

# ROP TEAM

Atelier NSEC 2023 - Analyse dynamique de pilotes windows

 July 17 2023-05-18

# Mise en place du laboratoire

L'atelier nécessite 2 machines (virtuelles ou physiques), une qui sera l'objet de l'analyse (cible) et une autre qui sera utilisée pour exécuter le déboggeur.

## Informations de connexion pour la VM

Utilisateur: administrator

Mot de passe: workshop2023!w00t

Copiez la VM depuis le lien dans #workshop sur discord ou depuis les clés usb prêtées pour l'occasion

Passez la clé au suivant après la copie.

# Introduction - Pourquoi attaquer les modules du noyau

Le temps que tous aient eu le temps de récupérer l'image.

- Le code qui s'exécute au niveau du noyau est privilégié par rapport aux logiciels même s'exécutant avec les privilèges system ou administrateur.
- Plus d'accès au niveau de la mémoire et des registres de cpu.

# Types d'exploits

On retrouve tous les classiques

- Lecture arbitraire, mémoire physique ou virtuelle
- Écriture arbitraire
- Dépassement de mémoire tampon, mitigé mais existe encore
- Use after free
- TOCTOU, race condition

# Stratégies d'exploitation

Exemple de stratégie: Elévation de privilèges

- Lire la liste des processus
- Parcourir la liste des processus, trouver un avec un token privilégié
- lire le token
- écrire le token dans son propre processus ou le processus de notre choix

Mon approche est la suivante:

1- Trouver des exemples de vulnérabilités exploités, modules kernel vulnérables

Quelques exemples:

- [https://github.com/chompie1337/Windows\\_LPE\\_AFD\\_CVE-2023-21768](https://github.com/chompie1337/Windows_LPE_AFD_CVE-2023-21768)
- <https://github.com/alfarom256/CVE-2022-3699>
- <https://github.com/mathisvickie/CVE-2021-21551>
- <https://github.com/alfarom256/HPHardwareDiagnostics-PoC>

## 2- Comprendre les stratégies d'exploitation

- cloner token d'un processus privilégié
- lire de la mémoire sensible (kernel based mimikatz)
- nuire aux outils de sécurité (<https://github.com/wavestone-cdt/EDRSandblast>)
- Dépassement de mémoire tampon sans écraser le canary. Voir s'il n'y a pas moyen d'influencer le flow d'exécution en écrasant les autres variables locales.



### 3- Trouver les fonctions importées utilisée dans la stratégie d'exploitation utilisée

- zwopensection
- mmapiospace
- memcpy, memmove
- halsetbusdata
- rdmsr, wrmsr
- etc

4- Amasser une liste de pilotes, cherchez sur les postes ou les environnements dont vous êtes responsable.

5- Recherche dans chaque drivers les fonctions utilisées lors d'exploits précédents

Les fonctions ainsi trouvées peuvent donner des indications sur la stratégie d'exploitation à privilégier

6- Analyser, fuzzer les modules contenant une ou plusieurs de ces fonctions afin de trouver de nouvelles vulnérabilités

## Articles publiés sur le sujet

<https://voidsec.com/reverse-engineering-and-exploiting-dell-cve-2021-21551>

<https://github.com/AzAgarampur/CorsairLLeak>

Fuzzer pour ioctl

<https://github.com/VoidSec/ioctlpus>

Loldrivers

<https://loldrivers.io/>

<https://github.com/magicword-io/LOLDrivers>

<https://medium.com/magicwordio/living-off-the-land-drivers-1-0-release-95af7d59fb89>

Windows syscalls

<https://github.com/j00ru/windows-syscalls>

# Mise en place du laboratoire - partie 2

## Note à propos de la virtualisation de la cible

Les deux produits de virtualisation testés lors de la présentation de cet atelier sont KVM et VirtualBox.

Microsoft emploie un protocole propriétaire au lieu de TCP pour les communications réseau du déboggeur s'il détecte que la virtualisation est utilisée.

La connexion au déboggeur fonctionne sans modifications sous le hyperviseur **VirtualBox**.

Dans le cas ou le hyperviseur **KVM** est utilisé, la section hyperv de la feuille XML décrivant la machine cible doit être modifiée. Remplacez le contenu de la balise `<hyperv>` par le suivant:

```
<hyperv>
  <relaxed state='on' />
  <vapic state='on' />
  <spinlocks state='on' retries='4096' />
  <vpindex state='on' />
  <runtime state='on' />
  <synic state='on' />
  <stimer state='on'>
    <direct state='on' />
  </stimer>
  <reset state='on' />
  <vendor_id state='on' value='KVMKVMKVM' />
  <frequencies state='on' />
  <reenlightenment state='on' />
  <tlbflush state='on' />
  <ipi state='on' />
  <evmcs state='on' />
</hyperv>
```

Référence: <https://www.osr.com/blog/2021/10/05/using-windbg-over-kdnet-on-qemu-kvm/>

Notez que les autres technologies de virtualisation n'ont **pas** été testées.

# Configuration des machines virtuelles

## VirtualBox

Si vous utilisez `VirtualBox`, il est sugg  rer de cr  er une machine virtuelle nomm  e `workshop`    partir de l'image disque fournie, `workshop.qcow2`.

Apr  s, cr  er un premier clone (Linked Clone) nomm   `cible` ou `target`    partir de la VM cr   e pr  c  demment.

Puis, cr  er un deuxi  me clone (toujours linked clone) nomm   `debugger` ou `windbg`, toujours    partir de la premi  re VM.

R  glez le r  seau sur `internal` ou `host only`, pas `NAT`, ni `bridged` pour   viter d'exposer la VM vuln  rable au r  seau.

Enfin, activez le copier-coller (bi-directionnal clipboard) afin de pouvoir copier la cl   g  n  r  e par `kdnet` dans le d  boggeur.

# KVM

Sous `KVM`, il faut créer une première VM nommée `cible` en choisissant l'option `import from disk image`. Puis, lors de la sélection du stockage, créer une image disque du même nom que la VM mais en spécifiant `workshop.qcow2` dans l'option `backing store`.

**Add a Storage Volume**

Create storage volume

Details XML

Create a storage unit to be used directly by a virtual machine.

Name: debugged .qcow2

Format: qcow2

▼ Backing store

Path: /home/vm/machines/workshop.qcow2 Browse...

**Storage Volume Quota**  
machines's available space: 765.49 GiB

Capacity: 20.0 - + GiB

☐ Allocate entire volume now

Cancel Finish



## Préparation de la cible

- Installer windows software development kit (SDK)

<https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/>

Récupérez l'exécutable `kdnet.exe` ainsi que le fichier `VerifiedNICList.xml` depuis `C:\Program Files (x86)\Windows Kits\10\Debuggers\x64` vers `c:\temp` ou un autre répertoire de votre choix.

Sur la machine virtuelle fournie, ces fichiers se trouvent dans le répertoire

`C:\workshop2023\kdnet`

## Chargement d'un pilote sous windows

Remplacez `vulnerable` par le nom du pilote que vous souhaitez tester.

Dans la VM fournie, le pilote à tester se trouve à l'emplacement suivant:

```
C:\Users\Administrator\Desktop\go-kernel-exploit\vulnerable.sys
```

```
sc.exe create vulnerable type=kernel start=demand error=normal binpath=C:\Users\Administrator\Desktop\go-kernel-exploit\vulnerable.sys
```

Pour un démarrage automatique d'un pilote

```
sc.exe create vulnerable type=kernel start=auto error=normal binpath=C:\Users\Administrator\Desktop\go-kernel-exploit\vulnerable.sys
```

Valider que le module est bien chargé

```
sc.exe query vulnerable
```

Sur la VM fournie, lancez `powershell` dans le répertoire `C:\workshop2023\kdnet`. Lancez `kdnet.exe` pour activer le débogage réseau. Notez la clé, qui devra être entrée dans le déboguer pour effectuer la connexion.

```
kdnet.exe <HostComputerIPAddress> <YourDebugPort>
```

Par exemple: PS C:\kdnet> .\kdnet.exe 192.168.222.63 54321

```
Enabling network debugging on Intel(R) 82574L Gigabit Network Connection.  
Manage-bde.exe not present. Bitlocker presumed disabled.
```

To debug this machine, run the following command on your debugger host machine.

```
windbg -k net:port=54321,key=30rhan6xwlb4t.14m2pvtts6v6.1k5454xoata1o.2ee6badkh1gcv
```

Then reboot this machine by running `shutdown -r -t 0` from this command prompt.

# Préparation du déboguer

## Facultatif

- Installer windows software development kit (SDK)

<https://developer.microsoft.com/en-us/windows/downloads/windows-sdk/>

- Récupérez "windbg preview" depuis le microsoft store. Le déboggeur "preview" présente plusieurs améliorations par rapport à la version ancestrale.

<https://apps.microsoft.com/store/detail/windbg-preview/9PGJGD53TN86>

## Contournement du magasin (sideloading)

Cependant, sous Windows Serveur 2019, le `Microsoft Store` n'est pas disponible. Il peut être aussi souhaitable de télécharger l'application sans devoir s'authentifier auprès de Microsoft.

Un service web permet de récupérer l'URL qui permet de télécharger l'application depuis les serveur de Microsoft.

<https://store.rg-adguard.net>

L'URL aura la forme suivante, choisir le fichier avec l'extension **APPX**:

```
http://tlu.dl.delivery.mp.microsoft.com/filestreamingservice/files/4c032be9-8e5e-490e-9d84-3eb4392501ad?P1=1669065332&P2=404&P3=2&P4=Fnt2RhuIDxo15%2bpU9ax9jzi245Jp4FpcZE2y0JCS98H1m0JSA2gzjk8l3ydg19%2bZ3Xb2q6jQZUEgxERIyjt02A%3d%3d
```

## Installation d'un paquet APPX

Une fois le fichier **APPX** téléchargé, il doit être installé avec la commande powershell suivante

```
Add-AppxPackage -Path "C:\Path\to\File.Appx"
```

## Problème d'accès / réseau

Si une erreur se produit lors du lancement de windbg, simplement copier les fichiers exécutables depuis `C:\Program`

`Files\WindowsApps\Microsoft.WinDbg_1.2103.1004.0_neutral__8wekyb3d8bbwe` dans un autre répertoire.

Ceci est dû au fait que la propriété des fichiers est attribuée à `TrustedInstaller` et que l'administrateur n'a qu'un accès limité.

Sur la VM fournie, utilisez le raccourci sur le bureau.

Pour lancer le débogueur, dans windbg preview, allez dans le menu file / attach to kernel puis copiez le port ainsi que la clé obtenue lors de l'exécution de kdnet sur la cible.

## Connexion avec windbg

Lancez windbg preview

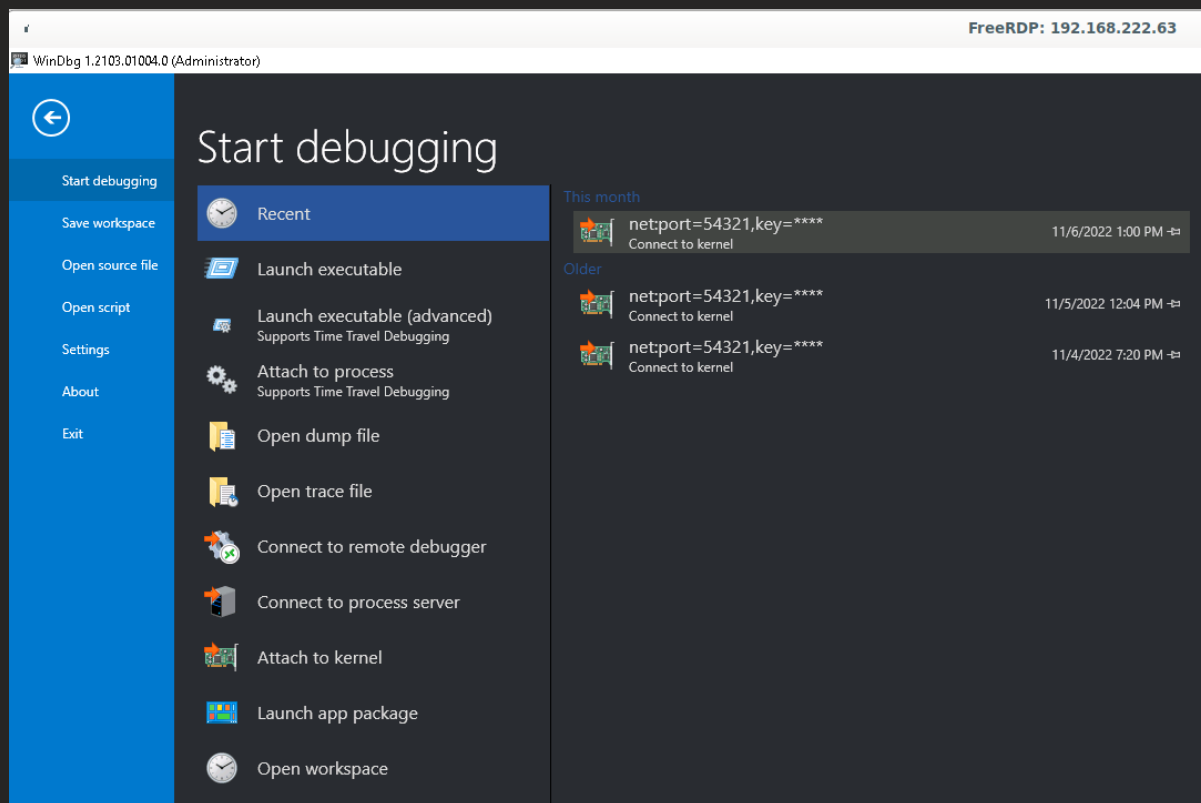
- Accédez au menu `file`, cliquez sur `attach to kernel` puis `connection string`
- Entrez la clé ainsi que le port configurés sur la cible

`kdnet.exe` a retourné

`port=54321, key=30rhan6xwlb4t.14m2pvttms6v6.1k5454xoata1o.2ee6badkh1gcv` donc:

- Port = 54321
- Key = 30rhan6xwlb4t.14m2pvttms6v6.1k5454xoata1o.2ee6badkh1gcv





## Accès réseau par windbg

Windbg doit pouvoir récupérer les symboles requis depuis les domaines suivantes

- msdl.microsoft.com
- blob.core.windows.net

Une connexion sortante vers ces deux domaines doit être autorisée lors de la première exécution. Réglez le proxy dans le navigateur IE/Edge (s'il y a lieu) La machine hébergeant le débogueur peut être mise hors ligne après.

Pour recharger les symboles, utilisez la commande `reload`

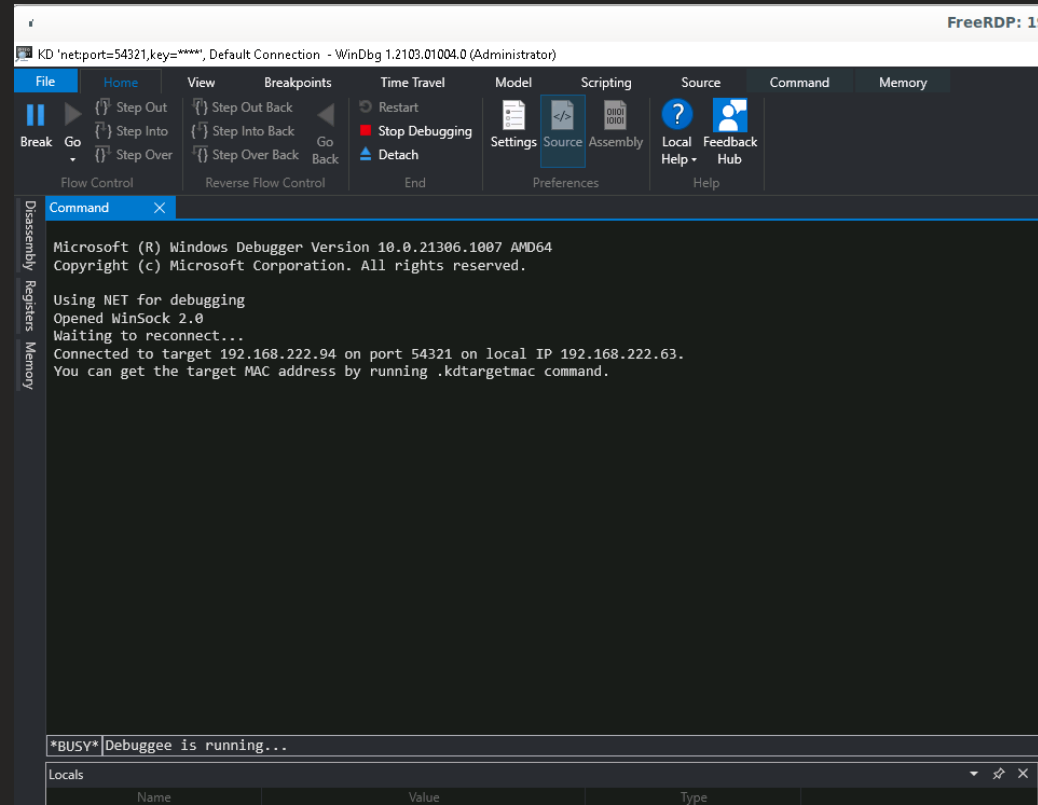
```
.reload nt
```

Pour activer/désactiver les messages concernant le chargement des symboles, utilisez les commandes suivantes

```
!sym noisy  
!sym quiet
```

# Statut BUSY:debuggee is running

Vous devriez voir **BUSY** et **debuggee is running**



Redémarrez la cible tel qu'indiqué avec **shutdown -r -t 0** et attendez que **debuggee is running** apparaisse de nouveau après le redémarrage

## Commandes de base dans windbg

Référence: <https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debug-universal-drivers---step-by-step-lab--echo-kernel-mode->

Cheat sheet: <https://github.com/repnz/windbg-cheat-sheet>

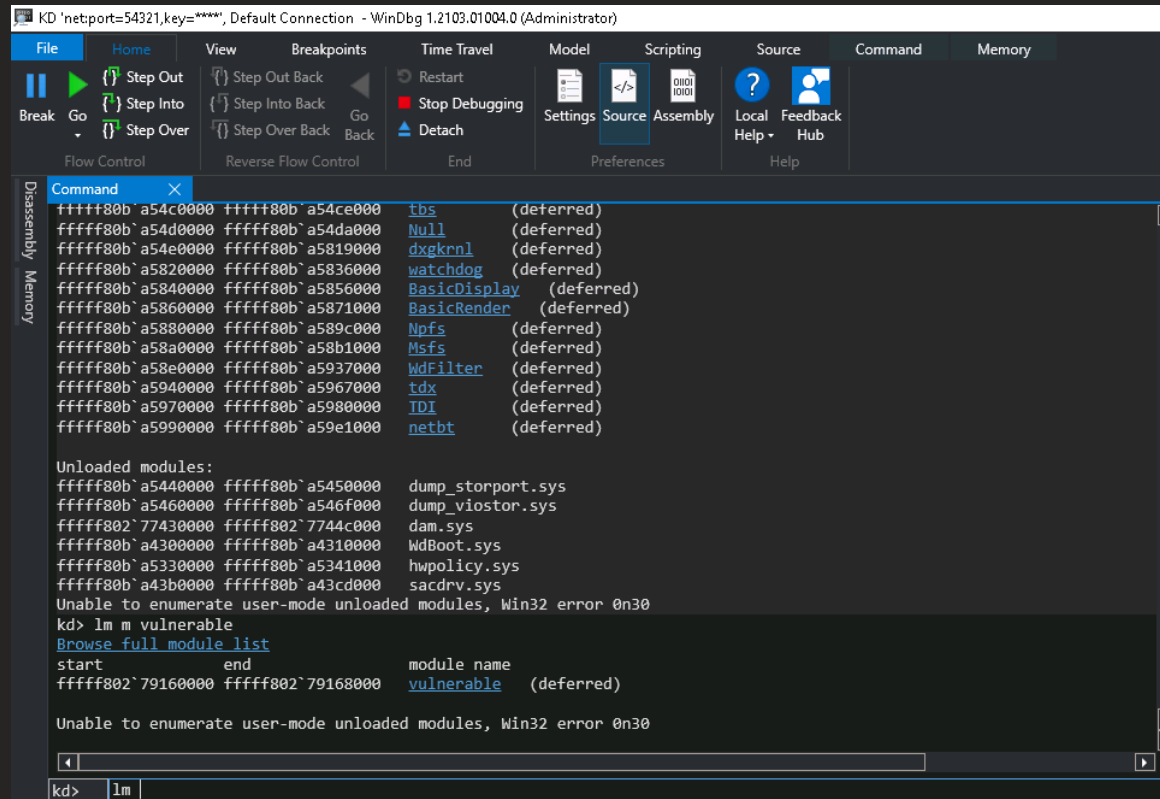
Voici quelques commandes afin de valider le bon fonctionnement de windbg

`lm`: Lists les modules

`x /D nt!io*` Lister les fonctions du module **nt** dont le nom commence par *io\**

# Pour lister un module en particulier

`lm m vulnerable` Lister le module vulnerable:



The screenshot shows the WinDbg interface with the Command window open. The title bar reads "KD 'net:port=54321,key=\*\*\*\*', Default Connection - WinDbg 1.2103.01004.0 (Administrator)". The Command window displays the output of the command `lm m vulnerable`, which lists loaded modules and their addresses. The output is as follows:

```
fffff80b`a54c0000 fffff80b`a54ce000 tbs (deferred)
fffff80b`a54d0000 fffff80b`a54da000 Null (deferred)
fffff80b`a54e0000 fffff80b`a5819000 dxgkrnl (deferred)
fffff80b`a5820000 fffff80b`a5836000 watchdog (deferred)
fffff80b`a5840000 fffff80b`a5856000 BasicDisplay (deferred)
fffff80b`a5860000 fffff80b`a5871000 BasicRender (deferred)
fffff80b`a5880000 fffff80b`a589c000 Npfs (deferred)
fffff80b`a58a0000 fffff80b`a58b1000 Msfs (deferred)
fffff80b`a58e0000 fffff80b`a5937000 WdFilter (deferred)
fffff80b`a5940000 fffff80b`a5967000 tdx (deferred)
fffff80b`a5970000 fffff80b`a5980000 IDI (deferred)
fffff80b`a5990000 fffff80b`a59e1000 netbt (deferred)

Unloaded modules:
fffff80b`a5440000 fffff80b`a5450000 dump_storport.sys
fffff80b`a5460000 fffff80b`a546f000 dump_viosstor.sys
fffff802`77430000 fffff802`7744c000 dam.sys
fffff80b`a4300000 fffff80b`a4310000 WdBoot.sys
fffff80b`a5330000 fffff80b`a5341000 hwpolicy.sys
fffff80b`a43b0000 fffff80b`a43cd000 sacdrv.sys
Unable to enumerate user-mode unloaded modules, Win32 error 0n30
kd> lm m vulnerable
Browse full module list
start end module name
fffff802`79160000 fffff802`79168000 vulnerable (deferred)

Unable to enumerate user-mode unloaded modules, Win32 error 0n30
```

The Command window also shows the command prompt `kd> lm` at the bottom.

# Visite guidée de vulnerable.sys

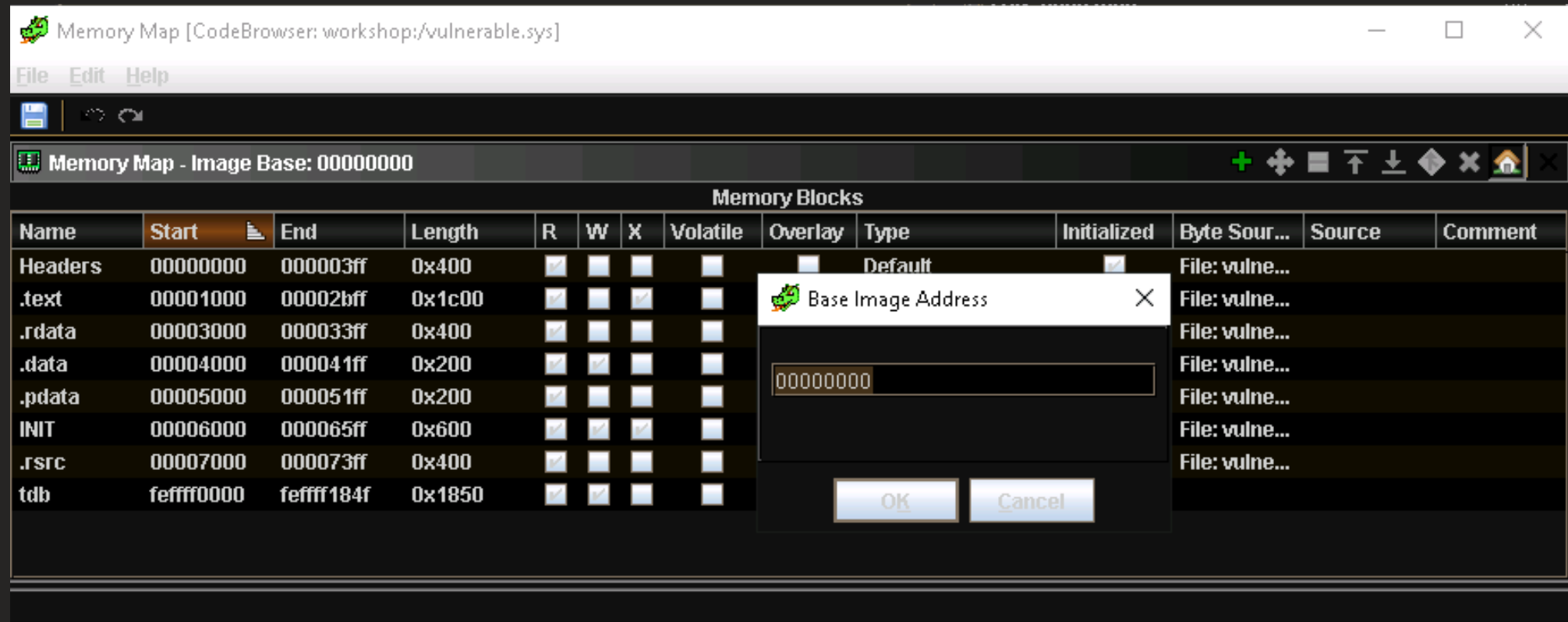
Une analyse statique rapide permet de préparer le terrain pour l'analyse dynamique.

## Préparation

- Créez un nouveau projet Ghidra
- Importez le module vulnerable
- Lancez le code browser, lancez l'analyse

Pour faciliter la création de points d'arrêt dans windbg, réglez l'adresse de base à 0x0

Cliquez sur `display memory map` et sur l'icone en forme de maison `set image base` et inscrivez 0



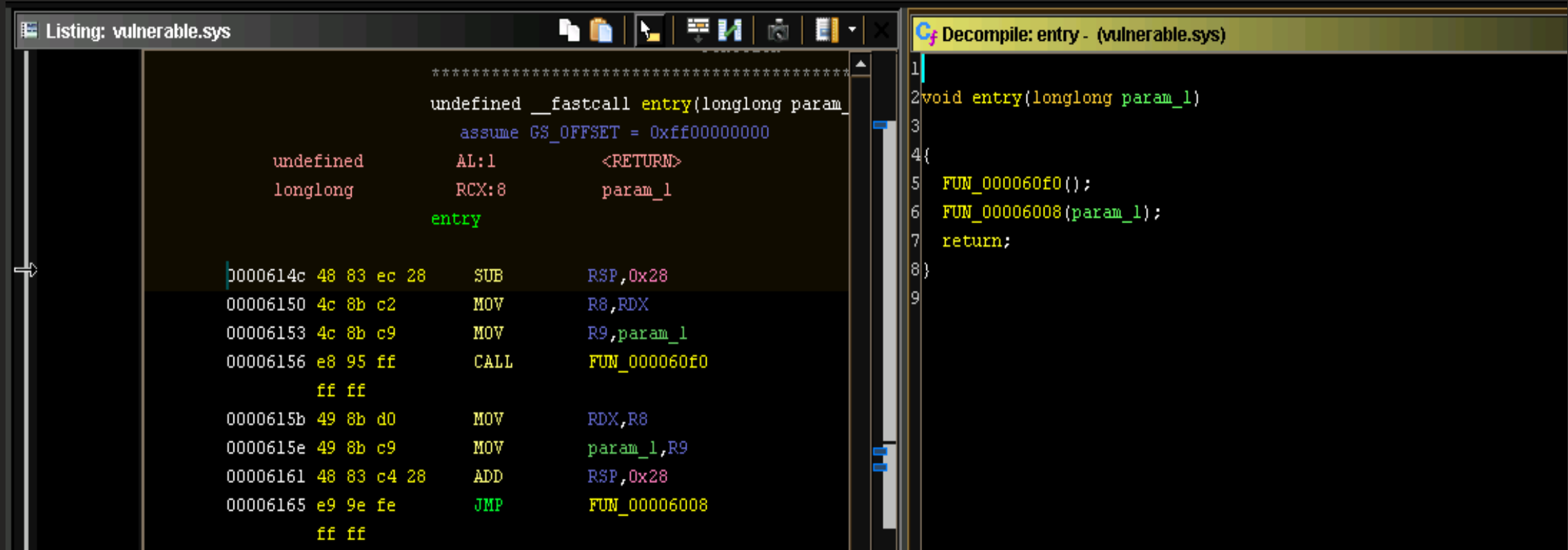


Cela permet d'utiliser directement l'adresse d'une fonction depuis ghidra pour créer un point d'arrêt dans windbg même si le ASLR est présent.

Windbg permet de simplement d'utiliser le nom du module pour représenter l'adresse ou il est chargé au moment du test.

Par exemple, la fonction qui sert de point d'entrée (entry):

```
bp vulnerable+614c
```



The screenshot displays a debugger window with two panes. The left pane, titled 'Listing: vulnerable.sys', shows assembly code for the 'entry' function. The right pane, titled 'Decompile: entry - (vulnerable.sys)', shows the decompiled C++ code for the same function.

**Assembly Listing:**

```
*****
undefined __fastcall entry(longlong param_1)
    assume GS_OFFSET = 0xff00000000
    undefined AL:1 <RETURN>
    longlong RCX:8 param_1
    entry
0000614c 48 83 ec 28 SUB RSP,0x28
00006150 4c 8b c2 MOV R8,RDX
00006153 4c 8b c9 MOV R9,param_1
00006156 e8 95 ff CALL FUN_000060f0
          ff ff
0000615b 49 8b d0 MOV RDX,R8
0000615e 49 8b c9 MOV param_1,R9
00006161 48 83 c4 28 ADD RSP,0x28
00006165 e9 9e fe JMP FUN_00006008
          ff ff
```

**Decompile: entry - (vulnerable.sys)**

```
1
2 void entry(longlong param_1)
3
4 {
5     FUN_000060f0();
6     FUN_00006008(param_1);
7     return;
8 }
9
```

## Analyse statique rapide dans Ghidra

Appliquer la procédure proposée par SpecterOps pour faciliter l'analyse statique.

Voir le billet de Matt Hand ici:

<https://posts.specterops.io/methodology-for-static-reverse-engineering-of-windows-kernel-drivers-3115b2efed83>

Dépôt GitHub pour ntddk\_64.gdt:

<https://github.com/0x6d696368/ghidra-data/tree/master/typeinfo>

Cependant, Ghidra donne un message d'erreur lors de l'étape "Apply function data types".

Si c'est le cas, mettre ntddk\_64.gdt dans

\$GHIDRAHOME/./Ghidra/Features/Base/data/typeinfo/win32 et relancez Ghidra

Cette étape a été fait d'avance dans la VM fournie.

## Documentation structures de données du kernel windows

<https://www.vergiliusproject.com/>

Mauvais certificat mais quand même utile

<https://undocumented.ntinternals.net/>

# Point d'entrée d'un pilote

Commencez l'analyse statique par `entry`

Listing: vulnerable.sys

```
*****
undefined __fastcall entry(longlong param_
    assume GS_OFFSET = 0xff00000000

    undefined    AL:1    <RETURN>
    longlong     RCX:8    param_1
    entry

0000614c 48 83 ec 28    SUB     RSP,0x28
00006150 4c 8b c2      MOV     R8,RDX
00006153 4c 8b c9      MOV     R9,param_1
00006156 e8 95 ff      CALL   FUN_000060f0
    ff ff
0000615b 49 8b d0      MOV     RDX,R8
0000615e 49 8b c9      MOV     param_1,R9
00006161 48 83 c4 28    ADD     RSP,0x28
00006165 e9 9e fe      JMP     FUN_00006008
    ff ff
```

Decompile: entry - (vulnerable.sys)

```
1
2 void entry(longlong param_1)
3
4 {
5     FUN_000060f0();
6     FUN_00006008(param_1);
7     return;
8 }
9
```

Creation d'un Ioctl, fonction vulnerable+614c

Voir documentation pour IoCreateDevice

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

Voir documentation pour driverdispatch callback:

[https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nc-wdm-driver\\_dispatch](https://learn.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nc-wdm-driver_dispatch)

# Handler: vulnerable+101c

Les numéros de IOCTL deviennent visibles dans le code décompilé Ex: 0x222808

```
Decompile: FUN_0000106c - (vulnerable.sys)
28  longlong in_CR8;
29
30  p_Var3 = (param_2->Tail).Overlay.field3_0x30.field1_0x10.CurrentStackLocation;
31  p1Var4 = *(longlong **)((longlong)&(param_2->AssociatedIrp).MasterIrp + 4);
32  iVar9 = 0;
33  (param_2->IoStatus).field0_0x0.Status = 0;
34  (param_2->IoStatus).Information = 0;
35  if (p_Var3->MajorFunction != '\x0e') goto LAB_000018a9;
36  uVar17 = (p_Var3->Parameters).QueryDirectory.FileIndex;
37  if (uVar17 < 0x222859) {
38      if (uVar17 == 0x222858) {
39          uVar11 = FUN_00001a4c((ushort *)p1Var4);
40          (param_2->IoStatus).field0_0x0.Status = (int)uVar11;
41          if (-1 < (int)uVar11) {
42              (param_2->IoStatus).Information = 0x204;
43          }
44          goto LAB_000018a9;
45      }
46      if (uVar17 < 0x222831) {
47          if (uVar17 != 0x222830) {
48              if (uVar17 != 0x222808) {
49                  if (uVar17 == 0x22280c) {
50                      puVar10 = (undefined4 *)MmMapIoSpace(*p1Var4,*(undefined4 *) (p1Var4 + 1),0);
51                      if (puVar10 == (undefined4 *)0x0) {
52                          iVar9 = -0x3fffffff;
53                      }
54                      else {
55                          puVar14 = (undefined4 *)p1Var4[2];
56                          iVar16 = *(int *) (p1Var4 + 1);
57                          puVar13 = puVar10;
58                          while (iVar16 != 0) {
59                              iVar2 = *(int *) ((longlong)p1Var4 + 0xc);
60                              if (iVar2 == 0) {
                                  uVar6 = *(undefined *)puVar14;
```

# Analyse dynamique partie 1, fuzzing

Le fuzzing permet d'appuyer notre analyse de deux façons:

- Il permet de comprendre la fonction des variables locales car leur nom significatif a été perdu lors de la compilation. On peut réassigner un nom significatif aux variables après avoir compris leur fonction.

- Le fuzzing agit aussi comme véhicule pour circuler à l'intérieur du code pour atteindre les parties qui nous interesse.

On peut découvrir quelles valeurs envoyer dans le ioctl afin de pouvoir traverser certaines conditions (if, else, switch). Ce travail peut aussi être appuyé par des logiciels comme **Angr**.



## Application user space, go-kernel-fuzz.exe

Utilise DeviceIoControl() pour communiquer avec le pilote (driver) via le périphérique (device) créé avec IoCreateDevice par le pilote.

Voir doc pour DeviceIoControl

```
BOOL DeviceIoControl(  
    [in]          HANDLE          hDevice,      # Descripteur de fichier  
    [in]          DWORD           dwIoControlCode,  
    [in, optional] LPVOID         lpInBuffer,   # Buffer du userspace vers kernel  
    [in]          DWORD           nInBufferSize,  
    [out, optional] LPVOID        lpOutBuffer,  # Buffer du kernel vers userspace  
    [in]          DWORD           nOutBufferSize,  
    [out, optional] LPDWORD       lpBytesReturned,  
    [in, out, optional] LPOVERLAPPED lpOverlapped  
);
```

Voir répertoire `C:\Users\Administrator\Desktop\go-kernel-fuzz` sur la VM fournie

Se compile avec go build

```
go build
```

Le code source se trouve dans le fichier `main.go`

Voir fonction fuzzIOctl

Utilise `syscall.DeviceIoControl()` pour communiquer avec le pilote après avoir ouvert le fichier crée par le pilote avec `IoCreateDevice()`.

Utilise le générateur de patron de pwntools pour fuzzer les ioctls.

## Commandes windbg utiles

Voici les principales commandes windbg utilisées durant la préparation de cet atelier.

## Affichage de données

db = display bytes (attention au endianness)

dq = display quad word (64 bits, tien compte du endianness et affiche les adresses à l'endroit

Suffixe 'L' pour la quantité (address range)

ex: dq rdx L1 pour afficher un quad word à l'adresse pointée par rdx

Peut-être utilisé sur un registre qui contient un pointeur

Exemple, consultez le contenu de la mémoire pointée par le registre rsi après l'exécution de l'instruction suivante:

```
0x....134a 488b7218      mov      rsi,qword ptr [rdx+18h]  
  
dq rsi
```

## Écriture en mémoire

Si on a besoin d'un déboguage actif

eq address value

Ex: correction du canary, inscrire **0xffffebd8b36ebc49** a l'adresse **fffff187`e256e7b0** sur la pile pour restaurer un canary écrasé par un dépassement de mémoire tampon.

Pour le cas ou on ne voudrait pas s'attarder immédiatement au contournement du canary.

```
kd> eq fffff187`e256e7b0 fffffebd8`b36ebc49
kd> dq rsp
fffff187`e256e750  00000000`0000020c 00000000`00000030
fffff187`e256e760  fffff187`e256e7e9 fffff802`0260004d
fffff187`e256e770  fffff9e85`0000020c 00000000`00000030
fffff187`e256e780  00000000`20206f49 00000000`00000000
fffff187`e256e790  00000000`00000000 02020202`02020202
fffff187`e256e7a0  03030303`03030303 00000000`0000007f
fffff187`e256e7b0  fffffebd8`b36ebc49 00000000`00000018
fffff187`e256e7c0  00000000`00000002 ffffc90b`e9705890
```

## Affichage et édition des registres

Par exemple, pour le cas où on aurait besoin de modifier la valeur de retour d'une fonction, inscrite dans le registre rax.

commande r

Affichage: r registre

Édition: r registre valeur

Par exemple, mettre 0x18 dans le registre rax et valider que rax a bien été modifié.

```
kd> r rax=18  
kd> r rax  
rax=000000000000000018
```

## Exploration des structures

commande dt

Pour la structure driver object

```
kd> dt nt!_DRIVER_OBJECT
```

Pour la structure irp

```
kd> dt nt!_IRP
```

## Points d'arrêts (breakpoints)

Mettre breakpoint a l'adresse du handler

Base address + 0x106c, notez que windbg assume l'utilisation de l'hexadécimal par défaut, pas besoin du préfixe `0x`.

```
bp vulnerable+106c
```

Pour lister les breakpoints

```
bl
```



Pour retirer un breakpoint

```
bc + numéro du breakpoint
```

Pour retirer tous les breakpoints d'un coup

```
bc *
```

## Lister les fonctions disponibles

lister les fonctions disponibles dans les modules chargés. Supporte l'asterisque `*` pour faciliter la recherche avec une partie du nom.

`x /D module!fonction`

`x /D vulnerable!*`

`x /D nt!mmmapiospace`

`x /D nt!io*`

## windbg scripting

<https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/windbg-scripting-preview>



.for loops



.foreach

## en cas de crash

!analyze -v

## Obtenir un stack strace

commande k, par exemple kPn

```
kd> kPn
# Child-SP          RetAddr           Call Site
00 ffffffff802`772658d8 ffffffff802`774e2a40 nt!DbgBreakPointWithStatus
01 ffffffff802`772658e0 ffffffff802`774e2911 kdnic!TXTransmitQueuedSends+0x120
02 ffffffff802`77265920 ffffffff802`74ed5439 kdnic!TXSendCompleteDpc+0x141
03 ffffffff802`77265960 ffffffff802`74ed6307 nt!KiProcessExpiredTimerList+0x159
04 ffffffff802`77265a50 ffffffff802`7506e74a nt!KiRetireDpcList+0x4a7
05 ffffffff802`77265c60 00000000`00000000 nt!KiIdleLoop+0x5a
```

## Résoudre les erreurs

```
kd> !gle  
LastErrorValue: (Win32) 0 (0) - The operation completed successfully.  
LastStatusValue: (NTSTATUS) 0xc0000034 - Object Name not found.
```

Sinon cherchez (NTSTATUS) + code erreur

# Exercice de Fuzzing

## Fuzzing des IOCTL

Pour savoir quoi écrire dans DeviceIOControl() afin d'obtenir une influence sur le comportement du code dans le noyau.

Utilisation de pwntools pour un fuzzer de base

Génération d'un patron en python avec pwntools

```
from pwn import *  
a = cyclic_gen()  
a.get(24)  
b'aaaabaaacaaadaaaeaaaafaaa'
```

## Tracage d'un ioctl

Testez par exemple avec **0x222808**

Activez un point d'arret à l'endroit où les données provenant du userspace seraient susceptibles d'être utilisées à votre avantage

Exemple: vulnerable+108c

La commande db rsi permet de constater que le registre pointe sur le patron copié dans le tampon d'entrée (bufin)

```
kd> db rsi
```

```
ffff998f`c7227c40  61 61 61 61 62 61 61 61-63 61 61 61 64 61 61 61  aaaabaaacaaadaaa
ffff998f`c7227c50  65 61 61 61 66 61 61 61-67 61 61 61 68 61 61 61  eaaafaaagaaahaaa
ffff998f`c7227c60  69 61 61 61 6a 61 61 61-6b 61 61 61 6c 61 61 61  iaaajaaakaaalaaa
```

Suivez l'exécution avec "step into" (commande **t**) jusqu'à l'adresse vulnerable+10ab, vous verrez le numéro du ioctl copié dans le registre r8

```
mov     r8d,dword ptr [rax+18h]
```

Après l'exécution de cette instruction, le registre r8 devrait contenir 0x222808



## Gros switch case ou tas de if/else

Observez le "switch case" avec les différents # de ioctl à partir de vulnerable+10cb

Mettre un breakpoint à cet emplacement

```
bp vulnerable+10cb
```

Suivez avec "step into" pour voir la comparaison du numéro consigné dans r8 avec des valeurs statiques. On doit faire le saut à vulnerable+11ec

```
kd> t
vulnerable+0x10d7:
fffff800`7a8510d7 4181e808282200 sub      r8d,222808h
kd> t
vulnerable+0x10de:
fffff800`7a8510de 0f8408010000 je       vulnerable+0x11ec (fffff800`7a8511ec)
```

D'abord, à 11ef, on voit les 8 premiers octets copiés depuis notre patron vers le registre rcx.

Ce registre correspond au premier argument transmis à la fonction mmapiospace(). Par conséquent, il correspond à l'adresse physique en mémoire que l'on pourra accéder car on contrôle son contenu

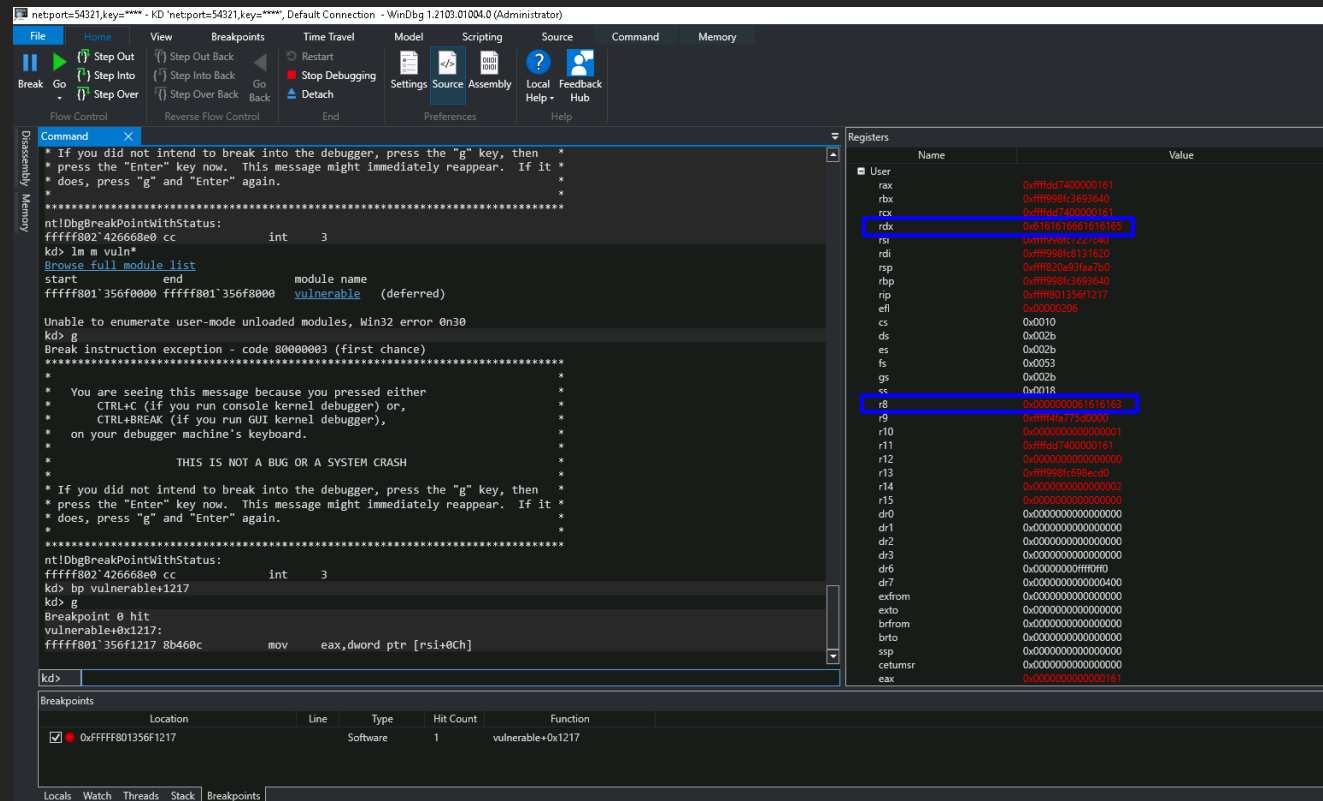
Effectivement, on a:

```
rcx = 0x6161616261616161
```

On saute l'appel à mmapiospace() avec la fonction "step over".

Ensuite, à 120a, on voit 8 octets de notre patron copié depuis rsi + 0x10 dans rdx

```
ffffff801`356f120a 488b5610      mov      rdx,qword ptr [rsi+10h]
```

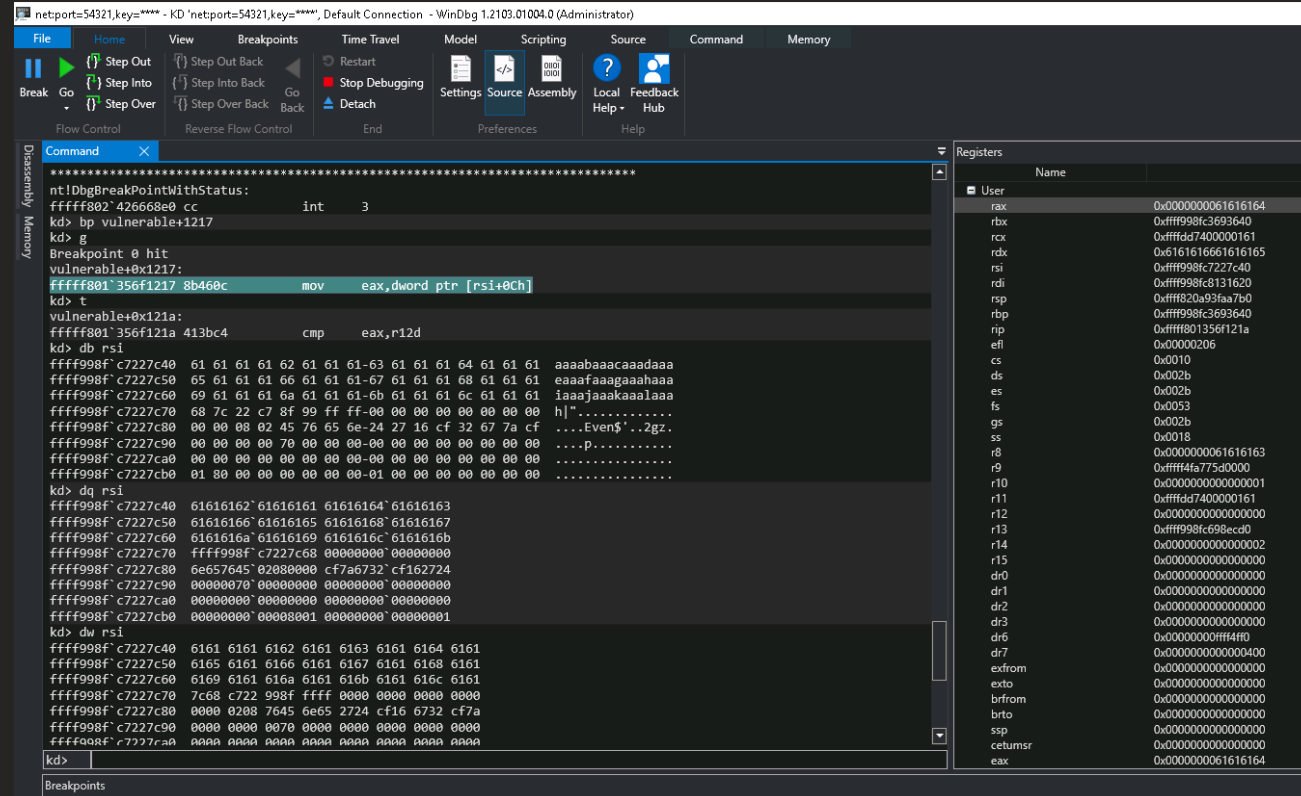


En regardant le code dans ghidra, on peut voir que cette variable sera utilisée comme tampon de destination. Il serait donc utile de transmettre un pointeur qui aboutira dans notre programme en userspace.

```
rdx = 0x6161616661616165
```

Ensuite, a 1217, on voit 4 octets (double word) être copié dans rax

```
ffffff801`356f1217 8b460c      mov     eax,dword ptr [rsi+0Ch]
```



The screenshot shows the WinDbg interface with the following details:

- Command Window:** Displays the command `nt!DbgBreakPointWithStatus:` and the assembly instruction `ffffff801`356f1217 8b460c mov eax,dword ptr [rsi+0Ch]`, which is highlighted in green.
- Registers Window:** Shows the state of various registers. The `rax` register is highlighted, showing the value `0x0000000061616164`.
- Disassembly Window:** Shows the assembly code for the current instruction, including the `mov` instruction and subsequent instructions like `cmp` and `db`.

Le registre rax prend donc une valeur influencée par notre patron

```
rax = 0x00000000061616164
```

Cette valeur doit être égale à 0, 1 ou 2 selon si on veut lire des octets (char), des mots de 2 octets (word) ou 4 octets (dword).

Enfin, à 120e, le registre r8 est initialisé avec le nombre d'itérations souhaité pour la copie. Avec le patron, le registre r8 prend la valeur suivante

```
r8 = 0x00000000061616163
```

## Récapitulons Registres contenant des parties du patron cyclique produit par pwntools



Avant mmapiospace()

rcx = 0x6161616261616161



Après mmapiospace()

rax (mode de copie) = 0x0000000061616164

rdx (tampon de destination) = 0x6161616661616165

r8 (nombre d'itération) = 0x0000000061616163

Avant de recherche dans le patron, on doit `unhexlify()` et inverser (car little-endian) les fragments du patron cyclique.

```
rcx = binascii.unhexlify("6161616261616161")[::-1] = b'aaaabaaa'
```

```
rax = binascii.unhexlify("61616164")[::-1] = b'daaa'
```

```
rdx = binascii.unhexlify("6161616661616165")[::-1] = b'aaaafaaa'
```

```
r8 = binascii.unhexlify("61616163")[::-1] = b'caaa'
```



Bien sûr, `aaaabaaa` correspond au débût de notre patron, donc rcx avant mmapiospace() en position 0. Donc, l'adresse physique qui sera lue se trouve en position 0

Avec la fonction find() du générateur, on obtient

```
In [9]: a.find(b'daaa')
```

```
Out[9]: (12, 0, 12)
```

donc rax, mode de copie en position 12

```
In [13]: a.find(b'eaaa')
```

```
Out[13]: (16, 0, 16)
```

rdx, tampon de sortie en position 16

Et enfin,

r8, nombre d'itération en position 8

Les 8 premiers octets sont passés à mmMapIoSpace comme pointeur sur la mémoire physique à récupérer. Le deuxième argument passé à mmMapIoSpace correspond à la taille de la mémoire à récupérer. La même valeur est donc aussi utilisée pour le nombre d'itérations dans la boucle (registre r8).

La structure à envoyer dans bufin pour une lecture aura donc l'aspect suivant:

```
type READMSG {  
    address uint64  
    size, uint32  
    mode, uint32  
    buffer, *byte  
}
```

La taille totale de la structure est donc de 24 octets et non 48

# Exploitation

## Exercice avec la preuve de concept

Voyez la preuve de concept dans `go-kernel-exploit`

- Contournement du ASLR à partir de `NtQuerySystemInformation()` appelé depuis le userspace
- Accès à la mémoire physique
- Manque traduction de mémoire virtuelle vers mémoire physique. (rammap le fait)