MWS Cup 2013

事前課題 2「文書型マルウェア解析」解答例

1　文書型マルウェア解析（10 点）

別途配布される MWS Cup 2013 用に作成された以下の文書型マルウェア（以下、「検体」という）について、設問に答えよ。

ファイル名	q2.jtd
ファイルサイズ	111,112 バイト
ファイル形式	一太郎文書
MD5 ハッシュ値	92dbbe9022e388db02ac3e5b05ca0c6e
SHA1 ハッシュ値	e093e087e8b630174ce5d50fe13485c2600f57d0
SHA256 ハッシュ値	8d492c3741938f3496c713894dcebfe4348a5129a1e523aa93b13e777623ca28

1.1　設問1

1.1.1 問題文

以下のソフトウェアを用いて動的解析を行い、検体が脆弱性の悪用に成功するかどうかを答えよ。（2 点）

- 一太郎ビューア最新版
- 一太郎 2013 玄 体験版

1.1.2 Team Enu の解法

まず本検体の動的解析環境として VirutualBox4.2.18 上 [7] に Windows7 Professional SP1 [8] を用意した。また、本検体を動的解析するにあたり解析環境上に一太郎ビューア最新版 [9] (以下一太郎ビューア)と一太郎 2013 玄 体験版 [10] (以下一太郎 2013)を用意した。

最初に本検体を静的解析した結果、本検体は脆弱性を悪用して複数のシェルコードを実行していることが分かった。シェルコードを解析した結果、ユーザを騙すための表示用文書(document.jtd)を cmd.exe を使用して表示するコードを発見した。

```
0000CA4D        lea     edi, [ebp-248h] ; Load Effective Address
0000CA53        mov     dword ptr [edi], '.dmc'
0000CA59        mov     dword ptr [edi+4], '.exe' ; cmd.exe /c "document.jtdのファイルパス"で
0000CA60        mov     dword ptr [edi+8], '" c/' ; 表示用の文書を表示
0000CA67        add     edi, 0Ch        ; Add
0000CA6D        rep movsb               ; Move Byte(s) from String to String
0000CA6F        dec     edi             ; Decrement by 1
0000CA70        mov     byte ptr [edi], 22h ; '"'
0000CA73        inc     edi             ; Increment by 1
0000CA74        mov     byte ptr [edi], 0
0000CA77        pop     edi
0000CA78        lea     eax, [ebp-248h] ; Load Effective Address
0000CA7E        call    $+5             ; Call Procedure
0000CA83        add     dword ptr [esp], 10h ; Add
0000CA8A        push    0
0000CA8C        push    eax
0000CA8D        push    dword ptr [edi+38h] ; GetCurrentThread
0000CA90        jmp     dword ptr [edi+34h] ; WinExec
```

図 1.表示用文書(document.jtd)を cmd.exe を使用して表示するコード

そこで動的解析を用いて検体実行時に表示用文書（図1）が表示されるか否かの確認を行った。具体的な確認方法について以下に示す。

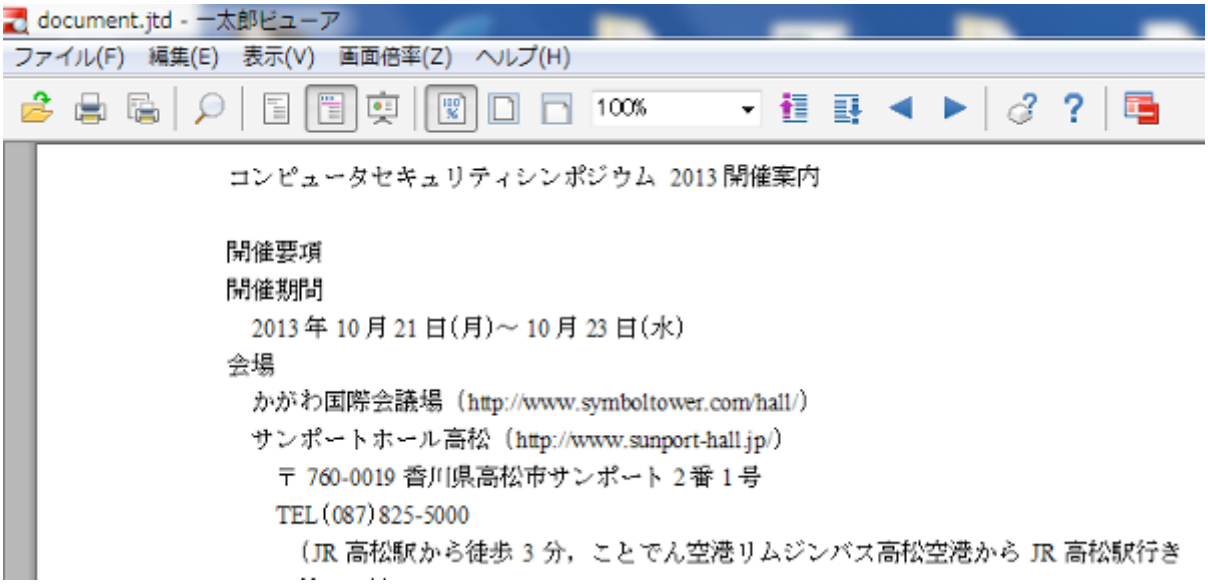

図 2.ユーザを騙すための表示用文書 document.jtd

まず、本検体を一太郎ビューアと一太郎 2013 の両方で開いてみた所、エラーメッセージのみが表示され、表示用文書が表示されないことを確認した。さらに ProcessMonitor [11] を用いて、表示用文書が表示されるか否かを確認した。具体的には ProcessMonitor のフィルタ機能（図 3）を使用して全プロセスの中から cmd.exe のプロセスのみ抽出した。その結果 cmd.exe のプロセス自体起動されていないことが確認できた。

上記の動的解析の結果、脆弱性の悪用成功時に表示される表示用文書が表示されないことから、本検体は脆弱性の悪用に成功していないと言える。

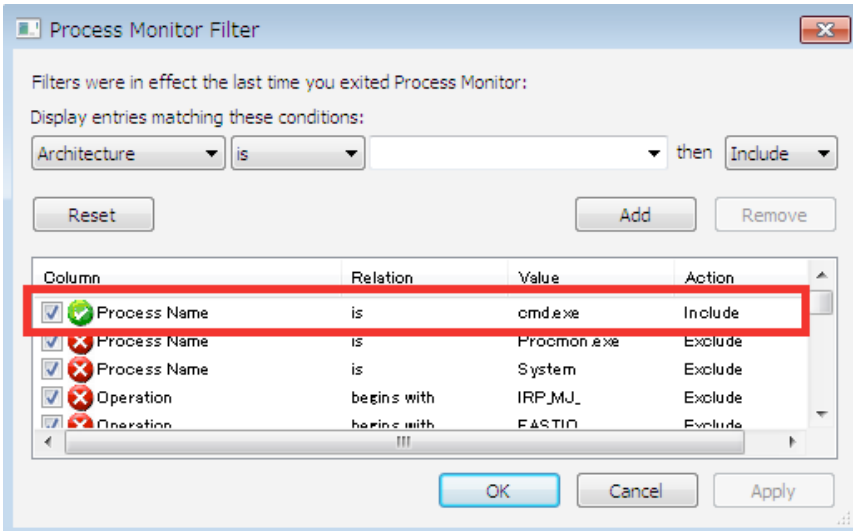

図 3.Process Monitor 上で cmd.exe プロセスのみ表示するフィルタ設定

1.1.3 GOTO Love and 初代森研の解法

VMWare 上の Windows に一太郎ビューア最新版と一太郎 2013 玄 体験版をインストールし、動的解析を行った。Windows のバージョンは、XP SP3 32bit, 7 32bit/64bit の 3 種類で動的解析を行っている。

一太郎ビューア最新版で q2.jtd を開いた場合、「ファイルを読み込むことができません。」というメッセージボックスが表示され、q2.jtd を開くことができなかった。

一太郎 2013 玄 体験版で q2.jtd を開いた場合、「ファイルのデータに不正な構造が見つかったため、読込を中止しました。」というメッセージボックスが表示され、q2.jtd を開くことができなかった。

念のため、q2.jtd を開く際に以下のツールを用いて動的解析を行い、適宜に通常の一太郎文書を開いた場合と比較したが、怪しい結果は得られなかった。

- FakeNet [7] (エミュレーションサーバ・外部との通信をキャプチャするため)
- Prefetch Parser [8] (一太郎のプロセスがロードしたファイルを確認するため)
- Regshot [9] (q2.jtd 読込前後のレジストリ差分を取得するため)

上記の結果から最新版のビューアと玄では、検体は脆弱性の悪用に成功しないものと思われる。

また一太郎 2012 (バージョン不明、ただしインストール CD からインストールした直後のものでアップデートはしていない) を使用して動的解析をおこなったところ、「デコード成功！ 作成されるマルウェアは当日課題で出題します」というメッセージボックスが表示され、一太郎 2012 はフリーズした。フリーズした一太郎 2012 を強制終了すると、自動的に document.jtd という一太郎文書が開かれており、その内容は MWS の日程についてのものであった。

この一太郎 2012 では、検体は脆弱性の悪用に成功しているものと思われる。これと比較しても、一太郎ビューア最新版と一太郎 2013 玄 体験版では、脆弱性の悪用に失敗しているといえる。

1.1.4 GOTO Love and 初代森研の解答

- 一太郎ビューア 2013
  - バージョン ：23.0.2.0
  - 実行環境 ：Windows XP SP3 32bit, Windows 7 32bit / 64bit
  - 結果 ： 失敗
- 一太郎 2013 玄 体験版
  - バージョン ：0.3.2351
  - 実行環境 ：Windows XP SP3 32bit, Windows 7 32bit / 64bit
  - 結果 ： 失敗

1.1.5 人海戦術チームの解法

上記ソフトウェアを用意し，検体を動作させる．その結果，各種リソース(ファイル，ディレクトリ，ミューテックス，レジストリ，プロセス，スレッドなど)への変化や外部通信などのアクティビティについて，記録し観測する．記録には，Capture-BAT を利用する．

また，特定の OS のみで動作する検体の可能性も考慮する．そのため，動作環境は Windows XP(SP3)と Windows 7(SP1)を用意する．

動的解析に用いるソフトウェアの情報は下記の通りである．

- 一太郎ビューア 2013 (バージョン: 23.0.2.0)
- 一太郎 2013 玄 体験版(バージョン: 0.3.2331)
- 一太郎 2013 玄 体験版(バージョン: 0.3.2351)

1.1.6 人海戦術チームの解答

	Windows XP(SP3)	Windows 7(SP1)
一太郎ビューア 2013	×	×
一太郎 2013 玄(バージョン: 0.3.2331)	×	×
一太郎 2013 玄(バージョン: 0.3.2351)	×	×

凡例:〇悪用成功， ×悪用失敗

1.2 設問2

1.2.1 問題文

検体の内部にはエンコードされたコード片（シェルコード）が複数存在している。これを探し出し、それぞれのオフセットおよびエンコード方法を答えよ。(2 点)

ここでオフセットとは、ファイルの先頭から該当シェルコードの実行開始地点（通常は jmp 命令や call 命令である）までの距離を指す。また、あらかじめ解答欄に用意された行の数だけシェルコードが存在するとは限らない。

1.2.2 Team Enu の解法

本検体には複数のシェルコードがエンコードされた状態で格納されていることが、問題文で提示されている。そこで、シェルコードの格納場所をツール等で特定しつつ、そのエンコード方法を静的解析により解析した。解析の結果3つのシェルコードと、そのエンコード方法が分かった。その具体的な方法を以下に記述する。

最初に本検体から strings コマンドを使用して、表示可能なデータのみ抽出した。その結果、"document.jtd"や cmd.exe /c"などの文字列を発見した。発見文字列は Windows コマンドのような文字列であることから、文字列周辺にシェルコードが存在するのではないかと推測出来る。そこでバイナリエディタ Bz [12] を使用して、発見文字列周辺を検索したところ、(先頭からのオフセット)0xC800～0xCAF2 までシェルコードと推定される正体不明のデータ（以下第一のシェルコード）(図4）が発見出来た。

図 4.シェルコードと推定されるデータ

発見した第一のシェルコードを IDApro6.4 [13] で解析した結果、シェルコード中に xor デコードルーチン（図5）があることが分かった。本ルーチンの詳細な挙動について以下に示す。

```
seg000:0000C945     decode_routine_:                        ; CODE XREF: seg000:0000C952↓j
seg000:0000C945                     xor     byte ptr [ebx+ecx], 89h ; Logical Exclusive OR
seg000:0000C949                     cmp     ecx, 0C2h ; 'ツ' ; Compare Two Operands
seg000:0000C94F                     jz      short loc_C954  ; Jump if Zero (ZF=1)
seg000:0000C951                     inc     ecx             ; Increment by 1
seg000:0000C952                     jmp     short decode_routine_ ; Jump
```

図 5.第一のシェルコード中にある xor デコードルーチン

本ルーチンでは、本検体ファイル中（オフセット 0xD200～）に存在している "0x58 0x65 0x50 0x49 0xDF 0xFE 0xEB 0xBE" という8バイトのデータを目印に、デコードを行っている。具体的には上記 8 バイトのデータが存在している直後のデータを1バイトずつ 0x89 と 0xC2 バイト分だけ xor している。本ルーチンの処理通りデコードを行うと、シェルコードの一部と推定されるデータ(以下第二のシェルコード）が出現した。

第二のシェルコードを xor デコードする際に、"0x58 0x65 0x50 0x49 0xDF 0xFE 0xEB 0xBE"という8バイトのデータを目印に解読したことから、第一のシェルコードの上部に存在する"0x57 0x64 0x50 0x49 0xEF 0xFE 0xEB 0xBE"も何らかのデコードの目印ではないかと推測出来る。そこで上記8バイトのデータを参考にバイナリエディタでデコードルーチンを探した。その結果、0x2170～0x2300 付近に"0x64 0x57 0x49 0x50"と"0xFE 0xEF 0xEB 0xBE"というデータがあることを発見した。これは、第一のシェルコード上部に存在する8バイトのデータを2バイトずつ、上位1バイトと下位1バイトを交換したものであると分かる。このことから、0x2170 から2バイトずつ読み込み、上位1バイトと下位1バイトを交換したところ、0x217C からシェルコード(以下第三のシェルコード)が現れた。(図6)

図6.シェルコードと推定されるデータ

第三のシェルコードを解析した結果、シェルコードの中に xor デコードルーチンがあることが分かった。本ルーチンは第一のシェルコード上部に存在する"0x57 0x64 0x50 0x49 0xEF 0xFE 0xEB

0xBE"のデータ直後から、1バイトずつ 0x99 と 0xEE バイト分だけ xor している。(図7)

```
decode_routine:                         ; CODE XREF: sub_31+F6↓j ; sub_31+D5↑j
                xor     byte ptr [ebx+ecx], 99h
                cmp     ecx, 0EEh ; '
                jz      short loc_129
                inc     ecx
                jmp     short decode_routine
```

図 7.第三のシェルコード中にある xor デコードルーチン

第三のシェルコードもエンコードされていることから、デコード処理を行っている新たなシェルコードを探した。しかし、残念ながら発見することはできなかったので、それは私たちの今後の課題とする。

上記の解析から、第一から第三のシェルコードの場所とエンコード方法が分かった。本設問では、ファイルの先頭から該当シェルコードの実行開始地点を求められている事から、各シェルコードが最初に実行している命令に着目した。その結果、第一のシェルコードでは 0xC808、第二のシェルコードでは 0xD208、第三のシェルコードでは、0x2191 から最初の処理が実行されていた。[1]

以上が本検体に含まれるシェルコードのオフセットおよびエンコード方法の解析方法である。

1.2.3 人海戦術チームの解法

検体をバイナリエディタで表示し，データ分布の異なる箇所を中心にチェックを実施.
チェック内容としては，

- 文字列として認識可能なもの(ファイル名や API 名の一部，アイキャッチ(識別子)など)の抽出
- call / jmp / pop / xor / ret / loop / dec / inc などの命令に相当するバイナリデータの前後からディスアセンブル
- 特定のデータパターン(連続した値や規則性を持って出現する値など)部分を抽出し，エンコード方法の推測.
- ファイル格納時とソフトウェアのメモリ上でのエンディアンが異なる(上位下位バイトスワップ)箇所については，実機メモリ上でのデータパターンでチェック.

上記のような分析を行い，平文のシェルコード部分とエンコードされているシェルコード部分に分割.その中からデコード処理を特定し，エンコード方法を特定する.

1.2.4 人海戦術チームの解答

ファイル先頭からのオフセット	エンコード方法
0x2178	0x2178 から 364 バイト分がバイト反転(上位下位バイトスワップ)している.
0xC808	0xC808 から 238 バイト分を 0x99 で XOR している.
0xD208	0xD208 から 194 バイト分を 0x89 で XOR している.

[1] 出題者注: 最初の処理が実行されている箇所のみが正答と異なる。

1.3 設問 3

1.3.1 問題文

以下の項目を、シェルコードの処理順序のとおりに並び替えよ。(2 点)

1. document.jtd の作成
2. document.jtd の表示
3. 本体となる DLL ファイルの作成
4. 本体となる DLL ファイルの読み込み

なお、処理内容として存在していない場合には、該当項目を除外すること。

1.3.2 NU14 の解法

0x2278 付近にある 0xc800 以降のデコード処理を読んで実装し、0xc808 以降の処理を逆アセンブルして読んだ。その結果この部分は次のような処理を行なっていることがわかった。
まず、0xd200 のコードのデコード、DLL と表示用文書のデコードを行う。そして 0xd208 以降の処理を呼び出す。このときデコードした DLL のアドレスを渡している。処理が戻ったら、テンポラリディレクトリに document.jtd というファイル名で表示用文書を保存し、cmd.exe に保存したファイル名を渡すことにより表示を行っている。
さらに、0xd208 以降の処理もデコード・逆アセンブルして読んだ。各種 API の解決を行った後、0xd340 へジャンプする。そして最初の処理で、渡されたメモリ上にあるデコードした DLL の領域と、API のテーブルを渡して 0xd448 を呼び出している。その中で、PE フォーマットをパースしながらメモリ上に配置する処理が見て取れるため、自力で DLL をロードしていると予想した。

1.3.3 人海戦術チームの解法

下記の手順で python.exe を解析対象(ホストプログラム)とし，デバッガ経由でシェルコードの挙動を確認する.

1. 検体から，シェルコードを抽出．ファイル上ではバイト反転(上位下位バイトスワップ)しているので復元を行う.
2. Python で検体ファイルを開く(＝検体ファイルへのファイルハンドル確保)
3. Python で読み書き実行権限のあるメモリ領域を確保.
4. 上記で確保したメモリ領域に#1 で抽出したシェルコードをコピー.
5. デバッガで python にアタッチし，EIP をシェルコードに移動させる.
6. デバッガ経由でシェルコードを実行させ，挙動を確認する.

1.3.4 人海戦術チームの解答

1. 本体となる DLL ファイルの読み込み(※)
2. document.jtd の作成

3. document.jtd の表示

※LoadLibrary()相当の処理をシェルコードにて実施. この部分を読み込みと判断している.

1.4 設問 4

1.4.1 問題文

検体の内部には、本体となる DLL ファイルおよびユーザをだますための表示用文書がエンコードされた状態で格納されている。これらのファイルをデコードする処理部分を解析し、ファイルおよびデコードに必要な情報がどのような構造で埋め込まれているのかを答えよ。(2 点)

1.4.2 Team Enu の解法

まず、(先頭からのオフセット)0xC808～に存在するシェルコードを解析すると、0xC954 から図 11 のような xor デコードルーチンを発見した。上記の xor デコードルーチンを解析すると以下のことが分かった。

```
seg000:0000C954     loc_C954:                               ; CODE XREF: seg000:0000C94F↑j
seg000:0000C954                 mov     eax, [ebp-0Ch]
seg000:0000C957                 add     eax, 0E200h         ; Add
seg000:0000C95C                 mov     esi, [eax]
seg000:0000C95E                 add     eax, 4              ; Add
seg000:0000C963                 add     esi, [eax]          ; Add
seg000:0000C965                 add     eax, 4              ; Add
seg000:0000C96A                 xor     ecx, ecx            ; Logical Exclusive OR
seg000:0000C96C
seg000:0000C96C     Decode_File:                            ; CODE XREF: seg000:0000C986↓j
seg000:0000C96C                 mov     dl, [eax+ecx]
seg000:0000C96F                 mov     dh, [eax+ecx+1]
seg000:0000C973                 xor     dx, 4948h           ; Logical Exclusive OR
seg000:0000C978                 xor     dx, cx              ; Logical Exclusive OR
seg000:0000C97B                 mov     [eax+ecx], dh
seg000:0000C97E                 mov     [eax+ecx+1], dl
seg000:0000C982                 inc     ecx                 ; Increment by 1
seg000:0000C983                 inc     ecx                 ; Increment by 1
seg000:0000C984
seg000:0000C984     loc_C984:                               ; Compare Two Operands
seg000:0000C984                 cmp     ecx, esi
seg000:0000C986                 jb      short Decode_File ; Jump if Below (CF=1)
```

図 11.エンコードされた DLL と表示用文書をデコードしているルーチン

最初に 0xE200 にあるデータ"0x7E00"と、0xE204 にあるデータ"0x5200"を足し合わせ"0xD000"という値を取得している。その後 0xE208 から 0xD000 バイト分、2バイトずつ読み込んで xor デコードをしている。本 xor デコード処理では2バイトずつ読み込み 0x4948 と xor した後で、さらに現在までに xor 処理したバイト数と xor している。そして xor デコード処理後、上位1バイトと下位1バイトを交換した後で、元のデータを上書きしている。

上記の xor デコードルーチン通り処理を行うと、DLL ファイルと表示用文書のファイルが結合された状態で出力された。そこでさらに 0xC808～に存在するシェルコードを解析すると、0xE204 にあるデータ"0x5200"を、表示用文書のデータサイズとして使用していることが、図 12 の処理の部分から分かった。そのことから、0xE200 にあるデータは、DLL のファイルサイズ、0xE204 にあるデータは表示用文書のファイルサイズであることが分かった。

以上が、埋め込まれた DLL と表示用文書のデコード方法である。

```
seg000:0000C9E1          call    dword ptr [eax+10h] ; call GlobalAlloc
seg000:0000C9E4          mov     [ebp-1Ch], eax
seg000:0000C9E7          mov     edi, eax
seg000:0000C9E9          mov     eax, [ebp-0Ch]
seg000:0000C9EC          mov     esi, [eax+0E200h]
seg000:0000C9F2          add     esi, eax        ; Add
seg000:0000C9F4          add     esi, 0E208h     ; Add
seg000:0000C9FA          mov     ecx, [eax+0E204h]
seg000:0000CA00          rep movsb               ; Move Byte(s) from St
```

図 12.0xE204 にあるデータをファイルサイズとして扱っている

1.4.3 NU14 の解法

ファイル全体を逆アセンブルした結果から、適切な命令列であると考えられる処理を探しだした。

0xc954 以降に 0xe200 以降のブロックのデコード処理を発見したため，デコードするスクリプトを作成して内容を抽出し，DLL と表示用文書が含まれていることを確認した.

1.4.4 NU14 の解答

ファイル先頭からのオフセット	サイズ	内容
0xe200	4 バイト	エンコードされた DLL のサイズ
0xe204	4 バイト	エンコードされた表示用文書のサイズ
0xe208	任意のサイズ	エンコードされた DLL のデータ
0x16008(=0xe208+DLL size)	任意のサイズ	エンコードされた表示用文書のデータ
0xc954	50 バイト	デコード処理

1.5 設問 5

1.5.1 問題文

設問 4 で解析したデコード処理・情報を元に、エンコードされた DLL および文書ファイルを抽出するスクリプトを作成せよ。スクリプトは 40 行以内に収めるようにすることとし、言語に指定はない。(2 点)

1.5.2 GOTO Love and 初代森研の解法

設問 4 で発見したデコーダ部分の処理から、デコードのアルゴリズムは以下のようになる。

1. デコードの開始アドレスを eax レジスタに代入する。ループを用いて処理を行うため、ecx レジスタをカウンタとして用いる。初期値は 0 である。
2. [eax+ecx]から 1 バイト読み込み、それを 16 ビットレジスタ（dx）の下位 8 ビットに入れる。さらに[eax+ecx+1]から 1 バイト読み込み、それを同じ dx レジスタの上位 8 ビットに入れる。
3. dx レジスタの値と 0x4948 とを xor し、その値を dx レジスタに代入する。

4. dx レジスタと cx レジスタ（cx は ecx の下位 16 ビットをさす）の値とを xor し、その値を dx レジスタに代入する。
5. dx レジスタの上位 8 ビットを[eax+ecx]に、下位 8 ビットを[eax+ecx+1]に代入する。
6. カウンタである ecx レジスタに 2 を加える。
7. ecx レジスタの値がデコード長である 0xD000 と等しくなければ、2 に戻る。等しければ、デコード終了。

なお、DLL の直後に表示用文書があるので、デコード長はこの 2 つのサイズの合計（0x7E00 + 0x5200 = 0xD000）である。

上記のアルゴリズムをもとに作成した Python スクリプト、parse_q2.py が 2.5.3 の解答である。
parse_q2.py は 3 つの引数をとり、1 つ目の引数：q2.jtd、2 つ目の引数：抽出した DLL の出力ファイル、3 つ目の引数：抽出した表示用文書の出力ファイル、となっている。

1.5.3 GOTO Love and 初代森研の解答

```
#! /usr/bin/python
# -*- coding: utf-8 -*-

import sys
import struct

if len(sys.argv) < 4:
    print 'usage: ' + sys.argv[0] + ' q2.jtd output1(DLL) output2(document)'
    quit()

infile = open(sys.argv[1], 'rb')
outfile = open(sys.argv[2], 'wb')

count = 0
length = 0xD000        # DLL's length (0x7E00) + document's length (0x5200)
infile.seek(0xE208)
byte = infile.read(1)
for i in range(0, length/2):
    low = struct.unpack('B', byte)[0]
    high = struct.unpack('B', infile.read(1))[0]

    count_low = count & 0xFF
    count_high = (count >> 8) & 0xFF

    low = low ^ 0x48 ^ count_low
    high = high ^ 0x49 ^ count_high

    if count == 0x7E00: # DLL's length : 0x7E00
        outfile.close()
        outfile = open(sys.argv[3], 'wb')
```

```
        outfile.write(struct.pack('B', high))
        outfile.write(struct.pack('B', low))

        byte = infile.read(1)
        count += 2

infile.close()
outfile.close()
```

1.5.4 Alkaneters の解法

0xc954 辺りのデコードルーチンを Ruby スクリプトで実装した。

1.5.5 Alkaneters の解答

```
IN  = "q2.jtd"
OUT = "q2_e208"

DW_E200 = 0x7e00
DW_E204 = 0x5200

esi = DW_E200 + DW_E204
eax = 0xe208

q2 = open(IN, "rb")
q2.seek(eax, IO::SEEK_SET)
q2 = q2.read(esi).bytes.to_a

decoded = []
ecx = 0
while ecx < esi
        dl = q2[ecx]
        dh = q2[ecx + 1]

        dx = (dh << 8) + dl
        dx = dx ^ 0x4948
        cx = ecx & 0xffff
        dx = dx ^ cx
        dh = dx >> 8
        dl = dx & 0xff

        decoded << dh
        decoded << dl
```

```
    ecx = ecx + 2
end

open(OUT + ".dll", "wb") do |f|
    f.write(decoded[0..DW_E200-1].pack("C*"))
end
open(OUT + ".jtd", "wb") do |f|
    f.write(decoded[DW_E200..-1].pack("C*"))
end
```

1.5.6 NU14 の解法

q2.jtd を開き 0xe200 の位置から DLL と文書ファイルの長さを読み取り、続くデータをデコードしてファイルに書きだすスクリプトを Perl で作成した。

1.5.7 NU14 の解答

```
my $file = "q2.jtd";
my $dllfile = "nanika.dll";
my $jtdfile = "document.jtd";
open my $fh, $file or die;
seek $fh, 0xe200, 0;
read $fh, my($dll_length), 4;
$dll_length = unpack("L<", $dll_length);
read $fh, my($jtd_length), 4;
$jtd_length = unpack("L<", $jtd_length);

my $decoded = "";
for my $i (0..($dll_length + $jtd_length)/2-1) {
        read $fh, my($s), 2;
        $decoded .= pack("S>", unpack("S<", $s)^0x4948^(($i*2)&0xffff));
}

my $wfh;
open $wfh, ">", $dllfile or die;
print $wfh substr($decoded, 0, $dll_length);
undef $wfh;

open $wfh, ">", $jtdfile or die;
print $wfh substr($decoded, $dll_length, $jtd_length);
undef $wfh;
```