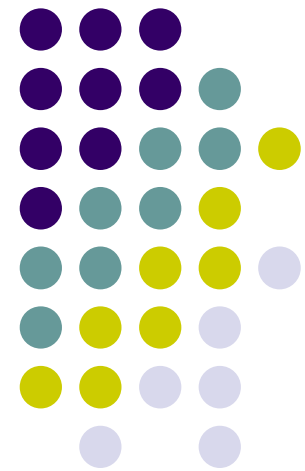


(自己流)  
リバーチャレンジ!

by ゆまの

yumano3 (at) gmail <dot> com



# 使ったもの

- Oll4Dbg
- 紙と鉛筆
- strings.exe
- 勘





## 大きな流れ(補足)

- 使っているAPIを調べる
- 意味ありげな文字列を探す
- 上の情報をヒントに意味あるロジックを推測
- 分岐の条件をひとつずつ分析



# Step 1 ファイルを開く

- OllyDbgを起動する
- [File>Open]を開き、ファイルを指定

The screenshot shows the OllyDbg interface with the following details:

- Assembly View:** Shows assembly instructions for the `level1.bin` module. The instruction at address `0040111D` is `CALL level1.00401554`. A comment indicates `EAX ESP EBP にスタックのポインタ`. Other instructions include `MOV EAX, EDI`, `CALL level1.00402C00`, and `MOV DWORD PTR DS:[ESI+4], ECX`.
- Registers (FPU):** Shows the state of CPU registers. `EAX` is `00000000`, `ECX` is `0012FFB0`, and `EDX` is `7C94E4F4`. The `EIP` register is `0040110C`.
- Hex Dump:** Shows a memory dump starting at address `00403800`. The first few bytes are `00 00 00 A9 13 40 00 00 00 00 00 00 00 00 00 00`.
- Call Stack:** Shows the current call stack with `0012FFC7` at the top, pointing to `kernel32.7C81707`.



## Step 2 APIを見てみる

- APIリストを取得 (CTRL+N)
- テキスト入力のAPIはGetCommandLineAくらい  
ほかのテキスト系APIにはGetWindowTextとかDlgItemTextとかあります

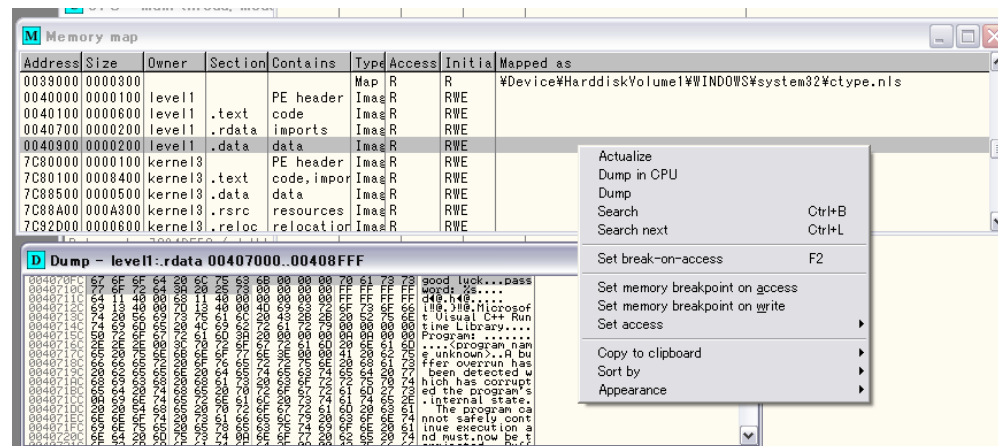
Address	Section	Type	Name	Comment
0040703C	.rdata	Import	KERNEL32.FlushFileBuffers	
00407040	.rdata	Import	KERNEL32.FreeEnvironmentStringsA	
00407048	.rdata	Import	KERNEL32.FreeEnvironmentStringsW	
00407088	.rdata	Import	KERNEL32.GetACP	
00407088	.rdata	Import	KERNEL32.GetCommandLineA	
00407090	.rdata	Import	KERNEL32.GetCPIInfo	
00407030	.rdata	Import	KERNEL32.GetCurrentProcess	
0040701C	.rdata	Import	KERNEL32.GetCurrentProcessId	
00407018	.rdata	Import	KERNEL32.GetCurrentThreadId	
00407044	.rdata	Import	KERNEL32.GetEnvironmentStrings	
00407054	.rdata	Import	KERNEL32.GetEnvironmentStringsW	
0040705C	.rdata	Import	KERNEL32.GetFileType	
00407050	.rdata	Import	KERNEL32.GetLastError	
00407074	.rdata	Import	KERNEL32.GetLocaleInfoA	

- enterを押すとAPIを使っている場所がわかる  
のでブレイクポイントを貼っとく (F2)



## Step 3 文字列を試みる

- 思い出したように文字列の抽出 (cygwin使ってます)
  - `cat level1.bin | strings.exe -radix=d > lv1.stv`
- そんなに出ました
  - 28924 good luck
  - 28936 password: %s
- alt+Mで"メモリマップを開いて該当場所検索





## Step 4 good luckを追跡

- メモリマップで該当のセクション(vdataセクション)を選んで"Dump in CPU"で表示
- Dumpウィンドウで"good luck"の文字列を検索  
(CTRL+B) (アドレス4070FCで見)
- 先頭の文字列" g" を指定して、Find Referenceで参照しているところにJump
  - 00401126 |. BE FC704000 MOV ESI, level1, 004070FC
  - 0040112B |. B9 0A000000 MOV ECX, 0A ← good luckをESIに代入
  - 00401130 |. 33D2 XOR EDX, EDX
  - 00401132 |. F3:A6 REPE CMPS BYTE PTR ES:[EDI], BYTE PTR DS:[ESI]
- 00401132で"good luck"と何かを比較!



## Step 4 good luckを追跡

- 比較しているところにブレイクポイントを張って実行してみるけど、通過しない・・・orz
- F8を押しながら、1行ずつ実行していくと比較の直前で分岐で終了していることが判明
- [ESP+4]が3だと通るらしい。

- 00401110 /\$ 837C24 04 03 CMP DWORD PTR SS:[ESP+4], 3
- 00401115 |. 74 06 JE SHORT level1.0040111D
- 00401117 |. B8 01000000 MOV EAX, 1
- 0040111C |. C3 RETN





## Step 4 good luckを追跡

- とりあえず、分岐処理をNOPで埋めて(\*)、実行してみると第1引数とgood luckを比較していることが判明。

```
CPU - main thread, module level1
00401115  90 NOP
00401116  90 NOP
00401117  90 NOP
00401118  90 NOP
00401119  90 NOP
0040111A  90 NOP
0040111B  90 NOP
0040111C  90 NOP
0040111D  > 8B4424 08 MOV EAX,DWORD PTR SS:[ESP+8]
00401121  . 56 PUSH ESI
00401122  . 57 PUSH EDI
00401123  . 8B78 04 MOV EDI,DWORD PTR DS:[EAX+4]
00401126  . BE FC704000 MOV ESI,level1.004070FC ASCII "good luck"
0040112B  . B9 0A000000 MOV ECX,0A
00401130  . 33D2 XOR EDX,EDX
00401132  . F3:A6 REPE CMPS BYTE PTR ES:[EDI],BYTE PTR DS:[ESI] good luckとくらべ
00401134  . 5F POP EDI
00401135  . 5E POP ESI
00401136  . 75 0C JNZ SHORT level1.00401144
00401138  . 8B40 08 MOV EAX,DWORD PTR DS:[EAX+8]
0040113B  . 50 PUSH EAX
0040113C  . E8 4FFFFFFF CALL level1.00401090
00401141  . 83C4 04 ADD ESP,4
00401144  > 33C0 XOR EAX,EAX
00401145  . 90 NOP

Registers (FPU)
EAX 00380C68
ECX 0000000A
EDX 00000000
EBX 7FFD0000
ESP 0012FEE0
EBP 0012FFC0
ESI 004070FC ASCII "good luck"
EDI 00380C95 ASCII "hosehose"
EIP 00401132 level1.00401132
C 0 ES 0023 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty -UNORM D0A8 01050104 00000000
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0

Address Hex dump ASCII
00409000 00 00 00 00 A3 13 40 00 00 00 00 00 00 00 00 00 .....jll@.....
0040901F  F5 1F 40 00 7C 38 40 00 8E 3C 40 00 00 00 00 00  @.|8@.n<@...
00409020  00 00 00 00 9B 20 40 00 00 00 00 00 00 00 00 00  . @.....
00409030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .....
00409040  70 44 9C AC 1B 22 40 00 01 00 00 00 9C 73 0D 00  0樓-"@...楼
```

\*NOPで埋めると処理がスキップされます



## Step 5 PTR SS:[ESP+4] = 3

- [ESP+4]とは？
- [ESP]はスタックポインタ。それに+4してるので、一つ前にPUSHした値 = 関数呼び出し直前にPUSHした [4096FC] のデータです。
  - 00401344 . FF35 FC96400>PUSH DWORD PTR DS:[4096FC]
- DS:[4096FC] にアクセスする処理にブレークポイントを張ってみよう
- Dumpデータの [4096FC] の位置を右クリック、[Breakpoint> Hardware on access>Byte] とやると、メモリにアクセスされた瞬間に処理がストップ。
- 実行すると 402832 でストップした。



## Step 5 PTR SS: [ESP+4] = 3

- 0040279E~0040283Fを眺める
- GetFileNameとか使っているのので、引数を取  
得しているっぽい
- おもむろに引数を二つにしてみる
- DS: [4096FC]が3になった！
- 第2引数を見つけないとダメポ

ここはアドレスが変更される処理を見つける方法を書きたかったのです  
IDA Proを使うとStep5の分析はいりませんでした(おまけ参照)



## Step 6 password: %s

- 004010FBにpassword: %sがあるので着目。
- サブルーチン00401090~00401FBを無事通過できればよい
- 眺めるとpassword: %sの下にジャンプされる分岐がある。飛び元アドレスは004010B4, 004010CB, 004010D8

004010F5	. 83F8 0C	CMP EAX,0C	
004010F8	. ^7C F7	JL SHORT level1.004010F1	
004010FA	. 56	PUSH ESI	
004010FB	. 68 08714000	PUSH level1.00407108	ASCII "password: %s"
00401100	. E8 81000000	CALL level1.00401186	
00401105	. 83C4 08	ADD ESP,8	
00401108	> 5E	POP ESI	
00401109	. C3	RETN	

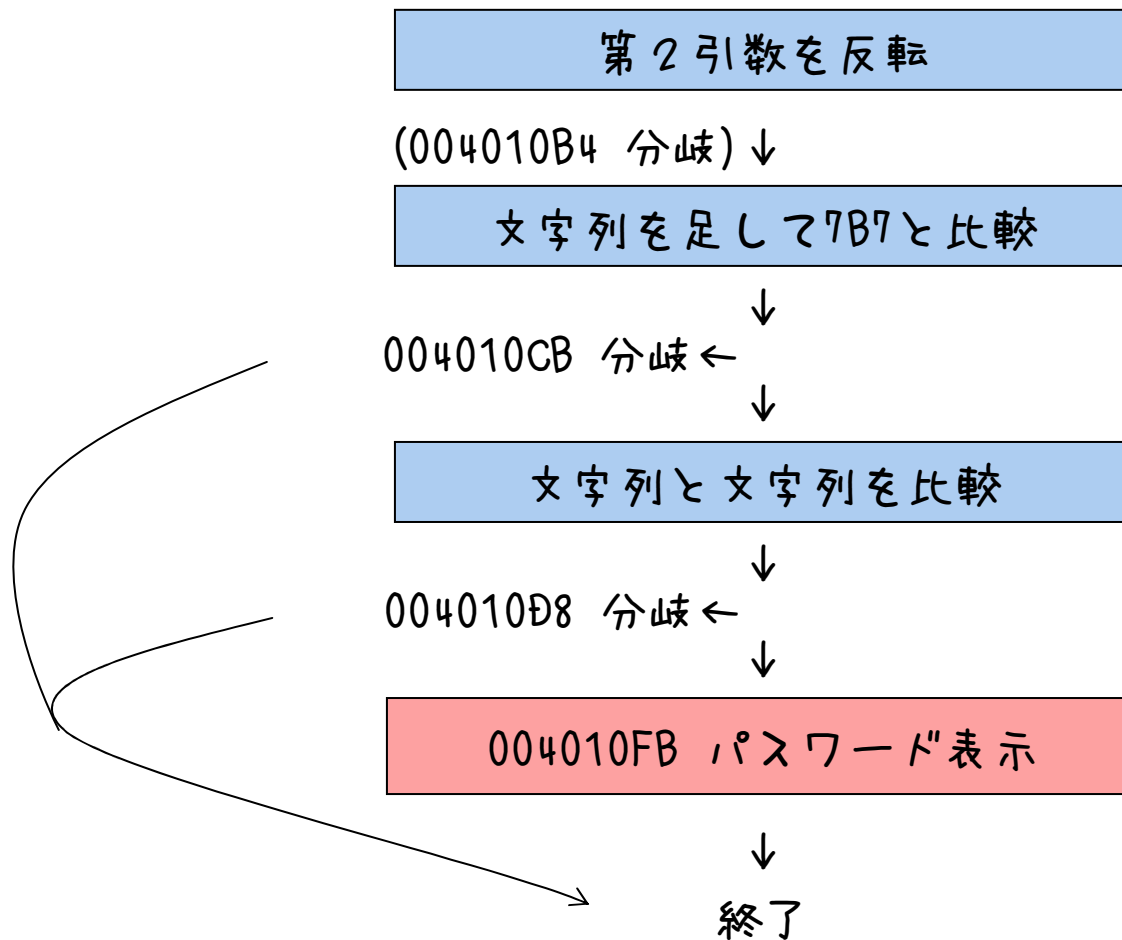
Jumps from 004010B4, 004010CB, 004010D8

実行したい処理の下にジャンプするところを探すと目的の分岐が見つかるよ



## Step 6 password: %s (補足)

- この辺の説明がわかりにくいので図にした





## Step 6 password: %s

- 下から順に確認してみる 00401D8
  - 004010CE |. E8 2DFFFFFF CALL level1.00401000 ; EAX=0だとダメポ
  - 004010D3 |. 83C4 04 ADD ESP, 4
  - 004010D6 |. 85C0 TEST EAX, EAX
  - 004010D8 |. 74 2E JE SHORT level1.00401108 ; ジャンプはダメ
- 関数00401000でEAXが0になるとダメポ
- 関数の中身を見ると、040100C~00401053の所で、データを作成。それを何かと比較
- データが同じだったらEAXは1!  
第2引数っぽいけど、そのままだと入力できない



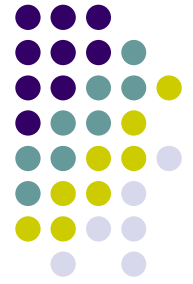
## Step 6 password: %s

- 004010CBを試してみる . . .

● 004010B8	> 0FB6C0	/MOVZX EAX, AL
● 004010BB	. 03D0	ADD EDX, EAX ; EDXにEAXを足す
● 004010BD	. 8A41 01	MOV AL, BYTE PTR DS:[ECX+1]
● 004010C0	. 41	INC ECX
● 004010C1	. 84C0	TEST AL, AL
● 004010C3	. ^75 F3	¥JNZ SHORT level1.004010B8
● 004010C5	. 81FA B707000>	CMP EDX, 7B7 ; EDXが7B7でないとダメ
● 004010CB	75 3B	JNZ SHORT level1.00401108

- 何かを足すと7B7っぽい。

- ちなみにさっきのデータの合計値は7B7



## Step 6 password: %s

- 004010B4を見てみる。
- これはそのままでも通過している。
- その上の部分 (04010A0~04010AC) をF8で1行ずつ実行しながら眺めると、第2引数を一文字ずつ反転していることがわかる。
- 反転した文字を0040100C~00401053のデータと比較。比較される文字列を反転するとHello World! これが第2引数かな？



おしまい



```
C:\ コマンド プロンプト
D:\>level1.bin "good luck" "Hello World!"
password: Ifmmp!Xpsme"
D:\>_
```

解けたあ〜♪  
でも、2時間かかったよ・・・orz



# おまけ SS:[ESP+4] = 3のところ

当日、指摘されたところ・・・

- IDA Pro使ったらmain関数がいっぱい表示

```
.text:00401110 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401110 _main          proc near          ; CODE XREF: start+16E↓p
.text:00401110
.text:00401110 arg_0          = dword ptr 4
.text:00401110 arg_4          = dword ptr 8
.text:00401110
• .text:00401110          cmp     [esp+arg_0], 3
• .text:00401115          jz     short loc_40111D
• .text:00401117          mov    eax, 1
• .text:0040111C          retn
.text:0040111D ; ~~~~~
```

- [ESP+4] (DS:[4096FC]) はargcだってさ。

```
• .data:004096FC ; int argc
.data:004096FC argc          dd 0
.data:004096FC ; DATA XREF: start+168↑r
; __setargv+8F↑w
```

頑張ってコード読んでたのは無駄な努力だった・・・orz  
悔しいのでついでにIDA Proバージョン作ってみた



# おまけ 後日 IDA Proでも解析

IDA Pro 4.9Free版を使ってみました♪

- [File>Open]か[New]を開く。
- 実行形式はPEファイルを選択する
- level1.binを開くとこんな感じの画面

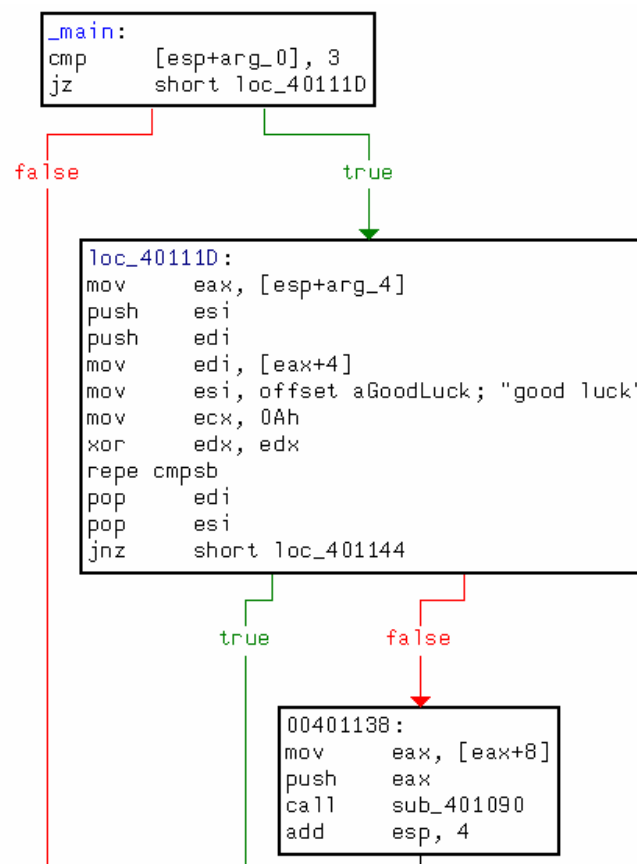
The screenshot shows the IDA Pro 4.9 Free interface. The main window displays assembly code for a subroutine. The Name window on the right lists symbols like `_main`, `_report_failure`, and `_printf`. The Strings window at the bottom right shows a string `good luck` at address `0000000A`.

```
.text:00401110  
.text:00401110 ; SUBROUTINE  
.text:00401110  
.text:00401110 ; int __cdecl main(int argc, const char **argv, const  
.text:00401110 _main proc near ; CODE XREF:  
.text:00401110 arg_0 = dword ptr 4  
.text:00401110 arg_4 = dword ptr 8  
.text:00401110  
* .text:00401110 cmp [esp+arg_0], 3  
.text:00401115 jz short loc_40111D  
.text:00401117 mov eax, 1  
.text:0040111C retn  
.text:0040111D ;  
.text:0040111E
```



# おまけ IDA Proバージョン

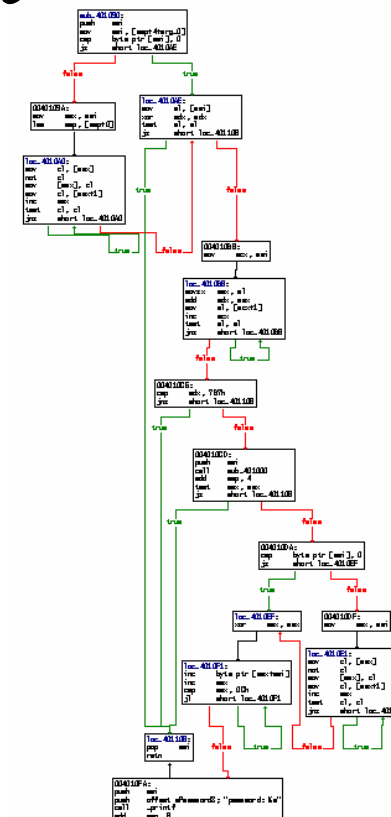
- いきなり main 関数のバレバレ
- F12 を押すと  
フローチャート表示
- `[esp+arg_0]=3`だと  
正解っぽい。
- `arg_0`は `argc` っぽい
- `[Debugger>  
Process options...]`で  
引数を2個指定



# おまけ IDA Proバージョン



- vepe cmpsbの後にjnz short loc\_401144とあるので、引数がgood luckでなければtrue(プログラム終了)
- falseの場合は401090を呼び出し
- 401090にカーソルあわせてF12  
グラフが表示される。(こんなの→)
- 4010FAでパスワード表示してる。
- 401108にいくとプログラム終了。



# おまけ IDA Proバージョン



- あとはStep6と同じようにがっつり解析。
- 呼び出しがあったらルーチンの中身を確認って感じでやればいいのだと思う。

IDA ProはこちらのページでDL可能です  
<http://www.hex-rays.com/idapro/>

おまけのおしまい

希望があればツールの使い方ネタもやりたいです