virus BULLETIN Covering the global threat landscape

THROUGH THE LOOKING GLASS: WEBCAM INTERCEPTION AND PROTECTION IN KERNEL MODE

Ronen Slavin & Michael Maltsev
Reason Software, USA

INTRODUCTION

When we talk about digital privacy, the computer's webcam is one of the most relevant components. We all have a tiny fear that someone might be looking through our computer's camera, spying on us and watching our every move [1]. And while some of us think this scenario is restricted to the realm of movies, the reality is that malware authors and threat actors don't shy away from incorporating such capabilities into their malware arsenals [2].

Camera manufacturers protect their customers by incorporating into their devices an indicator LED that illuminates when the camera is in use. Despite the fact that a poorly designed indicator LED (implemented as a firmware function rather than as a hardware function) could allow an attacker to use the camera without turning on the indicator [3], most cameras force the illumination of the LED on a hardware level. So, are we protected by the LED indicator? Not completely. We might not notice the LED lighting up, and even if we do notice it, by that time our private snapshot is likely to have already been taken and sent to the attacker's server.

In this paper, our goal is to dive into the video capturing internals on *Windows*, and to implement a driver that protects our private snapshots before they leave the kernel mode.

USER-MODE VIDEO-CAPTURING APIs

There are several user-mode image/video-capturing APIs in *Windows*, and there is plenty of information about them on the Internet [4, 5]. The following are the most relevant APIs:

- VFW (Video for *Windows*): An obsolete technology from the Win16 era. Still available in modern *Windows* versions for legacy applications [6].
- DirectShow: Previously part of the DirectX SDK, currently part of the Platform SDK. Probably the most popular API for interacting with the camera on *Windows*.
- Media Foundation: Intended to replace DirectShow, but apparently is not there yet [7].

Other *Windows* APIs, more relevant for scanners nowadays, are: TWAIN, a multi-platform API for scanners and cameras, and WIA (Windows Image Acquisition), which provides a still image acquisition API.

ATTACK VECTORS

Let's pretend for a moment that we're the bad guys. We have gained control of a victim's computer and we can run any code on it. We would like to use his camera to get a photo or a video to use for our nefarious purposes. What are our options?

The simplest option is just to use one of the user-mode APIs mentioned previously. By default, *Windows* allows every app to access the computer's camera, with the exception of *Store* apps on *Windows 10*. The downside for the attackers is that camera access will turn on the indicator LED, giving the victim an indication that somebody is watching him.

A sneakier method is to spy on the victim when he turns on the camera himself. Patrick Wardle described a technique like this for *Mac* [8], but there's no reason the principle can't be applied to *Windows*, albeit with a slightly different implementation – e.g. we can hook into *Skype* and intercept its video-capturing APIs. Once the victim begins chatting via *Skype*, we'll get a peek at every frame that goes from the camera to *Skype*.

We can also capture the video stream in kernel mode by implementing a driver. We can even get away without having to sign it [9]. (We will see how to implement such a driver later in this paper.)

Other options probably exist but require advanced skills or some amount of luck, e.g., as we've already mentioned, it's sometimes possible to misuse a poorly designed camera to turn off the indicator LED.

VIDEO CAPTURING IN KERNEL MODE

Unlike the user-mode APIs, there's not much information on the Internet about the kernel-mode internals of video capturing. The only places I could find relevant information were the *OSR Online* forums [10] (where Tim Roberts replied to nearly every camera-related question with extremely helpful information) and the windows-camera-class-filter-driver *GitHub* repository [11], which implements a simple camera class filter driver. The following is a motivational quote from Tim Roberts about kernel streaming [12]: 'The exact ioctl interface to kernel streaming drivers is not documented, and it doesn't follow the rules used by other ioctls.' That's not very encouraging, but we're determined, so let's dive in.

From the windows-camera-class-filter-driver repository and based on other *OSR Online* posts, we learn that implementing

an upper filter camera driver is the way to go. We'll use a copy of *Microsoft*'s generic toaster filter driver [13] as a starting point. Let's add some code to print every IRP (I/O request packet) that passes through our new filter driver, then install it by using the inf file from the windows-camera-class-filter-driver repository [14]. Note that we install an upper filter driver for the 'Image' and the 'Camera' device classes. In versions of *Windows* prior to *Windows 10* version 1709, a camera device is registered under the 'Image' class, together with devices such as still-image capture devices and scanners. *Windows 10* version 1709 added the new 'Camera' class [15] specifically for cameras.

You can find the source code of the driver in our *GitHub* repository [16].

Once we install our filter driver and restart the computer (or detach and reattach our camera), we'll be able to look at our filter driver's logs. You can attach *WinDbg* to the target computer, or use *DebugView* [17] with kernel output turned on to see the logs.

I ran the built-in *Windows Camera* app on *Windows 10* with the filter driver installed, then closed it and got the log output shown in [18]. There are many repeating and uninteresting IRP calls, so to save you from the boring parts, Listing 1 shows the IRPs that I found most relevant.

Unsurprisingly, it all starts with the IRP_MJ_CREATE IRP, which is sent in order to obtain a handle to the device object. In fact, our filter driver logs two IRP_MJ_CREATE IRPs with different filenames. The first IRP's filename is '\global', and the second is '{146F1A80-4791-11D0-A5D6-28DB04C10000}\...???!?!?'. Wait, what? It surprised me at first, but it turns out that this is just how the kernel streaming communication works. If the filename starts with the KSNAME_Pin GUID, it is followed by binary data which describes the communication format (compression, resolution, etc.). Note that we see two IRP_MJ_CREATE IRPs, since a kernel streaming client program opens a file handle to the device and separate file handles to each desired camera pin (one pin in our case).

Afterwards, we see the other IRP of great importance, IOCTL_KS_PROPERTY, specifically when used with the KSPROPERTY_CONNECTION_STATE parameter. This IRP allows the camera's streaming state to be controlled.

```
major: IRP_MJ_CREATE  filename: 5C 00 67 00 6C 00 6F 00 62 00 61 00 6C 00

...

major: IRP_MJ_CREATE  filename: 7B 00 31 00 34 00 36 00 46 00 31 00 41 00 38 00 30 00 2D 00 34 00 37
00 39 00 31 00 2D 00 31 00 31 00 44 00 30 00 2D 00 41 00 35 00 44 00 36 00 2D 00 32 00 38 00 44 00 42
00 30 00 34 00 43 00 31 00 30 00 30 00 30 00 30 00 7D 00 5C 00 A0 66 87 1A CE 62 CF 11 A5 D6 28 DB 04
C1 00 00 00 00 00 00 00 00 00 00 20 B3 47 47 CE 62 CF 11 A5 D6 28 DB 04 C1 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 00 40 B0 00 00 00 00 00 00 00 00
30 2A 00 00 00 00 00 76 69 64 73 00 00 10 00 80 00 00 AA 00 38 9B 71 4D 4A 50 47 00 00 10 00 80 00 00
AA 00 38 9B 71 A0 76 2A F7 0A EB D0 11 AC E4 00 00 C0 CC 16 BA 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 8D 27 00 00 00 00 15 16 05 00 00 00 00
00 00 00 00 00 00 00 00 00 10 00 00 00 09 00 00 00 80 11 8D 00 00 00 00 00 28 00 00 00 00 05 00 00 D0
02 00 00 01 00 18 00 4D 4A 50 47 00 30 2A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

...

major: IRP_MJ_DEVICE_CONTROL  ioctl: IOCTL_KS_PROPERTY      request: Connection
       KSPROPERTY_CONNECTION_STATE   set type: KSSTATE_ACQUIRE

major: IRP_MJ_DEVICE_CONTROL  ioctl: IOCTL_KS_PROPERTY      request: Connection
       KSPROPERTY_CONNECTION_STATE   set type: KSSTATE_PAUSE

major: IRP_MJ_DEVICE_CONTROL  ioctl: IOCTL_KS_PROPERTY      request: Connection
       KSPROPERTY_CONNECTION_STATE   set type: KSSTATE_RUN

major: IRP_MJ_DEVICE_CONTROL  ioctl: IOCTL_KS_READ_STREAM

major: IRP_MJ_DEVICE_CONTROL  ioctl: IOCTL_KS_READ_STREAM

...

major: IRP_MJ_DEVICE_CONTROL  ioctl: IOCTL_KS_PROPERTY      request: Connection
       KSPROPERTY_CONNECTION_STATE   set type: KSSTATE_STOP

major: IRP_MJ_CLEANUP

major: IRP_MJ_CLEANUP

major: IRP_MJ_CLOSE

major: IRP_MJ_CLOSE
```

Listing 1: Most relevant IRPs.

The state can be KSSTATE_ACQUIRE, KSSTATE_PAUSE, KSSTATE_RUN or KSSTATE_STOP. The first request changes the state to KSSTATE_ACQUIRE, which causes the device to allocate the resources required for the streaming. The others are quite obvious.

The next interesting IRP is IOCTL_KS_READ_STREAM. While the stream is running, this IRP allows data to be read from the camera pin. This is where the actual frames with our images are flowing.

CAMERA PROTECTION DRIVER OPTIONS

Now let's play the good guys. If we want to implement a kernel driver to defend our camera from attackers, what are our options? It looks like the most obvious option is to block IRP_MJ_CREATE. This is possible, but it may have undesired consequences, since it can also affect apps which are not actually using the camera. Here, Tim Roberts explains why trying to use a camera which is already in use by a different program fails on KSSTATE_ACQUIRE, and not earlier on IRP_MJ_CREATE:

> 'At the transition to KSSTATE_ACQUIRE, the driver is supposed to fail if the hardware resources are already committed. This convention was invented during the days of Windows Media Center, because it opened an instance in a service at boot time, even if it wasn't recording anything. If drivers failed during Create, no other devices could use the camera.'

As you can see, every program can open a handle to the camera device without actually using it, and blocking it might break things. So if streaming takes place only on KSSTATE_ACQUIRE, we can block this instead.

BLOCKING KSSTATE_ACQUIRE

This is a simple solution and it works. All we need to do is to intercept IOCTL_KS_PROPERTY and to stop it when the command KSSTATE_ACQUIRE or KSSTATE_RUN is sent. We'll reject the IRP with the STATUS_INSUFFICIENT_RESOURCES error when the KSSTATE_ACQUIRE command is sent, just as the camera does when it's already being used by a different program. We will reject KSSTATE_RUN with the STATUS_ACCESS_DENIED error just in case. You can find a POC implementation in our *GitHub* repository [16].

Note that the kernel streaming ioctls use the 'neither' buffering method [19], which means that what we get is raw user-mode pointers which can be accessed only from the context of the calling process. Therefore, we handle IOCTL_KS_PROPERTY in the EvtIoInCallerContext callback [20].

Note also that rejecting KSSTATE_ACQUIRE with STATUS_INSUFFICIENT_RESOURCES will make the program think that the camera is already being used, which is something a typical capturing program should be able to handle. We can try to use another error code, such as STATUS_ACCESS_DENIED, but then the capturing program might not be able to handle the error gracefully.

ACCESSING THE CAMERA FRAMES

A downside to blocking the IOCTL_KS_PROPERTY IRP is that the application which is trying to use the camera receives an error, so it is likely that the attacker will guess that something is wrong and will try to find other ways to use the camera. We can use an alternative approach: we'll allow the program to start the camera streaming, but before each frame leaves the kernel mode, we'll replace it with one of our own. For example, we can send a black image or even an image of a dark room so the attacker will assume that the streaming is working, but will get nothing valuable.

As we already mentioned, we need to intercept IOCTL_KS_READ_STREAM in order to access/modify the camera frame. This time, we want to post-process the request, i.e. make additional changes to the data after it has been processed by the camera driver. We can do that by registering a completion routine that will be called before handing the data back to the application. We're dealing with the 'neither' buffering method, and a completion routine is not guaranteed to run in the context of the caller process, so we might not be able to use the raw user-mode pointers. Fortunately, the camera driver maps the buffers for us, so that they can be accessed from any context. We can access the header of the frame via **Irp->AssociatedIrp.SystemBuffer** and the frame data itself by mapping **Irp->MdlAddress** with **MmGetSystemAddressForMdlSafe**.

You can find a POC implementation for this technique in our *GitHub* repository [16]. Note that in my simplified implementation I merely fill the image frame buffer with zero bytes. Most camera capture programs just show a black image in this case. A more robust solution would need to send back a valid image in the correct format and resolution.

Note: in the 'Attack vectors' section we mentioned that we would see how to capture a camera stream in kernel mode. Just as we filled every frame with zero bytes in our example, we can read from the frame buffer and do anything we want with it.

A note about **ksthunk**: if you install the POC driver and try this protection method, you might find that it doesn't work. This can happen if you're running a 32-bit capturing program on 64-bit *Windows*. To solve this issue, there's one more little thing that we need to do: switch the loading order of our driver and the ksthunk helper driver. Ksthunk is an upper filter driver shipped with *Windows* and is designed to provide streaming compatibility for 32-bit programs on a 64-bit

system. After installation, our driver ends up at the bottom of the list of upper drivers, which means that it loads last, which in turn means that it's at the top of the driver stack, i.e. it's the first filter driver to receive the incoming IRPs and the last one to post-process them. My experiments showed that sometimes ksthunk sets the **Irp->MdlAddress** value to zero during its post-processing. We want ksthunk to post-process the IRPs last, so that we'll get the chance to access the captured frame at **Irp->MdlAddress**. All we need to do is to re-order the lines in the UpperFilters registry entry (see Figures 1 and 2). Refer to the driver inf file for the full path.

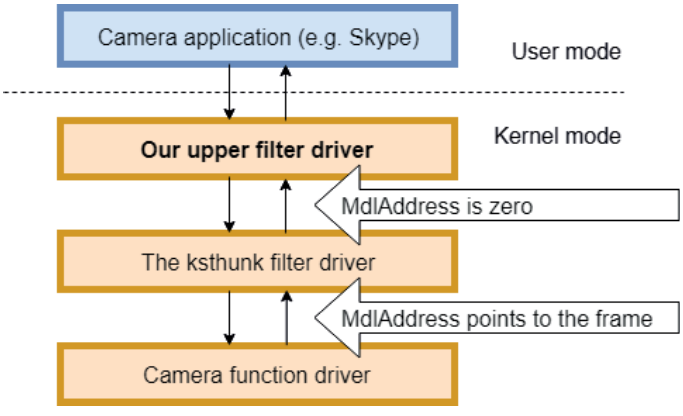

Figure 1: After driver installation.

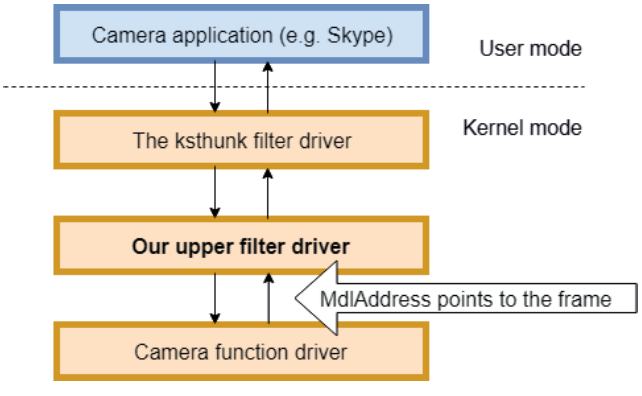

Figure 2: The desirable order.

IDENTIFYING THE BLOCKED PROCESS

What if we want to allow camera capturing for some programs but not for others? For example, it makes sense to allow *Skype* to access the camera, but not some random process which just appeared on the computer. By checking the execution context of the incoming IOCTLs, we can deduce which process sent them.

Unfortunately, *Windows 10 Anniversary Update* invalidates this assumption. Starting from *Windows 10 Anniversary Update*, when an application wants to capture from a camera, it doesn't talk to the device directly as it did before. Instead, it asks a service called Frame Server [21] to talk to the camera device on its behalf, acting as a proxy between the program and the device. The change was introduced to allow several programs to share the same camera concurrently, which in practice is allowed only for system services such as *Windows Hello*. Because of that, the execution context of camera-related IRPs will always be of an svchost process, which makes it impossible to distinguish the true capture program from kernel mode.

Identifying the relevant processes which use the camera through the Frame Server is a topic for separate research, but as a workaround, the new Frame Server can be disabled [22] by editing the registry. To disable the Frame Server globally, create the **EnableFrameServerMode** value of the type DWORD, and set it to zero, in the following branches:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
Windows Media Foundation\Platform
```

And for WOW64 programs:

```
HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\
Microsoft\Windows Media Foundation\Platform
```

It can also be disabled for selected devices [21]:

Create the **EnableDshowRedirection** value of the type DWORD and set it to zero, in the following branch:

```
HKEY_LOCAL_MACHINE\SYSTEM\
CurrentControlSet\Enum\
USB\<DeviceVID&PID>\<DeviceInstance>\Device
Parameters
```

And for selected programs:

Create the **Application** value of the type SZ and set it to the process file name (e.g. Skype.exe), in the following branch:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\OEM\
DshowBridge\<number>
```

And for WOW64 programs:

```
HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\
Microsoft\OEM\DshowBridge\<number>
```

Note that you must verify that the process that is running is not being tampered with. For example, if you trust *Skype* blindly, an attacker could inject malicious code into *Skype* to bypass the protection, or wait for the user to start using *Skype* and capture the video frames in user mode.

CONCLUSION

In this paper, we have presented possible attack vectors that involve the computer camera and our privacy, and possible ways to defend ourselves by implementing a kernel driver. Note that a properly implemented kernel driver should prevent any attempt at a user-mode attack. If the attacker is

running malicious code in kernel mode, our defending kernel driver may help if the attacker is not aware of it, but it can most certainly be bypassed.

ACKNOWLEDGEMENT

Thanks to Tim Roberts for his help in reviewing this paper.

REFERENCES

[1] http://www.abc.net.au/triplej/programs/hack/webcam-hackers-catch-man-wanking-demand-ransom/7668434.

[2] https://wikileaks.org/spyfiles/document/hackingteam/31_remote-control-system-v5-1/31_remote-control-system-v5-1.pdf.

[3] https://blog.erratasec.com/2013/12/how-to-disable-webcam-light-on-windows.html#.W5vPKoFtnHw.

[4] http://www.drdobbs.com/windows/apis-for-image-capture-applications/240156834.

[5] https://stackoverflow.com/questions/1627448/virtual-webcam-driver/1627703#1627703.

[6] http://alax.info/blog/1633.

[7] https://stackoverflow.com/questions/33539683/what-is-the-status-of-microsoft-media-foundation/33555619#33555619.

[8] https://speakerdeck.com/patrickwardle/virusbulletin-2016-getting-duped-piggybacking-on-webcam-streams-for-surreptitious-recordings.

[9] https://github.com/hfiref0x/TDL.

[10] http://www.osronline.com/page.cfm?name=ListServer.

[11] https://github.com/flowerinthenight/windows-camera-class-filter-driver.

[12] http://www.osronline.com/showthread.cfm?link=255336.

[13] https://github.com/Microsoft/Windows-driver-samples/tree/master/general/toaster/toastDrv/kmdf/filter/generic.

[14] https://github.com/flowerinthenight/windows-camera-class-filter-driver/blob/master/ccfltr/ccfltr.inf.

[15] https://docs.microsoft.com/en-us/windows-hardware/drivers/stream/camera-driver-inf-file-class-setting.

[16] https://github.com/ReasonSoftware.

[17] https://docs.microsoft.com/en-us/sysinternals/downloads/debugview.

[18] https://gist.github.com/RaMMicHaeL/851fe0ceb70632cc5587c2f50bd0893d.

[19] https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/using-neither-buffered-nor-direct-i-o.

[20] https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/content/wdfdevice/nc-wdfdevice-evt_wdf_io_in_caller_context.

[21] https://docs.microsoft.com/en-us/windows-hardware/drivers/stream/dshow-bridge-implementation-guidance-for-usb-video-class-devices.

[22] https://lifehacker.com/windows-10-anniversary-update-broke-millions-of-webcams-1785545990.

Editor: Martijn Grooten

Head of Testing: Peter Karsai

Security Test Engineers: Scott James, Tony Oliveira, Adrian Luca, Ionuţ Răileanu, Chris Stock

Sales Executive: Allison Sketchley

Editorial Assistant: Helen Martin

Developer: Lian Sebe

Tel: +44 (0)1235 555139 Email: editor@virusbulletin.com

Web: https://www.virusbulletin.com/