Windows

Exploiting the Hyper-V IDE Emulator to Escape the Virtual Machine

Joe Bialek (@JosephBialek)

MSRC Vulnerabilities & Mitigations Team

Microsoft

Why Is Microsoft Talking About Hyper-V?

Critical to customer (cloud) security

We want to show how much impact bug reports can have

We have paid $625,000 in Hyper-V bounties since Blackhat USA 2018

Rewind to sometime in 2017…

A Story About a Bug (CVE-2018-0959)

I’m reviewing Hyper-V emulators

Emulated Storage is old code, a little sketchy in some places…

```
void
IdeChannel::WriteDataPort(
    _In_                               IDE_DRIVE_STATE& Drive,
    _In_                               UINT8            AccessSize,    // Number of bytes being written (1, 2, or 4)
    _In_                               UINT16           AccessCount,   // Always 1
    _In_reads_bytes_(AccessSize*AccessCount) const VOID*      Buffer         // Buffer containing data to write (from IO port)
    )
{
    UINT8* curBuffer;
    if (Drive.Saved.UseCommandBuffer) {
        curBuffer = (UINT8*)Drive.CommandBuffer;    // Used for CDROM
    } else {
        curBuffer = Drive.TrackCacheBuffer + Drive.Saved.DriveStateBufferOffset;    // Used for IDE hard drive
    }

    if (curBuffer == NULL || !Drive.Saved.BufferPtrValid) {
        ...
    } else {
        UINT32 curByte = Drive.Saved.CurrentByte;
        UINT32 length = AccessCount * AccessSize;

        if (curByte + length > Drive.Saved.TotalBytes)
        {
            ...
            length = Drive.Saved.TotalBytes - curByte;
        }

        // Copy the data.
        RtlCopyMemory(curBuffer + curByte, Buffer, length);     // This looks suspicious
        curByte += length;

        Drive.Saved.CurrentByte = curByte;

...
    }
}
```

Attacker controlled data from IO port write

There are a lot of variables being used to compute the address passed to RtlCopyMemory

Some of these variables aren't validated here since they were expected to be validated when set…

A Story About a Bug (CVE-2018-0959)

I get side tracked on pentesting storage because…

Visual Studio team shows us GSL::Span

Fast-fail if you attempt to access memory out-of-bounds

Emulated Storage seemed like a great candidate

I start porting in GSL::Span to see how it works out for “real” system code

A Wild Crash Dump Appears

A researcher I work closely with sends a crash dump to me

Tells me a POC is on the way, the crash dump is a friendly heads up

Crash dump implicates the Emulated Storage component

The researcher ended up creating an exploit and receiving a $150,000 bounty

```
0:005> kc
 # Call Site
...
04 vfbasics!VerifierStopMessage
05 vfbasics!AVrfpCheckFirstChanceException
06 vfbasics!AVrfpVectoredExceptionHandler
07 ntdll!RtlpCallVectoredHandlers
08 ntdll!RtlCallVectoredExceptionHandlers
09 ntdll!RtlDispatchException
0a ntdll!KiUserExceptionDispatch
0b ucrtbase!MoveSmall
0c VmEmulatedStorage!IdeChannel::WriteDataPort
0d VmEmulatedStorage!IdeChannel::WritePort
0e VmEmulatedStorage!IdeChannel::AltWriteIoPort
0f VmEmulatedStorage!IdeControllerDevice::NotifyIoPortWrite
10 vmwp!VmbComIoPortHandlerAdapter::WriteCallback
11 vmwp!VmbCallback::NotifyIoPortWrite
12 vmwp!VND_HANDLER_CONTEXT::NotifyIoPortWrite
13 vmwp!EmulatorVp::DispatchIoPortOperation
14 vmwp!EmulatorVp::TrySimpleIoEmulation
15 vmwp!EmulatorVp::TryIoEmulation
16 vmwp!VndIce::HandleExecutionRequest
17 vmwp!VndCompletionHandler::HandleVndCallback
```

Verifier crashes due to access violation

WriteDataPort – suspicious function

IO Port handler for emulated storage

```
0:005> .frame c
… VmEmulatedStorage!IdeChannel::WriteDataPort+0x7e

0:005> dv /v
@rbx                Drive = 0x000001df`b17f7e28
00000098`984ff560  Buffer = 0x00000098`984ff5b8
@rdi                curBuffer = 0x000001df`a6d45e00 "--- memory read error at address 0x000001df`a6d45e00
@esi                length = 4
@ebp                curByte = 0

0:005> dx -r1 ((VmEmulatedStorage!IDE_DRIVE_STATE *)0x1dfb17f7e28)
((VmEmulatedStorage!IDE_DRIVE_STATE *)0x1dfb17f7e28) : 0x1dfb17f7e28 [Type: IDE_DRIVE_STATE *]
    [+0x000] Saved            [Type: IDE_DRIVE_SAVED_STATE]
    [+0x098] Attachment       : 0x1dfb3b916b0 [Type: IdeAttachment *]
    [+0x0a0] CommandBuffer    : 0x1dfb17fc000 : 0x0 [Type: unsigned short *]
    [+0x0a8] TrackCacheBuffer : 0x1dfa6d20000 : 0x0 [Type: unsigned char *]
    [+0x0b0] TrackCacheSize   : 0x10000 [Type: unsigned int]

0:005> dx -r1 (*((VmEmulatedStorage!IDE_DRIVE_SAVED_STATE *)0x1dfb17f7e28))
(*((VmEmulatedStorage!IDE_DRIVE_SAVED_STATE *)0x1dfb17f7e28))
...
    [+0x040] DriveStateBufferOffset : 0x25e00 [Type: unsigned int]
```

curBuffer = TrackCacheBuffer + DriveStateBufferOffset

DriveStateBufferOffset looks WAY too big

Storage Emulation Architecture

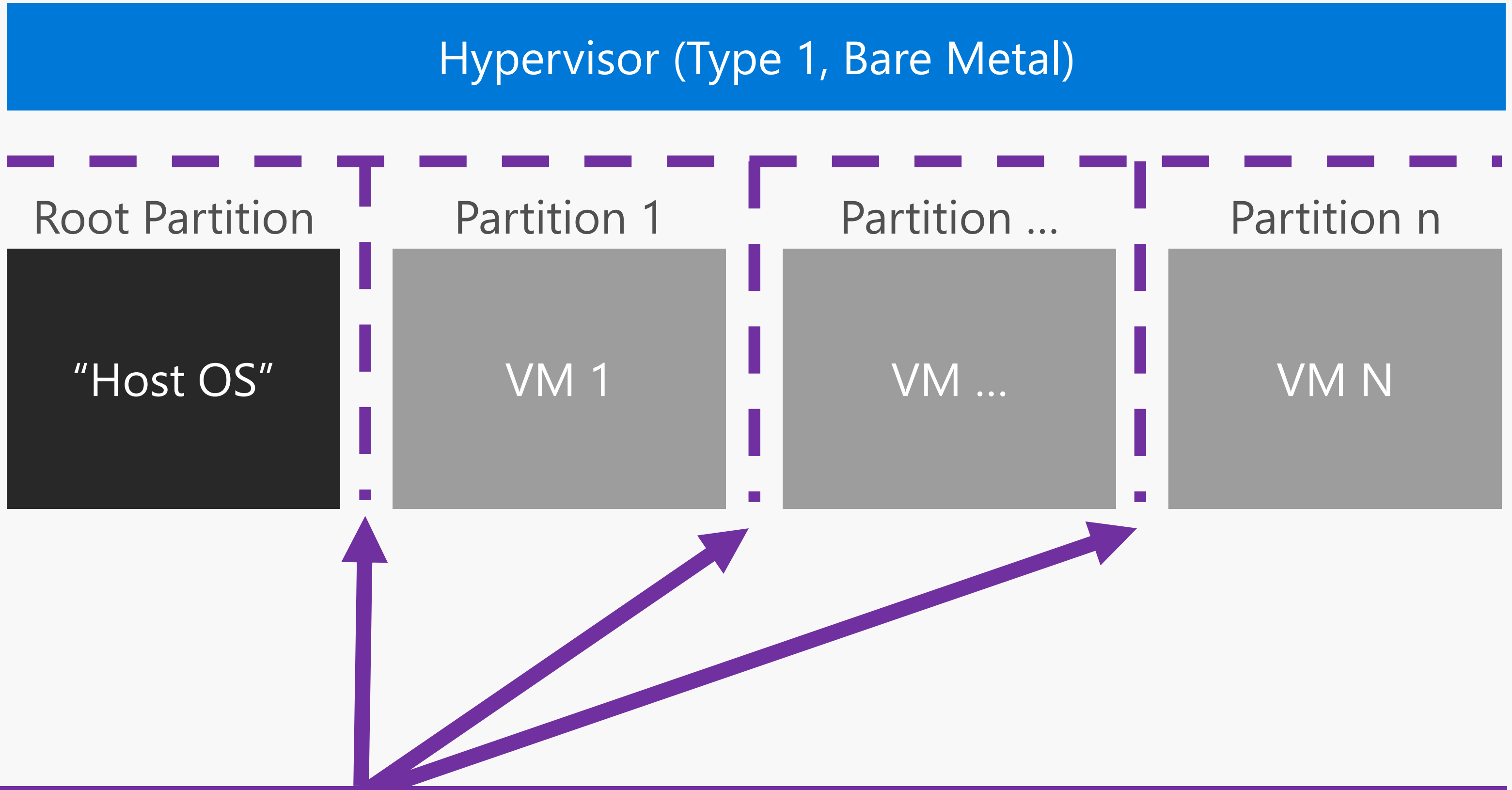
Hyper-V Architecture: Hypervisor

Manages physical address space of partitions (via EPT)

Manages virtualization specific hardware configuration

Handles intercepts (i.e. HyperCall, in/out instructions, CPUID instruction, EPT page fault, etc.)

Interrupt delivery to guests



Hypervisor EPT enforces physical memory isolation between partitions

Most Hyper-V attack surface is not in the hypervisor

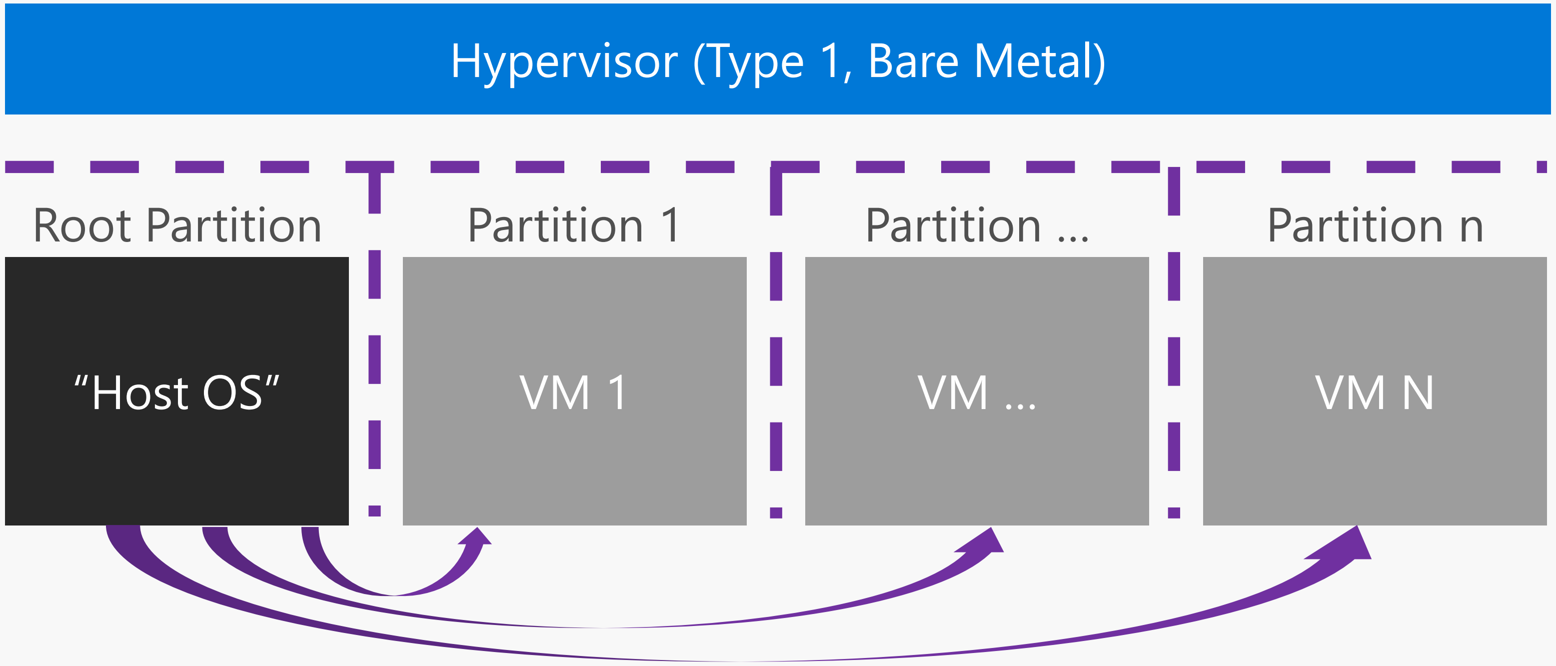
Hyper-V Architecture: Root Partition

Manages other VM's (create/destroy/etc.)

Access to the physical memory of other partitions

Access to all hardware

Provides services such as device emulation, para-virtualized networking/storage, etc.



Root partition can access other partitions' physical memory

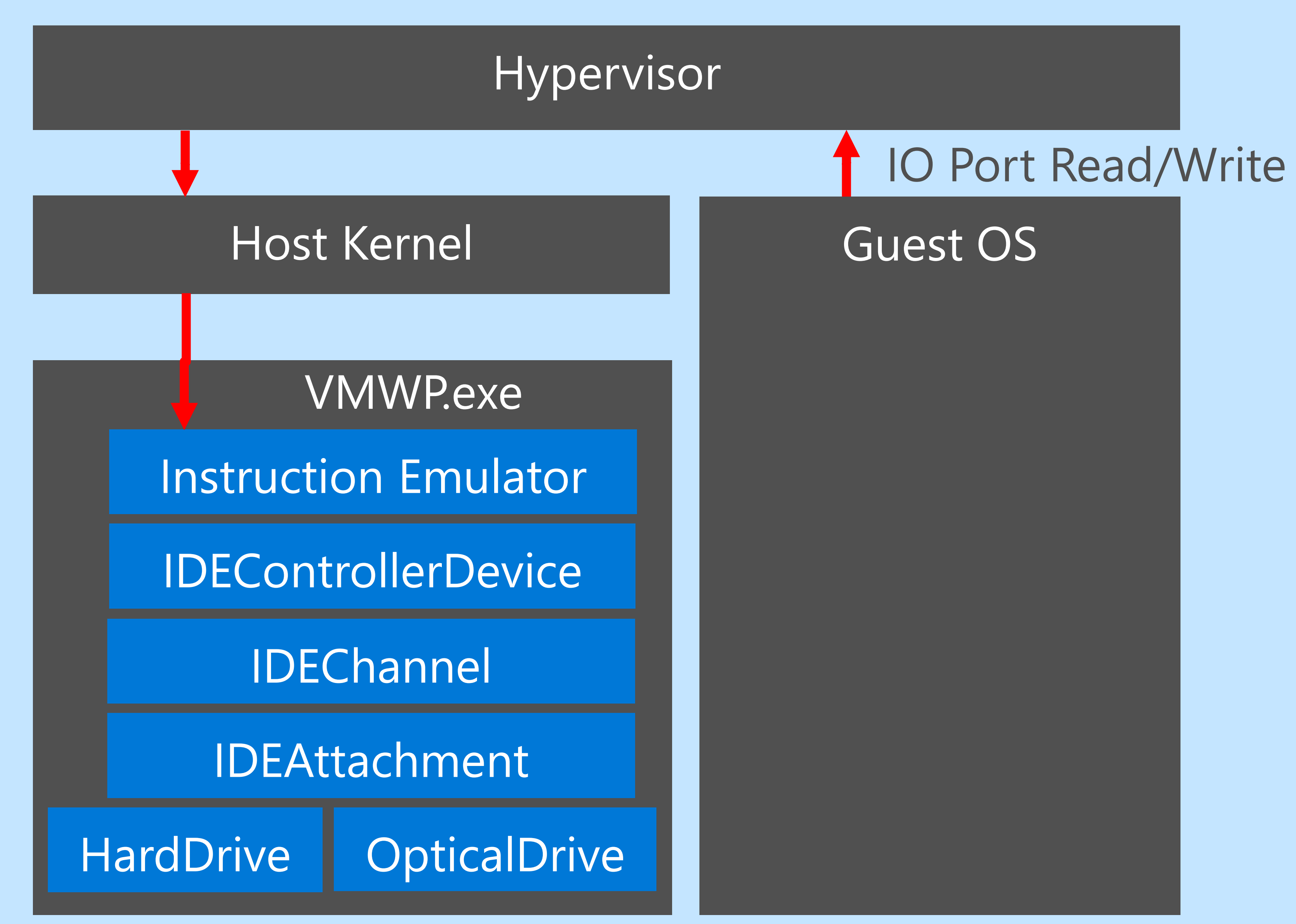
Most Hyper-V attack surface is in the root partition

IDE Emulator runs in the VM Worker Process (VMWP)

1 VMWP per VM

In/Out instructions allow 1, 2, or 4 byte read/write

IO Ports For This Talk: 1F0-1F7



The Bug

Triggering the Bug

```
WriteIoPort(0x1F3, 0x10, 1);

WriteIoPort(0x1F2, 0x77, 1);

WriteIoPort(0x1F7, 0x30, 1);

for (DWORD bytesWritten = 0;
     bytesWritten < 0x200;
     bytesWritten += 4) {

    WriteIoPort(0x1F0, 0x41414141, 4);
}
```

Put the device in the desired state

```
for (DWORD i = 0; i < 0x10000; i++) {
    WriteIoPort(0x1F7, 0x30, 1);
}
```

Each write increments the DriveStateBufferOffset by 0x200. Make it huge!

```
WriteIoPort(0x1F0, 0x13371337, 4);
```

Trigger a write to the TrackCacheBuffer

```
LeakedData = ReadIoPort(0x1F0, 4);
```

Trigger a read from TrackCacheBuffer

Reach WriteDataPort by writing to IO Port 0x1F0

```
void
IdeChannel::WriteDataPort(
    _In_                                IDE_DRIVE_STATE& Drive,
    _In_                                UINT8            AccessSize,    // Number of bytes being written (1, 2, or 4)
    _In_                                UINT16           AccessCount,   // Always 1
    _In_reads_bytes_(AccessSize*AccessCount) const VOID*      Buffer         // Buffer containing data to write (from guest)
    )
{
    UINT8* curBuffer;
    if (Drive.Saved.UseCommandBuffer) {
        curBuffer = (UINT8*)Drive.CommandBuffer;    // Used for CDROM
    } else {
        curBuffer = Drive.TrackCacheBuffer + Drive.Saved.DriveStateBufferOffset;     // Used for IDE hard drive
    }

    if (curBuffer == NULL || !Drive.Saved.BufferPtrValid) {
        ...
    } else {
        UINT32 curByte = Drive.Saved.CurrentByte;
        UINT32 length = AccessCount * AccessSize;

        if (curByte + length > Drive.Saved.TotalBytes)
        {
            ...
            length = Drive.Saved.TotalBytes - curByte;
        }

        // Copy the data.
        RtlCopyMemory(curBuffer + curByte, Buffer, length);     // Out-of-bounds
        curByte += length;

        Drive.Saved.CurrentByte = curByte;

        ...
    }
}
```

Offset incorrectly set too large

Out-of-bounds pointer

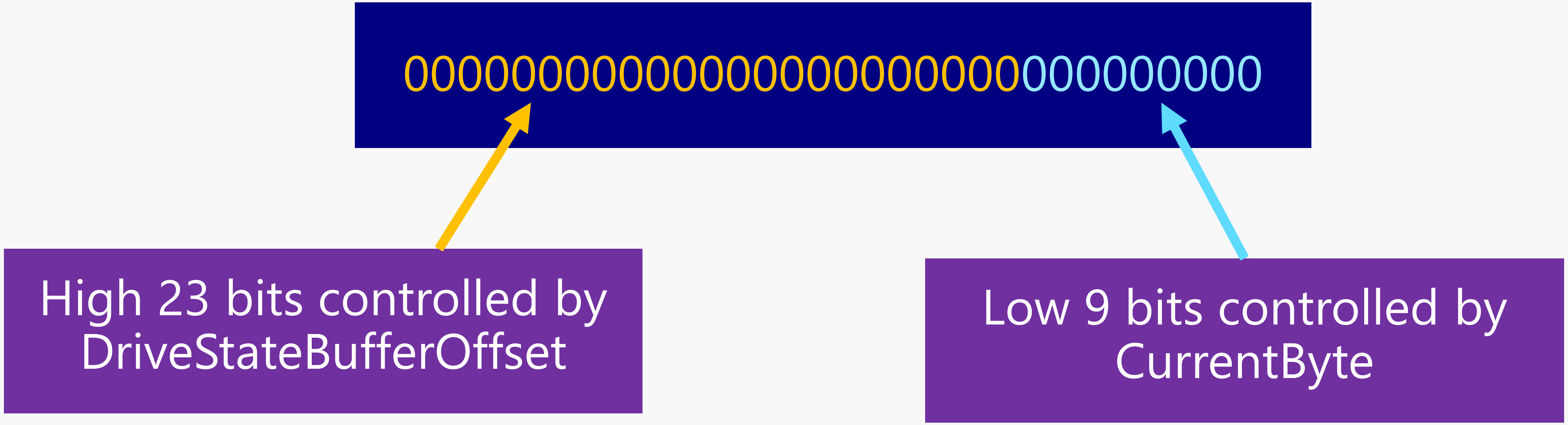
CurrentByte between 0-511, incremented each time this function is called

1, 2, or 4 bytes

Guest controlled data

Relative write primitive (attacker controlled data, 32-bit index)

32-bit offset added to TrackCacheBuffer



Reach ReadDataPort by writing to IO Port 0x1F0

```
void IdeChannel::ReadDataPort(
    _In_                                IDE_DRIVE_STATE& Drive,
    _In_                                UINT8            AccessSize,  // Number of bytes being read (1, 2, or 4)
    _In_                                UINT16           AccessCount, // Always 1
    _Out_writes_bytes_(AccessSize*AccessCount) PVOID     Buffer       // Buffer containing data to read (to guest)
    )
{
    UINT8* curBuffer;
    if (Drive.Saved.UseCommandBuffer) {
        curBuffer = (UINT8*)Drive.CommandBuffer;
    } else {
        curBuffer = Drive.TrackCacheBuffer + Drive.Saved.DriveStateBufferOffset;
    }

    if (curBuffer == NULL || !Drive.Saved.BufferPtrValid) {
        ...
    } else {
        UINT32 curByte = Drive.Saved.CurrentByte;
        UINT32 length = AccessSize * AccessCount;

        if (curByte + length > Drive.Saved.TotalBytes) {
            ...
            length = Drive.Saved.TotalBytes - curByte;
        }

#pragma prefast(suppress: __WARNING_BUFFER_COPY_NO_PREDICATES, "Copy is correctly bounded by buffer length [AccessSize*AccessCount].")
        RtlCopyMemory(Buffer, curBuffer + curByte, length);
        curByte += length;

        Drive.Saved.CurrentByte = curByte;
        ...
    }
}
```

Offset incorrectly set too large

Out-of-bounds pointer

CurrentByte between 0-511, incremented each time this function is called

1, 2, or 4 bytes

Buffer will be copied back to the guest by the instruction emulator

Relative read primitive (32-bit index)

The GSL::Span version of emulated storage fast-fails when the POC is run

Too bad we hadn’t shipped it yet

Exploiting Server 2012R2

Constraints

Emulated path is slow -- timing attacks / races are probably not practical

Generation 1 VM’s only (no emulated storage on Generation 2)

CFG, ASLR, DEP enabled

VMWP.exe is 64-bit only

TrackCacheBuffer allocated using VirtualAlloc

Allocations are made sequentially to reduce fragmentation

Allocations are 64KB aligned

Result: Allocations *may* be at predictable offsets from each other

Example making allocations with VirtualAlloc

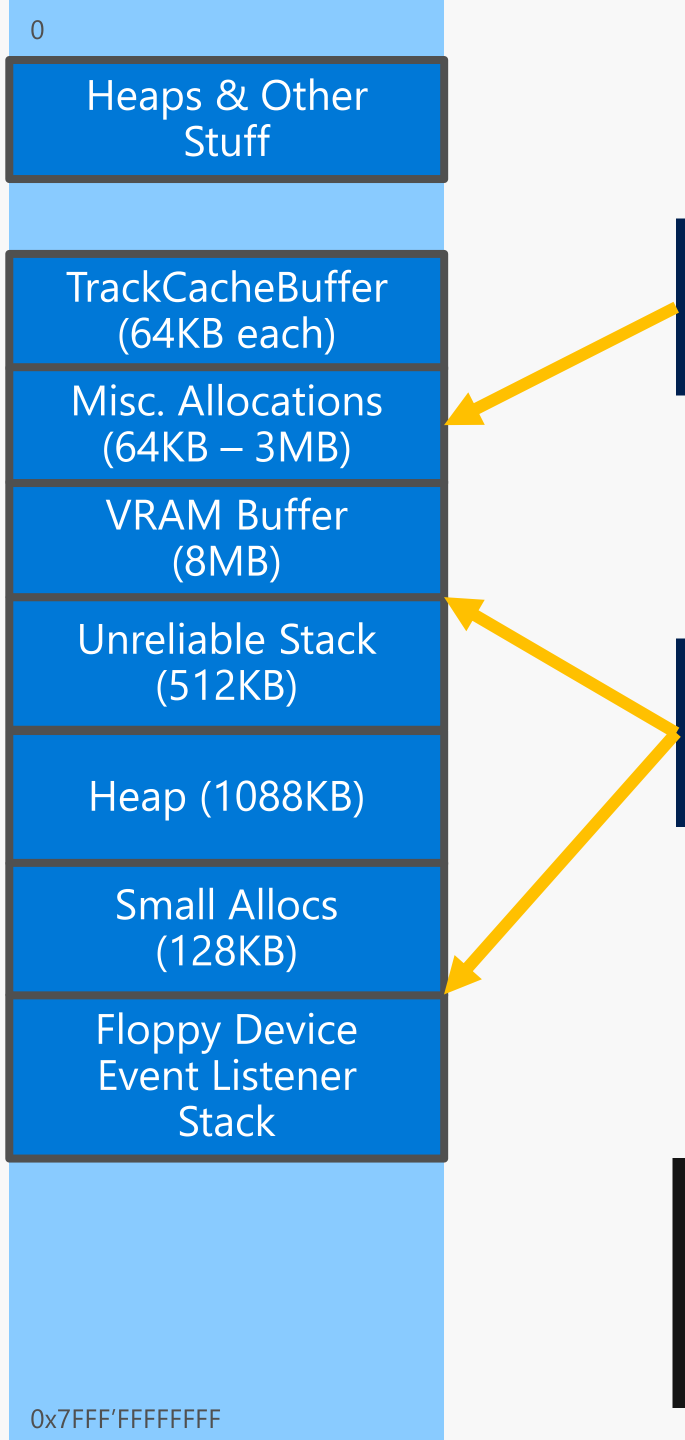
```
Alloc 1: 0x2a0bf8a0000
Alloc 2: 0x2a0bf8b0000
Alloc 3: 0x2a0bf8c0000
Alloc 4: 0x2a0bf8d0000
…
…
```

Can I find an interesting allocation that is a predictable offset from the TrackCacheBuffer?

For more info: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2012_07_BlackHatUSA/BHUSA12_Johnson_Miller_Exploit_Mitigation_Improvements_in_Windows_8.pdf

0000008A29E70000	Private Data	128 K	128 K	128 K	64 K	64 K	TrackCacheBuffer
0000008A29E90000	Private Data	128 K	128 K	128 K	4 K	4 K	
0000008A29EB0000	Mapped File	48 K	48 K		32 K		
0000008A29EC0000	Private Data	4 K	4 K	4 K	4 K	4 K	
0000008A29ED0000	Private Data	4 K	4 K	4 K	4 K	4 K	
0000008A29EE0000	Private Data	4 K	4 K	4 K	4 K	4 K	
0000008A29EF0000	Private Data	4 K	4 K	4 K	4 K	4 K	
0000008A29F00000	Private Data	4 K	4 K	4 K	4 K	4 K	
0000008A29F10000	Thread Stack	512 K	44 K	44 K	4 K	4 K	
0000008A29F90000	Private Data	12 K	12 K	12 K	12 K	12 K	
0000008A29FA0000	Private Data	44 K	44 K	44 K	44 K	44 K	
0000008A29FB0000	Private Data	48 K	48 K	48 K	48 K	48 K	
0000008A29FC0000	Private Data	60 K	60 K	60 K	60 K	60 K	
0000008A29FD0000	Private Data	32 K	32 K	32 K	32 K	32 K	
0000008A29FE0000	Heap (Private Data)	64 K	16 K	16 K	8 K	8 K	
0000008A29FF0000	Private Data	56 K	56 K	56 K	56 K	56 K	
0000008A2A000000	Private Data	16 K	16 K	16 K	16 K	16 K	
0000008A2A010000	Private Data	8,192 K	8,192 K	8,192 K	8,192 K	8,192 K	VRAM Buffer
0000008A2A810000	Thread Stack	512 K	52 K	52 K	24 K	24 K	
0000008A2A890000	Heap (Private Data)	1,044 K	1,028 K	1,028 K	1,028 K	1,028 K	
0000008A2A9A0000	Private Data	52 K	52 K	52 K	52 K	52 K	
0000008A2A9B0000	Thread Stack	512 K	44 K	44 K	8 K	8 K	Floppy Stack
0000008A2AA30000	Thread Stack	512 K	44 K	44 K	4 K	4 K	
0000008A2AAB0000	Thread Stack	512 K	44 K	44 K	8 K	8 K	

Memory Layout Generalization

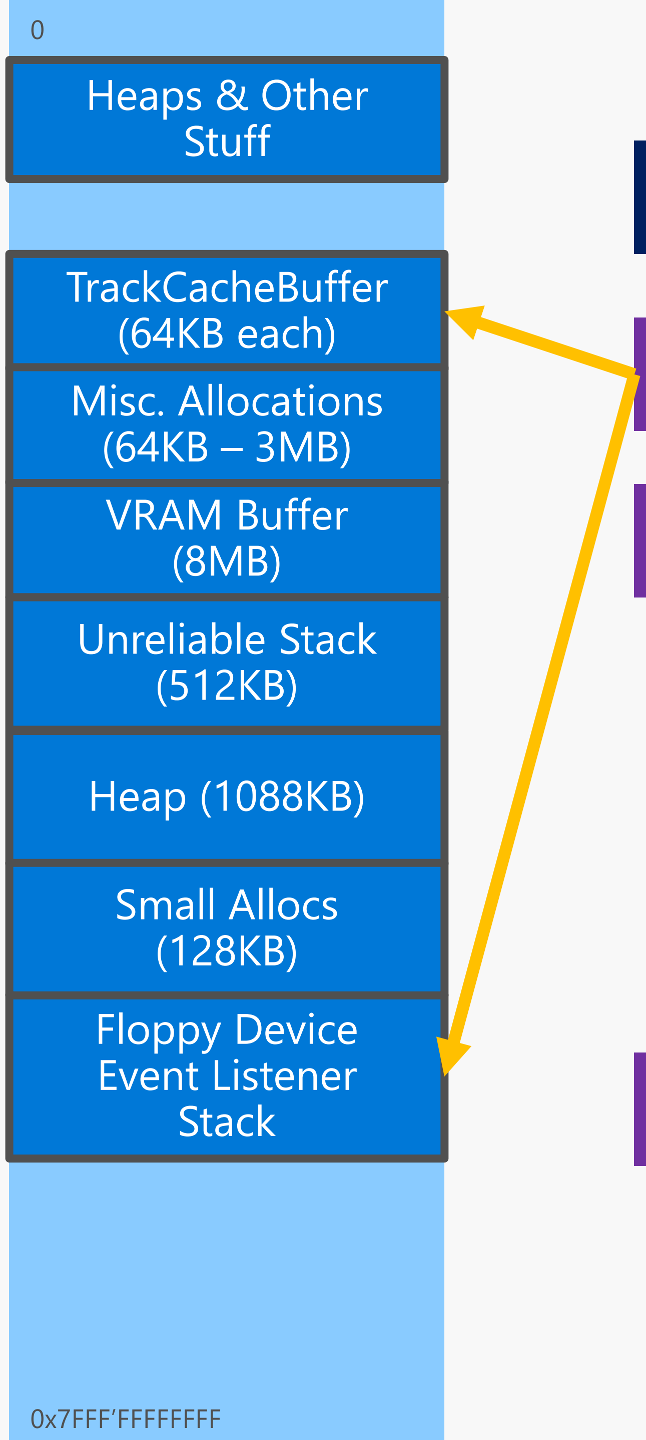


Amount of data varies greatly, but rarely more than a few MB.

Usually a fixed offset of 1728KB between end of VRAM Buffer and start of Floppy Device Event Listener Stack

Notes:

- Unreliable Stack was an unreliable corruption target (stack died a lot).
- Heap (1088KB) had a single 1024KB allocation (plus heap header). Wasn't a good corruption target.
- Small Allocs varied between 0, 64KB, and 128KB. It usually seemed to be 128KB, though.
- VRAM buffer maps Guest Physical Memory using an Aperture. It is effectively a shared section with guest memory.



Corrupting a stack would allow an immediate CFG bypass

No fixed offset between TrackCacheBuffer and stack

Need a reliable way to read/write the stack

Maybe the VRAM buffer would be helpful?

VRAM Buffer

VRAM Buffer Expanded

8MB

8MB-64KB

8MB-128KB

…

…

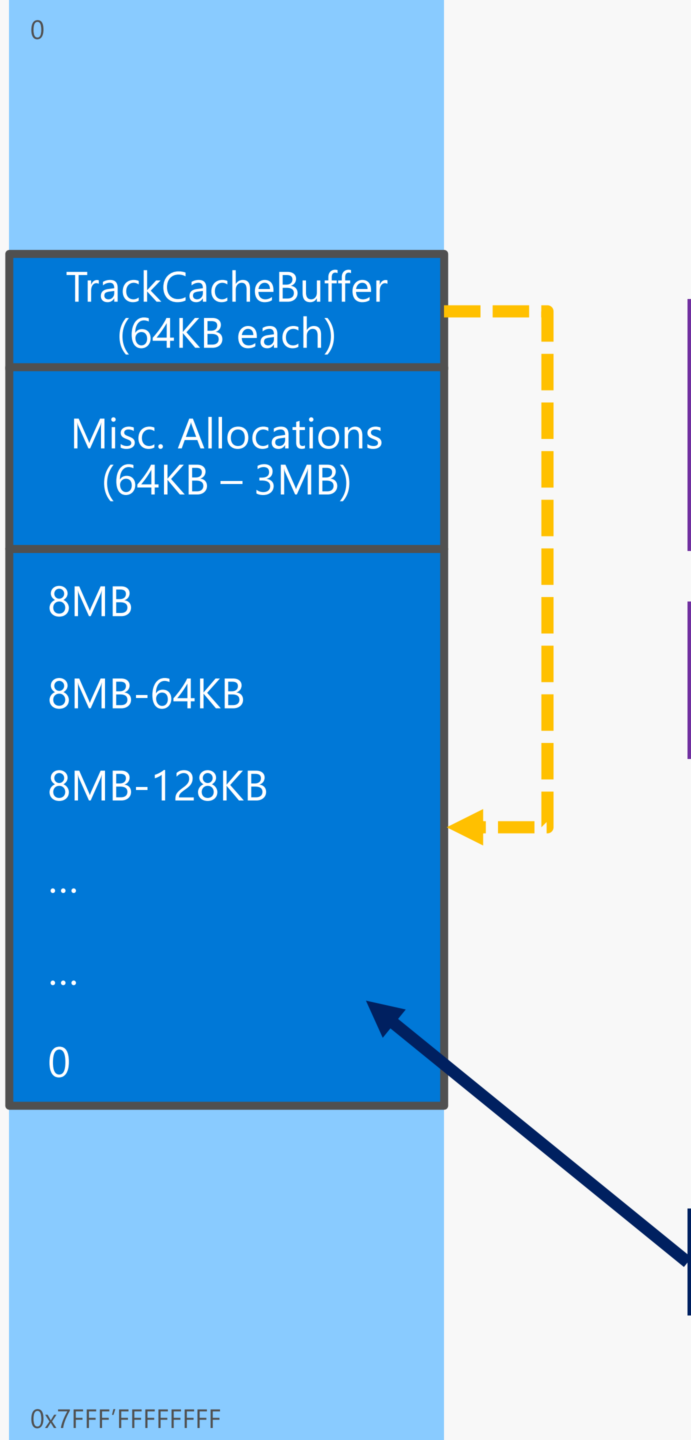
0

8MB Aperture (shared section that maps guest memory)

Any changes to this guest memory is immediately visible in the VMWP

Fill the guest memory with a pattern. Pattern indicates how many bytes are left in the VRAM Buffer

Skipping Past Misc. Allocations

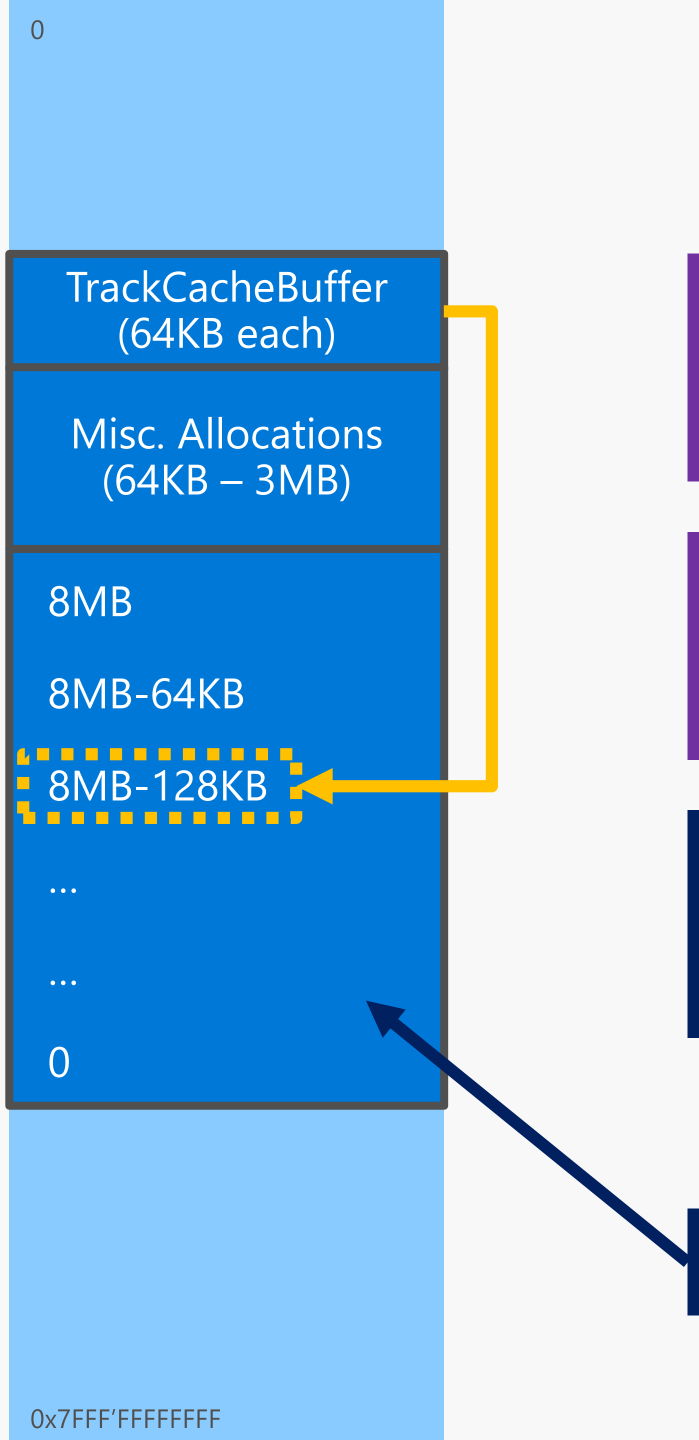


Indexing 8MB off the end of the track cache buffer usually results in skipping past "Misc. Allocations" and landing somewhere in the VRAM Buffer

Impossible to predict the precise location it will be in the VRAM buffer

VRAM Buffer with markers at every 64KB aligned address

Skipping Past Misc. Allocations



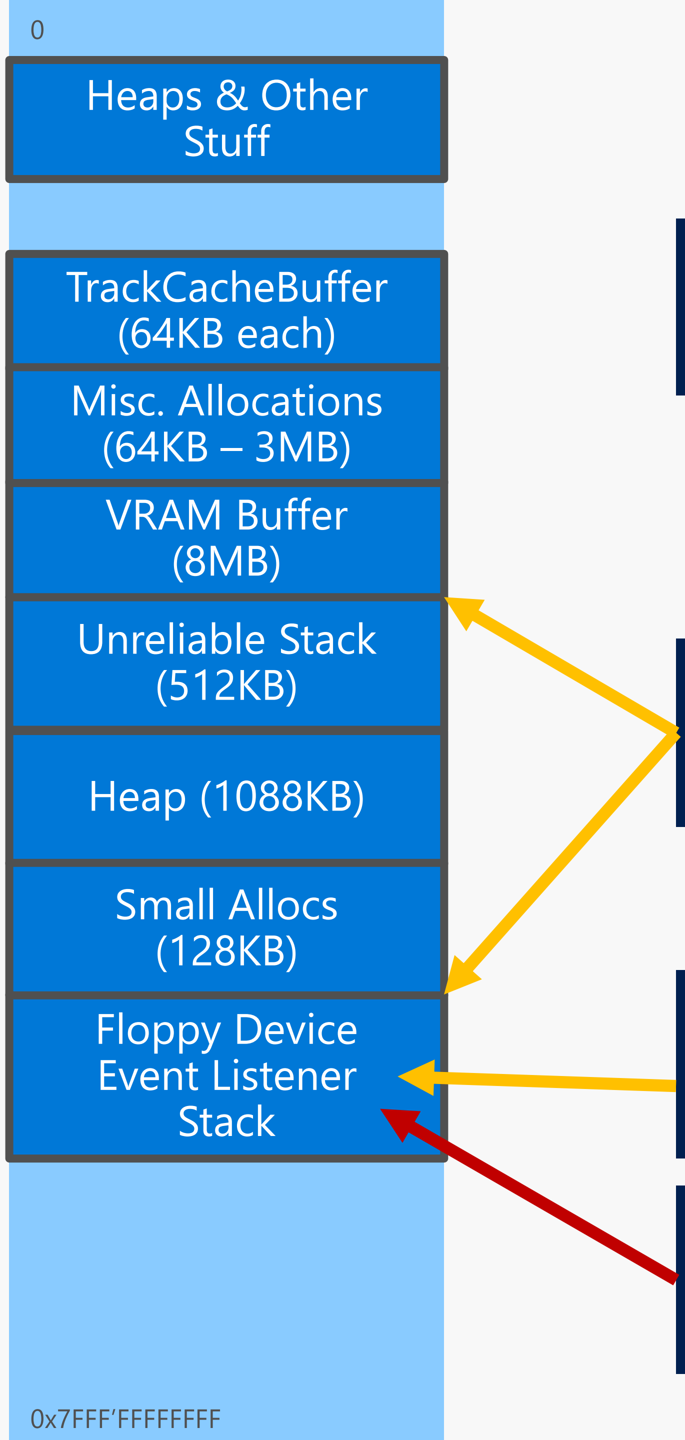
Use out-of-bounds read. Data returned == marker in VRAM Buffer

Marker indicates how many bytes of the VRAM buffer follow

Offset from start of TrackCacheBuffer to end of VRAM Buffer == 64KB (TrackCacheBufferSize)+ 8MB + Marker

VRAM Buffer with markers at every 64KB aligned address

Exploiting



I now know the offset from TrackCacheBuffer to end of VRAM Buffer

Usually a fixed offset of 1728KB between end of VRAM Buffer and start of Floppy Device Event Listener Stack

Use OOB read to read the Floppy Device Event Listener stack, get code addresses from it

Use OOB write to write ROP payload to Floppy Device Event Listener stack and corrupt return instruction pointer

Triggering Payload

Need the Floppy stack to unwind so my corrupted return instruction pointer is used

This stack waits on events. Events never come since nobody inserts floppy disks in their VM. No events == no stack unwinding

Rebooting the VM triggers an event that causes the thread to unwind and shut down

In other words, rebooting the VM triggers the payload

Payload uses ROP to call WinExec to launch calc

Demo

Exploiting Windows 10 1709

What’s Changed?

Address Space Layout

Stacks/TEB’s/PEB’s mapped in their own isolated region (thanks Jordan Rabet)

Exploit Mitigations

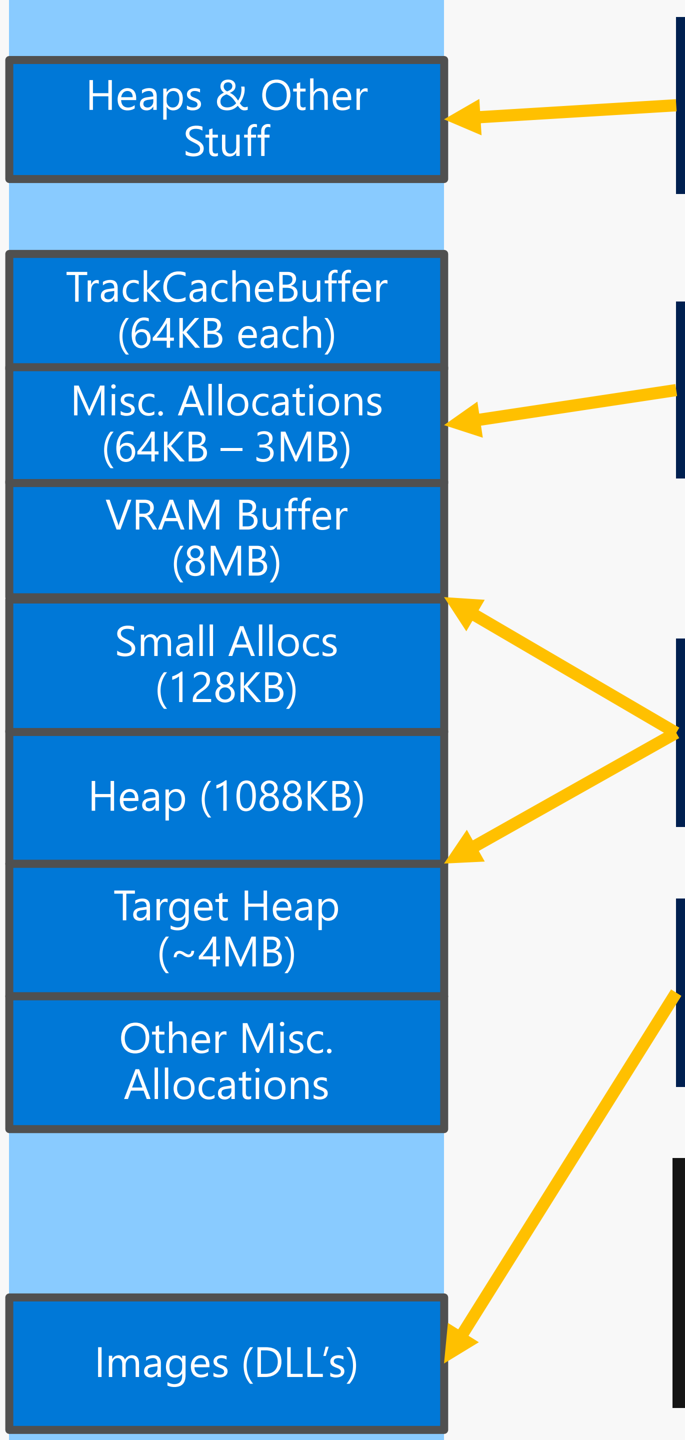
ACG – VMWP cannot create executable pages

CIG – VMWP can only load Microsoft signed code

NoChildProc – VMWP cannot spawn child processes

CFG Improvements (bypasses fixed)

Stack mapping change means exploit can no longer read/write the stacks



These heaps are where most device emulator objects are allocated ☹

Amount of data varies greatly, but never more than a few MB

Usually a fixed offset of 1216KB between end of VRAM Buffer and start of "Target Heap"

Multiple terabytes away from VirtualAlloc allocations. Cannot directly read with OOB read/write

Notes:
- Heap (1088KB) had a single 1024KB allocation (plus heap header). Wasn't a good corruption target.
- Small Allocs varied between 0, 64KB, and 128KB. It usually seemed to be 128KB, though.
- VRAM buffer maps Guest Physical Memory using an Aperture. It is effectively a shared section with guest memory.
- Target Heap was the only heap that was accessible from the OOB read/write (aside from the 1088KB heap).

Exploitation Ideas

1. Use data-corruption to boost my relative read-write primitive to arbitrary read-write, then corrupt a stack to get arbitrary code execution

2. Corrupt a function pointer and use a CFG bypass to get arbitrary code execution

3. Panic ☺

Exploitation Ideas

1. Use data-corruption to boost my relative read-write primitive to arbitrary read-write, then corrupt a stack to get arbitrary code execution

2. Corrupt a function pointer and use a CFG bypass to get arbitrary code execution

3. Panic ☺

First Attempt

Look for sprayable allocations with interesting data to corrupt

Log all heap allocations, inspect call stack to determine if guest can influence it

Check if allocation has interesting data to corrupt to get arbitrary read/write

Didn’t pan out ☹

Generation 1 VM’s have next to no allocations the guest can influence

None of these allocations were interesting to corrupt

Second Attempt

Look at existing allocations to find good corruption targets

Needs to have a long lifetime (don't want it to free while being corrupted)

Needs to have interesting data to corrupt to get arbitrary read/write

Specific idea: Corrupt emulator state

Make IO port reads/writes get written to a pointer I set

Make reads/writes to apertures go to a pointer I set

Second Attempt

Didn't pan out ☹

Emulators were almost never allocated in the Target Heap

None of the allocations in the Target Heap were good data corruption targets

Exploitation Ideas

1. Use data-corruption to boost my relative read-write primitive to arbitrary read-write, then corrupt a stack to get arbitrary code execution

2. Corrupt a function pointer and use a CFG bypass to get arbitrary code execution

3. Panic ☺

Third Attempt

Find Hyper-V binary with CFG bypass

Binaries that weren't compiled with CFG

Binaries that have specific indirect calls missing instrumentation

Didn't pan out ☹

No missing instrumentation bugs found

Some CFG bypasses identified, but they relied on winning races between 2 threads or knowing your current threads stack address. Neither was viable here

Final Attempt

Abuse course-grained nature of CFG

Make valid indirect calls to further corrupt process state

Obtain arbitrary read-write via indirect calls

Building Blocks

VideoDirtListener class commonly allocated in Target Heap

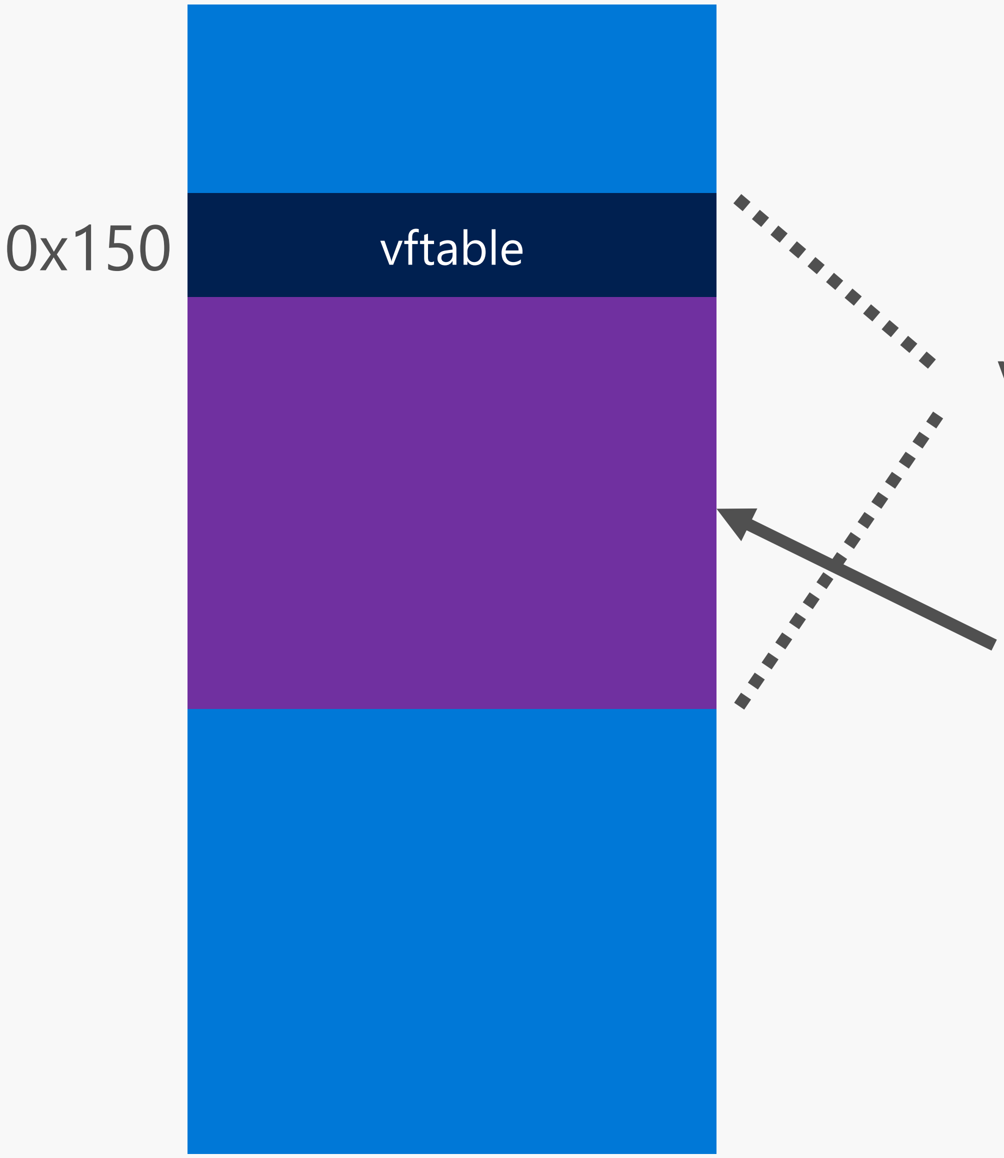
Has virtual functions, only called when VM reboots

Constraint: Need a place to put payload since most heap allocations will be destroyed when VM reboots

Has large (over 4096 bytes) buffer that is unused during normal operation, solves the above constraint

Address of COMBASE.dll and RPCRT4.dll commonly found in Target Heap

VideoDirtListener

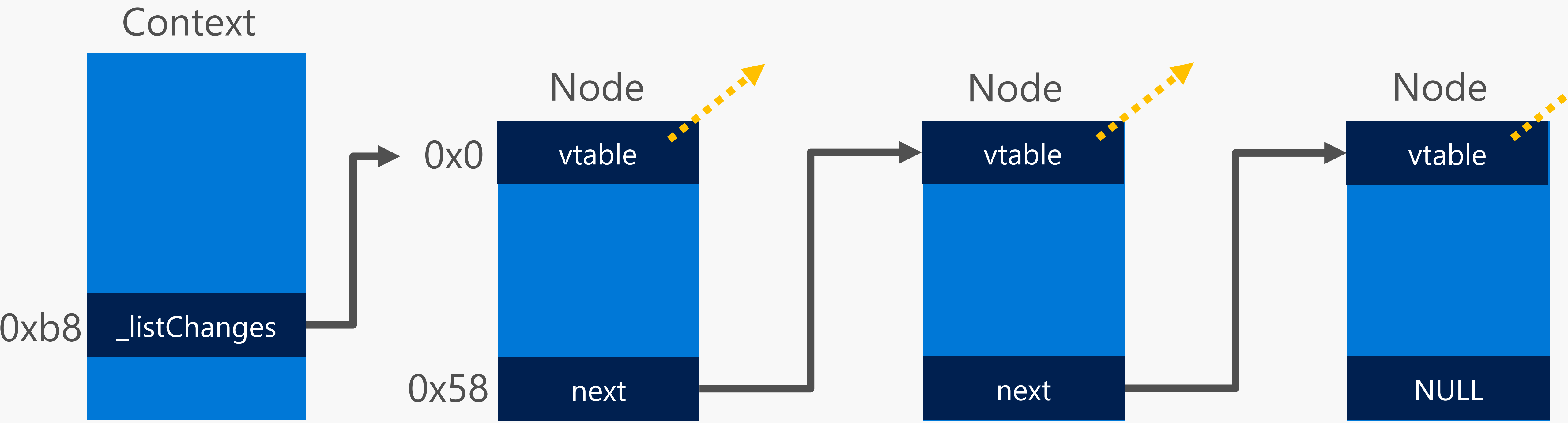


VideoDirt object embedded in VideoDirtListener

A bit over a 4096 bytes not generally used so a good place to store payload

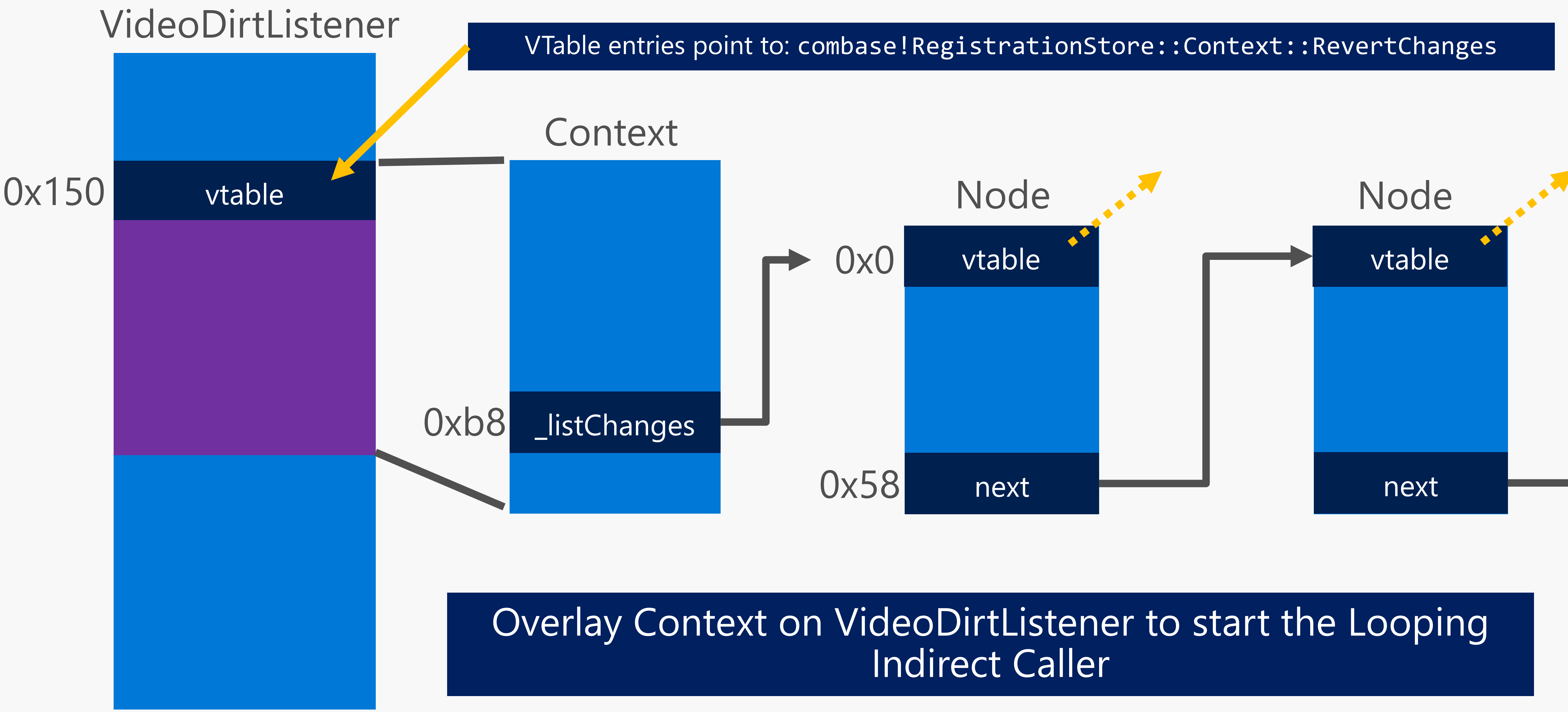
CFG Gadgets – Looping Indirect Caller

`combase!RegistrationStore::Context::RevertChanges`



Loops through linked list of Node* and makes virtual function call on each one

CFG Gadgets – Looping Indirect Caller



Overlay Context on VideoDirtListener to start the Looping Indirect Caller

CFG Gadgets – Control Indirect Call Parameters

NdrServerCall2 research done by Thomas Garnier

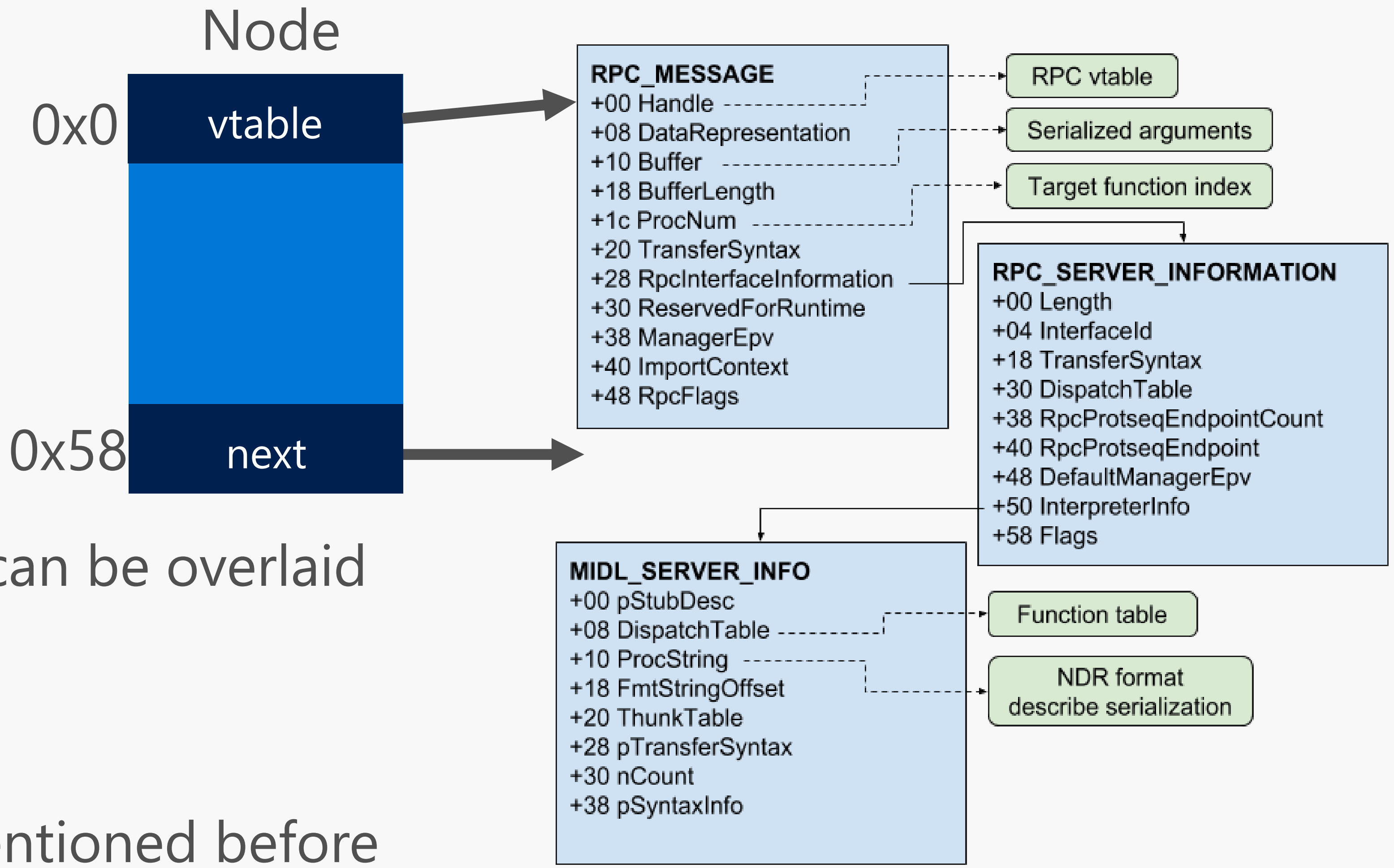
Create crafted buffer, pass to NdrServerCall2 as first parameter

NdrServerCall2 unmarshalls this buffer, makes indirect call to specified function with specified parameters

Indirect call target must be valid CFG target, parameters can be anything

For more info: https://medium.com/@mxatone/mitigation-bounty-from-read-write-anywhere-to-controllable-calls-ca1b9c7c0130

CFG Gadgets – Control Indirect Call Parameters



Node and RPC_MESSAGE can be overlaid without conflict

Entire payload gets put in VideoDirtListener array mentioned before

Image From: https://medium.com/@mxatone/mitigation-bounty-from-read-write-anywhere-to-controllable-calls-ca1b9c7c0130

CFG Gadgets - Memcpy

GenericBaseStream<Istream,AllocationWrapper>::GetCopy

```
… GetCopy(BYTE *pBuff)
{
    memcpy(pBuff,
           this->_pifData->abData,
           this->_cbBufferSize);
    return S_OK;
}
```

Valid CFG call target that calls memcpy with caller supplied arguments

CFG Gadgets – What I Have

VideoDirtListener object can be corrupted to initially hijack control flow

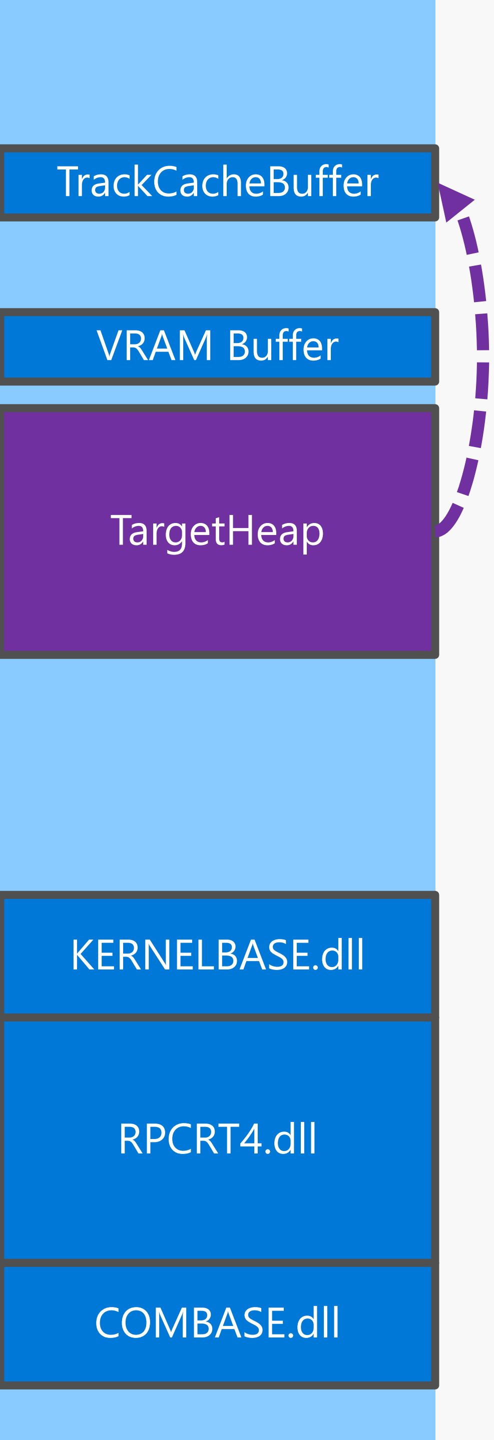
Make an arbitrary number of indirect calls with controlled parameters

Call memcpy with controlled parameters (arbitrary read/write primitive)

Must reboot the VM to trigger these payloads

Final Exploit - Strategy

- Get arbitrary read/write primitive
- Leak the address of kernelbase.dll
- Use kernelbase!VirtualProtect to make __guard_check_icall_fptr writeable
- Neutralize CFG by overwriting __guard_check_icall_fptr with a no-op function
- Execute arbitrary code via indirect call



Use relative read primitive to read the entire Target Heap

Find the following addresses

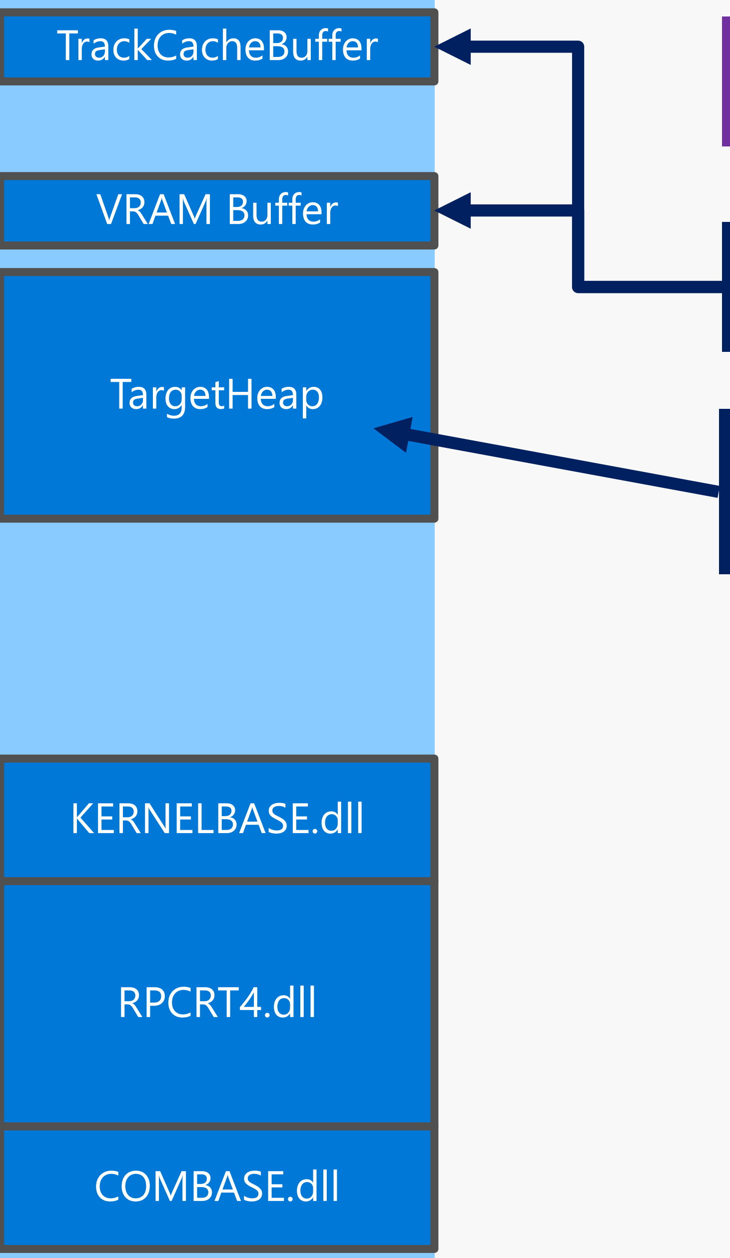
RPCRT4.dll

COMBASE.dll

VideoDirtListener

Note, the VideoDirtListener is not always in the TargetHeap, if it isn’t I need to reboot the VM and try again

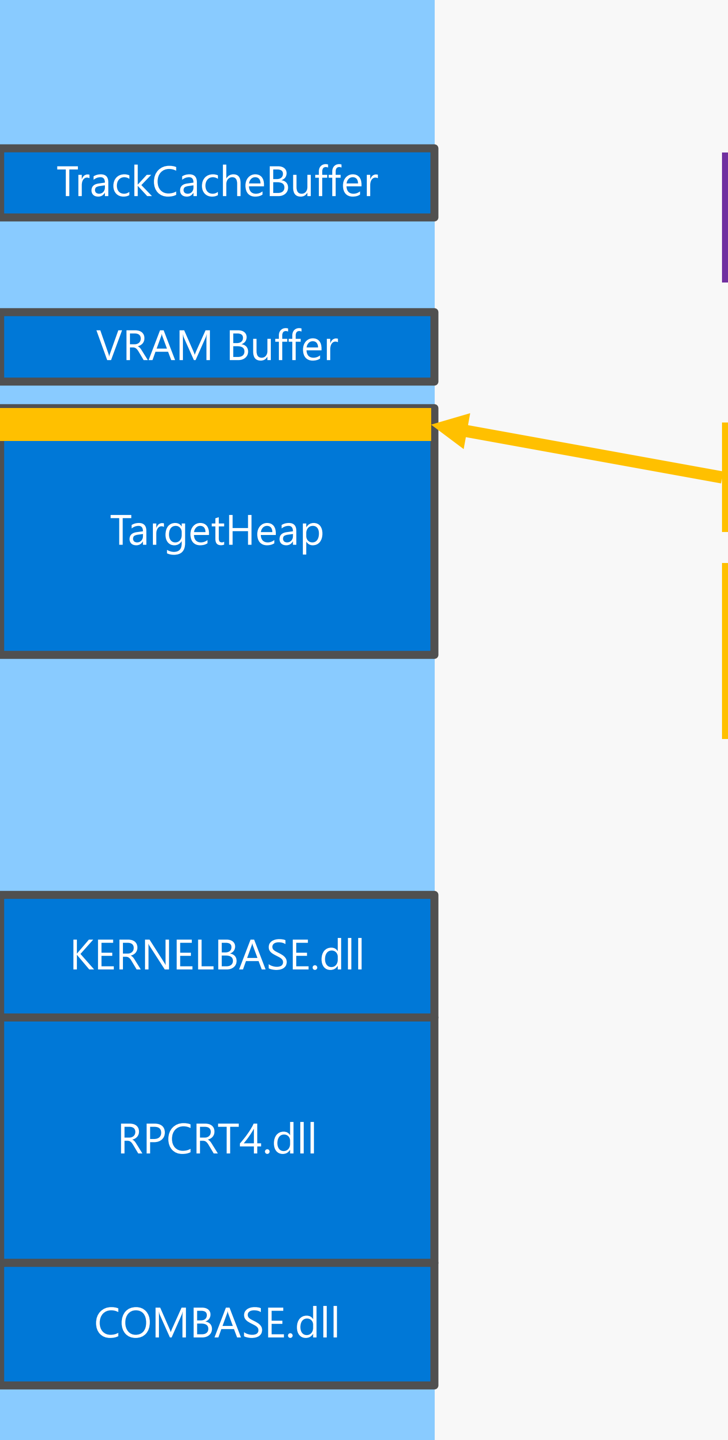
Next I will use memcpy gadget, but what destination?



Buffers free'd because the VM is rebooting

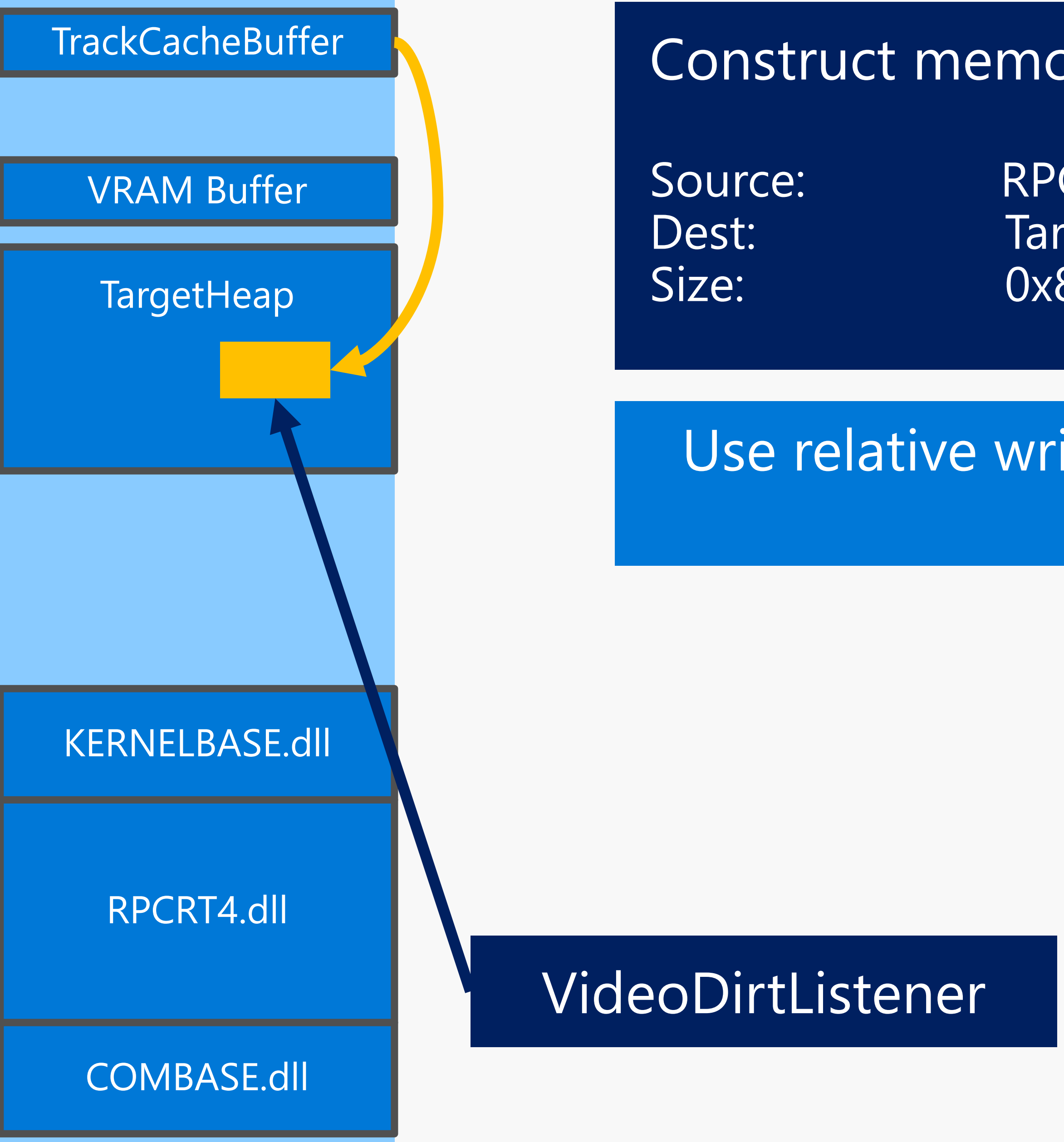
Most allocations in heap free'd because the VM is rebooting

Next I will use memcpy gadget, but what destination?



HEAP_SEGMENT header (not destroyed by VM rebooting)

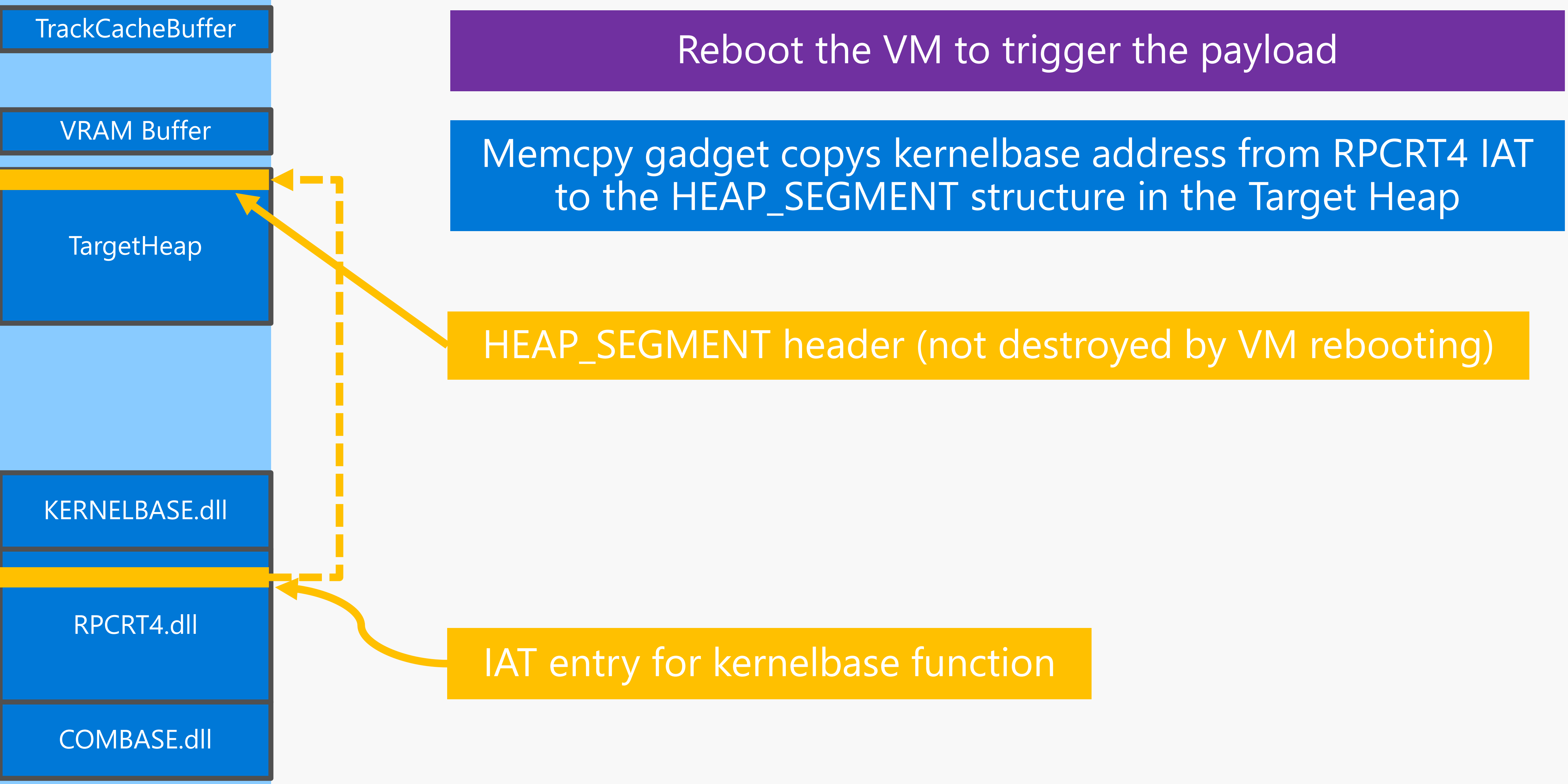
Placed at the start of the TargetHeap (contains heap metadata)

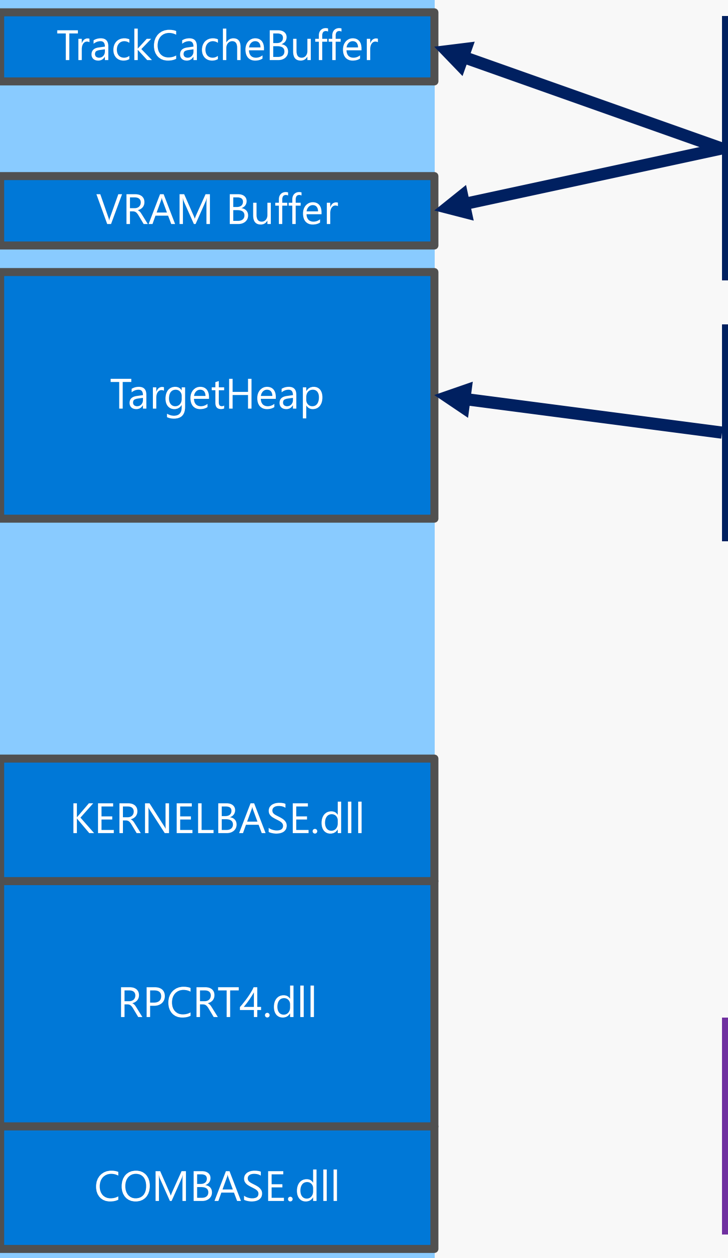


Construct memcpy payload with the following parameters:

Source:	RPCRT4 IAT entry kernelbase!ResolveDelayLoadedAPI
Dest:	TargetHeap+0x58
Size:	0x8

Use relative write primitive to corrupt VideoDirtListener with this payload

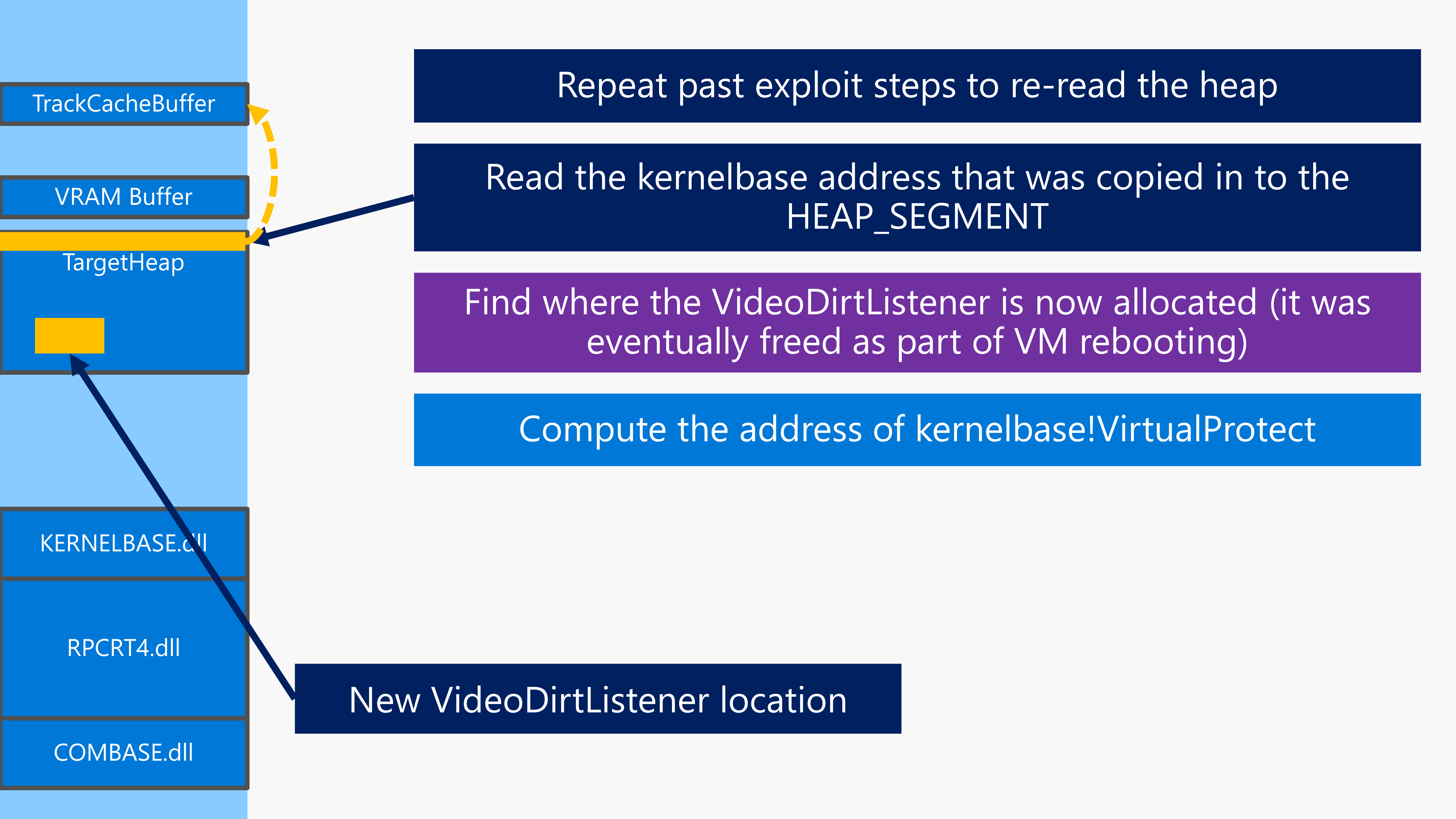

Reboot the VM to trigger the payload
Memcpy gadget copys kernelbase address from RPCRT4 IAT to the HEAP_SEGMENT structure in the Target Heap
HEAP_SEGMENT header (not destroyed by VM rebooting)
IAT entry for kernelbase function
TrackCacheBuffer
VRAM Buffer
TargetHeap
KERNELBASE.dll
RPCRT4.dll
COMBASE.dll

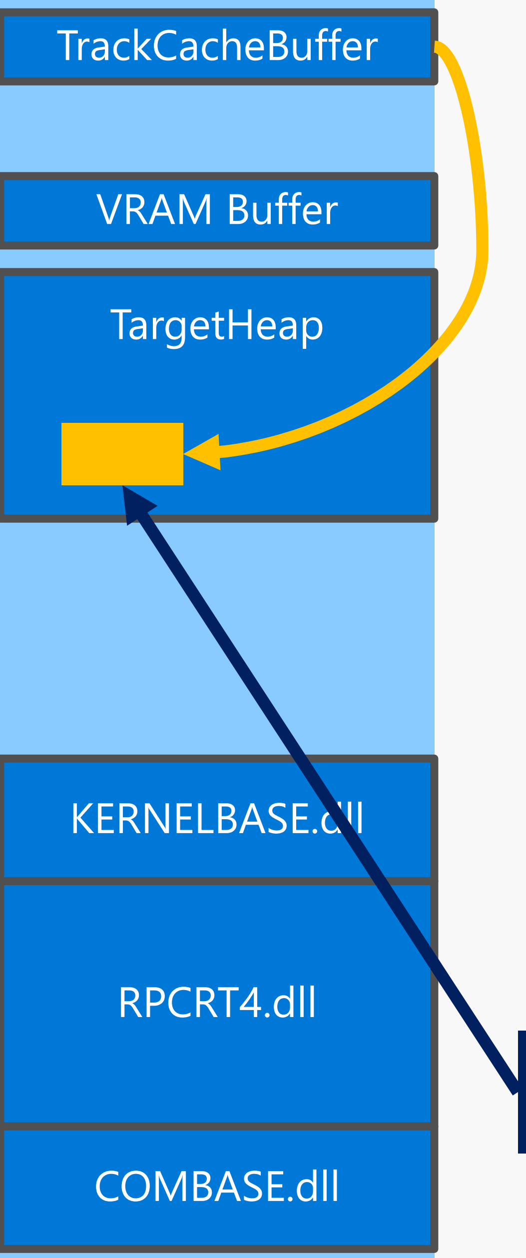


After the VM is rebooted, due to VirtualAlloc determinism, TrackCacheBuffer and VRAM Buffer are allocated at similar addresses

Many heap allocations were freed (and re-allocated) but the heap itself is never freed and is at the same address

After VM reboots, TrackCacheBuffer can be re-exploited in the same way due to the memory layout not changing much


Repeat past exploit steps to re-read the heap
Read the kernelbase address that was copied in to the HEAP_SEGMENT
Find where the VideoDirtListener is now allocated (it was eventually freed as part of VM rebooting)
Compute the address of kernelbase!VirtualProtect
TrackCacheBuffer
VRAM Buffer
TargetHeap
KERNELBASE.dll
RPCRT4.dll
COMBASE.dll
New VideoDirtListener location



Build new payload which does the following:

Call VirtualProtect to make __guard_check_icall_fptr writeable

Call memcpy to overwrite __guard_check_icall_fptr with a no-op function

CFG is now neutralized. Indirectly call the start instruction of a ROP chain.

Use relative write primitive to overwrite VideoDirtListener with this payload

New VideoDirtListener location

Payload

Due to exploit mitigations, payload must be written using ROP

Cannot create arbitrary executable pages or load unsigned images

Cannot create new processes or load images over the network

My payload simply writes a folder to “C:\” to prove arbitrary code execution

Demo

Sandbox Escape

Sandboxing

VMWP originally designed with least-privilege in mind

Runs with per-process unique SID, not as SYSTEM

Unfortunately this account is part of “Authenticated Users”

At the time, VMWP held SeImpersonatePrivilege (game over)

Achieving SYSTEM privilege was a matter of writing a big payload, I didn’t have time to do it

Learnings & Investments

Exploitation Difficulty

Metric	2012 R2	1709
Time-to-write-exploit	3 days	20 days
Exploit Reliability	~40%	<10%

Heap guard pages dropped reliability (2012R2 exploit avoided the heap)

Having to run the exploit at least twice in 1709 dropped reliability

CFG forces me to find specific objects and DLL addresses that aren’t always available in the heap, requires additional exploit attempts (more chances to fail)

Language Safety

The GSL::Span port that was completed as this vulnerability was reported killed the bug

Proved the value of Span – Hyper-V has been using it aggressively since (over 200 files)

MSVC team has done optimization work to ensure Span performance is good

This vulnerability report directly contributed to these investments

Language Safety

Microsoft has been aggressively investing in bug class elimination recently

InitAll Project – Kill uninitialized memory bugs

Using safer language features – Smart pointers, span

Static analysis tooling (Semmle) to eliminate specific bug patterns

Microsoft has been investigating the use of safer languages like Rust

Sandboxing

Virtualization is our most robust security boundary, but sandboxing still interesting

Hyper-V has investigated a strong sandbox for the VMWP as defense-in-depth

Working prototype for specific scenarios, not production ready

Tactical: VMWP removed SeImpersonatePrivilege and other sensitive privileges

This work was largely motivated by this vulnerability report and exploit

Kernel-Mode to User-mode

Microsoft has written user-mode and kernel-mode Hyper-V exploits

User-mode offers key advantages that Hyper-V wants to take advantage of

More robust exploit mitigations

Ability to use safer languages (safe C++ features, Rust, etc.)

Potential for extreme sandboxing for defense-in-depth

Hyper-V is investigating moving kernel-code to user-mode

Exploit Mitigations

Microsoft uses exploits to analyze the effectiveness of current and in-development mitigations

CFG Export Suppression wasn’t enabled, would have broken this exploit. It is now enabled

XFG would further break this exploit, announced at BlueHat Shanghai 2019

Microsoft uses exploits to analyze the impact of potential CPU features

How would memory tagging impact this exploit? What about CHERI?

Your Bug Reports Matter

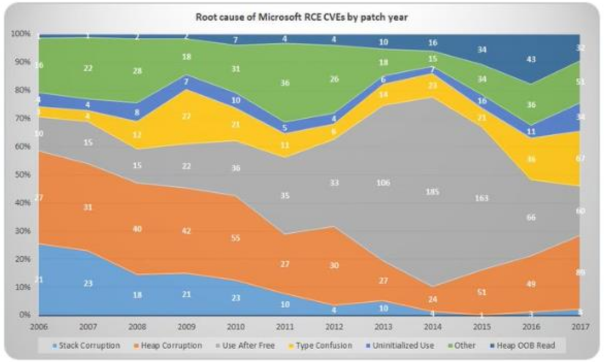
All vulnerability reports are used to analyze bug trends & identify hotspots that need mitigation work or pentesting

Some bug reports (like this one) have much greater impact

Matt Miller
@epakskape

Fellow data nerds: here's a snapshot of the vulnerability root cause trends for Microsoft Remote Code Execution (RCE) CVEs, 2006 through 2017.

A few callouts: heap corruption, type confusion, and uninit increased in 2017. Use after free steady y/y but proportionally declined.



4.4 million paid in bounties in the past year

$625,000 paid in Hyper-V bounties in the past year

Hyper-V Bug Bounty (as of August 2019)

RCE w/ Exploit (Guest-to-Host Escape)	Up to $250,000 (Hypervisor/Kernel) Up to $150,000 (User-mode)
RCE (Guest-to-Host Escape)	Up to $200,000 (Hypervisor/Kernel) Up to $100,000 (User-mode)
Information Disclosure	Up to $25,000 (Hypervisor/Kernel) Up to $15,000 (User-mode)
Denial of Service	Up to $15,000 (Hypervisor/Kernel)

See aka.ms/bugbounty for details

Microsoft Bounty Program

Encourage and reward high impact security research

Azure
up to $300K
(Just increased August 2019)

Hyper-V
up to $250K

Microsoft Identity
up to $100K

Mitigation Bypass
up to 100K

Windows Insider Preview **up to $50K**

Online Services
up to $20K

Azure DevOps
up to $20K

And more…

$4.4 million in bounty awards Jul 2018 – Jul 2019