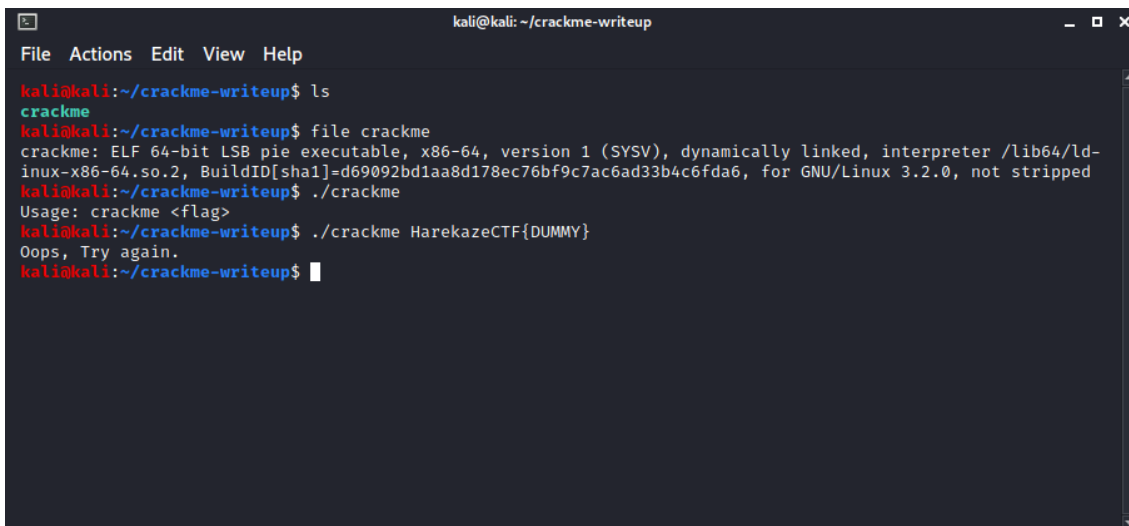


# [Rev] Crackme

---

## 表層解析

問題のプログラム「crackme」がなんのファイルか調査する。「file」コマンドを実行すると ELF のバイナリファイルであることがわかり、実行すると使用方法が表示される。この問題は「crackme <flag>」ということで、引数に Flag の文字列を渡せばよさそうである。試しに適当な flag「HarekazeCTF{DUMMY}」と入れると、「Oops, Try again.」と表示される。

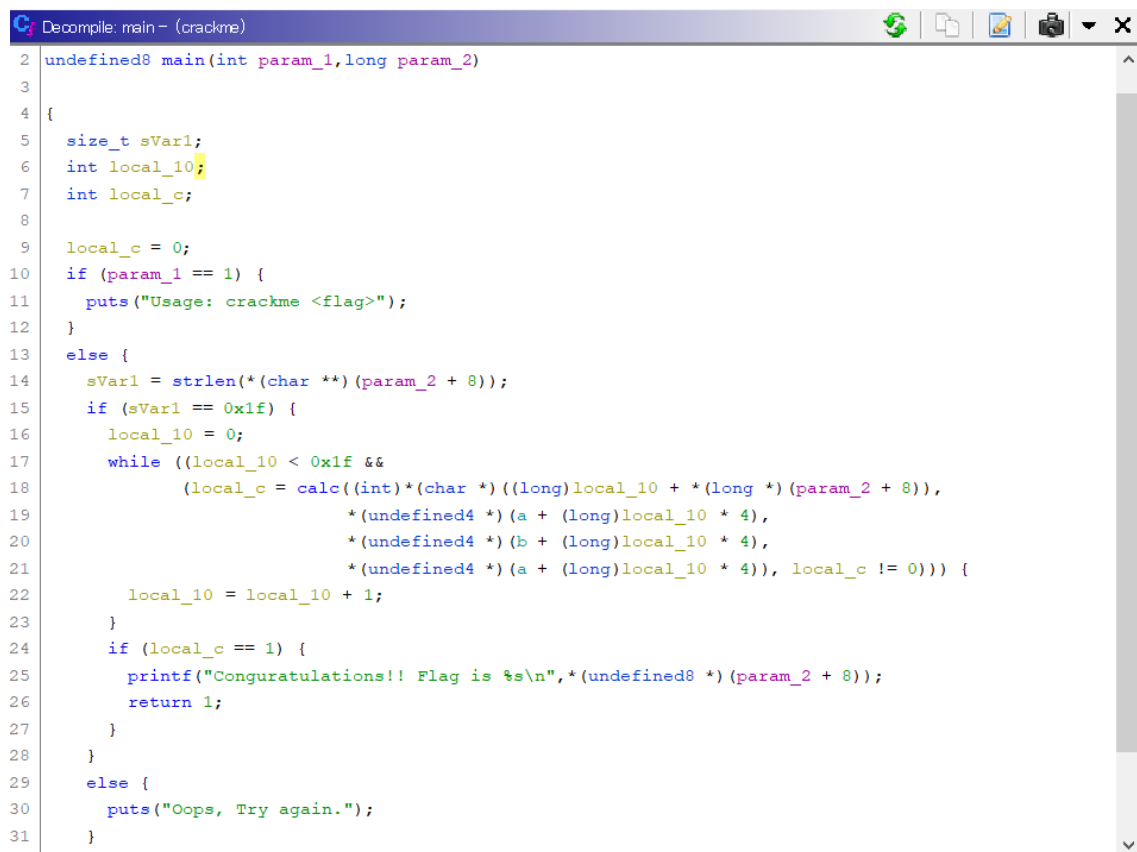
A terminal window titled 'kali@kali: ~/crackme-writeup' with a menu bar (File, Actions, Edit, View, Help). The terminal shows the following commands and output:

```
kali@kali:~/crackme-writeup$ ls
crackme
kali@kali:~/crackme-writeup$ file crackme
crackme: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d69092bd1aa8d178ec76bf9c7ac6ad33b4c6fda6, for GNU/Linux 3.2.0, not stripped
kali@kali:~/crackme-writeup$ ./crackme
Usage: crackme <flag>
kali@kali:~/crackme-writeup$ ./crackme HarekazeCTF{DUMMY}
Oops, Try again.
kali@kali:~/crackme-writeup$
```

図 1 : file コマンドの結果

## Ghidra を使った静的解析によるフラグ入手

今回は Ghidra で解析をする。解析するツールはお好きなものを使用してもらって OK です。



```
Decompile: main - (crackme)
2 undefined8 main(int param_1,long param_2)
3
4 {
5     size_t sVar1;
6     int local_10;
7     int local_c;
8
9     local_c = 0;
10    if (param_1 == 1) {
11        puts("Usage: crackme <flag>");
12    }
13    else {
14        sVar1 = strlen((char *) (param_2 + 8));
15        if (sVar1 == 0x1f) {
16            local_10 = 0;
17            while ((local_10 < 0x1f &&
18                (local_c = calc((int) *(char *) ((long) local_10 + *(long *) (param_2 + 8)),
19                    *(undefined4 *) (a + (long) local_10 * 4),
20                    *(undefined4 *) (b + (long) local_10 * 4),
21                    *(undefined4 *) (a + (long) local_10 * 4)), local_c != 0))) {
22                local_10 = local_10 + 1;
23            }
24            if (local_c == 1) {
25                printf("Conguratulations!! Flag is %s\n",*(undefined8 *) (param_2 + 8));
26                return 1;
27            }
28        }
29        else {
30            puts("Oops, Try again.");
31        }
32    }
```

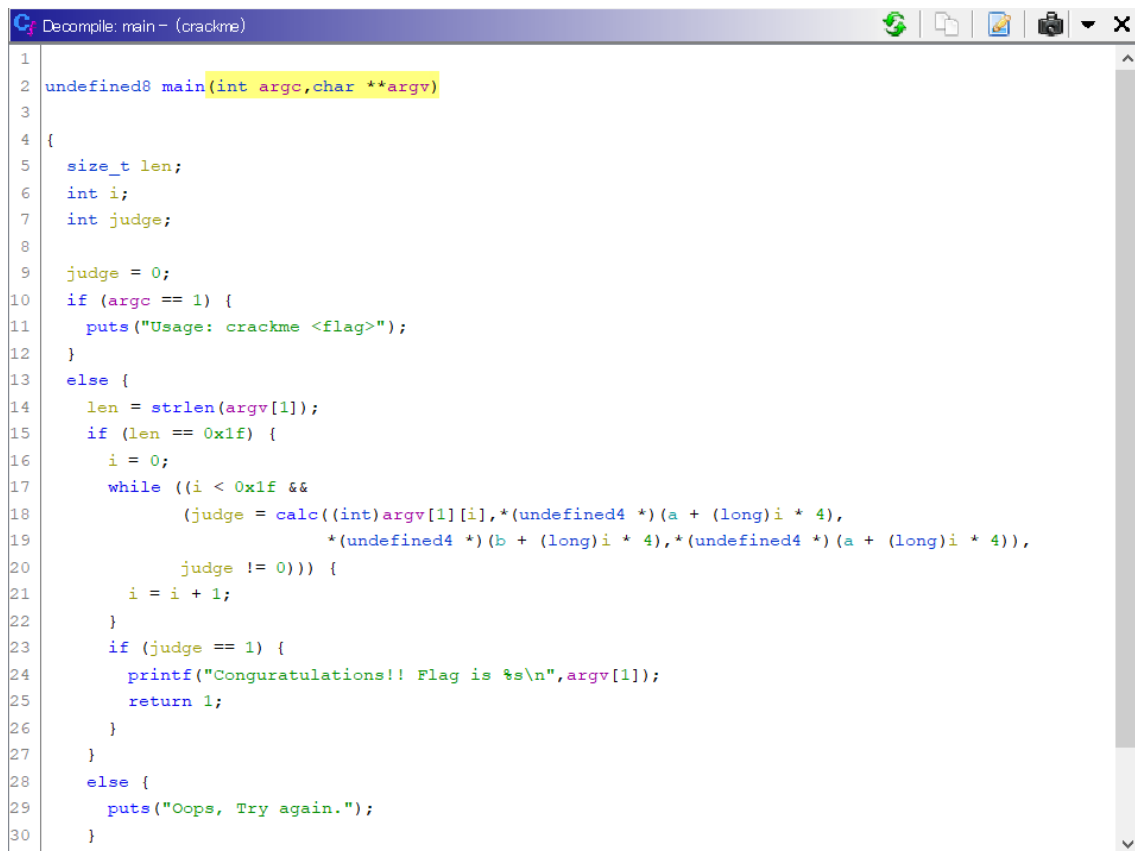
図 2 : Ghidra でのデコンパイル結果

見てみると、「Conguratulations!! Flag is %s\n」の文字列と「Oops, Try again.」の文字列が見つかる。また、最初のほうには「Usage: crackme <flag>」とあり、使用方法の文字列が見当たる。

Flag は 24 行目の if 文で分岐している。ここでは local\_c が 1 であれば、「Conguratulations」と表示し、この local\_c は calc という関数の結果として受け取っていることがわかる。

また、calc 関数は while 文で実行されており、この while 文は local\_10 が 0x1f(31)まで繰り返して実行される。このことを踏まえて Ghidra で変数書き換えを行って見やすくする。また、メイン関数の引数は、引数の数 argc と引数 argv なのでこの辺りも書き換

えてしまう。すると次のようになる。



```
1
2 undefined8 main(int argc, char **argv)
3
4 {
5     size_t len;
6     int i;
7     int judge;
8
9     judge = 0;
10    if (argc == 1) {
11        puts("Usage: crackme <flag>");
12    }
13    else {
14        len = strlen(argv[1]);
15        if (len == 0x1f) {
16            i = 0;
17            while ((i < 0x1f &&
18                (judge = calc((int)argv[1][i], *(undefined4 *) (a + (long)i * 4),
19                    *(undefined4 *) (b + (long)i * 4), *(undefined4 *) (a + (long)i * 4)),
20                judge != 0))) {
21                i = i + 1;
22            }
23            if (judge == 1) {
24                printf("Conguratulations!! Flag is %s\n", argv[1]);
25                return 1;
26            }
27        }
28        else {
29            puts("Oops, Try again.");
30        }
31    }
```

図 3 : Ghidra の変数名書き換え

calc 関数には、argv[1][i]、グローバル変数 a のデータとグローバル変数 b のデータが渡されている。i\*4 という要素で参照していることから、4 バイトのデータ型である。

これで main 関数がどういう処理をするのかが大体わかった。次の処理をする。

- ① 引数をチェックして、引数の数が足りなければ「Usage」を print する。
- ② 引数をチェックして与えた文字列の長さが 31 かどうかをチェックする
- ③ 文字列の長さが 31 の場合、31 回 calc 関数を実行する。
- ④ すべての calc 関数の return の値が 1 の場合、「Conguratulations!!」を表示する

上記動作をするので、calc 関数の内容を見ていく。

```
Decompile: calc - (crackme)
1
2 bool calc(char param_1,int param_2,int param_3)
3
4 {
5     return param_3 + (int)param_1 * (int)param_1 + param_1 * param_2 == 0;
6 }
7
```

calc 関数はとてもシンプルな関数で、次の計算結果が 0 であれば、1 を返す処理を行う。

$$param\_1^2 + param\_1 * param\_2 + param\_3 = 0$$

シンプルに calc 関数も書き換えておくと次のようになる。

```
Decompile: calc - (crackme)
1
2 bool calc(char x,int y,int z)
3
4 {
5     return z + (int)x * (int)x + x * y == 0;
6 }
7
```

ここまでをまとめると、calc 関数にはグローバル変数 a と b と与えた文字列で 31 回二次方程式の計算処理を行っている。

ここまでわかれば後は計算するだけである。どんなやりかたで計算してもらってもよい。31 回の計算くらい余裕というのであれば手計算してもらってもよいが、GhidraScript を使うことにする。

```

ghidra_solver.py
a_addr = 0x104060
b_addr = 0x1040e0
a_lis = []
for i in range(32):
    a_lis.append(getInt(toAddr(a_addr + (4 * i))))

# a_lis = [122, 69, 62, 88, 61, 87, 53, 57, 108, 104, 95, 73, 58, 56, 84, 97, 52, 96, 56, 141, 86, 76, 133, 58, 64, 61,
b_lis = []
for i in range(32):
    b_lis.append(getInt(toAddr(b_addr + (4 * i))))

# b_lis = [-13968, -16102, -20064, -19089, -17976, -17848, -21350, -15958, -11725, -15792, -11550, -24108, -19323, -202

import math
def solve(b,c):
    D = math.sqrt(b**2 - (4 * c))
    x1 = (-b + D) / 2
    x2 = (-b -D) /2
    return x1,x2

ans = []
for i in range(32):
    x1,x2 = solve(a_lis[i],b_lis[i])
    ans.append(int(x1))

print(''.join([chr(x) for x in ans]))
#HarekazeCTF{quadrat1c_3quati0n}\x00'

```

図 4 : Ghidra Script

このスクリプトを実行すると答えが得られる。答えは

「HarekazeCTF{quadrat1c\_3quati0n}」となる。プログラムに引数として渡すと期待通りの結果となる。

```

kali@kali: ~/crackme-writeup
File Actions Edit View Help
kali@kali:~/crackme-writeup$ ls
crackme
kali@kali:~/crackme-writeup$ file crackme
crackme: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-
linux-x86-64.so.2, BuildID[sha1]=d69092bd1aa8d178ec76bf9c7ac6ad33b4c6fda6, for GNU/Linux 3.2.0, not stripped
kali@kali:~/crackme-writeup$ ./crackme
Usage: crackme <flag>
kali@kali:~/crackme-writeup$ ./crackme HarekazeCTF{DUMMY}
Oops, Try again.
kali@kali:~/crackme-writeup$ ./crackme HarekazeCTF{quadratic_3quati0n}
Congratulations!! Flag is HarekazeCTF{quadrat1c_3quati0n}
kali@kali:~/crackme-writeup$

```