Vulnerable Driver Manipulation

***Abstract*—This research paper documents the process of using a vulnerable Windows kernel driver exposing a physical memory read and write[1] primitive to call any function inside of the Windows kernel; while also teaching you the basics of paging and physical memory.**

***Keywords*—Vulnerable Driver, Physical Memory, Virtual Address, Relative Virtual Address (RVA), Inline Hooking, Arbitrary Physical Memory Access**

I. Introduction

Exploiting vulnerable Windows drivers to leverage kernel execution is not a new concept. Although software that exploits vulnerable drivers has been around for a long time, there has yet to be a highly modular library of code that can be used to exploit multiple drivers exposing the same vulnerability. Windows drivers exposing arbitrary physical memory read and write primitives are the most abundant form of vulnerable drivers. These drivers are used for many things ranging from reading CPU fan speeds to flashing BIOS. Although there are thousands of drivers that expose this primitive; doing anything useful with these drivers is not necessarily a straightforward task. In this research paper, I will be describing the steps on how to obtain kernel execution with an arbitrary physical memory read and write primitive. Furthermore, I will be demonstrating how simple it is to find and exploit such drivers whilst providing example code along the way.

II. Locating a Vulnerable Driver

Finding a driver that exposes arbitrary physical memory read and write is as easy as googling the phrases: BIOS flashing utility for Windows, CPU fan speed utility for Windows, or ASUS overclocking utility for Windows. There are hundreds if not thousands of these drivers which allow for arbitrary physical memory read and write. In this research paper I'm going to specifically speak about phymem.sys; a Supermicro BIOS flashing Windows utility which I discovered during the process of making the introduction to this paper. Although there is an abundance of vulnerable drivers that expose physical memory read and write; typically the ranges of physical memory that can be manipulated are restricted. In the case of phymem.sys, only the first 4GB of physical memory can be arbitrarily read and written to. Be aware of these potential memory range restrictions when hunting for a vulnerable driver yourself.

Once you think you have found a vulnerable driver determining that it is in fact vulnerable can be done by concluding that user controlled data is passed to either: MmMapIoSpace[2], ZwMapViewOfSection, or MmCopyMemory[3]. This user controlled data is delivered to the driver's device control major function by calling DeviceIoControl. In the case of phymem.sys, user controlled data is passed to MmMapIoSpace.

III. Interfacing With a Vulnerable Driver

After determining that a driver is vulnerable the next step is to figure out how to interface with said vulnerable driver. The three most important values one should look for when reverse engineering the IRP_MJ_DEVICE_CONTROL function is: I/O control codes, IOCTL input and output buffer lengths, and finally input and output buffer(s)[4]. By observing how the user controlled data is used; a structure can be constructed.

```
InputBufferLength = StackLocation->Parameters
    .DeviceIoControl
    .InputBufferLength;

OutPutBufferLength = StackLocation->Parameters
    .DeviceIoControl
    .OutputBufferLength;

// IRP_MJ_DEVICE_CONTROL...
if (StackLocation->MajorFunction == 0xE)
{
  v22 = StackLocation->Parameters
      .Read
      .ByteOffset
      .LowPart;

  // 0x80002000 (MAP_PHYSICAL_MEMORY)
  switch (v22 + 0x7FFFE000)
  {
  case 0u:
      // mind lengths for DeviceIoControl...
    if (InputBufferLength != 16i64 ||
        OutPutBufferLength != 8i64)
        return {}; // invalid lengths...
    else
    {
      // 4gb of physical memory limit
      UserControlledPhysicalAddress.QuadPart =
          *((_QWORD*)SystemBuffer + 1)
              & 0xFFFFFFFFi64;

      VirtualAddress = MmMapIoSpace
          (UserControlledPhysicalAddress,
              *(_QWORD*)SystemBuffer, NULL);
      // IoAllocateMdl
      // MmBuildMdlForNonPagedPool...
      // MmMapLockedPagesSpecifyCache...
```

Listing 1. IRP_MJ_DEVICE_CONTROL Function of Phymem.sys

In the case of phymem.sys, the input buffer length is 16 bytes, and the output buffer length is 8 bytes. Looking at how SystemBuffer is used in Listing 1 you can see that it is a structure containing two QWORD sized fields. Further inspection concludes that the first QWORD field contains the size value in bytes of how much physical memory to map, and the second QWORD field is the physical address of memory to be mapped. As you can see in Listing 1, line 33, the top 32bits of the physical address is ignored. This limits the physical address to 32bits in size and thus the driver only allows us to map physical memory which is located in the first 4GB of physical memory.

```
#define MAP_PHYSICAL_MEMORY 0x80002000
#define UNMAP_PHYSICAL_MEMORY 0x80002004

// 16 bytes
typedef struct _map_phys_t
{
  union
  {
    std::uintptr_t map_size;  // + 0x0
    std::uintptr_t virt_addr; // + 0x0
  }
  std::uintptr_t phys_addr; // + 0x8
} map_phys_t, *pmap_phys_t;
```

Listing 2. Structure Passed via DeviceIoControl

Once a structure has been defined; interfacing with the vulnerable driver is just a matter of loading the driver into the kernel using NtLoadDriver[5], and then controlling the driver with DeviceIoControl.

IV. Scanning Physical Memory

Although physical memory may seem ambiguous, it is organized into fixed sized chunks called pages. Each page on a 64-bit system using a four layer paging table configuration is 4kB[6]. Within this chunk size memory is contiguous. The last 12bits of every 64-bit virtual address is called the page offset. Knowing this one can scan every single page at a specific offset for specific bytes.

```
PAGE:00000001C01265F0 ; ======= S U B R O U T I N E =======
PAGE:00000001C01265F0 public NtGdiDdDDICreateContext
PAGE:00000001C01265F0 NtGdiDdDDICreateContext proc near
PAGE:00000001C01265F0    mov     [rsp+arg_8], rbx
PAGE:00000001C01265F5    mov     [rsp+arg_10], rdi
PAGE:00000001C01265FA    mov     [rsp+arg_18], r12
PAGE:00000001C01265FF    push    r13
PAGE:00000001C0126601    push    r14
PAGE:00000001C0126603    push    r15
PAGE:00000001C0126605    sub     rsp, 200h
PAGE:00000001C012660C    mov     rax, cs:__security_cookie
PAGE:00000001C0126613    xor     rax, rsp
PAGE:00000001C0126616    mov     [rsp+218h+var_28], rax
PAGE:00000001C012661E    mov     r14, rcx
PAGE:00000001C0126621    mov     [rsp+218h+var_170], rcx
PAGE:00000001C0126629    mov     [rsp+218h+var_1A0], rcx
PAGE:00000001C012662E    or      rdi, 0FFFFFFFFFFFFFFFFh
PAGE:00000001C0126632    mov     [rsp+218h+var_1C0], edi
```

Listing 3. NtGdiDdDDICreateContext First Few Instructions

In Listing 3, the page offset for the system routine NtGdiDdDDICreateContext is 0x5F0. Simply scanning every single physical page at offset 0x5F0 for opcodes in NtGdiDdDDICreateContext is enough to get a handful of results. Testing each occurrence is required in order to determine that we have found the real[7] NtGdiDdDDICreateContext in physical memory. Scanning each page one at a time is quite slow so to expedite the process VDM creates a new thread for each physical memory range[8].

V. Elevating to Kernel Execution

Everytime an occurrence of NtGdiDdDDICreateContext's bytes is found in physical memory, a test is conducted to determine if the correct memory has been located. This test places some assembly code over the first few instructions of NtGdiDdDDICreateContext. NtGdiDdDDICreateContext is then called to see if the desired instructions were executed. Finally regardless of the situation the original bytes are restored.

```
bool vdm_ctx::valid_syscall(void* syscall_addr) const
{
  static std::mutex syscall_mutex;
  syscall_mutex.lock();

  static const auto proc =
    GetProcAddress(
      LoadLibraryA(syscall_hook.second),
      syscall_hook.first
    );

  // 0:  48 31 c0    xor rax, rax
  // 3 : c3          ret
  std::uint8_t shellcode[] = { 0x48, 0x31, 0xC0, 0xC3 };
  std::uint8_t orig_bytes[sizeof shellcode];

  // save original bytes and install shellcode...
  vdm::read_phys(
      syscall_addr,
      orig_bytes,
      sizeof orig_bytes);

  vdm::write_phys(
      syscall_addr,
      shellcode,
      sizeof shellcode);

  auto result = reinterpret_cast<
      NTSTATUS(__fastcall*)(void)>(proc)();

  vdm::write_phys(
      syscall_addr,
      orig_bytes,
      sizeof orig_bytes);

  syscall_mutex.unlock();
  return result == STATUS_SUCCESS;
}
```

Listing 4. Checking Physical Page to Ensure its the Right One

Now that we know the correct location of NtGdiDdDDICreateContext's routine in physical memory; we can install an inline hook at the beginning of the function everytime we want to call a specific function in the kernel, and then restore the original bytes once the syscall has finished. Locating specific routines in the kernel can be done with simple arithmetic. The location of kernel module base addresses can be obtained simply with NtQuerySystemInformation using SystemModuleInformation. This allows us to calculate the absolute virtual address of any kernel function we want. Simply by loading the driver[9] which contains the function desired and subtracting the address of it from the base address of the loaded driver a relative virtual address is produced. Subsequently the inverse operation (addition) can be applied to the kernel modules base address to produce the absolute kernel virtual address of the desired function. In conjunction with the ability to inline hook

NtGdiDdDDICreateContext this allows a VDM user to call any kernel function they desire.

VI. Using A Vulnerable Driver With VDM

VDM allows a programmer to easily integrate a vulnerable driver into the project simply by coding four functions used by the rest of the project. The four functions that are required for VDM to work are: vdm::load_drv, vdm::unload_drv, vdm::read_phys, and vdm::write_phys. Once these functions have been programmed appropriately the library will take care of the rest. Most drivers map and unmap physical memory, so when programming vdm::read_phys and vdm::write_phys map the physical memory, use memcpy, then unmap the physical memory.

VII. Limitations

- VDM will not work on HVCI systems.
- Inline hook on syscall is not thread safe and can cause system instability.

VIII. Conclusion

VDM abstracts the concept of a vulnerable driver that exposes physical memory read and write to a method in which you can call into any kernel function you desire. The overabundance of vulnerable drivers exposing this primitive allows VDM to be much more modular and thus much more attractive than other public options.

IX. Examples

- VDM: https://githacks.org/xerox/vdm
- TDL: https://github.com/hfiref0x/TDL
- KDU: https://github.com/hfiref0x/KDU
- nasa-mapper: https://githacks.org/xerox/nasa-mapper
- kdmapper: https://github.com/z175/kdmapper
- gdrv-loader: https://github.com/alxbrn/gdrv-loader
- drvmapper: https://github.com/not-wlan/drvmap

Notes

1. Although I say read and write physical memory, this typically is manifested by mapping and unmapping physical memory via MmMapIoSpace and ZwMapViewOfSection. There are some drivers that directly expose reading and writing to physical memory via MmCopyMemory.

2. MmMapIoSpace by itself is not enough to map physical memory into a user-mode process. IoAllocateMdl, MmMapLockedPages, and a few other functions are required.

3. MmCopyMemory can be used to read and write physical memory when Flags parameter is MM_COPY_MEMORY_PHYSICAL.

4. In the case of phymem.sys "SystemBuffer" is used for both the input and output buffer.

5. I have made a C++17 header-only library for this by the name of loadup. It can be found here: https://githacks.org/xerox/loadup.

6. Although pages are 4kB by default, a four layer paging table system supports 2MB pages and even 1GB pages, when a page is large, the part of the address used to describe the index into a paging table is instead used to describe an offset into the physical page.

7. These bytes are the ones executed when NtGdiDdDDICreateContext is invoked.

8. https://www.remkoweijnen.nl/blog/2009/03/20/reading-physical-memory-size-from-the-registry/

9. LoadLibraryEx using DONT_RESOLVE_DLL_REFERENCES for its dwFlag parameter.