進め！リバースエンジニアリングの道

第2回 時間制限の回避

文●愛甲健二

IDAProによる解析

前回、OllyDbgを使うことでMessage BoxA関数を呼び出す箇所（00402998）を特定しました。次は、このエラーメッセージを表示するに至る「過程」を調べます。

1192年の5月15日にしか起動しないということは、このcrackme.exeを実行したその日が1192年の5月15日であるかどうかをプログラムの中で判断しているわけです。その判断フローをプログラミング的に記述するならば以下になります。

1. 現在日時を取得
2. 取得した年と1192を比較
3. 取得した月と5を比較
4. 取得した日と15を比較
5. 2～4のいずれかが異なればエラー

そしてこの判断フローが、Message BoxA関数が呼び出されている箇所（00402998）より前に必ず存在するはずです。もし2～4のいずれか1つが異なればMessageBoxA関数を用いてエラーメッセージを表示し、逆に2～4の条件すべてが真ならば正常な動作を行います。ということは「現在日時を取得している箇所」もしくは「1192、5、15といった数値と比較している箇所」を見つければ、時間制限回避の糸口がつかめるかもしれません。

では、IDAProを用いてcrackme.exeのアドレス00402998あたりを解析してみましょう。IDAProにcrackme.exeをドラッグしてください。IDAProで開く際にいくつか読み込み確認などのウインドウが表示されますが、すべてデフォルトでかまいません。

crackme.exeを読み込んだら、アドレス00402998あたりを見てください（図1）。IDA View-Aウィンドウのスクロールバーを操作してもよいですが、メニューから「Jump」→「Jump to address...」を選択し、表示されたダイアログボックスに00402998と入力してもよいでしょう。

アドレス00402998から少し上にさかのぼっていくとGetSystemTime関数を呼び出している箇所があります（図2）。

時間比較処理の解読

GetSystemTime関数は、その名のとおり時間を取得する関数です。MSDN※で詳細を調べると、引数にSYSTEMTIME構造体のアドレスを指定して呼び出すようです。MSDNによれば、GetSystemTimeの定義は以下のとおりです。

```
.text:0040296C                  lea     ecx, [esp+820h+Text]
.text:00402970                  push    offset aWVgvgngogigavU ; "こ
.text:00402975                  push    ecx             ; LPSTR
.text:00402976                  call    ds:wsprintfA
.text:0040297C                  add     esp, 14h
.text:0040297F                  push    0               ; uType
.text:00402981                  push    offset Caption  ; "Error"
.text:00402986                  lea     edx, [esp+81Ch+Text]
.text:0040298A                  push    edx             ; lpText
.text:0040298B                  call    ds:GetActiveWindow
.text:00402991                  push    eax             ; hWnd
.text:00402992                  call    ds:MessageBoxA
.text:00402998
.text:00402998 loc_402998:                              ; CODE XREF:
```

図1　MessageBoxA関数が呼び出されている箇所

```
.text:00402912                  mov     [esp+814h+var_4], eax
.text:00402919                  lea     eax, [esp+814h+SystemTime]
.text:0040291D                  push    eax             ; lpSystemTi
.text:0040291E                  call    ds:GetSystemTime
.text:00402924                  mov     ax, [esp+814h+SystemTime.wYea
.text:00402929                  cmp     ax, 4A8h
.text:0040292D                  mov     cx, [esp+814h+SystemTime.wDay
.text:00402932                  mov     dx, [esp+814h+SystemTime.wMon
.text:00402937                  jnz     short loc_402960
.text:00402939                  cmp     dx, 5
.text:0040293D                  jnz     short loc_402960
.text:0040293F                  cmp     cx, 0Fh
.text:00402943                  jnz     short loc_402960
.text:00402945                  mov     ecx, [esp+814h+hInstance]
```

図2　GetSystemTime関数が呼び出されている箇所

※　http://msdn.microsoft.com/

```
void WINAPI GetSystemTime(
        __out  LPSYSTEMTIME
lpSystemTime
);
```

以上の内容を踏まえて、GetSystemTime関数が呼び出されている箇所以降の処理を読んでいきましょう。

```
00402919  lea   eax, [SystemTime]
0040291D  push  eax  ; lpSystemTime
0040291E  call  ds:GetSystemTime
00402924  mov   ax, [SystemTime.wYear]
00402929  cmp   ax, 4A8h
0040292D  mov   cx, [SystemTime.wDay]
00402932  mov   dx, [SystemTime.wMonth]
00402937  jnz   short loc_402960
00402939  cmp   dx, 5
0040293D  jnz   short loc_402960
0040293F  cmp   cx, 0Fh
00402943  jnz   short loc_402960
```

まず0040291EでGetSystemTime関数が呼び出されています。これによって現在時刻がSystemTimeに格納されます。00402924のSystemTime.wYearは現在の「年」ですので、10進数で2010年、16進数で7DAh年となります。その7DAhがaxレジスタにmovされて、次の00402929のcmp命令でaxと4A8h（10進数で1192）が比較されます。axは7DAhなので比較結果は「偽」ですね。よって00402937のjnz命令によりloc_402960へジャンプします。

SystemTime.wDayについても見てみましょう。SystemTime.wDayは現在の「日」であり、0040292Dにてcxレジスタにmovされます。 そして0040293Fのcmp命令にて0Fhと比較されています。0Fhは10進数で15ですね。つまり15日ではないならば、次の00402943でloc_402960へジャンプします。同じくSystemTime.wMonthは現在の「月」で、00402932にてdxレジスタにmovされ、00402939で5と比較されます。もしdxが5ではなければloc_402960へジャンプします。

以上の処理をC言語風に書くと次のようになります。

```
SYSTEMTIME SystemTime;
GetSystemTime(&SystemTime);
ax = SystemTime.wYear;
if(ax!=4A8h)
    goto loc_402960;
cx = SystemTime.wDay;
dx = SystemTime.wMonth;
if(dx!=5)
    goto loc_402960;
if(cx!=0Fh)
    goto loc_402960;
```

どうやらこの部分が時間比較を行っている処理と考えて間違いなさそうです。では、この処理の「どこ」を「どのように」変更すれば時間制限を回避できるでしょうか？

方法はいくつかありますが、最も単純なのは条件分岐であるjnz命令をすべてnop（90h）で塗りつぶすことです。nopとは「何も処理しない」というアセンブラ命令なので、jnzをすべて「何も処理しない」と変更することでloc_402960へジャンプさせません。具体的には00402937からの2バイト、0040293Dからの2バイト、そして00402943からの2バイトをすべて90hにします。これでエラーメッセージは表示されず正常にcrackme.exeが起動します。

命令コードの書き換え

OllyDbgでエラーメッセージを表示している箇所を特定し、IDAProで現在日時の比較を行う処理部分を解読しました。次にやるべきことは、2010年に起動してもプログラムがエラーメッセージを表示せずに、正常に起動するよう実行ファイルを書き換えることです。

では、まずはcrackme.exeをOllyDbg上で起動し、OllyDbg上で該当箇所を変更しましょう。変更箇所は00402937からの2バイトを90h、0040293Dからの2バイトを90h、そして00402943からの2バイトを90hですね。では、00402937以降を逆アセンブルウインドウに表示させます。逆アセンブル画面で<Ctrl>+<G>、もしくは右クリックして表示されるメニューから「移動」→「アドレス」と選択してください（図3）。そして00402937と入力します（図4）。

逆アセンブル画面に00402937以降が表示

されたら、変更したい命令コードを選択してスペース、もしくは変更したい命令コードを右クリックして表示されるメニューから「逆アセ修正」を選択します（図5）。nopに変更したいので「NOP」と入力してアセンブルボタンをクリックします（図6）。これでjnz命令が1つ潰れました。同じ方法であと2つのjnz命令も潰してください。

3つのjnz命令をnopに変更したらパッチウインドウを表示させます。<Ctrl>+<P>を押すか、もしくはメニューから「表示」→「パッチ」と選択してください。ここに命令コードを変更した履歴が表示されます（図7）。OllyDbgのログに残り続けるので、OllyDbgを終了後、再度crackme.exeを読み込んでもパッチウィンドウには修正内容が残り、好きなタイミングでパッチ適用のON、OFFが可能です。

では、さっそく実行しましょう。OllyDbg上で、上記のパッチを適用した状態でF9でcrackme.exeを実行します（図8）。

正常にcrackme.exeが起動したら成功です。

実行ファイルの書き換え

OllyDbg上ならば、crackme.exeの命令コードを自由に書き換えられますが、できればOllyDbgがない状態でも時間制限を回避したいところです。そうなると、やはりcrackme.exeファイル自体を書き換える必要があります。

確認すると、先ほどOllyDbg上で書き換えた部分のもともとのデータは、00402937からの2バイトが75h 27h、0040293Dからの2バイトが75h 21h、そして00402943からの2バイトが75h 1Bhでした。さっきはこれらをすべてメモリ（OllyDbg）上で90hに変更したのですが、それをバイナリエディタを使って実行ファイルに対して行います。

00402937以降のデータ列は「75 27 66 83 FA 05」ですので、このデータ列をバイナリエディタを使ってcrackme.exeから探します。Stirlingでcrackme.exeを開き、メニューから「検索・移動」→「検索」を選択し「75 27 66 83 FA 05」を入力して検索します（図9）。すると00001D37からの6バイトがヒットしました。他にヒットする場所はないので、メモリ上のアドレス00402937は、実行ファイルの中では00001D37に対応するとわかりました。

ここまでわかれば、あとは00001D37からの2バイトを90hに、00001D3Dからの2バイトを90hに、そして00001D43からの2バイトを90hに上書きすれば、OllyDbgなしでもcrackme.exeを起動できそうです（図10）。では、変更を保存して実行してください。図8と同じウィンドウが表示されたら成功です。

次はパスワード制限の回避？

crackme.exeが正常に実行されると、次はユーザー名とパスワードを要求してきます。よって、次はこのパスワード制限を回避したいところです。

バイナリエディタ紹介

●Stirling

Stirlingは高機能なバイナリエディタで、特に検索、置換、比較に関してかなり優秀な性能を持っています。リバースエンジニアリングにおいても必要な機能がそろっており、基本的にバイナリエディタはStirlingだけで問題ないレベルです。ただ、唯一の欠点を挙げるとすれば、ギガバイト単位のファイルを扱えない（扱えないこともないが動作が極端に遅くなる）点です。私はこの部分を補完するためだけにBZエディタも使っています。

http://www.vector.co.jp/soft/win95/util/se079072.html

●Binary Editor BZ

BZエディタは機能面や安定性ではStirlingに劣りますが、ギガバイト単位のファイルを全く問題なく読み込めます。ソフトウェア解析においてはそれほど大きなファイルを扱うことはまれですが、ファイルシステムなどを日常的に扱うフォレンジック技術においては、巨大なファイルを相手にすることばかりなので、むしろStirlingよりも重宝されるバイナリエディタです。

http://www.forest.impress.co.jp/lib/stdy/program/progeditor/binaryeditbz.html

命令コードを書き換えて制限を回避

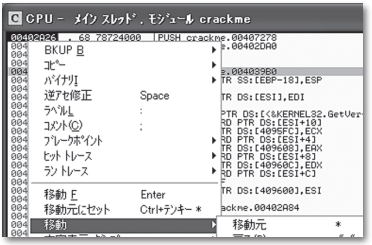

図3　右クリックから「移動」→「アドレス」を選択

図4　入力欄に00402937を入力して00402937以降を表示させる

図5　逆アセンブルコードの修正

図6　jnz命令をnop命令に変更

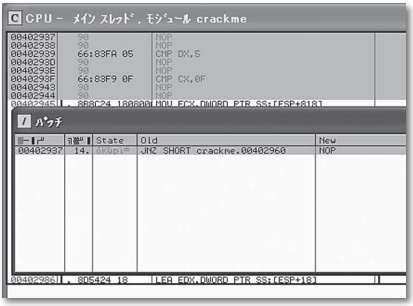

図7　パッチウインドウの表示

図8　crackme.exeの実行

実行ファイルを書き換えて制限を回避

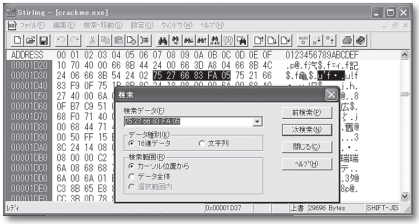

図9　Stirlingでデータ列を検索した結果

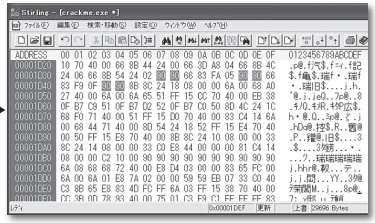

図10　実行ファイルの該当箇所を90hに上書き

しかし、それに挑戦する前に一度アセンブラについて解説しておきます。アセンブラの読解力は、ソフトウェア解析においてはかなり重要な能力の1つですので、次回はcrackme.exeはひとまず置いておいて、アセンブラについて学習していくことにしましょう。