FLARE

Flare-On 7: Challenge 3 – Wednesday (mydude.exe)

Challenge Author: Blaine Stancill (@MalwareMechanic)

Introduction

The challenge ZIP file (`wednesday.zip`) contains a folder named `wednesday` and a file named `README.txt`. As seen in Figure 1, `README.txt` provides a cryptic message "BE THE WEDNESDAY" followed by a column of single letters, along with the word DUDE, and a few directions to get us started.

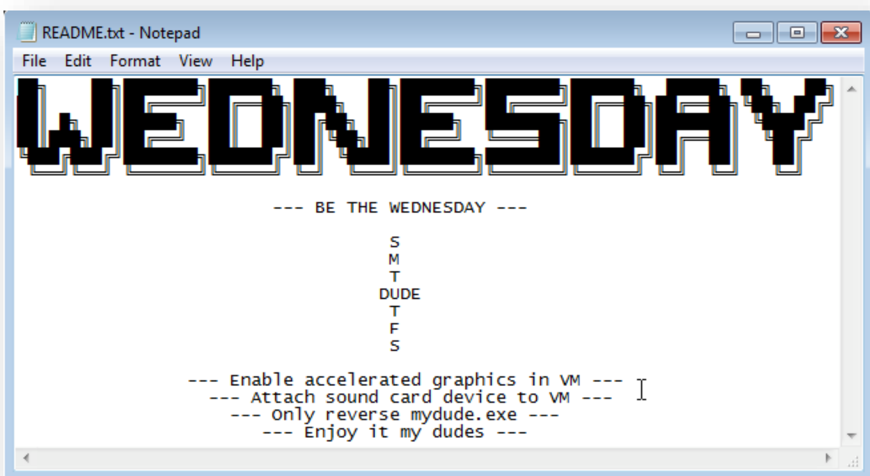

Figure 1: README.txt contents

Opening the folder `wednesday` reveals a folder named `data` and multiple files. The `data` folder contains additional folders with file types such as fonts, graphics, and sounds. Figure 2 displays the contents of the graphics folder named `gfx`. Inside this folder are square tile images with letters that match those in the letter column of `README.txt`. Additionally, we see a sprite sheet named `dude.png` with sprites of a frog who must be the "DUDE" alluded to in `README.txt`.

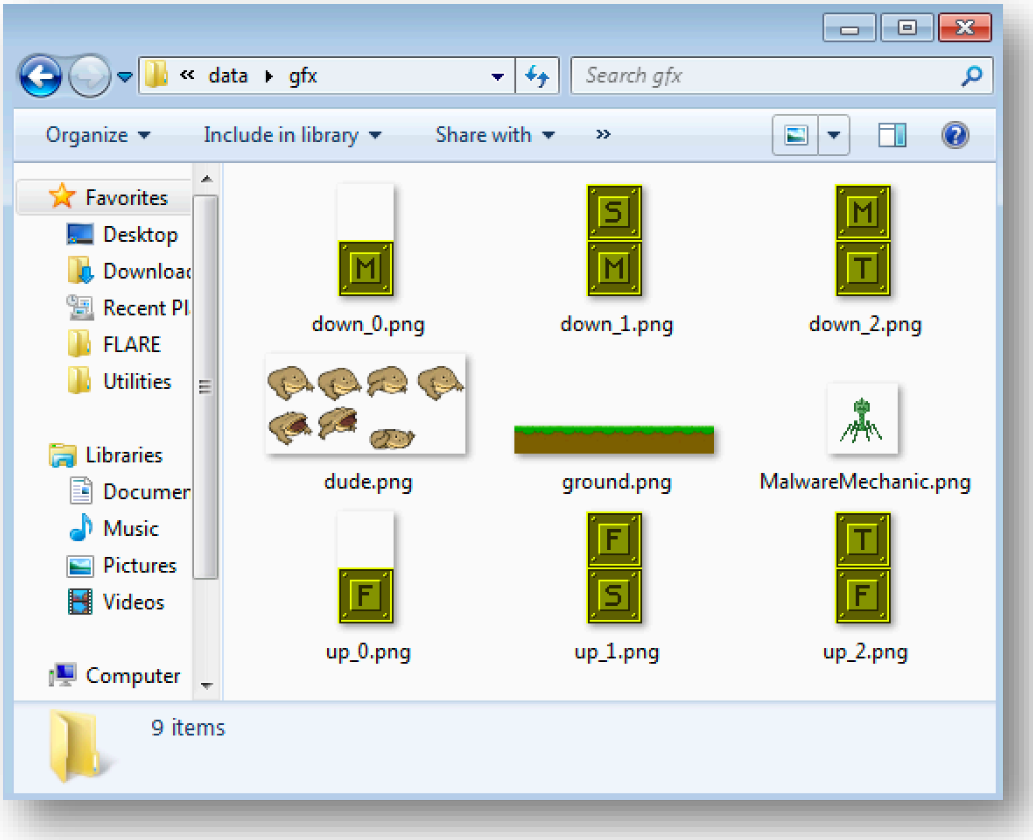

Figure 2: gfx folder contents

At first glance the sheer number of files seems intimidating; however, `README.txt` specified we should only reverse the file `mydude.exe`. We can safely conclude that the other files are merely support files for `mydude.exe` and ignore them.

Game Overview

Executing `mydude.exe` opens a window menu, as seen in Figure 3, displaying a bouncing frog, two menu buttons (DUDE and EXIT), and instructions at the bottom of the window:

- Jump (up arrow)
- Duck (down arrow)
- Quit (ESC)

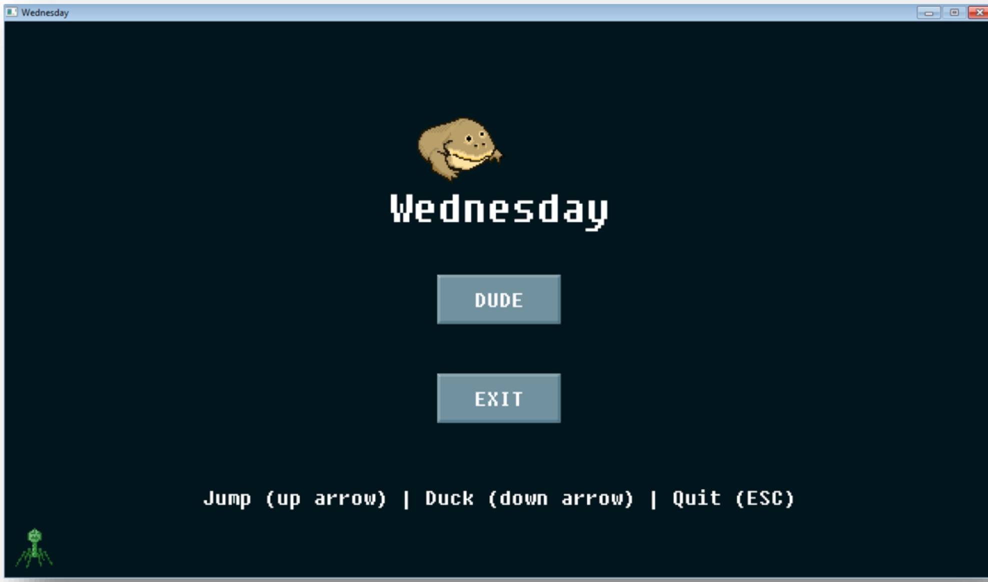

Figure 3: Starting menu for mydude.exe

It appears this is a game! Clicking the "DUDE" button begins the game. As we start to test play, we notice different square tiles, or obstacles, move towards our player character, the "DUDE" frog, as seen in Figure 4.

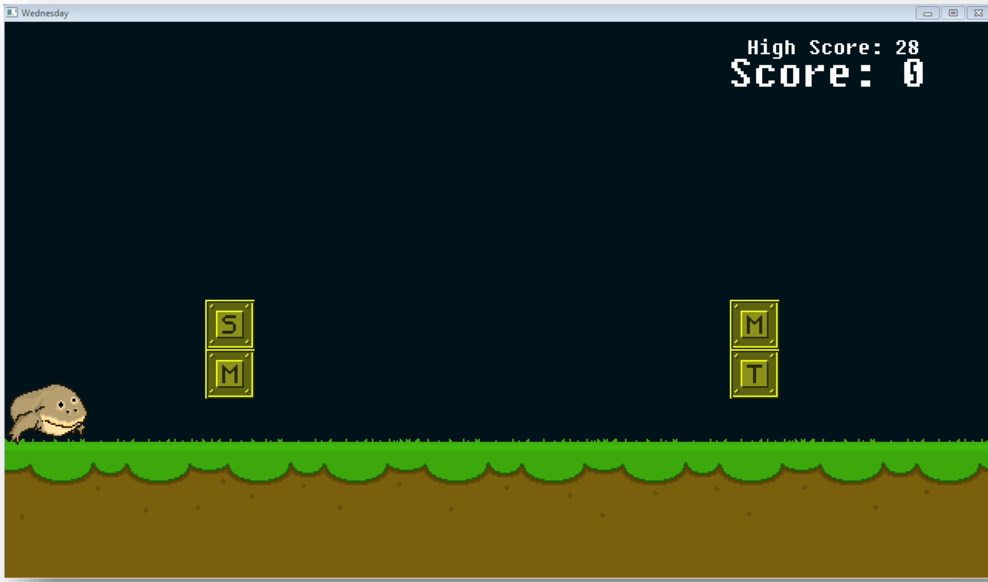

Figure 4: Obstacles moving towards the player character

Our player actions are limited to jumping and ducking as described in the initial menu. After trying to jump/duck the obstacles careening towards our player character, we notice there is a specific way we must jump/duck. Looking back at `README.txt`, we were provided a hint on how to play: "BE THE WEDNESDAY". We were also provided a column of letters with the word "DUDE" embedded in the middle. Thus, we can hypothesize the obstacle letters represent days of the week (i.e., Sunday through Saturday) and our player character is the day Wednesday. As such, we must jump/duck to "BE THE WEDNESDAY". Testing our hypothesis, we jump over the obstacle representing Monday (i.e., `M`), as seen in Figure 5, and subsequently fail/die and must start over.

Figure 5: Incorrectly jumping over Monday obstacle

Testing our hypothesis again, we duck under the obstacle representing Sunday and Monday (i.e., $\frac{S}{M}$) as seen in Figure 6. This time we don't die and our score increases to one.

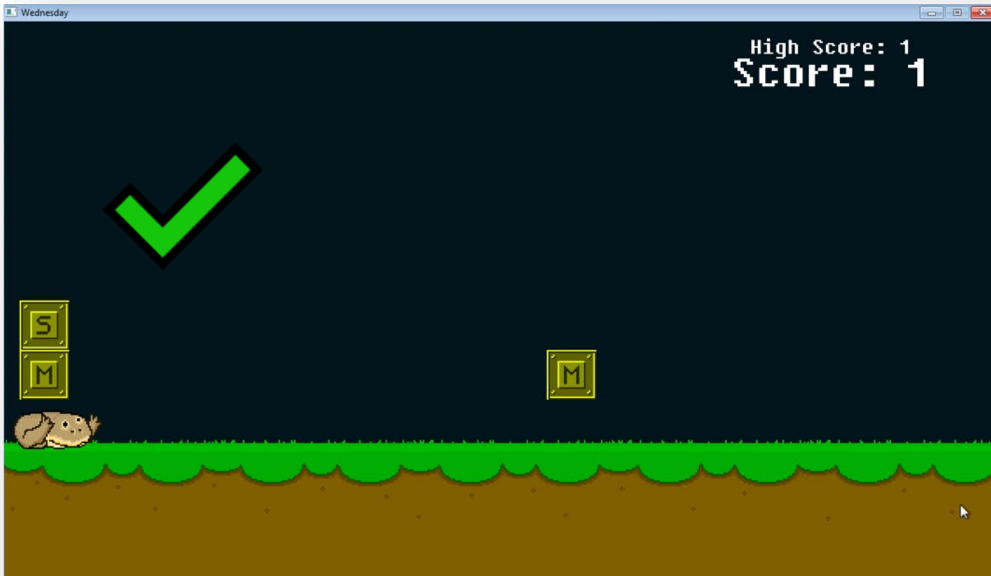

Figure 6: Correctly ducking under Sunday/Monday obstacle

We can safely assume we need to:

- Duck under tiles labeled: M, $\frac{\mathrm{S}}{\mathrm{M}}$, and $\frac{\mathrm{M}}{\mathrm{T}}$
- Jump over tiles labeled: F, $\frac{\mathrm{T}}{\mathrm{F}}$, and $\frac{\mathrm{F}}{\mathrm{S}}$

We can strengthen our assumption by looking back at the file names in the `gfx` folder, seen in Figure 2, and noticing how the tile image names (i.e., "up", "down") correspond to when our player must jump or duck.

Solutions

Now that we know how to play the game, the question is: how do we solve it for the Flare-On Challenge? I've seen three different methods to solve this challenge:

1. Find the challenge flag buffer
2. Patch the file and run
3. Create a bot

The first two methods will be covered below. Creating a bot is out of scope and left as an exercise for the reader.

Solution 1 – What day is it?

Before diving into our disassembler of choice (I'll be using IDA Pro), we perform basic static analysis on `mydude.exe` to determine points of interest. Figure 7 outlines interesting strings we can pivot on in our disassembler.

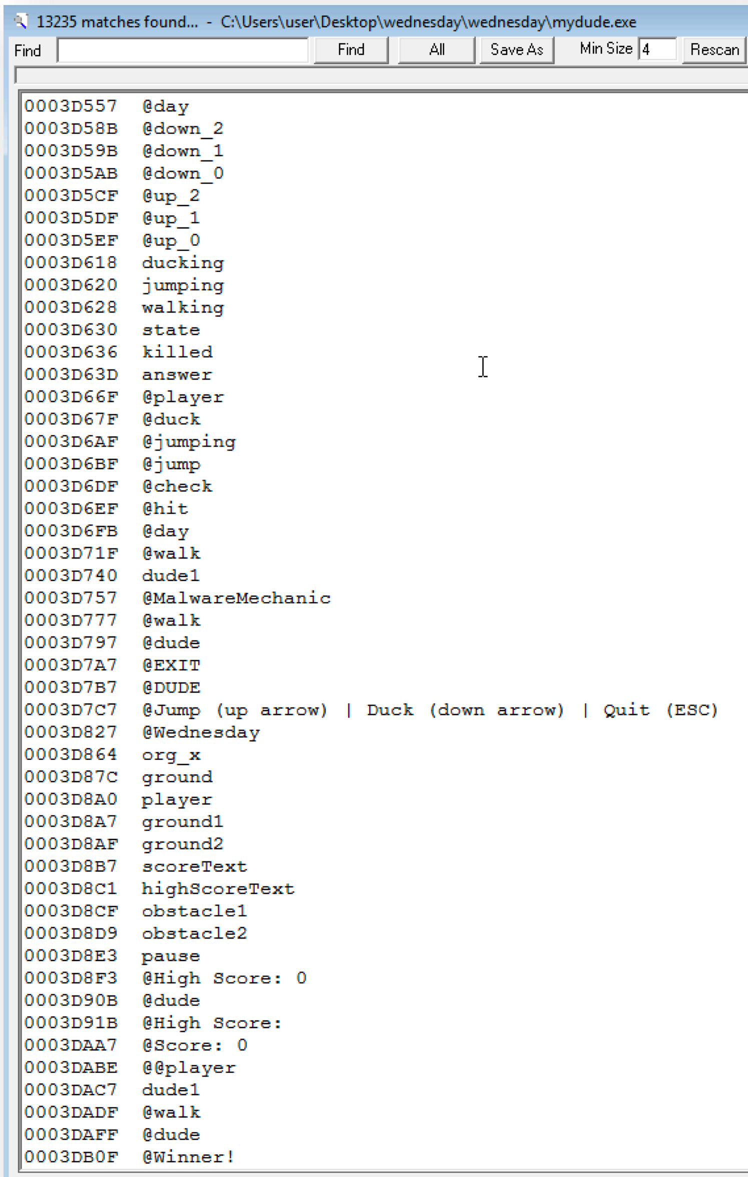

Figure 7: Interesting strings in `mydude.exe`

Opening `mydude.exe` in IDA Pro shows it contains DWARF information (i.e., debug information) that IDA can parse. Contained in the DWARF information are symbol names for both functions and variables making our disassembly more readable.

Browsing the disassembly and strings, it becomes apparent `mydude.exe` was created in the Nim programming language (https://nim-lang.org/). This is a slight challenge as there aren't many reversing tutorials focused on Nim. However, a good reference on Nim's underlying memory model is http://zevv.nl/nim-memory/.

So where to begin our analysis? Well, having played the game repeatedly we may have noticed there are two obstacles on the screen at a time (obstacle1 and obstacle2 seen in the strings) and the way we have to jump or duck them forms a pattern. The beginning of the pattern is: down, down, up, up, down, down, down, up, down, up, etc.

Since there's a pattern, there's likely an algorithm or static buffer used to assign days to each obstacle before they appear on screen. Let's follow this rabbit hole!

Searching for functions containing the name "Day" turns up the likely candidate function named `@assignDay__cz9bfHkuka9cVFg87ZfWKc8g@8` at virtual address (VA) `0x004317D0`. This function is responsible for assigning a day graphic and collision box to an associated obstacle. The calling convention for this function is `__fastcall` which is the standard calling convention for Nim procedures (see `nimcall` in https://nim-lang.org/docs/manual.html). The function expects two arguments in the registers `ECX` and `EDX`. In the function's first basic block, we see the lower 8-bits of the register `EDX` is compared to the value one determining the branching condition. In either branch location a global pointer to an array of strings is referenced. Outlined below are the strings and their corresponding global pointer VA.

- `0x0043E778: down_0, down_1, down_2`
- `0x0043E7BC: up_0, up_1, up_2`

These strings refer to the filenames referenced in Figure 2 of the day tile images. We now know the register `EDX` is responsible for determining if an obstacle is assigned a "down" or "up" day tile image. We also know from playing the game, the image is important as it also affects what action our player must take (i.e., jump or duck). Our job now is to trace backwards to determine where the value in `EDX` came from.

There are a few paths backward we may take, and I'll describe one of them. Cross-referencing `@assignDay__cz9bfHkuka9cVFg87ZfWKc8g@8` we see the functions below.

- `@reset__SAtOZDlchGyR6ynmbkI6aw@24` at VA `0x00431A20`
- `@init__SAtOZDlchGyR6ynmbkI6aw_2@24` at VA `0x00431AD0`

Let's use the function named `@reset__SAtOZDlchGyR6ynmbkI6aw@24`. Observing the disassembly for this function, as shown in Figure 8, we see the value assigned to `EDX` comes from the value at the stack location `[ESP + 0x40]`. Adjusting for the return address as well as the initial `PUSH` and `SUB` instructions, this value would have been stored on the stack at location `[ESP + 0x10]` from the calling function (i.e., the parent of `@reset__SAtOZDlchGyR6ynmbkI6aw@24`).

```
00431A20  push    esi
00431A21  push    ebx
00431A22  mov     ebx, ecx
00431A24  sub     esp, 24h
00431A27  mov     eax, [esp+30h]
00431A2B  mov     esi, [esp+40h]
00431A2F  mov     byte ptr [ecx+18h], 0
00431A33  mov     [esp+10h], eax
00431A37  mov     eax, [esp+34h]
00431A3B  mov     [esp+14h], eax
00431A3F  mov     eax, [esp+38h]
00431A43  mov     [esp+18h], eax
00431A47  mov     eax, [esp+3Ch]
00431A4B  mov     [esp+1Ch], eax
00431A4F  mov     eax, [ecx+0F8h]
00431A55  mov     byte ptr [eax+18h], 0
00431A59  mov     eax, esi
00431A5B  movsx   edx, al
00431A5E  call    @assignDay__cz9bfHkuka9cVFg87ZfWKc8g@8
```

Figure 8: Disassembly of function `@reset__SAtOZDlchGyR6ynmbkI6aw@24`

Cross-referencing `@reset__SAtOZDlchGyR6ynmbkI6aw@24` we see the functions below.

- `@resetEverything__Q1G0gjmnsnF8mVSgZnKS4w_3@4` at VA `0x00433A50`
- `@update__Arw3f6ryHvqdibU49aaayOg@12` at VA `0x00433D20`

Let's follow the function named `@resetEverything__Q1G0gjmnsnF8mVSgZnKS4w_3@4`. Observing the disassembly (Figure 9), we see the value stored at stack location `[ESP + 0x10]` is determined by two global variables named `_obstacles__Xqz7GG9aS72pTPD9ceUjZPNg` and `_day_index__HImZp3MMPNE3pGzeJ4pUlA`. Interesting!

```
00433CFF  mov     eax, _obstacles__Xqz7GG9aS72pTPD9ceUjZPNg
00433D04  mov     edi, ds:_day_index__HImZp3MMPNE3pGzeJ4pUlA
00433D0A  jmp     loc_433BA6
[...SNIP...]
00433BA6  movsx   eax, byte ptr [eax+edi+8]
00433BAB  mov     ecx, [ebx+40h]
00433BAE  mov     [esp+10h], eax
00433BB2  mov     eax, ds:_TM__V45tF8B8NBcxFcjfe7lhBw_9
00433BB7  mov     [esp], eax
00433BBA  mov     eax, ds:dword_43EB34
00433BBF  mov     [esp+4], eax
00433BC3  mov     eax, ds:dword_43EB38
00433BC8  mov     [esp+8], eax
00433BCC  mov     eax, ds:dword_43EB3C
00433BD1  mov     [esp+0Ch], eax
00433BD5  call    @reset__SAtOZDlchGyR6ynmbkI6aw@24
```

Figure 9: Disassembly of function `@resetEverything__Q1G0gjmnsnF8mVSgZnKS4w_3@4`

Based on the byte-pointer deference at VA `0x00433BA6`, we can guess the "obstacles" global variable is likely an array, with 8-bytes of header data, assuming "day_index" is actually an index. Following the global variable `_obstacles__Xqz7GG9aS72pTPD9ceUjZPNg` at VA `0x0043A860`, we see it points to `_TM__V45tF8B8NBcxFcjfe7lhBw_5` at VA `0x0043EB40`. Navigating to VA `0x0043EB40` presents us with the data shown in Figure 10.

```
0043EB40 28                    _TM__V45tF8B8NBcxFcjfe7lhBw_5 db  28h
0043EB40
0043EB41 01                                  db    1
0043EB42 00                                  db    0
0043EB43 00                                  db    0
0043EB44 28                                  db  28h ; (
0043EB45 01                                  db    1
0043EB46 00                                  db    0
0043EB47 40                                  db  40h ; @
0043EB48 00                                  db    0
0043EB49 00                                  db    0
0043EB4A 01                                  db    1
0043EB4B 01                                  db    1
0043EB4C 00                                  db    0
0043EB4D 00                                  db    0
0043EB4E 00                                  db    0
0043EB4F 01                                  db    1
0043EB50 00                                  db    0
0043EB51 01                                  db    1
0043EB52 01                                  db    1
0043EB53 01                                  db    1
0043EB54 00                                  db    0
0043EB55 01                                  db    1
0043EB56 00                                  db    0
0043EB57 00                                  db    0
0043EB58 00                                  db    0
0043EB59 01                                  db    1
0043EB5A 00                                  db    0
0043EB5B 01                                  db    1
0043EB5C 01                                  db    1
0043EB5D 01                                  db    1
0043EB5E 01                                  db    1
0043EB5F 01                                  db    1
0043EB60 00                                  db    0
0043EB61 01                                  db    1
0043EB62 01                                  db    1
0043EB63 00                                  db    0
0043EB64 01                                  db    1
0043EB65 00                                  db    0
```

Figure 10: Raw Nim sequence data

If we look closely, we observe a structure with 8-bytes of header data followed by bytes of ones and zeros. This is a Nim sequence having the structure below.

```
struct nim_seq {
  DWORD length;
  DWORD reserved;
  BYTE data[];
};
```

Figure 11: Nim sequence structure

Reformatting the data gives us the better visual below.

```
_TM__V45tF8B8NBcxFcjfe7lhBw_5 dd 128h
                dd 40000128h
                db 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1
                db 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1
                db 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1
                db 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0
                db 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1
                db 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1
                db 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1
                db 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1
                db 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1
                db 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1
                db 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0
                db 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0
                db 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1
                db 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0
                db 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1
                db 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1
                db 0, 1, 1, 0, 1, 1, 0, 1
```

Figure 12: Reformatted Nim sequence data

In Figure 12, we see the array has a length of `0x128`, or `296`, representing the number of elements. These elements are ultimately responsible for determining if a down or up day tile image is assigned to an obstacle. It also corresponds exactly to the initial pattern from before: down, down, up, up, down, down, down, up, down, up, etc.

Anytime I see a buffer of ones and zeros I assume it's binary data. Let's convert these bytes of ones and zeros into actual binary to see if it means anything.

Taking the first eight bytes (the size of an ASCII character) forms the binary value `0b00110001`, or `0x31`, which represents the ASCII character '`1`'. The next eight bytes form `0b01110100`, or `0x74`, corresponding to the ASCII character '`t`'. I think we're on to something! Instead of doing the conversion manually, we'll use the Python script below.

```python
import binascii
ones_and_zeros = [0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1,
                   0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1,
                   0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1,
                   1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0,
                   0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1,
                   1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1,
                   0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1,
                   1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1,
                   0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1,
                   1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1,
                   0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0,
                   0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0,
                   0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1,
                   1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0,
                   1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1,
                   1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1,
                   0, 1, 1, 0, 1, 1, 0, 1]

output = ""

def _8bit_chunks(array):
    for i in range(0, len(array), 8):
        yield array[i:i+8]

for chunk in _8bit_chunks(ones_and_zeros):
    tmp = 0
    for i, e in enumerate(chunk[:-1]):
        tmp = (tmp | e) << 1
    tmp |= chunk[-1]
    output += chr(tmp)

print(output)
```

Figure 13: Python decoding script

Executing the Python script in Figure 13 results in the flag:

`1t_i5_wEdn3sd4y_mY_Dud3s@flare-on.com`

We've done it! It turns out our player character was jumping and ducking to the bytes representing the Flare-On Challenge flag!

Solution 2 – Patch it like it's hot!

Perhaps tracking down a static buffer of memory isn't your thing and that's okay. How else would we solve this challenge? After playing it for a bit, you may start wondering how the score gets incremented. Let's track this rabbit down!

Hopping into our favorite disassembler, I'll use IDA Pro, we again utilize the DWARF debug information to our advantage. Searching across all symbol names for the keyword "score" finds the results in Table 1 below.

Symbol Name	Virtual Address	Type
`szScoretext`	`0x0043EAB7`	`String`
`szHighscoretext`	`0x0043EAC1`	`String`
`_high_score__zZRxWe9cBeocEphfWmZaLtA`	`0x00443D60`	`Data`
`_prev_score__55xT1lC51wWU8x2SoheEqg`	`0x00443D64`	`Data`
`_score__h34o6jaI3AO6iOQqLKaqhw`	`0x0044DDB0`	`Data`

Table 1: Symbol names containing the keyword "score"

Based on these names, it appears the current score global variable is stored at VA `0x0044DDB0` with name `_score__h34o6jaI3AO6iOQqLKaqhw` (denoted `_score` for the remainder of this walkthrough). Cross-referencing the global variable `_score`, we see it's used in multiple locations as outlined in Table 2 below.

Address	Reference Instruction
`@onCollide__9byAjE9cSmbSbow3F9cTFQfLg@8:loc_432261`	`mov ebx, ds:_score`
`@onCollide__9byAjE9cSmbSbow3F9cTFQfLg@8+159`	**`mov ds:_score, ebx`**
`@show__Q1G0gjmnsnF8mVSgZnKS4w@4`	`mov ds:_score, 0`
`@resetEverything__Q1G0gjmnsnF8mVSgZnKS4w_3@4+3D`	`mov ds:_score, 0`
`@update__Arw3f6ryHvqdibU49aaayOg@12:loc_433E79`	`mov ecx, ds:_score`
`@update__Arw3f6ryHvqdibU49aaayOg@12+1DC`	`mov ecx, ds:_score`

Table 2: Cross references to _score global variable

The second entry in Table 2 is the only entry in which `_score` is assigned a value. Navigating to VA `0x00432279` within the function named `@onCollide__9byAjE9cSmbSbow3F9cTFQfLg@8`, we see a distinct path that must be taken in order for global variable `_score` to be incremented. Figure 14 below highlights this path.

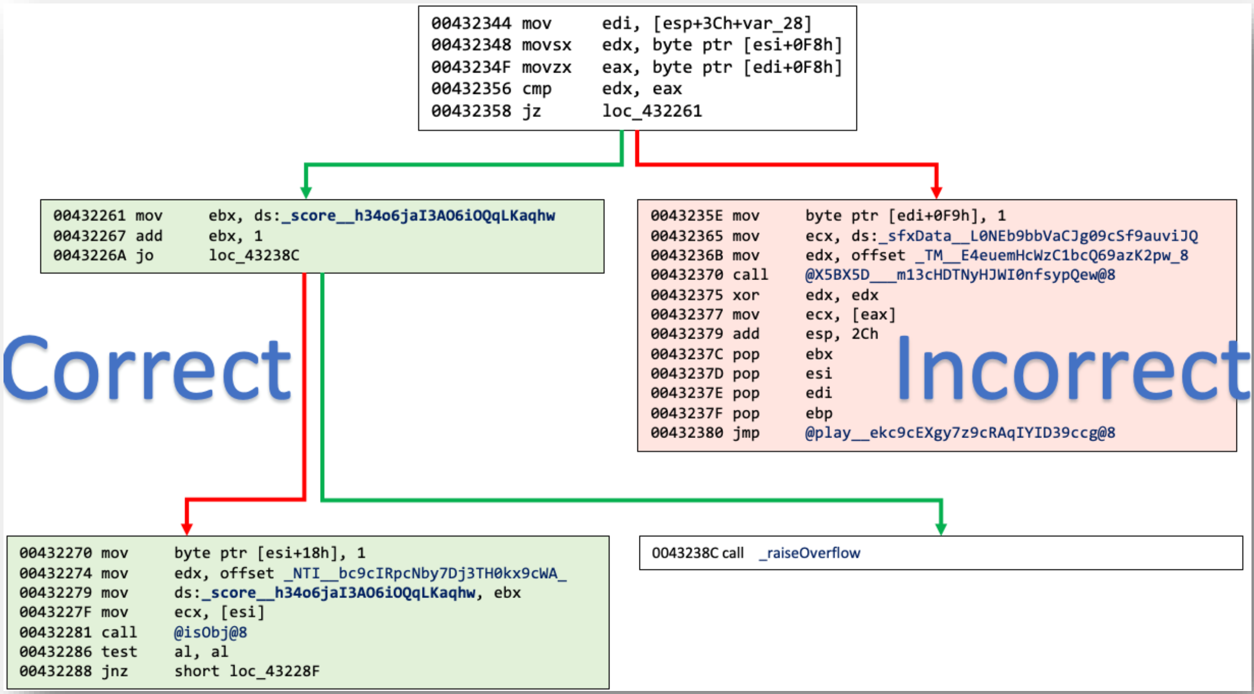

Figure 14: Path to increment the global variable _score

The function named `@onCollide__9byAjE9cSmbSbow3F9cTFQfLg@8` first determines if the player character has collided with one of the obstacles. If not colliding, it checks if the player performed the correct action (jumping or ducking) for the associated obstacle at VA `0x00432358`. This action check is specifically what Figure 14 outlines. If the player performed the correct action, the global variable `_score` is incremented. Otherwise, a sound is played and the game resets.

So, how can we bypass this action check and beat the game? We can patch it! Modifying the jump-if-zero (`JZ`) instruction at VA `0x00432358` to an unconditional jump (`JMP`) should work for our needs as shown in Figure 15 below.

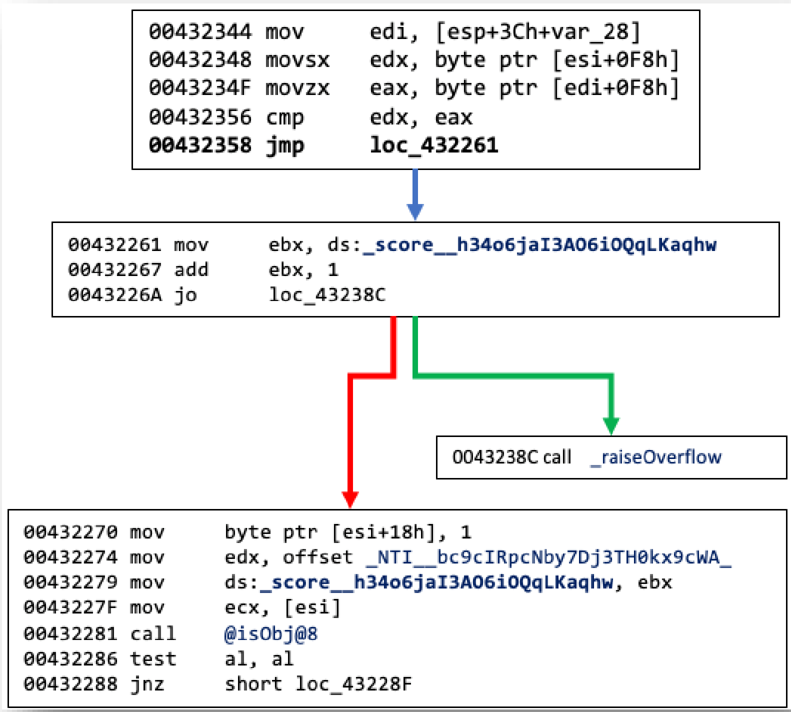

Figure 15: Patched action check

With this patch in place, we have two options for beating the game:

1. Find and patch the collision check
2. Hold the down arrow and duck under all 296 obstacles

For simplicity, I'll go with option #2. Running the patched game executable while holding the down arrow, we easily duck under all the obstacles. After about 10 minutes of game play, we beat the game and are presented with the win screen as seen in Figure 16 below.

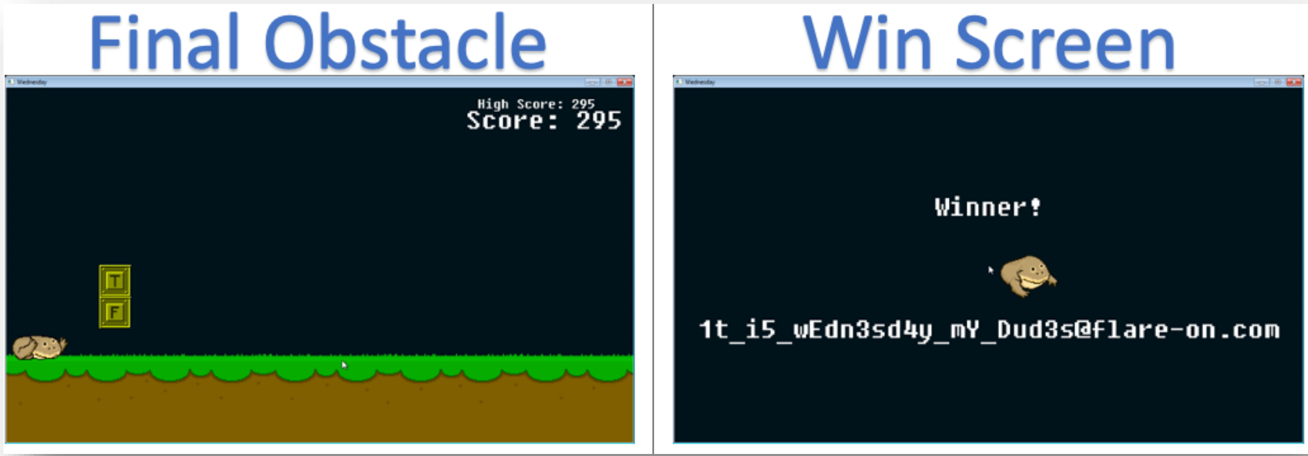

Figure 16: Ducking under the final obstacle

The win screen presents us with the flag:

`1t_i5_wEdn3sd4y_mY_Dud3s@flare-on.com`

Miscellany

It's likely many attempted to jump straight to the win screen. However, this would prove unfruitful as the text displayed on the win screen is created as the game is played.

As discussed in the two solutions above, each player action is checked for correctness against the corresponding obstacle. Not only is each obstacle assigned a day tile image, but also a one or zero byte indicating which action needs to be taken by the player. If the correct action is taken, the corresponding one or zero byte is appended to a Nim sequence encapsulated within the player object.

The win screen generates the text displayed by calling the function named `@getPlayerText__Uox5Ls3Q9bP7F7vcih9ag2vQ@4` at VA `0x00434880`. This function iterates the player's sequence and converts it into an ASCII string similar to the Python script in Figure 13. The resulting string is then displayed.