リバースエンジニアリングチャレンジ 2009 解答

■ LEVEL1：atoi が意味するものは？

level1.bin が何のタイプのファイルであるかを、file コマンドにより確認する。

```
-----
# file level1.bin
level1.bin: MS-DOS executable PE for MS Windows (console) Intel 80386 32-bit
-----
```

Windows 用の実行ファイル（PE フォーマット）であることが分かる。

IDAPro を使い、プログラムを解析する。

```
-----  level1.bin
.text:00401150 _main           proc near
.text:00401150
.text:00401150 argc            = dword ptr  8
.text:00401150 argv            = dword ptr  0Ch
.text:00401150 envp            = dword ptr  10h
.text:00401150
.text:00401150                 push    ebp
.text:00401151                 mov     ebp, esp
.text:00401153                 push    ebx
.text:00401154                 mov     ebx, [ebp+argv]
.text:00401157                 cmp     [ebp+argc], 5
.text:0040115B                 jge     short loc_401165
.text:0040115D                 mov     eax, 1
.text:00401162                 pop     ebx
.text:00401163                 pop     ebp
.text:00401164                 retn
.text:00401165
.text:00401165                 push    dword ptr [ebx+4] ; s
.text:00401168                 call    _atol
.text:0040116D                 pop     ecx
.text:0040116E                 cmp     eax, 65h
```

```
.text:00401171                 jz      short loc_40117B
.text:00401173                 mov     eax, 1
.text:00401178                 pop     ebx
.text:00401179                 pop     ebp
.text:0040117A                 retn
.text:0040117B
.text:0040117B                 push    dword ptr [ebx+8] ; s
.text:0040117E                 call    _atol
.text:00401183                 pop     ecx
.text:00401184                 cmp     eax, 5Ah
.text:00401187                 jz      short loc_401191
.text:00401189                 mov     eax, 1
.text:0040118E                 pop     ebx
.text:0040118F                 pop     ebp
.text:00401190                 retn
.text:00401191
.text:00401191                 push    dword ptr [ebx+0Ch] ; s
.text:00401194                 call    _atol
.text:00401199                 pop     ecx
.text:0040119A                 cmp     eax, 38h
.text:0040119D                 jz      short loc_4011A7
.text:0040119F                 mov     eax, 1
.text:004011A4                 pop     ebx
.text:004011A5                 pop     ebp
.text:004011A6                 retn
.text:004011A7
.text:004011A7                 push    dword ptr [ebx+10h] ; s
.text:004011AA                 call    _atol
.text:004011AF                 pop     ecx
.text:004011B0                 cmp     eax, 30h
.text:004011B3                 jz      short loc_4011BD
.text:004011B5                 mov     eax, 1
.text:004011BA                 pop     ebx
.text:004011BB                 pop     ebp
.text:004011BC                 retn
.text:004011BD
```

```
.text:004011BD                 push    dword ptr [ebx+10h] ; s
.text:004011C0                 call    _atol
.text:004011C5                 pop     ecx
.text:004011C6                 push    eax
.text:004011C7                 push    dword ptr [ebx+0Ch] ; s
.text:004011CA                 call    _atol
.text:004011CF                 pop     ecx
.text:004011D0                 push    eax
.text:004011D1                 push    dword ptr [ebx+8] ; s
.text:004011D4                 call    _atol
.text:004011D9                 pop     ecx
.text:004011DA                 push    eax
.text:004011DB                 push    dword ptr [ebx+4] ; s
.text:004011DE                 call    _atol
.text:004011E3                 pop     ecx
.text:004011E4                 push    eax
.text:004011E5                 push    offset format   ; "PASSWORD: %c%c%c%c¥n"
.text:004011EA                 call    _printf
.text:004011EF                 add     esp, 14h
.text:004011F2                 xor     eax, eax
.text:004011F4                 pop     ebx
.text:004011F5                 pop     ebp
.text:004011F6                 retn
.text:004011F6 _main           endp
-----
```

004011EA にて printf 関数が呼ばれており、その出力がパスワードとなっている。パスワードは%c を使い 4 文字であることが分かる。その 4 文字は、上にさかのぼり、スタックへ積まれる値を確認することで判明する。

004011BD 以降の処理にて、atol が 4 回呼び出されている。また、その戻り値 eax がそれぞれスタックへ格納されている。以上のことから、ebx+4、ebx+8、ebx+0Ch、ebx+10h それぞれのアドレスにある 1 文字ずつ（計 4 文字）がパスワードだと分かる。

では、「これらのアドレスにある文字は何か？」を、さらにさかのぼって調べる。

00401154 にて、ebx に argv が格納される。つまりプログラムに渡される引数の先頭アドレスが ebx に入る。続いて 00401157 にて、引数の数を判定する。

00401165 にて、プログラムに渡された第一引数をスタックへ積み、atol を呼ぶ。0040116E にて、atol の戻り値と 65h を比較、異なるならプログラムを終了し、イコールならば次の処理へ進む。

0040117B にて、プログラムに渡された第二引数をスタックへ積み、atol を呼ぶ。00401184 にて、atol の戻り値と 5Ah を比較、異なるならプログラムを終了し、イコールならば次の処理へ進む。

（以後、繰り返し）

結果、「65h、5Ah、38h、30h」の 4 バイトがスタックへ積まれた状態で printf が呼び出され、%c にて出力される。つまり「eZ80」が解答となる。

■ LEVEL2：このゲーム、クリアできる？

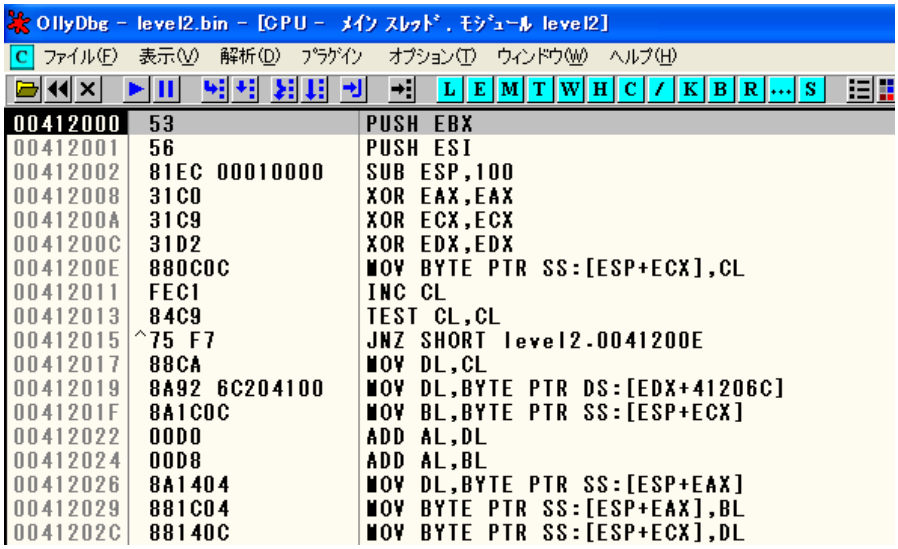
level2.bin が何のタイプのファイルであるかを、file コマンドにより確認する。

```
-----
# file level2.bin
level2.bin: MS-DOS executable PE for MS Windows (GUI) Intel 80386 32-bit
-----
```

Windows 用の実行ファイル（PE フォーマット）であることが分かる。

IDAPro で解析してもよいが、実行ファイルに暗号系の pack 処理がなされているため、まずは OllyDbg を使い、プログラムを解析する。

00412000 がエントリポイントとなる。



OllyDbg 上にて、復号処理が終わるまで処理を進める。復号処理は 00412067 で終わる。

```
OllyDbg - level2.bin - [CPU - メイン スレッド, モジュール level2]
00412050   3091 00104000    XOR BYTE PTR DS:[ECX+401000],DL
00412056   41               INC ECX
00412057   81F9 FB100000    CMP ECX,10FB
0041205D  ^75 D8            JNZ SHORT level2.00412037
0041205F   81C4 00010000    ADD ESP,100
00412065   5E               POP ESI
00412066   5B               POP EBX
00412067  -E9 0DFCFEFF      JMP level2.00401C79
0041206C  v E3 68           JECXZ SHORT level2.004120D6
0041206E   3E:3ABE FB59B23E CMP BH,BYTE PTR DS:[ESI+3BB259FB]
00412075   16               PUSH SS
00412076   91               XCHG EAX,ECX
00412077   A8 BE            TEST AL,0BE
00412079   9C               PUSHFD
0041207A   15 1E92BB24      ADC EAX,24BB921E
0041207F  -E9 8B09D161      JMP 62122A0F
00412084  -E9 D3035D80      JMP 809E245C
00412089   B1 BF            MOV CL,0BF
```

F8 で 1 命令進めると、本来のプログラムの処理へジャンプする。

```
OllyDbg - level2.bin - [CPU - メイン スレッド, モジュール level2]
00401C79   . E8 A6030000    CALL level2.00402024
00401C7E   .^E9 36FDFFFF    JMP level2.004019B9
00401C83   r> 55            PUSH EBP
```

OllyDbg では、Ctrl＋A で再解析をすると、逆アセンブルコードに変わる。unpack を行ったので、ここで一度 dump を取っておいてもよい。

```
OllyDbg - level2.bin - [CPU - メイン スレッド, モジュール level2]
004019B9   PUSH 5C
004019BB   PUSH level2.004032A8
004019C0   CALL level2.00401F7C
004019C5   XOR EBX,EBX
004019C7   MOV DWORD PTR SS:[EBP-1C],EBX
004019CA   MOV DWORD PTR SS:[EBP-4],EBX
004019CD   LEA EAX,DWORD PTR SS:[EBP-6C]
004019D0   PUSH EAX
004019D1   CALL DWORD PTR DS:[<&KERNEL32.GetStartupInfoW>]
004019D7   MOV DWORD PTR SS:[EBP-4],-2
004019DE   MOV DWORD PTR SS:[EBP-4],1
004019E5   MOV EAX,DWORD PTR FS:[18]
004019EB   MOV ESI,DWORD PTR DS:[EAX+4]
004019EE   MOV DWORD PTR SS:[EBP-20],EBX
004019F1   MOV EDI,level2.00404770
004019F6   PUSH EBX
004019F7   PUSH ESI
004019F8   PUSH EDI
```

続いてメイン処理の解析に移るが、まずは IsDebuggerPresent 関数を無効にする。

```
OllyDbg - level2.bin - [CPU - メイン スレッド, モジュール kernel32]
7C813093  33C0   XOR EAX,EAX
7C813095  C3     RETN
```

さらに、以下のような IsDebuggerPresent 関数の「変更監視」を無効にする。

```
OllyDbg - level2.bin - [CPU - メイン スレッド, モジュール level2]
00401540  > 8A90 34404000  MOV DL,BYTE PTR DS:[EAX+404034]
00401546  . 3A9401 344040  CMP DL,BYTE PTR DS:[ECX+EAX+404034]
0040154D  .v0F85 83000000  JNZ level2.004015D6
00401553  . 83C0 01        ADD EAX,1
00401556  . 83F8 0E        CMP EAX,0E
00401559  .^7C E5          JL SHORT level2.00401540
```

コードを NOP にしてもよいが、複数個所から呼ばれているため、データ部分の 00404034 以降の先頭 3 バイトを、「33h、C0h（xor eax, eax）、C3h（retn）」にする。

```
アドレス   Hexダンプ                                          ASCII
00404000  61 FB 3C 43 9E 04 C3 BC FF FF FF FF FF FF FF FF  a･C･ﾃｼ
00404010  FE FF FF FF 01 00 00 00 14 00 00 00 05 00 00 00  ････
00404020  2C 01 00 00 90 01 00 00 28 00 00 00 7C 01 00 00  ,･･･ ･･
00404030  01 00 00 00 33 C0 C3 00 00 00 8B 40 30 0F B6 40  ････3ﾀﾃ･
00404040  02 C3 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ･ﾃ･･････
00404050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ････････
00404060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ････････
00404070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ････････
```

これで、デバッガ上でゲームをプレイできる。

あとは、プレイヤーのテーブルが上昇しないようにバイナリを書き換えるだけだが、ここで重要なのは、どこに「テーブルの位置情報があるか？」を探すこと。テーブルは一定の速度で上昇しているため、左上を(x=0, y=0)の座標だとすると、上昇は y を減算していることになる。よって、一定の速度で減算されている値をメモリ上から探す。

結果、00404024 からの 4 バイトがテーブルの y 軸座標だと分かる。

```
ｱﾄﾞﾚｽ   Hexﾀﾞﾝﾌﾟ                                          ASCII
00404000 88 67 B7 01 77 98 48 FE FF FF FF FF FF FF FF FF  ･ｷ┌▼路･
00404010 FE FF FF FF 01 00 00 00 14 00 00 00 05 00 00 00  ････････
00404020 6C 01 00 00 88 00 00 00 71 02 00 00 E5 01 00 00  l┌･･･････
00404030 00 00 00 00 33 C0 C3 00 00 00 8B 40 30 0F B6 40  ････3ﾀﾃ･
00404040 02 C3 00 00 00 00 00 00 01 00 00 00 48 28 3D 00  ┐ﾃ......
00404050 E8 27 3D 00 00 00 00 00 00 00 00 00 00 00 00 00  ･=.....
00404060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........
00404070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........
ﾌﾟﾛｾｽが終了しました 終了ｺｰﾄﾞ：0
```

よって、00404024 を減算している処理をコードの中から探す。

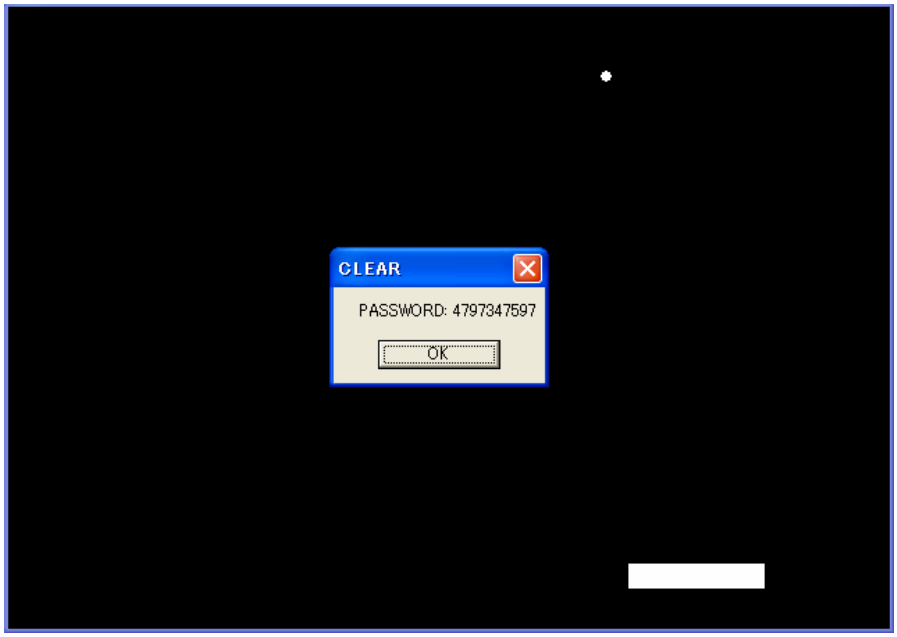
```
OllyDbg - level2.bin - [CPU - ﾒｲﾝ ｽﾚｯﾄﾞ, ﾓｼﾞｭｰﾙ level2]
C ﾌｧｲﾙ(F) 表示(V) 解析(D) ﾌﾟﾗｸﾞｲﾝ オプション(T) ウィンドウ(W) ヘルプ(H)
L E M T W H C / K B R ... S
004012AA  . B9 01000000    MOV ECX,1
004012AF  .v7C 10          JL SHORT level2.004012C1
004012B1  . 290D 24404000  SUB DWORD PTR DS:[404024],ECX
004012B7  . B8 05000000    MOV EAX,5
004012BC  . A3 54474000    MOV DWORD PTR DS:[404754],EAX
004012C1  > 3BDD           CMP EBX,EBP
```

004012B1 の命令を NOP に変更する。

```
OllyDbg - level2.bin - [CPU - ﾒｲﾝ ｽﾚｯﾄﾞ, ﾓｼﾞｭｰﾙ level2]
C ﾌｧｲﾙ(F) 表示(V) 解析(D) ﾌﾟﾗｸﾞｲﾝ オプション(T) ウィンドウ(W) ヘルプ(H)
L E M T W H C / K B R ... S
004012AA  . B9 01000000    MOV ECX,1
004012AF  .v7C 10          JL SHORT level2.004012C1
004012B1    90             NOP
004012B2    90             NOP
004012B3    90             NOP
004012B4    90             NOP
004012B5    90             NOP
004012B6    90             NOP
004012B7  . B8 05000000    MOV EAX,5
004012BC  . A3 54474000    MOV DWORD PTR DS:[404754],EAX
```

これでテーブルが上昇しなくなる。

あとはゲームをクリアすると、パスワードが得られる。



解答は「4797347597」となる。

なお、このテキストでは「ゲームを正常にプレイできるまで」を記述したが、解答パスワードを得るだけならば、他にも効率的な解答方法が存在する。

また、余談になるが、今回 level2.bin を作成する上で、書籍「Windows ゲームプログラミング 第 2 版　赤坂 玲音 (著)」を大いに参考にさせていただいた。それに伴いパスワードの 4797347597 は、この書籍の ISBN-10 番号となっている。

■ LEVEL3：7 つのエンコーダ

level3.bin が何のタイプのファイルであるかを、file コマンドにより確認する。

```
-----
# file level3.bin
level3.bin: Zip archive data, at least v2.0 to extract
-----
```

ZIP アーカイブであることが分かるため、unzip で展開する。

```
-----
# unzip level3.bin
Archive:  level3.bin
  inflating: password.e0.e1.e2.e3.e4.e5.e6
  inflating: encode3.bin
  inflating: encode5.bin
  inflating: encode1.bin
  inflating: encode2.bin
  inflating: encode4.bin
  inflating: encode6.bin
# ls
encode1.bin  encode3.bin  encode5.bin  password.e0.e1.e2.e3.e4.e5.e6
encode2.bin  encode4.bin  encode6.bin
-----
```

展開すると、複数のファイルが作成される。すべてのファイルに対して file コマンドで調べる。

```
-----
# file *
encode1.bin: DOS floppy 1440k, x86 hard disk boot sector
encode2.bin: MS-DOS executable PE for MS Windows (console) Intel 80386 32-bit
encode3.bin: ELF 32-bit LSB executable, ARM, version 1 (ARM), for GNU/Linux 2.4.1,
dynamically linked (uses shared libs), for GNU/Linux 2.4.1, stripped
```

```
encode4.bin: MS-DOS executable PE for MS Windows (console) Mono/.Net assembly
encode5.bin: MS-DOS executable PE for MS Windows (console) Intel 80386 32-bit
encode6.bin: MS-DOS executable PE for MS Windows (console) Intel 80386 32-bit
password.e0.e1.e2.e3.e4.e5.e6: data
```

file コマンドの結果から、password.e0.e1.e2.e3.e4.e5.e6 以外は何かしらの実行ファイルであることが分かる。よって、それらを逆アセンブルし、解析する。

□ encode6.bin 解析

encode6.bin は、PE フォーマットであるため、試しに拡張子を exe に変更する。すると実行ファイルが Perl と関係がありそうな「らくだ」のアイコンに変わる。Windows バイナリであること、Perl に関係しそうなこと、この 2 点から、strings で ActivePerl に関係する文字列を摘出する。

```
# strings encode6.bin | grep ActivePerl
ActivePerl::BUILD
ActivePerl::PRODUCT
ActivePerl::VERSION
ActivePerl
Usage: ActivePerl::PRODUCT()
Usage: ActivePerl::BUILD()
ActivePerl Build 819 [267479]
```

strings の結果から、ActivePerl で書かれたプログラムを exe ファイルに pack したと推測、また、PAR-Packer-0.973 という文字列がバイナリの中から見つかる。

```
# strings encode6.bin | grep PAR
（省略）
PAR->import(@par_args);
C:¥Temp¥PAR-Packer-0.973¥myldr¥par.pdb
PAR_PROGNAME
```

　以上のことから、PAR-Packer で pack された実行ファイルだと考え、unzip で展開し、実スクリプトを得る。

```
# unzip encode6.bin
Archive:  encode6.bin
   creating: lib/
   creating: script/
 (省略)
  inflating: lib/warnings/register.pm
  inflating: script/encode6.pl
  inflating: script/main.pl
```

　展開されたファイル script/encode6.pl が実スクリプト。

```
# cat script/encode6.pl
#!/usr/local/bin/perl

use Crypt::CBC;

$key1 = 'I LOVE PAR!';
$key2 = 'ACTIVE PERL';
$key3 = 'CRYPT::CBC';

open(R, "$ARGV[0]") or die "err";
open(W, ">$ARGV[1]") or die "err";

binmode R;
binmode W;

$cipher1 = Crypt::CBC->new($key1, 'DES');
$cipher2 = Crypt::CBC->new($key2, 'DES');
```

```
$cipher3 = Crypt::CBC->new($key3, 'DES');

$data = ""; $new = "";
while(read(R, $new, 1)){
        $data .= $new;
}

$data = $cipher1->encrypt($data);
$data = $cipher2->encrypt($data);
$data = $cipher3->encrypt($data);

print W ($data);

close W;
close R;
-----
```

encode6.bin はエンコーダになっているため、これに対応するデコーダを作成する。

```
-----  decode6.pl
#!/usr/bin/perl

use Crypt::CBC;

$key1 = 'I LOVE PAR!';
$key2 = 'ACTIVE PERL';
$key3 = 'CRYPT::CBC';

open(R, "$ARGV[0]") or die "err";
open(W, ">$ARGV[1]") or die "err";

binmode R;
binmode W;

$cipher1 = Crypt::CBC->new($key1, 'DES');
$cipher2 = Crypt::CBC->new($key2, 'DES');
```

```
$cipher3 = Crypt::CBC->new($key3, 'DES');

$data = ""; $new = "";
while(read(R, $new, 1)){
        $data .= $new;
}

$data = $cipher3->decrypt($data);
$data = $cipher2->decrypt($data);
$data = $cipher1->decrypt($data);

print W ($data);

close W;
close R;
-----
```

以上で、encode6.bin の解析は終了。

なお、今回使用した Perl スクリプトを EXE へパックするツール「PAR」は、以下の Web サイトから情報を得ることができる。

PAR
http://par.perl.org/wiki/Main_Page

□ encode5.bin 解析

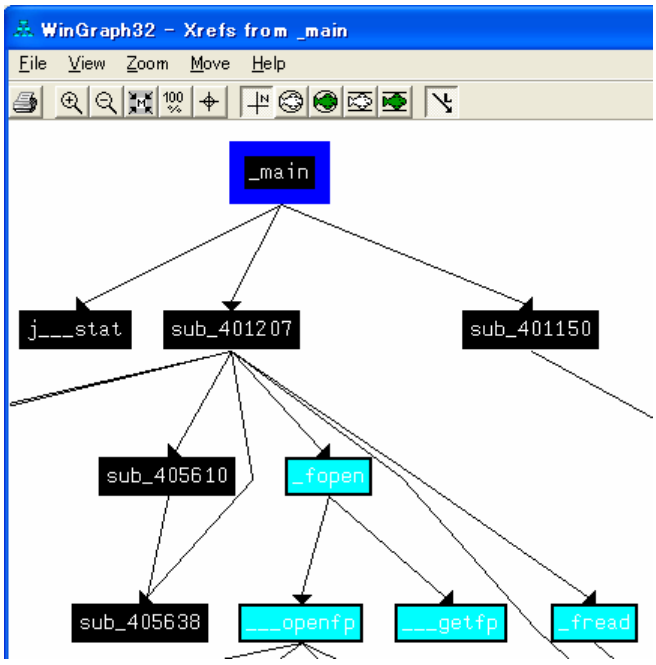
encode5.bin は x86 の Windows 実行ファイル。IDAPro や OllyDbg で解析することでアルゴリズムを特定できる。

```
-----  encode5.bin
.text:00401306                 push    ebp
.text:00401307                 mov     ebp, esp
.text:00401309                 add     esp, 0FFFFFF90h
.text:0040130C                 push    ebx
.text:0040130D                 mov     ebx, [ebp+argv]
```

```
.text:00401310                 cmp     [ebp+argc], 3
.text:00401314                 jz      short loc_40131B
.text:00401316                 or      eax, 0FFFFFFFFh
.text:00401319                 jmp     short loc_401392
.text:0040131B
.text:0040131B                 lea     edx, [ebp+statbuf]
.text:0040131E                 push    edx             ; statbuf
.text:0040131F                 push    dword ptr [ebx+4] ; path
.text:00401322                 call    j___stat
.text:00401327                 add     esp, 8
.text:0040132A                 test    eax, eax
.text:0040132C                 jz      short loc_401333
.text:0040132E                 or      eax, 0FFFFFFFFh
.text:00401331                 jmp     short loc_401392
-----
```

プログラムに渡される引数は 2 つ、そのひとつ目の引数が stat へ渡されているため、ひとつ目の引数にはファイル名を渡すのだと判断できる。ただし、main 関数内では、ファイルオープン系の関数が呼ばれていないため、より下位の関数を解析する必要がある。

関数の呼び出しマップを確認する。



右から順番に見ていく。

sub_401150 を読むと、RIFF や WAVE といった文字列が目に付く。よって、ここは wav ファイルのヘッダを作成する処理だと推測できる。

続いて sub_401207 を読む。読み込み用（rb）の fopen と書き込み用（wb）の fopen が呼ばれている。どうやら、この関数内でプログラムに渡されたファイルの read が行われているようだ。

sub_401207 は 3 つの引数をとるが、第一引数が読み込みファイル、第二引数が書き込みファイル、第三引数が 2Ch バイト分の書き込みデータとなっている。sub_401150 にて wav ファイルのヘッダが作成されていたため、そのヘッダ情報が第三引数に渡されると推測。あとは、読まれたデータがどのようにして書き込まれていくか、を調べれば、デコーダを作成できる。

```
-----  encode5.bin
.text:00401283                push    [ebp+stream]    ; stream
.text:00401286                push    1               ; n
.text:00401288                push    1               ; size
.text:0040128A                lea     edx, [ebp+ptr]
.text:0040128D                push    edx             ; ptr
.text:0040128E                call    _fread
.text:00401293                add     esp, 10h
.text:00401296                test    eax, eax
.text:00401298                jz      short loc_4012ED
.text:0040129A                xor     ebx, ebx
.text:0040129C                                        // [ebp+ptr] = read_1byte();
.text:0040129C                call    sub_405638      // eax = rand();
.text:004012A1                and     ax, 0FFC3h      // ax &= 0xFFC3;
.text:004012A5                xor     edx, edx        // edx = 0;
.text:004012A7                mov     dl, [ebp+ptr]   // dl = [ebp+ptr];
.text:004012AA                and     dx, 0Fh         // dx &= 0x0F;
.text:004012AE                shl     edx, 2          // edx <<= 2;
.text:004012B1                or      ax, dx          // ax = ax | dx;
.text:004012B4                mov     [edi], ax       // write_2byte(ax);
```

```
.text:004012B7                 call    sub_405638    // eax = rand();
.text:004012BC                 and     ax, 0C3FFh    // ax &= 0xC3FF;
.text:004012C0                 xor     ecx, ecx      // ecx = 0;
.text:004012C2                 mov     cl, [ebp+ptr] // cl = [ebp+ptr];
.text:004012C5                 and     cx, 0F0h      // cx &= 0xF0;
.text:004012CA                 shl     ecx, 6        // ecx <<= 6;
.text:004012CD                 or      ax, cx        // ax = ax | cx;
.text:004012D0                 mov     [edi+2], ax   // write_2byte(ax);
.text:004012D4                 push    esi             ; stream
.text:004012D5                 push    1               ; n
.text:004012D7                 push    4               ; size
.text:004012D9                 push    edi             ; ptr
.text:004012DA                 call    _fwrite
.text:004012DF                 add     esp, 10h
.text:004012E2                 inc     ebx              // ebx++;
.text:004012E3                 cmp     ebx, 100h        // if(ebx < 100h)
.text:004012E9                 jl      short loc_40129C //   goto loc_40129C;
-----
```

1 バイト読み、4 バイト書き込んでいる。sub_405638 は rand 関数。この処理を 1 バイトに対して 256 回繰り返している（つまり結果的に 1 バイトが 1024 バイトになる）。

以上の解析結果から、デコーダを作成する。

```
-----  decode5.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE *fp;
    char buff[1024];

    fp = fopen(argv[1], "rb");

    fread(buff, 0x2C, 1, fp);
    while(1024 == fread(buff, 1, 256 * 4, fp))
```

```
    {
        int c;
        unsigned short a, b;
        a = *((unsigned short *)buff);
        b = *((unsigned short *)(buff + 2));

        c = (int)(((a >> 2) & 0x000F) | ((b >> 6) & 0x00F0));
        fputc(c, stdout);
    }
    fclose(fp);
    return 0;
}
-----
```

以上で、encode5.bin の解析は終了。

□ encode4.bin 解析

encode4.bin は 64 ビット Windows の PE フォーマットバイナリ。64 ビット環境がなければ動作しないが、ndisasm.exe で逆アセンブルが可能。

```
-----
C:¥nasm-2.07>ndisasm.exe -b 64 -e 1024 encode4.bin > encode4.dis
-----
```

64 ビットモードで、先頭 1024 バイト（テキストセクション）から逆アセンブル。

```
-----  encode4.dis
00000000  48                rex.w
00000001  81EC78010000      sub esp,0x178
00000007  488B05F21F0000    mov rax,[rel 0x2000]
0000000E  4833C4            xor rax,rsp
00000011  4889842450010000  mov [rsp+0x150],rax
00000019  83F903            cmp ecx,byte +0x3     // if(argc >= 3)
0000001C  4889AC2498010000  mov [rsp+0x198],rbp
00000024  488BEA            mov rbp,rdx
```

```
00000027  7D0A              jnl 0x33              //     goto 0x33
00000029  B801000000        mov eax,0x1
0000002E  E9BF010000        jmp dword 0x1f2
00000033  48899C2490010000  mov [rsp+0x190],rbx
0000003B  33DB              xor ebx,ebx           // ebx = 0;
0000003D  488D4C2450        lea rcx,[rsp+0x50]
00000042  8BC3              mov eax,ebx           // eax = ebx; (eax = 0)
00000044  66666690          xchg ax,ax
00000048  66666690          xchg ax,ax
0000004C  66666690          xchg ax,ax
00000050  8801          +-> mov [rcx],al          // [ecx] = (al = 0..255);
00000052  83C001        |   add eax,byte +0x1     // eax++;
00000055  4883C101      |   add rcx,byte +0x1     // rcx++;
00000059  3D00010000    |   cmp eax,0x100         // if(eax < 0x100)
0000005E  7CF0          +-- jl 0x50               //     goto 0x50;
00000060  4889B42470010000  mov [rsp+0x170],rsi
00000068  33C9              xor ecx,ecx
0000006A  4889BC2468010000  mov [rsp+0x168],rdi
00000072  4C89A42460010000  mov [rsp+0x160],r12
0000007A  FF15A8100000      call qword [rel 0x1128] // time64
00000080  488BC8            mov rcx,rax
00000083  FF15A7100000      call qword [rel 0x1130] // srand
00000089  488BFB            mov rdi,rbx
0000008C  BE00010000        mov esi,0x100
00000091  4C8D2548250000    lea r12,[rel 0x25e0]
00000098  FF1582100000  +-> call qword [rel 0x1120] // rand
0000009E  99            |   cdq
0000009F  0FB6D2        |   movzx edx,dl
000000A2  03C2          |   add eax,edx
000000A4  0FB6C0        |   movzx eax,al
000000A7  2BC2          |   sub eax,edx
000000A9  42803C2701    |   cmp byte [rdi+r12],0x1
000000AE  7426          |   jz 0xd6
000000B0  4863D0        |   movsxd rdx,eax
000000B3  42803C2201    |   cmp byte [rdx+r12],0x1
000000B8  741C          |   jz 0xd6
```

```
000000BA  0FB6441450    |   movzx eax,byte [rsp+rdx+0x50]
000000BF  0FB64C3C50    |   movzx ecx,byte [rsp+rdi+0x50]
000000C4  42C6042701    |   mov byte [rdi+r12],0x1
000000C9  88443C50      |   mov [rsp+rdi+0x50],al
000000CD  42C6042201    |   mov byte [rdx+r12],0x1
000000D2  884C1450      |   mov [rsp+rdx+0x50],cl
000000D6  4883C701      |   add rdi,byte +0x1
000000DA  4883EE01      |   sub rsi,byte +0x1
000000DE  75B8          +-- jnz 0x98
000000E0  488B7D08          mov rdi,[rbp+0x8]
000000E4  488B7510          mov rsi,[rbp+0x10]
000000E8  488D542420        lea rdx,[rsp+0x20]
000000ED  488BCF            mov rcx,rdi
000000F0  FF157A100000      call qword [rel 0x1170] // stat64i32
000000F6  85C0              test eax,eax
000000F8  0F85D2000000      jnz dword 0x1d0
000000FE  48634C2434        movsxd rcx,dword [rsp+0x34]
00000103  8D5001            lea edx,[rax+0x1]
00000106  FF1554100000      call qword [rel 0x1160] // calloc
0000010C  4885C0            test rax,rax
0000010F  4C8BE0            mov r12,rax
00000112  0F84B8000000      jz dword 0x1d0
00000118  488D15C1100000    lea rdx,[rel 0x11e0]
0000011F  488BCF            mov rcx,rdi
00000122  FF1530100000      call qword [rel 0x1158] // fopen
00000128  4885C0            test rax,rax
0000012B  488BF8            mov rdi,rax
0000012E  0F8493000000      jz dword 0x1c7
00000134  4C63442434        movsxd r8,dword [rsp+0x34]
00000139  4C8BC8            mov r9,rax
0000013C  BA01000000        mov edx,0x1
00000141  498BCC            mov rcx,r12
00000144  FF15FE0F0000      call qword [rel 0x1148] // fread
0000014A  488BCF            mov rcx,rdi
0000014D  FF15ED0F0000      call qword [rel 0x1140] // fclose
00000153  488D158A100000    lea rdx,[rel 0x11e4]
```

```
0000015A  488BCE           mov rcx,rsi
0000015D  FF15F50F0000     call qword [rel 0x1158] // fopen
00000163  4885C0           test rax,rax
00000166  488BE8           mov rbp,rax
00000169  745C             jz 0x1c7
0000016B  395C2434         cmp [rsp+0x34],ebx
0000016F  8BFB             mov edi,ebx
00000171  7E2D             jng 0x1a0
00000173  498BF4           mov rsi,r12
00000176  66666690         xchg ax,ax
0000017A  666690           xchg ax,ax
0000017D  666690           xchg ax,ax
00000180  0FB606        +-> movzx eax,byte [rsi]          // al = *rsi;
00000183  488BD5        |   mov rdx,rbp                   // rdx = file handle;
00000186  0FB64C0450    |   movzx ecx,byte [rsp+rax+0x50] // cl = rsp+0x50[al];
0000018B  FF15A70F0000  |   call qword [rel 0x1138]       // fputc
00000191  83C701        |   add edi,byte +0x1       // edi++;
00000194  4883C601      |   add rsi,byte +0x1       // rsi++;
00000198  3B7C2434      |   cmp edi,[rsp+0x34]      // if(edi < read_file_size)
0000019C  7CE2          +-- jl 0x180                //     goto 0x180;
0000019E  6690             xchg ax,ax
000001A0  0FB64C1C50    +-> movzx ecx,byte [rsp+rbx+0x50] // ecx = (ebx = 0..255)
000001A5  488BD5        |   mov rdx,rbp                   // edx = file handle;
000001A8  83F155        |   xor ecx,byte +0x55            // cl ^= 0x55;
000001AB  FF15870F0000  |   call qword [rel 0x1138]       // fputc
000001B1  4883C301      |   add rbx,byte +0x1       // rbx++;
000001B5  48            |   rex.w
000001B6  81FB00010000  |   cmp ebx,0x100           // if(ebx < 256)
000001BC  7CE2          +-- jl 0x1a0                //     goto 0x1a0;
000001BE  488BCD           mov rcx,rbp
000001C1  FF15790F0000     call qword [rel 0x1140] // fclose
000001C7  498BCC           mov rcx,r12
000001CA  FF15800F0000     call qword [rel 0x1150] // free
000001D0  4C8BA42460010000 mov r12,[rsp+0x160]
000001D8  488BBC2468010000 mov rdi,[rsp+0x168]
000001E0  488BB42470010000 mov rsi,[rsp+0x170]
```

```
000001E8  488B9C2490010000  mov rbx,[rsp+0x190]
000001F0  33C0              xor eax,eax
000001F2  488BAC2498010000  mov rbp,[rsp+0x198]
000001FA  488B8C2450010000  mov rcx,[rsp+0x150]
00000202  4833CC            xor rcx,rsp
00000205  E826000000        call dword 0x230
0000020A  48                rex.w
0000020B  81C478010000      add esp,0x178
00000211  C3                ret
-----
```

ndisasm.exe では関数呼び出しの名前解決がなされないため、自前で作成する。

```
-----  getaddr.c
#include <stdio.h>

int main(int argc, char *argv[])
{
        FILE *fp;
        long offset;
        char funcname[256];

        if(argc < 2){
                fprintf(stderr, "%s <offset>¥n", argv[0]);
                return 1;
        }

        offset = (long)strtoul(argv[1], NULL, 16);

        if((fp = fopen("encode4.bin", "rb")) == NULL){
                fprintf(stderr, "error: fopen¥n");
                return -1;
        }

        fseek(fp, offset, SEEK_SET);
        fread(&offset, 4, 1, fp);
```

```
        offset -= 0x1000;

        fseek(fp, offset + 2, SEEK_SET);
        fread(funcname, 1, sizeof(funcname), fp);

        fclose(fp);

        printf("[%08X] %s¥n", strtoul(argv[1], NULL, 16), funcname);
        return 0;
}
```

これで　call qword [rel 0x____]　という部分をすべて関数名に置き換えられる。同じディレクトリに encode4.bin が存在する状態で以下のコマンドを実行する。

```
$ gcc getaddr.c -o getaddr
$ ./getaddr 1128
[00001128] _time64
```

00000019 より、プログラムは 2 つの引数をとることが分かるので、これまでの流れから考えて、ひとつ目の引数から読み込みファイルを開き、ふたつ目の引数のファイルへ書き込む、という処理だと推測できる。

ポイントとなる処理は 2 つの fputc を元に解析する。

00000180 から 0000019C までの処理がひとつ目の fputc（ファイルへの書き込み）処理、000001A0 から 000001BC までがふたつ目の fputc 処理である。

jl 命令より、ひとつ目の fputc では、読み込んだファイルサイズ分だけ書き込みを行い、ふたつ目の fputc では、256 バイト固定でファイルへの書き込みを行う。また、終端に書き込まれるふたつ目の fputc の 256 バイトのデータは 1 バイトずつ 0x55 で xor されている（000001A8）。

ひとつ目の fputc の処理を解析すると、00000180 の esi は、さかのぼると r12 からコピーされているため、読み込んだファイルデータの先頭である。00000186 にて、rsp+eax+0x50 が最終的に fputc される 1 バイトのデータとなるため、読み込んだデータ 1 バイトをオフセットとしたアドレスにあるデータが、書き込みファイルへ書き込まれるデータとなる。

ふたつ目の fputc の処理を解析すると、rsp+ebx+0x50（ebx は 0..255）を 1 バイトずつ 0x55 で xor してファイルへ書き込んでいる。

以上のことから、このエンコーダは、rsp+0x50 に 256 バイトのテーブルが存在し、そのテーブルを使った転置により、エンコードが行われるとわかる。そして、読み込みファイルサイズ分すべてを 1 バイトずつエンコードし書き込んだら、最後にテーブルそのものを 256 バイト分ファイルへ書き込んで、処理が終了となる。

よって、decode4.c は以下のようなプログラムとなる。

```
-----  decode4.c
#include <stdio.h>
#include <sys/stat.h>

int main(int argc, char *argv[])
{
        FILE *fp;
        struct stat st;
        unsigned char *buff;
        unsigned char *table;
        int i;

        if(stat(argv[1], &st) != 0)
                return 1;

        buff = calloc(st.st_size, 1);
        fp = fopen(argv[1], "rb");
        fread(buff, 1, st.st_size, fp);
        fclose(fp);
```

```
        table = buff + (st.st_size - 256);

        for(i=0; i < 256; i++){
                table[i] ^= 0x55;
        }

        for(i=0; i < (st.st_size - 256); i++){
                fputc((int)table[buff[i]], stdout);
        }

        free(buff);
        return 0;
}
```

以上で、encode4.bin の解析は終了。

ちなみに、デコーダを作るだけならば、テーブル生成の処理を解析する必要はない。しかし、もし興味があるならば、0000003D 以降の処理を読めば、256 バイトのテーブルが毎回ランダムで生成されることが確認できるだろう。

□ encode3.bin 解析

encode3.bin は、ARM/Linux 環境の 32 ビット ELF ファイルである。製品版、もしくは無料 Demo 版の IDAPro で解析できる。また、qemu を使うことで、x86 上でバイナリを動かすことができる。

encode3.bin を解析する上での大きなポイントは、どの時点で Base32 だと気づけるか? という点である。以下にいくつかの解答例を示す。

ARM であるためこのプログラムは x86 上では動作しないが、qemu を使うことで、encode3.bin を x86 上でエミュレートし実行できる。つまり、引数へ適当なファイルを渡してみて、その実行結果からアルゴリズムを推測する方法がある。

```
# cat > e3.samp
```

```
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD
^C (Ctrl + C)
# qemu-arm -L /usr/arm-linux-gnu/ encode3.bin e3.samp e3.samp2
# cat e3.samp2
IFAUCQKBIFAUCQSCIJBEEQSCIJBUGQ[DINBUGQ[EIRCEIRCEIRCAU
-----
```

複数回 encode3.bin を実行してみると、エンコード後のデータ（e3.samp2）は ASCII コード表における、'A'以降の 32 バイトのデータしか出現しないため、32 文字で構成されることが分かる。勘の良い人ならば、もしかしたらこのことからだけで Base32 アルゴリズムを使用していると推測できるかもしれない。そして、事実、それは当たっている。ここに気づけたならば、アセンブルコードを解析しなくとも、実行結果を確認しただけで Base32 デコーダを作成し、encode3.bin を攻略できただろう。

また、仮に Base32 アルゴリズムだと最後まで気づかなくとも、encode3.bin の中に、どこからも呼び出されていないデコード処理が残っていることを突き止めれば、encode3.bin を数バイト書き換えるだけで、デコーダを作成できる。

```
-----  encode3.bin
.text:000094C8                 LDR     R3, [R11,#n]
.text:000094CC                 LDR     R0, [R11,#ptr]
.text:000094D0                 MOV     R1, R3
.text:000094D4                 LDR     R2, [R11,#var_10]
.text:000094D8                 BL      sub_8D64
-----
```

sub_8D64 は 3 つの引数をとる。R0 が入力データアドレス、R1 が入力データのサイズ、R2 が出力データ用領域のアドレス。

この sub_8D64 は見ての通りエンコード処理を行う関数だが、逆アセンブルコードを読んでいくと、このエンコード関数は 00008D64～00009098 までとなっており、その後の 0000909C 以降がデコード関数となっている。そして、この関数（0000909C）はどこからも呼び出されてはいないが、確かにバイナリ内には存在する。

よって、000094D8 の BL 命令のジャンプ先を 0000909C に変更するだけで、デコーダが作成できる。

encode3.bin

```
ADDRESS   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
000014B0  01 10 A0 E3 03 20 A0 E1 1C 30 1B E5 0A FC FF EB   ..... ...0......
000014C0  1C 00 1B E5 1D FC FF EB 4C 30 1B E5 14 00 1B E5   .......・0......
000014D0  03 10 A0 E1 10 20 1B E5 21 FE FF EB 00 30 A0 E1   ..... ..!....0..
000014E0  20 30 0B E5 14 00 1B E5 1A FC FF EB 80 30 1B E5    0.........・0..
```

decode3.bin

```
ADDRESS   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
000014B0  01 10 A0 E3 03 20 A0 E1 1C 30 1B E5 0A FC FF EB   ..... ...0......
000014C0  1C 00 1B E5 1D FC FF EB 4C 30 1B E5 14 00 1B E5   .......・0......
000014D0  03 10 A0 E1 10 20 1B E5 EF FE FF EB 00 30 A0 E1   ..... .襄....0..
000014E0  20 30 0B E5 14 00 1B E5 1A FC FF EB 80 30 1B E5    0.........・0..
```

ジャンプ先を 0000909C に変更した decode3.bin を作成した。

```
-----
# qemu-arm -L /usr/arm-linux-gnu/ decode3.bin e3.samp2 e3.samp3
# cat e3.samp3
AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDD
-----
```

もちろん、これら以外の解答方法として、純粋にアセンブルコードをすべて読み解き、デコーダを作成しても、同様に問題は解ける。

以下に、decode3.pl を示す。

```
-----  decode3.pl
#!/usr/bin/perl

use MIME::Base32 qw (rfc);

open(FILE, "$ARGV[0]") or die "open";
read(FILE, $buff, (-s "$ARGV[0]"));
close(FILE);

$buff =~ s/¥[/2/g;
```

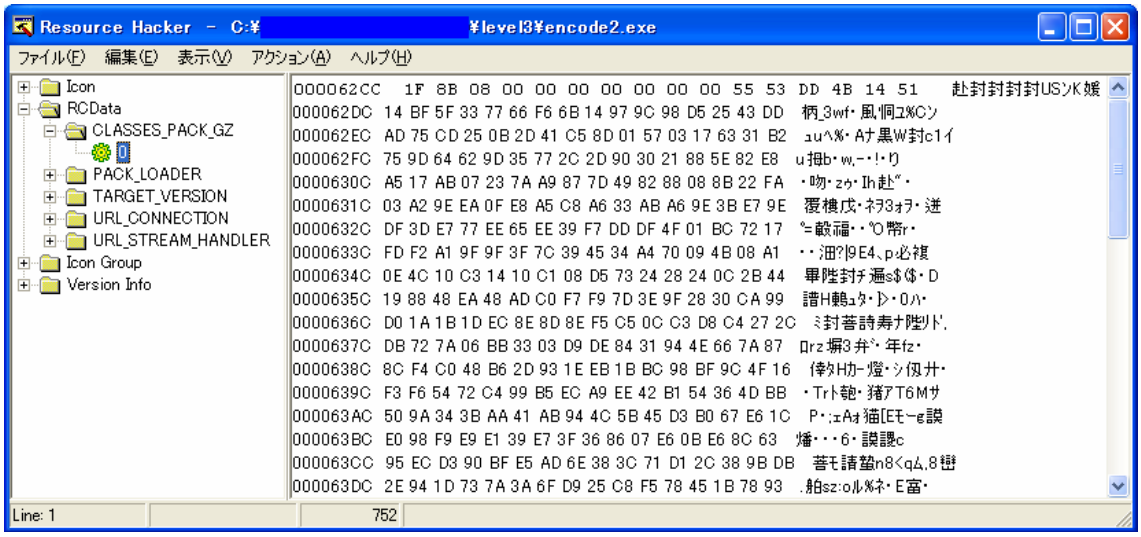
```
$buff =~ s/¥¥/3/g;
$buff =~ s/¥]/4/g;
$buff =~ s/¥^/5/g;
$buff =~ s/¥_/6/g;
$buff =~ s/¥`/7/g;

print MIME::Base32::decode($buff);
-----
```

以上で、encode3.bin の解析は終了。

□ encode2.bin 解析

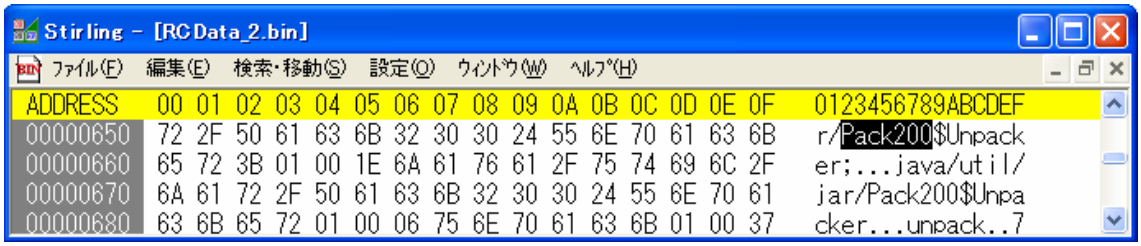
encode2.bin は、encode6.bin と似ており、さほど難しくない。拡張子を exe に変更すると、Java を意味するコーヒーのアイコンが表示される。

EXE ファイル内のリソースを確認すると CLASSES_PACK_GZ というステータスがある。



すべてのリソースを個別ファイルに出力すると、RCData_1.bin から RCData_5.bin までの 5 つの特殊なファイルが作成される。

RCData_2.bin をバイナリエディタで開くと、Pack200 という文字列がある。



pack200、unpack200 は Java にデフォルトでついているコマンドで、JAR ファイルを圧縮、展開するツール。これを使って、RCData_1.bin を JAR ファイルに戻す。

```
-----
C:¥Java¥jre1.6.0_05¥bin>unpack200.exe RCData_1.bin RCData_1.jar
-----
```

展開された JAR ファイル（RCData_1.jar）を、さらに zip 展開すると、class ファイルが出来上がる。

この class ファイルを jad でデコンパイルする。

```
-----
C:¥>jad encode2.class
Parsing encode2.class...  Generating encode2.jad
-----
```

encode2.jad をテキストエディタで開くと、エンコーダのプログラムとなる。

```
-----  encode2.jad
// Decompiled by Jad v1.5.8g.  Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.kpdus.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   encode2.java

import java.io.*;
```

```
public class encode2
{

    public encode2()
    {
    }

    public static void main(String args[])
    {
        byte abyte0[] = {
            14, 8, 5, 13, 15, 2, 12, 11, 1, 10,
            9, 7, 6, 3, 0, 4
        };
        byte abyte1[] = new byte[16];
        byte abyte2[] = new byte[16];
        try
        {
            FileInputStream fileinputstream = new FileInputStream(args[0]);
            FileOutputStream fileoutputstream = new FileOutputStream(args[1]);
            int j1 = 16;
            for(int i = 0; j1 == 16; i += 16)
            {
                j1 = fileinputstream.read(abyte1);
                for(int j = 0; j < j1; j++)
                {
                    byte byte0 = abyte0[(abyte1[j] & 0xf0) >> 4];
                    byte byte2 = abyte0[abyte1[j] & 0xf];
                    abyte1[j] = (byte)(byte0 << 4 | byte2);
                }

                if(j1 == 16)
                {
                    for(int k = 0; k < j1; k++)
                        abyte2[k] = abyte1[abyte0[k]];
```

```
                } else
                {
                    for(int l = 0; l < j1; l++)
                        abyte2[l] = abyte1[l];

                }
                for(int i1 = 0; i1 < j1; i1++)
                {
                    byte byte1 = abyte2[i1];
                    abyte2[i1] = (byte)(((byte1 != 255 ? byte1 + 1 : 0) ^ i1) & 0xff);
                }

                fileoutputstream.write(abyte2, 0, j1);
            }

            fileinputstream.close();
            fileoutputstream.close();
        }
        catch(IOException ioexception)
        {
            System.out.println(ioexception);
        }
    }
}
```

これに対応するデコーダを作成する。

```
-----  decode2.java
// Decompiled by Jad v1.5.8e2. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://kpdus.tripod.com/jad.html
// Decompiler options: packimports(3)
// Source File Name:   encode2.java

import java.io.*;
```

```
public class decode2
{

    public decode2()
    {
    }

    public static void main(String args[])
    {
        byte abyte0[] = {
            14, 8, 5, 13, 15, 2, 12, 11, 1, 10,
            9, 7, 6, 3, 0, 4
        };
        byte abyte1[] = new byte[16];
        byte abyte2[] = new byte[16];
        try
        {
            FileInputStream fileinputstream = new FileInputStream(args[0]);
            FileOutputStream fileoutputstream = new FileOutputStream(args[1]);
            int j1 = 16;
            for(int i = 0; j1 == 16; i += 16)
            {
                j1 = fileinputstream.read(abyte1);

                for(int i1 = 0; i1 < j1; i1++)
                {
                    byte byte1 = (byte)(abyte1[i1] ^ i1);
                    abyte1[i1] = (byte)(byte1 != 0 ? byte1 - 1 : 255);
                }

                for(int j = 0; j < j1; j++)
                {
                    byte byte0 = abyte0[(abyte1[j] & 0xf0) >> 4];
                    byte byte2 = abyte0[abyte1[j] & 0xf];
                    abyte1[j] = (byte)(byte0 << 4 | byte2);
                }
```

```
                if(j1 == 16)
                {
                    for(int k = 0; k < j1; k++)
                        abyte2[k] = abyte1[abyte0[k]];

                } else
                {
                    for(int l = 0; l < j1; l++)
                        abyte2[l] = abyte1[l];

                }


                fileoutputstream.write(abyte2, 0, j1);
            }

            fileinputstream.close();
            fileoutputstream.close();
        }
        catch(IOException ioexception)
        {
            System.out.println(ioexception);
        }
    }
}
-----
```

以上で、encode2.bin の解析は終了。

□ encode1.bin 解析

encode1.bin は file コマンドにより「x86 hard disk boot sector」のファイルであると分かる。ブートセクタであるため、おそらく 16 ビットで動作すると仮定し、16 ビットモードで逆アセンブルする。エントリポイントは先頭 80 バイトから。

```
-----
C:¥nasm-2.07>ndisasm.exe -b 16 -e 80 encode1.bin
00000000  B80000            mov ax,0x0
00000003  8ED0              mov ss,ax
00000005  BC007C            mov sp,0x7c00
00000008  8ED8              mov ds,ax
0000000A  8EC0              mov es,ax
0000000C  B100              mov cl,0x0   // cl = 0x00;
0000000E  B400      +-----> mov ah,0x0   // ah = 0x00;
00000010  CD16      |       int 0x16     // al(ah) = getchar();
00000012  3C00      |       cmp al,0x0   // if(al != 0x00)
00000014  7502      |   +-- jnz 0x18     //     goto 0x18;
00000016  88E0      |   |   mov al,ah    // al = ah;
00000018  3C20      |   +-> cmp al,0x20  // if(al == 0x00)
0000001A  7402      |   +-- jz 0x1e      //     goto 0x1e;
0000001C  28C8      |   |   sub al,cl     // al -= cl;
0000001E  B40E      |   +-> mov ah,0xe    // ah = 0x0e;
00000020  BB0F00    |       mov bx,0xf    // bx = 0x0f;
00000023  CD10      |       int 0x10      // putc(al);
00000025  80C101    |       add cl,0x1    // cl++;
00000028  80F941    |       cmp cl,0x41   // if(cl == 0x41)
0000002B  7402      |   +-- jz 0x2f       //     goto 0x2f;
0000002D  EBDF      +-- | - jmp short 0xe // goto 0x0e;
0000002F  B100      |   +-> mov cl,0x0    // cl = 0;
00000031  EBDB      +------ jmp short 0xe // goto 0x0e;
-----
```

処理自体はとても簡単。00000010 より、int 0x16 で標準入力から値をとってきて、入力値から現在のオフセットの分だけ減算した値（0000001C）を int 0x10 で出力している。ただし、減算値は 0x41 になると 0 に初期化される（00000028）。

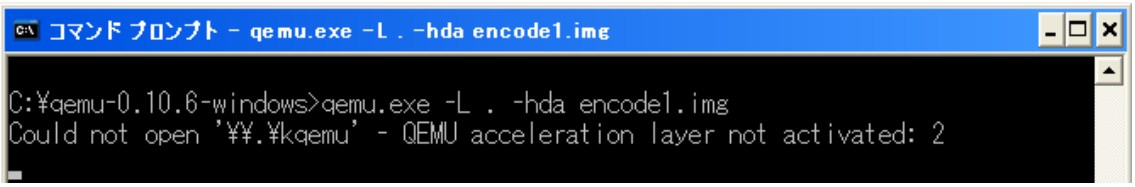
デコーダは以下になる。

```
-----  decode1.c
#include <stdio.h>
```

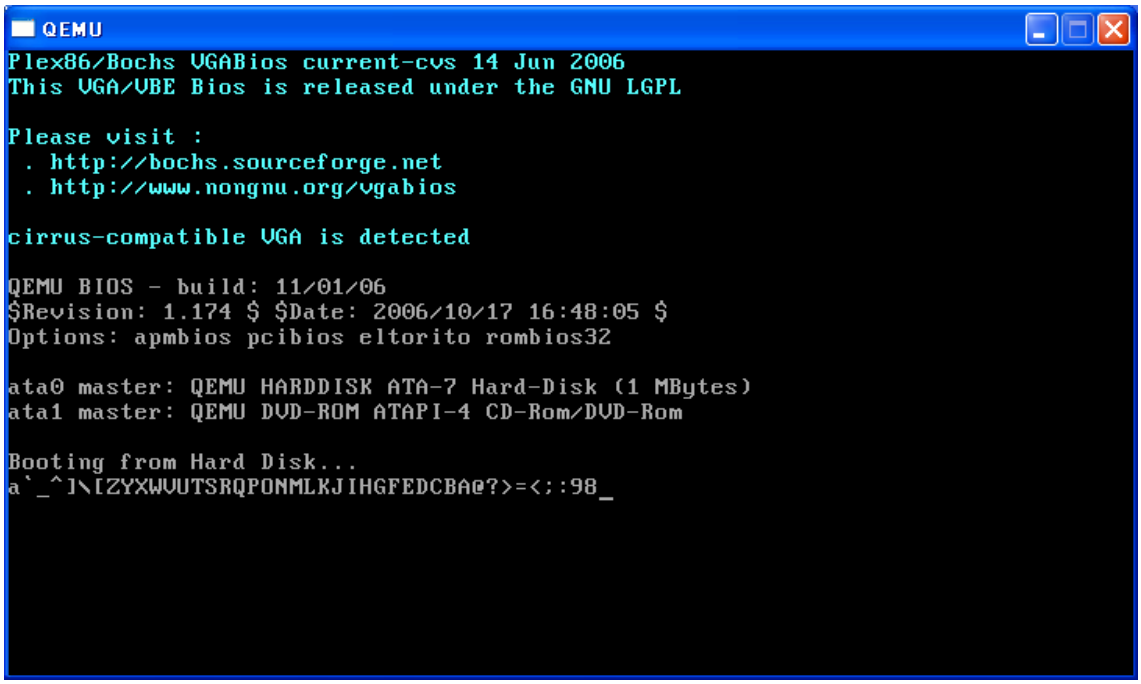
```
int main(int argc, char *argv[])
{
        FILE *fp;
        int al, cl;

        if((fp = fopen(argv[1], "rb")) == NULL)
                return 1;

        cl = 0;
        while((al = fgetc(fp)) != EOF){
                if(al != 0x20)
                        al += cl;
                putchar(al);
                cl++;
                if(cl == 0x41)
                        cl = 0;
        }

        fclose(fp);
        return 0;
}
-----
```

以上で、encode1.bin の解析は終了。

ちなみに qemu で実行すると、よりダイレクトにエンコード処理が理解できる。



以下の画像は「A」を入力し続けた例。



□ encode0 解析

これまで作成した decode6 から decode1 までを順番に password.e0.e1.e2.e3.e4.e5.e6 ファイルに適用させると、最後のエンコード（encode0）が出現する。

```
-----  password.e0
length q caller lc and print chr oct ord q qx q and print chr ord uc q map m and print chr ord uc qw q s q and print chr ord uc q msgctl m and print chr ord uc q qw q and print chr ord uc q local and print chr ord uc q qr q and print chr oct ord q chr lc and print chr oct ord uc q chr uc and print chr ord q q q and print chr ord uc q sin s and print chr ord qw q ne q and print chr oct ord q gt log and print chr ord uc q local and print chr ord q eval ne and print chr ord uc q each le and print chr ord uc q stat s and print chr ord q tie lt and print chr ord uc q xor x and print chr ord qw q no q and print chr oct hex ord uc qw q y q and print chr ord q stat s and print chr ord uc q pop and print chr oct hex ord uc q my m and print chr ord uc q lt eval and print chr ord q chr lc and print chr ord uc q ref or and print chr oct hex ord uc qw q y q and print chr ord qw q for q and print chr ord uc q dump and and print chr ord q lt eval and print chr ord uc qw q use q and print chr ord uc q or no and print chr ord q ge log
-----
```

password.e0 は、Perl スクリプトとなっており、実行することで最終的なパスワードが得られる。

```
-----
# perl password.e0
PASSWORD: InNOvATiOn_tO_ThE_fUtURe
-----
```

よって、LEVEL3 の解答は「InNOvATiOn_tO_ThE_fUtURe」となる。

なお、最後の Perl スクリプトは、ppencode と呼ばれるもので、以下のサイトから詳細な情報を得ることができる。

ppencode
http://www.namazu.org/~takesako/ppencode/demo.html

■ LEVEL4：PowerPC big-endian

level4.bin が何のタイプのファイルであるかを、file コマンドにより確認する。

```
-----
$ file level4.bin
level4.bin: ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV), for GNU/Linux 2.6.8, dynamically linked (uses shared libs), for GNU/Linux 2.6.8, stripped
-----
```

PowerPC 環境で動作する ELF ファイルだと分かる。PowerPC は製品版の IDAPro Advanced や objdump で解析できる。

PowerPC 環境用にコンパイルされているが、qemu を使うことで x86 上でも実行できる。実行には ppc 用の libc と ld の 2.8.90 以上が必要（libc-2.8.90.so、ld-2.8.90.so）。

```
-----
$ qemu-ppc -L /usr/powerpc-linux-gnu/ level4.bin
$
-----
```

実行しても何も表示されない。

objdump を使い、逆アセンブルする。

```
-----
$ /usr/powerpc-linux/bin/objdump -d level4.bin > level4.dis
-----
```

あとは逆アセンブルコードを読むのみ。

```
-----  level4.dis
100013f0:       94 21 f9 d0     stwu    r1,-1584(r1)
100013f4:       7c 08 02 a6     mflr    r0
100013f8:       90 01 06 34     stw     r0,1588(r1)
```

```
100013fc:       93 a1 06 24     stw     r29,1572(r1)
10001400:       93 e1 06 2c     stw     r31,1580(r1)
10001404:       7c 3f 0b 78     mr      r31,r1
10001408:       90 7f 00 0c     stw     r3,12(r31)
1000140c:       90 9f 00 08     stw     r4,8(r31)
10001410:       80 02 8f f8     lwz     r0,-28680(r2)
10001414:       90 1f 06 1c     stw     r0,1564(r31)
10001418:       38 00 00 00     li      r0,0
1000141c:       38 1f 00 10     addi    r0,r31,16
10001420:       39 20 04 84     li      r9,1156
10001424:       7c 03 03 78     mr      r3,r0 // arg1 = ptr
10001428:       38 80 00 00     li      r4,0  // arg2 = 0
1000142c:       7d 25 4b 78     mr      r5,r9 // arg3(size) = 1156
10001430:       48 00 04 c1     bl      100018f0 <_IO_stdin_used-0x108> // memset
10001434:       38 00 00 01     li      r0,1
10001438:       98 1f 01 d9     stb     r0,473(r31) // *(ptr+473-16) = 1
1000143c:       38 00 00 01     li      r0,1
10001440:       98 1f 01 fd     stb     r0,509(r31) // *(ptr+509-16) = 1
10001444:       38 00 00 01     li      r0,1
10001448:       98 1f 02 1c     stb     r0,540(r31) // *(ptr+540-16) = 1
1000144c:       38 00 00 01     li      r0,1
10001450:       98 1f 02 1d     stb     r0,541(r31) // *(ptr+541-16) = 1
10001454:       38 00 00 01     li      r0,1
10001458:       98 1f 02 20     stb     r0,544(r31) // *(ptr+544-16) = 1
1000145c:       38 00 00 01     li      r0,1
10001460:       98 1f 02 21     stb     r0,545(r31) // *(ptr+545-16) = 1
10001464:       38 00 00 01     li      r0,1
10001468:       98 1f 02 22     stb     r0,546(r31) // *(ptr+546-16) = 1
1000146c:       80 1f 00 0c     lwz     r0,12(r31) // r0 = argc
10001470:       7c 03 03 78     mr      r3,r0
10001474:       48 00 04 5d     bl      100018d0 <_IO_stdin_used-0x128> // srand
10001478:       48 00 04 c9     bl      10001940 <_IO_stdin_used-0xb8> // rand
1000147c:       7c 60 1b 78     mr      r0,r3    // r0 = (r3 = rand return val)
10001480:       54 09 05 3e     clrlwi  r9,r0,20 // r9 = r0 & 0x0FFF
10001484:       38 1f 00 10     addi    r0,r31,16
10001488:       7c 03 03 78     mr      r3,r0 // arg1 = ptr
```

```
1000148c:       7d 24 4b 78     mr      r4,r9 // arg2 = r9(random val)
10001490:       4b ff f8 5d     bl      10000cec <_IO_stdin_used-0xd0c>
10001494:       38 1f 04 94     addi    r0,r31,1172
10001498:       90 1f 05 14     stw     r0,1300(r31) // 1300(r31) = r0
1000149c:       38 00 00 00     li      r0,0         // r0 = 0
100014a0:       90 1f 05 18     stw     r0,1304(r31) // 1304(r31) = r0
100014a4:       48 00 00 4c     b       100014f0 <_IO_stdin_used-0x508>
100014a8:   +-> 83 bf 05 18     lwz     r29,1304(r31)// r29 = 1304(r31)
100014ac:   |   39 7f 00 10     addi    r11,r31,16   // r11 = ptr
100014b0:   |   80 1f 05 18     lwz     r0,1304(r31) // r0 = 1304(r31)
100014b4:   |   54 09 08 3c     rlwinm  r9,r0,1,0,30 // r9 = (r0<<1) & 0x7FFFFFFF
100014b8:   |   55 20 20 36     rlwinm  r0,r9,4,0,27 // r0 = (r9<<4) & 0x0FFFFFFF
100014bc:   |   7c 09 02 14     add     r0,r9,r0  // r0 += r9
100014c0:   |   7c 0b 02 14     add     r0,r11,r0 // r0 += r11
100014c4:   |   7c 03 03 78     mr      r3,r0 // r3 = r0
100014c8:   |   4b ff f7 99     bl      10000c60 <_IO_stdin_used-0xd98>
100014cc:   |   7c 6b 1b 78     mr      r11,r3 // r11 = func_c60();
100014d0:   |   57 a0 10 3a     rlwinm  r0,r29,2,0,29 // r0 = (r29<<2)&0x3FFFFFFF
100014d4:   |   39 5f 06 20     addi    r10,r31,1568 // r10 = r31 + 1568
100014d8:   |   7d 2a 02 14     add     r9,r10,r0    // r9 = r10 + r0
100014dc:   |   39 29 fe 74     addi    r9,r9,-396   // r9 -= 396
100014e0:   |   91 69 00 00     stw     r11,0(r9)    // 0(r9) = r11
100014e4:   |   81 3f 05 18     lwz     r9,1304(r31) // r9 = 1304(r31)
100014e8:   |   38 09 00 01     addi    r0,r9,1      // r0 = r9 + 1
100014ec:   |   90 1f 05 18     stw     r0,1304(r31) // 1304(r31) = r0
100014f0:   |   80 1f 05 18     lwz     r0,1304(r31) // r0 = 1304(r31)
100014f4:   |   2f 80 00 1f     cmpwi   cr7,r0,31 // if(r0 <= 31) then jump
100014f8:   +-- 40 9d ff b0     ble+    cr7,100014a8 <_IO_stdin_used-0x550>
100014fc:       80 1f 05 14     lwz     r0,1300(r31)
10001500:       7c 03 03 78     mr      r3,r0
10001504:       38 80 00 80     li      r4,128
10001508:       4b ff f0 65     bl      1000056c <_IO_stdin_used-0x148c>
1000150c:       7c 60 1b 78     mr      r0,r3 // r0 = func_56c(1300(r31), 128)
10001510:       2f 80 25 37     cmpwi   cr7,r0,9527 // if(r0 != 9527) then jump
10001514:   +-- 40 9e 00 48     bne-    cr7,1000155c <_IO_stdin_used-0x49c>
10001518:   |   38 1f 05 1c     addi    r0,r31,1308
```

```
1000151c:   |   7c 03 03 78     mr      r3,r0
10001520:   |   38 80 00 00     li      r4,0
10001524:   |   38 a0 01 00     li      r5,256
10001528:   |   48 00 03 c9     bl      100018f0 <_IO_stdin_used-0x108> // memset
1000152c:   |   38 1f 05 1c     addi    r0,r31,1308
10001530:   |   80 7f 05 14     lwz     r3,1300(r31)
10001534:   |   38 80 00 80     li      r4,128
10001538:   |   7c 05 03 78     mr      r5,r0
1000153c:   |   38 c0 01 00     li      r6,256
10001540:   |   4b ff f6 21     bl      10000b60 <_IO_stdin_used-0xe98>
10001544:   |   38 1f 05 1c     addi    r0,r31,1308
10001548:   |   3d 20 10 00     lis     r9,4096
1000154c:   |   38 69 1a 40     addi    r3,r9,6720 // arg1 = PASSWORD: %s
10001550:   |   7c 04 03 78     mr      r4,r0      // arg2 = string
10001554:   |   4c c6 31 82     crclr   4*cr1+eq
10001558:   |   48 00 03 c9     bl      10001920 <_IO_stdin_used-0xd8> // printf
1000155c:   +-> 38 00 00 00     li      r0,0
10001560:       7c 03 03 78     mr      r3,r0
10001564:       80 1f 06 1c     lwz     r0,1564(r31)
10001568:       81 22 8f f8     lwz     r9,-28680(r2)
1000156c:       7c 00 4a 79     xor.    r0,r0,r9
10001570:       39 20 00 00     li      r9,0
10001574:       41 82 00 08     beq-    1000157c <_IO_stdin_used-0x47c>
10001578:       48 00 03 b9     bl      10001930 <_IO_stdin_used-0xc8>
1000157c:       81 61 00 00     lwz     r11,0(r1)
10001580:       80 0b 00 04     lwz     r0,4(r11)
10001584:       7c 08 03 a6     mtlr    r0
10001588:       83 ab ff f4     lwz     r29,-12(r11)
1000158c:       83 eb ff fc     lwz     r31,-4(r11)
10001590:       7d 61 5b 78     mr      r1,r11
10001594:       4e 80 00 20     blr
-----
```

最初は右も左も分からない状態であるため、PASSWORD という文字列を始点としてボトムアップ式に解析する。10001558 の bl 命令で printf 関数が呼ばれており、引数に PASSWORD という文字列が渡されている。

printf 関数呼び出しから少しさかのぼると 10001514 にて、条件分岐が行われ、printf 関数が呼ばれるか否か、が判断される。分岐の条件は「1000056c 関数の戻り値が 9527 であるか否か？」そして 1000056c 関数は 2 つの引数を取る。ひとつはポインタ、もうひとつは 128 という値。

2 つ目の引数 128 がサイズであると仮定しても、さすがに 128 バイトのデータ列に対するブルートフォースは無理がある。しかし、解答を得るためには、1000056c 関数の戻り値が 9527 になるような第一引数を与える必要がある（第二引数は 128 で固定）。

1000056c 関数の第一引数に渡されているのは 1300(r31)である。この値は 10001498 にて設定されている。また、100014a8 から 100014f8 までにて、32 回の繰り返し処理が行われている。32 回というのは 100014f4 の cmpwi 命令にて判明する。

32 回の繰り返し処理を解析すると、100014ac にて addi r11,r31,16 という命令が見つかる。この値 r11 は、10001490 にて呼ばれている 10000cec 関数の第一引数になっている。さらにこの関数の第二引数は 0 から 0xFFF までのランダム値である。

10000cec 関数の第一引数の出処を突き止めると、10001430 の memset にて、1156 バイト分ゼロクリアされており、さらに不特定な 7 箇所に 1 がセットされている。

これまでの流れをまとめる。

まず、1156 バイトのよく分からないデータ列が存在する。このデータ列は、ほとんどが 0 だが、一部（7 箇所だけ）1 となっている。そしてこのデータ列は 10000cec 関数の第一引数として渡される。この関数は 2 つの引数を持ち、もうひとつは 0 から 0xFFF までの乱数である。次に 1156 バイトのよく分からないデータ列を使って 32 回の何かしらの演算を行う。そして、その結果を 1300(r31)として 1000056c 関数の第一引数に渡している。ちなみに 1000056c 関数の第二引数は 128 固定。この関数の戻り値が 9527 か否か？　が最終的な解答への道となる。

以上のことから、0 から 0xFFF までの乱数となっている箇所をすべてブルートフォース的に試せば、解答が得られるのではないか？　と推測できる。ちなみに、srand に渡されている値は main 関数の第一引数である argc である。つまり、このプログラムに渡す引数の数を変更することで rand の戻り値が変わる。

```
-----
$ qemu-ppc -L /usr/powerpc-linux-gnu/ level4.bin 1
$ qemu-ppc -L /usr/powerpc-linux-gnu/ level4.bin 1 2
$ qemu-ppc -L /usr/powerpc-linux-gnu/ level4.bin 1 2 3
$ qemu-ppc -L /usr/powerpc-linux-gnu/ level4.bin 1 2 3 4
$ qemu-ppc -L /usr/powerpc-linux-gnu/ level4.bin 1 2 3 4 5
$ qemu-ppc -L /usr/powerpc-linux-gnu/ level4.bin 1 2 3 4 5 6 …
（省略）
$ qemu-ppc -L /usr/powerpc-linux-gnu/ level4.bin 1 2 3 4 5 6 … 4093 4094 4095
-----
```

上記のようなことを行えば、どこかのタイミングでパスワードが得られるかもしれない。格好の良い解き方ではないが…。

それなりに性能の高いマシンを用意し、CPU ごとに可能な限り並列化させると、数分～数十分程度で解答が得られる。

```
-----  brt.pl
#!/usr/bin/perl

@list = (
        0,      100,   200,   300,   400,   500,   600,   700,   800,   900,
        1000,  1100,  1200,  1300,  1400,  1500,  1600,  1700,  1800,  1900,
        2000,  2100,  2200,  2300,  2400,  2500,  2600,  2700,  2800,  2900,
        3000,  3100,  3200,  3300,  3400,  3500,  3600,  3700,  3800,  3900,
        4000,  4100
);

&main;
exit;

sub make_arg
{
        my ($i, $m, $s);
        $m = $_[0]; $s = "";
        for($i=0; $i < $m; $i++){
```

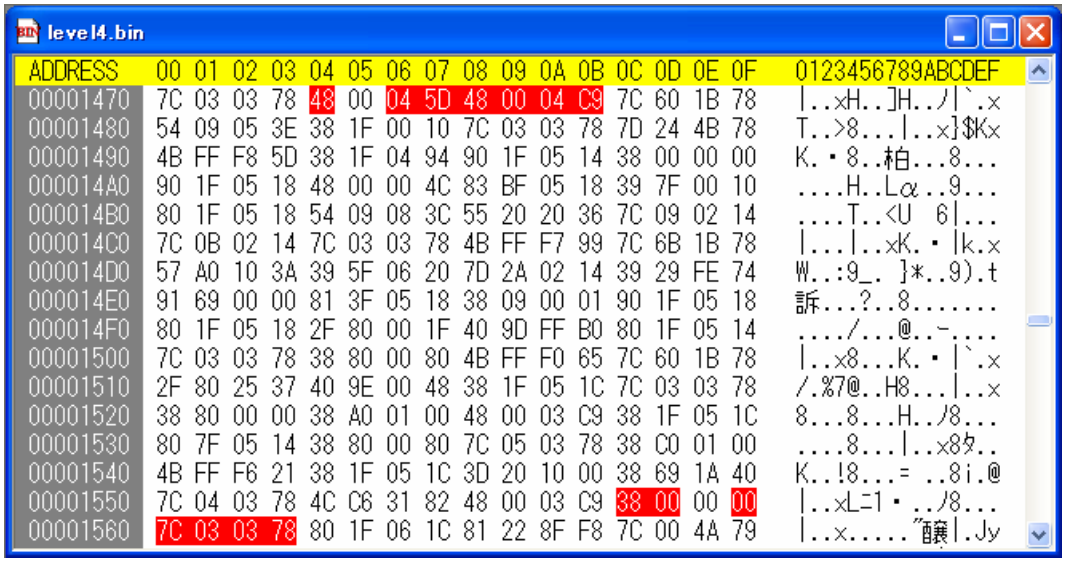
```
                $s .= " 0";
        }
        return $s;
}

sub child
{
        my ($i, $s, $b);
        $s = $_[0]; $b = $_[1];
        for($i=$s; $i < $b; $i++){
          system('qemu-ppc -L /usr/powerpc-linux-gnu/ level4.bin'. &make_arg($i));
        }
        exit;
}

sub main
{
        my ($i);
        for($i=0; $i < 41; $i++){
                $pid = fork;
                &child($list[$i], $list[$i+1]) if($pid == 0);
        }
        while(1){
                wait;
                print "END..¥n";
        }
}
```

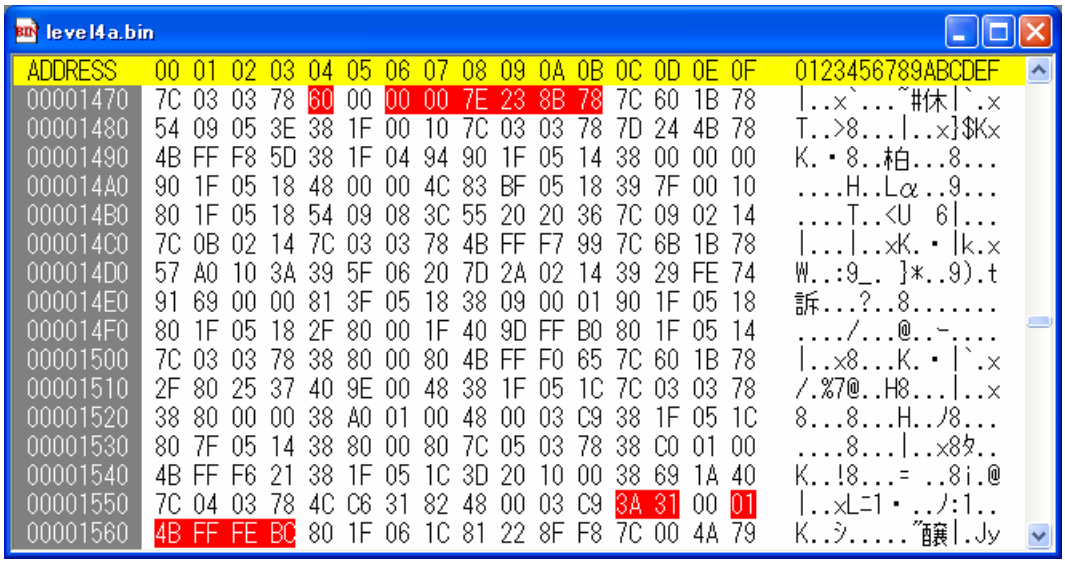
上記のプログラムを動作させる。

```
$ perl brt.pl
PASSWORD:AAeBgBoP8cAmHuhwYfAvylDz+Mg99jC4l58BGBEHMnwJBCnyiLAEYB6wHnJfD4M+bhhAAos
lAwlGjozfQo/gXELv4Mgz7+HuAyf/zS4jOG/96sAgjHYgEJxiP4RvoiAkDiAgFHoj4Bz5x6L+/OEUZen
sTOFJZKeBQYyX4iGPH/w=
```

数分後にパスワードが出力された。よって、解答は、
「AAeBgBoP8cAmHuhwYfAvylDz+Mg99jC4l58BGBEHMnwJBCnyiLAEYB6wHnJfD4M+bhhAAoslAwlGjozfQo/gXELv4Mgz7+HuAyf/zS4jOG/96sAgjHYgEJxiP4RvoiAkDiAgFHoj4Bz5x6L+/OEUZensTOFJZKeBQYyX4iGPH/w=」となる。

手法は同じだが、バイナリを直接書き換えることでも、ブルートフォースが可能。





```
$ qemu-ppc -L /usr/powerpc-linux-gnu/ level4a.bin
PASSWORD:AAeBgBoP8cAmHuhwYfAvylDz+Mg99jC4l58BGBEHMnwJBCnyiLAEYB6wHnJfD4M+bhhAAos
lAwlGjozfQo/gXELv4Mgz7+HuAyf/zS4jOG/96sAgjHYgEJxiP4RvoiAkDiAgFHoj4Bz5x6L+/OEUZen
sTOFJZKeBQYyX4iGPH/w=
-----
```

こちらは 1 プロセスで動かしたためか、20 分～30 分辺りで終了した。

変更箇所は以下の通り。

```
-----
$ diff level4.dis level4a.dis
2c2
< level4.bin:     ファイル形式 elf32-powerpc
---
> level4a.bin:     ファイル形式 elf32-powerpc
1080,1081c1080,1081
< 10001474:     48 00 04 5d     bl      100018d0 <_IO_stdin_used-0x128>
< 10001478:     48 00 04 c9     bl      10001940 <_IO_stdin_used-0xb8>
---
> 10001474:     60 00 00 00     nop
> 10001478:     7e 23 8b 78     mr      r3,r17
1138,1139c1138,1139
< 1000155c:     38 00 00 00     li      r0,0
< 10001560:     7c 03 03 78     mr      r3,r0
---
> 1000155c:     3a 31 00 01     addi    r17,r17,1
> 10001560:     4b ff fe bc     b       1000141c <_IO_stdin_used-0x5dc>
-----
```

srand、rand 呼び出しを削除して、本来 rand の戻り値となるはずの場所に r17 の値を投げている。r17 はループカウンタとして 0 から加算し続けるように変更。r17 はプログラム内で特に使われていない空きレジスタだったため使用した。

最後に level4.bin で使われているアルゴリズムを紹介する。

まず、10000cec 関数は、ライフゲームと呼ばれるゲームのアルゴリズムを使用している。ライフゲームとは、1970 年にイギリスの数学者ジョン・ホートン・コンウェイ（John Horton Conway）によって作られた生命の誕生、進化、淘汰などのプロセスを簡単にシミュレートするプログラムである。

次に戻り値が 9527 と比較される 1000056c 関数だが、これは TCP/IP ネットワークを流れる IP ヘッダのチェックサムを計算するアルゴリズムである。2 バイトの CRC 値が戻り値として返る。

printf 関数の前に呼ばれる 10000b60 関数は、これは参照している文字列を見ても予想がつく通り、Base64 エンコーダである。

では、最後に x86 で動作する解析プログラムを示す。level4.bin を隅々まで解析したならば、作成可能だろう。

```
-----  get_L4pass.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#define H  32
#define W  32

#include <winsock2.h>


unsigned short ip_checksum(unsigned short *data, int size)
{
  unsigned long sum = 0;

  while(size > 1){
    sum += *(data++);
    size -= 2;
  }
```

```
  if(size > 0)
    sum += (*data) & 0xff00;

  sum = (sum & 0xffff) + (sum >> 16);

  return ((~(unsigned short)((sum >> 16) + (sum & 0xffff))));
}


static void encode(unsigned char *src, int srcsize, unsigned char *dest)
{
  unsigned char base64[64] =
    "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

  int loop_num, last_num, i, j;
  unsigned long bin2base;

  loop_num = srcsize / 3;
  last_num = srcsize % 3;

  for(i=0, j=0; i < loop_num; i++, j++){
    bin2base   = 0; bin2base |= src[i*3+0];
    bin2base <<= 8; bin2base |= src[i*3+1];
    bin2base <<= 8; bin2base |= src[i*3+2];
    dest[j*4+0] = base64[ ((bin2base>>18) & 0x3F) ];
    dest[j*4+1] = base64[ ((bin2base>>12) & 0x3F) ];
    dest[j*4+2] = base64[ ((bin2base>> 6) & 0x3F) ];
    dest[j*4+3] = base64[ ((bin2base>> 0) & 0x3F) ];
  }

  if(last_num > 0){
    if(last_num == 1){
      bin2base   = 0; bin2base |= src[i*3+0];
      bin2base <<= 8; // bin2base |= src[i*3+1];
      bin2base <<= 8; // bin2base |= src[i*3+2];
      dest[j*4+0] = base64[ (unsigned long)((bin2base>>18) & 0x3F) ];
```

```
      dest[j*4+1] = base64[ (unsigned long)((bin2base>>12) & 0x3F) ];
      dest[j*4+2] = '=';
      dest[j*4+3] = '=';
    }
    if(last_num == 2){
      bin2base   = 0; bin2base |= src[i*3+0];
      bin2base <<= 8; bin2base |= src[i*3+1];
      bin2base <<= 8; // bin2base |= src[i*3+2];
      dest[j*4+0] = base64[ (unsigned long)((bin2base>>18) & 0x3F) ];
      dest[j*4+1] = base64[ (unsigned long)((bin2base>>12) & 0x3F) ];
      dest[j*4+2] = base64[ (unsigned long)((bin2base>> 6) & 0x3F) ];
      dest[j*4+3] = '=';
    }
  }
}


int base64_encode(unsigned char *src, int s_size, char *d_buff, int d_buff_size)
{
  int dest_size;

  if(s_size <= 0)
    return -1;

  dest_size = (((s_size / 3) + ((s_size % 3) ? 1 : 0)) * 4);
  if(dest_size > d_buff_size)
    return -1;

  encode(src, s_size, (unsigned char *)d_buff);
  return dest_size;
}


unsigned long array2long(char *array)
{
  int i;
```

```
  unsigned long n = 0;

  for(i=0; i < 32; i++){
    n <<= 1;
    n |= (array[i]) ? 1 : 0;
  }

  return n;
}


int lifegame(char (*buff)[W+2], int gen)
{
  int i, j, k;
  char map[H+2][W+2], back[H+2][W+2];

  unsigned long bin[32];
  unsigned char *pass;
  char base64[256];

  // init
  for(i=0; i < H+2; i++){
    for(j=0; j < W+2; j++){
      map[i][j]  = buff[i][j];
      back[i][j] = 0;
    }
  }

  // generation
  for(k=1; k <= gen; k++){
    for(i=1; i <= H; i++){
      for(j=1; j <= W; j++){
        if(map[i][j]){
          buff[i-1][j-1] = 1;
          back[i-1][j-1]++; back[i-1][j+0]++; back[i-1][j+1]++;
          back[i+0][j-1]++;                   back[i+0][j+1]++;
```

```
          back[i+1][j-1]++; back[i+1][j+0]++; back[i+1][j+1]++;
        }else{
          buff[i-1][j-1] = 0;
        }
      }
    }

    for(i=0; i <= H; i++){
      for(j=0; j <= W; j++){
        if((back[i][j] >= 3) && !(map[i][j]))
          map[i][j] = 1;
        else if(((back[i][j] == 2) || (back[i][j] == 3)) && (map[i][j]))
          map[i][j] = 1;
        else
          map[i][j] = 0;
        back[i][j] = 0;
      }
    }

    pass = (unsigned char *)bin;

    for(i=0; i < 32; i++)
      bin[i] = array2long(buff[i]);

    if(ip_checksum((unsigned short *)pass, 32 * 4) == 9527){
      for(i=0; i < 32; i++)
        bin[i] = ntohl(bin[i]);
      memset(base64, '¥0', sizeof(base64));
      base64_encode(pass, 32 * 4, base64, sizeof(base64));
      printf("PASSWORD: %s¥n", base64);
    }
  }

  return 0;
}
```

```
int main(void)
{
  char buff[H+2][W+2] = {

    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
```

```
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
    { 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 },
  };

  lifegame((char (*)[W+2])buff, 0xFFF);
  return 0;
}
-----
```

上記プログラムは Windows 環境において動作する。x86 はリトルエンディアンであるため、バイトオーダーの変換に winsock の ntohl 関数を使っている。また、処理速度向上のために (10 年前のマシンでも問題なく解析できるように)、lifegame 関数を何度も呼ばず、1 回の呼び出しで 0 から 0xFFF までを処理するようアルゴリズムを最適化している。

以上で、level4.bin の解析は終了となる。

■ 使用ツール一覧

最後に、今回解析に使用したツール一式を紹介する。

OllyDbg
http://www.ollydbg.de/

IDAPro（無料版、Demo 版、製品版）
http://www.hex-rays.com/idapro/idadownfreeware.htm
http://www.hex-rays.com/idapro/idadowndemo.htm

nasm & ndisasm
http://d.hatena.ne.jp/kenjiaiko/20090907/1252306069

qemu（エミュレータ）
http://www.nongnu.org/qemu/

Stirling（バイナリエディタ）
http://www.vector.co.jp/soft/win95/util/se079072.html

jad（Java デコンパイラ）
http://web.archive.org/web/20080214075546/http://www.kpdus.com/jad.html#download

Linux 環境と有用な多くのコマンドラインツール
http://www.kernel.org/

■ 最後に

参加者の方々の向上心と技術力に敬意を表すと共に、コンピュータネットワークの更なる発展と繁栄を願い、当イベントの終了とさせていただきます。ご参加、有難うございました。

ネットエージェント株式会社