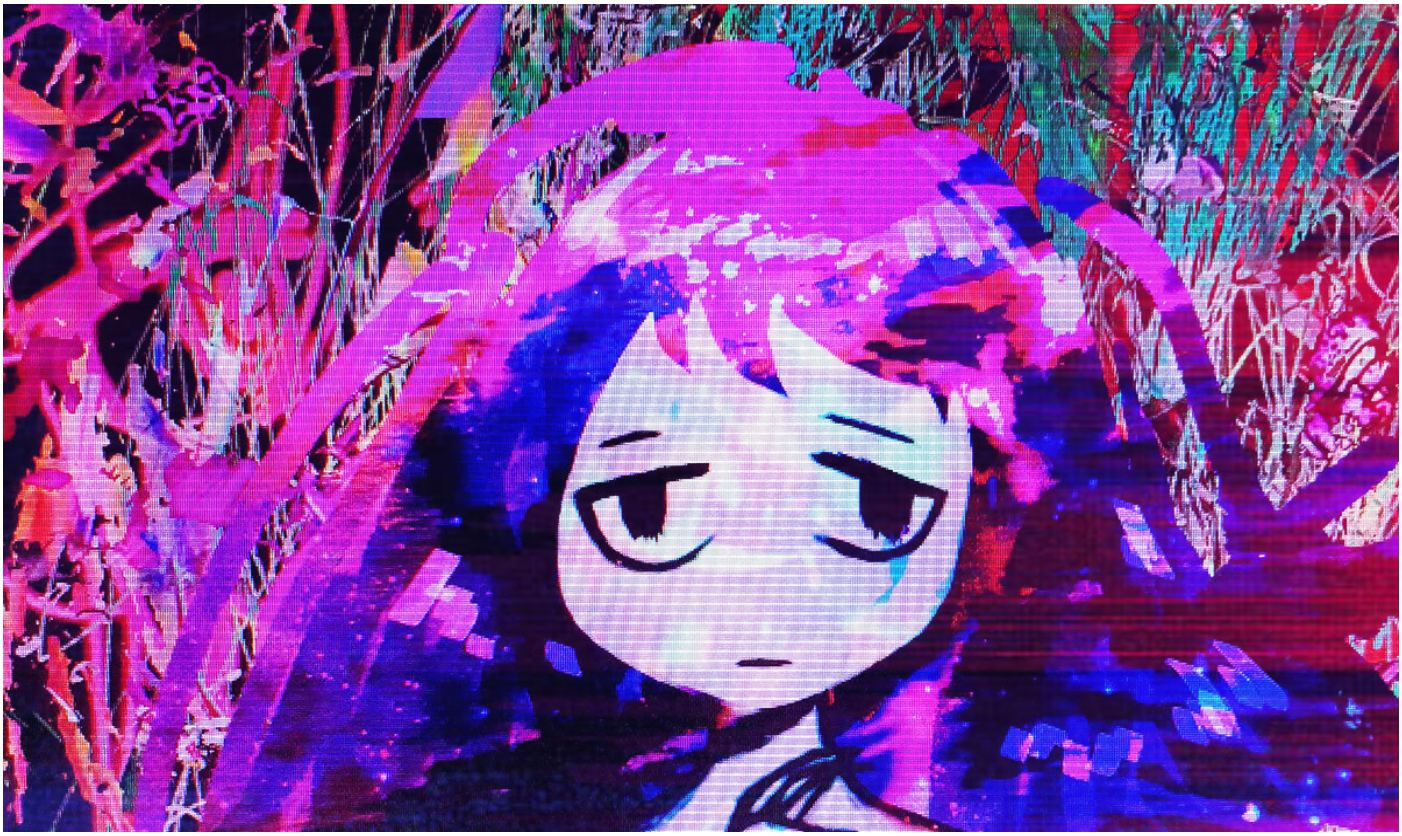

Drowning In The Kernel #1: Driver Signature Enforcement!

SKVLLZ.

Хм.

Допустим, мы должны убедиться, что у какого-либо драйвера есть собственная цифровая подпись. Так и еще чтоб она была валидной! Логично, что для этого нужен какой-либо механизм, совершающий подобные проверки. Ну, что же, да, таков есть в Windows и ни для кого это не секрет.

Садитесь удобнее, сегодня мы попытаемся понять, где обитает **DSE** и почему он причиняет боль многим любителям говна в режиме ядра.

Что такое DSE?

DSE – аббревиатура от словосочетания **Driver Signature Enforcement**. На самом деле это весьма комплексный механизм **Code Integrity**, он существует для «защиты» от неподписанных компонентов ядра, следовательно предполагает, что каждый драйвер устройства (или иной другой системный файл, выгружаемый в адресное пространство ядра) должен иметь собственную цифровую подпись. В противном случае, система просто откажется грузить этот компонент.

Стоит на всякий случай сказать, что изменение системных файлов убивает подпись, поэтому, мои маленькие любители подрочить в ядре, это не прокатит и система скажет, что такой файл не загрузит! Так что да, эта технология существует по большому счету для защиты от вредоносов, работающих на уровне ядра. И, в принципе, он справляется вполне себе неплохо!

DSE включен по стандарту, но пользователям (только из под админа, конечно же) предоставляется возможность выключить на время проверку подписей (или включить *Test Signing Mode*, позволяющий загружать самоподписанные драйвера), например, через bcdedit. Правда, это чаще всего нужно только для разработчиков драйверов, когда они тестируют свои наработки.

С понятием **DSE** мы вроде разобрались, перейдем к теперь к его инициализации.

Инициализация Driver Signature Enforcement и Code Integrity

DSE является компонентом ядра, а значит, надо искать примерно на этом уровне или чуточку ниже.

DSE берет свое начало с фазы «ранней загрузки» системы – когда свое дело закончили загрузчики (в данном случае – winload.exe/winload.efi). Это дело выгружается с основными драйверами системы. Нужный нам покемон находится в CI.dll (CodeIntegrity). Если обратить внимание на то, что экспортирует данная DLL мы однозначно можем сказать, что нам нужна функция под именем **CiInitialize**:

0004	00044c40	000bc09f	CiCheckSignedFile
0005	00044d00	000bc0b1	CiFindPageHashesInCatalog
0006	00044d80	000bc0cb	CiFindPageHashesInSignedFile
0007	00044dc0	000bc0e8	CiFreePolicyInfo
0008	000604c0	000bc0f9	CiGetCertPublisherName
0009	00044b20	000bc110	CiGetPEInformation
000a	00043400	000bc123	CiInitialize
000b	00060880	000bc130	CiSetTrustedOriginClaimId
000c	000531a0	000bc14a	CiValidateFileObject
000d	00044bc0	000bc15f	CiVerifyHashInCatalog

Однако, давайте не спешить, мы зайдем немного с другого угла и чуточку уйдем от CI.dll.

Все же не менее важным для ОС, как ни странно, является ядро, а это значит, что упоминание **CiInitialize** должно быть и там.
В действительности, так и есть, более того, эта функция также вызывается во время инициализации системы. *Барабанная дробь*...

Давайте посмотрим на функцию из ntoskrnl.exe – **SepInitializeCodeIntegrity**.

```
__int64 SepInitializeCodeIntegrity()
{
    unsigned int CiOptions; // edi
    _LIST_ENTRY *p_BootDriverListHead; // rbx
    _LOADER_PARAMETER_EXTENSION *Ext; // rcx
    _LOADER_PARAMETER_CI_EXTENSION *CodeIntegrityData; // rdx

    CiOptions = 6;
    memset(&CodeIntegrityCallbacks, 0, 0xC4u);  // nt!SeCiCallbacks

    p_BootDriverListHead = 0i64;
    SeCiCallbacks = 0xD0;
    qword_140C1DAA8 = 0xA000008i64;             // nt!SeCiCallbacks + 0xc8

    if ( KeLoaderBlock )
    {
        Ext = *(KeLoaderBlock + 0xF0);
        if ( Ext )
        {
            CodeIntegrityData = Ext->CodeIntegrityData;
            if ( CodeIntegrityData )
                CiOptions = CodeIntegrityData->CodeIntegrityOptions;
        }
        if ( *(KeLoaderBlock + 0xD8) && (SepIsOptionPresent)() )
            SeCiDebugOptions |= 1u;
        if ( KeLoaderBlock )
            p_BootDriverListHead = (KeLoaderBlock + 0x30);
    }
    return CiInitialize(CiOptions, p_BootDriverListHead, &SeCiCallbacks, SeCiPrivateApis);
}
```

Именно в этой функции мы получаем информацию о существовании **CiInitialize** впервые, правда, уже через само ядро. Думаю, стоит разобрать поближе несколько моментов.

И начну я пожалуй с **SeCiPrivateApis**, так как они требуют меньше объяснений и о них можно сказать прямо сейчас. Из названия логично предположить, что **SeCiPrivateApis** содержит в себе список некоторых функций, с которыми работает модуль **Code Integrity**. Данный список содержит в себе оффсеты на функции: **PsQueryProcessSignatureMitigationPolicy**, **VslHvciInterface** (это оффсет на **VslCreateSecureAllocation**), **SepZwLockRegistryKey**, **PsQuerySectionSignatureInformation**, **SepSetRuntimeUpdatableSigningLevel**, **RtlValidProcessProtection**, **MmGetImageFileSignatureInformation** и **SepGetSystemSigningLevel**.

Данные функции не будут разбираться в этой статье, однако, я покажу несколько из них.

```
char __fastcall MmGetImageFileSignatureInformation(__int64 a1)
{
    __int64 v1; // rcx
    char v2; // bl
    __int64 v3; // rax
    _QWORD *v4; // rdi
    unsigned __int8 v6; // [rsp+30h] [rbp+8h] BYREF

    v1 = *(a1 + 40);
    v2 = 0;
    v6 = 0;
    if ( !v1 )
        return 0;
    v3 = MiLockSectionControlArea(v1, 0i64, &v6);
    v4 = v3;
    if ( !v3 )
        return 0;
    if ( (*(v3 + 56) & 3) != 0 )
    {
        ExReleaseSpinLockExclusiveFromDpcLevel(v3 + 72);
        __writecr8(v6);
    }
    else
    {
        ++*(v3 + 24);
        MiRemoveUnusedSegment(v3);
        ++v4[6];
        ExReleaseSpinLockExclusiveFromDpcLevel(v4 + 9);
        __writecr8(v6);
        v2 = *(*v4 + 15i64) >> 4;
        MiDereferenceControlAreaBySection(v4, 1i64);
    }
    return v2;
}
```

```
__int64 __fastcall PsQuerySectionSignatureInformation(_EPROCESS *a1, __int64 a2)
{
    void *SectionObject; // rcx
    __int64 v3; // rax
    _BYTE *v4; // r8

    if ( a1 != KeGetCurrentThread()->Process )
        return 3221225659i64;
    SectionObject = a1->SectionObject;
    if ( !SectionObject )
        return 3221225473i64;
    v3 = MiSectionControlArea(SectionObject, a2, a2);
    *v4 = *(*v3 + 15i64) >> 4;
    return 0i64;
}
```

Назначение **SeCiCallbacks** я разберу в следующем параграфе. Также, не выкидывайте из головы структуру **_LOADER_PARAMETER_CI_EXTENSIONS**, с ней есть одна забавная особенность.

Кстати, забыл показать call stack. Да, функция и в правду вызывается во время инициализации:

```
1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff9008`b3606918 fffff803`7e9a05dd     nt!SepInitializeCodeIntegrity
01 ffff9008`b3606920 fffff803`7ec64f7a     nt!SepInitializationPhase1+0x231
02 ffff9008`b36069f0 fffff803`7ec3f464     nt!SeInitSystem+0x1e
03 ffff9008`b3606a30 fffff803`7e98cfa3     nt!Phase1InitializationDiscard+0x940
04 ffff9008`b3606be0 fffff803`7e55c855     nt!Phase1Initialization+0x23
05 ffff9008`b3606c10 fffff803`7e605808     nt!PspSystemThreadStartup+0x55
06 ffff9008`b3606c60 00000000`00000000     nt!KiStartSystemThread+0x28
```

От ядра к модулю

Хорошо, мы поняли, что за инициализацию **DSE** в ядре отвечает функция **SepInitializeCodeIntegrity**, данная функция срабатывает во время инициализации самой системы и она вызывает нужный нам **CiInitialize**.
Самое время глянуть на **CiInitialize**! Иииииии...

```
__int64 __fastcall CiInitialize(
        int CiOptions,
        _LIST_ENTRY *p_BootDriverListHead,
        __int64 SeCiCallbacks,
        __int64 SeCiPrivateApis)
{
  wil_InitializeFeatureStaging();
  _security_init_cookie();
  if ( wil_details_featureChangeNotification )
  {
    RtlUnregisterFeatureConfigurationChangeNotification();
    wil_details_featureChangeNotification = 0i64;
  }
  return CipInitialize(CiOptions, p_BootDriverListHead, SeCiCallbacks, SeCiPrivateApis);
}
```

Да, это очередной враппер над еще одной функцией – **CipInitialize**. Интересно, что тут мы имеем упоминание *Microsoft WIL* (Windows Implementation Library). Это нас не должно особо волновать, данные вещи нам не нужны и не мешают, но, решил просто это отметить.

Хорошо, **CiInitialize** – это враппер. Значит, идем прямиком в **CipInitialize**! И, да, это она, именно она отвечает за инициализацию нашего **DSE**. Это весьма большая функция, поэтому придется разместить её на нескольких скринах. Надеюсь, вы не против.

```
__int64 __fastcall CipInitialize(
        unsigned int CiOptions,
        _LIST_ENTRY *p_BootDriverListHead,
        __int64 CiCallbacks,
        __int64 CiPrivateApis)
{
  __int128 *Off1; // rax
  __int128 v10; // xmm0
  __int128 Off2; // xmm1
  struct _LIST_ENTRY *i; // rdi
  int v13; // r8d
  int v14; // r9d
  int v15; // r8d
  int v16; // r9d
  int MonitorStatus; // edi
  int Status; // eax
  bool IsHvciSupported; // zf
  _QWORD *pPreOsDriverList; // rdx
  PVOID *v21; // rcx
  PVOID PreOsDriverList; // rax

  g_CiOptions = CiOptions;
  g_CiSystemProcess = PsGetCurrentProcess();
  if ( *CiCallbacks != 0xE8 )
    return 0xC00000F1i64;
  if ( *(CiCallbacks + 224) != 0xA000008i64 )
    return 0xC0000059i64;
  Off1 = *(CiPrivateApis + 8);
  if ( Off1 )
  {
    v10 = *Off1;
    g_HvciSupported = 1;
    g_CiVslHvciInterface = v10;
    xmmword_1C0038430 = Off1[1];
    xmmword_1C0038440 = Off1[2];
    xmmword_1C0038450 = Off1[3];
    xmmword_1C0038460 = Off1[4];
    xmmword_1C0038470 = Off1[5];
    xmmword_1C0038480 = Off1[6];
  }
  g_CiPrivateNtosApis = *CiPrivateApis;
  xmmword_1C00384F0 = *(CiPrivateApis + 16);
  xmmword_1C0038500 = *(CiPrivateApis + 32);
  Off2 = *(CiPrivateApis + 48);
  qword_1C00384B8 = &g_BootDriverList;
  g_BootDriverList = &g_BootDriverList;
  xmmword_1C0038510 = Off2;
  if ( p_BootDriverListHead )
  {
    for ( i = p_BootDriverListHead->Flink;
```

```
  xmmword_1C0038510 = Off2;
  if ( p_BootDriverListHead )
  {
    for ( i = p_BootDriverListHead->Flink;
          i != p_BootDriverListHead && CipInitializeBootDriverState(&i[1]) >= 0;
          i = i->Flink )
    {
      ;
    }
  }
  g_CiInitLock = 0i64;
  MinCrypK_DisableEcdsa();
  HashpSelfTest();
  if ( !MincrypK_TestPKCS1SignVerify(32772, &unk_1C00C67E0, v13, v14, &unk_1C00C66E0, 128, &unk_1C00C6860, 128)
    || !MincrypK_TestPKCS1SignVerify(32780, &unk_1C00C6760, v15, v16, &unk_1C00C6B60, 256, &unk_1C00C6C60, 256) )
  {
    __fastfail(0x14u);
  }
  SymCryptParallelSha256Selftest();
  MonitorStatus = CiRegisterSiloMonitor();
  if ( MonitorStatus >= 0 )
  {
    KeQueryPerformanceCounter(&g_CiPerfFrequency);
    g_CipPerfLock = 0i64;
    g_CipPolicyStatusLock = 0i64;
    g_CiWimListLock = 0i64;
    CipHvciTreeLock = 0i64;
    CipHvciTree = 0i64;
    Status = XciInitialize(CiOptions, p_BootDriverListHead, CiCallbacks, CiPrivateApis);
    MonitorStatus = Status;
    if ( Status == 0xC00000BB )
    {
      MonitorStatus = 0;
      goto LABEL_19;
    }
    if ( Status >= 0 )
    {
      g_CiDeveloperMode |= 0x20000u;
LABEL_19:
      IsHvciSupported = g_HvciSupported == 0;
      *(CiCallbacks + 0x20) = CiValidateImageHeader;
      *(CiCallbacks + 0x28) = CiValidateImageData;
      *(CiCallbacks + 0x18) = CiQueryInformation;
      *(CiCallbacks + 0) = CiSetFileCache;
      *(CiCallbacks + 0x10) = CiGetFileCache;
      *(CiCallbacks + 0x30) = CiHashMemory;
      *(CiCallbacks + 0x38) = KappxIsPackageFile;
      *(CiCallbacks + 0x40) = CiCompareSigningLevels;
      *(CiCallbacks + 0x48) = CiValidateFileAsImageType;
      *(CiCallbacks + 0x50) = CiRegisterSigningInformation;
      *(CiCallbacks + 0x58) = CiUnregisterSigningInformation;
      *(CiCallbacks + 0x60) = CiInitializePolicy;
      *(CiCallbacks + 0x88) = CipQueryPolicyInformation;
      *(CiCallbacks + 0x90) = CiValidateDynamicCodePages;
      *(CiCallbacks + 0x98) = CiQuerySecurityPolicy;
      *(CiCallbacks + 0xA0) = CiRevalidateImage;
      *(CiCallbacks + 0xA8) = &CiSetInformation;
      *(CiCallbacks + 0xB0) = CiSetInformationProcess;
      *(CiCallbacks + 0xB8) = CiGetBuildExpiryTime;
      *(CiCallbacks + 0xC0) = &CiCheckProcessDebugAccessPolicy;
      *(CiCallbacks + 0xC8) = CiGetCodeIntegrityOriginClaimForFileObject;
      *(CiCallbacks + 0xD0) = CiDeleteCodeIntegrityOriginClaimMembers;
      *(CiCallbacks + 0xD8) = CiDeleteCodeIntegrityOriginClaimForFileObject;
      if ( !IsHvciSupported )
      {
        *(CiCallbacks + 0x78) = CiGetStrongImageReference;
        *(CiCallbacks + 0x68) = CiReleaseContext;
        *(CiCallbacks + 0x80) = CiHvciSetImageBaseAddress;
      }
      PESetPhase1Initialization(p_BootDriverListHead);
      if ( MonitorStatus >= 0 )
        return MonitorStatus;
    }
  }
  while ( 1 )
  {
    PreOsDriverList = g_BootDriverList;
    if ( g_BootDriverList == &g_BootDriverList )
      break;
    pPreOsDriverList = *g_BootDriverList;
    if ( *(*g_BootDriverList + 8i64) != g_BootDriverList || (v21 = *(g_BootDriverList + 1), *v21 != g_BootDriverList) )
      __fastfail(3u);
    *v21 = pPreOsDriverList;
    pPreOsDriverList[1] = v21;
    ExFreePoolWithTag(PreOsDriverList, 0);
  }
  return MonitorStatus;
}
```

Отлично, мы нашли, где инициализируется **DSE**! Давайте разберем наиболее примечательные детали.

```
Off1 = *(CiPrivateApis + 8);
if ( Off1 )
{
  pVslHvciInterface = *Off1;
  g_HvciSupported = 1;
  g_CiVslHvciInterface = pVslHvciInterface;
  xmmword_1C0038430 = Off1[1];
  xmmword_1C0038440 = Off1[2];
  xmmword_1C0038450 = Off1[3];
  xmmword_1C0038460 = Off1[4];
  xmmword_1C0038470 = Off1[5];
  xmmword_1C0038480 = Off1[6];
}
```

Данный отрывок проверяет, существует ли оффсет на **VslHvciInterface**. Если да – заполняет таблицу **g_CiVslHvciInterface**. Я так и не смог понять, точно ли это то, что должно быть в данной таблице (да-да, называйте меня дауном-долбоебом-уебком-etc), поэтому, не упомяну ничего здесь во избежание ошибок. Но, предполагаю, что загружает оно какую-то часть функций отсюда:

g_CiVslHvciInterface position	Function	SSCN
g_CiVslHvciInterface	*VslCreateSecureAllocation*	*0x13*
g_CiVslHvciInterface + 0x08	*VslFillSecureAllocation*	*0x14*
g_CiVslHvciInterface + 0x10	*VslMakeCodeCatalog*	*0x15*
g_CiVslHvciInterface + 0x18	*VslCreateSecureImageSection*	*0x16*
g_CiVslHvciInterface + 0x20	*VslValidateSecureImagePages*	*0xC1*
g_CiVslHvciInterface + 0x28	*VslFinalizeSecureImageHash*	*0x17*
g_CiVslHvciInterface + 0x30	*VslFinishSecureImageValidation*	*0x18*
g_CiVslHvciInterface + 0x38	*VslPrepareSecureImageRelocations*	*0x19*
g_CiVslHvciInterface + 0x40	*VslRelocateImage*	*0x1A*
g_CiVslHvciInterface + 0x48	*VslCloseSecureHandle*	*0x1B*
g_CiVslHvciInterface + 0x50	*VslGetNestedPageProtectionFlags*	*0xE7*
g_CiVslHvciInterface + 0x58	*VslValidateDynamicCodePages*	*0x1C*
g_CiVslHvciInterface + 0x60	*VslTransferSecureImageVersionResource*	*0x1D*

Table 1: Functions referenced by *g_CiVslHvciInterface*

Скорее всего оно загружает все, начиная с **VslCreateSecureAllocation** (присутствует в **CiPrivateApis**) + 0x10.

Здесь при вызове **HashpSelfTest** CI.dll тестирует собственную подпись, что очевидно. В условии тестируются собственные «Root Table».

```
MinCrypK_DisableEcdsa();
HashpSelfTest();
if ( !MincrypK_TestPKCS1SignVerify(32772, &unk_1C00C67E0, v13, v14, &unk_1C00C66E0, 128, &unk_1C00C6860, 128)
  || !MincrypK_TestPKCS1SignVerify(32780, &unk_1C00C6760, v15, v16, &unk_1C00C6B60, 256, &unk_1C00C6C60, 256) )
{
  __fastfail(0x14u);
}
SymCryptParallelSha256Selftest();
```

```
mov     ecx, 80h
lea     rax, [CI!RootTable+0x4d0 (fffff8057a8c5860)]
mov     dword ptr [rsp+38h], ecx
lea     rdx, [CI!RootTable+0x450 (fffff8057a8c57e0)]
mov     qword ptr [rsp+30h], rax
lea     rax, [CI!RootTable+0x350 (fffff8057a8c56e0)]
mov     dword ptr [rsp+28h], ecx
mov     ecx, 8004h
mov     qword ptr [rsp+20h], rax
call    CI!MincrypK_TestPKCS1SignVerify (fffff8057a867b54)
test    eax, eax
je      CI!CipInitialize+0x366 (fffff8057a84407a)
mov     ecx, 100h
lea     rax, [CI!RootTable+0x8d0 (fffff8057a8c5c60)]
lea     rax, [CI!RootTable+0x8d0 (fffff8057a8c5c60)]
mov     dword ptr [rsp+38h], ecx
lea     rdx, [CI!RootTable+0x3d0 (fffff8057a8c5760)]
mov     qword ptr [rsp+30h], rax
lea     rax, [CI!RootTable+0x7d0 (fffff8057a8c5b60)]
mov     dword ptr [rsp+28h], ecx
mov     ecx, 800Ch
mov     qword ptr [rsp+20h], rax
call    CI!MincrypK_TestPKCS1SignVerify (fffff8057a867b54)
test    eax, eax
```

А вот здесь у нас есть немножечко примеси Xbox (**XciInitialize** импортируется из ext-ms-win-ci-xbox-l1-1-0), живите с этим.

```
Status = XciInitialize(CiOptions, p_BootDriverListHead, CiCallbacks, CiPrivateApis);
MonitorStatus = Status;
if ( Status == 0xC00000BB )
{
```

А вот и самое главное – таблица коллбеков! Именно функции этой таблицы и отвечают за проверку подписей, а значит, мы технически нашли, то, что нам нужно!

```
    if ( Status == 0xC00000BB )
    {
      MonitorStatus = 0;
      goto LABEL_19;
    }
    if ( Status >= 0 )
    {
      g_CiDeveloperMode |= 0x20000u;
LABEL_19:
      IsHvciSupported = g_HvciSupported == 0;
      *(CiCallbacks + 0x20) = CiValidateImageHeader;
      *(CiCallbacks + 0x28) = CiValidateImageData;
      *(CiCallbacks + 0x18) = CiQueryInformation;
      *(CiCallbacks + 8) = CiSetFileCache;
      *(CiCallbacks + 0x10) = CiGetFileCache;
      *(CiCallbacks + 0x30) = CiHashMemory;
      *(CiCallbacks + 0x38) = KappxIsPackageFile;
      *(CiCallbacks + 0x40) = CiCompareSigningLevels;
      *(CiCallbacks + 0x48) = CiValidateFileAsImageType;
      *(CiCallbacks + 0x50) = CiRegisterSigningInformation;
      *(CiCallbacks + 0x58) = CiUnregisterSigningInformation;
      *(CiCallbacks + 0x60) = CiInitializePolicy;
      *(CiCallbacks + 0x88) = CipQueryPolicyInformation;
      *(CiCallbacks + 0x90) = CiValidateDynamicCodePages;
      *(CiCallbacks + 0x98) = CiQuerySecurityPolicy;
      *(CiCallbacks + 0xA0) = CiRevalidateImage;
      *(CiCallbacks + 0xA8) = &CiSetInformation;
      *(CiCallbacks + 0xB0) = CiSetInformationProcess;
      *(CiCallbacks + 0xB8) = CiGetBuildExpiryTime;
      *(CiCallbacks + 0xC0) = &CiCheckProcessDebugAccessPolicy;
      *(CiCallbacks + 0xC8) = CiGetCodeIntegrityOriginClaimForFileObject;
      *(CiCallbacks + 0xD0) = CiDeleteCodeIntegrityOriginClaimMembers;
      *(CiCallbacks + 0xD8) = CiDeleteCodeIntegrityOriginClaimForFileObject;
      if ( !IsHvciSupported )
      {
        *(CiCallbacks + 0x78) = CiGetStrongImageReference;
        *(CiCallbacks + 0x68) = CiReleaseContext;
        *(CiCallbacks + 0x80) = CiHvciSetImageBaseAddress;
      }
      PESetPhase1Initialization(p_BootDriverListHead);
      if ( MonitorStatus >= 0 )
        return MonitorStatus;
    }
  }
```

Самое главное, что делает **CipInitialize** – заполняет таблицу **CiCallbacks**, а **CiInitialize** передает заполненную таблицу ядру. Теперь, мы поняли, как инициализируется механизм проверки подписей. Осталось понять, каким образом проверяются загружаемые системные модули.

Мы любим только проверенные образы

Когда необходимая таблица с указателями на функции заполнилась – система может начинать совершать проверку загружаемых образов в пространство ядра.

Опытным путем было найдено несколько функций, отвечающих за это. Например, **SeValidateImageHeader**:

```
nt!SeValidateImageHeader:
fffff805`798f7564 488bc4          mov     rax, rsp
fffff805`798f7567 48895808        mov     qword ptr [rax+8], rbx
fffff805`798f756b 48897010        mov     qword ptr [rax+10h], rsi
fffff805`798f756f 57              push    rdi
fffff805`798f7570 4881eca0000000  sub     rsp, 0A0h
fffff805`798f7577 33f6            xor     esi, esi
fffff805`798f7579 488bda          mov     rbx, rdx
fffff805`798f757c 4839351d235300  cmp     qword ptr [ntkrnlmp!SeCiCallbacks+0x20 (fffff80579e298a0)], rsi
fffff805`798f7583 488bf9          mov     rdi, rcx
fffff805`798f7586 488970f0        mov     qword ptr [rax-10h], rsi
fffff805`798f758a 448bde          mov     r11d, esi
fffff805`798f758d 8970e8          mov     dword ptr [rax-18h], esi
fffff805`798f7590 0f8422b41300    je      ntkrnlmp!SeValidateImageHeader+0x13b454 (fffff80579a329b8)
fffff805`798f7596 8b9424f0000000  mov     edx, dword ptr [rsp+0F0h]
fffff805`798f759d f6c201          test    dl, 1
fffff805`798f75a0 0f85c2000000    jne     ntkrnlmp!SeValidateImageHeader+0x104 (fffff805798f7668)
fffff805`798f75a6 488b8c24f8000000 mov    rcx, qword ptr [rsp+0F8h]
fffff805`798f75ae 488d842490000000 lea    rax, [rsp+90h]
fffff805`798f75b6 4889442478      mov     qword ptr [rsp+78h], rax
fffff805`798f75bb 488b842418010000 mov    rax, qword ptr [rsp+118h]
fffff805`798f75c3 4c895c2470      mov     qword ptr [rsp+70h], r11
fffff805`798f75c8 4889442468      mov     qword ptr [rsp+68h], rax
fffff805`798f75cd 488b842410010000 mov    rax, qword ptr [rsp+110h]
fffff805`798f75d5 4889442460      mov     qword ptr [rsp+60h], rax
fffff805`798f75da 8a842408010000  mov     al, byte ptr [rsp+108h]
fffff805`798f75e1 88442458        mov     byte ptr [rsp+58h], al
fffff805`798f75e5 8a842400010000  mov     al, byte ptr [rsp+100h]
fffff805`798f75ec 88442450        mov     byte ptr [rsp+50h], al
fffff805`798f75f0 488b05a9225300  mov     rax, qword ptr [ntkrnlmp!SeCiCallbacks+0x20 (fffff80579e298a0)]
```

Здесь проводится проверка, существует ли **SeCiCallbacks** (или **CiCallbacks**, как я обозначал ранее) со смещением 0x20 (**CiValidateImageHeader**), если да – производим проверку, в противном случае возвращаем NTSTATUS равный 0xC0000428 (STATUS_INVALID_IMAGE_HASH). Call stack для **SeValidateImageHeader** выглядит так:

```
00 ffff8a07`dfc05f78 fffff805`798f7148     nt!SeValidateImageHeader
01 ffff8a07`dfc05f80 fffff805`798125a0     nt!MiValidateSectionCreate+0x438
02 ffff8a07`dfc06160 fffff805`7981a44e     nt!MiValidateSectionSigningPolicy+0xac
03 ffff8a07`dfc061c0 fffff805`79807a1b     nt!MiCreateNewSection+0x59a
04 ffff8a07`dfc06320 fffff805`79807064     nt!MiCreateImageOrDataSection+0x2db
05 ffff8a07`dfc06410 fffff805`7957be68     nt!MiCreateSection+0xf4
06 ffff8a07`dfc06590 fffff805`79954ea2     nt!MiCreateSystemSection+0xa4
07 ffff8a07`dfc06630 fffff805`7995277e     nt!MiCreateSectionForDriver+0x126
08 ffff8a07`dfc06710 fffff805`79951fd2     nt!MiObtainSectionForDriver+0xa6
09 ffff8a07`dfc06760 fffff805`79951e66     nt!MmLoadSystemImageEx+0x156
0a ffff8a07`dfc06900 fffff805`7993545c     nt!MmLoadSystemImage+0x26
```

Т.е, путь до проверки заголовка образа проходит от загрузки (**MmLoadSystemImage**) до самой валидации (**SeValidateImageHeader** и **CiValidateImageHeader**).

Это справедливо и для других функций, например для **SeValidateImageData** и **CiValidateImageData**!

WinDBG и DSE

Забавно, но стоит отметить, что *WinDBG*, если мы занимаемся отладкой ядра/бутменеджера/бутлоадера, сам отключает **DSE**, судя по всему это связано со структурой **_LOADER_PARAMETER_CI_EXTENSIONS**:

```
1: kd> dt nt!_LOADER_PARAMETER_CI_EXTENSION
   +0x000 CodeIntegrityOptions : Uint4B
   +0x004 UpgradeInProgress : Pos 0, 1 Bit
   +0x004 IsWinPE           : Pos 1, 1 Bit
   +0x004 CustomKernelSignersAllowed : Pos 2, 1 Bit
   +0x004 StateSeparationEnabled : Pos 3, 1 Bit
   +0x004 Reserved          : Pos 4, 28 Bits
   +0x008 WhqlEnforcementDate : _LARGE_INTEGER
   +0x010 RevocationListOffset : Uint4B
   +0x014 RevocationListSize : Uint4B
   +0x018 CodeIntegrityPolicyOffset : Uint4B
   +0x01c CodeIntegrityPolicySize : Uint4B
   +0x020 CodeIntegrityPolicyHashOffset : Uint4B
   +0x024 CodeIntegrityPolicyHashSize : Uint4B
   +0x028 CodeIntegrityPolicyOriginalHashOffset : Uint4B
   +0x02c CodeIntegrityPolicyOriginalHashSize : Uint4B
   +0x030 WeakCryptoPolicyLoadStatus : Int4B
   +0x034 WeakCryptoPolicyOffset : Uint4B
   +0x038 WeakCryptoPolicySize : Uint4B
   +0x03c SecureBootPolicyOffset : Uint4B
   +0x040 SecureBootPolicySize : Uint4B
   +0x044 Reserved2         : Uint4B
   +0x048 SerializedData    : [1] UChar
```

К сожалению, у меня она почему-то решила не загружаться (110% я криворукий долбоеб), поэтому вам придется поверить мне на слово, но обычно параметр **CodeIntegrityOptions** равен 6, однако, при подключенном *WinDBG* это поле скорее всего становится 0. Скорее всего это связано с тем, что настройки при отладке ядра изменяют это поле, а заодно вместе с ними и *WinDBG*.

И, да, это один из способов обхода механизма **DSE**.

Выводы

В данной статье я немного разобрал механизм инициализации **Driver Signature Enforcement**, а также немного рассказал, как он работает. Конечно, здесь непаханное поле экспериментов и можно еще что-то глянуть даже.

Как-то так, не скучайте, не болейте, кушайте хорошо. Водички побольше пейте, воооооот.