

ENIGMA UNVEILED

Solution:

Downloading the Challenge Binary

- The first step was to download the challenge binary from the provided link or platform.

Analyzing the File Format

- Used the `file` command to determine the type of binary:



```

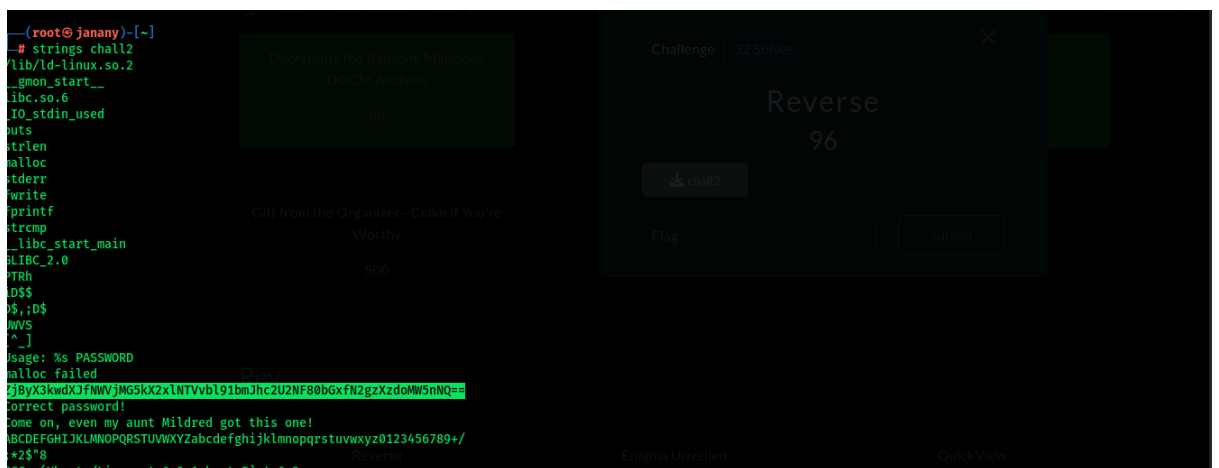
(root@janany)-[~]
# file chall2
chall2: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d6c6d7850110cf50e0b6b3c73530197110a6b6d8, for GNU/Linux 3.2.0, not stripped

```

The output revealed that the binary is in **ELF format**, meaning it's a Linux-based executable file.

Inspecting the Binary for Strings

- As this was the first challenge in reversing, instead of diving into advanced tools like IDA, I opted for a simpler approach using the `strings` command:



```

(root@janany)-[~]
# strings chall2
/lib/ld-linux.so.2
_gmon_start_
libc.so.6
_IO_stdin_used
puts
strlen
malloc
stderr
fwrite
printf
strcmp
libc_start_main
ELIBC_2.0
PTRh
ID$$
P$,;ID$
MVS
^_]
Usage: %s PASSWORD
malloc failed
3ByX3kwdXJfNWVjMG5kX2xINTVvb191bmJhc2U2NF80bGxN2gzXzdoMW5nNQ==
Correct password!
Come on, even my aunt Mildred got this one!
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
+2$*8

```

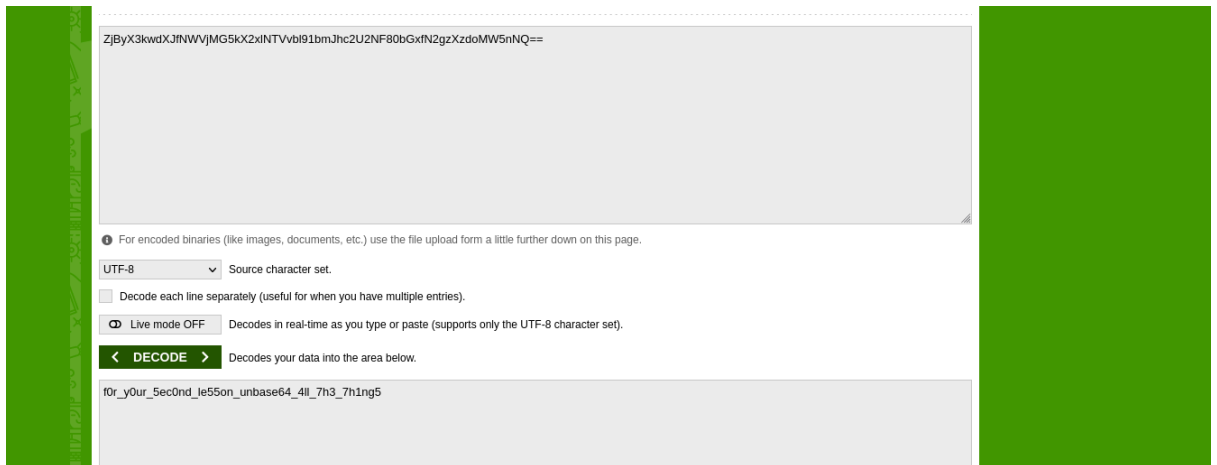
- The command displayed all human-readable strings within the binary.

Identifying the Flag

- While scanning the output of `strings`, I spotted a **base64-encoded string** that looked like the potential flag.

Decoding the Flag

- Copied the base64-encoded string and decoded it using an online decoder (or via a command-line tool like `base64`):



ZjByX3kwdXJfNWVjMG5kX2xINTVvb91bmJhc2U2NF80bGxN2gzXzdoMW5nNQ==

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

UTF-8 Source character set.

☐ Decode each line separately (useful for when you have multiple entries).

☒ Live mode OFF Decodes in real-time as you type or paste (supports only the UTF-8 character set).

< DECODE > Decodes your data into the area below.

f0r_y0ur_5ec0nd_le55on_unbase64_4ll_7h3_7h1ng5

- The decoded output revealed the flag!

Reverse

Solution:

Downloading the Challenge Binary

- The first step was to download the challenge binary from the provided link or platform.

Analyzing the File Format

- Used the `file` command to determine the type of binary:

```
(root@janany)~# file reverse
reverse: ELF 64-bit LSB pie executable, x86_64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d6c6d7850110cf50e0beb3c73539197110a6b6d8, for GNU/Linux 3.2.0, not stripped
```

The output revealed that the binary is in ELF format, meaning it's a Linux-based executable file.

Inspecting the Binary for Strings

Inspecting the Binary for Strings

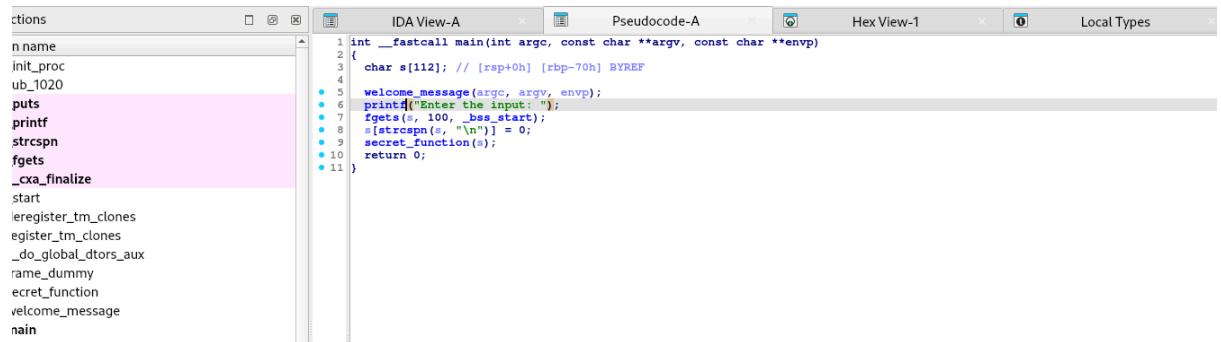
- I used the `strings` command and unfortunately no interesting information I could find

Opening the Binary in IDA

- Loaded the binary into **IDA Pro** for deeper analysis.

Navigating to the **main** Function

- Using IDA's decompiler, I went straight to the **main** function, which often contains essential logic for program execution.
- While analyzing the **main** function, I noticed a call to an **interesting secret function**. This function caught my attention as it wasn't immediately apparent from **main**.

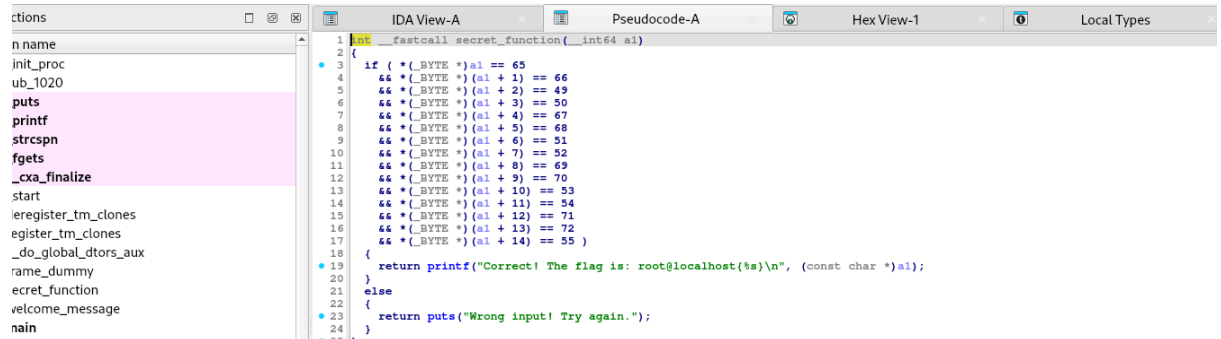


```

1 int __fastcall main(int argc, const char **argv, const char **envp)
2 {
3     char s[112]; // [rsp+0h] [rbp-70h] BYREF
4
5     welcome_message(argc, argv, envp);
6     printf("Enter the input: ");
7     fgets(s, 100, _bss_start);
8     if (strcspn(s, "\n") == 0)
9     {
10         secret_function(s);
11     }
12     return 0;
13 }
  
```

Analyzing the Secret Function

- Decompiled the secret function using IDA's pseudo-code view to understand its logic.
- Within the function, I found a **sequence of random hexadecimal values** stored in an array. This was the critical clue indicating the flag might be constructed from these values.



```

1 int __fastcall secret_function(_int64 a1)
2 {
3     if ( *((_BYTE *)a1) == 65
4         && *((_BYTE *)a1 + 1) == 66
5         && *((_BYTE *)a1 + 2) == 49
6         && *((_BYTE *)a1 + 3) == 50
7         && *((_BYTE *)a1 + 4) == 67
8         && *((_BYTE *)a1 + 5) == 68
9         && *((_BYTE *)a1 + 6) == 51
10        && *((_BYTE *)a1 + 7) == 52
11        && *((_BYTE *)a1 + 8) == 69
12        && *((_BYTE *)a1 + 9) == 70
13        && *((_BYTE *)a1 + 10) == 53
14        && *((_BYTE *)a1 + 11) == 54
15        && *((_BYTE *)a1 + 12) == 71
16        && *((_BYTE *)a1 + 13) == 72
17        && *((_BYTE *)a1 + 14) == 55 )
18     {
19         return printf("Correct! The flag is: root@localhost{&s}\n", (const char *)a1);
20     }
21     else
22     {
23         return puts("Wrong input! Try again.");
24     }
25 }
  
```

Converting Hex Values to ASCII

```

int __fastcall secret_function(__int64 a1)
{
    if ( *(_BYTE *)a1 == 'A'
        && *(_BYTE *) (a1 + '\x01') == 'B'
        && *(_BYTE *) (a1 + 2) == '1'
        && *(_BYTE *) (a1 + 3) == '2'
        && *(_BYTE *) (a1 + 4) == 'C'
        && *(_BYTE *) (a1 + 5) == 'D'
        && *(_BYTE *) (a1 + 6) == '3'
        && *(_BYTE *) (a1 + 7) == '4'
        && *(_BYTE *) (a1 + 8) == 'E'
        && *(_BYTE *) (a1 + 9) == 'F'
        && *(_BYTE *) (a1 + 10) == '5'
        && *(_BYTE *) (a1 + 11) == '6'
        && *(_BYTE *) (a1 + 12) == 'G'
        && *(_BYTE *) (a1 + 13) == 'H'
        && *(_BYTE *) (a1 + 14) == '7' )
    {
        return printf("Correct! The flag is: root@loca
    }
    else
    {
        return puts("Wrong input! Try again.");
    }
}

```

Constructing the Flag

- Placed the decoded string within the given flag format `root@localhost{}`.

Submitting the Flag

- Submitted the constructed flag and successfully completed the challenge.