National Institute Of Science And Technology Of Tunisia

Decompilers and Packers

Authors :

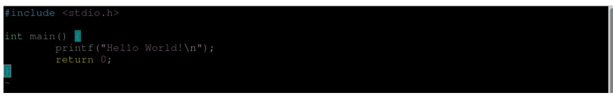
Name and Forename :

Med Ali Ouachani

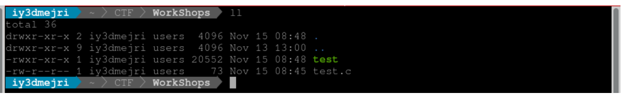
Iyed Mejri

Year :
2022 - 2023

I - Disassemblers / Decompilers :



```c
#include <stdio.h>

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

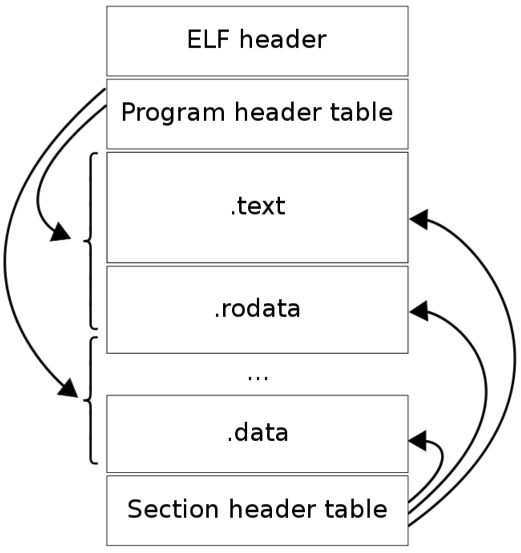


```
iy3dmejri  ~ 〉 CTF 〉 WorkShops  ll
total 36
drwxr-xr-x 2 iy3dmejri users  4096 Nov 15 08:48 .
drwxr-xr-x 9 iy3dmejri users  4096 Nov 13 13:00 ..
-rwxr-xr-x 1 iy3dmejri users 20552 Nov 15 08:48 test
-rw-r--r-- 1 iy3dmejri users    73 Nov 15 08:45 test.c
iy3dmejri  ~ 〉 CTF 〉 WorkShops  
```

If we delete the test.c file, we can no more access to the source code we have written since it has been transformed into machine code. Using the command ‘cat‘ to read the executable file we will just have a lot of gibberish.

```
iy3dmejri  ~ 〉 CTF 〉 WorkShops  cat test
@%$@@@�00aa   ���-�=�=HP�-�=�=�888@xxxDDS�td888@P�td   $$Q�tdR�td�-�=�=00/lib64/ld-linux-x86-64.so.20GNU����GNUP�p��]��?�|y����GNUH d s "puts__libc_start_main__cxa_finalizelibc.so.6GLIBC_2.2
.5GLIBC_2.34_ITM_deregisterTMCloneTable__gmon_start___ITM_registerTMCloneTable'u�i      1���=�0��@�?�?�?�?�?@��H�H��/H��t��H���5�/�%�/@�%�/h��������1�I��^H��H���PTE1�1�H�=��[/�f.�H�=�/H��/H9�
H�=/��.�c����/]�f.��ff.�@���g���UH��H��H��������]���H�H��Hello World!
                                                                                      ���T,���<%���|zRx
                                                                                                          ����&D$4���� FJ
                                                                                                                          �?�;*3$"\�����A�C
U
0�'
T������o���
�
<<\@&�5�C@&!�5O����o���o���o�=6@GCC: (GNU) 12.2.0@&7
                4�R&��(%�)lintf*&���`TA�Th�s�!�
                                             �         ����H��/        �"

!����%
      �x��E�
}Q     ��
Q�����&������;���
                 �(��E�����������-���   �"
�77�W�������J����������v��        x��'�1�;(E�O�Y�c�m�w���1�
                                                        ���!��{���
                                                                  ��

�
 F�

   ��
          �
3         �
3         �"
3
 ��<��    �          L
O+B�D�E q
       F
        �x          ��
3G�      �L�)O�9
                ;          int�
                              H�5�x-a�5�
                                        I7I$

                                            >
                                             &I:
                                                ;
                                                 9
                                                  I
:
 ;
  9
   I8
:    <
```

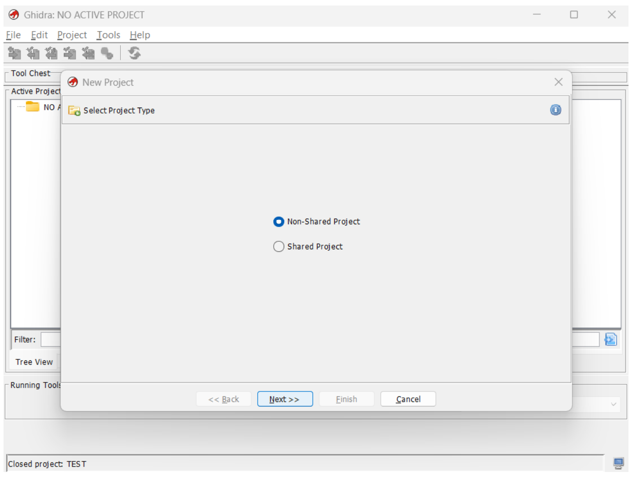
As we talked before every type of file has a certain structure containing metadata and other information that would let other programs know how to manipulate it. Talking about the ELF format (PE files are certainly the same as ELFs), this is the structure of an ELF file:



The program header table contains data describing segments (one or more section) and other information needed to prepare for execution. The section header table contains information describing each elf the file’s sections (address, name, size...). The other sections .text, .rodata, .data... would have data related to our program. For example the .text section would contain the compiled code we have written, the .rodata (Read Only Data) section would contain the string "Hello World!" that we have written at the beginning since that's a constant and it's not stored in any variable so it should be in the read only data section. The .data section would have our initialized global variables while defined but not initialized global variables are put in the .bss section. So in order to recover the source code of a compiled program we should disassemble/decompile the code inside the .text section. And for this purpose we will be using tools called disassemblers which will look up the .text section and disassemble and decompile it for us. The tools we are gonna use are **Ghidra** and **IDA Pro**. IDA is better then **Ghidra**, it's just my opinion please don't kill me. At the moment, we will be focusing on Ghidra since it's a free tool and everyone should be able to download it and install it on their machines.

NOTE: I'm saying disassemblers and not decompilers since tools we are using are converting code from machine code to assembly then to a C like syntax with the help of a plugin that usually accompanies the disassembler.

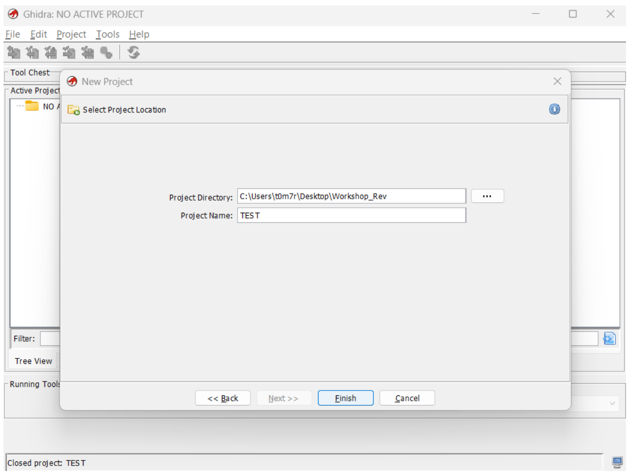
After opening Ghidra, we should create a project that will save all modifications and work we do. To do so we can go to 'File -> New Project' or press **CTRL + N**.



We choose **Non-Shared Project** and hit **Next**.

NOTE: The simple difference between non-shared and shared project is that a shared project is just like git repository that is pushed on a server (if you are looking to collaborate with others on a project), on the other hand the non-shared project is just a local project on your machine.

Now choose the project directory and give your project a name and hit **Finish**.



Project Directory: C:\Users\t0m7r\Desktop\Workshop_Rev
Project Name: TEST

Now we need to import the binary we like to analyze, we could either press **I** or go to **File -> Import File...** or even drag and drop the binary. For this workshop, let's analyze a compiled C program that I created. Here is the source code **login.c**:

```c
#include <stdio.h>
#include <string.h>

int password;

int main() {
        char username[0x10];

        printf("Username: ");
        fgets(username, 0x10, stdin);
        if (!strncmp(username, "Iyed", 4)) {
                printf("Password: ");
                scanf("%d", &password);
                if (password == 1337)
                        puts("Welcome on the board!");
                else
                        puts("Incorrect password!");
        }
        else
                puts("Incorrect username!");

        return 0;
}
```

To compile it, we would use the **gcc** (Gnu C Compiler) by using this command **gcc -o login login.c**. **-o** option with **login** is used to tell the compiler we want our executable to be output to a file named **login** (if we don't do so the compiler will just output the executable to a file named **a.out**). Before importing the executable to Ghidra let's start with some basic analysis + some dynamic analysis. Assuming we just have the executable 'login' but not the **login.c** source, let's start by using the **file** linux command to take a look at some information about the executable.

```
iy3dmejri > ~ > CTF > WorkShops > file login
login: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,

BuildID[sha1]=89630ec15bba4e54bbe255512ab1748ddceae861,
for GNU/Linux 4.4.0,
with debug_info, not stripped
```

Here is what we care about for the moment:
- The binary is a 64-bit ELF - The architecture it's meant to run on is x86_64 (AMD 64) - It's dynamically linked (Another link type you might have is **Statically Linked** which means that all library functions that were called are linked with all the functions they need in order to work into the code of our program) - It's not stripped

We know **ELF** (Executable and Linkable format) is the executable format for the Unix/Linux systems. Dynamically linked means the linker is going to link the library functions with our main program at runtime (Linker was discussed in the first document). Not stripped means that the binary includes debugging information and stripped removes this debugging information and any extra information that is not needed for the binary to run (For example function names and symbols, they are just addresses at the end of the day).

Starting with some basic dynamic analysis, executing the program and interacting with it by giving it different inputs (assuming that we don't know anything about the **login.c** file).

```
iy3dmejri > ~ > CTF > WorkShops > ./login
Username: test
Incorrect username!
}
```

```
iy3dmejri > ~ > CTF > WorkShops > ./login
Username: admin
Incorrect username!
}
```

Okay nothing intersting. We could run the executable under **ltrace** to trace (follow the execution flow instruction per instruction) library function calls maybe it's using some library function's to check our username.

```
iy3dmejri > ~ > CTF > WorkShops > ltrace ./login
printf("Username: ")
fgets(Username: asdf
"asdf\n", 16, 0x7f2b0dfac9c0)
strncmp("asdf\n", "Iyed", 4)
puts("Incorrect username!"Incorrect username!
)
+++ exited (status 0) +++
}
```

And effectively it was using **strncmp** to compare the first 4 bytes of our input with the string **Iyed** so we know for the moment that the **username** is **Iyed**. Entering it and taking a look again:

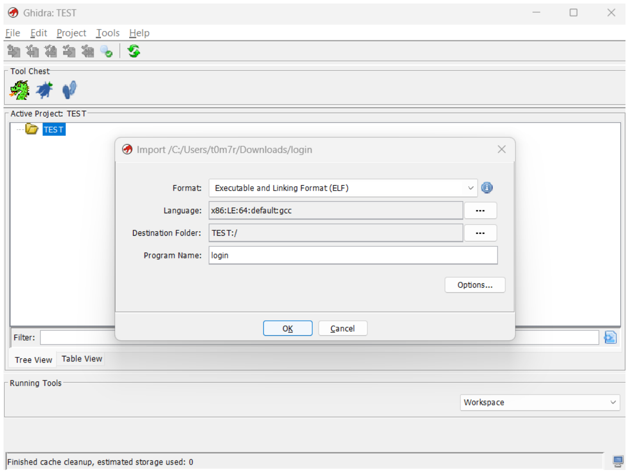
```
iy3dmejri > ~ > CTF > WorkShops > ./login
Username: Iyed
Password: 1234
Incorrect password!
```

Now it's asking for a password, let's try with **ltrace** again.

```
iy3dmejri > ~ > CTF > WorkShops > ltrace ./login
printf("Username: ")
fgets(Username: Iyed
"Iyed\n", 16, 0x7f3954b209c0)
strncmp("Iyed\n", "Iyed", 4)
printf("Password: ")
__isoc99_scanf(0x559f3ff7e01f, 0x559f3ff8004c, 0, 0Password: 1234
)                                                                  = 1
puts("Incorrect password!"Incorrect password!
)
+++ exited (status 0) +++
```

Sadly, **ltrace** is not helping here since the program is not checking for the password with a library function call maybe it's simply comparing the number we enter with another one using == operator. So let's start the static analysis process.

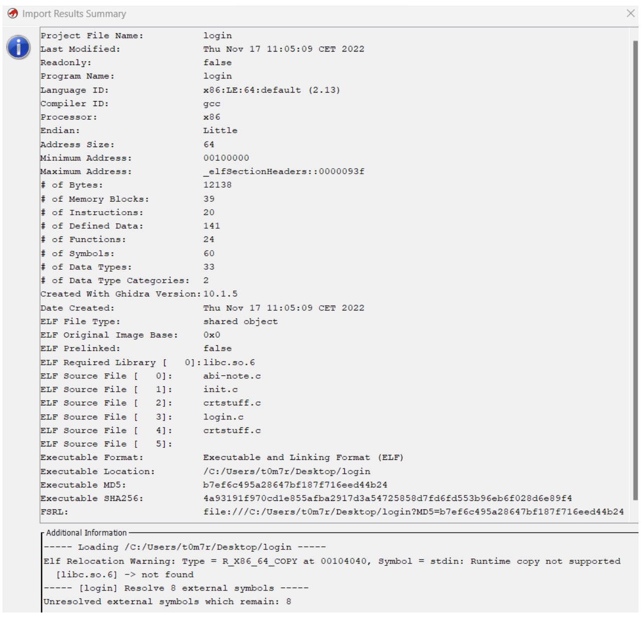
Now importing the binary into Ghidra. It will detect the basic info that the **file** command returned to us and some additional information about the size of the binary, the required shared libraries...



Import /C:/Users/t0m7r/Downloads/login

Format:	Executable and Linking Format (ELF)
Language:	x86:LE:64:default:gcc
Destination Folder:	TEST:/
Program Name:	login

Finished cache cleanup, estimated storage used: 0



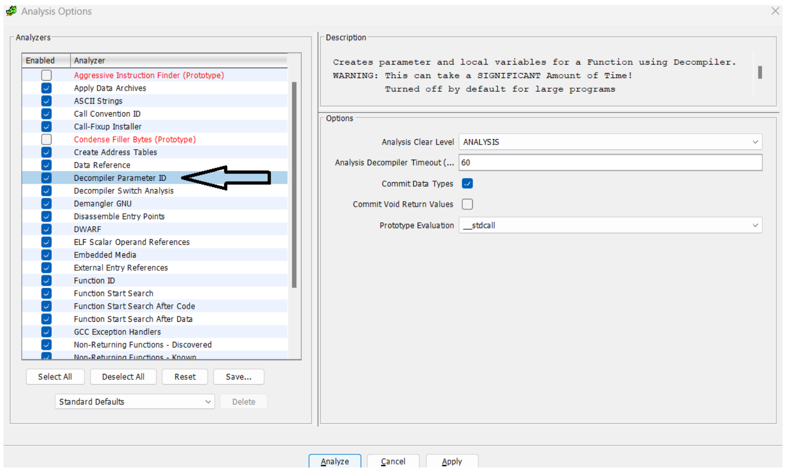
Import Results Summary

```
Project File Name:           login
Last Modified:               Thu Nov 17 11:05:09 CET 2022
Readonly:                    false
Program Name:                login
Language ID:                 x86:LE:64:default (2.13)
Compiler ID:                 gcc
Processor:                   x86
Endian:                      Little
Address Size:                64
Minimum Address:             00100000
Maximum Address:             _elfSectionHeaders::0000093f
# of Bytes:                  12138
# of Memory Blocks:          39
# of Instructions:           20
# of Defined Data:           141
# of Functions:              24
# of Symbols:                60
# of Data Types:             33
# of Data Type Categories:   2
Created With Ghidra Version:10.1.5
Date Created:                Thu Nov 17 11:05:09 CET 2022
ELF File Type:               shared object
ELF Original Image Base:     0x0
ELF Prelinked:               false
ELF Required Library [   0]:libc.so.6
ELF Source File [   0]:      abi-note.c
ELF Source File [   1]:      init.c
ELF Source File [   2]:      crtstuff.c
ELF Source File [   3]:      login.c
ELF Source File [   4]:      crtstuff.c
ELF Source File [   5]:
Executable Format:           Executable and Linking Format (ELF)
Executable Location:         /C:/Users/t0m7r/Desktop/login
Executable MD5:              b7ef6c495a28647bf187f716eed44b24
Executable SHA256:           4a93191f970cd1e855afba2917d3a54725858d7fd6fd553b96eb6f028d6e89f4
FSRL:                        file:///C:/Users/t0m7r/Desktop/login?MD5=b7ef6c495a28647bf187f716eed44b24
```

Additional Information

```
----- Loading /C:/Users/t0m7r/Desktop/login -----
Elf Relocation Warning: Type = R_X86_64_COPY at 00104040, Symbol = stdin: Runtime copy not supported
  [libc.so.6] -> not found
----- [login] Resolve 8 external symbols -----
Unresolved external symbols which remain: 8
```

Double clicking the file name to open the **codeBrowser** in Ghidra. Hitting YES on the would you like to analyze message. I mostly add to the default analyzers enabled the **Decompiler** Parameter ID' option.



Analysis Options

Enabled	Analyzer
☐	Aggressive Instruction Finder (Prototype)
✓	Apply Data Archives
✓	ASCII Strings
✓	Call Convention ID
✓	Call-Fixup Installer
☐	Condense Filler Bytes (Prototype)
✓	Create Address Tables
✓	Data Reference
✓	Decompiler Parameter ID
✓	Decompiler Switch Analysis
✓	Demangler GNU
✓	Disassemble Entry Points
✓	DWARF
✓	ELF Scalar Operand References
✓	Embedded Media
✓	External Entry References
✓	Function ID
✓	Function Start Search
✓	Function Start Search After Code
✓	Function Start Search After Data
✓	GCC Exception Handlers
✓	Non-Returning Functions - Discovered
✓	Non-Returning Functions - Known

Description

```
Creates parameter and local variables for a Function using Decompiler.
WARNING: This can take a SIGNIFICANT Amount of Time!
         Turned off by default for large programs
```

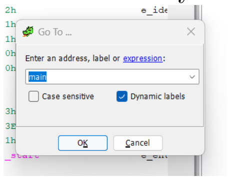
Options

Analysis Clear Level	ANALYSIS
Analysis Decompiler Timeout (...	60
Commit Data Types	✓
Commit Void Return Values	☐
Prototype Evaluation	__stdcall

Decompiler Parameter ID improves the decompilation of the program by determining the parameters and their types, the return type, and the calling convention information about the function, making the function signature clear.

NOTE: A function's signature basically includes parameters, their types and return type (Just like its declaration). If you feel that the function's signature that the decompiler generated is somewhat wrong you could edit it yourself. All types of variables and variable names are lost in the compilation process since from a low-level point of view we are just poking and peeking data into memory and it's going to be a number at the end of the day so the processor doesn't really care about types, the decompiler would try it's best to guess the correct types of certain variables based on their uses. Example: int test_func(char *a, int b)

Now starting our analysis by going to the main function and reading its code. A way of doing so, in a non stripped binary since symbols like function names are preserved, is to press **G** to access the **Go To ...** functionality in Ghidra and type main.



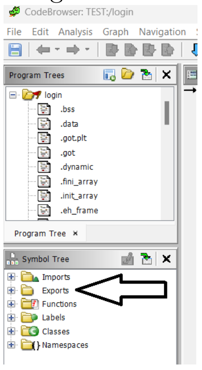
Go To ...

Enter an address, label or expression:

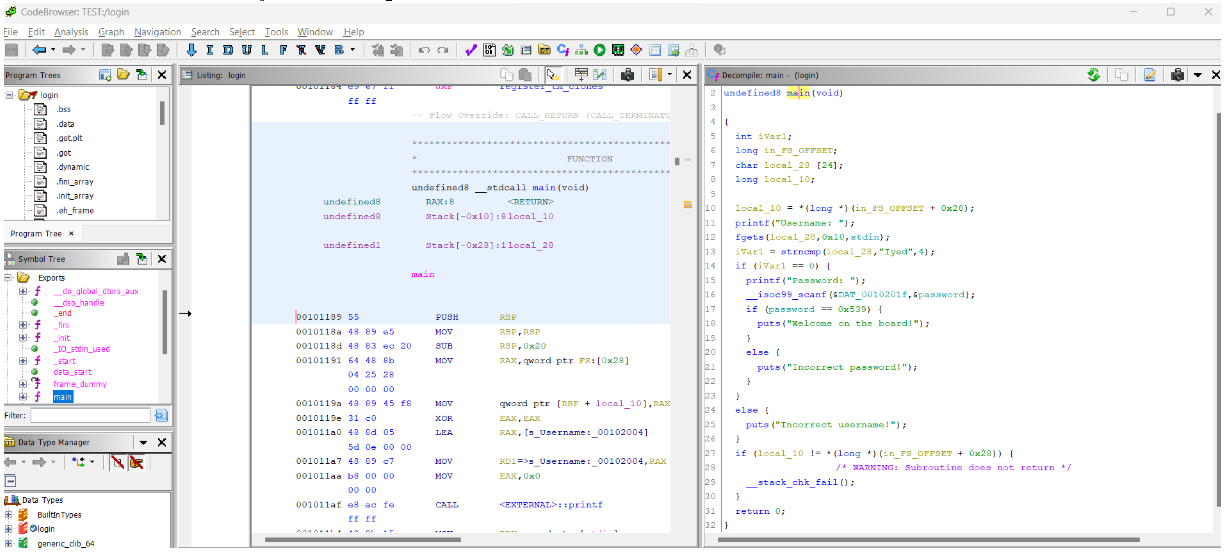
main

☐ Case sensitive ☑ Dynamic labels

Another way of doing it, is by going to the **Symbol Tree** window and then to the **Exports** folder which is a folder containing the symbols of our program (function names, global variable names...) and choosing the 'main' function.



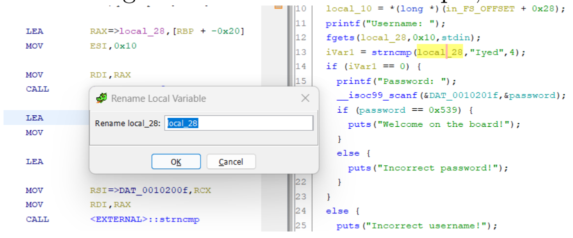
This is the disassembly + decompilation of our 'main' function.



```
                          ****************************************
                          *                      FUNCTION
                          ****************************************
                          undefined8 __stdcall main(void)
        undefined8          RAX:8          <RETURN>
        undefined8          Stack[-0x10]:8 local_10

        undefined1          Stack[-0x28]:1 local_28

                          main

00101189 55           PUSH      RBP
0010118a 48 89 e5     MOV       RBP,RSP
0010118d 48 83 ec 20  SUB       RSP,0x20
00101191 64 48 8b     MOV       RAX,qword ptr FS:[0x28]
         04 25 28
         00 00 00
0010119a 48 89 45 f8  MOV       qword ptr [RBP + local_10],RAX
0010119e 31 c0        XOR       EAX,EAX
001011a0 48 8d 05     LEA       RAX,[s_Username:_00102004]
         5d 0e 00 00
001011a7 48 89 c7     MOV       RDI=>s_Username:_00102004,RAX
001011aa b8 00 00     MOV       EAX,0x0
         00 00
001011af e8 ac fe     CALL      <EXTERNAL>::printf
         ff ff
```

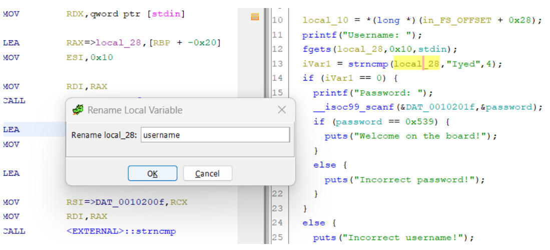
```c
undefined8 main(void)

{
  int iVar1;
  long in_FS_OFFSET;
  char local_28 [24];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  printf("Username: ");
  fgets(local_28,0x10,stdin);
  iVar1 = strncmp(local_28,"Iyed",4);
  if (iVar1 == 0) {
    printf("Password: ");
    __isoc99_scanf(&DAT_0010201f,&password);
    if (password == 0x539) {
      puts("Welcome on the board!");
    }
    else {
      puts("Incorrect password!");
    }
  }
  else {
    puts("Incorrect username!");
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return 0;
}
```

As we would see comparing this to the source code of **login.c** I showed you before, local variable names are lost while global variables are kept just because it's a nonstripped binary and we see global variables being colored in blue. For example, the **username** local variable is no more called **username**.



```
LEA        RAX=>local_28,[RBP + -0x20]
MOV        ESI,0x10

MOV        RDI,RAX
CALL

LEA
MOV

LEA

MOV        RSI=>DAT_0010200f,RCX
MOV        RDI,RAX
CALL       <EXTERNAL>::strncmp
```

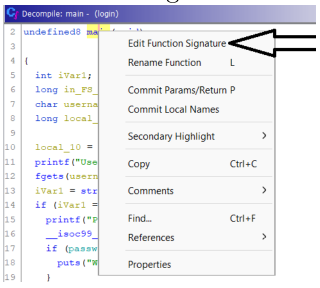
```
local_10 = *(long *)(in_FS_OFFSET + 0x28);
printf("Username: ");
fgets(local_28,0x10,stdin);
iVar1 = strncmp(local_28,"Iyed",4);
if (iVar1 == 0) {
  printf("Password: ");
  __isoc99_scanf(&DAT_0010201f,&password);
  if (password == 0x539) {
    puts("Welcome on the board!");
  }
  else {
    puts("Incorrect password!");
  }
}
else {
  puts("Incorrect username!");
```



```
MOV        RDX,qword ptr [stdin]

LEA        RAX=>local_28,[RBP + -0x20]
MOV        ESI,0x10

MOV        RDI,RAX
CALL

LEA
MOV

LEA

MOV        RSI=>DAT_0010200f,RCX
MOV        RDI,RAX
CALL       <EXTERNAL>::strncmp
```

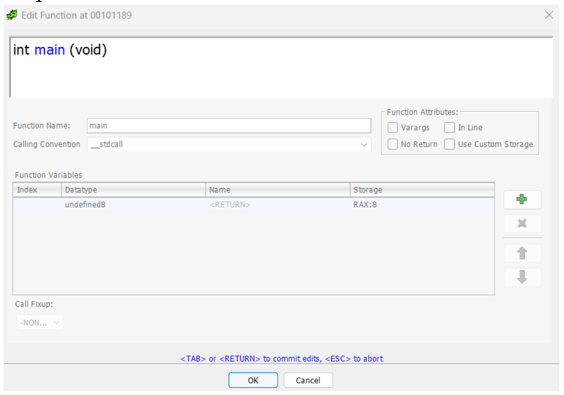
```
local_10 = *(long *)(in_FS_OFFSET + 0x28);
printf("Username: ");
fgets(local_28,0x10,stdin);
iVar1 = strncmp(local_28,"Iyed",4);
if (iVar1 == 0) {
  printf("Password: ");
  __isoc99_scanf(&DAT_0010201f,&password);
  if (password == 0x539) {
    puts("Welcome on the board!");
  }
  else {
    puts("Incorrect password!");
  }
}
else {
  puts("Incorrect username!");
```

Also we could see that the decompiler unfortunately didn't detect the return type of the main function we see **undefined8** just before the function's name. In this case it's useless to edit the main's function signature but I'm doing it for the sake of learning. In order to change the function's signature, we first click on it then right click and choose **Edit Function Signature**.



```
undefined8 ma

{
  int iVar1;
  long in_FS_
  char userna
  long local_

  local_10 =
  printf("Use
  fgets(usern
  iVar1 = str
  if (iVar1 =
    printf("P
    __isoc99_
    if (passw
      puts("W
    }
```

Replace **undefined8** with **int** And hit **OK**.



Edit Function at 00101189

```
int main (void)
```

Function Name: main

Calling Convention: __stdcall

Function Attributes: [ ] Varargs [ ] In Line [ ] No Return [ ] Use Custom Storage

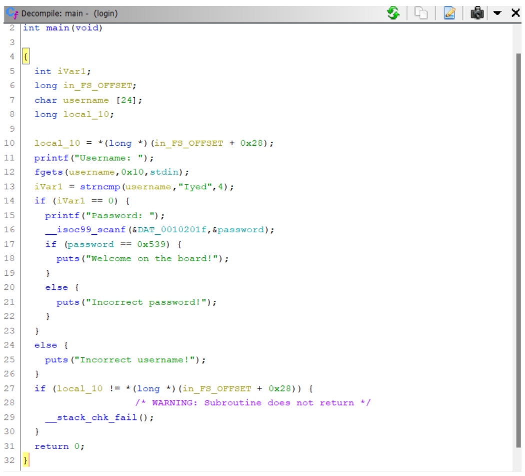
Function Variables

Index	Datatype	Name	Storage
	undefined8	<RETURN>	RAX:8

Call Fixup: -NON...

<TAB> or <RETURN> to commit edits, <ESC> to abort

This is how the decompilation is looking right now:

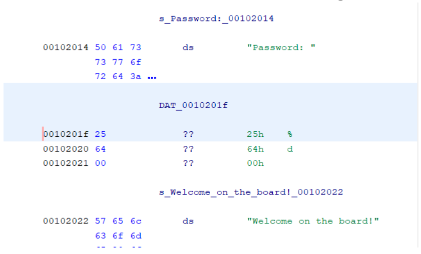


Decompile: main - (login)

```
 2 int main(void)
 3 
 4 {
 5   int iVar1;
 6   long in_FS_OFFSET;
 7   char username [24];
 8   long local_10;
 9 
10   local_10 = *(long *)(in_FS_OFFSET + 0x28);
11   printf("Username: ");
12   fgets(username,0x10,stdin);
13   iVar1 = strncmp(username,"Iyed",4);
14   if (iVar1 == 0) {
15     printf("Password: ");
16     __isoc99_scanf(&DAT_0010201f,&password);
17     if (password == 0x539) {
18       puts("Welcome on the board!");
19     }
20     else {
21       puts("Incorrect password!");
22     }
23   }
24   else {
25     puts("Incorrect username!");
26   }
27   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
28                     /* WARNING: Subroutine does not return */
29     __stack_chk_fail();
30   }
31   return 0;
32 }
```

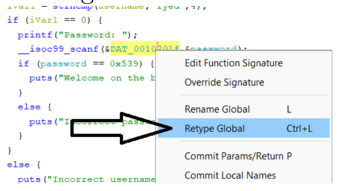
From the line ‘13‘ we know that the ‘username‘ must be ‘Iyed‘ since it’s using the ‘strncmp‘ function to compare the first 4 bytes of our username input with the string ‘Iyed‘ and if the result is 0 (meaning if both strings are equal line ‘14‘) it will print to the screen ‘Password: ‘ and use ‘scanf‘ function with a global variable called **DAT_0010201f** as the first argument and ‘password‘ global variable as the second argument. We know that ‘scanf‘ uses the first variable which is a pointer to a string containing the formats of data to read and place inside the variables mentioned in 2nd, 3rd, 4th... arguments. The decompiler here didn’t detect the type of the global variable **DAT_0010201f** as an array of characters in C otherwise it would have showed the string contained inside of it instead of it’s name. Just like the puts and printf functions the decompilers knows here that their value should be a pointer to a string so he interprets every first arguments of these functions as a string and shows the content of it.

You could double click on a global and Ghidra will show you its content in the disassembly window.



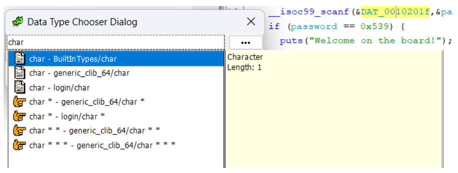
```
                    s_Password:_00102014

00102014 50 61 73       ds         "Password: "
         73 77 6f
         72 64 3a ...

                    DAT_0010201f

0010201f 25             ??         25h    %
00102020 64             ??         64h    d
00102021 00             ??         00h

                    s_Welcome_on_the_board!_00102022

00102022 57 65 6c       ds         "Welcome on the board!"
         63 6f 6d
```

In order to change a variable’s type in Ghidra we first click on it as usual then we either press ‘CTRL + L‘ or right click and choose ‘Retype Global‘ if it was a global or ‘Retype Variable‘ if it was a local.



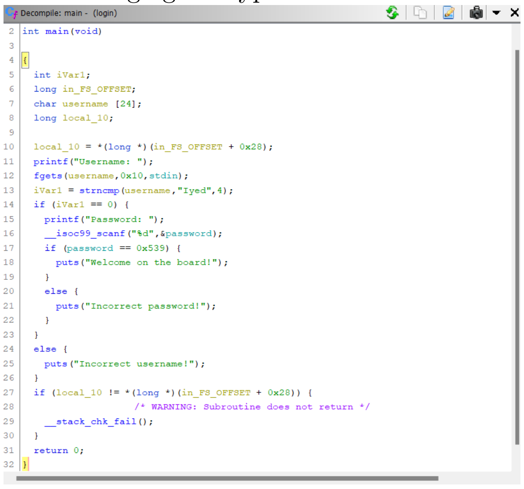
```
if (iVar1 == 0) {
  printf("Password: ");
  __isoc99_scanf(&DAT_0010201f,&password);
  if (password == 0x539) {
    puts("Welcome on the b
  }
  else {
    puts("I
  }
}
else {
  puts("Incorrect username
```

Replace ‘undefined‘ or the previous detected type with the type you guessed and think is correct. You could click on the ‘...‘ to open a menu containing a lot of types and choose the one you desire or starting typing the type to replace with and Ghidra will show different types corresponding to what you are entering at that moment. In this case we will choose ‘char‘ type, we know it’s an array of characters so it technically should be ‘char [size]‘. You could type ‘char [3]‘ since the string in this case is ‘%d‘ and taking count of the null byte or just type ‘char‘ and Ghidra will stop when it hits the first null byte.


Data Type Chooser Dialog
char
char - BuiltInTypes/char
char - generic_clib_64/char
char - login/char
char * - generic_clib_64/char *
char * - login/char *
char * * - generic_clib_64/char * *
char * * * - generic_clib_64/char * * *
Character
Length: 1

```
__isoc99_scanf(&DAT_0010201f,&pa
if (password == 0x539) {
  puts("Welcome on the board!");
```

After changing it’s type here is the new result of the decompilation.



```
 2 int main(void)
 3 
 4 {
 5   int iVar1;
 6   long in_FS_OFFSET;
 7   char username [24];
 8   long local_10;
 9 
10   local_10 = *(long *)(in_FS_OFFSET + 0x28);
11   printf("Username: ");
12   fgets(username,0x10,stdin);
13   iVar1 = strncmp(username,"Iyed",4);
14   if (iVar1 == 0) {
15     printf("Password: ");
16     __isoc99_scanf("%d",&password);
17     if (password == 0x539) {
18       puts("Welcome on the board!");
19     }
20     else {
21       puts("Incorrect password!");
22     }
23   }
24   else {
25     puts("Incorrect username!");
26   }
27   if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
28                     /* WARNING: Subroutine does not return */
29     __stack_chk_fail();
30   }
31   return 0;
32 }
```

Now from lines ‘16‘ ‘17‘ we can conclude that the program is taking an integer from our input, saving it into the global variable ‘password‘ and finally comparing it with the number ‘0x539‘ in hexadecimal so ‘1337‘ in decimal. If they were equal it will output ‘Welcome on the board¡ otherwise it will output ‘Incorrect password¡ on the screen That being said, our input for the password is 1337 since %d is expecting a decimal number as an input. Trying the inputs we found on the program now:

```
iy3dmejri > ~ > CTF > WorkShops > ./login
Username: Iyed
Password: 1337
Welcome on the board!
```

Just a little note here, we could also use the ‘strings‘ command to just show the printable characters that our binary contains. When we used ‘cat‘ to read the file directly we got a lot of gibberish, using the ‘strings‘ command will ignore other non printable characters and show us more clean result. How would this help us? We could make a guess and extract the username from the output of this command. Running it would return a bit of a large output that I can’t show all of it here. But taking a look at the first lines:

```
iy3dmejri > ~ > CTF > WorkShops > strings login
/lib64/ld-linux-x86-64.so.2
fgets
stdin
puts
__stack_chk_fail
__libc_start_main
__cxa_finalize
printf
__isoc99_scanf
strncmp
```

```
libc.so.6
GLIBC_2.7
GLIBC_2.4
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u3UH
Username:
Iyed
Password:
Welcome on the board!
Incorrect password!
Incorrect username!
;*3$"
GCC: (GNU) 12.2.0
.B#>M$#=1
LLu=/
```

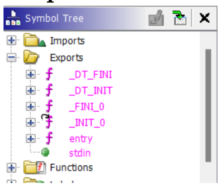
We see **Iyed** under the **Username** string so maybe this is the **username** and you try it to make sure. Unfortunately, the password comparison is used along the assembly instructions with the instruction:

```
0010121d 8b  05  29        MOV         EAX,dword ptr [password]
2e  00  00
00101223 3d  39  05        CMP         EAX,0x539
00  00
```

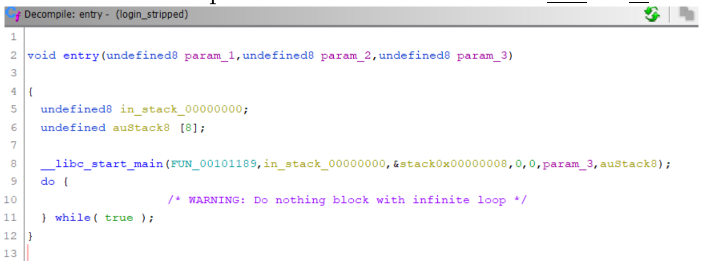
It's an instruction in machine code so it's not printable, sadly we can't get it from the **strings** command. Now taking a look at the case where this same binary was compiled in a way to remove debugging symbols and output a 'stripped' executable. To have such an executable, we will recompile the same **login.c** source code with this command now **gcc -s -o login_stripped login.c**. (**-s option means strip.**)

```
iy3dmejri > ~ > CTF > WorkShops > gcc -s -o login_stripped login.c
iy3dmejri > ~ > CTF > WorkShops > file login_stripped
login: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,

BuildID[sha1]=f1422e49fe53add0b5d3410c6a5c5bd8331ba51e, for
GNU/Linux 4.4.0, stripped
```

Redoing the same steps in order to open the binary with Ghidra and analyze it. Now take a look at the **Exports**:



There is no ‘main‘ function no more because symbols were lost when compiled with strip option. So we need to find the ‘main‘ function and rename it so we could always find it as main and access it easily. In order to do so, we will access the ‘entry‘ function from the ‘Exports‘ as the entry is one of the first functions that the program starts execution from and it is the function that will drop us into the ‘main‘ function with the help of another function called ‘___libc_start_main‘.



```c

void entry(undefined8 param_1,undefined8 param_2,undefined8 param_3)

{
  undefined8 in_stack_00000000;
  undefined auStack8 [8];

  __libc_start_main(FUN_00101189,in_stack_00000000,&stack0x00000008,0,0,param_3,auStack8);
  do {
                    /* WARNING: Do nothing block with infinite loop */
  } while( true );
}

```

The first argument passed to ‘___libc_start_main‘ is the address to the ‘main‘ function. In this case the main function is ‘FUN_00101189‘. We access it, we rename it to main and we just redo the same work in order to get the username and password.

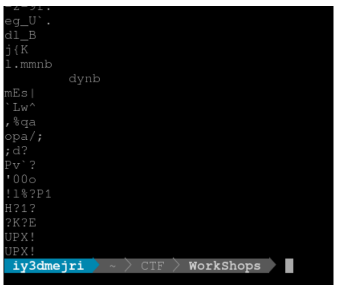
NOTE: Global variable’s names are also lost with the strip option so we need to guess the name of each variable as we have done before with the local variable that turned out to be username.

II - Introduction To Packers:

A packer is basically an executable compressor, which means that it will compress our original program alter its data (maybe encode it or something) then it will add a routine (a block of code) that will decompress that compressed program and jump and start executing it. An example of a famous packer is **UPX**. Back into the **login.c** sample, after compiling the program we could pack it with ‘upx‘ by executing this command ‘upx -o login__packed login‘.

```
iy3dmejri > ~ > CTF > WorkShops > upx -o login\_packed login
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2022
UPX git-69ca63+ Markus Oberhumer, Laszlo Molnar & John Reiser   Oct 28th 2022

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     20904 ->      8168   39.07%   linux/amd64   login\_packed

Packed 1 file.
iy3dmejri > ~ > CTF > WorkShops > ll
total 44
drwxr-xr-x 2 iy3dmejri users  4096 Nov 17 20:23 .
drwxr-xr-x 6 iy3dmejri users  4096 Nov 16 20:22 ..
-rwxr-xr-x 1 iy3dmejri users 20904 Nov 17 20:23 login
-rw-r--r-- 1 iy3dmejri users   384 Nov 17 08:41 login.c
-rwxr-xr-x 1 iy3dmejri users  8168 Nov 17 20:23 login_packed
```

As we could see after compression, the size decreased from **20904** bytes to ‘8168‘ bytes. On larger binaries, running ‘strings‘ might not be helpful anymore as data has been compressed and altered from its original form, however the ‘strings‘ command might help in detecting that it was packed with ‘upx‘ as this packer injects its signature into the binary.



```
eg_U`.
dl_B
j{K
l.mmnb
        dynb
mEs|
`Lw^
,%qa
opa/;
;d?
Pv`?
'00o
!l%?P1
H?1?
?K?E
UPX!
UPX!
 iy3dmejri  ~  CTF  WorkShops
```

In order to unpack an executable packed with ‘upx‘, we simply use:

```
iy3dmejri > ~ > CTF > WorkShops > upx -d login_packed -o login_unpacked
                       Ultimate Packer for eXecutables
                          Copyright (C) 1996 - 2022
UPX git-69ca63+ Markus Oberhumer, Laszlo Molnar & John Reiser   Oct 28th 2022

        File size         Ratio      Format      Name
   --------------------   ------   -----------   -----------
     29727 <-      8168   27.48%   linux/amd64   login_unpacked

Unpacked 1 file.
iy3dmejri > ~ > CTF > WorkShops > ll
total 68
drwxr-xr-x 2 iy3dmejri users  4096 Nov 17 20:27 .
drwxr-xr-x 6 iy3dmejri users  4096 Nov 16 20:22 ..
-rwxr-xr-x 1 iy3dmejri users 20904 Nov 17 20:23 login
-rw-r--r-- 1 iy3dmejri users   384 Nov 17 08:41 login.c
-rwxr-xr-x 1 iy3dmejri users  8168 Nov 17 20:23 login_packed
-rwxr-xr-x 1 iy3dmejri users 20904 Nov 17 20:23 login_unpacked
```

NOTE: Sometimes upx might ask for a binary with greater size in order to do the compressing job correctly, to do so you could compile your program and output a statically linked executable instead a dynamically linked one as in statically linked binaries size is larger due to having library functions merged with the code of our program

Packers are more complicated than just that and sometimes it could be so painful to unpack it, especially if it was a very good custom packer. But generally what we do is start the executable in a debugger (let’s say gdb) and place a breakpoint when the packer starts executing and then once it starts executing the real program you could set a breakpoint there (just to easily reaccess that point and not follow the packer’s execution flow again) then dump the binary at that state and reverse engineer it as we normally do.