

Introduction au développement de Solution de Triche

Sommaire :

- ***Présentation du Projet***
- ***Installation de la VM***
- ***Quelques Notions Importantes***
- ***Review des Tools***
- ***Un Cheat No-Code***
- ***Premier pas en Reverse Engineering***
- ***Comprendre et faire du Reverse Engineering***
- ***Pointer Scan et Pattern Scan***
- ***Premier Cheat Code***
- ***Les Anti-cheats***
- ***Conclusion***

1. Présentation du Projet :

1.0. Pourquoi ?

Dans le cadre de nos études, nous devons rendre des projets sur un thème libre, et dans l'espoir de nous améliorer en développement, en reverse engineering et comprendre les fonctionnements bas niveau dans un environnement ludique, nous avons décidé de reverser plusieurs jeux-vidéos et de sortir nos notes sous forme de livre gratuit et accessible dans le but de donner envie à de jeunes étudiants / développeurs / néophytes de se lancer dans la création de cheat, de plus les ressources françaises sont peu nombreuses.

Nous basons notre projet en tant que développement de cheat sur des jeux sans logiciels anti-cheat, ceci est le premier rapport écrit, d'autres apparaîtront sûrement pour voir certains sujets en profondeur. Soyez indulgents, nous n'avons eu que peu de temps pour travailler sur le sujet et nous n'avons que des bases, nous partons comme vous, et nous allons progresser ensemble de manière ludique.

1.1. Notre équipe !

Nous sommes une équipe basée en France, tous issus de la même école.

4dorable

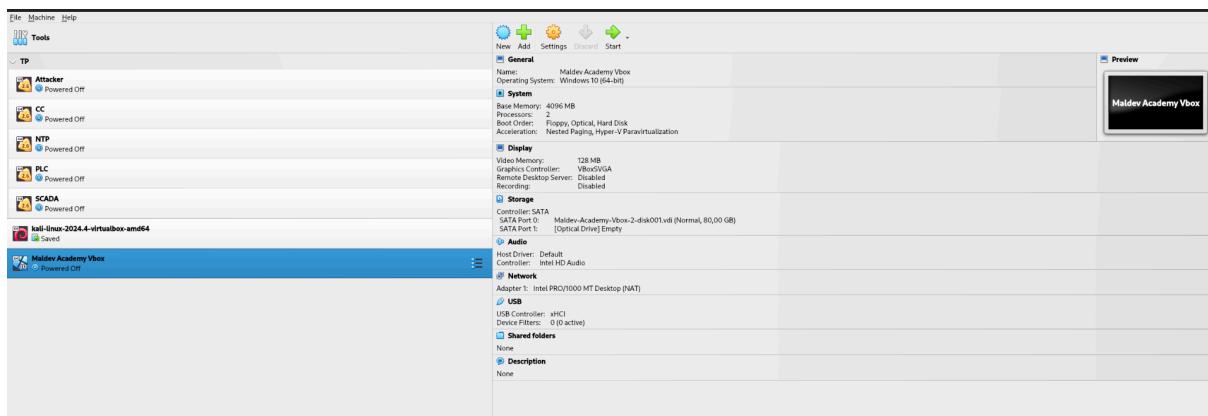
Nous avons tous un background technique différent mais on se rejoint sur le développement C et C++ que nous maîtrisons tous à un niveau au moins intermédiaire.

2. Installation de la VM :

2.0. Virtual Box

Dans le cadre de notre livre, nous allons utiliser une machine virtuelle (VM). Un logiciel de supervision nous sera utile, dans notre cas nous utiliserons VirtualBox par pure habitude et gratuité.

Vous pouvez le télécharger [ici](#).



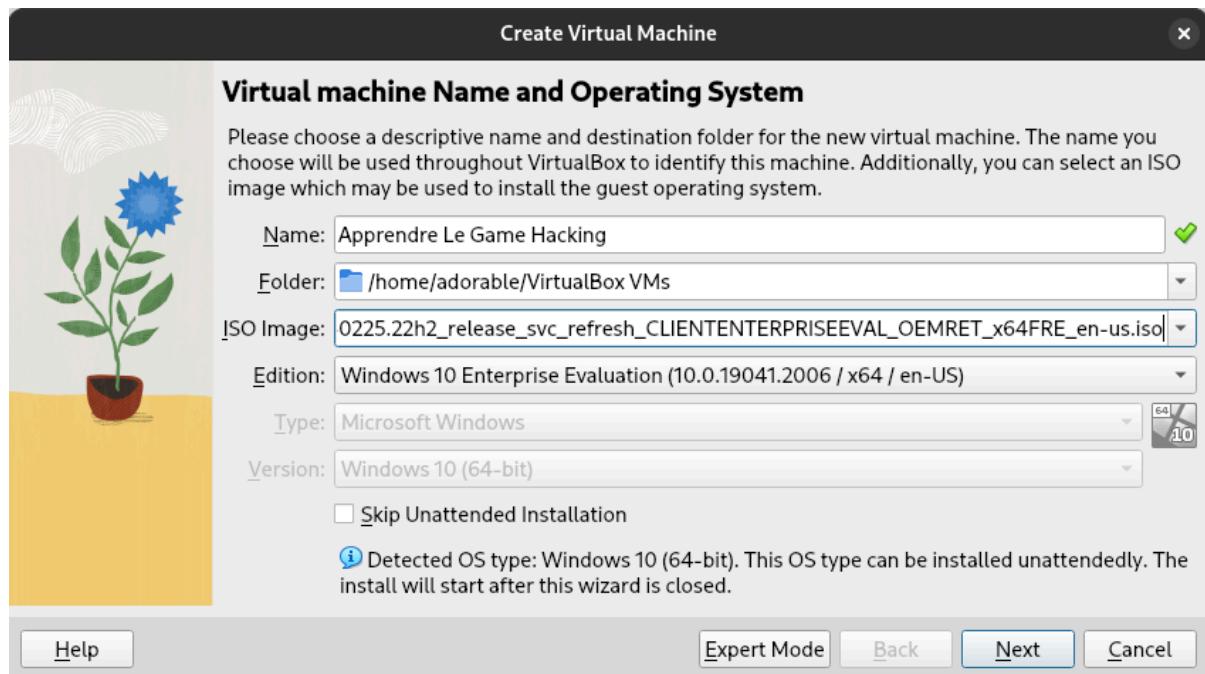
Voilà à quoi cela ressemble, votre interface est sûrement vide, nous allons pallier à cela de suite !

2.1. Windows...mais virtuel

Afin de simplifier, nous allons nous concentrer sur le système d'exploitation Windows car il détient la plus grosse part de marché en ce qui concerne les ordinateurs personnels.

Une fois notre hyperviseur installé, nous pouvons y mettre notre image ISO Windows (10 ou 11 on s'en fout), et commencer à installer certains de nos outils.

Vous pouvez télécharger votre ISO sur ce [site](#) (qui est celui de Windows donc aucun risque), et vous pouvez ensuite cliquer sur 'New' dans VirtualBox, attribuer un nom pour votre machine, puis sélectionner le fichier ISO.



Suivez les instructions et attendez que Windows s'installe. Vous aurez un environnement Windows, dans votre environnement actuel !

2.2. Installation de nos outils

On va utiliser un script afin de nous télécharger des outils de Game Hacking. Allez sur Powershell puis copiez-collez ces commandes.

```
Set-ExecutionPolicy Unrestricted
```

```
Install-BoxstarterPackage -PackageName  
https://raw.githubusercontent.com/GameHackingAcademy/vmsetup/master/vmsetup.txt  
-DisableReboots
```

NB: Il pourrait y avoir une erreur de permissions, dans ce cas, exédez cette commande puis reprenez les autres commandes.

```
Set-ExecutionPolicy Unrestricted
```

Ces commandes nous permettent d'installer un package de base qui nous permettra de créer nos premiers cheats !

Vous trouverez l'explication des différents outils au chapitre dédié.

2.3. Installation de nos outils

Pour pouvoir s'amuser à comprendre et confectionner des cheats, il faut aussi des jeux dont l'anti-cheat n'est pas présent. Je vous laisse avec ma recommandation de jeux (ceci seront utilisés pour la suite de ce livre mais les techniques seront les mêmes, je vous invite à regarder de vous-mêmes)

Copier coller sur l'invite de commande Powershell

```
choco install wesnoth --version=1.14.9 -y
```

Lien pour télécharger Assault Cube

[Assault Cube](#)

3. Quelques notions importantes :

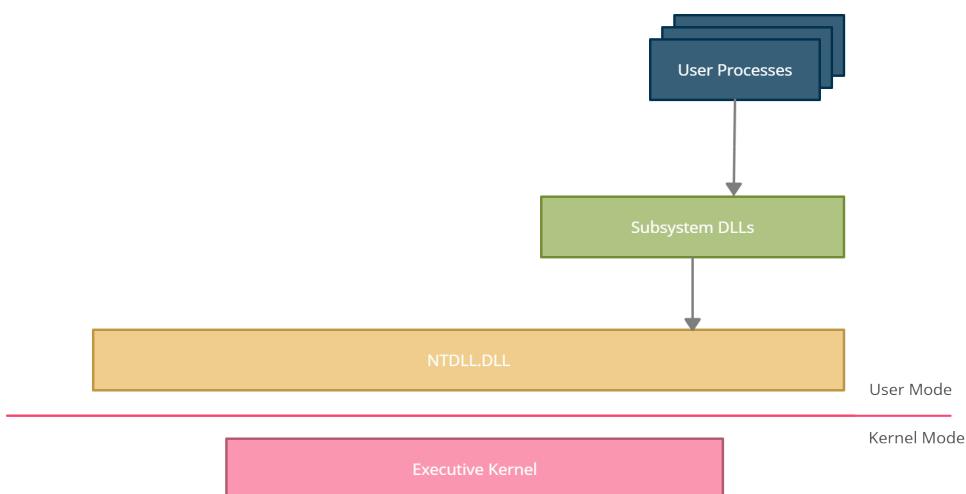
Je pense qu'il est important de rappeler certaines bases de Windows, tout d'abord car elles sont intéressantes mais aussi parce que le milieu du cheat et anti-cheat est majoritairement basé sur le système Windows.

3.0.1 L'architecture Windows :

Le processeur d'une machine exécutant le système d'exploitation Windows peut fonctionner sous deux modes différents : le mode utilisateur et le mode noyau.

Les applications s'exécutent en mode utilisateur et les composants du système d'exploitation s'exécutent en mode noyau, prenons l'exemple de lorsqu'une application souhaite accomplir une tâche, comme créer un fichier ne peut pas être créé seul. Et la seule entité capable d'accomplir cette tâche est le noyau. Les applications doivent donc suivre un flux d'appel de fonction spécifique.

Comme une image vaut 1000 mots, voici un schéma récupéré sur Internet :

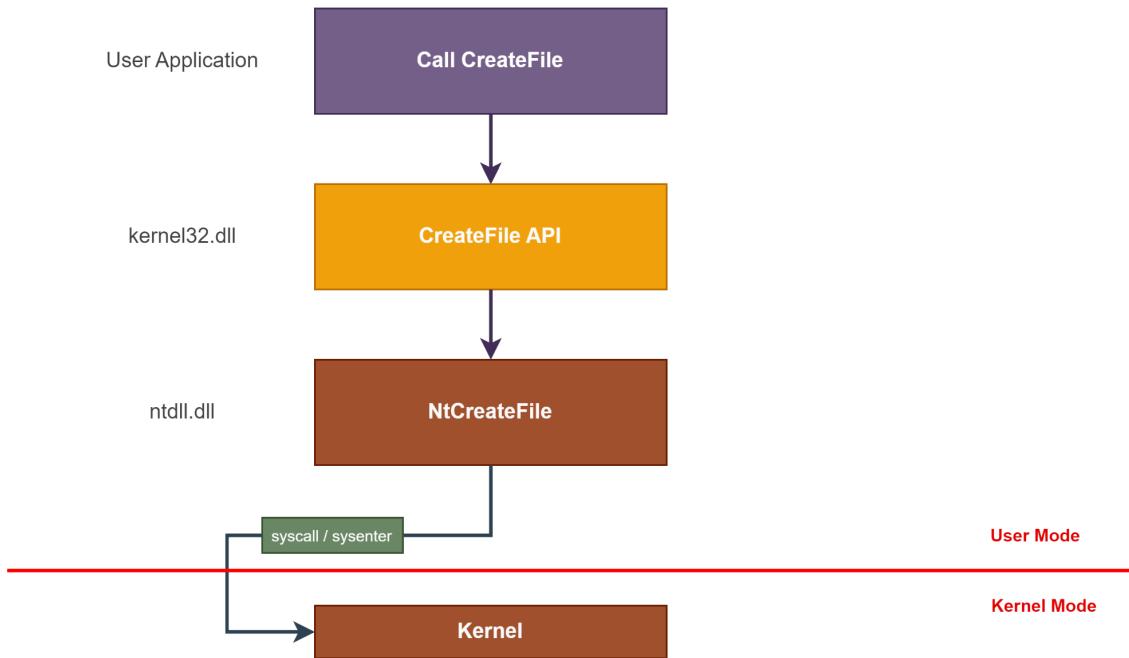


1. Processus utilisateur - Un programme/une application exécuté par l'utilisateur tel que le Bloc-notes, Google Chrome ou Microsoft Word.

2. DLL de sous-système : DLL contenant des fonctions API appelées par les processus utilisateurs. Par exemple, kernel32.dll exporte la fonction CreateFile de l'API Windows (WinAPI). D'autres exemples de DLL de sous-système courantes sont ntdll.dll, advapi32.dll et user32.dll.
3. Ntdll.dll - Une DLL à l'échelle du système qui est la couche la plus basse disponible en mode utilisateur. Il s'agit d'une DLL spéciale qui crée la transition du mode utilisateur au mode noyau. Elle est souvent appelée API native ou NTAPI.
4. Noyau exécutif - Il s'agit du noyau Windows qui appelle d'autres pilotes et modules disponibles en mode noyau pour effectuer des tâches. Le noyau Windows est partiellement stocké dans un fichier appelé ntoskrnl.exe sous « C:\Windows\System32 ».

3.0.2. Flux d'appel de fonction :

L'image ci-dessous montre un exemple d'application qui crée un fichier. Elle commence par l'appel de la fonction WinAPI CreateFile par l'application utilisateur, disponible dans kernel32.dll. Kernel32.dll est une DLL critique qui expose les applications à WinAPI et peut donc être vue chargée par la plupart des applications. Ensuite, CreateFile appelle sa fonction NTAPI équivalente, NtCreateFile, qui est fournie via ntdll.dll. Ntdll.dll exécute ensuite une instruction sysenter (x86) ou syscall (x64) d'assemblage, qui transfère l'exécution en mode noyau. La fonction NtCreateFile du noyau est ensuite utilisée, qui appelle les pilotes et modules du noyau pour effectuer la tâche demandée.



3.0.3. Appeler directement la NTAPI :

Il est important de noter que les applications peuvent invoquer des appels système (c'est-à-dire des fonctions NTDLL) directement sans avoir à passer par l'API Windows. L'API Windows agit simplement comme un wrapper pour l'API native. Cela étant dit, l'API native est plus difficile à utiliser car elle n'est pas officiellement documentée par Microsoft. De plus, Microsoft déconseille l'utilisation des fonctions de l'API native car elles peuvent être modifiées à tout moment sans avertissement.

3.1.1 Les Bases du Développement :

Pour cette section, je ne peux vous donner que des références pour apprendre la programmation. Il ne s'agit pas d'être un expert de la programmation, mais vous serez à même de lire du code en C++ et de l'assembleur x86 donc comprendre les instructions et pouvoir faire quelques projets en C++ est un grand avantage.

Voici quelques références qui pourraient être utiles:

- C++
 - Assembleur

3.2.1 Les Bases du Jeu Vidéo :

Les jeux vidéos ne sont que des applications comme les autres, ils ont plus ou moins complexes et sont composés de plusieurs parties comme:

- Les graphismes : tout ce qu'on voit à l'écran.
- Le son : les musiques et effets sonores.
- Les contrôles : clavier, manette, souris, tout ça.
- La physique : genre la gravité, les collisions.
- La logique du jeu : les règles, ce qui se passe quand on appuie sur un bouton.

Vu que chaque partie est un vrai casse-tête à gérer, la plupart des jeux utilisent des fonctions externes qu'on regroupe dans ce qu'on appelle une bibliothèque. Ces bibliothèques servent à éviter de tout coder soi-même et de devoir "réinventer la roue". Par exemple, pour afficher des images ou des formes à l'écran, on utilise souvent des bibliothèques comme OpenGL.

Alors, pourquoi je te parle de ça ? Parce que pour certains types de cheats, il s'agit de savoir quelles bibliothèques externes le jeu utilise. Un wallhack interne au jeu, par exemple : c'est un hack qui permet de voir les joueurs à travers les murs. Pour créer ce type de hack, on peut modifier la bibliothèque graphique du jeu.

Cela dit, pour la plupart des hacks, on va plutôt jouer avec la logique du jeu. En gros, c'est ce code qui gère comment le jeu fonctionne.

Exemple: la logique décide de la hauteur d'un saut ou de combien de fric le joueur ramasse. En modifiant ce bout de code, tu pourrais sauter plus haut que les développeurs l'avaient permis ou devenir plus riche.

3.3. Les différents types de cheats:

Prenons un peu de temps pour définir quelques termes, et catégoriser. En naviguant sur les forums, on tombe très vite sur un terme technique, à savoir les "trainers". Un trainer désigne tout programme qui a pour but de modifier la mémoire d'un jeu vidéo et donc son comportement. Un trainer est capable de "freeze" une valeur en mémoire en bouclant

dessus afin de reset la valeur en permanence. Un trainer a pour but de simplifier la triche et d'être accessible aux personnes non-techniques.

Cheat Engine permet la création de collections de cheats, qu'on appelle "cheat tables". En plus de pouvoir modifier ou freeze des valeurs en mémoire, il est également capable d'exécuter des scripts écrits en Lua (un langage de script que nous n'étudierons pas pour l'instant) en appuyant sur une touche, le tout étant configurable. Une fois que vous avez une panoplie de cheats qui fonctionne, vous pouvez les partager avec vos amis en exportant cette cheat table. Ce sera un trainer chargeable par vos amis à condition qu'ils aient installé Cheat Engine: ils pourront freeze les valeurs, les modifier, et configurer le cheat table à leur guise.

Maintenant que nous avons clarifié ce point, il nous faut plonger dans les différentes catégories de cheats, afin d'avoir une belle vue d'ensemble.

On regroupe les cheats selon l'endroit où ils s'exécutent: dans la mémoire du jeu, dans un processus à part, au-dessus d'une VM, dans le noyau du système ou encore sur un ordinateur à part via DMA (Direct Memory Access).

3.3.1. Les Cheats Internes

Ces cheats exploitent les DLL, des bibliothèques devant être chargées dans la mémoire du jeu pour fonctionner. Un cheat peut utiliser...

- Des entrées de registre Windows.
- Des "code cave" c'est-à-dire un ensemble d'octets non utilisés au sein de la mémoire d'un processus.
- Une "remote thread", c'est-à-dire un fil d'exécution pouvant accéder à l'espace mémoire du processus ciblé.
- Et autres.

... afin de charger une DLL contenant le code du cheat. Une fois injectée, la DLL s'exécute au sein même de l'espace mémoire du jeu et peut accéder directement aux données du jeu.

Avantages:

- De meilleures performances, car le cheat n'a pas besoin de passer par des appels systèmes pour modifier la mémoire.

Inconvénients:

- Les cheats internes peuvent faire planter le processus cible.
- Ce sont les cheats les plus simples à détecter, par le biais de différentes techniques que nous présenterons plus tard.

3.3.2. Les Cheats Externes

Ce sont des programmes autonomes que l'on peut exécuter comme n'importe quelle application normale. Ils exploitent des fonctions Windows, notamment ReadProcessMemory et WriteProcessMemory, pour lire et modifier la mémoire d'un autre programme en cours d'exécution (comme un jeu, par exemple).

Avantages:

- Plus difficiles à détecter.
- Plus faciles à développer.

Inconvénients:

- Moins performants, car ils doivent passer par l'API Windows chaque fois qu'ils veulent lire ou écrire dans la mémoire.

3.3.3. Les Cheats Hyperviseurs

Il s'agit là d'exécuter le jeu dans une machine virtuelle, mais de lancer le cheat depuis la machine hôte.

Avantages:

- Presque impossibles à détecter.

Inconvénients:

- Demandent des ressources pour une machine virtuelle.
- Communiquer à travers la machine virtuelle est très complexe.
- Certains anti-cheats ne permettront pas la connexion au multijoueur s'ils détectent un hyperviseur. (Ainsi Faceit AntiCheat)

ne permet pas le lancement de Counter-Strike si Windows Subsystem For Linux est actif, par le biais de la détection d'Hyper-V, la technologie de virtualisation de Windows).

3.3.4. Les Cheats Kernel

Ces cheats exécutent tout ou partie de leur code dans le noyau du système d'exploitation.

Avantages:

- Invisibles aux yeux des anti-cheats s'exécutant hors du noyau.

Inconvénients:

- Complexes à développer.
- Les modules kernel non signés sont en général rejetés par le système à moins que certaines options de sécurité Windows soient désactivées. Or, si ces options sont désactivées, alors un anti-cheat peut empêcher le lancement du jeu.

3.3.5. Les Cheats DMA

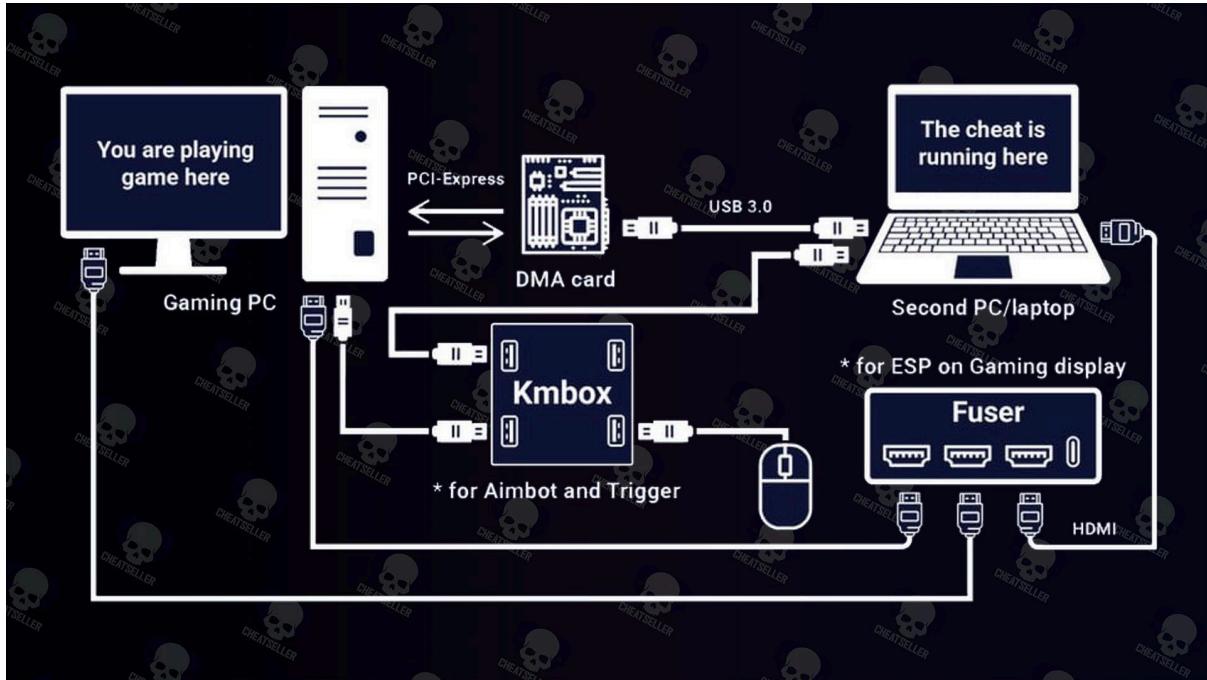
Ces cheats utilisent des cartes DMA (Direct Memory Access), qui peuvent être attachées aux barrettes de RAM de l'ordinateur hébergeant le jeu. La carte DMA lit la RAM et envoie son contenu à un autre ordinateur, qui lui gère la logique du cheat, et peut commander à la carte DMA d'écrire dans la RAM de l'ordinateur contenant le jeu et ce sans passer par le processeur de l'ordinateur cible.

Avantages:

- Complètement invisibles aux yeux de la machine exécutant le jeu.

Inconvénients:

- L'équipement nécessaire, une carte DMA et un ordinateur en plus, rendent l'opération assez coûteuse.



Un exemple de setup de cheat DMA. Le KMbox est un périphérique permettant de contrôler le clavier et la souris, et le fuser synthétise toutes les entrées et sorties sur un seul câble USB.

(Source de l'image: cheatseller.com, une marketplace de cheats pour de nombreux jeux populaires).

4. Review des Tools :

4.0. Introduction à la Boîte à Outils

Dans cette section, nous allons explorer les outils essentiels pour tout aspirant développeur de cheats ou reverse engineer. Ces outils sont nos armes, nos pinceaux, et nos tournevis : sans eux, impossible de sculpter ou de réparer quoi que ce soit. L'objectif est de vous donner un aperçu clair, mais aussi quelques conseils pour maximiser leur utilisation.

4.1. Cheat Engine : L'indispensable

Cheat Engine est l'outil de base pour toute manipulation mémoire. Simple d'utilisation et très puissant, il vous permet de scanner, modifier, et manipuler les processus en temps réel.

Pourquoi Cheat Engine est si populaire ?

- Interface intuitive pour les débutants.
- Outil très documenté avec une communauté active.
- Possibilité de créer des scripts Lua pour automatiser les cheats.

Les étapes de base :

1. **Attacher un processus** : Sélectionnez l'application cible (ex. : un jeu vidéo en cours d'exécution).
2. **Scanner une valeur** : Cherchez par exemple votre nombre de munitions dans le jeu.
3. **Modifier les résultats** : Une fois la bonne adresse trouvée, ajustez-la (par exemple, passez vos munitions de 10 à 999).

Options avancées :

- **Debugger intégré** : Permet de surveiller et manipuler le flux d'exécution.
- **Pointeurs et structures** : Identifier des valeurs dynamiques qui changent d'adresse à chaque exécution du jeu.

- **Création de cheats automatisés** : Sauvegarder des scripts pour un usage futur.

4.2. x64 GDB : Le Débogueur des Pros

Le GNU Debugger (GDB) est un outil pour les développeurs et reverse engineers confirmés. Il permet d'interagir avec les programmes à un niveau plus profond. Idéal pour suivre l'exécution d'un programme et modifier ces instructions en live.

Ce que vous pouvez faire avec :

- Placer des breakpoints pour arrêter le programme à un endroit précis.
- Inspecter la mémoire et les registres.
- Modifier les instructions ou données directement en mémoire.

Quelques commandes incontournables

- `b [fonction/ligne]` : Pose un breakpoint (point d'arrêt) sur une fonction ou une ligne précise.
- `run` : Lance l'exécution du programme jusqu'au breakpoint.
- `info registers` : Affiche l'état des registres du processeur (utile pour traquer vos variables).
- `x/[n] [adresse]` : Explore la mémoire à une adresse donnée.

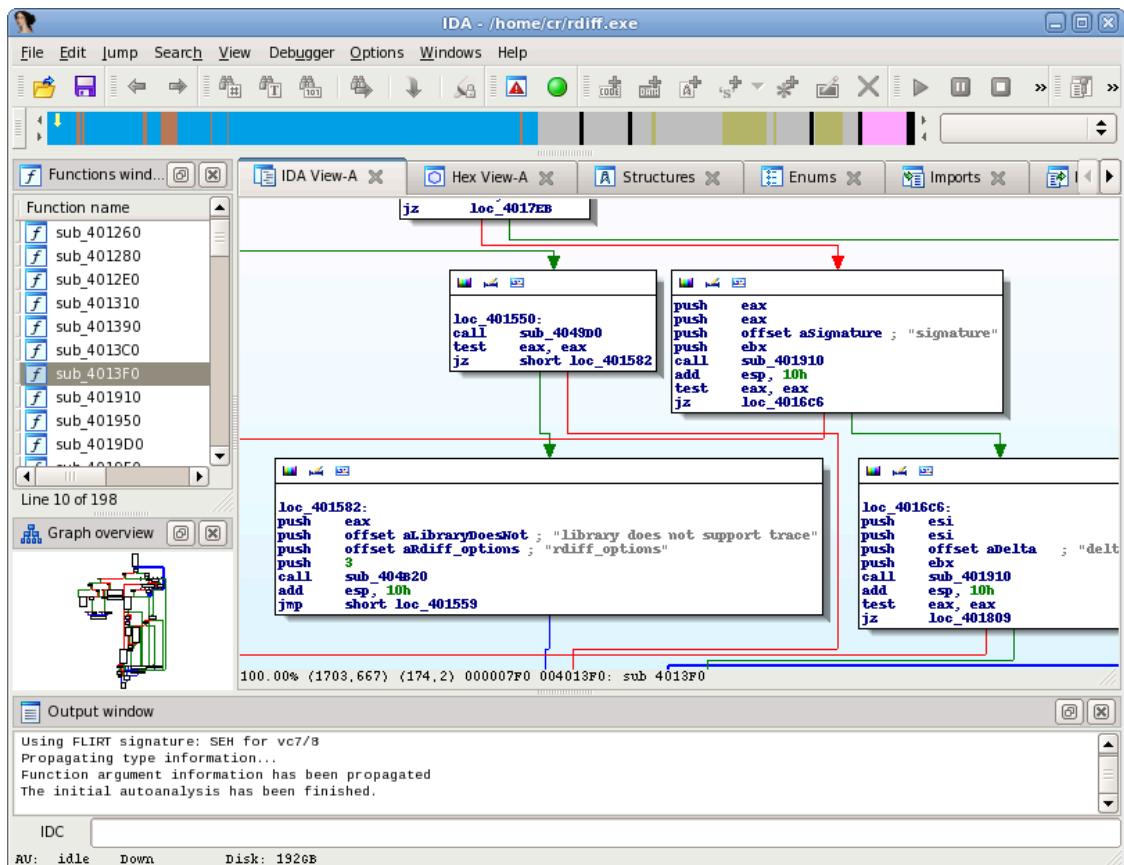
Petit hack : Coupler GDB avec un plugin comme gef ou peda améliore drastiquement l'expérience, avec des visualisations claires des piles, registres et instructions.

4.3. Désassembleurs : IDA Pro et Ghidra

Les désassembleurs sont la clé pour comprendre un programme sans son code source. Ces outils permettent de traduire les binaires en pseudo-code et de tracer les comportements.

IDA Pro

IDA est l'outil préféré des professionnels (et des hackers en herbe ayant trouvé une licence...). Il transforme des binaires obscurs en un pseudo-code lisible, mettant en lumière les fonctions et les variables.



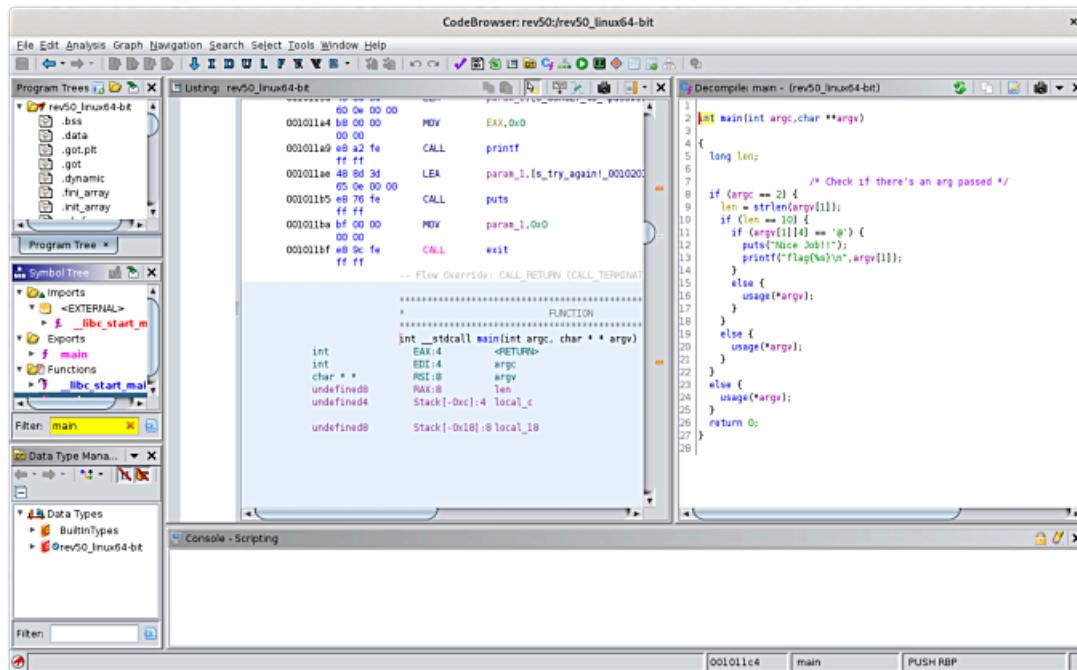
Points forts

- Interface interactive et graphique.
- Support pour de multiples architectures (ARM, x86, x64, etc.).
- Analyse avancée des appels de fonctions et des structures.

Limitation : IDA Pro coûte une petite fortune, mais une version gratuite (limitée) existe.

Ghidra

Développé par la NSA, Ghidra est une alternative gratuite et open source. Si IDA est l'arme du pro, Ghidra est l'arme du passionné ou de l'apprenti.



Points forts

- Décompilation puissante, générant du pseudo-code C.
- Support collaboratif pour travailler à plusieurs sur un même projet.
- Gratuité et extensibilité via des scripts.

4.4. REClass : cartographier la mémoire

Quand vous travaillez avec des jeux, vous aurez souvent besoin de comprendre comment les objets (joueurs, ennemis, items) sont représentés en mémoire. REClass est votre meilleur allié pour ça.

Fonctionnalités principales

- Analyse dynamique des structures mémoire.
- Construction visuelle des classes, permettant de repérer leurs champs et offsets.

- Export facile pour des projets externes.

Exemple d'utilisation

1. Attachez REClass à un jeu comme Assault Cube.
2. Naviguez dans les structures mémoire, identifiez la classe "Player" et notez les offsets des variables importantes (santé, position, etc.).
3. Exportez ces informations pour les utiliser dans un script.

4.5. Outils complémentaires

OllyDbg

Un débogueur pratique pour les applications 32 bits. Moins complexe que GDB, il est parfait pour débuter.

Radare2

Un framework puissant pour le reverse engineering, mais avec une courbe d'apprentissage abrupte.

Process Hacker

Outil permettant de surveiller les processus, leurs threads, et leurs interactions avec le système d'exploitation.

5. Un Cheat 'No-Code' :

5.0. Lancement du programme cible :

Pour ce chapitre, nous allons voir comment modifier la valeur liée à l'adresse mémoire d'un programme afin d'avoir un avantage sur les autres joueurs.

Et cela, sans écrire une seule ligne de code !

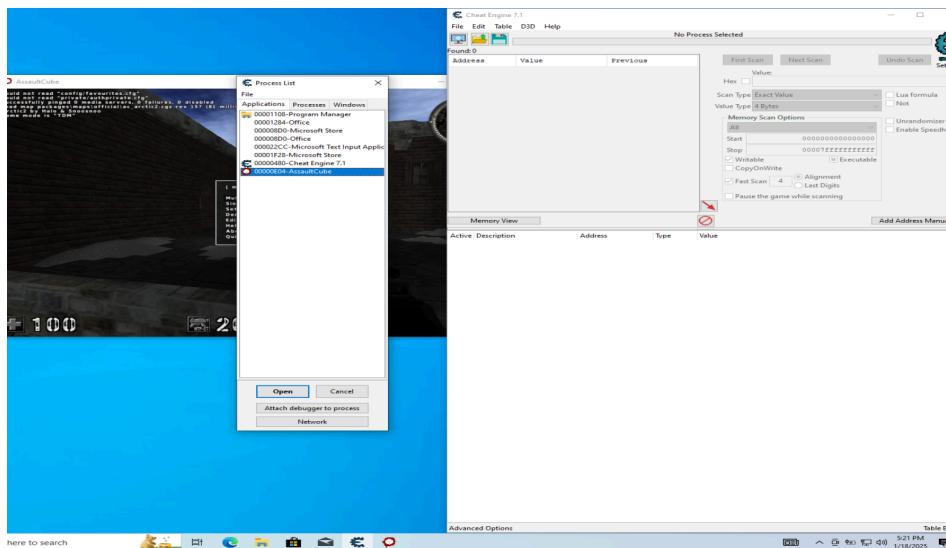
Commencez par installer le jeu cible :

<https://assault.cubers.net/download.html>

AssaultCube est un jeu Open Source et sans anticheat, aucun contournement d'Anti cheat. Nous pourrons alors modifier la mémoire comme nous le voulons.

5.1. Lancement de Cheat Engine :

Vous pouvez tout simplement lancer Assault Cube & Cheat Engine en mode Administrateur puis attacher le programme cible :



Appuyez sur Open et vous vous retrouverez prêt à modifier les valeurs du jeu.

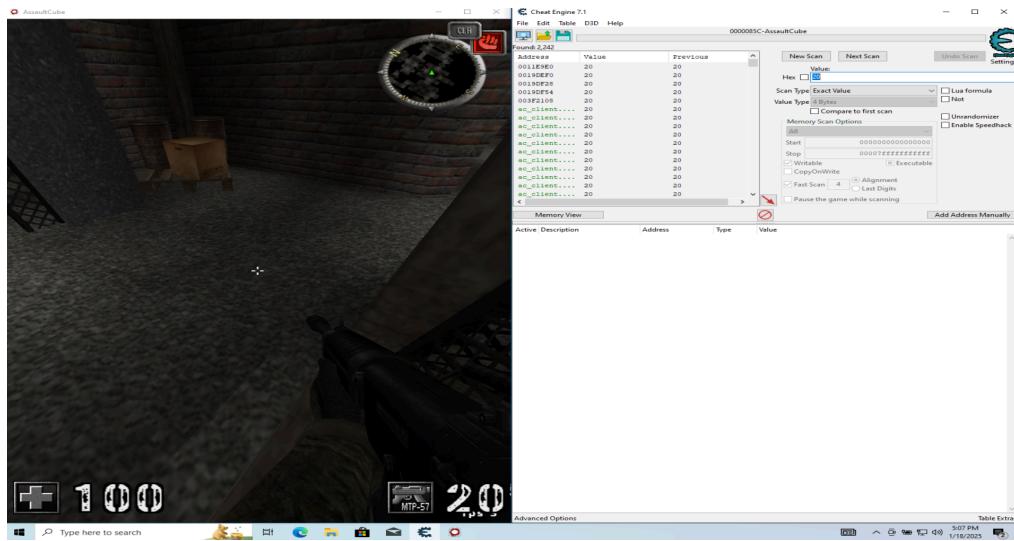
5.2. Modification du nombre de munitions avec Cheat Engine :

Lancez Cheat Engine et le jeu Assault Cube. Vous pouvez voir le nombre de munitions, le nombre de points de vie etc...

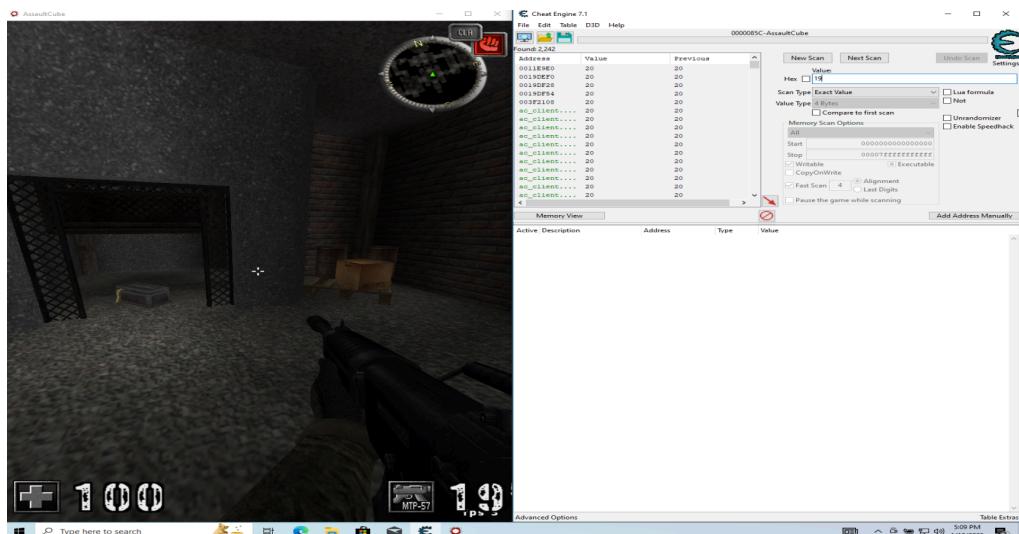
Ces valeurs sont inscrites dans la mémoire vive du jeu, avec Cheat Engine, nous pouvons accéder à cette valeur et la modifier. Voyons comment cela peut se faire.

Tout d'abord, trouvons une valeur que nous souhaitons modifier, dans notre cas, nous nous focaliserons sur le nombre de munitions, qui, lorsque l'on lance le jeu est de 20.

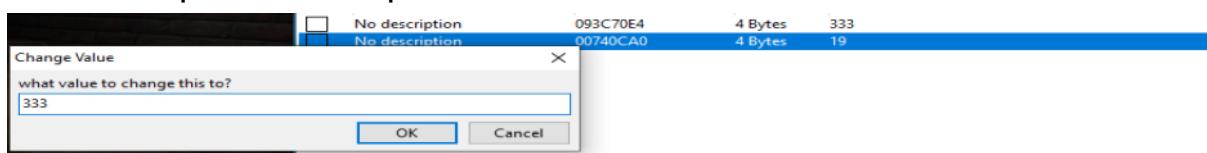
Cherchons cette valeur :



Plusieurs valeurs correspondent à la valeur de 20 dans notre programme, allons modifier la valeur en tirant une nouvelle fois et cliquons sur next scan pour voir les adresses impactées par notre modifications.

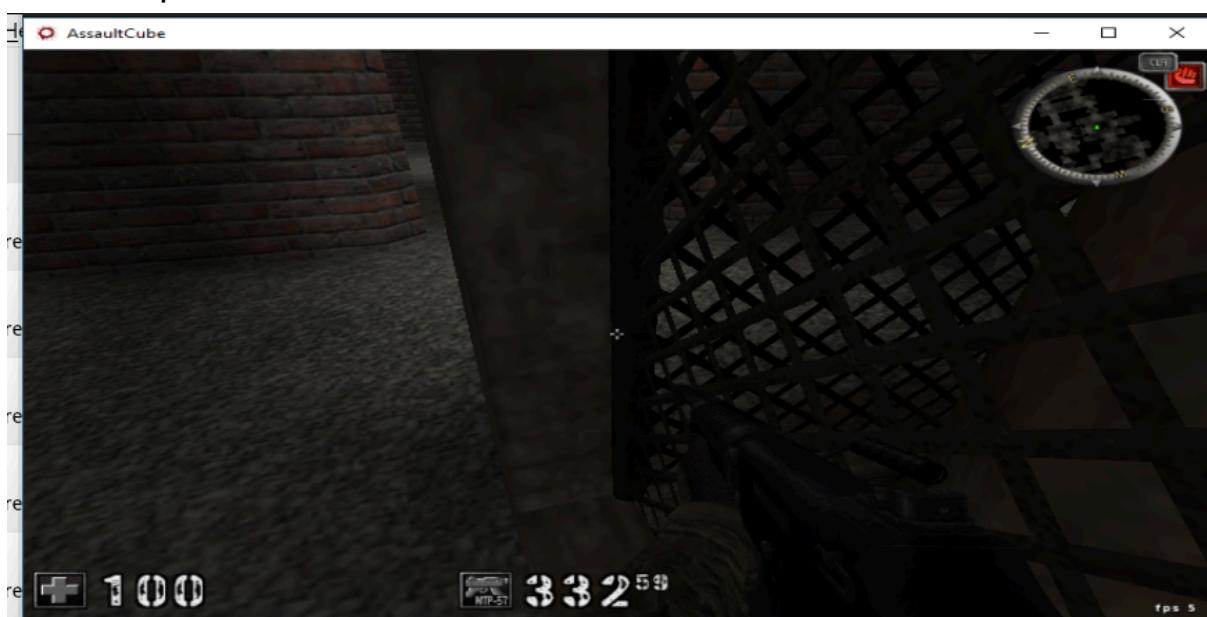


Cela nous laisse avec 2 valeurs restantes à deux adresses différentes.
Double cliquez dessus puis modifiez ces valeurs.



L'une des valeurs va modifier le nombre de munitions affichées à l'écran et l'autre valeur n'influe en rien.

Voilà après modification à quoi cela ressemble les modifications faites, n'hésitez pas à tester :



5.3. Comment et pourquoi cela marche ?

Les valeurs des munitions sont stockées à une adresse mémoire. En réalité, chaque valeur est stockée dans la mémoire vive (RAM).

C'est-à-dire que la valeur de notre nombre de munitions pourrait être stockée à une adresse 0x1337.

Cheat Engine permettant de scanner la mémoire d'un processus, notre but est de localiser puis de changer la valeur enregistrée à l'adresse mémoire qui nous intéresse. On indique alors la valeur connue du nombre de munitions, puis on effectue un premier scan, on modifie la valeur de notre nombre de munitions (en tirant) et on effectue un nouveau scan pour faire apparaître les adresses mémoires qui ont changé.

Cela marche grâce au fait que le jeu ne protège pas leurs données en mémoire (ce qui est souvent le cas pour les jeux solo) et qu'il n'y a pas d'Anti Cheat.

Fun fact: Il m'est arrivé de me faire ban par Riot Games car je faisais des challenge de reverse et que j'ai lancé un jeu sans fermer les applications de reverse. Leur Anti Cheat étant très invasif, ils ont détecté mon logiciel de reverse comme un outil externe de cheat.

Nous parlerons des Anti Cheats modernes à l'avenir dans un but purement informel et non-technique.

6. Premiers Pas en Reverse Engineering:

6.0. Petite Introduction au Reverse Engineering:

Le Reverse Engineering, ou rétro-ingénierie en français, est une discipline fascinante. Pour faire simple, cela consiste à analyser un système existant pour comprendre comment il fonctionne, souvent sans avoir accès à son code source ou à ses spécifications originales. L'objectif est d'en apprendre davantage sur sa structure interne, son comportement et ses mécanismes...de trouver des failles ou des points d'ancre pour confectionner des cheats.

Un petit tips intéressant, dès que je programme quelque chose, je m'amuse à regarder dans un désassembleur à quoi ressemble mon code, cela permet de développer quelques mécanismes en Reverse et c'est très intéressant.

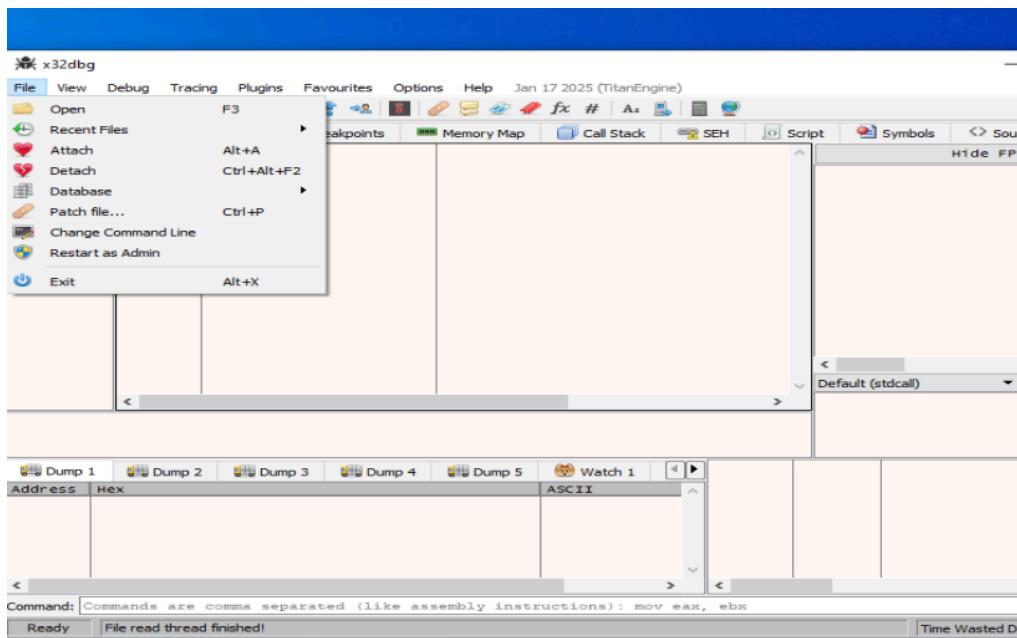
6.0. Demonstration sur Assault Cube:

Je pense qu'il est temps de se plonger dans un exercice plutôt simple de Reverse en modifiant le code d'Assault Cube directement.

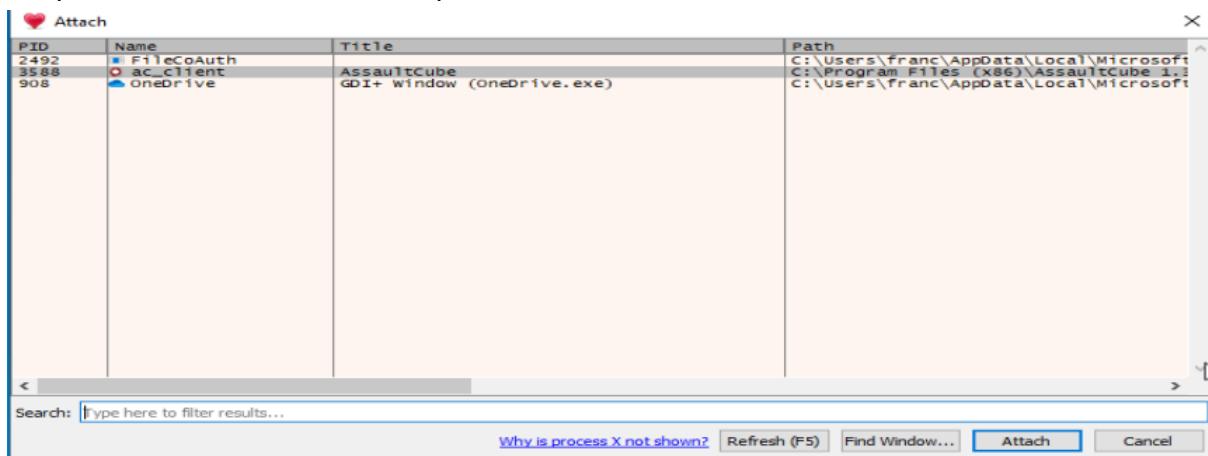
Notre première étape est de lancer x 32 gdb (Assault Cube est en 32 bits donc il ne sera pas visible par x64 gdb).

Le seul prérequis pour cette section est d'avoir l'adresse mémoire qui stocke notre nombre de munitions, si vous ne l'avez pas, retour au chapitre précédent, cela vous entraînera.

1. Une fois le programme lancé il faut attacher le processus Assault Cube à notre débogueur

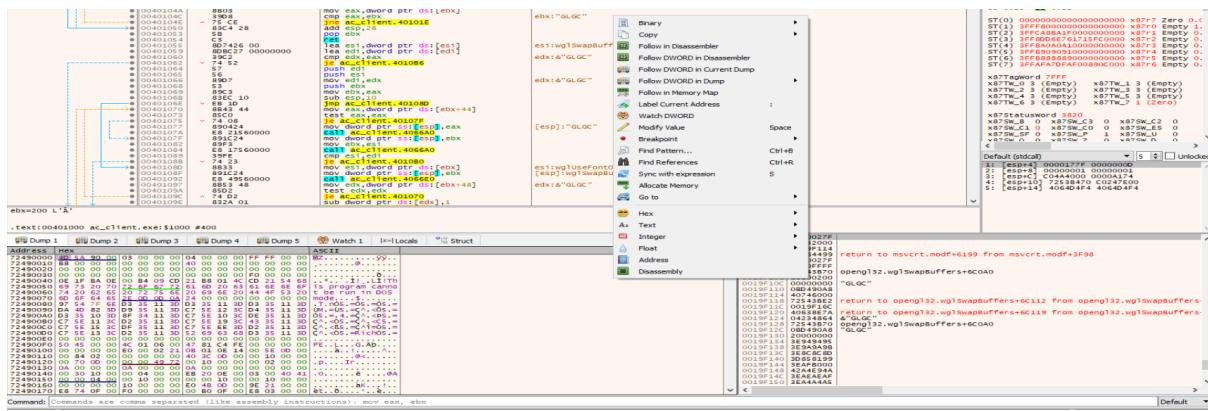


Jusqu'ici vous suivez ? Plutôt simple !

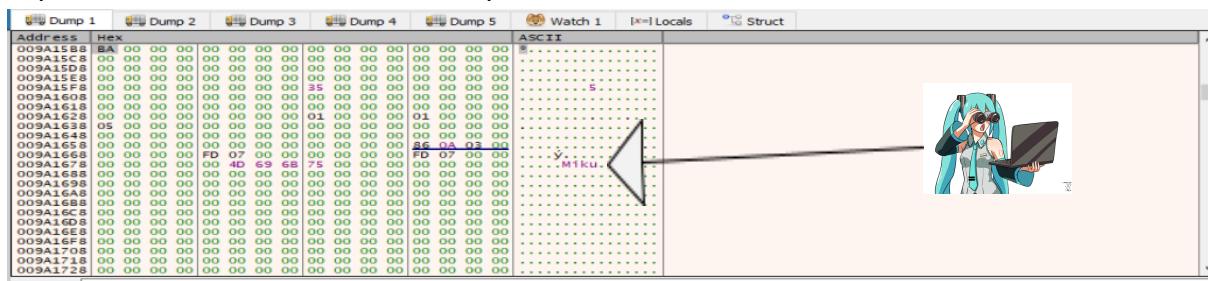


2. Allons regarder ce qui se passe à l'adresse mémoire précédemment récupérée.

Clique droit sur l'interface dump1 > Got to > Copier coller l'adresse mémoire > Entrée



On voit à l'adresse suivante une valeur en Hexadécimal (notre nombre de munitions). On clique droit sur notre adresse > breakpoint > hardware, write > dword



Une simple conversion et on retrouve notre nombre de munitions actuel.

From
To

Enter hex numbers

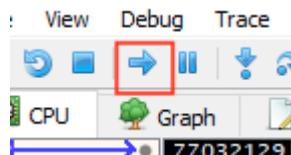
16

= Convert
x Reset
Swap

Decimal number (3 digits)

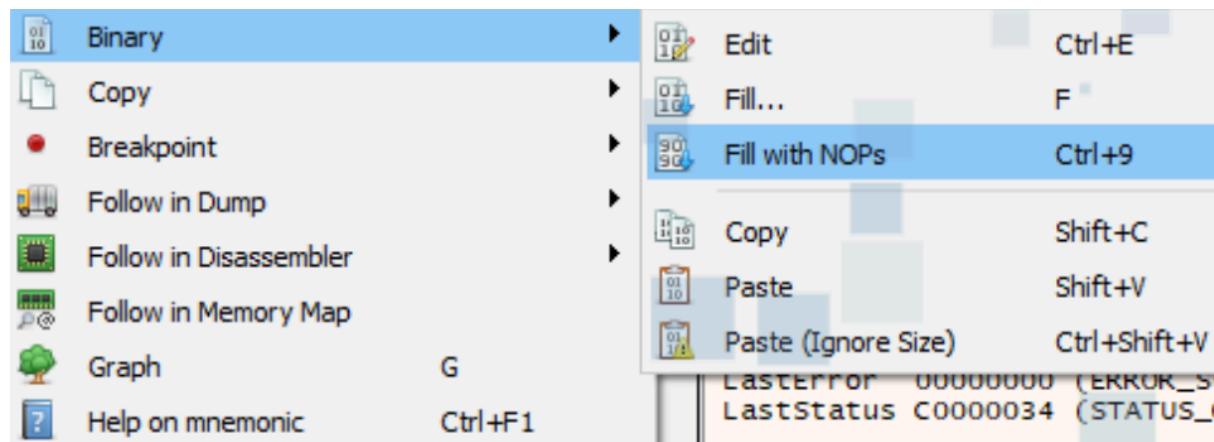
186
10

On peut ensuite cliquer sur la flèche pour reprendre l'exécution du programme.



On se retrouve avec cette exécution de code, au-dessus de notre breakpoints, on voit l'instruction DEC qui va décrémenter notre nombre de munitions.

On peut cliquer droit, binary, fill with NOPs



Supprimez le breakpoint puis revenez sur le jeu et vous verrez que vous pourrez tirer en illimité.

Cette partie n'était qu'une petite introduction, préparez-vous à entrer dans le sujet encore plus profondément, à base de Code Caves, d'allocation dynamique de mémoire et enfin de la programmation en C++!

Bonne chance, c'est long pour tout le monde mais s'essayer et progresser vous fera aimer cette partie ! Ne vous découragez pas...

7. Comprendre et faire du Reverse

7.0. Introduction pour le chapitre:

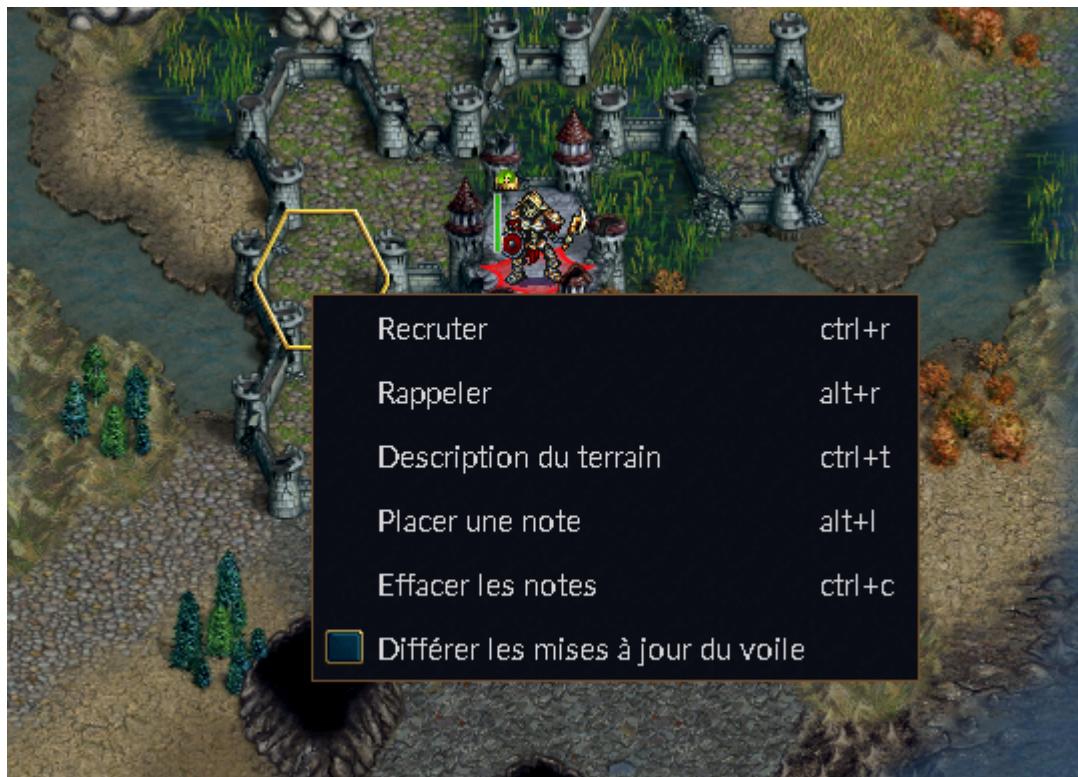
Pour cette section, nous allons effectuer un hack sur le jeu Bataille pour Wesnoth afin de pouvoir mettre en lumière les appels de fonctions et comment modifier le code à notre avantage.

Pour ce faire, nous allons reprendre ce que nous avons fait dans les leçons précédentes et y rajouter une brique.

Cela peut sembler redondant mais c'est utile de répéter les mêmes choses pour être sûr de comprendre.

7.1. Repérer ce que l'on veut changer:

En essayant de recruter une unité, voilà le menu que j'ai en face de moi, on va essayer de comprendre les dessous de ce code, et de modifier le comportement de cette fonctionnalité.





Commençons par comprendre le deçà du mécanisme de recrutement !
Voici les actions que nous avons faites :

File: [615076.jpg](#) (196 KB, 1280x1970)

Anonymous 01/26/25(Sun)14:57:53 No.1663937 ►

> be me
> clique sur le menu
> un menu apparaît
> puis sur recruter
> un nouveau menu apparaît
> choisi un soldat
> compare mon nombre d'or au prix du soldat
> soustrait mon nombre d'or

7.2. Comprendre ce que l'on veut changer:

Voilà comment on pourrait imaginer les fonctions et comment celles-ci interagissent entre elles.

```
handle_context_menu()
    recruit_unit()
        find_unit_in_unit_list()
            subtract_unit_cost()
                subtract_gold()
```

Nous savons comment récupérer la fonction `subtract_gold()` depuis notre débogueur. Notre but va donc être de remonter jusqu'au menu, puis de le modifier pour en changer le fonctionnement.

Bien entendu, dans notre débogueur nous n'avons pas un code en clair mais une succession d'instructions, l'appel d'une fonction se fait avec l'instruction 'call' et le retour par l'instruction 'ret' ou 'ret'.

Petit exemple:

```
main:
    mov eax, 0 ; → 1 (ordre de passage)
    call increase_eax ; → 2
    mov ebx, eax ; → 6

increase_eax:
    add eax, 1 ; → 3
    mov ecx, eax ; → 4
    ret
```

On y voit 2 fonctions:

- main
- increase_eax

NB : Le signe ‘;’ est un commentaire en langage assembleur. Ce qui y suit n'est pas compris dans le code

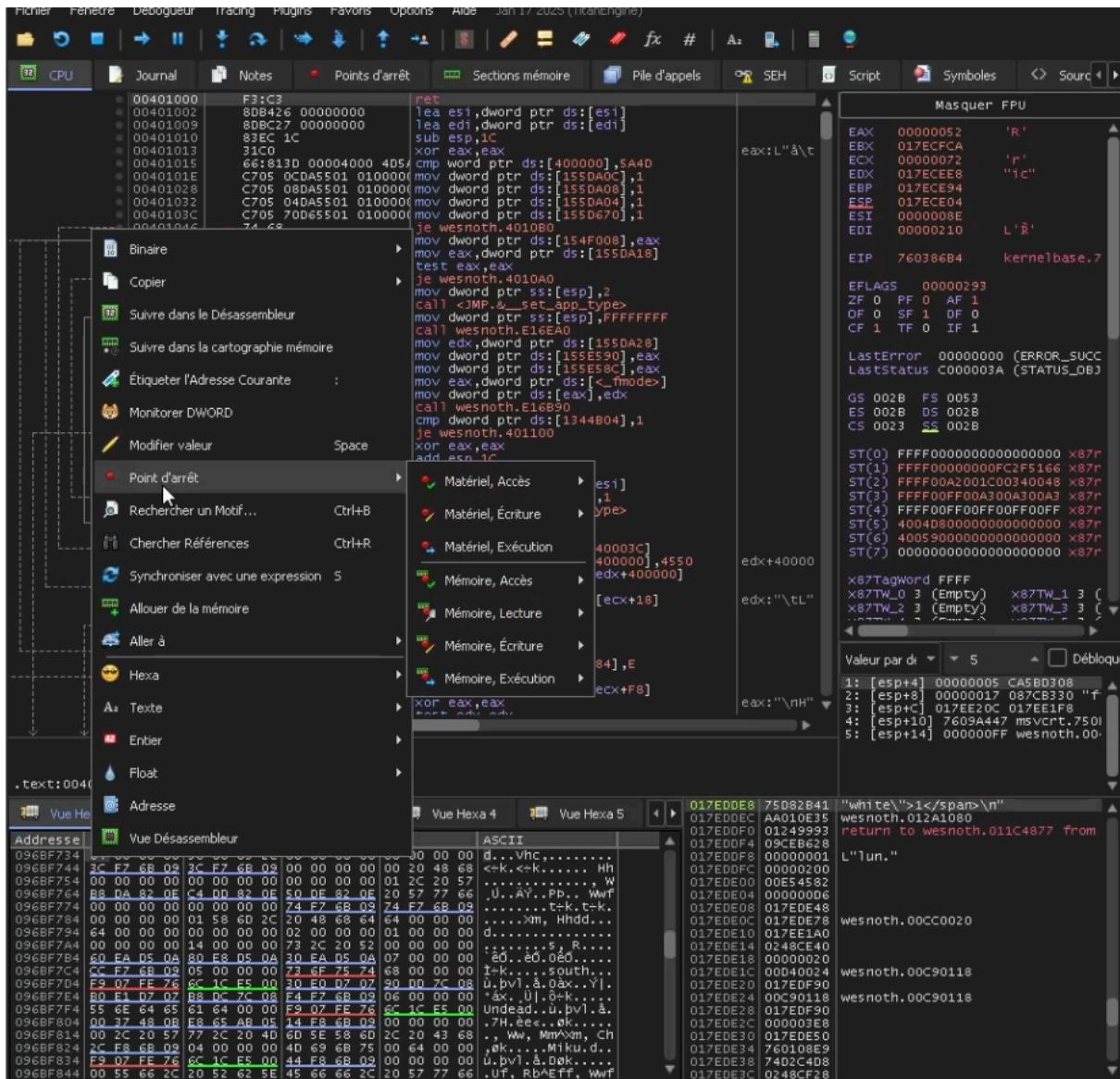
Voici une explication du code ci-dessus pour vous faciliter la compréhension.

Le code débutera toujours par la fonction main.

1. *On attribue la valeur 0 dans le registre eax*
2. *On appelle la fonction increase_eax*
3. *Dans la fonction increase_eax*
 - a. *On ajoute 1 au registre eax*
 - b. *On met la valeur de eax dans ecx*
 - c. *On retourne à la fonction main*
4. *On insere la valeur de eax (1) à ebx*

7.3. Mise en Pratique:

Après avoir récupéré l'adresse mémoire qui contenait les informations sur le nombre de pièces d'or, nous allons placer un breakpoint à cette adresse.

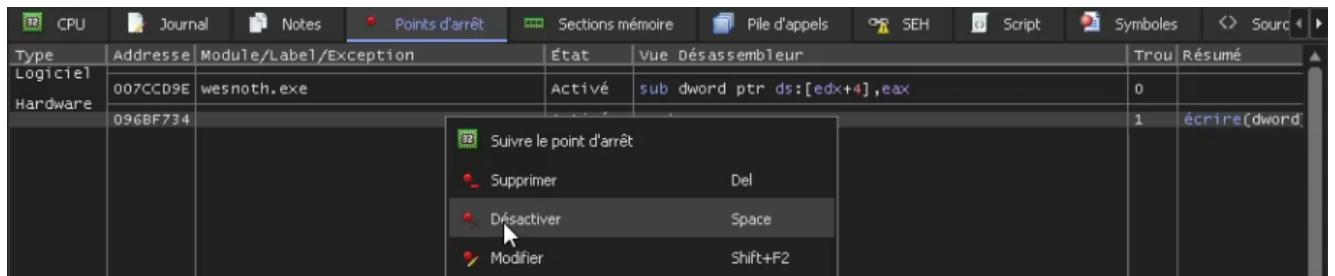


On tente de recruter une unité, le débogueur nous arrête à notre breakpoint précédemment placé, et on repère l'instruction qui soustrait notre nombre de pièces d'or.

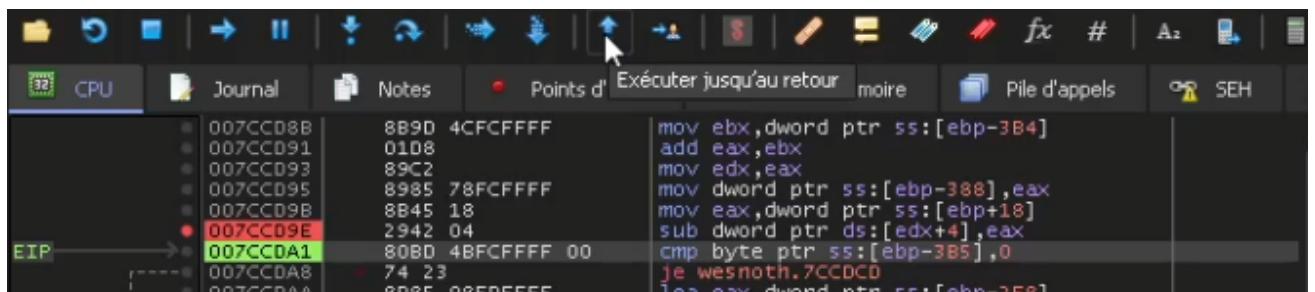
En cliquant sur le bouton à gauche, on va placer un breakpoint à cette instruction.



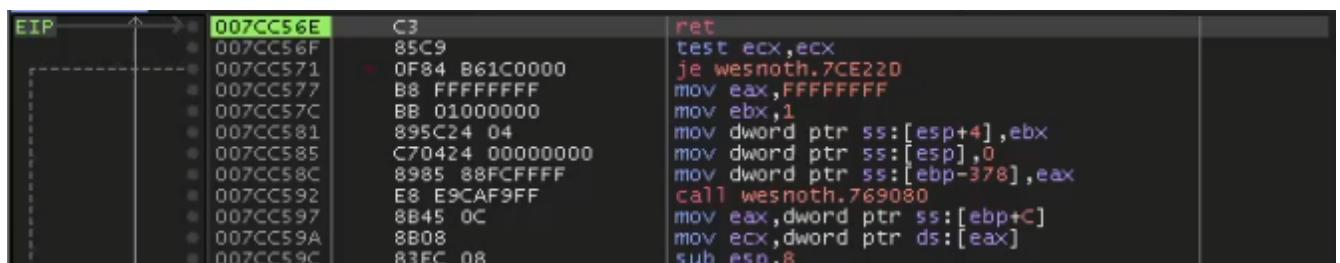
On peut désactiver l'ancien breakpoint.



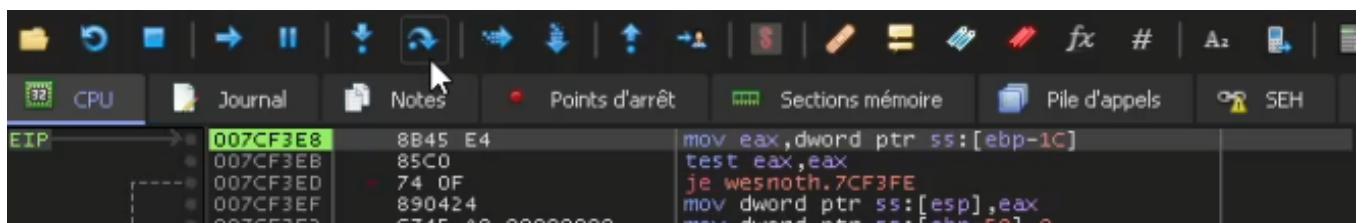
Une fois le breakpoint mit, on peut exécuter le code jusqu'à son instruction retour.



On se retrouve donc avec cela.



Puis en cliquant sur StepOver, nous nous retrouvons avec cela.



Ensuite, répétez le même processus jusqu'à tomber sur quelque chose comme cela :

Fichier Fenêtre Débogueur Tracing Plugins Favoris Options Aide Jan 17 2025 (TitanEngine)

CPU Journal Notes Points d'arrêt Sections mémoire Pile d'appels SEH

```

00CAF1A 8B01 mov eax,dword ptr ds:[ecx]
00CAF1C 8D7426 00 lea esi,dword ptr ds:[esi]
00CAF20 FF50 54 call dword ptr ds:[eax+54]
00CAF23 BO 01 mov al,1
    E9 D3F9FFFF jmp wesnoth.CCA8FD
    8B01 mov eax,dword ptr ds:[ecx]
    8D7426 00 lea esi,dword ptr wesnoth.00CAF20
    FF50 40 call dword ptr add esp,28
    BO 01 mov al,1 pop ebx
    E9 C3F9FFFF jmp wesnoth.ret 10
    8B01 mov eax,dword ptr ds:[ecx]
    8D7426 00 lea esi,dword ptr ds:[esi]
    FF50 3C call dword ptr ds:[eax+3C]
    BO 01 mov al,1
    E9 B3F9FFFF jmp wesnoth.CCA8FD
    8B01 mov eax,dword ptr ds:[ecx]
    8D7426 00 lea esi,dword ptr ds:[esi]
    FF50 38 call dword ptr ds:[eax+38]
    BO 01 mov al,1
    E9 A3F9FFFF jmp wesnoth.CCA8FD
    8B01 mov eax,dword ptr ds:[ecx]
    8D7426 00 lea esi,dword ptr ds:[esi]
    FF50 34 call dword ptr ds:[eax+34]
    BO 01 mov al,1
    E9 93F9FFFF jmp wesnoth.CCA8FD
    8B01 mov eax,dword ptr ds:[ecx]
    8D7426 00 lea esi,dword ptr ds:[esi]
    FF50 30 call dword ptr ds:[eax+30]
    BO 01 mov al,1
    E9 83F9FFFF jmp wesnoth.CCA8FD
    8B01 mov eax,dword ptr ds:[ecx]
    8D7426 00 lea esi,dword ptr ds:[esi]
    FF50 2C call dword ptr ds:[eax+2C]
    BO 01 mov al,1
    E9 73F9FFFF jmp wesnoth.CCA8FD
    8B01 mov eax,dword ptr ds:[ecx]
    8D7426 00 lea esi,dword ptr ds:[esi]
    FF50 28 call dword ptr ds:[eax+28]
    BO 01 mov al,1
    E9 63F9FFFF jmp wesnoth.CCA8FD
    8B01 mov eax,dword ptr ds:[ecx]
    8D7426 00 lea esi,dword ptr ds:[esi]
    FF90 2C010000 call dword ptr ds:[eax+12C]
    BO 01 mov al,1
    E9 54F9FFFF jmp wesnoth.CCA8FD
    8B01 mov eax,dword ptr ds:[ecx]
    8D7426 00 lea esi,dword ptr ds:[esi]
    FF90 50010000 call dword ptr ds:[eax+150]
    BO 01 mov al,1
    E9 45F9FFFF jmp wesnoth.CCA8FD
    8B01 mov eax,dword ptr ds:[ecx]

```

a1=0

.text:00CAF23 wesnoth.exe:\$8CAF23 #8CA323

Pourquoi je sais que nous sommes remontés jusqu'à l'appel de la fonction de recrutement ?

Et bien en analysant le code, j'ai vu que l'appel ressemblait à un truc du style :

`call dword ptr ds:[eax+0x54]`

Alors que toutes les autres appelaient une fonction de ce style :

je wesnoth.ABCD123

Voilà comment on peut imaginer le code du menu :

```
void* context_menu_functions[MAX_FUNCTIONS] = {  
    terrain_description,  
    recruit_unit,  
    ...  
}  
  
context_menu_functions[option_selected]();
```

7.4. Changer le code

On sait que l'adresse [eax+0x54] contient le choix recruit_units.

Changeons l'offset à 0x68, cela permettra d'ouvrir un mode de debug !

Ce code stocke un pointeur vers chaque fonction dans un tableau. La variable 'option_selected' peut ensuite être utilisée pour récupérer la fonction correcte depuis le tableau et l'exécuter. Nous aborderons les pointeurs dans une leçon future. Il est important de noter que, même si nous avions une idée incorrecte du code original, la structure globale des branches sera toujours évidente dans le code d'un jeu.

Je vous laisse voir ce qui marche ou non...



8. Notions avancées de reverse engineering

8.1. La dynamicité de la mémoire

Nous avons donc fait notre premier cheat no-code, et nous comprenons un peu mieux le reverse engineering.

Notre cheat fonctionne, nous déçons les bots sans avoir à recharger, et nous commençons déjà à rêver de cheats plus avancés. Des triggerbots, des aimbots, des wallhacks, des multicheats...

Sauf que notre enthousiasme retombe très vite lorsque nous quittons le jeu et que nous le relançons, car surprise! Plus rien ne marche.

Pourquoi?

Pour des raisons de sécurité, l'espace mémoire des processus est placé de manière aléatoire dans la mémoire d'un ordinateur grâce à un processus nommé l'ASLR (Address Space Layout Randomization). Les adresses mémoire des points de vie de notre joueur, des munitions de notre arme, ainsi que des DLLs utilisées par le jeu sont donc complètement différentes à chaque lancement du jeu.

Cela est fait pour empêcher des attaques du style "retour vers la libc", où un hacker peut exécuter des fonctions de bibliothèques système, ou écraser la mémoire de processus importants pour le bon fonctionnement du système.

Nous ne nous y attarderons pas davantage. L'ASLR est un grand bien pour la sécurité de nos ordinateurs, même si elle gêne nos cheats et nos rêves d'armes qui ne rechargent jamais.

Il faut savoir que bien que l'endroit où un binaire est chargé change à chaque fois qu'il est lancé, le binaire et sa structure ne changent pas. Il est donc possible de trouver des moyens d'accéder aux éléments du jeu qui nous intéressent de manière constante, et ce, en suivant les pointeurs!

8.2. Un pointeur, c'est quoi?

Un pointeur est une valeur dans l'espace mémoire comme une autre. La seule différence avec d'autres valeurs telles que les munitions ou le username d'un joueur, est que le pointeur contient **l'adresse d'une autre valeur**, et non un nombre ou une chaîne de caractères.

Les pointeurs sont utilisés par les développeurs pour faire référence à des éléments du jeu et les modifier. Souvent, les développeurs veulent accéder à des valeurs très importantes, comme les points de vie du joueur, partout dans leur code, et pour cela ils en feront une variable globale statique.

Sans trop rentrer dans les détails du développement de jeux, il faut savoir que ces variables sont stockées dans l'exécutable à un endroit prévisible.

Même si une variable n'est pas globale et statique, et que son emplacement varie à chaque changement de carte par exemple, en étudiant le code du jeu, nous pouvons retracer les **chemins de pointeurs** (pointer paths) pour accéder à nos points de vie, notre argent, nos munitions.

Dans l'exemple précédent, on savait que telle adresse contenait notre argent.

Avant le lancement de la carte, un autre bout de code assembleur a dû accéder à cet emplacement en mémoire pour l'initialiser, et nous pourrions retrouver le code en question et voir quel autre code a pu initialiser cela à son tour.

Cela nous donnerait donc un pointer path, que nous pourrions suivre pour accéder à nos pièces d'or à chaque lancement.

Tout cela est bien beau, mais cela peut prendre du temps et s'avérer plus ou moins complexe selon le jeu et le nombre de bouts de code ayant besoin d'un emplacement mémoire en particulier, et c'est donc pour cela que nous allons bruteforce les pointer paths comme des bourrins! Je présente notre nouvel ami, le pointer scan!

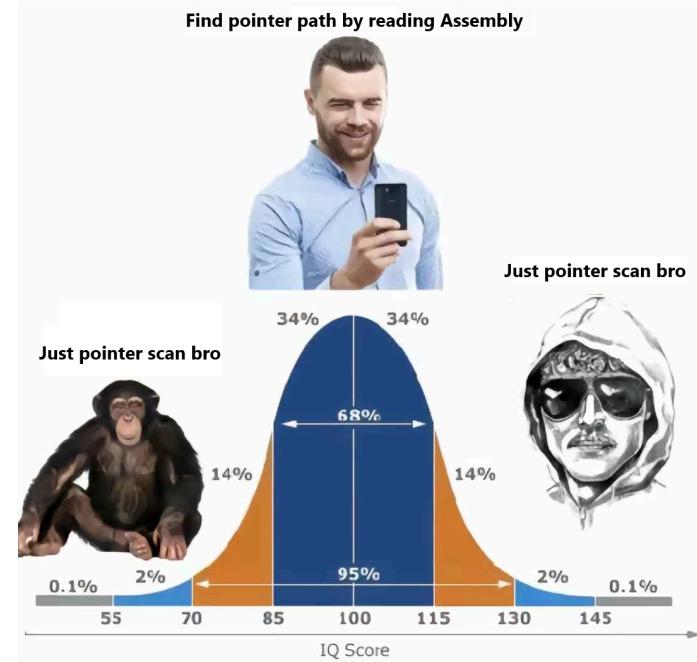
8.3. Le pointer scan

Le scan de pointeur, que nous allons effectuer sous Cheat Engine dans cette section, présente un grand gain de temps pour tout bon reverser/modder/cheater.

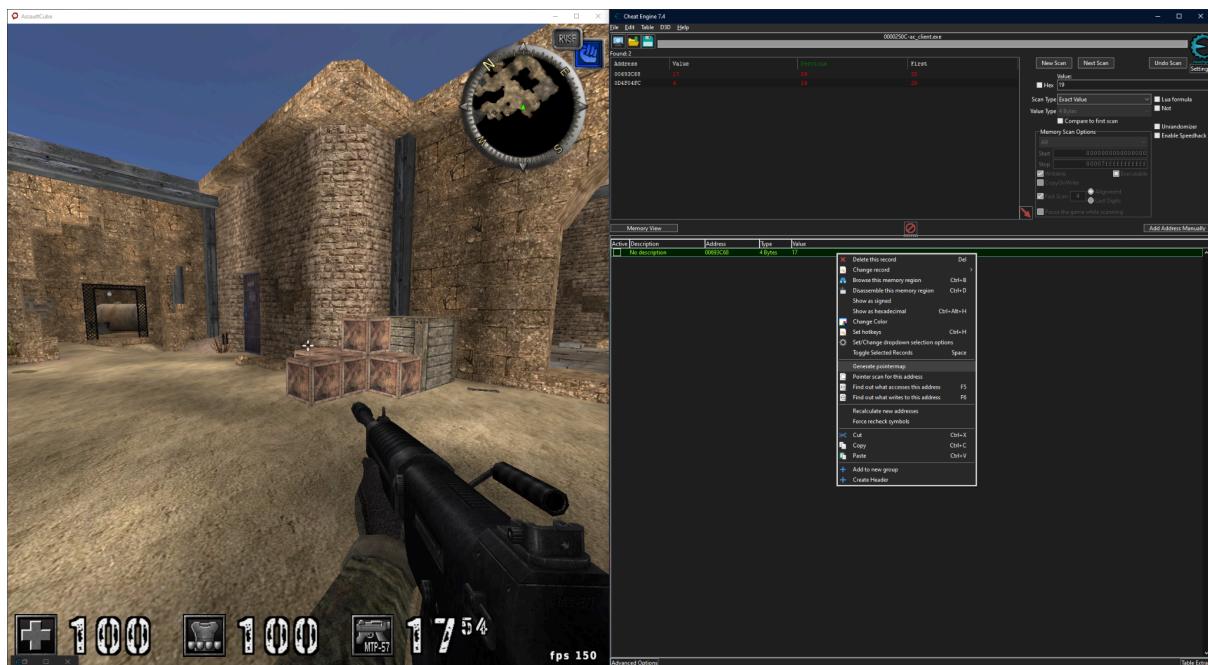
Beaucoup d'informations en ligne sont erronées, et le but de cette section va être de vous montrer à quel point ce concept est simple. Pour ce faire, nous allons trouver un pointer path nous menant vers les munitions dans notre chargeur sous AssaultCube.

La méthode est la suivante:

- On trouve notre valeur de munitions en mémoire.
- On copie absolument tous les pointeurs qu'on trouve en mémoire, et on les regroupe dans un “pointermap”.
- On ferme le jeu puis on le relance.
- On retrouve notre valeur de munitions à nouveau.
- On effectue un pointer scan, en prenant le pointermap précédent pour comparer et permettre à Cheat Engine d'éliminer de nombreux chemins de pointeurs erronés.
- Cheat Engine va bruteforce, en suivant tous les chemins de pointeurs qu'il peut trouver, et nous présenter seulement ceux qui ont abouti à notre valeur de munitions.
- On nettoie les résultats, et on obtient un ou plusieurs chemins de pointeurs nous permettant d'accéder à nos munitions à chaque lancement.



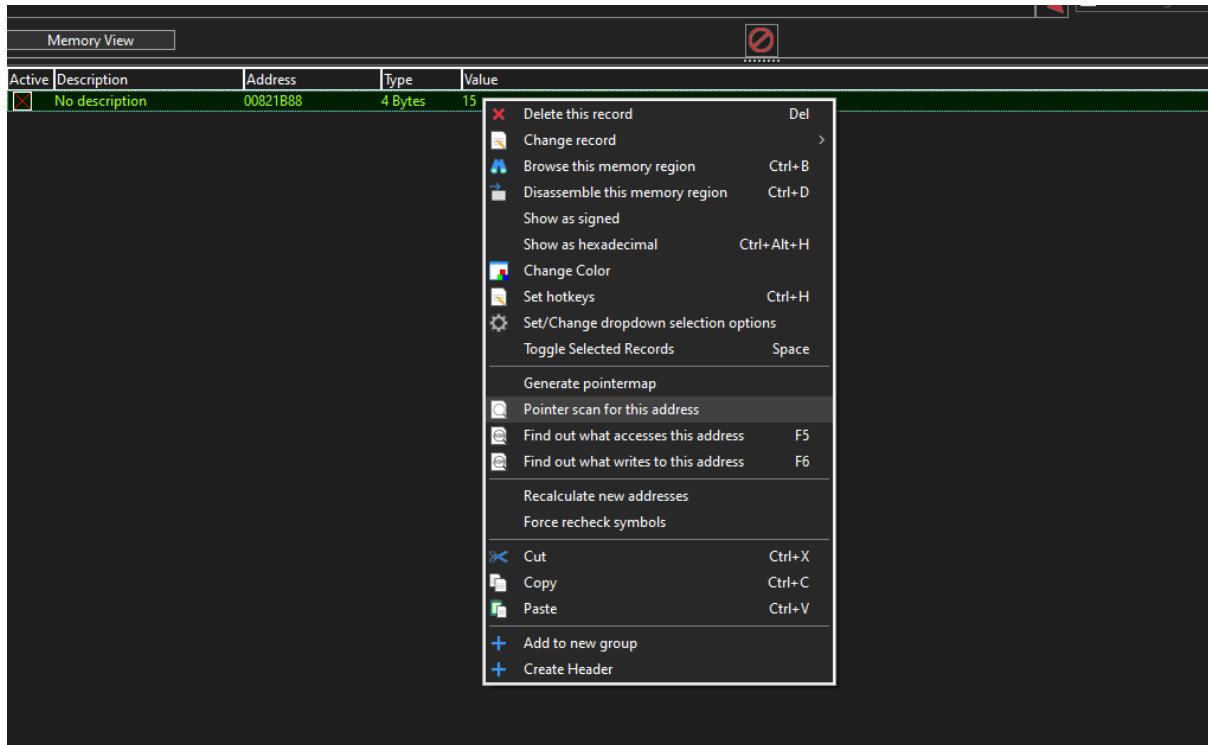
Démonstration!



Nous avons trouvé notre valeur de munitions en mémoire, on fait clic-droit dessus et on génère le pointermap (en cliquant sur “Generate pointermap”).

Une fois le pointermap sauvégarde, on ferme le jeu.
On le rouvre, et on retrouve nos munitions une deuxième fois.

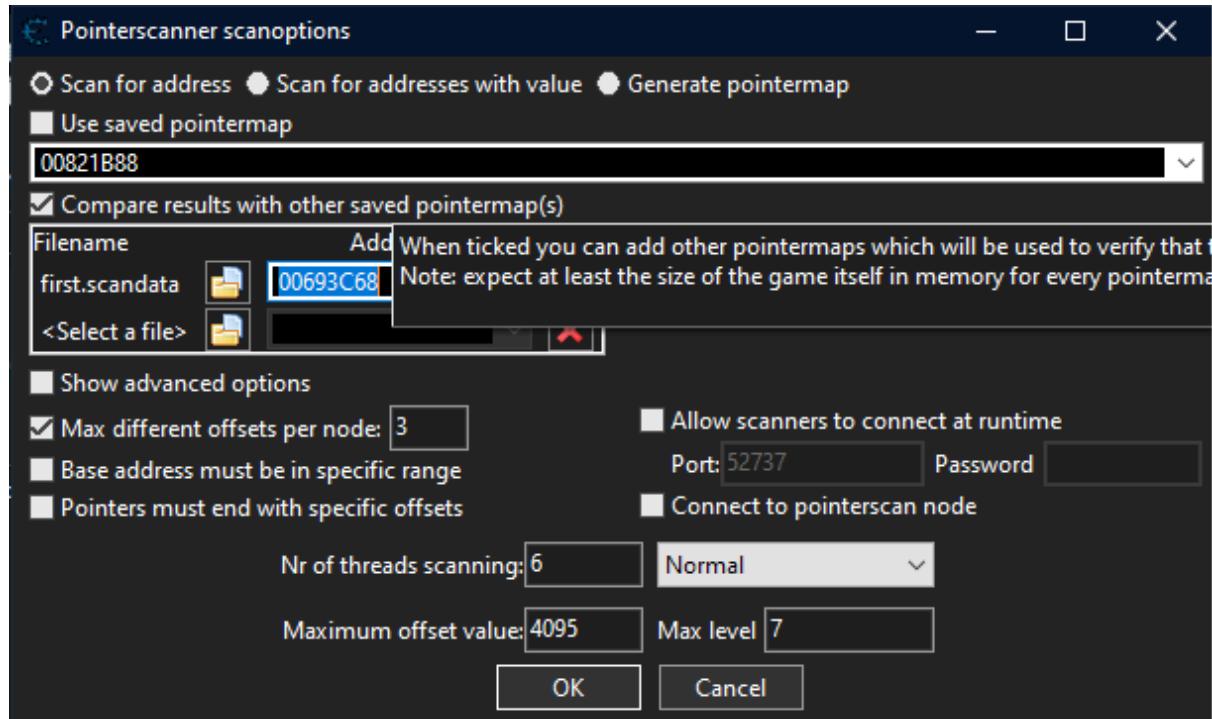
Une fois trouvé, on fait clic-droit sur nos munitions et nous effectuons un pointer scan en cliquant sur “Pointer scan for this address”.



Dans l'interface qui apparaît, ne changez rien car les options sont parfaites pour le cas actuel. Sur des jeux plus complexes, il faudra juste penser à augmenter le “Max level”, c'est-à-dire la profondeur maximale à laquelle Cheat Engine va bruteforce, et l'offset maximum.

N'oubliez pas de choisir l'option “Compare results with other saved pointermaps” et de sélectionner notre pointermap précédent, ainsi que l'ancienne adresse de nos munitions.

Elle a été sauvegardée avec notre pointermap, et elle apparaît dans le dropdown.



Pointer scan : ptrscan.PTR

File Distributed pointer scan Pointer scanner

4 Bytes

Pointer paths:683

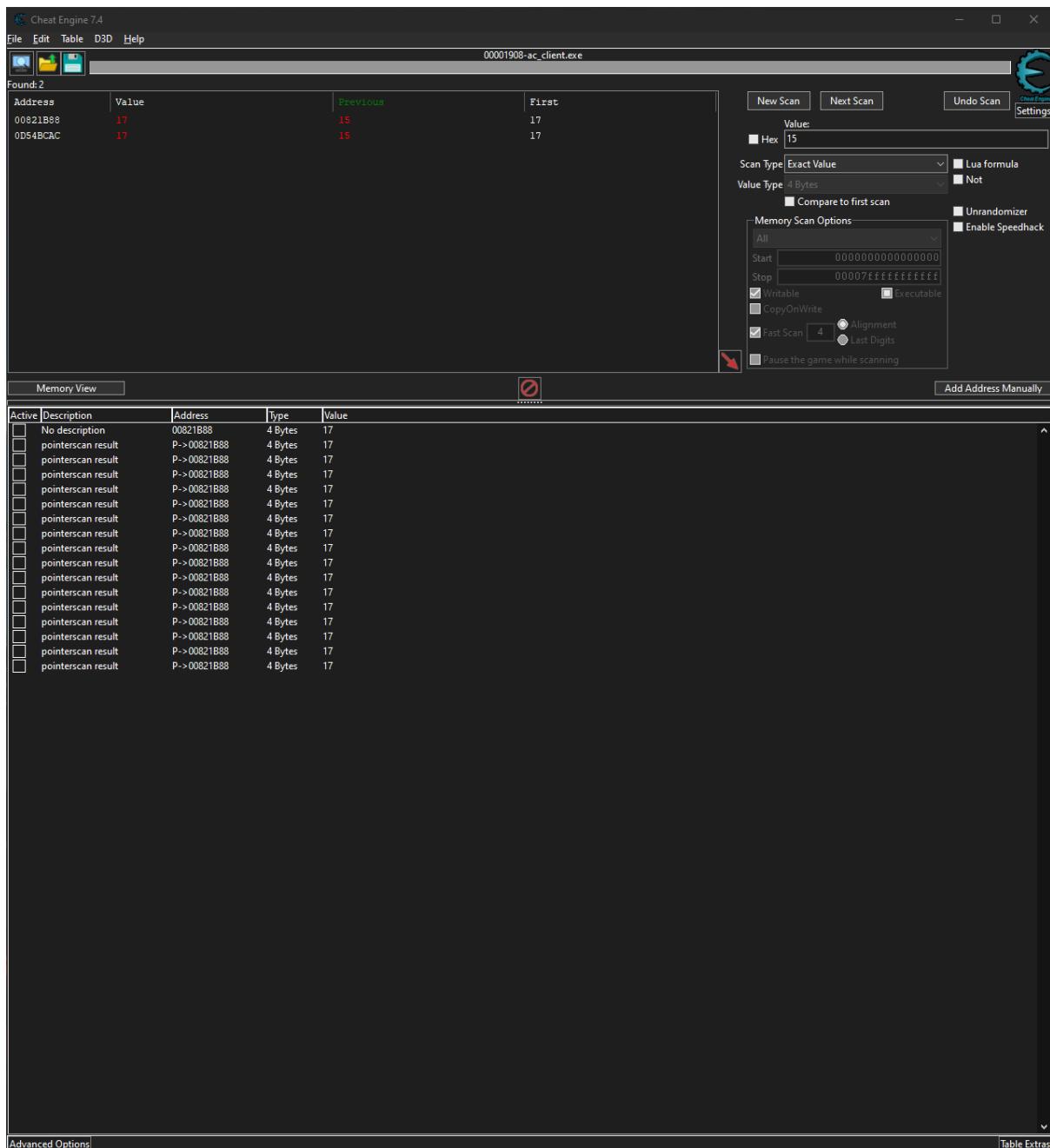
Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4	Offset 5	Offset 6	Points to:
"ac_client.exe"+00183828	8	170	34	64	98	220		00821B88 = 15
"ac_client.exe"+00183828	8	C7C	30	64	30	98	220	00821B88 = 15
"ac_client.exe"+00183828	8	87C	64	64	30	98	220	00821B88 = 15
"ac_client.exe"+00183828	8	588	30	98	64	220		00821B88 = 15
"ac_client.exe"+00183828	8	714	30	64	220			00821B88 = 15
"THREADSTACK0"-00000...	18	1F4	18	1D8	8	140		00821B88 = 15
"THREADSTACK0"-00000...	8	140						00821B88 = 15
"THREADSTACK0"-00000...	18	20C	8	140				-
"THREADSTACK0"-00000...	18	1F8	18	204	8	140		-
"THREADSTACK0"-00000...	18	1FC	8	140				00821B88 = 15
"THREADSTACK0"-00000...	18	1F8	8	140				00821B88 = 15
"THREADSTACK0"-00000...	18	1F8	18	1E4	8	140		00821B88 = 15
"THREADSTACK0"-00000...	18	1E0	10	34				00821B88 = 15
"THREADSTACK0"-00000...	18	1F4	18	1DC	10	34		00821B88 = 15
"THREADSTACK0"-00000...	18	1EC	10	28				00821B88 = 15
"THREADSTACK0"-00000...	18	1F4	18	1E0	10	30		00821B88 = 15
"THREADSTACK0"-00000...	18	1F4	18	1E8	10	28		-
"THREADSTACK0"-00000...	18	1F8	18	1E4	10	28		-
"THREADSTACK0"-00000...	18	1F8	18	208	10	24		-
"THREADSTACK0"-00000...	18	1F4	18	204	10	24		00821B88 = 15
"THREADSTACK0"-00000...	18	1D8	14	18				00821B88 = 15
"THREADSTACK0"-00000...	18	1F8	18	1D4	14	14		00821B88 = 15
"THREADSTACK0"-00000...	18	1F4	18	1DC	14	10		00821B88 = 15
"THREADSTACK0"-00000...	18	210	14	0				00821B88 = 15
"THREADSTACK0"-00000...	18	1F8	18	200	14	0		00821B88 = 15
"THREADSTACK0"-00000...	18	1F8	18	1F8	14	0		00821B88 = 15
"THREADSTACK0"-00000...	18	1F4	18	1FC	14	0		00821B88 = 15
"ac_client.exe"+0017E0A8	140							00821B88 = 15
"ac_client.exe"+0018AC00	140							00821B88 = 15
"ac_client.exe"+00195404	140							00821B88 = 15
"ac_client.exe"+00183828	8	3DC	570					00821B88 = 15
"ac_client.exe"+00183828	8	B74	30	744				00821B88 = 15
"ac_client.exe"+00183828	8	DFC	30	64	744			00821B88 = 15
"ac_client.exe"+00183828	8	790	30	64	64	744		00821B88 = 15
"ac_client.exe"+00183828	8	F80	64	30	64	64	744	00821B88 = 15
"ac_client.exe"+00183828	8	7F4	98	30	64	64	744	00821B88 = 15
"ac_client.exe"+00183828	8	E48	64	64	64	744		00821B88 = 15
"ac_client.exe"+00183828	8	A70	98	64	64	744		00821B88 = 15
"ac_client.exe"+00183828	8	950	98	64	744			00821B88 = 15
"ac_client.exe"+00183828	8	714	30	98	744			00821B88 = 15
"ac_client.exe"+00183828	8	780	30	30	64	98	744	00821B88 = 15
"ac_client.exe"+00183828	8	C04	30	64	64	98	744	00821B88 = 15
"ac_client.exe"+00183828	8	338	98	64	98	744		00821B88 = 15
"ac_client.exe"+00183828	8	588	30	98	98	744		00821B88 = 15

Voilà à quoi ressemble le pointer scan. Pas de panique! Dans cette liste figure de nombreux chemins de pointeurs qui sont erronés en réalité.

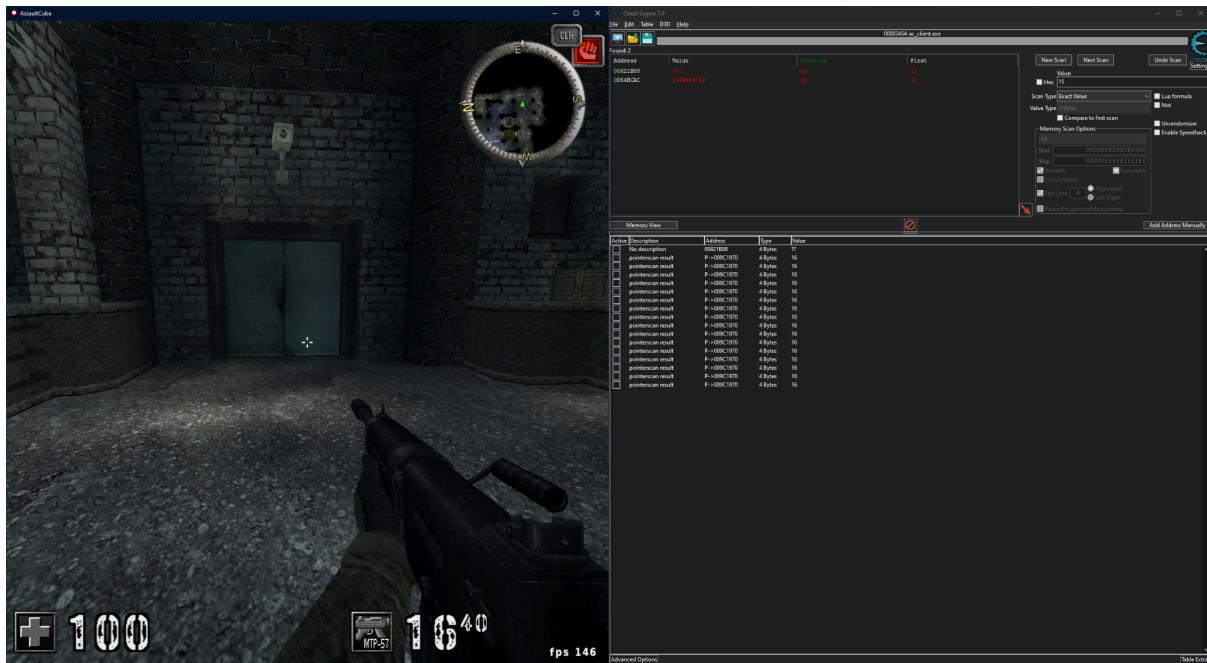
Pour trier les vrais des faux, il suffit de tirer avec notre arme et d'observer quels chemins de pointeurs correspondent à notre valeur de munitions. Pour en filtrer davantage, il peut également être intéressant de recharger l'arme.

Sur le screenshot ci-dessus, j'ai tiré avec l'arme une seule fois, et pourtant on observe que 5 pointer paths sont déjà faux et pointent sur rien (regardez la colonne de droite).

Un double-clic permet d'ajouter un pointer path à notre liste d'adresses, donc sélectionnons en plusieurs, aléatoirement.



Voilà, après en avoir sélectionné environ deux dizaines, je quitte le jeu.
Faites attention à cliquer sur “Yes” quand Cheat Engine demande s'il faut garder la liste d'adresses.



Je rouvre le jeu (la preuve, la map est différente), et on observe que tous mes pointer paths, sauf le premier, sont corrects et pointent bien sur la valeur de mes munitions. On peut filtrer davantage en relançant le jeu plusieurs fois et en changeant de carte.

Et voilà! Maintenant, peu importe si on quitte et relance le jeu, on a toujours accès à nos munitions. Une vraie plus-value lorsqu'on veut passer notre cheat à notre ami qui n'y connaît rien aux ordinateurs, sans qu'il ait besoin de retrouver la valeur de munitions en scannant sa mémoire.

C'est un prérequis également pour le cheat externe compilé de la prochaine section!

Nous pouvons cliquer sur File puis Save pour enregistrer notre liste d'adresses pour nos manipulations futures.

Malheureusement, même avec le pointer scan, nos pointer paths risquent tout de même de se casser! Comment ça, vous dites, j'ai bien scanné les pointeurs, et la structure de l'exécutable de change pas!

Eh bien malheureusement si, la structure de l'exécutable change notamment lorsque les développeurs effectuent des mises à jour et recompilent leur jeu. Personne ne sait alors comment le compilateur

changera la structure globale du programme, et nos pointer paths seront alors inutiles, car les instructions assembleurs auront changé.

Il nous faudra alors répéter le processus du pointer scan afin d'obtenir des offsets fonctionnant tout le temps, ou du moins jusqu'à la prochaine mise à jour...

Répéter ce processus peut s'avérer être fastidieux, c'est pourquoi nous pouvons gagner un peu de temps grâce au pattern scanning.

8.4. Le pattern scanning

Le pattern scanning peut se traduire en français par “scan de tendances” ou bien “scan de schémas/motifs”.

Le pattern scanning est aussi connu sous le nom de *signature scanning* ou encore plus simplement de *array of bytes scanning*.

L'appellation *array of bytes scanning* dévoile la simplicité du processus: nous fouillons un exécutable à la recherche d'une chaîne d'octets spécifique. Nous utiliserons désormais les termes pattern et signature de manière interchangeable.

L'intérêt est que même si l'exécutable change, nous regarderons seulement une partie du binaire, à savoir des instructions assembleur qui accèdent à la zone mémoire qui nous intéresse. Si cette partie du code n'a pas été touchée par la mise à jour, nous pouvons obtenir des offsets valables.

```
.text:10273B50 55          push    ebp
.text:10273B51 8B EC        mov     ebp, esp
.text:10273B53 56          push    esi
.text:10273B54 8D 75 04      lea    esi, [ebp+4]
.text:10273B57 8B 0E          mov    ecx, [esi]
.text:10273B59 E8 72 7F 55 00 call   sub_107CBAD0
.text:10273B5E 8B 0E          mov    ecx, [esi]
.text:10273B60 E8 CB 7F 55 00 call   sub_107CBB30
.text:10273B65 BB 00 D8 A4 2B 15 | mov    ecx, g_pLocalPlayer
```

Dans cet exemple, nous utilisons IDA, et nous avons nommé la variable qui nous intéresse `g_pLocalPlayer`.

A gauche des instructions assembleur, on peut voir les octets représentant ces instructions, et c'est ces octets que nous allons chercher dans l'exécutable.

Malheureusement, on ne peut pas scanner ces octets exacts, car g_pLocalPlayer est une variable, et les variables peuvent changer de valeur et d'adresse au cours de l'exécution, donc tout scan pour ces octets exacts échouerait.

Nos signatures incluent donc des wildcards, comme les shells. Ces signatures respectent en général le format popularisé par IDA, et s'appellent communément les "IDA patterns". Les octets représentant les variables sont tout simplement remplacés par des points d'interrogation, et il y a en général un offset qui permet d'accéder à la variable.

Le screen qui suit concerne un cheat dont je n'ai pas retrouvé la source.

```
sigs.glowObjectManager = Scan("client", "0F 11 05 ? ? ? ? 83 C8 01") + 3;
```

Néanmoins, on voit le principe. Dans sa fonction de setup, le cheat appelle une fonction scan qui effectuera le pattern scan. Il prend en argument le nom du processus du jeu ciblé, ici "client", et le pattern en deuxième argument. On voit bien les points d'interrogation à la place des variables. Le chiffre 3 à la fin permet d'accéder directement à la variable: en effet, le pattern scan retourne l'adresse du premier octet appartenant au match du pattern. Ici, il faut avancer de 3 octets pour accéder au premier point d'interrogation, c'est-à-dire au premier octet de la variable.

Voici comment un cheat peut se pérenniser malgré les mises à jour grâce au pattern scan. Il peut attraper des variables passées dans le code assembleur et tant que cette instruction n'est pas affectée par une mise à jour, alors il peut continuer à fonctionner.

Le choix de l'instruction est donc capital, il faut que ce soit une instruction qui pointe vers une zone mémoire dont l'adresse ne change pas. Ainsi les meilleures cibles pour faire des patterns scans sont les bouts de code qui initialisent des structures, comme par exemple une

grosse structure globale contenant le joueur local, car on obtient ainsi probablement un bloc mémoire qui ne changera pas de place.

Prenons un exemple plus compliqué, où les variables changeront très probablement de place. Si on veut obtenir l'adresse d'une liste de joueurs, on ne ciblera pas une instruction qui manipule, disons, l'argent d'un struct joueur dans la liste. En effet, la liste peut changer de taille, et lorsqu'un joueur quitte la partie, la zone mémoire référencée par l'instruction sera vide.

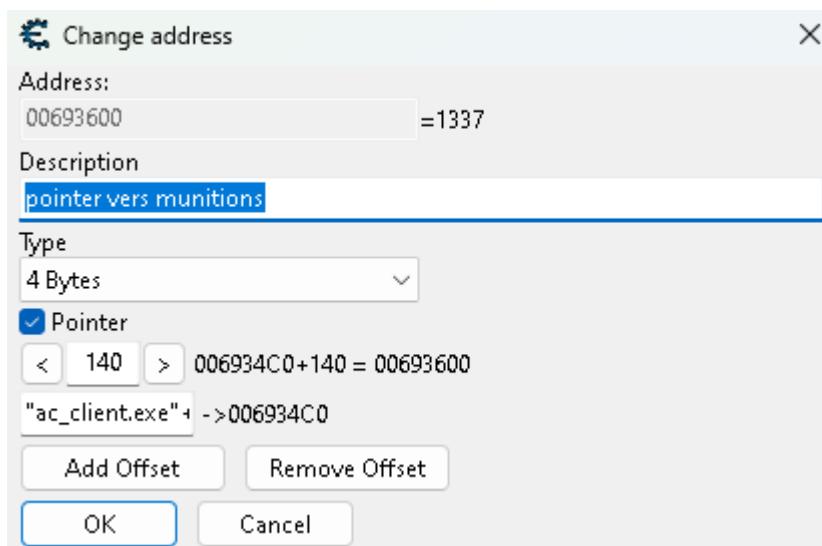
On tentera donc de trouver le moment où la liste de joueurs est initialisée, car si on utilise le même procédé pour trouver des références fixes au nombre de joueurs et à la taille d'un struct joueur, on pourra accéder à n'importe quel joueur dans la liste.

Le signature scan est un procédé qui est également utilisé par les anti-cheats, et nous y reviendrons dans une future partie.

9. Le Premier Cheat Code en C++

9.1. Après le Pointer Scan

On a pu trouver le pointeur et son offset dans la partie précédente. En ayant accès à ce pointeur, nous pouvons regarder son offset et décortiquer tout cela pour faciliter la compréhension. La compréhension de ce que l'on a sous nos yeux est très importante et vous facilitera pour la partie code.



"ac_client.exe"+0017E0A8 -> 006934C0 (puisque l'on ne voit pas bien)

Comment lire l'offset, que comprendre de cette image... Basiquement, il faut le lire du bas vers le haut.

*ac_client.exe → Base Address
0017E0A8 → Entity Address*

Base Address + Entity Address = 006934C0 = Entity

0x140 → Offset Munition

Entity + Offset Munition = location Munition

Maintenant que l'on sait un peu plus ce que cela veut dire, on va pouvoir refaire des modifications sur la mémoire du jeu et faire en sorte que cela soit valable même après avoir quitté le jeu.

9.2. La partie Code

Dans cette partie, nous allons faire plus ou moins ce que l'on faisait à la main sur CheatEngine, c'est-à-dire, récupérer l'adresse mémoire qui contient la valeur qui nous intéresse puis la modifier.

Le code est en entier à la suite de cette section.

Nous allons maintenant passer à l'explication bout à bout de ce code.

A. Les Headers.

```
#include <iostream>
#include <Windows.h>
#include <TlHelp32.h>
```

- a. `#include <iostream>` → Permet d'utiliser `std::cout` et `std::cin` pour afficher des messages.
- b. `#include <Windows.h>` → Fournit les API Windows
- c. `#include <TlHelp32.h>` → Contient les fonctions `CreateToolhelp32Snapshot`, `Module32First`, et `Module32Next` qui servent à récupérer les informations sur un processus et ses modules.

B. La Fonction GetModuleBase.

```
uintptr_t GetModuleBase(DWORD procID, const wchar_t*
modName) {
    uintptr_t moduleBase = 0;
    HANDLE hSnap =
CreateToolhelp32Snapshot(TH32CS_SNAPMODULE | TH32CS_SNAPMODULE32, procID);
    if (hSnap != INVALID_HANDLE_VALUE) {
        MODULEENTRY32 modEntry;
        modEntry.dwSize = sizeof(modEntry);
        if (Module32First(hSnap, &modEntry)) {
            do {
                if (!_wcsicmp(modEntry.szModule, modName))
{
                    moduleBase =
(uintptr_t)modEntry.modBaseAddr;
                    break;
                }
            } while (Module32Next(hSnap, &modEntry));
        }
    }
    CloseHandle(hSnap);
    return moduleBase;
}
```

Cette fonction sert à récupérer l'adresse de base d'un module (ex. ac_client.exe) chargé en mémoire dans le processus cible.

- a. Prend en entrée l'ID du processus (procID) et le nom du module (modName).
- b. Crée un snapshot de tous les modules chargés dans le processus.

- c. Parcourt tous les modules pour trouver celui qui correspond au nom donné (modName).
- d. Retourne l'adresse de base du module trouvé.

L'adresse de base change à chaque lancement du jeu, donc on doit la retrouver dynamiquement.

C. Le Main. L'Initialisation.

```
int main() {
    DWORD procID = 0;
    HWND hwnd = FindWindowA(NULL, "AssaultCube");
    GetWindowThreadProcessId(hwnd, &procID);
    HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, NULL,
procID);
```

Trouver l'ID du processus (PID) :

- a. FindWindowA(NULL, "AssaultCube") → Trouve la fenêtre du jeu en utilisant son nom.
- b. GetWindowThreadProcessId(hwnd, &procID) → Récupère l'ID du processus associé à la fenêtre.

Ouvrir le processus en mémoire :

- c. OpenProcess(PROCESS_ALL_ACCESS, NULL, procID) → Ouvre le processus avec tous les droits d'accès.

D. Trouver l'adresse des Munitions.

```
uintptr_t base = GetModuleBase(procID, L"ac_client.exe");
    uintptr_t ent = base + 0x17E0A8;
    uintptr_t entDef;
    ReadProcessMemory(hProc, (BYTE*)ent, &entDef,
sizeof(entDef), 0);
    int ammo = entDef + 0x140;
```

Obtenir l'adresse de base du jeu en appelant GetModuleBase().

Accéder à la structure de l'entité locale :

- a. base + 0x17E0A8 → Adresse du joueur dans la mémoire du jeu.
- b. ReadProcessMemory() → Lit l'adresse de l'entité dans entDef.
- c. ammo = entDef + 0x140 → Adresse mémoire des munitions du joueur.

E. Gestion des erreurs.

```
if (procID == NULL) {
    printf("mauvais proc id\n");
    Sleep(3000);
    exit(-1);
} else {
    if (hProc == NULL) {
        perror("Mauvais process\n");
        Sleep(3000);
        exit(-1);
    }
}
```

- a. Vérifie si l'ID du processus (procID) ou le handle du processus (hProc) est valide.
- b. Si procID == NULL → Le jeu n'est pas ouvert.
- c. Si hProc == NULL → Impossible d'ouvrir le processus (ex. problème de permission).

F. Écriture dans la memoire.

```
else {
    while(true) {
        int ammotest = 1337;
        WriteProcessMemory(hProc, (BYTE*)ammo,
&ammotest, sizeof(ammotest), 0);
        Sleep(1);
    }
}
```

- a. Une boucle infinie modifie la valeur des munitions en mémoire (ammo).
- b. WriteProcessMemory() remplace la valeur des munitions par 1337.
- c. Sleep(1) évite une surcharge du CPU en ajoutant une légère pause.

9.3. Le code en Entier

```

#include <iostream>
#include <Windows.h>
#include <vector>
#include <TLHelp32.h>

uintptr_t GetModuleBase(DWORD procID, const wchar_t* modName) {
    uintptr_t moduleBase = 0;
    HANDLE hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE | TH32CS_SNAPMODULE32, procID);
    if (hSnap != INVALID_HANDLE_VALUE) {
        MODULEENTRY32 modEntry;
        modEntry.dwSize = sizeof(modEntry);
        if (Module32First(hSnap, &modEntry)) {
            do {
                if (!_wcsicmp(modEntry.szModule, modName)) {
                    moduleBase = (uintptr_t)modEntry.modBaseAddr;
                    break;
                }
            } while (Module32Next(hSnap, &modEntry));
        }
    }
    CloseHandle(hSnap);
    return moduleBase;
}

int main() {
    DWORD procID = 0;
    HWND hwnd = FindWindowA(NULL, "AssaultCube");
    GetWindowThreadProcessId(hwnd, &procID);
    HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, NULL, procID);
    uintptr_t base = GetModuleBase(procID, L"ac_client.exe");
    uintptr_t ent = base + 0x17E0A8;
    uintptr_t entDef;
    ReadProcessMemory(hProc, (BYTE*)ent, &entDef, sizeof(entDef),
0);
}

```

```
int ammo = entDef + 0x140;

if (procID == NULL) {
printf("mauvais proc id\n");
Sleep(3000);
exit(-1);
}

else {
if (hProc == NULL) {
perror("Mauvais process\n");
Sleep(3000);
exit(-1);
}

else {
while(true) {
int ammotest = 1337;
WriteProcessMemory(hProc, (BYTE*)ammo, &ammotest,
sizeof(ammotest), 0);
Sleep(1);
}
}
return 0;
}
```

10. Les Anti-Cheats

10.1. Les fondamentaux

Ce chapitre présentera les bases des fonctionnement des anti-cheats, afin de comprendre comment les cheats les contournent.

Penchons-nous tout d'abord sur la nature des cheats. Les cheats ont de nombreuses catégories, ils peuvent être internes, externes, noyau... ; ils peuvent être des aimbots, des wallhacks...

Mais, fondamentalement, tout cheat doit lire la mémoire d'un processus, et potentiellement la modifier afin d'appliquer ces hacks.

Par conséquent, le boulot de n'importe quel anti-cheat est d'empêcher les autres processus d'avoir accès à la mémoire du jeu, que ce soit en lecture ou en écriture.

Voici quelques stratégies couramment utilisées par les anti-cheats.

10.2. Surveiller la mémoire du processus

Les anti-cheats vérifient très souvent l'intégrité des fichiers du jeu pour empêcher les patchs. Ils peuvent également hooker des appels WinAPI tels LoadLibrary pour bloquer les DLLs, ou OpenProcess pour empêcher tout processus d'avoir un Handle vers le jeu.

Le désavantage de bloquer toutes les DLLs est que l'on bloque les utilisations légitimes de ces injections. Par exemple, les overlays Steam ou Discord sont grossièrement des injections DLL.

L'anti-cheat peut également vérifier périodiquement si les valeurs en mémoire font sens, ou bien si les points de vie du joueur sont soudainement dans les millions.

10.3. Analyser les signatures des autres processus

Les anti-cheats peuvent scanner le processus du jeu ainsi que les autres processus pour vérifier la présence de signatures connues pour appartenir à des cheats. Ainsi, certains jeux se fermeront s'ils détectent Cheat Engine par exemple.

10.4. L'autorité du serveur

Enfin, un moyen simple d'empêcher les cheats basiques dans les jeux multijoueurs est de vérifier les actions du client. Par exemple, on ne validera pas l'achat d'un item sans vérifier si la quantité d'argent du joueur est suffisante.

10.5. Migrer vers le noyau

Comme la plupart des joueurs ont pu le constater, les anti-cheats kernel (aussi appelés KAC) sont de plus en plus répandus. Le fait de vivre dans le noyau permet à l'anti-cheat de détecter des appels systèmes tels Read/WriteProcessMemory, et bien d'autres techniques utilisées par les cheats.

10.6. Empêcher la virtualisation

Les jeux peuvent refuser de se lancer s'ils détectent la présence d'hyperviseurs.

10.7. Bannir en vagues

Une fois les cheaters détectés, la stratégie la plus courante consiste à les bannir en vague. Cela permet d'infliger un coup plus important à la réputation d'un cheat, ainsi que de compliquer la tâche du cheater, qui met à jour son cheat au fur et à mesure des mises à jour du jeu, et qui aura plus de mal à comprendre comment son cheat est détecté. Comme il n'est pas banni instantanément, il devra retracer tout l'historique de changements qu'il a apportés à son cheat.

Conclusion

Le sujet de la triche dans les jeux vidéos est fascinant car il permet d'étudier le fonctionnement des ordinateurs à un niveau très bas, et nous permet d'obtenir des résultats très funs et très impressionnantes.

Nous en profitons pour rappeler que ce guide a été écrit dans un but purement ludique, et que nos cheats fonctionnent sur le client AssaultCube contre des bots car c'est là leur but: s'amuser, et non gêner d'autres joueurs.

Les cheats sont un gros enjeu pour les joueurs et les développeurs, car selon [une étude de l'université du Birmingham datant de 2024](#), recensant les 80 plus grands sites de vente de cheats, on estime que ces sites génèrent entre 12,7 et 73,2 millions de dollars par an, et servent entre 30,000 et 174,000 joueurs par an. L'étude précise que ces chiffres n'incluent pas les cheats vendus sur des forums ainsi que les cheats partagés gratuitement, et que par conséquent le nombre de gens qui trichent dans les jeux vidéos est bien plus grand que ça.

De plus, avec la fin du support de Windows 10, une minorité grandissante se tourne vers Linux pour le gaming.



On remarque que sur le graphique ci-dessus, issu de Google Trends, les recherches pour le gaming sur Linux ont presque doublé depuis 2023, année où la date de fin de vie de Windows 10 a été annoncée. Les gens se tournent donc soit vers des machines virtuelles ou des couches de compatibilité POSIX comme [wine](#).

Cela est très intéressant du point de vue de la cybersécurité, car une machine virtuelle accède à la carte graphique via ce qu'on appelle le [GPU passthrough](#), qui est une solution récente de Nvidia. Un cheat pourrait donc potentiellement permettre d'échapper d'une VM et atteindre l'hôte par le biais de la carte graphique et de pilotes présentant des vulnérabilités.