QubesOS articles

Nvidia driver debugging

Author:
Neowutran

Contents

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.

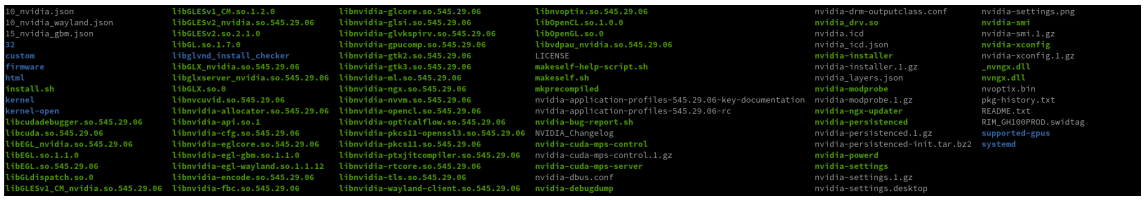
1.4.2 Structure of the driver files

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

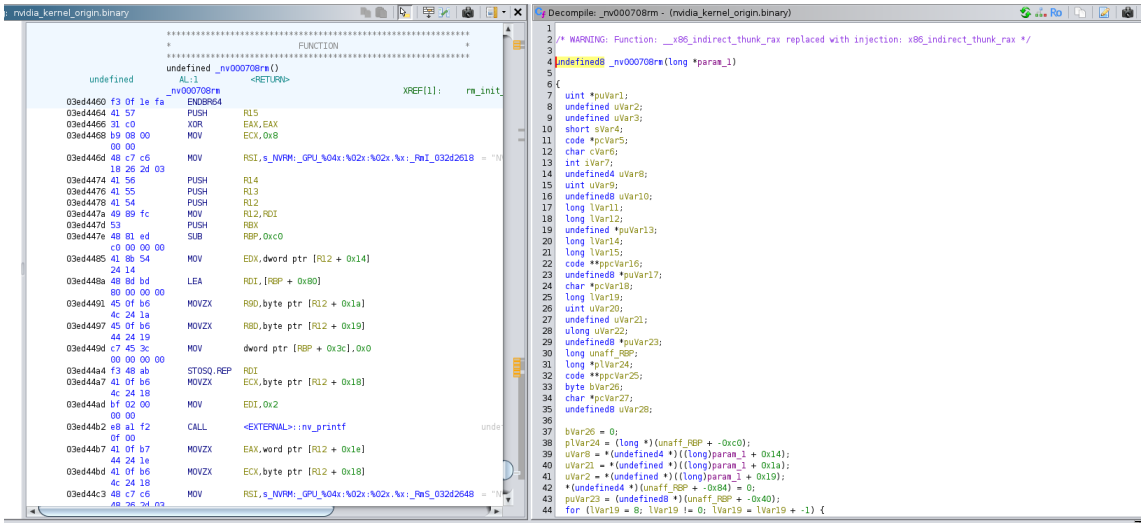


Figure 3: nvidia driver, function "_nv000708rm" - Extract 2

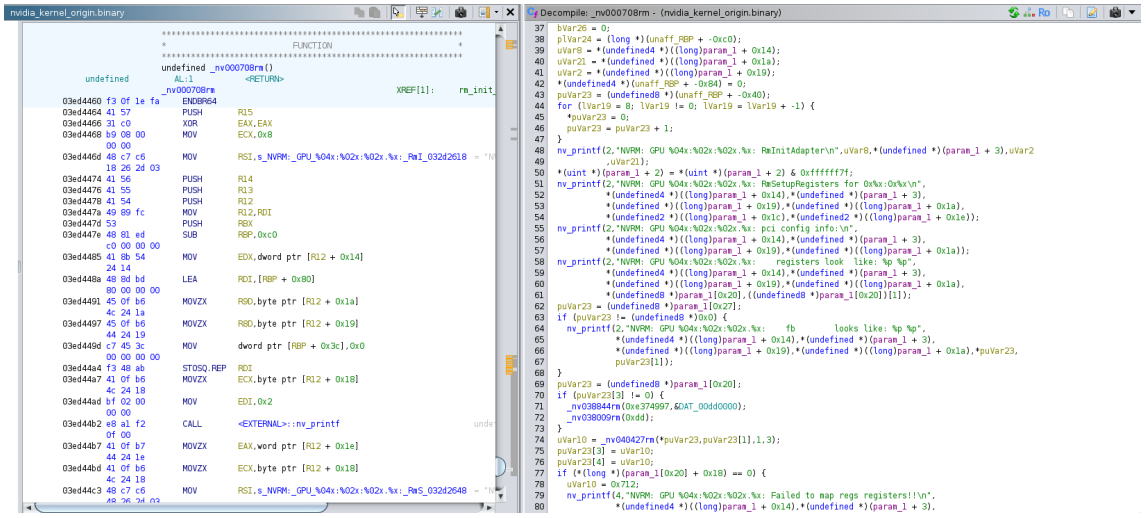


Figure 4: "nvidia-open" driver, function "RmInitAdapter" - Extract 1

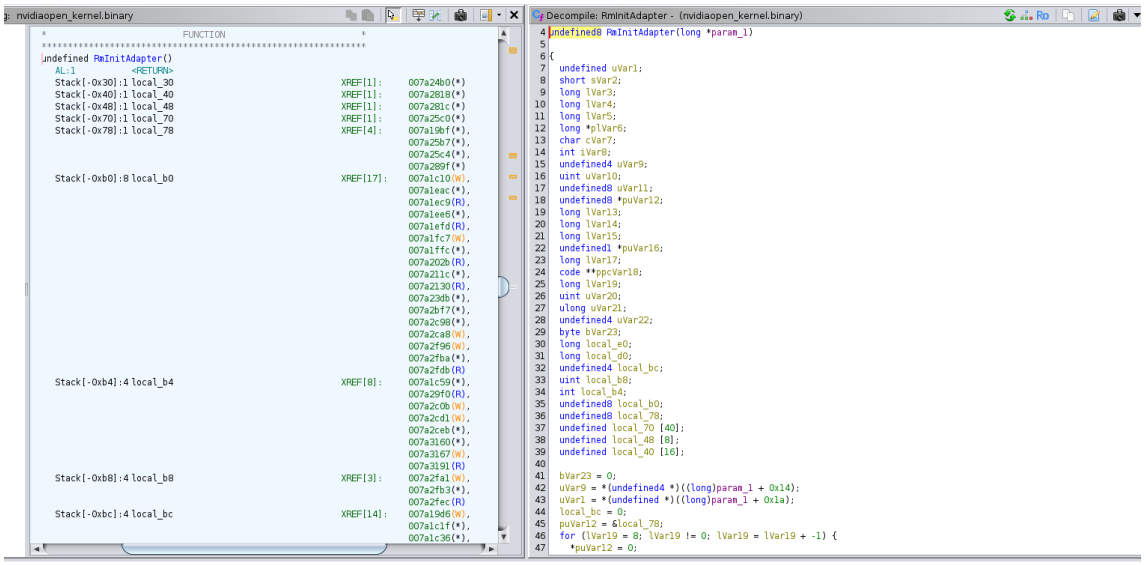
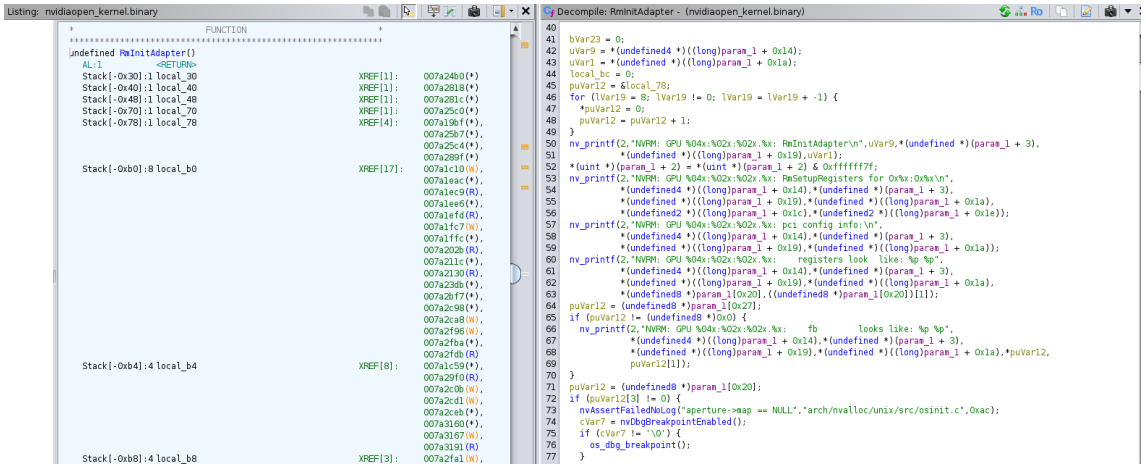


Figure 5: ”nvidia-open” driver, function ”RmInitAdapter” - Extract 2



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.

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

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

+ #ifeq ($(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.

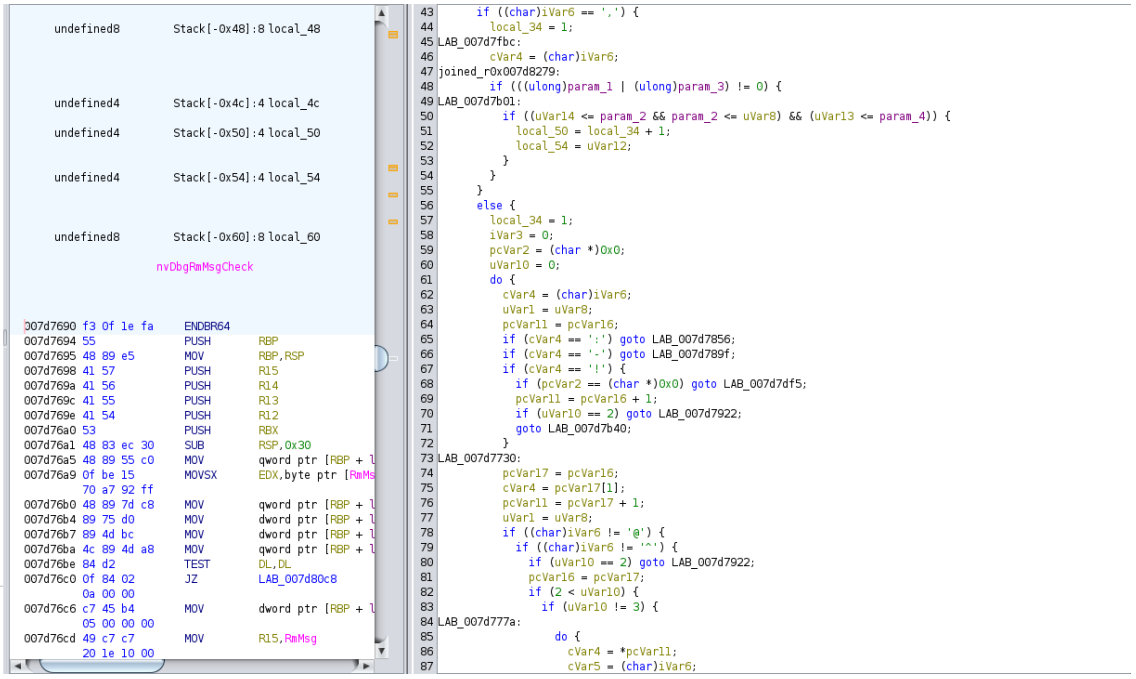
The first one can be found in the file 'os-interface.c', a short extract :

```
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_conftest 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

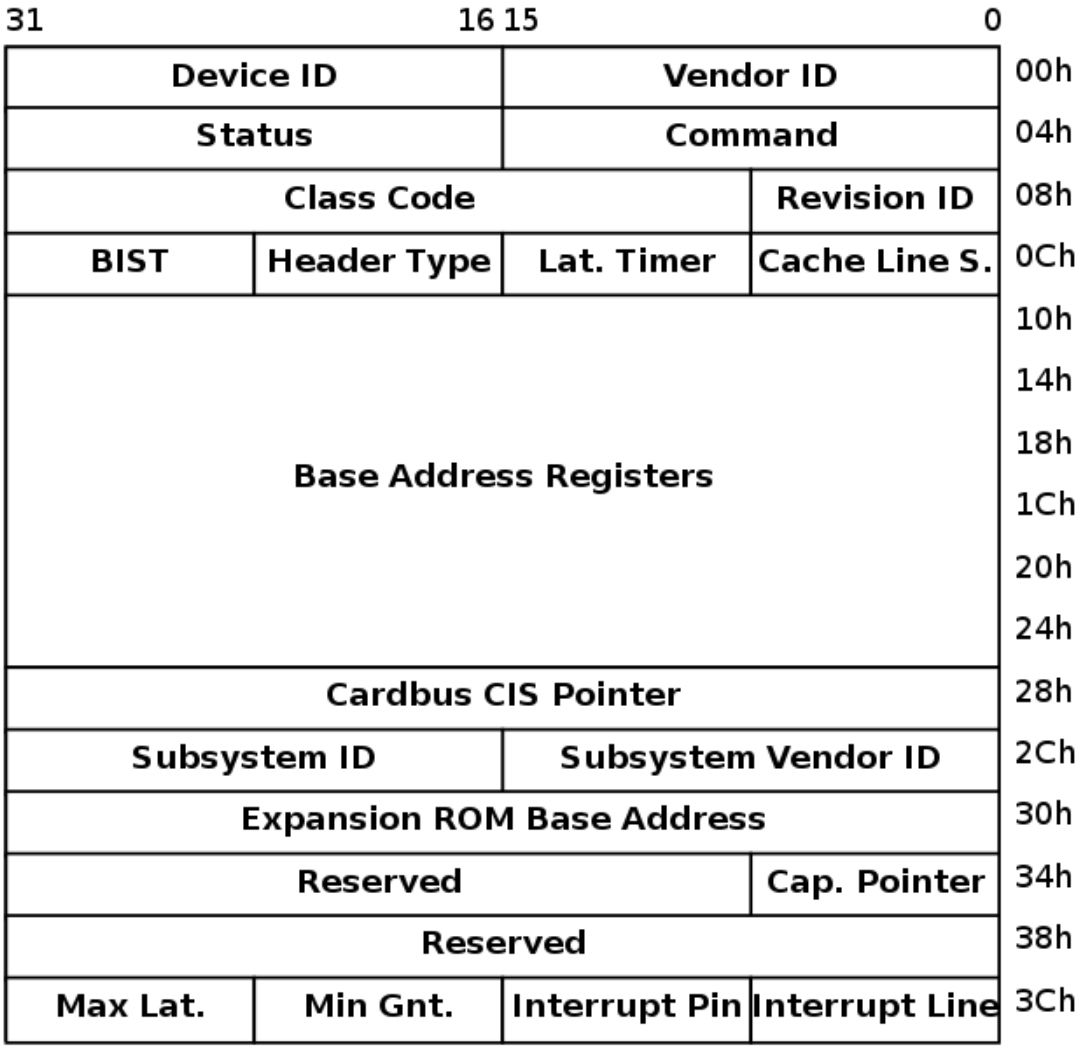
Let's forbid any write operation to offset 4:

```
 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



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
- ibm
- 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=XXXXXXXXXXXXXXXXX
```

(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.

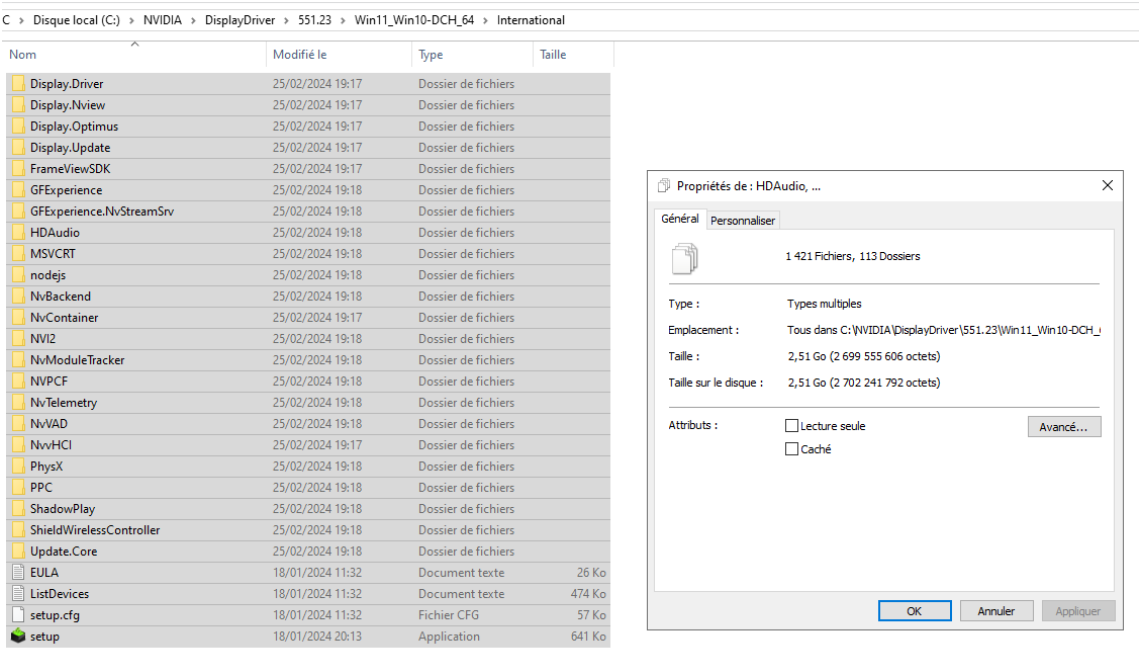
1.5.2 Creating a simplified driver to be sure the crash is not influenced by anything other than "nvlddmkm.sys"

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.

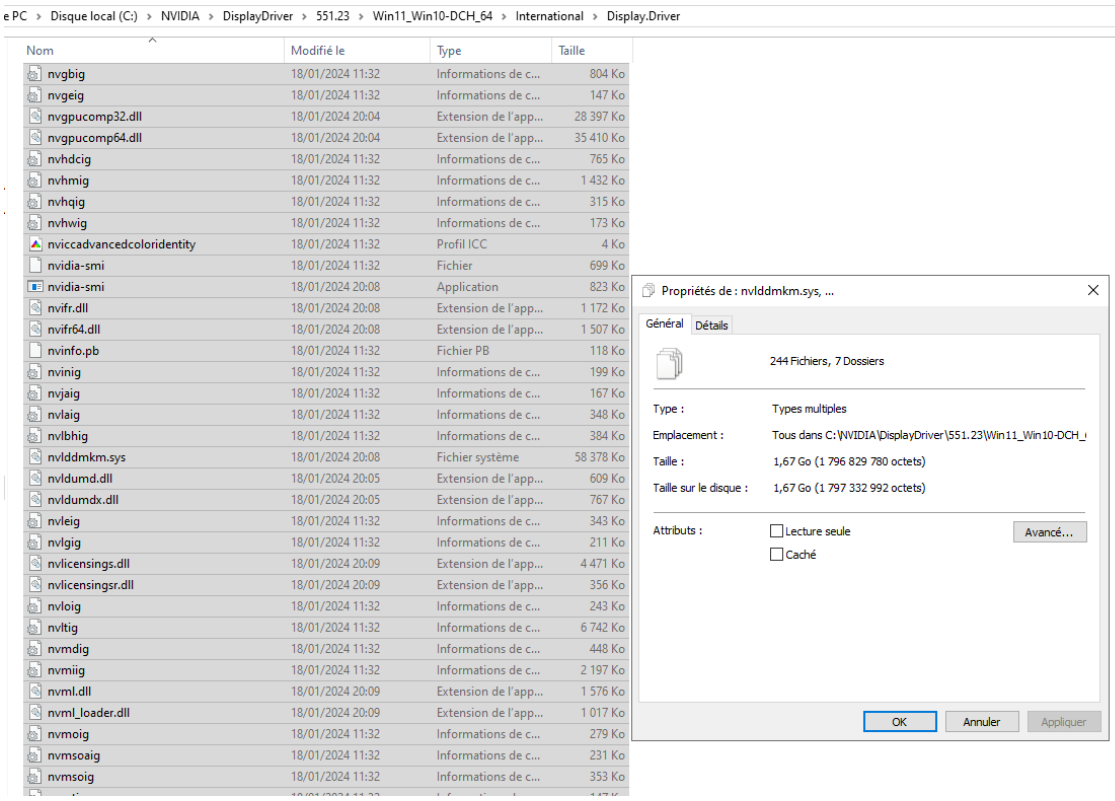
I started from that:

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



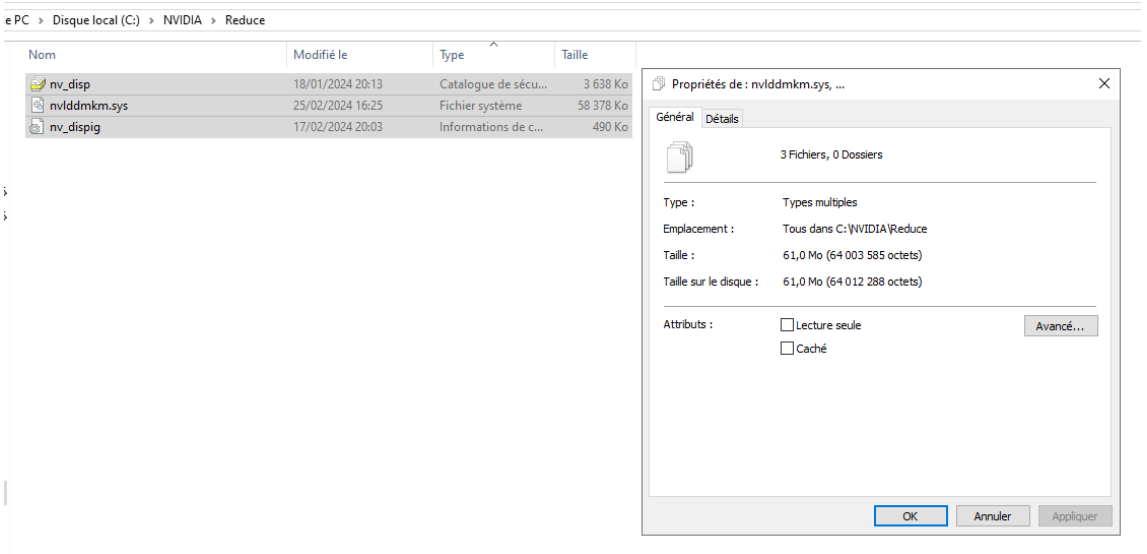
and a look inside the 'Display.Driver' folder:

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:

nvidia 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 `HalSetBusDataBy` ⌋ `Offset`

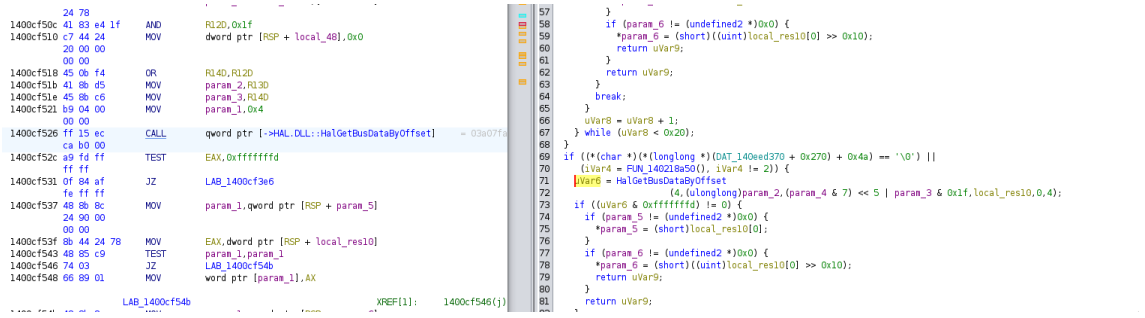
```
                        *************************************************************
                        *                    POINTER to EXTERNAL FUNCTION           *
                        *************************************************************
                        undefined HalSetBusDataByOffset()
          undefined         AL:1           <RETURN>
                        61  HalSetBusDataByOffset  <<not bound>>
                        PTR_HalSetBusDataByOffset_140bdc010             XREF[3]:     FUN_1400cfb40:1400cfd0b(R),
                                                                                     FUN_1400cfd30:1400cfefb(R),
                                                                                     FUN_1400cff20:1400d00e7(R)
     140bdc010 88 7f a0       addr        HAL.DLL::HalSetBusDataByOffset
               03 00 00
               00 00

                        *************************************************************
                        *                    POINTER to EXTERNAL FUNCTION           *
                        *************************************************************
                        undefined HalGetBusDataByOffset()
          undefined         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

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

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

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



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

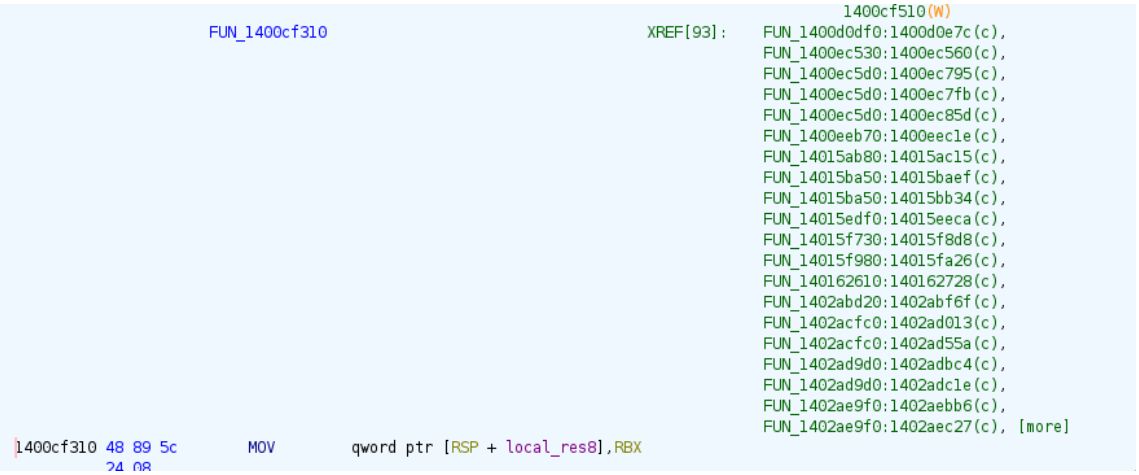


Figure 17: Extract of the list of the caller functions for the function calling the problematic `HalGetBusDataByOffset` - Part 2

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`

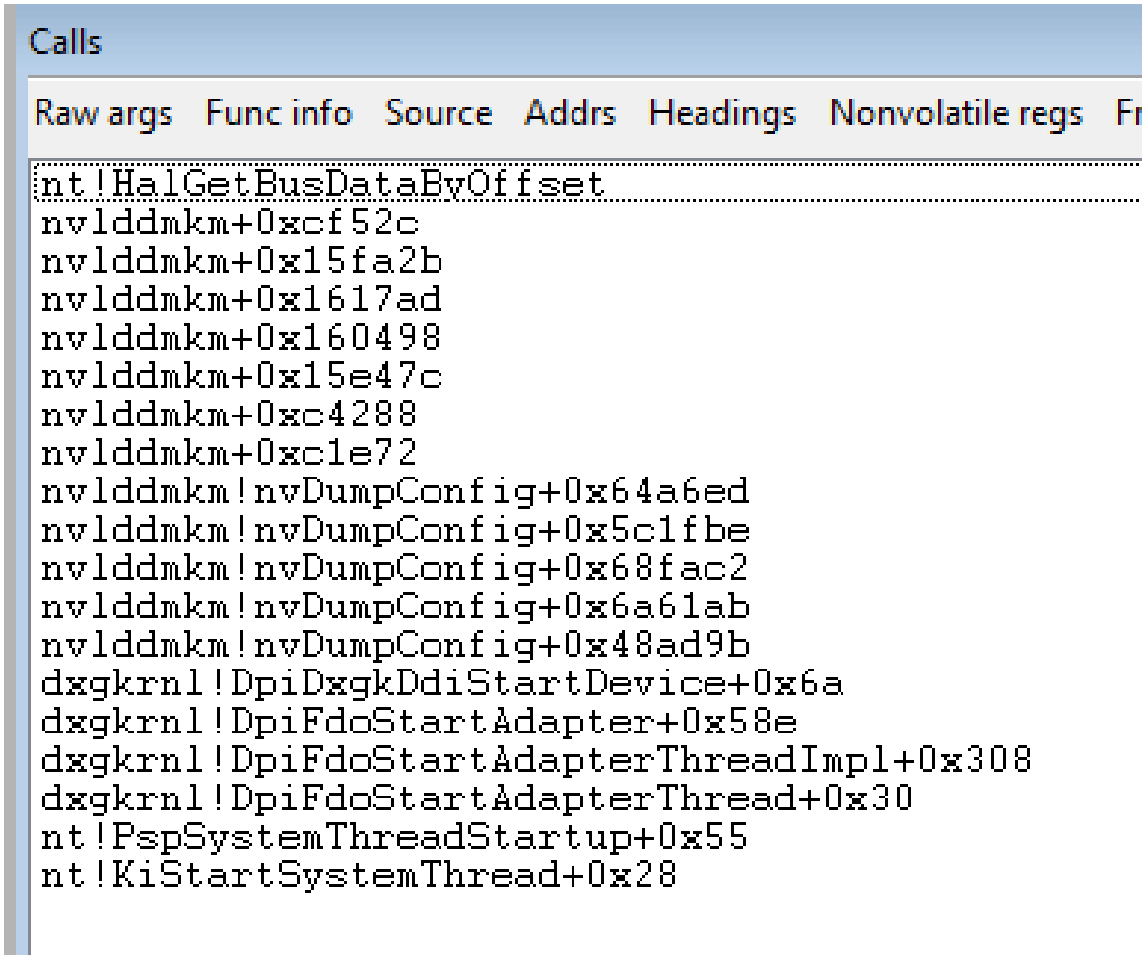
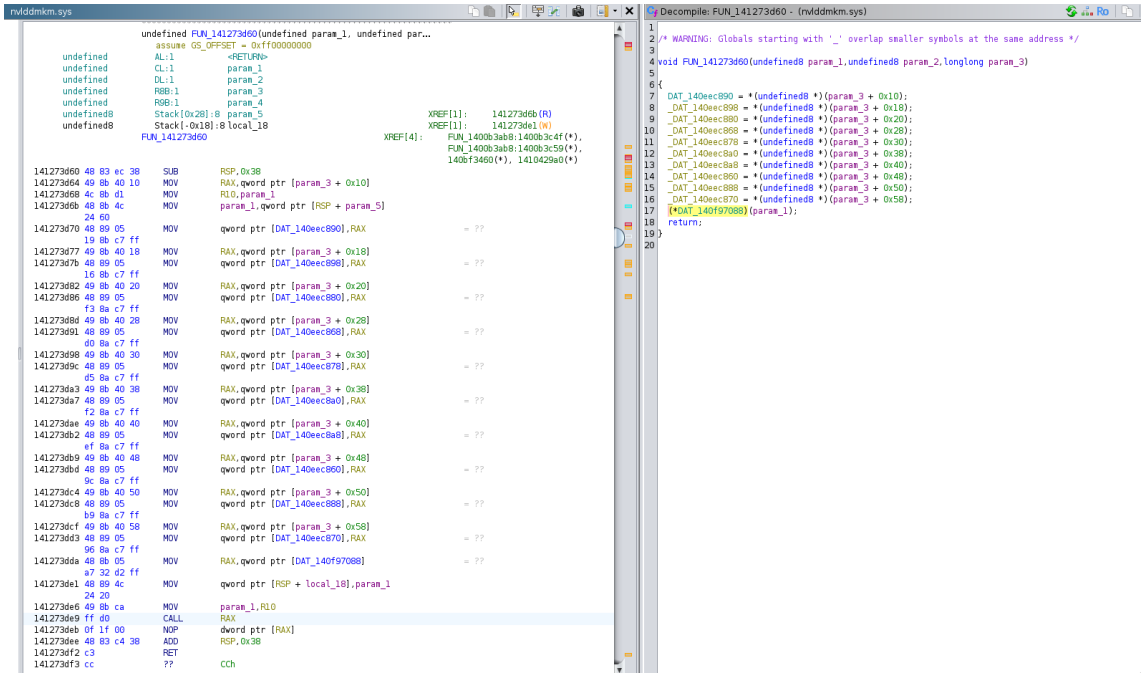


Figure 19: Ghidra: function "nvlddmkm!nvDumpConfig+0x48ad9b"



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

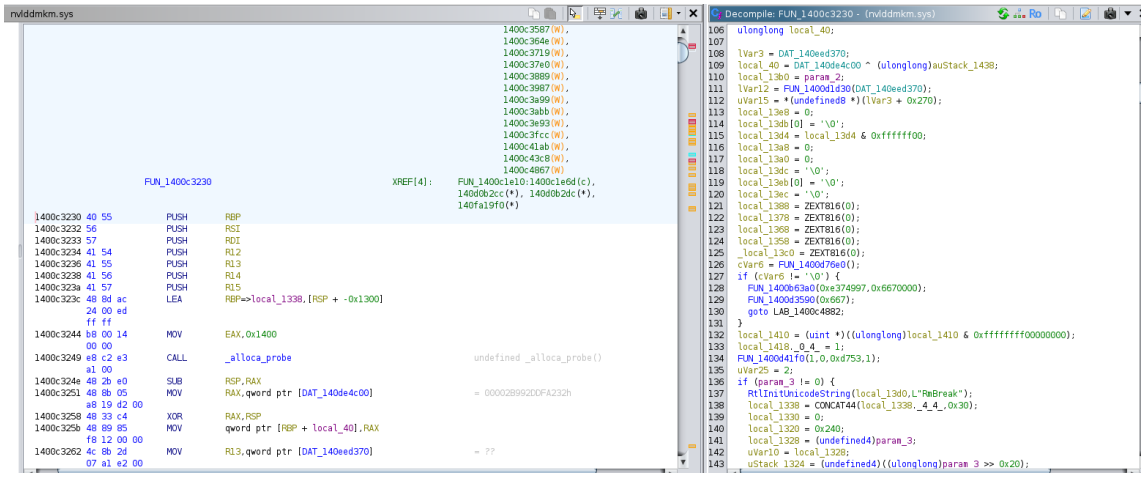


Figure 21: First function I needed to patch: Small extract of the pseudo C code leading to the function call, and assembly code calling the said function

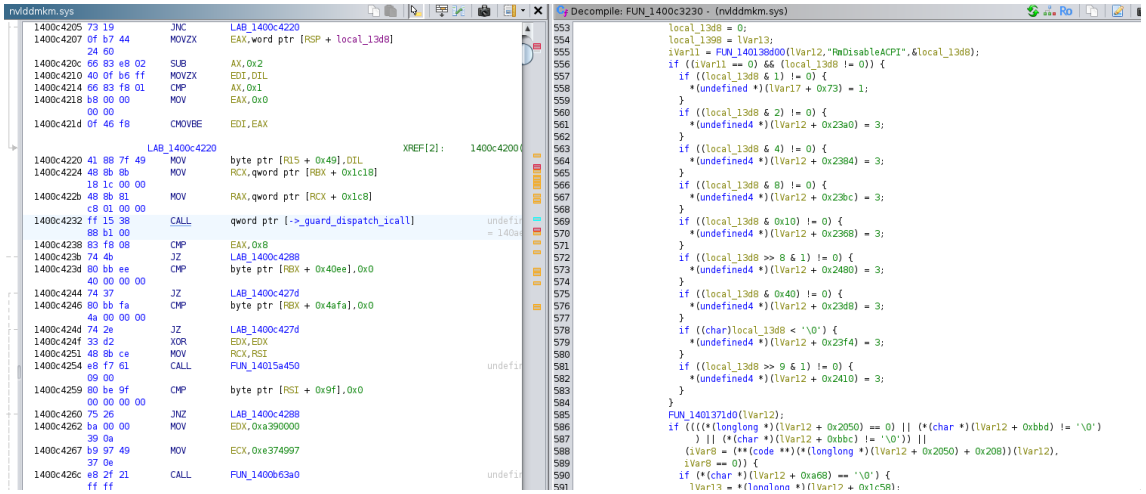


Figure 22: Second function I needed to patch: What call it, and a small extract of the beginning of the function

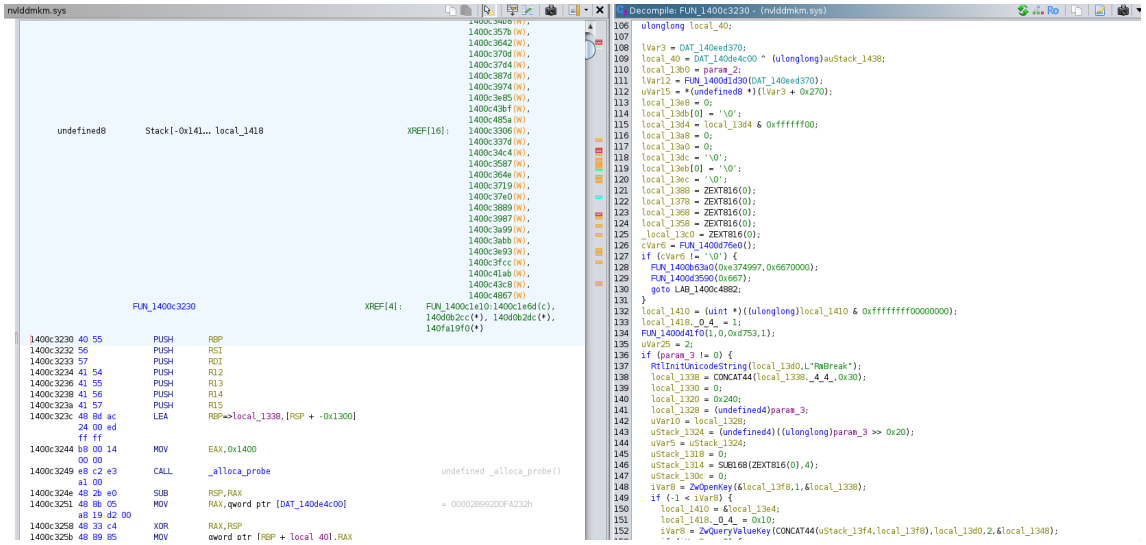
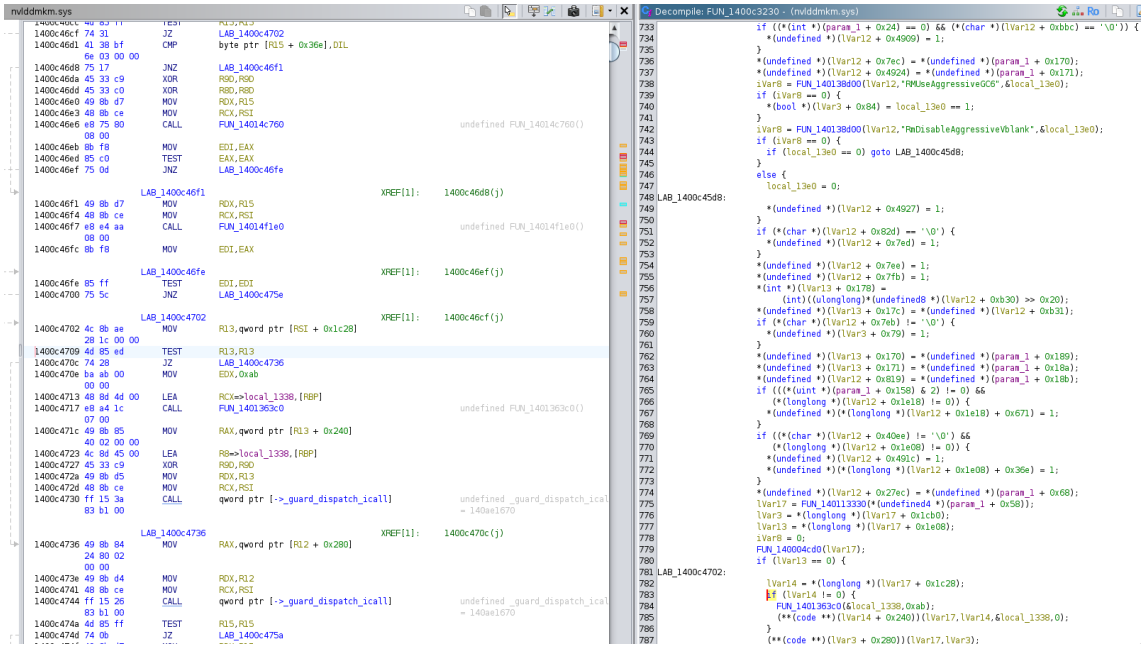


Figure 23: Second function I needed to patch: Small extract of the pseudo C code leading to the function call, and assembly code calling the said function



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

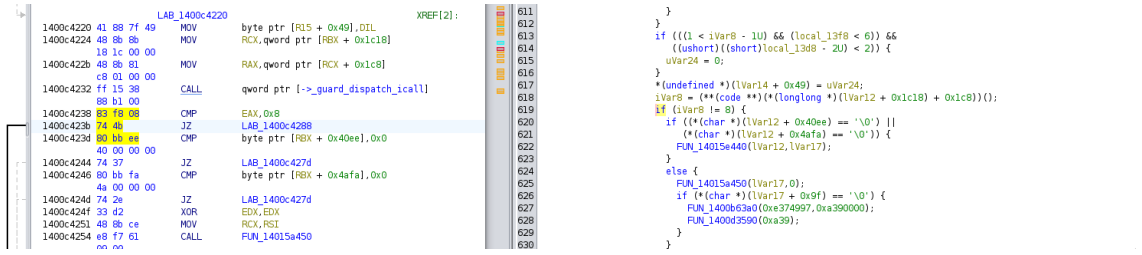
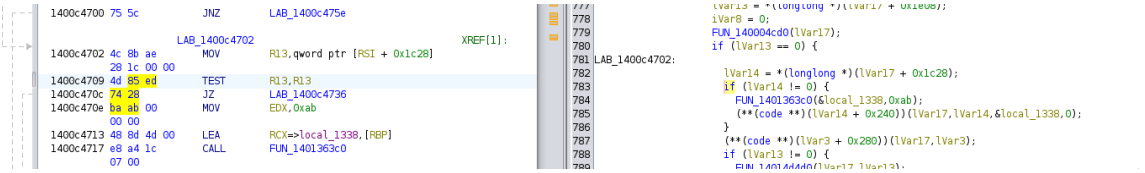


Figure 25: Code of the second function to patch



We will replace the conditional jump with a non-conditional jump, that way we never call the function that will call `HalGetBusDataByOffset` and interpret them leading to the crash. Below, two lines of bash that correctly patch the driver:

```
bbe -e 's/\x4D\x85\xED\x74\x28\xBA\xAB\x00\x00\x00/\x4D\x85\xED\xEB\x28\x
↪  BA\xAB\x00\x00\x00/' nvlddmkm.sys >
↪  nvlddmkm.sys.partialpatch
bbe -e 's/\x83\xF8\x08\x74\x4B\x80\xBB\xEE\x40\x00\x00\x00/\x83\xF8\x08\x
↪  EB\x4B\x80\xBB\xEE\x40\x00\x00\x00/' nvlddmkm.sys.partialpatch >
↪  nvlddmkm.sys.patched
```

At this point, the driver wasn't crashing anymore, but I still got no display. I guessed that I needed some of the nvidia files I deleted earlier for my test driver. So I reinstalled the official nvidia driver and replaced nvlddmkm.sys with my own version.

Et tada ! The driver now works !

However note that the driver now has an incorrect signature and by default Windows will refuse to load it. Using the Windows 'Recovery options' you can force Windows to load drivers with an invalid signature.

But let's improve that. We will modify the signature of the nvidia driver.

1.5.5 Signing the nvidia driver

Step 1: Install the WDK Follow the official microsoft documentation for that step

Step 2: Extract the nvidia driver Download the nvidia driver from the official website and launch it. Pause the installation just after the extraction of the files (when the UI popup asking you if you want to just install the driver or the driver + GeForce)

Step 3: Patch nvlddmkm.sys The extracted nvidia file will be by default somewhere like `C:\NVIDIA\DisplayDriver\551.61\Win11_Win10-DCH_64\International\Display.Driver`. Copy nvlddmkm.sys to a Linux qube to apply the patch mentioned earlier. Replace the 'nvlddmkm.sys' file on windows with the patched 'nvlddmkm.sys'.

Step 4: Sign the driver You will find below a BAT script to modify and run as administrator. It will do the following things:

- 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 'nvidiapath' and 'wdkpath'.

```
set nvidiapath="C:\NVIDIA\DisplayDriver\551.61\Win11_Win10-DCH_64\Interna
↪  tional\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 %nvidiapath%\nvlddmkm.sys
%wdkpath%\x86\Inf2Cat.exe /driver:%nvidiapath% /os:10_x64
%wdkpath%\x64\signtool.exe sign /v /sm /s Root /n "QubesNvidia" /debug /a
↪  /fd sha1 /t http://timestamp.digicert.com %nvidiapath%\nvlddmkm.sys
%wdkpath%\x64\signtool.exe sign /v /sm /s Root /n "QubesNvidia" /debug /a
↪  /fd sha1 /t http://timestamp.digicert.com %nvidiapath%\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 ? Writting 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 thoses 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:

```c
#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 definitly 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