SECCON Beginners 2025 Kanazawa

Reversingに入門してみよう

n01e0

事前準備

① Ghidraをインストールし，指定のファイルを読み込み，一番右のパネル内に
`puts("Hello World!");`が表示される (事前準備スライドP.3~)

② x86-64のLinux実行環境を用意し，GDBとPedaをインストールし，指定のファイルを読み込む
(事前準備スライドP.48~)

自己紹介

n01e0 - Reo Shiseki

GMOペパボ所属

BunkyoWesterns

Reversing・Pwn・Misc・Web担当

https://feneshi.co/about

共著者

hi120ki - Hiroki Akamatsu

Web・Reversing担当

- CTF大会開催はいいぞ - 魔女のお茶会 2021冬

講義の流れ

Reversing分野や基礎知識の説明 → 実際にバイナリを解析する演習

質問・疑問があれば適宜sli.doにコメントをお願いします

https://www.slido.com/ #ctf4b

本演習の目的

CTFのReversing分野でよく出題される実行可能ファイルの解析問題の基礎と解析ツールの使い方を知ってもらう

1. CTFのReversing分野の紹介
2. 実行可能ファイルとは
3. ELFファイル解析の基本
4. ELFファイルを解析しながら問題を解いてみる
5. 実行可能ファイルの解析が役に立つ場面の紹介

1. CTFのReversing分野とは

ReversingはReverse Engineeringとも言われ，機械や製品・プログラムの内部の構造や動作を解析すること

特にCTFではプログラムの動作を解析する問題がよく出題される

(例)

- 実行可能ファイル(Executable file)
  - Linux ELF
  - Windows PE
- モバイルアプリ
  - APK(Android Application Package)
- スクリプトファイル
- pcapファイル
- ファームウェア

SECCON Beginners CTFでの出題傾向

- Linux ELF
  - 2020 : mask yakisoba sneaky
  - 2021 : only_read children please_not_trace_me be_angry firmware
  - 2022 : Quiz Recursive Ransom please_not_debug_me
  - 2023 : half heaven leak poker three
- Windows PE
  - 2022 : WinTLS
- APK
  - 2020 : siblangs
- スクリプトファイル
  - 2020 : ghost
- pcapファイル
  - 2022 : Ransom
- ファームウェア
  - 2021 : firmware

特にBeginner~Easy難易度はLinux ELFファイルの出題がほとんど

Linux ELFファイル

```c
#include <stdio.h>

int main(void) {
  printf("Hello World!\n");
  return 0;
}
```

```
$ gcc hello.c -o hello
$ ./hello
Hello World!
```

Linux上でコンパイラgccを使ってC言語のソースコードをコンパイルしたときに出力されるファイルはLinux ELFファイル

2. 実行可能ファイルとは

コンピュータが処理を実行するための命令を含むファイル

ELFはExecutable and Linkable Formatの略
多くのLinux系やBSD系のOSで実行可能ファイル形式として採用されている

Windowsでいうところのexeファイルで，Windowsだとダブルクリックで起動したりするが，
Linux ELFファイルは主にターミナルで実行する

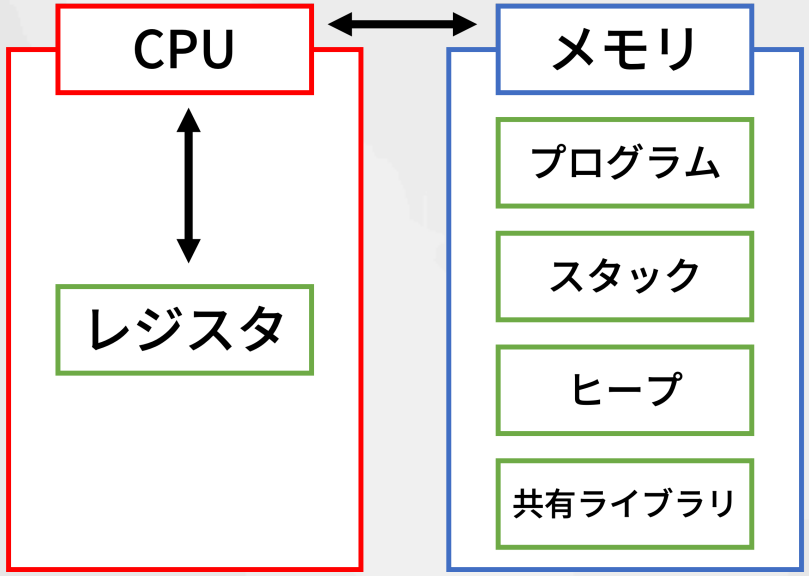
(gccで作成したELFファイル hello を実行する)

```
$ ./hello
Hello World!
```

そもそもプログラムはどう動いているのか

実行可能ファイル形式のプログラムはメモリ上にコピーされ，機械語から命令を1つずつCPUが実行していく

CPUは高速な保存領域のレジスタや，メモリのスタック領域やヒープ領域にデータを書き込んだり読み込んだりしながら命令を実行していく



そもそもプログラムはどう動いているのか

ソースコード

```c
#include <stdio.h>

int main(void)
{
  printf("Hello World!\n");
  return 0;
}
```

アセンブリコード

```
main:
  push  %rbp
  mov %rsp, %rbp
  lea .LC0(%rip), %rdi
  call  puts@PLT
  mov $0, %eax
  pop %rbp
  ret
```

機械語(バイナリ)

```
1EFAE977 FFFFFF55 4889E548 8D3DC00E
0000E8E7 FEFFFFB8 00000000 5DC3F30F
1EFA4157 4C8D3D5B 2C000041 564989D6
41554989 F5415441 89FC5548 8D2D4C2C
0000534C 29FD4883 EC08E87F FEFFFF48
C1FD0374 1F31DB0F 1F800000 00004C89
F24C89EE 4489E741 FF14DF48 83C30148
39DD75EA 4883C408 5B5D415C 415D415E
415FC366 662E0F1F 84000000 0000F30F
1EFAC300 0000F30F 1EFA4883 EC084883
C408C300 00000000 00000000 00000000
```

gccのようなソースコードから直接実行可能ファイルを出力する際

- ソースコードをコンパイルしてアセンブリコードを生成
- アセンブリコードをアセンブルして機械語を生成し，実行可能ファイルとして出力する(厳密にはこの中でリンクなど他の処理も行う)

そもそもプログラムはどう動いているのか

ソースコード

```c
#include <stdio.h>

int main(void)
{
  printf("Hello World!\n");
  return 0;
}
```

アセンブリコード

```
main:
  push  %rbp
  mov %rsp, %rbp
  lea .LC0(%rip), %rdi
  call  puts@PLT
  mov $0, %eax
  pop %rbp
  ret
```

機械語(バイナリ)

```
1EFAE977 FFFFFF55 4889E548 8D3DC00E
0000E8E7 FEFFFFB8 00000000 5DC3F30F
1EFA4157 4C8D3D5B 2C000041 564989D6
41554989 F5415441 89FC5548 8D2D4C2C
0000534C 29FD4883 EC08E87F FEFFFF48
C1FD0374 1F31DB0F 1F800000 00004C89
F24C89EE 4489E741 FF14DF48 83C30148
39DD75EA 4883C408 5B5D415C 415D415E
415FC366 662E0F1F 84000000 0000F30F
1EFAC300 0000F30F 1EFA4883 EC084883
C408C300 00000000 00000000 00000000
```

- アセンブリコードからソースコードへは正確に戻せない
- アセンブリコードと機械語は一対一対応なので戻せる

→ アセンブリコードを読めるようになることで，プログラムがどう動くかを「正確」に知ることができる

Pythonはどう動いているのか

ソースコードをそのまま実行できるPythonのようなインタープリター型言語はどう動くのか

→ Pythonのソースコードを実行しているプログラムはELFファイルなどの実行可能ファイル(=機械語からなるバイナリファイル)
→ 正確にPythonの動作を知りたい場合，アセンブリコードを読めるようになる必要がある…?

```
$ file python
python: ELF 64-bit LSB shared object, x86-64,
version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=43f02f5bee4427e07ed423fb8f31e77617aaaefc,
for GNU/Linux 3.2.0, with debug_info, not stripped
```

3. ELFファイル解析の基本

ソースコードに記述された処理は機械語に変換されるとどんな処理が行われているか分かりづらくなる
また，マルウェア等では処理を秘匿するために更に処理が分かりづらくなるような工夫がされることもある

→ 機械語(=アセンブリコード)から処理を正確に把握する能力を養う

ELFファイル解析の流れ

1. 表層解析

ファイルの種類を調べる

ファイルに含まれる文字列からファイルの種類や動作の手がかりを取得する

2. 静的解析

バイナリに含まれる機械語の命令からプログラムの動作を把握する

3. 動的解析

ファイルを実行しながら処理を追うことでプログラムの動作を把握する

ELFファイル解析でよく使うツール

1. 表層解析

fileコマンド	ファイル形式を推測
stringsコマンド	ファイルに含まれる可読文字列を表示

2. 静的解析

Ghidra	NSAによって開発されているOSSの解析ツール
IDA	Hex Rays社によって開発されている有償(機能が制限された無償版もあり)の解析ツール

3. 動的解析

GDB	GNU Project debugger 特にLinuxではデファクトスタンダードなデバッグ&動的解析ツール

4. ELFファイルを解析しながら問題を解いてみる

表層解析ツール・静的解析ツールでは seiteki_kaiseki

https://ctf4b.epitomize.dev/bin/seiteki_kaiseki

動的解析ツールでは douteki_kaiseki

https://ctf4b.epitomize.dev/bin/douteki_kaiseki

を実際に解きながら紹介します

ここから作業画面に切り替わります．この資料には作業内で行うことを書いてあるので適宜参考にしてください

4-1. 表層解析ツール

解析対象のプログラム

seiteki_kaiseki

https://ctf4b.epitomize.dev/bin/seiteki_kaiseki

```
$ wget https://ctf4b.epitomize.dev/bin/seiteki_kaiseki
$ chmod +x seiteki_kaiseki
$ ./seiteki_kaiseki
Please input password:
hoge
Incorrect Bye...
```

実行するとパスワードの入力を求められる

fileコマンド

```
$ sudo apt-get install -y file
```

でインストールできる

```
$ file seiteki_kaiseki
seiteki_kaiseki: ELF 64-bit LSB shared object, x86-64,
version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=65b22e0cbb1adc2bafcbfa40f93c18fbf2573465,
for GNU/Linux 3.2.0, not stripped
```

x86-64環境で動作するELFファイルである事が確認できる

stringsコマンド

```
$ sudo apt-get install -y binutils
```

でインストールできる

```
$ strings seiteki_kaiseki
...
ctf4b{we1cme_4o_rev4r3ing_2o22}
Please input password:
Please input PIN:
Correct!!!
Incorrect Bye...
...
```

加工されていないそのままの形でバイナリ内に埋め込まれている文字列はstringsコマンドで取得できる

stringsコマンド

怪しい文字列 `ctf4b{we1cme_4o_rev4r3ing_2o22}` を入力してみると

```
$ ./seiteki_kaiseki
Please input password:
ctf4b{we1cme_4o_rev4r3ing_2o22}
Please input PIN:
```

今度はPINの入力を求められる

それらしい数字はstringsの結果から見つからないので，静的解析をしてみる

4-2. 静的解析ツール

Ghidraの使い方を紹介

0. Ghidraを起動する
1. 起動後File→New Projectを選択
2. Non-shared projectを選択
3. 適当なProject　DirectoryとProject Nameを入力
4. Tool Chestの緑のアイコンをクリック
5. File→Import Fileから配布バイナリを選択
6. 「analyze now?」にYes
7. Analysis OptionはデフォルトでOK
8. 左の真ん中のSymbol TreeのFunctionsを展開しmainを選択
9. 真ん中に逆アセンブル結果，右に逆コンパイル結果が表示される

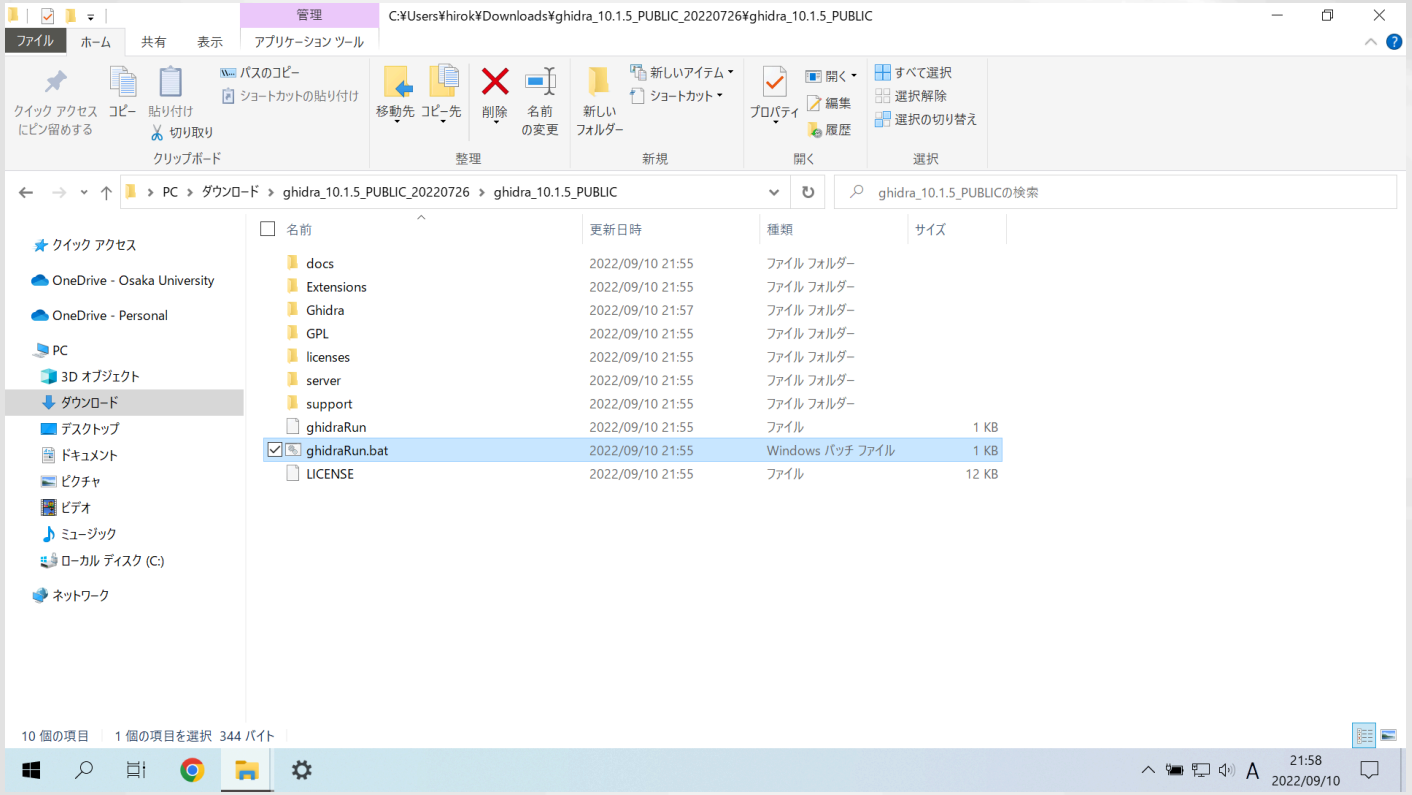
0. Ghidraを起動する - Mac

ターミナルでGhidraのzipファイルを解凍したディレクトリに移動し`./ghidraRun`で実行

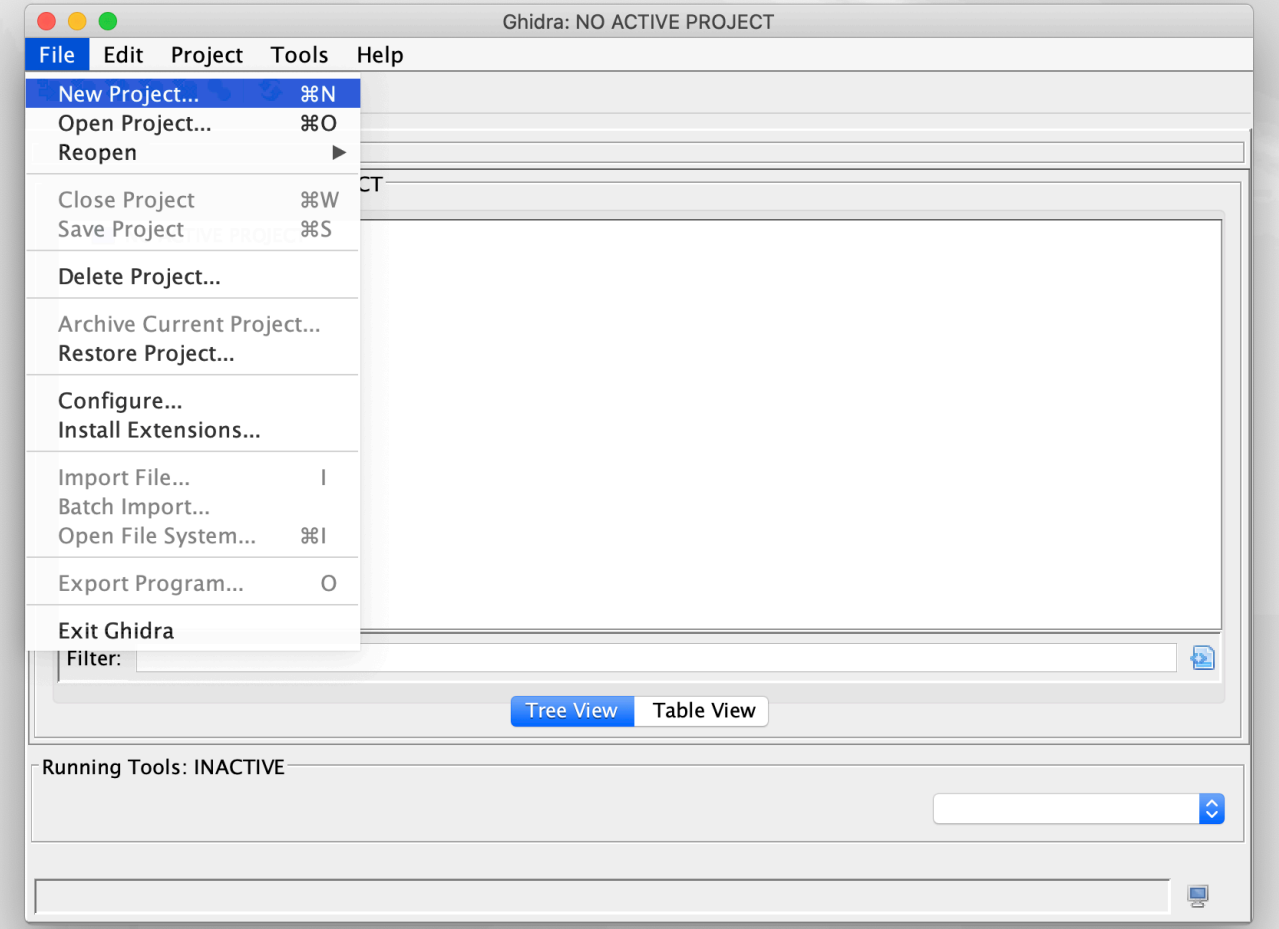
```
$ cd ~/Documents/
$ cd ghidra_10.1.5_PUBLIC/
$ ./ghidraRun
```

0. Ghidraを起動する - Windows

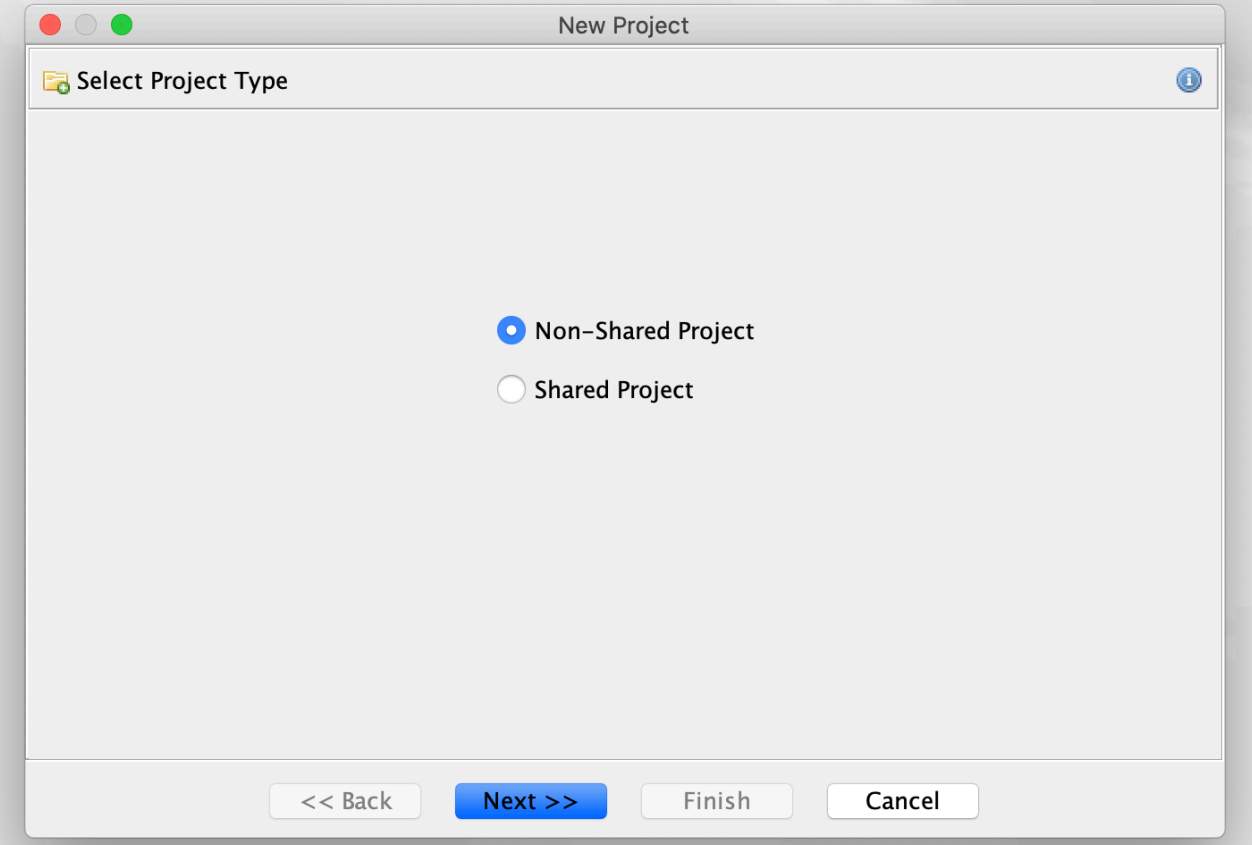
GhidraのzipファイルをWindowsで展開したフォルダの中の「ghidrarun.bat」をクリックしGhidraを起動します



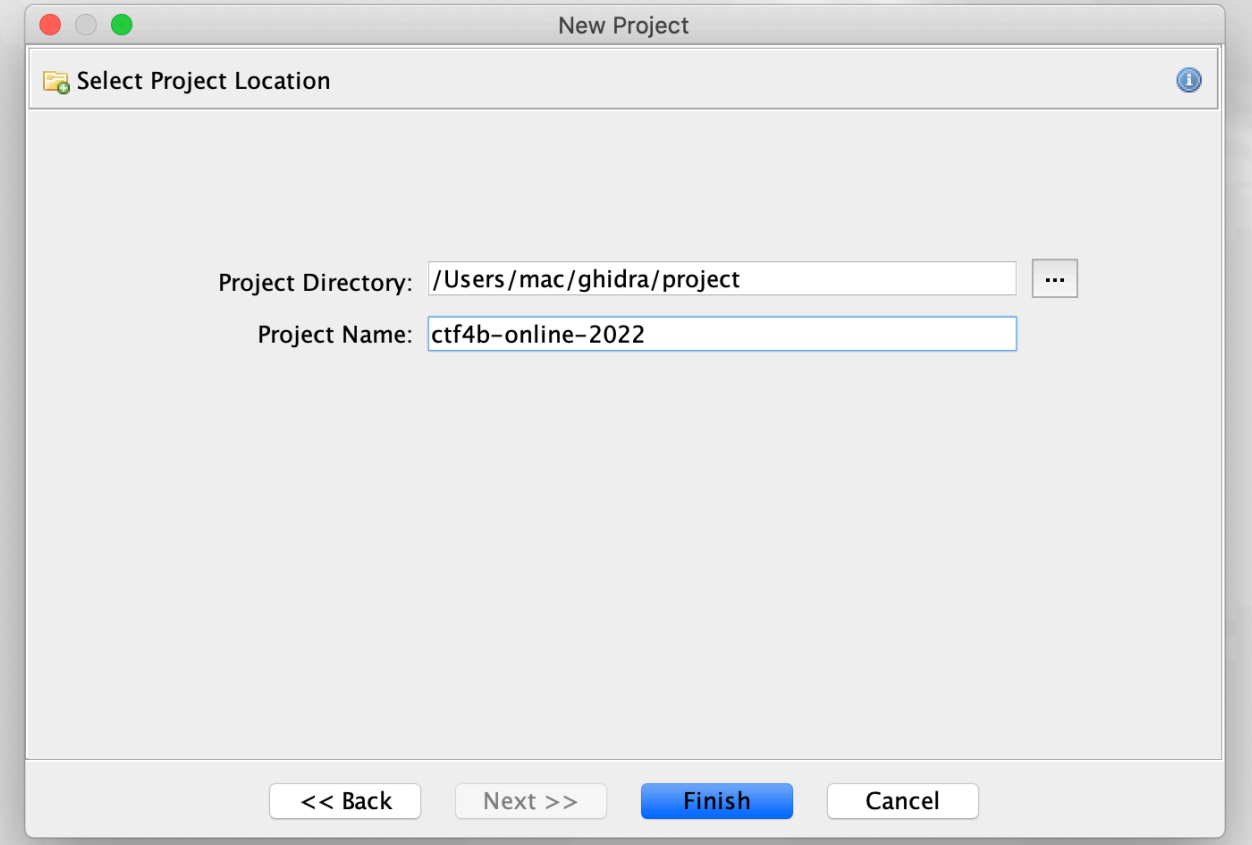
1. 起動後左上のFile→New Projectを選択



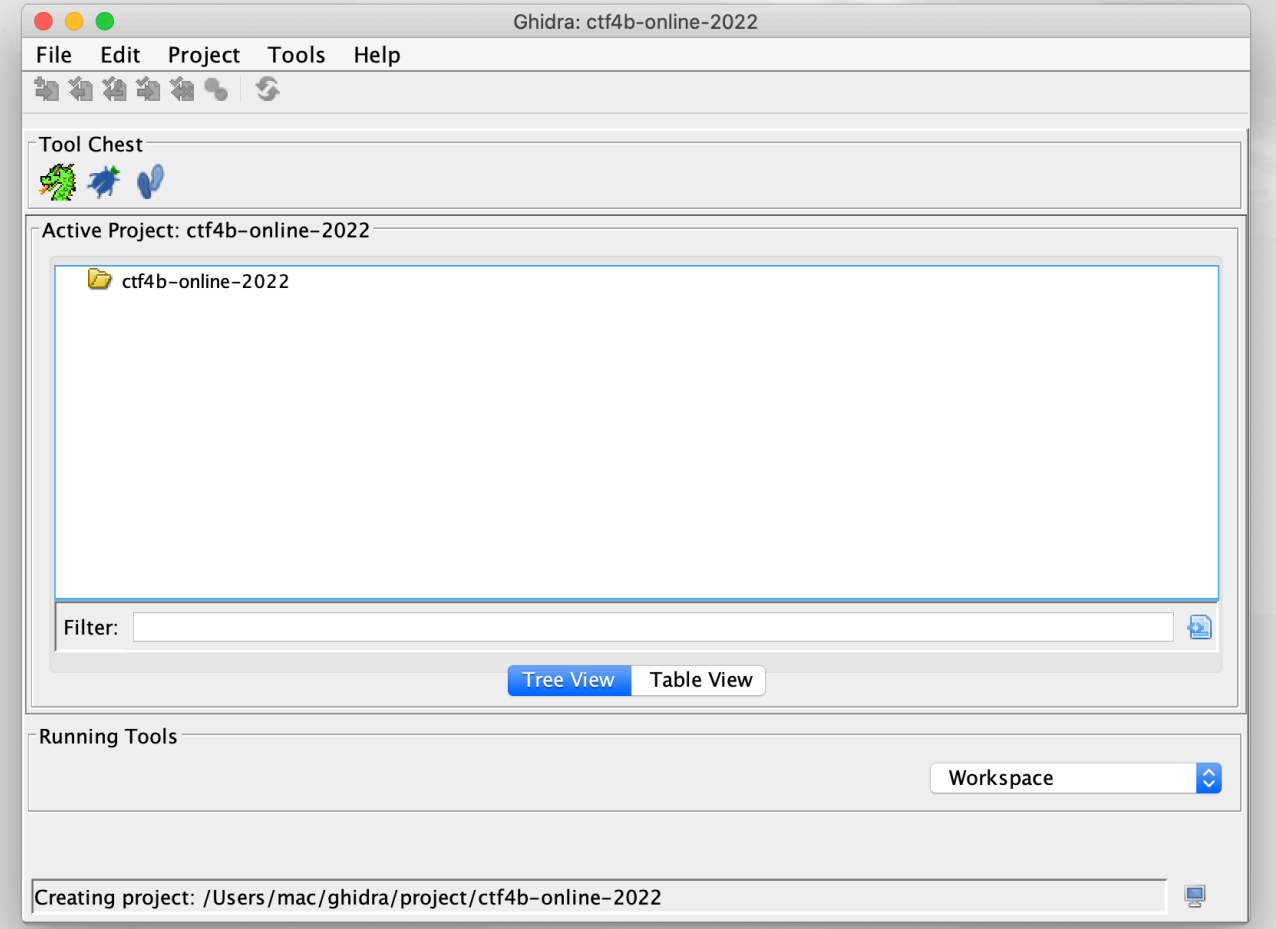
2. Non-shared projectを選択



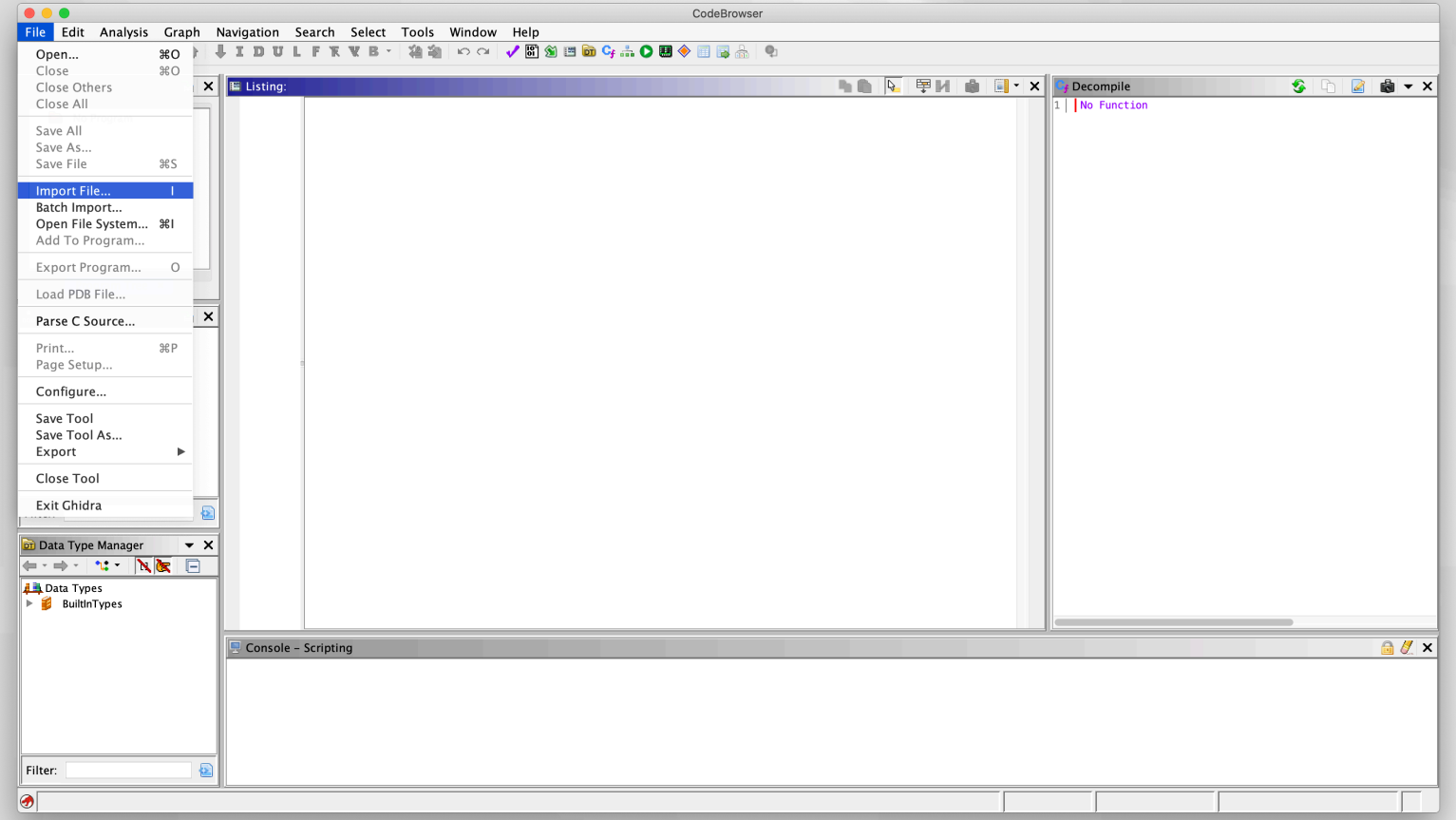
3. 適当なProject　DirectoryとProject Nameを入力



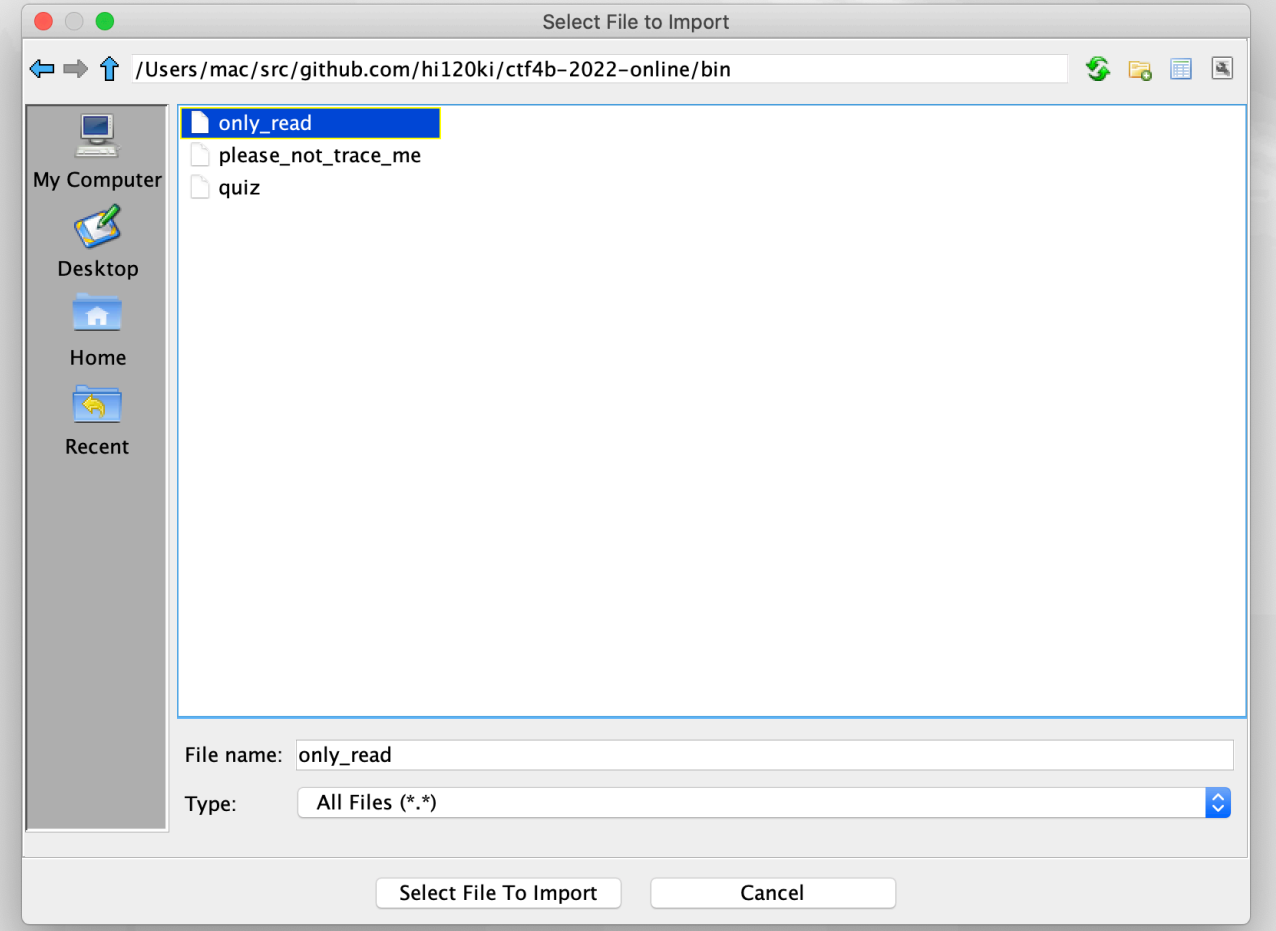
4. Tool Chestの緑のドラゴンのようなアイコンをクリック



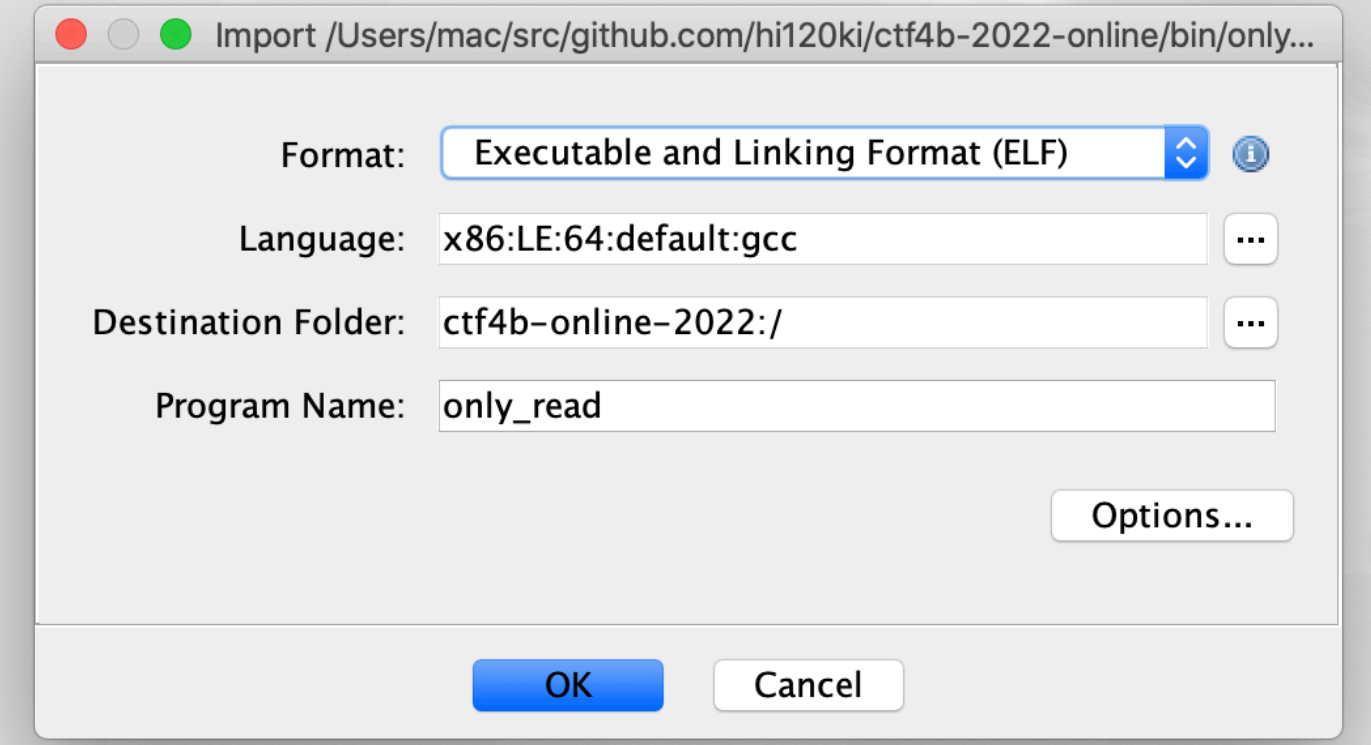
5. 左上のFile→Import Fileから配布バイナリを選択



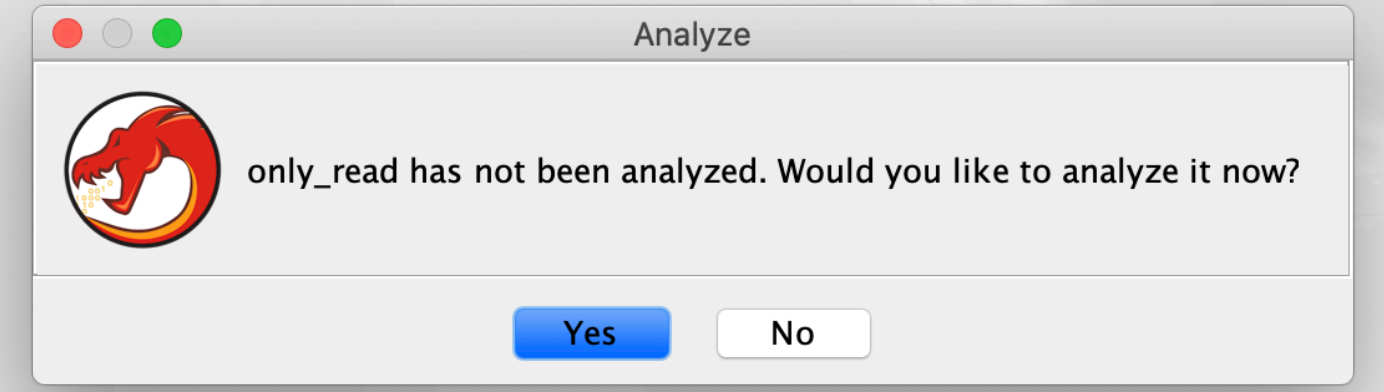
5. File→Import Fileから配布バイナリを選択



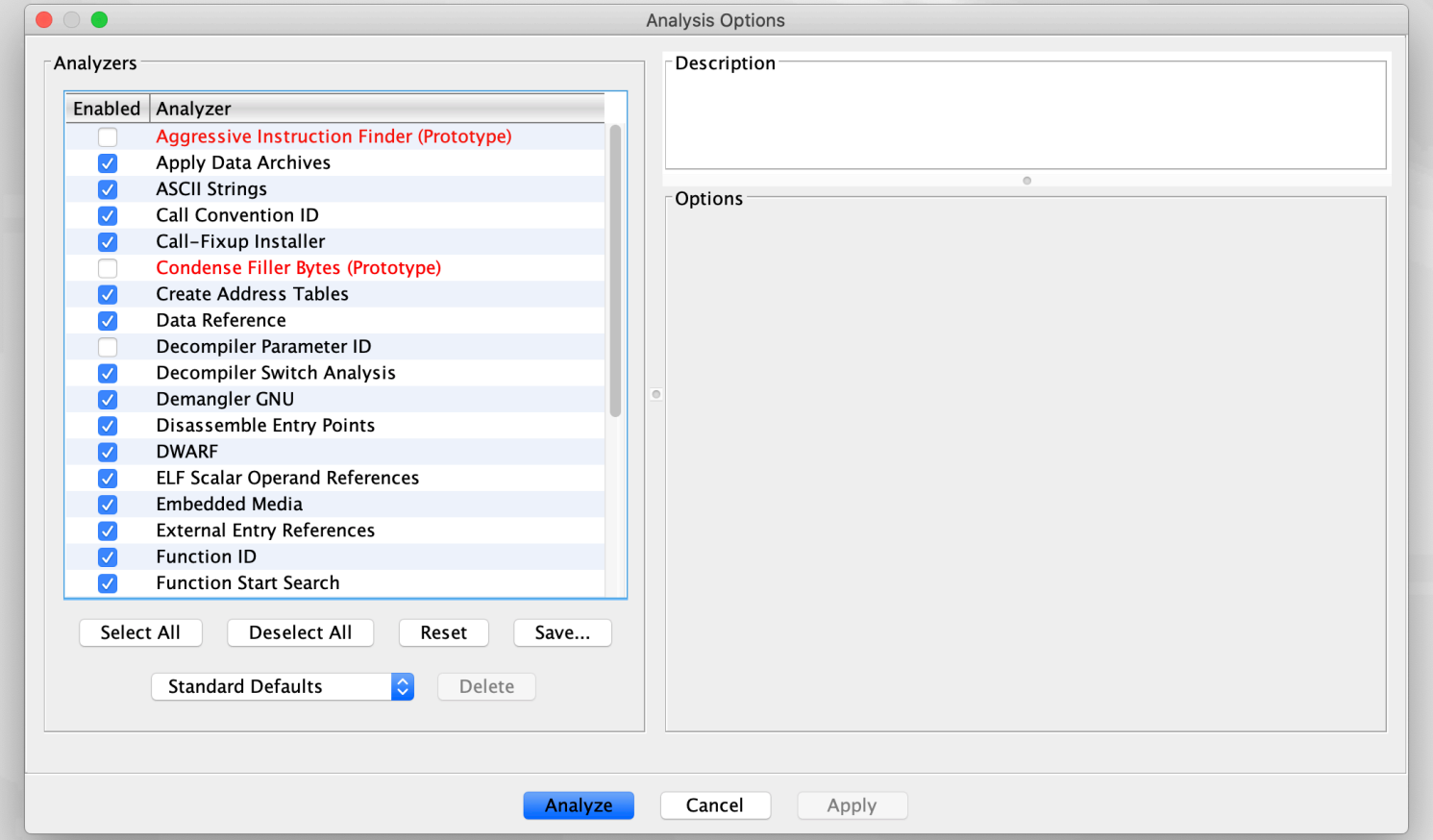
5. File→Import Fileから配布バイナリを選択



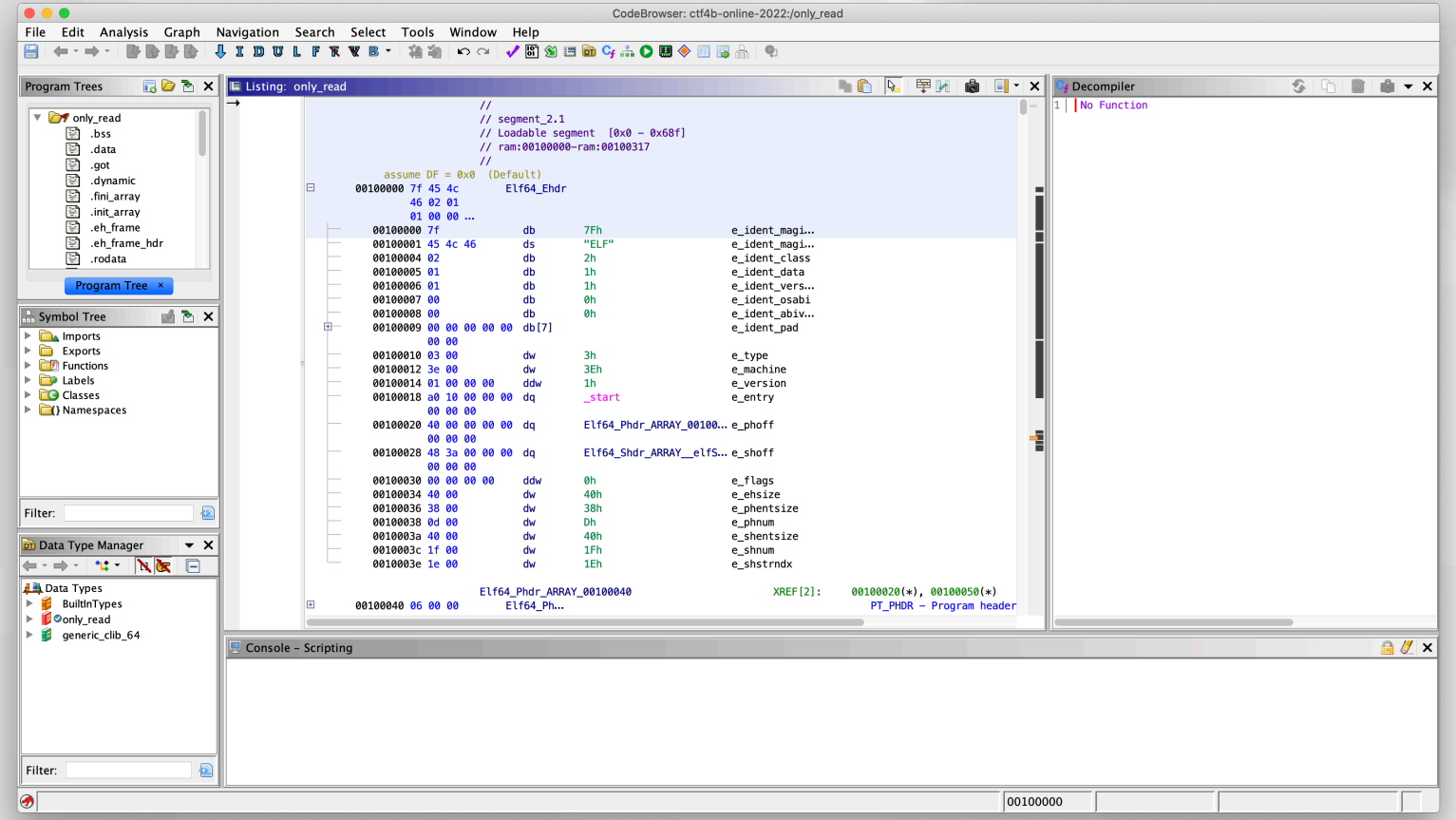
6. 「analyze now?」にYes



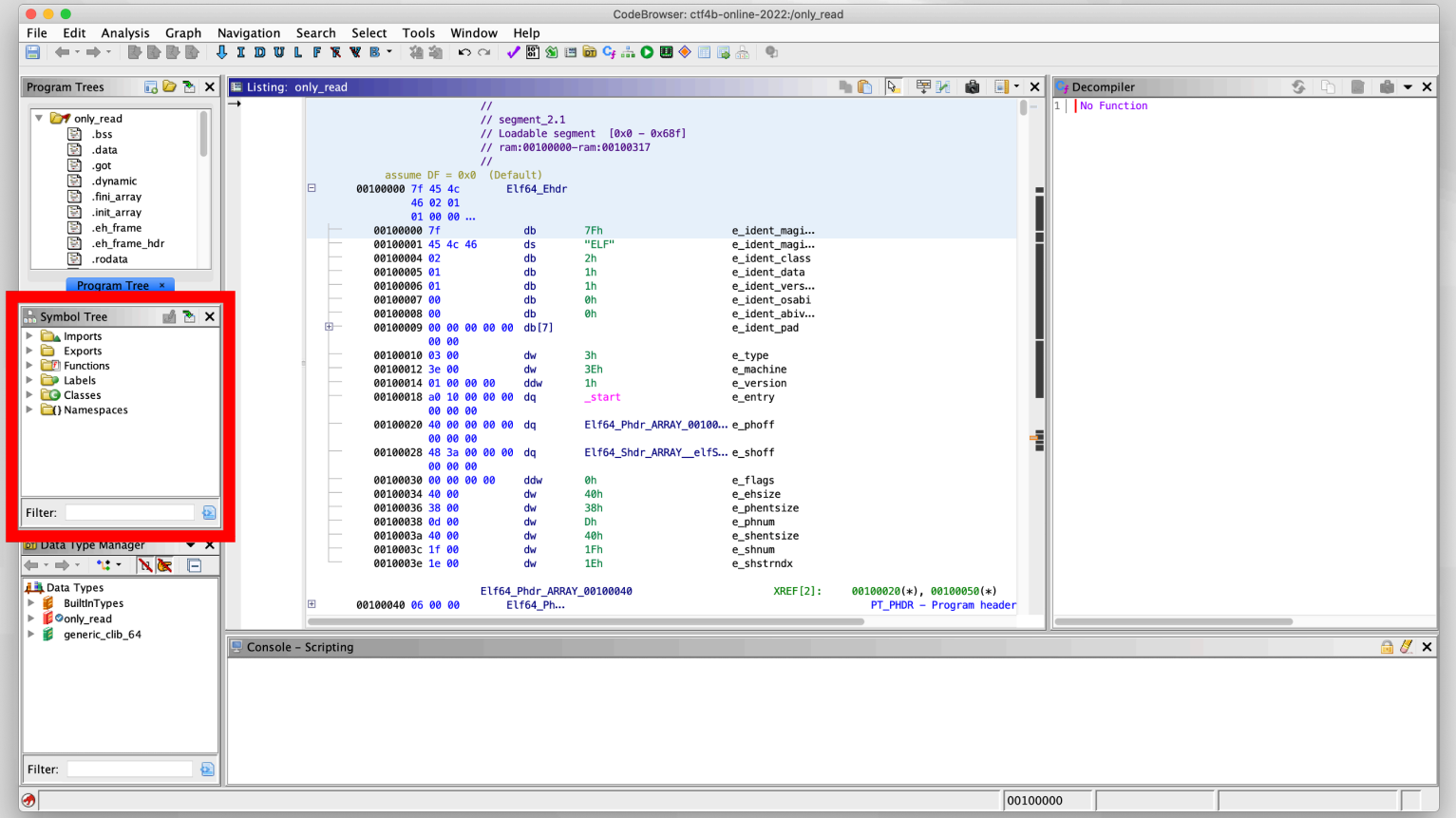
7. Analysis Optionはデフォルトで OK



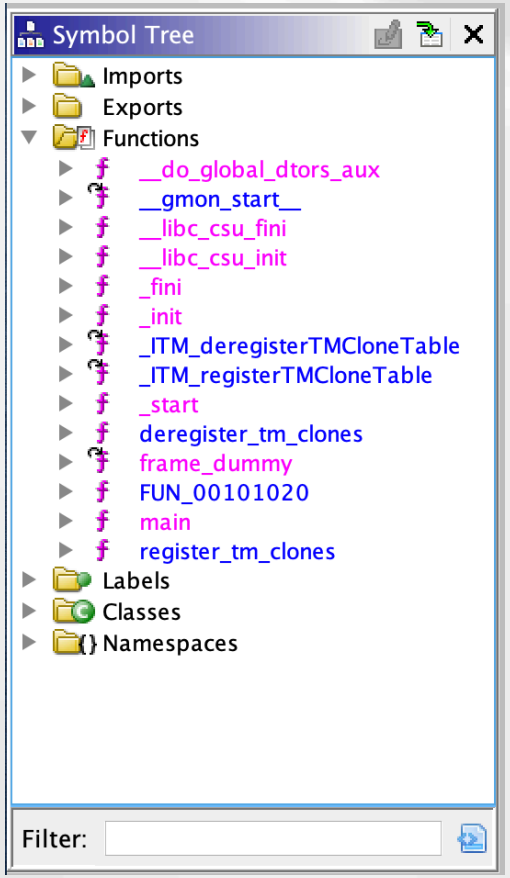
8. 左の真ん中のSymbol TreeのFunctionsを展開しmainを選択



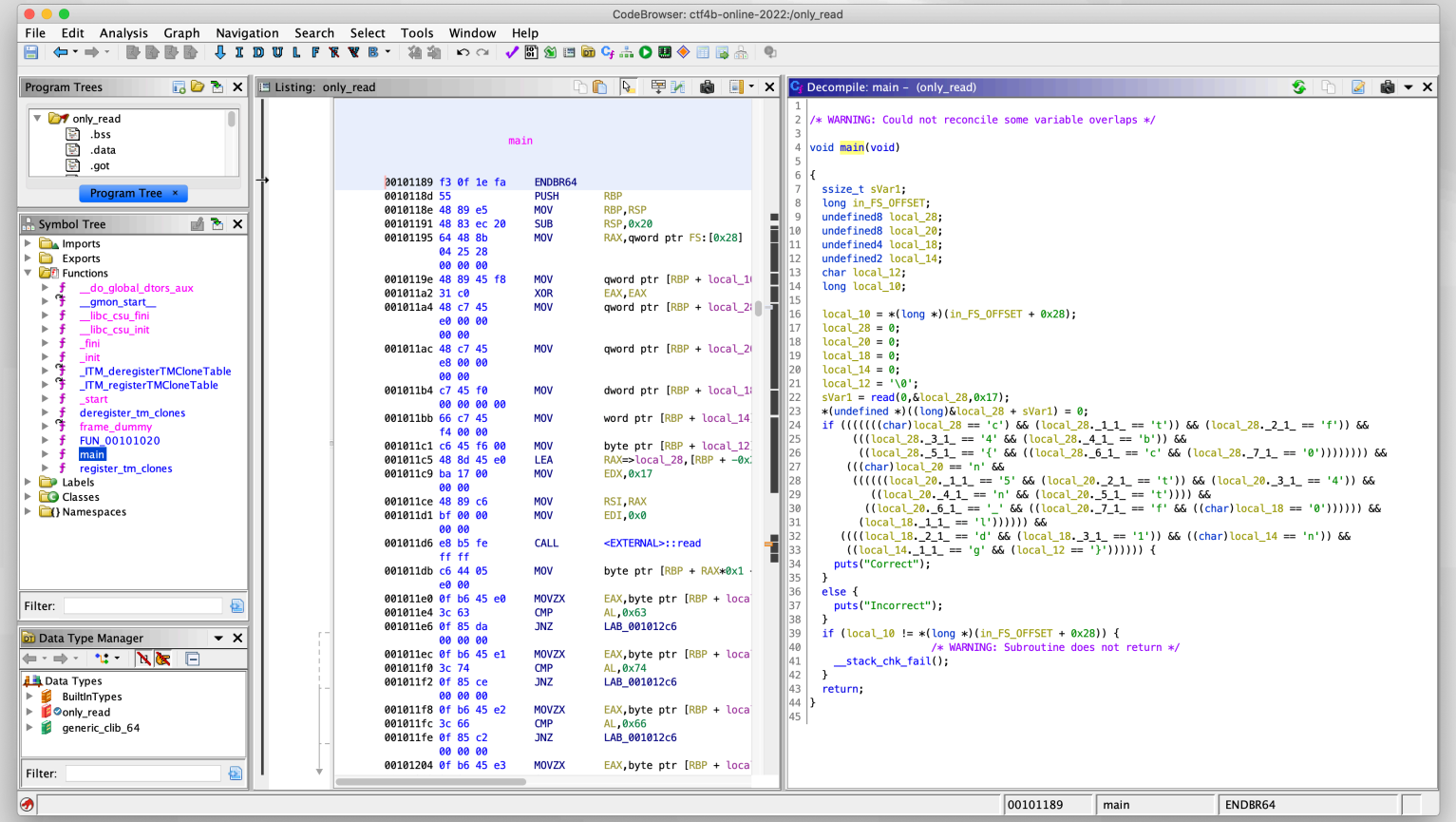
8. 左の真ん中のSymbol TreeのFunctionsを展開しmainを選択



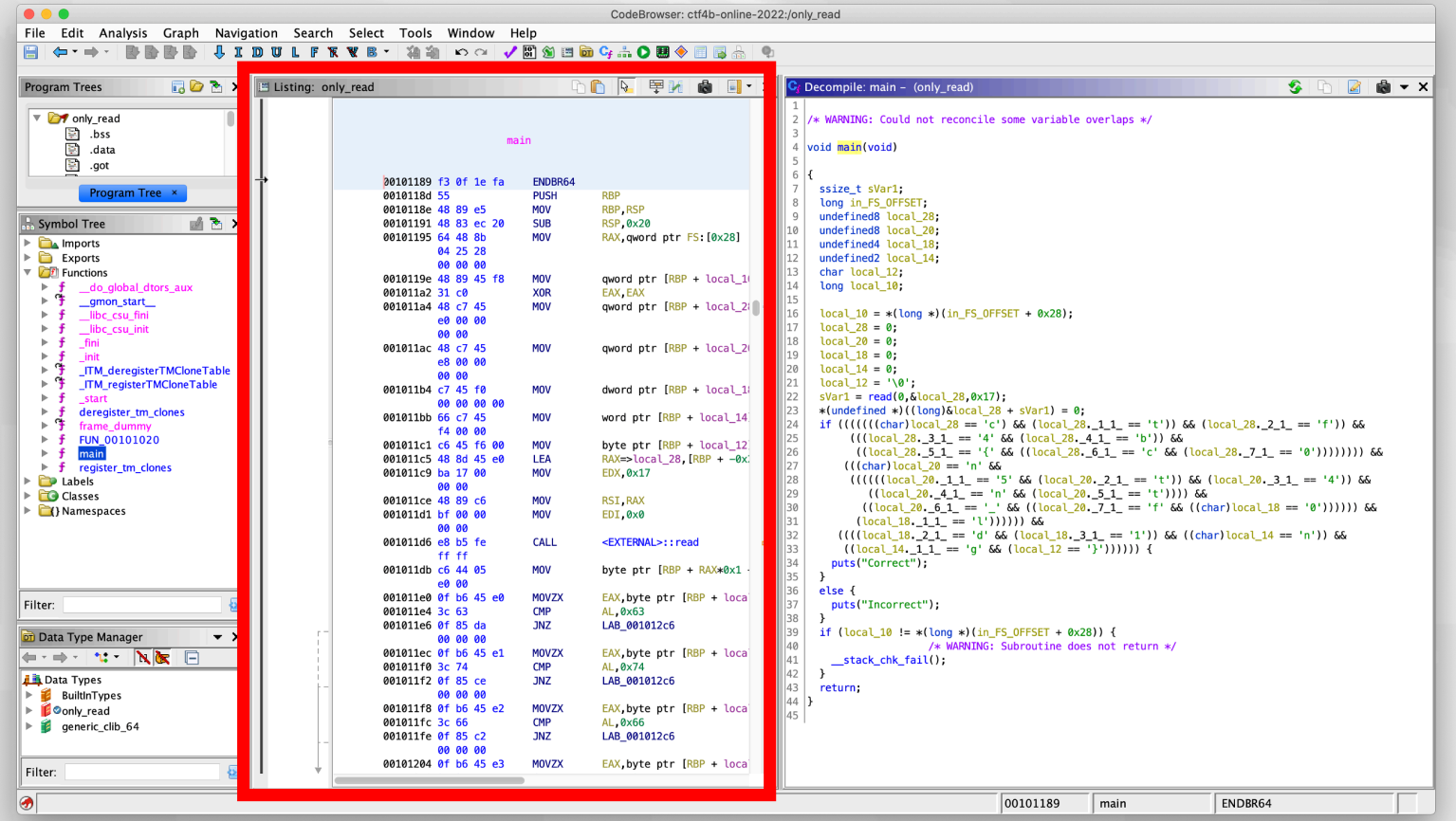
8. 左の真ん中のSymbol TreeのFunctionsを展開しmainを選択



9. 真ん中に逆アセンブル結果，右に逆コンパイル結果が表示される



(解法1) 逆アセンブル結果を見る



(解法1) 逆アセンブル結果を見る

```
アドレス    バイナリ          命令        オペランド
001012bd    be 50 00          MOV         ESI,0x50
            00 00
```

アドレス：メモリの中で命令が格納される番地．メモリの中に格納された命令は基本的に順番に実行されていくがJMP系の命令などで任意のアドレスの命令を実行していくようにすることもできる

バイナリ：いわゆる機械語(アセンブリコード)．ELFファイルの中身で命令とオペランドに対応するものが表示されている

命令: メモリのアドレスや中身・レジスタ・定数を引数(オペランドと呼ぶ)として行われる処理の名称．MOV命令は第2オペランドの値を第1オペランドにコピーする

レジスタ：CPUと高速にデータのやり取りができる記憶素子．EDX, RSI, RAXはレジスタの1種

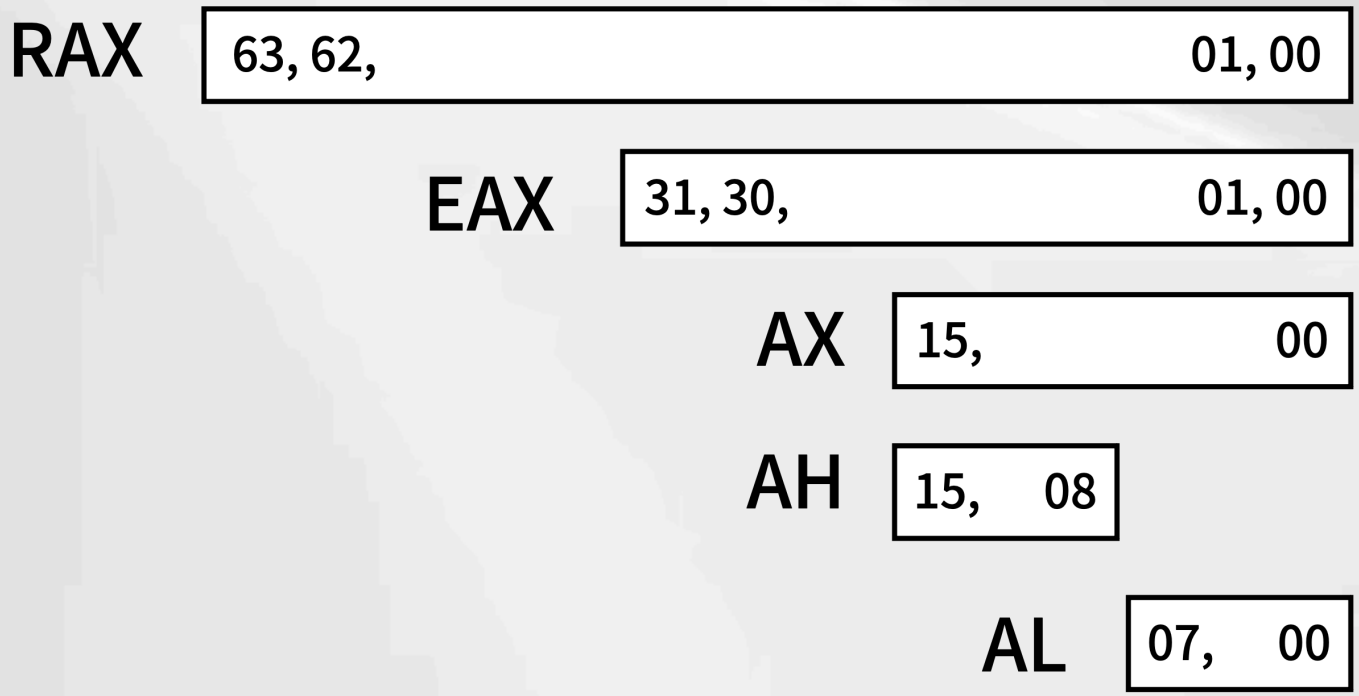
レジスタの種類

汎用レジスタ	RAX，RCX，RDX，RBX
インデックスレジスタ	RSI, RDI
特殊レジスタ	RBP, RSP, RIP
拡張汎用レジスタ	R8~R15

これ以外にもセグメントレジスタ・フラグレジスタなどがある

関数を呼び出す際，UNIXではRDI, RSI, RDX, RCX, R10, R8, R9の順番で第一引数から引数を格納し，返り値にはRAXが使用される

汎用レジスタのサイズごとの呼び方



汎用レジスタはアクセスしたい領域のサイズごとに呼び方が変わり，64bitすべて使う場合はRAX，
下位32bitの場合はEAX，16bitではAX，AXのうち，上位8bitはAH・下位8bitをALとして扱う

```
001012f0 48 8b 15        MOV        RDX,qword ptr [stdin]
         29 2d 00 00
001012f7 48 8d 45 a0     LEA        RAX=>local_68,[RBP + -0x60]
001012fb be 10 00        MOV        ESI,0x10
         00 00
00101300 48 89 c7        MOV        RDI,RAX
00101303 e8 b8 fd        CALL       <EXTERNAL>::fgets
         ff ff
```

(LEA命令は第2オペランドのアドレスを計算し，第1オペランドにコピーする．ここではRBP-0x60)

このとき

- 第1引数 RDIには書き込み先メモリ領域のアドレス
- 第2引数 ESIには書き込む文字数0x10
- 第3引数 RDXには読み込むもととなるstdin(標準入力)

を指定し，関数を呼び出すCALL命令で `fgets` を指定することで，標準入力から0x10(=16)文字読み込みメモリ内に保存する処理が実行される

```
00101308 48 8d 45 a0     LEA        RAX=>local_68,[RBP + -0x60]
0010130c 48 89 c7        MOV        RDI,RAX
0010130f e8 b5 fe        CALL       check_pin
         ff ff
```

その後，先程標準入力からの文字列を格納したメモリのアドレスを第一引数(RDI)として関数`check_pin`を呼び出している

※再確認

“ UNIXではRDI, RSI, RDX, RCX, R10, R8, R9の順番で第一引数から引数を格納する ”

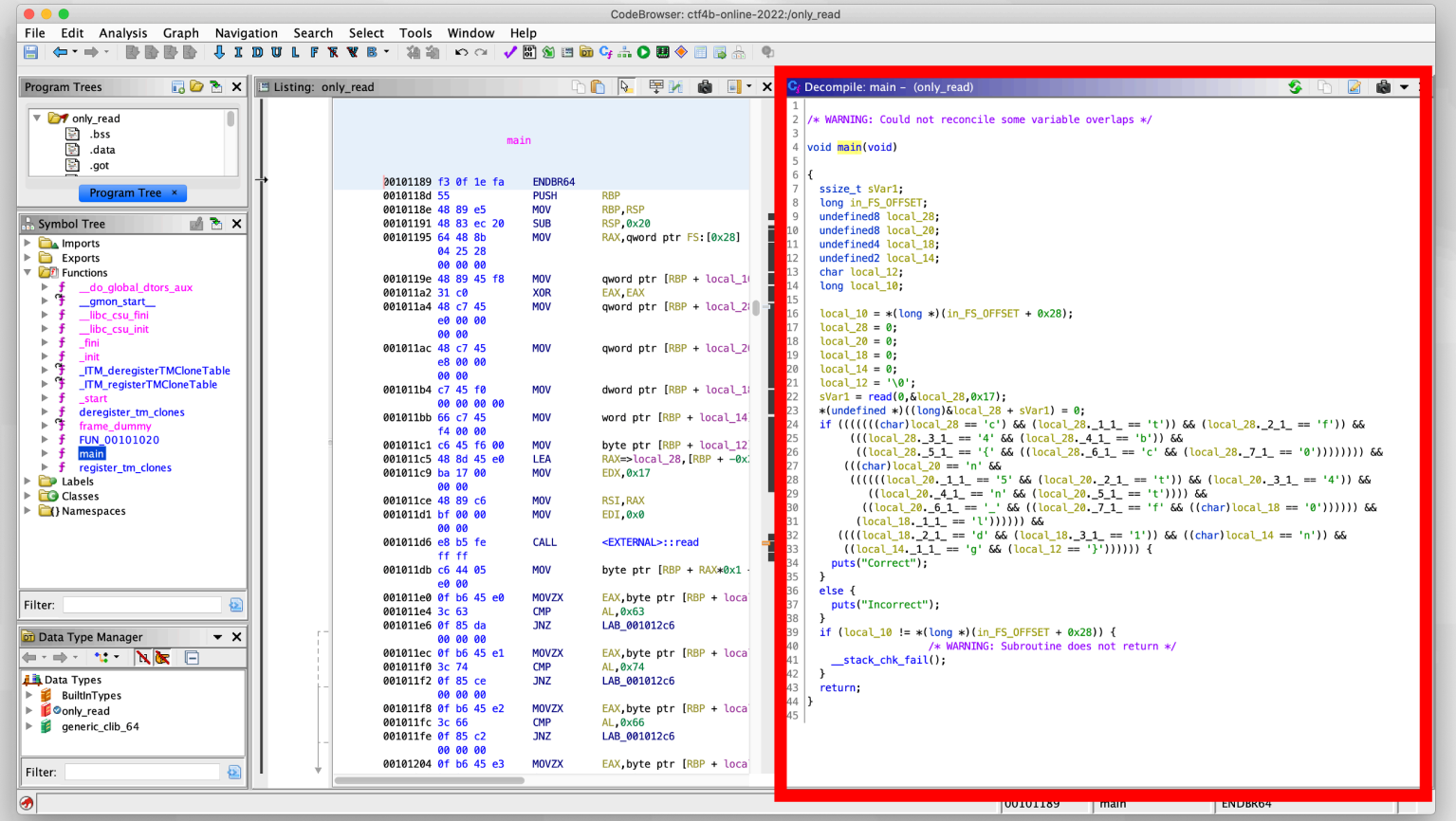
```
001011f2 48 8b 45 f8      MOV       RAX,qword ptr [RBP + local_10]
001011f6 0f b6 00         MOVZX     EAX,byte ptr [RAX]
001011f9 3c 31            CMP       AL,0x31
001011fb 75 34            JNZ       LAB_00101231
...
```

そしてメモリ内に保存された入力文字列を1byteずつEAXにコピーして`0x31`や`0x32`と比較している

`0x31` `0x32`は文字コードの標準的な規格であるアスキー文字として`1` `2`と対応する

→ CMP命令で比較している数値を順番にアスキー文字として変換するとPIN`1230`が得られる.

(解法2) Ghidraの右のDecompile画面を見る



(解法2) Ghidraの右のDecompile画面を見る

```
undefined8 main(void)

{
  ...
  puts("Please input password:");
  fgets((char *)&local_68,0x50,stdin);
  iVar2 = strcmp((char *)&local_68,PASSWORD);
  if (iVar2 == 0) {
    puts("Please input PIN:");
    fgets((char *)&local_68,0x10,stdin);
    cVar1 = check_pin(&local_68);
    if (cVar1 == '\0') {
      puts("Incorrect Bye...");
    }
    ...
```

PINを検査している`check_pin`関数をダブルクリックする

(解法2) Ghidraの右のDecompile画面を見る

```
undefined4 check_pin(char *param_1)

{
  undefined4 uVar1;
  size_t sVar2;

  sVar2 = strlen(param_1);
  if (sVar2 == 5) {
    if ((((*param_1 == '1') && (param_1[1] == '2')) && (param_1[2] == '3')) && (param_1[3] == '0'))
    {
      uVar1 = 1;
    }
    else {
      uVar1 = 0;
    }
  }
  else {
    uVar1 = 0;
  }
  return uVar1;
}
```


文字列の長さが5(4桁の文字列と改行 `\n`)であることを判定し，さらに1文字ずつ `1` `2` `3` `0` と一致し

4-3. 動的解析ツール

GDBの使い方を紹介

0. 表層解析・静的解析
1. バイナリの読み込み
2. 解析の始め方
3. Break Pointによる一時停止
4. レジスタの書き換え

0. 表層解析・静的解析

まずは表層解析・静的解析で情報を集める

```
$ file douteki_kaiseki
douteki_kaiseki: ELF 64-bit LSB shared object,
x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=283ffaa3fb208372c6b8737c8e8c4947d901c440,
for GNU/Linux 3.2.0, not stripped
```

```
$ strings douteki_kaiseki
/lib64/ld-linux-x86-64.so.2
...
main
print_flag
...
```

0. 表層解析・静的解析

Ghidraでmain関数を見てみると

```
undefined8 main(void)

{
  int iVar1;
  iVar1 = super_secure_function();
  if (iVar1 == 0x2022) {
    puts("Sorry...");
  }
  else {
    print_flag();
  }
  return 0;
}
```

super_secure_function関数の返り値が0x2022以外の時flagが表示される

super_secure_function関数をダブルクリックして処理を見てみると必ず0x2022を返すようになっている

```
undefined8 super_secure_function(void)

{
  return 0x2022;
}
```

→ super_secure_function関数の返り値を0x2022以外に書き換え，print_flag関数に到達できるようにする

2. 解析の始め方

douteki_kaisekiをLinux x64環境上のGDBで読み込む

```
$ wget https://ctf4b.epitomize.dev/bin/douteki_kaiseki
$ chmod +x douteki_kaiseki
$ gdb douteki_kaiseki
...
gdb-peda$ start # main関数で一時停止
...
gdb-peda$ c # 実行を続ける
Continuing.
Sorry...
[Inferior 1 (process 110354) exited normally]
```

GDBでの代表的なコマンド

コマンド名(エイリアス)	動作
run, r	プログラムの実行開始
start	プログラムを実行開始し，main関数で停止
continue, c	実行を再開
step, s	1命令ずつ実行(ステップイン：関数内に入る)
next, n	1命令ずつ実行(ステップアウト：関数内に入らない)
quit, q	終了する
disassemble <関数名>, disas	関数のディスアセンブル結果の表示
break <*0x00>, b	指定したアドレス，関数にブレークポイントを追加
delete <*0x00>, d	指定したアドレスのブレークポイントの削除
delete 1	指定した番号(一覧で確認できる)のブレークポイントの削除
info b, i b	ブレークポイントの一覧
info functions, i func	バイナリ中で定義されている関数の一覧
set $<レジスタ>=<値>	指定したレジスタの値を上書きする

3. Break Pointによる一時停止

デバッガにプログラムのアドレスを設定しておくことで，そのアドレスの命令を実行する直前でプログラムを一時停止する機能

さらに停止中はレジスタやメモリの値を書き換えることができる

→ super_secure_function関数の実行が終了したところで実行を停止させ，返り値を0x2022以外に書き換え，print_flag関数に到達できるようにする

まずはGDB上でプログラム内のsuper_secure_function関数がどのアドレスで呼び出されているか調査する

```
$ gdb douteki_kaiseki
...
gdb-peda$ start # main関数で勝手に一時停止してくれる
...
gdb-peda$ disas main # main関数を逆アセンブルする
Dump of assembler code for function main:
=> 0x000055555555221 <+0>:     endbr64
   0x000055555555225 <+4>:     push   rbp
   0x000055555555226 <+5>:     mov    rbp,rsp
   0x000055555555229 <+8>:     mov    eax,0x0
   0x00055555555522e <+13>:    call   0x555555555212 <super_secure_function>
   0x000055555555233 <+18>:    cmp    eax,0x2022
   0x000055555555238 <+23>:    je     0x555555555246 <main+37>
   0x00055555555523a <+25>:    mov    eax,0x0
   0x00055555555523f <+30>:    call   0x555555555189 <print_flag>
```

`main+13`でsuper_secure_function関数が呼び出され、その直後に返り値が格納されているEAXレ

4. レジスタの書き換え

```
gdb-peda$ b *main+18 # break pointを設定
Breakpoint 2 at 0x555555555233
gdb-peda$ c # 実行を再開する
Continuing.
...
Breakpoint 2, 0x0000555555555233 in main ()
gdb-peda$ set $rax=0 # RAXレジスタの値を0にする
gdb-peda$ c # 実行を再開する
Continuing.
ctf4b{r3g1st3r_3d1t1ng}
```

→ 通常では到達できないprint_flag関数を実行できフラグ `ctf4b{r3g1st3r_3d1t1ng}` を取得できた

5. 実行可能ファイルの解析が役に立つ場面の紹介

デバッガーは名前の通り，プログラムのデバッグに使える事があります

また，セキュリティエンジニアのうち，マルウェア解析等をする場合は今回紹介したツールなどを使う事があります

プログラムの動作の仕組みや中身を知ることで，普段ソフトウェアを触ったり，プログラムを書く際の解像度が上がります

最後に

表層解析ではfileコマンド・stringsコマンド，静的解析ではGhidra，動的解析ではGDBをインストール方法から基本的な使い方までを紹介しました

SECCON Beginners CTFの過去問は https://github.com/seccon から入手できます

また，他にもネット上では常設CTFや書籍，様々なCTFの過去問，解答付きの問題を入手することができます

ぜひいろんな問題を今回紹介したツールたちで解いてみてください

appendix

資料本編では紹介を省いてきた部分の説明などを行います

手元で解析をする際の参考にお使いください

ASCII

ASCIIは文字を数値で表現する方法の1つで，幅広く使われている

例えば `'A'` は `0x41`，`'a'` は `0x61`，`'0'` は `0x30` など

Pythonで `ord`，`chr` を用いることで相互変換できる

```
>>> ord('A')
65
>>> chr(65)
'A'
>>> hex(ord('a'))
'0x61'
```

ASCIIの一覧

`ascii`コマンドで確認できる

```
$ ascii -x
   00 NUL    10 DLE    20       30 0    40 @    50 P    60 `    70 p
   01 SOH    11 DC1    21 !     31 1    41 A    51 Q    61 a    71 q
   02 STX    12 DC2    22 "     32 2    42 B    52 R    62 b    72 r
   03 ETX    13 DC3    23 #     33 3    43 C    53 S    63 c    73 s
   04 EOT    14 DC4    24 $     34 4    44 D    54 T    64 d    74 t
   05 ENQ    15 NAK    25 %     35 5    45 E    55 U    65 e    75 u
   06 ACK    16 SYN    26 &     36 6    46 F    56 V    66 f    76 v
   07 BEL    17 ETB    27 '     37 7    47 G    57 W    67 g    77 w
   08 BS     18 CAN    28 (     38 8    48 H    58 X    68 h    78 x
   09 HT     19 EM     29 )     39 9    49 I    59 Y    69 i    79 y
   0A LF     1A SUB    2A *     3A :    4A J    5A Z    6A j    7A z
   0B VT     1B ESC    2B +     3B ;    4B K    5B [    6B k    7B {
   0C FF     1C FS     2C ,     3C <    4C L    5C \    6C l    7C |
   0D CR     1D GS     2D -     3D =    4D M    5D ]    6D m    7D }
   0E SO     1E RS     2E .     3E >    4E N    5E ^    6E n    7E ~
   0F SI     1F US     2F /     3F ?    4F O    5F _    6F o    7F DEL
```

x86_64アセンブリ

x86_64アセンブリの各命令について，代表的なものを取り上げながら，それにまつわる用語の解説なども行う

参考になるサイト(一部)

- https://hikalium.github.io/opv86/
- https://www.mztn.org/lxasm64/amd00.html
- https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

シンタックスについて

x86_64アセンブリの書き方には，AT&T記法とIntel記法があるが，この資料ではIntel記法で記載する

- AT&T記法
  - `movl $1, %eax`など，レジスタのprefixに`%`を，即値のprefixに`$`を付ける
  - オペランドの順番は`src, dst`
- Intel記法
  - `mov eax, 1`など，レジスタ，即値にprefixがつかない
  - オペランドの順番は`dst, src`

用語について

- オペランド
  - 演算の対象となる値
  - 要は引数
- 即値
  - オペランドとして指定される値
  - 要は定数
- src
  - `source`
  - 演算のソースとなるオペランド
- dst
  - `destination`
  - 演算の対象となるオペランド

即値の表記について

`0x`から始まる数値は16進数

`0b`から始まる数値は2進数

その他，特に指定の無い場合は10進数

サイズについて

演算の対象となるデータのサイズごとに，`QWORD`などの指定がある

- `BYTE`
  - 1バイト(8Bit)
- `WORD`
  - 2バイト(16Bit)
- `DWORD`
  - 4バイト(32Bit)
- `QWORD`
  - 8バイト(64Bit)

また，オペランドをアドレスとして使う際は`PTR`を付ける

これらは，メモリの中身を読む時等によく使われる

特所なレジスタについて

本編でも少し触れた特殊なレジスタの一部を説明する

RIP

次に実行するる命令が格納されているアドレス

フラグレジスタ

一部の命令では，演算の結果により値が変わり，bit毎に役割がある

例

- `OF`
  - オーバーフローフラグ
  - 算術演算の結果がオーバーフローした時にセットされる
- `SF`
  - 符号(Signed)フラグ
  - 演算結果の最上位bitと同じ値になる．2の補数での正負を表す
- `ZF`
  - ゼロフラグ
  - 演算結果が0だっと時にセットされる
- `CF`
  - キャリーフラグ
  - シフトや算術演算などで最上位または最下位のbitからあふれた場合にセットされる

呼び出し規約について

改めて，関数の引数を指定するとき，x86_64ではレジスタを使う

第1引数	第2引数	第3引数	第4引数	第5引数	第6引数	第7引数以降
rdi	rsi	rdx	rcx	r8	r9	スタック

スタックについて

x86_64では，メモリ上の特定の領域をスタックと呼ばれるデータ構造として扱う

スタックとのデータのやりとりは主に`push`と`pop`で行う

スタックはLIFO(Last In First Out)のデータ構造で，最後に`push`した値が`pop`で出てくる

わかりづらい点として，スタックはメモリのアドレス上で小さい方に向かって伸びていく

また，スタックの単位は8byteなので`rsp`は8ずつ増減する

※
スタックの単位は8byteだが，実際に使う際は8byteごととは限らない
例えば，サイズが4byteの変数を2つ，スタックの同じ段に入れることもある

スタック操作系命令

- push
- pop

push

`rsp`を -8 (スタックはアドレスの小さい方に伸びていくので)し，`rsp`のアドレスにオペランドの値を書き込む

- オペランドに指定できる値
  - 即値
  - レジスタ
  - メモリ

例

```
push rbp
```

pop

`rsp`のアドレスから値を取り出し，オペランドに書き込み，`rsp`を +8する

- オペランドに指定できる値
  - 即値
  - レジスタ
  - メモリ

例

```
pop rdi
```

スタックフレーム

スタックは主に，関数ごとにスタックフレームという単位で扱われる

スタックフレームを表現するのには，主に`rbp`，`rsp`のレジスタが以下の目的で使われる

- rbp
  - 現在のスタックの底(一番下，最後に出てくる値)のアドレス
- rsp
  - 現在のスタックの一番上(次のpopで出てくる値)のアドレス

関数毎にスタックフレームを分ける事で，ローカル変数の概念を実現している

基本的に，ローカル変数にアクセスする際は`rbp`からのインデックスが使われる(例: `lea rax, [rbp-0x20]`)

アドレスの小さい方に伸びていく為，`[rbp-0x20]`はスタック上からの4番目(0x20/8番目)の値

スタックフレームの確保

バイナリのdisassemble結果を見ると，関数の最初に

```
push rbp
mov rbp, rsp
sub rsp, 0x60
```

といった操作をしていることがある

これがスタックフレームの確保に必要な処理

1. `rbp`(前の関数のスタックフレームの底)をスタック上に`push`
2. `rbp`を`rsp`(前の関数のスタックフレームの一番上)で上書き
3. `rsp`を`0x60`小さくする．つまり`0x60/8 = 12`個の値を格納できるスタックを確保する

スタックフレームの破棄

関数の末尾では，`ret`の前に`leave`をしている

これは

```
mov rsp, rbp
pop rbp
```

と等価で，つまり確保した時と逆の事を行っている

`mov rsp, rbp`によって，今までスタックの底だったアドレスをスタックトップとしている．
すると，先程確保した時に`push`してあった前のスタックフレームの底(`rbp`に入っていた値)が今度はスタックトップに出てくるので，`pop rbp`で復元する

これらの操作によって，関数毎に分けられた領域をスタックフレームとして使う事ができる

関数関係の命令

- call
- jmp系
- cmp
- ret

call

`call`は関数(サブルーチン)の呼び出しに使う命令．

関数の処理が終わった際に戻ってくるべきアドレス(リターンアドレス)をスタックに`push`し，オペランドのアドレスに遷移する

リターンアドレスは`call`の次の命令のアドレス

疑似コードで書くと

```
push {{callの次の命令のアドレス}}
jmp function
```

ret

1. RSPが指すアドレスからRIPにデータを格納する
2. RSPを8増加させる

要は `pop rip`

先程の `leave` でスタックトップに復元したリターンアドレスをRIPにpopしてくる事で遷移できる

jmp系命令

オペランドのアドレスに遷移する

```
jmp function
mov rip, function ; 実際にはこんな命令は無いが，こういう事
```

条件付きjmp

`jne(Jump if Not Equal)`や，`jge(Jump if Greater or Equal)`等，条件(フラグレジスタの状態)によって遷移するかどうかが決まるもの

主にcmpの結果をもとに変わる．頻出は以下

- `jne`
  - 値が等しくなかったときに遷移する
- `je`
  - 値が等しかったときに遷移する
- `jle`
  - 値が小さいか等しいときに遷移する
- `jge`
  - 値が大きいか等しいときに遷移する
- `jl`
  - 値が小さいときに遷移する
- `jg`
  - 値が大きいときに遷移する

cmp

dstとsrcの値を比較する命令

上の条件付きjmp命令と一緒に使うことが多い

```
cmp al, 0x8
jne 0x400123
```

alレジスタの値が0x8と等しくない場合，cmpによりゼロフラグが0になり，jneによってアドレス0x400123の命令に遷移する

endbr64について

最近ROPという攻撃への対策として追加された命令

この命令が無い場所には遷移できなくする機能がある

が，基本的には意味が無いので，何もしない命令だと思って良い

データ転送系命令

- mov
- lea
- xchg

mov

名前こそ `MOVe` だが，実際の挙動はコピー

`src` オペランドの値を `dst` オペランドにコピーする

- srcに指定できる値
  - 即値
  - レジスタ
  - メモリ
- dstに指定できる値
  - レジスタ
  - メモリ

例

```
mov rax, 0 # raxに0をコピー
mov qword ptr [rbp-8], rax ; rbp - 8をアドレスとして解釈し，そこにraxの値をコピー
```

lea

(主に)アドレスを計算する命令

`src`オペランドのアドレスを計算し，`dst`オペランドに書き込む

あくまで計算するだけで，**そのアドレスを参照し，中身を取り出したりはしない．**

- srcに指定できる値
  - 即値
  - レジスタ
  - メモリ
- dstに指定できる値
  - レジスタ
  - メモリ

```
lea rax, [rbp-0x60] ; rbp-0x60を計算し，raxに書き込む
```

lea

アドレス以外の計算に使われる事もある

例

```
lea rax, [rax+0x10] ; rax+0x10を計算し，raxに書き込む
add rax, 0x10       ; 上のleaと等価
```

xchg

`src`と`dst`を交換する命令

- srcに指定できる値
  - レジスタ
  - メモリ
- dstに指定できる値
  - レジスタ
  - メモリ

例

```
xchg rax, rdi ; raxとrdiを交換する
```

上の例は関数を呼び出したあと，その返り値(rax)を第一引数(rdi)として関数を呼びたいときなどに使える

算術演算系命令

- add
  - inc
- sub
  - dec
- mul
- div

算術演算系命令のオペランドに指定できるのは，`src`，`dst`共に

- レジスタ
- メモリ

の2種

add

`src`の値を`dst`に足す

例

```
add rbx, 0x10 ; rbx += 16
add rcx, rbx ; rcx += rbx
```

inc

オペランドの値に1を足し，格納する(インクリメント)

例

```
inc rcx ; rcx++
```

sub

`src`の値を`dst`から引く

```
add rsp, 0x8 ; rsp -= 8
```

dec

オペランドの値から1を引き，格納する(デクリメント)

例

```
dec rcx ; rcx--
```

mul

オペランドの値を，サイズ毎にRAX・EAX・AX・ALレジスタに掛けて格納する(符号無し)

例

```
mul ebx ; eax *= rbx
mul rcx ; rax *= rcx
```

符号有りの場合は`imul`を使う

div

オペランドの値で，RAX・EAX・AX・ALレジスタを割り格納する(符号無し)

例

```
div ebx ; eax /= ebx
```

符号有りの場合は`idiv`を使う

シフト系命令

- shl, sal
- shr, sar

shl, sal

`dst`の値を`src`の値の分左シフトして格納する

足りなくなった部分には自動的に0が入る

例

```
mov al, 0b00011110
shl al, 2 ; rax <<= 2 (raxは0b01111000になる)
sal al, 4 ; rax <<= 3 (raxは0b11000000になる)
```

(shlとsalは同じ処理をする)

shr, sar

`dst`の値を`src`の値の分，右シフトして格納する

足りなくなった部分には自動的に0が入る

例

```
mov al, 0b11011000
sar al, 4 ; raxの符号は変えずに(下位63Bitのみ)右シフトする (alは0b10000101)
shr al, 4 ; al >>= 4 (alは0b00001000)
```

sarは`dst`の符号にあたる最上位ビットを保持したままシフトする

論理演算系命令

- and
- or
- xor
- not

and

`src`と`dst`のbit単位でのandを取り，`dst`に格納する

第1引数	第2引数	結果
0	0	0
0	1	0
1	0	0
1	1	1

例

```
mov al, 0b01001010 ; alに0b01001010を格納
and al, 0b01111100 ; al &= 0b01111100 結果，alは0b01001000となる
```

or

`dst`と`src`のorを取り，`dst`に格納する

第1引数	第2引数	結果
0	0	0
0	1	1
1	0	1
1	1	1

例

```
mov al, 0b01001010 ; alに0b01001010を格納
or al,  0b00011110 ; al &= 0b00011110 結果，alは0b01011110となる
```

xor

`dst`と`src`のxorを取り，`dst`に格納する

第1引数	第2引数	結果
0	0	0
0	1	1
1	0	1
1	1	0

例

```
mov al, 0b01001010 ; alに0b01001010を格納
xor al, 0b11110011 ; al ^= 0b11110011 結果，alは0b10110001となる
```

```
xor rax, rax ; raxにどんな値が入っていてもxorの結果は0になる(割とよくある)
```

not

`dst`のnot(bit反転)をとり，`dst`に格納する

第1引数	結果
0	1
1	0

例

```
mov al, 0b10101010 ; alに0b10101010を格納
not al ; al ~= al 結果，alは0b01010101となる
```

おまけ演習(1)

以下の関数をC言語で書き表してみましょう

```
push rbp
mov rbp, rsp
sub rsp, 0x10
mov dword [rbp - 0x8], 0x64
mov dword [rbp - 0x10], 0x100
mov edx, dword [rbp - 0x8]
mov eax, dword [rbp - 0x10]
add eax, ebx
leave
ret
```

回答例

```
int func() {
    int a = 0x64;
    int b = 0x100;
    return a + b;
}
```

おまけ演習(2)

以下の関数をC言語で書き表してみましょう

```
    push rbp
    mov rbp, rsp
    sub rsp, 0x10
    mov dword [rbp-0x8], 0x30
    jmp check
loop:
    mov eax, dword [rbp-0x8]
    mov edi, eax
    call putchar
    add dword [rbp-0x8], 1
check:
    cmp dword [rsp-0x8], 0x39
    jle loop
    leave
    ret
```

回答例

```
void func() {
    for (int c = 0x30; c <= 0x39; c++) {
        putchar(c);
    }
}
```