◆リバースエンジニアリングチャレンジ 2008 解答

■はじめに

ここで示すものは、あくまでも解答へ辿り着くためのひとつの方法にすぎない。より早く、よりエレガントな解答方法が存在するかもしれないという前提で、当テキストを参照していただきたい。

■LEVEL1 解答

level1.bin は Windows の実行ファイル（PE フォーマット）であり、Windows 環境下で実行でき、OllyDbg で解析できる。このプログラムは CUI アプリケーションであるため、コマンドプロンプトから実行する必要がある。

main 関数（実質的なエントリポイント）はメモリアドレス「00401110」以下となる。

```
-----
00401110  /$ 837C24 04 03   CMP DWORD PTR SS:[ESP+4],3
00401115  |. 74 06          JE SHORT level1.0040111D
00401117  |. B8 01000000    MOV EAX,1
0040111C  |. C3             RETN
-----
```

[ESP+4]は main 関数へ渡される引数の数(プログラム自身のファイル名も含まれる)で、引数の数が 3 ではないならば、1 を返してプログラムを終了する。引数の数が 3 つなら、以下の処理へ進む。

```
-----
0040111D  |> 8B4424 08      MOV EAX,DWORD PTR SS:[ESP+8]
00401121  |. 56             PUSH ESI
00401122  |. 57             PUSH EDI
00401123  |. 8B78 04        MOV EDI,DWORD PTR DS:[EAX+4]
00401126  |. BE FC704000    MOV ESI,level1.004070FC ;  ASCII "good luck"
```

```
0040112B  |. B9 0A000000    MOV ECX,0A
00401130  |. 33D2           XOR EDX,EDX
00401132  |. F3:A6          REPE CMPS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI]
00401134  |. 5F             POP EDI
00401135  |. 5E             POP ESI
00401136  |. 75 0C          JNZ SHORT level1.00401144
-----
```

第 1 引数を取得して、その文字列と「good luck」という文字列を比較し、同じなら次の処理へ進み、異なるならプログラムを終了させる処理となる。アドレス 00401132 の REPE 命令が比較処理で、00401136 の JNZ 命令が分岐処理。よって、プログラムの第 1 引数には「good luck」を渡さなければならない。

```
-----
00401138  |. 8B40 08        MOV EAX,DWORD PTR DS:[EAX+8]
0040113B  |. 50             PUSH EAX
0040113C  |. E8 4FFFFFFF    CALL level1.00401090
00401141  |. 83C4 04        ADD ESP,4
00401144  |> 33C0           XOR EAX,EAX
00401146  ¥. C3             RETN
-----
```

今度は、プログラムの第二引数をスタックへ格納して、アドレス 00401090 へジャンプしている。00401141 以降がプログラム終了処理であるため、重要な部分は、おそらくジャンプ先の 00401090 以降で確認できると推測できる。

```
-----
00401090  /$ 56             PUSH ESI
00401091  |. 8B7424 08       MOV ESI,DWORD PTR SS:[ESP+8]
00401095  |. 803E 00         CMP BYTE PTR DS:[ESI],0
00401098  |. 74 14           JE SHORT level1.004010AE
0040109A  |. 8BC6           MOV EAX,ESI
0040109C  |. 8D6424 00       LEA ESP,DWORD PTR SS:[ESP]
004010A0  |> 8A08           /MOV CL,BYTE PTR DS:[EAX]
004010A2  |. F6D1           |NOT CL
004010A4  |. 8808           |MOV BYTE PTR DS:[EAX],CL
```

```
004010A6  |. 8A48 01        |MOV CL,BYTE PTR DS:[EAX+1]
004010A9  |. 40             |INC EAX
004010AA  |. 84C9           |TEST CL,CL
004010AC  |.^75 F2          ¥JNZ SHORT level1.004010A0
-----
```

00401098 までは、スタック経由で呼び出し元から渡された値（引数）が 0 なら関数を終了する処理。それ以降を見ると、引数から渡された文字列を 1 バイトずつ NOT し、TEST 命令により、0 が見つかればそこでループを脱出する。

```
-----
004010AE  |> 8A06           MOV AL,BYTE PTR DS:[ESI]
004010B0  |. 33D2           XOR EDX,EDX
004010B2  |. 84C0           TEST AL,AL
004010B4  |. 74 52          JE SHORT level1.00401108
004010B6  |. 8BCE           MOV ECX,ESI
004010B8  |> 0FB6C0         /MOVZX EAX,AL
004010BB  |. 03D0           |ADD EDX,EAX
004010BD  |. 8A41 01        |MOV AL,BYTE PTR DS:[ECX+1]
004010C0  |. 41             |INC ECX
004010C1  |. 84C0           |TEST AL,AL
004010C3  |.^75 F3          ¥JNZ SHORT level1.004010B8
-----
```

第 2 引数を 1 バイト単位で NOT にかけたあと、そのデータ列を 1 バイト単位で区切り、すべてを加算する。そして、加算結果を EDX に入れる。

```
-----
004010C5  |. 81FA B7070000  CMP EDX,7B7
004010CB  |. 75 3B          JNZ SHORT level1.00401108
-----
```

加算結果 EDX が 07B7h でないならばプログラムを終了する。つまり、入力値の合否判定をここで行っている。

```
-----
```

```
004010CD  |. 56             PUSH ESI
004010CE  |. E8 2DFFFFFF    CALL level1.00401000
-----
```

第２引数の１バイトずつを NOT したデータをスタックへ入れ、00401000 へジャンプする。

```
-----
00401000  /$ 83EC 10        SUB ESP,10
00401003  |. A1 40904000    MOV EAX,DWORD PTR DS:[409040]
00401008  |. 894424 0C      MOV DWORD PTR SS:[ESP+C],EAX
0040100C  |. B0 93          MOV AL,93
0040100E  |. B1 90          MOV CL,90
00401010  |. 53             PUSH EBX
00401011  |. 56             PUSH ESI
00401012  |. 8B7424 1C      MOV ESI,DWORD PTR SS:[ESP+1C]
00401016  |. 884424 0A      MOV BYTE PTR SS:[ESP+A],AL
0040101A  |. 884424 0B      MOV BYTE PTR SS:[ESP+B],AL
0040101E  |. 884424 11      MOV BYTE PTR SS:[ESP+11],AL
00401022  |. 884C24 0C      MOV BYTE PTR SS:[ESP+C],CL
00401026  |. 884C24 0F      MOV BYTE PTR SS:[ESP+F],CL
0040102A  |. 8D4424 08      LEA EAX,DWORD PTR SS:[ESP+8]
0040102E  |. 33C9           XOR ECX,ECX
00401030  |. C64424 08 B7   MOV BYTE PTR SS:[ESP+8],0B7
00401035  |. C64424 09 9A   MOV BYTE PTR SS:[ESP+9],9A
0040103A  |. C64424 0D DF   MOV BYTE PTR SS:[ESP+D],0DF
0040103F  |. C64424 0E A8   MOV BYTE PTR SS:[ESP+E],0A8
00401044  |. C64424 10 8D   MOV BYTE PTR SS:[ESP+10],8D
00401049  |. C64424 12 9B   MOV BYTE PTR SS:[ESP+12],9B
0040104E  |. C64424 13 DE   MOV BYTE PTR SS:[ESP+13],0DE
-----
```

00401000 以降では、まず特定の固定値を作成している。この固定値は、[ESP+8]～[ESP+13]までの 12 バイトのデータ列である。

```
-----
```

```
00401053  |. 2BF0           SUB ESI,EAX
00401055  |> 8A5C0C 08      /MOV BL,BYTE PTR SS:[ESP+ECX+8]
00401059  |. 8D440C 08      |LEA EAX,DWORD PTR SS:[ESP+ECX+8]
0040105D  |. 8A1406         |MOV DL,BYTE PTR DS:[ESI+EAX]
00401060  |. 3AD3           |CMP DL,BL
00401062  |. 75 1A          |JNZ SHORT level1.0040107E
00401064  |. 41             |INC ECX
00401065  |. 83F9 0C        |CMP ECX,0C
00401068  |.^7C EB          ¥JL SHORT level1.00401055
-----
```

その固定値列（ESP+8）と、さきほど１バイトずつ NOT したデータ列（ESI）を比較し、すべて同じならば以降の処理へ進む（00401060）。よって、その固定値列を１文字ずつ NOT したデータが２番目のキーワードだと考えられる。

固定値は「B7 9A 93 93 90 DF A8 90 8D 93 9B DE」なので、これを NOT したデータは、以下になる。

```
-----  ans1.cpp
#include <stdio.h>
int main()
{
        int i;
        unsigned char p[32] = {
                0xB7, 0x9A, 0x93, 0x93, 0x90, 0xDF, 0xA8, 0x90,
                0x8D, 0x93, 0x9B, 0xDE, 0x00
        };
        for(i=0; i < 0x0C; i++)
                p[i] ^= 0xFF;
        printf("bin: ");
        for(i=0; i < 0x0C; i++)
                printf("%02X", p[i]);
        printf("¥n");
        printf("str: %s¥n", p);
        return 0;
}
-----
```

```
-----  コマンドプロンプト
C:¥>ans1.exe
bin: 48656C6C6F20576F726C6421
str: Hello World!
-----
```

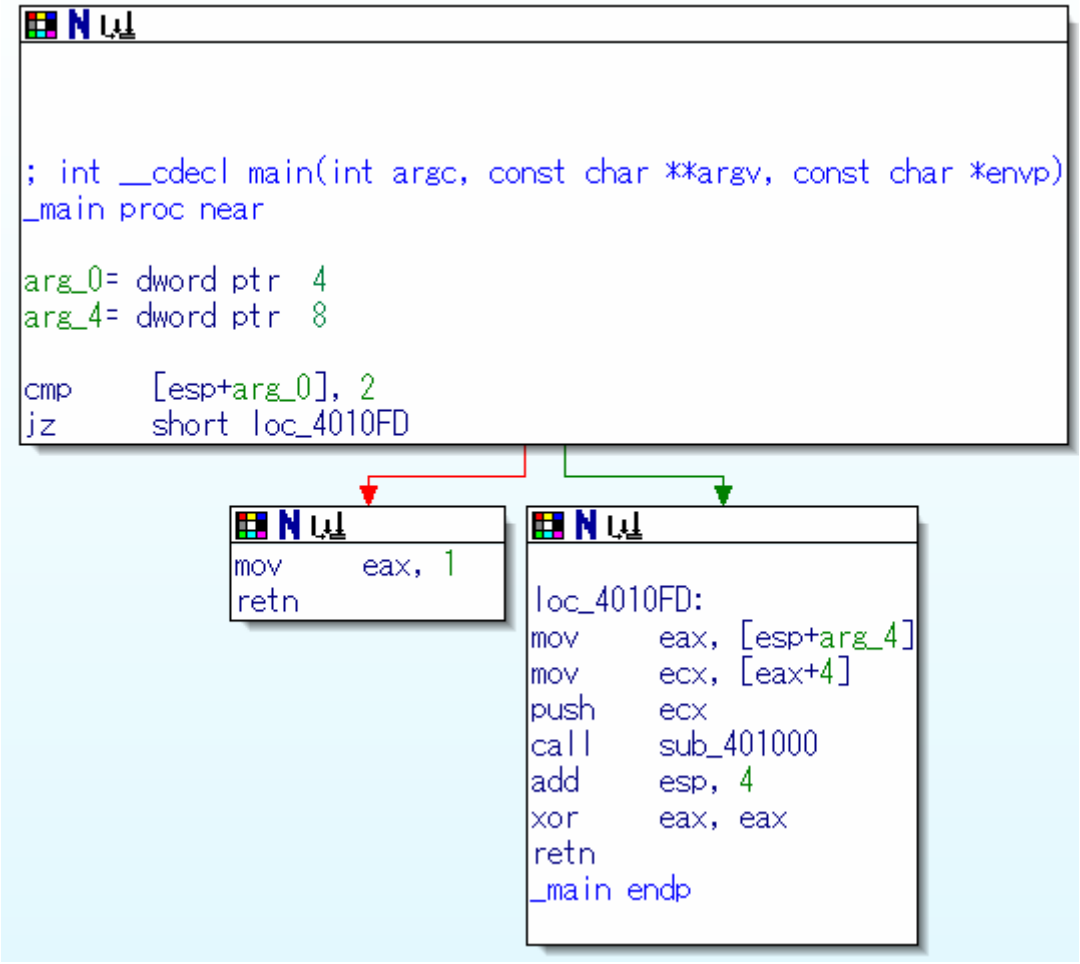
よって、プログラムに渡すべき 1 番目の引数が「good luck」、2 番目の引数が「Hello World!」となる。それらを引数に渡すと、以下のパスワードが得られる。

```
-----  コマンドプロンプト
C:¥>level1.exe "good luck" "Hello World!"
password: Ifmmp!Xpsme"
-----
```

よって、LEVEL2 へ進むためのパスワードは「Ifmmp!Xpsme"」となる。

■LEVEL2 解答

level2.bin は Windows の実行ファイル（PE フォーマット）であり、Windows 環境下で実行できる。まずは IDAPro で解析する。

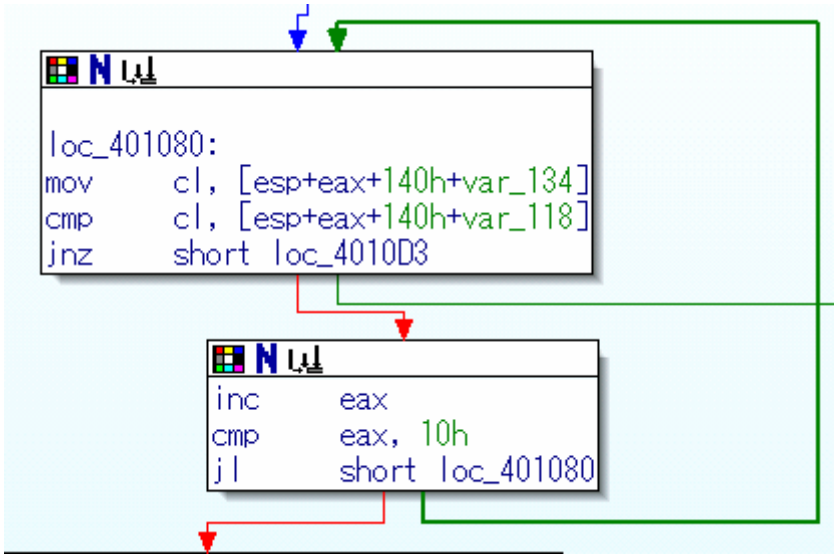


プログラムは引数をとり、その引数を sub_401000 関数に渡している。

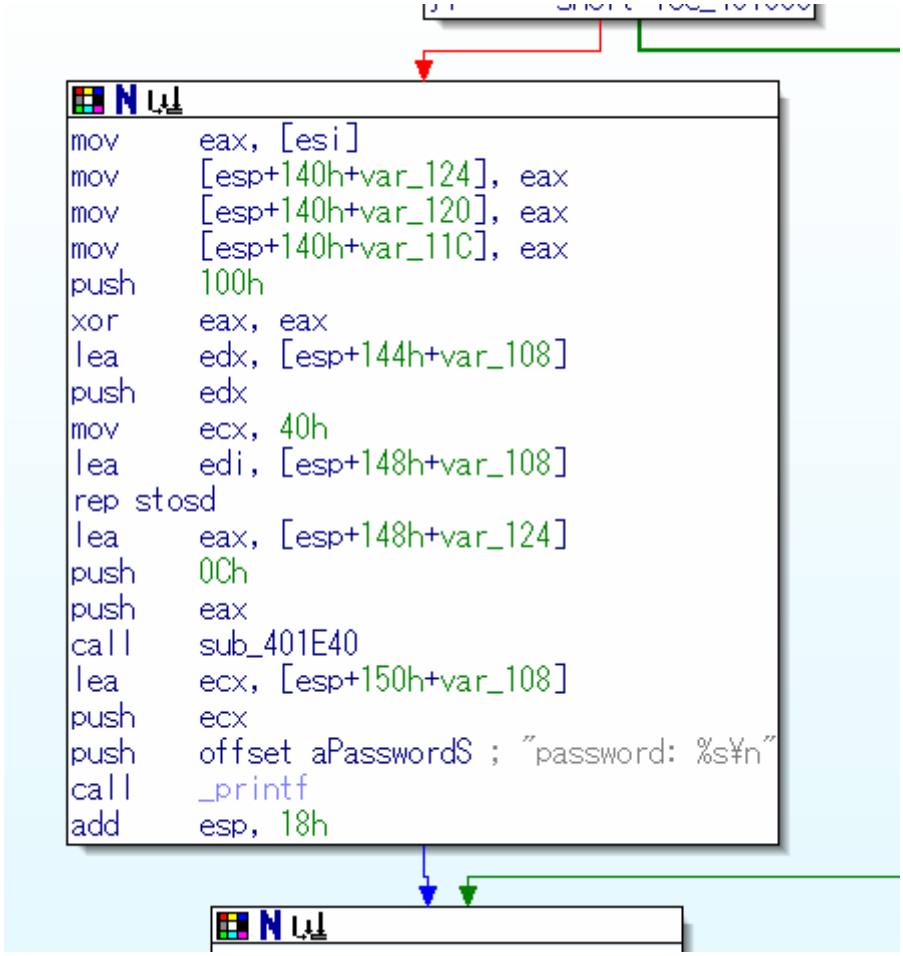
そして、sub_401000 関数へ進むと、以下のコードになっている。

```
push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF8h
sub     esp, 138h
mov     eax, dword_409040
push    esi
mov     esi, [ebp+arg_0]
mov     [esp+13Ch+var_4], eax
push    edi
lea     eax, [esp+140h+var_118]
push    eax
push    4
push    esi
mov     [esp+14Ch+var_134], 43h
mov     [esp+14Ch+var_133], 2Bh
mov     [esp+14Ch+var_132], 0F7h
mov     [esp+14Ch+var_131], 0F4h
mov     [esp+14Ch+var_130], 49h
mov     [esp+14Ch+var_12F], 90h
mov     [esp+14Ch+var_12E], 0A3h
mov     [esp+14Ch+var_12D], 75h
mov     [esp+14Ch+var_12C], 7Fh
mov     [esp+14Ch+var_12B], 0D6h
mov     [esp+14Ch+var_12A], 0EEh
mov     [esp+14Ch+var_129], 19h
mov     [esp+14Ch+var_128], 0B8h
mov     [esp+14Ch+var_127], 78h
mov     [esp+14Ch+var_126], 81h
mov     [esp+14Ch+var_125], 51h
call    sub_401C80
add     esp, 0Ch
xor     eax, eax
```

まず目立つのは、ローカル変数の 134h から 125h までの 10h バイトに、固定値データ列「43 2B F7 F4 49 90 A3 75 7F D6 EE 19 B8 78 81 51」が入っていること。さらに、sub_401C80 関数が呼ばれており、そのための引数が 3 つ、スタックへ格納されている。EAX はローカル変数（118h 以降）の 10h バイトの領域（結果の格納先）、4 はおそらくサイズ、そして ESI は、sub_401000 関数へ渡された（プログラムに渡された）データ列。sub_401C80 の内部を解析すると、どうやら何かのハッシュ生成関数だと推測できる（MD5 ライクな関数である）。sub_401C80 実行後には 118h 以降に 16 バイトのハッシュ値が入る。



入力値の「ハッシュ値」と「固定データ列」を比較、もし 10h バイトの中の 1 バイトでも異なっていたらプログラムを終了する。もし、すべて一致したら、次の処理へ進む。



ここは、ハッシュ値と、固定データ列がイコールとなった場合にのみ進む処理である。EAX に引数の先頭 4 バイトを格納し、4 バイトのローカル変数 124h、120h、11Ch にそれぞれ EAX を入れる。

sub_401E40 関数（内部を解析すると base64 生成関数だと分かる）は、push 命令の数から 4 つの引数をとる。1 つ目は、先頭 4 バイトのデータを 3 回コピーして作成された 12 バイトのデータ列。2 つ目は、12（0Ch）、3 つ目はスタックに確保されたローカル変数領域のアドレス、4 つ目は、100h。

1 つ目の引数のサイズが、4 バイト×3 つ分なので 0Ch バイト、それを base64 で文字列に変換するため、出力されるパスワードは、10h バイトだと分かる。

しかし、そもそもプログラムへの入力値の先頭 4 バイトからパスワードが生成されているため、このままではパスワードは求められない。分かっていることは、正確な入力値 4 バイトを sub_401C80 関数にかけたら、固定データ列「43 2B F7 F4 49 90 A3 75 7F D6 EE 19 B8 78 81 51」が得られるということ。

ハッシュ値の元となっているデータサイズは 4 バイトという小さいサイズであるため、おそらく総当りによるパスワードクラックが可能。よって、0x00000000～0xFFFFFFFF までの数値を総当りするツールを作成する。

ちなみに、このプログラムで使われている関数 sub_401C80 は、処理を解析すると MD5 のように思えるが、実際は純粋な MD5 アルゴリズムではないため、他から MD5 ライブラリを持ってきても正しい結果が得られない。よって、この level2.exe プログラム自体を、マッピングし、関数を呼び出す機構を作る必要がある。

以下がパスワード取得ツールのソースコードとなる。

```
-----  level2_crack.cpp

#include <windows.h>

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

int main()
{
        FILE *fp;

        BYTE ok[16] = {
```

```
        0x43, 0x2B, 0xF7, 0xF4, 0x49, 0x90, 0xA3, 0x75,
        0x7F, 0xD6, 0xEE, 0x19, 0xB8, 0x78, 0x81, 0x51,
    };

    if((fp = fopen("level2.exe", "rb")) == NULL)
        return -1;

    PBYTE bExeData = (PBYTE)VirtualAlloc(
        NULL, 1024 * 1024, MEM_COMMIT, PAGE_READWRITE);
if(bExeData == NULL){
        fclose(fp);
        return -1;
    }

    int c, i;
    for(i=0; (c = fgetc(fp)) != EOF; i++)
        bExeData[i] = (BYTE)c;

    fclose(fp);

    PBYTE func = bExeData + 0x00001C80;

    for(DWORD a=0; a != 0xFFFFFFFF; a++){

        BYTE hash[16];
        BYTE *h = hash;
        DWORD b = a;
        DWORD *p = &b;

        __asm{
            //int 3h
            push h
            push 4
            push p
            call func
            add esp, 0Ch
```

```
                };

                if(a%0x01000000 == 0)
                        printf("%08X¥n", a);

                for(i=0; i < 16; i++){
                        if(ok[i] != hash[i])
                                break;
                        else
                                if(i == 15)
                                        printf("KEY: %08X¥n", a);
                }
        }

        VirtualFree(bExeData, 1024 * 1024, MEM_DECOMMIT);
        return 0;
}
```

0x00001C80 は、sub_401C80 関数へのオフセット。

このままでは、実行しても level2.exe の中にあるセキュアクッキー（カナリア）に引っかかるため、その部分だけ、バイナリエディタで NOP に変更しておく必要がある。

・level2.exe の 00001C83 からの 5 バイトを 90 に変更
・level2.exe の 00001CF1 からの 5 バイトを 90 に変更
・level2.exe の 00001AB3 からの 5 バイトを 90 に変更
・level2.exe の 00001C73 からの 5 バイトを 90 に変更

このように変更した level2.exe ファイルと、上記のツールを同じフォルダに入れ、上記のプログラムを実行することで、正当な 4 バイトの値が判明する。

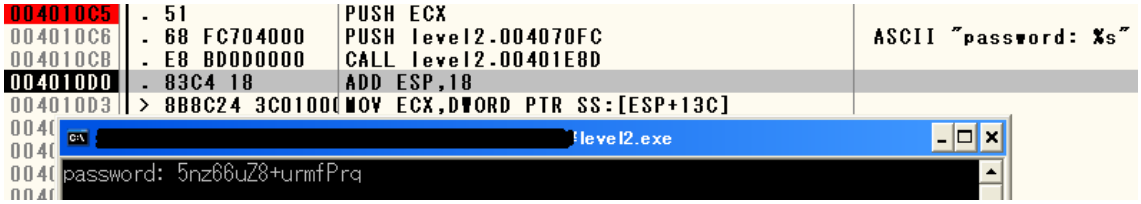
---　コマンドプロンプト
（省略）
14000000
15000000
KEY: 15058319

16000000
（省略）

今度は、level2.exe を OllyDbg で開き、適当な引数をつけて、00401024 にブレイクポイントを仕掛け、実行する。

```
-----  level2.exe
00401021   |. 50              PUSH EAX
00401022   |. 6A 04           PUSH 4
00401024   |. 56              PUSH ESI
-----
```

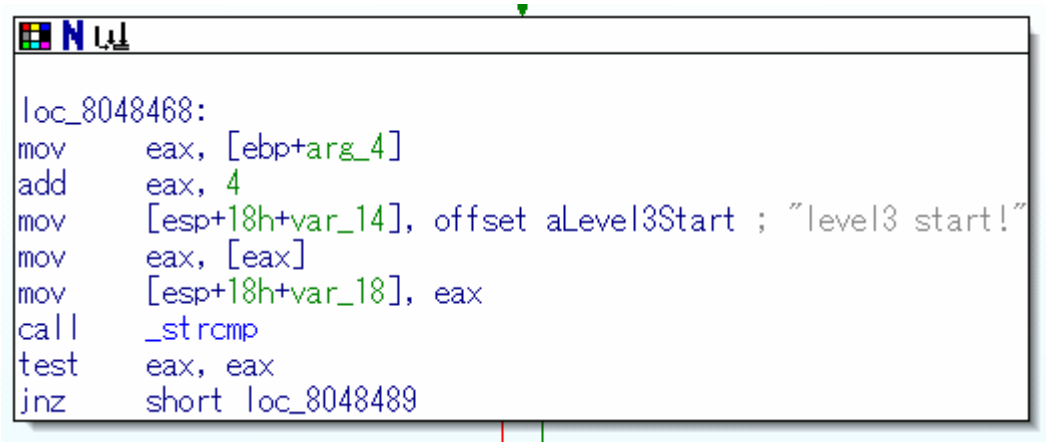
ESI には、元となるデータのアドレスが入っているため、ESI が指しているアドレス先の値を、得られたキー（15058319）に変更する。これで実行を継続するとパスワードが得られる。



LEVEL3 へのパスワードは「5nz66uZ8+urmfPrq」となる。

■LEVEL3 解答

level3.bin は Linux の実行ファイル（ELF フォーマット）であり、Linux 環境下で実行できる。まずは IDAPro で解析する。



main 関数の最初の部分で、strcmp を呼び出している。ここで、「level3 start!」という文字列と入力データを比較しているため、まずは、「level3 start!」という文字列が 1 つ目のキーワード。これを Linux 環境で実行する。

```
-----
# ./level3 'level3 start!' > level3_1.bin
-----
```

大量のバイナリデータが出力されるため、それらを level3_1.bin ファイルへ保存する。今度はその出力ファイルが何かを調べる。

```
-----
# file level3_1.bin
level3_1.bin: POSIX tar archive
-----
```

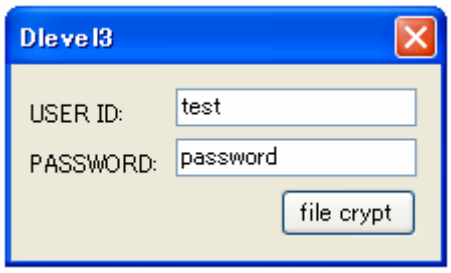
file コマンドより、tar で固められたアーカイブであることが分かる。展開すると以下のファイル群が表示される。

```
-----
# tar xf level3_2.tar.gz
# cd Dlevel3/
# ls
Dlevel3c  Dlevel3p.exe
-----
```

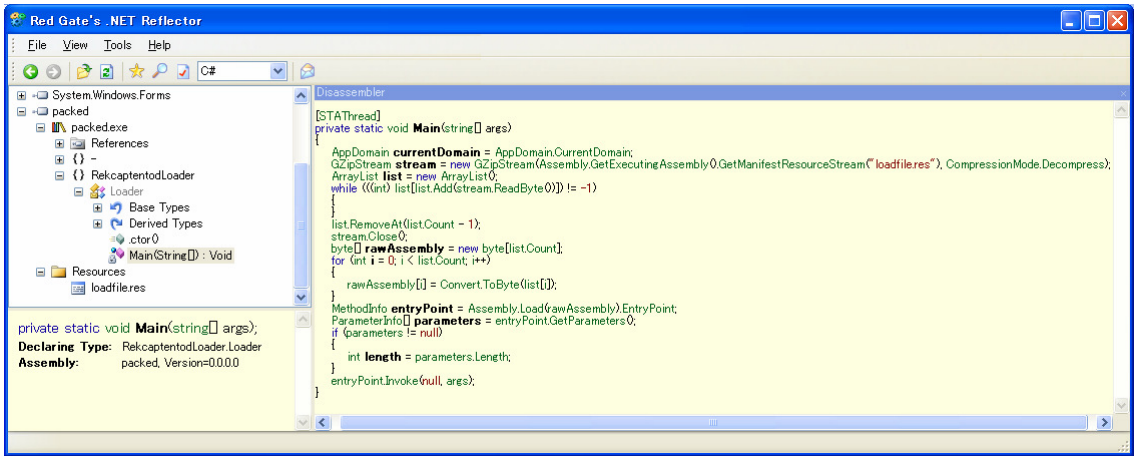
生成された 2 つのファイルを、さらに file コマンドで調べる。

```
-----
# file Dlevel3p.exe
Dlevel3p.exe: MS Windows PE 32-bit Intel 80386 GUI executable
# file Dlevel3c
Dlevel3c: data
-----
```

Dlevel3p.exe は Windows 環境で動作する通常の実行ファイルであり、Dlevel3c はただのデータファイルだと分かる。



Windows 上で Dlevel3p.exe を実行すると、上記のようなウィンドウが表示される。
.NET アプリケーションなので.NET デコンパイラ（Reflector）でソースコードを表示する。



Main 関数を見ると、以下のソースコードになっている。

```
-----
[STAThread]
private static void Main(string[] args)
{
    AppDomain currentDomain = AppDomain.CurrentDomain;
    GZipStream stream = New GZipStream(
        Assembly.GetExecutingAssembly().GetManifestResourceStream(
        "loadfile.res"), CompressionMode.Decompress);
    ArrayList list = new ArrayList();
    while (((int) list[list.Add(stream.ReadByte())]) != -1)
    {
    }
    list.RemoveAt(list.Count - 1);
    stream.Close();
    byte[] rawAssembly = new byte[list.Count];
    for (int i = 0; i < list.Count; i++)
    {
        rawAssembly[i] = Convert.ToByte(list[i]);
    }
    MethodInfo entryPoint = Assembly.Load(rawAssembly).EntryPoint;
    ParameterInfo[] parameters = entryPoint.GetParameters();
    if (parameters != null)
    {
        int length = parameters.Length;
    }
    entryPoint.Invoke(null, args);
}
-----
```

ソースコードを読むと、loadfile.res ファイルが読み込まれ、gzip で展開されて、その展開されたコードが実行されていることが分かる。つまり、loadfile.res は zip 圧縮されたファイルである。よって、loadfile.res を exe ファイルから取り出し、展開して本体のプログラムを作成する。この作成された exe ファイルがプログラムの実体となる。

よって、今度はこのプログラムに対して、デコンパイルを行う。

Disassembler

```
private void button1_Click(object sender, EventArgs e)
{
    if ((this.textBox1.Text != "") && (this.textBox2.Text != ""))
    {
        int hashCode = this.textBox1.Text.GetHashCode();
        int num2 = this.textBox2.Text.GetHashCode();
        int num3 = hashCode ^ num2;
        if (File.Exists("Dlevel3.zip"))
        {
            int num4;
            FileStream stream = new FileStream("Dlevel3.zip", FileMode.Open, FileAccess.Read);
            FileStream stream2 = new FileStream("Dlevel3c", FileMode.Create, FileAccess.Write);
            for (int i = 0; (num4 = stream.ReadByte()) != -1; i++)
            {
                switch ((i % 4))
                {
                    case 0:
                        stream2.WriteByte((byte) (((byte) (num4 & 0xff)) ^ ((byte) (num3 & 0xff))));
                        break;

                    case 1:
                        stream2.WriteByte((byte) (((byte) (num4 & 0xff)) ^ ((byte) ((num3 >> 8) & 0xff))));
                        break;

                    case 2:
                        stream2.WriteByte((byte) (((byte) (num4 & 0xff)) ^ ((byte) ((num3 >> 0x10) & 0xff))));
                        break;

                    case 3:
                        stream2.WriteByte((byte) (((byte) (num4 & 0xff)) ^ ((byte) ((num3 >> 0x18) & 0xff))));
                        break;
                }
            }
            stream.Close();
            stream2.Close();
        }
    }
}
```

Dlevel3c が作られる過程を示したプログラムが表示される。どうやら、この Dlevel3c というファイルは、Dlevel3.zip というファイルを暗号化して作られたものだと推測できる。そして、暗号化の方法は、xor により使用されている 4 バイトのデータ列であるため、この 4 バイトデータ列が分かれば、Dlevel3c を Dlevel3.zip に変換できる。

zip ファイルの先頭 4 バイトは、マジックワードとなっており、必ず「50 4B 03 04」である。また、Dlevel3c の先頭 4 バイトは「06 97 A5 5F」である。つまり、「50 4B 03 04」を何かしらのデータで xor した結果が、「06 97 A5 5F」となる。ということは、「50 4B 03 04」と「06 97 A5 5F」を xor した結果が、今回の xor の元データとなる。そのパスワードで、Dlevel3c ファイルを 4 バイトずつ xor していくことで、Dlevel3.zip が得られる。

以下に、zip ファイルへの変換コードを示す。

```
-----  ans1.cpp
#include <stdio.h>
```

```
int main(void)
{
        int c, i;
        FILE *fpr, *fpw;

        unsigned char key[4] = {
                0x06 ^ 0x50,  0x97 ^ 0x4B,
                0xA5 ^ 0x03,  0x5F ^ 0x04,
        };

        if((fpr = fopen("Dlevel3c", "rb")) == NULL)
                return -1;

        if((fpw = fopen("Dlevel3.zip", "wb")) == NULL){
                fclose(fpr);
                return -1;
        }

        for(i=0; (c = fgetc(fpr)) != EOF; i++)
                fputc((int)(((unsigned char)c) ^ key[i % 4]), fpw);

        fclose(fpr);
        fclose(fpw);
        return 0;
}
-----
```

これで生成された zip ファイルを展開すると、2 つのファイルが生成される。

ひとつが、Java のクラスファイルで、もうひとつがデータファイルである。
class ファイルを jad によりデコンパイルすると、以下のソースコードが得られる。

```
-----
// 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:   Jlevel3.java

import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.DESKeySpec;

public class Jlevel3
{

    public Jlevel3()
    {
    }

    public static void main(String args[])
    {
        try
        {
            int dlen = 0;
            byte input[] = new byte[16384];
            try
            {
                FileInputStream in = new FileInputStream("Jlevel3");
                int ch;
                int i;
                for(i = 0; (ch = in.read()) != -1; i++)
                    input[i] = (byte)ch;

                dlen = i;
                in.close();
            }
            catch(IOException e)
```

```
            {
                System.out.println(e);
            }
            String key = "I love coffee";
            DESKeySpec dk = new DESKeySpec(key.getBytes());
            SecretKeyFactory kf = SecretKeyFactory.getInstance("DES");
            SecretKey sk = kf.generateSecret(dk);
            Cipher c = Cipher.getInstance("DES");
            c.init(1, sk);
            byte encrypted[] = c.doFinal(input, 0, dlen);
            try
            {
                FileOutputStream out = new FileOutputStream("Jlevel3c");
                out.write(encrypted);
                out.close();
            }
            catch(IOException e)
            {
                System.out.println(e);
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

どうやら、Jlevel3c を作成した過程を示したプログラムであることが分かる。また、今度は「I love coffee」という文字列をキーとして、暗号アルゴリズム DES を使用しているようだ。よって、DES の復号プログラムを作成する。

```
// 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:   Jlevel3.java

import javax.crypto.spec.*;
import java.io.*;
import javax.crypto.*;
import javax.crypto.spec.DESKeySpec;
import java.security.*;

public class Jlevel3
{

    public Jlevel3()
    {
    }

    public static void main(String args[])
    {
        try
        {
            int dlen = 0;
            byte input[] = new byte[16384];
            try
            {
                FileInputStream in = new FileInputStream("Jlevel3c");
                int ch;
                int i;
                for(i = 0; (ch = in.read()) != -1; i++)
                    input[i] = (byte)ch;

                dlen = i;
                in.close();
            }
            catch(IOException e)
            {
                System.out.println(e);
```

```
            }
            String key = "I love coffee";
            DESKeySpec dk = new DESKeySpec(key.getBytes());
            SecretKeyFactory kf = SecretKeyFactory.getInstance("DES");
            SecretKey sk = kf.generateSecret(dk);
            Cipher c = Cipher.getInstance("DES");
            AlgorithmParameters iv = c.getParameters();
            c.init(Cipher.DECRYPT_MODE, sk, iv);
            byte decrypted[] = c.doFinal(input, 0, dlen);
            try
            {
                FileOutputStream out = new FileOutputStream("Jlevel3");
                out.write(decrypted);
                out.close();
            }
            catch(IOException e)
            {
                System.out.println(e);
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

クラスファイルのデコンパイルにより取得できたソースコードに多少の変更を加えるだけで、復号プログラムを作成できる。

Jlevel3c を復号したファイル、Jlevel3 は、file コマンドで調べると、zip ファイルであることが分かる。

```
# file Jlevel3
Jlevel3: Zip archive data, at least v2.0 to extract
```

よって、拡張子を変更し、zip 展開する。すると、Llevel3 ファイルが生成される。このファイルは、ELF であり、Linux の実行ファイルであることが分かる。

```
# file Llevel3
Llevel3: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.4.1,
dynamically linked (uses shared libs), not stripped
```

実行しても何も起きないため、IDAPro で調べる。すると、以下のコードが得られる。

```
.text:08048418                public main
.text:08048418 main           proc near               ; DATA XREF: _start+17↑o
.text:08048418
.text:08048418 var_4          = dword ptr -4
.text:08048418
.text:08048418                push    ebp
.text:08048419                mov     ebp, esp
.text:0804841B                sub     esp, 8
.text:0804841E                and     esp, 0FFFFFFF0h
.text:08048421                mov     eax, 0
.text:08048426                sub     esp, eax
.text:08048428                mov     [ebp+var_4], offset funcA
.text:0804842F                mov     eax, [ebp+var_4]
.text:08048432                call    eax
.text:08048434                mov     eax, 0
.text:08048439                leave
.text:0804843A                retn
.text:0804843A main           endp
```

funcA という名前の関数を、関数ポインタを使って呼び出している。その funcA を見ると、以下のコードになっている。

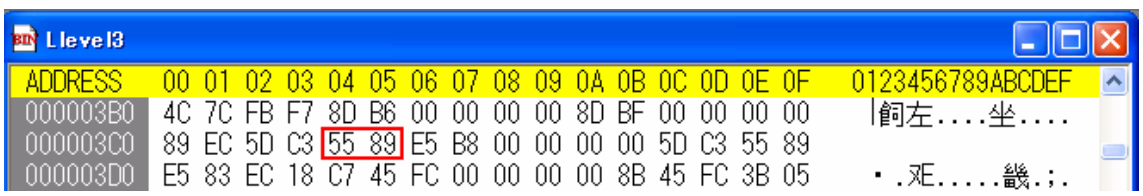
```
.text:080483C4                public funcA
.text:080483C4 funcA          proc near               ; DATA XREF: main+10↓o
.text:080483C4                push    ebp
.text:080483C5                mov     ebp, esp
.text:080483C7                mov     eax, 0
.text:080483CC                pop     ebp
.text:080483CD                retn
.text:080483CD funcA          endp
.text:080483CD
```

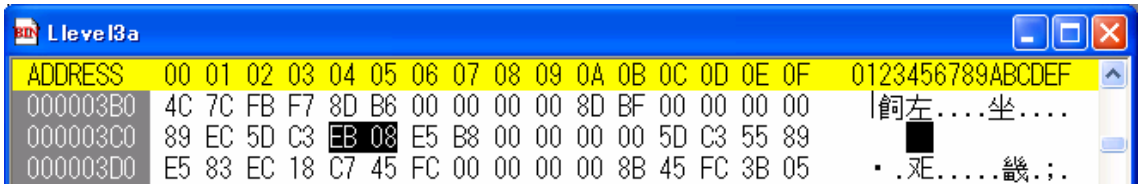
コードとしては何も処理していない。この funcA という関数のすぐ下に funcB という関数が存在する。

```
.text:080483CE funcB           proc near
.text:080483CE
.text:080483CE var_18          = dword ptr -18h
.text:080483CE var_14          = dword ptr -14h
.text:080483CE var_4           = dword ptr -4
.text:080483CE
.text:080483CE                 push    ebp
.text:080483CF                 mov     ebp, esp
.text:080483D1                 sub     esp, 18h
.text:080483D4                 mov     [ebp+var_4], 0
.text:080483DB
.text:080483DB loc_80483DB:                            ; CODE XREF: funcB+41↓j
.text:080483DB                 mov     eax, [ebp+var_4]
.text:080483DE                 cmp     eax, size
.text:080483E4                 jl      short loc_80483E8
.text:080483E6                 jmp     short loc_8048411
.text:080483E8 ; ---------------------------------------------------------------
.text:080483E8
.text:080483E8 loc_80483E8:                            ; CODE XREF: funcB+16↑j
.text:080483E8                 mov     eax, ds:stdout@@GLIBC_2_0
.text:080483ED                 mov     [esp+18h+var_14], eax
.text:080483F1                 mov     eax, [ebp+var_4]
.text:080483F4                 add     eax, start
.text:080483FA                 movzx   eax, byte ptr [eax]
.text:080483FD                 xor     al, 99h
.text:080483FF                 movzx   eax, al
.text:08048402                 mov     [esp+18h+var_18], eax
.text:08048405                 call    _fputc
.text:0804840A                 lea     eax, [ebp+var_4]
.text:0804840D                 inc     dword ptr [eax]
.text:0804840F                 jmp     short loc_80483DB
.text:08048411 ; ---------------------------------------------------------------
```

この関数がどうやら何かを行っていると予想できるため、funcA の代わりに funcB が実行されるようマシン語を書き換える。

funcA 関数の先頭に jmp 命令を置き、jmp 先を funcB 関数にする。





これで、funcA 関数の代わりに、funcB 関数が実行される。

```
# ./Llevel3a > level3.bin
# file level3.bin
level3.bin: PNG image data, 382 x 52, 8-bit/color RGB, non-interlaced
```

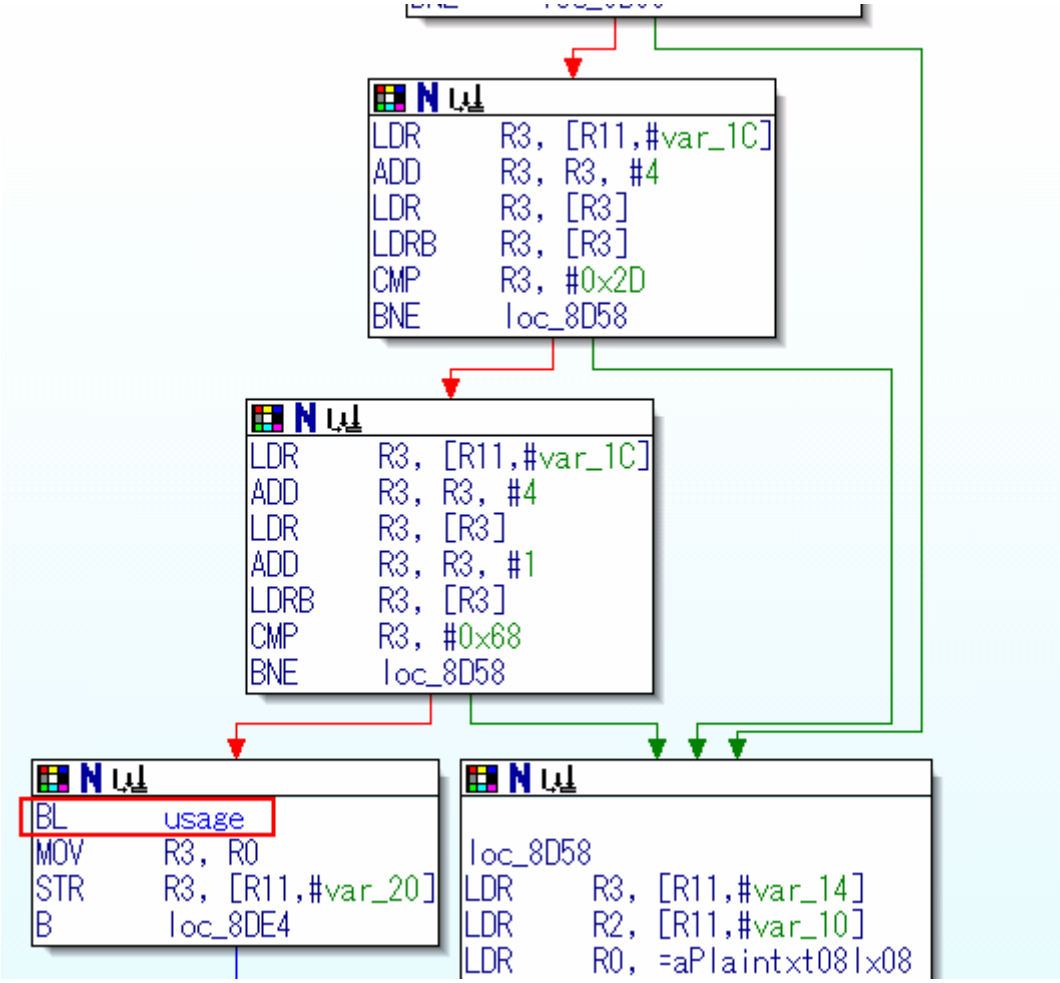
出力結果をファイルへ保存し、そのファイルの種類を調べる。すると png ファイルであることが分かる。その png ファイルを開くと、パスワードが表示される。

よって、LEVEL4 へのパスワードは「Do_you_like_ARM?」となる。

■LEVEL4 解答

level4.bin は ARM 上で動作する Linux 用バイナリ（ELF フォーマット）である。

まず、IDAPro で level4 のバイナリを開く。すると、ARM のマシン語で main 関数以降の処理が見える。シンボル情報（関数名）は消されていないため、blowfish という暗号アルゴリズムを用いていることが分かる。また、下へ進むと、usage が設定されている。



usage より前にある 2 つの分岐点により、引数に「-h」があれば、usage が実行されることが分かる。

```
-----
# ./level4 -h
password: XXXXXXXXXXXXXXX
cryption: 2F45DD54AD8CF0EF
password: XXXXXXXXXXXXXXX
-----
```

「-h」をつけると、上記の出力になる。「-h」をつけないと、以下の出力になる。

```
-----
# ./level4
```

```
plaintxt: 4142434445464748
cryption: 627F2D58606E17B4
plaintxt: 4142434445464748
-----
```

「4142434445464748」というデータ列を暗号化すると「627F2D58606E17B4」となり、それを復号すると「4142434445464748」に戻るという処理。そして、「-h」をつけた場合の出力は、暗号文が「2F45DD54AD8CF0EF」である場合の、平文がパスワードと考えられる。つまり、level4 の最終的なパスワードは、「2F45DD54AD8CF0EF」を復号することで求まる。

通常実行時の挙動から、復号処理もこのプログラムの中に入っていると考えられる。となると、復号処理を実行される直前のメモリ空間に「2F45DD54AD8CF0EF」を割り込ませ、この暗号文に対する平文を表示させるように、バイナリを変更する。

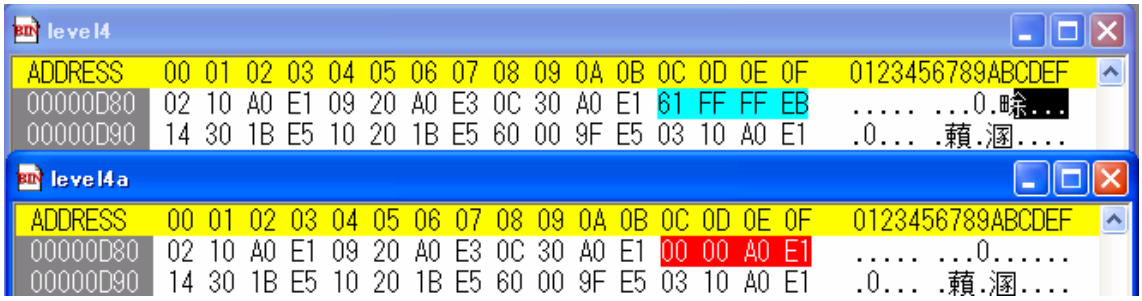
```
MOV     R1, R2
MOV     R2, #9
MOV     R3, R12
BL      blowfish
LDR     R3, [R11,#var_14]
LDR     R2, [R11,#var_10]
LDR     R0, =aCryption08lx08
MOV     R1, R3
BL      sub_83A4
LDR     R2, =aArmadillo
SUB     R12, R11, #0x14
MOV     R3, #8
STR     R3, [SP,#0x28+var_28]
MOV     R0, #0
MOV     R1, R2
MOV     R2, #9
MOV     R3, R12
BL      blowfish
LDR     R3, [R11,#var_14]
LDR     R2, [R11,#var_10]
LDR     R0, =aPlaintxt08lx08
MOV     R1, R3
BL      sub_83A4
MOV     R2, #0
STR     R2, [R11,#var_20]
```

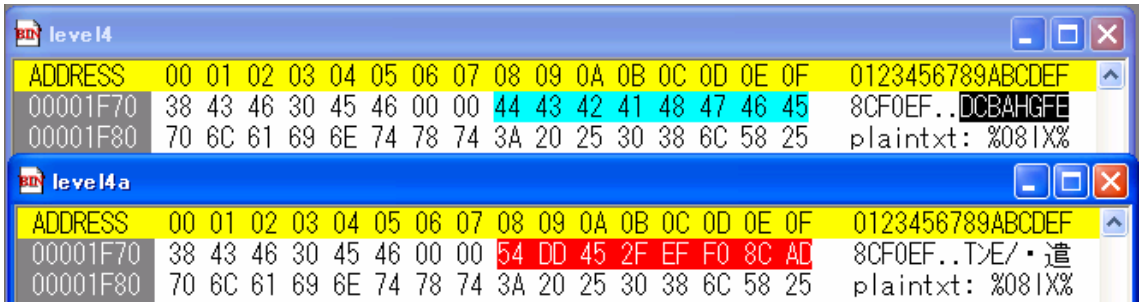
まず、2 度呼び出される blowfish 関数の最初の 1 つを NOP に変更する。ここはおそらく、

最初が「暗号化」次が「復号」を担っていると考えられる。つまり、まず暗号化を行わないようにする。

次に、初期値「4142434445464748」を、復号したい暗号文「2F45DD54AD8CF0EF」に変更する。最初の暗号化は行われないため、初期値「2F45DD54AD8CF0EF」はそのままの形で、復号処理へ進むことができる。これで、ディスプレイに、「2F45DD54AD8CF0EF」の復号データが表示される。

これら 2 つを、バイナリレベルで書き換える。





このように変更したバイナリを再度実行する。

```
-----
# ./level4a
plaintxt: 2F45DD54AD8CF0EF
cryption: 2F45DD54AD8CF0EF
plaintxt: 41524D3741524D39
-----
```

平文「41524D3741524D39」が出力される。

よって、LEVEL4 の最終パスワードは「41524D3741524D39」である。

ちなみに、「41524D3741524D39」は、テキストで「ARM7ARM9」となる。

上記は ARM 環境がある場合での解説である。もし ARM 実行環境がない場合は、IDAPro で静的にアルゴリズムを解析する必要がある。その場合の注意点は、ここで使われている blowfish は通常の blowfish ではなく、若干の変更が加えられているため、静的解析により変更点を特定し、アルゴリズムを再現した上で復号処理を行う必要がある。