Broken Synapse

Je suis un grand fan de Frozen Synapse depuis sa sortie en 2011. Il s'agit d'un jeu de stratégie en ligne comparable à une partie d'échecs dans laquelle chaque joueur déciderait de son coup en même temps et découvrirait le résultat de ses actions à l'issue du tour.

L'idée d'aller regarder un peu le fonctionnement interne du jeu m'avait traversé la tête plusieurs fois au cours de ces dernières années et je me suis enfin résolu à consacrer un peu de temps à ce projet inutile. Ma motivation était la suivante : à l'instar d'un logiciel de poker en ligne mal codé, j'avais la certitude que le client du jeu recevait des informations sur la position de l'adversaire qu'il suffisait de regarder pour obtenir un avantage tactique en duel. Pas que j'aie l'intention de tricher - mais j'étais très curieux de savoir si c'était possible.

Le premier réflexe a été d'utiliser Wireshark pour regarder un peu ce qui transite par le réseau. Un filtre basique (`ip.addr == 62.197.39.230`, l'IP étant visible dans la configuration) permet d'observer tous les échanges avec le serveur. Bonne nouvelle, rien n'est chiffré. Le protocole ressemble à ceci :

```
textcom     prelogon
textcom     command     saltRec             29007       2
textcom     login       [user]              [password]  33
textcom     loggedIn    30787               [user]
textcom     command     setMyOS             windows.steam
writeFile   psychoff/activeGames.txt        121         gz    161
textcom     command     refreshPeopleOnline
```

Il me faut un peu de temps pour trouver quel algorithme est utilisé pour hacher le mot de passe. Il s'agit de `MD5(salt + MD5(password))`, à un détail près : le MD5 du mot de passe est passé en majuscules avant d'être re-haché.

En quelques heures, j'ai un client Python qui peut se connecter et afficher les informations des parties en cours ou finies. C'est une bonne base pour les ligues qui souhaiteraient automatiser la vérification de résultats. Le script peut également récupérer les fichiers décrivant les tours déjà écoulés.

Ce sont précisément ces fichiers que je suspecte de contenir des informations non présentées au joueur ; malheureusement il s'agit d'un format binaire et non-documenté :

```
0000h: 06 00 00 00 88 31 33 32 31 38 33 34 09 6D 75 6C  ....ˆ1321834.mul
0010h: 74 69 74 75 72 6E 09 63 61 66 66 69 6E 61 74 6F  titurn.caffinato
0020h: 72 09 32 09 34 09 30 09 4D 61 74 63 68 6D 61 64  r.2.4.0.Matchmad
0030h: 65 09 30 09 31 09 30 09 53 6E 69 70 65 72 5A 77  e.0.1.0.SniperZw
0040h: 6F 6C 66 09 30 09 2D 31 09 30 20 31 09 35 32 20  olf.0.-1.0 1.52
0050h: 36 32 09 36 31 30 20 32 34 38 09 34 32 37 36 09  62.610 248.4276.
0060h: 31 38 38 09 31 38 09 31 32 39 09 63 61 66 66 69  188.18.129.caffi
0070h: 6E 61 74 6F 72 09 53 6E 69 70 65 72 5A 77 6F 6C  nator.SniperZwol
0080h: 66 09 2D 34 30 2E 30 09 30 09 30 09 30 00 00 00  f.-40.0.0.0.0...
0090h: 00 08 00 00 00 00 00 00 00 0D 0A 6D 61 63 68 69  ...........machi
00A0h: 6E 65 47 75 6E 00 01 00 00 00 01 00 00 00 00 FF  neGun..........ÿ
00B0h: FF FF FF 00 00 80 BF 03 00 00 00 00 00 00 00 00  ÿÿÿ..€¿........
00C0h: 00 00 00 40 AB 19 44 D0 44 CE 41 01 00 00 00 00  ...@«.DÐDÎA.....
00D0h: 00 80 BF 00 00 00 00 01 00 00 00 2E 00 00 00 00  .€¿.............
00E0h: 00 00 00 01 00 00 00 40 AB 19 44 D0 44 CE 41 5B  .......@«.DÐDÎA[
```

```
00F0h: 67 16 44 E2 23 6E 42 00 00 00 00 00 00 00 00 00  g.Dâ#nB.........
0100h: 00 00 00 5B 67 16 44 E2 23 6E 42 01 00 00 00 00  ...[g.Dâ#nB.....
0110h: 00 80 BF 00 00 00 00 02 00 00 00 39 00 00 00 69  .€¿........9...i
0120h: CE 9F C2 9C DE 7B C0 00 39 00 00 00 37 DC 80 C2  ÎŸÂœÞ{À.9...7Ü€Â
0130h: 34 AF 3D 42 00 00 00 00 00 0E 72 6F 63 6B 65 74  4¯=B......rocket
0140h: 4C 61 75 6E 63 68 65 72 00 02 00 00 00 01 00 00  Launcher........
0150h: 00 00 FF FF FF FF 00 00 80 BF 05 00 00 00 00 00  ..ÿÿÿÿ..€¿......
0160h: 00 00 00 00 00 00 7E 5F 2D 44 DB 39 B7 43 01 00  ......~_-DÛ9·C..
0170h: 00 00 00 00 80 BF 00 00 00 00 03 00 00 00 2E 00  ....€¿..........
0180h: 00 00 00 00 00 00 27 00 00 00 4B 01 E7 43 FD C2  ......'...K.çCýÂ
0190h: A0 43 05 00 00 00 01 00 00 00 7E 5F 2D 44 DB 39   C........~_-DÛ9
```

La familiarité avec le jeu permet de reconnaitre des éléments à gauche et à droite. Identifiant de la partie, nom des joueurs et de quelques unités... On voit assez rapidement que le fichier est découpé en deux parties : un header relativement lisible avec du texte séparé par des tabulations, puis des données pour le moment illisibles.

Je commence par le header : le tout premier champ (6) est vraisemblablement un numéro de version, et l'octet suivant (0x88) semble contenir la taille du header. Pour identifier le reste des valeurs lorsqu'elles ne sont pas évidentes, il faut mettre les mains dans le cambouis. Je récupère les fichiers d'un grand nombre de parties et fais un grand tableau sur papier pour essayer de deviner ce qui correspond à quoi. Je finis par retrouver les plus importants (à quel tour en est-on, le nom des joueurs, leur classement, niveau et historique de victoires/défaites, etc.). Là aussi, un script python est plus parlant qu'un millier de mots. Le tableau suivant décrit la majorité des champs :

Index du champ	Description
0	L'identifiant de la partie.
1	?
2	Nom de l'adversaire.
3	Quel "côté" de la carte appartient au joueur.
4	Le tour actuel.
5	Si le joueur a validé son tour.
6	La description de la partie.
7	Si on se trouve à la phase de "pari".
8	Si la partie est terminée.
9	Si on est en train d'assister à une partie.
10	Le nom du joueur qui a été "remplacé" lorsqu'on assiste à une partie.
11	Une estimation de qui va gagner basée sur le ELO des joueurs et l'historique de leurs parties.
13	L'historique de victoires/défaites entre les deux joueurs. Les deux valeurs sont séparées par un espace.
14	L'historique de victoires/défaites global du premier joueur.
15	L'historique de victoires/défaites global du second joueur.
16	Le classement du premier joueur.
17	Le classement du second joueur.
18	Le niveau du premier joueur.
19	Le niveau du second joueur.
20	Le nom du premier joueur.
21	Le nom du second joueur.

22	Le score de la partie.
23	Si la partie utilise des tours chronométrés.
24	La durée des tours s'ils sont chronométrés.
25	Si l'adversaire a validé son tour.

La seconde partie du fichier est autrement plus difficile. Les chaînes qui traînent (machineGun, rocketLauncher) laissent penser qu'il s'agit d'informations sur les unités. Je tâtonne un long moment mais à l'issue mais échoue à faire des progrès significatifs : même si je trouve la position de départ et l'identifiant du propriétaire, impossible de comprendre comment les waypoints (le chemin suivi par chaque unité sur la carte) sont structurés.

Première approche

Il est temps de remonter les manches : je lance le jeu sous débugger, mets un breakpoint sur les primitives réseau comme recv, et tente de retrouver la fonction qui interprète les données reçues. Malheureusement, je me retrouve rapidement à boucler dans un switch-case géant. La page Wikipedia de Frozen Synapse confirme mes suspicions : le jeu utilise un moteur appelé Torque Game Engine et celui-ci utilise son propre langage de script. Je suis en train de reverser l'interpréteur. Un petit tour dans le répertoire d'installation révèle une quantité importante de fichiers ".cs.dso", qui contiennent le bytecode généré par leur langage. En toute vraisemblance, ce sont eux qui contiennent la logique qui m'intéresse.

Seconde approche

Reverser une VM comme ça, à froid, est loin d'être chose facile. Là où j'ai de la chance, c'est que Torque Engine 2D a été rendu open-source en 2013. Après pas mal de lecture, j'identifie les points d'intérêt suivant dans le projet :

- L'endroit où sont lus les headers des fichiers DSO.
- La fonction CodeBlock::exec, chargée d'évaluer le bytecode.
- Le fichier astNodes.cc et CodeBlock::compile, qui compilent les scripts en bytecode.
- La page Wiki décrivant la syntaxe du langage.

Pour résumer rapidement, la VM du Torque Game Engine ne possède aucun registre. Elle utilise trois stacks (piles en bon français) pour effectuer l'ensemble de ses opérations : une pour les chaines de caractères, une pour les entiers et une pour les flottants. Par exemple, pour effectuer une addition, les deux opérandes sont poussées sur la pile des flottants, puis l'opcode OP_ADD indique à la VM que ces deux valeurs doivent être dépilées et que le résultat doit être poussé à la place.

La structure des fichiers DSO n'est pas très complexe et reflète ce mode de fonctionnement : le header contient un numéro de version, des tables de valeurs immédiates (chaines, entiers et flottants) utilisées dans le script, puis les instructions à proprement parler. Voici la structure détaillée en partant de l'offset 0 :

Nom	Taille du champ (octets)	Description
version	4	La version de la VM qui a généré le bytecode. Celui-ci ne peut-être exécuté qu'en cas de correspondance exacte (pas de rétro-compatibilité).
gst_size	4	Taille de la table de chaînes de caractères globale.

global_string_table	*gst_size*	La table de chaînes de caractères globale. L'ensemble des chaînes utilisées dans le script se trouvent juxtaposées ici, séparées par des octets nuls. Elles sont désignées via un offset dans cette "table". Par exemple, si l'opcode `OP_LOADIMMED_STR` est suivi d'un entier X, alors il fait référence à la chaîne qui commence à l'offset 8+X et qui s'arrête au premier octet nul rencontré.
fst_size	4	Taille de la table des chaînes de caractères utilisées dans les fonctions.
function_string_table	*fst_size*	La table de chaînes de caractères des fonctions. Elle fonctionne de manière identique à la précédente, mais elle contient les chaînes qui sont utilisées dans le scope des fonctions (i.e. si on est dans une fonction et qu'une chaîne est référencée, c'est dans cette table qu'il faut aller la chercher et non pas dans la précédente). Je n'ai pas vraiment compris pourquoi ils ont pris la peine de les séparer en deux tables.
gft_size	4	Taille de la table de flottants globale.
global_float_table	8 * *gft_size*	Un tableau de `float`s utilisés dans le script. On y accède via des index dans ce tableau en partant de zéro (2 ferait référence au 3ème flottant lu).
fft_size	4	Taille de la table des flottants utilisés dans les fonctions.
function_float_table	8 * *fft_size*	La table de flottants des fonctions. Comme pour les chaînes de caractères, les flottants utilisés dans les fonctions sont placés dans une table à part.
code_size	4	Taille du bytecode en opcodes.
linebreak_count	4	Nombre de breakpoints.
Code	?	Le flux d'opcodes qui constitue le script. La taille de celui-ci n'est pas prévisible, car chaque valeur lue peut avoir une taille de un ou cinq octets. Les valeurs comprises entre 0 et 254 sont codées sur un seul, mais les autres sont représentées par `0xFF` suivi de la valeur codée sur 4 octets. Les opcodes sont référencés par offsets directs en partant de 0 dans ce tableau : l'opcode qui sera exécuté après un saut dont la destination est X est `code[X]`.
linebreak_pairs	2 * 4 * *linebreak_count*	Des informations relatives aux breakpoints. Au nom, on devine qu'il s'agit d'un offset dans le flux d'opcodes où breaker et d'un numéro de ligne associé dans le script original, mais pour être honnête je n'ai pas creusé plus que ça.
it_size	4	Nombre d'entrées dans l'*IdentTable*.
ident_table	?	Sans doute à cause de la manière dont le bytecode est généré par Torque, les références vers les chaînes de caractères ne sont pas présentes dans le flux d'opcodes (un offset nul est laissé partout). Cette table permet de patcher le flux d'opcodes pour y

remettre les bons offsets. Chaque entrée a la structure suivante :

Nom	**Taille du champ** (octets)	**Description**
real_offset	4	Offset correct à insérer.
Count	4	Nombre d'emplacements à patcher.
Locations	4 * *count*	Une liste d'offsets dans le flux d'opcodes où la valeur (0) soit être remplacée par *real_offset*.

Le script suivant permet de parser entièrement un fichier DSO.

Ce qu'il faut retenir, c'est qu'à l'instar des fichiers Java compilés, les noms de fonctions et de variables sont toujours présents dans le fichier. Il est donc possible de regénérer un code quasiment identique à l'original !

Ecriture d'un décompilateur

Maintenant que l'ensemble des informations contenues dans le fichier DSO a été récupéré, je peux commencer l'écriture d'un décompilateur. Pour m'aider dans la tâche, j'utilise la version open-source du Torque Game Engine pour générer des scripts de complexité croissante et mets des breakpoints dans leur VM afin de regarder exactement comment elle interprète chaque opcode. A chaque fois que j'en rencontre un nouveau, je l'implémente en Python. En pratique, j'écris un émulateur pour leur langage de script qui exécute le bytecode, ce qui me permet de suivre l'état des différentes variables de la VM (en particulier, ses piles).

En premier lieu, je m'occupe des opérations arithmétiques (OP_ADD, OP_SUB, etc.). Les deux valeurs au sommet de la pile des flottants sont leurs arguments. Par exemple, pour effectuer une addition, le bytecode suivant est généré :

```
OP_LOADIMMED_FLT [offset dans la table des flottants]
OP_LOADIMMED_FLT [offset dans la table des flottants]
OP_ADD
```

Les deux premiers opcodes chargent des valeurs immédiates sur la pile des flottants, puis OP_ADD les dépile puis pousse leur somme à la place. En ce qui concerne mon décompilateur, je sauvegarde l'opération effectuée et non son résultat (par exemple, au lieu de pousser "5" sur la pile des flottants, je pousserais la chaîne de caractères "2 + 3"). Mécaniquement, le code source réapparait.

Le principe est identique pour les comparaisons (OP_CMPEQ, etc.), au détail près que c'est la pile des entiers qui est utilisée pour celles-ci.

La VM Torque n'effectue aucune optimisation : il n'y a aucun piège.

Passons aux affectations :

```
OP_LOADIMMED_STR [offset dans la table des chaînes]
OP_SETCURVAR [offset dans la table des chaînes]
OP_SAVEVAR_STR
```

Deux chaînes sont utilisées dans cet exemple d'affectation de string : la première est la valeur assignée, qui est mise sur la pile idoine par le premier opcode. La seconde est le nom de la variable de destination : l'émulateur doit donc garder en mémoire quelle est la variable courante. Enfin, l'exécution du dernier opcode récupère la valeur au sommet de la pile et l'affecte à la variable courante.

Ici, il est trivial de ré-écrire le code original : il suffit de récupérer les deux chaines et de mettre un signe "égal" entre les deux.

Vous avez compris le principe : je vais donc sauter directement à la partie la plus compliquée : les structures de contrôle (if/then/else, for/while) et déclarations de fonctions. Elles m'obligent à m'éloigner du comportement de la VM originale.

En effet, à la fin d'un "if", la VM continue d'exécuter normalement les instructions comme si de rien n'était - moi, je dois refermer l'accolade. Lorsque l'opcode OP_RETURN est rencontré, la VM sait qu'elle doit sortir de la fonction, mais en ce qui me concerne, il peut y avoir plusieurs points de sortie et je ne peux pas me baser sur eux pour déterminer si elle s'arrête ici ou non.

Ma solution est donc d'insérer des opcodes "maison" (type META_ENDFUNC ou META_ENDWHILE) aux endroits pertinents pour me "souvenir" qu'il s'agit d'une fin de structure (au sens large). Le problème lorsqu'on rajoute des opcodes, c'est que les sauts absolus se désynchronisent. J'ai fait la supposition, il me semble juste, qu'il n'est pas possible de faire des sauts qui sortent de la structure de contrôle : mes opcodes n'étant ajoutés qu'en fin de structure, toutes les adresses qui se trouvent avant demeurent inchangées. S'il y avait un mot clef goto dans le langage de script, cette supposition deviendrait vraisemblablement fausse.

La seconde difficulté est de parvenir à identifier à quel genre de structure on a affaire lorsqu'on rencontre un saut conditionnel. La solution pour celà est de regarder quel est l'opcode qui se trouve juste avant la destination du saut. J'ai abouti à la table suivante :

Opcode précédant la destination du saut	Type de structure
OP_JMP	Le bytecode ressemble à ceci : `[test dont le résultat est poussé sur la pile]` `*  OP_JMPIFNOT` `|  [branche "True", exécutée si la condition est remplie]` `|  OP_JMP  ----------,` ``-> [branche "False"] |` `[suite du code] <--'`

	Il s'agit donc d'un *if-then-else*.
OP_JMP	Un cas particulier du précédent : si l'opcode qui précède OP_JMP est un `OP_LOAD*`, on est peut-être dans le cas de l'opérateur ternaire (`a ? b : c`). Il faut alors émuler le code de la branche pour voir l'état des piles à l'issue - on ne saura qu'à ce moment-là.

OP_JMPIFNOT ou OP_JMPIFFNOT	Une boucle *for* ou *while* :
	`[test de sortie de boucle]`
	`* OP_JMPIFNOT`
	`| ,-> [instructions de la boucle]`
	`| | [test de sortie de boucle]`
	`| * OP_JMPIFNOT`
	`` `---> [suite du code] ``

N'importe quoi d'autre	On est alors dans le cas d'un *if* simple, sans *else* :
	`[test]`
	`* OP_JMPIFNOT`
	`| [branche "True", exécutée si la condition est remplie]`
	`` `-> [suite du code] ``

Quelques subtilités ont été passées sous silence, mais j'ai fini par aboutir à un décompilateur permettant de ré-écrire le code original quasiment à l'identique. Les curieux pourront lire ce script pour en savoir plus.

Le code n'est pas particulièrement propre ni élégant, mais il fonctionne.

Décompilation de Frozen Synapse

Armé d'un décompilateur fonctionnel, je peux donc aller lire le code du jeu pour trouver les informations qui m'intéressent. Victoire ? Pas si vite. Mes scripts échouent complètement à générer les sources du jeu. Le problème est identifié rapidement : les fichiers DSO de Frozen Synapse ont été générés vers 2011, plusieurs versions du moteur auparavant. Le (vieux) bytecode du jeu n'est pas compatible avec les versions actuelles.

Mais il ne peut pas tant avoir changé que ça, n'est-ce pas ? Armé de tout mon courage, je lance IDA. Astuce : je choisis de reverser la version Linux du jeu, en espérant que les développeurs aient oublié de stripper les symboles et je suis récompensé. Cela me permet de retrouver rapidement mes marques dans le moteur de jeu compilé. Je me concentre sur les différences avec la version open-source et identifie deux nouveautés majeures :

- La liste des opcodes s'est décalée à deux endroits en raison d'ajouts.
- Les offsets dans la table des chaînes sont donnés sur 4 octets dans l'ancienne version, et sur 8 octets dans la nouvelle (des commentaires dans le code laissent penser que c'est lié à une version 64 bits du moteur).

Qu'à cela ne tienne, je corrige mon décompilateur pour qu'il s'adapte à la version du bytecode - beaucoup de souffrances résumées en une phrase.

Au final, je parviens à récupérer les sources du jeu. Mieux : le code étant 99% fidèle à l'original, je peux même l'utiliser à la place du bytecode compilé (la VM Torque recrée le bytecode à la volée lorsqu'on lui donne des scripts).

Modification du jeu

Outre la possibilité de regarder la logique du jeu, je suis en mesure de modifier très proprement le comportement du jeu. Il me suffit d'ouvrir un script avec le bloc-notes, d'écrire quelques lignes de code, et le tour est joué !

Pour mémoire, le but de l'opération était de voir s'il n'était pas possible de découvrir la position des unités ennemies.

Plutôt que de mettre en place un proxy pour intercepter le traffic réseau puis parser les données reçues, pourquoi ne pas tout simplement désactiver le brouillard de guerre dans le client du jeu ? Rien de plus facile : voici `psychoff/gameScripts/mtInGame.cs` 2.0 :

```
function loadMTStage4()
{
    if ($manager.visMode != 0.0)
    {
        error("****** Forcing \"Light\" visibility...");
        $manager.visMode = 0;
       $gmPrefix = "\"Light\" ";
       MatchInfoVisGraphic.setBitmap("common/gui/images/15black");
    }
    else
    {
        $gmPrefix = "Light";
        MatchInfoVisGraphic.setBitmap("common/gui/images/15white");
    }
   // ...
}
```

Le résultat dans toute sa gloire :

Conclusion

La morale est que dans une architecture client-serveur, toute information qui ne doit pas être connue du client ne devrait tout simplement pas lui être envoyée.

A l'heure où j'écris ces lignes, Frozen Synapse 2 vient d'être annoncé, et je suis curieux de découvrir si Mode7 Games réutilisera le même moteur et si ces travaux pourront être réutilisés sur la prochaine version !