第２９話（マルウェア解析Ｉ）

Ｇｈｉｄｒａ（ギドラ）ツールの利用

タヌキ、リバースエンジニアリングを実施するには、それなりのツールが必要だ。
リバースエンジニアリングのツールには、「IDA Freewire」や「Ghidra」がある。「IDA Freewire」は商用製品の「IDA Pro」のフリー版である。「Ghidra」はアメリカ国家安全保障局（NSA）が開発したオープンソースソフトウェアである。今、注目されているツールなので此方を使うことにする。
これまで同様に「Ghidra」の操作の解説をするつもりは無いので、「Ghidra」の詳細は自分で勉強してくれ。「Ghidra」の解説書だけで６００ページ以上の専門書になるほどだ。

キツネ、了解した。操作に必要な部分は、ネットや本を見て自分で勉強するよ。ところで、「Ghidra」の内容を理解する為の前提条件はなんだ。

タヌキ、「Ghidra」の内容を理解する為の前提条件はアセンブラ言語とＣ言語とバイナリ（１６進数）ファイルの理解だ。
でも、心配しなくて良いよ。「Ghidra」の操作以外は実例を使って、わかり易く説明するから。
リバースエンジニアリングを通して、アセンブラとＣ言語の重要性を再認識してくれ。

実例とは何だ？何を使うつもりだ。

テキストファイル（data）を読み込み、内容を画面に表示する、という標準入出力（stdio.h）しか使わない簡単な C 言語プログラム（mfd.c）を使う。
簡単な既知のソースプログラムを使うとスクリプトが４行程度なので「Ghidra」で、どのように逆コンパイルされるか、さらに逆アセンブルされるのか、区分けが明瞭になるのだ。ただ、注意しなければならないのは、ソースから実行形式のファイルが作られる時にＣ言語のプログラムでもある stdio.h が取り込まれている点である。以下にその状態を図示する。

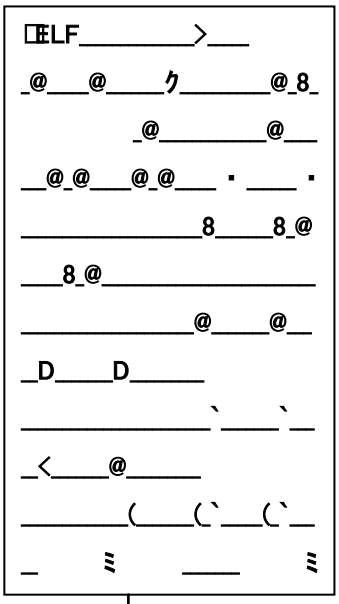
mfd.c（ソース）

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int fd;
    char buf[10];
    fd = open("data",0);
    read(fd,buf,5);
    write(1,buf,5);
    close(fd);
}
```

CentOS で C 言語の実行ファイル作成
$ gcc -o emfd.exe mfd.c
実行ファイル ソース

同時に stdio.h を取り込む

emfd.exe を無理に開いた状態

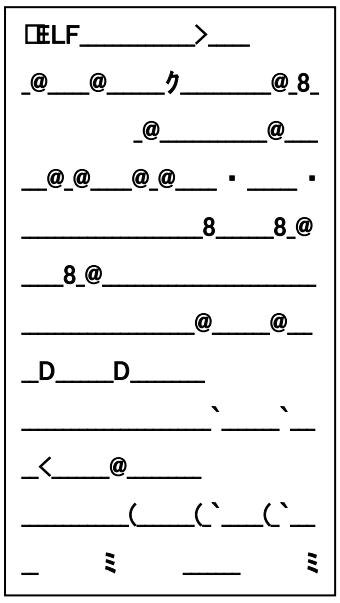


「Ghidra」で逆コンパイル&逆アセンブル

なるほどね。実行ファイル（emfd.exe）をメモ帳などで開いてもグチャグチャでわけわかんないけれども、今度はこれを基にして、アセンブラ言語や元のソースに戻し、比較しながら「Ghidra」の仕組みを勉強するのか。やるね、キツネ。

タヌキ！マルウエアは、実行ファイル（exe）に組み込まれる。
次は、実行ファイルからのリバースエンジニアリングの流れだ。

emfd.exe



バイナリエディッタで開く

emfd.exe(機械語)

```
31 ed 49 89 d1 5e 48 89 e2 48 83 e4 f0 50 54 49
c7 c0 f0 06 40 00 48 c7 c1 80 06 40 00 48 c7 c7
0d 06 40 00 e8 a7 ff ff ff f4 66 0f 1f 44 00 00
b8 57 10 60 00 55 48 2d 50 10 60 00 48 83 f8 0e
48 89 e5 77 02 5d c3 b8 00 00 00 00 48 85 c0 74
f4 5d bf 50 10 60 00 ff e0 0f 1f 80 00 00 00 00
b8 50 10 60 00 55 48 2d 50 10 60 00 48 c1 f8 03
48 89 e5 48 89 c2 48 c1 ea 3f 48 01 d0 48 d1 f8
75 02 5d c3 ba 00 00 00 00 48 85 d2 74 f4 5d 48
89 c6 bf 50 10 60 00 ff e2 0f 1f 80 00 00 00 00
80 3d 85 0a 20 00 00 75 11 55 48 89 e5 e8 7e ff
ff ff 5d c6 05 72 0a 20 00 01 f3 c3 0f 1f 40 00
48 83 3d 38 08 20 00 00 74 1e b8 00 00 00 00 48
85 c0 74 14 55 bf 20 0e 60 00 48 89 e5 ff d0 5d
e9 7b ff ff ff 0f 1f 00 e9 73 ff ff ff 55 48 89
e5 48 83 ec 10 be 00 00 00 00 bf 10 07 40 00 b8
00 00 00 00 e8 e7 fe ff ff 89 45 fc 48 8d 4d f0
8b 45 fc ba 05 00 00 00 48 89 ce 89 c7 b8 00 00
00 00 e8 99 fe ff ff 48 8d 45 f0 ba 05 00 00 00
48 89 c6 bf 01 00 00 00 b8 00 00 00 00 e8 5e fe
ff ff 8b 45 fc 89 c7 b8 00 00 00 00 e8 5f fe ff
ff c9 c3 66 2e 0f 1f 84 00 00 00 00 00 0f 1f 00
41 57 41 89 ff 41 56 49 89 f6 41 55 49 89 d5 41
54 4c 8d 25 78 07 20 00 55 48 8d 2d 78 07 20 00
```

逆アセンブル

emfd.exe のアセンブラ言語

```
0040060d 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]
00400630 8b 45 fc        MOV     EAX,dword ptr [RBP + local_c]
00400633 ba 05 00 00 ...  MOV     EDX,0x5
00400638 48 89 ce        MOV     RSI,RCX
0040063b 89 c7           MOV     EDI,EAX
0040063d b8 00 00 00 ...  MOV     EAX,0x0
00400642 e8 99 fe ff ...  CALL    <EXTERNAL>::read
00400647 48 8d 45 f0     LEA     RAX=>local_18,[RBP + -0x10]
0040064b ba 05 00 00 ...  MOV     EDX,0x5
00400650 48 89 c6        MOV     RSI,RAX
00400653 bf 01 00 00 ...  MOV     EDI,0x1
00400658 b8 00 00 00 ...  MOV     EAX,0x0
0040065d e8 5e fe ff ...  CALL    <EXTERNAL>::write
00400662 8b 45 fc        MOV     EAX,dword ptr [RBP + local_c]
00400665 89 c7           MOV     EDI,EAX
00400667 b8 00 00 00 ...  MOV     EAX,0x0
0040066c e8 5f fe ff ...  CALL    <EXTERNAL>::close
00400671 c9              LEAVE
00400672 c3              RET
```

main（）関数の始まり

リバースエンジニアリングの流れが良く分かるぞ。16進数の 55 が main 関数の始まりか。すごいなキツネ！

逆コンパイル

そうだ、アセンブラ言語から逆コンパイルで大元のソースファイルのスクリプトに戻るというわけだ。
それで、逆コンパイルされたスクリプトと大元のソースファイルのスクリプトを比較することによってmain()関数の内容が殆ど同じことが分かると思うよ。違いは変数名ぐらいかな。大元の変数名までは読み取れないからね。

逆コンパイル

```
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
```

［逆コンパイルしたファイルと大元のソースファイル（mfd.c）の比較］

逆コンパイル

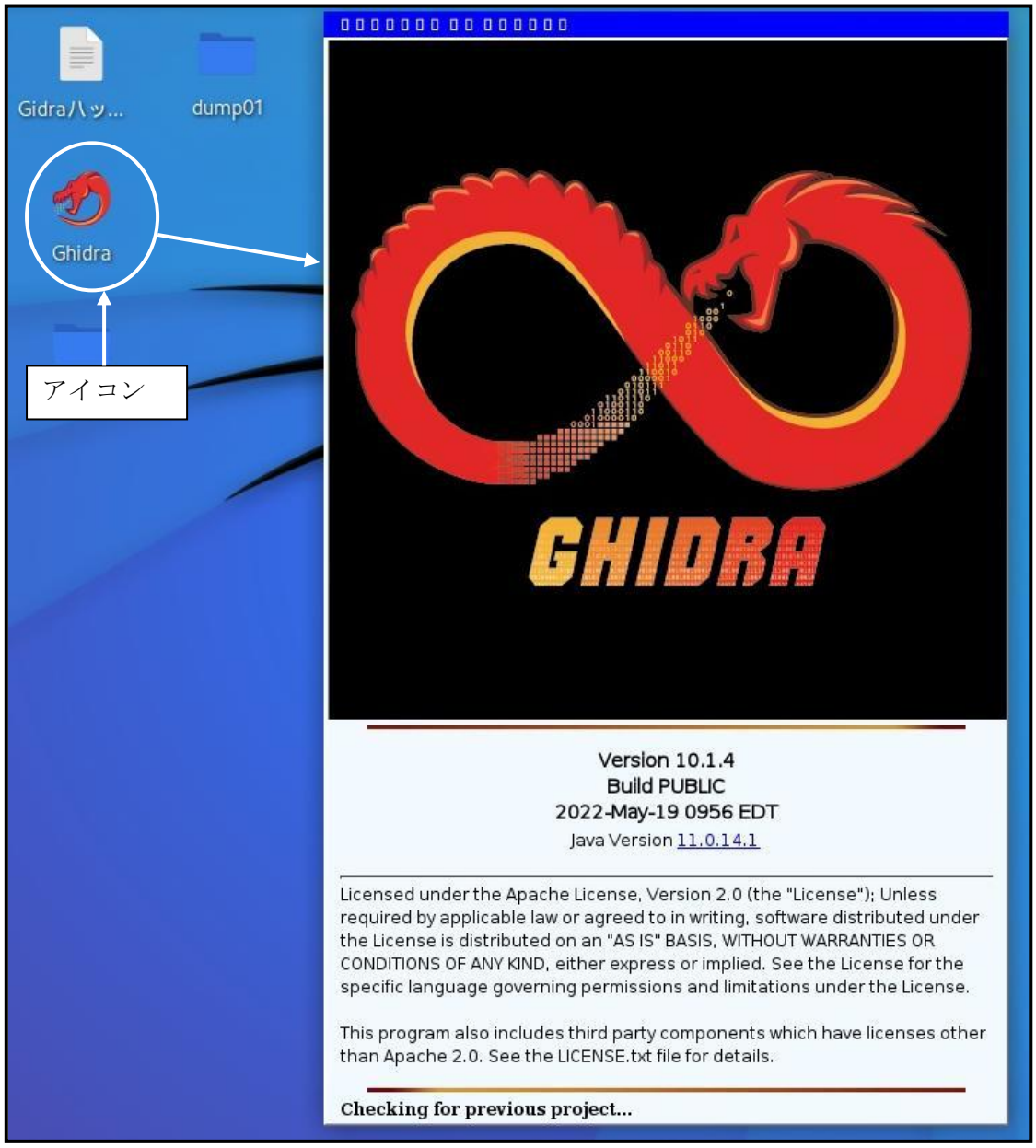
```
1
2  void main(void)
3
4  {
5   undefined local_18[12];
6   int local_c;
7
8   local_c = open("data",0);
9   read(local_c,local_18,5);
10  write(1,local_18,5);
11  close(local_c);
12  return;
13 }
14
```

大きな違い
変数名

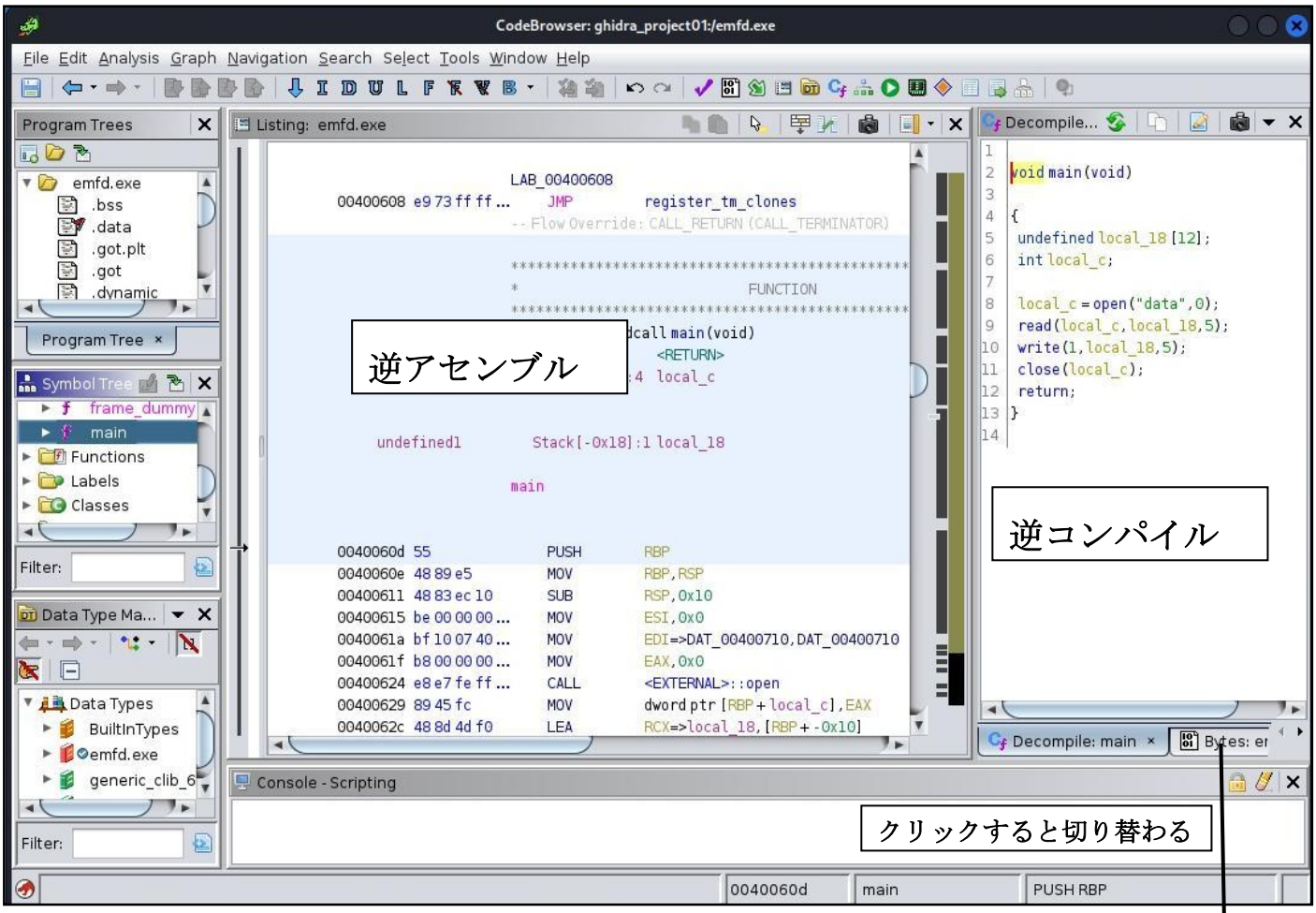
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int fd;
    char buf[10];

    fd = open("data",0);
    read(fd,  buf,  5);
    write(1,  buf,  5);
    close(fd);
}
```

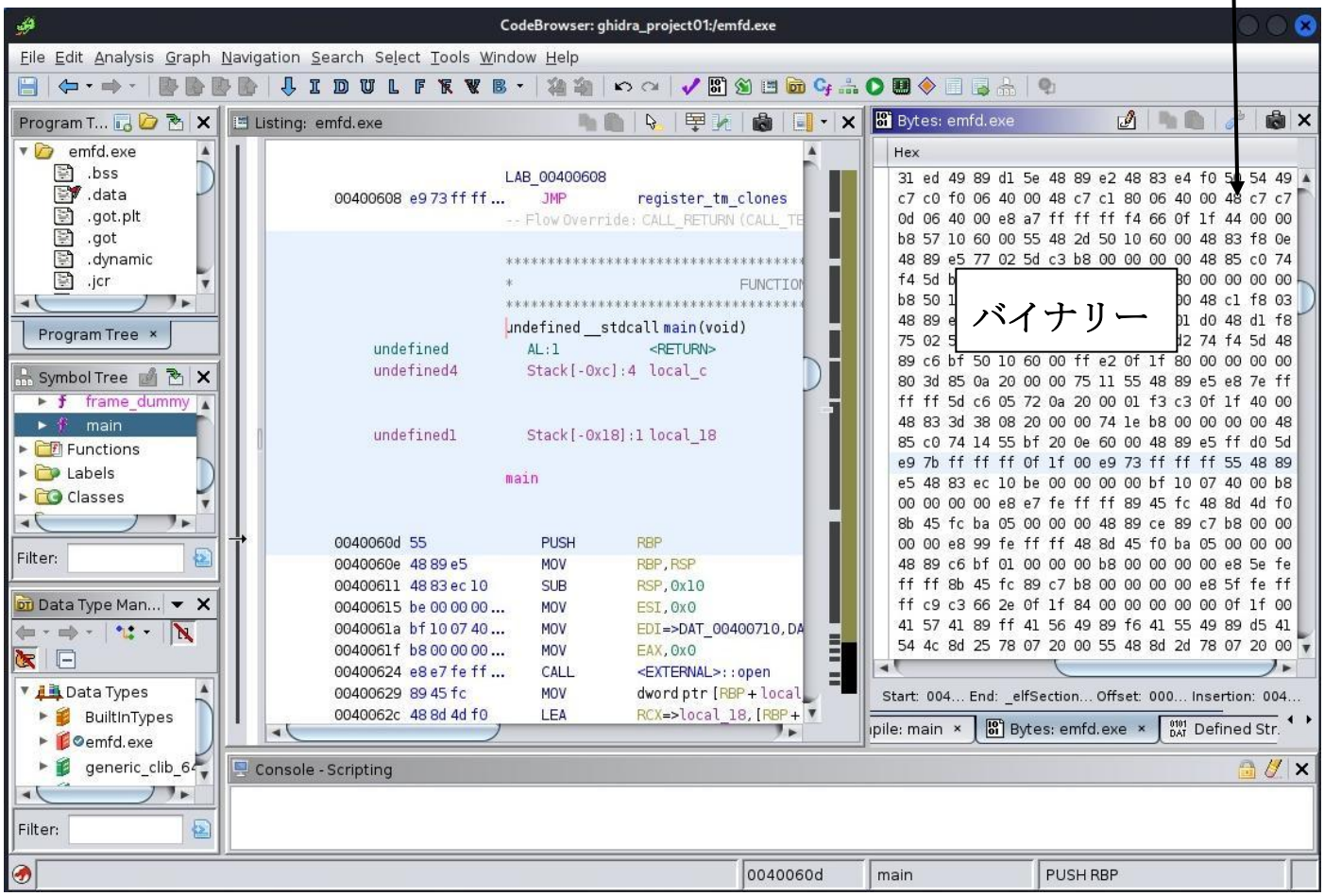
タヌキ、実行ファイル（emfd.exe）を読み込んで、バイナリエディッタ、逆アセンブル、逆コンパイルを一気に行って、表示してくれるのが「Ghidra」ツールなのだ。以下に（emfd.exe）を読み込んで「Ghidra」で表示した した図を提示する。ただ、いつものように「Ghidra」の「kali Linux」へのインストール手順はここでは解説しないので、必要ならばネットで調べてくれ。

［Ghidra（ギドラ）の起動］



[emfd.exe を Ghidra にインポート（読み込み）した状態]





キツネ、「Ghidra」（ギドラ）って仮想の怪獣なのだろうが、すごそうなオープニング画面だね。
何か、悪いことをしそうな感じだね。

リバースエンジニアリング自体、開発されたプログラムの内容を盗む、ということも含まれているからね。でも、ここではマルウェアを解析し、仕組みを知り、PC を守るということに使うのだ。

では、「第３０話」では、「Ghidra」を使った emfd.exe の勉強だ。