Flare-On 7: Challenge 1 – fidler.exe

Challenge Author: Nick Harbour

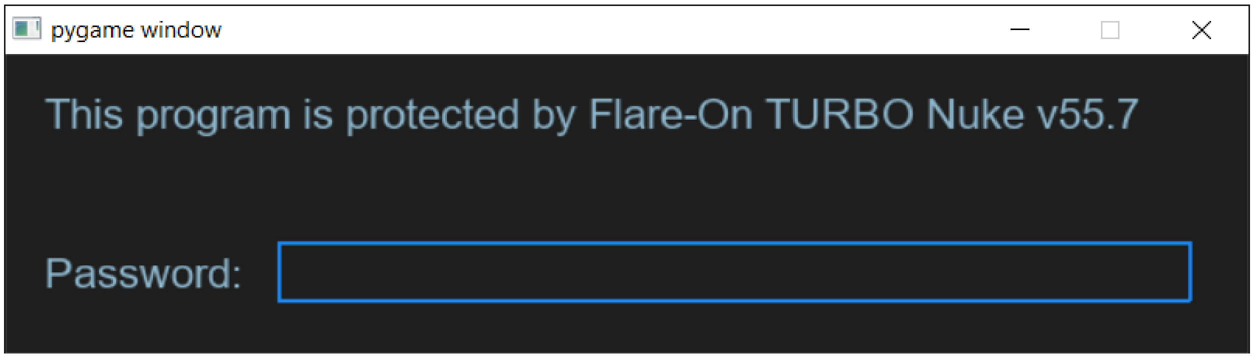
Before downloading the challenge, you were presented with the following prompt:

```
This is a simple game. Win it by any means necessary and the victory screen will
reveal the flag. Enter the flag here on this site to score and move on to the next
level.

This challenge is written in Python and is distributed as a runnable EXE and matching
source code for your convenience. You can run the source code directly on any Python
platform with PyGame if you would prefer.
```

This challenge contains both a compiled binary and Python source code for convenience. If you are running a Windows workstation you may launch the self-contained `fidler.exe` binary without the need to install any Python packages on your system. If you prefer to run this challenge in a Non-windows environment, then you will need to install Python 3 and the PyGame package. Once installed, you may launch the challenge by running the `fidler.py` source file. The other source code file found in this package, `controls.py`, contains code used in the challenge. You do not need to reverse engineer this file to solve this challenge and its contents are not discussed in this document.

When you launch the challenge, you see the following prompt:

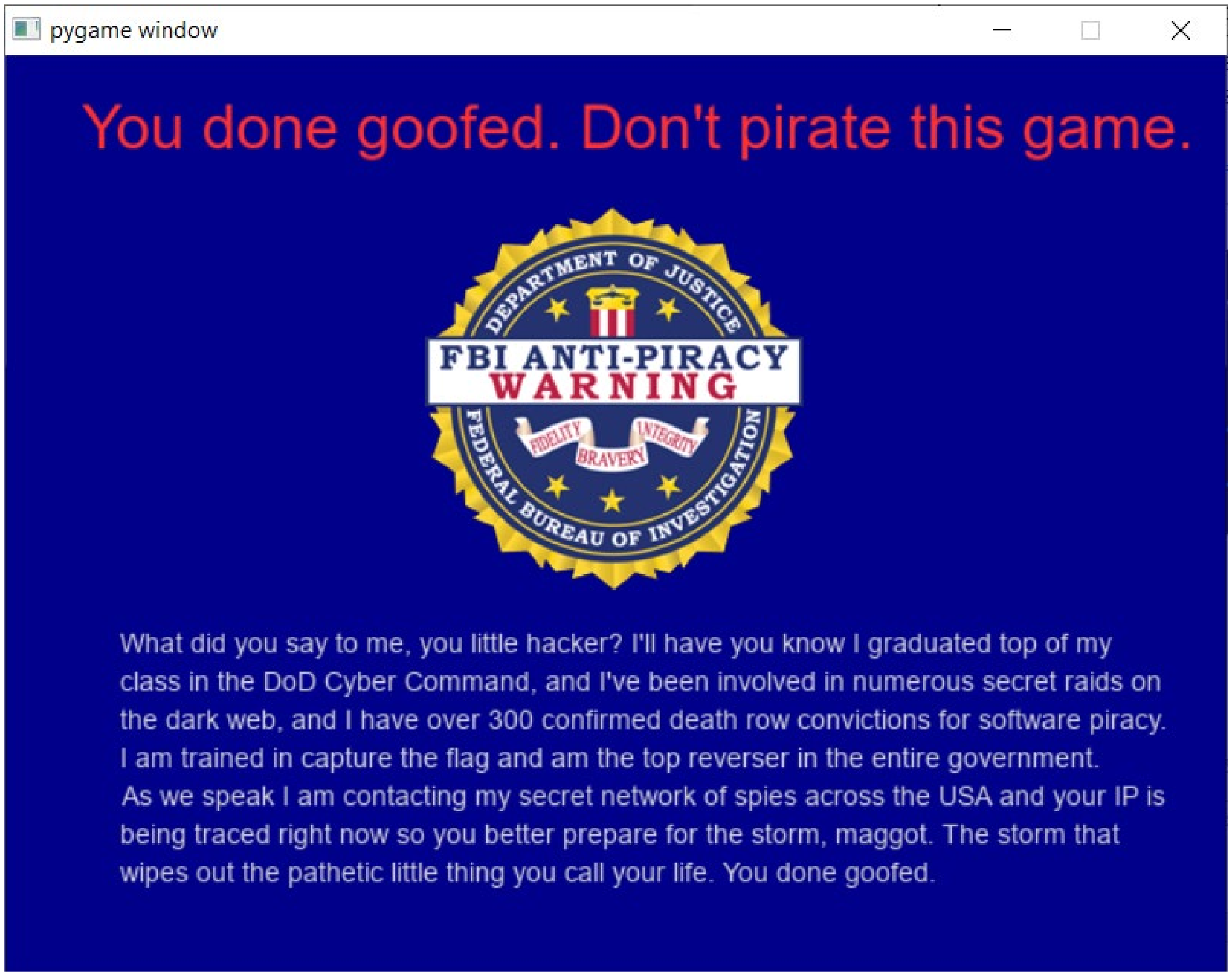


This program is protected by Flare-On TURBO Nuke v55.7

Password:

Figure 1: Password Prompt

If you click into the blue textbox and enter the incorrect password, you are presented with the following error screen:



You done goofed. Don't pirate this game.

What did you say to me, you little hacker? I'll have you know I graduated top of my class in the DoD Cyber Command, and I've been involved in numerous secret raids on the dark web, and I have over 300 confirmed death row convictions for software piracy. I am trained in capture the flag and am the top reverser in the entire government. As we speak I am contacting my secret network of spies across the USA and your IP is being traced right now so you better prepare for the storm, maggot. The storm that wipes out the pathetic little thing you call your life. You done goofed.

Figure 2: Password Error Screen

This error screen contains a modified form of the famous "Navy Seal Copypasta"[1]. This screen represents the end of the program and you must try again with the correct password to proceed to the next stage of the challenge. To determine the correct password, let's start by examining the source code file `fidler.py`.

At the bottom of `fidler.py` is the `main()` function that dictates the overall flow of the program. This relevant code is shown below in Figure 3.

```
def main():
    if password_screen():
        game_screen()
    else:
        password_fail_screen()
    pg.quit()

if __name__ == '__main__':
```

[1] https://knowyourmeme.com/memes/navy-seal-copypasta

```
    main()
```

Figure 3: main() function

The `main()` function describes the overall flow of the program. Luckily, the flow is very simple. It calls the function `password_screen()` first, and if that returns `True` then it proceeds to the `game_screen()`. If it fails, it proceeds instead to `password_fail_screen()`. The `password_screen()` function displays the prompt shown in Figure 1 and collects and validates the input. Lines 31-35 from the `password_screen()` function, shown below in Figure 4, show the logic that occurs when input is submitted from the password text box.

```
        if input_box.submitted:
            if password_check(input_box.text):
                return True
            else:
                return False
```

Figure 4: Password Input Check

In this fragment of code, the result of the `password_check()` function must return `True` for the function to return `True` and the challenge to proceed to the game screen. This fragment passes the text from the password input text box to the `password_check()` function as a parameter. Let's examine the `password_check()` function to learn how it detects a valid password. The function is found on lines 9-12 and is shown below in Figure 5.

```
def password_check(input):
    altered_key = 'hiptu'
    key = ''.join([chr(ord(x) - 1) for x in altered_key])
    return input == key
```

Figure 5: password_check() function

The `password_check()` function builds a key value then compares the input text to that key. It returns the truth value of the string comparison. The first line of the function defines a variable intuitively named `altered_key` which is assigned the value "`hiptu`". In the next line of code, the key variable is created by subtracting 1 from every character in the `altered_key` variable. Since this is a simple algorithm and a short key, there is no need to write a special script to decode this value. If we subtract 1 from each character value in the string "`hiptu`" the result is "`ghost`". Enter "`ghost`" into the password field to proceed to the game screen shown below in Figure 6.

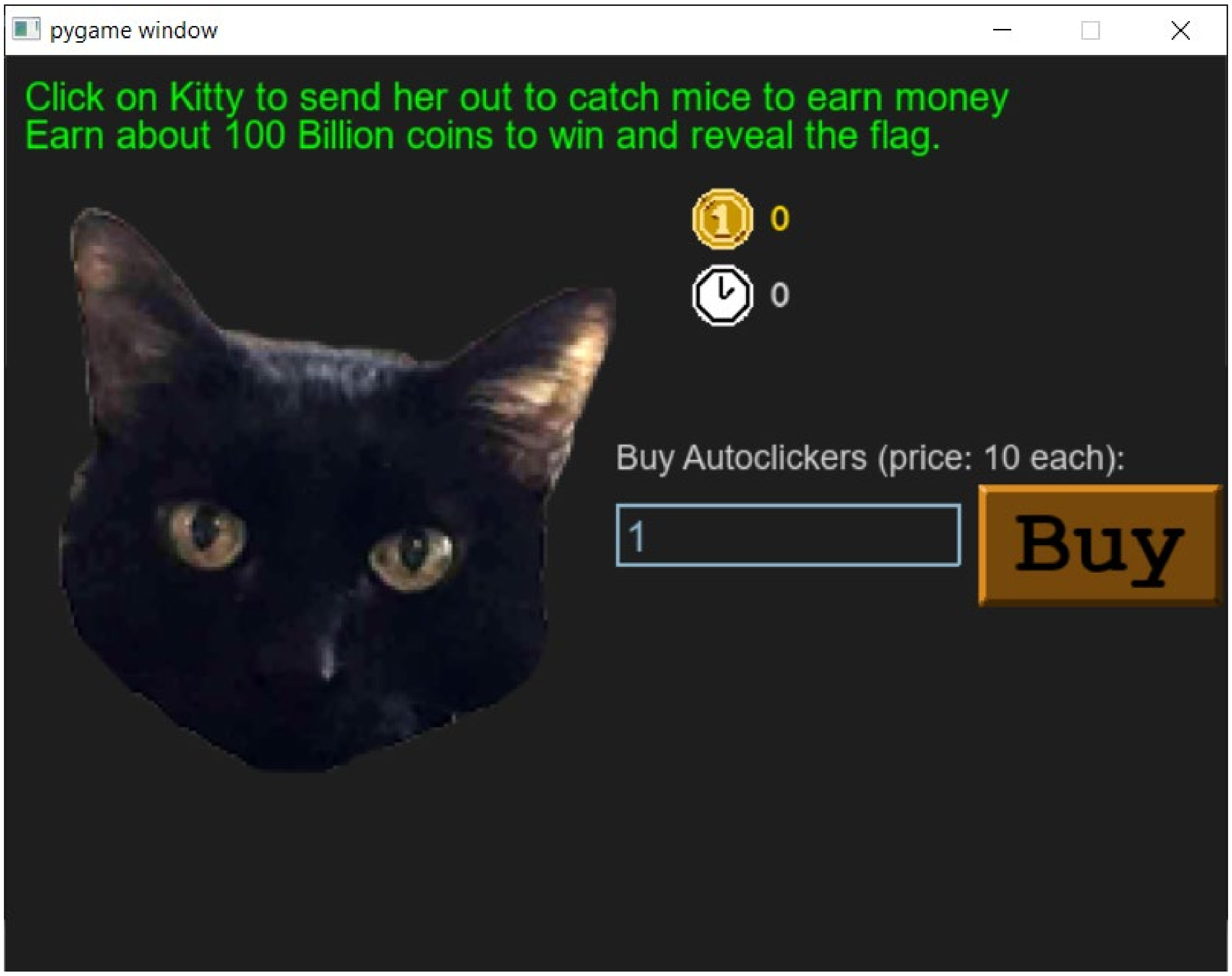

pygame window
Click on Kitty to send her out to catch mice to earn money
Earn about 100 Billion coins to win and reveal the flag.
0
0
Buy Autoclickers (price: 10 each):
1
Buy

Figure 6: Game Screen

The most straightforward path to victory is simply to play the game. If you click on the cat ten times, you will have enough coins to buy one auto-clicker. Each autoclicker will click the cat automatically once per second. Repeat this until you have at least ten autoclickers, then buy ten autoclickers at a time. Buy ten at a time until you're earning enough money to buy 100 at a time, then 1000, etc. After a few minutes of repeatedly re-investing in autoclickers you should hit the 100-Billion-coin mark and the game will automatically advance to the victory screen with the appropriate flag shown below in Figure 10. For a detailed explanation of the game logic to derive a manual solution, please continue reading.

Lines 146-152 from `fidler.py` in the `game_screen()` function define the condition required to achieve victory and advance to the victory screen showing the flag. The relevant code is shown below in Figure 7.

```
    while not done:
        target_amount = (2**36) + (2**35)
        if current_coins > (target_amount - 2**20):
            while current_coins >= (target_amount + 2**20):
```

```
            current_coins -= 2**20
        victory_screen(int(current_coins / 10**8))
        return
```

Figure 7: Victory Condition Game Logic

The code from Figure 7 begins by defining a variable named `target_amount` which is set to the expression “(2**36) + (2**35)”. This expression evaluates to 103,079,215,104. This is the exact number of coins needed to proceed to the victory screen. The next three lines of code detect if the current coin amount is beyond the target amount minus “2**20” (1,048,576). If the current coin amount is above this threshold, the code enters a loop which will subtract 1,048,576 from the current coin amount until the result is less than 1,048,576 above the target value. This approach will not produce a consistent `current_coins` value, but it will guarantee that the value is between 103,078,166,528 and 103,079,215,104.

The second to last line of code shown in Figure 7 shows the program calling the `victory_screen()` function with a parameter that is produced from the `current_coins` variable. It divides the current_coins amount by “10**8”, which evaluates to 100,000,000. Since the `current_coins` variable will be a number between 103,078,166,528 and 103,079,215,104 at this point in the code, the result of this division will always be the number 1,030.

Line 215 of the victory_screen() function is responsible for changing the text of the placeholder flag control with the value of the actual final flag. This line of code is shown here:

```
    flag_content_label.change_text(decode_flag(token))
```

Figure 8: Flag Text Change

Examining the code in Figure 8 above it is clear that the next place we need to examine next is the decode_flag() function. We know that the token value passed to this function will be 1030 as described above.

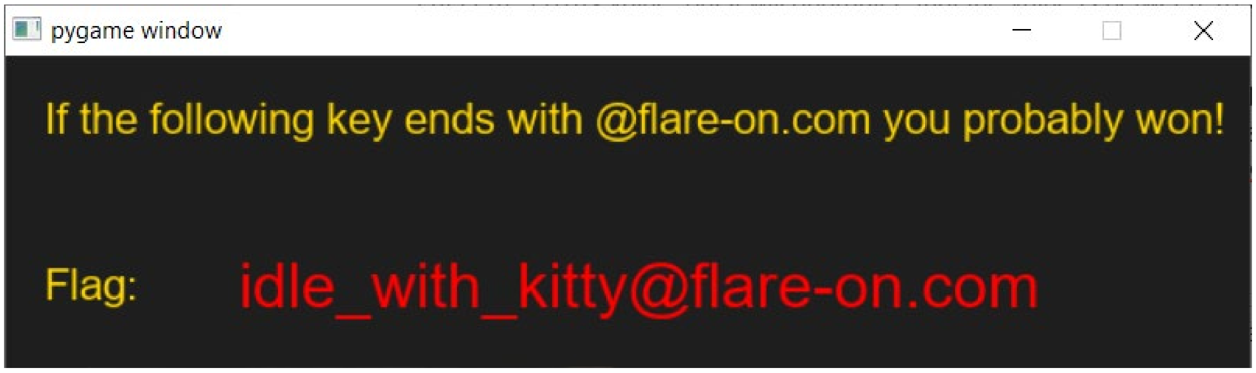
```
def decode_flag(frob):
    last_value = frob
    encoded_flag = [1135, 1038, 1126, 1028, 1117, 1071, 1094, 1077, 1121,
1087, 1110, 1092, 1072, 1095, 1090, 1027,
                    1127, 1040, 1137, 1030, 1127, 1099, 1062, 1101, 1123,
1027, 1136, 1054]
    decoded_flag = []

    for i in range(len(encoded_flag)):
        c = encoded_flag[i]
        val = (c - ((i%2)*1 + (i%3)*2)) ^ last_value
        decoded_flag.append(val)
        last_value = c

    return ''.join([chr(x) for x in decoded_flag])
```

Figure 9: decode_flag() function

The `decode_flag()` function contains an array of values named `encoded_flag`. It will use the token value passed to this function (which was 1030) to decode this value and store it into the variable `decoded_flag`. The For loop found on lines 194-198 is responsible for iterating through the encoded flag and performing the custom operation to each value, storing the result in the `decoded_flag` array. Each value in the encoded flag is manipulated by first subtracting a value computed based on the index in the array the value occurs, represented by the expression "(i%2)*1 + (i%3)*2". It is important to remember for this expression that the '%' operator represents the Modulo operation, calculating the remainder of an integer division. The value is then XORed with the last value which was computed in the loop. During the first iteration of the loop, the value that it is XORed with is the token value 1030. If you perform this loop over each value in the array and join them together with the text join statement on line 200 you will produce the string "`idle_with_kitty@flare-on.com`". The result of this function is visible on the victory screen as shown below in Figure 10.



If the following key ends with @flare-on.com you probably won!

Flag: idle_with_kitty@flare-on.com

Figure 10: Victory Screen with Flag