Reversing - Le /who sur MoP (Cas pratique)

Galathil

Important : Ce tutoriel a été rédigé par Nobodie pour le site wow-emu.fr. Je le recopie ici à l'identique, Il se peut que certaines informations soient obsolètes à ce jour. Il manque les images, leurs présences seront cependant signalées dans le texte. Bonne lecture !

Reversing - Le /who sur MoP (Cas pratique)

Prélude : Ce sujet n'est pas original, il a été transféré d'un précédent forum d'émulation en date d'Octobre 2013. J'en suis l'auteur, je le transfère donc pour les besoins de ce forum. Bien qu'il soit en partie ré-écrit, il se peut que certains termes soient inadaptés ou ne soient plus d'actualité.

Bonjour à tous,

beaucoup de gens me l'ont demandé, voici un guide explicatif sur le travail que représente le reversing de paquets sur MoP.

1) Préambule : les impasses de ce guide

Contrairement à d'habitude, je me dois de faire un préambule car plusieurs points techniques seront occultés par choix. A divers passages de ce guide, je donnerais des valeurs sans expliquer la façon de les obtenir, et cela pour plusieures raisons. Premièrement, nous travaillons actuellement sur une extension attendue qu'est Mists-Of-Pandaria, je ne donnerais donc pas certains éléments pouvant amener certaines personnes sur une piste bien éclairée lorsque nous travaillons dans le noir le plus complet. Deuxièmement, il s'agit ici d'un guide servant à montrer dans les grandes lignes comment se déroule ce travail, les différentes étapes pour corriger une fonctionnalité du jeu, et le temps nécessaire à accomplir chacune de ces étapes. Un tutoriel sera probablement fait plus tard, dans lequel j'entrerais dans les détails pour chacune de ces étapes, mais il s'agit ici d'un guide qui se veut généraliste, à destination d'un grand nombre de lecteurs.

Etant donné le type de connaissances nécessaires à cette partie du développement d'un serveur privé, je présenterais ce guide sous forme d'une suite de captures d'écran suivies d'explications afin d'avoir un visuel sur le travail effectué. Bonne lecture à vous

2) Prérequis des connaissances nécessaires pour comprendre ce guide

Il est nécessaire de posséder un minimum de culture générale sur le développement. Je ne reviendrais pas sur les concepts de langage machine, de code assembleur, de gestion de la mémoire, de binaire, d'héxadécimal, de pseudocode, de buffer. Si vous ne connaissez pas ces termes, vous pourrez trouver une explication basique sur wikipedia. Il n'est pas nécessaire de connaître cela dans les détails, un simple survol de ces termes et de leur définition suffira pour ce guide.

3) le Reversing, qu'est-ce que c'est ?

Cette deuxième partie pourrait également faire partie du préambule. Comme vous le savez si vous avez déjà lu un de mes guides sur le fonctionnement d'un serveur de jeu, la partie la plus sensible, celle qui met le plus de bâtons dans les roues des développeurs, c'est le protocole de communication entre le client et le serveur. C'est par ce protocole que transitent tous les informations, et généralement ce protocole change à chaque version, obligeant les développeurs à mettre leur serveur à jour pour s'adapter à la dernière version. L'échange client<-> serveur se fait à l'aide de paquets de données. Lorsque ces paquets vont du client vers le serveur, on dit que ce sont des paquets CMSG. Lorsque le sens est inversé, on les appellent des paquets SMSG. La plupart des paquets s'articulent sous une forme simple de "question-réponse". Le client souhaite obtenir des informations -> il envoie un paquet CMSG au serveur. Le serveur traite cette requête via une fonction qu'on appelle un handler, puis renvoie le paquet SMSG contenant les informations souhaitées par le client. Celui-ci traite les informations via un autre handler, puis affiche le résultat en jeu (par exemple, le message d'un NPC, le spell lancé par un joueur, etc).

Le travail de Reversing, au sens large du terme, consiste à exploiter au maximum les données fournies par le client (principalement le code machine du client, mais aussi les DBCs, les MAPs, les messages d'erreurs tels que les Wow Errors, etc) afin de résoudre un problème. Dans notre cas, et dans le cadre de ce guide, nous allons nous limiter au code machine du client que nous désassemblons afin de retrouver le handler client qui envoie un certain message au serveur (et la structure du paquet envoyé), ainsi que le handler client qui reçoit un autre

handler client qui envoie un certain message au serveur (et la structure du paquet envoyé), ainsi que le handler client qui reçoit un autre message en provenance du serveur (et la structure du paquet reçu).

4) Le /Who, un exemple de fonctionnalité simple

Nous allons prendre en effet la fonctionnalité du /who comme exemple pour ce guide. Il s'agit d'une fonctionnalité simple mettant en scène deux paquets assez courts. Lorsqu'on entre des informations dans la fenêtre du /who, et qu'on clique sur "envoyer", un paquet CMSG est émis vers le serveur. Celui-ci reçoit le message, le parse (cela signifie découper le paquet correctement pour en extraire les variables) via son handler, effectue les requêtes SQL/ingame correspondantes pour obtenir la liste des personnages connectés répondant aux critères de recherches du joueur, puis renvoie un paquet SMSG vers le client contenant toutes ces informations. Le client reçoit alors ce message dans son handler, parse les informations reçues, et affiche la liste des personnages dans son interface.

5) Les outils nécessaires au Reversing

Nous allons commencer par les différents logiciels que j'utilise pour travailler. Premièrement, j'utilise QtCreator pour tous mes projets de développement Cpp. C'est cet IDE qui me permet de développer sur Wow.

[Image: 141223063632518797.jpg]

Ensuite, j'utilise un petit logiciel que nous avons développé (l'ensemble de l'équipe du projet Mist-Eria) afin de calculer des valeurs à partir des Opcodes (qui, je le rappelle, servent à identifier un paquet - savoir à quoi il sert, en quelque sorte).

[Image: 141223063632605068.jpg]

La troisième application est un désassembleur. J'utilise pour ma part IDA ainsi que son plugin HexRays qui me permet de décompiler le code machine (= code assembleur) et d'obtenir un pseudocode C pas trop moche.

[Image: 141223064732395959.jpg]

J'utilise également Notepad++ qui me sert de fourre-tout lorsque j'ai besoin d'identifier des textes. Il m'arrive également d'utiliser une feuille et un crayon, mais comme il est difficile d'en faire une capture d'écran, j'espère que vous savez tous de quoi il s'agit

[Image: 141223063632649773.jpg]

J'utilise enfin la console serveur qui va me permettre de visualiser certaines informations en provenance du client. Nous l'utiliserons d'ailleurs juste après, dès la prochaine partie de ce guide.

[Image: 14122306363374784.jpg]

6) Le paquet CMSG_WHO

La première étape est d'identifier l'Opcode (qui je le rappelle permet d'identifier le paquet lorsqu'il est envoyé/reçu). Pour cela, rien de plus simple, il suffit de taper /who dans la console pour obtenir la fenêtre du /who, puis de cliquer sur le bouton Envoyer. On obtient alors un message côté serveur qui nous indique avoir reçu un paquet dont l'opcode n'est géré par aucun handler. On peut voir ici qu'il s'agit de l'Opcode 203 (ou 0xCB en héxa).

[Image: 141223063634642741.jpg]

On le garde quelque part sur un papier/un bloc-note pour plus tard.

[Image: 14122306363464632.jpg]

On effectue ensuite une recherche dans IDA pour trouver toutes les occurence de la valeur 0xCB dans le client

[Image: 141223063634574917.jpg]

On obtient 3124 occurences parmi lesquelles se trouve la ligne qui correspond à l'envoi de cette Opcode. Plusieurs techniques nous permettent de rendre cette recherche instantannée, que je ne dévoilerais pas ici. Sachez simplement qu'il est possible de trouver cette information sans erreur directement dans le client.

[Image: 141223063634215062.jpg]

Une fois la bonne occurence trouvée, on accède au code de la fonction dans laquelle on peut effectivement voir un appel à la fonction CDataStore::PutInt32 (qui correspond à l'envoi d'un int32 dans le buffer) avec en paramètre la valeur 203 (qui correspond à l'Opcode trouvé

CDataStore__PutInt32 (qui correspond à l'envoi d'un int32 dans le buffer) avec en paramètre la valeur 203 (qui correspond à l'Opcode trouvé ci-dessus). On voit just en dessous un bout de code barbare (la ligne thiscall avec des DWORD puis v3 + 4). Cette fonction est en fait celle qui générera l'ensemble du paquet CMSG avant envoi vers le serveur. Le problème est que l'appel ne se fait pas directement, il s'agit d'un appel virtuel. Les explications sur l'obtention de l'adresse réelle du handler dépassent le cadre de ce guide, sachez simplement qu'il nous est possible de trouver l'adresse réelle du handler à partir de cette fonction.

[Image: 141223063634729996.jpg]

Nous obtenons l'adresse de la fonction chargée de remplir le paquet identifié par l'opcode 0xCB. Celui-ci contient bien des appels aux fonctions CDataStore__*, nous confirmant que nous sommes bien dans le handler de l'opcode. Voici son code :

```
int __thiscall sub_7114B0(int this, int a2)
{
  int v2; // eax@1
  int v3; // esi@1
  char v4; // cl@2
  int v5; // eax@3
  unsigned int v6; // eax@3
  int v7; // edx@3
  char v8; // cl@4
  unsigned int v9; // edi@5
  int v10; // ebx@6
  int v11; // eax@7
  int v12; // edx@7
  char v13; // cl@8
  unsigned int v14; // edi@10
  int v15; // ebx@11
  int v16; // eax@12
  int v17; // ecx@12
  char v18; // dl@13
  int v19; // eax@15
  char v20; // cl@16
  unsigned int v21; // edi@17
  int v23; // [sp+8h] [bp-10h]@1
  __int16 v24; // [sp+Ch] [bp-Ch]@1
  int v25; // [sp+10h] [bp-8h]@1
  int v26; // [sp+14h] [bp-4h]@1

  v3 = this;
  CDataStore__PutInt32(*(_DWORD *)(this + 20));
  CDataStore__PutInt32(*(_DWORD *)(v3 + 44));
  CDataStore__PutInt32(*(_DWORD *)(v3 + 16));
  CDataStore__PutInt32(*(_DWORD *)(v3 + 24));
  v2 = v3 + 64;
  v23 = a2;
  v24 = 0;
  LOBYTE(v26) = 0;
  v25 = v3 + 64;
  do
    v4 = *(_BYTE *)v2++;
  while ( v4 );
  sub_702380((int)&v23, v2 - (v3 + 65), v26);
  v6 = *(_DWORD *)(v3 + 28);
  LOBYTE(v26) = 0;
  sub_702260((int)&v23, v6, v26);
  v7 = *(_DWORD *)(v3 + 48);
  LOBYTE(v26) = 0;
  sub_7107F0(v7, v26);
```

```
  v5 = v3 + 112;
  LOBYTE(v26) = 0;
  do
    v8 = *(_BYTE *)v5++;
  while ( v8 );
  sub_702410((int)&v23, v5 - (v3 + 113), v26);
  v9 = 0;
  if ( *(_DWORD *)(v3 + 48) )
  {
    v10 = 0;
    do
    {
      v11 = v10 + *(_DWORD *)(v3 + 52);
      LOBYTE(v26) = 0;
      v12 = v11 + 1;
      do
        v13 = *(_BYTE *)v11++;
      while ( v13 );
      sub_702410((int)&v23, v11 - v12, v26);
      ++v9;
      v10 += 128;
    }
    while ( v9 < *(_DWORD *)(v3 + 48) );
  }
  FlushBits((int)&v23);
  v14 = 0;
  if ( *(_DWORD *)(v3 + 48) )
  {
    v15 = 0;
    do
    {
      v17 = v15 + *(_DWORD *)(v3 + 52);
      v16 = v17;
      v26 = v17 + 1;
      do
        v18 = *(_BYTE *)v16++;
      while ( v18 );
      CDataStore__PutData(v17, v16 - v26);
      ++v14;
      v15 += 128;
    }
    while ( v14 < *(_DWORD *)(v3 + 48) );
  }
  v19 = v3 + 112;
  do
    v20 = *(_BYTE *)v19++;
  while ( v20 );
  CDataStore__PutData(v3 + 112, v19 - (v3 + 113));
  v21 = 0;
  if ( *(_DWORD *)(v3 + 28) )
  {
    do
      CDataStore__PutInt32(*(_DWORD *)(*(_DWORD *)(v3 + 32) + 4 * v21++));
    while ( v21 < *(_DWORD *)(v3 + 28) );
  }
  return CDataStore__PutData(v25, strlen((const char *)v25));
}
```

[Image: 141223063635253972.jpg]

Jusqu'à présent, cette recherche a été quasiment instantannée (moins d'une minute pour atteindre ce handler. Mais les choses vont vite se compliquer. Maintenant que nous avons identifié le handler, il nous faut identifier la structure. Cela se fait en plusieurs parties. Premièrement nous créons une structure dans IDA nous facilitant la tâche d'identification.

[Image: 141223063634964691.jpg][Image: 141223063635198665.jpg]

A présent, il nous faut identifier la structure du paquet. Cela nous permet de savoir comment agencer les données. Pour un paquet de cette taille, cela prend environ 10 minutes en étant expérimentés. A l'époque ou nous avons travaillé dessus, nous débutions dans ce domaine et il nous a fallu près de 4 heures. Cette étape nous permet uniquement de trouver la structure, nous ne savons pas encore quelles données mettre à quelle place. J'élude la partie sur la recherche, mais il s'agit uniquement de lecture du code source, de compréhension du pseudocode, du langage assembleur parfois, et d'une certaine pratique et d'une logique de développement.

[Image: 141223063635588268.jpg]

Après ce travail sur la structure, il nous faut identifier les données à insérer à chaque endroit. Je ne donnerais pas la technique pour le faire, mais il faut compter une trentaine de minutes pour tout identifier. Là encore, tout est question de pratiques, les premiers paquets ont mis bien plus longtemps. A ce moment du code, nous pensons avoir la bonne structure, mais rien ne nous permet de confirmer cela. Le seul moyen d'en être sûrs est de patienter jusqu'à la fin pour bien vérifier que le comportement du /who est correct. Ce code source est celui implanté directement dans le serveur.

[Image: 141223063635824483.jpg]

L'ensemble de ces étapes nous a permis de trouver la structure du paquet CMSG_WHO en une quarantaine de minutes.

7) Le paquet SMSG_WHO

Maintenant que nous avons (peut-être correctement) identifié le paquet client, il nous faut trouver le paquet serveur pour envoyer la réponse au client. Ici, je vais donner immédiatement la valeur de l'Opcode sans dévoiler la façon de le trouver. Les opcodes serveur sont effectivement plus complexes à trouver que les opcodes client et les façons de les identifier dépassent le cadre de ce guide, tout en étant trop critiques pour être dévoilées pour l'instant. Il s'agit donc de l'Opcode 4433 (0x1151 en héxadécimal).

[Image: 141223063635550229.jpg]

L'ensemble des paquets serveurs passent par la même fonction appelée le ProcessMessage que vous pouvez trouver ici :

```
void __thiscall NetClient__ProcessMessage(void *this, int a2, unsigned int a3, int a4)
{
  int v4; // ebx@1
  void *v5; // edi@1
  unsigned int v6; // esi@1
  unsigned int v7; // eax@7
  unsigned int v8; // eax@11
  int v9; // ecx@12

  ++dword_10A0E64;
  v4 = a3;
  v5 = this;
  CDataStore__GetInt32(a3, (int)&a3);
  v6 = a3;
  if ( a3 & 0x80A )
  {
    if ( (a3 & 0xA58) == 2120 )
    {
      sub_7221C0(v5, 0, a2, a3, v4);
    }
    else
    {
      if ( (a3 & 0x248) == 520 )
```

```
      {
        JAM_ProcessMovement(v5, 0, a2, a3, v4);
      }
      else
      {
        v7 = a3 & 0x90A;
        if ( v7 == 2304 )
        {
          JAM_ProcessQuest(v5, 0, a2, a3, v4);
        }
        else
        {
          if ( v7 == 2048 )
          {
            JAM_ProcessSpell(v5, 0, a2, a3, v4);
          }
          else
          {
            (*(void (__thiscall **)(void *, unsigned int))(*(_DWORD *)v5 + 68))(v5, a3);
            v8 = v6 & 1 | ((v6 & 4 | ((v6 & 0x30 | (v6 >> 1) & 0x7FC0) >> 1)) >> 1);
            if ( (v6 & 0x4A) == 2 && (v9 = *((_DWORD *)v5 + v8 + 340)) != 0 )
              ((void (__cdecl *)(_DWORD, unsigned int, int, int))(v9 - ((v6 | (v6 << 16)) ^ 0x62A3A31D)))(
                *((_DWORD *)v5 + v8 + 2388),
                v6,
                a2,
                v4);
            else
              (*(void (__thiscall **)(int))(*(_DWORD *)v4 + 24))(v4);
          }
        }
      }
    }
  }
  else
  {
    JAM_ClientDispatch(0, a2, a3, v4);
  }
}
```

Cette fonction effectue différents calculs sur l'opcode afin de le rediriger vers différentes fonctions chargées de traiter chaque paquet.

[Image: 141223063636716227.jpg]

Afin de trouver la sub (le handler) correspondante au paquet serveur, nous utilisons notre logiciel OpcodeTool afin de calculer différentes valeurs que nous appelons Offsets. Cela nous amène à une fonction contenant un switch. Dans ce switch, nous retrouvons l'Offset trouvé, qui nous indique directement la fonction appelée.

[Image: 141223063636106990.jpg] [Image: 141223063635874786.jpg]

Une fois la sub identifiée, nous arrivons à la sub principale qui initialise quelques données. Cette fonction ne nous intéresse pas, nous poursuivons notre chemin vers la sub suivante indiquée en rouge sur l'image.

[Image: 141223063636229429.jpg]

Nous arrivons enfin au handler gérant le paquet SMSG_WHO. Son code se trouve ici :

```
int __thiscall sub_7DE9E0(void *this, int a2, int a3)
{
  int result; // eax@1
```

```
int v4; // ebx@1
unsigned int v5; // edi@1
void *v6; // esi@1
int v7; // edx@1
unsigned int v8; // ecx@3
int v9; // ecx@6
int v10; // edi@9
int v11; // eax@10
int v12; // ecx@10
int v13; // edx@10
int v14; // edi@11
int v15; // edx@12
int v16; // edx@12
int v17; // [sp+Ch] [bp-10h]@1
__int16 v18; // [sp+10h] [bp-Ch]@1
int v19; // [sp+14h] [bp-8h]@10
int v20; // [sp+18h] [bp-4h]@1

v4 = a2;
v6 = this;
v20 = 0;
CDataStore__GetInt32(a2, (int)&v20);
LOBYTE(a2) = 0;
v7 = a2;
*((_DWORD *)v6 + 8) = v20;
v17 = v4;
v18 = 2048;
result = sub_733B60(&v17, v7);
v5 = result;
if ( (unsigned int)result > *((_DWORD *)v6 + 4) )
{
  if ( (unsigned int)result > *((_DWORD *)v6 + 6) )
  {
    v8 = *((_DWORD *)v6 + 7);
    if ( !v8 )
      v8 = sub_75E730(result);
    if ( v5 % v8 )
      v9 = v5 + v8 - v5 % v8;
    else
      v9 = v5;
    result = sub_7B1820(v9);
  }
}
*((_DWORD *)v6 + 4) = v5;
v10 = 0;
a2 = 0;
if ( *((_DWORD *)v6 + 4) )
{
  do
  {
    LOBYTE(v20) = 0;
    v11 = sub_70A730(v20);
    v12 = *((_DWORD *)v6 + 5);
    LOBYTE(v19) = 0;
    v13 = v19;
    *(_DWORD *)(v10 + v12) = v11;
    *(_DWORD *)(v10 + *((_DWORD *)v6 + 5) + 48) = CDataStore__Read7Bits((int)&v17, v13);
    result = a2 + 1;
    v10 += 164;
    a2 = result;
```

```
      a2 = result;
    }
    while ( (unsigned int)result < *((_DWORD *)v6 + 4) );
  }
  v14 = 0;
  v20 = 0;
  if ( *((_DWORD *)v6 + 4) )
  {
    do
    {
      v15 = *((_DWORD *)v6 + 5);
      a2 = *(_DWORD *)(v14 + v15 + 48);
      CDataStore__ReadStringWithSize(v4, (void *)(v14 + v15 + 48), a2);
      *(_BYTE *)(v14 + *((_DWORD *)v6 + 5) + a2 + 48) = 0;
      v16 = *((_DWORD *)v6 + 5);
      a2 = *(_DWORD *)(v14 + v16);
      CDataStore__ReadStringWithSize(v4, (void *)(v14 + v16), a2);
      *(_BYTE *)(v14 + *((_DWORD *)v6 + 5) + a2) = 0;
      a2 = 0;
      CDataStore__GetInt32(v4, (int)&a2);
      *(_DWORD *)(v14 + *((_DWORD *)v6 + 5) + 148) = a2;
      a2 = 0;
      CDataStore__GetInt32(v4, (int)&a2);
      *(_DWORD *)(v14 + *((_DWORD *)v6 + 5) + 152) = a2;
      BYTE3(a2) = 0;
      CDataStore__GetInt8(v4, (int)((char *)&a2 + 3));
      *(_BYTE *)(v14 + *((_DWORD *)v6 + 5) + 156) = BYTE3(a2);
      a2 = 0;
      CDataStore__GetInt32(v4, (int)&a2);
      *(_DWORD *)(v14 + *((_DWORD *)v6 + 5) + 144) = a2;
      a2 = 0;
      CDataStore__GetInt32(v4, (int)&a2);
      *(_DWORD *)(v14 + *((_DWORD *)v6 + 5) + 160) = a2;
      result = v20 + 1;
      v14 += 164;
      v20 = result;
    }
    while ( (unsigned int)result < *((_DWORD *)v6 + 4) );
  }
  return result;
}
```

[Image: 141223063636817258.jpg]

A cette étape, le travail est identique au CMSG. Il nous faut dans un premier temps identifier la structure, chose faite en une dizaine de minutes.

[Image: 141223063636356255.jpg]

Une fois la structure identifiée, il faut identifier les données à insérer. Cela se fait à nouveau en une trentaine de minutes avec la pratique. A ce stade, nous avons terminé le travail de reversing.

[Image: 141223063637109023.jpg]

Si le travail a été fait correctement, il ne reste plus qu'à profiter du résultat et à admirer une belle liste des personnages connectés, ici Teraah et moi-même pour l'exemple. Voilà le fruit d'un peu moins de deux heures de travail pour gérer la fonctionnalité du /WHO.

[Image: 141223063636872671.jpg]

8) Généralisation du reversing à l'ensemble des paquets

Maintenant que vous avez conscience du travail effectué pour corriger une fonctionnalité simple telle que le /WHO, il est de mon devoir de rappeler la totalité du travail à accomplir. Premièrement, il existe environ 2500 paquets à corriger. Nous en avons vu 2 ensemble au travers de ce guide. Ensuite, il s'agit d'un système assez simple. Le fait que ce soit un paquet très visuel (on clique sur un bouton, le paquet est envoyé. On envoie le message, le paquet est reçu) nous permet de tester très facilement les différentes valeurs envoyés. Cela raccourcit grandement les étapes d'identification des variables qui prennent tout de même 30 minutes avec cette facilité. Je vous laisse imaginer ce qui arrive lorsque les paquets sont des paquets "invisibles" tels que tous les paquets envoyés avant l'accès au jeu (connexion au serveur) ou encore les systèmes nécessitant plusieurs envois/réceptions de messages (le système de quêtes par exemple nécessite près de 10 paquets différents pour afficher la quête dans son journal). Ensuite, les handlers de ces deux paquets sont relativement cours, avec moins de 200 lignes environ. Il faut savoir que les handlers de certains paquets de quêtes, de déplacement, ou de spells, font plus de 3000 lignes. Vous comprendrez que le risque d'erreur est très grand et qu'il arrive souvent de devoir relire plusieurs fois le handler pour ne louper aucune variable. Enfin, il s'agissait ici de variables plutôt simples à trouver, car l'ancien système de /who (des version Cataclysm) était semblable à celui-ci. Imaginez le travail lorsqu'il s'agit de paquets totalement inconnus, tels que le système de bataille de Pets par exemple.

Malgré la complexité et le côté fastidieux du reversing, sachez que c'est vraiment la partie la plus plaisante du travail car c'est à ce moment qu'on apprend le plus. On comprend comment fonctionne le jeu, mais également l'ordinateur, la mémoire, et tout ce qui fait tourner un tel système. En comprenant ce système, vous êtes rompus à toutes les situations de développement et c'est un pur bonheur que de réussir à passer outre la complexité pour que le client affiche ce que vous souhaitez.

C'est ici que s'achève ce guide plutôt long. Comme d'habitude, j'espère obtenir de nombreuses remarques constructives afin d'améliorer les points sensibles et de faire en sorte que ce guide soit accessible au moins chevronné des développeurs.

Merci de m'avoir lu jusqu'au bout.