進め！リバースエンジニアリングの道

第4回 手動で逆コンパイル

文●愛甲健二

逆コンパイルとは

一般的に、逆アセンブルとは「マシン語からアセンブラ命令へコードを変換すること」ですが、逆コンパイルはそれをさらに一歩進めて「マシン語から十分に可読性のある高級言語へ変換すること」を指します。JavaやNETなどは、バイトコードからソースコードへの復元が容易に行えるため、いくつかの逆コンパイラもすでに存在しますが、ネイティブのマシン語についてはやはり難しく、例えば、x86系のマシン語からC/C++のソースコードへ正確に変換する逆コンパイラといったものは、今のところありません。しかし、マシン語といえども人間が読めば当然理解できますので、人の手による手動での逆コンパイルは可能です。今回はsample.exeを使って、手動による逆コンパイルを行っていきましょう。

前回「sample.exeと全く同じプログラムをC言語で作成してください」という課題を出させていただきました（前回までの記事は付録DVD-ROMに収録していますので、必要に応じて参照してください）。C言語にすると20行程度エンコード処理の解析動で逆コンパイルを行う」というのはアセンブルコード解読の練習にもなりますので、ぜひとも面倒くさがらずにやってみてください。

ではsample.exeを解析していきましょう。

エンコード処理の解析

IDAPro※で出力されたアセンブルコードを元に解析していきますので、まずはsample.exeをIDAProで開いてください。

先頭から読み進めてもよいのですが、課題では、すでにencode.cppが記述されており、黒塗りされた4行のみを特定すればよいので、メインとなるエンコード処理のみを解析します。下記がsample.exeの該当部分です。

```
.text:0040109F sub_40109F proc near
.text:0040109F   push  edx
.text:004010A0   and   eax, 3Fh
.text:004010A3   mov   edx, offset aFkl
.text:004010A8   add   edx, eax
.text:004010AA   xor   eax, eax
.text:004010AC   mov   al, [edx]
.text:004010AE   pop   edx
.text:004010AF   retn
```

sub_40109Fが1文字に対するエンコード処理であり、C言語風に書くとreturn (aFkl[eax & 0x3F])となります。004010A0のandでeaxの値の下位6ビットのみを有効にしてeaxを64未満の値にし、aFklの配列に対応させます。IDAProで見るとわかりやすいですが、004010A3のaFklは「FKLBaCAcTUDgGHslxRyJzMhiNjOtPulvQwEpqSXVmWoYkZ0bdefnr1¥x00MessageBo」という途中に0x00を含む64バイトのデータ列のアドレスになっており、aFklにeaxを加算することで、このデータ列の中のいずれか1つの文字のアドレスを指します。その指した値を戻り値としてalにコピーし、関数を終了します。

sub_40109Fは1文字に対するエンコード処理ですが、これを文字列に対して行うのが、次のsub_4010B0です。内部でsub_40109Fを呼び出します。

```
.text:004010B0 sub_4010B0 proc near
.text:004010B0 arg_0 = dword ptr  8
.text:004010B0 arg_4 = dword ptr  0Ch
.text:004010B0   push  ebp
.text:004010B1   mov   ebp, esp
.text:004010B3   push  edx
.text:004010B4   push  ecx
.text:004010B5   mov   edx, [ebp+arg_0]
.text:004010B8   mov   ecx, [ebp+arg_4]
.text:004010BB loc_4010BB:
.text:004010BB   xor   eax, eax
.text:004010BD   dec   ecx
.text:004010BE   mov   al, [edx+ecx]
.text:004010C1   call  sub_40109F
.text:004010C6   mov   [edx+ecx], al
.text:004010C9   test  ecx, ecx
.text:004010CB   jnz   short loc_4010BB
```

※　無料で入手できるIDAProのバージョンが、4.9から5.0に上がっています。ジャンプ系の命令を適切に解釈し、逆アセンブルウィンドウにおいてもグラフィカルに分岐を表示する機能も加わっているので、ぜひ使ってみてください。http://www.hex-rays.com/idapro/idadownfreeware.htm

```
.text:004010CD   pop   ecx
.text:004010CE   pop   edx
.text:004010CF   leave
.text:004010D0   retn  8
```

sub_4010B0は文字列に対してエンコードを行う関数で、004010C1にて1文字エンコーダーのsub_40109Fを呼び出しています。関数としては2つの引数arg_0（＝文字列のアドレス）、arg_4（＝文字列のサイズ）を受け取ります。

実際にエンコードを行っている部分は004010BB～004010CBまでのループ処理で、arg_0の後ろから1文字ずつをalにコピーし、sub_40109Fを呼び出し、その戻り値をまたarg_0の同じ場所に格納しています。004010C9にて、文字列のサイズであるecxが0になるまでエンコード処理を行っているため、すべての文字に対してsub_40109Fを実行します。

以上から、課題となっていたencode.cppは以下のように書けます。

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    unsigned int i;
    char data[256];
    char t[] = "FKLBaCAcTUDgGHsIx"
      "RyJzMhiNj0tPulvQwEpqSXVmWo"
      "YkZ0bdefnr1""¥x00""MessageBo";
    if(argc != 2)
      sprintf(data, "HELLO");
    else
      sprintf(data, "%s", argv[1]);
    for(i=0; i < strlen(data); i++)
      data[i] = t[data[i] & 0x3F];
    printf("data=%s¥n", data);
    return 0;
}
```

ちなみに、このアルゴリズムは不可逆ですのでデコードできません。当たり前ですが1バイト値、256個のいずれかの値を示すデータを0x3FでAND演算して、64個のいずれかの値に置換するので、例えばencode.cppの配列tの先頭である「F」という値をデコードしようにも1100 0000（192）、1000 0000（128）、0100 0000（64）、0000 0000（0）の4つの値に戻る可能性を持っています。ただ、入力が表示可能文字だけであったり、テキストとして意味を成しているものならば、ある程度の推測はできるかと思います。

また、ここまで読み進めて気がついた方もいるかもしれませんが、アセンブルコードは、1命令ごとの意味を理解するのは簡単ですが、関数やループといった「まとまった処理」としての意味を理解するのは難しいのです。しかし、これが正確にできなければ解析のしようがありませんので、命令をひととおり覚えたら、次はまとまった処理を解読する訓練をしましょう。

全体の解析

今回の課題はすでにencode.cppが与えられていたので、sub_40109F、sub_4010B0の2つの関数を読むだけでよかったのですが、本来の解析業務では、右も左もわからない状況ですので、最初からアセンブルコードを読み進めていくしかありません。とはいっても、いちばん初めの取っ掛かりは特徴的な文字列であったり、callされてそうなAPIにブレイクポイントを仕掛けて、といった感じで少しずつ解析場所を狭めていき、ある程度のところまで来たら「これはどのような処理をする関数なのか?」を特定していきます。まぁsample.exeのような小さいプログラムは、最初から全部の関数を読んでいってもよいですが。

では、せっかくなのでエンコードには直接関係ない関数の方も読んでいきましょう。なお、誌面の関係上、アセンブルコードを載せられないので、以降はIDAProで閲覧しながら読み進めてください。

• sub_401000

まずsub_401000ですが、これは「esiからediへecxの値だけデータをコピーする関数」です。00401001でコピー先がalなので、1バイト単位でコピーされます。コードも短いので比較的簡単に解読できると思います。

```
void sub_401000(void)
{
    do{
        ecx--;
        edi[ecx] = esi[ecx];
    }while(ecx != 0);
}
```

• sub_40100C

続いてsub_40100Cですが、これは引数（arg_0）を1つ受け取る関数です。アドレス00401011で、arg_0はesiへ格納され、00401018でalへコピーされ、0040101Aでそのalが0か否かを評価されます。そして0でなければ、loc_401016へ戻ります。

つまり、引数arg_0にはおそらく文字列のようなものが格納され、そのarg_0の終端文字0x00が見つかるまでloc_401016へ戻る、とい

う処理だと推測できます。そしてloc_401016のループ処理の中で唯一関係のないレジスタecxが、00401017でインクリメントされており、0040101Eで関数の戻り値となるeaxへ格納されるため、終端文字0x00が見つかるまでの長さが関数の戻り値として返されます。

以上からsub_40100Cは、引数として受け取った文字列の「0x00を含めた」長さを返す関数だとわかります。また、1文字目は無視しているので、1文字目が0x00だった場合はスルーされます。

```
int sub_40100C(char *arg_0)
{
    ecx = 0; esi = arg_0;
    do{
        esi++; ecx++;
    }while(*esi != 0);
    return (ecx + 1);
}
```

• sub_401027

sub_401027も引数arg_0を1つ受け取って処理する関数ですが、今度は00401041以降にエラー処理らしきものがあります。eaxに-1を入れていますので、戻り値が-1になる場合があるということです。戻り値が-1になるから必ずしもエラー処理というわけではありませんが、一般的にプログラムを作成する際、関数の戻り値を負にするのはエラー処理である場合が多いので、確定はできませんがそう推測しましょう。

そしてloc_401041へ進むのは00401033からのjzであり、条件はalが0の場合です。つまり、引数arg_0の先頭から探していって終端文字0x00が見つかったらエラーとなるわけです。そして00401035でalと0x20を比較しています。0x20はスペースなので、終端文字が見つかる前にスペースが見つかったら、00401039と0040103Bでスペースの次の文字のアドレスを戻り値としてeaxへコピーし、関数を終了しています。

以上から、sub_401027は引数arg_0を先頭から検索し、スペースが見つかったらその次の文字のアドレスを戻り値として返し、もしスペースが見つからなければ-1を返す関数だとわかります。

```
int sub_401027(char *arg_0)
{
    esi = arg_0;
    do{
        esi++;
        if(*esi == 0)
            return -1;
    }while(*esi != 0x20);
    return (esi + 1);
}
```

• sub_40104B

sub_40104BはGetCommandLine、CommandLineToArgvWといったAPIを呼び出しています。これらはプログラム実行時にコマンドラインに入力された文字列を取得する関数で、ここで取得した文字列を0040106Fにてsub_401027へ渡しています。GetCommandLineWはUNICODE版で、GetCommandLineAはASCII版です。

最初に、CommandLineToArgvWを使ってプログラムへ渡された引数の数pNumArgsを取得して、その数が2だったならば、引数が渡されていると判断し、改めてGetCommandLineAを使いASCII文字としてコマンドライン文字列を取得します。

このGetCommandLineは、実行されたプログラム名も含めたコマンドライン文字列を取得しますので、sub_401027を呼び出してスペースを探索し、引数となる部分だけを取り出します。ただ、この実装だと実行プログラムパスの中にスペースが使われていたらアウトですが、そこは仕様ということでお許しください(汗)。

スペースがなければ-1を返し、スペースがあれば「プログラム実行時にコマンドラインから引数が渡された」と考えてloc_401083へジャンプします。loc_401083以降では、sub_40100Cを呼び出し、引数の長さを取得して、その引数のサイズ分だけarg_0の指すアドレスへデータをコピーしています。

```
int sub_40104B(char *arg_0)
{
    int pNumArgs;
    LPWSTR *lpszArgs = CommandLineToArgvW
        (GetCommandLineW(), &pNumArgs);
    if(pNumArgs != 2)
        return -1;
    esi = sub_401027(GetCommandLineA());
    if(esi == -1)
        return -1;
    ecx = sub_40100C(esi);
    edi = arg_0;
    sub_401000();
    return 0
}
```

エントリポイント

最後にエントリポイントの処理です。最初に256バイトの領域をスタックに確保して、sub_40104Bを呼び出して、プログラムへ渡された引数を確認します。もし引数がなければ

crackme.exe　パスワード比較箇所

```
00402858 PUSH crackme_.004071E4                    ; /String2 = "WizardBible"
0040285D LEA EDX,DWORD PTR SS:[ESP+10]             ; |
00402861 PUSH EDX                                  ; |String1
00402862 CALL DWORD PTR DS:[<&KERNEL32.lstrcmpA>]  ; \lstrcmpA
00402868 TEST EAX,EAX
```

「HELLO」という文字列をbuffへ格納します。

sub_4010B0はエンコードを行う関数なので、buffの中にあるデータ列をエンコードし再びbuffに入れ、それをMessageBoxAで表示してプログラム終了です。

```
void start(void)
{
    char buff[256];
    if(sub_40104B(buff) != 0){
        ecx = 6;
        esi = "HELLO";
        edi = buff;
        sub_401000();
    }
    sub_4010B0(buff,
        sub_40100C(buff) - 1);
    MessageBoxA(GetActiveWindow(),
        buff, "MessageBox", 0);
}
```

以上でsample.exeのすべてのコードを読み終えましたが、いかがだったでしょうか。基本的なアセンブラ命令を覚えてさえいれば、それほど難しくはなかったと思います。

逆コンパイルはとても地味で面倒くさい作業ですが、マシン語を読み続けていれば必ず答えが出るという点においては、努力が報われやすい技術だとも言えます。

コンピューター技術には時として、天才的な発想力や、高度な数学的知識や、的確に問題点を探す嗅覚などが必要になる場合もあります。

図1　crackme_ex.exeを実行。「ユーザー名が間違っています」と表示される

図2　ユーザー名「WizardBible」が判明。今度は「パスワードが間違っています」となる

しかし、ことリバースエンジニアリングに限ればそういったものよりも、地道に解読していく根気こそが最も重要なスキルかもしれません。

再びcrackme.exeへ

第2回目以降何も触っていなかったcrackme.exeですが、アセンブラも学習したことですし、再びcrackme.exeの解析に戻りましょう。第2回目の最後、ユーザー名とパスワードを要求するダイアログボックスが表示されたところで解析は止まっていますから、ここから始めましょう（図1）。もし内容を忘れている方がいましたら、付録DVD収録の第2回を参照してください。また、元々のcrackme.exeと、日付制限を回避したcrackme_ex.exeもDVDに収録しています。

では解析を始めます。 まずテキストボックスから入力されたテキストを得ているため、GetWindowText、GetDlgItemText、GetDlgItemInt辺りにブレイクポイントを仕掛けてプログラムを実行します。そしてテキストボックスに適当な文字列を入力して、OKボタンをクリックすると、GetWindowTextA関数で処理が止まるので、そのまま関数を抜けるとアドレス00402818へたどり着きます。

00402818がテキストボックスから入力されたテキストを得ている場所なので、ここから下に降りていくとパスワードと比較している箇所が見つかります（上掲の部分）。

String2が「WizardBible」で、String1がユーザー名として入力された文字列です。その2つをlstrcmpAで比較しています。以上から、ユーザー名は「WizardBible」だと考えて間違いなさそうです。試しにこのユーザー名を再度入力してみましょう（図2）。

図1では、ユーザー名が間違っているとのメッセージが表示されましたが、図2ではパスワードが間違っているとのメッセージに変わりました。つまり、ユーザー名は合っているというわけです。これでユーザー名は判明しました。次はパスワードです。

パスワードは何？

00402868以降の処理を読み進めていけばパスワードも判明しそうですが、今回はここまでとし、この続きは次回にて行いたいと思います。では、またお会いしましょう。