

QubesOS articles

---

## Nvidia driver debugging

---

*Author:*  
Neowutran



# Contents

1	Debugging Windows and Linux Nvidia drivers	2
1.1	Related issues	2
1.2	Description of the issue	2
1.3	Finding the code that broke the 'nvidia' driver.	2
1.4	Fixing the Linux Nvidia driver	2
1.4.1	Deciding on the road to take for this debug session.	2
1.4.2	Structure of the driver files	3
1.4.3	Creating a common build process for 'nvidia' and 'nvidia-open' drivers	3
1.4.4	Similarities between the functions of 'nvidia' and 'nvidia-open' driver	5
1.4.5	Add traces in the driver	7
1.4.6	Interesting differences between our traces of the 'nvidia-open' and 'nvidia'	11
1.4.7	Patching the "nvidia" driver	11
1.5	Fixing the Windows Nvidia driver	15
1.5.1	Setting up remote windows kernel debugging on QubesOS	16
1.5.2	Creating a simplified driver to be sure the crash is not influenced by anything other than "nvlddmkm.sys"	18
1.5.3	Debugging and modifying 'nvlddmkm.sys'	20
1.5.4	Patching "nvlddmkm.sys"	26
1.5.5	Signing the nvidia driver	27
1.6	Conclusion	28
1.6.1	Extra mile	28
1.6.2	Follow up	29
1.7	References	30

# 1. DEBUGGING WINDOWS AND LINUX NVIDIA DRIVERS

---

## 1.1 Related issues

- [QubesOS issue n°8783](#)
- [QubesOS issue n°8631](#)
- [QubesOS issue n°9003](#)
- [Forum post from Solene in "Create a Gaming HVM"](#)
- [Forum post from Cameron](#)

## 1.2 Description of the issue

### Before

- For Linux, the 'nvidia' and 'nvidia-open' worked fine for GPU passthrough.
- For Windows, the 'nvidia' driver worked fine for GPU passthrough.

### Now

- For Linux, the 'nvidia-open' driver still works as before
- For Linux, the 'nvidia' driver now crash with a message like this one: `GPU 0000:00:06.0: GPU has fallen off the bus.`
- For Windows, the 'nvidia' driver causes a Blue Screen of Death: `System_Thread_Exception_Not_Handled` in 'nvlddmkm.sys'

## 1.3 Finding the code that broke the 'nvidia' driver.

I tested multiple version of 'xen' and 'xen-hvm-stubdom-linux\*' until I found which commit broke the 'nvidia' driver, and specifically which modification.

It got broken by this commit: [Add MSI-X support to stubdom.](#)

And specifically by this patch file:

[0001-hw-xen-xen\\_pt-Save-back-data-only-for-declared-regis.patch](#)

However, I am not able to see something wrong with this patch. It seems that the 'nvidia' driver tries to make an illegal call on the PCI bus, this commit actually enforce the interdiction. The 'nvidia' driver doesn't know how to handle the error and crash. So let's debug that.

## 1.4 Fixing the Linux Nvidia driver

### 1.4.1 Deciding on the road to take for this debug session.

Since I knew the patch that broke the 'nvidia' driver is : [0001-hw-xen-xen\\_pt-Save-back-data-only-for-declared-regis.patch](#).

I knew that the issue was related to things about accessing PCI configuration, read or write, most probably write.

I also knew that when the driver crashed, this message was printed on the kernel log.

```
NVRM: Xid (PCI:0000:00:06): 79, pid='<unknown>', name=<unknown>, GPU has
↳ fallen off the bus.
```

```
NVRM: GPU 0000:00:06.0: GPU has fallen off the bus.
```

```
NVRM: A GPU crash dump has been created. If possible, please run
```

```
NVRM: nvidia-bug-report.sh as root to collect this data before
```

```
NVRM: the NVIDIA kernel module is unloaded.
```

```
NVRM: GPU 0000:00:06.0: RmInitAdapter failed! (0x23:0xf:1426)
```

```
NVRM: GPU 0000:00:06.0: rm_init_adapter failed, device minor number 0
```

So I decided to find some way to add as much log as possible to understand what are the last few things being executed before it crashes. And I would combine the static analysis using 'ghidra' with the trace log that the driver will generate in order to orientate myself in the binary.

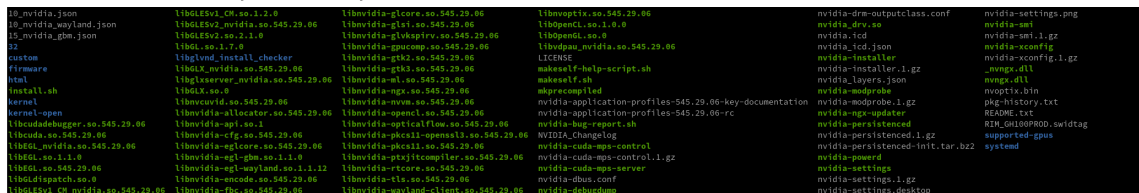
## 1.4.2 Structure of the driver files

I downloaded the latest release of the 'nvidia' Linux driver on the 'nvidia' website and extracted its content:

```
./NVIDIA-Linux-x86_64-535.154.05.run -x
```

The most interesting thing here is the two folders 'kernel' and 'kernel-open', for respectively the 'nvidia' and 'nvidia-open' drivers.

Figure 1: Listing of the files extracted from NVIDIA-Linux-x86\_64-535.154.05.run (some files have been manually added by me)



Inside those folders we will find C files, header files, Makefile, and the 'nvidia' kernel blob named 'nv-kernel.o\_binary'. Both the 'nvidia' and 'nvidia-open' drivers have a 'nv-kernel.o\_binary' blob but it is not the same blob.

## 1.4.3 Creating a common build process for 'nvidia' and 'nvidia-open' drivers

I created a file 'install.sh', its goal is to compile and install the driver. I took the inspiration from the Archlinux build process for the 'nvidia-open' package.

```
#!/bin/bash
```

```
make SYSSRC="/usr/src/linux"
```

```
_extradir="/usr/lib/modules/$(</usr/src/linux/version)/extramodules"
```

```
install -Dt "${_extradir}" -m644 *.ko
```

```
# Force module to load even on unsupported GPUs
```

```
mkdir -p /usr/lib/modprobe.d
```

```
echo "options nvidia NVreg_OpenRmEnableUnsupportedGpus=1" >
```

```
↳ /usr/lib/modprobe.d/nvidia-open.conf
```

depmod -a

A patch to force 'nvidia' to comply with the GPL licence [found on "linuxquestions"](#) written by "J\_W":

```
--- a/kernel/common/inc/nv-linux.h
+++ b/kernel/common/inc/nv-linux.h
@@ -1990,2 +1990,23 @@

+#if defined(CONFIG_HAVE_ARCH_PFN_VALID) || LINUX_VERSION_CODE <
+  ↪ KERNEL_VERSION(6,1,76)
+#  define nv_pfn_valid pfn_valid
+#else
+/* pre-6.1.76 kernel pfn_valid version without GPL
+  ↪ rcu_read_lock/unlock() */
+static inline int nv_pfn_valid(unsigned long pfn)
+{
+    struct mem_section *ms;
+
+    if (PHYS_PFN(PFN_PHYS(pfn)) != pfn)
+        return 0;
+
+    if (pfn_to_section_nr(pfn) >= NR_MEM_SECTIONS)
+        return 0;
+
+    ms = __pfn_to_section(pfn);
+    if (!valid_section(ms))
+        return 0;
+
+    return early_section(ms) || pfn_section_valid(ms, pfn);
+}
+#endif
+
+#endif /* _NV_LINUX_H_ */
--- a/kernel/nvidia/nv-mmap.c
+++ b/kernel/nvidia/nv-mmap.c
@@ -576,3 +576,3 @@
+        if (!IS_REG_OFFSET(nv, access_start, access_len) &&
-            (pfn_valid(PFN_DOWN(mmap_start))))
+            (nv_pfn_valid(PFN_DOWN(mmap_start))))
+        {
--- a/kernel/nvidia/os-mlock.c
+++ b/kernel/nvidia/os-mlock.c
@@ -102,3 +102,3 @@
+        if ((nv_follow_pfn(vma, (start + (i * PAGE_SIZE)), &pfn) < 0) ||
-            (!pfn_valid(pfn)))
+            (!nv_pfn_valid(pfn)))
+        {
@@ -176,3 +176,3 @@
-        if (pfn_valid(pfn))
```

```
+   if (nv_pfn_valid(pfn))
+   {
```

From the Archlinux 'nvidia-open' package: 'nvidia-open-tfm-ctx-aligned.patch'.

```
kernel-open/nvidia/libspdm_shash.c | 4 +--
1 file changed, 2 insertions(+), 2 deletions(-)
```

```
diff --git c/kernel-open/nvidia/libspdm_shash.c
↪ i/kernel-open/nvidia/libspdm_shash.c
index 10e9bff..d0ef6b2 100644
--- c/kernel-open/nvidia/libspdm_shash.c
+++ i/kernel-open/nvidia/libspdm_shash.c
@@ -87,8 +87,8 @@ bool lkca_hmac_duplicate(struct shash_desc *dst, struct
↪ shash_desc const *src)

    struct crypto_shash *src_tfm = src->tfm;
    struct crypto_shash *dst_tfm = dst->tfm;
-   char *src_ipad = crypto_tfm_ctx_aligned(&src_tfm->base);
-   char *dst_ipad = crypto_tfm_ctx_aligned(&dst_tfm->base);
+   char *src_ipad = crypto_tfm_ctx_align(&src_tfm->base,
↪ crypto_tfm_alg_alignmask(&src_tfm->base) + 1);
+   char *dst_ipad = crypto_tfm_ctx_align(&dst_tfm->base,
↪ crypto_tfm_alg_alignmask(&dst_tfm->base) + 1);
    int ss = crypto_shash_statesize(dst_tfm);
    memcpy(dst_ipad, src_ipad, crypto_shash_blocksize(src->tfm));
    memcpy(dst_ipad + ss, src_ipad + ss,
    ↪ crypto_shash_blocksize(src->tfm));
```

From the Archlinux 'nvidia-open' package: 'nvidia-open-gcc-ibt-sls.patch'.

```
--- a/src/nvidia-modeset/Makefile
+++ b/src/nvidia-modeset/Makefile
@@ -142,6 +142,7 @@ ifeq ($(TARGET_ARCH),x86_64)
    CONDITIONAL_CFLAGS += $(call TEST_CC_ARG, -fno-jump-tables)
    CONDITIONAL_CFLAGS += $(call TEST_CC_ARG,
    ↪ -mindirect-branch=thunk-extern)
    CONDITIONAL_CFLAGS += $(call TEST_CC_ARG, -mindirect-branch-register)
+   CONDITIONAL_CFLAGS += $(call TEST_CC_ARG, -mharden-sls=all)
endif

CFLAGS += $(CONDITIONAL_CFLAGS)
```

I also had other issues related to 'GPL' licence infractions by the 'nvidia' driver. I wanted to be able to debug it quickly so I just overrode the license declaration of 'nvidia/nv.c' to set it to 'GPL'.

```
MODULE_LICENSE("Dual MIT/GPL");
```

#### 1.4.4 Similarities between the functions of 'nvidia' and 'nvidia-open' driver

In the 'nvidia' driver, all the names of the functions of 'nv-kernel.o\_binary' have been removed and replaced with a placeholder name like `_nv02057rm`. However in the 'nvidia-open' driver,

the original names are still here, and some functions are very similar.

I used this finding to rename some of the function of 'nvidia' drivers version of 'nv-kernel.o\_binary' to their original name. Which help quite a bit to understand what is going on and how it works.

Example:

Figure 2: nvidia driver, function "\_nv000708rm" - Extract 1

```
***** FUNCTION *****  
undefined_nv000708rm()  
*****  
AL_1  
-RETURN->  
XREF[1]:  rn_init  
03e44460 f3 0f 1e fa ENDBR64  
03e44464 41 57 PUSH R15  
03e44466 31 c0 XOR EAX,EAX  
03e44468 b9 08 00 MOV ECX,0x8  
00 00  
03e4446d 48 c7 c6 MOV RSI,s_NVRM_GPU_04x:02x:02x:Rx:_Rm1_032d2618 = "N  
18 26 2d 03  
03e44474 41 56 PUSH R14  
03e44476 41 55 PUSH R13  
03e44478 41 54 PUSH R12  
03e4447a 49 89 fc MOV R12,RDI  
03e4447d 53 PUSH RDX  
03e4447e 48 81 ed SUB RBP,0xc0  
c0 00 00 00  
03e44485 41 86 54 MOV EDX,dword ptr [R12 + 0x14]  
24 14  
03e4448a 48 8d bd LEA RDI,[RBP + 0x80]  
80 00 00 00  
03e44491 45 0f b6 MOVZX R0D,byte ptr [R12 + 0x1a]  
4c 24 1a  
03e44497 45 0f b6 MOVZX R0D,byte ptr [R12 + 0x19]  
44 24 19  
03e4449d c7 45 3c MOV dword ptr [RBP + 0x3c],0x0  
00 00 00 00  
03e444a4 f3 48 ab STOSQ,REP RDI  
03e444a7 41 0f b6 MOVZX ECX,byte ptr [R12 + 0x18]  
4c 24 18  
03e444ad bf 02 00 MOV EDI,0x2  
00 00 00 00  
03e444b2 e8 a1 f2 CALL <EXTERNAL>:nv_printf  
0f 00  
03e444b7 41 0f b7 MOVZX EAX,word ptr [R12 + 0x1e]  
44 24 1e  
03e444bd 41 0f b6 MOVZX ECX,byte ptr [R12 + 0x18]  
4c 24 18  
03e444c3 48 c7 c6 MOV RSI,s_NVRM_GPU_04x:02x:02x:Rx:_Rm5_032d2648 = "N  
48 36 3d 03
```

```
1 /* WARNING: Function: __x86_indirect_thunk_rax replaced with injection: x86_indirect_thunk_rax */  
2  
3  
4 undefined8_nv000708rm(long *param_1)  
5  
6 {  
7     uint *puVar1;  
8     undefined4 uVar2;  
9     undefined uVar3;  
10    short sVar4;  
11    code *pcVar5;  
12    char cVar6;  
13    int iVar7;  
14    undefined4 uVar8;  
15    uint uVar9;  
16    undefined8 uVar10;  
17    long lVar11;  
18    long lVar12;  
19    undefined *pVar13;  
20    long lVar14;  
21    long lVar15;  
22    code **pcVar16;  
23    undefined8 *pVar17;  
24    char *pcVar18;  
25    long lVar19;  
26    uint uVar20;  
27    undefined uVar21;  
28    ulong uVar22;  
29    undefined8 *pVar23;  
30    long unaff_RBP;  
31    long *pVar24;  
32    code **pcVar25;  
33    byte bVar26;  
34    char *pcVar27;  
35    undefined8 uVar28;  
36  
37    bVar26 = 0;  
38    pVar24 = (long *) (unaff_RBP + -0xc0);  
39    uVar9 = *(undefined4 *) ((long) param_1 + 0x14);  
40    uVar21 = *(undefined *) ((long) param_1 + 0x1a);  
41    uVar2 = *(undefined *) ((long) param_1 + 0x19);  
42    *(undefined4 *) (unaff_RBP + -0x84) = 0;  
43    pVar23 = (undefined8 *) (unaff_RBP + -0x40);  
44    for (lVar19 = 8; lVar19 != 0; lVar19 = lVar19 + -1) {  
45        *pVar23 = 0;  
46        pVar23 = pVar23 + 1;  
47    }  
48    nv_printf(2, "NVRM: GPU %04x:02x:02x:Rx: RmInitAdapter\n", uVar8, *(undefined *) (param_1 + 3), uVar2  
49        , uVar21);  
50    *(uint *) (param_1 + 2) = *(uint *) (param_1 + 2) & 0xfffffff7;  
51    nv_printf(2, "NVRM: GPU %04x:02x:02x:Rx: RmSetupRegisters for 0x%x:0x%x\n",  
52        *(undefined4 *) ((long) param_1 + 0x14), *(undefined4 *) (param_1 + 3),  
53        *(undefined *) ((long) param_1 + 0x19), *(undefined *) ((long) param_1 + 0x1a),  
54        *(undefined2 *) ((long) param_1 + 0x1c), *(undefined2 *) ((long) param_1 + 0x1e));  
55    nv_printf(2, "NVRM: GPU %04x:02x:02x:Rx: pci config infos:\n",  
56        *(undefined4 *) ((long) param_1 + 0x14), *(undefined *) (param_1 + 3),  
57        *(undefined *) ((long) param_1 + 0x19), *(undefined *) ((long) param_1 + 0x1a));  
58    nv_printf(2, "NVRM: GPU %04x:02x:02x:Rx: registers look like: %p %p",  
59        *(undefined4 *) ((long) param_1 + 0x14), *(undefined *) (param_1 + 3),  
60        *(undefined *) ((long) param_1 + 0x19), *(undefined *) ((long) param_1 + 0x1a),  
61        *(undefined8 *) param_1[0x20], ((undefined8 *) param_1[0x20])[1]);  
62    pVar23 = (undefined8 *) param_1[0x21];  
63    if (pVar23 != (undefined8 *) 0x0) {  
64        nv_printf(2, "NVRM: GPU %04x:02x:02x:Rx: fb looks like: %p %p",  
65            *(undefined4 *) ((long) param_1 + 0x14), *(undefined *) (param_1 + 3),  
66            *(undefined *) ((long) param_1 + 0x19), *(undefined *) ((long) param_1 + 0x1a), pVar23,  
67            pVar23[1]);  
68    }  
69    pVar23 = (undefined8 *) param_1[0x20];  
70    if (pVar23[3] != 0) {  
71        nv038044m(0x374997, &CAT_00d00000);  
72        nv038009m(0xddd);  
73    }  
74    uVar10 = nv040427m(pVar23, pVar23[1], 1, 3);  
75    pVar23[3] = uVar10;  
76    pVar23[4] = uVar10;  
77    if (*(long *) (param_1[0x20] + 0x18) == 0) {  
78        uVar10 = 0x712;  
79        nv_printf(4, "NVRM: GPU %04x:02x:02x:Rx: Failed to map regs registers!!\n",  
80            *(undefined4 *) ((long) param_1 + 0x14), *(undefined *) (param_1 + 3),
```

Figure 3: nvidia driver, function "\_nv000708rm" - Extract 2

```
37    bVar26 = 0;  
38    pVar24 = (long *) (unaff_RBP + -0xc0);  
39    uVar9 = *(undefined4 *) ((long) param_1 + 0x14);  
40    uVar21 = *(undefined *) ((long) param_1 + 0x1a);  
41    uVar2 = *(undefined *) ((long) param_1 + 0x19);  
42    *(undefined4 *) (unaff_RBP + -0x84) = 0;  
43    pVar23 = (undefined8 *) (unaff_RBP + -0x40);  
44    for (lVar19 = 8; lVar19 != 0; lVar19 = lVar19 + -1) {  
45        *pVar23 = 0;  
46        pVar23 = pVar23 + 1;  
47    }  
48    nv_printf(2, "NVRM: GPU %04x:02x:02x:Rx: RmInitAdapter\n", uVar8, *(undefined *) (param_1 + 3), uVar2  
49        , uVar21);  
50    *(uint *) (param_1 + 2) = *(uint *) (param_1 + 2) & 0xfffffff7;  
51    nv_printf(2, "NVRM: GPU %04x:02x:02x:Rx: RmSetupRegisters for 0x%x:0x%x\n",  
52        *(undefined4 *) ((long) param_1 + 0x14), *(undefined4 *) (param_1 + 3),  
53        *(undefined *) ((long) param_1 + 0x19), *(undefined *) ((long) param_1 + 0x1a),  
54        *(undefined2 *) ((long) param_1 + 0x1c), *(undefined2 *) ((long) param_1 + 0x1e));  
55    nv_printf(2, "NVRM: GPU %04x:02x:02x:Rx: pci config infos:\n",  
56        *(undefined4 *) ((long) param_1 + 0x14), *(undefined *) (param_1 + 3),  
57        *(undefined *) ((long) param_1 + 0x19), *(undefined *) ((long) param_1 + 0x1a));  
58    nv_printf(2, "NVRM: GPU %04x:02x:02x:Rx: registers look like: %p %p",  
59        *(undefined4 *) ((long) param_1 + 0x14), *(undefined *) (param_1 + 3),  
60        *(undefined *) ((long) param_1 + 0x19), *(undefined *) ((long) param_1 + 0x1a),  
61        *(undefined8 *) param_1[0x20], ((undefined8 *) param_1[0x20])[1]);  
62    pVar23 = (undefined8 *) param_1[0x21];  
63    if (pVar23 != (undefined8 *) 0x0) {  
64        nv_printf(2, "NVRM: GPU %04x:02x:02x:Rx: fb looks like: %p %p",  
65            *(undefined4 *) ((long) param_1 + 0x14), *(undefined *) (param_1 + 3),  
66            *(undefined *) ((long) param_1 + 0x19), *(undefined *) ((long) param_1 + 0x1a), pVar23,  
67            pVar23[1]);  
68    }  
69    pVar23 = (undefined8 *) param_1[0x20];  
70    if (pVar23[3] != 0) {  
71        nv038044m(0x374997, &CAT_00d00000);  
72        nv038009m(0xddd);  
73    }  
74    uVar10 = nv040427m(pVar23, pVar23[1], 1, 3);  
75    pVar23[3] = uVar10;  
76    pVar23[4] = uVar10;  
77    if (*(long *) (param_1[0x20] + 0x18) == 0) {  
78        uVar10 = 0x712;  
79        nv_printf(4, "NVRM: GPU %04x:02x:02x:Rx: Failed to map regs registers!!\n",  
80            *(undefined4 *) ((long) param_1 + 0x14), *(undefined *) (param_1 + 3),
```

Figure 4: "nvidia-open" driver, function "RmInitAdapter" - Extract 1

```

FUNCTION
*****
undefined RmInitAdapter()
AL:1 <RETURN>
Stack[-0x30]:1 local_30
Stack[-0x40]:1 local_40
Stack[-0x48]:1 local_48
Stack[-0x70]:1 local_70
Stack[-0x78]:1 local_78

Stack[-0xb0]:8 local_b0
XREF[17]: 007a1c10(W),
007a1eac(*),
007a1ec9(0),
007a1ee6(*),
007a1efd(R),
007a1fc7(W),
007a1ffc(*),
007a202b(R),
007a211c(*),
007a2130(R),
007a23db(*),
007a2bf7(*),
007a2c98(*),
007a2ca8(W),
007a2f96(W),
007a2fba(*),
007a2fdb(R)

Stack[-0xb4]:4 local_b4
XREF[8]: 007a1c59(*),
007a29f0(0),
007a2c0e(W),
007a2cd1(W),
007a2ceb(*),
007a3160(*),
007a3167(W),
007a3191(R)

Stack[-0xb8]:4 local_b8
XREF[3]: 007a2f83(*),
007a2fec(R)

Stack[-0xbc]:4 local_bc
XREF[14]: 007a19db(W),
007a1c1f(*),
007a1c38(*),

4 undefined8 RmInitAdapter(long *param_1)
5 {
6
7   undefined uVar1;
8   short sVar2;
9   long lVar3;
10  long lVar4;
11  long lVar5;
12  long *plVar6;
13  char cVar7;
14  int iVar8;
15  undefined4 uVar9;
16  uint uVar10;
17  undefined8 uVar11;
18  undefined8 *pUVar12;
19  long lVar13;
20  long lVar14;
21  long lVar15;
22  undefined1 *pUVar16;
23  long lVar17;
24  code **ppcVar18;
25  long lVar19;
26  uint uVar20;
27  ulong uVar21;
28  undefined4 uVar22;
29  byte bVar23;
30  long local_e0;
31  long local_d0;
32  undefined4 local_bc;
33  uint local_b8;
34  int local_b4;
35  undefined8 local_b0;
36  undefined8 local_78;
37  undefined local_70 [40];
38  undefined local_48 [8];
39  undefined local_40 [16];
40
41  bVar23 = 0;
42  uVar9 = *(undefined4 *)((long)param_1 + 0x14);
43  uVar1 = *(undefined *)((long)param_1 + 0x1a);
44  local_bc = 0;
45  puVar12 = &local_78;
46  for (lVar19 = 8; lVar19 != 0; lVar19 = lVar19 + -1) {
47    *puVar12 = 0;
48    puVar12 = puVar12 + 1;
49  }
50  nv_printf(2,"NVRM: GPU %04x:%02x:%02x:%x: RmInitAdapter\n",uVar9,*(undefined *)(param_1 + 3),
51            *(undefined *)((long)param_1 + 0x19),uVar1);
52  *(uint *)(param_1 + 2) + *(uint *)(param_1 + 2) & 0xfffff;
53  nv_printf(2,"NVRM: GPU %04x:%02x:%02x:%x: RmSetupRegisters for 0x%x:0x%x\n",
54            *(undefined4 *)((long)param_1 + 0x14),*(undefined *)(param_1 + 3),
55            *(undefined *)((long)param_1 + 0x19),*(undefined *)((long)param_1 + 0x1a));
56  *(undefined *)((long)param_1 + 0x1c),*(undefined *)((long)param_1 + 0x1e));
57  nv_printf(2,"NVRM: GPU %04x:%02x:%02x:%x: pci config info:\n",
58            *(undefined4 *)((long)param_1 + 0x14),*(undefined *)(param_1 + 3),
59            *(undefined *)((long)param_1 + 0x19),*(undefined *)((long)param_1 + 0x1a));
60  nv_printf(2,"NVRM: GPU %04x:%02x:%02x:%x: registers look like: %p %p",
61            *(undefined4 *)((long)param_1 + 0x14),*(undefined *)(param_1 + 3),
62            *(undefined *)((long)param_1 + 0x19),*(undefined *)((long)param_1 + 0x1a),
63            *(undefined8 *)param_1[0x20],((undefined8 *)param_1[0x20])[1]);
64  puVar12 = (undefined8 *)param_1[0x27];
65  if (puVar12 != (undefined8 *)0x0) {
66    nv_printf(2,"NVRM: GPU %04x:%02x:%02x:%x: %b looks like: %p %p",
67              *(undefined4 *)((long)param_1 + 0x14),*(undefined *)(param_1 + 3),
68              *(undefined *)((long)param_1 + 0x19),*(undefined *)((long)param_1 + 0x1a),*puVar12,
69              puVar12[1]);
70  }
71  puVar12 = (undefined8 *)param_1[0x20];
72  if (puVar12[3] != 0) {
73    nvAssertFailedNoLog("sperture-map == NULL","arch/nvalloc/unix/src/osinit.c",0xac);
74    cVar7 = nvDbgBreakpointEnabled();
75    if (cVar7 != '\0') {
76      os_dbg_breakpoint();
77    }

```

Figure 5: "nvidia-open" driver, function "RmInitAdapter" - Extract 2

```

Listing: nvidiaopen_kernel.binary
*****
undefined RmInitAdapter()
AL:1 <RETURN>
Stack[-0x30]:1 local_30
Stack[-0x40]:1 local_40
Stack[-0x48]:1 local_48
Stack[-0x70]:1 local_70
Stack[-0x78]:1 local_78

Stack[-0xb0]:8 local_b0
XREF[17]: 007a1c10(W),
007a1eac(*),
007a1ec9(0),
007a1ee6(*),
007a1efd(R),
007a1fc7(W),
007a1ffc(*),
007a202b(R),
007a211c(*),
007a2130(R),
007a23db(*),
007a2bf7(*),
007a2c98(*),
007a2ca8(W),
007a2f96(W),
007a2fba(*),
007a2fdb(R)

Stack[-0xb4]:4 local_b4
XREF[8]: 007a1c59(*),
007a29f0(0),
007a2c0e(W),
007a2cd1(W),
007a2ceb(*),
007a3160(*),
007a3167(W),
007a3191(R)

Stack[-0xb8]:4 local_b8
XREF[3]: 007a2f83(*),
007a2fec(R)

40  bVar23 = 0;
41  uVar9 = *(undefined4 *)((long)param_1 + 0x14);
42  uVar1 = *(undefined *)((long)param_1 + 0x1a);
43  local_bc = 0;
44  puVar12 = &local_78;
45  for (lVar19 = 8; lVar19 != 0; lVar19 = lVar19 + -1) {
46    *puVar12 = 0;
47    puVar12 = puVar12 + 1;
48  }
49
50  nv_printf(2,"NVRM: GPU %04x:%02x:%02x:%x: RmInitAdapter\n",uVar9,*(undefined *)(param_1 + 3),
51            *(undefined *)((long)param_1 + 0x19),uVar1);
52  *(uint *)(param_1 + 2) + *(uint *)(param_1 + 2) & 0xfffff;
53  nv_printf(2,"NVRM: GPU %04x:%02x:%02x:%x: RmSetupRegisters for 0x%x:0x%x\n",
54            *(undefined4 *)((long)param_1 + 0x14),*(undefined *)(param_1 + 3),
55            *(undefined *)((long)param_1 + 0x19),*(undefined *)((long)param_1 + 0x1a));
56  *(undefined *)((long)param_1 + 0x1c),*(undefined *)((long)param_1 + 0x1e));
57  nv_printf(2,"NVRM: GPU %04x:%02x:%02x:%x: pci config info:\n",
58            *(undefined4 *)((long)param_1 + 0x14),*(undefined *)(param_1 + 3),
59            *(undefined *)((long)param_1 + 0x19),*(undefined *)((long)param_1 + 0x1a));
60  nv_printf(2,"NVRM: GPU %04x:%02x:%02x:%x: registers look like: %p %p",
61            *(undefined4 *)((long)param_1 + 0x14),*(undefined *)(param_1 + 3),
62            *(undefined *)((long)param_1 + 0x19),*(undefined *)((long)param_1 + 0x1a),
63            *(undefined8 *)param_1[0x20],((undefined8 *)param_1[0x20])[1]);
64  puVar12 = (undefined8 *)param_1[0x27];
65  if (puVar12 != (undefined8 *)0x0) {
66    nv_printf(2,"NVRM: GPU %04x:%02x:%02x:%x: %b looks like: %p %p",
67              *(undefined4 *)((long)param_1 + 0x14),*(undefined *)(param_1 + 3),
68              *(undefined *)((long)param_1 + 0x19),*(undefined *)((long)param_1 + 0x1a),*puVar12,
69              puVar12[1]);
70  }
71  puVar12 = (undefined8 *)param_1[0x20];
72  if (puVar12[3] != 0) {
73    nvAssertFailedNoLog("sperture-map == NULL","arch/nvalloc/unix/src/osinit.c",0xac);
74    cVar7 = nvDbgBreakpointEnabled();
75    if (cVar7 != '\0') {
76      os_dbg_breakpoint();
77    }

```

The function `_nv000708rm` and the function `'RmInitAdapter'` are so similar that we can believe they are the same / at least have the exact same role, so we can rename `_nv000708rm` to `'RmInitAdapter'`.

#### 1.4.5 Add traces in the driver

**Enabling the debugging flags in the Makefile** The original Makefile already contains some debug flags probably used by the 'nvidia' team, so I removed the condition to always have the most verbose flags.

```

+ #ifdef $(NV_BUILD_TYPE),release
+ # NVIDIA_CFLAGS += -UDEBUG -U_DEBUG -DNDEBUG
+ #endif

+ #ifdef $(NV_BUILD_TYPE),develop
+ # NVIDIA_CFLAGS += -UDEBUG -U_DEBUG -DNDEBUG -DNV_MEM_LOGGER
+ #endif

+ #ifdef $(NV_BUILD_TYPE),debug

```



```
NVIDIA_CFLAGS += -DDEBUG -D_DEBUG -UNDEBUG -DNV_MEM_LOGGER
+ #endif
```

Patching the nvidia kernel and C files Some times in the binary we can see calls to `nvDbg_Printf`:

```
nvDbg_Printf("src/kernel/gpu/bus/arch/maxwell/kern_bus_gm107.c",0x520,
            "kbusSetupBar2CpuAperture_GM107",4,
            "NVRM: BAR2 pteBase not initialized by
            ↪ fbPreInit_FERMI!\n");
```

However messages that should have been printed were never actually printed. The reason is the presence of two functions that filter what messages are actually printed or not.

The first one can be found in the file 'os-interface.c', a short extract :

```
int NV_API_CALL nv_printf(NvU32 debuglevel, const char *printf_format,
↪ ...)
{
    va_list arglist;
    int chars_written = 0;
    NvBool bForced = (NV_DBG_FORCE_LEVEL(debuglevel) == debuglevel);
    debuglevel = debuglevel & 0xff;

    // This function is protected by the "_nv_dbg_lock" lock, so it is
    ↪ still
    // thread-safe to store the print buffer in a static variable, thus
    // avoiding a problem with kernel stack size.
    static char buff[NV_PRINT_LOCAL_BUFF_LEN_MAX];

    /*
     * Print a message if:
     * 1. Caller indicates that filtering should be skipped, or
     * 2. debuglevel is at least cur_debuglevel for DBG_MODULE_OS (bits
     ↪ 4:5). Support for print
     * modules has been removed with DBG_PRINTF, so this check should be
     ↪ cleaned up.
     */
    if (bForced ||
        (debuglevel >= ((cur_debuglevel >> 4) & 0x3)))
    {
        size_t loglevel_length = 0, format_length = strlen(printf_format);
        size_t length = 0;
        const char *loglevel = "";

        switch (debuglevel)
        {
            case NV_DBG_INFO:          loglevel = KERN_DEBUG; break;
            case NV_DBG_SETUP:        loglevel = KERN_NOTICE; break;
            case NV_DBG_WARNINGS:     loglevel = KERN_WARNING; break;
            case NV_DBG_ERRORS:       loglevel = KERN_ERR; break;
```

```

    case NV_DBG_HW_ERRORS:    loglevel = KERN_CRIT; break;
    case NV_DBG_FATAL:       loglevel = KERN_CRIT; break;
}

```

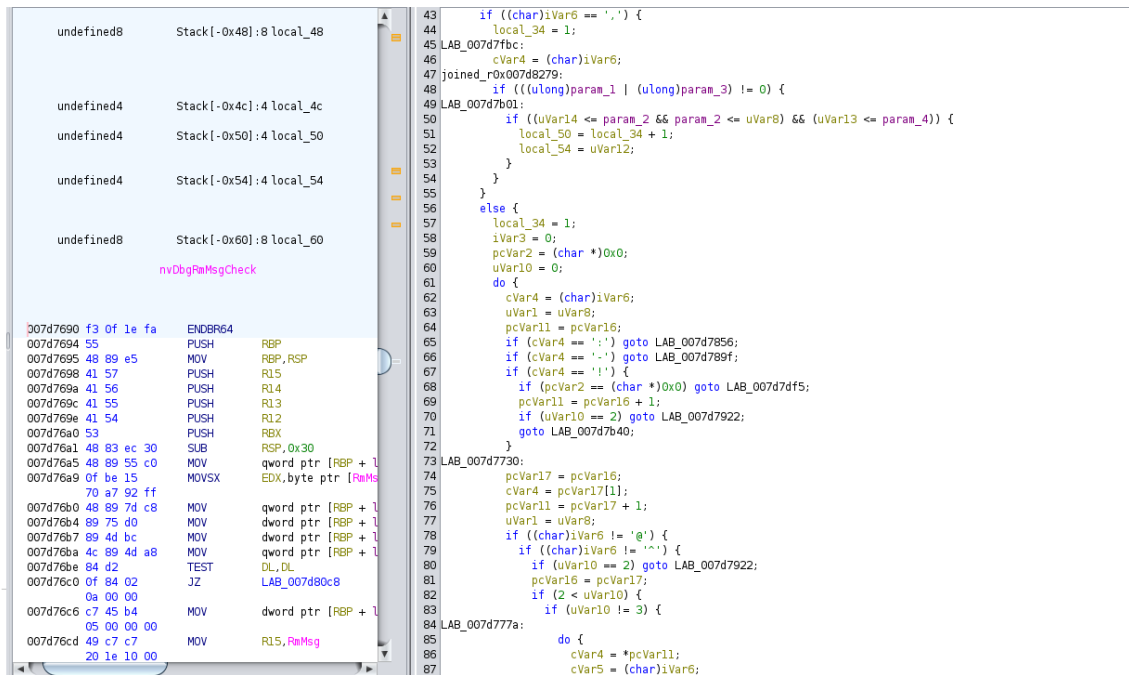
```

loglevel_length = strlen(loglevel);

```

and the function `nvDbgRmMsgCheck` that filter out some messages

Figure 6: Extract of the function `nvDbgRmMsgCheck`



I patched those functions not to filter out any messages. However I didn't end up getting any useful new debug log for my case.

**Enabling '-finstrument-functions' functionality in GCC** We are going to use the GCC functionality '-finstrument-functions' to print a message every time a function is called. You can read about this functionality on [balau82 blog](#). We will create a new folder, add its own build process and add the C file that will actually trace all function calls.

We modify our script 'install.sh' to add the build of our new C file responsible to trace all functions calls:

```

#!/bin/bash

make SYSSRC="/usr/src/linux" -f Makefile_trace
make SYSSRC="/usr/src/linux"

_extradir="/usr/lib/modules/$(</usr/src/linux/version)/extramodules"
install -Dt "${_extradir}" -m644 *.ko

# Force module to load even on unsupported GPUs

```

```

mkdir -p /usr/lib/modprobe.d
echo "options nvidia NVreg_OpenRmEnableUnsupportedGpus=1" >
↳ /usr/lib/modprobe.d/nvidia-open.conf
depmod -a

```

We create the dedicated makefile 'Makefile\_trace':

```

KERNEL_SOURCES := $(SYSSRC)

KERNEL_OUTPUT := $(KERNEL_SOURCES)
KBUILD_PARAMS :=

KERNEL_UNAME ?= $(shell uname -r)
KERNEL_MODLIB := /lib/modules/$(KERNEL_UNAME)
ifeq ($(KERNEL_SOURCES), $(KERNEL_MODLIB)/source)
    KERNEL_OUTPUT := $(KERNEL_MODLIB)/build
    KBUILD_PARAMS := KBUILD_OUTPUT=$(KERNEL_OUTPUT)
endif

CC ?= gcc
LD ?= ld
OBJDUMP ?= objdump
NV_KERNEL_MODULES ?= $(wildcard trace)

KBUILD_PARAMS += V=1
KBUILD_PARAMS += -C $(KERNEL_SOURCES) M=$(CURDIR)
KBUILD_PARAMS += NV_KERNEL_SOURCES=$(KERNEL_SOURCES)
KBUILD_PARAMS += NV_KERNEL_OUTPUT=$(KERNEL_OUTPUT)
KBUILD_PARAMS += NV_KERNEL_MODULES="trace"

.PHONY: modules module clean clean_conf test modules_install
modules clean modules_install:
    @$(MAKE) "LD=$(LD)" "CC=$(CC) -g -fno-stack-protector -no-pie"
    ↳ -rdynamic -O0" "OBJDUMP=$(OBJDUMP)" $(KBUILD_PARAMS) $@

```

And we create the kernel makefile 'trace/trace.Kbuild':

```

NVIDIA_OBJECTS = trace/trace.o

obj-m += trace/trace.o
nvidia-y := $(NVIDIA_OBJECTS)

$(call ASSIGN_PER_OBJ_CFLAGS, $(NVIDIA_OBJECTS))

```

The C file responsible to trace all the function calls 'trace/trace.c', without forgetting to prepend the profile function with `__attribute__((no_instrument_function))` to prevent infinite recursion:

```

#include <linux/printk.h>

__attribute__((no_instrument_function))
void __cyg_profile_func_enter (void *func, void *caller)

```

```

{
printk("NEO TRACE; ENTER: %pSR %pSR\n", func, caller);
}

__attribute__((no_instrument_function))
void __cyg_profile_func_exit (void *func, void *caller)
{
printk("NEO TRACE; EXIT: %pSR %pSR\n", func, caller);
}

```

In the file "nvidia/nvidia.Kbuild", add the line :

```
NVIDIA_OBJECTS += trace/trace.o
```

Repeat the process for every '\*.kbuild' file all the folder and subfolders, except, of course, for the file 'trace.Kbuild'.

In the original makefile 'Makefile', we add the GCC options '-finstrument-functions':

```

@$(MAKE) "LD=$(LD)" "CC=$(CC) -g -finstrument-functions -rdynamic -O0"
↪ "OBJDUMP=$(OBJDUMP)" $(KBUILD_PARAMS) $@

```

**Modifying nvidia C files** I also modified the nvidia C files to add some kernel logs. For example, to trace the parameters for the function `os_pci_write_dword`:

```

printk(KERN_ALERT "NEOWUTRAN os_pci_write_dword : try to write %u %u
↪ \n", offset, value);

```

Now let's use all those logs (and the knowledge acquired reading the code, reversing the binary and modifying the build chain) to fix the driver.

#### 1.4.6 Interesting differences between our traces of the 'nvidia-open' and 'nvidia'

Using all the logs we set up, right before the driver crash, we see calls to the function `os_pci_write_dword`:

```

os_pci_write_dword: write value 21 to offset 196
os_pci_write_dword: write value 1049603 to offset 4
os_pci_write_dword: write value 1049607 to offset 4
os_pci_write_dword: write value 63488 to offset 12
os_pci_write_dword: write value 8452096 to offset 12

```

For the 'nvidia-open' driver, we only see those references to the function `os_pci_write_dword`:

```

os_pci_write_dword: write value 21 to offset 196
os_pci_write_dword: write value 63488 to offset 12
os_pci_write_dword: write value 8452112 to offset 12

```

That is an interesting difference, no calls to offset 4 in the 'nvidia-open' driver.

#### 1.4.7 Patching the "nvidia" driver

Let's forbid any write operation to offset 4:

```

NV_STATUS NV_API_CALL os_pci_write_dword(
    void *handle,
    NvU32 offset,
    NvU32 value
)
{
    if (offset >= NV_PCIE_CFG_MAX_OFFSET)
        return NV_ERR_NOT_SUPPORTED;
+   printk(KERN_ALERT "NEOWUTRAN os_pci_write_dword : try to write %u %u
↪ \n",offset, value);
+   if (offset != 4){
        pci_write_config_dword( (struct pci_dev *) handle, offset, value);
+   }else{
+       return NV_ERR_NOT_SUPPORTED;
+   }
    return NV_OK;
}

```

I compiled and installed the driver, and the driver is now working as expected!

From the informations I could gather, offset 4 for `pci_write_config_dword` would represent the "command" field of the "PCI Configuration Headers" structure:

Figure 7: Code of the second function to patch

31		16 15		0		
Device ID		Vendor ID				00h
Status		Command				04h
Class Code			Revision ID			08h
BIST	Header Type	Lat. Timer	Cache Line S.			0Ch
Base Address Registers						10h
						14h
						18h
						1Ch
						20h
Cardbus CIS Pointer						24h
Subsystem ID			Subsystem Vendor ID			28h
Expansion ROM Base Address						2Ch
Reserved				Cap. Pointer		30h
Reserved						34h
Reserved						38h
Max Lat.	Min Gnt.	Interrupt Pin	Interrupt Line			3Ch

#### Command

By writing to this field the system controls the device, for example  
 ↳ allowing the device to access PCI I/O memory,

#### References:

- [unitn.it](http://unitn.it)
- [ibm](http://ibm.com)
- [Wikipedia](http://Wikipedia)

See below, and automated version to patch and install the 'nvidia' driver. Note that it does not include the patch to fix the 'nvidia' GPL violation that currently block the installation of the 'run' file provided on the official nvidia website:

```
#!/bin/bash
NVIDIA_RUN_FILE=${1?Indicate to the nvidia run file, example:}
↳ ". /NVIDIA-Linux-x86_64-545.29.06.run"
echo "$NVIDIA_RUN_FILE"
{
while true
```

```

do
  if [ -f "NVIDIA/kernel/nvidia/os-pci.c" ] ; then
    sed -i "s/\(pci_write_config_dword.*\)\/if (offset != 4){\1}else{return
    ↪ NV_ERR_NOT_SUPPORTED;}" NVIDIA/kernel/nvidia/os-pci.c
    echo "PATCHED ! "
    break
  fi
done
} &

```

```

SETUP_NOCHECK=1 bash "$NVIDIA_RUN_FILE" --target NVIDIA --ui=none
↪ --no-x-check

```

And version with the GPL violation fix that is required at the time of writing this (2024-03-01):

```

#!/bin/bash
NVIDIA_RUN_FILE=${1?Indicate to the nvidia run file, example:
↪ ".\/NVIDIA-Linux-x86_64-545.29.06.run"}
echo "$NVIDIA_RUN_FILE"
{
  while true
  do
    if [ -f "NVIDIA/kernel/nvidia/os-pci.c" ] && [ -f
    ↪ "NVIDIA/kernel/common/inc/nv-linux.h" ] && [ -f
    ↪ "NVIDIA/kernel/nvidia/nv-mmap.c" ] && [ -f
    ↪ "NVIDIA/kernel/nvidia/os-mlock.c" ]; then
      sed -i "s/\(pci_write_config_dword.*\)\/if (offset != 4){\1}else{return
      ↪ NV_ERR_NOT_SUPPORTED;}" NVIDIA/kernel/nvidia/os-pci.c
      gpl=$(cat <<EOF
--- a/kernel/common/inc/nv-linux.h
+++ b/kernel/common/inc/nv-linux.h
@@ -1990,2 +1990,23 @@
+
+#if defined(CONFIG_HAVE_ARCH_PFN_VALID) || LINUX_VERSION_CODE <
↪ KERNEL_VERSION(6,1,76)
+# define nv_pfn_valid pfn_valid
+#else
+/* pre-6.1.76 kernel pfn_valid version without GPL
↪ rcu_read_lock/unlock() */
+static inline int nv_pfn_valid(unsigned long pfn)
+{
+
+    struct mem_section *ms;
+
+    if (PHYS_PFN(PFN_PHYS(pfn)) != pfn)
+        return 0;
+
+    if (pfn_to_section_nr(pfn) >= NR_MEM_SECTIONS)
+        return 0;
+
+    ms = __pfn_to_section(pfn);

```

```

+         if (!valid_section(ms))
+             return 0;
+
+         return early_section(ms) || pfn_section_valid(ms, pfn);
+}
+#endif
+ #endif /* _NV_LINUX_H_ */
--- a/kernel/nvidia/nv-mmap.c
+++ b/kernel/nvidia/nv-mmap.c
@@ -576,3 +576,3 @@
+         if (!IS_REG_OFFSET(nv, access_start, access_len) &&
-             (pfn_valid(PFN_DOWN(mmap_start))))
+             (nv_pfn_valid(PFN_DOWN(mmap_start))))
+         {
--- a/kernel/nvidia/os-mlock.c
+++ b/kernel/nvidia/os-mlock.c
@@ -102,3 +102,3 @@
+         if ((nv_follow_pfn(vma, (start + (i * PAGE_SIZE)), &pfn) < 0) ||
-             (!pfn_valid(pfn)))
+             (!nv_pfn_valid(pfn)))
+         {
@@ -176,3 +176,3 @@
-         if (pfn_valid(pfn))
+         if (nv_pfn_valid(pfn))
+         {
EOF
)
  cd NVIDIA && echo "$gpl" | patch -p1
  echo "PATCHED ! "
  break
fi
done
} &

SETUP_NOCHECK=1 bash "$NVIDIA_RUN_FILE" --target NVIDIA --ui=none
↪ --no-x-check

```

## 1.5 Fixing the Windows Nvidia driver

I tried to apply the same patch in the nvidia Windows driver (nvlddmkm.sys). The equivalent function of `pci_write_config_dword` for Windows seems to be `HalSetBusDataByOffset`. But no matter how I tried to apply the patch to forbid call to `HalSetBusDataByOffset` when the parameter "offset" is equal to 4, it always result in a BSOD.

So we will need to dig deeper, so let's configure what is needed to use "windbg" on the Windows kernel.



## 1.5.1 Setting up remote windows kernel debugging on QubesOS

You can follow [the official Microsoft documentation](#).

However you will see in the section 'Supported network adapters' that the network adapter used by QubesOS (/Xen) is not in the [compatibility list](#). So first we will need to modify the virtual network adapter used for xen stubdom. Xen support the virtual version of the network adapter 'Intel 1000': 'e1000'. This adapter is in the compatibility list of Microsoft for remote kernel debugging.

So let's patch 'qemu-stubdom-linux-full-rootfs' to modify the virtual network adapter used for Windows qubes:

```
mkdir stubroot
cp /usr/libexec/xen/boot/qemu-stubdom-linux-full-rootfs
↪ stubroot/qemu-stubdom-linux-full-rootfs.gz
cd stubroot
gunzip qemu-stubdom-linux-full-rootfs.gz
cpio -i -d -H newc --no-absolute-filenames <
↪ qemu-stubdom-linux-full-rootfs
rm qemu-stubdom-linux-full-rootfs
nano init
```

After the line

```
# Extract network parameters and remove them from dm_args
```

add:

```
dm_args=$(echo "$dm_args" | sed 's/rtl8139/e1000/g')
```

Then execute:

```
find . -print0 | cpio --null -ov \
--format=newc | gzip -9 > ../qemu-stubdom-linux-full-rootfs
sudo mv ../qemu-stubdom-linux-full-rootfs /usr/libexec/xen/boot/
```

Alternatively, the following dom0 script "patch\_stubdom.sh" does all the previous steps:

```
#!/bin/bash
```

```
patch_rootfs(){
```

```
filename=${1?Filename is required}
```

```
cd ~/
```

```
sudo rm -R "patched_$filename"
```

```
mkdir "patched_$filename"
```

```
cp /usr/libexec/xen/boot/$filename "patched_$filename/$filename.gz"
```

```
cp /usr/libexec/xen/boot/$filename "$filename.original"
```

```
cd patched_$filename
```

```
gunzip $filename.gz
```

```

    cpio -i -d -H newc --no-absolute-filenames < "$filename"
    sudo rm $filename

patch_string=$(cat <<'EOF'
dm_args=$(echo "$dm_args" | sed 's/rtl8139/e1000/g')
\# Extract network parameters and remove them from dm_args
EOF
)

awk -v r="$patch_string" '{gsub(/^\# Extract network parameters and remove
↪ them from dm_args/,r)}1' init > init2
cp init /tmp/init_.$filename
mv init2 init
chmod +x init

find . -print0 | cpio --null -ov \
--format=newc | gzip -9 > ../$filename.patched
sudo cp ../$filename.patched /usr/libexec/xen/boot/$filename

cd ~/

}

patch_rootfs "qemu-stubdom-linux-rootfs"
patch_rootfs "qemu-stubdom-linux-full-rootfs"

```

```
echo "stubdom have been patched."
```

The following command will display a message informing you if your network card is compatible with remote windows kernel debugging or not:

```
"C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\kdnet.exe"
```

After patching the stubdom, it should display that the network card is indeed compatible with remote windows kernel debugging.

We can now continue following the Microsoft documentation.

I also watched some part of [this video](#) to be sure I didn't forget any steps.

In my setup, I have two qubes. One 'Windbg' qube, that runs 'windbg' to debug the remote kernel. And 'Windows10' qube, that is my qube with the GPU passthrough and the buggy nvidia driver that need to be fixed.

Those two qubes need to communicate between each other on the network. Connect them to the same 'netvm', let's say, 'sys-firewall'.

We need to configure nftable to allow communication between those two. Below example command I ran on 'sys-firewall'.

```
sudo nft add rule ip qubes custom-forward ip saddr 10.137.0.42 ip daddr
↪ 10.137.0.79 udp dport 54444 ct state new,established counter accept
```

And for convenience, on my 'Windbg' qube I also created a shortcut that launch windbg with all the needed parameters:

```
"C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\windbg.exe" -k
↪ net:port=54444,key=XXXXXXXXXXXXXXXXXXXX
```

(and the corresponding configuration in the "Windows10" qube was:

```
bcdedit /dbgsettings net hostip:10.137.0.79 port:54444
bcdedit /set "{dbgsettings}" busparams 0.6.0
)
```

You should now be able to debug the windows kernel remotely using QubesOS.

### 1.5.2 Creating a simplified driver to be sure the crash is not influenced by anything other than "nvlddmkm.sys"

The BSoD that led to all of this was about 'nvlddmkm.sys'. So that is the file I started debugging. However due to the massive number of files included, I wasn't so sure that the bug only related to 'nvlddmkm.sys'.

So I decided to remove as many files as possible from the driver files.

I started from that:

Figure 8: Number of files and size of the official Windows nvidia driver

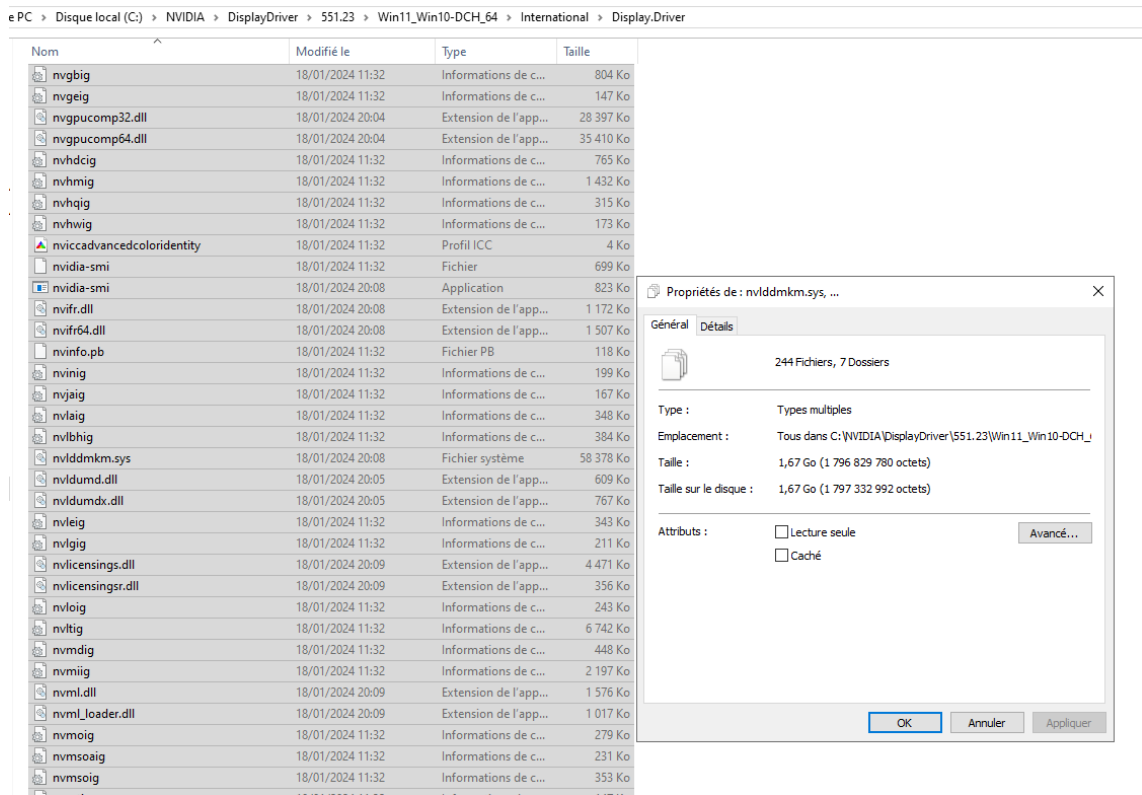
The screenshot shows a Windows File Explorer window with the address bar set to 'C:\Disque local (C:) > NVIDIA > DisplayDriver > 551.23 > Win11\_Win10-DCH\_64 > International'. The main pane displays a list of files and folders. To the right, the 'Propriétés de : HDAudio, ...' window is open, showing the 'Général' tab with the following information:

- 1 421 Fichiers, 113 Dossiers
- Type : Types multiples
- Emplacement : Tous dans C:\NVIDIA\DisplayDriver\551.23\Win11\_Win10-DCH\_64
- Taille : 2,51 Go (2 699 555 606 octets)
- Taille sur le disque : 2,51 Go (2 702 241 792 octets)
- Attributs :  Lecture seule,  Caché

Nom	Modifié le	Type	Taille
Display.Driver	25/02/2024 19:17	Dossier de fichiers	
Display.Nview	25/02/2024 19:17	Dossier de fichiers	
Display.Optimus	25/02/2024 19:17	Dossier de fichiers	
Display.Update	25/02/2024 19:17	Dossier de fichiers	
FrameViewSDK	25/02/2024 19:17	Dossier de fichiers	
GFXExperience	25/02/2024 19:18	Dossier de fichiers	
GFXExperience.NvStreamSrv	25/02/2024 19:18	Dossier de fichiers	
HDAudio	25/02/2024 19:18	Dossier de fichiers	
MSVCRT	25/02/2024 19:18	Dossier de fichiers	
nodejs	25/02/2024 19:18	Dossier de fichiers	
NvBackend	25/02/2024 19:18	Dossier de fichiers	
NvContainer	25/02/2024 19:17	Dossier de fichiers	
NV12	25/02/2024 19:18	Dossier de fichiers	
NvModuleTracker	25/02/2024 19:18	Dossier de fichiers	
NVPCF	25/02/2024 19:18	Dossier de fichiers	
NvTelemetry	25/02/2024 19:18	Dossier de fichiers	
NvVAD	25/02/2024 19:18	Dossier de fichiers	
NvVHCI	25/02/2024 19:17	Dossier de fichiers	
PhysX	25/02/2024 19:18	Dossier de fichiers	
PPC	25/02/2024 19:18	Dossier de fichiers	
ShadowPlay	25/02/2024 19:18	Dossier de fichiers	
ShieldWirelessController	25/02/2024 19:18	Dossier de fichiers	
Update.Core	25/02/2024 19:18	Dossier de fichiers	
EULA	18/01/2024 11:32	Document texte	26 Ko
ListDevices	18/01/2024 11:32	Document texte	474 Ko
setup.cfg	18/01/2024 11:32	Fichier CFG	57 Ko
setup	18/01/2024 20:13	Application	641 Ko

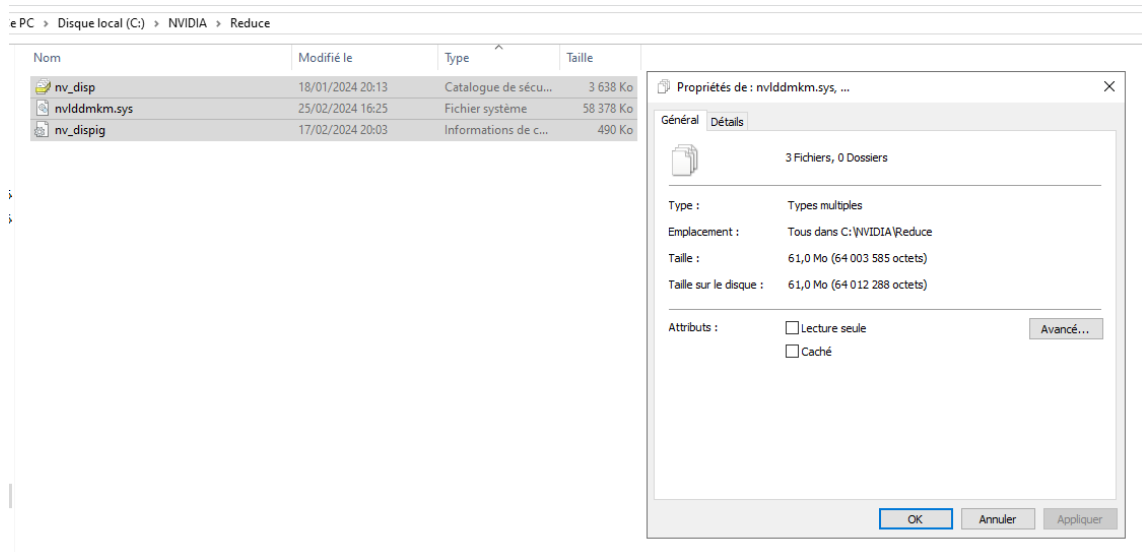
and a look inside the 'Display.Driver' folder:

Figure 9: Number of files and size of the 'Display.Driver' part of official Windows nvidia driver



And I successfully removed nearly all the files. I ended up with that:

Figure 10: Number of files and size of my custom driver folder, for easier debugging (It is a subpart of 'Display.Driver')



The file 'nv\_disp.inf' is heavily modified, but still contains few thousand lines. The whole thing is available in attachment:

📎 nvvidia debug driver

I the installed the driver from command line:

```
pnputil /add-driver nv_disp.inf /install
```

And then I manually launched the driver. Using the Windows GUI 'Device Manager', I selected the GPU, clicked to update its driver and manually selected the newly installed driver from the list.

And the driver crash, causing the same Blue Screen of Death as before. So now I am sure the bug is contained inside 'nvlddmkm.sys'. So let's debug it.

### 1.5.3 Debugging and modifying 'nvlddmkm.sys'

I only used the 'windbg' GUI and some pretty basic command, most of them were:

```
sxe ld nvlddmkm.sys
bp function_name "r; g"
bp function_name
bc X
```

I first tried to monitor when HalSetBusDataByOffset was called, because it is the Windows equivalent of pci\_write\_config\_dword. However, it seems the driver always crash before even calling this function.

Then I guessed, maybe a call to HalGetBusDataByOffset trigger the same problem as for pci\_write\_config\_dword on Linux. I monitored the calls to HalGetBusDataByOffset and indeed, hundreds or thousands of calls to this function are made before the driver crash.

Figure 11: List of functions that calls either HalGetBusDataByOffset or HalSetBusDataByOffset

```
*****
*          POINTER to EXTERNAL FUNCTION          *
*****
undefined HalSetBusDataByOffset()
AL:1 <RETURN>
61 HalSetBusDataByOffset <<not bound>>
PTR_HalSetBusDataByOffset_140bdc010 XREF[3]: FUN_1400cfb40:1400cfd0b(R),
FUN_1400cfd30:1400cfe7b(R),
FUN_1400cff20:1400d00e7(R)

140bdc010 88 7f a0      addr  HAL.DLL::HalSetBusDataByOffset
03 00 00
00 00

*****
*          POINTER to EXTERNAL FUNCTION          *
*****
undefined HalGetBusDataByOffset()
AL:1 <RETURN>
25 HalGetBusDataByOffset <<not bound>>
PTR_HalGetBusDataByOffset_140bdc018 XREF[4]: FUN_1400cf310:1400cf526(R),
FUN_1400cf570:1400cf745(R),
FUN_1400cf760:1400cf937(R),
FUN_1400cf950:1400cfb24(R)

140bdc018 a0 7f a0      addr  HAL.DLL::HalGetBusDataByOffset
03 00 00
00 00
140bdc020 00          ??    00h
```

Figure 12: Extract of the function that will call HalGetBusDataByOffset, with a configurable offset parameter, and asking for a WORD result

```

1400cfab 83 e5 07 AND     EBP,0x7
1400cfae 40 0f b6 d7 MOVZX  EDI,DIL
1400cfb0 c1 e5 05 SHL     EBP,0x5
1400cfb5 4c 8d 4c LEA    R9=>local_res8,[RSP + 0x60]
24 60
1400cfba 83 e6 1f AND     ESI,0x1f
1400cfbd c7 44 24 MOV    dword ptr [RSP + local_30],0x2
28 02 00
00 00
1400cfb1 0b ee OR     EBP,ESI
1400cfb7 44 89 7c MOV    dword ptr [RSP + local_38],R15D
24 20
1400cfbc 44 8b c5 MOV    R8D,EBP
1400cfbf b9 04 00 MOV    ECX,0x4
00 00
1400cf24 ff 15 ee CALL   qword ptr [->HAL_DLL::HalGetBusDataByOffset] + 03a07fa
c4 b0 00
1400cf2a 0f b7 44 MOVZX  EAX,word ptr [RSP + local_res8]
24 60
1400cf2f e9 da fe JMP    LAB_1400cfae
ff ff

LAB_1400cf34 XREF[1]: 140fa21
1400cf34 cc INT3
1400cf35 cc ?? CCh
1400cf36 cc ?? CCh
1400cf37 cc ?? CCh
1400cf38 cc ?? CCh
1400cf39 cc ?? CCh

```

```

37 (* (int *) (Var4 + 0x48e8) == 3) || (bVar1 = KeGetCurrentIrql(), bVar1 < bVar6) {
38   iVar2 = (**code *) (Var3 + 0x38) * (undefined8 *) (Var3 + 8), 0, local_res8, param_2, 2);
39   if (iVar2 == 0) {
40     return 0xffff;
41   }
42   return local_res8[0];
43 }
44 bVar1 = KeGetCurrentIrql();
45 if (bVar1 < bVar6) {
46   return 0xffff;
47 }
48 FUN_1400b63a0(0xed2ad0a, 0x16b10000);
49 uVar5 = 0x1801;
50 goto LAB_1400cfa04;
51 }
52 uVar7 = uVar7 + 1;
53 while (uVar7 < 0x20);
54 if ((param_2 < 0x100) || ((*char *) (Var3 + 0x53) != '\0')) {
55   if ((*char *) (longlong *) (DAT_140ed370 + 0x270) + 0x4a) == '\0') ||
56     (iVar2 = FUN_140218a50(1, iVar2 != 2)) {
57     HalGetBusDataByOffset
58       (4, param_1 & 0fff, (uint) (param_1 >> 0x10) & 7) << 5 | (uint) (param_1 >> 8) & 0x1f
59         , local_res8, param_2, 2);
60     return local_res8[0];
61   }
62   FUN_1400b63a0(0xed2ad0a, 0x39e0000);
63   FUN_1400b6de0(0x3ed);
64 }
65 else {

```

Figure 13: Extract of the function that will call `HalGetBusDataByOffset`, with a configurable offset parameter, and asking for a DWORD result

```

1400cf90 83 e5 07 AND     EBP,0x7
1400cf91 40 0f b6 d7 MOVZX  EDI,DIL
1400cf95 c1 e5 05 SHL     EBP,0x5
1400cf98 4c 8d 4c LEA    R9=>local_res8,[RSP + 0x60]
24 60
1400cf9d 83 e6 1f AND     ESI,0x1f
1400cf9f c7 44 24 MOV    dword ptr [RSP + local_30],0x4
28 04 00
00 00
1400cf9b 0b ee OR     EBP,ESI
1400cf9d 44 89 7c MOV    dword ptr [RSP + local_38],R15D
24 20
1400cf9f 44 8b c5 MOV    R8D,EBP
1400cf9f b9 04 00 MOV    param_1,0x4
00 00
1400cf97 ff 15 db CALL   qword ptr [->HAL_DLL::HalGetBusDataByOffset] + 03a07fa
c6 b0 00
1400cf94 0f b6 44 MOVZX  EAX,dword ptr [RSP + local_res8]
1400cf91 e9 d8 fe JMP    LAB_1400cf8e
ff ff

LAB_1400cf96 XREF[1]: 140fa21bc(*)
1400cf96 cc INT3
1400cf97 cc ?? CCh
1400cf98 cc ?? CCh
1400cf99 cc ?? CCh
1400cf9a cc ?? CCh
1400cf9b cc ?? CCh
1400cf9c cc ?? CCh

```

```

41 (* (char *) (longlong *) (Var4 + 0x48e8) + 0x1747 != '\0') ||
42 (* (int *) (Var4 + 0x48e8) == 3) || (bVar1 = KeGetCurrentIrql(), bVar1 < bVar6) {
43   iVar2 = (**code *) (Var3 + 0x38) * (undefined8 *) (Var3 + 8), 0, local_res8,
44     (ulonglong) param_2 & 0xffffffff, 4);
45   if (iVar2 == 0) {
46     return 0xffffffff;
47   }
48   return local_res8[0];
49 }
50 bVar1 = KeGetCurrentIrql();
51 if (bVar1 < bVar6) {
52   return 0xffffffff;
53 }
54 FUN_1400b63a0(0xed2ad0a, 0x16b10000);
55 uVar5 = 0x1801;
56 goto LAB_1400cf814;
57 }
58 uVar7 = uVar7 + 1;
59 while (uVar7 < 0x20);
60 if ((uint) param_2 < 0x100 || ((*char *) (Var3 + 0x53) != '\0')) {
61   if ((*char *) (longlong *) (DAT_140ed370 + 0x270) + 0x4a) == '\0') ||
62     (iVar2 = FUN_140218a50(1, iVar2 != 2)) {
63     HalGetBusDataByOffset
64       (4, param_1 & 0fff, (uint) (param_1 >> 0x10) & 7) << 5 | (uint) (param_1 >> 8) & 0x1f
65         , local_res8, (uint) param_2, 4);
66     return local_res8[0];
67   }
68   FUN_1400b63a0(0xed2ad0a, 0x39e0000);
69   FUN_1400b6de0(0x39f);
70 }

```

Figure 14: Extract of the function that will call `HalGetBusDataByOffset`, with a configurable offset parameter, and asking for a BYTE result

```

1400cf71c 83 e5 07 AND     EBP,0x7
1400cf71f 40 0f b6 d7 MOVZX  EDI,DIL
1400cf723 c1 e5 05 SHL     EBP,0x5
1400cf726 4c 8d 4c LEA    R9=>local_res8,[RSP + 0x60]
24 60
1400cf72b 83 e6 1f AND     ESI,0x1f
1400cf72e c7 44 24 MOV    dword ptr [RSP + local_30],0x1
28 01 00
00 00
1400cf736 0b ee OR     EBP,ESI
1400cf738 44 89 7c MOV    dword ptr [RSP + local_38],R15D
24 20
1400cf73d 44 8b c5 MOV    R8D,EBP
1400cf73d b9 04 00 MOV    ECX,0x4
00 00
1400cf75 ff 15 cd CALL   qword ptr [->HAL_DLL::HalGetBusDataByOffset] + 03a07fa
c8 b0 00
1400cf74b 0f b6 44 MOVZX  EAX,byte ptr [RSP + local_res8]
1400cf750 e9 d6 fe JMP    LAB_1400cf62b
ff ff

LAB_1400cf755 XREF[1]: 140fa21b0(*)
1400cf755 cc INT3
1400cf756 cc ?? CCh
1400cf757 cc ?? CCh
1400cf758 cc ?? CCh
1400cf759 cc ?? CCh
1400cf75a cc ?? CCh
1400cf75b cc ?? CCh

```

```

57 }
58 if (param_6 != (undefined2 *) 0x0) {
59   *param_5 = (short) (uint) local_res10[0] >> 0x10;
60   return uVar9;
61 }
62 return uVar9;
63 }
64 break;
65 }
66 uVar8 = uVar8 + 1;
67 while (uVar8 < 0x20);
68 if ((*char *) (longlong *) (DAT_140ed370 + 0x270) + 0x4a) == '\0') ||
69   (iVar4 = FUN_140218a50(1, iVar4 != 2)) {
70   uVar6 = HalGetBusDataByOffset
71     (4, (ulonglong) param_2, (param_4 & 7) << 5 | param_3 & 0x1f, local_res10, 0, 4);
72   if ((uVar6 & 0xffffffff) != 0) {
73     if (param_5 != (undefined2 *) 0x0) {
74       *param_5 = (short) local_res10[0];
75     }
76   }
77   if (param_6 != (undefined2 *) 0x0) {
78     *param_6 = (short) (uint) local_res10[0] >> 0x10;
79   }
80   return uVar9;
81 }

```

Figure 15: Extract of the function that will call `HalGetBusDataByOffset`, with the offset parameter defined to 0, and asking for a DWORD result

```

1400cf50c 41 83 e4 1f AND     R1D,0x1f
1400cf510 c7 44 24 MOV    dword ptr [RSP + local_48],0x0
20 00 00
00 00
1400cf518 45 0b f4 OR     R1D,R1D
1400cf51b 41 8b 05 MOV    param_2,R1D
1400cf51e 45 0b c6 MOV    param_3,R1D
1400cf521 b9 04 00 MOV    param_1,0x4
00 00
1400cf526 ff 15 cc CALL   qword ptr [->HAL_DLL::HalGetBusDataByOffset] + 03a07fa
ca b0 00
1400cf52c a9 fd ff TEST   EAX,0xffffffff
ff ff
1400cf531 0f 84 ff JZ     LAB_1400cf36e
fe ff ff
1400cf537 48 8b 8c MOV    param_1,qword ptr [RSP + param_5]
24 90 00
00 00
1400cf53f 8b 44 24 78 MOV    EAX,dword ptr [RSP + local_res10]
1400cf543 48 85 c9 TEST   param_1,param_1
1400cf546 74 03 JZ     LAB_1400cf54b
1400cf548 66 89 01 MOV    word ptr [param_1],AX

LAB_1400cf54b XREF[1]: 1400cf546(j)

```

I then tried to patch the calls to `HalGetBusDataByOffset` to forbid any call related to offset 4.

But the driver was still crashing. I then decided to patch the calls to `HalGetBusDataByOffset` in multiple ways:

- Only allow some offset
- Only forbid some offset
- Forbid any call

But the driver was still crashing. — As a note, for my patches I overrode some part of the PE that I knew was not used (“code cave”), and eventually modified the section protection with `LordPE` —

At this point I was wondering if there was not any other kernel call that potentially read or write the pci config. And I was also wondering if I didn’t misunderstood something. However I did the previous steps correctly and I was 100% sure that the crash was related to manipulating pci config. So I spend some time reading Microsoft documentation to list all the possible kernel call that will interact with pci configuration. But I was left with only `HalSetBusDataByOffset` and `HalGetBusDataByOffset`, other kernel functions exist, but are not used by the nvidia driver.

So, I guessed that if I am 100% sure that I did the previous steps correctly, then the only possibility I was able to come up with was: Somewhere in the code, `HalGetBusDataByOffset` is called, if the result retrieve does not match what the nvidia driver was expecting, the driver commit seppuku (voluntary or accidentally). So either I needed to intercept calls to `HalGetBusDataByOffset` and made sure that the value returned by the function was the expected value. Or, find where the problematic code logic is implemented and disable it completely.

I also verified in the Linux driver, there is no equivalent, I didn’t see those thousands of calls to kernel function to read the pci configuration. So I guessed that this behaviour on Windows was not vital to the driver and decided to monitor the call stacks leading to a call to `HalGetBusDataByOffset`, and then start from as close as possible from one of the driver entry points to try to understand why `HalGetBusDataByOffset` got called and by what.

Figure 16: Extract of the list of the caller functions for the function calling the problematic `HalGetBusDataByOffset` - Part 1

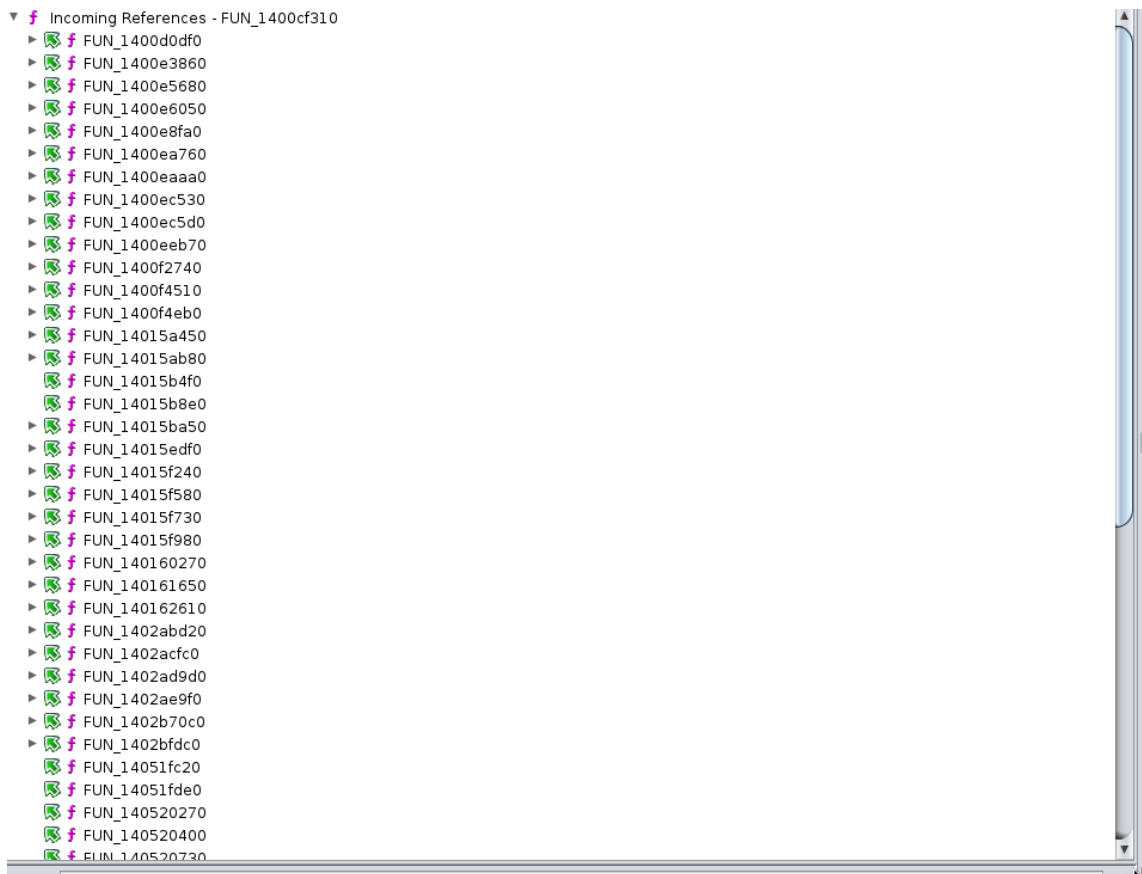
```

FUN_1400cf310                                     XREF[93]: 1400cf510(W)
                                                    FUN_1400d0df0:1400d0e7c(c),
                                                    FUN_1400ec530:1400ec560(c),
                                                    FUN_1400ec5d0:1400ec795(c),
                                                    FUN_1400ec5d0:1400ec7fb(c),
                                                    FUN_1400ec5d0:1400ec85d(c),
                                                    FUN_1400eeb70:1400eec1e(c),
                                                    FUN_14015ab80:14015ac15(c),
                                                    FUN_14015ba50:14015baef(c),
                                                    FUN_14015ba50:14015bb34(c),
                                                    FUN_14015edf0:14015eeca(c),
                                                    FUN_14015f730:14015f8d8(c),
                                                    FUN_14015f980:14015fa26(c),
                                                    FUN_140162610:140162728(c),
                                                    FUN_1402abd20:1402abf6f(c),
                                                    FUN_1402acf0:1402ad013(c),
                                                    FUN_1402acf0:1402ad55a(c),
                                                    FUN_1402ad9d0:1402adbc4(c),
                                                    FUN_1402ad9d0:1402adc1e(c),
                                                    FUN_1402ae9f0:1402aebb6(c),
                                                    FUN_1402ae9f0:1402aec27(c), [more]

1400cf310 48 89 5c      MOV     qword ptr [RSP + local_res8],RBX
          24 08

```

Figure 17: Extract of the list of the caller functions for the function calling the problematic `HalGetBusDataByOffset` - Part 2



I found that it was the section related to `nvDumpConfig` of "nvlddmkm.sys" that end up calling `HalGetBusDataByOffset` and later lead to the crash.

Figure 18: Windbg GUI showing the callstacks after a breakpoint on `HalGetBusDataByOffset`



## Calls

```

Raw args  Func info  Source  Addr  Headings  Nonvolatile regs  Fr
-----
nt!HalGetBusDataByOffset
nvlddmkm+0xcf52c
nvlddmkm+0x15fa2b
nvlddmkm+0x1617ad
nvlddmkm+0x160498
nvlddmkm+0x15e47c
nvlddmkm+0xc4288
nvlddmkm+0xc1e72
nvlddmkm!nvDumpConfig+0x64a6ed
nvlddmkm!nvDumpConfig+0x5c1fbe
nvlddmkm!nvDumpConfig+0x68fac2
nvlddmkm!nvDumpConfig+0x6a61ab
nvlddmkm!nvDumpConfig+0x48ad9b
dxgkrnl!DpiDxgkDdiStartDevice+0x6a
dxgkrnl!DpiFdoStartAdapter+0x58e
dxgkrnl!DpiFdoStartAdapterThreadImpl+0x308
dxgkrnl!DpiFdoStartAdapterThread+0x30
nt!PspSystemThreadStartup+0x55
nt!KiStartSystemThread+0x28

```

Figure 19: Ghidra: function "nvlddmkm!nvDumpConfig+0x48ad9b"

```

nvlddmkm.sys
Decompile: FUN_141273660 - (nvlddmkm.sys)

undefined FUN_141273660(undefined param_1, undefined param_2, undefined param_3)
AL:1 <SETUP>
CL:1 param_1
DL:1 param_2
R8B:1 param_3
R9B:1 param_4
Stack[0x28]:8 param_5
Stack[0x18]:8 local_18
FUN_141273660
XREF[1]: 14127366b(R)
XREF[4]: FUN_1400b3ab8:1400b3c4ff(*), FUN_1400b3ab8:1400b3c59(*), 140b13460(*), 1410429a0(*)

141273660 48 83 ec 38 SUB RSP,0x38
141273664 49 8b 40 10 MOV RAX,qword ptr [param_3 + 0x10]
141273668 4c 8b d1 MOV R10,param_1
14127366b 48 8b 4c MOV param_1,qword ptr [RSP + param_5]
141273670 48 89 05 MOV qword ptr [DAT_140ec890],RAX
141273677 49 8b 40 18 MOV RAX,qword ptr [param_3 + 0x18]
14127367b 48 89 05 MOV qword ptr [DAT_140ec898],RAX
141273682 49 8b 40 20 MOV RAX,qword ptr [param_3 + 0x20]
141273686 48 89 05 MOV qword ptr [DAT_140ec890],RAX
14127368d 49 8b 40 28 MOV RAX,qword ptr [param_3 + 0x28]
141273691 48 89 05 MOV qword ptr [DAT_140ec898],RAX
141273698 49 8b 40 30 MOV RAX,qword ptr [param_3 + 0x30]
14127369c 48 89 05 MOV qword ptr [DAT_140ec878],RAX
1412736a3 49 8b 40 38 MOV RAX,qword ptr [param_3 + 0x38]
1412736a7 48 89 05 MOV qword ptr [DAT_140ec8a0],RAX
1412736ae 49 8b 40 40 MOV RAX,qword ptr [param_3 + 0x40]
1412736b2 48 89 05 MOV qword ptr [DAT_140ec8a8],RAX
1412736b9 49 8b 40 48 MOV RAX,qword ptr [param_3 + 0x48]
1412736bd 48 89 05 MOV qword ptr [DAT_140ec860],RAX
1412736c4 49 8b 40 50 MOV RAX,qword ptr [param_3 + 0x50]
1412736c8 48 89 05 MOV qword ptr [DAT_140ec868],RAX
1412736cf 49 8b 40 58 MOV RAX,qword ptr [param_3 + 0x58]
1412736d3 48 89 05 MOV qword ptr [DAT_140ec870],RAX
1412736da 48 8b 05 MOV RAX,qword ptr [DAT_140f97088]
1412736e1 48 89 4c MOV qword ptr [RSP + local_18,param_1]
1412736e6 49 8b ca MOV param_1,R10
1412736e9 ff 40 CALL RAX
1412736eb ff 1f NOP dword ptr [RAX]
1412736ee 48 83 c4 38 ADD RSP,0x38
1412736f2 c3 RET
1412736f3 cc CCH

```

I then went down the calls stacks and found that, without surprise, some functions gather data about the OS, OS configuration, etc .... And inside those functions, there are multiple

calls to others function that end up (function that calls a function that calls a function ...that led to conditionally calling HalGetBusDataByOffset).

The function nvDumpConfig exists in the Linux driver too, and perform some of the same checks as the nvDumpConfig function on Windows.

For the function that ended up calling HalGetBusDataByOffset, they were conditionally called, and I decided to make sure they were never reached by replacing the conditional call by an unconditional call. I repeated the debug process until I was able to get out of that function without ever calling HalGetBusDataByOffset.

Figure 20: First function I needed to patch: What call it, and a small extract of the beginning of the function

The screenshot shows a debugger window with two panes. The left pane displays assembly code for the function FUN\_1400c3230, starting with a PUSH RBP instruction at address 1400c3230. The right pane shows the decompiled pseudo-C code, starting with 'ulonglong local\_40;' and 'lVar3 = DAT\_1400e4370;'. The assembly code includes instructions like MOV, CALL, SUB, MOV, XOR, and MOV, with various registers and memory addresses. The decompiled code includes variable declarations, assignments, and conditional logic.

Figure 21: First function I needed to patch: Small extract of the pseudo C code leading to the function call, and assembly code calling the said function

The screenshot shows a debugger window with two panes. The left pane displays assembly code for the function LAB\_1400c4220, starting with a JNC instruction at address 1400c4205. The right pane shows the decompiled pseudo-C code, starting with 'local\_1398 = 0;' and 'local\_1399 = lVar13;'. The assembly code includes instructions like MOVZX, SLB, MOVZX, CMP, MOV, CMOVBE, MOV, CALL, CMP, JZ, XOR, MOV, CALL, CMP, JNZ, MOV, MOV, and CALL. The decompiled code includes variable declarations, assignments, and conditional logic.

Figure 22: Second function I needed to patch: What call it, and a small extract of the beginning of the function

Figure 23: Second function I needed to patch: Small extract of the pseudo C code leading to the function call, and assembly code calling the said function

After two well-positioned jump patches, the driver stopped crashing. In the next section, we will actually patch "nvlddmkm.sys".

### 1.5.4 Patching "nvlddmkm.sys"

Below, some screenshot from 'ghidra' showing the assembly code that needs to be patched:

Figure 24: Code of the first function to patch



- Create a new root certificate
- Remove the existing signature from "nvlddmkm.sys"
- Generate a new CAT file for the driver
- Sign "nvlddmkm.sys" with the new root certificate
- Sign "nv\_disp.cat" with the new root certificate

You will need to modify the value of 'nvidiopath' and 'wdkpath'.

```
set nvidiopath="C:\NVIDIA\DisplayDriver\551.61\Win11_Win10-DCH_64\Internat
ional\Display.Driver"
set wdkpath="C:\Program Files (x86)\Windows Kits\10\bin\10.0.22621.0"

%wdkpath%\x64\MakeCert.exe -r -n "CN=QubesNvidia" -ss Root -sr
LocalMachine
%wdkpath%\x64\signtool.exe remove /s %nvidiopath%\nvlddmkm.sys
%wdkpath%\x86\Inf2Cat.exe /driver:%nvidiopath% /os:10_x64
%wdkpath%\x64\signtool.exe sign /v /sm /s Root /n "QubesNvidia" /debug /a
/fd sha1 /t http://timestamp.digicert.com %nvidiopath%\nvlddmkm.sys
%wdkpath%\x64\signtool.exe sign /v /sm /s Root /n "QubesNvidia" /debug /a
/fd sha1 /t http://timestamp.digicert.com %nvidiopath%\nv_disp.cat
```

**Step 5: Resume the installation of the nvidia driver** You can now finish the installation using the nvidia GUI that is still running from Step 2.

## 1.6 Conclusion

A QubesOS patch of xen stubdom restricted access to the PCI configuration (read and write). Following this patch, the 'nvidia' driver on Linux and the nvidia driver on Windows stopped working. However, the 'nvidia-open' driver for Linux still works well.

I believed the bug to be on the nvidia side, so I decided to try to patch both Linux and Windows nvidia drivers to make them work. Through reverse engineering, dynamic and static analysis, I was able to patch the Linux and Windows nvidia proprietary driver to make them work. And that is a nice training to ready myself to take on the AWE course :)

### 1.6.1 Extra mile

I, however, didn't track the bug down to the exact line of assembly and necessary conditions, I just ensured that the problematic case was not reached. It could be interesting to go deeper than what I did here to understand (This could be complex or not, I just didn't try to get the answer to that yet).

For the Linux 'nvidia' driver:

- For a nvidia GPU, what is the meaning the decimal value "1049603" and "1049607" for the "command" field of the pci configuration header ?
- Why exactly does it crash ? Writing to offset 4 trigger an error code not expected by the nvidia driver ? ...

- Knowing the answer to that, is it legitimate for QubesOS to block this write ? (Does the patch is actually buggy ?) ...
- If the QubesOS patch is indeed correct, is there a better way to solve this issue (Bug report on the nvidia side to handle this special case ? Tricking the system into thinking the call to `pci_write_config_dword` actually worked ?)

For the Windows nvidia driver:

- What are all those calls being do for offset 0 for `HalGetBusDataByOffset` ? retrieving what exact information ?
- For a GPU, what is offset 0 for `HalGetBusDataByOffset` ? ...
- Knowing the answer to that, is it legitimate for QubesOS to block this read ? (Does the patch is actually buggy ?) ...
- If the QubesOS patch is indeed correct, is there a better way to solve this issue (Bug report on the nvidia side to handle this special case ? Tricking the system to return artificial data for call to `HalGetBusDataByOffset` on an not authorized offset ?)

## 1.6.2 Follow up

I originally linked this article on the [github issue 9003](#).

The following discussion lead to investigate potential issue in qemu, and it lead to discovering a integer overflow issue in the patch [0005-hw-xen-xen\\_pt-Save-back-data-only-for-declared-regis.patch](#)

Test code I wrote to confirm the integer overflow:

```
#include <stdint.h>
#include <stdio.h>

// gcc XXX.c ; ./a.out

int main(int argc, char *argv[]) {

    int emul_len = 4;
    uint32_t write_val = 0x100403;

    // Integer overflow
    uint32_t mask1 = ((1 << (emul_len * 8)) - 1);
    printf("%x %x \n", mask1, write_val & mask1);

    // The value here is probably calculated at compile time using int64 so
    ↪ the overflow doesn't occur ?
    uint32_t mask2 = ((1 << (4 * 8)) - 1);
    printf("%x %x \n", mask2, write_val & mask2);

    // Fixed
    uint32_t mask3 = (((uint64_t)1 << (emul_len * 8)) - 1);
    printf("%x %x \n", mask3, write_val & mask3);
}
```

```
}
```

And the final patch that definitely fix all of this was to modify this line

```
uint32_t mask = ((1 << (emul_len * 8)) - 1);
```

to

```
uint32_t mask = ((1L << (emul_len * 8)) - 1);
```

[The pull request.](#)

Extra miles finished and problem solved !

## 1.7 References

Things I have read, that where usefull and that I didn't already directly mentionned in this post:

- [Removing signature from PE](#)
- [Installing unsigned drivers on Windows 10](#)