1. Санкт-Петербургский Политехнический Университет Петра Великого
2. **Институт кибербезопасности и защиты информации**

**ЛАБОРАТОРНАЯ РАБОТА № 2**

1. «Реверс-инжиниринг SMM модулей UEFI BIOS»
2. по дисциплине «Принципы построения, проектирования и эксплуатации автоматизированных информационных систем»
3. Выполнил
4. студент гр. 4851003/70801 Гасанов Э.А.

<*подпись*>

1. Преподаватель
2. старший преподаватель Чернов А.Ю.

<*подпись*>

1. Санкт-Петербург
2. 2021

**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 С422)
* 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 HOB-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 a priori file (рисунок 1).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 1 – Содержимое a 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.

Изображение выглядит как стол

Автоматически созданное описание

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 вызов протоколов происходит согласно стеку DepEx из рисунка 2.

Таблица 1 – Точка входа в драйвер

|  |  |
| --- | --- |
| Дизассемблер | TianoCore - PiSmmIpl.c |
| 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 | 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); |
| (\*(void (\_\_fastcall \*\*)(EFI\_TPL))mSmmAccess)(mSmmAccess); | //  // Open all SMRAM ranges  //  Status = mSmmAccess->Open (mSmmAccess);  // SMRAM window is now open. |
| Пытаемся найти самое больше “окно” в процессоре для SMRAM  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;  --v3;  }  while ( v3 );  Таким образом находится место в памяти процессора под SMRAM, которое будет использоваться SMM IPL и SMM Core | //  // 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) {  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];  }  }  }  } |
| 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)  v12 = sub\_15EC(qword\_4498, (char \*)Buffer + 32 \* qword\_4140 - 32, aSmmc);  // qword\_4140= gSmmCorePrivate->SmramRangeCount  Продолжение в таблице 4 | CpuArch = NULL;  Status = gBS->LocateProtocol (&gEfiCpuArchProtocolGuid, NULL, (VOID \*\*)&CpuArch);  //  // 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  ); |

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 |
| \_\_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 | EFI\_SMRAM\_DESCRIPTOR \* GetFullSmramRanges  (  OUT UINTN \*FullSmramRangeCount  )  {  //  // Get SMM Configuration Protocol if it is present.  //  SmmConfiguration = NULL;  Status = gBS->LocateProtocol (&gEfiSmmConfigurationProtocolGuid, NULL, (VOID \*\*)&SmmConfiguration); |
| 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 | //  // Get SMRAM reserved region count.  //  SmramReservedCount = 0;  if (SmmConfiguration != NULL)  {    while  (SmmConfiguration->SmramReservedRegions[SmramReservedCount].SmramReservedSize != 0) {  SmramReservedCount++;  }  } |
| У меня декомпилировалось сразу в “иначе”  То есть SMM Configuration Protocol даёт зарезервированную точку “вхождения” в SMRAM  if ( SmramReservedCount )  {  v41 = 16 \* (v2 + 2 \* SmramReservedCount);    Buffer = (void \*)sub\_221C(4i64); // sub\_221C имеет AllocatePool  v5 = 0i64;  v6 = (char \*)Buffer;  v7 = SmramReservedCount;  do  {  v8 = (char \*)(v5 + \*SmmConfiguration);  if ( v6 != v8 )  sub\_3700( // содержит qmemcpy == аналог CopyMem  v6,  v8,  16i64);  v5 += 16i64;  v6 += 16;  --v7;  }  while ( v7 );  Тогда v2 - SmramRangeCount  Тогда v41 - Size  Buffer – SmramReservedRanges    v5= SmramRangeCount + 2 \* SmramReservedCount  Size = 32 \* v5;  v10 = sub\_221C(4i64, 32 \* v5); // sub\_221C имеет AllocatePool  v11 = 0i64;  v43 = 0i64;  v12 = v10;  v13 = sub\_221C(4i64, Size); // sub\_221C имеет AllocatePool  v39 = v13;  Тогда v39=v13= SmramRanges  do  {  v14 = 0i64;  v15 = 0;  v38 = 0i64;  if ( !SmramRangeCount )  break; // - Skip zero size entry.    опускаем подробности проверок в цикле  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 | 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);  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++)  {  //  // Skip zero size entry.  //  If (SmramReservedRanges[Index2].SmramReservedSize!= 0)  {  if (SmmIsSmramOverlap (  &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); |
| 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 );  }  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; | //  // 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++) {  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; |

Таблица 3 – Запуск образа ядра SMM

|  |  |
| --- | --- |
| Дизассемблер | PiSmmIpl.c |
| \_\_int64 \_\_fastcall sub\_15EC  (\_\_int64 a1, \_QWORD \*a2, \_\_int64 a3)  {  Каждый том прошивки драйвера должен создавать экземпляр 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;  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 | // 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);  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); |
| \_\_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); | //  // 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);  } |

Таблица 4 – Установка протоколов

|  |  |
| --- | --- |
| Дизассемблер | TianoCore; PiSmmIpl.c |
| gBS\_43E0->InstallMultipleProtocolInterfaces(  &Handle,  &EFI\_SMM\_BASE2\_PROTOCOL\_GUID\_4080,  off\_4110,  &EFI\_SMM\_COMMUNICATION\_PROTOCOL\_GUID\_4090,  &off\_4120,  0i64  );  off\_4110:    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;  }  off\_4120:    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;  } | //  // 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;  }  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 = mSmmControl2->Trigger (mSmmControl2,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;  }  } |
| 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  {  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 ); | 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.  //  // 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,  mSmmIplEvents[Index].Guid,  &mSmmIplEvents[Index].Event  );  ASSERT\_EFI\_ERROR (Status);  }  } |

Отдельно рассмотрим протокол READY\_TO\_LOCK\_PROTOCOL (таблица 5).

Таблица 5 – Использование READY\_TO\_LOCK\_PROTOCOL

|  |  |
| --- | --- |
| Дизассемблер | TianoCore; PiSmmIpl.c |
| 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] );  sub\_13B8(a1, (\_\_int64 \*)&EFI\_DXE\_SMM\_READY\_TO\_LOCK\_PROTOCOL\_GUID\_40D0);  byte\_43C4 = 1; | 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);  }  }  //  // 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; |

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

Таблица 6 – Нахождение точки входа в SMM

|  |  |
| --- | --- |
| Дизассемблер | TianoCore; PiSmmIpl.c |
| EFI\_SMM\_RESERVED\_SMRAM\_REGION - 64 бита весит, поэтому +64 приведет на следующее поле структуры  \_\_int64 sub\_21F4()  {  \_\_int64 result; // rax  if ( qword\_43F0 )  return (\*(\_\_int64 (\_\_fastcall \*\*)(\_QWORD, \_\_int64 \*))(qword\_4400 + 64))(0i64, &qword\_43F0);  }  В силу того, что у нас нет вызова функции LocateProtocole, второй параметр 0 | 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); |

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 256Кб). 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 очередной 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. Информируем, что протокол DxeSmmReadyToLock установлен и устанавливаем флаг, что SMM закрыт.

**Модуль SmmHddSecurity**

Находим точку входа в SmmHddSecurity драйвер (рисунок 3).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 3 – Экспортируемая точка входа

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

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 4 – Функция sub\_330

Произведем переименования фактических параметров функции sub\_330 (рисунок 5)

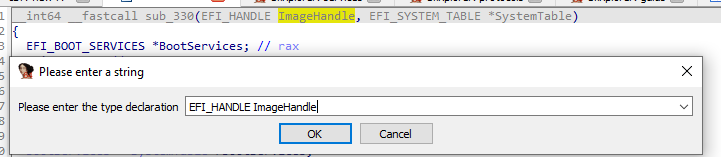


Рисунок 5 - Переименования аргументов

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

Изображение выглядит как текст

Автоматически созданное описание

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).

Изображение выглядит как текст

Автоматически созданное описание

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

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 8 – Установлен верный тип данных

Аналогично поступаем с остальными параметрами (рисунки 8 и 9)

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 8 – Перед изменением типа данных на верный

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 9 – Верный тип данных

Проверено в EDK2, что последний параметр GetCapabilities является типом данных EFI\_SMRAM\_DESCRIPTION (рисунки 10 и 11)

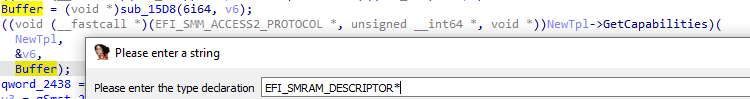


Рисунок 10 – GetCapabilities и его последний параметр

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 11 – Верное имя

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

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 12 – Верный тип данных

Итоговый результат (рисунок 13)

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 13 – Итоговая функция InitAmi

Выставим в точке входа переменную статуса (рисунок 14)

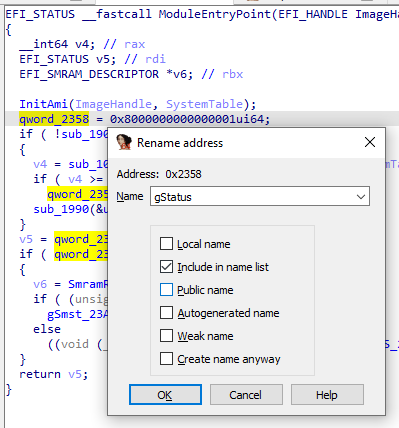


Рисунок 14 – Верное по контексту имя переменной

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

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 15 - Перехват исключений

Заходим в функцию sub\_107C (рисунок 16)

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 16 – Новое имя переменной

Изменяем тип данных gSMST\_ на \_EFI\_SMM\_SYSTEM\_TABLE2\* (рисунок 17)

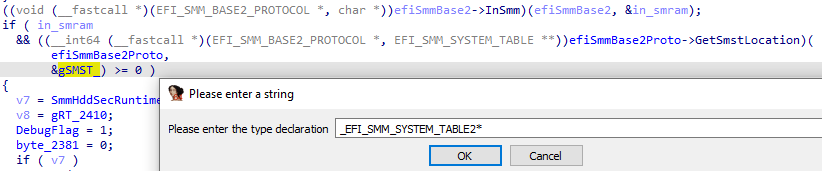


Рисунок 17 – Новый тип данных

Далее рассмотрим функции из рисунка 18.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 18 – Некоторые функции

Происходит подмена RUNTIME\_SERVICES таблицы на SMM’ную версию, чтобы разделить режимы RING 0 и SMM (рисунок 19).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 19 – Подмена таблицы на SMM’ные

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

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 20 – Функция HddSecInitSmmStatusCodeProtocol

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 21 – Функция HddSecInitSmmSomeProtocol

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

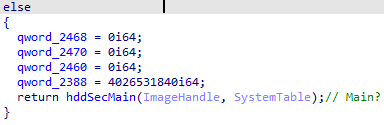


Рисунок 22 – Вход в hddSecMain

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

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 23 – Определение hddSecMain

В функции Handlers находятся обработчики прерываний (SWSMI) (рисунок 24) Изображение выглядит как текст

Автоматически созданное описание

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).

Изображение выглядит как текст

Автоматически созданное описание

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)

Изображение выглядит как текст

Автоматически созданное описание

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.

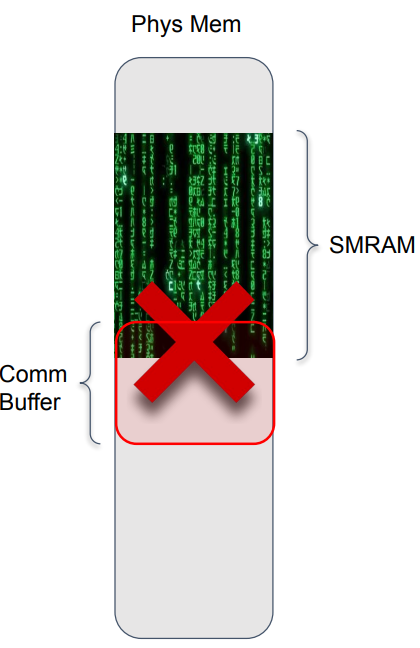


Рисунок 27 – Перекрытие SMRAM CommBuffer’ом [2]

В связи с вышеуказанными доводами опишем вектор атаки. Но сделаем оговорку, что атакующий уже имеет привилегии уровня ядра (ring0). Атакующий должен поместить СommBuffer прямо под SMRAM (SMRAM\_BASE-1) c 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).

Изображение выглядит как текст

Автоматически созданное описание

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).

Изображение выглядит как текст

Автоматически созданное описание

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).

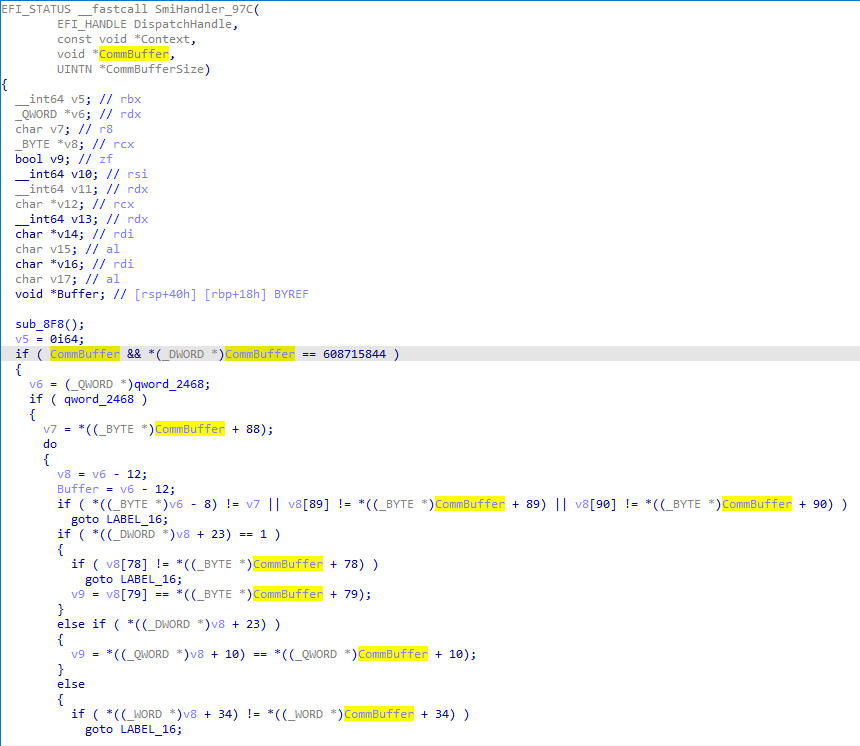


Рисунок 30 – SmiHandler\_97C

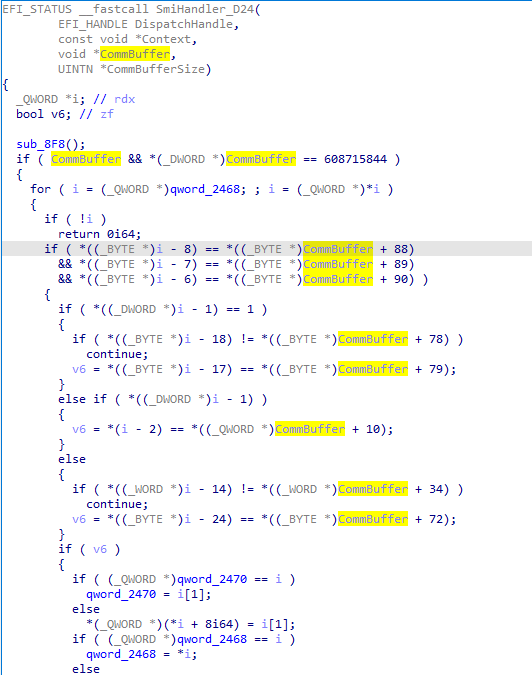


Рисунок 31 - SmiHandler\_D24

Изображение выглядит как текст

Автоматически созданное описание

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).

Изображение выглядит как текст

Автоматически созданное описание

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)

Изображение выглядит как текст

Автоматически созданное описание

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).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 35 – Точка входа в SdioSmm

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

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 36 – Переименование параметра в LocateProtocol

На рисунке 37 происходит переименование переменной статуса.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 37 – Новое имя переменной статуса

Таким образом, получаем следующий ModuleEntryPoint (рисунок 38).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 38 – Новый ModuleEntryPoint

Аналогично заменяем фактические параметры функции на нужные (рисунок 39)

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 39 – Установка имён и типов данных фактических параметров функции

Далее изменяем тип данных (рисунок 40).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 40 – Изменение типа данных согласно GUID

Изменим qword\_2010 на gSMST и его тип данных на \_EFI\_SMM\_SYSTEM\_TABLE2\* (рисунок 41).

Изображение выглядит как текст

Автоматически созданное описание

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.

Изображение выглядит как текст

Автоматически созданное описание

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.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 43 – функция sub\_A0C, ставшая SdioSmmInintSmmStatusCodeProtocol

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 44 – функция sub\_A50, ставшая SdioSmmInintSomeProtocols

Таким образом, SdioSmmEntryPoint принимает вид (рисунок 45)

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 45 – Измененная точка входа

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

Изображение выглядит как текст

Автоматически созданное описание

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).

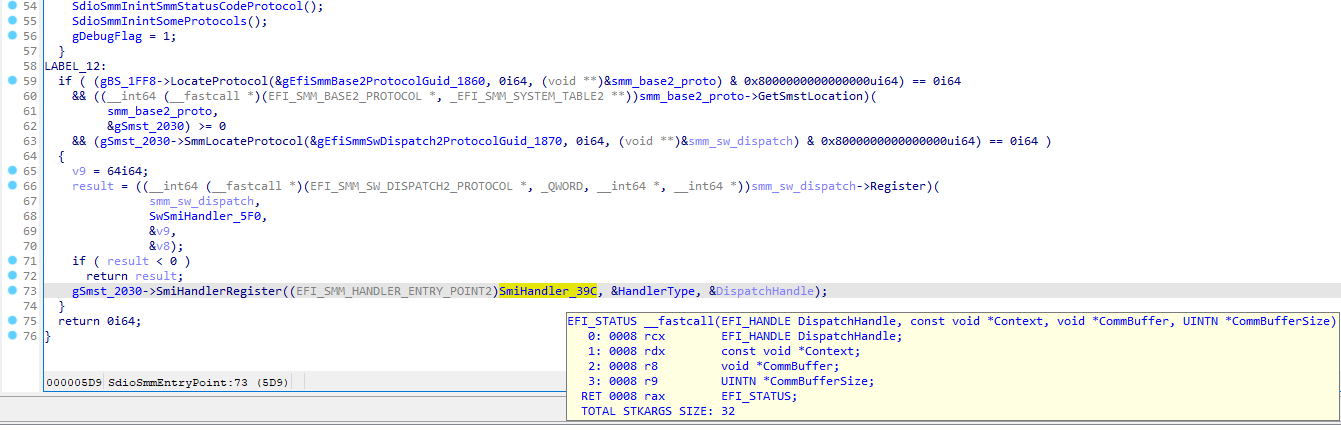


Рисунок 47 – Произведенные изменения

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.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 48 – Обработчик SwSmiHandler\_5F0

Но SwSmiHandler\_5F0 не имеет векторов атаки, так как было реализовано исправление (рисунок 49).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 49– Исправления уязвимости [3]

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

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 50 - SmiHandler\_39C

Снова проведем автоматический анализ (рисунок 51).

Изображение выглядит как текст

Автоматически созданное описание

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 AMI 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; |

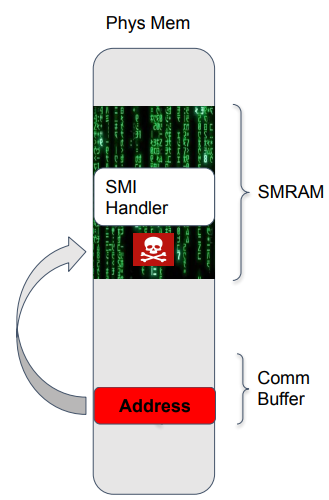


Рисунок 52 – Иллюстрация уязвимости [2]

## **Модуль SbRunSmm**

Находим экспортируемую точку входа (рисунок 53).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 53 – Первоначальный вид точки входа

Проводим переименования фактических параметров функции sub\_310 (рисунок 54).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 54 – Фактические аргументы

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

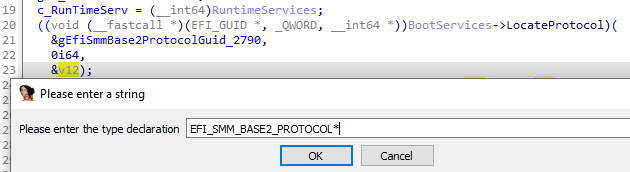


Рисунок 55 – Переопределение типа данных

После этого задаём имя согласно контексту (рисунок 56).

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 56 – Задание имени smmBase2Proto

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

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 57 – Первоначальный вид до изменений

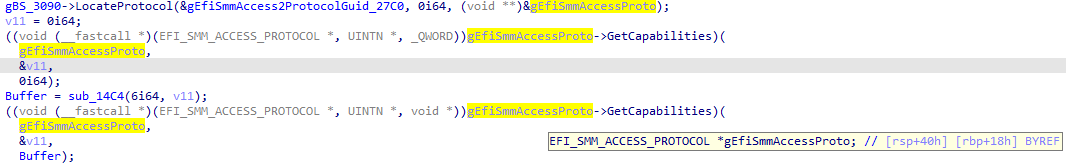


Рисунок 58 – Выставлены тип данных EFI\_SMM\_ACCESS\_PROTOCOL и имя из контекста

Обратившись к исходникам AMI\_BIOS стало ясно, что в функции GetCapabilities для последнего параметра используется тип данных EFI\_SMRAM\_DESCRIPTOR\* (рисунок 59).

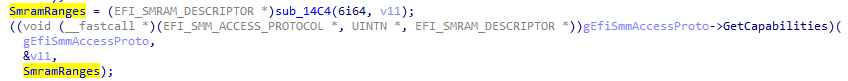


Рисунок 59– Верный тип данных у SmramRanges

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

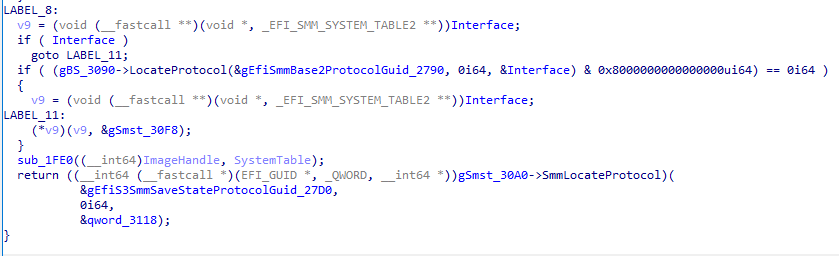


Рисунок 60 – До применения изменений

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 61 – Примененные изменения

На рисунках 62 и 63 показан финальный вид функции sub\_310, переименованной в Init.

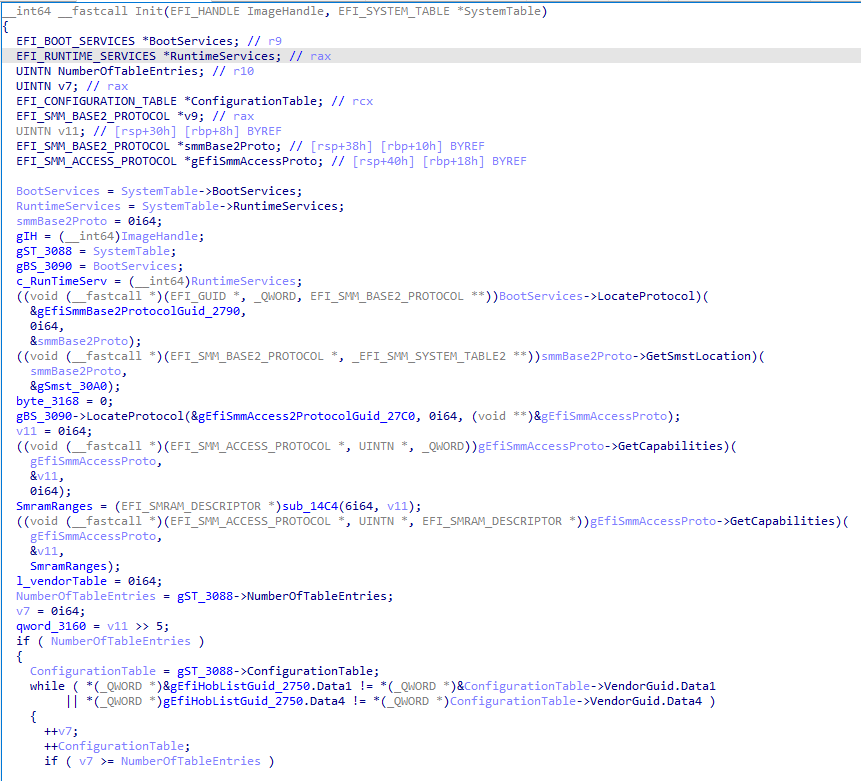


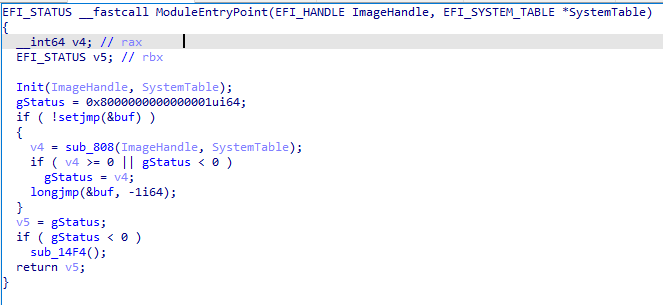
Рисунок 62 – Функция Init

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 63 – Функция Init (продолжение)

На рисунке 64 представлена измененная точка входа.



Затем процедура sub\_808 переименовывается в SbRunSmmEntryPoint и на рисунках 65 и 66 показана эта функция до внесения изменений.

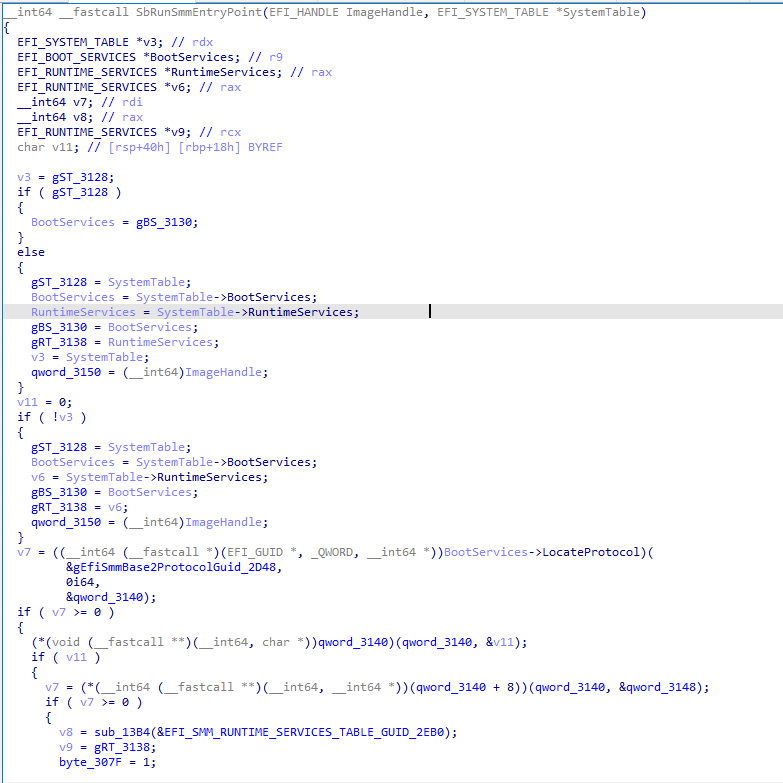


Рисунок 65– До внесений изменений в функцию SbRunSmmEntryPoint

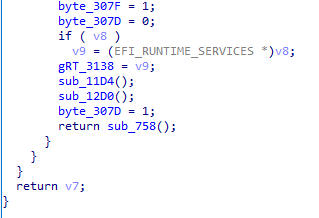


Рисунок 66 – До внесений изменений в функцию SbRunSmmEntryPoint (продолжение)

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)

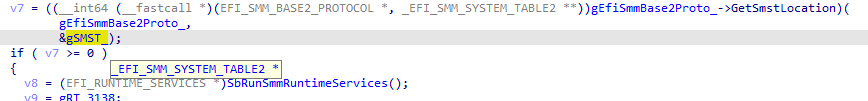


Рисунок 67 – Установление верного типа данных

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

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 68 – Функция SbRunSmmRuntimeServices

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

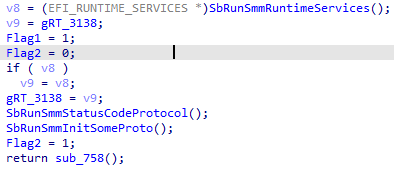


Рисунок 69– Оставшиеся функции

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 70 – Инициализация протоколов EfiSmmStatusCodeProtocol и EfiStatusCodeRuntimeProtocol

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 71 – Инициализация протоколов AmiSmmDebugServiceProtocol и AmiDebugServiceProtocol

Функция sub\_758 содержит в себе обработчик прерывания (рисунок 72). Далее sub\_758 переименовывается в Handler.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 72 – Функция sub\_758

На рисунке 73 представлен обработчик.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 73 – Callback функции добавлены аргументы обработчика

Рассмотрим содержимое SwSmiHandler\_4B4 (рисунок 74)

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 74 – Runtime сервис, доступный в обработчике

Обратимся к результатам проверки efiXplorer (рисунок 75)

Изображение выглядит как текст

Автоматически созданное описание

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,SetVatiable) 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’а

# **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 выполнился. 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 напряжении 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/Sentinel-One/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)