easycrackme

Introduction

Category: Reverse Engineering
Points: 482
Solves: 10

Honestly, I was very happy when I saw the word 'easy' in the challenge name. Unfortunately for me, it was far from easy! However, after the challenge, I realised in retrospect that the challenge can indeed be very easy if you know what you are doing.

This challenge requires us to enter a key that will undergo six series of checks defined in the binary. Once we find a key that can pass all six checks, we can submit the key as the flag and obtain points for this challenge. The last three checks make use a series of hexadecimal decoding, XOR and base64 decoding.

I believe there was a little hiccup in the creation of the binary, as multiple different inputs were able to pass check 4, which wasn't intentional. I personally got a non-intended answer and luckily managed to get the points by opening a ticket. Nonetheless, in the write-up below, I will attempt to solve for the intended flag as well, with the newfound knowledge that check 4 was supposed to be a hexadecimal decode function.

Solving the challenge

Initial reconnaissance

The challenge prompt simply provides us with the easycrackme binary, which we will be using to find the flag.

We can start analysing the file by using the `file` command in our terminal, which lets us know that it's an ELF executable. We can use multiple tools to disassemble and decompile this binary -- I personally used Ghidra and radare2.

Moving on, we can also use the `strings` command to see what we are able to extract from the binary. There is a portion of the output that seems to be worth examining:

```
=== Check 1 ===
-- Failed check
=== Check 2 ===
greyhats{
=== Check 3 ===
=== Check 4 ===
olympics
=== Check 5 ===
=== Check 6 ===
tokyo
Tell me the key:
*** Passed all checks ***
*** Submit the key as the flag ***
```

From the output seen above, we can deduce that there are six checks packed into the binary, and that the key that passes all the checks will be our flag to be submitted. There are also some strings -- `greyhats{`, `olympics` and `tokyo` which seem to be important as well.

Seems good enough! Let's try running the binary to see what we get.

> Tell me the key:

Looks like this is where we enter the string which will undergo the six checks. Seeing how there were some peculiar strings above, and with the knowledge that the key will be the flag for this challenge, I tried my luck with the string `greyhats{tokyo_olympics}`.

> === Check 1 ===
> -- Failed check

Looks like my luck didn't even get me through the first check... Oh well, looks like that's about what we can gather without taking the binary apart. Let's go ahead and dive deeper!

Decompiling the binary

Let's try to decompile the binary first and hope that we are able to make sense of the higher-level code generated. I used Ghidra to decompile the binary into C.

After letting Ghidra analyse the binary, I went on to look for functions that correspond to the six checks. Under Symbol Tree -> Functions, I found quite a number of unnamed functions (`FUN_00101XXX`), so I went ahead to inspect them.

Upon inspection, I found the strings `=== Check 1 ===`, `=== Check 2 ===`, ... all the way to `=== Check 6 ===`. This is where the six checks are located! Let us now look at what each check does.

For each check, I briefly state what I am able to deduce from the code generated by Ghidra, followed by a summary what each check tells us about the key.

Check 1

Uses the `strlen()` function to obtain the number of characters in our input. If it is not 38 characters long, the check fails.

Basically, our key has to be **38 characters long**.

Check 2

Uses the `strncmp()` function to compare the first 9 characters of our key with the string `greyhats{`. If they are not the same, check fails.

In short, our key has to **start with** `greyhats{`.

Check 3

Compares the last character in our input with the character `}`. `strlen()` is called in the process to grab the last character.

In other words, our key has to **end with** `}`.

Checkpoint

Let us summarise what we know so far.

The first three checks are pretty straightforward and tell us about the general format of the key:

- It has 38 characters
- Is in the form of `greyhats{xxx}`, where `xxx` is currently unknown.

Our hints for `xxx` should be in the last three checks. This is where things get a little more complicated.

Check 4 (hexadecimal decode)

As this check is a little more complex, I will represent it in the form of numbered steps.

1. Initialise an empty string, which we can call `strDecode`.
2. Runs a loop that determines the characters that will make up `strDecode`.
    - Let us imagine the key in this format: `greyhats{xxxx_yyyy}`, where `xxx` shall be represented by the string `str1`.
    - The function will look at `str1` 2 characters at a time, to determine each character of `strDecode`. In other words, every 2 characters in `str1` makes up 1 character in `strDecode`.
    - Let us denote the copies of the first and second characters in each 2-character chunk as `c1` and `c2` respectively.
    - The following pseudocode roughly describes the transformation of `c1` and `c2`:

    ```
    if c1 or c2 is a number:
      c1 or c2 -= 48
    else:
      if c1 or c2 is within the range  [a-f] inclusive:
        c1 or c2 -= 87
    ```

    - `c1` is then shifted left by 4 bits.
    - The bitwise OR operator is applied on `c1` and `c2`.
    - The last 8 bits of the result from the previous operation is then added to `strDecode` as a character.
3. Compares `strDecode` with the string `olympics`. If `strDecode` does not contain the string `olympics`, the check fails.

In short, we have to ensure that the `str1` that we provide in the key must end up forming the word `olympics` after the series of transformations. Let us consider which two characters we must enter to add an 'o' into `strDecode`.

The following section is how I managed to get the flag during the competition, when I was unaware of what the function was actually doing. If you wish to solve it the way the creator intended for it to be solved, please skip to the next section.

Inefficient method (unaware of hexadecimal decoding)

We can tell that 'o' has a character code of 111, which is represented in binary form as 0b01101111 (the '0b' in front just denotes that the following numbers are in binary form). Before we proceed, let us first split `c1` and `c2` into smaller parts:

- `c1a` and `c2a` shall be the first 4 bits of `c1` and `c2` respectively
- `c1b` and `c2b` shall be the last 4 bits of `c1` and `c2` respectively.

Now, we must find two characters `c1` and `c2` such that

- `c1b | c2a` = 0b0110
- `c2b` = 0b1111

One point to note is that the character code of any number, from 0 to 9, is simply 48 added to the number. For example, the character code of 5 is (48+5=53). Thus, since the code subtracts 48 from any number, we can use '0' to obtain a transformed `c1` of 0. If `c1` is 0, `c1b` will also be 0, and the conditions become simpler:

- `c2a` gives us 0b0110
- `c2b` gives us 0b1111

Or in simpler terms, `c2` just needs to give us the character code of 'o' after transformation. If we consider the transformation that occurs, we realise that any character that does not fall within the letters a to f will not be transformed. Thus, to get 'o', we can set `c1='0'` and `c2='o'`.

Proper method

Although I managed to get the flag through the previous method, there is a much better way to solve this. All we require is just these 2 simple observations:

- Numbers (0 to 9) are transformed into their numerical values, meaning the character '5' would be transformed into the number 5.
- Letters, from a-f, are transformed into their hexadecimal values, meaning the character 'a' would be transformed into the number 10.

This would have led to the discovery that this check is just performing a hexadecimal decode on `str1`, and it can be solved by converting each character in the string `olympics` to their hexadecimal representations. Thus, we could have gotten the expected `str1` by running this in Python:

```
print(''.join([hex(ord(c))[2:] for c in 'olympics']))
```

And this leaves us with `str1` = '6f6c796d70696373'.

In short, the key must **start with** `greyhats{6f6c796d70696373_`.

Check 5

This check is a simple XOR function, which does the following:

1. Extract the text between the first and second underscore in the key, which we can denote as `str2`.
2. Initialise an empty string, which we can denote as `strDecode2`

3. Perform the following functions on each character of `str2`, and add the result to `strDecode2`
    - If the position of the current character in `str2` is even, XOR the character with 32.
    - If it is odd, XOR the character with 33.
4. Compare `strDecode2` with the string 'in'. If it is not equal, check fails.

As the inverse function of XOR is XOR itself, we only have to XOR 'i' with 32 and 'n' with 33 to decode what `str2` should be.

```
''.join([chr(ord(c) ^ 32 + i) for i, c in enumerate('in')])
```

And we now know `str2` = 'IO'.

Now we know the key **starts with** `greyhats{6f6c796d70696373_IO_`.

Check 6

This check was like the gatekeeper of the flag. It was extremely difficult for me and I spent a few hours just trying to pass this check. However, this is probably not the case for those sharp minds who could identify that this was a base64 decode function.

This function was not so straightforward, as it messed up my Ghidra decompiler. It was such a mess that I had no idea what was going on, so I had to resort to using Radare2's disassembler and debugger and piece together what was going on manually. This was what I could figure out from painstakingly stepping over each instruction one by one. (sad reacts only)

1. Extracts the string between the 2nd underscore and the '}' in the input, which we can denote as `str3`. The flag basically looks like `greyhats{6f6c796d70696373_IO_${str3}` at this point.
2. Checks if the last 3 bits of the length of `str3` is zero. In other words, check if the number of characters in `str3` is a multiple of 4. If it isn't the check fails.
3. Initialises an empty string, which we can denote as `strDecode3`.
4. Copies a lookup table, which will be used in the next step, into memory (storing it in a variable)
5. Goes through a very complicated loop that was a huge pain to read in assembly code.
    - This loop goes through `str3` in 4-character blocks. Let us denote `c1, c2, c3, c4` as the first, second, third and fourth character in the block.
    - It starts off by subtracting 43 from `c1`, then using the number to check for a number in the lookup table. It took me a long time to figure the table out, but here it is (lookup index on the left):
        - 0: 61
        - 4: 63
        - 5 to 14: 52 to 61
        - 23 to 48: 1 to 25
        - 54 to 80: 26 to 51
    - Add the result to a temporary variable, which we shall call `temp1`
    - Shift the `temp1` 6 bits to the left
    - Repeat the lookup process with `c2` - 43.
    - Apply the bitwise OR operator to `temp1` and the result.

    - Check if `c3` is '='. If `c3` is not '=', the lookup process repeats with `c3` - 43, and the result is sent through the bitwise OR operator with `temp1`.
    - Shift `temp1` 6 bits to the left.
    - Repeat the previous 2 steps with `c4`.
    - Take the first 8 bits of `temp1` and add it as a character to `strDecode3`
    - Check if `c3` is '='. If `c3` is not '=', take the next 8 bits of `temp1` and append it to `strDecode3`
    - Repeat the previous step with `c4`
    - Move on to the next 4 characters
6. Compares `strDecode3` with the string 'tokyo'. If `strDecode3` is not 'tokyo', the check fails.

Since I was unaware that it was a base64 decode, I had to solve it manually. The following section describes how I did it character by character. If you wish to see how it is intended to be solved, please skip to the next section.

Inefficient method (unaware of base64 decode)

First, I represented the first three characters of 'tokyo' in its binary form: 0b011101000110111101101011

Afterwards, I split them into 4 segments of 6 bits each, and manually reversed each character. The following is an example of how I reversed the first character in `str3`

- 0b011101
    - 29 in decimal form
    - Index 57 in lookup table
    - `c1` must be `chr(57+43)` = 'd'

Admittedly, this is a very inefficient process, but it got the job done and I managed to get the flag. Of course, the next method is much easier and it would have saved me a lot of hours.

Efficient method

We just have to encode "tokyo" with base64.

```python
import base64
print(base64.b64encode("tokyo"))
```

We now know that `str3` must be 'dG9reW8='

Thus, we now have the full flag: `greyhats{6f6c796d70696373_IO_dG9reW8=}`

Thoughts

This was a very fruitful challenge. It has taught me a lot about the common types of encoding and how they look like when implemented. Hope I will never commit the same mistake ever again!