進め！リバースエンジニアリングの道

第6回 クラックルーチンの注入

文●愛甲健二

クラック手法を考える

今回は、第5回までの解析結果を元にcrackme_ex.exeのパスワードを突き止めます。理論はすでにわかっているため、あとは解析手法を選ぶだけですが、せっかく「進め！ リバースエンジニアリングの道」と題して連載していますので、コンパイラやスクリプト言語などは使わずに、アセンブラだけで実現したいと思います。よって、crackme_ex.exe自体をクラック用に修正してパスワードを探し当てましょう。

OllyDbgを使ってクラック用の処理を追加します。まずはHACKER JAPAN ONLINEよりファイルをダウンロードして解凍します。アンチウイルスソフトによっては警告が出る場合がありますが、除外項目にしてください。この中のcrackme_ex.exeをOllyDbgで読み込むと00406368以降が空いていますので、ここにクラックルーチンを追加します。

ただ、クラックルーチンを追加しただけでは00406368へ処理が進みませんので、正規ルーチンの適当な場所から00406368へ処理をジャンプさせる必要があります。よって、00402891のcall命令のジャンプ先を004019F0から00406368へ変更します。これで、ユーザー名とパスワードを入力し、OKボタンをクリックすることでクラック処理が走ります（図1）。

では、パスワード解析のメイン処理となるクラックルーチンを書いていきましょう。

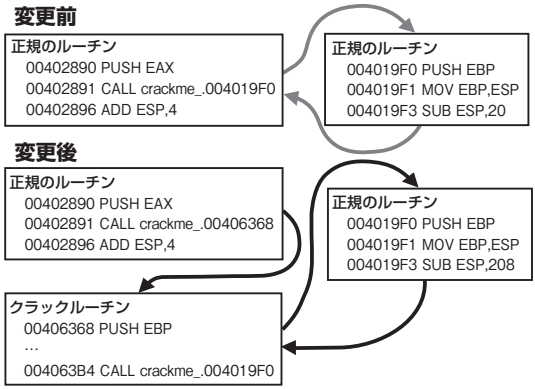

図1　パスワードクラックの処理フロー

OllyDbgでクラックルーチン実行

パスワードクラックにはいくつかの手法がありますが、今回はdictionaryA.txt（DVD-ROMにも収録しています）を使用した辞書攻撃を行います。よって、00406368以降に追加するクラック処理は以下の流れで実現しましょう。

1. 辞書ファイルを開く
2. 文字列を1行取得
3. 文字列をスタックへ
4. 004019F0を呼び出す
5. 2へ戻る（繰返し）

まずは辞書ファイルを開く処理ですが、これはCreateFileAを使います。ファイルを読み込み専用で開くので、第1引数に対象となるファイル名、第2引数にGENERIC_READ（80000000h）、第5引数にOPEN_EXISTING（3h）、第6引数にFILE_ATTRIBUTE_NORMAL（80h）をセットします。

次に文字列を取得する部分ですが、これはReadFileを使いましょう。ReadFileで1文字ずつ取得していき、改行コード「0D0Ah」を見つけたら文字列として認識し、次ステップへ進む、という処理にします。文字列を取得したら、それをスタックへpushして004019F0をcallします。004019F0内では、渡されたパスワード（第1引数）が正しいかどうかの判定が行われます。

これらの処理をC言語風に書くと以下になります。辞書ファイル名はdictionaryA.txtだとアセンブラ上では扱いにくいため、便宜上「DIC」としています。

```
func_00406368(char *s)
{
  char buff[4] = "DIC";
  eax = CreateFileA(
    buff,80000000h,
    0,0,3h,80h,0);
  esi = eax;
_READ_STR:
  ebx = s;
_READ_CHAR:
  ReadFile(esi,ebx,1,buff,0);
  al = *ebx;
  ebx++;
  if(al != 0x0D)
    goto _READ_CHAR;
  ebx--;
  *ebx = '¥0';
  004019F0(s);
  ReadFile(esi,ebx,1,buff,0);
  goto _READ_STR;
}
```

00406368に記述するコードは、004019F0の代わりとしてcallされるため、実行時にはスタックに「入力されたパスワード」のアドレスが格納されています。ただ、本来の処理とは異なり、パスワードは辞書ファイルから読み込むため、このスタックにある「入力されたパスワード」は必要ないのですが、せっかくなのでメモリ領域だけは使わせてもらいましょう。ebxレジスタにsを格納し、この領域へ読み込んだ文字列を格納します。

ReadFileで1文字ずつ読み込み、0Dhが見つかったら改行の代わりに00hを入れて、004019F0を呼び出します。Windowsの場合は0D0Ahが改行コードであるため、0Dhを読み込むと次のバイトは0Ahです。よって、0Ahを1回読み捨てて_READ_STRへジャンプし、最初へ戻ります。

また、ReadFileにbuffを渡していますが、buff(="DIC")はCreateFileA呼び出し時にしか使用しないため、その後は読み込んだバイト数を格納するための領域として使用します。

このコードをアセンブラにすると以下になります。OllyDbgでcrackme_ex.exeを読み込み、00406368以降に以下のコードを書き込んでください。

```
00406368 PUSH EBP
00406369 MOV EBP,ESP
0040636B PUSH 434944 // "DIC"
00406370 MOV EAX,ESP
00406372 PUSH 0
00406374 PUSH 80
00406379 PUSH 3
0040637B PUSH 0
0040637D PUSH 0
0040637F PUSH 80000000
00406384 PUSH EAX
00406385 CALL kernel32.CreateFileA
0040638A MOV EBX,EBP
0040638C ADD EBX,8
0040638F MOV ESI,EAX
00406391 MOV EDI,EBP
00406393 SUB EDI,4
00406396 PUSH 0
00406398 PUSH EDI
00406399 PUSH 1
0040639B PUSH EBX
0040639C PUSH ESI
```

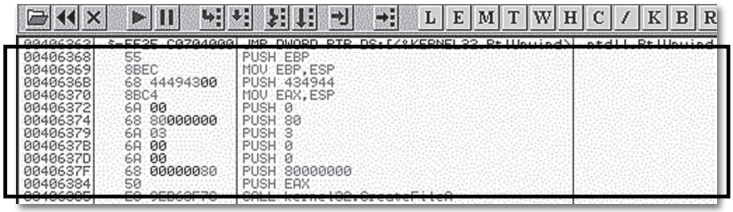

図2 00406368の書き換え

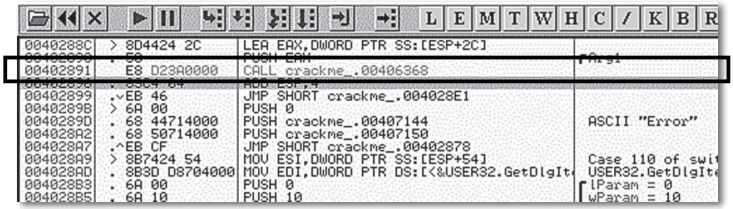

図3 00402891の書き換え

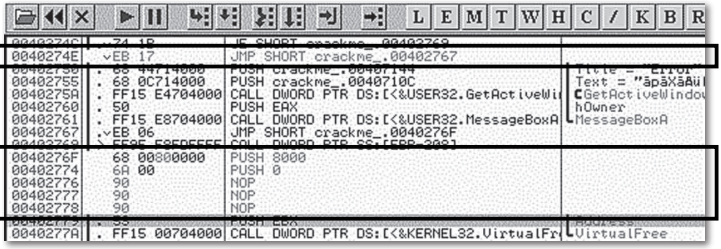

図4　0040274Eと0040276Fの書き換え

```
0040639D CALL kernel32.ReadFile
004063A2 MOV AL,BYTE PTR DS:[EBX]
004063A4 INC EBX
004063A5 CMP AL,0D
004063A7 JNZ SHORT 00406396
004063A9 DEC EBX
004063AA XOR EAX,EAX
004063AC MOV BYTE PTR DS:[EBX],AL
004063AE MOV EBX,EBP
004063B0 ADD EBX,8
004063B3 PUSH EBX
004063B4 CALL 004019F0
004063B9 ADD ESP,4
004063BC PUSH 0
004063BE PUSH EDI
004063BF PUSH 1
004063C1 PUSH EBX
004063C2 PUSH ESI
004063C3 CALL kernel32.ReadFile
004063C8 JMP SHORT 00406396
```

以上でクラックルーチンの作成は終了です（図2）。

あとは、00402891のcall命令のジャンプ先を004019F0から00406368へ書き換えましょう（図3）。これで、パスワードクラックを行う実行イメージの完成かと思えますが、実はあと2つ細かな修正点があります。

1つは、エラーメッセージに関する処理です。004019F0は引数に「入力されたパスワード」を得る関数で、そのパスワードが正解ならばDESで復号された200hバイトのデータ列をマシン語として実行し、間違いならばエラーを意味する文字列をメッセージボックスに表示しますが、パスワードクラック中に毎回間違いのメッセージボックスが表示されても困るので、あらかじめこの処理をjmp命令で飛ばしておきます。

もう1つは、0040277Aで呼び出されているVirtualFreeの引数をMEM_RELEASE（8000h）、サイズを0にしておくことです。ぶっちゃけた話をするとただのプログラムのバグで（汗）、元々のcrackme.exeはメモリを正常に解放していませんでした。よって、パスワードクラックによって何度も004019F0を呼び出すと、途中でVirtualAlloc呼び出し時に「メモリが確保できない」というエラーが発生します。それを防ぐために、VirtualFree呼び出しを問題ないコードに変更します。

これら2つの修正を終えると、パスワードクラック用実行イメージの完成です（図4）。ファイル名を「DIC」に変更した辞書ファイルを同じディレクトリに置き、OllyDbg上で、修正後のイメージを実行してください。数分ほど待った後、パスワードが解析されたことを意味するメッセージボックスが表示されます（図5）。

成功のメッセージボックスが表示されている状態で0040276Fにブレイクポイントをセットし、メッセージボックスのOKボタンをクリックすると、0040276Fで処理が止まります。この状態でスタックを確認するとパスワード「OtmPpLvQ」が確認できます（図6）。

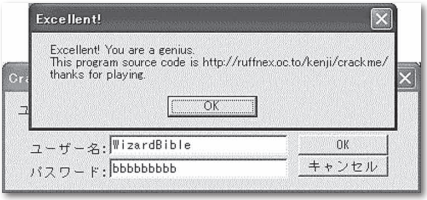

図5　解析成功のメッセージボックス

crackme.exeについて

見事パスワードがわかり、crackme.exeのすべてが解析されました。メッセージボックスに書かれているURL「http://ruffnex.oc.to/kenji/crackme/」へアクセスするとユーザー名とパスワードを求められますので、ユーザー名「WizardBible」、パスワード「OtmPpLvQ」と入力すると、crackme.exeのソースコードがダウンロードできます。興味がある方はぜひソースコードを眺めてみてください。

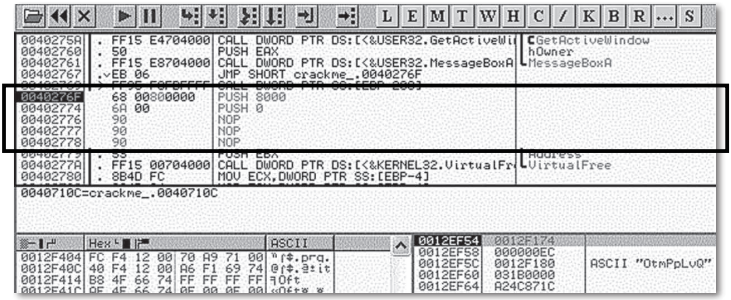

図6　パスワードの確認

また、パスワードクラック用に改造したcrackme_ex_fin.exeもダウンロードファイルに含まれています。これは本記事にてOllyDbg上で行った修正を実行ファイルへ適用したものです。Windows XP SP3 Professionalにて動作確認していますが、CreateFileAとReadFileのアドレスを決め打ちしていますので、動作しない場合はこれらのアドレスを修正してください。また以前にも書きましたが、crackme.exeはWindows7上では正常に動作しません。よって、特に第6回目の本記事はWindows7以外の環境でテストしてください。

Windows7で動作しない理由、成功時にメッセージボックスが表示されるメカニズムなどが知りたい方は、00402769辺りにブレイクポイントを仕掛け、正常なパスワードを入力し、成功時に実行される処理を眺めてみてください。リバースエンジニアリング技術からは少し離れますが、Windowsシステムに関するさまざまなことがわかるかと思います。

最後に

これで「進め！ リバースエンジニアリングの道」は終了となります。読者の皆様、本当にお疲れ様でした。本連載は、なるべく初心者向けであり、かつ、普遍的な技術をテーマに書かせていただきましたが、いかがだったでしょうか。

現在、ソフトウェア開発の現場においてはやはりJavaや.NETが主流であり、Webサービスの世界ではJavaScript、Ruby、Pythonといったプログラミング言語が好まれています。そう考えると、21世紀においてアセンブラを学ぶ意味は果たしてあるのか？ と思われる読者も多いと思います。

はっきり言ってしまえば、アセンブラの知識はコンピューターセキュリティの世界においても、マルウェア解析や脆弱性監査に使われる程度ですし、それらも近年では自動化され、人の手で行うソフトウェア解析の活躍の場はさらに減ってきているかもしれません。ただ、だからといって、その技術が不必要になることは決してないと思います。ビジネスに繋がる機会は少ないかもしれませんが、ソフトウェア技術の根幹はやはりバイナリデータであり、最終的にCPUが解釈するのはマシン語です。そのベースとなるスキルを持っていることで解決できる技術的問題も多々あります。そして何より、無意味かどうかよりも、楽しいかどうかでソフトウェア技術を学び続けたいというのが個人的な本音です。

読者の方々が、本連載を通して少しでもリバースエンジニアリングに興味を持っていただけたなら幸いです。

新連載予告　▶▶▶▶▶ 実践的なリバースエンジニアリングを学ぼう！

次回から「進め！ リバースエンジニアリングの道Next（仮）」と題して、改めて連載をスタートします。リバースエンジニアリングを扱う、という意味ではさほど変わりませんが、これまでの連載が、デバッガの使い方やアセンブラを中心に扱ったのに対し、次回以降では、CTF（Capture the Flag）やマルウェア解析を題材にして、より実践的なリバースエンジニアリングに関する内容がメインとなります。

マルウェア解析や脆弱性監査を行う際は、当然デバッガを使ったり、アセンブラを読む必要があるのですが、実際はそれだけではありません。これらはあくまで基礎的な技術であり、業務を遂行するためには、それらにプラスしてまたいくつかのベーススキルが必要になります。

よって、次回以降は、リバースエンジニアリングという技術をよりセキュリティに応用していく方向で連載を進めていくことになります。なるべく現場レベル、業務レベルでの解析を紹介、解説していきたいと思いますので、興味がある方は、ぜひ引き続きよろしくお願い致します。