

# CS255 – Lab 1 – Reverse Engineering

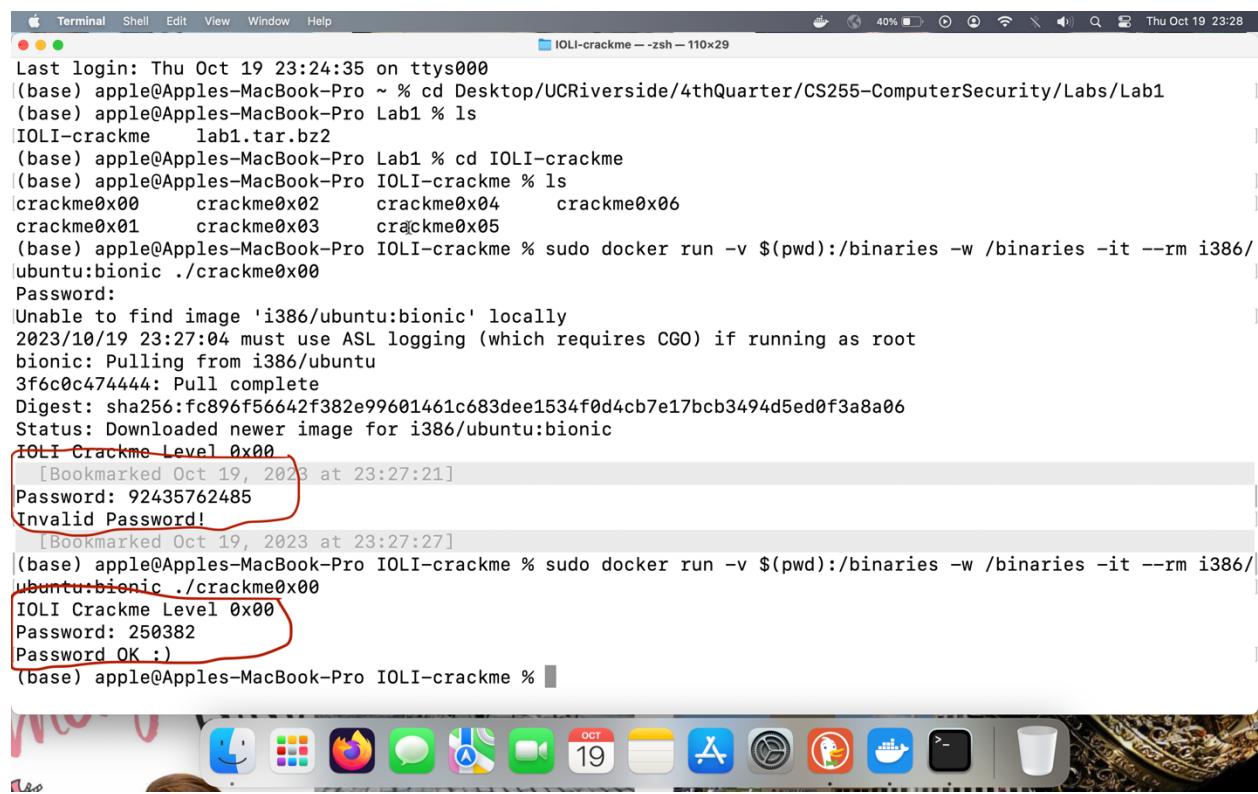
- Lohith Kumar Bhambore

## Setup

- For analyzing these crackme files and running them, I used Ghidra and Docker desktop....
- Following the instructions as posted in the video on Yuja, I installed Ghidra to decompile and analyze the code files
- To run the code, I installed docker desktop and used the command shared on piazza –  
`<< sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bionic  
./crackme0x00>>`

## Crackme0x00

Password – 250382



The screenshot shows a macOS Terminal window titled "IOLI-crackme -- zsh -- 110x29". The terminal output is as follows:

```
Last login: Thu Oct 19 23:24:35 on ttys000
(base) apple@Apples-MacBook-Pro ~ % cd Desktop/UCRiverside/4thQuarter/CS255-ComputerSecurity/Labs/Lab1
(base) apple@Apples-MacBook-Pro Lab1 % ls
IOLI-crackme lab1.tar.bz2
(base) apple@Apples-MacBook-Pro Lab1 % cd IOLI-crackme
(base) apple@Apples-MacBook-Pro IOLI-crackme % ls
crackme0x00 crackme0x02 crackme0x04 crackme0x06
crackme0x01 crackme0x03 crackme0x05
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/
ubuntu:bionic ./crackme0x00
Password:
Unable to find image 'i386/ubuntu:bionic' locally
2023/10/19 23:27:04 must use ASL logging (which requires CGO) if running as root
bionic: Pulling from i386/ubuntu
3f6c0c474444: Pull complete
Digest: sha256:fc896f56642f382e99601461c683dee1534f0d4cb7e17bcb3494d5ed0f3a8a06
Status: Downloaded newer image for i386/ubuntu:bionic
IOLI Crackme Level 0x00
[Bookmarked Oct 19, 2023 at 23:27:21]
Password: 92435762485
Invalid Password!
[Bookmarked Oct 19, 2023 at 23:27:27]
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/
ubuntu:bionic ./crackme0x00
IOLI Crackme Level 0x00
Password: 250382
Password OK :)
(base) apple@Apples-MacBook-Pro IOLI-crackme %
```

The terminal window is set against a background of a Mac OS X desktop with various application icons visible in the Dock.

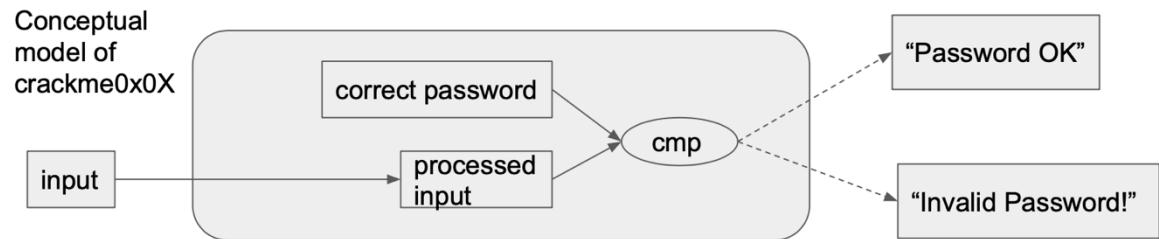
## Analysis –

- Using Ghidra to decompile binary file crackme0x00 we can find the main function (as seen in the screenshot below)

```

1 undefined4 main(void)
2 {
3     int iVar1;
4     char local_1c [24];
5
6     printf("IOLI Crackme Level 0x00\n");
7     printf("Password: ");
8     scanf("%s",local_1c);
9     iVar1 = strcmp(local_1c,"250382");
10    if (iVar1 == 0) {
11        printf("Password OK :)\n");
12    }
13    else {
14        printf("Invalid Password!\n");
15    }
16    return 0;
17 }
18
19 }
```

- A static analysis of the main function reveals that the file reads input from the user using `scanf` and stores it in a string variable called `local_1c`
- The next step is comparison of `local_1c` and correct password. Shown below is a generic model of how we can get “Password OK” or “Invalid Password” (reference – slides by Zhengchuan Liang)



- In this case the program uses `strcmp` function which returns 0 if the two strings being compared are the same. The result is stored in a variable called `iVar1`.
- Using `strcmp`, the comparison is with the processed input and the correct password which is exposed and the value is “250382”
- And it prints “Password OK :)” if the input string is the same as “250382”.
- In all other cases it prints “Invalid password!”

## Crackme0x01

Password – 5274

```
(base) apple@Apples-MacBook-Pro IOLI-crackme % docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bionic ./crackme0x01
IOLI Crackme Level 0x01
>Password: 1348314
Invalid Password!
(base) apple@Apples-MacBook-Pro IOLI-crackme % docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bionic ./crackme0x01
IOLI Crackme Level 0x01
>Password: 5274
Password OK :)
(base) apple@Apples-MacBook-Pro IOLI-crackme %
```



## Analysis –

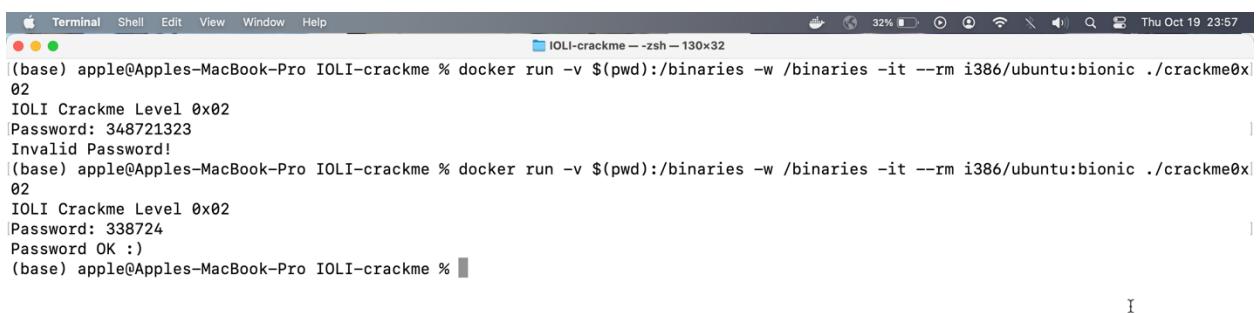
- Using Ghidra to decompile binary file crackme0x01 we can find the main function (as seen in the screenshot below)

```
CodeBrowser: cs255/crackme0x01
File Edit Analysis Graph Navigation Search Select Tools Window Help
Program Trees Listing: crackme0x01
Program Tree x
Symbol Tree x
Data Type Manager x
Data Types
  BuiltInTypes
  crackme0x01
  generic_clib
Filter: x
Console - Scripting
Decompile: main - (crackme0x01)
1 undefined4 main(void)
2 {
3     int local_8;
4
5     printf("IOLI Crackme Level 0x01\n");
6     printf("Password: ");
7     scanf("%d",&local_8);
8     if (local_8 == 0x149a) {
9         printf("Password OK :)\n");
10    }
11   else {
12       printf("Invalid Password!\n");
13   }
14   return 0;
15 }
16
17 }
```

- A static analysis of the main function reveals that the file reads input from the user using `scanf` and stores it in a integer variable called `local_8`
- This time the comparison is b/w two numbers, one the value input by the user and the second one is a hexadecimal value ‘`0x149a`’ which in decimal is ‘`5274`’.
- It doesn’t matter if we are comparing hexadecimal and decimal numbers as the values are all converted to their binary representations. This is just a simple obfuscation on the correct password.
- Thus, if the user input is ‘`5274`’, the program prints “Password OK : )”
- For any other user input, it prints “Invalid Password!”

## Crackme0x02

Password – 338724



```
apple@Apple-MacBook-Pro IOLI-crackme % docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bionic ./crackme0x02
IOLI Crackme Level 0x02
>Password: 348721323
Invalid Password!
apple@Apple-MacBook-Pro IOLI-crackme % docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bionic ./crackme0x02
IOLI Crackme Level 0x02
>Password: 338724
Password OK :)
apple@Apple-MacBook-Pro IOLI-crackme %
```



Analysis –

- Using Ghidra to decompile binary file `crackme0x02` we can find the main function (as seen in the screenshot below)

The screenshot shows the Immunity Debugger interface with the following windows:

- Program Trees**: Shows the project structure with files like .bss, .data, .got.plt, .got, and .dynamic.
- Listing: crackme0x02**: Displays the assembly code for the main function.
- Decompile: main - (crackme0x02)**: Shows the C decompiled code corresponding to the assembly.
- Console - Scripting**: An empty command-line interface.

The assembly code in the Listing tab:

```

main:
    .text
    .section .text
    .globl main
main:
    .L1:    push    ebp
    .L2:    mov     ebp,esp
    .L3:    sub     esp,0x18
    .L4:    and     esp,0xfffffff0
    .L5:    mov     eax,0x0
    .L6:    add     eax,0xf
    .L7:    add     eax,0xf
    .L8:    shr     eax,0x4
    .L9:    shl     eax,0x4
    .L10:   sub    esp,0x4

```

The decompiled code in the Decompile tab:

```

1 undefined4 main(void)
2 {
3     int local_8;
4
5     printf("IOLI Crackme Level 0x02\n");
6     printf("Password: ");
7     scanf("%d",&local_8);
8     if (local_8 == 0x52b24) {
9         printf("Password OK :)\n");
10    }
11    else {
12        printf("Invalid Password!\n");
13    }
14    return 0;
15 }
16
17 }
18

```

- This code is very similar to the previous one (0x01). This time again we have a comparison b/w two numbers, one the value input by the user and the second one is a hexadecimal value ‘0x52b24’ which in decimal is ‘338724’.
- It doesn’t matter if we are comparing hexadecimal and decimal numbers as the values are all converted to their binary representations (A simple obfuscation once again)
- Thus, if the user input is ‘338724’, the program prints “Password OK : )”
- And just as before, for any other input, it prints “Invalid Password!”

## Crackme0x03

### Password – 338724

```

apple@apples-MacBook-Pro IOLI-crackme % docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bionic ./crackme0x03
IOLI Crackme Level 0x03
Password: fwjdbcf13
Invalid Password!
apple@apples-MacBook-Pro IOLI-crackme % docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bionic ./crackme0x03
IOLI Crackme Level 0x03
Password: 338724
Password OK!!! :)
apple@apples-MacBook-Pro IOLI-crackme %

```



## Analysis –

- Using Ghidra to decompile binary file crackme0x03 we can find the main function (as seen in the screenshot below)

The screenshot shows the Ghidra interface with the following windows:

- Program Trees:** Shows the project structure with 'crackme0x03' selected.
- Listing:** Displays assembly code for address 0x00 to 0x04. It includes instructions like CALL, LEAVE, and RET, along with labels LAB\_08048496 and LAB\_08048497.
- Decompile:** Shows the decompiled C code for the main function. The code reads a password from standard input, prints it, and then calls a test function with the password and the value 0x52b24. It returns 0.
- Symbol Tree:** Shows symbols including 'main', 'local\_8', 'local\_c', 'local\_10', 'local\_2c', and 'local\_30'.
- Data Type Manager:** Shows built-in types and user-defined types for 'crackme0x03' and 'generic\_clib'.
- Console - Scripting:** An empty console window.

- A static analysis of the main function reveals that the file reads input from the user using `scanf` and stores it in a local variable called `local_8`
- Now, instead of direct comparison, the program calls a `test` function with `local_8` and the hexadecimal value '0x52b24' as arguments. It finally returns 0.
- Next let's look at the `test` function

```

1 void test(int param_1,int param_2)
2 {
3     if (param_1 == param_2) {
4         shift("Sdvvzrug#RN$$$$#=");
5     }
6     else {
7         shift("Lqydolg#Sdvvzrug$");
8     }
9     return;
10}
11
12
13

```

- The test function compares the two integer parameters, and if they are equal, it calls the shift function with the string “Sdvvzrug#RN\$\$\$\$#=,” as an argument
- If they are not equal, it still calls the shift function but with a different string – “Lqydolg#Sdvvzrug\$”
- Now let's take a look at the shift function

```

1 void shift(char *param_1)
2 {
3     size_t sVar1;
4     uint local_80;
5     char local_7c [120];
6
7     local_80 = 0;
8     while( true ) {
9         sVar1 = strlen(param_1);
10        if (sVar1 <= local_80) break;
11        local_7c[local_80] = param_1[local_80] + -3;
12        local_80 = local_80 + 1;
13    }
14    local_7c[local_80] = '\0';
15    printf("%s\n",local_7c);
16    return;
17}
18
19
20

```

- The shift function takes a string called param\_1 as an argument

- It keeps track of the size of param\_1 using sVar1 and uses a local\_80 as a counter for the while loop.
- The code in the while loop is basically a simple character shift. It is subtracting 3 from the ASCII values of each character in the string param\_1 and updates it into the corresponding location in a new char array local\_7c.
- It is essentially creating a new string one character at a time.
- Lastly it terminates the string by adding '\0' character to local\_7c and then prints it.
- By performing this shift operation (subtracting 3 in ASCII value for each character) on the string "Sdvvzrug#RN\$\$\$#=," gives us – "Password OK!!! : )"
- With string "Lqydolg#Sdvvzrug\$", it gives us – "Invalid Password!"
- This just provides a simple obfuscation on the output strings printed, but the value that is actually being compared with the user input is still '0x52b24' or '338724'
- Thus, we can conclude that if the user enters the value '338724', it prints "Password OK!!! : )"
- In all other cases, it prints "Invalid Password!"

## Crackme0x04

Password –

Sum of the digits in the input starting from the leftmost one should be 15 at some point (incrementally)

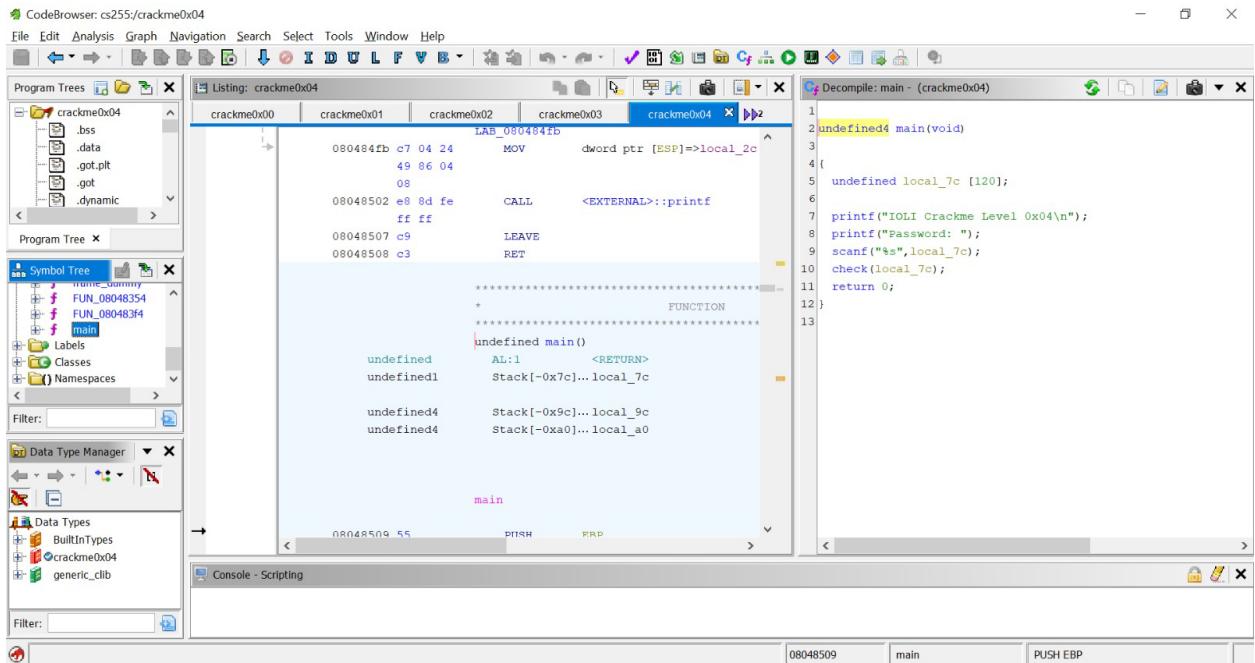
eg: 78, 781, 7813143251435 etc.

```
apple@Apples-MacBook-Pro IOLI-crackme % Terminal Shell Edit View Window Help
apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/
ubuntu:bionic ./crackme0x04
>Password:
Sorry, try again.
>Password:
IOLI Crackme Level 0x04
>Password: 78
>Password OK!
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/
ubuntu:bionic ./crackme0x04
IOLI Crackme Level 0x04
>Password: 781
>Password OK!
(base) apple@Apples-MacBook-Pro IOLI-crackme % 781243254235425
zsh: command not found: 781243254235425
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/
ubuntu:bionic ./crackme0x04
IOLI Crackme Level 0x04
>Password: 7813143251435
>Password OK!
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/
ubuntu:bionic ./crackme0x04
IOLI Crackme Level 0x04
>Password: 99871343
>Password Incorrect!
(base) apple@Apples-MacBook-Pro IOLI-crackme %
```



## Analysis –

- Using Ghidra to decompile binary file crackme0x04 we can find the main function (as seen in the screenshot below)



The screenshot shows the Ghidra interface with the following windows:

- Program Trees**: Shows the project structure with files like .bss, .data, .got.plt, .got, and .dynamic.
- Symbol Tree**: Shows symbols including frame\_dummy, FUN\_08048354, FUN\_080483f4, and main.
- Data Type Manager**: Shows data types like BuiltinTypes, crackme0x04, and generic\_clib.
- Listing: crackme0x04**: Shows assembly code for the main function, starting with MOV dword ptr [ESP] to local\_2c.
- Decompile: main - (crackme0x04)**: Shows the decompiled C# code for the main function:

```
1 undefined4 main(void)
2 {
3     undefined local_7c [120];
4
5     printf("IOLI Crackme Level 0x04\n");
6     printf("Password: ");
7     scanf("%s",local_7c);
8     check(local_7c);
9     return 0;
10 }
11
12 }
```

- Console - Scripting**: An empty console window.

- A static analysis of the main function reveals that the file reads input from the user using scanf and stores it in a local string variable called local\_7c
- Now, instead of direct comparison with the correct password, the program calls a check function with local\_7c as the argument. It finally returns 0.
- Next let's look at the check function

The screenshot shows the Immunity Debugger interface with the following windows:

- Program Trees**: Shows the file structure of the binary.
- Listing: crackme0x04**: Displays the assembly code for the current function.
- Decompile: check - (crackme0x04)**: Shows the corresponding C decompilation of the check function.
- Symbol Tree**: Lists local variables and their types.
- Data Type Manager**: Shows available data types.
- Console - Scripting**: A command-line interface at the bottom.

The assembly listing shows the following code snippet for the `check` function:

```

08048484 55      PUSH    EBP
08048485 89 e5   MOV     EBP,ESP
08048487 83 ec 28 SUB    ESP,0x28
0804848a c7 45 f8 MOV    dword ptr [EBP + local_c], 00 00 00
00
08048491 c7 45 f4 MOV    dword ptr [EBP + local_10], 00 00 00
00
LAB_08048498 8b 45 08 MOV    EAX,dword ptr [EBP + param_1]
0804849b 09 04 24 MOV    dword ptr [ESP] =>local_2c

```

The decompiled C code for the `check` function is:

```

void check(char *param_1)
{
    size_t sVar1;
    char local_11;
    uint local_10;
    int local_c;
    int local_8;

    local_c = 0;
    local_10 = 0;
    while( true ) {
        sVar1 = strlen(param_1);
        if (sVar1 <= local_10) {
            printf("Password Incorrect!\n");
            return;
        }
        local_11 = param_1[local_10];
        sscanf(&local_11,"%d",&local_8);
        local_c = local_c + local_8;
        if (local_c == 0xf) break;
        local_10 = local_10 + 1;
    }
    printf("Password OK!\n");
    /* WARNING: Subroutine does not return */
}

```

- The `check` function takes a string `param_1` as an argument. It also has a bunch of local variables
- `sVar1` to keep size of the string `param_1`.
- A char variable `local_11` to read and temporarily store each character of the string `param_1`
- An integer `local_10` which works as a counter for the while loop, which is initialized to 0
- An integer `local_c` which stores the sum so far and is initialized to 0.
- And an integer `local_8` which stores the int value of the character `local_11`
- In the while loop, there is a check if the counter `local_10` exceeds the size of `param1`, in which case, it prints “Password Incorrect!” and returns.
- If the condition is false, i.e the current char is not the last char of the string `param_1`, then it reads that character and stores it in `local_11`
- It then uses `sscanf` to convert that character to an integer and stores it in `local_8`
- Next, it increments adds the value of `local_8` to the sum so far, i.e to `local_c` and stores it in `local_c`
- Next, it checks if the sum so far (`local_c`) is equal to `0xf` (which is 15 in decimal system). If that is true, it breaks out of the loop.
- And then it prints “Password OK!” and returns
- Else, it increments the counter `local_10` by 1 and goes to the next iteration.
- Since it is checking the sum of digits incrementally from the left and prints “Password OK!” as long as the sum so far reaches exactly 15 and doesn’t check the rest of the digits in the string, all valid passwords are the ones in which if the sum of digits of the input from the left equal 15 at any time.
- Invalid passwords are all in which the sum of digits from the left is not equal to 15 at any time.

## Crackme0x05

Password –

Sum of the digits in the input starting from the leftmost one should be 16 at some point (incrementally)

And

Number must be even.

eg: 88, 4444, 970, 974568, 882 etc.

```
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bianic ./crackme0x05
IOLI Crackme Level 0x05
>Password: 97
Password Incorrect!
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bianic ./crackme0x05
IOLI Crackme Level 0x05
>Password: 970
Password OK!
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bianic ./crackme0x05
IOLI Crackme Level 0x05
>Password: 974568
Password OK!
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bianic ./crackme0x05
IOLI Crackme Level 0x05
>Password: 88
Password OK!
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bianic ./crackme0x05
IOLI Crackme Level 0x05
>Password: 882
Password OK!
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bianic ./crackme0x05
IOLI Crackme Level 0x05
>Password: 79
Password Incorrect!
(base) apple@Apples-MacBook-Pro IOLI-crackme % sudo docker run -v $(pwd):/binaries -w /binaries -it --rm i386/ubuntu:bianic ./crackme0x05
IOLI Crackme Level 0x05
>Password: 792
Password OK!
```

Analysis –

- Using Ghidra to decompile binary file crackme0x05 we can find the main function (as seen in the screenshot below)

```

1 undefined4 main(void)
2 {
3     undefined local_7c [120];
4
5     printf("IOLI Crackme Level 0x05\n");
6     printf("Password: ");
7     scanf("%s",local_7c);
8     check(local_7c);
9     return 0;
10 }
11
12 }
```

- A static analysis of the main function reveals that the file reads input from the user using scanf and stores it in a local string variable called local\_7c
- Now, instead of direct comparison with the correct password, the program calls a check function with local\_7c as the argument. It finally returns 0.
- Next let's look at the check function

```

1 void check(char *param_1)
2 {
3     size_t sVar1;
4     char local_11;
5     uint local_10;
6     int local_c;
7     int local_8;
8
9     local_c = 0;
10    local_10 = 0;
11    while( true ) {
12        sVar1 = strlen(param_1);
13        if (sVar1 <= local_10) break;
14        local_11 = param_1[local_10];
15        sscanf(&local_11,"%d",&local_8);
16        local_c = local_c + local_8;
17        if (local_c == 0x10) {
18            parell(param_1);
19        }
20        local_10 = local_10 + 1;
21    }
22    local_10 = local_10 + 1;
23 }
24 printf("Password Incorrect!\n");
25 return;
26 }
```

- The check function takes a string param\_1 as an argument. It also has a bunch of local variables similar to the previous one

- sVar1 to keep size of the string param\_1.
- A char variable local\_11 to read and temporarily store each character of the string param\_1
- An integer local\_10 which works as a counter for the while loop, which is initialized to 0
- An integer local\_c which stores the sum so far and is initialized to 0.
- And an integer local\_8 which stores the int value of the character local\_11
- Next we have an indefinite while loop, within which we check if length of string param\_1 is less than or equal to local\_10. In case it is true, we break out of the loop
- Next, we read a character from the string param\_1 (at the index local\_10 which is a counter) and store it in local\_11
- We then convert the character stored in local\_11 into an integer using sscanf and store it in local\_8
- We then add local\_8 to the sum so far, i.e. local\_c and store it in local\_c
- We now check, if the sum so far or local\_c is equal to the value '0x10', which is 16 in decimal system, if it is true we call the parell function while passing the string param\_1
- We then increment the value of the counter local\_10 by 1
- If the loop exits without the sum of digits from the left ever reaching 16 or 0x10, we print "Password Incorrect!"
- Now let's take a look at the parell function -

The screenshot shows the Immunity Debugger interface with the following windows:

- Program Trees**: Shows the project structure with files like crackme0x00, crackme0x01, etc.
- Symbol Tree**: Shows symbols such as FUN\_080483f4, main, local\_7c, local\_9c, local\_a0, and parell.
- Listing**: Shows assembly code for the parell function, including instructions like PUSH EBP, MOV EBP, SUB ESP, LEA EAX=>local\_8, MOV dword ptr [ESP + local\_14], and MOV EAX, dword ptr [EBP + param\_1].
- Decompile**: Shows the C-like decompiled code for the parell function, which reads a character from param\_1, converts it to an integer using sscanf, adds it to local\_8, and then checks if local\_8 is even (local\_8 & 1 == 0). If it is, it prints "Password OK!\n" and exits.
- Console - Scripting**: Shows the command line interface with the current address (08048484), function name (parell), and assembly instruction (PUSH EBP).

- The parell function takes a string argument called param\_1
- It also has a local integer variable called local\_8
- It first converts the string param\_1 into an integer and stores it in local\_8 using sscanf
- Now it checks if the number local\_8 is even using bitwise AND operation
- If it is even, it prints "Password OK!" and exits the program.

- Since it is checking the sum of digits incrementally from the left and prints “Password OK!” as long as the sum so far reaches exactly 16 and the number is even and doesn’t check the rest of the digits in the string
- all valid passwords are the ones in which if the sum of digits of the input from the left equal 16 at any time and that the input is even.
- Invalid passwords are all in which the sum of digits from the left is not equal to 16 at any time and cases where the sum is 16, but the number is not even.