

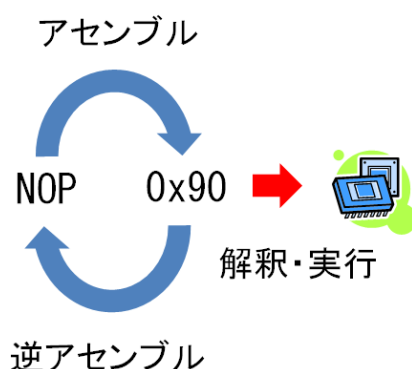
# Shellcoder's Challenge の手引き

## ■基礎知識

マシン語（機械語）とは、一般的にはプロセッサが理解できるバイナリのことを指します。私たちが普段使っているパソコンには大抵 x86 系のプロセッサが搭載されていますから、0xEBFE や 0x90、0x61 などがマシン語にあたります。このテキストでも x86 のマシン語、アセンブリ言語を対象とします。

マシン語だと読みにくいので、それを少し（人間にとって）読みやすくしたものがアセンブリ言語（アセンブラ）です。厳密には「アセンブラ」はアセンブリ言語からマシン語へ変換するプログラムのことですが、この辺りの厳密性はあまり意識されず、どちらもアセンブラと言っても意味は通じます。

そして、アセンブリ言語からマシン語に変換することを「アセンブル」、マシン語からアセンブリ言語に変換することを「逆アセンブル」と呼びます。



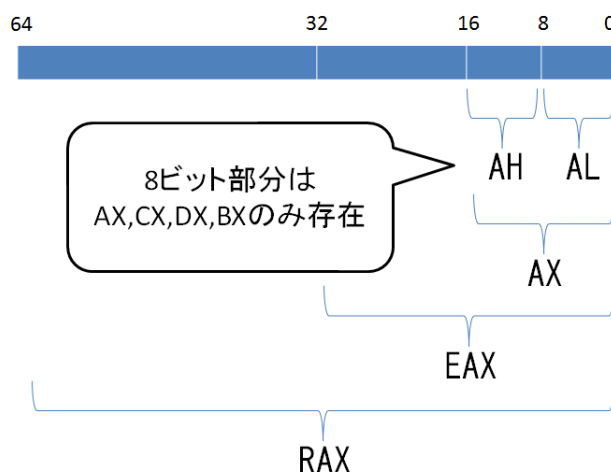
アセンブラを学ぶ上で一番はじめに覚えるのが（多分）レジスタです。

レジスタとは CPU が内部に持つ記憶領域です。その記憶領域にはそれぞれ EAX、ECX、EDX、EBX、ESP、EBP、ESI、EDI、EIP といった名前がついています（他にもあります）。そして、これらレジスタにはそれぞれ用途があり、たとえば ESP と EBP はスタックの管理に使用され、EIP は現在実行中の命令を指しています。それ以外の EAX～EBX、そして ESI、EDI は基本的に自由に使えるメモリ領域だと思ってください。

ちなみにこれらは 32 ビット環境を想定しています。もし 16 ビットなら AX、CX、DX、BX、SP、BP、SI、DI、IP という名前で、逆に 64 ビットなら RAX、RCX、RDX、RBX、RSP、RBP、RSI、RDI、RIP という名前になっており、それぞれの領域のサイズは 32 ビット環境なら 32 ビット（4 バイト）、16 ビット環境なら 16 ビット（2 バイト）、そして 64 ビット環境なら 64 ビット（8 バイト）になります。

そしてこれらはビットが上がるごとに拡張されているだけなので、例えば 32 ビット環境

で AX を使うと、それは EAX の下位 16 ビットを意味します。AX にさらに 16 ビット追加したものが EAX というイメージです。もちろん 64 ビット環境で EAX を使った場合も同じです。RAX の下位 32 ビットが参照されます。これは上記に上げたすべてのレジスタで同様です。



これらとは別にフラグレジスタと呼ばれるものがあります。フラグレジスタは条件分岐のために使用されます。それぞれ条件に応じて変更されるいくつかのフラグ (CF、PF、AF、ZF などなど) があるのですが、とりあえず ZF だけ覚えてください。

フラグレジスタは、例えば ZF が 1 ならばジャンプする、ZF が 0 ならばジャンプしないといった感じで `jmp` 系命令といっしょに使用されます。フラグレジスタを変更する代表的なものに `cmp` と `test` があります。このように使用されます。

```
0x004004ff:  cmpl  $0x0,%eax  # eax が 0 なら ZF を 1 にする
0x00400503:  je     0x40050c    # ZF が 1 なら 0x40050c へジャンプする
```

(※) AT&T 記法

次にアセンブラ命令について解説します。

一般的なプログラミング言語の予約語と比べても、アセンブラ命令の多さは尋常ではありません。たしか x86 だと 1000 近い命令があるみたいです。ただ Windows プログラマが Windows API の関数群をすべて覚えていないのと同じく、覚えておくべきアセンブラ命令もそれほど多くないです。もし知らない命令があったとしても、そのつど調べればよいだけであり、実際に覚えておかなければならないのはせいぜい 20 命令ほどです。

ここで筆者の独断と偏見から選別された、よく使われているアセンブラ命令を紹介します。あくまで筆者の感覚なので本当のところはわかりませんが、このくらいの命令を知っていればなんとかなるのではないかな、と思われる命令群です。

ちなみに、わかりやすさのために正確性をかなり犠牲にしているため、アセンブラに慣

れてきたらあらためて各命令について深く調べてみてください。

命令	例	意味	説明
MOV	MOV EAX, ECX	EAX = ECX	ECXの値をEAXへ格納
ADD	ADD EAX, ECX	EAX += ECX	EAXにECXを加算
SUB	SUB EAX, ECX	EAX -= ECX	EAXからECXを減算
INC	INC EAX	EAX++	EAXに1を加算
DEC	DEC EAX	EAX--	EAXから1を減算
LEA	LEA EAX, [ECX+4]	EAX = ECX+4	ECX+4をEAXへ格納
CMP	CMP EAX, ECX	if (EAX == ECX) ZF=1 else ZF=0	値を比較してフラグへ反映 EAXとECXが同じならばZF=1 EAXとECXが違えばZF=0
TEST	TEST EAX, EAX	if (EAX == 0) ZF=1 else ZF=0	値を0と比較してフラグへ反映 EAXが0ならば ZF=1 EAXが0以外ならばZF=0
JE (JZ)	JE 04001000	if (ZF==1) GOTO 04001000	ZFが1なら04001000へジャンプ
JNE (JNZ)	JNE 04001000	if (ZF==0) GOTO 04001000	ZFが0なら04001000へジャンプ
JMP	JMP 04001000	GOTO 04001000	無条件で04001000へジャンプ
CALL	CALL IstcmpW		IstcmpWの呼び出し
PUSH	PUSH 00000001		スタックへ00000001を格納
POP	POP EAX		スタックからEAXへ値を取得

(※) Intel 記法

プログラミングの経験があれば、半分以上の命令は、この表を眺めただけでなんとなく理解できるでしょう。説明が必要なのは cmp、test と je、jne だと思いますが、これらはアセンブラにおける条件分岐です。

一般的なプログラミング言語ならば、if や switch といった予約語で条件分岐を記述しますが、アセンブラのばあいはフラグを操作する cmp、test と、フラグの値によって処理が分岐するジャンプ系命令によってそれを実現します。

```
00401019 test    eax, eax          ◆ 比較の処理
0040101B jnz     short loc_401035 ◆ 条件分岐の処理
```

test は、ありていにいえばフラグのみを変化させる and 命令なのですが、そもそも and 命令とは何か、といった話になりますし、それがわかってもしまいち理解しにくいので、かなりざっくりと説明しますと、test eax,eax は eax が 0 ならば ZF を 1 にします。そして

test 命令はほとんどのばあいにおいて test eax,eax や test ecx,ecx といったように同じレジスタが 2 つ渡される形で使われます。なので同じレジスタが渡されている test 命令を見つけたら、そのレジスタが 0 ならば ZF を 1 にするのだな、と覚えておけばよいです。

あと、ついでに AT&T 記法と Intel 記法についても説明しておきます。アセンブリ言語には 2 つの記法があります。以下に EAX に 0 を入れる処理をそれぞれの記法で紹介します。

#### AT&T 記法

mov 0x0, eax # 左から右へ代入される

#### Intel 記法

mov eax, 0x0 # 右から左へ代入される

そもそも mov や jmp といったものをオペコードと呼び、それに続く eax や 0x0 といったものをオペランドと呼びます。そしてオペランドには「ソース」と「デスティネーション」の二種類があります。ソースはデータとして読み取られる方、デスティネーションは実行結果が格納される方です。

そして AT&T と Intel、それぞれの記法において、ソースとデスティネーションの順序が違います。出力されたアセンブラコードがどちらの記法なのかはわかりませんので、ざっと見て判断しなければなりません。といっても慣れれば簡単ですが…。

さて早足ではありましたが、基礎知識はこの辺にしてさっそく **Shellcoder's Challenge** に挑戦していきましょう。

## ■ レベル 0

ではさっそくレベル 0 の問題を見てみましょう。ちなみにこれは AT&T 記法ですね。

### Level 0 ★

以下のコードが実行された。  
処理が 0x00400519 にきたときの %eax の値を 10 進数 で答えよ。

```
(gdb) disas func
Dump of assembler code for function func:
0x004004f4 <+0>:    push    %rbp
0x004004f5 <+1>:    mov     %rsp,%rbp
0x004004f8 <+4>:    movl    $0x400,-0xc(%rbp)
0x004004ff <+11>:   movl    $0x3a98,-0x8(%rbp)
0x00400506 <+18>:   movl    $0xd,-0x4(%rbp)
0x0040050d <+25>:   mov     -0x8(%rbp),%eax
0x00400510 <+28>:   mov     -0xc(%rbp),%edx
0x00400513 <+31>:   add     %edx,%eax
0x00400515 <+33>:   sub     -0x4(%rbp),%eax
0x00400518 <+36>:   pop     %rbp
0x00400519 <+37>:   retq
End of assembler dump.
```

Hint1: よくわからない命令はとりあえず無視しよう

Hint2: -0xX(%rbp)といったものは変数と考えよう

Hint3: あたまた 0x がついている数値は 16 進数 で考えよう

最初はとても難しそうにみえますが、1 命令ずつ分解していけば簡単に理解できます。まずはヒントに従って、よくわからない命令は無視し、-0xX(%rbp)といったものは変数（自分の好きな名前）に置き換えていきましょう。また movl といったような最後に l や b や w がつく命令があったりしますが、無視して OK です。一応説明するとオペランドのサイズによって変わります。l は 4 バイト、w は 2 バイト、b は 1 バイトのオペランドに対して処理するときに使われます。

```
(gdb) disas func
```

```
Dump of assembler code for function func:
```

```
0x004004f4 <+0>:    push    %rbp
0x004004f5 <+1>:    mov     %rsp,%rbp
0x004004f8 <+4>:    movl    $0x400, val0C # val0C = 0x400
0x004004ff <+11>:   movl    $0x3a98, val08 # val08 = 0x3a98
0x00400506 <+18>:   movl    $0xd, val04 # val04 = 0xd
0x0040050d <+25>:   mov     val08,%eax # eax = val08 (0x3a98)
0x00400510 <+28>:   mov     val0C,%edx # edx = val0C (0x400)
0x00400513 <+31>:   add     %edx,%eax # eax = eax + edx
0x00400515 <+33>:   sub     val04,%eax # eax = eax - val04 (0xd)
```

```
0x00400518 <+36>:    pop    %rbp
```

```
0x00400519 <+37>:    retq
```

End of assembler dump.

最初の 2 行と最後の 2 行はよくわからないのでとりあえず置いておいて、残りの命令をわかりやすくしてみました。

先頭に 0x がついている数値は 16 進数です。16 進数と 10 進数の相互変換は Windows なら電卓、UNIX 系なら 1 行スクリプトが便利です。

```
# python -c 'print int(0x10), hex(10)'
```

```
16 0xa
```

では 0x がついている数値を 10 進数に変換しましょう。

```
# python -c 'print int(0x400), int(0x3a98), int(0xd)'
```

```
1024 15000 13
```

それぞれ 1024、15000、そして 13 となりました。ではこれを前提にもう一度コードを見てください。途中でレジスタが使われているだけですので、ちゃんと読めば答えが分かると思います。

```
0x0040050d <+25>:    mov    val08,%eax    # eax = val08(0x3a98)
```

```
0x00400510 <+28>:    mov    val0C,%edx    # edx = val0C(0x400)
```

```
0x00400513 <+31>:    add    %edx,%eax     # eax = eax + edx
```

```
0x00400515 <+33>:    sub    val04,%eax     # eax = eax - val04(0x0d)
```

最終的に eax は何になるでしょうか？

15000 + 1024 - 13 = ?????

以上でレベル 0 は攻略です。

## ■ レベル 1

レベル 1 もレベル 0 と同様に「わからない命令は無視」 & 「変数に置き換えていく」をやっていけば理解できます。

### Level 1 ★

以下のコードが実行された。  
処理が 0x0040051f にきたときの %eax の値を 10 進数 で答えよ。


```
(gdb) disas func
Dump of assembler code for function func:
0x004004f4 <+0>:    push    %rbp
0x004004f5 <+1>:    mov     %rsp,%rbp
0x004004f8 <+4>:    movl    $0x0,-0x4(%rbp)
0x004004ff <+11>:   movl    $0x0,-0x8(%rbp)
0x00400506 <+18>:   jmp     0x400512 <func+30>
0x00400508 <+20>:   mov     -0x8(%rbp),%eax
0x0040050b <+23>:   add     %eax,-0x4(%rbp)
0x0040050e <+26>:   addl    $0x1,-0x8(%rbp)
0x00400512 <+30>:   cmpl    $0x3ff,-0x8(%rbp)
0x00400519 <+37>:   jle     0x400508 <func+20>
0x0040051b <+39>:   mov     -0x4(%rbp),%eax
0x0040051e <+42>:   pop     %rbp
0x0040051f <+43>:   retq
End of assembler dump.
```

Hint1: アドレス 0x00400512 の cmpl は比較命令

Hint2: 次の jle は cmpl の比較結果によって処理が分岐する

レベル 1 のポイントはヒントにもあるように cmpl と jle による条件分岐です。jmp 系の命令は実際に矢印を書いてあげるとわかりやすいです。

```
0x004004f8 <+4>:    movl    $0x0,-0x4(%rbp)
0x004004ff <+11>:   movl    $0x0,-0x8(%rbp)
0x00400506 <+18>:   jmp     0x400512 <func+30>
0x00400508 <+20>:   mov     -0x8(%rbp),%eax
0x0040050b <+23>:   add     %eax,-0x4(%rbp)
0x0040050e <+26>:   addl    $0x1,-0x8(%rbp)
0x00400512 <+30>:   cmpl    $0x3ff,-0x8(%rbp)
0x00400519 <+37>:   jle     0x400508 <func+20>
0x0040051b <+39>:   mov     -0x4(%rbp),%eax
```



赤い矢印は一度しか実行されませんが、青い矢印は何度も実行されることが分かります。  
-0x8(%rbp)は最初 0x0 ですので、青い矢印のループ内で-0x8(%rbp)がインクリメント (1 加算) されていることから、この値が 0x3ff になるまで青い矢印ループは繰り返されると推測できます。

また-0x4(%rbp)も最初 0x0 ですが、青矢印のループを回るとに-0x8(%rbp)が加算されていきます。

以上から、こういうコードがイメージできるでしょう。

movl \$0x0, -0x4(%rbp)	Val04 = 0x0
movl \$0x0, -0x8(%rbp)	Val08 = 0x0
jmp 0x400512 <func+30>	Goto func+30
<func+20>	func+20:
mov -0x8(%rbp), %eax	Val08 = eax
add %eax, -0x4(%rbp)	Val04 = Val04 + eax
addl \$0x1, -0x8(%rbp)	Val08 = Val08 + 1
<func+30>	func+30:
cmpl \$0x3ff, -0x8(%rbp)	If (val08 <= 0x3ff)
jle 0x400508 <func+20>	Goto func+20
mov -0x4(%rbp), %eax	eax = Val04

よって 0x0～0x3ff までの数をすべて加算したものが、最終的に eax に代入されます。

以上でレベル 1 は攻略です。

では最後にここまでのポイントを列挙しておきます。

- 変数に置き換えると読みやすくなる
- ジャンプ系命令は矢印をつけるとわかりやすい
- わからない命令はとりあえず無視する（行き詰まったら Web で調べる）
- 先頭と終端の 2 命令は（とりあえずは）理解してなくても良い



## ■ レベル 2

レベル 0、レベル 1 で学んだことをベースにレベル 2 を解いていきましょう。レベル 2 からは手動で逆コンパイル（高級言語のソースコードに戻すこと）を目標にしていきます。また「その関数はいったい何をするものなのか？」も意識しながら解析してってください。

### Level 2 ★

%edi に 0x0a を入れた状態で以下のコードが実行された。  
処理が 0x0040051e にきたときの %eax の値を 10進数 で答えよ。

```
(gdb) disas func
Dump of assembler code for function func:
0x004004f4 <+0>:    push    %rbp
0x004004f5 <+1>:    mov     %rsp,%rbp
0x004004f8 <+4>:    sub     $0x10,%rsp
0x004004fc <+8>:    mov     %edi,-0x4(%rbp)
0x004004ff <+11>:   cmpl    $0x0,-0x4(%rbp)
0x00400503 <+15>:   jne     0x40050c <func+24>
0x00400505 <+17>:   mov     $0x1,%eax
0x0040050a <+22>:   jmp     0x40051d <func+41>
0x0040050c <+24>:   mov     -0x4(%rbp),%eax
0x0040050f <+27>:   sub     $0x1,%eax
0x00400512 <+30>:   mov     %eax,%edi
0x00400514 <+32>:   callq   0x4004f4 <func>
0x00400519 <+37>:   imul    -0x4(%rbp),%eax
0x0040051d <+41>:   leaveq  0
0x0040051e <+42>:   retq
End of assembler dump.
```

まずは 0x004004ff の cmpl 命令に注目しましょう。ここではレジスタ edi が 0 かどうかによって処理を分岐させています。

mov    %edi, -0x4(%rbp)	mov    %edi, -0x4(%rbp)
cmpl    \$0x0, -0x4(%rbp)	cmpl    \$0x0, -0x4(%rbp)
jne     0x40050c <func+24>	jne     0x40050c <func+24>
mov     \$0x1, %eax	mov     \$0x1, %eax
jmp     0x40051d <func+41>	jmp     0x40051d <func+41>
<func+24>	<func+24>
mov     -0x4(%rbp), %eax	mov     -0x4(%rbp), %eax
sub     \$0x1, %eax	sub     \$0x1, %eax
mov     %eax, %edi	mov     %eax, %edi
callq   0x4004f4 <func>	callq   0x4004f4 <func>
imul    -0x4(%rbp), %eax	imul    -0x4(%rbp), %eax
<func+41>	<func+41>

レジスタ edi が 0 だったら eax を 1 にして終了です。それ以外ならば edi (-0x4(%rbp)) -1 の値を edi へ入れてもう一度 func を呼び出します。そしてその結果と edi (-0x4(%rbp)) を乗算 (掛け算) した値を eax に入れて終了します。

この関数はいわゆる再帰呼び出しをやっています。レジスタ edi に 0x0a を入れて実行したという問題文なので、このまま処理を追っていても答えは出ますが、ここで「そもそもこれは何をする関数なのか？」を推測しましょう。

アセンブラコードを高級言語っぽく書きなおしてみましょう。

```
int func()
{
    int val04 = edi
    if(val04 == 0)
        return 1
    edi = val04 - 1
    return (val04 * func())
}
```

こういう関数を見たことはないでしょうか。これは階乗を求める関数です。なのでレジスタ edi に 0x0a を入れて実行したのならば、0x0a、つまり 10 の階乗が答えなわけです。

アセンブラのままではわかりにくいものも、少し高級言語っぽくしただけで途端にわかりやすくなったりするものです。

## ■ アセンブラの学び方

最後にアセンブラの学び方について解説します。

まず UNIX 系 OS の場合、gcc と gdb と objdump をインストールしてください。

waka.S

---

```
.global main
main:
    mov    $0x616b6157, %eax
    push   %ebx
    push   %eax
    mov    $4, %edx
    mov    $1, %ebx
    mov    $4, %eax
    mov    %esp, %ecx
    int    $0x80
    pop    %eax
    xor    %eax, %eax
    pop    %ebx
    ret
```

---

```
$ gcc waka.S -o waka
```

```
$ ./waka
```

Waka

こんな感じでアセンブラを試せます。

逆アセンブルする場合は gdb もしくは objdump を使います。

rl0.c

---

```
#include <stdio.h>
int func()
{
    int a = 1024;
    int b = 15000;
    int c = 13;
    return (a+b-c);
}
```

```

}
int main()
{
    printf("%d¥n", func());
    return 0;
}

```

---

console

---

\$ gcc r10.c -o r10

\$ gdb r10

GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04

(gdb) disas func

Dump of assembler code for function func:

```

0x0000000004004f4 <+0>:    push    %rbp
0x0000000004004f5 <+1>:    mov     %rsp,%rbp
0x0000000004004f8 <+4>:    movl    $0x400,-0xc(%rbp)
0x0000000004004ff <+11>:   movl    $0x3a98,-0x8(%rbp)
0x000000000400506 <+18>:   movl    $0xd,-0x4(%rbp)
0x00000000040050d <+25>:   mov     -0x8(%rbp),%eax
0x000000000400510 <+28>:   mov     -0xc(%rbp),%edx
0x000000000400513 <+31>:   add     %edx,%eax
0x000000000400515 <+33>:   sub     -0x4(%rbp),%eax
0x000000000400518 <+36>:   pop     %rbp
0x000000000400519 <+37>:   retq

```

End of assembler dump.

(gdb)

---

console

---

\$ objdump -d r10 | grep ¥<func¥>: -A 11

0000000004004f4 <func>:

```

4004f4:    55                      push    %rbp
4004f5:    48 89 e5               mov     %rsp,%rbp
4004f8:    c7 45 f4 00 04 00 00   movl    $0x400,-0xc(%rbp)
4004ff:    c7 45 f8 98 3a 00 00   movl    $0x3a98,-0x8(%rbp)
400506:    c7 45 fc 0d 00 00 00   movl    $0xd,-0x4(%rbp)
40050d:    8b 45 f8               mov     -0x8(%rbp),%eax

```

400510:	8b 55 f4	mov	-0xc(%rbp), %edx
400513:	01 d0	add	%edx, %eax
400515:	2b 45 fc	sub	-0x4(%rbp), %eax
400518:	5d	pop	%rbp
400519:	c3	retq	

アセンブラを書く機会というのはあまりないと思いますが、読むことに関してはわりとあるかもしれません。基本的に objdump があればどんなアセンブラコードだって読むことができますので、それをきっかけに勉強していくのがよいかもしれません。

gdb はデバッガなのでブレイクポイントなどもつけられますが、読むだけなら objdump の方がよいでしょう。

続いて Windows 環境の場合ですが、こちらは OllyDbg、IDA 辺りをインストールするとよいでしょう。

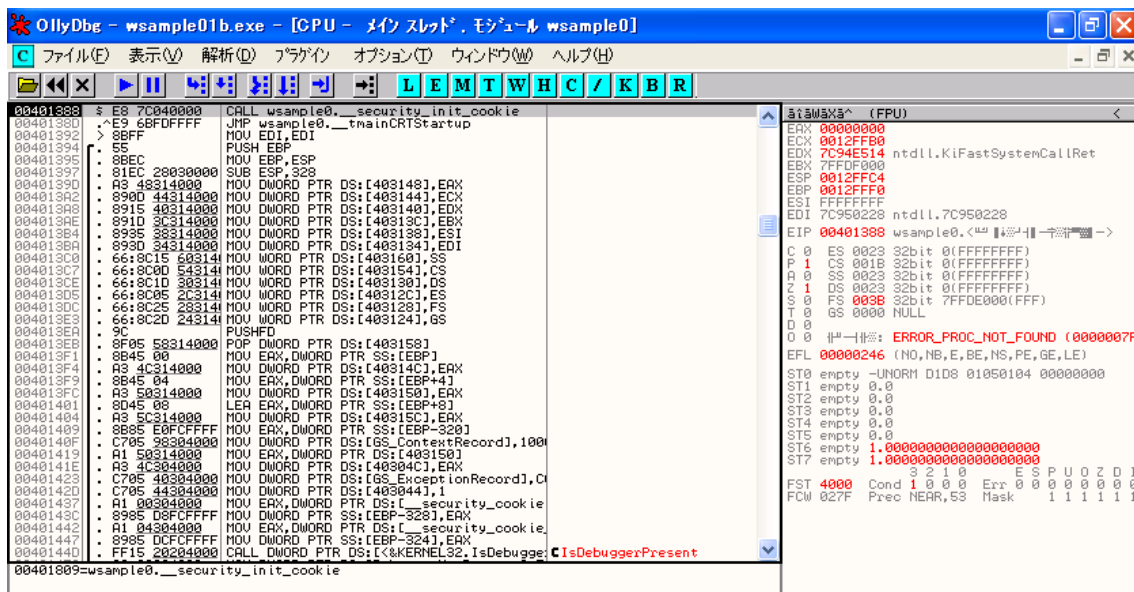
OllyDbg

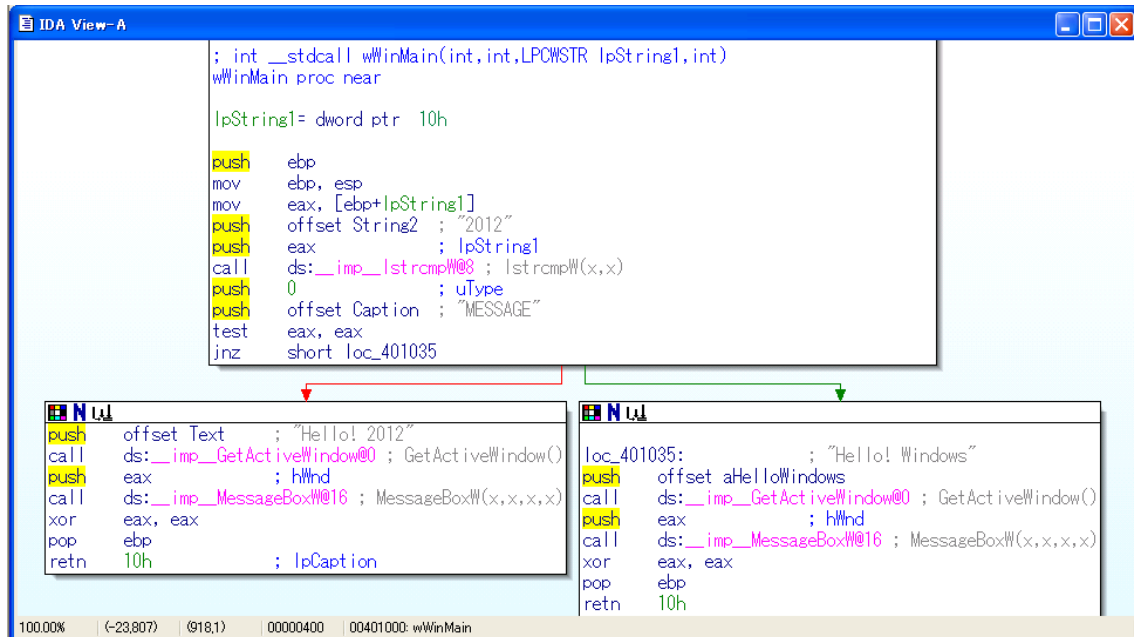
<http://www.ollydbg.de/>

IDA

<https://www.hex-rays.com/products/ida/support/download.shtml>

IDA は逆アセンブラ、OllyDbg はデバッガです。どちらも適当な EXE ファイルをドラッグするだけで OK です。





```
; int __stdcall wWinMain(int,int,LPCWSTR lpString1,int)
wWinMain proc near

lpString1= dword ptr 10h

push    ebp
mov     ebp, esp
mov     eax, [ebp+lpString1]
push    offset String2 ; "2012"
push    eax             ; lpString1
call    ds:__imp__strcmpW@8 ; strcmpW(x,x)
push    0               ; uType
push    offset Caption ; "MESSAGE"
test    eax, eax
jnz     short loc_401035

; "Hello! 2012"
push    offset Text
call    ds:__imp__GetActiveWindow@0 ; GetActiveWindow()
push    eax             ; hWnd
call    ds:__imp__MessageBoxW@16 ; MessageBoxW(x,x,x,x)
xor     eax, eax
pop     ebp
retn    10h             ; lpCaption

loc_401035:
; "Hello! Windows"
push    offset aHelloWindows
call    ds:__imp__GetActiveWindow@0 ; GetActiveWindow()
push    eax             ; hWnd
call    ds:__imp__MessageBoxW@16 ; MessageBoxW(x,x,x,x)
xor     eax, eax
pop     ebp
retn    10h
```

また NASM と alink を使って Windows 環境でアセンブラを試す方法を以下に書きました。もし興味があれば参考にしてください。

CodeIQ Blog

<http://codeiq.hatenablog.com/entry/2013/11/11/130145>

そして最後に Intel のリファレンスを DL できるサイトを。

日本語技術資料のダウンロード

<http://www.intel.co.jp/content/www/jp/ja/developer/download.html>