リバースエンジニアリングの実践そのいち　～Hover!バグ修正～

SET/K07 文月師走

1. リバースエンジニアリングとは？

> リバースエンジニアリング(Reverse engineering)とは、機械を分解したり、製品の動作を観察したり、ソフトウェアを解析するなどして、製品の構造を分析し、そこから製造方法や動作原理、設計図、ソースコードなどを調査する事である。

出展：Wikipedia

ソフトウェアだと、「設計してコーディングしてコンパイル」じゃなく、「逆コンパイルしてコードを類推して原理や動作を推測」という具合になります。

ここまでなら普通に技術的ですが、実例は結構怪しい内容も多いです。

著作権侵害回避のため、ソースから仕様を作り、別チームで再開発を行う

こういう手段があるらしいです。

組み込むと組み込んだ側も同じライセンスになる、つまり感染するライセンスへの対策でこれをやる場合もあるとか無いとか。

あるデバイスを別 OS で使うためデバイスドライバの機能を解析する

これである種の盗み見が発覚した事例もあるとか無いとか。

勝手にバグを見つけて修正

今回の主題。

勝手に海外ソフトを日本語化する

著作権的にアレなところがなくもないですね。まぁ、よく見かけます。

ゲームのパラメータとかいじっちゃおう

いやまぁ、かなーり黒いですが、結構あります。これは自分も割とよく…

シェアウェアや有料製品のプロテクト外しちゃおう

マックロケ。

そういう側面もある技術です。

2. 目標設定

やった後にこれ書いているので、目標の再確認です。

Win95 の頃の Microsoft の HOVER!は最近の PC だと BGM が出ない、これを直す。

ま、それだけです。

それだけってわりには遠回りなのが厄介ですが、それが面白くもあります。

3. 実践

手順について

今回は

逆アセンブル→問題個所特定→回避手段作成→テスト

という流れになります。

逆アセンブルは、既に存在する実行ファイルをアセンブリ言語へ変換する作業です。

解析は、変換されたアセンブリ言語のコードを読んだり実行してみたりして意味を理解する作業です。

回避手段はこの場合アセンブリ言語のソースに変更を加えたりしてそれを実行ファイルに書き戻す作業です。

テストはそのまま、成功したかの確認です。

前準備

必要なものを揃えます。とりあえず OllyDbg v1.10 日本語化パッチ済みを用意しました。

> **入手元：**
> OllyDbg 本体：OllyDbg
> http://www.ollydbg.de/
> 日本語化パッチ：Digital Travesia ～ でじたる とらべしあ ～
> http://hp.vector.co.jp/authors/VA028184/

以下の説明でもこれを使います。

逆アセンブル

最初の説明だと逆コンパイルですが、これは Win32 アプリケーション(普通の実行ファイル)だと難しいうえに誤差が多い技術であまり実用的ではありません。

とはいえ機械語だと…少なくとも私は読めません。

というわけで、機械語を人が読みやすい形で表現したとかいうアセンブリ言語へ変換することになります。

ここから既に人によってやり方が違いますが、自分の場合 OllyDbg を使います。

OllyDbg はその名のとおりデバッガで、逆アセンブラではありません。

が、デバッグ対象についての逆アセンブルを表示できるので代用できます。

「表示する機能がある」だけですので、手順としては

1. OllyDbg を起動して、HOVER.EXE を読み込む

のみで完了です。

通常、読み込むと自動的にプログラムを解析が始まります。

簡単なループ構造や分岐構造、外部関数(要するにほか DLL などの関数。API。)、ごく一部の関数の検出、あたりを自動でやってくれます。

プログラム作成時にインポートしたと思われる.LIB ファイル等があれば、同じアセンブルをもつ関数を検出することもできます。

手順は

1. CPU ウィンドウの逆アセンブルペインをアクティブにする。
2. ホットキー「Ctrl+O」。右クリックメニューからも出せます。
3. (未登録の場合のみ)オブジェクトファイルを追加ボタンを使って登録します。
4. 「スキャン<グループ番号>」ボタン
5. 終わったらスキャンを閉じて、念のため「Ctrl+A」で再解析します。

とりあえず、VC++6.0 付属のライブラリではほとんど一致しませんでした。

うまく一致すると多い時は 120 ぐらい発見できるので、コンパイラは VC++6.0 ではないと予想できます。

問題個所特定

問題個所を発見する行程です。一番重要ですねん。長いので作業の経過を書いてすませます。

1 原因推定のためのアプローチ

1.1 UI からたどる、編

1.1.1 ダイアログボックスの淡色化処理

理由：BGM が再生できない場合、BGM 設定が淡色化されるため。

結果：失敗。

原因：関係しそうな淡色化操作が見つからなかった。

1.1.2 ダイアログボックスの値設定

結果：失敗。

原因：ダイアログがものすごく遠回りに操作されていた。

成果：表示処理から辿るのは困難だとわかった。

1.2 設定の保存値からたどる、編

1.2.1 サウンド設定のレジストリ値

結果：収穫あり。

成果：4 つの設定値があり、それぞれのメモリ上でのアドレスを特定できた。

BGM に関連するフラグから API の遅延ロード処理が発見できた。

1.2.2 遅延ロード API

結果：微妙

成果：BGM 一時停止、BGM 再開(兼 BGM 再生)、BGM 停止、が見つかった。

1.3 機能からたどる、編

1.3.1 インポート API

理由：今回は半ば偶然これが重要であると分かったため。

実際のところ割と定番の方法だが、今回は後回しにしていた。

結果：怪しいとこ発見。

成果：midi デバイスの情報取得 API、midiOutGetDevCapsA の呼び出し発見。

失敗するならこの処理が怪しいと踏んだ。

2　意味解析

怪しいと思った処理の中身を理解できる形で読み下す作業。

読み下した分は頭の中に覚えるだけなので、例文程明確に言語化する必要は無いです。

先に元コードと読み下しの例文を提示しておきます。

読み下し内容-生の逆アセンブル：

```
01 DWORD num=midiOutGetNumDevs();   0041D380 CALL NEAR DWORD PTR DS:
                                                [<&WINMM.midiOutGetNumDevs>]
    01 の切片                       0041D386 MOV DWORD PTR SS:[EBP-80],EAX
02 if(num==0){                      0041D389 CMP DWORD PTR SS:[EBP-80],0
    02 の切片                       0041D38D JNZ HOVER.0041D3B8
    何かの処理                      0041D393 MOV EAX,DWORD PTR SS:[EBP-E4]
    何かの処理                      0041D399 MOV DWORD PTR DS:[EAX+410],0
    何かの処理                      0041D3A3 MOV EAX,DWORD PTR SS:[EBP-E4]
    何かの処理                      0041D3A9 MOV DWORD PTR DS:[EAX+40C],0
03 }else{                           0041D3B3 JMP HOVER.0041D47A
    何かの処理                      0041D3B8 MOV EAX,DWORD PTR SS:[EBP-E4]
    何かの処理                      0041D3BE MOV DWORD PTR DS:[EAX+410],0
 A //long l=0       for(ココ;;){    0041D3C8 MOV DWORD PTR SS:[EBP-84],0
    A の切片                        0041D3D2 JMP HOVER.0041D3DD
 B //l++            for(;;ココ){    0041D3D7 INC DWORD PTR SS:[EBP-84]
 C //l<num          for(;ココ;){    0041D3DD MOV EAX,DWORD PTR SS:[EBP-80]
    C の切片                        0041D3E0 CMP DWORD PTR SS:[EBP-84],EAX
    C の切片                        0041D3E6 JNB HOVER.0041D452
04 for(long l=0;l<num;l++){//A～C
```

```
05 MIDIOUTCAPS moc;
 D //sizeof(moc)          第 3 引数      0041D3EC PUSH 34
 E //&moc                 第 2 引数      0041D3EE LEA EAX,DWORD PTR SS:[EBP-B8]
    E の切片                            0041D3F4 PUSH EAX
 F //l                    第 1 引数      0041D3F5 MOV EAX,DWORD PTR SS:[EBP-84]
    F の切片                            0041D3FB PUSH EAX
06 long ret=midiOutGetDevCapsA          0041D3FC CALL NEAR DWORD PTR DS:
                                                    [<&WINMM.midiOutGetDevCapsA>]
       (l,%moc,sizeof(&moc))//D～F
    06 の切片                           0041D402 MOV DWORD PTR SS:[EBP-BC],EAX
07 if(ret!=0) break;                    0041D408 CMP DWORD PTR SS:[EBP-BC],0
    07 の切片                           0041D40F JNZ HOVER.0041D44D
 G //moc.wTechnology==MOD_MAPPER        0041D415 MOV EAX,DWORD PTR SS:[EBP-90]
    G の切片                            0041D41B AND EAX,0FFFF
    G の切片                            0041D420 CMP EAX,5
    G の切片                            0041D423 JE HOVER.0041D43D
 H  //moc.wTechnology==MOD_FMSYNTH      0041D429 MOV EAX,DWORD PTR SS:[EBP-90]
    H の切片                            0041D42F AND EAX,0FFFF
    H の切片                            0041D434 CMP EAX,4
    H の切片                            0041D437 JNZ HOVER.0041D44D
08 if(
    (moc.wTechnology==MOD_MAPPER)||
    (moc.wTechnology==MOD_FMSYNTH)
   ){//G～H
    何かのフラグセット                  0041D43D MOV EAX,DWORD PTR SS:[EBP-E4]
    その続き                            0041D443 MOV DWORD PTR DS:[EAX+410],1
09 }
00 }                                    0041D44D JMP HOVER.0041D3D7
    次の処理                            0041D452 MOV EAX,DWORD PTR SS:[EBP-E4]
```

2.1 大雑把な把握

よくあるコンパイル例：	
if(判定){処理}後続	**for(前処理;判定;差分){処理}後続**
〈判定〉	〈前処理〉
Jcc 後続のアドレス	JMP 判定のアドレス
〈処理〉	〈差分〉
〈後続〉	〈判定〉
if(判定){処理 A}else{処理 B}後続	Jcc 後続のアドレス
〈判定〉	〈処理〉
Jcc 処理 B のアドレス;	JMP 差分のアドレス
〈処理 A〉	〈後続〉
JMP 後続のアドレス	**ループの定番**
〈処理 B〉	ループ部分を別の if 文構造で括る場合も多い。

〈後続〉	多条件判断　if((判定 A)&&(判定 B)){処理}の場合
while(判定){処理}後続	〈判定 A〉
〈判定〉	Jcc 後続のアドレス
Jcc 後続のアドレス	〈判定 B〉
〈処理〉	Jcc 後続のアドレス
JMP 判定のアドレス	〈処理〉
〈後続〉	〈後続〉
do{処理}while(判定);後続	多条件判断　if((判定 A)||(判定 B)){処理}の場合
〈処理〉	〈判定 A〉
〈判定〉	Jcc 処理のアドレス
Jcc 処理のアドレス	〈判定 B〉
〈後続〉	Jcc 後続のアドレス
	〈処理〉
	〈後続〉

まずは、API からの流れを Jcc(JNZ とか JE とか)と JMP の位置と飛び先を元に大雑把に解釈していきます。

Jcc は条件ジャンプ命令といって、そこより以前の演算結果や判定命令の結果の持つ情報によってジャンプするかどうかが変わる命令です。

JMP は無条件ジャンプ命令といって、ここにたどり着いた場合は強制的にジャンプします。

ここでは C/C++っぽい形へ変換していくので、まずは基本的な制御文がどう変化されるか把握しておきます。

これらは基本形で、特に for 文などは変則的な展開が行われる事も多いです。

構造が理解できれば良かったりするので「この範囲がループ」「ここは条件分岐」程度の把握で構いません。

それが終われば、API の動作を追っていきます。

2.2 API の動作の把握

まず、手元の SDK から関連構造体と定数を調べ、構造体のオフセットを調査します。

```
                #define MAXPNAMELEN      32
                typedef UINT        VERSION;
 サイズ 位置    typedef struct{
  0x02  0x00        WORD    wMid;                 /* manufacturer ID */
  0x02  0x02        WORD    wPid;                 /* product ID */
  0x04  0x04        VERSION vDriverVersion;       /* version of the driver */
  0x20  0x08        char    szPname[MAXPNAMELEN]; /* product name (NULL terminated string) */
  0x02  0x28        WORD    wTechnology;          /* type of device */
  0x02  0x2A        WORD    wVoices;              /* # of voices (internal synth only) */
  0x02  0x2C        WORD    wNotes;               /* max # of notes (internal synth only) */
  0x02  0x2E        WORD    wChannelMask;         /* channels used (internal synth only) */
  0x04  0x30        DWORD   dwSupport;            /* functionality supported by driver */
                } MIDIOUTCAPS;
                #define MOD_MIDIPORT    1  /* output port */
                #define MOD_SYNTH       2  /* generic internal synth */
                #define MOD_SQSYNTH     3  /* square wave internal synth */
                #define MOD_FMSYNTH     4  /* FM internal synth */
                #define MOD_MAPPER      5  /* MIDI mapper */
```

```
#define MOD_WAVETABLE   6  /* hardware wavetable synth */
#define MOD_SWSYNTH     7  /* software synth */
```

これは構造体の宣言などのある SDK のヘッダファイルを読めば分かります…が、注意が一つ。

構造体で宣言される変数の順序やサイズやコンパイル設定によって、「4 バイト変数は先頭から 4 の倍数の位置」といった規約があり、隙間が開く場合があります。

大抵は 4 バイト以上の変数でも 4 バイト区切りにあわせられ、API の構造体は余りが出ないよう作られる事が多いので、API の場合は特に気にしなくて構いません。

2.3 処理の理解

さて、これらを考慮して例文のように読み下していくと、調べていないフラグ建て処理以外は大体の流れが復元できます。

「デバイスのテクノロジ番号が「FM ハードウェアシンセ」「ハードウェアシンセ」の場合のみ何らかのフラグを立て、それ以外では無視をする動作」を全デバイスに実行している、と解釈できます。

つまり、ここで未知のテクノロジやただのマッパが使われていたら失敗すると考えられるわけです。

多分ここが正解だとあたりを付けたら検証します。

3 トレース

実際の動きを観察したり、推測内容を検証する段階になります。

べつに疑問点がなく、解析結果に確信が持てるなら必要ない作業ではありますが…

3.1 結果

前段階で認識しない未知の定数のデバイスしかその PC には実装されていなかった。

より新しい SDK(のコピペサイト)を参照することで、「ハードウェア DSL 対応シンセ」「ソフトウェアシンセ」であることがわかった。

つまり原因は上位互換のテクノロジを無視した実装と、下位互換性に乏しい API の設計に原因があったといえる。

回避手段作成

ここまでわかってしまえばあとは簡単です。いろいろ選択肢はあるが、二種類ほどやってみることにしました。

元の対応デバイスより新しい既知のデバイスをすべて許可する。

```
0041D415 MOVZX EAX,WORD PTR SS:[EBP-90]
0041D41C CMP EAX,3
0041D41F JLE SHORT HOVER.0041D3D7
0041D421 CMP EAX,8
0041D424 JL SHORT HOVER.0041D43D
0041D426 JMP SHORT HOVER.0041D3D7
```

4～7 なら今まで 4 と 5 のときだけの処理(たぶん有効フラグを付けている)を行う。

許可するテクノロジの種類を差し替える。

```
0041D434   CMP EAX,7
```

MOD_FMSYNTH の代わりに MOD_SWSYNTH で動作するようにする。

テスト

実際に動かしてみる。

チェックボックスの無効化も解けて、BGM もなるようになった。

と、いうわけで成功です。

お疲れ様でした。

4. 所感

本文中では書きませんでしたが「ジャンプ先が自分の次の命令(つまり無視される)」や、「2バイトで済むのに4バイト使われている命令」、「ジャンプ命令にジャンプする命令」、「API の不在を考慮したコードと実在しないと起動さえ阻害しうる内容の混在」など、無駄のある命令が散見されました。

その無駄のある命令も全体で共通するわけではなく、場所によってどちらが使われるかの特徴に差が有りました。

…ということは、コンパイルオプションやコンパイラが複数パターン入り乱れているという事になります。

幾つかはライブラリファイルと仮定しても高速性を考慮すべきゲームでそんな非効率なコードを混ぜ込んでいたら結構危ないです。

利益優先と規模の大きさによる原因のような気はしますが、大手が作るからといって安定した高品位のものになるとは限らない、と考えさせられる結果になりました。

5. まとめ

物凄く大雑把にやりましたが、この説明で理解できるならこれを読むまでも無く分かる気もするくらいに説明不足な内容になった気がします。

興味を引く程度の効果はあると思いたいです。

またリバースエンジニアリングの実践について書くと思いますので、そのときはもうちょっとましになるよう努力します。