TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Jaanus Kääp 182507IVCM

HYPER-V VMBUS BASED TRAFFIC INTERCEPTION AND FUZZING

Master's thesis

Supervisor: Sille Laks
MSc

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Jaanus Kääp 182507IVCM

HYPER-V VMBUS PÕHISE ANDMELIIKLUSE INFOPÜÜK JA HÄGUSTUS

Magistritöö

Juhendaja: Sille Laks
MSc

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature, and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Jaanus Kääp

07.12.2020

Abstract

This thesis is written in English and is 78 pages long, including 6 chapters, 4 figures, and 10 tables.

The thesis provides an overview of Hyper-V virtualization software. The thesis is focused on the internal workings of one of Hyper-V virtualization software's main components called VMBus.

The thesis describes the general working logic of the VMBus based communication, several undocumented internal kernel functions and data structures as well as their usage. Based on the reverse engineering of those kernel components that has been performed by the author, the thesis describes internal workings of VMBus communication pathway, from the perspective that was necessary for developing the tools for monitoring, intercepting, and modifying data traffic moving through over VMBus.

The goal of the thesis is to create a good open source knowledge base and documentation for security researchers in order to simplify the beginning of Hyper-V vulnerability research targeting VMBus.

The thesis includes and analysis the most efficient methods that can be used for developing monitoring, interception, and fuzzing tools as well as a documented description of the tools by the author of the thesis. All tools developed are developed by the author of this work and will be publicly available under MIT licence from https://github.com/JaanusKaap/ThesisMaterials.

Annotatsioon

Hyper-V VMBus põhise andmeliikluse infopüük ja hägustus

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 78 leheküljel, 6 peatükki, 4 joonist ja 10 tabelit.

Lõputöö kirjeldab Hyper-V virtualiseerimise tarkvara tausta ja keskendub ühe hyper-V komponendi, VMBus, sisemisele implementatsioonile. VMBus on virtualiseeritud andmesiin, mida kasutatakse partitsioonide (virtualiseeritud masinate) vaheliseks suhtluseks.

Lõputöö kirjeldab VMBus põhise suhtluse põhiloogikat, Microsofti poolt dokumenteerimata kerneli funktsioone ja andmestruktuure ning näiteid nende kasutamisest. Tuginedes VMBus kerneli komponentide pöördkonstrueerimisele, annab töö ülevaate VMBus-i sisemisest toimimisest ulatuses, mis on vajalik arendamaks välja tööriistad, mis on võimeliseid neist läbiliikuvaid andmeid monitoorima, pealt kuulata ja muutma. Lõputöö eesmärk on luua avalikult kättesaadav ja avatud lähtekoodiga teadmistepagas mis võimaldaks infoturbeuurijatel edukamalt VMBus kaudu suhtlevatest komponentidest turvavigade leidmisega algust teha.

Lõputöö tulemuseks on erinevate metoodikate analüüs ja tööriistad mille abil on võimalik VMBuspõhist suhtlust monitoorida, pealt kuulda ja hägustada ning autori poolt koostatud tööriistade kohta käiv dokumentatsioon. Kõik lõputöö käigus autori poolt arendatud tarkvaralised tööriistad on avalikult kättesaadavad MIT litsentsi alusel aadressilt https://github.com/JaanusKaap/ThesisMaterials.

List of abbreviations and terms

Hypervisor	Software that enables creation and running virtual machines that work inside single host machine as they would work in separate machines
SLAT	Second Level Address Translation
IOMMU	Input–output memory management unit
VSP	Virtualization Service Providers
VSC	Virtualization Service Consumers
MDL	Memory description list
GPADL	Guest Physical Address Descriptor List
WDF	Windows Driver Frameworks
word	Size of 2 bytes or 16 bits
dword	Size of 4 bytes or 32 bits
qword	Size of 8 bytes or 64 bits
oword	Size of 16 bytes or 128 bits
MSRC	Microsoft Security Response Center
MIT licence	The primary terms and conditions of the MIT license are to grant permissions and indemnify developers for future use. It grants any person who obtains a copy of the software and associated files the right to use, copy, modify, merge, distribute, publish, sublicense, and sell copies of the software.
OS	Operating system
OSR	Offensive Security Research
KMCL	Kernel Mode Client Library
API	Application Programming Interface
VM	Virtual Machine
SDK	Software Development Kit
Root partition	The host OS, the name “partition” in this context is used by the Microsoft but can be translated as virtual machine.

Table of Contents

List of Figures

List of Tables

1 Introduction

Hyper-V is a virtualization tool that was released by Microsoft in year 2008 and from that point on has been included in all Windows Server versions and Pro/Enterprise versions of Windows 8 and Windows 10. It allowed the creation of virtual machines (VMs) and from 2016 it also allows adding additional security features in Windows operating system itself. A special version of Hyper-V is also used in Microsoft's Azure cloud platform called Azure Hypervisor.

The above-mentioned aspects of Hyper-V make it a potentially valuable target for the attackers. In case of a successful exploitation it allows a breakout from a VM, privilege escalation, or bypassing security features. Microsoft has acknowledged this and offers the highest bug bounties for vulnerabilities found from Hyper-V[1]. This has been done to motivate security researchers as the technical knowledge needed to research vulnerabilities from hypervisors is above average professional knowledge and a large amount of effort is needed for Hyper-V vulnerability research. Not only does researching Hyper-V require additional effort and self-development but Hyper-V is at the same time less popular for server virtualization as some of its alternatives like RedHat Virtualization and VMware vSphere. This means that it would be more beneficial time wise for the researchers to work on other virtualization environments that are used more widely. Hyper-V internals documentation has also not been published and this also makes the vulnerability research harder. For example, the communication channels that are used for communicating between different parts of the Hyper-V technology are not documented or are documented only from the external developer's perspective who is developing another operating systems to support Hyper-V.

The goal of the thesis is to increase the amount of public information about Hyper-V communication channels internals focusing on the VMBus based communications. Microsoft Virtual Machine Bus (VMBus) is a mechanism within the Hyper-V architecture that enables logical communication between partitions. The VMBus works as the internal communications channel to redirect requests to virtual devices. The thesis

will investigate the internal workings of the VMBus communications and mostly how it is used by the root partition. The thesis will also attempt to find possible ways to intercept the mentioned communications and to provide tools for doing this by the researchers.

Another goal of the thesis is to produce a full set of tools, that will be based on the performed research and allow showing information about VMBus channels internal configuration values and allow to intercept, record, and fuzz the data moving through VMBus. The research will make it easier for the author and other researchers to cover the VMBus attack surface and to reproduce or analyze any findings.

In order to achieve the goals for the thesis an empirical research will be done on the present research conducted in the field. The analysis will be used for carrying out actual technical research in order to increase the knowledge about Hyper-V communication channels internals with the focus on VMBus internals and to develop full set of tools that allow showing information about VMBus channels values and allow to intercept, record, and fuzz the data moving through VMBus.

Authors contribution is the reverse-engineering the parts of the kernel drivers that are used for VMBus based communication, the analysis of the overall logic of the communication and development of the techniques and tool for monitoring, intercepting and fuzzing the VMBus based traffic.

All reverse engineering described in this thesis is done on the 64 bit Windows 10 version 19041 (19041.1.amd64fre.vb_release.191206-1406).

All software and tools developed throughout this work are solely developed by the author and will be made open source with an MIT License that allows nearly unrestricted additional usage of the software and tools.

Parts of the work included in the thesis have been presented by the author at POC conference in South Korea in 2019 and at CyberShock conference in Latvia 2020.

2 Literature review

There is plenty of literature on the topic of hypervisors and Hyper-V [2][3][4][5] but there is not much literature that covers the internal workings of Hyper-V. There is some documentation provided by Microsoft for software developers on how to communicate with Hyper-V[6] and on how to communicate over VMBus via Windows API[7]. There is nearly no documentation or information about the internal workings of the VMBus and how Hyper-V internally handles it. The existing documentation is useful for starting the research but it is not enough to perform any other needed activities, such as data recording and fuzzing.

There are some published scientific and technology papers and articles that cover Hyper-V's security a bit [3][4][5] but most of them only cover setup methodology for administrators or how to secure the networks using Hyper-V. There appears to be no paper that covers the attack surface needed for VM breakout style of attacks or attacking the driver or device level communication through the Hyper-V. Those that do cover such topics in the first glance are unfortunately not an in-depth analysis and are not useful for security research. There are some examples of the existing public academic research that according to the title should cover similar topics but in reality do not cover any aspect of VMBus security research's point of view:

“A methodology for testing virtualisation security” by Scott Donaldson, Natalie Coull, and David McLuskie [8]

Very perfunctory description of testing the drivers inside the guest OS. The article claims to cover a testing virtualization environment, but in reality it covers fuzzing IOCTLs sent to the guest OS own drivers. While there exists a low probability that such testing can find issues in the host as well, it requires multiple issues in the chain. This is not how the guest-host communication is tested as it has a tendency to miss

almost all relevant issues and actual attack surface. It also covers no internal workings or anything relevant for Hyper-V research.

“Towards Testing the Software Aging Behavior of Hypervisor Hypercall Interfaces” by Lukas Beierlieb, Lukas Ifflander, Aleksandar Milenkoski, Charles F. Gonc¸alves, Nuno Antunes and Samuel Kounev [9]

The paper covers the overall logic how hypercall fuzzing could be done, it is not applicable for VMBus research as the research does not cover internals inside the hypevisor nor how debugging or any other part would work on VMBus. Hypercalls and VMBus are different things and the most important difference is that hypercalls do not work over VMBus. For researchers who are conducting research on VMBus, this paper adds no value, as the same information can already be found on some blog posts online that unlike this paper also include relevant kernel symbols.

In the other sources that are not counted as academic papers exists more information about Hyper-V internals, including information about VMBus. Most of this information can be found from the blog posts or slides from conference presentations from which some even originate from Microsoft Security Response Centre's (MSRC) team members. Most information regarding Hyper-V and VMBus at the beginning of author's research was collected from below mentioned resources as these are much more accurate, specific, and relevant to the research than information that can be found from currently published academic literature.

“Fuzzing para-virtualized devices in Hyper-V” by Microsoft Virtualization Security Team [10]

Microsoft Virtualization Security team's publication as a blog post that describes workings of the VMBus channels and how it can be fuzzed from guest OS side. It is one of the best currently published resources for anyone who is starting a VMBus based

research as it provides a good foundation about the logical implementation and relevant technical information. Unfortunately it does not cover channel handling internal logic sufficiently enough that would allow to create more generic tools for monitoring and modifying existing channel communications.

“A Dive in to Hyper-V Architecture & Vulnerabilities” by Nicolas Joly and Joe Bialek [11]

Conference presentation of research conducted by Nicolas Joly and Joe Bialek from MSRC Vulnerabilities & Mitigations team. This describes very broadly how different communication channels work inside Hyper-V controlled system. It is a relevant knowledge for the beginning of research on Hyper-V, but it does not go in depth with technical analysis nor does it provide any information nor guidelines for development of actual tools or techniques.

“Hardening Hyper-V through offensive security research” by Jordan Rabet [12]

Conference presentation of research conducted by Jordan Rabet from Microsoft Offensive Security Research (OSR) team. The research focuses on providing general overview of the VMBus main working logic and bugs that have been discovered so far. Similarly to the research on “A Dive in to Hyper-V Architecture & Vulnerabilities” by Nicolas Joly and Joe Bialek the current research also does not present in depth technical analysis or references to internals.

"Hyper-V internals” by Artur Kudyaev [13]

Hyper-V internals research by Artur Khudyaev on VMBus device stack and initialization implementation. It is a very thorough research, but unfortunately at the time of conducting my research the technical information has already aged a few years and does cover only some of the aspects needed for VMBus security research.

As can be seen from the referenced literature, there is some information about the VMBus working logic and also some information available in the documentation provided by Microsoft but simultaneously there is no provided information about suitable tools or even detailed techniques for researchers on how to monitor, intercept and fuzz VMBus traffic. This thesis will try to cover this research gap.

3 VMBus generic info

Microsoft Virtual Machine Bus (VMBus) is a mechanism within the Hyper-V architecture that enables logical communication between partitions. The VMBus works as the internal communications channel to redirect requests to virtual devices. In a simplified way - VMBus is a virtual bus that is used by the guest and root partitions to create communication channels between them. This is mostly used for access for a virtualized device that is controlled by the root partition. The root partition has drivers called Virtualization Service Providers (VSP) and they communicate with guest partition drivers called Virtualization Service Consumers (VSC). As names indicate, the VSC drivers relay data related to device communications to VSP drivers over the VMBus, and VSP drivers then handle the data. Handling the data might can refer to relaying it directly to a physical device, emulating it, or to any combination of the mentioned activities. [10]

VMBus itself is implemented as a ring buffer that is mapped to both virtual machines with the help of Hyper-V. For each data channel there exists two buffers - upstream buffer and downstream buffer. Notifications about new data channels are also implemented via Hyper-V with the help of synthetic interrupts. [10]

Because of the use of multiple VSPs and VSCs, there are also multiple channels over VMBus. The channels are called just - channels or VMBus channels and they are created by the root partition. Within VMBus exist regular channels and a special subtype of channels called pipes. Pipes are described more in depth in the next chapters.

Communication between regular channel endpoints works by using packets. One side will send the packet and the other side will handle the packet and respond to the packet if possible. The packet handling is based on callback functions and data within the packet can be included in two ways:

1. Data inside ring buffer - this is the data that is copied from the sender's partition memory to the ring buffer and then copied to the receiver's partition memory.

This means that from one side the data copying can take longer, but at the same time there is no possibility for different time-to-check to time-to-use bugs.

2. Included Guest Physical Address Descriptor List (GPADL) - this is a list that can be used to make some part of the guest partition memory available to receiver's partition. With the help of Hyper-V, some virtual memory from the sender's partition will be also mapped to the other partition. In the case of large data buffers, such method is much faster than copying the same buffer twice between ring buffer. At the same time nevertheless it makes time-to-check to time-to-use bugs possible because during the time receiver handles the data in this shared buffer, the sender can still change it. [12]

Application Programming Interface (API) functions for creating and connecting VMBus channels are partially publicly documented [7] and header files are included in driver development SDK. But outside of some functions, the rest of the internals are not public and the documentation provides a warning “Some information relates to pre-released product which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.” [7]. For example it means that all the structures used by the VMBus implementations are not available in the documentation or in the symbols server.

Authors contribution is the reverse-engineering the parts of the kernel drivers that are used for VMBus based communication, the analysis of the overall logic of the communication and development of the techniques and tool for monitoring, intercepting and fuzzing the VMBus based traffic

3.1 Research setup

The research environment setup was following:

- Host machine: Linux Mint (Linux Mint 18.3 Sylvia)

- Outer virtualization software: VMware Workstation 15
- Debugging OS: Windows 10 running in VMware Workstation
- Debugging software: Windbg (pykd extension)
- Reverse engineering tools: IDA Pro
- Research target OS: 64 bit Windows 10 version 19041 (19041.1.amd64fre.vb_release.191206-1406)
- Research target OS specific setup:
    - Kernel debugging enabled
    - Debugging over COM port (virtualized by Vmware)
- Crash detection: Via windbg running on “Debugging OS”

4 VMBus channels internal components

Although several of the VMBus channels's API functions are publicly documented, their internal workings inside windows kernel is not. Because of this the following chapter describes the pieces of VMBus internal logic that is found by reverse-engineering the API and the drivers that implement most of the VMBus logic.

As the actual research done covers a large amount of technical details, only parts that are needed to find ways to monitor VMBus channels or intercept and fuzz the traffic are in the scope of this thesis.

4.1 vmbkmclr.sys driver

Vmbkmclr.sys is the driver implementing most the API functionality for VMBus channel communication on host OS. It is used by the VSP drivers and its API is partially documented by Microsoft [7]. Since the documentation and also information provided by the vmbuskernelmodeclientlibapi.h header file from winsdk-10 points mostly to the functions exported by this driver, it would be reasonable to start reverse engineering from this driver.

4.1.1 Channel allocation function

Before a new channel can be activated and used it has to be allocated by the function VmbChannelAllocate. According to the documentation [14] it requires 2 input parameters and 1 output parameter. The input parameters are the parent device object and the boolean value determining whether the created channel is a server type. The output parameter points to the VMBCHANNEL structure (undocumented). While reverse engineering this part, it can be seen that the memory allocated for VMBCHANNEL structure is with pool type NonPagedPoolNx, in size of 3072 bytes,

and with tag Vkmc. After the allocation the VmbChannelAllocate function calls out the function KmclInitializeChannel (the name is based on Windows public symbols, the function itself is not exported under any name or ordinal). This will initialize the structure by zeroing out the entire structure and then setting the following values to the structure offsets as specified in table 6 at appendix 1. If the channel is not a server type and the kernel has AccessPartitionReferenceTsc privilege then it is followed by initialization described in table 1.

Table 1. Changes made in VMBCHANNEL structure by KmclInitializeChannel if not server

Offset	**Size**	**Value**
0x9E0	word	0x101
0x9F8	qword	Pointer to the allocated work item structure (IO_WORKITEM)
0xA80	byte	0x1
0x6F	dword	0x80000
0x6FC	dword	0x2000000
0xA00	sizeof(PAGED_LOOKASIDE_LIST)	Initializes NPAGED_LOOKASIDE_LIST structure inside channel structure via ExInitializeNPagedLookasideList with following parameters: 1. Lookaside = Pointer to structure offset 0xA00 2. Allocate = 0 3. Free = 0 4. Flags = ExDefaultNonPagedPoolType | POOL_NX_ALLOCATION 5. Size = 0x20 6. Tag = ‘Vkmc’ 7. Depth = 0
0x508	qword	Allowable CPU DBC usage

When the rest of the initialization was successful the function will acquire fast mutex from WPP_MAIN_CB.DeviceQueue.32 and will add the channel into the double linked list of channels. The first channel is referenced by the WPP_MAIN_CB.DeviceQueue.DeviceListHead (vmbkmclr!KmclChannelList) and the double link structure locates in the channel structure at offset 0x7A0.

4.1.2 Channel initializations function

After the Channel is allocated, there are multiple other functions that can be used to configure channel additionally. Table 7 at appendix 2 describes some of these functions by showing what changes each of these cause in VMBCHANNEL structure.

4.1.3 Channel enabling - VmbChannelEnable function

After the channel has been set up it is in a disabled state by default. In order to enable the channel, the VmbChannelEnable function has to be called out. According to the comments in Windows kernel software development kit (SDK) header file “On the host, this function offers a channel to the guest. On the guest, this function accepts an existing offer or waits for such an offer to arrive. In either case, VmbChannelEnable does not wait until the opposite endpoint offers/opens the channel and returns immediately. At this point, the channel is enabled but not open. When the host offers a channel that the guest is waiting on, or the guest decides to open an existing channel offer, kernel mode client library (KMCL) will invoke the \ref EvtChannelOpened callback.” [15]

As the entire VmbChannelEnable function is a rather large one including multiple calls to different functions inside vmbkmclr driver it will not be fully described here in detail because of the complexity. Instead the following list will provide a high level overview of the inner workings and some of the objects and structures used:

1. Writes log messages via WmiTraceMessage method.

2. Verifies that channel GUID is zero GUID.

3. Unless queue management is suppressed (in case of pipe channels for example), the “in queue” is initialized via InInitializeQueue function and “out queue” is initialized via OutInitializeQueue function. These functions are described in the next chapters.

4. The function uses FAST_MUTEX structure for synchronization referenced by the pointer at channel structure at offset 0x7F8.

5. If all checks are successful then new channel is offered to guest OS with function KmclpServerOfferChannel. These functions are described on separate chapter.

4.1.4 Channel enabling - InInitializeQueue function

InInitializeQueue function initializes the queue that is being used for incoming messages. This is not used in case of pipe subtype of channels. The changes made to channel structure by this function are described in table 8 at appendix 3.

4.1.5 Channel enabling - OutInitializeQueue function

OutInitializeQueue function appears to initialize queue that is used for outgoing messages. This is not used in case of pipe subtype of channels. The changes made to channel structure by this function are described in table 9 at appendix 4.

4.1.6 Channel enabling - KmclpServerOfferChannel function

This function is responsible for sending the channel offer to guest OS via VMBus. The next steps will give a broad overview of the steps the function goes through to do it:

1. If the channel already has VMBus handle set on offset 0x6E0 then it will generate a string \DosDevices\VMBus\offer\ID where ID equals the numeric value of the VMBus handle pointer. If the VMBus handle is not set then the

string that will be generated is \DosDevices\VMBus\offer\GUID where GUID equals to the VM id.

2. Based on the string generated during step 1, the function will request pointer to the file and device objects with call to the IoGetDeviceObjectPointer. The access mask of the call is 0x1F01FF. File object pointer is written into channel structure to the offset 0x988 and device object pointer to offset 0x980.

3. The function puts together a 176 byte buffer that contains a request and sends it to the VMBus via KmclpSynchronousIoControl function. This in turn uses IoBuildDeviceIoControlRequest function along with the control code 0x3EC01C to send it to the device (RootVMBus) written to channel structure at offset 0x980. The KmclpSynchronousIoControl function is synchronous and waits for the result before returning a value.

4. If previous steps have been successful then the function VmbusSendInterfaceQuery is called out. This will generate an IRP with major function IRP_MJ_PNP and a minor function IRP_MN_QUERY_INTERFACE to query interface for the communication from the RootVMBus device driver.

5. After the interface query has been successful, a call to KmclpInitializeVmbusConnection will be made. This function allocates IRP based on stack size of the RootVMBus device's stack size and writes its pointer to channel structure at the offset 0x9C0. After that there is a function pointer taken from channel structure at offset 0x900 (usually containing pointer to function vmbusr!BusChGetVmName) and called with following parameters:

    1. Qword taken from channel structure at offset 0x848.

    2. 0

    3. 0

    4. Pointer to channel structure offset 0x9B0.

    After the call to KmclpInitializeVmbusConnection has been successful, the function KmclpEnablePerformanceCounters is called to enable performance

counters. This fills some of the values in the channel structure, described in table 2.

6. When VMBus connection is initialized and guest VM is already waiting for the channel, then the function KmclpServerOpenChannel will be called. The function sets up the last pieces of the channel for it to function correctly and will be covered in depth in the next chapter.

Table 2. Changes made in VMBCHANNEL structure by KmclpEnablePerformanceCounters

Offset	**Size**	**Value**
0x7B2	word	Length of the instance name
0x7B8	qword	Pointer to the instance name
0x7C0	dword	Performance counter ID (incremental)

4.1.7 Channel enabling - KmclpServerOpenChannel function

This is the function that opens the channel from host side. It is triggered by the function KmclpServerOfferChannel in case the guest was already waiting for a particular channel or by function KmclpWaitForActionWorkerRoutine in case the channel was created from host side before it was requested by the guest. In both cases the function does the same thing and sets up the last actions for communication to begin. The overall logic how the function works:

1. Function pointer is taken from the channel structure at offset 0x868 and called out with following parameters:

    1. Qword from channel structure at offset 0x848.

    2. Pointer to be allocated a local 64bit integer variable (certainly OUT type).

    This function pointer is currently always pointing to vmbusr! BusFdoOpenChannel function.

2. Function pointer is taken from channel structure at offset 0x898 and called out with following parameters:

    1. Qword from channel structure at offset 0x848.

    2. Integer received from function vmbusr!BusFdoOpenChannel (second parameter).

    3. 2

    4. Pointer to MDL structure (certainly OUT type).

    This function pointer is by default pointing to vmbusr!BusChMapGpadlView function.

3. The MDL received back from vmbusr!BusChMapGpadlView is then locked by call to MmMapLockedPagesSpecifyCache with parameters that lock it to kernel access mode, MmCached cache type and, ExDefaultMdlProtection | 0x40000010 priority.

4. Calls PkInitializeRingBuffer that performs the following actions:

    1. Maps existing buffers to additional locations.

    2. Initializes ring buffer and its controls by using the function PkInitializeDoubleMappedRingBuffer.

5. Sets interrupt mask to skip count in channel structure at offset 0xD8 to point to channel structure at offset 0x1E8.

6. Calls function InOpenChannel that performs the following actions:

    1. Function pointer is taken from the channel structure at offset 0x908 and called with 2 parameters:

        1. Qword from channel structure at offset 0x848.

        2. Pointer to local 64bit integer variable (certainly OUT type).

This function pointer is currently always pointing to vmbusr! BusChGetLockChildPagesParams function.

2. Allocates memory block via ExAllocatePoolWithTag. The size is assigned from the last function call's second parameter and the tag is ‘Vkmc’.

3. The function pointer is taken from the channel structure at offset 0x8E0 and called with 3 parameters:

    1. Qword from channel structure at offset 0x848.

    2. Qword from channel structure at offset 0xB40.

    3. Pointer to channel structure at offset 0x590.

    This function pointer is by default pointing to vmbusr! BusChCreateAwWorkItem function.

4. Calls function InpReacquirePacketAllocationResources

4.1.8 Packet allocation

In order to send data over a regular channel, there is a special type of data structures used - VMBPACKET (undocumented). This data structure is created using the function VmbPacketAllocate that is partially documented by the header file and in Microsoft's documentation [8]. Even by using this information, the internal structure of the packet remains unknown. In order to understand its usage better the package needs to be reversed to some extent. After investigating the disassembled VmbPacketAllocate function, it is revealed that the buffer for the structure is allocated from channels lookaside list which is located in channel structure at offset 0x140. After the buffer is returned, it is filled as specified in table 3.

Table 3. Changes made in VMBPACKET structure by VmbPacketAllocate

Offset	**Size**	**Value**
0x9C	dword	Dword value from channel structure at offset 0x608
0	qword	Pointer to the channel structure
0xC	dword	0
0x10	qword	0
0x18	qword	0
0x2E	byte	0
0x60	qword	Pointer to packet structure offset 0xE0
0x68	dword	Dword value from channel structure at offset 0x608
0x90	dword	1
0x50	qword	0
0x8	dword	0
0x2D	byte	0
0x18	qword	Pointer to packet structure offset 0xE0 + dword value from channel structure at offset 0x608
0x2E	byte	8
0x48	qword	Pointer to function VmbPacketFree

Along with VmbPacketAllocate there is also a function named VmbPacketInitialize which is meant for use in situations where memory block that will be written to the structure is provided by the caller. VmbPacketInitialize function performs the same initialization as VmbPacketAllocate.

4.1.9 Packet initializations

After the packet has been allocated, there are multiple other functions that can be used for additional packet configuration. Table 4 describes some of the configurations by showing what changes each of these causes to the VMBPACKET structure.

Table 4. Changes made in VMBPACKET structure by set functions

VmbPacketSetCompletionRoutine		
Offset	**Size**	**Value**
0x48	qword	Pointer to completion callback function
VmbPacketSetCompletionRoutineEx		
Offset	**Size**	**Value**
0x50	qword	Callback context
0x58	qword	Pointer to compleation callback function
VmbPacketSetPointer		
Offset	**Size**	**Value**
0x10	qword	Pointer to context

4.1.10 Packet sending functions

There are multiple functions that allow sending packets over a channel. Their main logic is the same, but there are some differences that are described in the list below:

1. VmbPacketSend - most simplistic function, allows sending data in a packet buffer and/or external data (not copying over the ring buffer but shared directly).

2. VmbPacketSendWithExternalMdl - allows sending data in a packet buffer and/or external data like VmbPacketSend but additionally allows to specify MDL offset and MDL length. [16]

3. VmbPacketSendWithExternalPfns - allows sending data in a packet buffer and/or external data like VmbPacketSend but additionally allows to send a array of PFNs (Page Frame Numbers, effectively Physical addresses) instead of MDL-s. [17]

4. VmbChannelSendSynchronousRequest - allows sending data in a packet buffer and/or external data like VmbPacketSend, but waits for the response and returns completion packet directly. [18]

Because all the functions above are extended versions of the VmbPacketSend, then only this function needs to be reversed in order to understand the sending logic.

In the essence VmbPacketSend function is rather small and simple but its sub-functions are much more complicated and will be described separately. The main logic of the VmbPacketSend function as follows:

1. Packet structure for sending with function OutSetupGpaDirectPacket is set. The changes made in the structure are described in table 5.

2. In case a VMBUS_CHANNEL_FORMAT_FLAG_PAGED_BUFFER is set, then the function OutCopyAndSendPacket is called for sending the data in case paged memory can be handled. Otherwise the function OutSendPacket is called.

As OutCopyAndSendPacket just copies memory to unpaged location and then calls out the OutSendPacket function, only OutSendPacket is described in more detail

Table 5. Changes made in VMBPACKET structure by OutSetupGpaDirectPacket

Offset	Size	Value
0x38	dword	0
0x30	qword	Pointer to external data MDL structure
0x3C	dword	0
0x2D	byte	1
0x40	qword	0
0x2E	byte	1 if byte from packet structure at offset 0x2E has any bit except least significant set or 0x2 flag is set. Otherwise value is 0

4.1.11 Packet sending - OutSendPacket function

OutSendPacket function itself is a large and complicated function and it has a lot of dependencies. Because of the size and the high number of dependencies it was not reverse engineered fully for this thesis, but only analyzed as much as needed in order to understand its main logic and how packets are relayed. The list below will describe the most important parts of the function and its sub-functions. Not all functions nor subfunctions will be executed in all situations.

1. The function sets the IRQL to DISPATCH_LEVEL and acquires a spin lock via KeAcquireSpinLockRaiseToDpc function.

2. The function calls OutpEnqueuePollingDpc that queues a DPC for execution via KeInsertQueueDpc function. The KDPC pointer locates itself within the channel structure at offset 0x308. On success, the dword value in channel structure at offset 0x348 is incremented.

3. The function calls OutpPreparePacketForIsolation that will isolate the data buffer in order to get bounce buffers and create MDLs. The bounce buffer

location is written to packet structure offset 0x40 and MDL pointer is in the same structure and offset 0x30.

4. The function registers a work item with IoQueueWorkItem for a routine OutPacketAddMoreBounceWorkerRoutine and sends the packet to the queue for later use with function OutpEnqueuePacket.

5. In case a packet is not queued for a later sending, then after multiple checks, verifications, and additional setup, the packet is going to be sent along with one of the following functions (depending on the packet and overall setup): PkSendPacketSimple, PkSendPacketGpaDirectListm or PkSendPacketGpaDirect.

4.2 Vmbusr.sys driver

Vmbusr driver manages two important aspects of the VMBus communication. First it manages some higher and lower level core functionalities that are needed for VMBUS to work at all. Second and more important for this thesis specifically, it manages the pipe type of channels. As the pipe channels are read and written using typical NtReadFile and NtWriteFile functions (equivalent of a regular ReadFile and WriteFile), first it must be clarified what stacktraces such functions have.

After starting with NtReadFile and tracing through the entire call chain it can be concluded that the call to function vmbusr!PkGetReceiveBuffer is the deepest one. The entire call stack at that point is shown on following figure 1.

```
0: kd> u rip L1
vmbusr!PkGetReceiveBuffer:
fffff806`25392024 48895c2408      mov     qword ptr [rsp+8],rbx
0: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffaa82`35a29708 fffff806`25391cce     vmbusr!PkGetReceiveBuffer
01 ffffaa82`35a29710 fffff806`25391c56     vmbusr!PipeTryReadSingle+0x5e
02 ffffaa82`35a297a0 fffff806`253916b1     vmbusr!PipeTryRead+0x56
03 ffffaa82`35a297e0 fffff806`253915ef     vmbusr!PipeRead+0xb1
04 ffffaa82`35a29840 fffff806`229da977     vmbusr!RootFileReadWritePreprocess+0x6f
05 (Inline Function) --------`--------     Wdf01000!PreprocessIrp+0x2e [minkernel\wdf
06 (Inline Function) --------`--------     Wdf01000!DispatchWorker+0x17b [minkernel\w
07 (Inline Function) --------`--------     Wdf01000!FxDevice::Dispatch+0x199 [minkerr
08 ffffaa82`35a29870 fffff806`1ded1f35     Wdf01000!FxDevice::DispatchWithLock+0x267
09 ffffaa82`35a298d0 fffff806`1e2a6fb8     nt!IofCallDriver+0x55
0a ffffaa82`35a29910 fffff806`1e2aa1f9     nt!IopSynchronousServiceTail+0x1a8
0b ffffaa82`35a299b0 fffff806`1e0058b5     nt!NtReadFile+0x599
0c ffffaa82`35a29a90 00007ffa`b984be84     nt!KiSystemServiceCopyEnd+0x25
0d 00000014`549ff2e8 00007ffa`b72f8ae0     ntdll!NtReadFile+0x14
0e 00000014`549ff2f0 00000153`e4780000     0x00007ffa`b72f8ae0
0f 00000014`549ff2f8 00000153`e4ae34a8     0x00000153`e4780000
10 00000014`549ff300 00000153`e4d7a4e0     0x00000153`e4ae34a8
11 00000014`549ff308 00007ffa`b97efca0     0x00000153`e4d7a4e0
12 00000014`549ff310 00007ffa`b97efc3e     ntdll!TppIopValidateIo+0x18
13 00000014`549ff340 00007ffa`931ecd83     ntdll!TpStartAsyncIoOperation+0x2e
14 00000014`549ff370 00000000`00000000     0x00007ffa`931ecd83
```

Figure 1. Windbg output for pipe read operation stacktrace.

It is clear that after the userland process (vmwp.exe) calls NtReadFile and the WDF does its filtering and dispatching, the following functions are executed in vmbusr driver:

1. RootFileReadWritePreprocess

2. PipeRead
3. PipeTryRead
4. PipeTryReadSingle
5. PkGetReceiveBuffer

In next sub-chapters these functions are shortly analysed along with the input parameter types as these are important for monitoring and tracking the functions.

For NtWriteFile the logic is similar but when tracing the call chain, the PkGetSendBufferEx function is the deepest meaningful function called out when triggering NtWriteFile to the pipe channel. Overall the deepest is the PkpValidatePointer function but the function just verifies the pointer.

```
1: kd> u rip L1
vmbusr!PkGetSendBufferEx:
fffff806`2539257c 488bc4          mov     rax,rsp
1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffffaa82`37829668 fffff806`253924c0     vmbusr!PkGetSendBufferEx
01 ffffaa82`37829670 fffff806`2539240f     vmbusr!PkGetSendBuffer+0x34
02 ffffaa82`378296c0 fffff806`25392199     vmbusr!PipeTryWriteDeferred+0x73
03 ffffaa82`37829710 fffff806`25391802     vmbusr!PipeTryWriteIrp+0xb9
04 ffffaa82`378297c0 fffff806`253915be     vmbusr!PipeWrite+0xe2
05 ffffaa82`37829820 fffff806`229da977     vmbusr!RootFileReadWritePreprocess+0x3e
06 (Inline Function) --------`--------     Wdf01000!PreprocessIrp+0x2e [minkernel\wd
07 (Inline Function) --------`--------     Wdf01000!DispatchWorker+0x17b [minkernel\
08 (Inline Function) --------`--------     Wdf01000!FxDevice::Dispatch+0x199 [minker
09 ffffaa82`37829850 fffff806`1ded1f35     Wdf01000!FxDevice::DispatchWithLock+0x267
0a ffffaa82`378298b0 fffff806`1e2a6fb8     nt!IofCallDriver+0x55
0b ffffaa82`378298f0 fffff806`1e296def     nt!IopSynchronousServiceTail+0x1a8
0c ffffaa82`37829990 fffff806`1e0058b5     nt!NtWriteFile+0x66f
0d ffffaa82`37829a90 00007ffa`b984bec4     nt!KiSystemServiceCopyEnd+0x25
0e 00000014`54bfec18 00007ffa`b72f867a     ntdll!NtWriteFile+0x14
0f 00000014`54bfec20 00007ffa`81f9bdc6     KERNELBASE!WriteFile+0xfa
10 00000014`54bfec90 00000000`00000000     vmicvdev+0x1bdc6
```

Figure 2. Windbg output for pipe read operation stacktrace

It is clear that after the process vmwp.exe calls WriteFile and the WDF does its filtering and dispatching, the following functions are executed in the vmbusr driver:

1. RootFileReadWritePreprocess
2. PipeWrite

3. PipeTryWriteIrp
4. PipeTryWriteDeferred
5. PkGetSendBuffer
6. PkGetSendBufferEx

4.2.1 RootFileReadWritePreprocess function

This function is called out by both NtReadFile and NtWriteFile functions. It has 2 parameters, but important for this thesis only the second one that is the pointer to the IRP structure is relevant. Based on its major function the function vmbusr!PipeRead or vmbusr!PipeWrite is called out. The function will also derive a pointer to the underlying pipe structure (undocumented) from the IRP using the following logic:

1. Gets FsContext pointer from PIRP->Tail.Overlay.CurrentStackLocation->FileObject->FsContext
2. If dword from address FsContext+0x8 is 7, then the pipe structure pointer is taken from FsContext+0x168.
3. If dword from address FsContext+0x8 is 6, then the pipe structure pointer is taken from FsContext+0x50

The pointer to pipe structure is used as a first parameter to the PipeRead/PipeWrite function that is being called out.

4.2.2 PipeRead function

The function has two parameters:

1. Pipe structure pointer.
2. IPR structure pointer.

This function maps IRP contained MDLs to virtual memory, acquires a spinlock from the pipe structure (offset 0x0), and after that calls PipeTryRead directly or queues the call via PipeQueueIrp function.

4.2.3 PipeTryRead function

The function has three parameters:

1. Pipe structure pointer.
2. IPR structure pointer.
3. Out parameter for something.

Depending on the pipe structure configuration, one of the below functions will be called:

1. If the byte in the pipe structure offset 0x111 is larger than 0, then the function PipeTryReadMultiple is called with following parameters:
    1. Pipe structure pointer.
    2. IPR structure pointer.
    3. Out parameter for a specific feature.
2. If the byte in the pipe structure offset 0x111 is 0, then the function PipeTryReadSingle is called with the following parameters:
    1. Pipe structure pointer.
    2. IPR structure pointer.
    3. 0
    4. Out parameter for a specific feature.

4.2.4 PipeTryReadMultiple function

The function has three parameters:

1. Pipe structure pointer.
2. IPR structure pointer.
3. Out parameter for a specific feature.

This function endlessly calls out the PipeTryReadSingle function until it finally returns 0 value or until read size has reached the limit specified by the pipe structure (dword value pointed by pointer+0x8 at offset 0xB8 in the pipe structure). The function PipeTryReadSingle is called out with the following parameters:

1. Pipe structure pointer.
2. IPR structure pointer.
3. 1
4. Out parameter for a specific feature.

4.2.5 PipeTryReadSingle function

The function has four parameters:

1. Pipe structure pointer.
2. IPR structure pointer.
3. Flag showing is the pipe channel with chained MDLs.
4. Out parameter for a specific feature.

This is the actual function that gets shared to buffer via PkGetReceiveBuffer and reads data to local buffer which is returned as a result to NtReadFile

4.2.6 PipeWrite functions

The function has two parameters:

1. Pipe structure pointer.

2. IPR structure pointer.

This function maps IRP contained MDLs to virtual memory, acquires a spinlock from the pipe structure (offset 0x0), and after that either calls PipeTryWriteIrp directly or queues the call via PipeQueueIrp function.

4.2.7 PipeTryWriteIrp functions

The function has 3 parameters:

1. Pipe structure pointer.
2. IPR structure pointer.
3. In and out parameter – how many bytes to write or to be written.

The function will trigger copying of necessary data to ring buffer with the help of function PipeTryWriteDeferred and if needed, then function PipeMapChainedMdl to also get connected MDLs. After that the function triggers context switches with the Hypervisor via functions HviEnterKernelAperture and HviLeaveKernelAperture.

4.2.8 PipeTryWriteDeferred function

The function has 7 parameters:

1. Pipe structure pointer.
2. Unknown integer value, always 1.
3. Out integer parameter.
4. Flag of write size limit (0x4000).
5. Out parameter, bytes that can be written.
6. Unknown out parameter.
7. Out parameter – will return pointer to the buffer to the ring buffer.

This function gets pointer to the ring buffer and makes all preparations for data to be copied over. The ring buffer pointer is received via function PkGetSendBuffer.

5 Monitoring, intercepting and fuzzing traffic

In this chapter the analysis and reverse engineering done in the previous chapters will be combined into actual knowledge on how to monitor, intercept, and fuzz VMBus based traffic. This includes the knowledge how to perform it and actual tools that can be used for this. All tools will be open sourced and released under MIT License for anyone to use, extend, and repurpose.

All debugging plugins described in this paper are written in Python scripting language by the author of this thesis – they are easier to follow than the ones written in C++. But for this, the pykd library for python and pykd extension for Windbg are also needed.

5.1 Fuzzing basics

In next subchapters there are lot of mentions of “fuzzing”. Because of this the current subchapter gives brief explanation what fuzzing is, how fuzzing loop overall works, and how fuzzing can be used in current context. It has to be kept in mind, that specifics on how to make every detail needed for fuzzing work together, depends of the overall setup as specific details for each project have to be determined by the researcher conducting the specific fuzzing project during the research.

Fuzzing means providing random, mutated or generated inputs [19] to the target with the goal of causing some errors in input parsing. Inputs in current context are data buffers sent over VMBus ring buffer or shared as external data.

The fuzzing loop is known as following:

1. Create input
2. Run target with the input
3. If no misbehaviour detected, go to step 1

4. If misbehaviour detected, store the input that caused it, then go to step 1

In current case, the “misbehaviour “ detected is either invalid read or write by the target kernel or userland process. The detection part of the loop can most easily be done by the attached debugger but there are other options such as just detecting the crash of the kernel or exit of the targeted process. More exact approach has to be selected by the researcher him- or herself based on the overall fuzzing setup.

5.2 Reading channels

First requirement to start analysing VMBus channels is to get list of the channels with additional information such as type (pipe or not), human readable name (if exists), status, handling functions (if not pipe), etc. This information could easily be taken from VMBCHANNEL if the structure would be public, but since it is not, the reverse engineering results conducted throughout previous work needs to be used. Additionally this means that in case Microsoft does some internal changes to the structure then this information has to be renewed - in most cases the changes are small and simply the new offsets has to be taken from same locations. But in the current version (Windows 10 19041, 19041.1.amd64fre.vb_release.191206-1406) the important values and offsets in VMBCHANNEL structures, based on their usefulness for reverse engineering and security testing, can be taken from table 10 at appendix 5. The usefulness is determined by how well the values in these locations can be used to determine the behaviour of the channels or identify them.

When using Windbg the pointer to the first channel can be found from vmbkmclr! KmclChannelList. Unless the pointer at that location is pointing to itself, it is pointing to the double link list structure inside VMBCHANNEL structure. In order to get the first channel structure base address, the command "?poi(vmbkmclr!KmclChannelList)-0x7A0” is required. From that all other values can be read via Windbg commands. Some examples of such commands:

- In order to get interface type GUID bytes:

    - `db (poi(vmbkmclr!KmclChannelList)-0x7A0+0x61C) L10`

- In order to find out whether the pipe flag has been set:
    - `db (poi(vmbkmclr!KmclChannelList)-0x7A0+0x640) L1`

- In order to show human readable name:
    - `dt nt!_UNICODE_STRING (poi(vmbkmclr!KmclChannelList)-0x7A0+0x7C8) L1.`

With the help of LIST_ENTRY structure at 0x7A0 offsets, the entire chain of the channels can be traversed, but to do it manually is rather time consuming and therefore in appendix 1, the python script for Windbg can be found. The script displays all the channels including their detailed information. It displays the internal information such as GUID values (it is important that the GUID values have to be read as little endians not as big endians described by RFC 4122), some configuration values and callback functions. This information is sufficient for the research.

5.3 Intercepting regular channels using a debugger

Intercepting regular channels is rather simple when using a debugger. Since showChannels.py script returns the callback functions, it is straightforward to set breakpoints for these locations. For example, in order to intercept storage related packages sent via VMBus to storvsp driver (configuration displayed on figure 3), the command to use is:

```
bp storvsp!VspPvtKmclProcessPacket
```

```
Channel at 0FFFF860808F1A010
  --Normal channel--
  Interface type: ba6163d9-04a1-4d29-b605-72e2ffb1dc7f
  Interface instance: 1eb57cdc-6d84-4e7e-9e34-51fc5ca32a6c
  VM id: 00000000-0000-0000-0000-000000000000
  Pointer: 0xFFFF860805DA08A0
  Parent Device Object: 0xFFFF8608036D4D50
  Primary channel: 0x0
  Sub channel index: 0x0
  Callbacks
    packet callback = 0xFFFFF80438C931F0 (storvsp!VspPvtKmclProcessPacket)
    packet completion callback = 0xFFFFF80438C937E0 (storvsp!VspPvtKmclProcessingComplete)
    channel opened = 0xFFFFF80438C92ED0 (storvsp!VspPvtKmclChannelOpened)
    channel close = 0xFFFFF80438C92C50 (storvsp!VspPvtKmclChannelClosed)
    channel suspended = 0xFFFFF80438C93190 (storvsp!VspPvtKmclChannelSuspend)
    channel started = 0xFFFFF80438C92F90 (storvsp!VspPvtKmclChannelStarted)
    channel post started = 0xFFFFF80437104250 (vmbkmclr!InpChannelProcessingCompleteExNoOp)
```

Figure 3. ShowChannels.py script output

The breakpoint should be triggered rather fast and since the prototype of the handler function is known [20] and shown on figure 4, the parameters can be parsed.

```
typedef
_Function_class_(EVT_VMB_CHANNEL_PROCESS_PACKET)
_IRQL_requires_max_(DISPATCH_LEVEL)
VOID
EVT_VMB_CHANNEL_PROCESS_PACKET(
    _In_ VMBCHANNEL Channel,
    _In_ VMBPACKETCOMPLETION Packet,
    _In_reads_bytes_(BufferLength) PVOID Buffer,
    _In_ UINT32 BufferLength,
    _In_ UINT32 Flags
    );

typedef EVT_VMB_CHANNEL_PROCESS_PACKET *PFN_VMB_CHANNEL_PROCESS_PACKET;
```

Figure 4. Channel packet handling function prototype

In case there is a need to verify it, the first parameter should point to the channel structure, but in most cases only the last four parameters are relevant. Third and fourth parameters give input buffer with length that was received from the ring buffer. Second parameter points to the completion packet structure VMBPACKETCOMPLETION (different than VMBPACKET) that can be useful to recover shared memory

information for situations where GPADLs were sent along with the packet. These are determined by the fifth parameter. The completion packet structure is also used when driver finishes the handling of the packet using the function VmbChannelPacketComplete. [21]

The inner logic of how the GPADL conversion to MDL is done, is implemented by function VmbChannelPacketGetExternalData [22]. There are 2 important aspects that can be taken from the function:

1. If the GPADL to MDL conversion has already been done, then the pointer to MDL is cached in the packet structure at offset 0x30.

2. If the GPADL to MDL conversion has not been done, then long chain of function calls will be made that lock the referenced pages, acquire GPA lock, etc. For most part it is not useful to reverse the entire logic because in all situations where external data is being used, it is recovered by the drivers using this function. Because of that it is easier to add a breakpoint on return of this function and recover MDL structure after that. Or if an own driver is used for interception, then VmbChannelPacketGetExternalData function itself should be called.

Based on this information it can be seen how data moves through the handler function. For example in a case of channel shown on figure 3, researcher can track ongoing requests to the specified channel with such breakpoint:

```
bp storvsp!VspPvtKmclProcessPacket ".printf \"Input packet at 0x%p
with buffer at 0x%p with size 0x%X - \", rdx, r8, r9;
.if(poi(rsp+0x28)&1 > 0){.printf \"EXTRA DATA\\n\"}.else{.printf \"NO
EXTRA DATA\\n\"};"
```

If there is a wish to allow tracking of external data received, then it can easily be implemented by using a short Python script like the one added in appendix 2.

5.4 Intercepting regular channels using a driver

While intercepting channels with a debugger is straightforward, it is not always the best solution for intercepting the channels. Every request interception means that the entire OS is stopped, debugger has to get out the necessary data over the debugging channel and only then OS will resume its work. This will create lot of overhead. Because of that it is often more reasonable to perform the interception in host kernel using a special driver. The main workflow of such interception by the driver is rather simple:

1. Driver has to have an interception function with the prototype PFN_VMB_CHANNEL_PROCESS_PACKET. This will record all required data and then jump to the actual handler (stored in step 3).

2. Find structure of the channel to intercept.

3. Store ProcessPacketCallback pointer from the channel structure.

4. Overwrite ProcessPacketCallback pointer in the structure to point the driver its own function (described at step 1).

5. Keep interception working until it will be stopped and then restore the original pointer that was overwritten in step 4.

While the logic is rather simple, there is one larger problem – since developers should not handle the channel structures directly, there is no good way to get a location of the existing channels. In debugger solution the vmbkmclr!KmclChannelList symbol was used to find pointer to the first channel. This nevertheless is not an exported value by the driver, but a simple debug symbol. So based on these findings there are at least 3 options to get the pointer location by driver:

1. Hardcode the offset of the pointer for vmbkmclr driver. This is a shorter term solution as with every OS update it is possible that the offset will change.

2. Allocate and initialize own channel. This will be added to the linked list and after that the driver can move back through the list to find other channels.

3. Find offset from one of the vmbkmclr exported function in a way that is not likely to break with every OS update.

Author of the paper has been mostly using options 1 and option 3. Option 1 is rather reasonable for a researcher because the hardcoded value has to be renewed only once a month after Microsoft's patch Tuesday and it is a rather simple thing to find using a debugger:

```
?vmbkmclr!KmclChannelList-vmbkmclr
```

Using option 3 is a little bit more efficient for other situations – for example for situations where symbols are not available or when the tool is used by others who cannot make necessary changes themselves. The author has used DllInitialize function in order to solve that problem. In DllInitialize KmclChannelList is referenced as following:

```
mov     qword ptr cs:WPP_MAIN_CB.DeviceQueue.Type, rax
```

Based on the facts above and rest of the DllInitialize function, the driver has to perform the following steps:

1. Locate DllInitialize function from vmbkmclr driver exports.

2. Locate first 3 bytes with values 0x48 0x89 and 0x05 from the given function.

3. Read the following 4 bytes as an dword value (OFFSET).

4. Calculate the location of the KmclChannelList using the following formula: Location of step 2 + 0xFFFFFFFF00000000 + OFFSET + 0x7.

5. Calculation has to take the integer overflow into consideration.

The explained method can most likely endure multiple OS updates until DllInitialize function is changed in a way that creates different machine code.

In addition it also has to be noted, that when the driver stores recorded data to the filesystem, it should be done via work queue logic as in some cases the packet handlers are called out with IRQL higher than PASSIVE_LEVEL. This means that writing onto filesystems results in kernel crash. A mitigation measure is to always copy all the buffers to work queues and its effects can be reduced by only using them where IRQL is not at PASSIVE_LEVEL.

5.5 Testing/fuzzing regular channels using a debugger

After the interception part has been clarified it is now possible to move further to testing and fuzzing parts. A handler function is problematic to trigger by using debugger, but it's possible to intercept the existing requests and to make changes in them which in the other words can be referred to as performing mutational fuzzing. The author of the paper has found a vulnerability CVE-2019-0695 [23] in hypercalls using the given method. This method is rather efficient but also contains the following problems:

1. The number of requests made is controlled by the guest VM and it is hard to make it do more. While the requests being made are valid ones and suit very well for mutations, the bandwidth is not very good.

2. Since fuzzer is modifying active request that the guest OS is relaying on, then the modification will result in guest OS crashing rather fast, either by leaving the request hanging or acting otherwise unexpectedly. Because of crashing the guest OS the environment has to be recovered often and this is rather time consuming.

While debugger cannot trigger requests to handler function in a usual way, there is still another option how similar situation could be achieved to work up to a certain point. This handler function logic is mostly the following:

1. Handler function is called out by vmbkmclr driver.

2. Handler function gets input buffer and MDL values needed.

3. Handler function does its work.

4. Handler function marks request complete by using the function VmbChannelPacketComplete.

Because of this logic and the fact that VMBPACKETCOMPLETION structure can be restored by the debugger, it is possible to perform the following fuzzing loop:

1. Debugger adds breakpoint to the beginning of a handling function.
2. Debugger lets the OS run.
3. Debugger breaks at handler function breakpoint.
4. Debugger stores thread address and all registry values.
5. Debugger adds breakpoint to VmbChannelPacketComplete function.
6. Debugger removes breakpoint added in step 1
7. Debugger lets the OS run.
8. Debugger breaks at VmbChannelPacketComplete function.
9. If the thread is the same as stored in step 4, the registry values are restored to the ones in step 4 and the input buffer and/or MDL buffer will be randomly modified.
10. Return to step 7.

This kind of approach allows the usage of the same request to actually force handler function to handle the request multiple times, destroying the downside of not being able to start requests. But this implementation might not always work because of the internals of handler functions. There are multiple reasons for it not to work in same situations but in most cases this is still a viable approach.

5.6 Testing/fuzzing regular channels using a host kernel driver

After the fuzzing is completed by the OS kernel driver, then the subsequent approach is very similar to the interception part described in chapter 5.3. The only difference is that

the driver does not record the packet buffer and MDL buffer but randomly changes them. The changes have to be recorded for crash situations. The author of this paper usually implements the procedure in following way:

1. Driver allocates some memory in the kernel.
2. Location of the allocation is sent to the kernel debugger using DbgPrint function.
3. With every fuzzing iteration the thread pointer, buffer location, MDL location, and changes made are stored on location allocate at step 1.
4. If a crash occurs, the debugger can be used to recover changes made by the driver from location relayed to debugger during step 2.

5.7 Testing/fuzzing regular channels using a guest kernel driver

Similarly to the host kernel drivers, the guest kernel driver has to first allocate the location of the existing channels from the system. The only difference is that the driver used for this is not vmbkmclr.sys as previously but vmbkmcl.sys. Most logic and functionality remains the same. The functions and variable offsets are different, but can be located using same methods as when using vmbkmclr. After the target channel has been found, the regular Vmb functions such as VmbPacketSend [24] can be used to send data to the host kernel.

In such cases, the data sent by the driver has to be self generated and there are 3 options how to generate the required data:

1. By sending random data or dumb fuzzing [19]. Data being sent is randomly generated in full. This approach rarely works.
2. by sending modified data or mutational fuzzing [19]. This approach requires that some of the traffic is previously recorded and can now be randomly modified and sent over. This approach is commonly most time effective as this can be quickly implemented, does not require a lot of reverse-engineering, but generates almost correct inputs.

3. By generating correct data or smart fuzzing [19]. Using this approach, the researcher has to first reverse engineer the protocol being sent over and then write the generator that will generate inputs based on the protocol. This is the most comprehensive way to fuzz, but it is highly time consuming at the beginning.

The author of this paper is mostly using the mutational fuzzing in order to avoid huge time consumption for the topics that are not guaranteed to give the desired results. From the other aspect smart fuzzing is a good option in case the target is highly valuable and additional time consumption is not a problem.

5.8 Intercepting pipe channels using a debugger

As previously described in chapter 4.2, the pipe channels data is received by VMWP and via regular ReadFile API call. Due to this reason, there is no handler function to add a breakpoint for. But at the same time there are couple of functions that will always be called whenever data is read from the pipe channel. One of these functions is vmbusr! PipeTryReadSingle and it includes 2 important parameters:

1. The first parameter is a pointer to pipe structure data from where at offset 0x100 the pointer to channel structure can be found.

2. The second parameter is the IRP structure pointer that contains information about read length and the buffer where data is written.

While it is straightforward to intercept PipeTryReadSingle function, the interception is happening at the moment when the data has not yet been received. The breakpoint should be put at the end of the function to handle the situation after the read operation is over. At that point, the values of original function parameter registers (rcx and rdx) have been changed. Fortunately the values are still held around on other registers so it is not necessary to have additional breakpoint at the start of function for a parameter storage. At the end of the function, the original first parameter can be found from the register rbx and second parameter from register rsi.

Additionally to the information found above it should also be noted that these read operations often return 0 bytes, and thereby these cases should be also sorted out. In order to simply display read operation results to the debugger, the following windbg command can be used for it to function correctly (0x3ac is offset from the beginning of PipeTryReadSingle to its ret opcode):

```
bp vmbusr!PipeTryReadSingle+0x3ac ".printf \"Read from channel 0x%p -
size: 0x%X\\n\", poi(rbx+0x100), poi(rsi+0x38); .if(poi(rsi+0x38) ==
0){.echo \"NO DATA\";}.else{db poi(poi(rsi+0x8)+0x18)
poi(poi(rsi+0x8)+0x18)+poi(rsi+0x38)-1;};gh;"
```

5.9 Intercepting pipe channels using a driver

Intercepting pipe channel traffic using a driver requires extra effort than intercepting regular channels. As there are no handler functions, the interception requires injection of machine code snippet that will make the code flow jump to the driver code. The location for the code injection is the end of the function PipeTryReadSingle. The main logic for this is as follows:

1. The driver implements function/s capable of filtering and recording data and recovering to the status expected by the PipeTryReadSingle function.

2. The driver locates end of the PipeTryReadSingle function and adds a conditionless jump to the function described in step 1.

3. When the CPU reaches the execution of the end of the PipeTryReadSingle function, the codeflow will switch to the function described at step 1, where data is recorded and correct values of the registers are restored before returning to the PipeTryReadSingle caller function.

5.10 Testing/fuzzing pipe channels using a debugger

Exactly like with regular channels, the debugger cannot initiate writes to the channels, but has to work on the traffic already being generated. This means that that bandwidth cannot be very high – it’s highly dependent but usually not more than couple of requests per second. In addition there are lot on reads that do not return a result. These could

potentially be used to generate random responses - filling up everything needed and setting IRP structure status and values. The author has not tried to generate random responses as this did not seem to achieve any goals that would be considered valuable.

5.11 Testing/fuzzing pipe channels using a host kernel driver

The approach is the same as with the interception driver described at chapter 5.8 and fuzzing driver at 5.5. The access to the data will be achieved in a way described in chapter 5.8 and the fuzzing logic and modification storage implemented and achieved as in chapter 5.5.

5.12 Testing/fuzzing pipe channels from guest system

Write operations to the pipe channel can be triggered from guest system with both kernel driver and for most part also by a userland program. All code that runs outside the operating system's kernel belongs to the userland, that is sometimes also referred to as user space. In both cases usual NtWriteFile/WriteFile functions can be used and input generation methods have same options as described in chapter 5.6. As already previously, the author recommends using either mutational fuzzing or smart fuzzing for pipe channels from the guest system as dumb fuzzing almost never give results on hardened targets.

5.13 Tools developed based on the research

Based on reverse engineering performed throughout chapters 4 to 4.28 and methods created for monitoring, interception, and fuzzing in chapters 5 to 5.11, the author has created a new toolset for VMBus research. The toolset is available from github repository at https://github.com/JaanusKaap/ThesisMaterials.

The toolset code is broken into smaller pieces in order to make it easily understandable and simply modifiable. The driver code is separated to different drivers and not implemented on a single one, in order to clarify the understanding of each functionality.

The following list will describe different tools developed, what is their purpose and profitability for the researchers:

- **Windbg scripts https://github.com/JaanusKaap/ThesisMaterials/scripts**

  The scripts meant to run in Windbg debugger. All scripts have been written in python (preference of the author) and therefore require pykd library for python and Windbg itself.

  - **showChannels.py**

    Displays information about the VMBus channels locations, internal configuration values, and handler functions.

  - **recordChannel.py**

    Records regular channel traffic to the debugger machine filesystem with different filtering options.

  - **recordPipeChannel.py**

    Records pipe channel traffic to the debugger machine filesystem with different filtering options.

  - **fuzzChannelOnInterception.py**

    Fuzzes traffic passing through the regular channel handler function. Can be configured with different settings and allows fuzzing of external data. Crash detection is simple and no crash analysis is performed by the script.

  - **fuzzPipeChannelOnInterception.py**

    Fuzzes traffic passing through the pipe channel during read operation. Can be configured with different settings. Crash detection not included as the crash happens in VMWP process, the fuzzing data is logged.

  - **fuzzChannelRepetition.py**

    Fuzzes traffic passing through the regular channel handler function but with additional capability to replay the VMBPACKETCOMPLETION handling as explained in chapter 5.4. Can be configured with different settings and allows fuzzing of external data. Crash detection is simple and no crash analysis will be performed by the script.

- **Drivers https://github.com/JaanusKaap/ThesisMaterials/DriversTools**
  - **VMBusChannels.sys**

    Can locate information about the existing channels and return internal information about them to userland process.
  - **VMBusIntercept.sys**

    Can intercept traffic moving through the regular VMBus channels with capability to store this data on filesystem. Stores both ring buffer and external data.
  - **VMBusFuzz.sys**

    Can fuzz traffic moving through the regular VMBus channels while storing all current fuzzing information to exported location making it possible to be found by the debugger in case of a crash. Can fuzz both ring buffer and external data.
- **Libraries https://github.com/JaanusKaap/ThesisMaterials/DriversTools**

  Dll files are good way to make it easier for any language to communicate with drivers without requiring to implement entire driver communication logic.
  - **VMBusChannels.dll**

    Contains all methods needed for communication with VMBusChannels.sys driver.
  - **VMBusIntercept.dll**

    Contains all methods needed for communication with VMBusIntercept.sys driver.
  - **VMBusFuzz.dll**

    Contains all methods needed for communication with VMBusFuzz.sys driver.
- **Tools https://github.com/JaanusKaap/ThesisMaterials/DriversTools**

  Programs that allow user to communicate with drivers

- **VMBusChannels.exe**

  Tool that can communicate with VMBusChannels.sys driver in order to display the user information about existing VMBus channels.

- **VMBusIntercept.exe**

  Tool that can communicate with VMBusIntercept.sys driver in order to set up recording of ongoing VMBus regular channel traffic including some additional filtering and setup options.

- **VMBusFuzz.exe**

  Tool that can communicate with VMBusFuzz.sys driver in order to set up fuzzing of the VMBus regular channel traffic with some configuration options.

6 Summary

In this paper, the author reverse engineered and analysed some of the internal workings of VMBus implementation in Windows 10 version 19041 (19041.1.amd64fre.vb_release.191206-1406) running on x64 architecture.

The analysis was driven by the goal to be able to create methods for creation of tools to monitor, intercept, and fuzz data moving through VMBUS between guest and host systems. The resulting knowledge, tools and methods can now be used to perform additional vulnerability research with the end-goal of finding new security vulnerabilities and bugs in kernel drivers that provide virtualization service provider functionality support and in virtual machine working processes running in the host system userland. The tools created are publicly available from github under MIT license from https://github.com/JaanusKaap/ThesisMaterials.

References

1: Microsoft, Microsoft Hyper-V Bounty Program, , https://www.microsoft.com/en-us/msrc/bounty-hyper-v?rtc=1
2: Jason Kappel, Anthony Velte, Toby Velte, Microsoft Virtualization with Hyper-V: Manage Your Datacenter with Hyper-V, Virtual PC, Virtual Server, and Application Virtualization (Network Professional's Library), 2009
3: Lei Chen, Ming Xian, Jian Liu and Huimei Wang, Research on Virtualization Security in Cloud Computing, 2020
4: Purva Vishwakarma , Sumit Kumar, Desktop Virtualization on different system using Hyper-V , 2015
5: Federico Sierra-Arriaga, Rodrigo Branco, Ben Lee, Security Issues and Challenges for Virtualization Technologies, 2020
6: Microsoft, Hypervisor Top-Level Functional Specification, 2018, https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/tlfs
7: Microsoft, vmbuskernelmodeclientlibapi.h header, 2018, https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/vmbuskernelmodeclientlibapi/
8: Scott Donaldson; Natalie Coull; David McLuskie, A methodology for testing virtualisation security, 2017
9: Lukas Beierlieb; Lukas Ifflander; Aleksandar Milenkoski; Charles F. Gonc¸alves; Nuno Antunes; Samuel Kounev, Towards Testing the Software Aging Behavior ofHypervisor Hypercall Interfaces, 2019
10: Microsoft Virtualization Security Team, Fuzzing para-virtualized devices in Hyper-V, 2019, https://msrc-blog.microsoft.com/2019/01/28/fuzzing-para-virtualized-devices-in-hyper-v/
11: Nicolas Joly; Joe Bialek , A Dive in to Hyper-V Architecture & Vulnerabilities, 2018
12: Jordan Rabet, Hardening Hyper-V throughoffensive security research, 2018
13: Gerhart X, Hyper-V internals, 2015,
14: Microsoft, FN_VMB_CHANNEL_ALLOCATE callback function (vmbuskernelmodeclientlibapi.h), 2018, https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/vmbuskernelmodeclientlibapi/nc-vmbuskernelmodeclientlibapi-fn_vmb_channel_allocate
15: Microsoft, vmbuskernelmodeclientlibapi.h header file,
16: Microsoft, https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/vmbuskernelmodeclientlibapi/nc-vmbuskernelmodeclientlibapi-fn_vmb_packet_send_with_external_mdl, 2018, https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/vmbuskernelmodeclientlibapi/nc-vmbuskernelmodeclientlibapi-fn_vmb_packet_send_with_external_mdl
17: Microsoft, FN_VMB_PACKET_SEND_WITH_EXTERNAL_PFNS callback function (vmbuskernelmodeclientlibapi.h), 2018, https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/vmbuskernelmodeclientlibapi/nc-vmbuskernelmodeclientlibapi-fn_vmb_packet_send_with_external_pfns
18: Microsoft, FN_VMB_CHANNEL_SEND_SYNCHRONOUS_REQUEST callback function (vmbuskernelmodeclientlibapi.h), 2018, https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/vmbuskernelmodeclientlibapi/nc-vmbuskernelmodeclientlibapi-fn_vmb_channel_send_synchronous_request
19: F-Secure, OUR GUIDE TO FUZZING, , https://www.f-secure.com/en/consulting/our-thinking/15-minute-guide-to-fuzzing
20: Microsoft, EVT_VMB_CHANNEL_PROCESS_PACKET callback function (vmbuskernelmodeclientlibapi.h), 2018, https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/vmbuskernelmodeclientlibapi/nc-vmbuskernelmodeclientlibapi-evt_vmb_channel_process_packet
21: Microsoft, FN_VMB_CHANNEL_PACKET_COMPLETE callback function (vmbuskernelmodeclientlibapi.h), 2018, https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/vmbuskernelmodeclientlibapi/nc-vmbuskernelmodeclientlibapi-fn_vmb_channel_packet_complete
22: Microsoft, FN_VMB_CHANNEL_PACKET_GET_EXTERNAL_DATA callback function (vmbuskernelmodeclientlibapi.h), 2018, https://docs.microsoft.com/en-us/windows-

hardware/drivers/ddi/vmbuskernelmodeclientlibapi/nc-vmbuskernelmodeclientlibapi-fn_vmb_channel_packet_get_external_data
23: Microsoft, CVE-2019-0695, , https://msrc.microsoft.com/update-guide/en-US/vulnerability/CVE-2019-0695
24: Microsoft, FN_VMB_PACKET_SEND callback function (vmbuskernelmodeclientlibapi.h), 2018, https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/vmbuskernelmodeclientlibapi/nc-vmbuskernelmodeclientlibapi-fn_vmb_packet_send

Appendix 1 – Table of KmclInitializeChannel changes

Table 6. Changes made in VMBCHANNEL structure by KmclInitializeChannel

Offset	Size	Value
0x7F8	dword	0x1
0x810	sizeof(KEVENT)	Kernel event structure for synchronizationEvent with unsignaled status
0xAE8	dword	0x1
0xB00	sizeof(KEVENT)	Kernel event structure for synchronizationEvent with unsignaled status
0x0	dword	Pointer to the structure itself
0xB38	qword	Pointer to the structure offset 0xB30
0xB30	qword	Pointer to the structure offset 0xB30
0xB28	qword	Pointer to the structure offset 0xB20
0xB20	qword	Pointer to the structure offset 0xB20
0x9F0	qword	Pointer to the structure offset 0x9E8
0x9E8	qword	Pointer to the structure offset 0x9E8
0x3F0	byte	0x1
0x4B4	dword	If server flag is set, then 0x100, otherwise 0xFFFFFFFF
0x600	byte	Server flag
0x830	byte	If the server flag is not set, then value is OR-ed with 0x1
0x6F0	dword	0xFFFFFFFF

Offset	Size	Value
0x738	qword	Pointer to function KmclpDefaultChannelOpenedEx
0x740	qword	Pointer to function KmclpDefaultChannelClosed
0x748	qword	Pointer to function KmclpDefaultChannelClosed
0x750	qword	Pointer to function KmclpDefaultChannelClosed
0x758	qword	Pointer to function KmclpDefaultChannelClosed
0x798	qword	Pointer to function KmclpDefaultChannelClosed
0xB40	qword	Pointer to the parent device object
0x7F0	qword	Pointer to the allocated work item structure (IO_WORKITEM)
0x9C8	qword	Pointer to the allocated work item structure (IO_WORKITEM)

Appendix 2 – Table of initialization functions changes

Table 7. Changes made in VMBCHANNEL structure by initialization functions

VmbChannelInitSetBounceBufferSizes		
Offset	**Size**	**Value**
0x6F4	dword	Size of bounce buffer 1 Limitation: Has to be larger then 0x40000 and page sized
0x6F8	dword	Size of bounce buffer 2 Limitation: Has to be larger then 0x40000 and page sized
0x6FC	dword	Size of bounce buffer 3 Limitation: Has to be larger then 0x40000 and page sized
VmbChannelInitSetClientContextSize		
Offset	**Size**	**Value**
0x614	dword	Size of the client context
VmbChannelInitSetFlags		
Limitations	• Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x640	byte	Flag VMBUS_CHANNEL_INIT_FLAG_IS_PIPE

VmbChannelInitSetFriendlyName		
Limitations	• Dword at structure offset 0x7D8 has to be 0 • Qword at structure offset 0x7D0 has to be 0	
Offset	**Size**	**Value**
0x7C8	sizeof(UNICODE_STRING)	UNICODE_STRING structure referencing the human readable name of the channel
VmbChannelInitSetInlinePacketContextSize		
Limitations	• Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x618	dword	Size of the inline packet context size. The actual value set is input (value + 7) & 0xFFFFFFF8 Limitation: Can only be set when dword at structure offset 0x7D8 is 0
VmbChannelInitSetMaximumExternalData		
Limitations	• Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x60C	dword	Maximum data size
0x610	dword	Maximum chain length
VmbChannelInitSetMaximumPacketCount		
Limitations	• Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x604	dword	Maximum packet count

VmbChannelInitSetMaximumPacketSize		
Limitations	• Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x608	dword	Maximum packet size
VmbChannelInitSetPrimaryChannel		
Limitations	• Dword at structure offset 0x7D8 has to be 0 • Byte at structure offset 0xAE0 has to be 0 • Byte at primary channel structure offset 0x700 has to be 0 • Byte at primary channel structure offset 0x830 can’t have third bit set	
Offset	**Size**	**Value**
0xAD8	qword	Pointer to primary channel
0xAE0	byte	1
0xB50	word	Subchannel index
VmbChannelInitSetProcessPacketCallbacks		
Limitations	• Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x708	byte	0
0x710	qword	Pointer to ProcessPacketCallback function
0x718	qword	Pointer to ProcessingCompleteCallback function

VmbChannelInitSetProcessPacketCallbacksEx		
Limitations	• Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x708	byte	0
0x720	qword	CallbackContext value that is provided with all calls to ProcessPacketCallbackEx and ProcessingCompleteCallbackEx for this channel
0x710	qword	Pointer to ProcessPacketCallbackEx function
0x718	qword	Pointer to ProcessingCompleteCallbackEx function
VmbChannelInitSetShortLifetimeThreshold		
Limitations	• Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x63C	dword	Short lifetime threshold value
VmbChannelInitSetStateChangeCallbacks		
Limitations	• Dword at structure offset 0x7D8 has to be 0 • Callbacks structure has to be version 1 or 5	
Offset	**Size**	**Value**
0x728	byte	1 Limitation: Only set when input structure version is 5
0x738	qword	Pointer to EvtChannelOpened callback
0x740	qword	Pointer to EvtChannelClosed callback
0x748	qword	Pointer to EvtChannelSuspend callback
0x750	qword	Pointer to EvtChannelStarted callback
0x758	qword	Pointer to EvtChannelPostStarted callback

VmbChannelInitSuppressQueueManagement		
Limitations	• Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x601	byte	1
0x790	qword	Pointer
VmbChannelSetIncomingPollOnCompletion		
Offset	**Size**	**Value**
0x3F0	byte	Is set 0 if flag parameter is 0, otherwise 1
VmbChannelSetIncomingProcessingAtPassive		
Offset	**Size**	**Value**
0x3F2	byte	Is set 0 if flag parameter is 0, otherwise 1
0x3F1	byte	Is set 1 if flag parameter is not 0
VmbChannelSetPointer		
Offset	**Size**	**Value**
0x838	qword	Pointer to the parameter specified location
VmbServerChannelInitSetTargetInterfaceId		
Limitations	• Byte at structure offset 0x600 can’t be 0 • Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x61C	sizeof(GUID)	Interface type GUID
0x62C	sizeof(GUID)	Interface instance GUID

VmbServerChannelInitSetTargetVtl		
Limitations	• Byte at structure offset 0x600 can’t be 0 • Dword at structure offset 0x7D8 has to be 0 • VTL value provided can’t be larger then 2	
Offset	**Size**	**Value**
0x702	byte	VTL parameter value
VmbServerChannelInitSetVmId		
Limitations	• Byte at structure offset 0x600 can’t be 0 • Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x650	sizeof(GUID)	VM id
VmbServerChannelInitSetVmbusHandle		
Limitations	• Byte at structure offset 0x600 can’t be 0 • Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x6E0	qword	VMBus handle pointer

VmbChannelInitSetFlags		
Limitations	• Byte at structure offset 0x600 can’t be 0 • Dword at structure offset 0x7D8 has to be 0	
Offset	**Size**	**Value**
0x641	byte	VMBUS_SERVER_CHANNEL_INIT_FLAG_LOOPBACK_OFFER flag
0x642	byte	VMBUS_SERVER_CHANNEL_INIT_FLAG_ENUMERATE_DEVICE_INTERFACE flag
0x643	byte	VMBUS_SERVER_CHANNEL_INIT_FLAG_OFFER_AS_PIPE flag
0x644	byte	VMBUS_SERVER_CHANNEL_INIT_FLAG_FORCE_NEW_CHANNEL flag
0x645	byte	VMBUS_SERVER_CHANNEL_INIT_FLAG_TLNPI_PROVIDER_OFFER flag

Appendix 3 – Table of InInitializeQueue changes

Table 8. Changes made in VMBCHANNEL structure by InInitializeQueue

Offset	**Size**	**Value**
0x388	dword	0x0
0x510	sizeof(KDPC)	Initialized DPC object with DeferredRoutine pointer pointing to function InpProcessingDpcRoutine and DeferredContext pointing to channel structure itself
0x550	sizeof(KDPC)	Initialized DPC object with DeferredRoutine pointer pointing to function InpProcessingDpcRoutine and DeferredContext pointing to channel structure itself
0x598	dword	-1 or 0xFFFFFFFF
0x59C	dword	-1 or 0xFFFFFFFF
0x5A0	dword	-1 or 0xFFFFFFFF
0x448	qword	Pointer to channel structure offset 0x440
0x440	qword	Pointer to channel structure offset 0x440
0x398	qword	Pointer to channel structure offset 0x390
0x390	qword	Pointer to channel structure offset 0x390
0x458	qword	Pointer to channel structure offset 0x450
0x450	qword	Pointer to channel structure offset 0x450
0x4B0	dword	Dword from channel structure offset 0x614
0x3F3	byte	Single byte from channel structure offset 0x646
0x4BC	dword	0

Offset	Size	Value
0x4D8	qword	0
0x46C	dword	Dword from channel structure offset 0x63C
0x710	qword	Pointer to function InpChannelProcessPacketExNoOp
0x718	qword	Pointer to function KmclpDefaultChannelClosed
0x4E4	dword	Something related to sizes of external data
0x4E0	dword	Something related to sizes of external data

Appendix 4 – Table of OutInitializeQueue changes

Table 9. Changes made in VMBCHANNEL structure by OutInitializeQueue

Offset	Size	Value
0x100	sizeof(SPIN_LOCK)	Spin lock object initialized with KeInitializeSpinLock
0x240	sizeof(SPIN_LOCK)	Spin lock object initialized with KeInitializeSpinLock
0x110	qword	Pointer to structure at offset 0x0x110
0x118	qword	Pointer to structure at offset 0x0x110
0x288	sizeof(TIMER)	Timer structure initialized with KeInitializeTimer
0x2C8	sizeof(KDPC)	Initialized DPC object with DeferredRoutine pointer pointing to function OutpPollingDpcRoutine and DeferredContext pointing to channel structure itself
0x308	sizeof(KDPC)	Initialized DPC object with DeferredRoutine pointer pointing to function OutpPollingDpcRoutine and DeferredContext pointing to channel structure itself

Offset	Size	Value
0x140	sizeof(PAGED_LOOKASIDE_LIST)	Initializes NPAGED_LOOKASIDE_LIST structure inside channel structure via ExInitializeNPagedLookasideList with following parameters: 1. Lookaside = Pointer to structure offset 0x140 2. Allocate = 0 3. Free = 0 4. Flags = ExDefaultNonPagedPoolType | POOL_NX_ALLOCATION 5. Size = Dword taken from structure offset 0x618 + dword taken from structure offset 0x608 + 0x224 6. Tag = ‘Vkou’ 7. Depth = 0
0x1C0	byte	1
0x250	qword	0
0x258	qword	0
0x260	qword	0
0x268	qword	0
0x248	qword	Pointer to channel structure at offset 0x260
0x25C	dword	ExDefaultNonPagedPoolType
0x254	dword	If dword in structure at offset 0x604 is larger then 0x4000 then this is written, otherwise 0x4000
0x258	dword	0x636D6B56
0x250	dword	1

Appendix 5 – Table of VMBCHANNEL members

Table 10. Most useful members of VMBCHANNEL structure

Offset	**Size**	**Value**
0x0	qword	Pointer to structure itself (for verification)
0x604	dword	Maximum packet count
0x608	dword	Maximum packet size
0x614	dword	Client context size
0x61C	sizeof(GUID)	Interface type GUID
0x62C	sizeof(GUID)	Interface instance GUID
0x640	byte	Pipe flag
0x650	sizeof(GUID)	VM id GUID
0x6E0	qword	VMBus handle pointer
0x702	byte	VTL value
0x710	qword	Pointer to ProcessPacketCallback callback
0x718	qword	Pointer to ProcessingCompleteCallback callback
0x720	qword	CallbackContext value that is provided with all calls to ProcessPacketCallbackEx and ProcessingCompleteCallbackEx for this channel
0x738	qword	Pointer to EvtChannelOpened callback
0x740	qword	Pointer to EvtChannelClosed callback
0x748	qword	Pointer to EvtChannelSuspend callback

Offset	Size	Value
0x750	qword	Pointer to EvtChannelStarted callback
0x758	qword	Pointer to EvtChannelPostStarted callback
0x7A0	sizeof(`LIST_ENTRY`)	Double link list structure connecting all channels
0x7C8	sizeof(UNICODE_STRING)	UNICODE_STRING structure referencing the human readable name
0x838	qword	Channel set pointer
0xAD8	qword	Primary channel
0xB40	qword	Parent device object
0xB50	word	Subchannel index

Appendix 6 – showChannels.py script

```
import pykd
import uuid
header = pykd.getOffset("vmbkmclr!KmclChannelList")
nextPtr = pykd.loadQWords(header, 1)[0]
if header == nextPtr:
      print "No channels found!"
      exit()

def byteArray2ByteBuffer(arr):
      ret = ""
      for x in arr:
            ret += chr(x)
      return ret

while nextPtr != header:
      base = nextPtr – 0x7A0
      print "Channel at 0x%X" % base
      if pykd.loadQWords(base, 1)[0] !=  base:
            print "  INVALID CHANNEL"
            exit()
      pipe = pykd.loadBytes(base + 0x640, 1)[0]
      interfaceTypeGuid =
uuid.UUID(bytes_le=byteArray2ByteBuffer(pykd.loadBytes(base + 0x61C,
16)))
      interfaceInstanceGuid =
uuid.UUID(bytes_le=byteArray2ByteBuffer(pykd.loadBytes(base + 0x62C,
16)))
      vmIdGuid =
uuid.UUID(bytes_le=byteArray2ByteBuffer(pykd.loadBytes(base + 0x650,
16)))

      pointer = pykd.loadQWords(base + 0x838, 1)[0]
      primaryChannel = pykd.loadQWords(base + 0xAD8, 1)[0]
      parentDeviceObj = pykd.loadQWords(base + 0xB40, 1)[0]
      subchannelIndex = pykd.loadWords(base + 0xB50, 1)[0]
      packetCallback = pykd.loadQWords(base + 0x710, 1)[0]
      completeCallback = pykd.loadQWords(base + 0x718, 1)[0]
      channelOpened = pykd.loadQWords(base + 0x738, 1)[0]
```

```
      channelClosed = pykd.loadQWords(base + 0x740, 1)[0]
      channelSuspended = pykd.loadQWords(base + 0x748, 1)[0]
      channelStarted = pykd.loadQWords(base + 0x750, 1)[0]
      channelPostStarted = pykd.loadQWords(base + 0x758, 1)[0]

      if pipe > 0:
            print "  --Pipe--"
      else:
            print "  --Normal channel--"
      print "  Interface type: %s" % str(interfaceTypeGuid)
      print "  Interface instance: %s" % str(interfaceInstanceGuid)
      print "  VM id: %s" % str(vmIdGuid)
      print "  Pointer: 0x%X" % pointer
      print "  Parent Device Object: 0x%X" % parentDeviceObj
      print "  Primary channel: 0x%X" % primaryChannel
      print "  Sub channel index: 0x%X" % subchannelIndex
      print "  Callbacks"
      if packetCallback > 0:
            print "    packet callback = 0x%X (%s)" % (packetCallback,
pykd.findSymbol(packetCallback))
      if completeCallback > 0:
            print "    packet completion callback = 0x%X (%s)" %
(completeCallback, pykd.findSymbol(completeCallback))
      if channelOpened > 0:
            print "    channel opened = 0x%X (%s)" % (channelOpened,
pykd.findSymbol(channelOpened))
      if channelClosed > 0:
            print "    channel close = 0x%X (%s)" % (channelClosed,
pykd.findSymbol(channelClosed))
      if channelSuspended > 0:
            print "    channel suspended = 0x%X (%s)" %
(channelSuspended, pykd.findSymbol(channelSuspended))
      if channelStarted > 0:
            print "    channel started = 0x%X (%s)" % (channelStarted,
pykd.findSymbol(channelStarted))
      if channelPostStarted > 0:
            print "    channel post started = 0x%X (%s)" %
(channelPostStarted, pykd.findSymbol(channelPostStarted))

      nextPtr = pykd.loadQWords(nextPtr, 1)[0]
      print "\n"
```

Appendix 7 – Script handling buffer and MDL from packet handlers

```
import pykd
import uuid
import sys

def byteArray2ByteBuffer(arr):
      ret = ""
      for x in arr:
            ret += chr(x)
      return ret

def checkValidChannel(addr):
      return (pykd.loadQWords(addr, 1)[0] == addr)

def getChannelPtr(channel):
      if isinstance(channel, int) and not checkValidChannel(channel):
            return None
      if isinstance(channel, str):
            header = pykd.getOffset("vmbkmclr!KmclChannelList")
            nextPtr = pykd.loadQWords(header, 1)[0]
            if header == nextPtr:
                  return None

            while nextPtr != header:
                  base = nextPtr - 0x7A0
                  if not checkValidChannel(base):
                        return None
                  interfaceInstanceGuid =
uuid.UUID(bytes_le=byteArray2ByteBuffer(pykd.loadBytes(base + 0x62C,
16)))
                  if str(interfaceInstanceGuid) == channel:
                        return base
                  nextPtr = pykd.loadQWords(nextPtr, 1)[0]
            return None

if len(sys.argv) == 1:
```

```
	print "Missing channel address/instance GUID"
	exit()

if sys.argv[1].startswith("0x"):
	channel = getChannelPtr(int(sys.argv[1][2:], 16))
else:
	channel = getChannelPtr(sys.argv[1])

if channel is None:
	print "Could not find channel"

pykd.dbgCommand("bc *")
print "Channel found at 0x%X" % channel
packetCallback = pykd.loadQWords(channel + 0x710, 1)[0]
print "packet callback @ 0x%X (%s)" % (packetCallback,
pykd.findSymbol(packetCallback))
pykd.dbgCommand("bp 0x%X" % packetCallback)

packets = {}
breakAddr = pykd.getOffset("nt!DbgBreakPointWithStatus")
extAddr = pykd.getOffset("vmbkmclr!VmbChannelPacketGetExternalData")
pykd.dbgCommand("bp 0x%X" % extAddr)

cnt = 0
while True:
	pykd.dbgCommand("gh")
	if pykd.reg("rip") != packetCallback and pykd.reg("rip") !=
extAddr:
		if pykd.reg("rip") == breakAddr:
			break
		continue
	if pykd.reg("rip") == packetCallback:
		buf = pykd.reg("r8")
		bufSize = pykd.reg("r9")
		extDataFlag = pykd.loadQWords(pykd.reg("rsp") + 5*8, 1)[0]
& 1
		if extDataFlag == 0:
			print "Call to handler with buffer @ 0x%X with size
0x%X and no external data" % (buf, bufSize)
		else:
			packets[pykd.reg("rdx")] = (buf, bufSize)
	if pykd.reg("rip") == extAddr and pykd.reg("rcx") in packets:
		packet = pykd.reg("rcx")
		pmdl = pykd.reg("r8")
		pykd.dbgCommand("bd *")
```

```
            pykd.dbgCommand("gu")
            pykd.dbgCommand("be *")
            (buf, bufSize) = packets[packet]
            print "Call to handler with buffer @ 0x%X with size 0x%X
and external data:" % (buf, bufSize)
            while True:
                  mdl_next = pykd.loadQWords(pmdl, 1)[0]
                  print "  MDL @ 0x%X" % pmdl
                  if mdl_next == 0:
                        break
                  pmdl = mdl_next
            del packets[packet]
pykd.dbgCommand("bc *")
```