第３０話（マルウェア解析Ⅱ）

実行（ＥＸＥ）ファイルの解析

キツネ、マルウエア解析をする為に、どうして普通の実行ファイルの解析をしなければならないのだ？

タヌキ、マルウェアというのは多種多様なのだ。ここで全てのウィルスを扱うのは無理だし、ウィルスの名称や働きの解説をするつもりは無い。

オイラが興味を持っているマルウェアはファイル感染型ウィルスだ。ＣＩＨ（別名：Chernobyl チェルノブイリとも言う）ウィルスだ。これは、日常使用しているアプリケーションの一部を書き換え、アプリケーションを起動する度に、ＣＩＨウィルスのプログラムを呼び出し、何食わぬ顔をしながら裏で情報収集しながら外部に個人情報を流出させる、というものだ。

感染したアプリケーションは普通に起動し、使えるので感染に気が付かないのさ。また、ＣＩＨウィルスを呼び出すコードは、アプリケーションの空き領域に書き込まれるので、アプリケーションのファイル容量も増えないのさ。さらに、アプリケーションの起動よりもＣＩＨウィルスプログラムの方が先に呼び出される。

さらに、さらに、ＣＩＨウィルスがアプリケーションに書き込むコードはアセンブラ言語なので、Ghidra を用いての解析に最適なのだ。

ＣＩＨウィルスは主に Windows をターゲットにしているのじゃ無いのか？

その通り。だから CIH ウィルスを見つける為には、先ず実行ファイルの仕様を理解しておくことが必須なのだ。
Windows に標準添付されているノートパッドの場合を例で考えよう。notepad.exe などの Windows の実行ファイルはＰＥ(Portable Executable)フォーマットであり、今回用いた emfd.exe は Linux 上での実行ファイルなので ELF(Executable and Linkable Format) フォーマットなのだ。両者に若干の違いがあるが、基本的部分は同じようなものだ。emfd.exe は入出力だけの最も簡単なプログラムなので、これで解説していこうと思う。PE と ELF の構造の違いを以下に図示しておく。

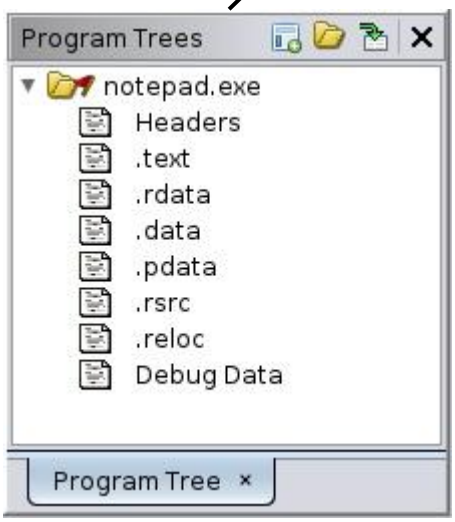
仮想アドレス　　ＰＥフォーマット

00400000

DOS ヘッダ
DOS スタブ
PE ヘッダ　シグネチャ / ファイルヘッダ / ｵﾌﾟｼｮﾅﾙﾍｯﾀﾞ
セクションテーブル
.text
.rdata
.data
:

00419363

notepad.exe(windows 実行ファイル)



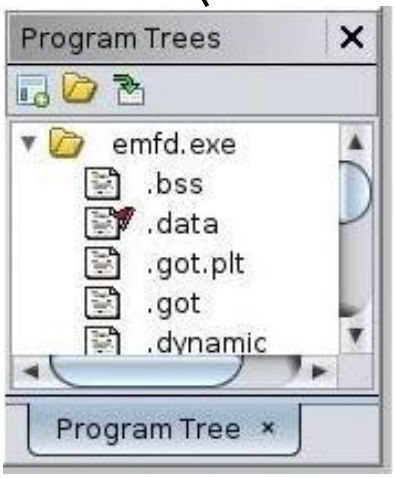
仮想アドレス　　ＥＬＦフォーマット

00400000

ELF ヘッダ
プログラムヘッダテーブル
.dss
.data
:
.rodata
セクションヘッダテーブル
:

00602020

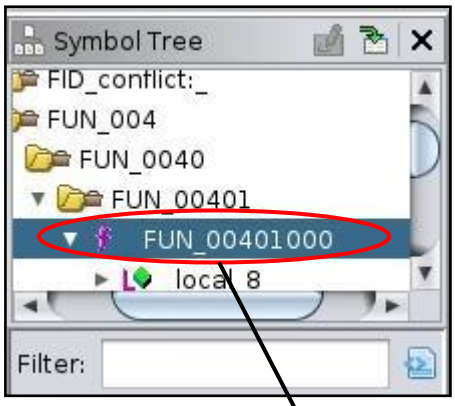
emfd.exe(Linux 実行ファイル)

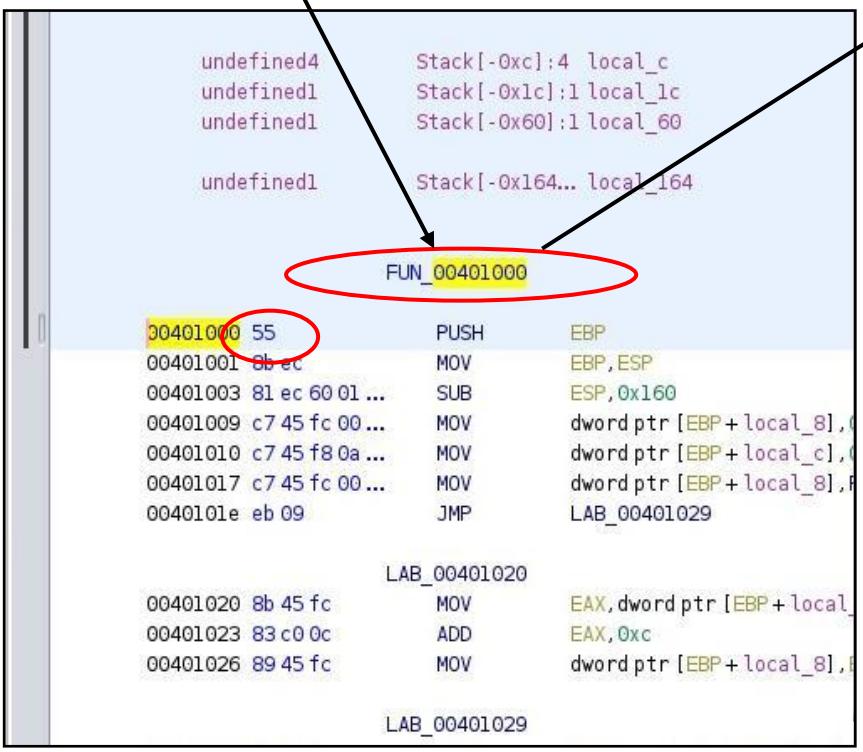


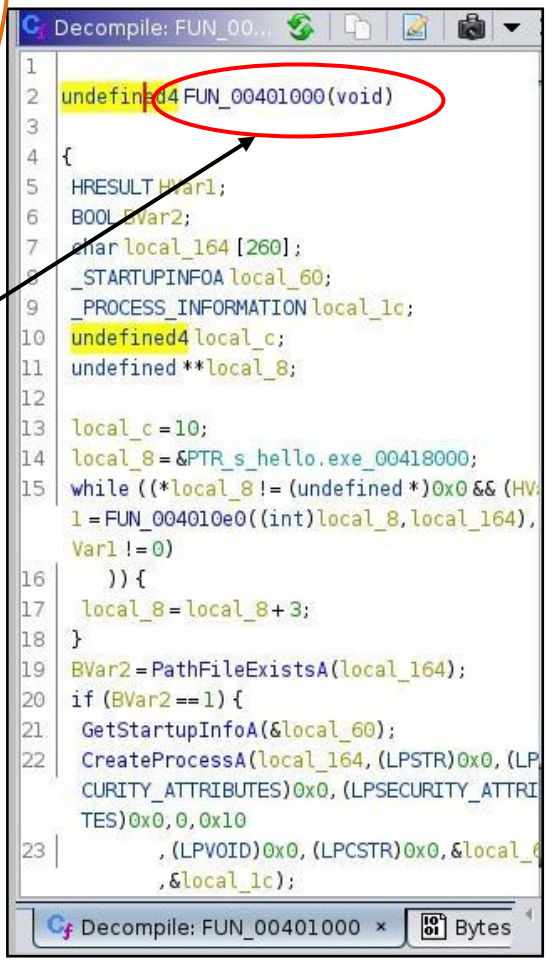
Ghidra で比較すると両者のフォーマット形式の違いが良くわかるな！

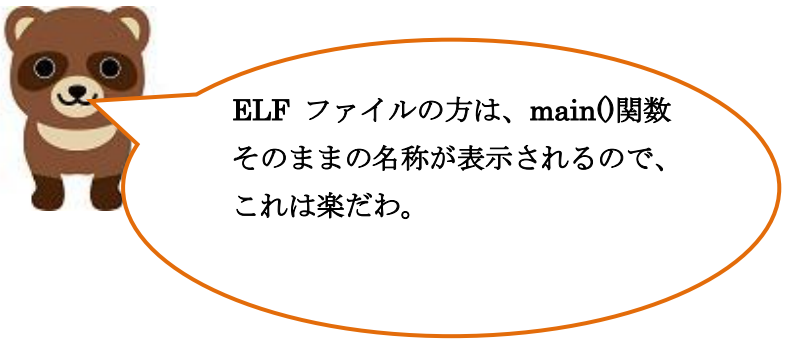
次は、C言語の実行ファイルを解析する時の注意点だ。C言語は、最初に main()関数から実行される。これは、PE ファイルも ELF ファイルも同じだ。ところが、ELF ファイルの main()関数の名称は main と表示されるのに、Windows の実行ファイルである PE(Portable Executable)ファイルの main()関数は、名称が main という名称にはなっていないのだ、それで main 関数に相当する関数を探さなければならない、という厄介な問題が発生するのだ。なぜなら、main()関数を見つけなければ解析が進まないのだ。やみくもに解析していくわけにもいかないしな。次に PE ファイルと ELF ファイルの main()関数の例を図示して置く。

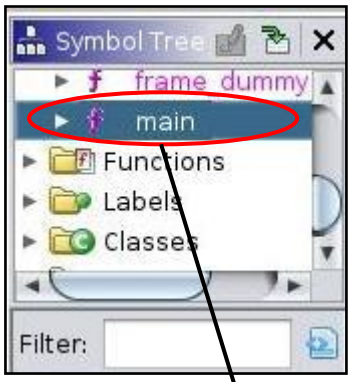
「FUN_00401000」関数が PE ファイルの main()関数か。これは、見つけるのは大変だわ。さらに、いつも「FUN_00401000」という名称とは限らないのだろ！

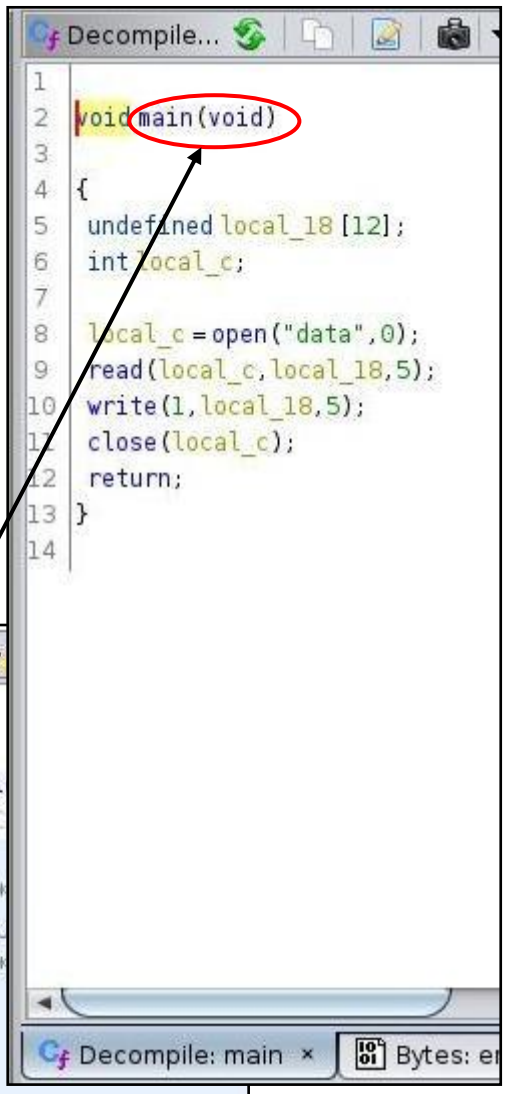


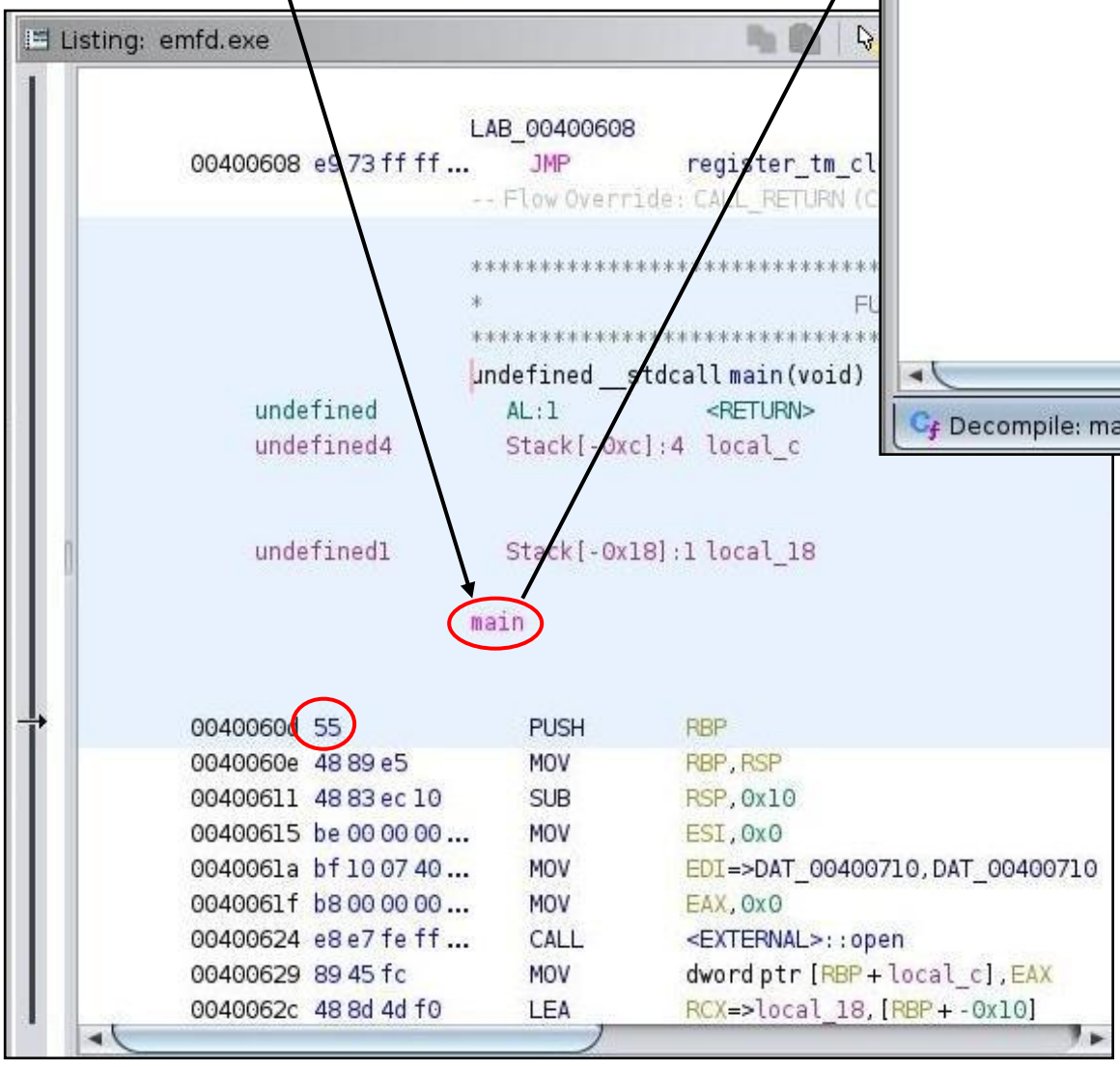




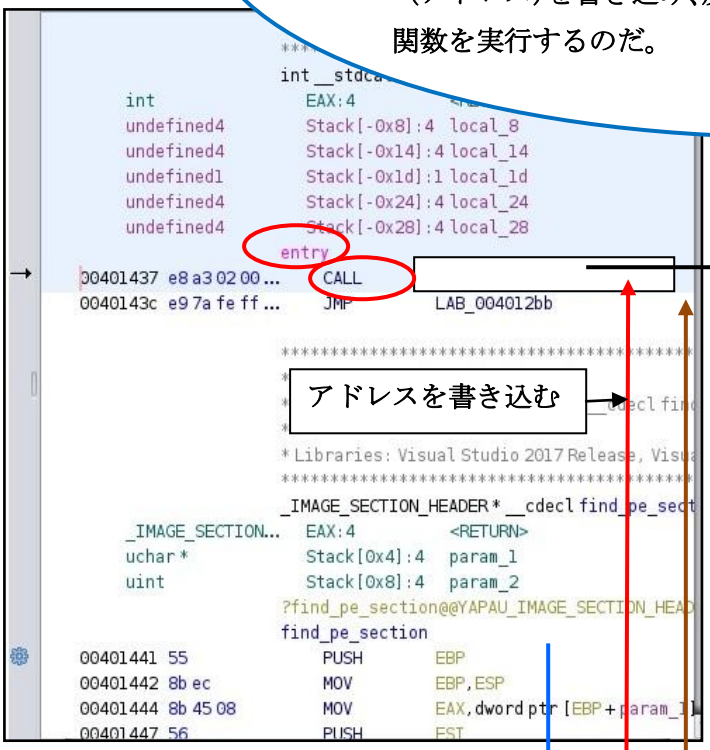

ELF ファイルの方は、main()関数
そのままの名称が表示されるので、
これは楽だわ。


Symbol Tree
frame_dummy
main
Functions
Labels
Classes
Filter:


Decompile...
1
2 void main(void)
3
4 {
5 undefined local_18[12];
6 int local_c;
7
8 local_c = open("data",0);
9 read(local_c,local_18,5);
10 write(1,local_18,5);
11 close(local_c);
12 return;
13 }
14
Decompile: main
Bytes: e

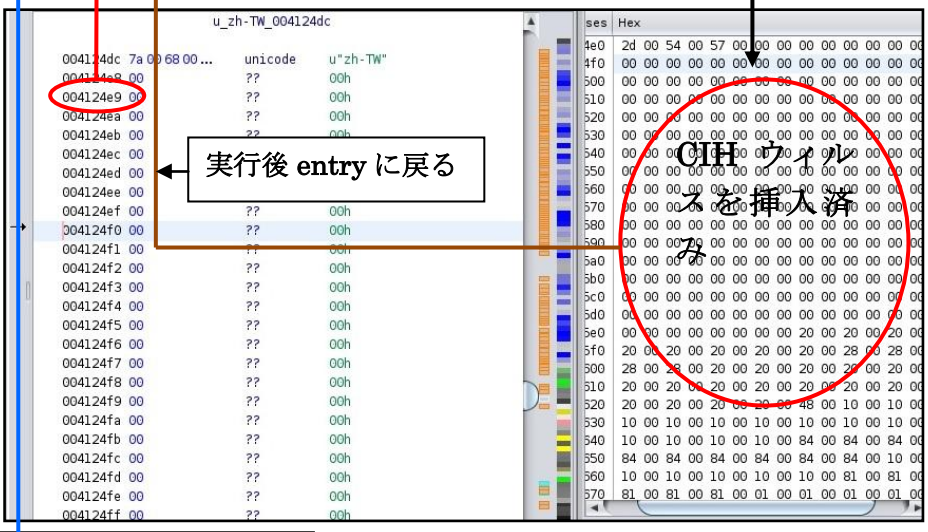

Listing: emfd.exe
LAB_00400608
00400608 e9 73 ff ff ... JMP register_tm_cl
-- Flow Override: CALL_RETURN (C
undefined __stdcall main(void)
undefined AL:1 <RETURN>
undefined4 Stack[-0xc]:4 local_c
undefined1 Stack[-0x18]:1 local_18
main
00400600 55 PUSH RBP
0040060e 48 89 e5 MOV RBP,RSP
00400611 48 83 ec 10 SUB RSP,0x10
00400615 be 00 00 00 ... MOV ESI,0x0
0040061a bf 10 07 40 ... MOV EDI=>DAT_00400710,DAT_00400710
0040061f b8 00 00 00 ... MOV EAX,0x0
00400624 e8 e7 fe ff ... CALL <EXTERNAL>::open
00400629 89 45 fc MOV dword ptr [RBP + local_c],EAX
0040062c 48 8d 4d f0 LEA RCX=>local_18,[RBP + -0x10]

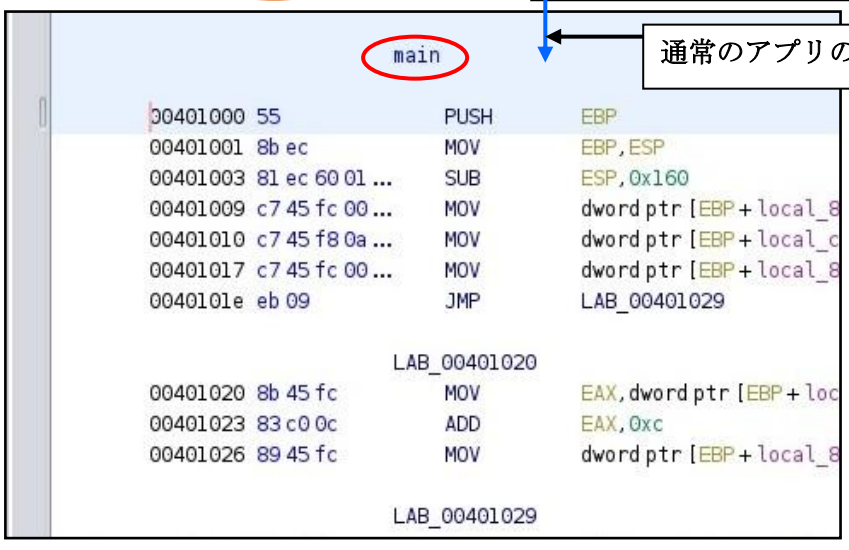
では、CIH ウイルスの感染の仕組みを図解しておくか。C 言語で作られた ELF フォーマット EXE ファイルには、entry()関数と main()関数が存在する。最初に entry()関数が実行されるので、その CALL 命令にウィルスを組み込んだスクリプトのある場所（アドレス）を書き込み、戻ってきたら、何食わぬ顔をして main()関数を実行するのだ。



なるほど、アプリには必ず使用されていない場所があるので、そこにウィルスを組むとアプリの容量も増えないのか。最初にそれを実行し、戻るのか！

entry()関数は、確かアプリの初期化を行う関数だよな。Windows アプリには entry 関数は無く、WinMain()関数が両方の働きをしているのかな？





通常のアプリの実行

「第３１話」では、実際に保存されているパスワードをどのように読み取るかだ。