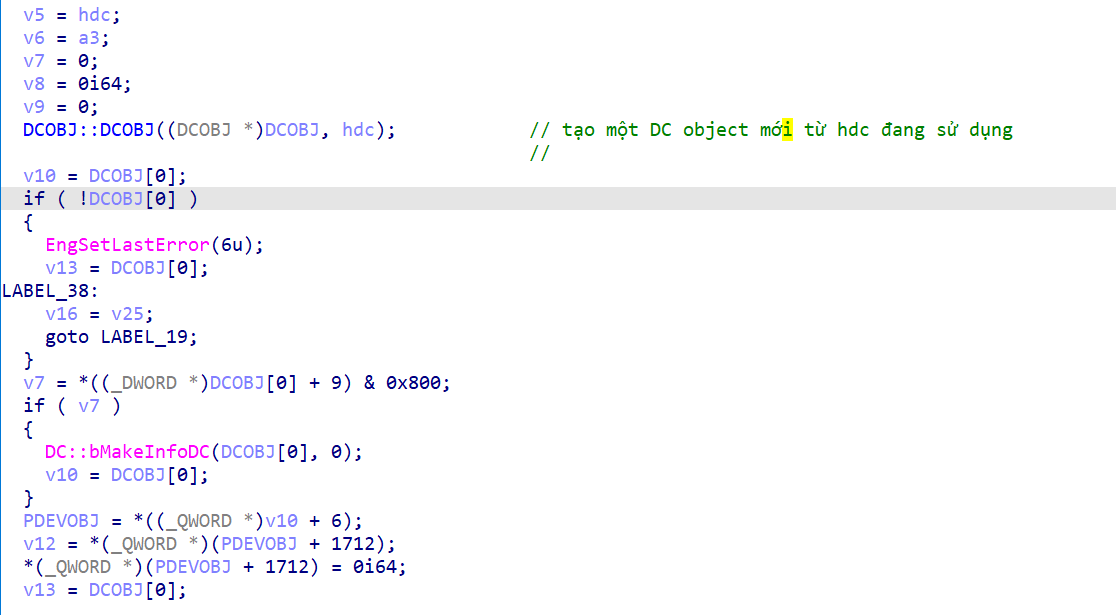
CVE-2021-40449

# I) Bug overview

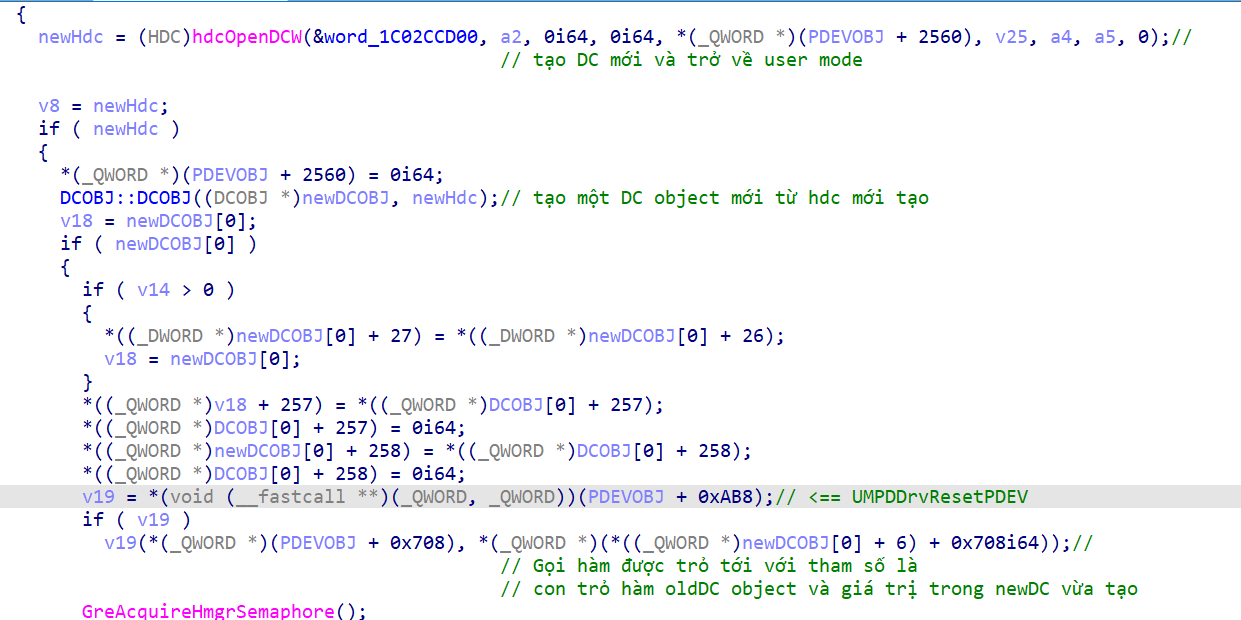
Lỗ hổng xảy ra tại hàm GreResetDCInternal trong win32kfull.sys, hàm này sẽ lấy con trỏ hàm trong object DC và thực thi hàm do con trỏ hàm trỏ tới ở user mode, nhưng không kiểm tra object DC. Khi giải phóng đối tượng DC trước lúc gọi con trỏ hàm và sử dụng lại bộ nhớ được cấp phát, ta có thể sửa đổi con trỏ hàm của đối tượng DC ban đầu để trỏ đến bất kỳ hàm kernel nào trong GreResetDCInternal bằng cách xây dựng một forge memory layout.

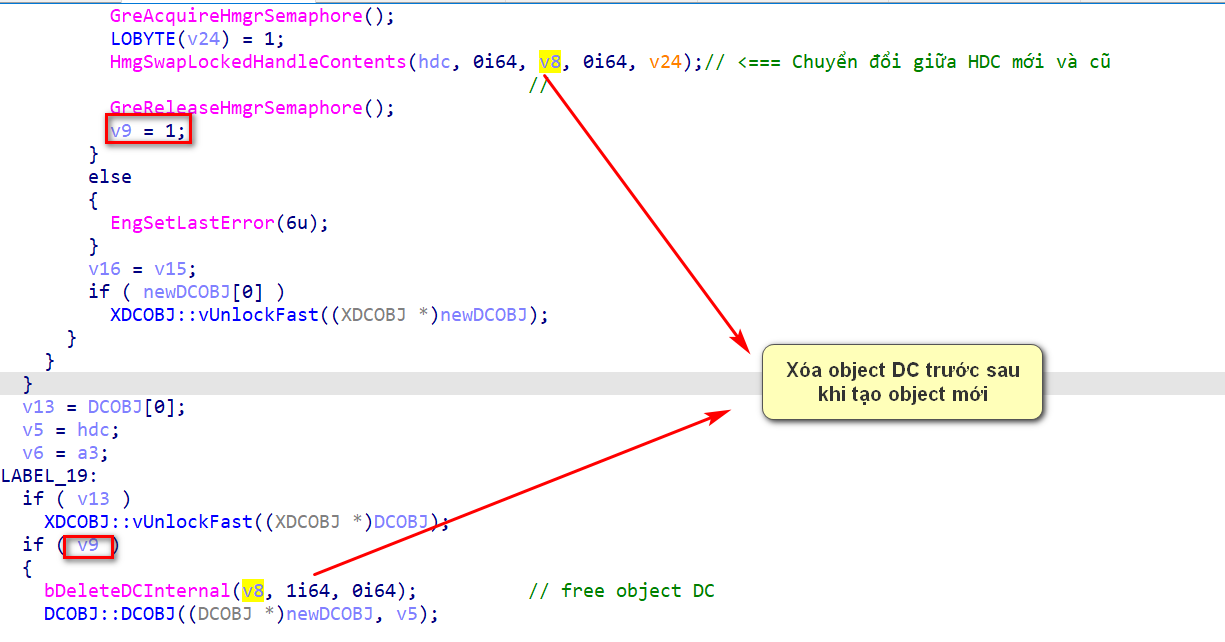
# II) Vulnerability analysis

* Đầu tiên hàm tạo một DCO object mới từ HDC truyền vào sau đó lấy PDEVOBJ ở offset 30h



* Gọi hàm ***hdcOpenDCW()*** để tạo DC object mới. Lỗ hổng được tạo ra do hàm này vì nó có chức năng gọi lại chế độ user mode
* Khi tạo một handle mới tạo bằng hàm ***hdcOpenDCW()*** thì không hề có sự kiểm tra về oldDC object đang được sử dụng, và có khả năng hdc đó đã được free => Lỗi use after free





* Nếu sau khi free old\_DC object và sử dụng lại bộ nhớ đã cấp phát, sửa đổi nội dung của con trỏ hàm bên trong và để kernel thực thi con trỏ hàm trong DC object, thì có thể đạt được mục đích cho phép kernel thực hiện leo thang đặc quyền.

# III) Vulnerability Exploitation Analysis

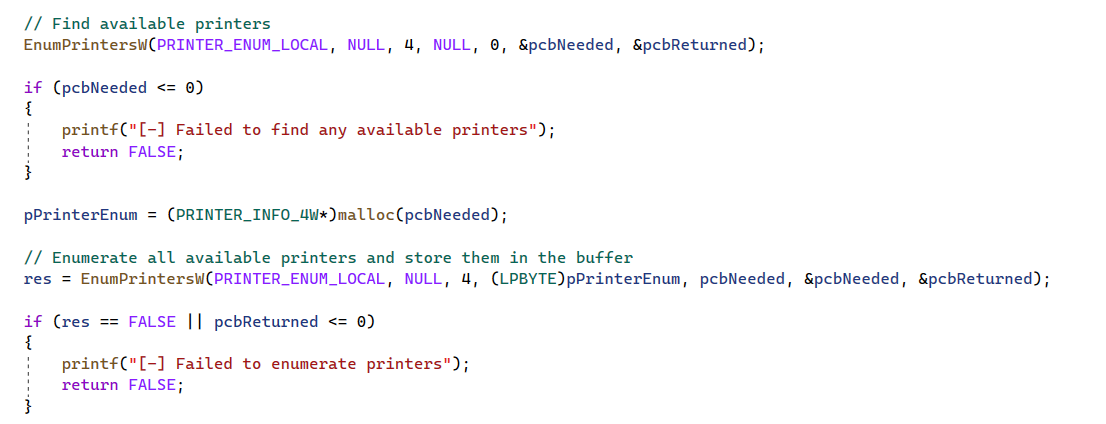
Để khai thác được lỗ hổng này, mục tiêu là để kernel callback con trỏ hàm của DC object sau khi giải phóng DC. Tóm gọn lại ta cần có các bước:

* + Hooking tới một usermode-callback và trigger UAF
  + Sử dụng lại không gian bộ nhớ đã cấp phát cho DC object đã được free
  + Hoàn thành forge memory layout để thực thi hàm gọi

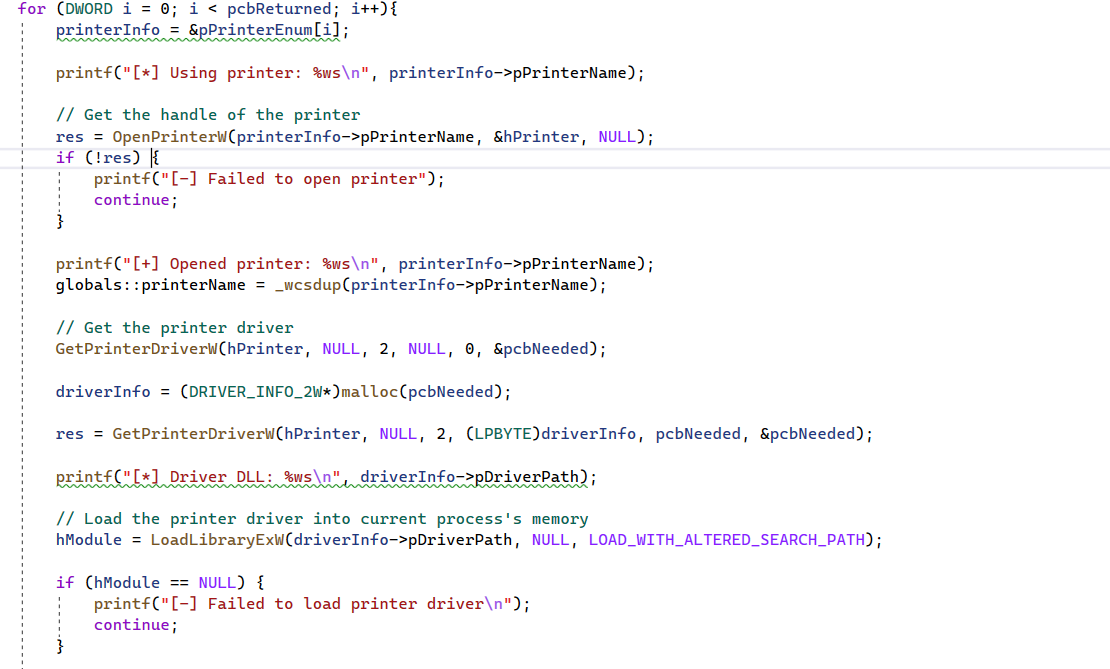
## 1) Hooking tới một usermode call back

Để tạo và hook tới một Device Context, ta có theo các bước sau:

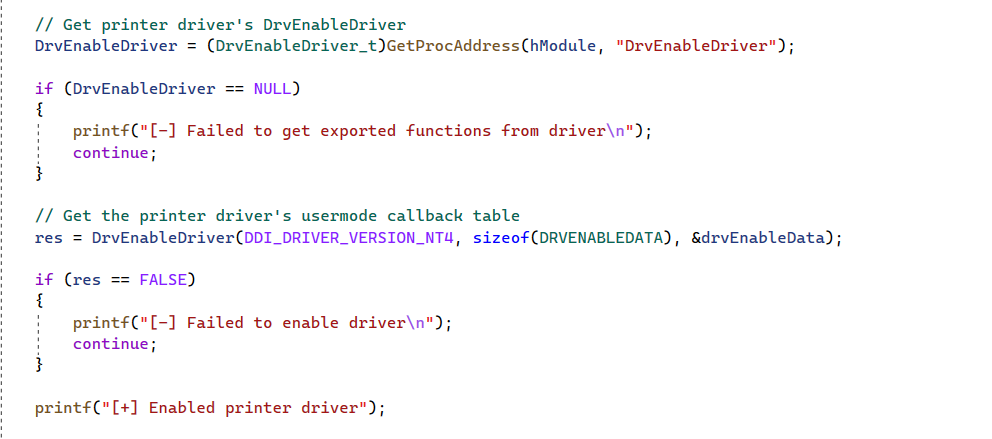
* Sử dụng ***EnumPrinters*** để tìm printer có sẵn.



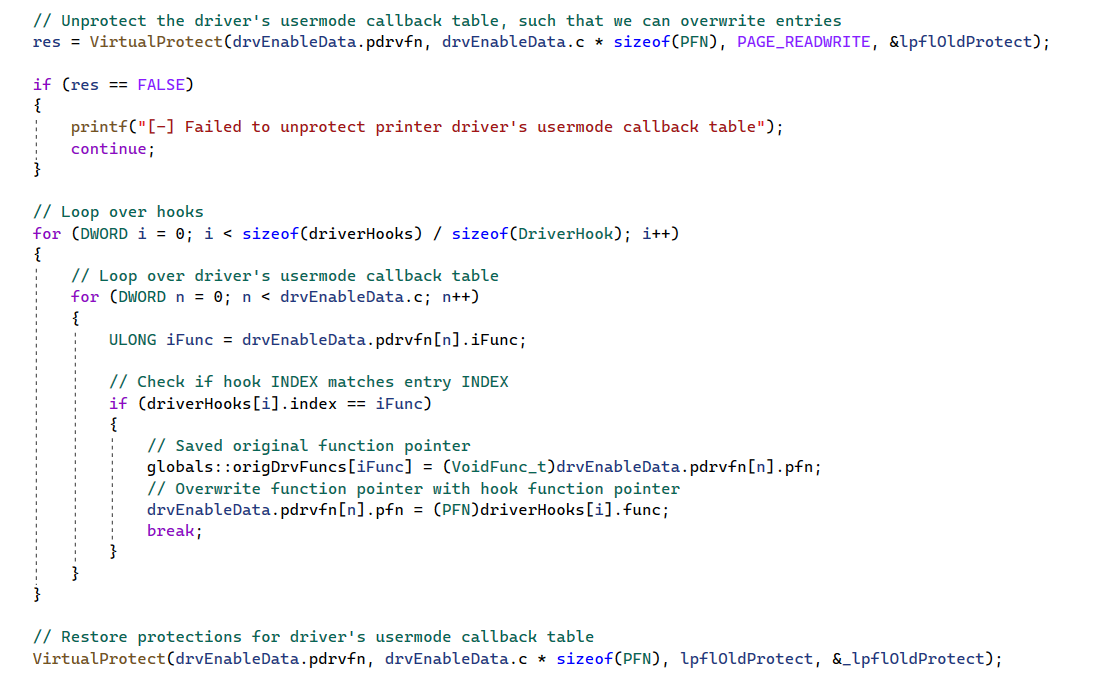
* Sử dụng ***OpenPrinter*** để lấy handle của printer, sử dụng ***GetPrinterDriver*** để lấy thông tin cần thiết, ***LoadLibraryExA*** tải printer driver vào không gian địa chỉ của tiến trình hiện tại.



* Sử dụng ***GetProcAddress*** để lấy địa chỉ chức năng của ***DrvEnableDriver*** và sử dụng ***DrvEnableDriver*** có được bảng usermode-callback.



* Sử dụng ***VirtualProtect*** để gỡ bỏ bảo vệ bảng usermode-callback của trình printer driver này, từ đó có thể thay thế nó bằng một con trỏ chức năng gọi lại tùy ý.
* Ghi đè các entries được chỉ định trong bảng usermode-callback và khôi phục chức năng bảo vệ callback table

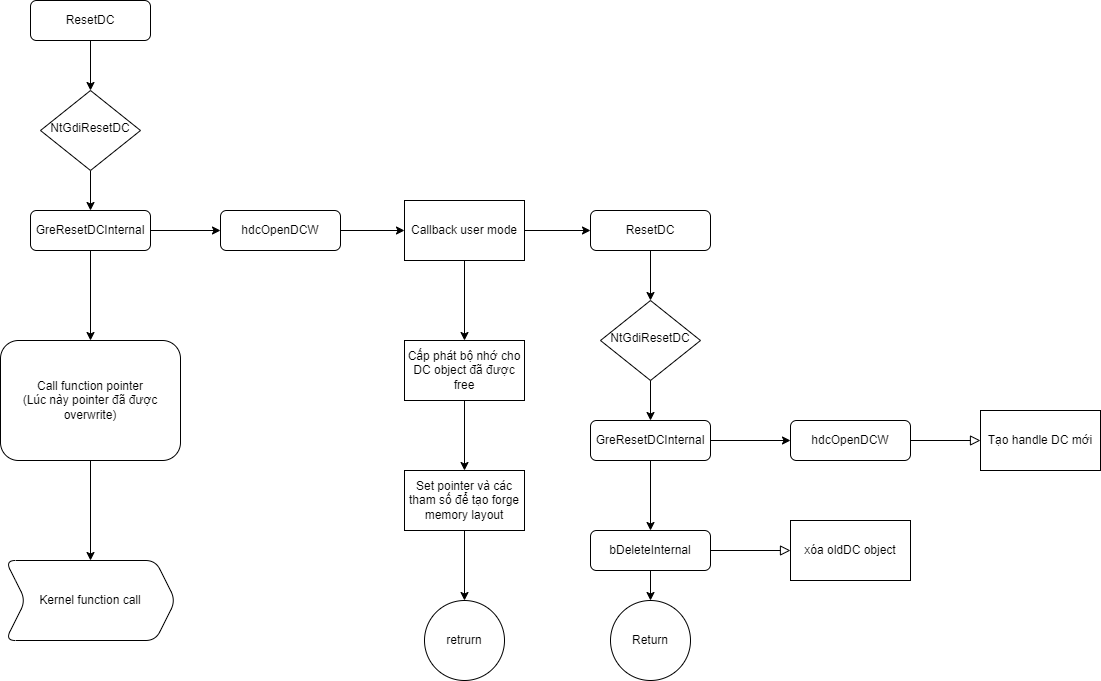


* Xây dựng được fake callback function và gọi lại hàm đó.

Vì sao lại sử dụng hook ***DrvEnableDriver*** ? Sử dụng hook này vì 2 lí do: trigger được lỗi use-after-free và kiểm soát được các đối số có thể truyền vào.

***ResetDC*** function sẽ giải phóng handle structure. ***ResetDC*** sẽ không kiểm tra xem handle có khả dụng hay không và usermode – callback sẽ được gọi trong ***ResetDC***. Điều này cho phép ta có thể giải phóng cùng một handle:

* + Khi ***ResetDC*** được gọi ở user mode, nó sẽ thực thi syscall ***NtGdiResetDC*** và chức năng bên trong của ***GreResetDCInternal***. Tại ***GreResetDCInternal()*** sẽ thực gọi hàm ***hdcOpenDCW()***
  + Hàm ***hdcOpenDCW()*** sẽ thực hiện callback lại user mode. Lúc này, ta sẽ gọi ***ResetDC*** lần thứ 2ở bên trong hook sẽ giải phóng được old DC object truyền vào
  + Sau khi thực hiện xong callback, ***hdcOpenDCW()*** quay trở lại ***GreResetDCInternal*** ở lần gọi đầu tiên, nhưng con trỏ không gian bộ nhớ vẫn trỏ tới old DC object đã bị giải phóng trước đó.
  + Trigger use-after-free thành công, ta có thể sử dụng nó để gọi tới kernel function tùy ý

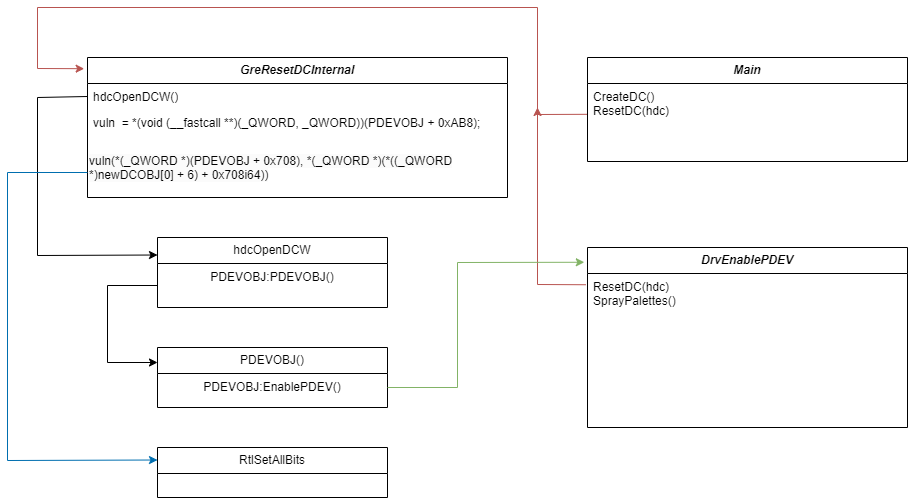


## 2) Sử dụng lại không gian bộ nhớ đã cấp phát cho DC object đã được free

Theo Kaspersky đã đề cập, để khai thác ta có thể sử dụng Gdi Palette và một kernel function call duy nhất để đạt được khả năng đọc/ghi bộ nhớ kernel tùy ý.

Sử dụng Palettes để chiếm bộ nhớ của DEV object trong oldDC object đã giải phóng và thay đổi con trỏ hàm UMPDDrvResetPDEV() => lệnh kernel function.

Sử dụng hàm RtlSet AllBits nhằm sửa đổi các quyền của process token để đạt được leo thang đặc quyền.



### Palettes Spray

Dựa vào “[windows rs3 exploitation primitive](https://census-labs.com/media/windows_10_rs2_rs3_exploitation_primitives.pdf)” thì kích thước của palettes có thể được kiểm soát nên có thể được dùng cho heap spraying.

* Tạo Palettes

*VOID* SprayPalettes(*DWORD* size)

{

    /\* Spray palettes to reclaim freed memory \*/

    DWORD palCount = (size - 0x90) / 4;

    DWORD palSize = sizeof(LOGPALETTE) + (palCount - 1) \* sizeof(PALETTEENTRY);

    LOGPALETTE\* lPalette = (LOGPALETTE\*)malloc(palSize);

    if (lPalette == NULL) {

        puts("[-] Failed to create palette");

        return;

    }

lPalette->palNumEntries = (WORD)palCount;

    lPalette->palVersion = 0x300;

    CreatePalette(lPalette);

}

* Tính toán kích thước có thể sử dụng từ DEV object. Ở Windows 10 build 1809, thì size của PDEVOBJ is 0xe30 nên ta có thể sử dụng tối đa size truyền vào là 0xe30 (<https://github.com/ZoloZiak/WinNT4/blob/master/private/ntos/w32/ntgdi/gre/pdevobj.hxx>)
* Theo POC được viết trên Windows 10 build 1607, ghi đè địa chỉ và con trỏ hàm ***RtlSetAllBits*** như sau:
  + p[0xD7] = globals::fakeRtlBitMapAddr
  + p[0x14D] = globals::rtlSetAllBits

Nhưng vì offset ở mỗi phiên bản lại khác nhau nên ta sẽ ghi đè offset trong khoảng POC đã dùng:

for (DWORD i = 0; i < 0x100; i++) {

        p[i] = fakeRtlBitMapAddr;

    }

    for (DWORD i = 0x100; i < (palSize)/8 ; i++) {

        p[i] = rtlSetAllBits;

    }

## 3) Hoàn thành forge memory layout để thực thi hàm gọi

### a) Create fakePRTL\_BITMAP

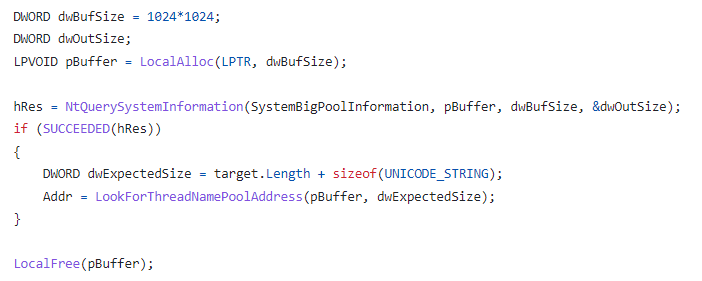
Từ win 8 trở lên, SMAP được bật nên kernel không thể lấy dữ liệu trong vùng người dùng, vì vậy không thể trực tiếp sử dụng tham số **PRTL\_BITMAP** cho hàm ***RtlSetAllBits()***

Để có thể tạo fakePRTL\_BITMAP ta có thể theo các bước sau:

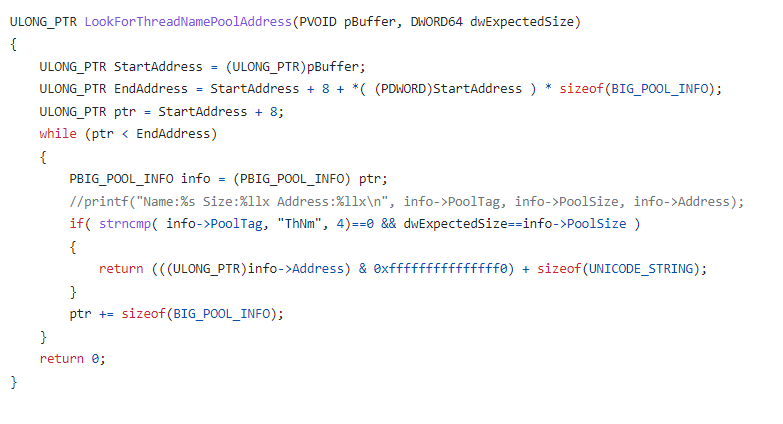
* Sử dụng **NtQuerySystemInformation** để leak địa chỉ kernel của Token
* Tạo fakePRTL\_BITMAP vào bên trong kernel memory bằng cách sử dụng **SetInformationThread**
* Leak địa chỉ của bitmap bằng cách sử dụng **NtQuerySystemInformation** với lớp **SystemBigPoolInformation**

**(**<https://gist.github.com/hugsy/d89c6ee771a4decfdf4f088998d60d19>**)**

* + Một kỹ thuật *exploit primitive* đã được đề cập bởi Kaspersky đó là dùng **NtQuerySystemInformation()** syscall cùng với **SystemBigPoolInformation(0x42)** để liệt kê “large pools” (bao gồm: size, kernel address,…)
  + Dump information:

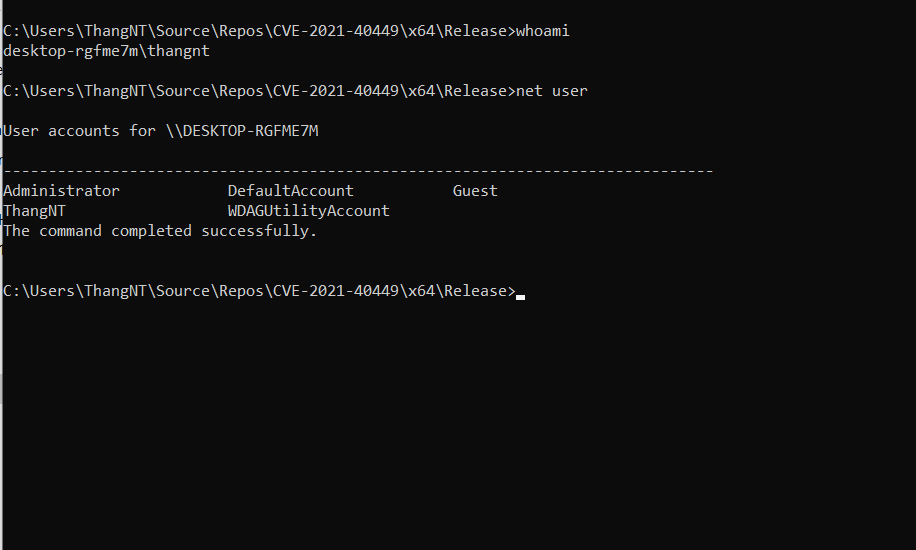


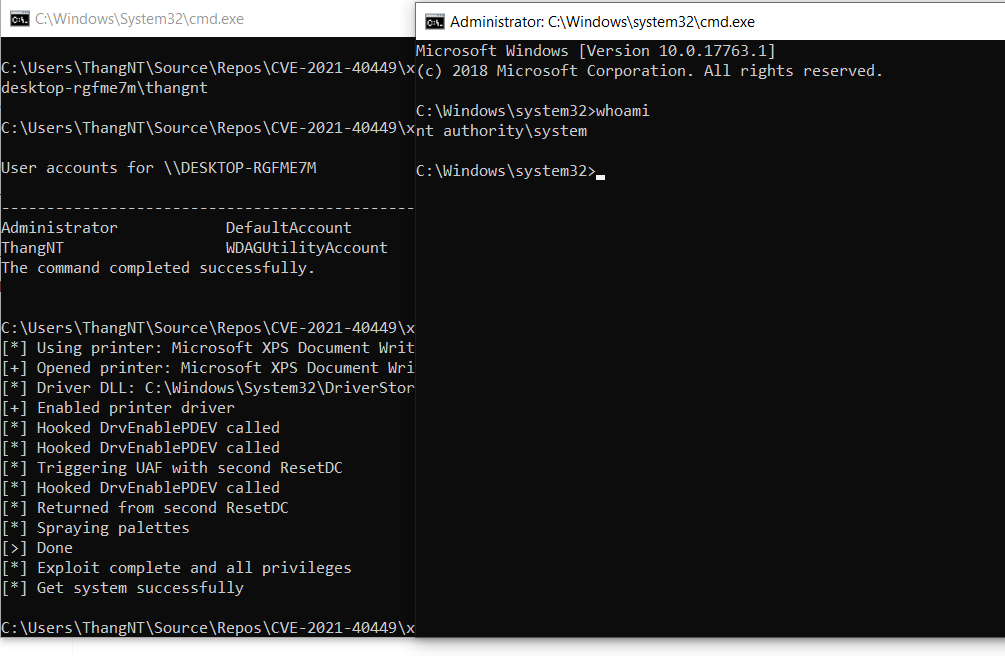
* + Tìm kiếm address với điều kiện size và **strncmp( info->PoolTag, "ThNm", 4)==0:**



* **fakePRTL\_BITMAP** BitMapHeder sẽ trỏ đến \_SEP TOKEN PRIVILEGES. Gọi ***RtlSetAllBits***(BitMapHeader) kích hoạt tất cả các đặc quyền bằng một kernel function call.
* Từ đây, người ta có thể lạm dụng các đặc quyền cao để lấy SYSTEM

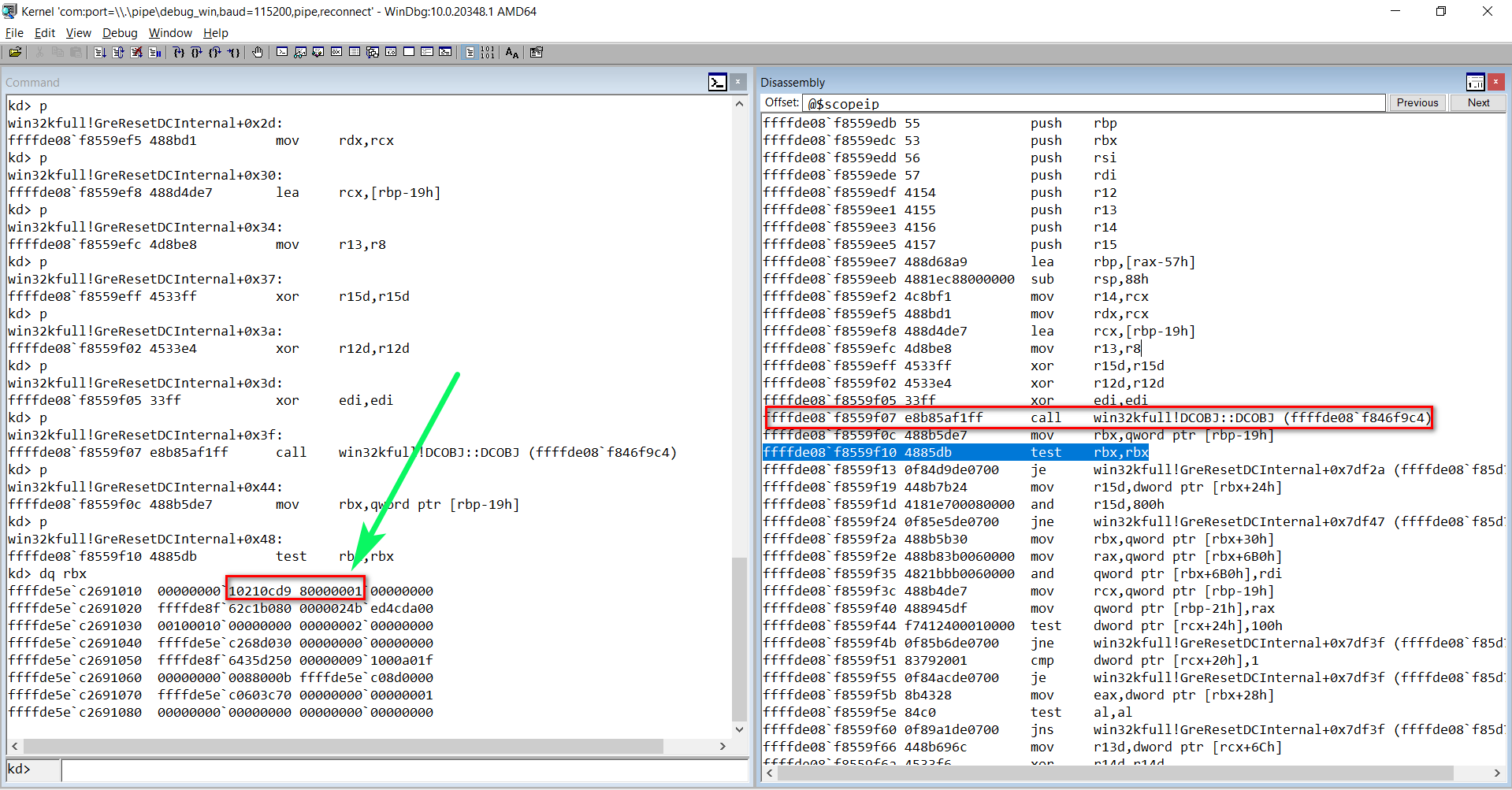
# IV) Result



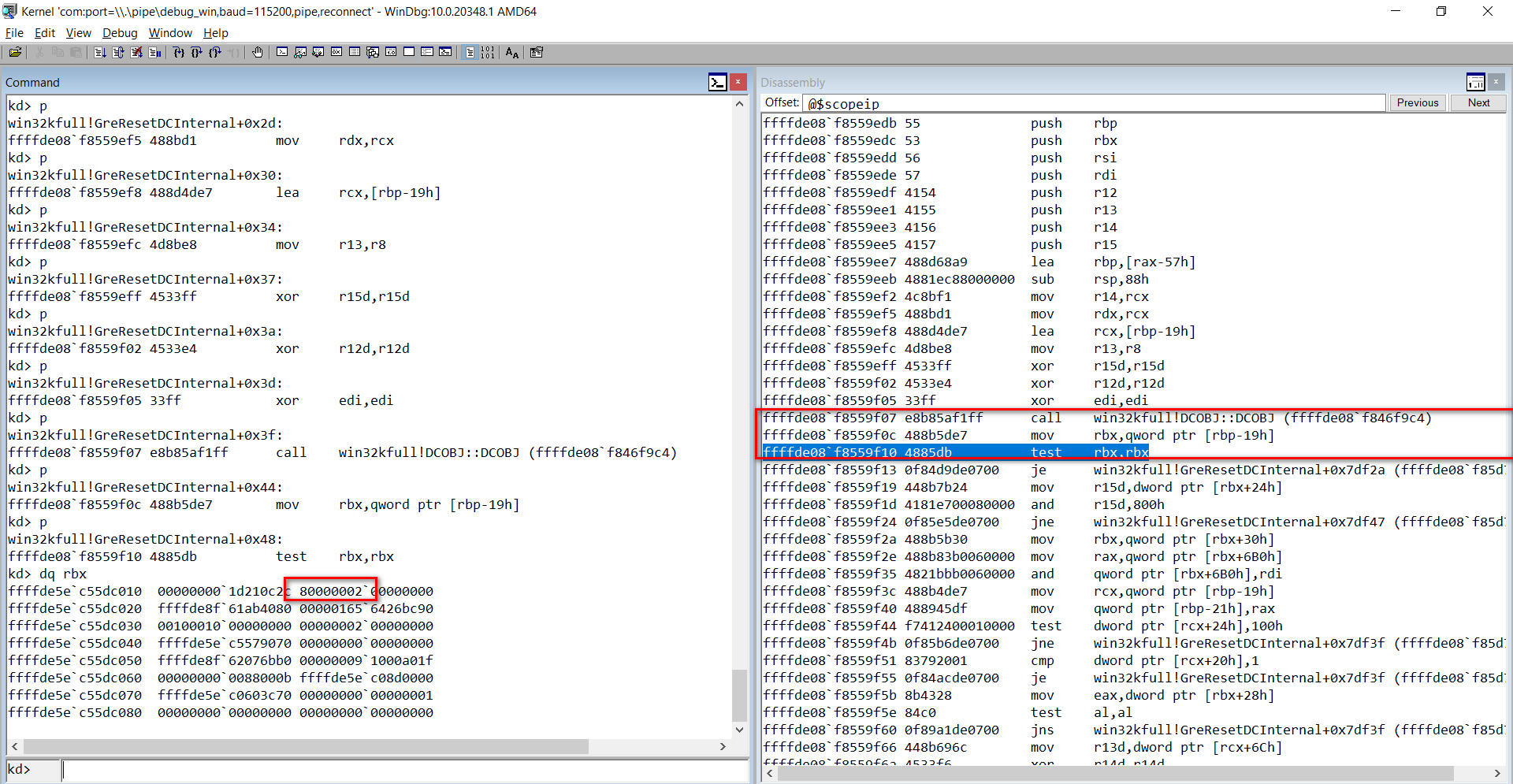


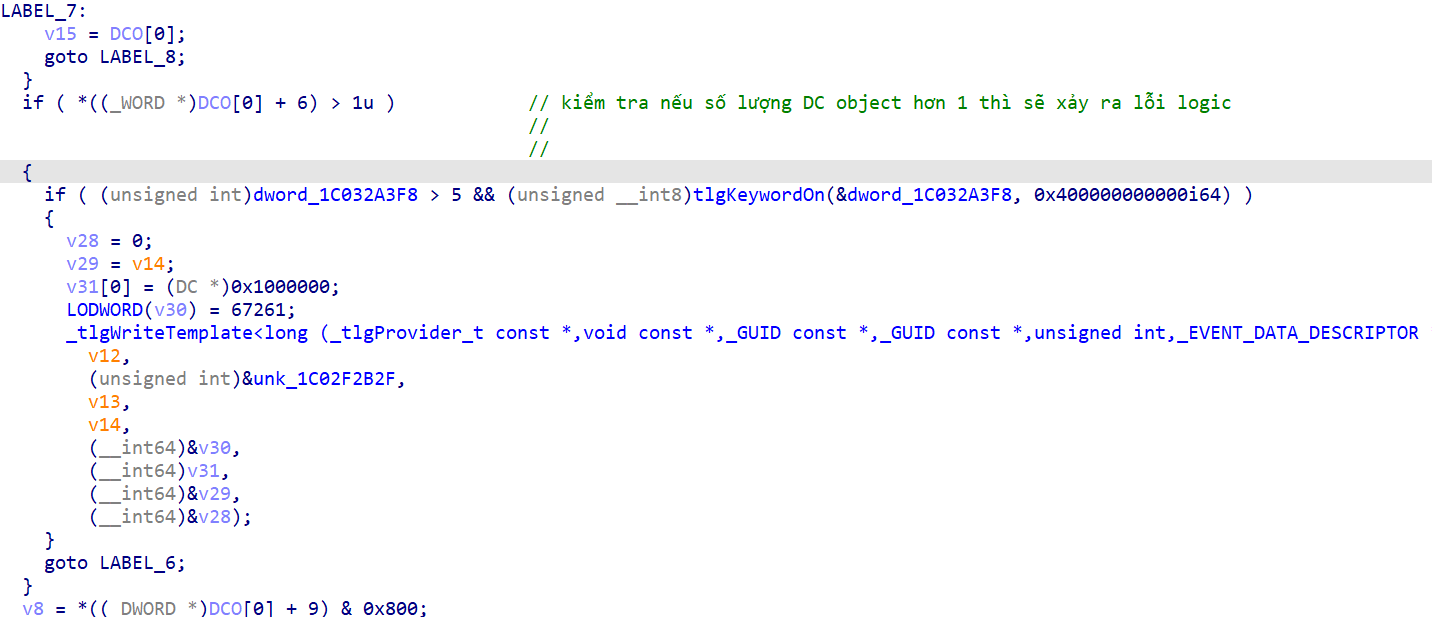
# V) The patch

Lỗ hổng xảy do khi gọi hàm **GreResetDCInternal** đã không kiểm tra con trỏ DC object xem có bất thường hay không. Khi tạo DC object sẽ được tạo, giá trị bộ đếm tham chiếu đối tượng sẽ tăng lên một lần mỗi lần được tham chiếu. Tiến trình xảy ra bình thường khi giá trị đếm chỉ bằng 1.



Nếu ResetDC được gọi lại trong callback function, **GreResetDCInternal** sẽ được gọi lần thứ hai và hàm tạo của DC Object sẽ được gọi lại và tham chiếu đối tượng DC sẽ được thêm lại và số lượng tham chiếu sẽ là 2 vào lúc này.



Do đó, việc xem xét trạng thái của DC object có thể được nhận ra bằng cách đánh giá số lượng tham chiếu của DC object. 

# VI) References

* <https://github.com/ly4k/CallbackHell/tree/cbeac75baccd66242fa9e3efa5d9d86892f02159>
* <https://securelist.com/mysterysnail-attacks-with-windows-zero-day/104509/>
* <https://cn-sec.com/archives/1868860.html>