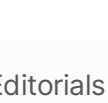
Tech Editorials

#CVE #Vulnerability #Windows #Kernel #Advisory

Streaming vulnerabilities from Windows Kernel - Proxying to Kernel - Part I



Angelboy 2024-08-23

English Version, 中文版本

Over the past few decades, vulnerabilities in the Windows Kernel have emerged frequently. The popular attack surface has gradually shifted from Win32k to CLFS (Common Log File System). Microsoft has continuously patched these vulnerabilities, making these targets increasingly secure. However, which component might become the next attack target? Last year, MSKSSRV (Microsoft Kernel Streaming Service) became a popular target for hackers. However, this driver is tiny and can be analyzed in just a few days. Does this mean there might not be new vulnerabilities?

This research will discuss an overlooked attack surface that allowed us to find more than ten vulnerabilities within two months. Additionally, we will delve into a proxy-based logical vulnerability type that allows us to bypass most validations, enabling us to successfully exploit Windows 11 in Pwn2Own Vancouver 2024.

(This research will be divided into several parts, each discussing different bug classes and vulnerabilities. This research was also presented at HITCON CMT 2024.)

Start from MSKSSRV

> For vulnerability research, looking at historical vulnerabilities is indispensable.

Initially, we aimed to challenge Windows 11 in Pwn2Own Vancouver 2024. Therefore, we began by reviewing past Pwn2Own events and recent in-the-wild Windows vulnerabilities, searching for potential attack surfaces. Historical trends show that Win32K, primarily responsible for handling GDI-related operations, has always been a popular target, with numerous vulnerabilities still emerging. Since 2018, CLFS (Common Log File System) has also gradually become a popular target. Both components are extremely complex, suggesting that there are likely still many vulnerabilities. However, becoming familiar with these components requires significant time, and many researchers are already examining them. Therefore, we did not choose to analyze them first.

Last year, after Synacktiv successfully exploited a vulnerability in MSKSSRV to compromise Windows 11 during Pwn2Own 2023, many researchers began to focus on this component. Shortly thereafter, a second vulnerability ,CVE-2023-36802, was discovered. At this time, chompie also published an excellent blog post detailing this vulnerability and its exploitation techniques. Given that this component is very small, with a file size of approximately 72 KB, it might only take a few days of careful examination to fully understand it. Therefore, we chose MSKSSRV for historical vulnerability analysis, with the hopeful prospect of identifying other vulnerabilities.

We will briefly discuss these two vulnerabilities but not go into much detail.

CVE-2023-29360 - logical vulnerability

The first one is the vulnerability used by Synacktiv in Pwn2Own Vancouver 2023.

```
__int64 __fastcall FsAllocAndLockMdl(void *user_addr, ULONG size, struct _MDL **a3)
{
  ...
  if ( user_addr && size && a3 )
  {
    Mdl = IoAllocateMdl(user_addr, size, 0, 0, 0LL);
    v6 = Mdl;
    if ( Mdl )
    {
      MmProbeAndLockPages(Mdl, 0, IoWriteAccess);
      *a3 = v6;
    }
```

This is a logical vulnerability in the MSKSSRV driver. When MSKSSRV uses MmProbeAndLockPages to lock user-specified memory address as framebuffer, the `AccessMode` was not set correctly. This leads to a failure to check whether the user-specified address belongs to the user space. If the user provides a kernel address, it will map the specified kernel address to user space for the user to use. Ultimately, this allows the user to write data to any address in the kernel. The exploitation is simple and very stable, making it become one of the most popular vulnerabilities.

For more details, please refer to Synacktiv's presentation at HITB 2023 HKT and Nicolas Zilio(@Big5_sec) 's blog post.

CVE-2023-36802 - type confusion

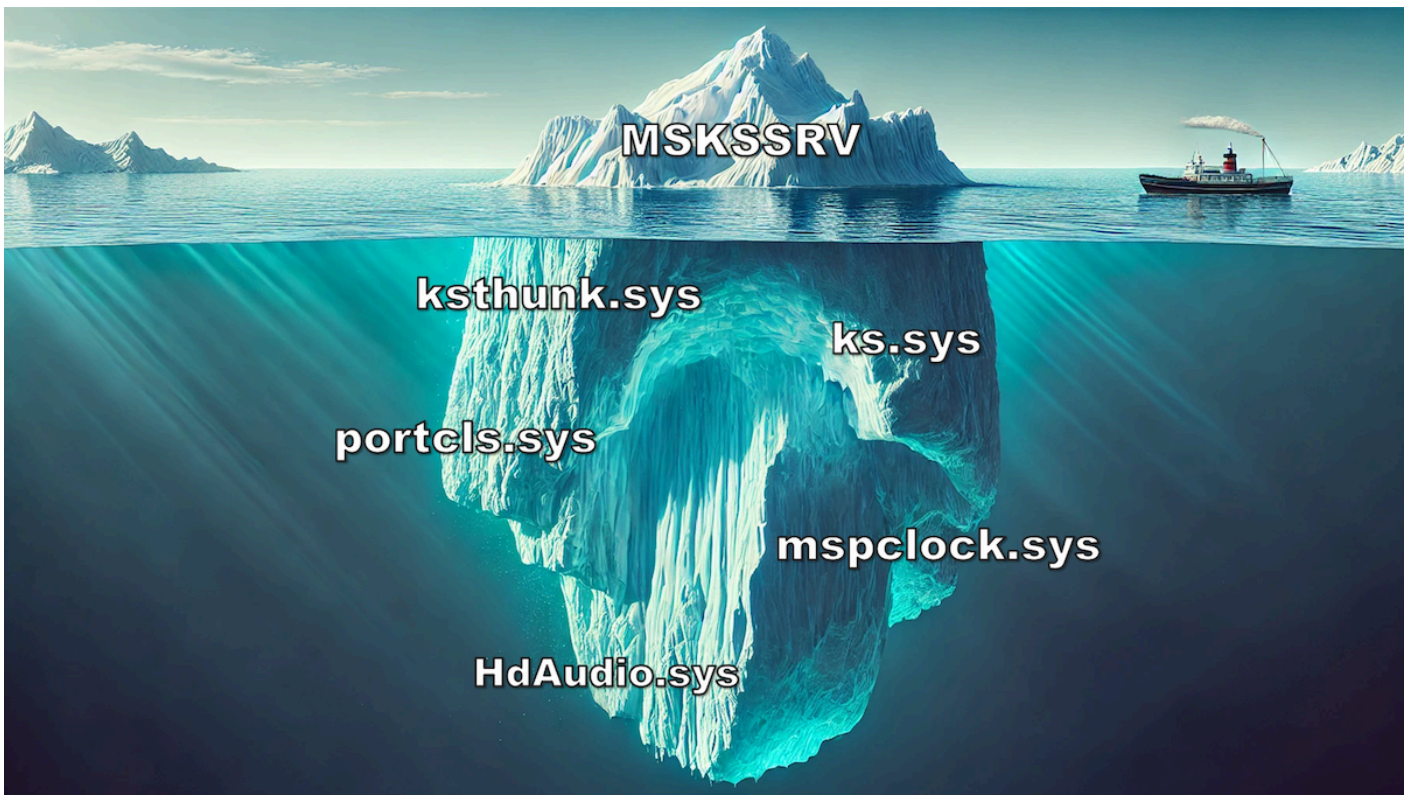
This vulnerability was discovered shortly after CVE-2023-29360 was released. It was already being exploited when Microsoft released their patches. This is a very easily discovered vulnerability. It uses the objects (`FSContextReg`, `FSStreamReg`) stored in `FILE_OBJECT->FsContext2` for subsequent processing. However, there is no check on the type of `FsContext2`, leading to type confusion. For detailed information, you can refer to the IBM X-Force blog, which provides a very thorough explanation.

Since then, there have been very few vulnerabilities related to MSKSSRV. Due to its relatively small size, MSKSSRV is quickly reviewed, and gradually, fewer and fewer people pay attention to it.

But is that the end of it ?

However, does this mean there are no more vulnerabilities?

In fact, the entire Kernel Streaming looks like the diagram below:

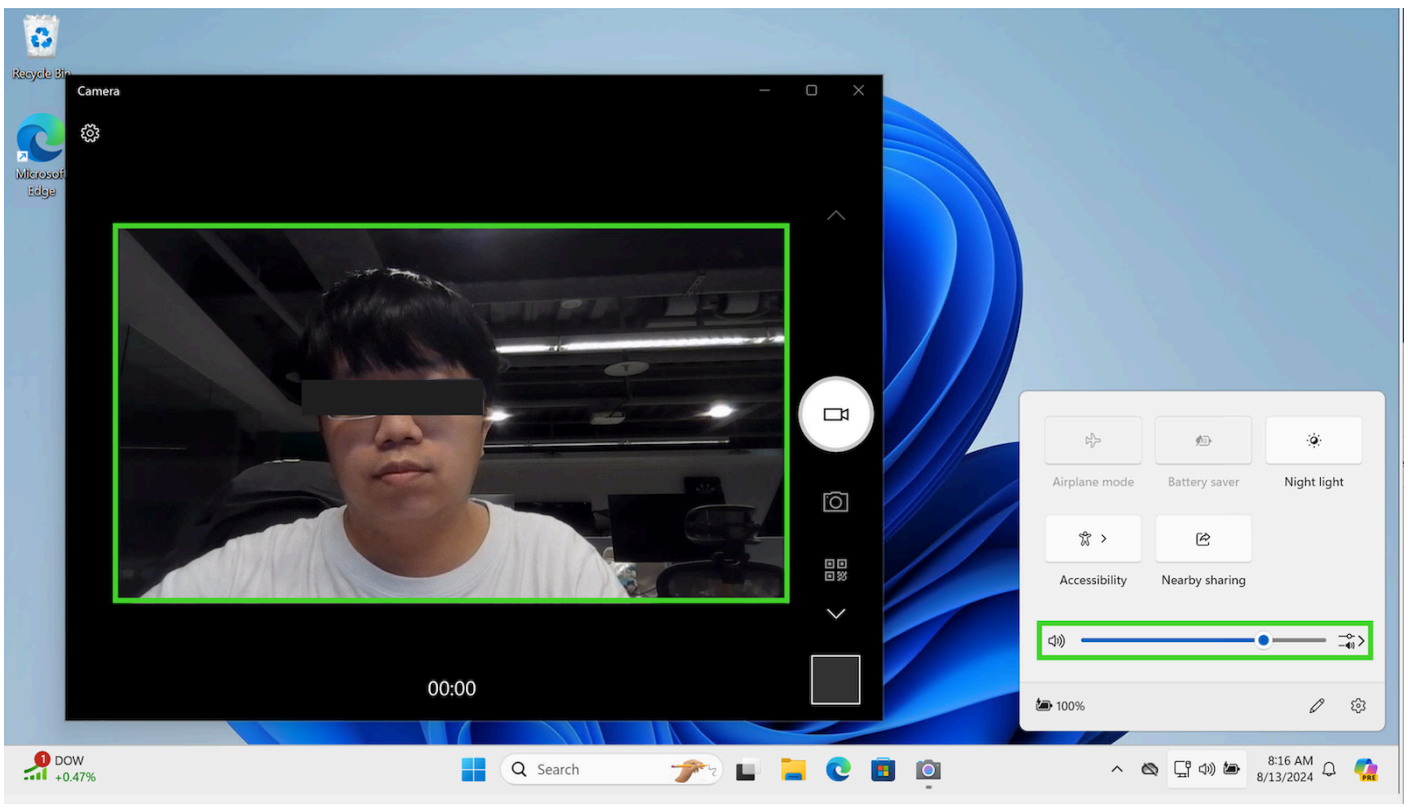


MSKSSRV is just the tip of the iceberg. In fact, there are many other potential components, and those listed in the diagram above are all part of Kernel Streaming. After delving into this attack surface, numerous vulnerabilities were eventually discovered, flowing like a stream.

By the way, while I was writing this blog, chompie also published about the vulnerability she used in this year's Pwn2Own Vancouver 2024, CVE-2024-30089, which is also a vulnerability in MSKSSRV. The vulnerability lies in handling the reference count, requiring a lot of attention and thought to discover. It is also quite interesting, but I won't discuss it here. I highly recommend reading this one.

Brief overview of Kernel Streaming

So, what is Kernel Streaming? In fact, we use it very frequently.



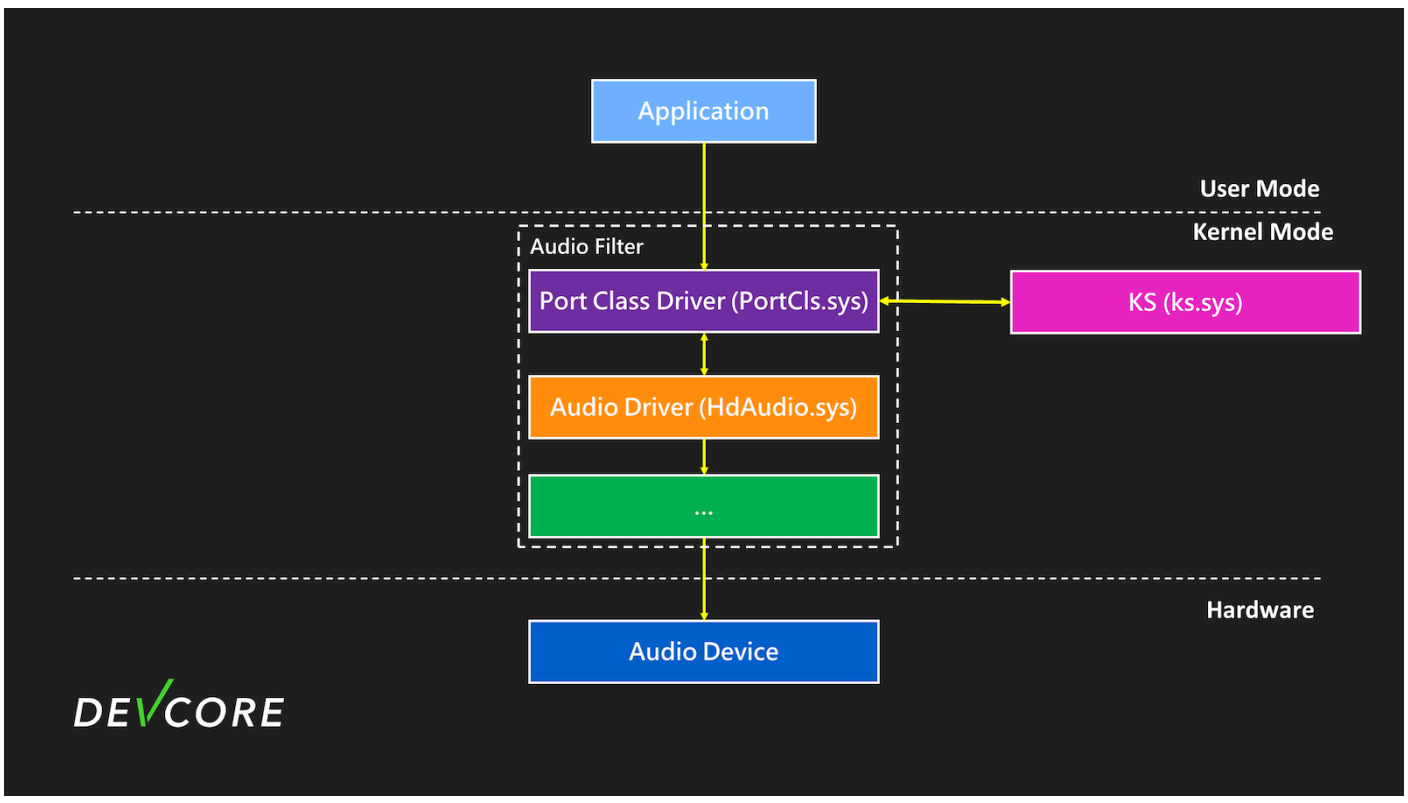
On Windows systems, when we open the webcam, enable sound, and activate audio devices such as microphones, the system needs to write or read related data such as your voice and captured images from your devices into RAM. It is essential to read data into your computer more efficiently during this process. Microsoft provides a framework called Kernel Streaming to handle these data, which **primarily operates in kernel mode**. It features low latency, excellent scalability, and a unified interface, making handling streaming data more convenient and efficient.

In Microsoft's Kernel Streaming, three multimedia class driver models are provided: port class, AVStream, and stream class. We will briefly introduce port

class and AVStream, as stream class is less common and more outdated and will not be discussed here.

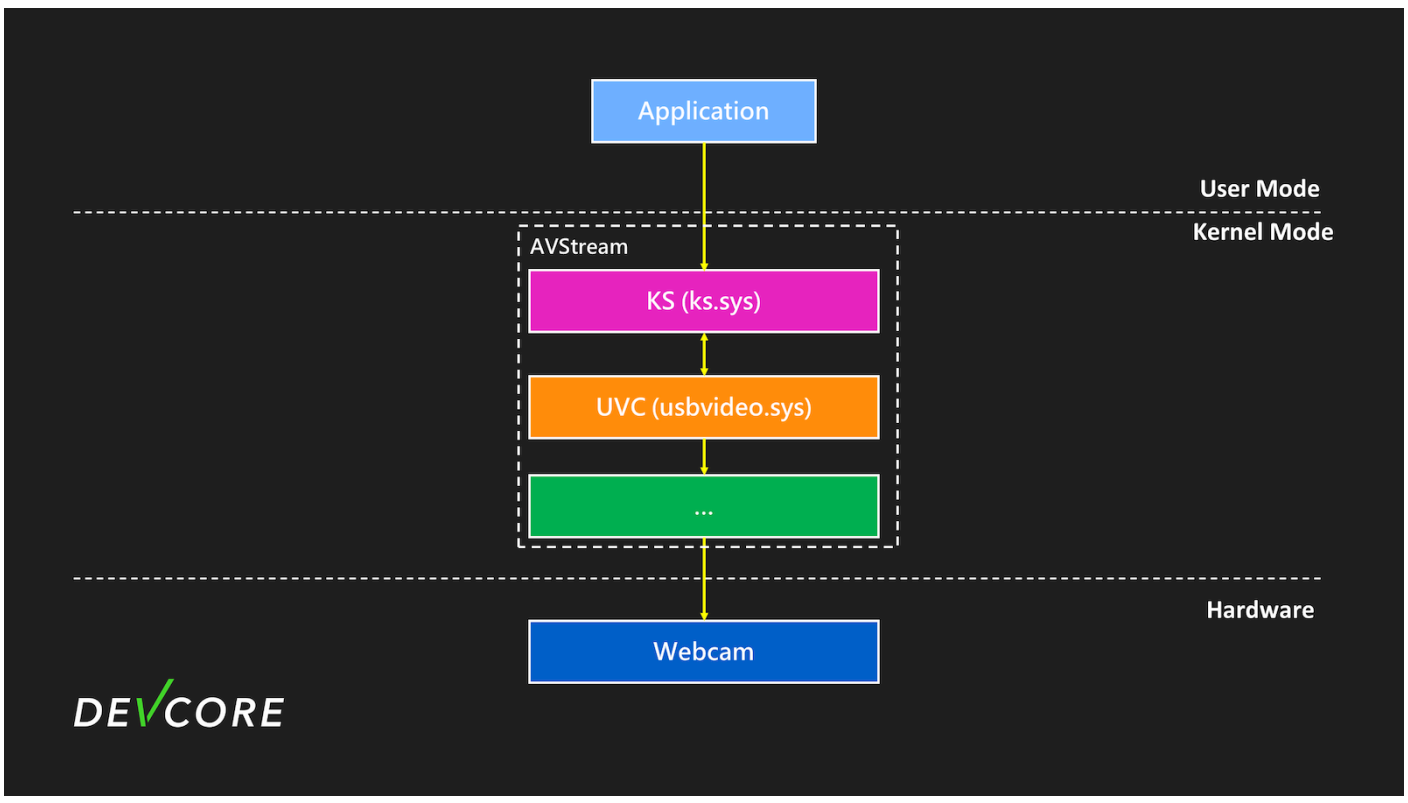
Port class

This type of driver is mostly used for PCI and DMA-based audio device hardware drivers. Currently, most audio-related processing, such as volume control or microphone-related processing, falls into this category. The main component library used would be `portcls.sys`.



AVStream

AVStream is a multimedia driver provided by Microsoft that primarily supports video-only and integrated audio/video streaming. Currently, most video-related processing, such as your webcam, capture card, etc., is associated with this category.



In fact, the use of Kernel Streaming is also very complex. We will only provide a brief description. For more detailed information, please refer to Microsoft Learn.

Interact with device

When we want to interact with audio devices or webcams, we need to open the device just like with any other device. Essentially, it interacts with the device driver in the same way. So, what would the names of these types of devices be? These names are not typically like `\Device\NamedPipe`, but rather something like the following:

```
\\?\hdaudio#subfunc_01&ven_8086&dev_2812&nid_0001&subsys_00000000&rev_1000#6&2f1f346a
```

Enumerate device

Here you can use APIs such as SetupDiGetClassDevs to enumerate devices. Generally, KS series devices are registered under `KSCATEGORY*`, such as audio devices which are registered under KSCATEGORY_AUDIO.

Additionally, you can use the APIs provided by KS, such as KsOpenDefaultDevice, to obtain the `handle` of the first matching PnP device in that category. Actually, it is just a wrapper around `SetupDiGetClassDevs` and `CreateFile`.

```
hr = KsOpenDefaultDevice(KSCATEGORY_VIDEO_CAMERA,GENERIC_READ|GENERIC_WRITE, &g_hDevi
```

Kernel Streaming object

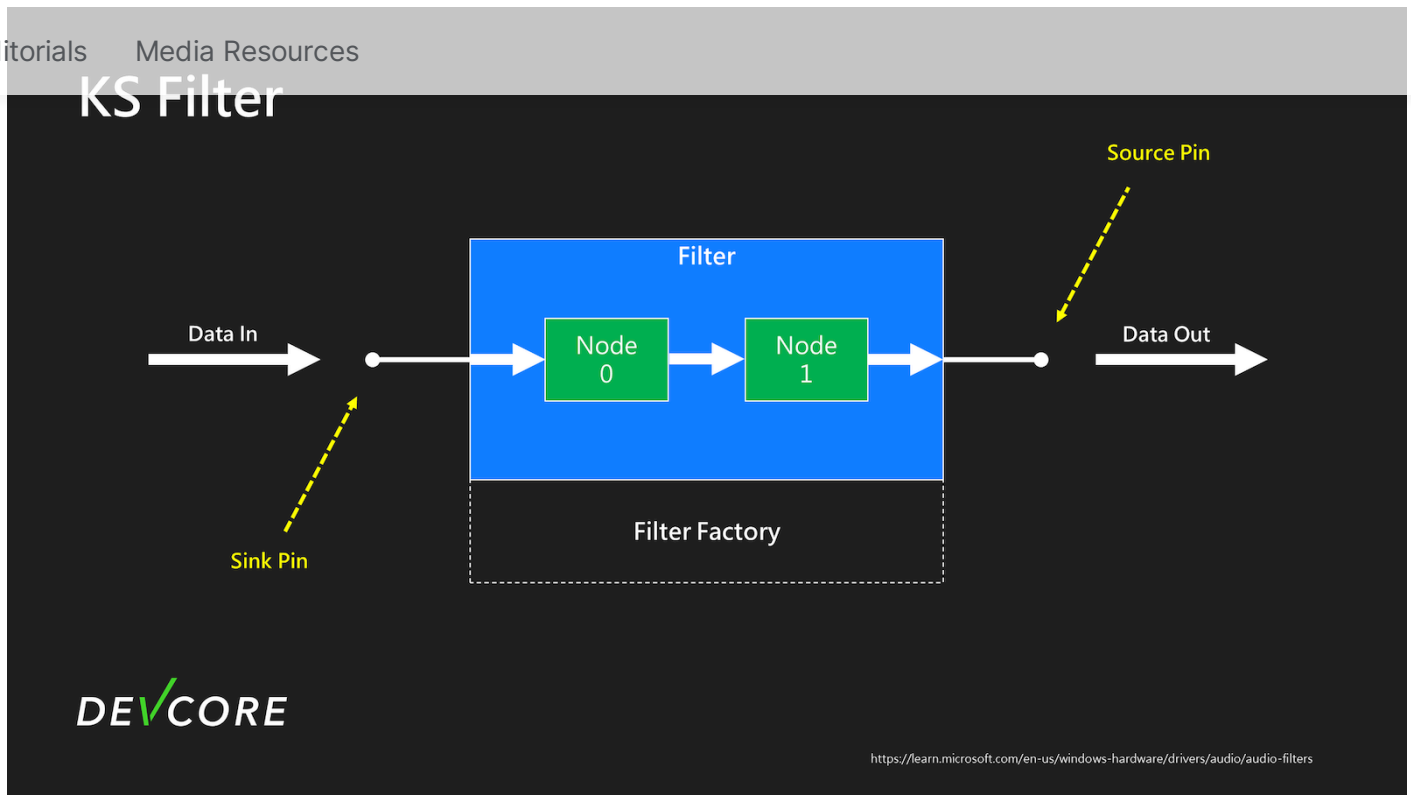
After we open these devices, Kernel Streaming will create some Kernel Streaming related instances, the most important of which are KS Filters and KS Pins.

These will be used during the Kernel Streaming process. We will only provide a brief introduction here. We will use audio filters as an example, as most others are quite similar.

KS filters

Each KS Filter typically represents a device or a specific function of a device. When we open an audio device, it usually corresponds to a KS filter object. When we read data from the audio device, this data is first processed through this KS Filter.

Conceptually, as shown in the diagram below, the large box in the middle represents a KS filter for the audio device. When we want to read data from the audio device, it is read into the filter from the left, processed through several nodes, and then output from the right.



(From: https://learn.microsoft.com/en-us/windows-hardware/drivers/audio/audio-filters)

KS pins

In the figure above, the points for reading and outputting data are called pins. The kernel also has corresponding KS pin Objects to describe the Pins, recording whether the Pin is a sink or a source, the data format for input and output, and so on. When we use it, we must open a pin on the filters to create an instance to read from or write to the device.

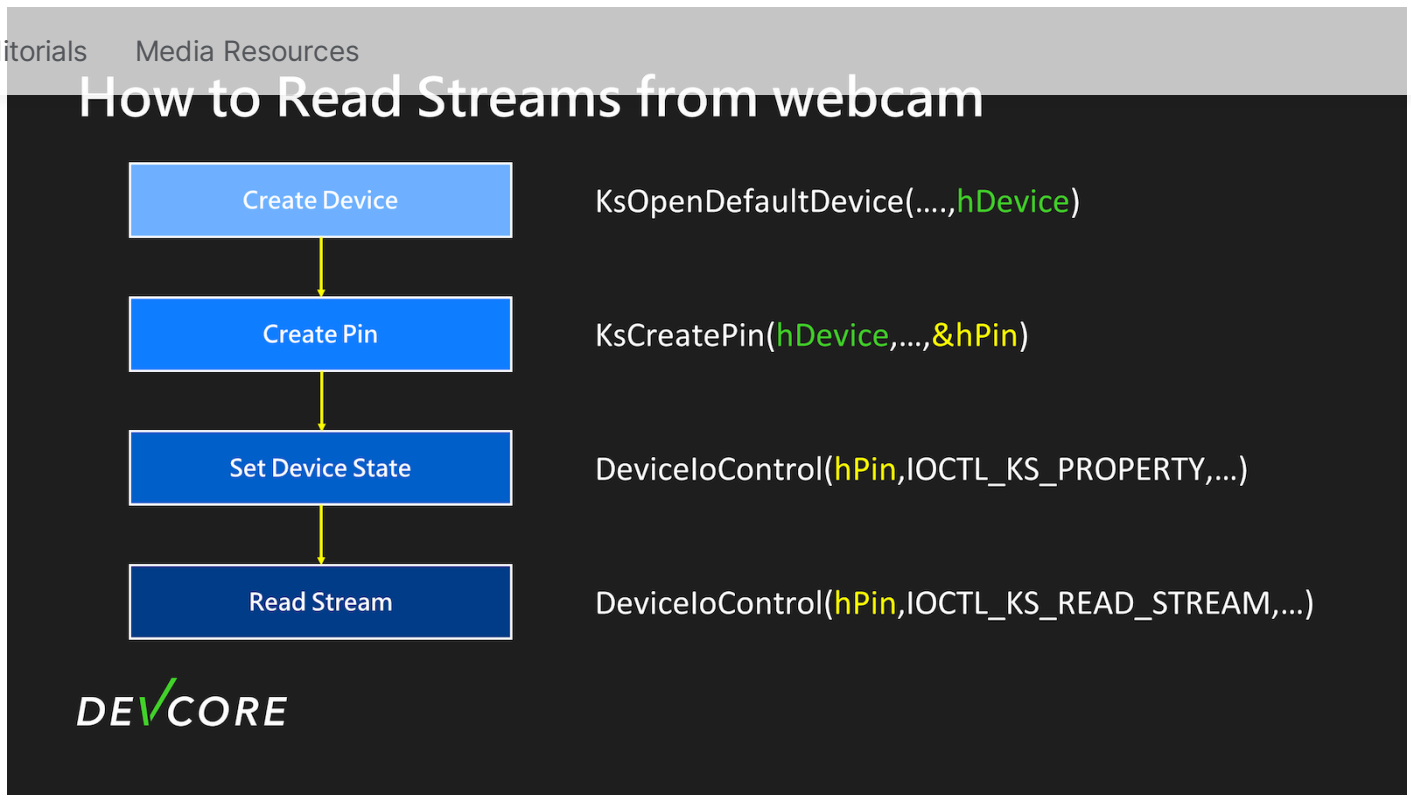
KS property

Each of these KS objects will have its own property, and each property corresponds to a specific feature. For instance, the data format mentioned earlier in the Pin, the volume level, and the status of the device are all properties. These properties typically correspond to a set of GUIDs. We can set these properties through IOCTL_KS_PROPERTY.

This greatly simplifies the development of multimedia drivers and ensures consistency and scalability across different devices.

Read streams from webcam

Here is a simple example illustrating how an application can read data from a webcam.



The most basic flow is roughly shown in this diagram:

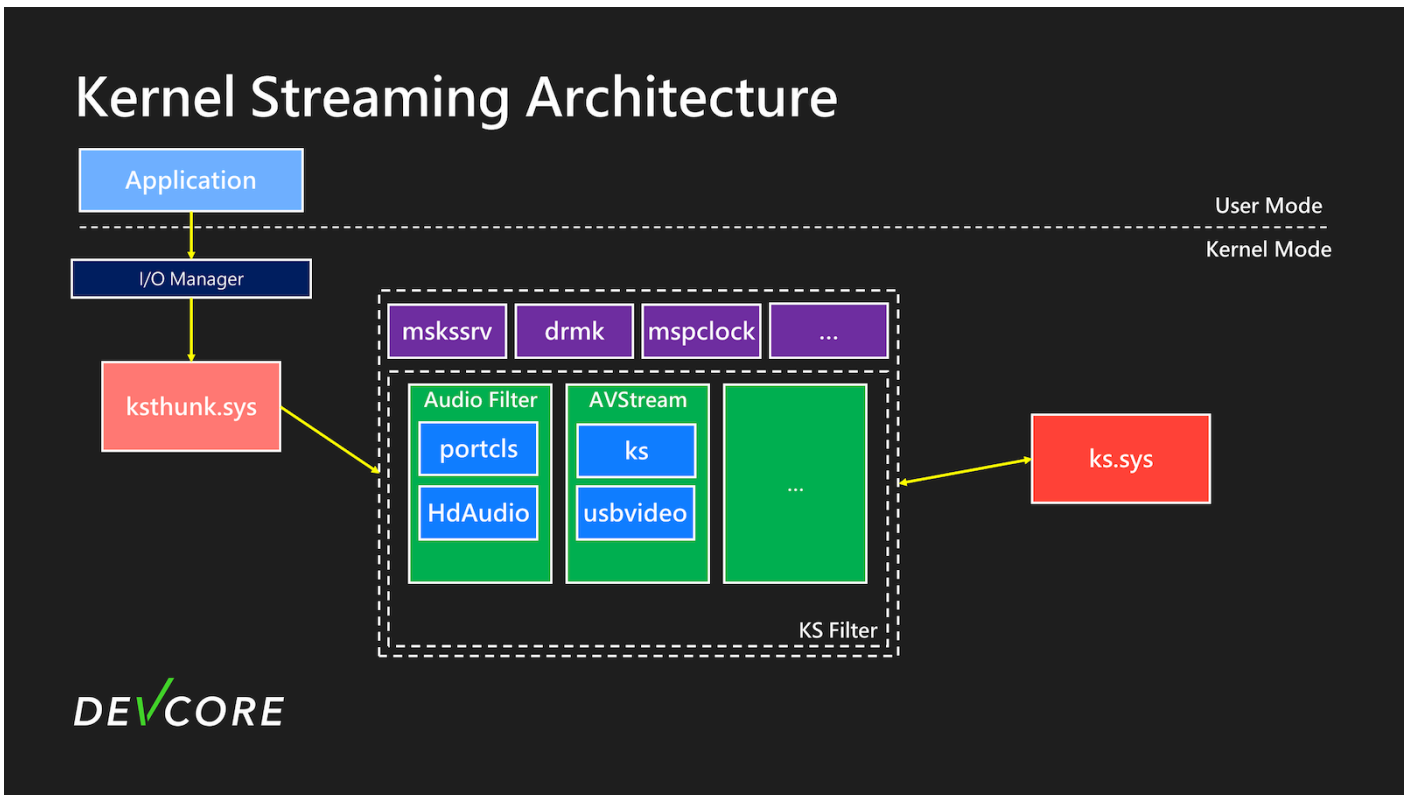
1. Open the device to obtain the device handle.
2. Use this device handle to create an instance of the Pin on this filter and obtain the Pin handle.
3. Use `IOCTL_KS_PROPERTY` to set the device state of the Pin to RUN.
4. Finally, you can use IOCTL_KS_READ_STREAM to read data from this Pin.

Kernel Streaming architecture

> For vulnerability research, we must first understand its architecture and consider the potential attack surfaces.

After gaining a preliminary understanding of the functionalities and operations of Kernel Streaming, we need to understand the architecture to find vulnerabilities. It's crucial to know how Windows implements these functions and what components are involved. This way, we can identify which sys files to analyze and where to start.

After our analysis, the overall architecture looks approximately like this diagram:



In the Kernel Streaming components, the most important ones are `ksthunk.sys` and `ks.sys`. Almost all functionalities are related to them.

ksthunk (Kernel Streaming WOW Thunk Service Driver)

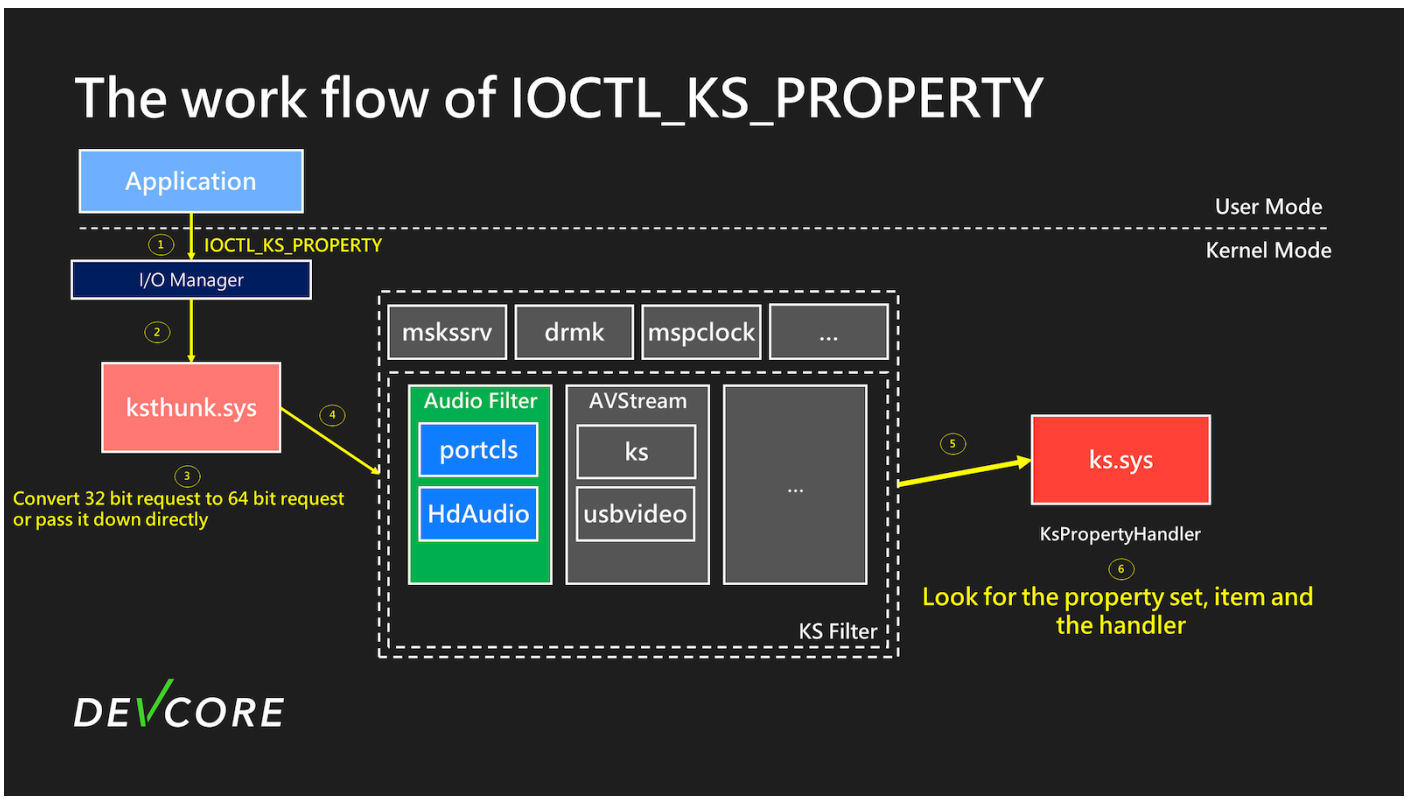
`ksthunk.sys` is the entry point in Kernel Streaming. Its function is quite simple: it converts 32-bit requests from the WoW64 process into 64-bit requests, allowing the underlying driver to handle the requests without additional processing for 32-bit structures.

ks (Kernel Connection and Streaming Architecture Library)

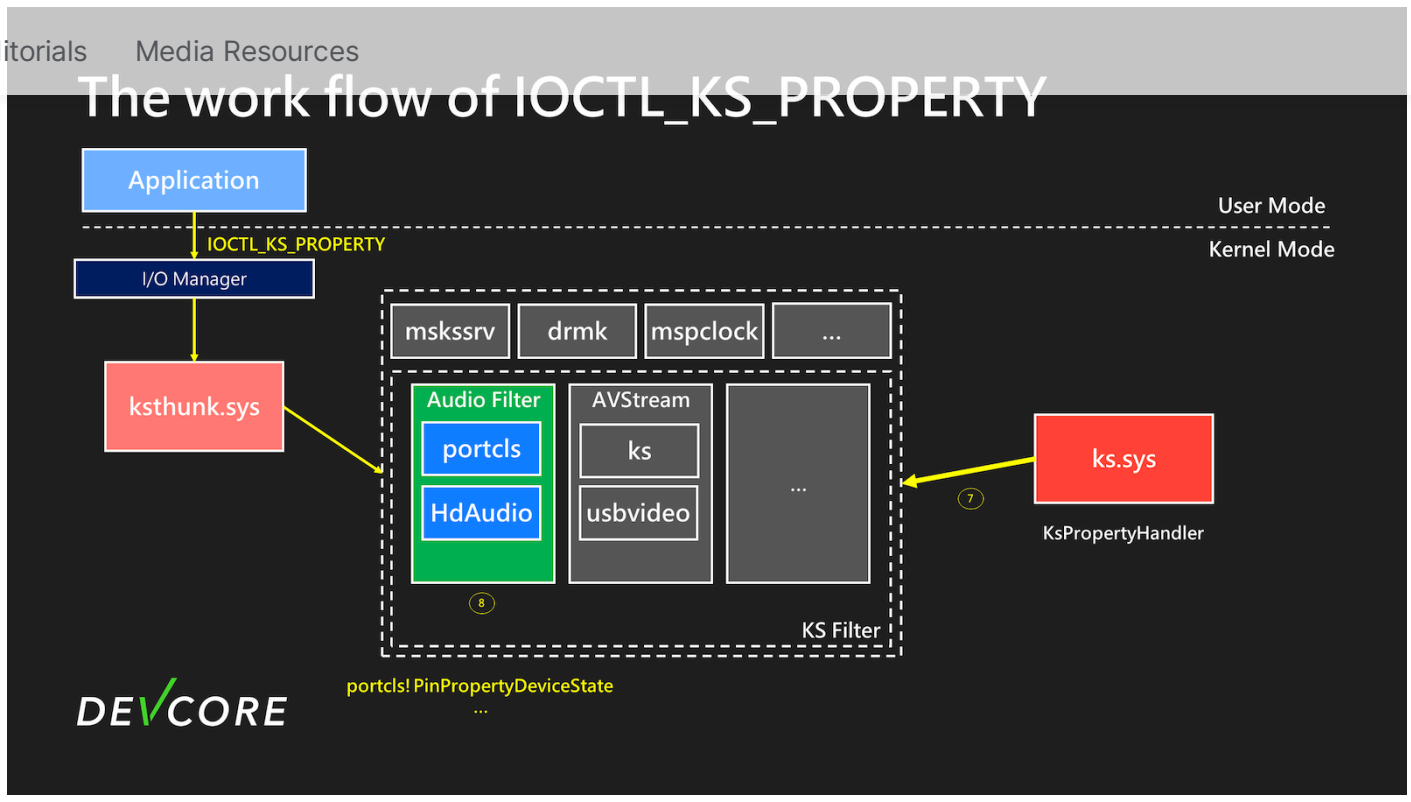
`ks.sys` is one of the **core components of Kernel Streaming**. It is the library of Kernel Streaming responsible for forwarding requests such as `IOCTL_KS_PROPERTY` to the corresponding device driver, and it also handles functions related to AVStream.

The work flow of IOCTL_KS_*

`IOCTL_KS_PROPERTY` will be used as an example here. When calling `DeviceIoControl`, as shown in the figure below, the user's request will be sequentially passed to the corresponding driver for processing.



At step 6, `ks.sys` will determine which driver and handler to hand over your request based on the KSPROPERTY you requested.



Finally, forward it to the corresponding driver, as shown in the figure above, where it is ultimately forwarded to the handler in `portcls` to operate the audio device.

You should now have a preliminary understanding of the architecture and process of Kernel Streaming. Next, it's time to look for vulnerabilities.

Based on the existing elements, which attack surfaces are worth examining?

From attacker's view

> Before digging for vulnerabilities, if you can carefully consider under what circumstances they are likely to occur, you can achieve twice the result with half the effort.

From a vulnerability researcher's perspective, there are a few key points to consider:

1. Property handler in each device

KS object for each device has its own properties, and each property has its own handler. Some properties are prone to issues during handling.

2. ks and ksthunk

`ks` and `ksthunk` have not had vulnerabilities for a long time, but they are the most accessible entry points and might be good targets. The last vulnerabilities(CVE-2020-16889 and CVE-2020-17045) were found in 2020 by @nghiadt1098.

3. Each driver handles a part of the content

In some functionalities of Kernel Streaming, certain drivers handle parts of the input individually, which may lead to inconsistencies.

After reviewing Kernel Streaming from the above perspectives, we quickly identified several relatively easy-to-discover vulnerabilities.

- portcls.sys
  - CVE-2024-38055 (out-of-bounds read when set dataformat for Pin)

- CVE-2024-38056
- ksthunk
  - CVE-2024-38054 (out-of-bounds write)
  - CVE-2024-38057

However, we will not be explaining these vulnerabilities one by one. Most of these are obvious issues such as unchecked length or index leading to out-of-bounds access. @Fr0st1706 also wrote an exploit for CVE-2024-38054 recently. We might slowly explain these in subsequent parts in the future. We will leave this for the readers to study for now.

During the review process, we discovered some interesting things. Do you think the following code snippet is really fine?

```
__int64 __fastcall CKSThunkDevice::CheckIrpForStackAdjustmentNative(__int64 a1, struc
{

    if ( irp->RequestorMode )
    {
        v14 = 0xC0000010;
    }
    else
    {
        UserBuffer = (unsigned int *)irp->UserBuffer;
        ...
        v14 = (*(__int64 (__fastcall **)(_QWORD, _QWORD, __int64 *))    (Type3InputBu
                *UserBuffer,
                0LL,
               v19);
    }
}
```

Seeing this code reminds me of CVE-2024-21338. Initially, the vulnerability had no checks, but after patching, ExGetPreviousMode was added. However, the check here uses the `RequestorMode` in IRP for validation. Generally, the `RequestorMode` from a user-called IOCTL will be `UserMode(1)`, so there shouldn't be any issues.

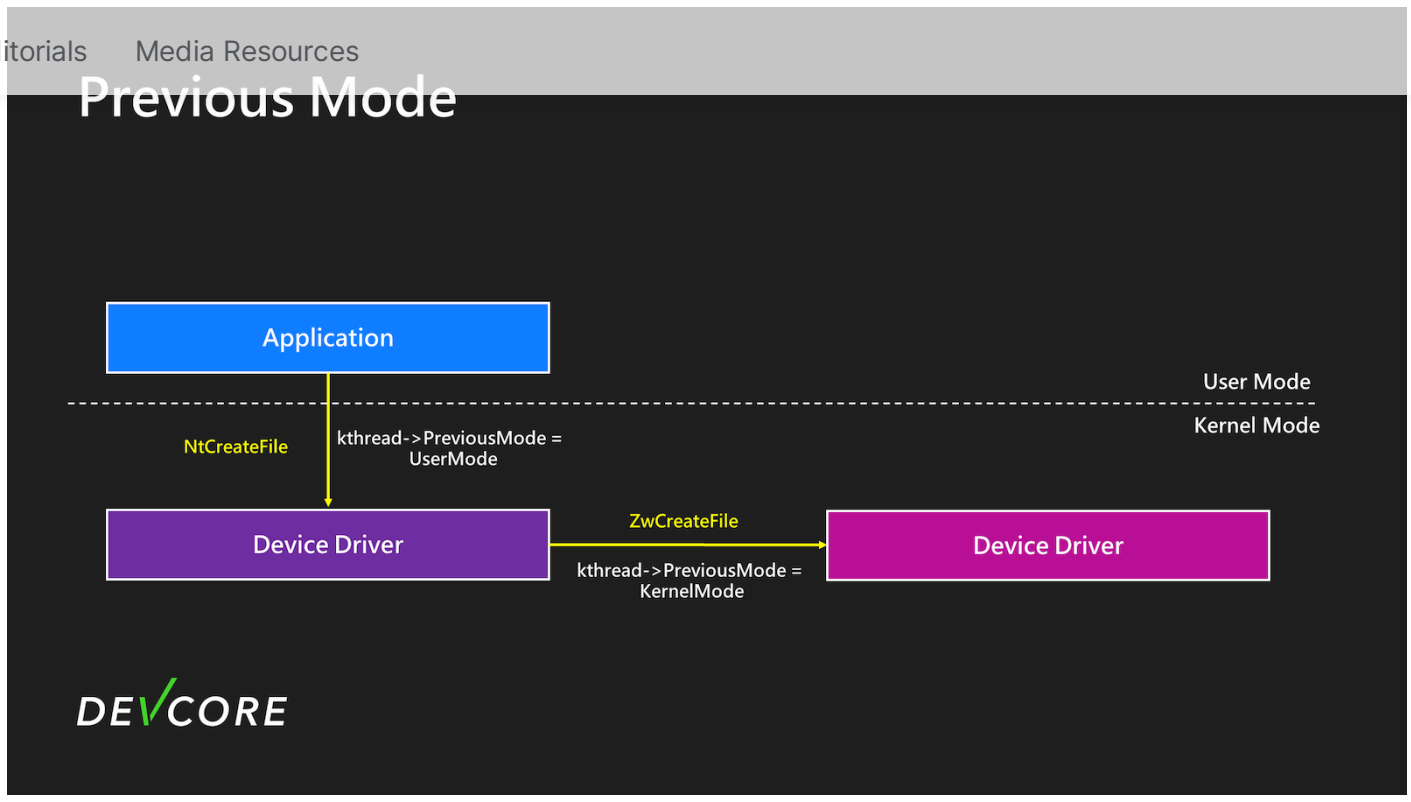
At this point, I also recall James Forshaw's article Windows Kernel Logic Bug Class: Access Mode Mismatch in IO Manager.

The overlooked bug class

First, we need to mention a few terms and concepts. If you are already familiar with `Previous Mode` and `RequestorMode`, you can skip to A logical bug class section.

PreviousMode

The first one is `PreviousMode`. In an Application, if a user operates on a device or file through `Nt*` System Service Call, upon entering the kernel, it will be marked as `UserMode(1)` in _ETHREAD's `PreviousMode`, indicating that this System Service Call is from the user. Conversely, if it is called from kernel mode, such as a device driver invoking the `Zw*` System Service Call, it will be marked as `KernelMode(2)`.



RequestorMode

Another similar field is the `RequestorMode` in the IRP. This field records whether your original request came from `UserMode` or `KernelMode`. In kernel driver code, this is a very commonly used field, typically derived from `PreviousMode`.

It is often used to decide whether to perform additional checks on user requests, such as Memory Access Check or Security Access Check. In the example below, if the request comes from `UserMode`, it will check the user-provided address. If it comes from the Kernel, no additional checks are performed to increase efficiency.

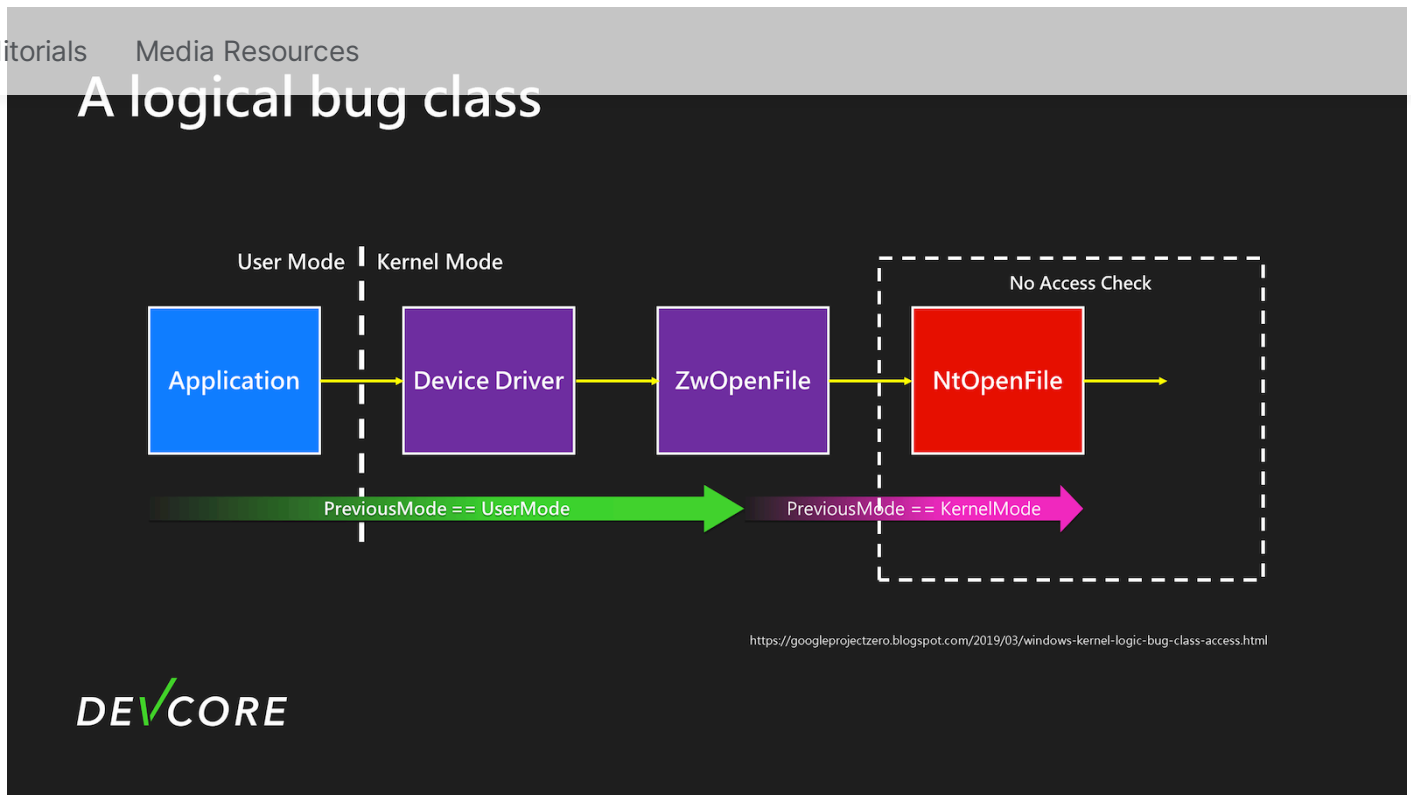
```
if ( Irp->RequestorMode )
{
  ProbeForRead(CurrentStackLocation->Parameters.DeviceIoControl.Type3InputBuffer, InputBufferLength, 1u)
  a4 = callback;
  outputLength = outlen;
}
```

But in reality, this has also led to some issues.

A logical bug class

James Forshaw's Windows Kernel Logic Bug Class: Access Mode Mismatch in IO Manager mentions a type of Bug Class.

What would happen if a user calls a System Service Call like `NtDeviceIoControlFile`, and then the driver handling it uses user-controllable data as parameters for `ZwOpenFile`?



After the driver calls `ZwOpenFile`, `PreviousMode` will switch to `KernelMode`, and when it uses `NtOpenFile` processing, most checks will be skipped due to `PreviousMode` being `KernelMode`. Subsequently, `Irp->RequestorMode` will become `KernelMode`, bypassing the Security Access Check and Memory Access Check. However, this largely depends on how the subsequent driver implements these checks. There might be issues if it relies solely on `RequestorMode` to decide whether to perform checks The actual situation is slightly more complex and related to the flags of `CreateFile`. For details, refer to the following articles:

- Windows Kernel Logic Bug Class: Access Mode Mismatch in IO Manager
- Hunting for Bugs in Windows Mini-Filter Drivers
- Local privilege escalation via the Windows I/O Manager: a variant finding collaboration

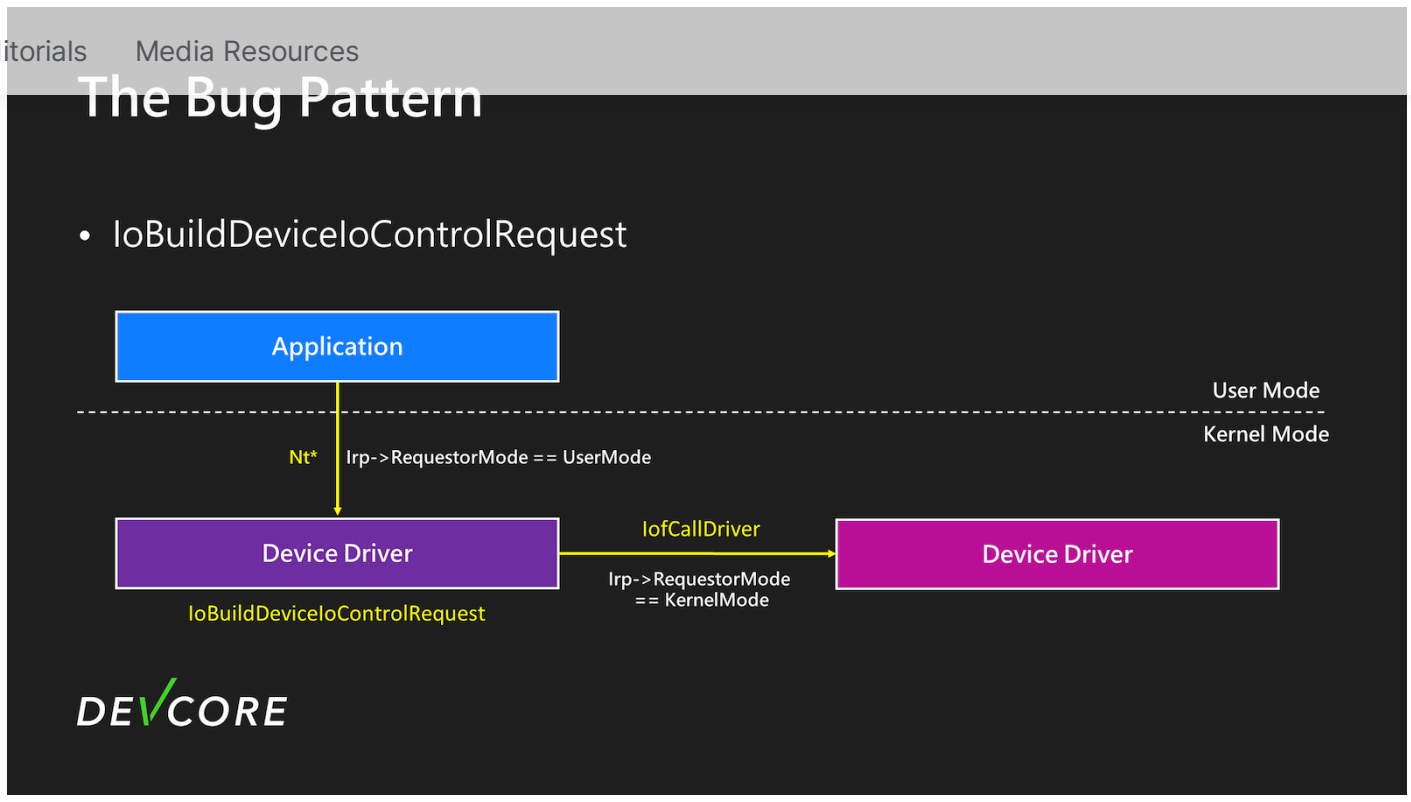
These studies mainly focused on the `Zw*` series of System Service Call. Are there other similar situations that could also cause this kind of logical vulnerability?

The new bug pattern

Actually, it is possible. When the device driver uses IoBuildDeviceIoControlRequest to create a `DeviceIoControl` IRP, it is easy to encounter such issues if not careful. This API is primarily used by kernel drivers to call IOCTL, and it helps you build the IRP. Subsequently, calling `IofCallDriver` allows you to call IOCTL within the kernel driver. On Microsoft Learn, there is a particular passage worth noting.

> If the caller supplies an *InputBuffer* or *OutputBuffer* parameter, this parameter must point to a buffer that resides in system memory. The caller is responsible for validating any parameter values that it copies into the input buffer from a user-mode buffer. The input buffer might contain parameter values that are interpreted differently depending on whether the originator of the request is a user-mode application or a kernel-mode driver. In the IRP that **IoBuildDeviceIoControlRequest** returns, the **RequestorMode** field is always set to **KernelMode**. This value indicates that the request, and any information contained in the request, is from a trusted, kernel-mode component.

By default, if you do not explicitly set the **`RequestorMode`**, it will directly call IOCTL with `KernelMode`.



Following this approach, we revisited Kernel Streaming and discovered an intriguing aspect.

```
NTSTATUS __stdcall KsSynchronousIoControlDevice(
    PFILE_OBJECT FileObject,
    KPROCESSOR_MODE RequestorMode,
    ULONG IoControl,
    PVOID InBuffer,
    ULONG InSize,
    PVOID OutBuffer,
    ULONG OutSize,
    PULONG BytesReturned)
{

    KeInitializeEvent(&Event, NotificationEvent, 0);
    NewIrp = IoBuildDeviceIoControlRequest(
        IoControl,
        RelatedDeviceObject,
        InBuffer,
        InSize,
        OutBuffer,
        OutSize,
        0,
        &Event,
        &IoStatusBlock);
    ...
    NewIrp->RequestorMode = RequestorMode;
    ...
    Status = IofCallDriver(RelatedDeviceObject, NewIrp);
}
```

The function where `IoBuildDeviceIoControlRequest` is used in Kernel Streaming is in `ks!KsSynchronousIoControlDevice` and it obviously involves calling IOCTL in the kernel using the aforementioned method. However, it appears that `Irp->RequestorMode` is properly set here, and different values are assigned based on

the parameters of KsSynchronousIoControlDevice. This will be a convenient library for kernel streaming driver developers.

However...

ks!CKsPin::GetState

```
BytesReturned = 0;
v5 = KsSynchronousIoControlDevice(m_Worker, 0, 0x2F0003u, &InBuffer, 0x18u, OutBuffer,
if ( v5 >= 0 && BytesReturned != 4 )
  v5 = -1073741306;
```

ks!SerializePropertySet

```
if ( SerialSize )
{
  v19 = KsSynchronousIoControlDevice(
          CurrentStackLocation->FileObject,
          0,
          CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode,
          PoolWithTag,
          InSize,
          (v16 + 0x20),
          SerialSize,
          &BytesReturned);
```

ks!UnserializePropertySet

```
    if ( OutSize > v13 )
      goto error2;
    status = KsSynchronousIoControlDevice(
               CurrentStackLocation->FileObject,
               0,
               CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode,
               New_KsProperty_req,
               InSize,
               OutBuffer,
               OutSize,
               &BytesReturned);
```

We found that in Kernel Streaming, all functions using `KsSynchronousIoControlDevice` consistently use **`KernelMode(0)`**. At this point, we can carefully inspect whether the places it is used have any security issues. Therefore, we convert the bug pattern in Kernel Streaming into the following points:

1. Utilized `KsSynchronousIoControlDevice`
2. Controllable InputBuffer & OutputBuffer
3. The second processing of IOCTL relies on **RequestorMode** for security checks

```
if ( irp->RequestorMode )
  ProbeForRead(CurrentStackLocation->Parameters.DeviceIoControl.Type3InputBuffer, inputbuf, 1u);
```

Following this pattern, we quickly found the first vulnerability.

The vulnerability & exploitation

CVE-2024-35250

This vulnerability is also the one we used in Pwn2Own Vancouver 2024. In the `IOCTL_KS_PROPERTY` of Kernel Streaming, to increase efficiency, `KSPROPERTY_TYPE_SERIALIZESET` and `KSPROPERTY_TYPE_UNSERIALIZESET` requests are provided to allow users to operate on multiple properties through a single call. These types of requests will be broken down into multiple calls by the `KsPropertyHandler`. For more details, refer to this one.

It is implemented in ks.sys

```
NTSTATUS __fastcall KspPropertyHandler(
        PIRP Irp,
        unsigned int propertysetscnt,
        KSPROPERTY_SET *propertyset,
        __int64 (__fastcall *a4)(_QWORD, _QWORD, _QWORD),
        int a5,
        __int64 NodeAutomationTable,
        unsigned int NodeCnt){

        // check if the UserProvideProperty->Set is in the propertyset

        ...
        if ( KsProperty_flag == KSPROPERTY_TYPE_UNSERIALIZESET )
          return UnserializePropertySet(Irp, sysbuf_, propertyset_);
        ...

}
```

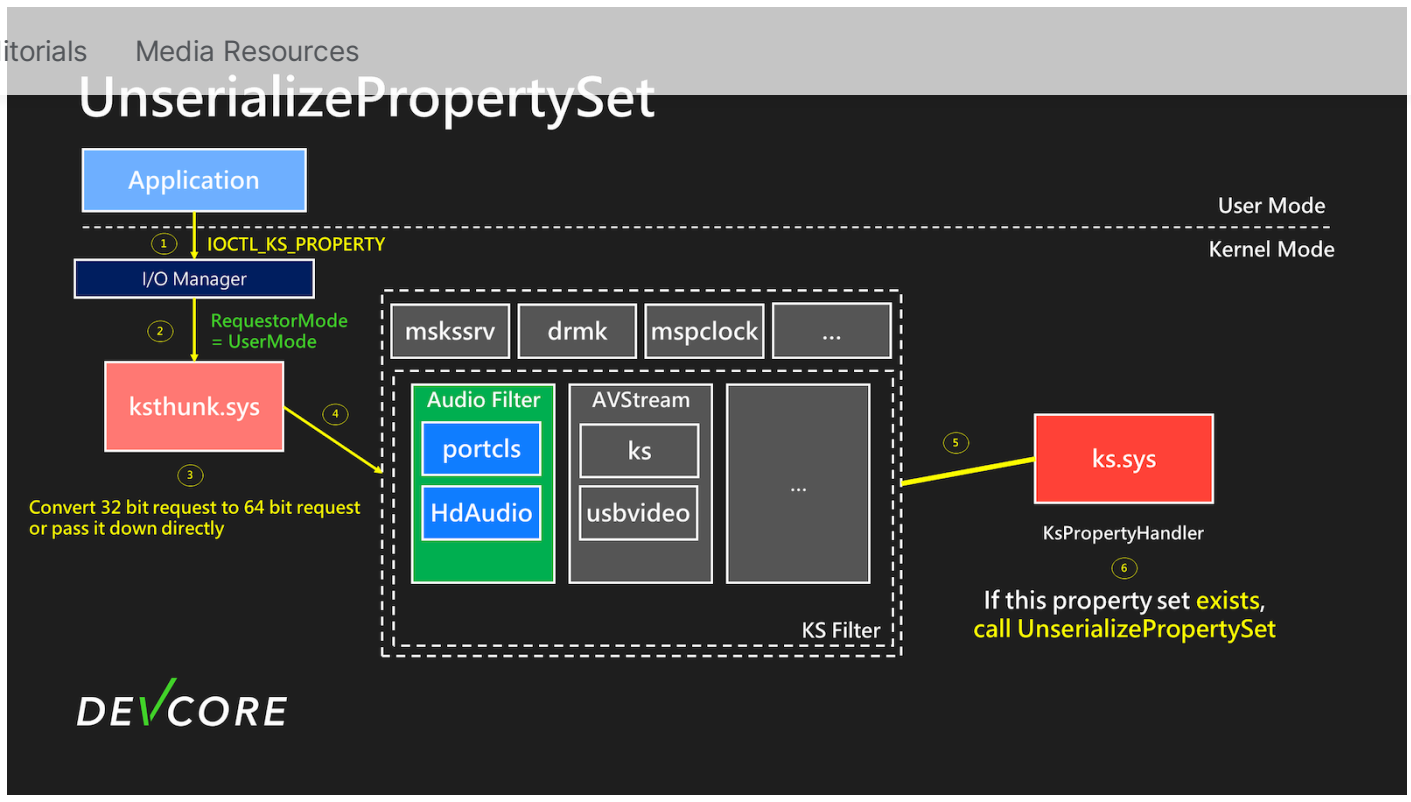
When handling property in `ks.sys`, if the `KSPROPERTY_TYPE_UNSERIALIZESET` flag is provided, `ks!UnserializePropertySet` will handle your request.

Let's take a look at `UnserializePropertySet`.

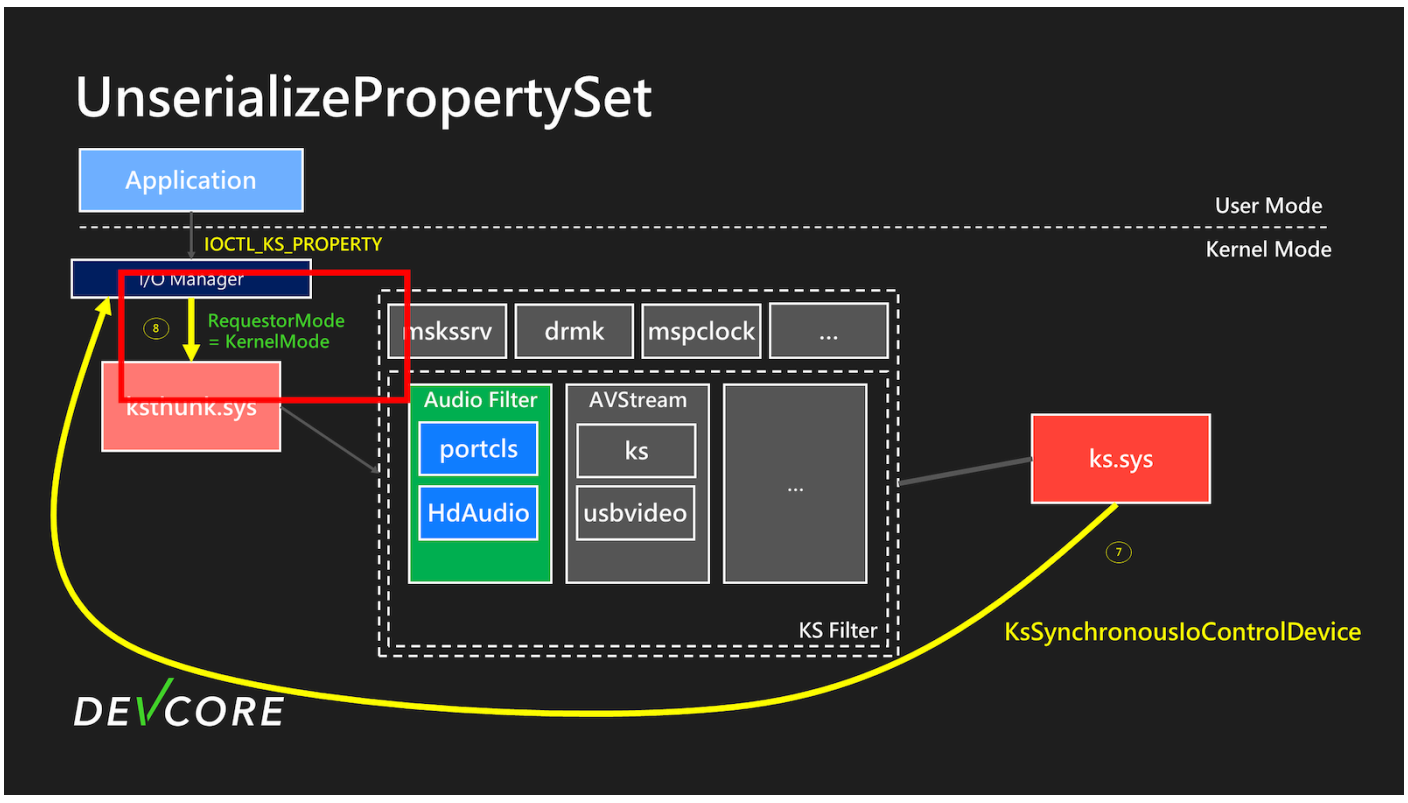
```
unsigned __int64 __fastcall UnserializePropertySet(
    PIRP irp,
    KSIDENTIFIER* UserProvideProperty,
    KSPROPERTY_SET* propertyset_)
{
    ...
    New_KsProperty_req = ExAllocatePoolWithTag(NonPagedPoolNx, InSize, 0x7070534Bu);
    ...
    memmove(New_KsProperty_req, CurrentStackLocation->Parameters.DeviceIoControl.Type
    ...
    status = KsSynchronousIoControlDevice(
            CurrentStackLocation->FileObject,
            0,
            CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode,
            New_KsProperty_req,
            InSize,
            OutBuffer,
            OutSize,
            &BytesReturned); //-----------[2]
    ...
}
```

You can see that during the processing, the original request is first copied into a newly allocated buffer at [1]. Subsequently, this buffer is used to call the new IOCTL using `KsSynchronousIoControlDevice` at [2]. Both `New_KsProperty_req` and `OutBuffer` are contents provided by the user.

The flow when calling `UnserializePropertySet` is roughly as illustrated below:



When calling IOCTL, as shown in `step 2` of the diagram, the I/O Manager will set `Irp->RequestorMode` to `UserMode(1)`. Until step 6, it will check if the requested property by the user exists in the KS object. If the property exists in the KS object and is set with `KSPROPERTY_TYPE_UNSERIALIZESET`, `UnserializePropertySet` will be used to handle the specified property.



Next, in step 7, `KsSynchronousIoControlDevice` will be used to perform the IOCTL again. At this point, the new `Irp->RequestorMode` will become `KernelMode(0)`, and the subsequent processing will be the same as a typical `IOCTL_KS_PROPERTY`. This part will not be elaborated further.

As a result, we now have a primitive that allows us to perform arbitrary `IOCTL_KS_PROPERTY` operations. Next, we need to look for places where it might be possible to achieve EoP (Elevation of Privilege).

The EoP

The first thing you will likely notice is the entry point `ksthunk.sys`.

Let's take a look at `ksthunk!CKSThunkDevice::DispatchIoctl`.

```
__int64 __fastcall CKSThunkDevice::DispatchIoctl(CKernelFilterDevice *a1, IRP *irp, u
```

```
{
  ...
  if ( IoIs32bitProcess(irp) && irp->RequestorMode ) //------[3]
  {
   //Convert 32-bit requests to 64-bit requests
  }
  else if ( CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode == IOCTL_K
  {
    return CKSThunkDevice::CheckIrpForStackAdjustmentNative((__int64)a1, irp, v11, a4
  }
}
```

You can see that `ksthunk` will first determine whether the request is from a WoW64 Process. If it is, it will convert the original 32-bit Requests into 64-bit at [3]. If the original request is already 64-bit, it will call `CKSThunkDevice::CheckIrpForStackAdjustmentNative` at [4] to pass it down.

```
__int64 __fastcall CKSThunkDevice::CheckIrpForStackAdjustmentNative(__int64 a1, struc
{
  ...
    if ( *(_OWORD *)&Type3InputBuffer->Set == *(_OWORD *)&KSPROPSETID_DrmAudioStream
        && !type3inputbuf.Id
        && (type3inputbuf.Flags & 2) != 0 )   //-----[5]
    {
        if ( irp->RequestorMode ) //-------[6]
        {
        v14 = 0xC0000010;
        }
        else
        {
        UserBuffer = (unsigned int *)irp->UserBuffer;
        ...
        v14 = (*(__int64 (__fastcall **)(_QWORD, _QWORD, __int64 *))(Type3InputBuffer
                *UserBuffer,
                0LL,
                v19); //------------[7]
        }
    }
}
```

We can notice that if we give the property set as `KSPROPSETID_DrmAudioStream`, there is additional processing at [5]. At [6], we can see that it first checks whether `Irp->RequestorMode` is `KernelMode(0)`. If it is called from user, it will directly return an error.

It should be pretty obvious here that if we call IOCTL with `KSPROPERTY_TYPE_UNSERIALIZESET` and specify the `KSPROPSETID_DrmAudioStream` property, it will be `KernelMode(0)` here. Additionally, it will directly use the input provided by the user as a function call at [7]. Even the first parameter is controllable.

After writing the PoC, we confirmed our results.

```
BUGCHECK_CODE:  3b

BUGCHECK_P1: c0000005

BUGCHECK_P2: fffff80173333380

BUGCHECK_P3: ffffaa88a40de100

BUGCHECK_P4: 0

CONTEXT:  ffffaa88a40de100 -- (.cxr 0xffffaa88a40de100)
rax=ffff404040404040 rbx=ffff838a3cef5b20 rcx=00000000deadbee0
rdx=0000000000000000 rsi=ffff838a3cef5da0 rdi=0000000000000001
rip=fffff80173333380 rsp=ffffaa88a40deb28 rbp=ffff838a3d45e0a0
 r8=ffffaa88a40deb78  r9=ffffaa88a40dec80 r10=fffff8016aa26e90
r11=0000000000000000 r12=ffffaa88a40dec80 r13=ffff838a3df23de0
r14=4fac41982f2c8ddd r15=ffff838a3d45e0a0
iopl=0         nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00050246
ksthunk!guard_dispatch_icall_nop:
fffff801`73333380 ffe0            jmp     rax {ffff4040`40404040}
Resetting default scope
```

Some people might wonder, under what device or situation would have `KSPROPSETID_DrmAudioStream`? Actually, most audio devices will have it, primarily used for setting DRM-related content.

Exploitation

Once arbitrary calls are achieved, accomplishing EoP is not too difficult. Although protections such as kCFG, kASLR, and SMEP will be encountered, the only protection that needs to be dealt with under Medium IL is kCFG.

- **kCFG**
- kASLR
  - NtQuerySystemInformation
- SMEP
  - Reuse Kernel Code
- ...

Bypass kCFG

Our goal here is straightforward: to create an arbitrary write primitive from a legitimate function, which can then be used to achieve EoP through typical methods like replacing the current process token with system token or abusing the token privilege.

It is intuitive to look directly for legitimate functions with names containing `set`, as they are more likely to do arbitrary writing. We directly take the export functions from `ntoskrnl.exe` to see if there are any good gadgets, as these functions are generally legitimate.

Name	Address	Ordinal
RtlNumberOfSetBitsInRangeEx	00000001405A7080	2441
RtlNumberOfSetBitsUlongPtr	**00000001403B0490**	**2442**
RtlSetActiveConsoleId	0000000140758470	2505
RtlSetAllBits	**000000014024EE60**	**2506**
RtlSetAllBitsEx	00000001403B3240	2507
RtlSetBit	**000000014029A5F0**	**2508**
RtlSetBitEx	000000014029D810	2509
RtlSetBits	**000000014024D8B0**	**2510**
RtlSetBitsEx	0000000140355B70	2511
RtlSetConsoleSessionForegroundProcessId	00000001407574E0	2512
RtlSetControlSecurityDescriptor	0000000140852320	2513
RtlSetDaclSecurityDescriptor	**0000000140697010**	**2514**
RtlSetDynamicTimeZoneInformation	00000001409BBA60	2515

We quickly found RtlSetAllBits.

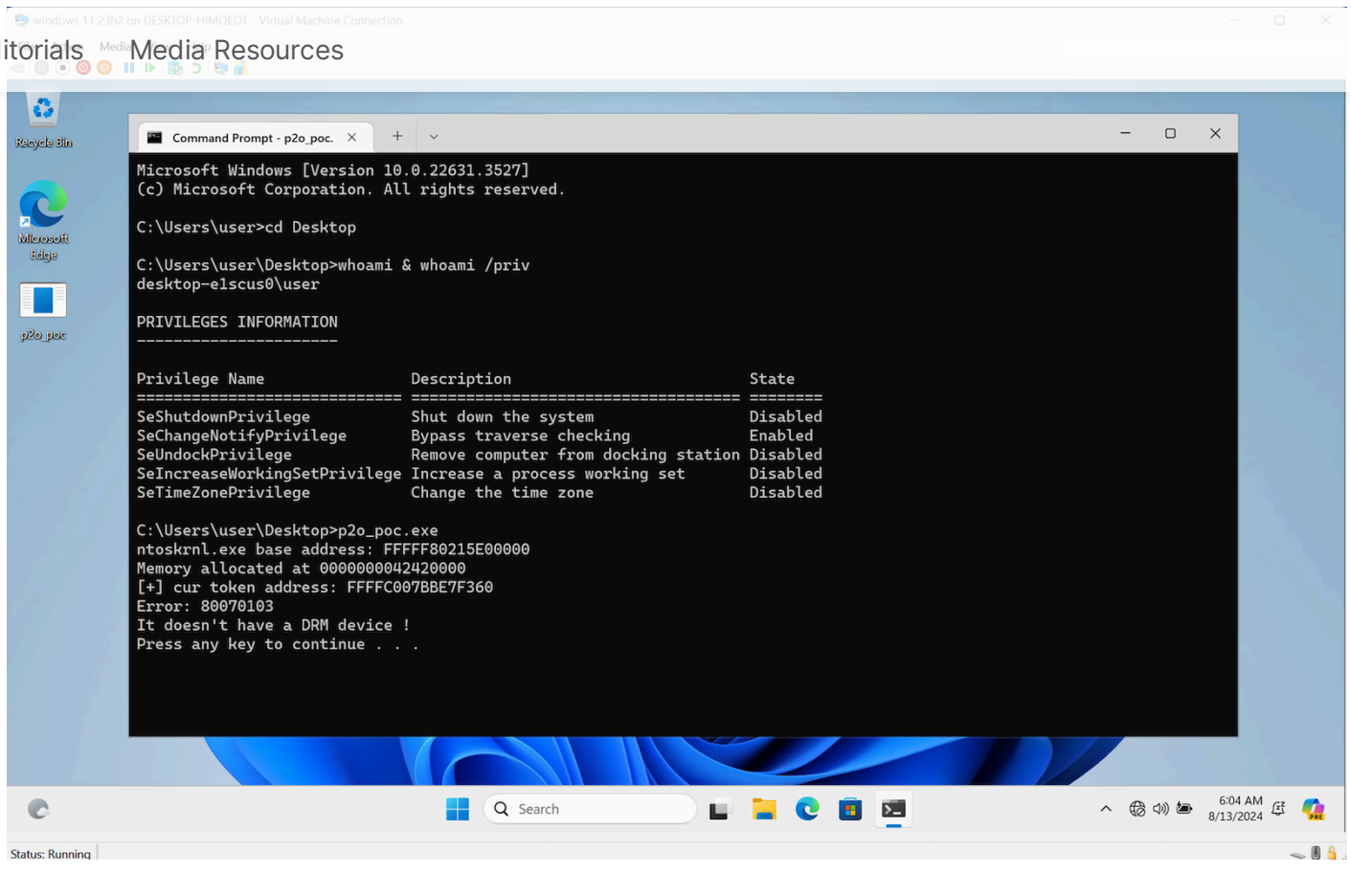
```
void __stdcall RtlSetAllBits(PRTL_BITMAP BitMapHeader)
{
  unsigned int *Buffer; // r8
  unsigned __int64 v2; // rdx

  Buffer = BitMapHeader->Buffer;
  v2 = (unsigned __int64)(4 * (((BitMapHeader->SizeOfBitMap & 0x1F) != 0) + (BitMapHeader->SizeOfBitMap >> 5))) >> 2;
  if ( v2 )
  {
    ...
    memset(Buffer, 0xFFu, 8 * (v2 >> 1));
    if ( (v2 & 1) != 0 )
      Buffer[v2 - 1] = -1;
  }
}
```

It is a very useful gadget and a legitimate function in kCFG. Additionally, it only requires controlling the first parameter `_RTL_BITMAP`.

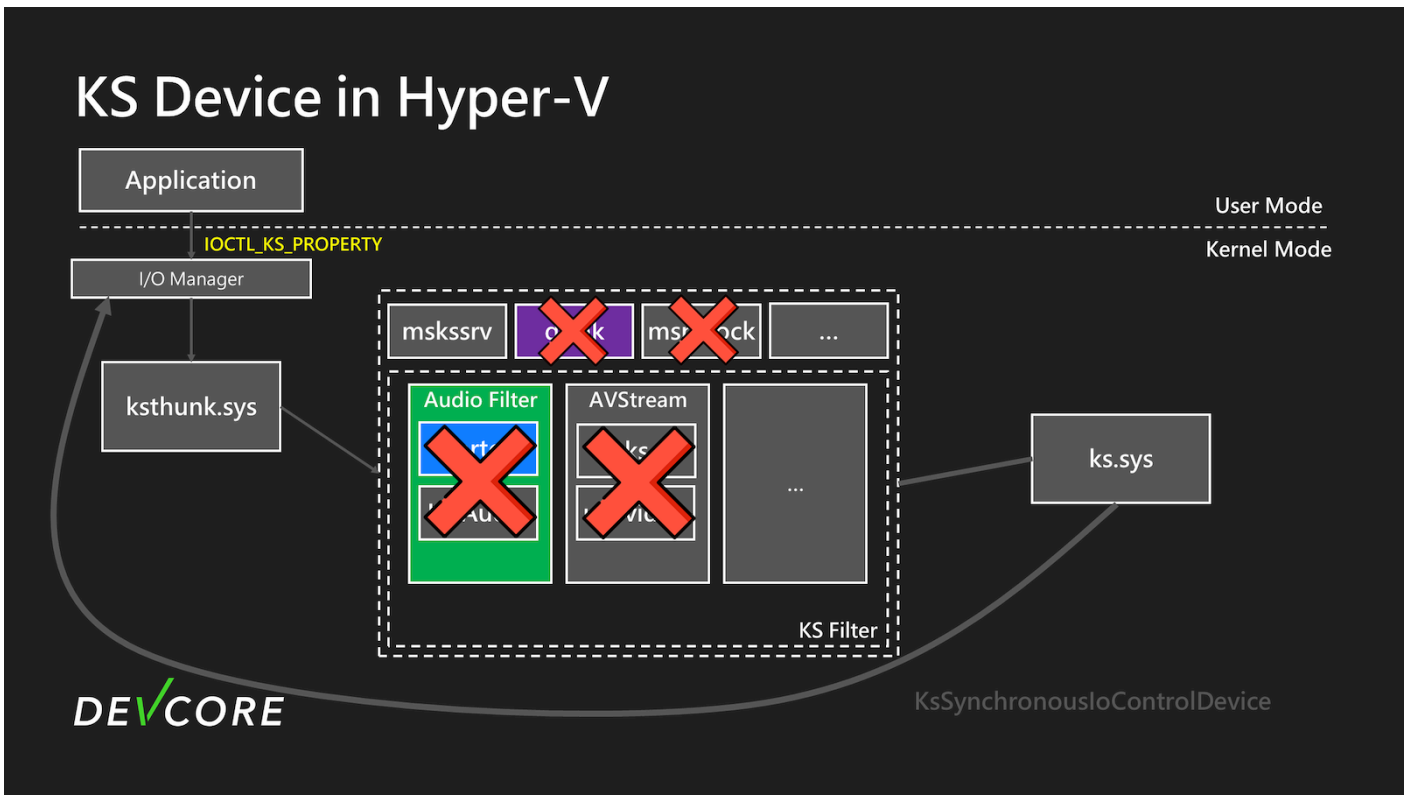
```
struct _RTL_BITMAP
{
    ULONG SizeOfBitMap;
    ULONG* Buffer;
};
```

We can assign the buffer to any address and specify the size, allowing us to set up a range of bits entirely. At this point, we are almost done. As long as `Token->Privilege` is fully set up, we can use the `Abuse Privilege` method to achieve EoP.

However, before the Pwn2Own event, we created a new Windows 11 23H2 VM on Hyper-V and ran the exploit. The result was a failure. It failed at the open device stage.



After investigation, we discovered that Hyper-V does not have an audio device by default, causing the exploit to fail.



In Hyper-V, only MSKSSRV is present by default. However, MSKSSRV does not have the `KSPROPSETID_DrmAudioStream` property, which prevents us from successfully exploiting this EoP vulnerability. Therefore, we must find other ways to trigger it or discover new vulnerabilities. We decided to review the entire process again to see if there were any other exploitable vulnerabilities.

CVE-2024-30084

After re-examining, it was found that `IOCTL_KS_PROPERTY` uses Neither I/O to transmit data, which means it uses the input buffer for data processing directly. Generally speaking, this method is not recommended as it often leads to **double fetch** issues.

```
NTSTATUS __fastcall KspPropertyHandler(
        PIRP Irp,
        unsigned int propertysetscnt,
        KSPROPERTY_SET *propertyset,
        ...){

        memmove(SystemBuffer[outlen_padding],
        CurrentStackLocation->Parameters.DeviceIoControl.Type3InputBuffer,
        InputBufferLength);
        ...
        Guid = *&SystemBuffer[outlen_padding];
        // Check if the Guid is in the property set

        ...
        if ( KsProperty_flag == KSPROPERTY_TYPE_UNSERIALIZESET )
          return UnserializePropertySet(Irp, sysbuf_, propertyset_);
        ...

}
```

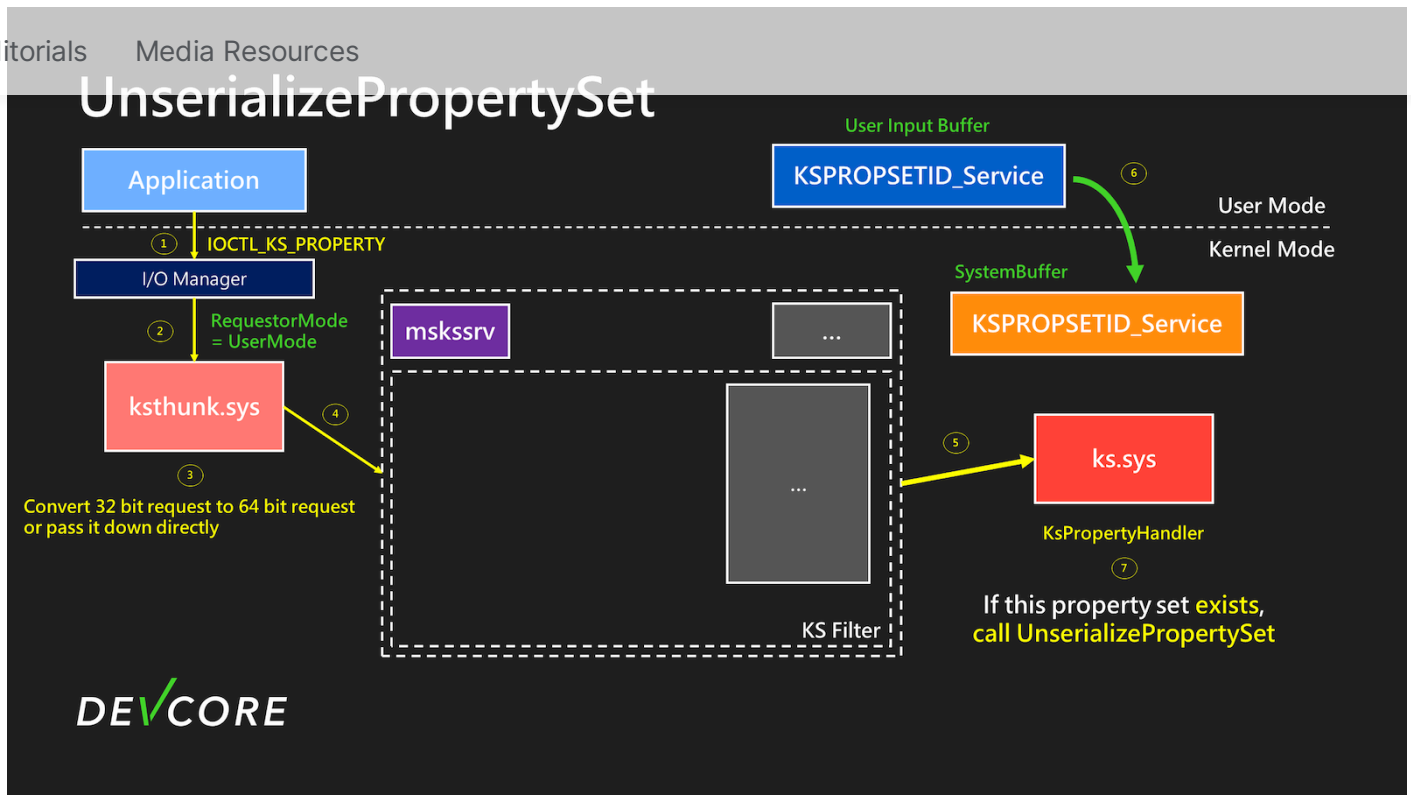
From the above figure of `KspPropertyHandler`, we can see that after the user calls IOCTL, the `Type3InputBuffer` is directly copied into a newly allocated buffer, containing the KSPROPERTY structure. This GUID in the structure is then used to check if the property set exists in the device. If it does, it will proceed to call `UnserializePropertySet`.

Let's take another look at `UnserializePropertySet`.

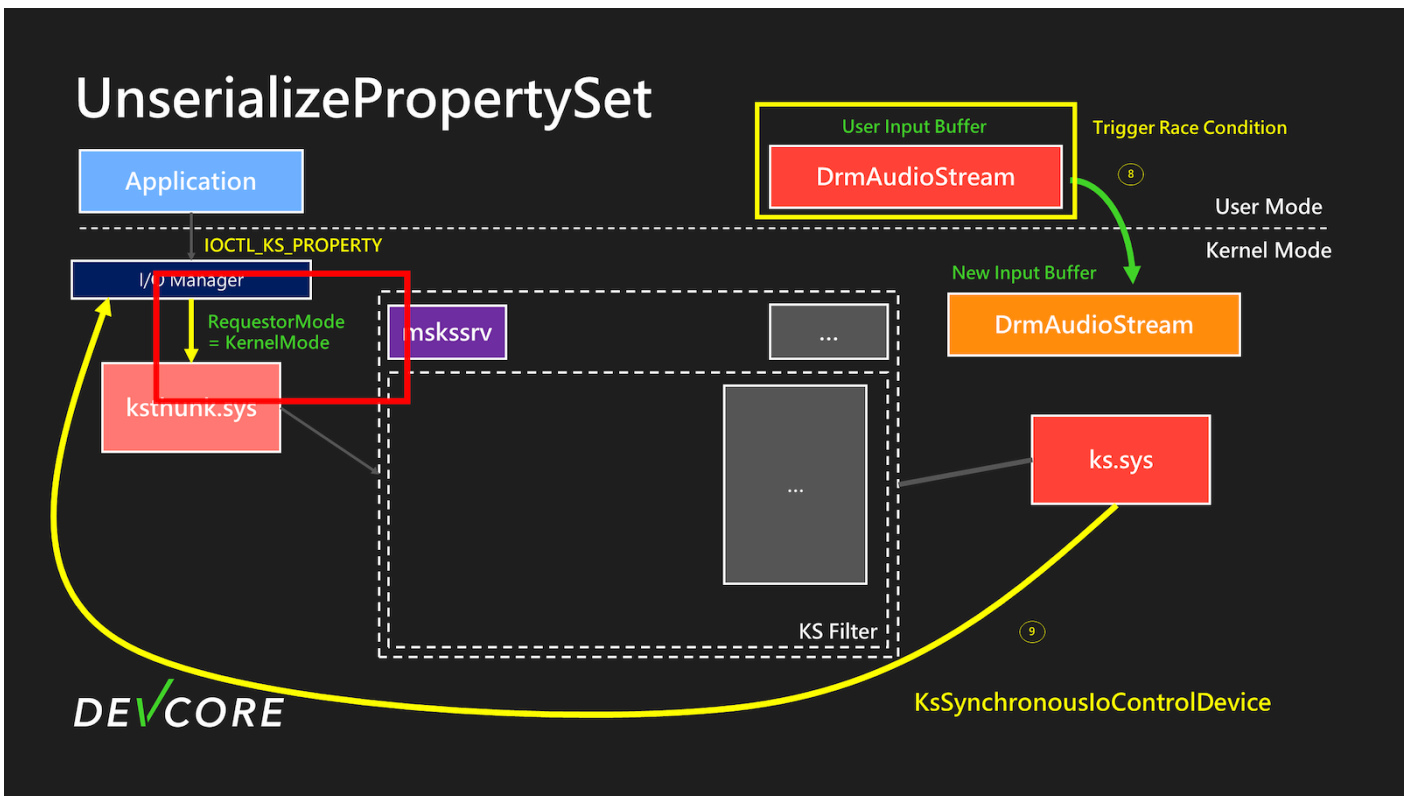
```
unsigned __int64 __fastcall UnserializePropertySet(
    PIRP irp,
    KSIDENTIFIER* UserProvideProperty,
    KSPROPERTY_SET* propertyset_)
{
    ...
    New_KsProperty_req = ExAllocatePoolWithTag(NonPagedPoolNx, InSize, 0x7070534Bu);
    ...
    memmove(New_KsProperty_req, CurrentStackLocation->Parameters.DeviceIoControl.Type3InputBuffer, InSize);
    ...
    status = KsSynchronousIoControlDevice(
            CurrentStackLocation->FileObject,
            0,
            CurrentStackLocation->Parameters.DeviceIoControl.IoControlCode,
            New_KsProperty_req,
            InSize,
            OutBuffer,
            OutSize,
            &BytesReturned);
    ...
}
```

Copy UserInput again ! ?

It **once again copies the user-provided data from `Type3InputBuffer`** as the input for the new IOCTL. Clearly, there exists a **double fetch** vulnerability here, so we have modified the entire process, as shown in the following diagram.

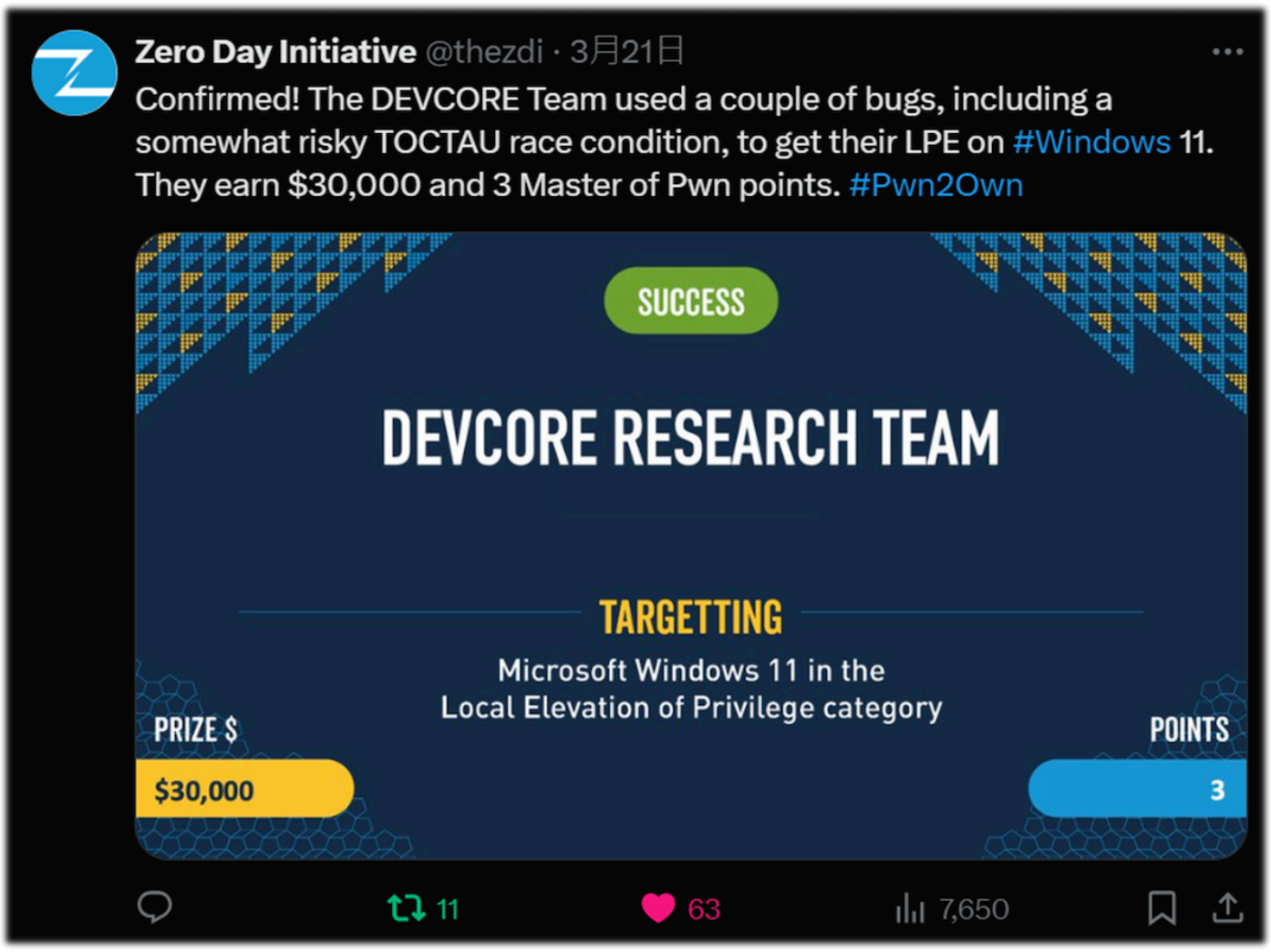


When we initially send `IOCTL_KS_PROPERTY`, we use the existing property `KSPROPSETID_Service` of MSKSSRV for subsequent operations. As shown in step 6 of the diagram, a copy of `KSPROPERTY` is first made into the `SystemBuffer`, and then this property is used to check whether it is in the support list of the KS object. Since MSKSSRV supports it, it will then call `UnserializePropertySet`.



After calling `UnserializePropertySet`, due to the double fetch vulnerability, we can change `KSPROPSETID_Service` with `KSPROPSETID_DrmAudioStream` between the check and use phases. Subsequently, `KSPROPSETID_DrmAudioStream` will be used as the request to send IOCTL, thereby triggering the CVE-2024-35250 mentioned above logic flaw. This makes the vulnerability exploitable in any environment.

Finally, we successfully compromised Microsoft Windows 11 during Pwn2Own Vancouver 2024.



After the Pwn2Own event, our investigation revealed that this vulnerability has existed since Windows 7 for nearly 20 years. Moreover, it is highly reliable and has a 100% success rate in exploitation. We strongly recommend that everyone update to the latest version as soon as possible.

To be continued

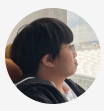
This article focuses on how we identified the vulnerabilities used in this year's Pwn2Own and the attack surface analysis of Kernel Streaming. After discovering this vulnerability, we continued our research on this attack surface and found another exploitable vulnerability along with other interesting findings.

Stay tuned for Part II, expected to be published in October of this year.

Reference

- Critically Close to Zero-Day: Exploiting Microsoft Kernel Streaming Service

- Windows Kernel Security - A Deep Dive into Two Exploits Demonstrated at Pwn2Own
- CVE-2023-29360 Analysis
- Racing Round and Round: The Little Bug That Could
- Windows Kernel Logic Bug Class: Access Mode Mismatch in IO Manager
- Hunting for Bugs in Windows Mini-Filter Drivers
- Local Privilege Escalation via the Windows I/O Manager: A Variant Finding & Collaboration



Angelboy
Senior Security Researcher
I am 👼 :)