



## 第 3 1 話 (マルウェア解析Ⅲ)

## パスワード入力プログラムの解析例



タヌキ! 最初は、実行ファイルであるパスワード入力プログラム (passAnaD03) を解析して既に記憶されているパスワードを割り出す、ということをやってみる。

このプログラム (passAnaD03) の場合、正しいパスワードがエンコード (16進数への変換) されてスタック領域に積まれている。その状態は、Ghidra でデコンパイルされたプログラム (C 言語) でも見ることができる。

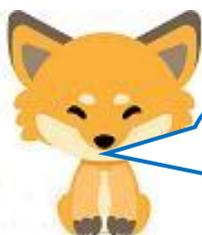
新規に入力されたパスワード (仮に Raccoon とする) と正しいパスワード (仮に yxxxxxxxxy とする) は両方ともエンコードされ、エンコードされたビット列を相互に XOR 演算をし、全てが0になれば入力されたパスワードが正しいと認証し、ログインできるという訳だ。

今回は、この正しいパスワード (仮に yxxxxxxxxy とする) を割り出し、ASCII 文字として表示しようという試みだ。

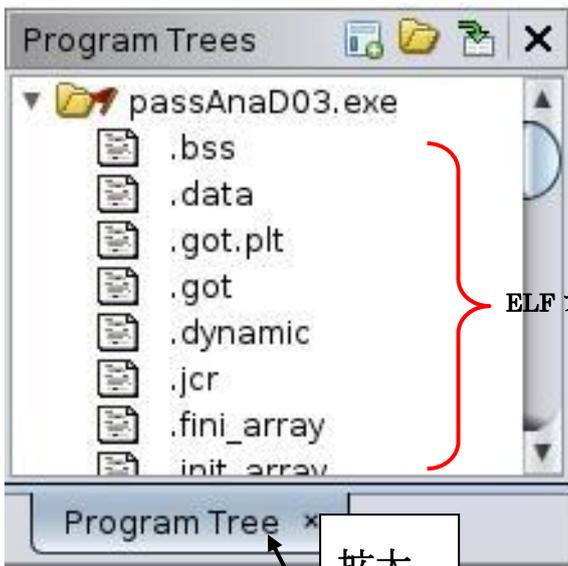
ただ、パスワードは16進にエンコードされるという単純なものではなく、実際にはハッシュ関数などでメッセージダイジェスト形式に変換されているので、簡単に復元することができない、ということに注意してもらいたい。ここは、あくまでも仕組みの勉強だ。



キツネ、これから実習に使う「passAnaD03」実行ファイルは、PE フォーマットと ELF フォーマットのどちらだ?

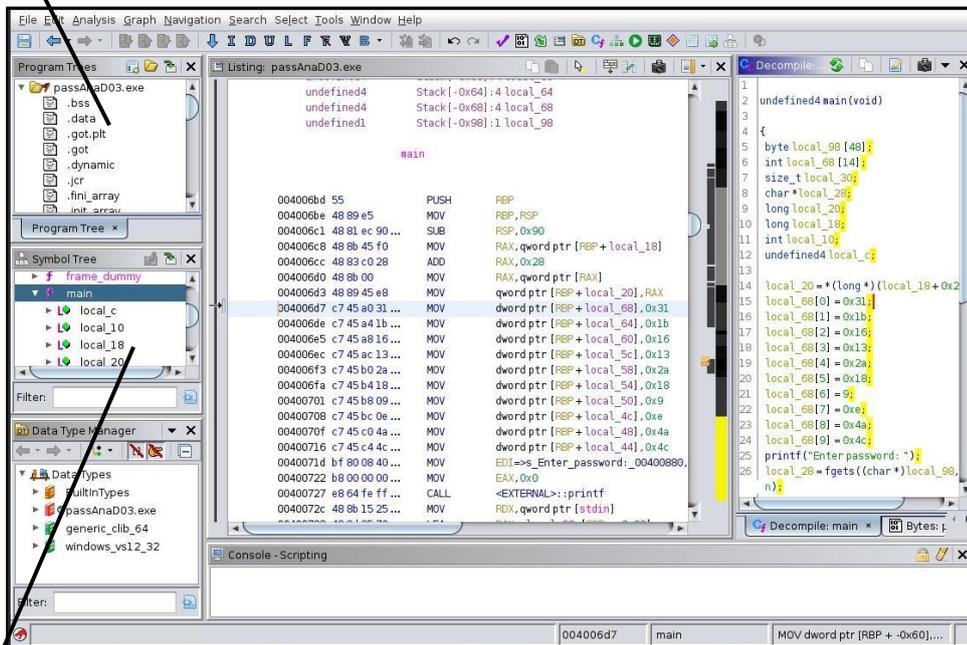


「passAnaD03」はソースプログラムが Linux 上の C 言語で開発されたものなので、ELF フォーマットだ。以下に、「passAnaD03」からパスワードを割り出す方法を図示する。



キツネ、これが「passAnaD03」実行ファイルを Ghidra にインポートした状態か。左の Program Trees を見ると、ELF フォーマットになっているな。

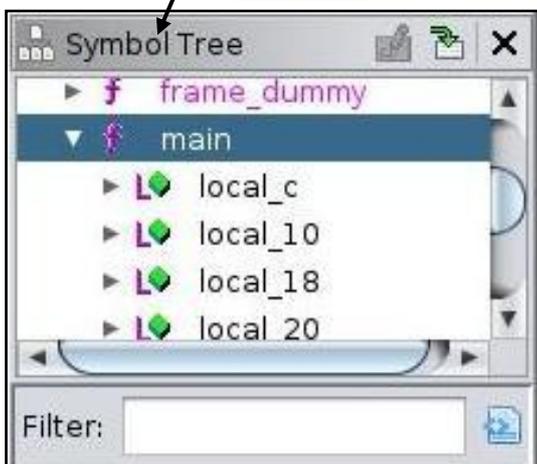
拡大



拡大



ELF バイナリ（実行ファイルのこと）は、main () で始まるのでここをクリックすれば、逆アセンブラスクリプトや逆コンパイラのスクリプトが表示されるのか。  
Windows の PE バイナリの場合も main() で始まるが、main という表示でないの、main() を探すのが大変だよな。



```
Listing: passAnaD03.exe

undefined4      Stack[-0x64]
undefined4      Stack[-0x68]
undefined1      Stack[-0x98]

main

004006bd 55          PUSH      RBP
004006be 48 89 e5     MOV       RBP,RSP
004006c1 48 81 ec 90 ... SUB       RSP,0x90
004006c8 48 8b 45 f0 ... MOV       RAX,qword ptr [RBP+local_18]
004006cc 48 83 c0 28 ... ADD       RAX,0x28
004006d0 48 8b 00     MOV       RAX,qword ptr [RAX]
004006d3 48 89 45 e8 ... MOV       qword ptr [RBP+local_20],RAX
004006d7 c7 45 a0 31 ... MOV       dword ptr [RBP+local_68],0x31
004006de c7 45 a4 1b ... MOV       dword ptr [RBP+local_64],0x1b
004006e5 c7 45 a8 16 ... MOV       dword ptr [RBP+local_60],0x16
004006ec c7 45 ac 13 ... MOV       dword ptr [RBP+local_5c],0x13
004006f3 c7 45 b0 2a ... MOV       dword ptr [RBP+local_58],0x2a
004006fa c7 45 b4 18 ... MOV       dword ptr [RBP+local_54],0x18
00400701 c7 45 b8 09 ... MOV       dword ptr [RBP+local_50],0x9
00400708 c7 45 bc 0e ... MOV       dword ptr [RBP+local_4c],0xe
0040070f c7 45 c0 4a ... MOV       dword ptr [RBP+local_48],0x4a
00400716 c7 45 c4 4c ... MOV       dword ptr [RBP+local_44],0x4c
0040071d bf 80 08 40 ... MOV       EDI=>s_Enter_password:_00400880,
00400722 b8 00 00 00 ... MOV       EAX,0x0
00400727 e8 64 fe ff ... CALL     <EXTERNAL>:;printf
0040072c 48 8b 15 25 ... MOV       RDX,qword ptr [stdin]
```



これが逆アセンブラされた main 関数の部分か。確かにスタックに積まれたパスワードの最初の仮想アドレス (004006d7) を読み取ることができる。16進数の 31 がエンコードされたパスワードの最初の文字だな。

拡大

```
Listing: passAnaD03.exe

undefined4      Stack[-0x64]:4 local_64
undefined4      Stack[-0x68]:4 local_68
undefined1      Stack[-0x98]:1 local_98

main

004006bd 55          PUSH      RBP
004006be 48 89 e5     MOV       RBP,RSP
004006c1 48 81 ec 90 ... SUB       RSP,0x90
004006c8 48 8b 45 f0 ... MOV       RAX,qword ptr [RBP+local_18]
004006cc 48 83 c0 28 ... ADD       RAX,0x28
004006d0 48 8b 00     MOV       RAX,qword ptr [RAX]
004006d3 48 89 45 e8 ... MOV       qword ptr [RBP+local_20],RAX
004006d7 c7 45 a0 31 ... MOV       dword ptr [RBP+local_68],0x31
004006de c7 45 a4 1b ... MOV       dword ptr [RBP+local_64],0x1b
004006e5 c7 45 a8 16 ... MOV       dword ptr [RBP+local_60],0x16
004006ec c7 45 ac 13 ... MOV       dword ptr [RBP+local_5c],0x13
004006f3 c7 45 b0 2a ... MOV       dword ptr [RBP+local_58],0x2a
004006fa c7 45 b4 18 ... MOV       dword ptr [RBP+local_54],0x18
00400701 c7 45 b8 09 ... MOV       dword ptr [RBP+local_50],0x9
00400708 c7 45 bc 0e ... MOV       dword ptr [RBP+local_4c],0xe
0040070f c7 45 c0 4a ... MOV       dword ptr [RBP+local_48],0x4a
00400716 c7 45 c4 4c ... MOV       dword ptr [RBP+local_44],0x4c
0040071d bf 80 08 40 ... MOV       EDI=>s_Enter_password:_00400880,
00400722 b8 00 00 00 ... MOV       EAX,0x0
00400727 e8 64 fe ff ... CALL     <EXTERNAL>:;printf
0040072c 48 8b 15 25 ... MOV       RDX,qword ptr [stdin]
```

```
Decompile: main(void)
{
    byte local_98[48];
    int local_68[14];
    size_t local_30;
    char* local_28;
    long local_20;
    long local_18;
    int local_16;
    undefined4 local_c;
    local_20 = *(long*)(local_18+0x2);
    local_68[0] = 0x31;
    local_68[1] = 0x1b;
    local_68[2] = 0x16;
    local_68[3] = 0x13;
    local_68[4] = 0x2a;
    local_68[5] = 0x18;
    local_68[6] = 9;
    local_68[7] = 0xe;
    local_68[8] = 0x4a;
    local_68[9] = 0x4c;
    printf("Enter password: ");
    local_28 = fgets((char*)local_98,
    n);
}
```

```

1
2 undefined4 main(void)
3
4 {
5  byte local_98 [48];
6  int local_68 [14];
7  size_t local_30;
8  char *local_28;
9  long local_20;
10 long local_18;
11 int local_10;
12 undefined4 local_c;
13
14 local_20 = *(long *) (local_18 + 0x2
15 local_68[0] = 0x31;
16 local_68[1] = 0x1b;
17 local_68[2] = 0x16;
18 local_68[3] = 0x13;
19 local_68[4] = 0x2a;
20 local_68[5] = 0x18;
21 local_68[6] = 9;
22 local_68[7] = 0xe;
23 local_68[8] = 0x4a;
24 local_68[9] = 0x4c;
25 printf("Enter password: ");
26 local_28 = fgets((char *)local_98,
n);

```



バイナリファイルからリバースエンジニアリングするのは大変だが、このように逆コンパイルしてくれれば、一目瞭然ではないか。スタックに積まれているパスワードも見えるし、「passAnaD03」を実行した時に入力されるパスワードを取り込む fgets()関数も読み取ることができるし、その後の XOR 処理も読み取れる。「passAnaD03」のコンパイル前のソースと違うのは変数名ぐらいなものかな。

拡大

```

Listing: passAnaD03.exe
main
004006bd 55      PUSH   RBP
004006be 48 89 e5 MOV    RBP,RSP
004006c1 48 81 e9 90... SUB    RSP,0x90
004006c8 48 8b 45 f0 MOV    RAX,qword ptr [RBP+local_18]
004006cc 48 83 c0 28 ADD    RAX,0x28
004006d0 48 8b 00 MOV    RAX,qword ptr [RAX]
004006d3 48 89 45 e8 MOV    RAX,qword ptr [RBP+local_20],RAX
004006d7 c7 45 a0 31... MOV    dword ptr [RBP+local_68],0x31
004006de c7 45 a4 1b... MOV    dword ptr [RBP+local_64],0x1b
004006e5 c7 45 a8 16... MOV    dword ptr [RBP+local_60],0x16
004006ec c7 45 ac 13... MOV    dword ptr [RBP+local_5c],0x13
004006f3 c7 45 b0 2a... MOV    dword ptr [RBP+local_58],0x2a
004006fa c7 45 b4 18... MOV    dword ptr [RBP+local_54],0x18
00400701 c7 45 b8 09... MOV    dword ptr [RBP+local_50],0x9
00400708 c7 45 bc 0e... MOV    dword ptr [RBP+local_4c],0xe
0040070f c7 45 c0 4a... MOV    dword ptr [RBP+local_48],0x4a
00400716 c7 45 c4 4c... MOV    dword ptr [RBP+local_44],0x4c
0040071d bf 80 08 40... MOV    EDI,>e_Enter_password:_00400880,
00400722 b8 00 00 00... MOV    EAX,0x0
00400727 e8 64 fe ff... CALL  <-EXTERNAL>:printf
0040072c 48 8b 15 25... MOV    RDX,qword ptr [stdin]

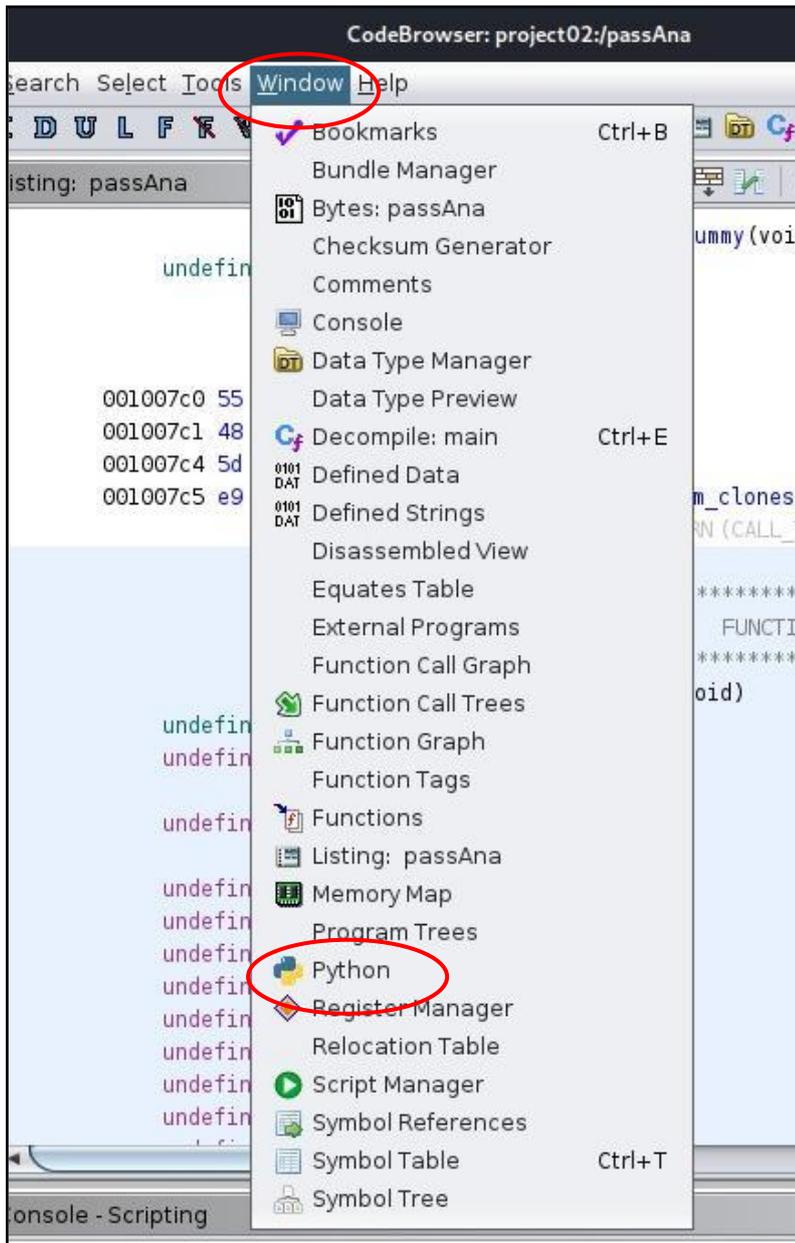
```



キツネ！バイナリファイルからディス（逆）アセンブルやデ（逆）コンパイルし、内容を読み取ることができることは理解した。読み取った後、どうするのだ？



タヌキの言うように、読み取った後が大切だよな。手順は、パスワードが積まれているスタック領域の最初の仮想アドレスを設定し、上から順に10個のエンコードされた16進データを取り出して、それをASCII文字に変換していくのだ。この作業をする為に、以下の図で示すように Ghidra に標準で付いている Python のコマンドプロンプトを使うのだ。



なるほどな、Python も使うのか。Python はインデント（字下げ）を守らないとエラーになるからな。



Python のコマンドプロンプトが開いたら、入力する命令やデータは次図のようになる。

Edit Help

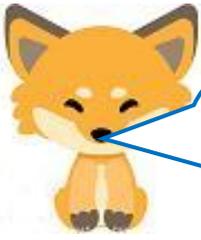
Python - Interpreter

```
>>> addr = toAddr(0x4006d7)
>>> inst = getInstructionAt(addr)
>>> enc = []
>>> for _ in range(10):
...     e = inst.getDefaultOperandRepresentation(1)
...     enc.append(int(e, 16))
...     inst = inst.getNext()
...
>>> enc
[49, 27, 22, 19, 42, 24, 9, 14, 74, 76]
>>> ''.join([chr(e ^ 0x7a) for e in enc])
'KaliPbst06'
>>>
```



なるほどな、toAddr()関数でスタックの最初の仮想アドレスを addr に保存し、enc 配列を定義し、getInstructionAt()関数で、仮想アドレス (4006d7) の部分のアセンブルされた 1 行を取り出している。繰り返し処理 (for) でパスワードの文字数分、10 回繰り返している。getDefaultOperandRepresentation(1)関数で inst からデータ (オペランドともいう) 部分 (0x31) を取り出している。取り出したものを、append () 関数で 16 進数と断って enc 配列に順に保存している。getNext()は言うまでも無く、次のアセンブルされた行に移動することだ。For 以下を入力する時には、インデントに注意しなければエラーになるよね。

繰り返し処理から抜ける場合には、単に「Enter」キーを押すだけで OK だ。enc と入力するだけで、配列に保存されたデータを確認することができる。次に enc 配列の値を 1 個ずつ変数 e に代入し、chr (e ^ 0x7a) で、XOR 処理をした文字コードを ASCII 文字に変換している。さらに、join()関数で 10 個の文字を結合し、一連の文字列にしている。それが**実際のパスワードである「KaliPbst06」だ**。キツネ、こんな解釈でいいかな？



タヌキの解釈で良いよ。  
正しいかどうかは、下図のように端末を開いて実行ファイル「passAnaD03」が存在するディレクトリに移動し、「passAnaD03」を実行してみることだ。

```
(kali㉿kali)-[~]  
└─$ ./passAnaD03.exe  
Enter password: abcdefghij  
Wrong ...
```



キツネ！これは、パスワードを割り出していないので、適当に「abcdefghij」と10個入力し、間違っているよ（Wrong）と拒否された状態だな。

```
(kali㉿kali)-[~]  
└─$ ./passAnaD03.exe  
Enter password: KaliPbst06  
Correct !!
```



次は、割り出したパスワード「**KaliPbst06**」を入力し、正しい（Correct）と受け入れられた状態だな。  
すごいなキツネ！



タヌキ、もう少し丁寧に説明しておこうか。  
 このパスワード解析には前提条件がある。  
 つまり、最初のパスワード「**KaliPbst06**」は、XOR（排他的論理和）でエンコードする、ということだ。XOR でエンコードされた文字は、XOR で元に戻すことができる、という前提条件で成り立っている。  
 論理演算子（AND、OR、XOR）については、タヌキよ、自分で勉強してね。以下に図解しようか。

パスワード：**KaliPbst06** のK（大文字）で考えよう！

K（大文字）の ASCII コードは16進数で「4b」だ。2進数で「01001011」だ。  
 これを、16進数「7a」（2進数で「01111010」）（任意の数値で良い）を用いて XOR でエンコードすると次のようになる。

[エンコード]

```

4 b   : 0 1 0 0 1 0 1 1
7 a   : 0 1 1 1 1 0 1 0
XOR   -----
      0 0 1 1 0 0 0 1  →  3 1 (エンコードした16進数)
  
```

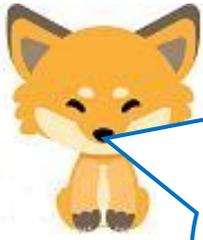
[元に戻す]

```

3 1   : 0 0 1 1 0 0 0 1
7 a   : 0 1 1 1 1 0 1 0
XOR   -----
      0 1 0 0 1 0 1 1  →  4 b (元に戻した16進数)
                               (ASCII コード：大文字のK)
  
```



なるほど、これを上記の Python で行っているのか。  
 すごいなキツネ！



タヌキ、今回オイラが参考にした文献はマイナビ発行の「Ghidra 実践ガイド」だ。そこが提供している学習用実行ファイル「Lebel1」を使用したのだが、「Lebel1」では、内容が把握しづらいのでパスワード解析ということで「passAnaD03」と変更して使わせてもらった。ここで断って置くぞ！  
他に、技術評論社の「UNIX ネットワークプログラミング入門」も参考にしている。