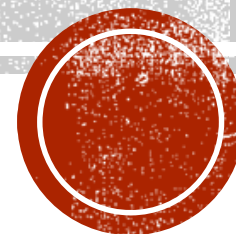


バイナリアンが
何を言っているか
わからない件

三村 聡志



自己紹介

- 三村 聡志 a.k.a. 親方



- CTF Team : wasamusume, mayuge(HITB)

- Twitter : @mimura1133

- Web Site : <http://mimumimu.net/>

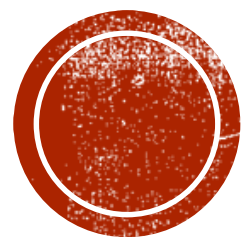
- CTF よりもソフト開発してることの方が多い。



講習内容

- そもそも、バイナリファイルとは？
- ファイルは何かを判定しよう
 - File
- ファイルの中から文字列を取り出してみよう
 - Strings
- 実行ファイルを解析してみよう
 - IDA Pro
- プログラム実行の仕組み

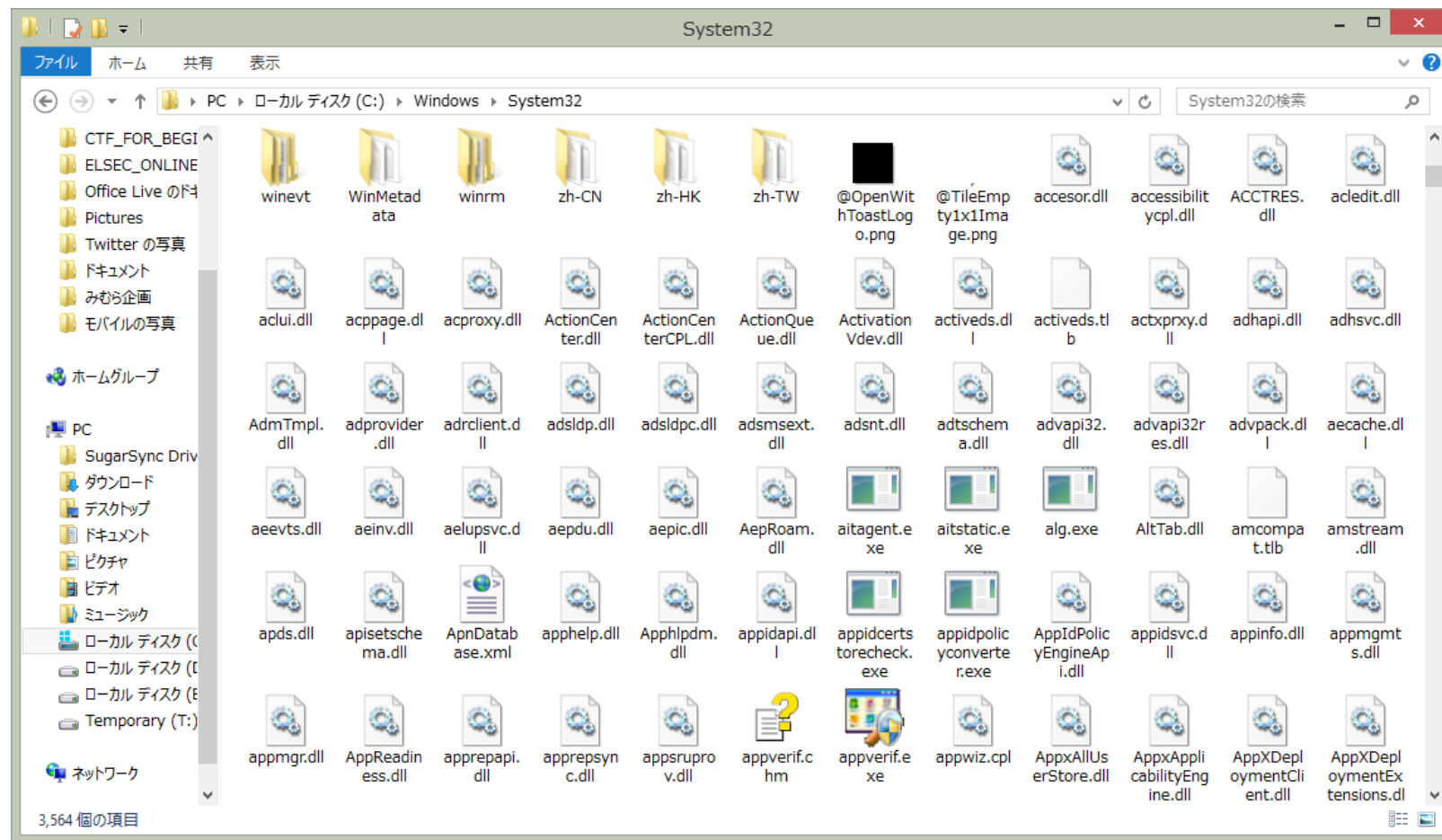




そもそも、
バイナリファイル
とは？

バイナリファイルとは

- コンピュータ内に記録されているファイル全般



バイナリファイルとは

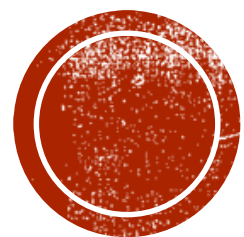
- 実行ファイル (.exe, .dll, .so, .elf etc..)
- 画像ファイル (.jpg, .gif, .png etc..)
- 音声ファイル (.mp3, .wav, .aac, .m4a etc..)
- 動画ファイル (.mp4, .m4v, .avi, .wmv etc..)
- 文書ファイル (.txt, .rtf, .doc, .jtd etc..)
 - “.txt” については、バイナリファイルに対比させて「テキストファイル」として扱う事もありますが今回は「バイナリファイル」に含めて取り扱います。
- などなど・・・



バイナリファイルとは

- CTF における「バイナリファイル」
 - 何のファイルか分からない状態で渡されるため、まず、ファイルの種類を判定する
 - それが実行ファイルなら、逆アセンブル
 - 既知のファイル形式ならその形式で読む
 - 分からなければ分析する





ファイルは何か
判定しよう

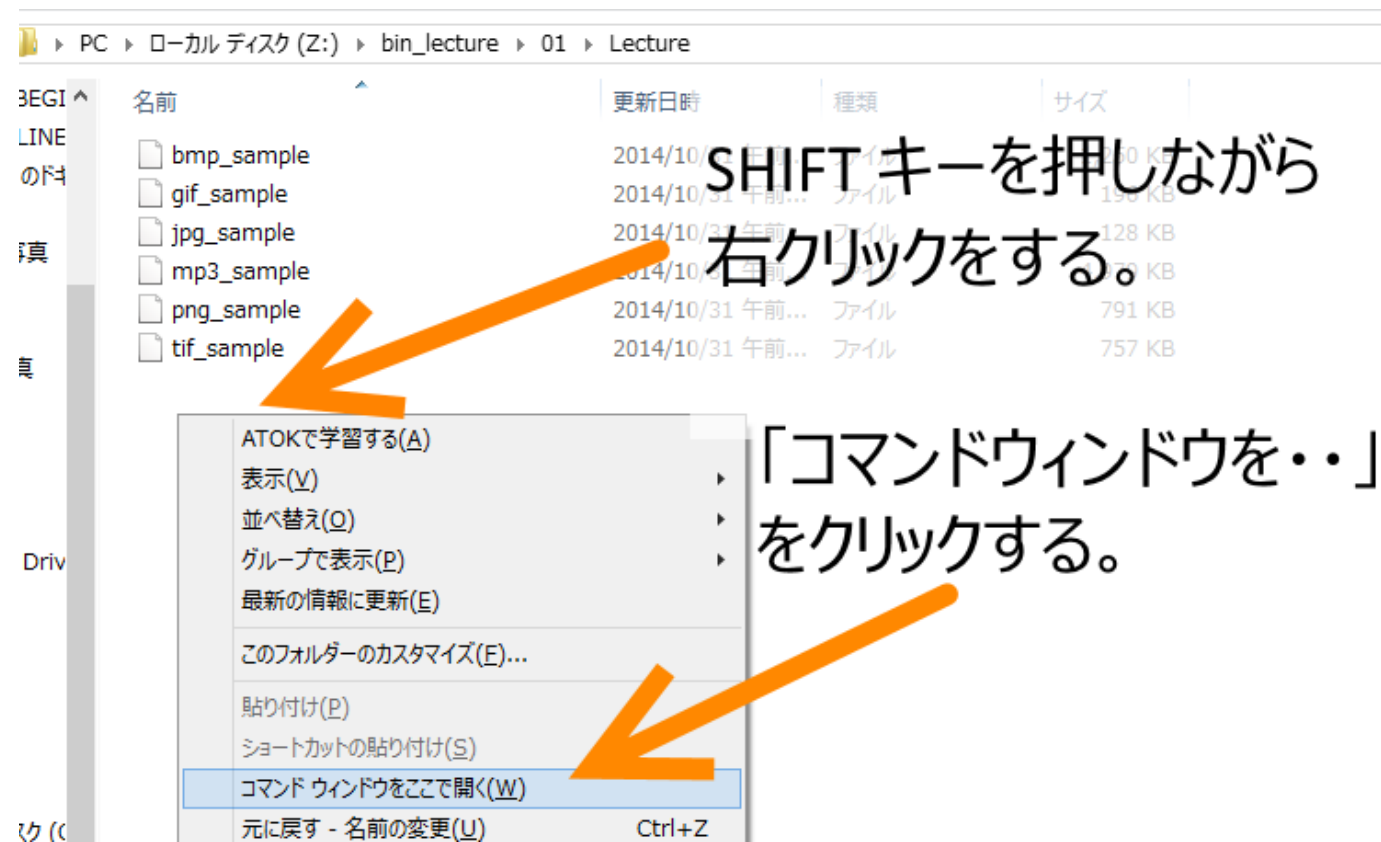
ファイルは何か判定しよう

- ファイルの種類を判定するには
“file” コマンドを使用します。
- それでは例を使って
一緒に判定していきましょう。



ファイルは何か判定しよう

- 配付資料の 01 フォルダを開き、
コマンドウィンドウを開きます。



ファイルは何か判定しよう

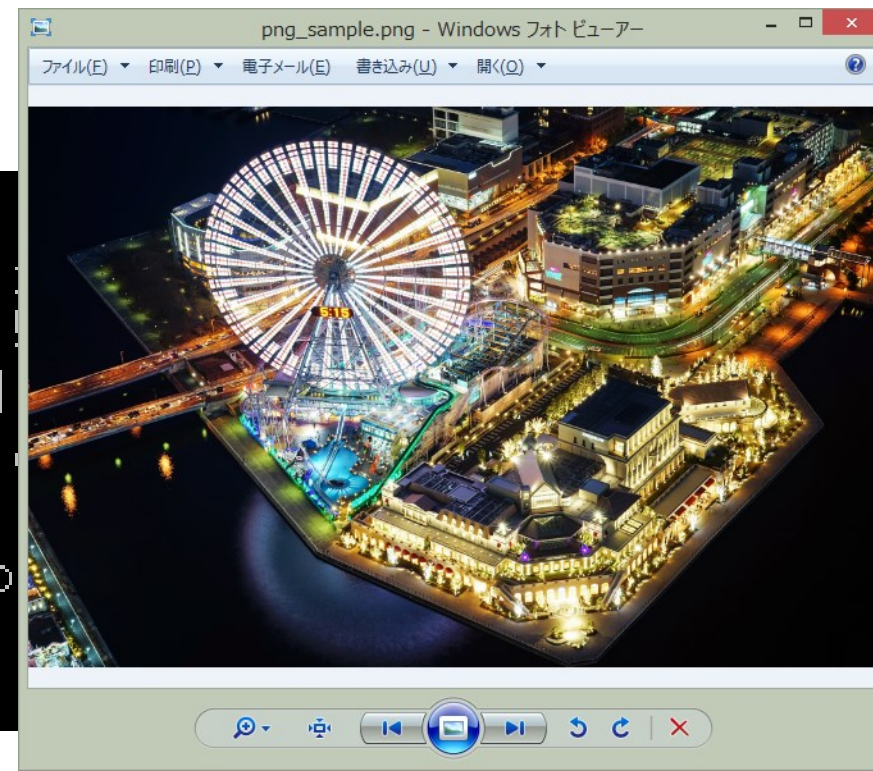
- “file *” と入力。
- ファイルの種類が判定されて出てくる。

```
Z:\bin_lecture\01\Lecture>file *
bmp_sample: PC bitmap, Windows 3.x format, 800 x 533 x 24
gif_sample: GIF image data, version 89a, 800 x 533
jpg_sample: JPEG image data, JFIF standard 1.01
mp3_sample: Audio file with ID3 version 2.3.0, contains: MPEG ADTS, layer III, v
1, 128 kbps, 44.1 kHz, JntStereo
png_sample: PNG image data, 800 x 533, 8-bit/color RGB, non-interlaced
tif_sample: TIFF image data, little-endian
```

ファイルは何か判定しよう

- 試しに、bmp_sample というファイルに
bmp_sample.bmp という拡張子を付けてみると
開ける事が分かる。

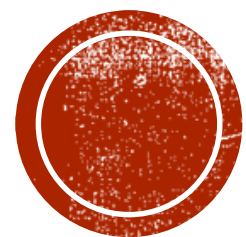
```
Z:\¥bin_lecture¥01¥Lecture>file *  
bmp_sample: PC bitmap, Windows 3.x format, 800 x 533, 32 bits/pixel  
gif_sample: GIF image data, version 89a, 800 x 533, 32 bits/pixel  
jpg_sample: JPEG image data, JFIF standard 1.01, 800 x 533, 32 bits/pixel  
mp3_sample: Audio file with ID3 version 2.3.0, 1, 128 kbps, 44.1 kHz, JntStereo  
png_sample: PNG image data, 800 x 533, 8-bit/color, 32 bits/pixel  
tif_sample: TIFF image data, little-endian, 800 x 533, 32 bits/pixel
```



ファイルは何か判定しよう

- この章のまとめ：
- `file` コマンドでファイルの種類が判定できる
- 必要に応じて拡張子を付ければファイルを開ける

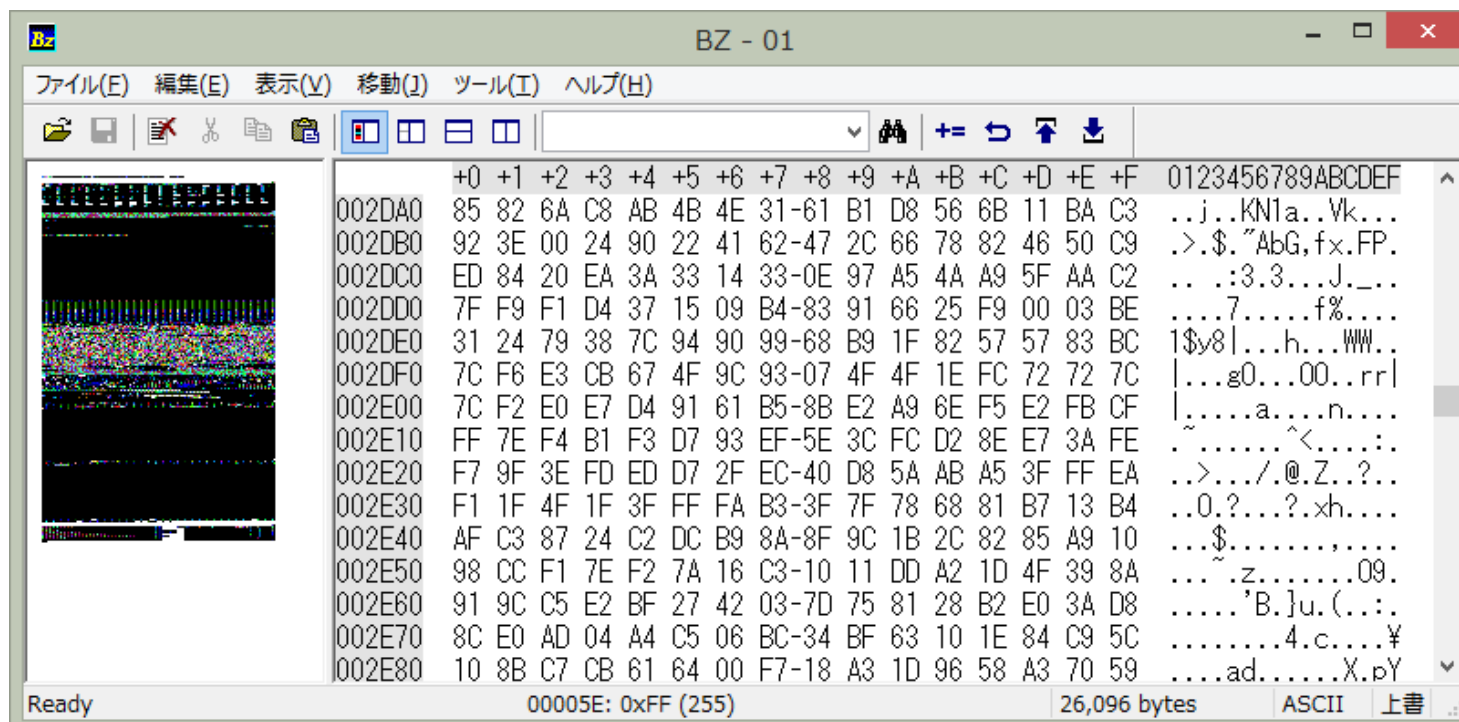




ファイルの中から
文字列を取り出して
みよう

文字列を取り出してみよう

- 文字列を取り出すとは？
- このようによく分からないデータの山から
人が読める文字を抽出すること。



文字列を取り出してみよう

- バイナリファイルから
文字列を抽出するためには “strings” コマンド を
使用します。
- それでは例を使ってやってみましょう。



文字列を取り出してみよう

- 配付資料内の 02 フォルダを開き、
コマンドウィンドウを開きます。
- フォルダの何もないところで、
SHIFT キーを押しながら右クリックし
「コマンドウィンドウをここで開く」を選択。



文字列を取り出してみよう

- まずは、“file 01”を実行して、ファイルの種類を判定してみます。

```
Z:\$bin_lecture¥02>file 01
01: data
```

- “data”と表示されて、ファイル形式が分からない



文字列を取り出してみよう

- 次に “strings -a 01” を実行
- “FLAG_IS_CHALLENGER_1985” という文字列
 - これが解答になる。

```
Z:\¥02>strings 01
```

```
Strings v2.5
```

```
Copyright (C) 1999-2012 Mark Russinovich
```

```
Sysinternals - www.sysinternals.com
```

```
bjbjzqza
```

```
FLAG_IS_CHALLENGER_1985
```

```
nn0h
```

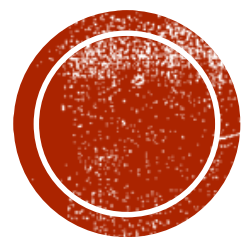
```
0j0W0
```

```
[Content_Types].xml
```

文字列を取り出してみよう

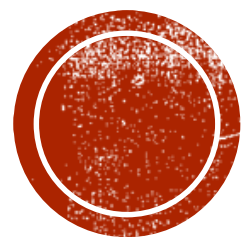
- この章のまとめ：
- Strings コマンドで
ファイル内の文字列を抽出できる。





実行ファイル
解析してみよう

そのまえに・・・



プログラムの 動く仕組みを知ろう

プログラムの動く仕組みを知ろう

- プログラムを組んだことがある人はどれぐらいいますか？
- その中で「コンパイラ言語」を使ったことがある人はどれぐらい？
 - C, C++ が代表的。



プログラムの動く仕組みを知ろう

- 次のプログラムは何をしていますか

```
#include <stdio.h>

void main() {
    printf("HELLO WORLD");
}
```



プログラムの動く仕組みを知ろう

- では、次のプログラムは何をしていますか

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near
push    offset Format      ; "HELLO WORLD"
call    ds:__imp__printf
add     esp, 4
xor     eax, eax
retn
_main endp
```

- 先ほどのプログラムと同じ動作をします



プログラムの動く仕組みを知ろう

- コンパイラ言語は通常次のような手順を追う



- CPU は「オブジェクトコード」を読み
プログラムを実行する



プログラムの動く仕組みを知ろう

- 最初のもものが「ソースコード」

```
#include <stdio.h>
void main() { printf("HELLO WORLD"); }
```

- 次に見せたものがそれをコンパイルした結果の「オブジェクトコード」（を逆アセンブルしたもの）

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near
push    offset Format    ; "HELLO WORLD"
call    ds:__imp__printf
add     esp, 4
xor     eax, eax
retn
_main endp
```



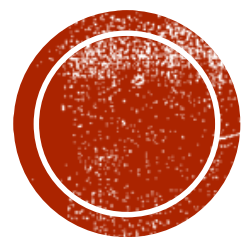
プログラムの動く仕組みを知ろう

- 実際の「オブジェクトコード」
(CPU が実際に読み取るもの)
 - 一般に、人間は読めないのでアセンブラに直したものを読む。

```
ADD; int __cdecl main(int argc, const char **argv, const char **envp)
00000000 _main proc near
00000000 push    offset Format      ; "HELLO WORLD"
call     ds:__imp__printf
add      esp, 4
xor      eax, eax
retn
_main endp
```

EF
§





実行ファイル
解析してみよう

実行ファイルを解析してみよう

- 実行ファイルの解析
 - 実際に動作させてみて結果をみる方法
 - 逆アセンブラにより処理内容を読む方法
 - Etc..
- まずは解析対象のファイルがどういうものかを見ていきましょう。



実行ファイルを解析してみよう

- 配付資料内の 03 フォルダを開いて
コマンドウィンドウを開きます。
- フォルダの何もないところで、
SHIFT キーを押しながら右クリックし
「コマンドウィンドウをここで開く」を選択。



実行ファイルを解析してみよう

- まずは `file` コマンドを使ってファイルを判定します。

```
Z:\03>file 01  
01: PE32 executable for MS Windows (console) Intel 80386 32-bit
```

- `exe` (実行可能形式) ファイルである事が分かる。



実行ファイルを解析してみよう

- ファイル名を“01”から“01.exe”に変更
- 実行してみる (“01” と打って Enter)

```
Z:\¥03>01  
INVALID KEY!!
```

- INVALID KEY と出るだけ。



実行ファイルを解析してみよう

- KEY を見つけるために、
実行ファイルの解析をしてみましょう。
- 今回は “IDA Pro” を用いて
exe ファイルの中身を追っていきます。



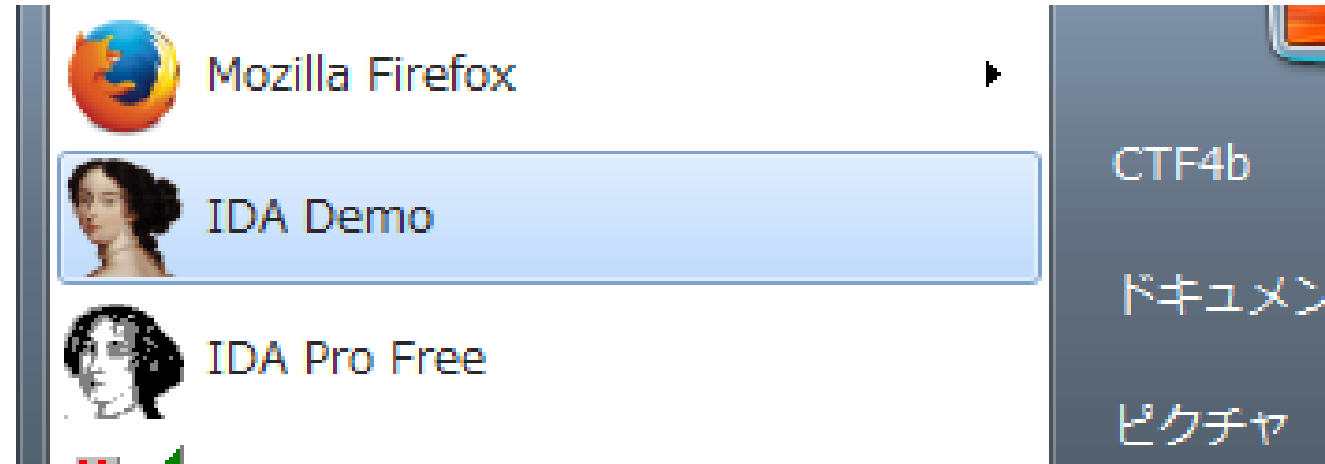
実行ファイルを解析してみよう

- IDA Pro とは？
 - 非常に高機能な逆アセンブラ
 - プログラムの実行の流れをフローグラフで表示
 - サブルーチン・API呼び出しの依存関係の可視化
 - 強力な検索・ジャンプ機能
 - 構造体，共用体，ビットフィールド等のサポート
 - 文字列，インポート，エクスポートの抽出
 - メモ，状態保存など解析途中での休止機能
 - デバッガの利用
 - スクリプティング（IDC, IDAPython）



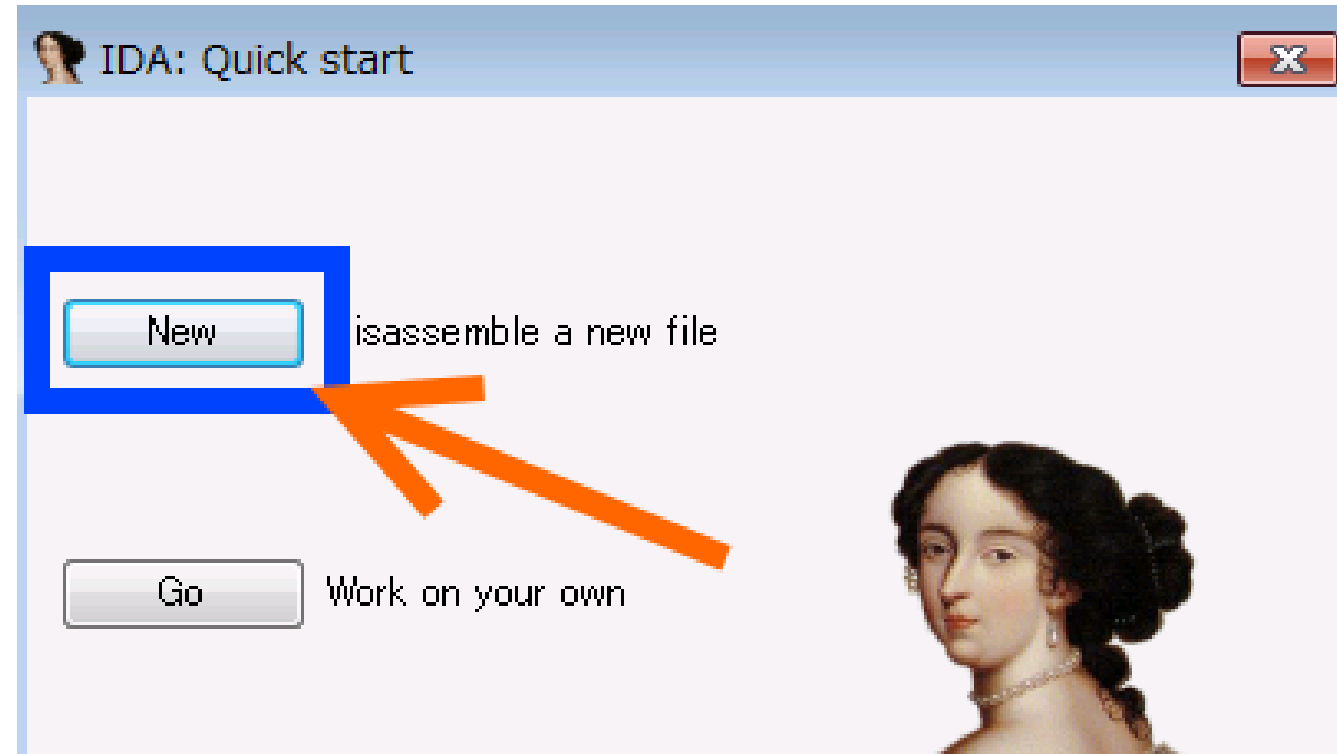
実行ファイルを解析してみよう

- 実際に起動してみよう
 - スタートメニューから “IDA Demo” を起動
- 今回はデモ版なので、30分で強制終了されます
- 起動時間の制約がない無料(free) 版もありますが、解析能力は劣ります。



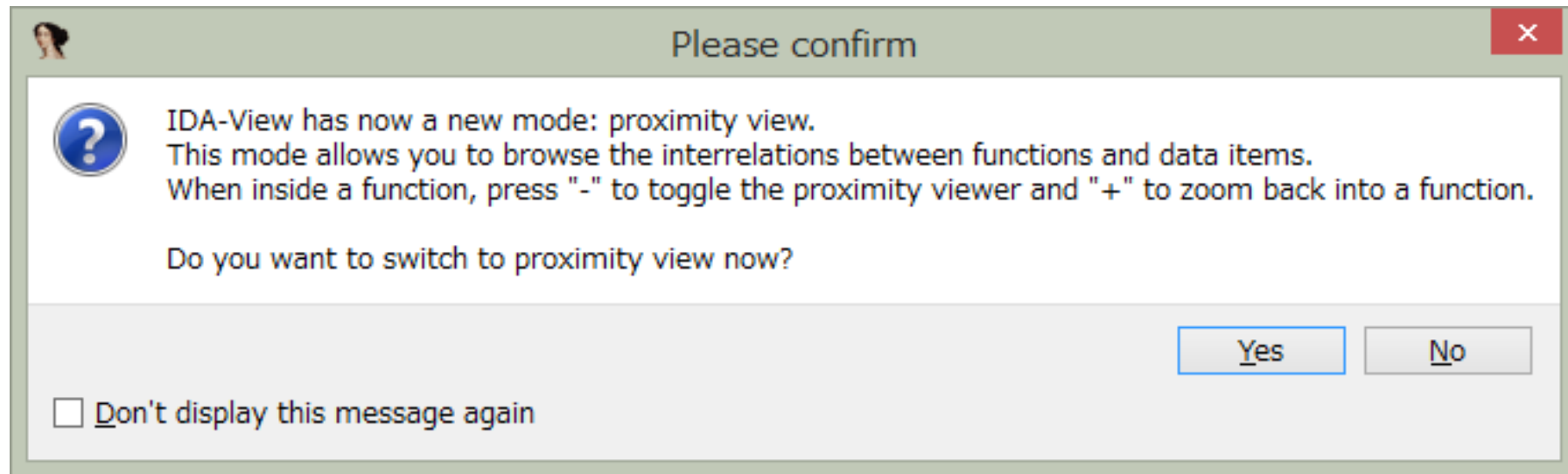
実行ファイルを解析してみよう

- 起動したら
出てきたダイアログにて“New”をクリックして、
解析対象のファイル（今回は 01.exe）を選択します



実行ファイルを解析してみよう

- 出てくるダイアログについて：

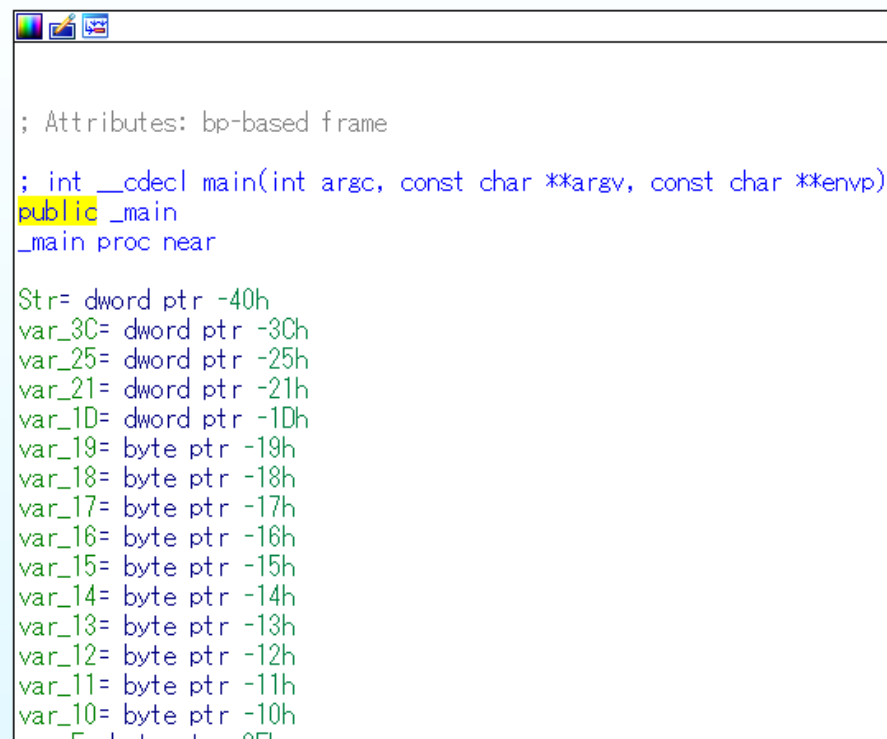


- このダイアログは“**No**”を選ぶ。



実行ファイルを解析してみよう

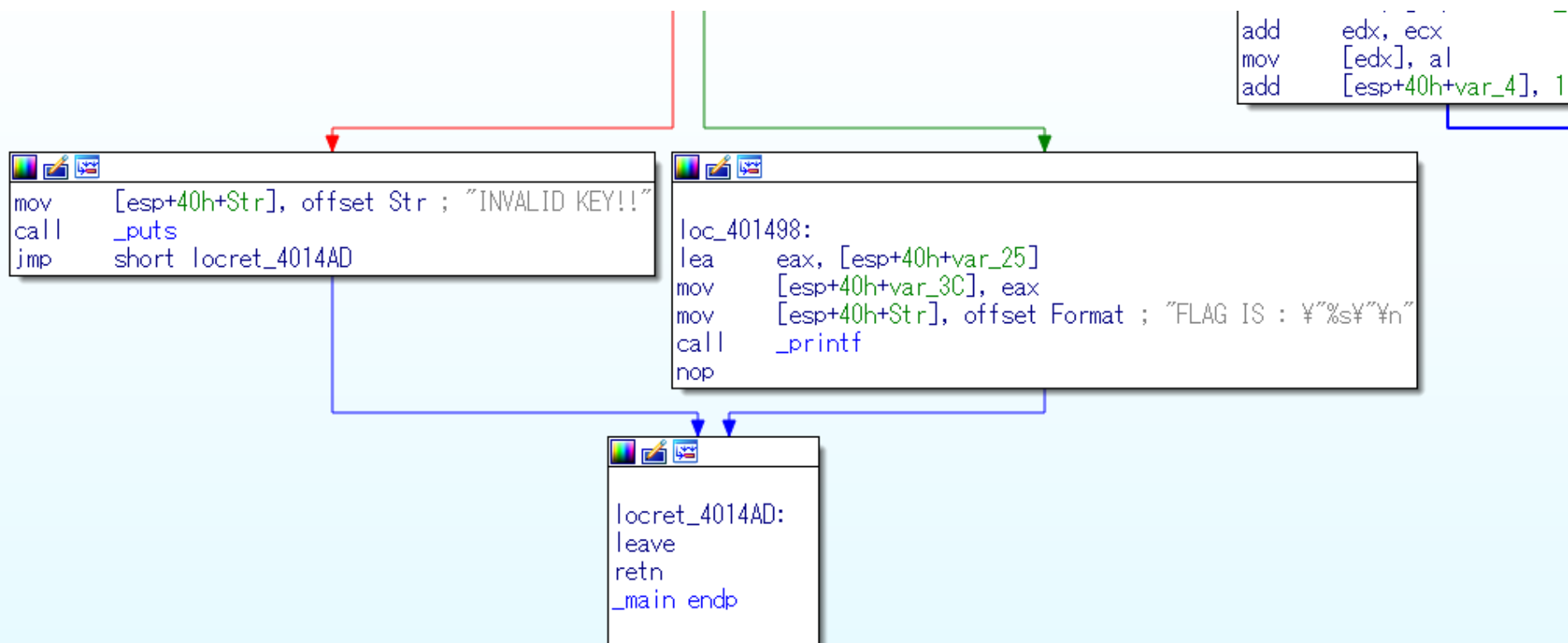
- 逆アセンブル結果が表示される
 - 表示が違っている場合は、スペースキーを何度か押します。

A screenshot of a debugger window showing assembly code. The window has a title bar with standard Windows icons. The code is displayed in a monospaced font with syntax highlighting. The code includes a comment about attributes, a function signature for _main, and a list of local variables with their types and offsets.

```
; Attributes: bp-based frame  
  
; int __cdecl main(int argc, const char **argv, const char **envp)  
public _main  
_main proc near  
  
Str= dword ptr -40h  
var_3C= dword ptr -3Ch  
var_25= dword ptr -25h  
var_21= dword ptr -21h  
var_1D= dword ptr -1Dh  
var_19= byte ptr -19h  
var_18= byte ptr -18h  
var_17= byte ptr -17h  
var_16= byte ptr -16h  
var_15= byte ptr -15h  
var_14= byte ptr -14h  
var_13= byte ptr -13h  
var_12= byte ptr -12h  
var_11= byte ptr -11h  
var_10= byte ptr -10h
```

実行ファイルを解析してみよう

- コードを実際に見ていきましょう。
 - コードの下の方にこんな部分がある。



実行ファイルを解析してみよう

jbe short loc_401454

INVALID KEY
という記述
がある。

```
mov     [esp+40h+var_5], 0
movzx   eax, [esp+40h+var_5]
test    eax, eax
jnz     short loc_401498
```

```
loc_401454:
mov     eax, [esp+40h+va
add     eax, eax
```

“FLAG IS :”
という記述
がある。

```
mov     [esp+40h+Str], offset Str ; "INVALID KEY!!"
call    _puts
jmp     short locret_4014AD
```

```
loc_401498:
lea     eax, [esp+40h+var_25]
mov     [esp+40h+var_30], eax
mov     [esp+40h+Str], offset Format ; "FLAG IS : ¥"%s¥"¥n"
call    _printf
nop
```



実行ファイルを解析してみよう

この部分を
どうにかすれば
フラグが出そう

```
mov     [esp+40h+var_5], 0
movzx   eax, [esp+40h+var_5]
test    eax, eax
jnz     short loc_401498
```

jbe short loc_401454

```
loc_401454:
mov     eax, [esp+40h+va
add     eax, eax
movzx   eax, [esp+eax+40i
xor     eax, 20h
lea     ecx, [esp+40h+va
mov     edx, [esp+40h+va
add     edx, ecx
mov     [edx], al
add     [esp+40h+var_4],
```

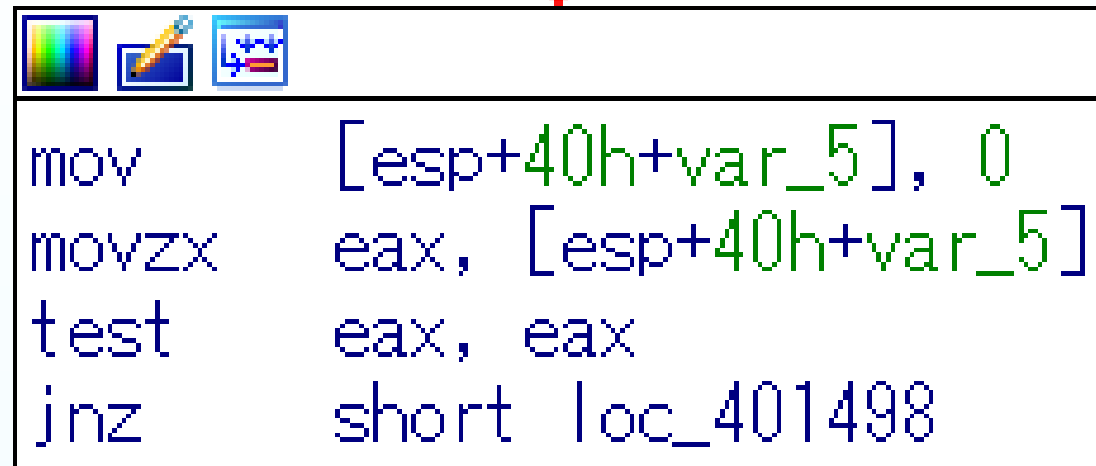
```
mov     [esp+40h+Str], offset Str ; "INVALID KEY!!"
call    _puts
jmp     short locret_4014AD
```

```
loc_401498:
lea     eax, [esp+40h+var_25]
mov     [esp+40h+var_3C], eax
mov     [esp+40h+Str], offset Format ; "FLAG IS : ¥"%s¥"¥n"
call    _printf
nop
```



実行ファイルを解析してみよう

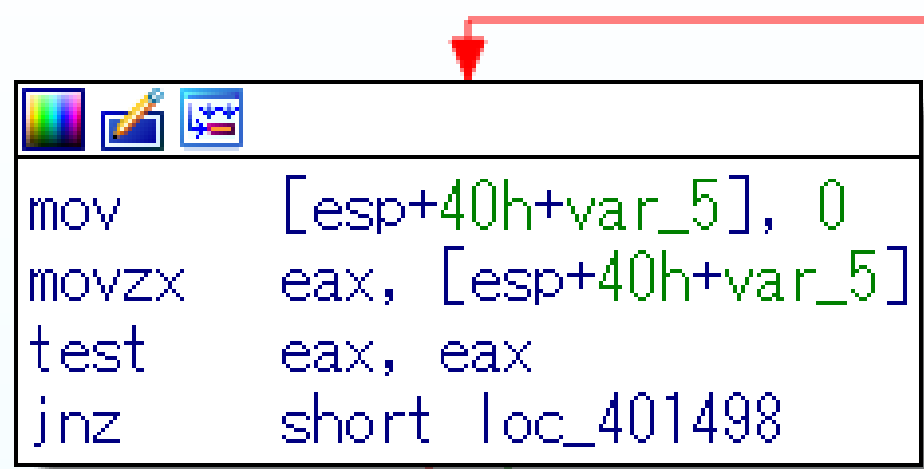
- INVALID KEY と FLAG IS.. の分岐部分
- 命令の実行結果は左側に入る (Intel 記法)
 - `mov [esp+40h+var_5], 0` なら
`[esp+40h+var_5]` に
0 を格納する動作。



```
mov      [esp+40h+var_5], 0
movzx    eax, [esp+40h+var_5]
test     eax, eax
jnz      short loc_401498
```

実行ファイルを解析してみよう

- **mov , movzx**
 - 右の物を左へコピー (MOVE)
- **test a,b**
 - a&b を演算し、結果は格納せずに「負数になるか」や「ゼロになるか」だけを求める。
- **jnz a (Jump Not Zero)**
 - 先行する演算の結果が 0 ではない場合指定した場所 (この場合は a) に飛ぶ。



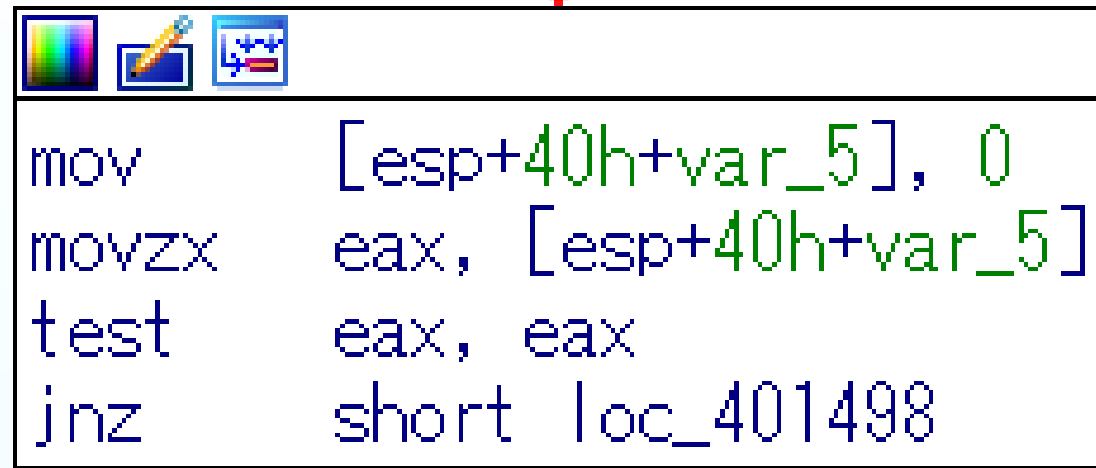
```
mov      [esp+40h+var_5], 0
movzx    eax, [esp+40h+var_5]
test     eax, eax
jnz      short loc_401498
```

A screenshot of assembly code from a debugger. A red arrow points from the top right to the 'jnz' instruction. The code is as follows:

実行ファイルを解析してみよう

- ということで処理はこんな感じに：

```
[esp+40h+var_5] = 0;  
eax = [esp+40h+var_5];  
if (eax&eax)  
{  
    “INVALID FLAG”  
}  
else {  
    “FLAG IS : **”  
}
```



```
mov    [esp+40h+var_5], 0  
movzx  eax, [esp+40h+var_5]  
test   eax, eax  
jnz    short loc_401498
```

- このままではどうやっても出ない。

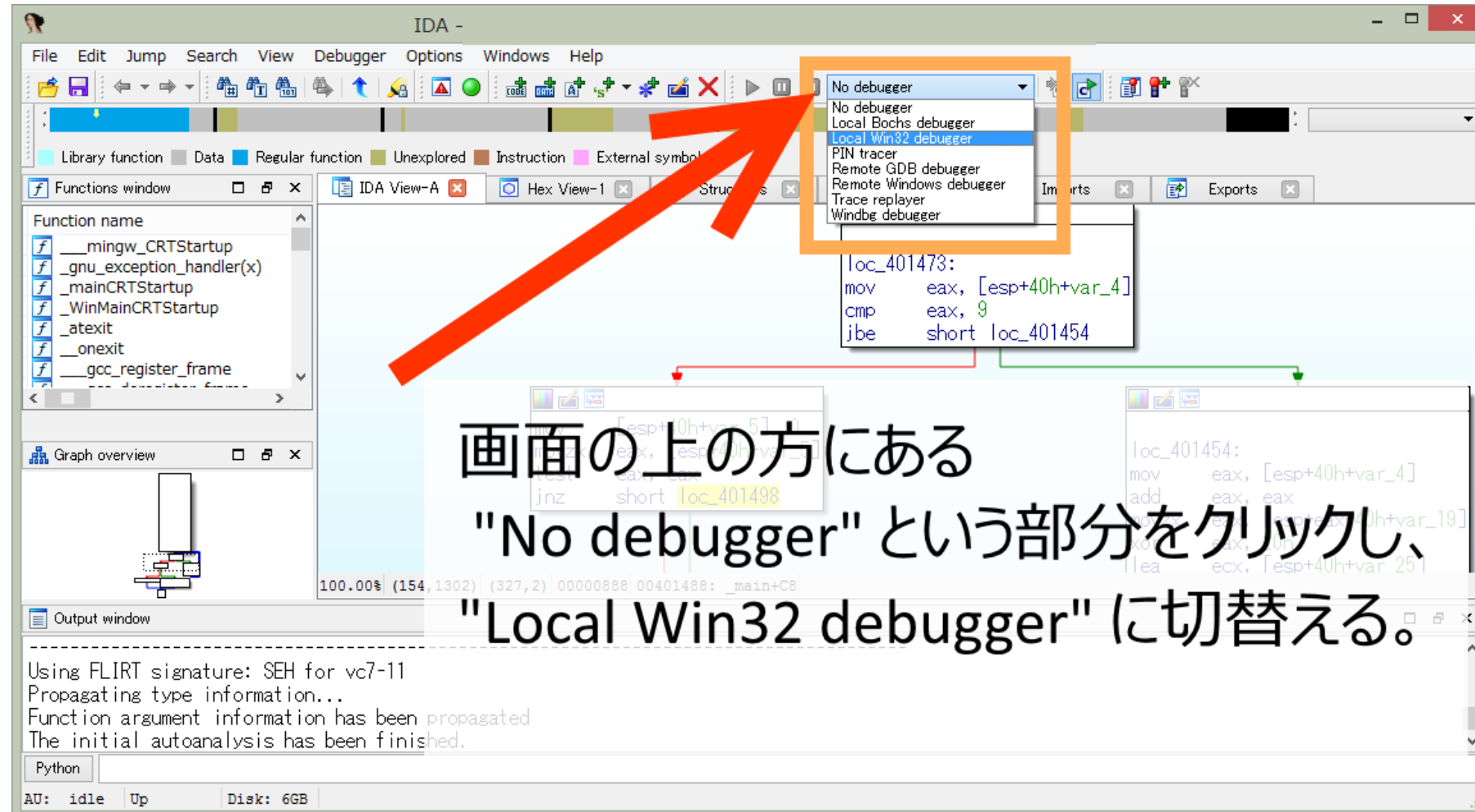
実行ファイルを解析してみよう

- プログラムの動作を変えてみよう
 - IDA Pro の機能を使って、プログラムの動作を変えてみよう



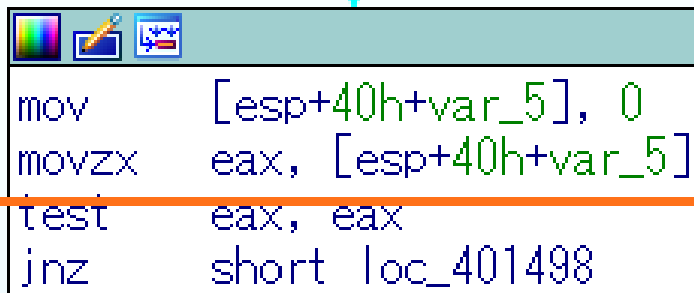
実行ファイルを解析してみよう

- まずはデバッガを設定します。



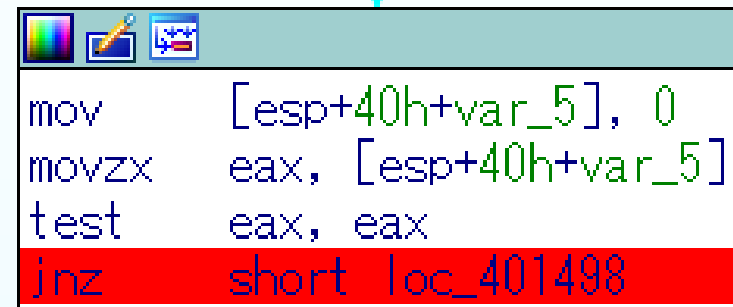
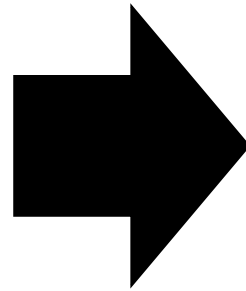
実行ファイルを解析してみよう

- 次に、「ブレークポイント」を設定します
 - これを設定すると、プログラムの動作がその箇所で停止するようになります。
- ブレークポイントの設定は“F2”キーで行えます。



```
mov     [esp+40h+var_5], 0
movzx   eax, [esp+40h+var_5]
test    eax, eax
jnz     short loc_401498
```

ここをクリックし、F2 キーを押す。



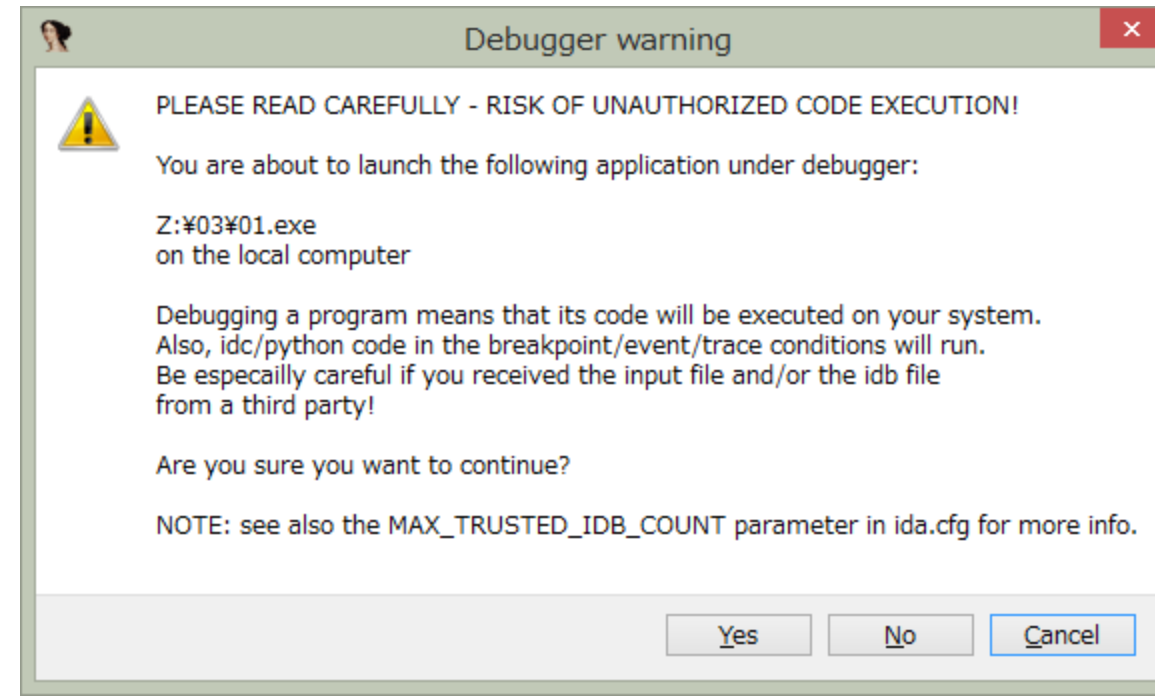
```
mov     [esp+40h+var_5], 0
movzx   eax, [esp+40h+var_5]
test    eax, eax
jnz     short loc_401498
```

設定されると、赤色になる



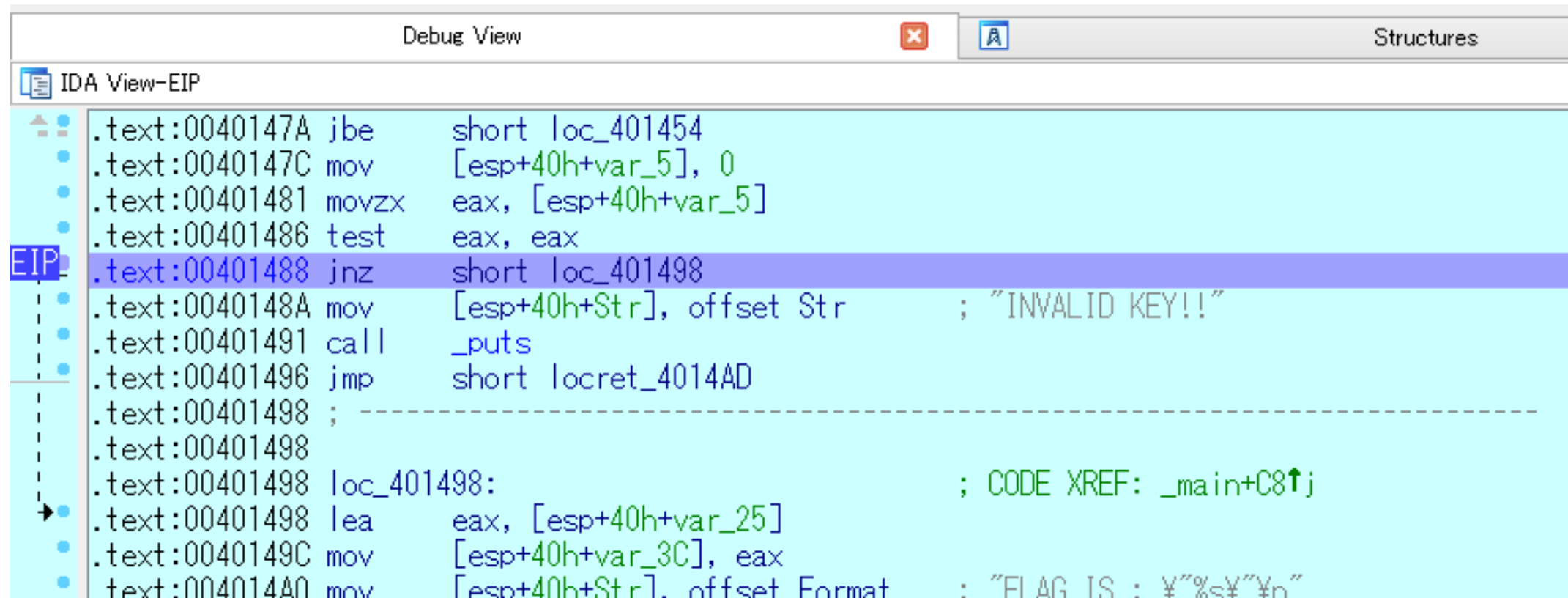
実行ファイルを解析してみよう

- 実行してみよう
 - F9 キーを押して実行開始
- 画面下のようなダイアログが出るが
“YES” を押す。
 - 実際にコードを実行するが
いいか、というような内容。



実行ファイルを解析してみよう

- ブレークポイントでプログラムが止まる
 - スペースキーを押して表示を切替えましょう。



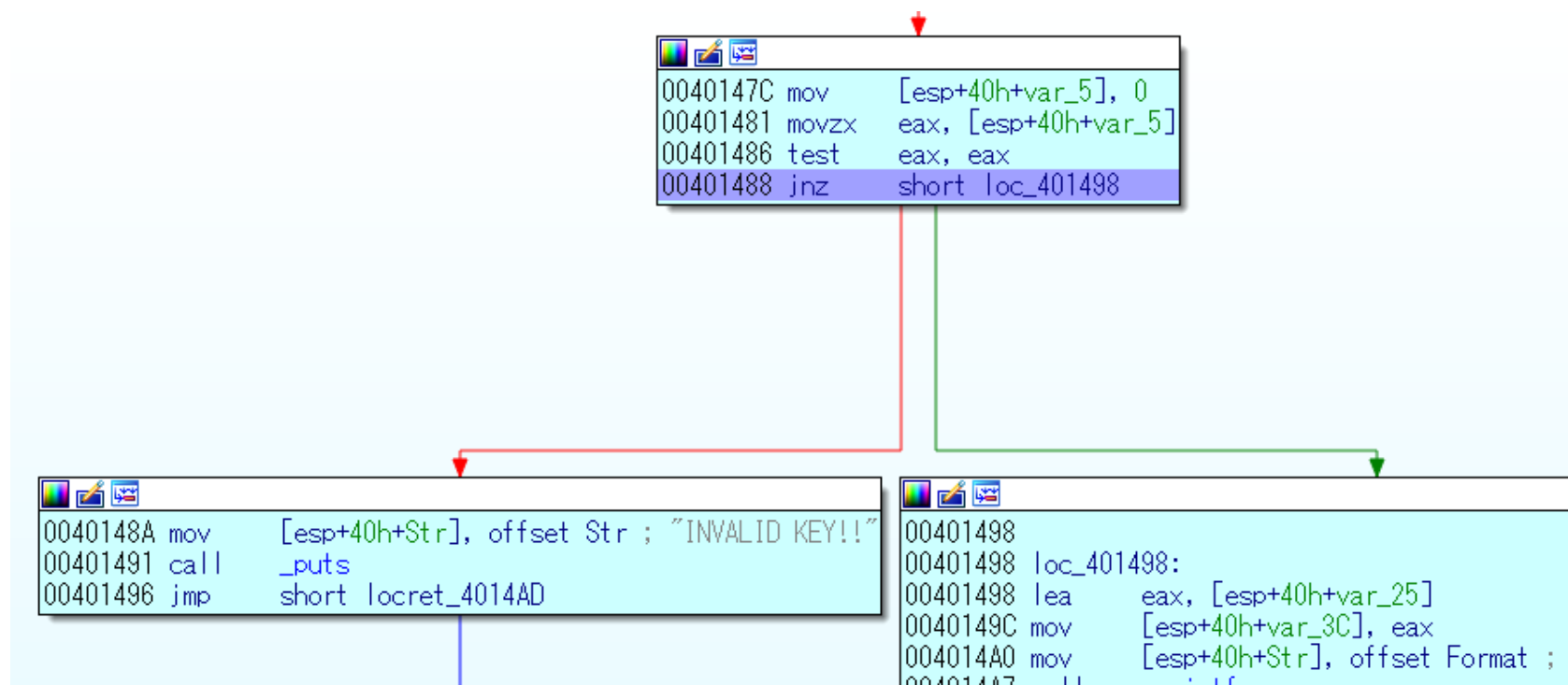
```
Debug View
Structures

IDA View-EIP

.text:0040147A jbe     short loc_401454
.text:0040147C mov     [esp+40h+var_5], 0
.text:00401481 movzx   eax, [esp+40h+var_5]
.text:00401486 test    eax, eax
EIP .text:00401488 jnz     short loc_401498
.text:0040148A mov     [esp+40h+Str], offset Str      ; "INVALID KEY!!"
.text:00401491 call    _puts
.text:00401496 jmp     short locret_4014AD
.text:00401498 ; -----
.text:00401498
.text:00401498 loc_401498:                ; CODE XREF: _main+C8↑j
.text:00401498 lea     eax, [esp+40h+var_25]
.text:0040149C mov     [esp+40h+var_3C], eax
.text:004014A0 mov     [esp+40h+Str], offset Format      ; "FLAG IS : ¥\"%s¥\"¥n"
```

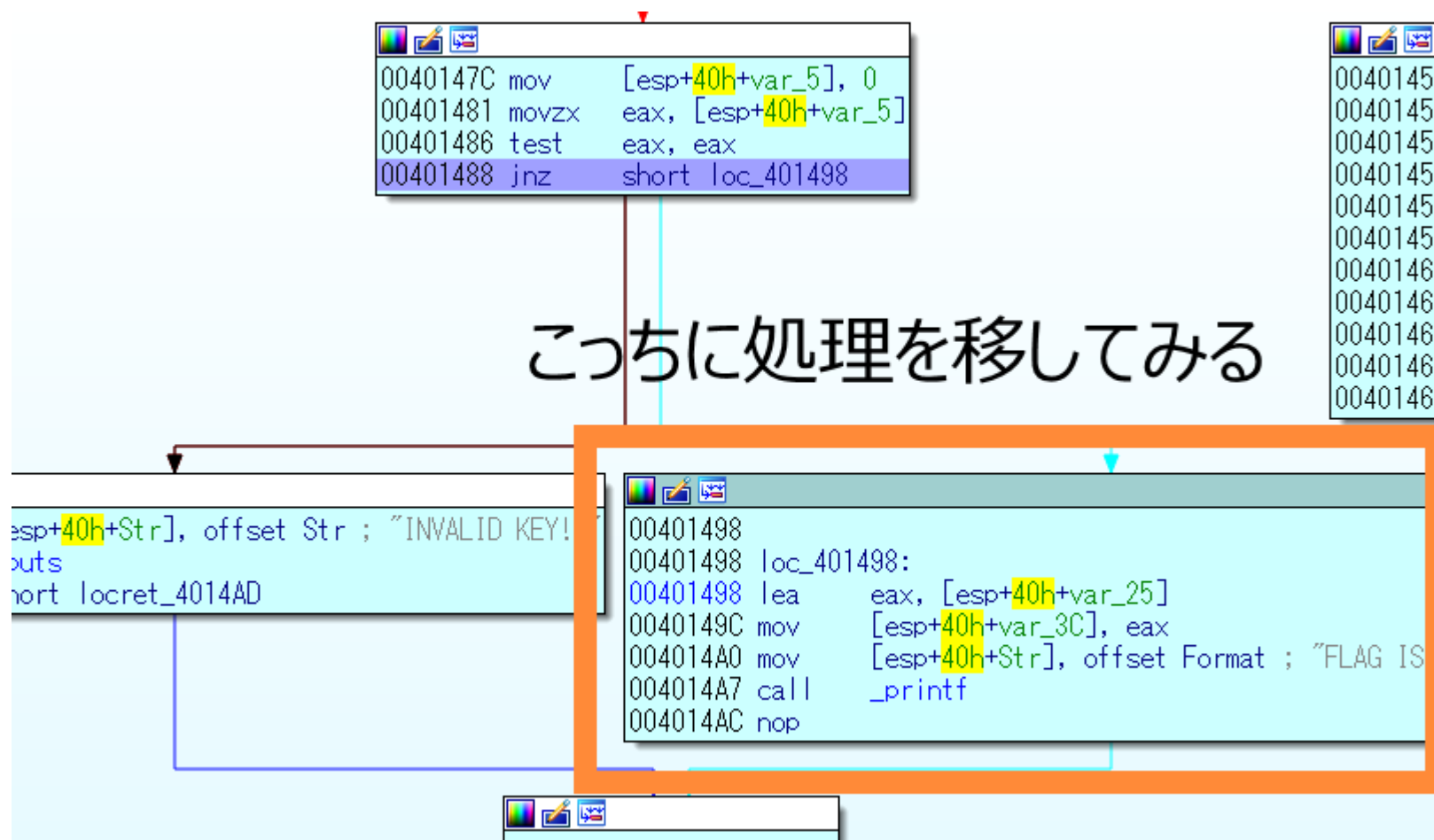
実行ファイルを解析してみよう

- グラフの画面に切り替わる。
また赤色の線が点滅している。
- 点滅している線は、次はこっちに進む、という意味を示す



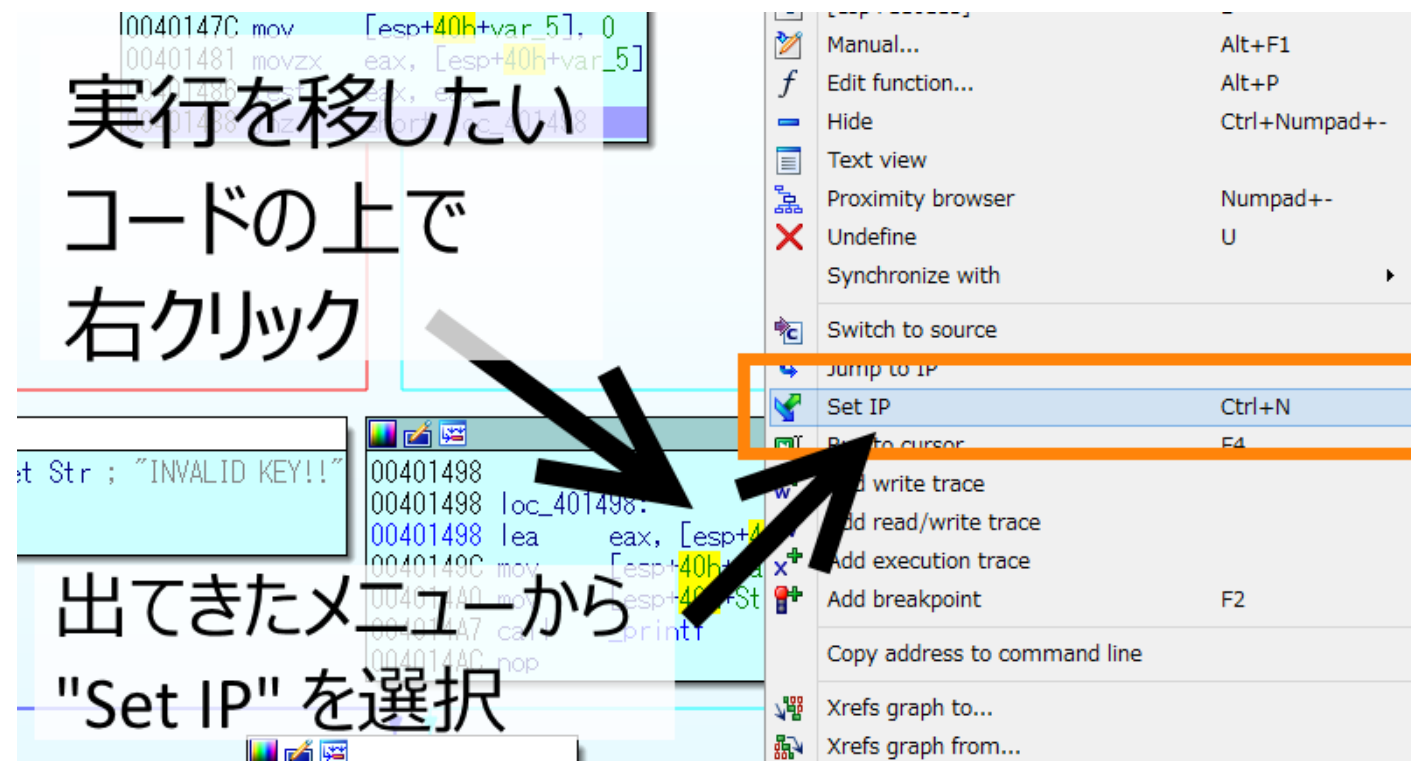
実行ファイルを解析してみよう

- それでは、実行先を変えてみましょう。



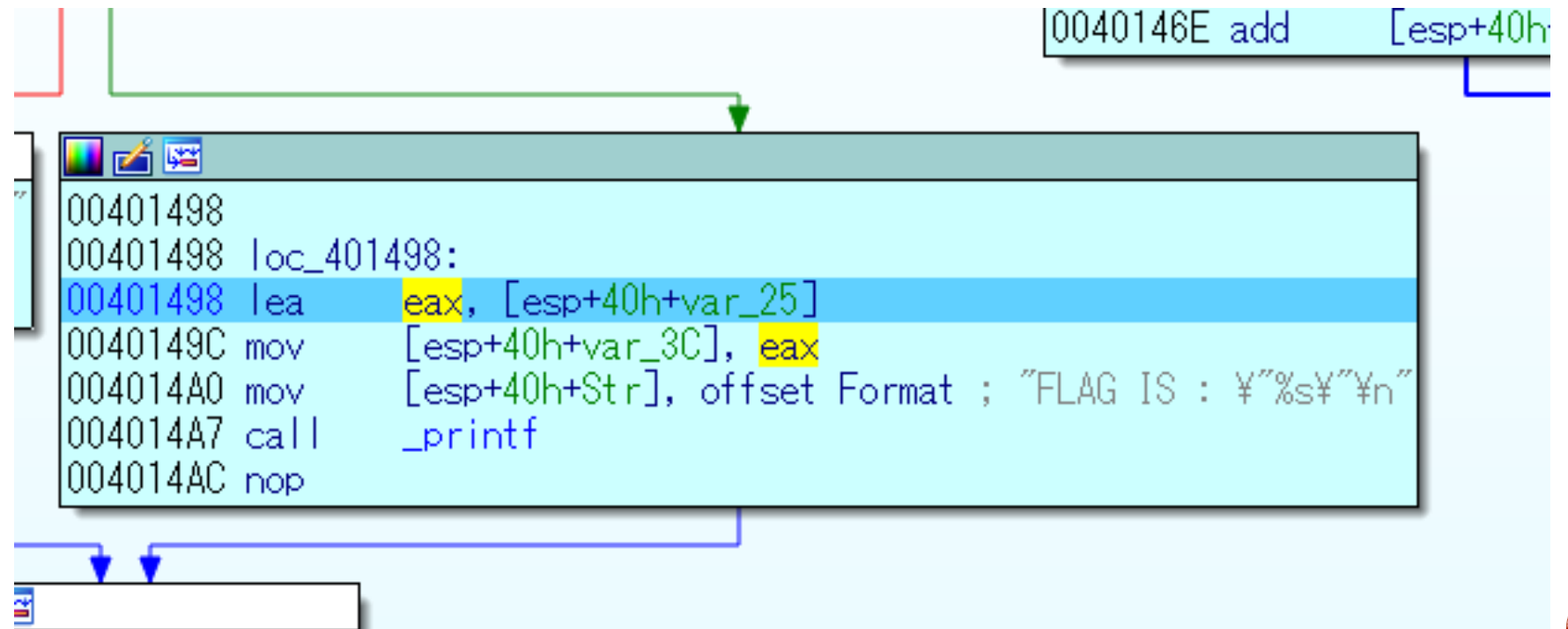
実行ファイルを解析してみよう

- 実行先を変えてみよう。
 - 実行先として指定したいコードの上で右クリックし、“Set IP”を選ぶ。



実行ファイルを解析してみよう

- コードの背景が青くなり、
処理がそちらに移動した事がわかる。

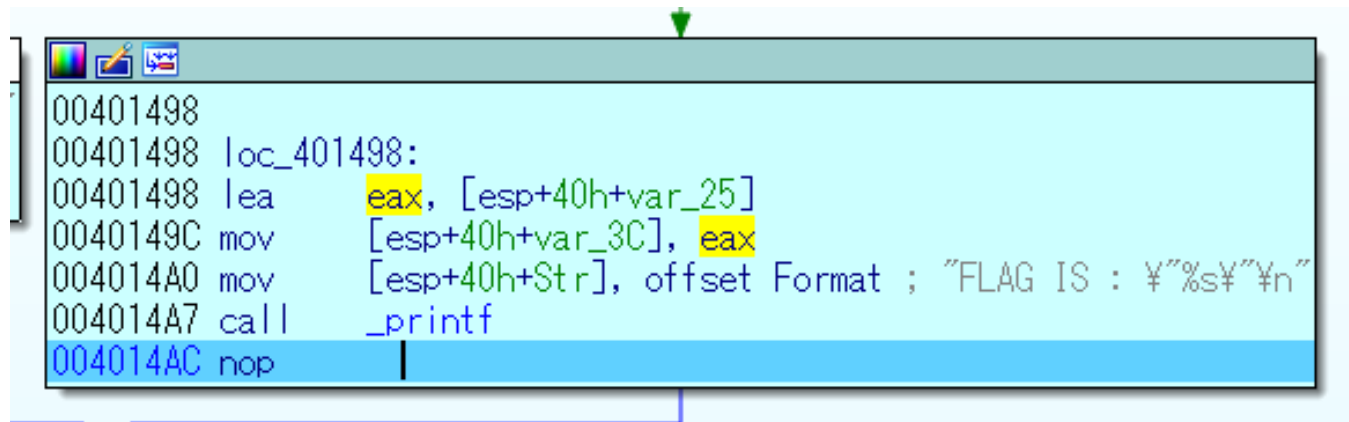


```
0040146E add [esp+40h]
00401498
00401498 loc_401498:
00401498 lea eax, [esp+40h+var_25]
0040149C mov [esp+40h+var_3C], eax
004014A0 mov [esp+40h+Str], offset Format ; "FLAG IS : ¥"%s¥"¥n"
004014A7 call _printf
004014AC nop
```



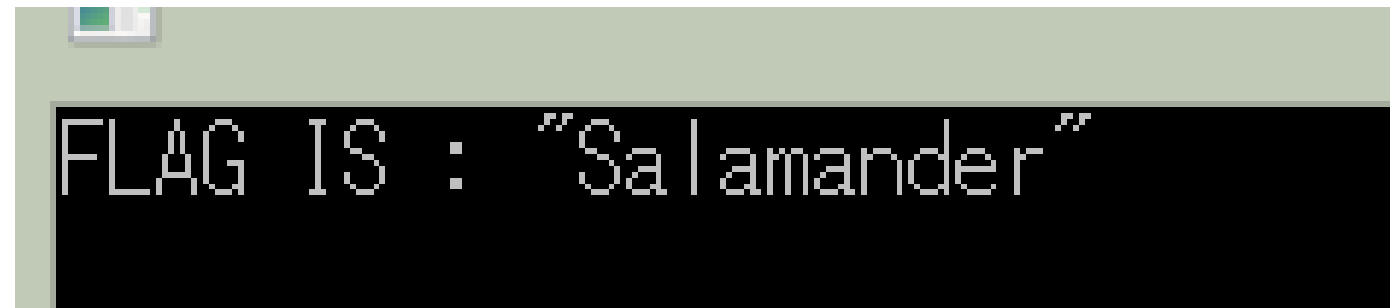
実行ファイルを解析してみよう

- その状態で F8 キーを押して
その部分の終わりまで処理を進めると、



```
00401498  
00401498 loc_401498:  
00401498 lea     eax, [esp+40h+var_25]  
0040149C mov     [esp+40h+var_3C], eax  
004014A0 mov     [esp+40h+Str], offset Format ; "FLAG IS : ¥"%s¥"¥n"  
004014A7 call    _printf  
004014AC nop     |
```

- フラグが表示された！

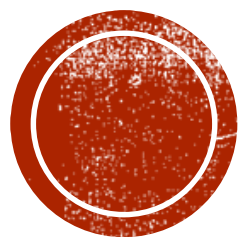


```
FLAG IS : "Salamander"
```

実行ファイルを解析してみよう

- この章のまとめ
- 実行ファイルを IDA Pro で解析できる
 - デバッグを開始するには “Local Win32 Debugger” を選択する
 - F2 キーでブレークポイントの設定
 - F9 キーで実行開始
 - “Set IP” で実行先を変更できる
 - F8 キーで一つずつ進めることが出来る





アセンブラを読んで プログラムの仕組み を見てみよう

まずは加減乗除

プログラムの仕組みを見てみよう

- まずは加算から

<code>int</code>	<code>main()</code>	<code>{</code>	<code>mov</code>	<code>[ebp+i], 0</code>
		<code>int i = 0;</code>	<code>mov</code>	<code>eax, [ebp+i]</code>
		<code>i = i + 20;</code>	<code>add</code>	<code>eax, 14h</code>
		<code>}</code>	<code>mov</code>	<code>[ebp+i], eax</code>

1. “0” を入れる。
2. “add” 命令を使って加算する。



プログラムの仕組みを見てみよう

- （記法によって異なりますが・・・）
今回の資料で用いられている“Intel”記法では

結果が 左 に入る と覚えておきましょう。

- [命令] [出力],[入力]
- `add eax,10h` → `eax = eax + 0x10;`



プログラムの仕組みを見てみよう

■次に減算

```
int main() {  
    int i = 0;  
    i = i - 20;  
}
```

```
mov    [ebp+var_4], 0  
mov    eax, [ebp+var_4]  
sub    eax, 14h  
mov    [ebp+var_4], eax
```

1. “0” を入れる。
2. “sub” 命令を使って減算する。



プログラムの仕組みを見てみよう

- 同様に・・・加減乗除には代表的な物として次のような命令があります
 - add : 加算
 - sub : 減算
 - imul : 乗算
 - idiv : 除算



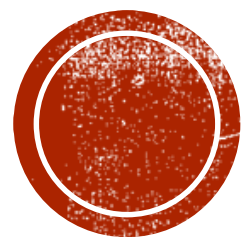
プログラムの仕組みを見てみよう

- ではこれはどうなるか、分かりますか

```
int main() {  
    int i = 0;  
    i = i * 20;  
}
```

```
mov    [ebp+var_4], 0  
mov    eax, [ebp+var_4]  
imul   eax, 14h  
mov    [ebp+var_4], eax
```





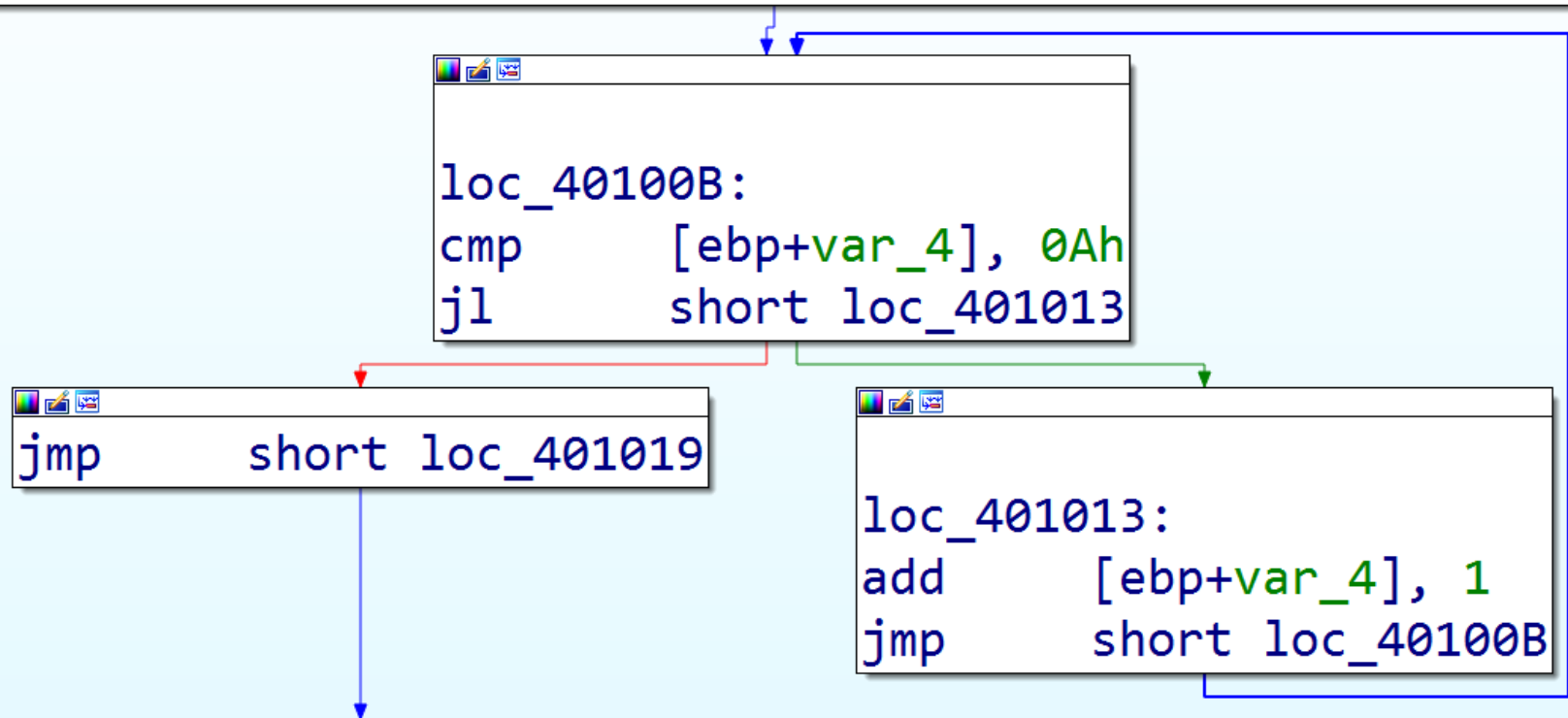
アセンブラを読んで
プログラムの仕組み
を見てみよう

次はループ

プログラムの仕組みを見てみよう

■ For ループ

```
mov     [ebp+var_4], 0
```



プログラムの仕組みを見てみよう

for (int i = 0; i < 10; i++) { }

mov [ebp+var_4], 0

loc_40100B:
cmp [ebp+var_4], 0Ah
jl short loc_401013

jmp short loc_401019

loc_401013:
add [ebp+var_4], 1
jmp short loc_40100B



プログラムの仕組みを見てみよう

- 値の比較命令には次のようなものがあります（一例）
 - JL : Jump if less (～より小さかったら・・・)
 - JG : Jump if greater (～より大きかったら・・・)
 - JE : Jump if equal (～と同じだったら・・・)
 - JGE : Jump if greater or equal (～以上なら)



プログラムの仕組みを見てみよう

- 値の比較は “演算命令＋ジャンプ命令” の組で記述します。
- ジャンプ命令：実行場所を変更する命令
- `cmp eax, 10h` # `eax` と `0x10` を比較
- `j1 loc_401000` # `eax < 0x10` なら `loc_401000` に飛ぶ



プログラムの仕組みを見てみよう

- では、これはどうなるか分かりますか

```
mov     [ebp+var_4], 14h
```

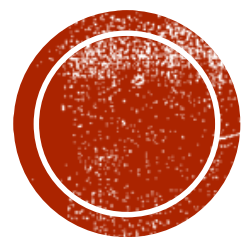
```
loc_40100B:  
cmp     [ebp+var_4], 0  
jg      short loc_401013
```

```
jmp     short loc_401019
```

```
for(int i = 20; i > 0; i--)  
{ }
```

```
loc_401013:  
sub     [ebp+var_4], 1  
jmp     short loc_40100B
```





アセンブラを読んで
プログラムの仕組み
を見てみよう

命令呼び出し

プログラムの仕組みを見てみよう

```
printf("HELLO WORLD");
```

```
push    offset Format    ; "HELLO WORLD"  
call    ds:__imp__printf
```



プログラムの仕組みを見てみよう

```
printf("HELLO WORLD");
```

```
push    offset Format    ; "HELLO WORLD"
```

```
call    ds:__imp__printf
```



プログラムの仕組みを見てみよう

- 命令を呼ぶときは
 - 引数を “push” して、
 - 命令を “call” します。
- Ex.) func(0x10)
 - push 10h
 - call func



プログラムの仕組みを見てみよう

- Push をする順番は「後ろから」

```
printf("%d", 1024);
```

```
push    400h  
push    offset Format    ; "%d"  
call    ds:printf
```



プログラムの仕組みを見てみよう

- なぜ後ろから？

- 引数の情報がスタック（Stack：積み重ね）領域に記録されるため。
- 引き出しから物を出すとき「先入れ後出し」なのを考える。
- スタック領域は一時的なデータの保存場所として使われる。



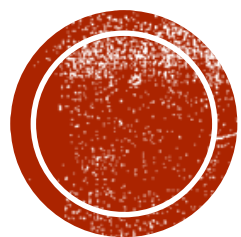
プログラムの仕組みを見てみよう

- では、これはどうなりますか

```
push    offset Str      ; "HELLO"  
call    ds:puts
```

```
puts("HELLO");
```





アセンブラを読んで
プログラムの仕組み
を見てみよう

関数呼び出し

プログラムの仕組みを見てみよう

```
void hoge(int i)
{ printf("%d", i); }

int main()
{ hoge(10); }
```

```
mov     eax, [ebp+arg_0]
push    eax
push    offset Format      ; "%d"
call    ds:printf
```

```
push    0Ah
call    hoge
```



● まとめ



まとめ

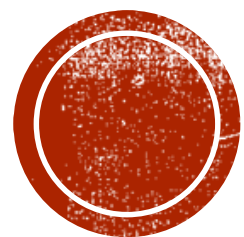
- バイナリの問題が出たら
まず “file” コマンドを実行して種類を判定しよう
- “strings” コマンドを実行して
バイナリの中に含まれる文字を出してみよう
- 実行ファイルであればデバッガで解析してみよう
 - 今回は IDA Pro を使いました。
- コンピュータは簡単な命令の組み合わせで動いている



まとめ

- たのしいバイナリの歩き方：
<http://www.amazon.co.jp/dp/B00EODUZFO/>
- アセンブリ言語の教科書：
<http://www.amazon.co.jp/dp/4887188293/>
- 解析魔法少女美咲ちゃん（中古のみ）：
<http://www.amazon.co.jp/dp/4798008532/>
- IDA Pro book：
<http://www.amazon.co.jp/dp/B005EI84TM/>
- リバースエンジニアリング：
<http://www.amazon.co.jp/dp/4873114489/>
- IPA セキュアプログラミング講座：
<https://www.ipa.go.jp/security/awareness/vendor/programmingv2>





(追加資料) レジスタの構成



レジスタ

- 一時的にデータを保存する場所
 - 自由に使えるレジスタ
 - `eax, ebx, ecx, edx` 等
 - 特別な役割を持つレジスタ
 - `esp, ebp, EFLAGS` 等
- 関数の返り値は基本的に`eax`に入る



EFLAGS レジスタ

- 計算を行う命令が自動で更新する
- 最後の計算結果の「状態」を記憶する
 - 計算結果が0か (Zero Flag)
 - 計算結果が負か (Signed Flag)
 - Etc..
- これらの情報は、主に分岐命令で使用される



レジスタの構成

- 同じレジスタが複数の名前を持つ
 - 名前によってサイズが違ふ
 - 名前の変化には規則がある
 - 例: `rax`, `eax`, `ax`, `ah`, `al`
- 扱う値の大きさに応じて名前を変える
 - 例えば、`eax`の下位8bitが欲しい時には`al`を読み出す



レジスタの構成

64bit

32bit

16bit

8bit

								rax (r*x)
								eax (e*x)
								ax (*x)
								al (*l)
								ah (*h)

- 各マスは
8bit(1byte)
- r*xは64bit環境用



スタック

- 一時的にデータを保存する場所
 - PUSH / POP命令でデータを出し入れする
 - PUSHで入れてPOPで出す
- 関数呼び出し時にはPUSH命令で引数を渡す

```
load_data("a.txt", 10); → PUSH 10  
PUSH "a.txt"  
CALL load_data
```



算術・論理命令

- 加減乗除, ビット演算 等を行う命令
- 結果は左側に格納される。
- 例：
 - 加算： `eax` に 1 を足し、結果を `eax` に格納する
`ADD eax, 1`
 - 減算： `ebx` から `ecx` 分を引き、結果を `ebx` に格納
`SUB ebx, ecx`



分岐命令

- 条件分岐命令と無条件分岐命令がある
- 条件分岐命令：
 - 先行の演算結果を基にジャンプするかどうかを判定する
 - プログラムで言うところの“if” 命令
- 無条件分岐命令：
 - 演算結果にかかわらずジャンプを行う。
 - プログラムで言うところの“goto” 命令



分岐命令の例

- 無条件分岐：
 - JMP（無条件に指定したアドレスに飛ぶ）
 - CALL（JMP と同じく飛ぶが、RET で戻れる）
- 条件分岐：
 - JZ (Jump Zero, 先行する演算結果が 0 なら飛ぶ)
 - JNZ (Jump Not Zero, 0 でなければ飛ぶ)



比較命令

- 数値を比較する命令
 - 例) `TEST eax, ebx`
 - 内部的には $(ebx - eax)$ をしたときに負数になるか、0 になるかを演算して `EFLAGS` レジスタに格納する
- `JZ` や `JNZ` の為の条件判定として用いられることが多い

