進め！リバースエンジニアリングの道

第3回 アセンブラの学習

文●愛甲健二

アセンブラの必要性

ソフトウェア解析において、アセンブラの知識は少なからず必要です。21世紀になり、JavaやC#がメジャーなプログラミング言語になって久しいこの時代においても、ソフトウェア解析、脆弱性監査の現場においては今なおアセンブルコードが読まれ続けています。

アセンブラについて一切知らなくとも、OllyDbgの使い方さえ知っていればそれなりに解析はできるのですが、やはりどうしてもある一定のレベルで壁にぶつかってしまいます。その意味で、アセンブラはリバースエンジニアリングやソフトウェアにおけるセキュリティ技術を学ぶ上での基礎といえるかもしれません。

さて、連載も3回目となり、リバースエンジニアリングに関する概要もある程度つかめてきたと思います。今回は一度基礎に立ち返る意味も含めて、アセンブラそのものについて学んでいくことにします。なお、本連載のバックナンバーは付録DVD-ROMに収録していますので、必要に応じて参照してください。

アセンブラとリンカ

EclipseやVisual Studioをはじめとした近年の開発環境は、ソフトウェア開発に関するあらゆるツールが内部統合されており、ビルドボタンをクリックするだけで実行ファイル生成までを自動で行ってくれます。しかし、実際は開発環境の内部でさまざまなツールが逐次動作しており、実行ファイル生成までにいくつもの処理を行っています。簡単に言うと、C/C++で書かれたファイル.cppがアセンブラファイル.asmに変換され、その.asmがオブジェクトファイル.objになり、その.objが実行ファイル.exeになる、といった具合です。.cppを.asmもしくは.objへ変換するツールを「コンパイラ」と呼び、.asmを.objへ変換するツールを「アセンブラ」と呼び、そして.objを.exeへ変換するツールを「リンカ」と呼びます。

つまり.asmから.exeを作るためにはアセンブラとリンカが必要です。本記事ではアセンブラにNASM※1、リンカにALINK※2を使います。どちらも付録DVD-ROMに収録していますが、確実に最新版を使いたい場合は脚注のURLからダウンロードしてください。

NASMのZIPファイルを展開するとnasm.exeとndisasm.exeがあります。nasm.exeがアセンブラで、ndisasm.exeが逆アセンブラです。ALINKのプログラム本体（ALINK.EXE）と、WindowsのAPI群を呼び出すためのライブラリ（WIN32.LIB）も展開し、nasm.exeと同じフォルダへコピーします。つまりnasm.exe、ndisasm.exe、ALINK.EXE、WIN32.LIBが同じフォルダに存在する状態にします。これでNASM、ALINKを用いてアセンブラでWindowsアプリケーションを作成する環境が整いました。

Hello World!

まずは、NASM、ALINKに慣れるために文字列「Hello World!」を表示するだけの簡単なプログラムを作成します。

NASMプログラミングでは、ソースコードの中に最低限以下の項目が必要です。

・利用するAPIの定義
・プログラムが開始される場所（エントリポイント）
・命令コードを配置する場所（テキストセクション）
・扱うデータを配置する場所（データセクション）

テキストエディタを使って、上記項目を含むソースコードを書いてみましょう（hello32.asm、下記）。「Hello World!」を表示するために、まずMessageBoxAというAPIを定義します。次に、global mainと記述し、エントリポイントをmainに仮設定します。あとはテキストセクションであるsection .text以降にプログラムを書いていき、その中で利用するデータをデータセクションであるsection .data以降に置きます。

```
extern MessageBoxA
section .text
global main

main:
    push dword 0
    push dword title
    push dword text
    push dword 0
    call MessageBoxA
    ret
```

※1 NASM http://www.nasm.us/
※2 ALINK http://alink.sourceforge.net/download.html

```
section .data
title: db 'MessageBox', 0
text:  db 'Hello World!', 0
```

MessageBoxAは4つの引数を取ります。親ウインドウハンドル、表示するメッセージ、表示するメッセージボックスのタイトル、表示するメッセージボックスの種類の4つです。APIに関する詳細はMSDN（Microsoft Developer Network）で検索できます※3。

とりあえずメッセージボックスを表示するだけならば、第1引数と第4引数は0で問題ありません。あとは表示するメッセージを「Hello World!」、タイトルを「MessageBox」として、引数を後ろから順番にスタックへpushします。そして、call MessageBoxAでAPIを呼び出します。

ここでpushとcallというアセンブラ命令が使われていますが、任意の関数（API）を呼び出すためには、その関数に必要な引数を後ろから順番にスタックへpushして、call命令を実行します。これがC言語でのMessageBoxA(0, text, title, 0)と同じ意味になります。

では実行してみましょう。コマンドプロンプトで、まずはNASMで-fwin32オプションを付けて.objファイルにアセンブルし、次にALINKを使い実行ファイルを生成します。正常に実行ファイルが生成されたら「Generat-ing PE file hello32.exe」と表示されます。

```
C:¥nasm>nasm -fwin32 hello32.asm
C:¥nasm>alink -oPE hello32 win32.
lib -entry main
（中略）
Generating PE file hello32.exe
```

実行ファイルが生成されたら、そのままダブルクリックして実行してください。「Hello World!」と書かれたメッセージボックスが表示されます。

作成したhello32.exeをOllyDbgで開くと、左上の逆アセンブルウインドウには先ほど記述したアセンブラ命令があり、左下のメモリウインドウにはsection .data以降に書いたデータがあります。そして1命令ずつ進めていくと、push命令が実行されるたびに右下のスタックウインドウへ値が格納されていきます。最後にcall MessageBoxAでメッセージボックスの表示です。

以上の実行経過を見て、アセンブラで記述したプログラムがOllyDbg上でどのように表示されるのかをイメージできたでしょうか？ また逆に、OllyDbg上に表示されている実行ファイルは元々どのようなソースコードだったのか。こういった「ソースコードと実行ファイルの比較」を頭の中でイメージできることが、未知のソフトウェアを解析する際にも非常に役立つスキルとなります。この技術を磨いていけば、OllyDbgで実行ファイルを解析しながら頭の中でC/C++のソースコードをイメージできるようになります。

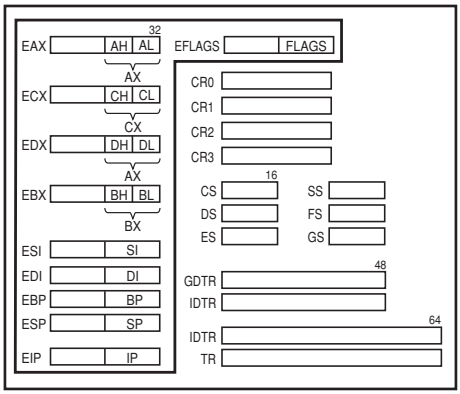

図　x86系CPUのレジスタ

レジスタ

誤解を恐れずに説明すると、レジスタとはCPUが持つメモリ空間です。OllyDbgを起動した際、右上のウインドウに表示されているのがレジスタです。

レジスタはCPUが保持するものなので、当然CPUごとに違います。例えばZ80とi386では保持するレジスタの数、サイズが異なりますし、マシン語命令もCPU（アーキテクチャ）ごとに違うものなので、x86とARMでは命令セットは異なります。本記事では一般的なPCにて使われるx86系CPUについて記述します。

図はx86系CPUが持つレジスタのうち、よく使われるものです。線で囲っているのが、一般的なアプリケーションを解析する際に最低限覚えておくべき部分です。

まず、EAXからEFLAGSまですべて32ビット（4バイト）です。そしてEAX～EBXまでは下位8ビットへAL、BLといった感じでアクセスできます。EAX～EDIについては基本的に自由に利用してよいレジスタとなっており「汎用レジスタ」と呼ばれます。

EBPは「ベースポインタ」と呼ばれ、スタックの底を示し、逆にESPはスタックのトップを示す「スタックポインタ」です。EIPは「命令ポインタ」と呼ばれ、現在実行している命令のアドレスを示します。汎用レジスタとは違い、これらのレジスタは用途に応じて使う必要があります。

最後にEFLAGSですが、これは「フラグレジスタ」と呼ばれ、演算の結果を元に変化するフラグが入ります。詳しい説明はCMP命令を解説する際に行います。

※3　MessageBox　http://msdn.microsoft.com/ja-jp/library/cc410914.aspx

マシン語命令（ニーモニック）

x86系CPUがサポートするマシン語命令はすでに1000命令を超えているらしく、すべての命令を覚えきるのは至難の業ですし、その必要もありません。ただ、使われる頻度が高い命令については、あらかじめ覚えておかなければ話になりませんので、本記事ではそこに重点を置いて解説していきます。

表1は、加算減算を行うものを中心としたもっとも基本的な命令です。足し算はADD命令、引き算はSUB命令で実現します※4。INC命令とDEC命令はそれぞれ「1だけ加算する」と「1だけ減算する」という命令です。MOVは値を代入する命令ですが、MOVZXはサイズを気にせずに値を代入する命令です。LEA命令も同様に、演算結果を「デスティネーションオペランド」へ格納する命令だと覚えてください。オペランドとは命令を実行する対象となるレジスタ、定数、メモリなどのことであり、デスティネーションオペランドは変わる値（NASMだと左側）、ソースオペランドは元となる値（NASMだと右側）です。なので、MOV EAX, 10hという命令があったら、NASMではEAXがデスティネーションオペランドであり、10hがソースオペランドとなります。

では、3×5を計算してみましょう。それには3を5回加算すればよいわけですが、ADD命令をプログラム内に5回も記述するのは美しくありませんので、表2で示した命令による条件分岐を使って3を5回加算するプログラムを書きます（addtest.asm）。

表1　マシン語命令（加減算関連）

命令	例	説明
MOV	MOV EAX, ECX	ECXの値をEAXへ格納（EAX = ECX）
MOVZX	MOVZX EAX, CX	CXの値をEAXへ格納（EAX = CX）
ADD	ADD EAX, ECX	EAXにECXを加算してEAXへ格納（EAX += ECX）
SUB	SUB EAX, ECX	EAXからECXを減算してEAXへ格納（EAX -= ECX）
INC	INC EAX	EAXに1を加算（EAX++）
DEC	DEC EAX	EAXから1を減算（EAX--）
LEA	LEA EAX, [ECX+4]	ECX+4の値をEAXへ格納（EAX = ECX+4）

表2　マシン語命令（条件分岐関連）

命令	例	説明
CMP	CMP EAX, 1	EAXと1を比較してフラグへ反映 もしEAX＞1ならばZF＝0、CF＝0 もしEAX＝1ならばZF＝1、CF＝0 もしEAX＜1ならばZF＝0、CF＝1
TEST	INC EAX	EAXに1を加算（EAX++）
JE（JZ）	JE <アドレス>	ZFフラグが1ならば<アドレス>へジャンプ
JNE（JNZ）	JNE <アドレス>	ZFフラグが0ならば<アドレス>へジャンプ
JMP	JMP <アドレス>	無条件で<アドレス>へジャンプ

表3　マシン語命令（スタック関連）

命令	例	説明
PUSH	PUSH EAX	ESPを4減算しESPのアドレスへEAXを格納
POP	POP EAX	ESPのアドレスの値をEAXへ格納しESPを4加算
CALL	CALL <アドレス>	次に実行すべき命令のアドレスをPUSHして<アドレス>へジャンプする
LEAVE	LEAVE	“MOV ESP, EBP”→“POP EBP”と同じ
RET	RET <値>	ESPのアドレスの値へジャンプする。その際、ESPは4＋<値>だけ加算される

```
section .text
global main
main:
    mov eax, 0h
    mov ecx, 5h
.while:
    add eax, 3h
    dec ecx
    cmp ecx, 0h
    jnz .while
    nop
```

addtest.asmは3×5の計算結果をEAXレジスタへ格納するプログラムです。CMP命令でECXレジスタと0を比較し、もしECXが0でなければ.whileへジャンプします。つまり.while:からjnz .whileまでの命令が5回ループします。OllyDbgで確認してください。

たいてい、CMP、TEST、そしてJZ、JNZといったジャンプ系命令はセットで使われます。CMP A, Bという命令を呼び出した際、AとBがイコールならばEFLAGSレジスタ内にあるZFフラグが1になります。そして、JE（JZ）命令はZFフラグが1ならジャンプする命令、JNE（JNZ）命令はZFフラグが0ならジャンプする命令です。つまり、CMP命令でZFフラグを変化させて、JNZ命令でECXが0になるまで.whileへ処理をジャンプさせます。ちなみに0との比較、つまり真偽のみの判定にはTEST命令もよく用いられます。

次はスタック操作系の命令です（表3）。OllyDbgでは右下のウインドウが現在のスタックを表すものですが、これはESPの値を元に表示されています。よってESPの値を変更すると右下のウインドウにあるアドレス帯域も変更されます。

スタックと聞くと、FILO（First In Last Out）のアルゴリズムを思い浮かべるかもしれませんが、マシン語レベルにおいてはESPを操作すること、つまり、PUSH、POPにてESPを増減させることでFILOを実現します。

また、マシン語レベルでは「関数呼び出し」や「関数への引数渡し」、そして「ローカル変数」

※4　実はアセンブラには乗算除算用の命令もあるのですが（MUL、DIV命令）、実際にはこれらはシフト命令で代用されることが多く、出現頻度は高くありませんので今回は省略しました。

などにスタックを利用します。関数を呼び出す際はCALL命令を使うのですが、呼び出した関数の処理が終われば、また呼び出し元に戻らなければなりません。よって、CALLする際にあらかじめ戻り先アドレスをスタックへ積んでおきます。これで呼び出した関数の処理が終了したら、RET命令で呼び出し元へ戻り、再び処理を続けられます（calltest.asm）。

```
section .text
global main
func:
    add eax, 10h
    ret
main:
    mov eax, 5h
    call func
    ret
```

「関数呼び出し」はわかったとして、次は「関数への引数渡し」と「ローカル変数」についてですが、これらはEBPとESPを使って実現します（calltest2.asm）。

```
section .text
global main
func:
    push ebp
    mov ebp, esp
    sub esp, 4h
    mov eax, dword [ebp+8]
    mov dword [ebp-4], eax
    add dword [ebp-4], 10h
    mov eax, dword [ebp-4]
    leave
    ret
main:
    mov eax, 5h
    push eax
    call func
    ret
```

calltest2.asmはcalltest.asmと動作としては同じプログラムですが、func関数への引数渡しにEAXレジスタではなく、スタックが使われています。またfunc関数が4バイトのローカルを使用しています。

まずfunc関数の1行目のpush ebpでEBPの値をスタックへ保存します。続いてESPの値をEBPに入れるのですが、これはいわゆるESPのバックアップと考えてください。ESPはPUSH、POP、CALLなどが呼ばれるたびに随時変更されるため、特定のメモリアドレスを指すのには向きません。しかし、func関数への引数はスタックへ積まれていますので、つまりはESPを加算したところにあります。なので、ESPの代わりにEBPを使って関数への引数にアクセスしようということで、push ebp、mov ebp, espと実行し、さらにfunc関数が終わったらこれらを元の値に戻しておかなければならないのでLEAVE命令がretの1つ前にあるわけです。

そして、次のsub esp, 4hはローカル変数のための領域を確保しています。この時点でfunc関数への引数には[ebp+8]としてアクセスでき、かつ、ローカル変数には[ebp-4]としてアクセスできます。これが関数呼び出しに関するマシン語レベルの実装です。

sample.exeを解析してみよう

さて、そろそろ終わりに近づいてきましたが、最後に、本連載初めての宿題を出させていただきたいと思います（笑）。付録DVD-ROMに、私が作成したプログラムsample.exeが収録されています。このプログラムは引数に任意の文字列を渡すと、何かしらのエンコードを行った後、メッセージボックスを使って結果を出力します。

宿題は、「今回学んだことを利用して、このsample.exeが使っているエンコード方法をC言語で再現する」ことです。つまり、sample.exeと全く同じプログラムをC言語で作成してください。といっても、いきなりプログラムそのものを再現するのは難しいため、作成するプログラムの大まかな部分を以下に示します。

```
int main(int argc, char *argv[])
{
    unsigned int i;
    char data[256];
    char t[] =
      "[redacted]"
      "[redacted]";
    if(argc != 2)
      sprintf(data, "HELLO");
    else
        sprintf(data, "%s",
argv[1]);
    for(i=0; [redacted]; i++)
      data[i] = [redacted];
    printf("data=%s¥n", data);
    return 0;
}
```

エンコードを行う際に重要な4行のみを削除しています。sample.exeを解析し、アセンブラ命令を解析して、黒塗りの部分を見事特定してください。次回、解答編を行いたいと思います。なおencode.cppをコンパイル、実行したい場合は<stdio.h>と<string.h>をincludeすることを忘れないでください。では、またお会いしましょう。