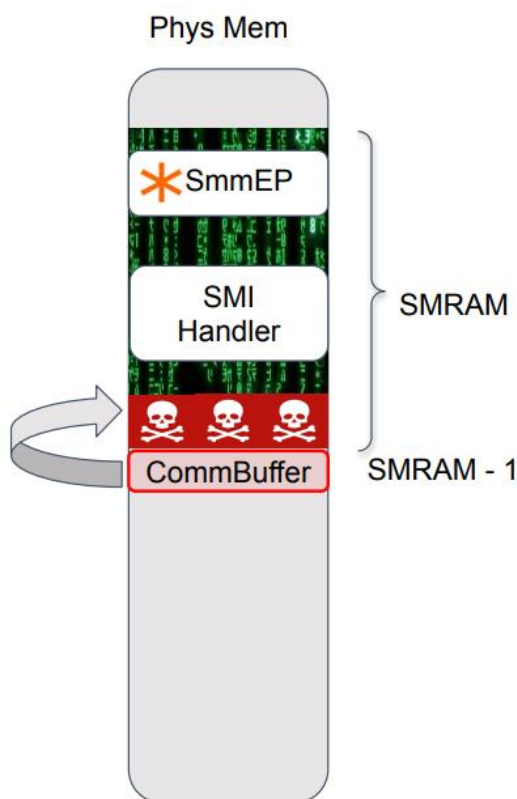


Рисунок 29 – Перекрытие SMRAM

Каждый такой обработчик “общается” через CommBuffer и его размер (CommBufferSize). Эти параметры являются источниками недоверенных входных данных, например, из операционной системы, которые могут заставить CommBuffer перекрыть SMRAM. Такой фаззинг может заставить исполнять вредоносный код в привилегированном режиме.

Таким образом, нужно проверять размер CommBufferSize.

Теперь необходимо проверить реальное положение дел. Обработчики SmiHandler_97C , SmiHandler_D24 и SmiHandler_F3C построены одинаковым образом (рисунки 30,31,32).

```

EFI_STATUS __fastcall SmiHandler_97C(
    EFI_HANDLE DispatchHandle,
    const void *Context,
    void *CommBuffer,
    UINTN *CommBufferSize)
{
    __int64 v5; // rbx
    __QWORD *v6; // rdx
    char v7; // r8
    __BYTE *v8; // rcx
    bool v9; // zf
    __int64 v10; // rsi
    __int64 v11; // rdx
    char *v12; // rcx
    __int64 v13; // rdx
    char *v14; // rdi
    char v15; // al
    char *v16; // rdi
    char v17; // al
    void *Buffer; // [rsp+40h] [rbp+18h] BYREF

    sub_8F8();
    v5 = 0i64;
    if ( CommBuffer && *((_DWORD *)CommBuffer) == 608715844 )
    {
        v6 = (__QWORD *)qword_2468;
        if ( qword_2468 )
        {
            v7 = *((_BYTE *)CommBuffer + 88);
            do
            {
                v8 = v6 - 12;
                Buffer = v6 - 12;
                if ( *((_BYTE *)v6 - 8) != v7 || v8[89] != *((_BYTE *)CommBuffer + 89) || v8[90] != *((_BYTE *)CommBuffer + 90) )
                    goto LABEL_16;
                if ( *((_DWORD *)v8 + 23) == 1 )
                {
                    if ( v8[78] != *((_BYTE *)CommBuffer + 78) )
                        goto LABEL_16;
                    v9 = v8[79] == *((_BYTE *)CommBuffer + 79);
                }
                else if ( *((_DWORD *)v8 + 23) )
                {
                    v9 = *((_QWORD *)v8 + 10) == *((_QWORD *)CommBuffer + 10);
                }
                else
                {
                    if ( *((_WORD *)v8 + 34) != *((_WORD *)CommBuffer + 34) )
                        goto LABEL_16;
                }
            } while (v8--);
        }
    }
}

```

Рисунок 30 – SmiHandler_97C

```

EFI_STATUS __fastcall SmiHandler_D24(
    EFI_HANDLE DispatchHandle,
    const void *Context,
    void *CommBuffer,
    UINTN *CommBufferSize)
{
    __QWORD *i; // rdx
    bool v6; // zf

    sub_8F8();
    if ( CommBuffer && *((_DWORD *)CommBuffer) == 608715844 )
    {
        for ( i = (__QWORD *)qword_2468; i = (__QWORD *)i )
        {
            if ( !i )
                return 0i64;
            if ( *((_BYTE *)i - 8) == *((_BYTE *)CommBuffer + 88)
                && *((_BYTE *)i - 7) == *((_BYTE *)CommBuffer + 89)
                && *((_BYTE *)i - 6) == *((_BYTE *)CommBuffer + 90) )
            {
                if ( *((_DWORD *)i - 1) == 1 )
                {
                    if ( *((_BYTE *)i - 18) != *((_BYTE *)CommBuffer + 78) )
                        continue;
                    v6 = *((_BYTE *)i - 17) == *((_BYTE *)CommBuffer + 79);
                }
                else if ( *((_DWORD *)i - 1) )
                {
                    v6 = *(i - 2) == *((_QWORD *)CommBuffer + 10);
                }
                else
                {
                    if ( *((_WORD *)i - 14) != *((_WORD *)CommBuffer + 34) )
                        continue;
                    v6 = *((_BYTE *)i - 24) == *((_BYTE *)CommBuffer + 72);
                }
            }
            if ( v6 )
            {
                if ( (__QWORD *)qword_2470 == i )
                    qword_2470 = i[1];
                else
                    *((_QWORD *)i + 8i64) = i[1];
                if ( (__QWORD *)qword_2468 == i )
                    qword_2468 = *i;
                else
            }
        }
    }
}

```

Рисунок 31 - SmiHandler_D24

```

EFI_STATUS __fastcall SmiHandler_F3C(
    EFI_HANDLE DispatchHandle,
    const void *Context,
    void *CommBuffer,
    UINTN *CommBufferSize)
{
    qword_2370 = *(_QWORD *)CommBuffer;
    sub_8F8();
    return 0i64;
}

```

Рисунок 32 - SmiHandler_F3C

Несмотря на то, что автоматизированное средство проверки brick.py указывает, что CommBufferSize нигде в этих обработчиках не используется, на это есть причина - мы видим однотипную проверку нулевого поля структуры на равенство константе 608715844 (кроме SmiHandler_F3C). Это приводит к тому, что злоумышленник уже должен угадать константу, чтобы исполнился блок кода. По этой причине нет проверки на CommBufferSize. Более того, в CommBuffer ничего не записывается, поэтому перекрытие здесь невозможно. Поэтому, несмотря на результаты автоматической проверки (где говорилось, что если не используется CommBufferSize, то вероятно перекрытие), при ручной проверке мы убедились, что данные обработчики неподвержены описанной выше уязвимости. Уязвимость была бы возможна, если в сам CommBuffer происходила запись и не было бы проверки на CommBufferSize.

Обработчик SmiHandler_E64 не имеет векторов воздействия (рисунок 33).

```

EFI_STATUS __fastcall SmiHandler_E64(
    EFI_HANDLE DispatchHandle,
    const void *Context,
    void *CommBuffer,
    UINTN *CommBufferSize)
{
    char v5[24]; // [rsp+30h] [rbp-18h] BYREF
    v5[0] = 0xD1;
    gEfi_S3_Save_state->Write(gEfi_S3_Save_state, 0i64, 0i64, 178i64, 1i64, v5);
    return 0i64;
}

```

Рисунок 33 - SmiHandler_E64

Обработчик `SmiHandler_EA4` не имеет возможный вектор воздействия, например, `SMM callout`, так как не используется ни `EFI_RUNTIME_SERVICE`, ни `EFI_BOOT_SERVICE` структуры (рисунок 34)

```
EFI_STATUS __fastcall SmiHandler_EA4(
    EFI_HANDLE DispatchHandle,
    const void *Context,
    void *CommBuffer,
    UINTN *CommBufferSize)
{
    __int64 CommBuff_zero_field; // rcx
    __int64 efi_smm_save_sate_reg_rbx; // rbx
    unsigned int v6; // ebp
    unsigned int v7; // edi
    __int64 *from_comm_buffer; // rsi
    __int64 v9; // r8
    __int64 v11; // [rsp+50h] [rbp+18h] BYREF

    CommBuff_zero_field = *(_QWORD *)CommBuffer;
    LODWORD(efi_smm_save_sate_reg_rbx) = 0;
    v11 = 0i64;
    if ( CommBuff_zero_field )
    {
        v6 = 0;
        v7 = *(_DWORD *)((unsigned int *)CommBufferSize + CommBuff_zero_field - 4);
        if ( v7 )
        {
            from_comm_buffer = (__int64 *)(CommBuff_zero_field + 16);
            do
            {
                if ( (_DWORD)efi_smm_save_sate_reg_rbx == 1935831918 )
                    break;
                efi_smm_save_sate_reg_rbx = *(from_comm_buffer - 2);
                v9 = *((unsigned int *)from_comm_buffer - 2);
                v11 = *from_comm_buffer;
                gEfi_S3_Save_state->Write(gEfi_S3_Save_state, 4i64, v9, efi_smm_save_sate_reg_rbx, 1i64, &v11);
                ++v6;
                from_comm_buffer += 3;
            }
            while ( v6 < v7 );
        }
    }
    return 0i64;
}
```

Рисунок 34 - `SmiHandler_EA4`

Некоторые обработчики имеют предупреждение `nested pointers` (рисунок 26), которые рассмотрим подробно на явном примере - **`SdioSmm`**.

Модуль SdioSmm

SdioSmm - это Secure digital input output Smm.

Находим экспортируемую точку входа (рисунок 35).

```
EFI_STATUS __fastcall ModuleEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    EFI_BOOT_SERVICES *BootServices; // rax
    __int64 v5; // rax
    void *v6; // rcx
    __int64 v7; // rax
    void (__fastcall **v9)(_QWORD, _QWORD); // [rsp+30h] [rbp+8h] BYREF

    BootServices = SystemTable->BootServices;
    gImageHandle_1F88 = (__int64)ImageHandle;
    gST_1F90 = SystemTable;
    gBS_1F98 = BootServices;
    v9 = 0i64;
    BootServices->LocateProtocol(&gEfiSmmBase2ProtocolGuid_1860, 0i64, (void **)&v9);
    ((void (__fastcall **)(_QWORD, _EFI_SMM_SYSTEM_TABLE2 **))v9)[1](v9, &gSmst_1FA0);
    byte_2020 = 0;
    v5 = gSmst_1FA0->SmmLocateProtocol(&gAmiSmmBufferValidationProtocolGuid_18A0, 0i64, &qword_1FE8);
    v6 = qword_1FE8;
    if ( v5 < 0 )
    {
        v6 = 0i64;
        qword_1F78 = 0x8000000000000000i64; // всё, что выше должно было быть отдельной функцией, но видимо она стала inline
        qword_1FE8 = v6;
        if ( !sub_15D0(&unk_1E80) )
        {
            v7 = sub_458((__int64)ImageHandle, SystemTable);
            if ( v7 >= 0 || qword_1F78 < 0 )
            {
                qword_1F78 = v7;
                sub_1660(&unk_1E80, -1i64);
            }
        }
        return qword_1F78;
    }
}
```

Рисунок 35 – Точка входа в SdioSmm

Проводим аналогичные переименования в LocateProtocol, по определению этой функции, его последний аргумент совпадает с GUID (рисунок 36)

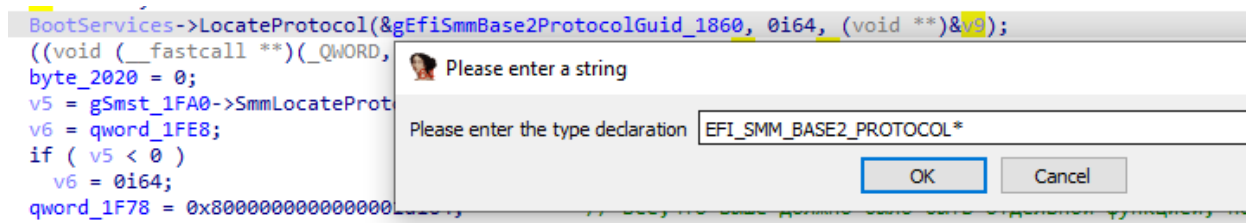


Рисунок 36 – Переименование параметра в LocateProtocol

На рисунке 37 происходит переименование переменной статуса.

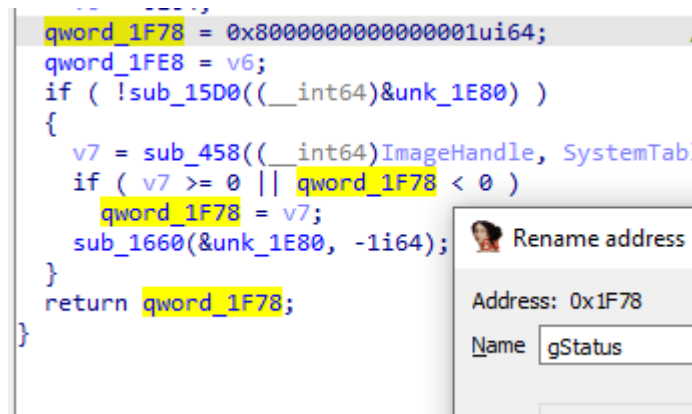


Рисунок 37 – Новое имя переменной статуса

Таким образом, получаем следующий ModuleEntryPoint (рисунок 38).

```
EFI_STATUS __fastcall ModuleEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    EFI_BOOT_SERVICES *BootServices; // rax
    __int64 v5; // rax
    void *v6; // rcx
    __int64 status; // rax
    EFI_SMM_BASE2_PROTOCOL *smmBase2Proto; // [rsp+30h] [rbp+8h] BYREF

    BootServices = SystemTable->BootServices;
    gImageHandle_1F88 = (__int64)ImageHandle;
    gST_1F90 = SystemTable;
    gBS_1F98 = BootServices;
    smmBase2Proto = 0i64;
    BootServices->LocateProtocol(&gEfiSmmBase2ProtocolGuid_1860, 0i64, (void **)&smmBase2Proto);
    ((void (__fastcall *) (EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))smmBase2Proto->GetSmstLocation)(
        smmBase2Proto,
        &gSmst_1FA0);
    byte_2020 = 0;
    v5 = gSmst_1FA0->SmmLocateProtocol(&gAmiSmmBufferValidationProtocolGuid_18A0, 0i64, &AmiSmmBufferValidation);
    v6 = AmiSmmBufferValidation;
    if ( v5 < 0 )
        v6 = 0i64;
    gStatus = 0x8000000000000001ui64; // всё, что выше должно было быть отдельной функцией, но видимо
    AmiSmmBufferValidation = v6;
    if ( !setjmp((__int64)&buf) )
    {
        status = SdioSmmEntryPoint(ImageHandle, SystemTable);
        if ( status >= 0 || gStatus < 0 )
            gStatus = status;
        longjmp(&buf, -1i64);
    }
    return gStatus;
}
```

Рисунок 38 – Новый ModuleEntryPoint

Аналогично заменяем фактические параметры функции на нужные (рисунок 39)

```
__int64 __fastcall SdioSmmEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    EFI_BOOT_SERVICES *BootServices; // r9
    EFI_RUNTIME_SERVICES *RuntimeServices; // rax
```

Рисунок 39 – Установка имён и типов данных фактических параметров функции

Далее изменяем тип данных (рисунок 40).

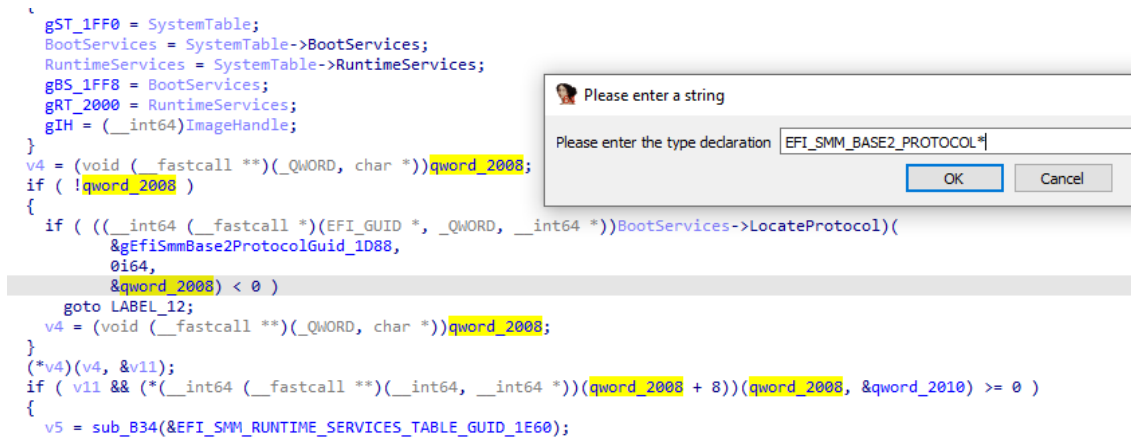


Рисунок 40 – Изменение типа данных согласно GUID

Изменим qword_2010 на gSMST и его тип данных на _EFI_SMM_SYSTEM_TABLE2* (рисунок 41).

```

8      goto LABEL_12;
9      gEfiSmmBase2Proto_ = gEfiSmmBase2Proto;
10     }
11     ((void (__fastcall *)(EFI_SMM_BASE2_PROTOCOL *, char *))gEfiSmmBase2Proto->InSmm)(gEfiSmmBase2Proto_, &in_smmram);
12     if ( in_smmram
13         && ((__int64 (__fastcall *)(EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))gEfiSmmBase2Proto->GetSmmLocation)(
14             gEfiSmmBase2Proto,
15             &gSMST ) >= 0 )
16     {
17         v5 = SdioSmmEFI_SMM_SYSTEM_TABLE2 *

```

Рисунок 41 – Аналогичные изменение имён и типов данных

Функция sub_B34 принимает вид (рисунок 42) и переименовывается в SdioSmmRuntimeServices. Здесь происходит подмена RUNTIME_SERVICES таблицы на SMM версию таблицы для разделения возможностей режима SMM и RING0.

```

UINT64 SdioSmmRuntimeServices()
{
    EFI_CONFIGURATION_TABLE *conf_table; // rbx
    __int64 v1; // r11

    if ( !gSMST )
        return 0i64;
    conf_table = gSMST->SmmConfigurationTable;
    if ( !gSMST->NumberOfTableEntries )
        return 0i64;
    while ( CmpGuid(conf_table, &EFI_SMM_RUNTIME_SERVICES_TABLE_GUID_1E60, 16i64) )
    {
        ++conf_table;
        if ( v1 == 1 )
            return 0i64;
    }
    return (UINT64)conf_table->VendorTable;
}

```

Рисунок 42 – Декомпилированная sub_B34 (SdioSmmRuntimeServices)

Далее рассмотрим функции sub_A0C() и sub_A50() – рисунки 43 и 44.

Первая – SdioSmmInintSmmStatusCodeProtocol, вторая – SdioSmmInintSomeProtocols. В первой происходит инициализация протокола EfiSmmStatusCodeProtocol.

```
int64 SdioSmmInintSmmStatusCodeProtocol()
{
    int64 result; // rax

    result = 0i64;
    if ( !gEfiSmmStatusCodeProtocol )
    {
        if ( gSMST_ )
            return ((int64 (__fastcall *))(EFI_GUID *, _QWORD, EFI_SMM_STATUS_CODE_PROTOCOL **))gSMST_->SmmLocateProtocol)(
                &gEfiSmmStatusCodeProtocolGuid_1DA8,
                0i64,
                &gEfiSmmStatusCodeProtocol);
        else
            return 0x8000000000000003ui64;
    }
    return result;
}
```

Рисунок 43 – функция sub_A0C, ставшая SdioSmmInintSmmStatusCodeProtocol

```
int64 SdioSmmInintSomeProtocols()
{
    int64 v1; // r11
    unsigned __int64 v2; // rbx
    void *v3; // rax

    if ( byte_1F83 )
    {
        if ( gAmiSmmDebugServProto )
            return 0i64;
        if ( !gSMST_ )
            return 0x8000000000000003ui64;
        v1 = ((int64 (__fastcall *))(EFI_GUID *, _QWORD, void **))gSMST_->SmmLocateProtocol)(
            &gAmiSmmDebugServiceProtocolGuid_1880,
            0i64,
            &gAmiSmmDebugServProto);
        if ( v1 < 0 )
            gAmiSmmDebugServProto = 0i64;
    }
    else
    {
        if ( gAmiDebugServicePpi )
            return 0i64;
        if ( gDebugFlag == 1 )
            return 0x8000000000000003ui64;
        v2 = ((int64 (__fastcall *))(__int64))gBS_1FF8->RaiseTPL(31i64);
        ((void (__fastcall *))(unsigned __int64))gBS_1FF8->RestoreTPL(v2);
        if ( v2 > 0x10 )
            return 0x8000000000000003ui64;
        v1 = gBS_1FF8->LocateProtocol(&gAmiDebugServiceProtocolGuid_1890, 0i64, &gAmiDebugServicePpi);
        v3 = gAmiDebugServicePpi;
        if ( v1 < 0 )
            v3 = 0i64;
        gAmiDebugServicePpi = v3;
    }
    return v1;
}
```

Рисунок 44 – функция sub_A50, ставшая SdioSmmInintSomeProtocols

Таким образом, SdioSmmEntryPoint принимает вид (рисунок 45)

```
int64 __fastcall SdioSmmEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    EFI_BOOT_SERVICES *BootServices; // r9
    EFI_RUNTIME_SERVICES *RuntimeServices; // rax
    EFI_SMM_BASE2_PROTOCOL *gEfiSmmBase2Proto; // rax
    UINT64 v5; // rax
    EFI_RUNTIME_SERVICES *_rt; // rcx
    __int64 result; // rax
    __int64 v8; // [rsp+20h] [rbp-28h] BYREF
    __int64 v9; // [rsp+28h] [rbp-20h] BYREF
    EFI_HANDLE DispatchHandle; // [rsp+30h] [rbp-18h] BYREF
    char in_smmram; // [rsp+60h] [rbp+18h] BYREF
    void *Interface; // [rsp+68h] [rbp+20h] BYREF

    v8 = 0i64;
    Interface = 0i64;
    DispatchHandle = 0i64;
    if ( gST_1FF0 )
    {
        BootServices = gBS_1FF8;
    }
    else
    {
        gST_1FF0 = SystemTable;
        BootServices = SystemTable->BootServices;
        RuntimeServices = SystemTable->RuntimeServices;
        gBS_1FF8 = BootServices;
        gRT_2000 = RuntimeServices;
        *(gSMST_ + 1) = (EFI_SMM_SYSTEM_TABLE *)ImageHandle;
    }
    gEfiSmmBase2Proto = gEfiSmmBase2Proto;
    if ( !gEfiSmmBase2Proto )
    {
        if ( ((__int64 (__fastcall *))(EFI_GUID *, _QWORD, EFI_SMM_BASE2_PROTOCOL *))BootServices->LocateProtocol)(
            &gEfiSmmBase2ProtocolGuid_1D88,
            0i64,
            &gEfiSmmBase2Proto ) < 0 )
            goto LABEL_12;
        gEfiSmmBase2Proto = gEfiSmmBase2Proto;
    }
    ((void (__fastcall *))(EFI_SMM_BASE2_PROTOCOL *, char *))gEfiSmmBase2Proto->InSmm(gEfiSmmBase2Proto, &in_smmram);
    if ( in_smmram )
    {
        && ((__int64 (__fastcall *))(EFI_SMM_BASE2_PROTOCOL *, EFI_SMM_SYSTEM_TABLE *))gEfiSmmBase2Proto->GetSmstLocation(
            gEfiSmmBase2Proto,
            &gSMST_ ) >= 0 )
        {
            v5 = SdioSmmRuntimeServices();
            _rt = gRT_2000;
            byte_1F83 = 1;
            gDebugFlag = 0;
            if ( v5 )
            {
                LABEL_12:
                if ( (gBS_1FF8->LocateProtocol(&gEfiSmmBase2ProtocolGuid_1860, 0i64, &::Interface) & 0x8000000000000000ui64) == 0i64
                    && *((__int64 (__fastcall **))(void *, _EFI_SMM_SYSTEM_TABLE2 **))::Interface + 1))::Interface, &gSmst_2030) >= 0
                    && (gSmst_2030->SmmLocateProtocol(&gEfiSmmSwDispatch2ProtocolGuid_1870, 0i64, &Interface) & 0x8000000000000000ui64) == 0i64 )
                {
                    v9 = 64i64;
                    result = ((*(__int64 (__fastcall **))(void *, __int64 (__fastcall *)(), __int64 *, __int64 *))Interface)(
                        Interface,
                        SwSmiHandler_5F0,
                        &v9,
                        &v8);
                    if ( result < 0 )
                        return result;
                    gSmst_2030->SmiHandlerRegister((EFI_SMM_HANDLER_ENTRY_POINT2)SmiHandler_39C, &HandlerType, &DispatchHandle);
                }
                return 0i64;
            }
        }
    }
}
```

Рисунок 45 – Измененная точка входа

Обработчики находятся в этой же функции. На рисунке 46 показан первоначальный вид

```
LABEL_12:
if ( (gBS_1FF8->LocateProtocol(&gEfiSmmBase2ProtocolGuid_1860, 0i64, &::Interface) & 0x8000000000000000ui64) == 0i64
    && *((__int64 (__fastcall **))(void *, _EFI_SMM_SYSTEM_TABLE2 **))::Interface + 1))::Interface, &gSmst_2030) >= 0
    && (gSmst_2030->SmmLocateProtocol(&gEfiSmmSwDispatch2ProtocolGuid_1870, 0i64, &Interface) & 0x8000000000000000ui64) == 0i64 )
{
    v9 = 64i64;
    result = ((*(__int64 (__fastcall **))(void *, __int64 (__fastcall *)(), __int64 *, __int64 *))Interface)(
        Interface,
        SwSmiHandler_5F0,
        &v9,
        &v8);
    if ( result < 0 )
        return result;
    gSmst_2030->SmiHandlerRegister((EFI_SMM_HANDLER_ENTRY_POINT2)SmiHandler_39C, &HandlerType, &DispatchHandle);
}
return 0i64;
}
```

Рисунок 46 - Найденные функции прерывания до изменений

Устанавливаем новые типы данных для `smm_base2_proto` и `smm_sw_dispatch` основываясь на их GUID, то есть `EFI_SMM_BASE2_PROTOCOL` и `EFI_SMM_SW_DISPATCH2_PROTOCOL` соответственно. А также устанавливаем callback функцию в Register (рисунок 47).



Рисунок 47 – Произведенные изменения

Таким образом, имеем два обработчика:

- SwSmiHandler_5F0 (рисунок 48)
- SmiHandler_39C (рисунок 50)

На рисунке видно, что используется адрес `loc_40E` (0x40E), в который злоумышленник может записать свои данные.

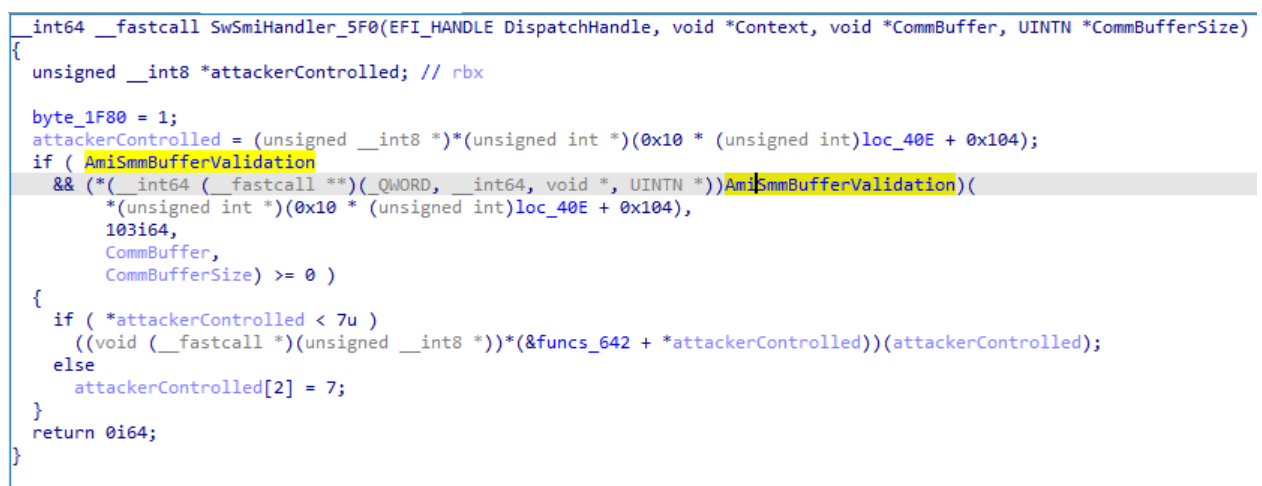


Рисунок 48 – Обработчик SwSmiHandler_5F0

Но `SwSmiHandler_5F0` не имеет векторов атаки, так как было реализовано исправление (рисунок 49).

Resolution

The fix is to avoid using the user-controlled contents at all when it points inside SMRAM, as shown below.

```
{
    // ...
    struct_v0 *userControlled = *(0x10 * MEMORY[0x40E] + 0x104);
    if (!EFI_ERROR(ValidateBufferIsOutsideSmram(userControlled, sizeof(struct_v0))))
    {
        if (userControlled->Offset0_FunctionCode < 7 )
        {
            // ...
        }
        else
        {
            userControlled->Offset2 = 7;
        }
    }
    return EFI_SUCCESS;
}
```

Рисунок 49– Исправления уязвимости [3]

Исправление состоит в том, чтобы не использовать контролируемое злоумышленником содержимое, когда оно указывает на SMRAM.

```
EFI_STATUS __fastcall SmiHandler_39C(
    EFI_HANDLE DispatchHandle,
    const void *Context,
    void *CommBuffer,
    UINTN *CommBufferSize)
{
    _QWORD *v4; // rdi
    unsigned __int8 i; // bl
    __int64 v6; // rcx
    _BYTE *v7; // rdx
    _BYTE *v8; // rcx

    if ( !byte_1F80 && CommBuffer && CommBufferSize )
    {
        v4 = *(_QWORD **)CommBuffer;
        if ( !Buffer )
        {
            gSmst_2030->SmmAllocatePool(EfiRuntimeServicesData, 0x1658ui64, &Buffer);
            sub_15B0((char *)Buffer, 0x1658ui64);
            *(_QWORD *)Buffer = *v4;
        }
        for ( i = 0; i < 8u; ++i )
        {
            v6 = 89i64 * i;
            v7 = &v4[v6 + 3];
            if ( !*v7 )
                break;
            v8 = (char *)Buffer + v6 * 8 + 24;
            if ( !*v8 && v8 != v7 )
                CopyMem(v8, v7, 0x2C8ui64);
        }
    }
    return 0i64;
}
```

Рисунок 50 - SmiHandler_39C

Снова проведем автоматический анализ (рисунок 51).

C:\Users\Seven\Documents\brick\brick\UX310UAK-AS.bin.output\SdioSmm.efi		
<ul style="list-style-type: none">• SUCCESS (efiXplorer_module.py): efiXplorer didn't detect any vulnerabilities• ERROR (check_nested_pointers.py): SMI SmmHandler_39C does not validate pointers nested in the CommBuffer• VERBOSE (check_nested_pointers.py): SMI SwSmmHandler_5F0 does not expose any attack surface• SUCCESS (smmram_overlap.py): No SMI that omits checking CommBufferSize was found• SUCCESS (toctou.py): No double fetches from the Comm Buffer were encountered• WARNING (legacy_protocols.py): Found legacy protocol EfiDiskIoProtocolGuid ce345171-ba0b-11d2-8e4f-00a0c969723b at 0x1d68• WARNING (legacy_protocols.py): Found legacy protocol EfiComponentNameProtocolGuid 107a772c-d5e1-11d4-9a46-0090273fc14d at 0x1d98• INFO (is_edk2.py): SdioSmm is not from EDK2, probably OEM specific• SUCCESS (scan_cseg.py): No SMIs that have CSEG specific behavior were found• SUCCESS (setvar_infoleak.py): No potential SetVariable() info leaks were detected		
IDA database: C:\Users\Seven\Documents\brick\brick\UX310UAK-AS.bin.output\SdioSmm.id4	Raw IDA log: C:\Users\Seven\Documents\brick\brick\UX310UAK-AS.bin.output\SdioSmm.log	efiXplorer report: C:\Users\Seven\Documents\brick\brick\UX310UAK-AS.bin.output\SdioSmm.json

Рисунок 51– Результат автоматического анализа

Эвристика этого инструмента состоит в следующем: сначала проверяется используется ли вообще в обработчике CommBuffer, если нет, то и опасности тоже нет. Затем проверяется используется ли вызов функции SmmIsBufferOutsideSmmValid (или его API аналога – ValidateMemoryBuffer). Если такая функция не используется, то проверяется используются ли nested_pointers, то есть указатели в if-else или switch case. Таким образом, может оказаться, что в обработчик приходит непроверенный указатель, то есть появляется новый вектор атаки.

Эта атака заключается в том, что при отсутствии проверки SmmIsBufferOutsideSmmValid (находится в SMMLockBox), атакующий может создать коммуникационный буфер где-то в физической памяти, которая ему доступна (не в SMRAM), затриггерить уязвимый SMI, который не проверит, что коммуникационный буфер указывает на SMRAM и исполнить код в ring - 2 (рисунок 52). Таким образом, можно совершить confused deputy attack на SMI обработчик. Эта атака сводится к тому, что путём такого обманного воздействия на SMI обработчик, мы можем косвенно исполнить код в привилегированном режиме, так как обработчик находится в SMRAM с максимальными привилегиями (и при том, что атакующий не имел на это прав – за него это сделал обработчик – он же confused deputy) [4].

Теперь необходимо проверить результаты автоматического анализа вручную. Действительно, как показано на рисунке 50 в обработчике не производится проверка функцией SmmIsBufferOutsideSmmValid, что может привести к исполнению произвольного кода в SMRAM. Таким образом, разработчики должны были добавить следующую проверку:

```
if (!SmmIsBufferOutsideSmmValid (CommBuffer, CommBufferSize))
return EFI_ACCESS_DENIED;
```

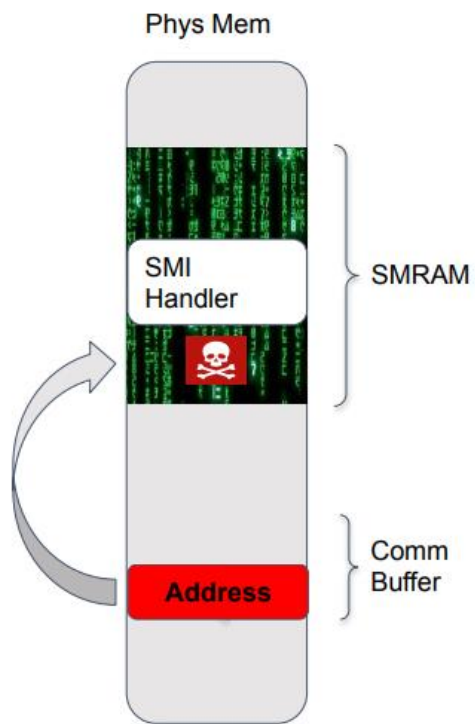



Рисунок 52 – Иллюстрация уязвимости [2]

Модуль SbRunSmm

Находим экспортируемую точку входа (рисунок 53).

```
EFI_STATUS __fastcall ModuleEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    __int64 v4; // rax
    EFI_STATUS v5; // rbx

    sub_310((__int64)ImageHandle, SystemTable);
    qword_3048 = 0x8000000000000001ui64;
    if ( !sub_2390(&unk_2F50) )
    {
        v4 = sub_808(ImageHandle, SystemTable);
        if ( v4 >= 0 || qword_3048 < 0 )
            qword_3048 = v4;
        sub_2420(&unk_2F50, -1i64);
    }
    v5 = qword_3048;
    if ( qword_3048 < 0 )
        sub_14F4(Buffer);
    return v5;
}
```

Рисунок 53 – Первоначальный вид точки входа

Проводим переименования фактических параметров функции sub_310 (рисунок 54).

```
int64 __fastcall sub_310(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *a2)
{
    EFI_BOOT_SERVICES *BootServices; // r9
    EFI_RUNTIME_SERVICES *RuntimeServices; // rax
    UINTN NumberOfTableEntries; // r10
    UINTN v7; // rax
    EFI_CONFIGURATION_TABLE *ConfigurationTable; //
    void (__fastcall **v9)(void *, _EFI_SMM_SYSTEM_
```

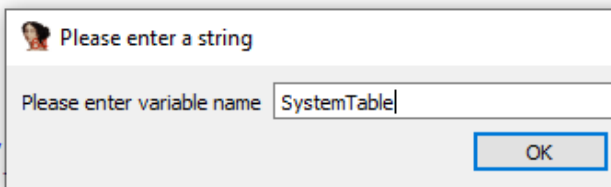


Рисунок 54 – Фактические аргументы

Далее производится переопределение последнего параметра в соответствии с указанным GUID (рисунок 55).

```
19  c_RunTimeServ = (__int64)RuntimeServices;
20  ((void (__fastcall *) (EFI_GUID *, _QWORD, __int64 *))BootServices->LocateProtocol)(
21  &gEfiSmmBase2ProtocolGuid_2790,
22  0i64,
23  &v12);
```

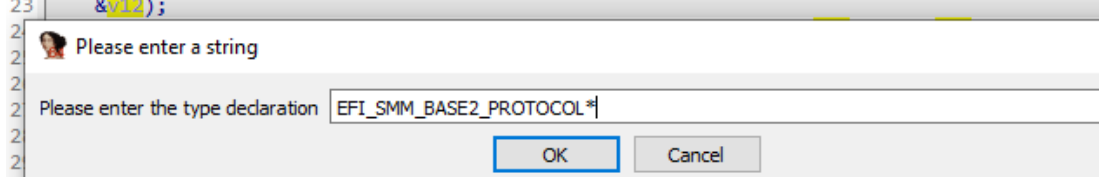


Рисунок 55 – Переопределение типа данных

После этого задаём имя согласно контексту (рисунок 56).

```
((void (__fastcall *)(EFI_GUID *, _QWORD, EFI_SMM_BASE2_PROTOCOL **))BootServices->LocateProtocol)(
    &gEfiSmmBase2ProtocolGuid_2790,
    0i64,
    &smmBase2Proto);
((void (__fastcall *)(EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))smmBase2Proto->GetSmstLocation)(
    smmBase2Proto,
    &gSmst_30A0);
```

Рисунок 56 – Задание имени smmBase2Proto

Затем аналогично вышеуказанному продолжаем проводить изменения (рисунки 57 и 58).

```
gBS_3090->LocateProtocol(&gEfiSmmAccess2ProtocolGuid_27C0, 0i64, &Interface);
v11 = 0i64;
*((void (__fastcall **))(void *, unsigned __int64 *, _QWORD))Interface + 3)((Interface, &v11, 0i64);
Buffer = (void *)sub_14C4(6i64, v11);
*((void (__fastcall **))(void *, unsigned __int64 *, void *))Interface + 3)((Interface, &v11, Buffer);
qword_30F0 = 0i64;
NumberOfTableEntries = gST_3088->NumberOfTableEntries;
```

Рисунок 57 – Первоначальный вид до изменений

```
gBS_3090->LocateProtocol(&gEfiSmmAccess2ProtocolGuid_27C0, 0i64, (void **)&gEfiSmmAccessProto);
v11 = 0i64;
((void (__fastcall *)(EFI_SMM_ACCESS_PROTOCOL *, UINTN *, _QWORD))gEfiSmmAccessProto->GetCapabilities)(
    gEfiSmmAccessProto,
    &v11,
    0i64);
Buffer = sub_14C4(6i64, v11);
((void (__fastcall *)(EFI_SMM_ACCESS_PROTOCOL *, UINTN *, void *))gEfiSmmAccessProto->GetCapabilities)(
    gEfiSmmAccessProto,
    &v11,
    Buffer);
```

EFI_SMM_ACCESS_PROTOCOL *gEfiSmmAccessProto; // [rsp+40h] [rbp+18h] BYREF

Рисунок 58 – Выставлены тип данных EFI_SMM_ACCESS_PROTOCOL и имя из контекста

Обратившись к исходникам AMI_BIOS стало ясно, что в функции GetCapabilities для последнего параметра используется тип данных EFI_SMRAM_DESCRIPTOR* (рисунок 59).

```
SmramRanges = (EFI_SMRAM_DESCRIPTOR *)sub_14C4(6i64, v11);
((void (__fastcall *)(EFI_SMM_ACCESS_PROTOCOL *, UINTN *, EFI_SMRAM_DESCRIPTOR *))gEfiSmmAccessProto->GetCapabilities)(
    gEfiSmmAccessProto,
    &v11,
    SmramRanges);
```

Рисунок 59– Верный тип данных у SmramRanges

На рисунках 60-61 показаны изменения, произошедшие в ходе реверс-инжиниринга.

```

LABEL_8:
v9 = (void (__fastcall **)(void *, _EFI_SMM_SYSTEM_TABLE2 **))Interface;
if ( Interface )
goto LABEL_11;
if ( (gBS_3090->LocateProtocol(&gEfiSmmBase2ProtocolGuid_2790, 0i64, &Interface) & 0x8000000000000000ui64) == 0i64 )
{
v9 = (void (__fastcall **)(void *, _EFI_SMM_SYSTEM_TABLE2 **))Interface;
LABEL_11:
(*v9)(v9, &gSmst_30F8);
}
sub_1FE0((__int64)ImageHandle, SystemTable);
return ((__int64 (__fastcall *)(EFI_GUID *, _QWORD, __int64 *))gSmst_30A0->SmmLocateProtocol)(
&gEfiS3SmmSaveStateProtocolGuid_27D0,
0i64,
&qword_3118);
}

```

Рисунок 60 – До применения изменений

```

LABEL_8:
v9 = gEfiSmmBase2Proto;
if ( gEfiSmmBase2Proto )
goto LABEL_11;
if ( (gBS_3090->LocateProtocol(&gEfiSmmBase2ProtocolGuid_2790, 0i64, (void **)&gEfiSmmBase2Proto) & 0x8000000000000000ui64) == 0i64 )
{
v9 = gEfiSmmBase2Proto;
LABEL_11:
((void (__fastcall *)(EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))v9->InSmm)(v9, &gSmst_30F8);
}
sub_1FE0((__int64)ImageHandle, SystemTable);
return ((__int64 (__fastcall *)(EFI_GUID *, _QWORD, EFI_S3_SMM_SAVE_STATE_PROTOCOL **))gSmst_30A0->SmmLocateProtocol)(
&gEfiS3SmmSaveStateProtocolGuid_27D0,
0i64,
&gEfiS3SmmSaveStateProto);
}

```

Рисунок 61 – Примененные изменения

На рисунках 62 и 63 показан финальный вид функции sub_310, переименованной в Init.

```

int64 __fastcall Init(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    EFI_BOOT_SERVICES *BootServices; // r9
    EFI_RUNTIME_SERVICES *RuntimeServices; // rax
    UINTN NumberOfTableEntries; // r10
    UINTN v7; // rax
    EFI_CONFIGURATION_TABLE *ConfigurationTable; // rcx
    EFI_SMM_BASE2_PROTOCOL *v9; // rax
    UINTN v11; // [rsp+30h] [rbp+8h] BYREF
    EFI_SMM_BASE2_PROTOCOL *smmBase2Proto; // [rsp+38h] [rbp+10h] BYREF
    EFI_SMM_ACCESS_PROTOCOL *gEfiSmmAccessProto; // [rsp+40h] [rbp+18h] BYREF

    BootServices = SystemTable->BootServices;
    RuntimeServices = SystemTable->RuntimeServices;
    smmBase2Proto = 0i64;
    gIH = (__int64)ImageHandle;
    gST_3088 = SystemTable;
    gBS_3090 = BootServices;
    c_RunTimeServ = (__int64)RuntimeServices;
    ((void (__fastcall *)(EFI_GUID *, _QWORD, EFI_SMM_BASE2_PROTOCOL **))BootServices->LocateProtocol)(
        &gEfiSmmBase2ProtocolGuid_2790,
        0i64,
        &smmBase2Proto);
    ((void (__fastcall *)(EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))smmBase2Proto->GetSmstLocation)(
        smmBase2Proto,
        &gSmst_30A0);
    byte_3168 = 0;
    gBS_3090->LocateProtocol(&gEfiSmmAccess2ProtocolGuid_27C0, 0i64, (void **)&gEfiSmmAccessProto);
    v11 = 0i64;
    ((void (__fastcall *)(EFI_SMM_ACCESS_PROTOCOL *, UINTN *, _QWORD))gEfiSmmAccessProto->GetCapabilities)(
        gEfiSmmAccessProto,
        &v11,
        0i64);
    SmmramRanges = (EFI_SMRAM_DESCRIPTOR *)sub_14C4(6i64, v11);
    ((void (__fastcall *)(EFI_SMM_ACCESS_PROTOCOL *, UINTN *, EFI_SMRAM_DESCRIPTOR *))gEfiSmmAccessProto->GetCapabilities)(
        gEfiSmmAccessProto,
        &v11,
        SmmramRanges);
    l_vendorTable = 0i64;
    NumberOfTableEntries = gST_3088->NumberOfTableEntries;
    v7 = 0i64;
    qword_3160 = v11 >> 5;
    if ( NumberOfTableEntries )
    {
        ConfigurationTable = gST_3088->ConfigurationTable;
        while ( *(_QWORD *)&gEfiHobListGuid_2750.Data1 != *(_QWORD *)&ConfigurationTable->VendorGuid.Data1
            || *(_QWORD *)&gEfiHobListGuid_2750.Data4 != *(_QWORD *)&ConfigurationTable->VendorGuid.Data4 )
        {
            ++v7;
            ++ConfigurationTable;
            if ( v7 >= NumberOfTableEntries )
            {
                ConfigurationTable = gST_3088->ConfigurationTable;
                while ( *(_QWORD *)&gEfiHobListGuid_2750.Data1 != *(_QWORD *)&ConfigurationTable->VendorGuid.Data1
                    || *(_QWORD *)&gEfiHobListGuid_2750.Data4 != *(_QWORD *)&ConfigurationTable->VendorGuid.Data4 )
                {
                    ++v7;
                    ++ConfigurationTable;
                    if ( v7 >= NumberOfTableEntries )
                    {
                        goto LABEL_8;
                    }
                }
                l_vendorTable = (__int64)gST_3088->ConfigurationTable[v7].VendorTable;
            }
        }
    }
    LABEL_8:
    v9 = gEfiSmmBase2Proto;
    if ( gEfiSmmBase2Proto )
        goto LABEL_11;
    if ( (gBS_3090->LocateProtocol(&gEfiSmmBase2ProtocolGuid_2790, 0i64, (void **)&gEfiSmmBase2Proto) & 0x8000000000000000ui64) == 0i64 )
    {
        v9 = gEfiSmmBase2Proto;
    }
    LABEL_11:
    ((void (__fastcall *)(EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))v9->InSmm)(v9, &gSmst_30F8);
    sub_1FE0((__int64)ImageHandle, SystemTable);
    return ((__int64 (__fastcall *)(EFI_GUID *, _QWORD, EFI_S3_SMM_SAVE_STATE_PROTOCOL **))gSmst_30A0->SmmLocateProtocol)(
        &gEfiS3SmmSaveStateProtocolGuid_27D0,
        0i64,
        &gEfiS3SmmSaveStateProto);
}

```

Рисунок 62 – Функция Init

```

v7 = 0i64;
qword_3160 = v11 >> 5;
if ( NumberOfTableEntries )
{
    ConfigurationTable = gST_3088->ConfigurationTable;
    while ( *(_QWORD *)&gEfiHobListGuid_2750.Data1 != *(_QWORD *)&ConfigurationTable->VendorGuid.Data1
        || *(_QWORD *)&gEfiHobListGuid_2750.Data4 != *(_QWORD *)&ConfigurationTable->VendorGuid.Data4 )
    {
        ++v7;
        ++ConfigurationTable;
        if ( v7 >= NumberOfTableEntries )
            goto LABEL_8;
    }
    l_vendorTable = (__int64)gST_3088->ConfigurationTable[v7].VendorTable;
}
LABEL_8:
v9 = gEfiSmmBase2Proto;
if ( gEfiSmmBase2Proto )
    goto LABEL_11;
if ( (gBS_3090->LocateProtocol(&gEfiSmmBase2ProtocolGuid_2790, 0i64, (void **)&gEfiSmmBase2Proto) & 0x8000000000000000ui64) == 0i64 )
{
    v9 = gEfiSmmBase2Proto;
}
LABEL_11:
((void (__fastcall *)(EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))v9->InSmm)(v9, &gSmst_30F8);
sub_1FE0((__int64)ImageHandle, SystemTable);
return ((__int64 (__fastcall *)(EFI_GUID *, _QWORD, EFI_S3_SMM_SAVE_STATE_PROTOCOL **))gSmst_30A0->SmmLocateProtocol)(
    &gEfiS3SmmSaveStateProtocolGuid_27D0,
    0i64,
    &gEfiS3SmmSaveStateProto);
}

```

Рисунок 63 – Функция Init (продолжение)

На рисунке 64 представлена измененная точка входа.

```
EFI_STATUS __fastcall ModuleEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    __int64 v4; // rax
    EFI_STATUS v5; // rbx

    Init(ImageHandle, SystemTable);
    gStatus = 0x8000000000000001ui64;
    if ( !setjmp(&buf) )
    {
        v4 = sub_808(ImageHandle, SystemTable);
        if ( v4 >= 0 || gStatus < 0 )
            gStatus = v4;
        longjmp(&buf, -1i64);
    }
    v5 = gStatus;
    if ( gStatus < 0 )
        sub_14F4();
    return v5;
}
```

Затем процедура sub_808 переименовывается в SbRunSmmEntryPoint и на рисунках 65 и 66 показана эта функция до внесения изменений.

```

__int64 __fastcall SbRunSmmEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
{
    EFI_SYSTEM_TABLE *v3; // rdx
    EFI_BOOT_SERVICES *BootServices; // r9
    EFI_RUNTIME_SERVICES *RuntimeServices; // rax
    EFI_RUNTIME_SERVICES *v6; // rax
    __int64 v7; // rdi
    __int64 v8; // rax
    EFI_RUNTIME_SERVICES *v9; // rcx
    char v11; // [rsp+40h] [rbp+18h] BYREF

    v3 = gST_3128;
    if ( gST_3128 )
    {
        BootServices = gBS_3130;
    }
    else
    {
        gST_3128 = SystemTable;
        BootServices = SystemTable->BootServices;
        RuntimeServices = SystemTable->RuntimeServices;
        gBS_3130 = BootServices;
        gRT_3138 = RuntimeServices;
        v3 = SystemTable;
        qword_3150 = (__int64)ImageHandle;
    }
    v11 = 0;
    if ( !v3 )
    {
        gST_3128 = SystemTable;
        BootServices = SystemTable->BootServices;
        v6 = SystemTable->RuntimeServices;
        gBS_3130 = BootServices;
        gRT_3138 = v6;
        qword_3150 = (__int64)ImageHandle;
    }
    v7 = ((__int64 (__fastcall *) (EFI_GUID *, _QWORD, __int64 *)) BootServices->LocateProtocol)(
        &gEfiSmmBase2ProtocolGuid_2D48,
        0i64,
        &qword_3140);
    if ( v7 >= 0 )
    {
        (*(void (__fastcall **)(__int64, char *)) qword_3140)(qword_3140, &v11);
        if ( v11 )
        {
            v7 = (*(__int64 (__fastcall **)(__int64, __int64 *)) (qword_3140 + 8))(qword_3140, &qword_3148);
            if ( v7 >= 0 )
            {
                v8 = sub_13B4(&EFI_SMM_RUNTIME_SERVICES_TABLE_GUID_2EB0);
                v9 = gRT_3138;
                byte_307F = 1;
            }
        }
    }
}

```

Рисунок 65— До внесений изменений в функцию SbRunSmmEntryPoint

```

        byte_307F = 1;
        byte_307D = 0;
        if ( v8 )
            v9 = (EFI_RUNTIME_SERVICES *)v8;
        gRT_3138 = v9;
        sub_11D4();
        sub_12D0();
        byte_307D = 1;
        return sub_758();
    }
}
return v7;
}

```

Рисунок 66 – До внесений изменений в функцию SbRunSmmEntryPoint
(продолжение)

Далее изменяем имя переменной на gSMST и её тип данных на _EFI_SMM_SYSTEM_TABLE2 (рисунок 67), чтобы в дальнейшем эта переменная могла быть верно использована в контексте других функций, например, SbRunSmmRuntimeServices (рисунок 68)

```
v7 = ((__int64 (__fastcall *) (EFI_SMM_BASE2_PROTOCOL *, _EFI_SMM_SYSTEM_TABLE2 **))gEfiSmmBase2Proto_>GetSmstLocation)(
    gEfiSmmBase2Proto_,
    &gSMST_);
if ( v7 >= 0 )
{
    _EFI_SMM_SYSTEM_TABLE2 *
    v8 = (EFI_RUNTIME_SERVICES *)SbRunSmmRuntimeServices();
    v9 = 0RT_3138;
```

Рисунок 67 – Установление верного типа данных

На рисунке 68 функция SbRunSmmRuntimeServices подменяет RUNTIME_SERVICES таблицы на SMMную версию, для того чтобы разделить возможности SMM и RING0.

```
void *SbRunSmmRuntimeServices()
{
    EFI_CONFIGURATION_TABLE *SmmConfigurationTable; // rbx
    __int64 v1; // r11

    if ( !gSMST_ )
        return 0i64;
    SmmConfigurationTable = gSMST_>SmmConfigurationTable;
    if ( !gSMST_>NumberOfTableEntries )
        return 0i64;
    while ( CmpGuid((unsigned __int64)SmmConfigurationTable, &EFI_SMM_RUNTIME_SERVICES_TABLE_GUID_2EB0) )
    {
        ++SmmConfigurationTable;
        if ( v1 == 1 )
            return 0i64;
    }
    return SmmConfigurationTable->VendorTable;
}
```

Рисунок 68 – Функция SbRunSmmRuntimeServices

Рассмотрим оставшиеся функции (рисунок 69). На рисунке 70 представлена функция SbRunSmmStatusCodeProtocol, на рисунке 71- SbRunSmmInitSomeProto.

```
v8 = (EFI_RUNTIME_SERVICES *)SbRunSmmRuntimeServices();
v9 = gRT_3138;
Flag1 = 1;
Flag2 = 0;
if ( v8 )
    v9 = v8;
gRT_3138 = v9;
SbRunSmmStatusCodeProtocol();
SbRunSmmInitSomeProto();
Flag2 = 1;
return sub_758();
```

Рисунок 69– Оставшиеся функции


```

int64 SbRunSmmStatusCodeProtocol()
{
    __int64 result; // rax

    result = 0i64;
    if ( Flag1 )
    {
        if ( gEfiSmmStatusCodeProto_ )
            return result;
        if ( !Flag2 && gSMST_ )
            return ((__int64 (__fastcall *))(EFI_GUID *, _QWORD, EFI_SMM_STATUS_CODE_PROTOCOL **))gSMST_->SmmLocateProtocol)(
                &gEfiSmmStatusCodeProtocolGuid_2D68,
                0i64,
                &gEfiSmmStatusCodeProto_);
        return 0x8000000000000003ui64;
    }
    if ( !gEfiStatusCode )
    {
        if ( Flag2 )
            return 0x8000000000000003ui64;
        return gBS_3130->LocateProtocol(&gEfiStatusCodeRuntimeProtocolGuid_2D88, 0i64, (void **)&gEfiStatusCode);
    }
    return result;
}

```

Рисунок 70 – Инициализация протоколов EfiSmmStatusCodeProtocol и EfiStatusCodeRuntimeProtocol

```

__int64 SbRunSmmInitSomeProto()
{
    __int64 v1; // r11
    unsigned __int64 v2; // rbx
    void *v3; // rax

    if ( Flag1 )
    {
        if ( gAmiSmmDebugServ )
            return 0i64;
        if ( !gSMST_ )
            return 0x8000000000000003ui64;
        v1 = ((__int64 (__fastcall *))(EFI_GUID *, _QWORD, __int64 *))gSMST_->SmmLocateProtocol)(
            &gAmiSmmDebugServiceProtocolGuid_27A0,
            0i64,
            &gAmiSmmDebugServ);
        if ( v1 < 0 )
            gAmiSmmDebugServ = 0i64;
    }
    else
    {
        if ( gAmiDebugServProto )
            return 0i64;
        if ( Flag2 == 1 )
            return 0x8000000000000003ui64;
        v2 = ((__int64 (__fastcall *))(__int64))gBS_3130->RaiseTPL(31i64);
        ((void (__fastcall *))(unsigned __int64))gBS_3130->RestoreTPL(v2);
        if ( v2 > 0x10 )
            return 0x8000000000000003ui64;
        v1 = gBS_3130->LocateProtocol(&gAmiDebugServiceProtocolGuid_27B0, 0i64, &gAmiDebugServProto);
        v3 = gAmiDebugServProto;
        if ( v1 < 0 )
            v3 = 0i64;
        gAmiDebugServProto = v3;
    }
    return v1;
}

```

Рисунок 71 – Инициализация протоколов AmiSmmDebugServiceProtocol и AmiDebugServiceProtocol

Функция sub_758 содержит в себе обработчик прерывания (рисунок 72). Далее sub_758 переименовывается в Handler.

```
int64 sub_758()
{
    __int64 v1[3]; // [rsp+20h] [rbp-18h] BYREF
    void (__fastcall **v2)(__QWORD, __int64 (__fastcall *)()), __int64 *, __int64 *; // [rsp+50h] [rbp+18h] BYREF
    __int64 v3; // [rsp+58h] [rbp+20h] BYREF

    v2 = 0i64;
    v3 = 0i64;
    v1[0] = 187i64;
    gRT_3138->ResetSystem = (EFI_RESET_SYSTEM)sub_152C;
    gRT_3138->GetTime = (EFI_GET_TIME)sub_1844;
    gRT_3138->SetTime = (EFI_SET_TIME)sub_1A1C;
    gRT_3138->GetWakeUpTime = (EFI_GET_WAKEUP_TIME)sub_1C5C;
    gRT_3138->SetWakeUpTime = (EFI_SET_WAKEUP_TIME)sub_1E38;
    ((void (__fastcall *))(EFI_GUID *, __QWORD, void (__fastcall **)(__QWORD, __int64 (__fastcall *)()), __int64 *, __int64 *)))gSMST_>SmmLocateProtocol)(
        &gEfiSmmSwDispatch2ProtocolGuid_2780,
        0i64,
        &v2);
    (*v2)(v2, SwSmiHandler_4B4, v1, &v3);
    return 0i64;
}
```

Рисунок 72 – Функция sub_758

На рисунке 73 представлен обработчик.

```
int64 Handler()
{
    __int64 v1[3]; // [rsp+20h] [rbp-18h] BYREF
    EFI_SMM_SW_DISPATCH2_PROTOCOL *gEfiSmmSwDispatchProto; // [rsp+50h] [rbp+18h] BYREF
    __int64 v3; // [rsp+58h] [rbp+20h] BYREF

    gEfiSmmSwDispatchProto = 0i64;
    v3 = 0i64;
    v1[0] = 187i64;
    gRT_3138->ResetSystem = (EFI_RESET_SYSTEM)sub_152C;
    gRT_3138->GetTime = (EFI_GET_TIME)sub_1844;
    gRT_3138->SetTime = (EFI_SET_TIME)sub_1A1C;
    gRT_3138->GetWakeUpTime = (EFI_GET_WAKEUP_TIME)sub_1C5C;
    gRT_3138->SetWakeUpTime = (EFI_SET_WAKEUP_TIME)sub_1E38;
    ((void (__fastcall *))(EFI_GUID *, __QWORD, EFI_SMM_SW_DISPATCH2_PROTOCOL **))gSMST_>SmmLocateProtocol)(
        &gEfiSmmSwDispatch2ProtocolGuid_2780,
        0i64,
        &gEfiSmmSwDispatchProto);
    ((void (__fastcall *))(EFI_SMM_SW_DISPATCH2_PROTOCOL *, __int64 (__fastcall *) (EFI_HANDLE, void *, void *, UINTN *), __int64 *, __int64 *))gEfiSmmSwDispatchProto->Register)(
        gEfiSmmSwDispatchProto,
        SwSmiHandler_4B4,
        v1,
        &v3);
    return 0i64;
}
```

Рисунок 73 – Callback функции добавлены аргументы обработчика

Рассмотрим содержимое SwSmiHandler_4B4 (рисунок 74)

```

    }
    while ( v10 );
    v12 = __inbyte(0x1804u);
    v16[0] = v12 | 1;
    gEfiS3SmmSaveStateProto->Write(gEfiS3SmmSaveStateProto, 0i64, 0i64);
    v13 = __indword(0x1830u);
    v19 = v13 & 0xFFFFFFFFB7;
    gEfiS3SmmSaveStateProto->Write(gEfiS3SmmSaveStateProto, 0i64, 2i64);
    v18[0] = 16;
    gEfiS3SmmSaveStateProto->Write(gEfiS3SmmSaveStateProto, 0i64, 1i64, 6144i64, 1i64, v18);
    DataSize[0] = 4125i64;
    gEfiSmmSwDispatchProto->GetVariable(aMesetup, &VendorGuid, 0i64, (UINTN *)DataSize, Data);
    if ( !Data[1755] && v14 >= 0 )
    {
        EFI_RUNTIME_SERVICES *
    }
```

Рисунок 74 – Runtime сервис, доступный в обработчике

Обратимся к результатам проверки efiXplorer (рисунок 75)

```

"vulns": {
  "smm_callout": [
    1670
  ]
}
```

Рисунок 75 - JSON с SMM callout функцией

Опишем вектор атаки SMM callout. Согласно спецификации в обработчике могут использоваться только EFI_SMM_SYSTEM_TABLE функции и SMM протоколы, но никак не EFI_RUNTIME_SERVICES (или EFI_BOOT_SERVICES) функции. Это ошибка разработчиков, так как обработчик SMI вызывает код, находящийся вне SMRAM. Тогда атакующий с правами на запись в физическую память может перезаписать адрес функции типа EFI_RUNTIME_SERVICE (например, GetVariable, SetVariable) на свой шеллкод и выполнить этот шеллкод в режиме SMM [5].

Таким образом, на рисунке 74 адрес функции GetVariable типа EFI_RUNTIME_SERVICE может быть перезаписан адресом шеллкода и этот шеллкод будет исполнен в режиме SMM. Это возможно, потому что не установлен SMM_Code_Chk_En, который запрещает запуск какого-либо кода, находящегося снаружи SMRAM (врезка 1). Другими словами, SMM_Code_Chk_En запрещает процессору исполнять код снаружи диапазона SMRR (System Management Range Register), когда процессор находится в режиме SMM.

```
#### common.smm_code_chk
```

```
[*] running module: chipsec.modules.common.smm_code_chk
```

```
[x] [=====
```

```
[x] [ Module: SMM_Code_Chk_En (SMM Call-Out) Protection
```

```
[x] [=====
```

```
[*] MSR_SMM_FEATURE_CONTROL = 0x00000000 << Enhanced SMM Feature Control (MSR 0x4E0 Thread 0x0)
```

```
  [00] LOCK      = 0 << Lock bit
```

```
  [02] SMM_Code_Chk_En = 0 << Prevents SMM from executing code outside the ranges defined by the SMRR
```

```
[*] MSR_SMM_FEATURE_CONTROL = 0x00000000 << Enhanced SMM Feature Control (MSR 0x4E0 Thread 0x0)
```

```
  [00] LOCK      = 0 << Lock bit
```

```
  [02] SMM_Code_Chk_En = 0 << Prevents SMM from executing code outside the ranges defined by the SMRR
```

```
[*] MSR_SMM_FEATURE_CONTROL = 0x00000000 << Enhanced SMM Feature Control (MSR 0x4E0 Thread 0x0)
```

```
  [00] LOCK      = 0 << Lock bit
```

```
  [02] SMM_Code_Chk_En = 0 << Prevents SMM from executing code outside the ranges defined by the SMRR
```

```
[*] MSR_SMM_FEATURE_CONTROL = 0x00000000 << Enhanced SMM Feature Control (MSR 0x4E0 Thread 0x0)
```

```
  [00] LOCK      = 0 << Lock bit
```

```
  [02] SMM_Code_Chk_En = 0 << Prevents SMM from executing code outside the ranges defined by the SMRR
```

```
WARNING: [*] SMM_Code_Chk_En is not enabled.
```

```
This can happen either because this feature is not supported by the CPU or because the BIOS forgot to enable it.
```

```
Please consult the Intel SDM to determine whether or not your CPU supports SMM_Code_Chk_En.
```

Врезка 1 – Проверка модулем `smm_code_chk` CHIPSEC'a

Вывод

В данной лабораторной работе был изучен подход к реверс-инжинирингу модулей UEFI. Стало очевидно, к чему стремились разработчики стандарта UEFI – к стандартизации и обобщению. Из процесса реверс-инжиниринга стало известно об общей структуре построения драйверов SMM, в том числе и о подмене (remap) стандартной RuntimeServices таблицы на SMM RuntimeServices таблицу для разделения ring 0 и ring -2.

Затем были исследованы обработчики прерываний в каждом из рассмотренных модулей: SmmHddSecurity, SdioSmm, SbRunSmm. Для того чтобы назвать потенциальный вектор атаки на их обработчики, необходимо было найти возможную уязвимость в каждом из обработчиков прерываний. В первом случае входе автоматической проверки с brick.py было выявлено возможное перекрытие коммуникационным буфером области SMRAM, однако при ручной проверке стало ясно, что это ложное срабатывание, так как в коммуникационный буфер ничего не записывалось и проверка размеров вовсе не понадобилась. Фаззинг в данном случае тоже не увенчается успехом, так как атакующему придётся угадать константу, чтобы блок кода выполнялся. В случае со вторым модулем только один из обработчиков потенциально уязвим к атаке confused deputy. Для предотвращения требуется использовать SmmIsBufferOutsideSmmValid, чтобы проверить, что коммуникационный буфер не указывает на SMRAM. Для второго обработчика этого модуля Satoshi Tanda совместно с ASUS выпустили исправление [3]. В последнем модуле уязвимость имеет место быть, так как разработчики использовали Runtime сервисную функцию, где её использовать нельзя, так как её можно перезаписать, а также не установлен SMM_Code_Chk_En, который запрещает процессору исполнять код снаружи SMRR (System Management Range Register), когда процессор находится в режиме SMM.

Таким образом, ASUS зачастую пренебрегает безопасностью своих прошивок, так как многие уязвимости, а соответственно и вектора атак известны уже на протяжении многих лет, но они до сих пор присутствуют во флагманских линейках их ноутбуков, таких как рассматриваемый в этой лабораторной продукт.

Список использованных источников

1. Assaf Carlsbad. brick.py - small tool designed to identify potentially vulnerable SMM modules. – 2021. – URL: <https://github.com/SentinelOne/brick> (дата обращения: 30.12.2021).
2. Assaf Carlsbad, Itai Liba. Automated vulnerability hunting in SMM using Brick. – 2021. – URL: <https://hardwear.io/netherlands-2021/presentation/automated-vulnerability-hunting-in-SMM.pdf> (дата обращения: 31.12.2021).
3. Satoshi Tanda. The report and the exploit of CVE-2021-26943, the kernel-to-SMM local privilege escalation vulnerability in ASUS UX360CA BIOS version 303. – 2021. – URL: <https://github.com/tandasat/smmexploit#resolution> (дата обращения: 01.01.2022).
4. Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alexander Matrosov, Mickey Shkatov. A New Class of Vulnerabilities in SMI Handlers.CanSecWest2015. – 2015. - https://edk2-docs.gitbook.io/edk-ii-secure-code-review-guide/code_review_guidelines_for_boot_firmware/external_input#smm-communication (дата обращения: 02.01.2022)
5. Dmytro Oleksiuk. Exploiting SMM callout vulnerabilities in Lenovo firmware. – 2016.- <http://blog.cr4.sh/2016/02/exploiting-smm-callout-vulnerabilities.html> (дата обращения: 03.01.2022)