目次

はじめに 1

第 1 章　準備 **3**

プログラミング言語 3

本書の構成 3

第 2 章　ZIP の構成 **4**

概要 4

End of central directory record 5

CRC 8

Local file header/Central directory header 12

Deflate 圧縮 19

暗号化 36

第 3 章　ZIP の暗号に対する攻撃 **41**

PkCrack 42

Biham と Kocher の攻撃法 42

圧縮されたファイルに対する既知平文攻撃 58

Stay の攻撃法 61

第 1 章

準備

プログラミング言語

本文中のコードは C++ で記述しています。 Visual C++ 2017 と、GCC 4.8.4 でコンパイルできることを確認しています。C++11 の機能を使用しているので、GCC や Clang でコンパイルする際には`-std=c++11` オプションを追加してください。

本文中のコード断片は、別のコード断片で定義した関数や変数を参照しています。コンパイルして動かす際には適宜コピーしてください。奥付のサイトで本文中のソースコードを公開しています。

本書の構成

2 章では、ZIP で使われている CRC や Deflate アルゴリズムの説明を交えながら、ZIP の構成を解説しています。

3 章では ZIP で用いられている暗号の攻撃方法を紹介しています。

自己解凍形式の ZIP では、1 個目のファイルの local file header の前にプログラムを置いて、OS が ZIP 自体を実行可能ファイルとして実行できるようにしています。先頭にプログラムがあっても仕様に則った ZIP なので、ファイルに置かれたプログラム以外の解凍ソフトでも解凍することができます。

さらに、end of central directory record の後に何らかのデータを置くこともできます。 次節で説明するように end of central directory はコメントを含んでいて可変長なので、そもそも解凍する際にファイル末尾から一定の距離を遡って end of central directory record とみなすことはできません。

End of central directory record には 1 個目のファイルの central directory header の位置が、各 central directory header には対応するファイルの local file header の位置が記録されています。解凍ソフトが ZIP を解凍するときには、この順で ZIP の中身を辿ります。ファイルの末尾に存在するとは限らないとなると、どのようにして central directory header の位置を知れば良いのでしょうか？ 仕様書にはその方法は書かれていません。ZIP の実装の一つである Info-Zip[*1]ではファイル全体から end of central directory record のシグネチャを探していました（zip 3.0 の zipfile.c を `ENDSIG` で検索すると処理が見つかります）。検索は末尾から行われていたので、やはり end of central directory record は末尾に置くと解凍するプログラムに優しいようです。

End of central directory record

Local file header と central directory record は ZIP 中の各ファイルに対応しているので、ファイルを 1 個も含まない空の ZIP は、end of central directory record を書き出すだけで作ることができます。

End of central directory record の構成は下表の通りです。

項目名	サイズ	説明
signature	4 bytes	`0x06054b50`
disk	2 bytes	このディスクが何番目か
centralDirDisk	2 bytes	central directory が開始するディスクの番号
entryNumber	2 bytes	このディスクの central directory 中のエントリー数
totalEntryNumber	2 bytes	central directory 中のエントリーの総数
centralDirSize	4 bytes	central directory のサイズ
centralDirOffset	4 bytes	central directory のファイル先頭からのオフセット

[*1] http://www.info-zip.org/

```
    uint32_t centralDirSize = 0;
    uint32_t centralDirOffset = 0;
    uint16_t commentLength = 0;
};

#pragma pack(pop)

int main()
{
    FILE *f = fopen("empty.zip", "wb");
    EndOfCentralDirectoryRecord end;
    fwrite(&end, sizeof end, 1, f);
    fclose(f);
}
```

```
00000000  50 4b 05 06 00 00 00 00  00 00 00 00 00 00 00 00  |PK..............|
00000010  00 00 00 00 00 00                                 |......|
00000016
```

後の実装で必要になるヘッダもここでまとめて`#include`しています。

`#pragma pack(push, 1)`はコンパイラが構造体のメンバーの間に隙間を空けることを抑止するためのものです。`#pragma`での指定に対応しているかどうかはコンパイラによって異なります。Visual C++ と GCC はこの指定を解釈します。また、ビッグエンディアンの環境ではファイルに書き出す前にエンディアンを変換する必要があります。

commentLength にコメントの長さを指定してコメントを書き出すことにより、ZIP にコメントを付けることができます。アーカイバによっては何らかの方法でこのコメントを表示します。

```
int main()
{
    FILE *f = fopen("comment.zip", "wb");
    EndOfCentralDirectoryRecord end;
    end.commentLength = 7;
    fwrite(&end, sizeof end, 1, f);
    fwrite("comment", 1, 7, f);
    fclose(f);
}
```

```
            int lsb = c&1;
            c = ((i<data.size() ? data[i]>>j&1 : 0) ^ (i<4 ? 1 : 0))<<31 | c>>1;
            if (lsb != 0)
                c ^= 0xedb88320;
        }
    return ~c;
}
```

入力データが処理に影響を与えるのは、最下位ビットに達したときなので読むのを遅延させることができます。入力データは除数と xor されます。c の初期値を 0xffffffff とすることで、入力データの先頭 32 bits を反転させる処理に変えることができます。

```
uint32_t crc_tmp3(vector<uint8_t> data)
{
    uint32_t c = 0xffffffffu;
    for (size_t i=0; i<data.size(); i++)
        for (int j=0; j<8; j++) {
            int lsb = c&1 ^ data[i]>>j&1;
            c >>= 1;
            if (lsb != 0)
                c ^= 0xedb88320;
        }
    return ~c;
}
```

このとき、内側のループの処理は c の下位 8 bits と data[i] の xor を取った値にしか依存していません。あらかじめ値を計算しておくことができます。

```
uint32_t crcTable[256];
void makeCrcTable()
{
    for (int i=0; i<256; i++) {
        uint32_t c = i;
        for (int j=0; j<8; j++)
            c = c>>1 ^ ((c&1)!=0 ? 0xedb88320 : 0);
        crcTable[i] = c;
    }
}
```

ば、圧縮方法として BZIP2 が使用可能になったのは 4.6 からなので、BZIP2 を使用した場合には 46 になります。この章で取り扱う圧縮や暗号化は 20 を指定すれば足ります。その他のバージョンは ZIP の仕様書を参照してください。

ファイルを暗号化したり、ファイルサイズの指定を後回しにしたりする場合には、flag の特定のビットを立てます。

compression は圧縮方法を示します。主に使用されるのは、0 の無圧縮と、8 の Deflate 圧縮です。

modifiedTime と modifiedDate には、MS-DOS のフォーマットで更新日時を指定します。特定の日時からの経過秒数ではなく、下表の通り各ビットに年や分などを設定します。CRC の算出とは異なり、最下位ビットを 0 としています。[*3]

	ビット	内容
modifiedDate	0-4	日（1-31）
	5-8	月（1-12）
	9-16	年-1980
modifiedTime	0-4	秒/2（0-29）
	5-10	分（0-59）
	11-15	時（0-23）

16 ビットに詰め込むために秒を 2 で割るので、更新時刻に奇数秒を指定することはできません。また、日時はローカル日時です。異なるタイムゾーンの環境に ZIP を持っていっても更新日時は変化しません（実際とは異なる日時にファイルが更新されたように見えます）。

Local file header の構成は次の通りです。

項目名	サイズ	説明
signature	4 bytes	0x02014b50
versionNeeded	2 bytes	解凍に必要なバージョン
versionMade	2 bytes	圧縮に使用されたソフトウェアのバージョン
flag	2 bytes	フラグ
compression	2 bytes	圧縮方法
modifiedTime	2 bytes	更新時刻
modifiedDate	2 bytes	更新日付

[*3] https://msdn.microsoft.com/ja-jp/library/cc429703.aspx

```
        local->tm_min<<5 |
        local->tm_sec/2;
    *dosDate =
        (local->tm_year + 1900 - 1980)<<9 |
        (local->tm_mon + 1)<<5 |
        local->tm_mday;
}

int main()
{
    makeCrcTable();

    string name[3] = {"file1.txt", "file2.txt", "file3.txt"};
    string content[3] = {"understand zip", "abc", ""};

    FILE *f = fopen("uncompressed.zip", "wb");

    uint16_t modTime, modDate;
    timeToDos(time(nullptr), &modTime, &modDate);

    CentralDirectoryHeader central[3];
    for (int i=0; i<3; i++) {
        central[i].versionMade = 0<<8 | 20;
        central[i].versionNeeded = 20;
        central[i].modifiedTime = modTime;
        central[i].modifiedDate = modDate;
        central[i].crc = crc(stringToUint8(content[i]));
        central[i].compressedSize = (uint32_t)content[i].size();
        central[i].uncompressedSize = (uint32_t)content[i].size();
        central[i].fileNameLength = (uint32_t)name[i].size();
        central[i].offset = (uint32_t)ftell(f);

        LocalFileHeader local;
        copyHeader(central[i], &local);

        fwrite(&local, sizeof local, 1, f);
        fwrite(name[i].c_str(), name[i].size(), 1, f);
        fwrite(content[i].c_str(), content[i].size(), 1, f);
    }

    EndOfCentralDirectoryRecord end;
    end.centralDirOffset = (uint32_t)ftell(f);
```

Deflate 圧縮

ZIP はいくつかの圧縮方法をサポートしていますが、主に Deflate 圧縮が用いられます。ZIP の仕様書には簡単にしか書かれていないので、詳細は Deflate の仕様書（RFC 1951）[*5]を読むと良いでしょう。

Deflate 圧縮では、(a) 同じ文字列が何度も出現することが多いことと、(b) 文字の出現頻度には偏りがあることに着目して圧縮します。

繰り返し出現する文字列の圧縮

`zip`（`122 105 112`）が 8 回繰り返されたデータを圧縮することを考えます。最初の 3 文字は過去に出現していない文字列なので、そのまま `122 105 112` を出力します。入力のまま出力される文字を Deflate では「リテラル」と呼びます。4 文字目以降の 21 文字は、「この先 21 文字は、3 文字前と同じである」と出力します。`122 105 112 (21, 3)`。この先何文字が一致しているか（この例では 21 文字）を「一致長」、何文字前と等しいか（この例では 3 文字前）を「一致距離」と言います。一致長と一致距離で出力する文字列と参照する文字列は重なることができます。この例でも、21 を出力した時点では参照する文字列は 3 文字しかありません。一致長と一致文字列で表された文字列を解凍する際には、（特別な処理をしない限り）先頭から 1 文字ずつ処理をする必要があります。

Deflate ではリテラルと一致長、さらにブロック（後述）が終了するかどうかのフラグををひとまとめにして、リテラル・一致長として取り扱います。リテラル・一致長が、256 未満ならばリテラル、256 ならばブロックの終了、257 以上ならば一致長となります。これによって、リテラルか一致長かを表す 1 ビットを削っています。

一致長は 3 から 258、一致距離は 1 から 32768 までの範囲を取ります。そのまま次項で説明するハフマン符号で取り扱うと、符号の数が多くなりすぎて効率が悪いので、拡張ビットを用いていくつかの値をまとめます。例えば、19 から 22 の一致長は 269 という同一のリテラル・一致長になり、4 通りの一致長を区別するため 2 ビットの拡張符号を直後に出力します。

リテラル・一致長（コード）と拡張ビットの長さ（拡張）や一致長の対応は下表の通りです。

[*5] https://www.ietf.org/rfc/rfc1951.txt

```
    return code;
}

int main()
{
    vector<string> code = huffman({1, 17, 2, 3, 5, 5});
    for (size_t i=0; i<code.size(); i++)
        cout<<i<<" "<<code[i]<<endl;
}
```

```
0 0000
1 1
2 0001
3 001
4 010
5 011
```

なお、このプログラム中の関数は Deflate 圧縮に使用することはできません。Deflate 圧縮ではビット列の長さに制限があるからです。例えば、`huffman` 関数に`{1, 2, 4, 8, 16, 32}`を渡すと 0 に 00000 という長いビット列が割り当てられることが確認できます。

ビット列の長さに上限 L があるときに最適なビット列の割り当てを求めるアルゴリズムとして、Package-merge が知られています[*7]。

n 個の文字の出現回数がそれぞれ $c_0, c_1, \ldots, c_{n-1}$ のとき、n 種類の出現回数に対応した「価値」と、$2^{-1}, 2^{-2}, \ldots, 2^{-L}$ の L 種類の「額面」を組み合わせた nL 枚のコインを用意します。なるべく「価値」が小さくなるように、額面の合計が $n-1$ になるようなコインの組み合わせを選びます。このとき、この組み合わせの中に i 番目の文字に対応するコインが h_i 枚含まれているならば、i 番目の文字に割り当てるべきビット列の長さは h_i となります。各文字に対応するコインは L 枚しかないため、ビット列の最大の長さは L です。

最適なコインを組み合わせは、額面 2^{-k} のコインを小さい順に 2 枚ずつセットにして 2^{-k+1} のコインとして扱うということを繰り返すことで求められます。

```
vector<int> packMerge(vector<int> count, int L)
```

[*7] https://en.wikipedia.org/wiki/Package-merge_algorithm

ブロック

Deflate ではデータはいくつかのブロックに分かれて格納されます。ブロックごとにハフマン符号表はリセットされますが、一致長と一致距離での参照はブロックを跨ぐことができます。例えば、1 個目のブロックが１ ２ ３で、2 個目のブロックの最初が (4, 2) のとき、2 個目のブロックは２ ３ ２ ３というデータを表しています。

各ブロックの先頭には、最後のブロックかどうかを示す 1 ビットのフラグ BFINAL と、ブロックの圧縮方法を示す 2 ビットの BTYPE があります。

BFINAL が 0 ならばこのブロック以降もブロックが続き、1 ならば終了です。

BTYPE が 00 のとき、ブロックは圧縮されていません。この場合、次のバイト境界までのビットは無視されます。その後、バイト単位でのデータサイズを示す 2 バイトの LEN と、LEN のビットを反転させた NLEN が続き、以降に LEN バイトのデータがそのまま格納されます。

BTYPE が 01 のとき、ブロックは各文字のビット長が仕様で規定された固定ハフマン符号表で圧縮されています。各文字のビット長は次の表の通りです。ハフマン符号表を送る必要が無いので、BTYPE の直後にハフマン符号で圧縮されたデータ本体が続きます。

リテラル・一致長コード	ビット長
0-143	8
144-255	9
256-279	7
280-287	8

一致距離コード	ビット長
0-31	5

リテラル・一致長コードの 286 と 287 と、一致距離コードの 30 と 31 は、ハフマン符号表の作成には使用しますが、圧縮の際に使用することはありません。

BTYPE が 11 のとき、データはカスタムハフマン符号表（圧縮する側が決めたハフマン符号表）で圧縮されます。ハフマン符号表は次のようにして送られます。

```
    3, 3, 3, 3, 4, 4, 4, 4,  5, 5, 5, 5, 0,
};
static int distExtTable[30] = {
    0, 0, 0, 0, 1, 1, 2, 2,  3, 3, 4, 4, 5, 5, 6, 6,
    7, 7, 8, 8, 9, 9,10,10, 11,11,12,12,13,13,
};

// 入力をリテラル・一致長と一致距離に圧縮する
vector<Code> literal;
vector<Code> dist;
// 添え字の3文字が過去に出現した位置を記録する
map<uint32_t, size_t> hash;
for (size_t i=0; i<data.size();)
{
    int len = 0;
    if (i+3 <= data.size()) {
        uint32_t h = data[i]<<16 | data[i+1]<<8 | data[i+2];
        if (hash.count(h) > 0 && i-hash[h] <= 32768) {
            // 過去に出現していれば、一致長と一致距離に圧縮
            size_t p = hash[h];
            len = 3;
            while (len<258 && i+len<data.size() && data[p+len]==data[i+len])
                len++;
            literal.push_back(calcCodeExt(len-3, literalExtTable, 29, 257));
            dist.push_back(
                calcCodeExt((int)(i-hash[h]), distExtTable, 30, 0));
        }
    }
    if (len == 0) {
        // 過去に出現していなければ、そのままリテラルとして書き出す
        literal.push_back(Code(data[i], 0, 0));
        dist.push_back(Code(-1, 0, 0));
        len = 1;
    }
    for (int j=0; j<len; j++)
        if (2 <= i+j)
            hash[data[i+j-2]<<16 | data[i+j-1]<<8 | data[i+j]] = i+j-2;
    i += len;
}
literal.push_back(Code(256, 0, 0));
dist.push_back(Code(-1, 0, 0));
```

```
int main()
{
    makeCrcTable();
    Key key;
    for (uint8_t c=0; c<8; c++) {
        key.update_keys(c);
        printf(" %02x", key.decrypt_byte());
    }
    printf("\n");
    //  1a 63 54 9f e7 41 0e 83
}
```

1、2、...、7 を入力して、予測ができない（ように見える）値が出力されていることがわかります。

ZIP ではファイルごとに暗号化が行われます。あまりそのような使い方はされませんが、ファイルごとに別のパスワードで暗号化したり、ZIP 中の特定のファイルだけを暗号化したりすることができます。

ZIP 中の暗号化されたファイルは、local file header とファイルの中身の間に、12 バイトの encryption header が置かれます。暗号化される前の encryption header の最後の 1 バイトもしくは 2 バイトは、ファイルの CRC 値の最上位バイトもしくは上位 2 バイトです。解凍プログラムは、encryption header を復号して最後のバイトが CRC 値に一致するか確認することで、パスワードが正しいかどうかを判定します。2 バイトのチェックは古い PKZIP で行われていたものであり、特に互換性を重視する場合以外は、攻撃者に余計な情報を与えないために 1 バイトのみを CRC にするべきでしょう。encryption header の前半 11 バイト（もしくは 10 バイト）はランダムなデータで埋めます。

仕様書には記述がありませんが、Info-zip の Zip 3.00 のソースコードを見ると、local file header と central directory header の `flag` の 3 ビット目が立っているときは CRC の代わりに `modifiedTime` を使うという実装になっています。PKWARE 社の SecureZIP もこの方法でパスワードのチェックをしていました。`flag` の 3 ビット目を立てると、ファイルサイズや CRC を local file header ではなくファイル本体の直後に出力して、標準入出力などのシークができないファイルで ZIP を取り扱うことができます。圧縮プログラムが ZIP ファイルを書き込むときも、解凍プログラムが ZIP を読み込むときもシークができない場合、encryption header を処理する時点でファイルの CRC が分からないのでこのような仕様になっているのでしょう。

```
    /* If last two bytes of header don't match crc (or file time in the
```

第 3 章

ZIP の暗号に対する攻撃

ZIP の暗号（Traditional PKWARE Encryption）は既知平文攻撃に対して脆弱であることが知られています。ある暗号文とその暗号文に対する平文を攻撃者が入手した場合、攻撃者が他の暗号文を解読することが可能になります。同じ ZIP 内のファイルは同じパスワードで暗号化されることが多いため、例えばソースコードをまとめた ZIP ファイル中に OSS として公開されているソースコードが含まれていると、解読が可能になります。また、多くのファイルはシグネチャなどで先頭部分が予測可能なため、この部分に対して既知平文攻撃を行い、残りの部分を解読することができる場合もあります。

攻撃を試す前に、Linux の zip コマンドで攻撃の対象となる暗号化された ZIP ファイルを作成します。`-e` で暗号化を、`-0` で圧縮しないことを指示しています。

```
$ zip -e -0 secret.zip kusano_k.png secret.txt
Enter password:
Verify password:
  adding: kusano_k.png (stored 0%)
  adding: secret.txt (stored 0%)
```

これらのファイルは奥付に記載のウェブサイトで公開しています。ここでは、kusano_k.png が既知のファイル、secret.txt は攻撃者が内容を知ることができないファイルであると想定しています。

2. key2 の配列の候補を求める（数百〜2^{22} 個程度）
3. 各 key2 の配列から、key1 の配列の候補を求める（key2 の配列 1 個あたり約 2^{16} 個）
4. key1 の配列の候補から key0 の候補を求める（key1 の配列 1 個に対して 1 個）
5. 暗号文と平文に矛盾しない key0 をフィルタリングする
6. ある位置での key0 と key1、key2 が手に入ったので、暗号化の処理を逆に辿り、パスワードを処理した直後の各 key の値を求める
7. パスワードを計算する

6 の時点での各 key の値があれば、同じパスワードで暗号化されたファイルを復号することができるので、7 の処理は必須ではありません。候補の数は key1 の配列の候補が 2^{22} 個の場合で、約 2^{38} 個となります。

key2

key2 から平文に xor する値を求める処理 decrypt_byte を見ると、（uint16_t にキャストしているので）上位 16 ビットと（2 と or を取っているので）2 番目のビットは返り値に影響が無いことが分かります。また、temp*(temp^1) という処理から最下位ビットも返り値に影響しません。影響があるのは key2 の 14 ビットであり、事前にテーブルを作っておくことで、平文と暗号文からこの 14 ビットを求めることができます。なお、decrypt_byte の各返り値には、この 14 ビットがそれぞれちょうど $2^6 = 64$ 個ずつ対応します。

```
uint8_t decrypt_byte(uint32_t key2) {
    uint16_t temp = (uint16_t)(key2 | 2);
    return (temp*(temp^1))>>8;
}
```

key2 の配列の候補は、ファイルを暗号化するときは逆に、ファイル後方から前方に向かって求めます。位置 p での key2 の値を key2[p] と書くと、key2[p] と key2[p+1] の間には次の関係が成り立ちます。crcTable の添え字の 1 バイトの値 key2[p]&0xff ^ key1>>24 は、この段階では key1 の値が分からないのでまとめて i としています。

```
                key2[p+1] = crc32(key2[p], key1>>24)
                          = key2[p]>>8 ^ crcTable[X],
              crcTable[X] = key2[p+1] ^ key2[p]>>8,
crcTableInv[key2[p+1]>>24] = key2[p+1]<<8 ^ key2[p] ^ X
                  key2[p] = key2[p+1]<<8 ^ crcTableInv[key2[p+1]>>24] ^ X
```

```
    }

    for (uint32_t k2: key2s) {
        printf("Try key2=%08x\n", k2);
        key2[11] = k2;
        if (guessKey2(10))
            break;
    }
}
```

```
4194304
9998 2686976
9997 2013184
9996 1560559
  :
14 4114
13 4076
12 4017
11 4174
Try key2=00152d80
Try key2=00207458
 f3075c74 dc250a93 e4d2ea01 8b5a6a00 fde9a310 f34498eb 7a991e30 d1c1feef dddcb…
 f3075ce8 dc250a93 e4d2ea01 8b5a6a00 fde9a310 f34498eb 7a991e30 d1c1feef dddcb…
  :
```

10,000 バイト後方から処理をすることで、key2[11] の候補を 1/1,000 に減らすことができました。

key1

key2 の配列の候補から、key1 の配列の候補を求めます。

key2[p-1] から key2[p] を計算する際には key1[p] の最上位バイト（key1[p]>>24）が使われるので、key2 の値から key1 の最上位バイトを逆算することができます。

```
    key2[p] = crc32(key2[p-1], key1[p]>>24)
            = key2[p-1]>>8 ^ crcTable[key2[p-1]&0xff ^ key1[p]>>24],
```

```
                basis[i] = x;
    }
    for (int i=0; i<32; i++)
        printf("%08x %08x\n", 1<<i, basis[i]);
}
```

パスワードを求めるプログラムは次のようになります。求めたパスワード用いて鍵を初期化してみることで、パスワードがその長さであったかどうかが分かります。

```
bool checkPassword(uint32_t key0, uint32_t key1, uint32_t key2,
    vector<uint8_t> password)
{
    Key key;
    for (uint8_t c: password)
        key.update_keys(c);
    return key.key0==key0 && key.key1==key1 && key.key2==key2;
}

//  CRCが start から endに変換する入力を求める
vector<uint8_t> recoverCrc(uint32_t start, uint32_t end, int n)
{
    uint32_t basis[32] = {
        0xdb710641U, 0x6d930ac3U, 0xdb261586U, 0x6d3d2d4dU,
        0xda7a5a9aU, 0x6f85b375U, 0xdf0b66eaU, 0x6567cb95U,
        0xcacf972aU, 0x4eee2815U, 0x9ddc502aU, 0xe0c9a615U,
        0x1ae24a6bU, 0x35c494d6U, 0x6b8929acU, 0xd7125358U,
        0x7555a0f1U, 0xeaab41e2U, 0x0e278585U, 0x1c4f0b0aU,
        0x389e1614U, 0x713c2c28U, 0xe2785850U, 0x1f81b6e1U,
        0x3f036dc2U, 0x7e06db84U, 0xfc0db708U, 0x236a6851U,
        0x46d4d0a2U, 0x8da9a144U, 0xc02244c9U, 0x5b358fd3U,
    };

    for (int i=0; i<n; i++)
        start = start>>8 ^ crcTable[start&0xff];

    uint32_t x = 0;
    for (int i=0; i<32; i++)
        if (((start^end)>>i&1) != 0)
            x ^= basis[i];

    vector<uint8_t> input;
```

ファイルの先頭数百 KB が既知であるとき

ファイルの先頭部分が一致していても、圧縮後の先頭部分は一致しません。Deflate 圧縮されたデータの先頭にはハフマン符号表が付き、ハフマン符号表の中身は deflate ブロック全体の符号の出現率に依存するからです。

圧縮ソフトの実装に依存しますが、Info-ZIP の deflate では、リテラル・一致長 32,768 個ごとにブロックに区切られていました。先頭のブロックの圧縮前のデータが同じならば、圧縮後のブロックも等しくなるでしょう。

ただ、ファイルの先頭数百 KB が既知であるという状況はあまり無さそうです。

先頭 16 バイト程度が既知で、固定ハフマン符号が用いられている場合

Deflate 圧縮でどれだけ前のデータを保持しているかや、探索にどの程度力を入れるかは圧縮ソフトや設定に依存します。しかし、各ファイルフォーマットのヘッダ程度のサイズであれば、圧縮結果に差はほとんどなく、固定ハフマン符号が用いられていると仮定すれば圧縮結果も推測できるでしょう。

圧縮ソフトのアルゴリズムによっては、同じバイトの連続が 2 個目から一致長と一致距離に符号化されるとは限らないことに注意する必要があります。例えば、`０００００００`というデータを圧縮するとき、`0（5，1）`と圧縮するのが最適ですが、`００（4，1）`と圧縮されることがあります。

先頭 16 バイト程度が既知で、カスタムハフマン符号が用いられている場合

このケースが一番多いと思いますが、攻撃が最も困難です。

先述したようにブロック全体の符号の出現率でハフマン符号表が変化するので、ハフマン符号表の部分を推測することが現実的ではありません。既知の先頭部分がどのように符号化されるかを推測するしかないでしょう。

ハフマン符号表の内容によって圧縮されたハフマン符号表の長さも異なります。いくつかのデータで試してみたところ、50 バイトから 100 バイト程度になりました。ハフマン符号表のサイズをビット単位で総当たりする必要があります。

Stay の攻撃法

ここまで見てきたように ZIP の暗号は既知平文攻撃に対して脆弱です。さらに、平文が全く分からない場合でも攻撃可能な方法が Stay によって提案されています[*3]。

論文が発表されたのは 2001 年なのでとっくに修正されているかと思いましたが、現在でもこの攻撃法は有効でした。ZIP の暗号化はそもそも脆弱なので実装のみを強化しても意味が無いという判断なのかもしれません。Info-ZIP の実装は Linux の `zip` コマンドとして広く使われています。

この攻撃法では、Info-ZIP の実装に不備があり、encryption header の乱数が予測可能であることを利用します。 Encryption header の乱数を予測することで既知平文とします。後述するように、生成された乱数をそのまま encryption header とするのではなく、乱数を一度 ZIP の暗号化と同じ方法で暗号化する（生成された乱数は 2 回暗号化される）ため、Biham と Kocher の攻撃法を用いることはできません。

Stay はある 1 バイトの暗号化には鍵の一部の情報しか寄与しないことに着目しました。寄与する部分のみを総当たりする、実際に暗号化の処理を行って結果が一致するもののみをフィルタリングして次の総当たりに回す、ということを繰り返します。Encryption header の 2 回の暗号化には同じ鍵が使われるので、1 回目の暗号化の結果（2 回目の暗号化の平文）も総当たりすることで、Info-ZIP の暗号化に対しても適用することができます。1 個のファイルに対する攻撃では Biham と Kocher の攻撃法よりも計算方法が大きくて実用にはなりませんが、パスワードを処理した直後の鍵は同じパスワードで暗号化された全てのファイルで共通であり、複数のファイルでフィルタリングすることで高速化しています。

論文には「5 個のファイルがあれば解ける」と書かれていますが、計算資源を注ぎ込めばより少ないファイルでも解読できそうです。以降の説明では 5 個のファイルを用いています。

処理の流れは次のようになります。`key0`、`key1`、`key2` はパスワードを処理した直後の値、`R`、`P`、`C` はそれぞれ暗号化前、1 回目の暗号化後、2 回目の暗号化後（対象の ZIP に保存される）の encryption header です。

1. 各ファイルの encryption header の先頭 1 バイトを集めて、乱数のシードを推測し、`R` を求める

[*3] STAY, Michael. ZIP attacks with reduced known plaintext. In: International Workshop on Fast Software Encryption. Springer, Berlin, Heidelberg, 2001. p. 125-134.

```
bool attack2()
{
    printf("attack2 %08x %08x %08x\n", key0_crc, key1_mul1, key2_crc);

    //  crc32(key0,0) & 0x0000ff00
    for (uint32_t g0=0; g0<0x100; g0++) {   //  0xdb
        key0_crc = key0_crc&~0xff00 | g0<<8;
        //  MSB(key1*0x08088405*0x08088405)
        for (uint32_t g1=0; g1<0x100; g1++) {   //  0x7f
            key1_mul2 = g1<<24;
            //  crc32(key2,0)&0x00ff0003
            for (uint32_t g2=0; g2<0x100; g2++) //  0xfb
            for (uint32_t g3=0; g3<4; g3++) {   //  0
                key2_crc = g2<<16 | key2_crc&~0xff0003 | g3;
                if (attack3(0))
                    return true;
            }
        }
    }
    return false;
}
```

Encryption header の 3 バイト目によるフィルタリング

2 バイト目よりも処理が複雑になっていますが、2 バイト目と同様に、encryption header の 3 バイト目を計算して正しく暗号文を生成できるもののみを通しています。

key1 の計算について、Stay の論文には 0x08 の記載が抜けているようです。1 回目の key1 の計算の +1 と、2 回目の key1 の計算での 0x08088405 との乗算で出てくるので加えています。

```
bool attack3(int f)
{
    if (f==5)
        return attack4();

    for (uint32_t g0=0; g0<2; g0++)
    for (uint32_t g1=0; g1<2; g1++) {
        key0_20lsb[f] = crc32(key0_crc^crcTable[R[f][0]], R[f][1]);
```

```
attack2 0000ff5f fe000000 00001738
  attack4 0000cb5f ff000000 007b1738
  attack4 0000cb5f ff000000 00fb1738
  attack4 0000db5f 7f000000 007b1738
  attack4 0000db5f 7f000000 00fb1738
key0: e4b115e8
key1: 01b5f84c
key2: 98b20d06
2870.06
```

全探索空間の 37% くらいのところ（最も外側のループは `LSB(crc32(key0,0))`）で解が見つかり、実行時間は Core i7-4790 3.6GHz で 1 時間弱でした。 論文には、Pentium II 500 MHz で 2 時間未満と書かれているので、もう少し高速化できそうです。このプログラムでは、キャリーの情報は `key1` の全てのビットが求められてからのフィルタリングにしか使用していませんが、`key1*0x08088405` の上限や下限を定めるのに活用できるそうです。