MANDIANT®
NOW PART OF Google Cloud

FLARE-ON CHALLENGE 9 SOLUTION

BY TINA JOHNSON (@0XTININJA)

Challenge 3: Magic 8 Ball

Challenge Prompt

You got a question? Ask the 8 ball!

7-zip password: flare

Solution

The challenge archive contains one executable (Magic8Ball.exe) and multiple library files. The *assets* folder contains a PNG file and multiple font files probably used by the Magic8Ball.exe program. While there are multiple files included in the challenge archive, it is clear that Magic8Ball.exe is the file of interest.

Game Overview

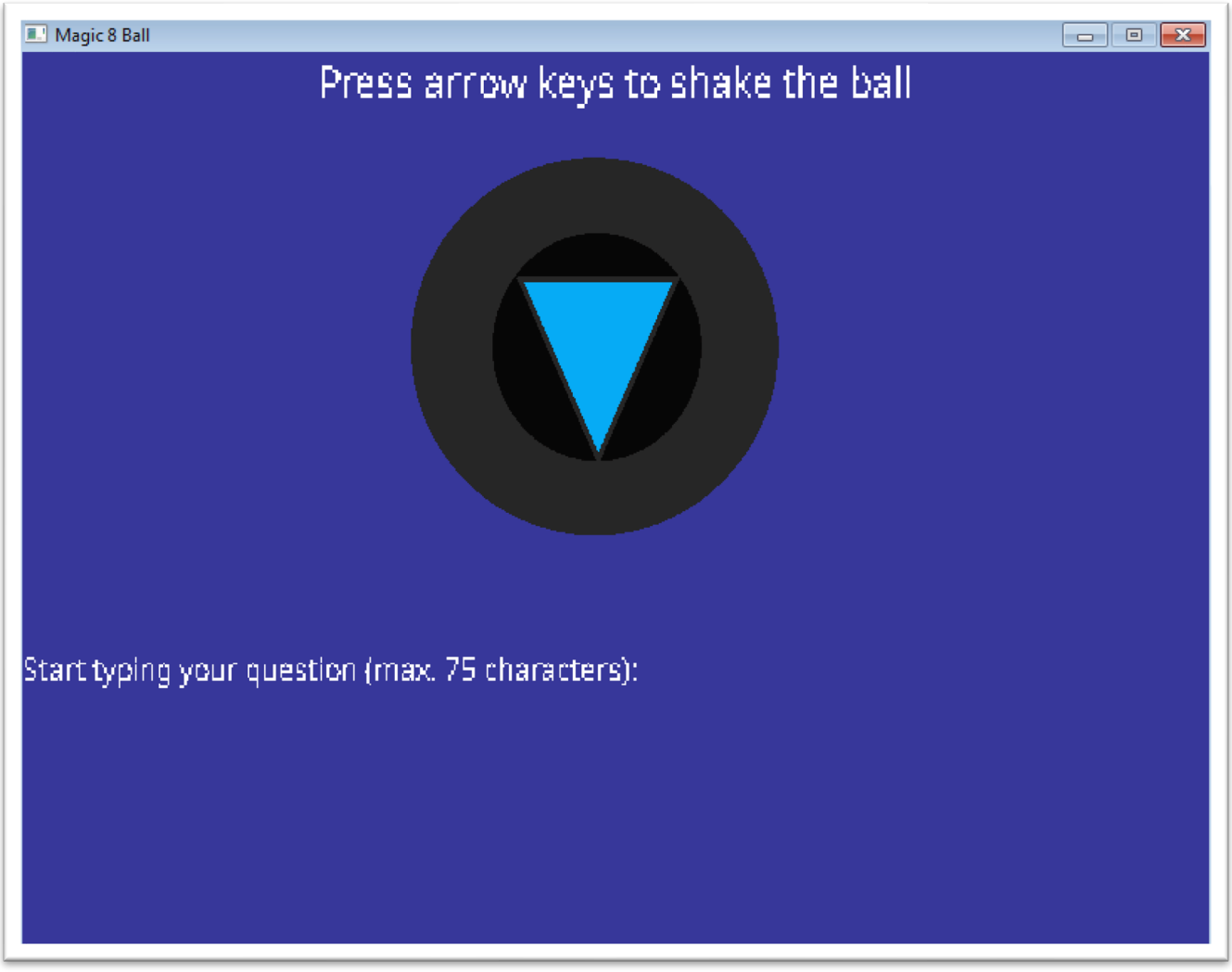


Figure 1 - Starting window of Magic8Ball.exe

On executing Magic8Ball.exe, a window opens as seen in Figure 1. The game instructs the user to "shake" the ball using the arrow keys. There's also a text telling the user to type in their question. Inferring from the name of the game

and the instructions on the window, it is understood that the game is trying to mimic a Magic 8 ball toy. Either entering a question or shaking the ball or both provides the user with a prediction such as shown in Figure 2.

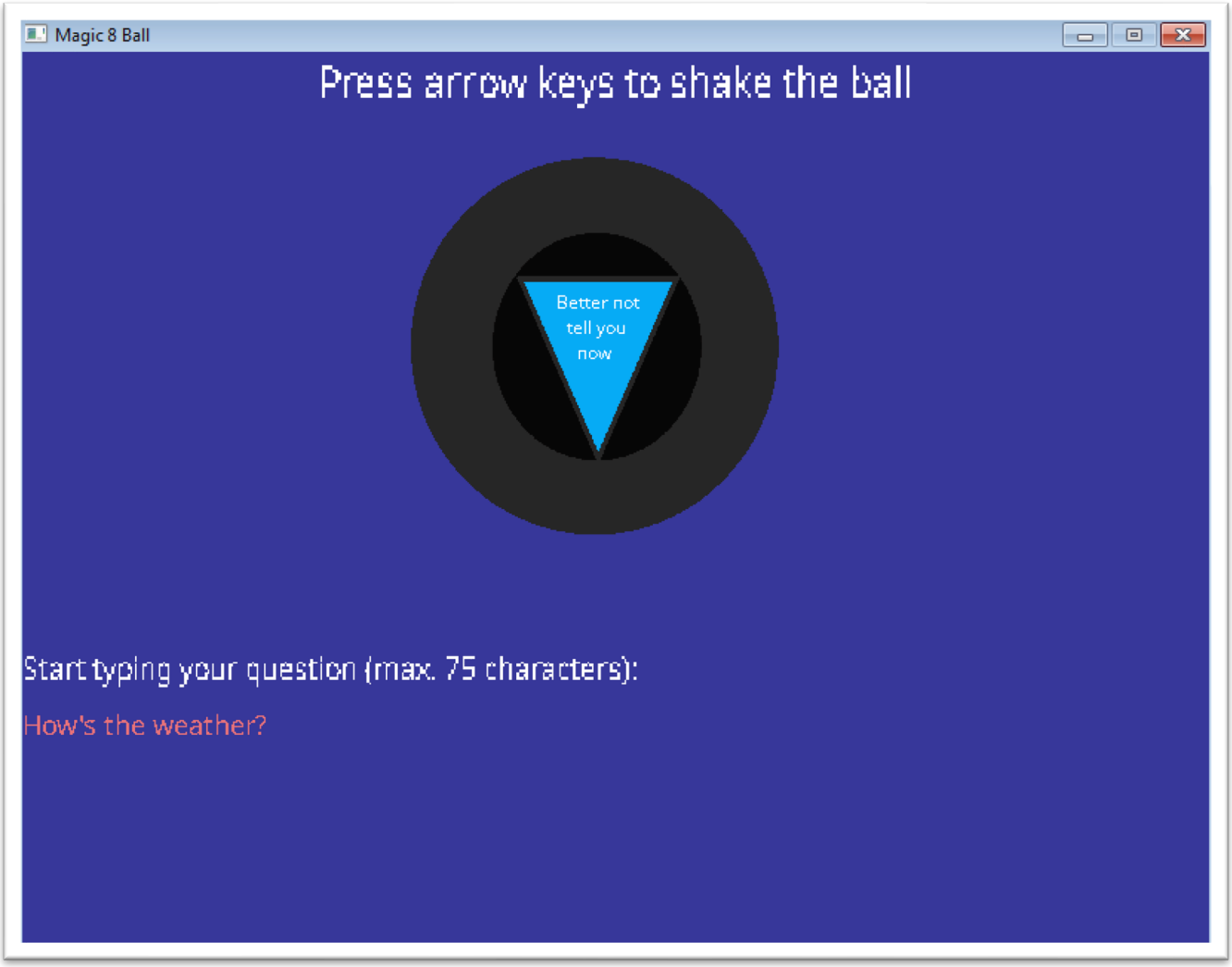


Figure 2 - Magic 8 Ball answers the user

Let's explore the binary to retrieve the flag. First step is to do basic static analysis. On looking at the output of *strings* command, we see few interesting strings that might guide our analysis:

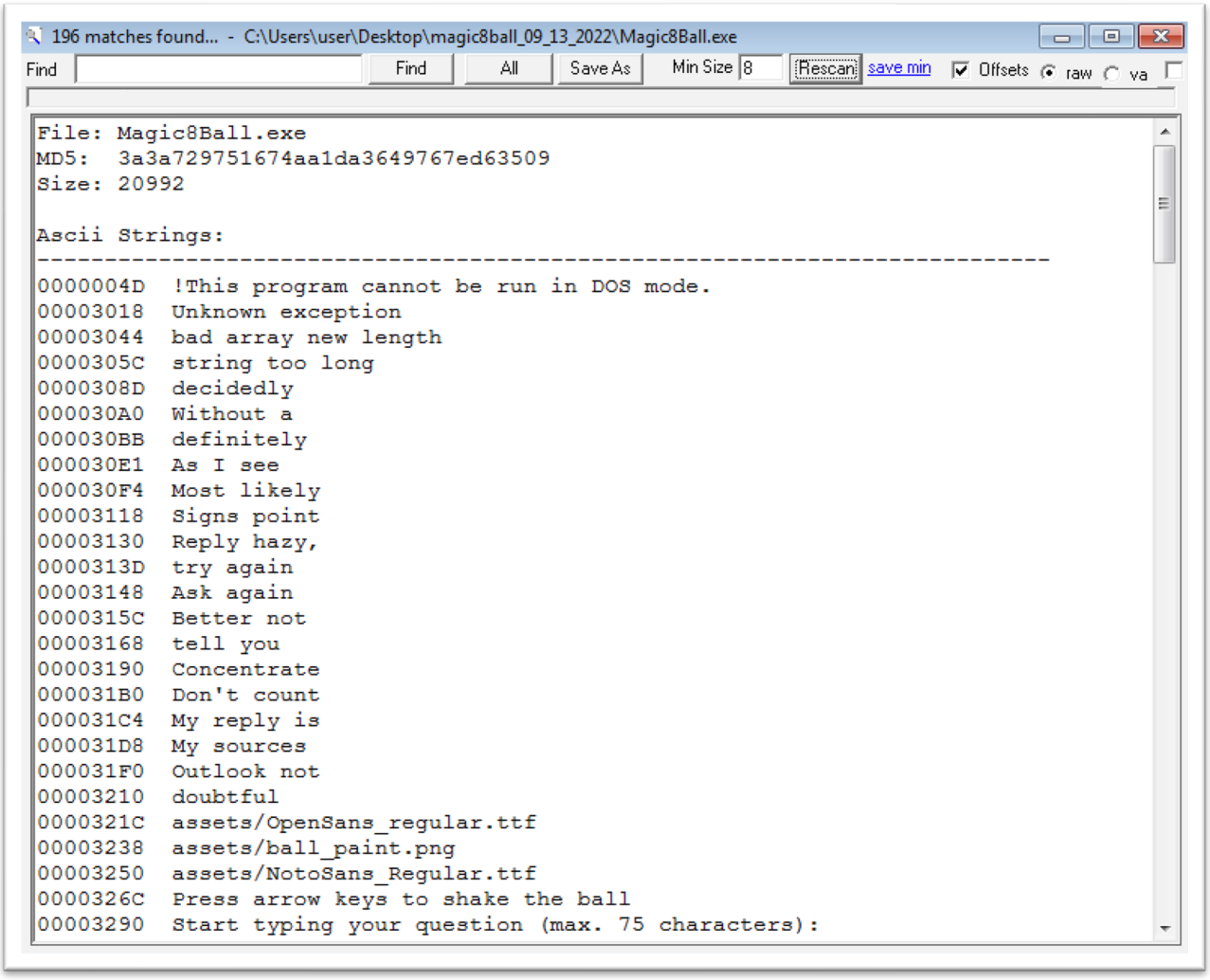
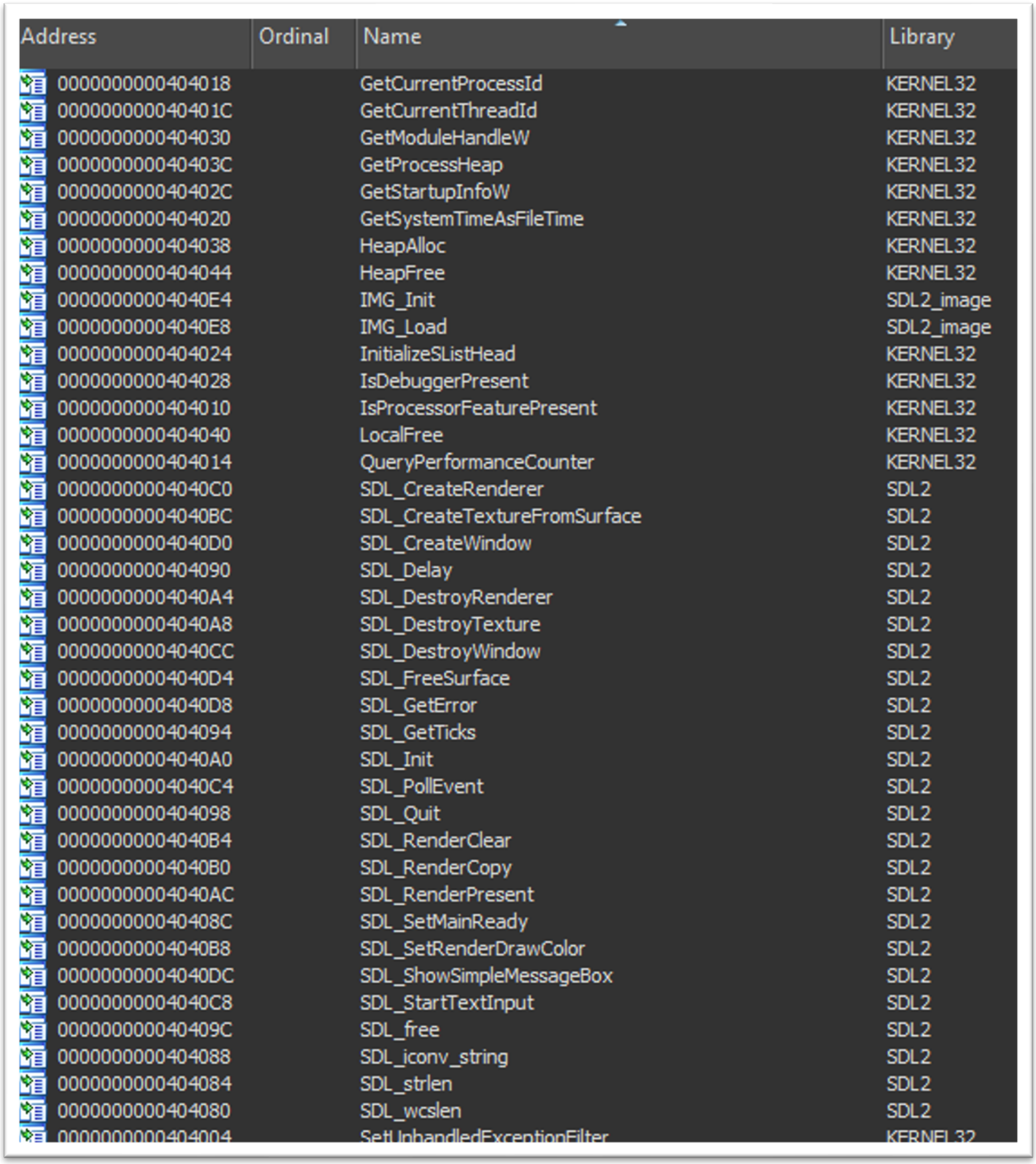


Figure 3 - Interesting Strings

We find some strings that are in fact the predictions provided to the user by the Magic 8 ball and we can confirm this by jumping to the file offsets of any of these strings. On looking at cross references to these strings (starting at 0x404270), we land at sub_4012B0. Since this function seem to do nothing more than initialize strings, we move on to find the cross references to the function and we land in sub_4027A0. We see multiple functions called within this function. You could choose to explore these functions one by one to understand the functionalities of the executable but a more efficient way would be looking at interesting imports of this PE file and jumping to functions that call them especially since we know some of the executable's functionalities (creates a window, accepts keyboard user input, etc). Looking at the Imports subview in IDA (Figure 4) we see a couple of interesting imports such as SDL_CreateWindow, SDL_PollEvent, SDL_StartTextInput. On searching for these function names, we see documentation that confirms that these are SDL2 library functions that can create windows, poll for events, and accept user text input events.



Address	Ordinal	Name	Library
0000000000404018		GetCurrentProcessId	KERNEL32
000000000040401C		GetCurrentThreadId	KERNEL32
0000000000404030		GetModuleHandleW	KERNEL32
000000000040403C		GetProcessHeap	KERNEL32
000000000040402C		GetStartupInfoW	KERNEL32
0000000000404020		GetSystemTimeAsFileTime	KERNEL32
0000000000404038		HeapAlloc	KERNEL32
0000000000404044		HeapFree	KERNEL32
00000000004040E4		IMG_Init	SDL2_image
00000000004040E8		IMG_Load	SDL2_image
0000000000404024		InitializeSListHead	KERNEL32
0000000000404028		IsDebuggerPresent	KERNEL32
0000000000404010		IsProcessorFeaturePresent	KERNEL32
0000000000404040		LocalFree	KERNEL32
0000000000404014		QueryPerformanceCounter	KERNEL32
00000000004040C0		SDL_CreateRenderer	SDL2
00000000004040BC		SDL_CreateTextureFromSurface	SDL2
00000000004040D0		SDL_CreateWindow	SDL2
0000000000404090		SDL_Delay	SDL2
00000000004040A4		SDL_DestroyRenderer	SDL2
00000000004040A8		SDL_DestroyTexture	SDL2
00000000004040CC		SDL_DestroyWindow	SDL2
00000000004040D4		SDL_FreeSurface	SDL2
00000000004040D8		SDL_GetError	SDL2
0000000000404094		SDL_GetTicks	SDL2
00000000004040A0		SDL_Init	SDL2
00000000004040C4		SDL_PollEvent	SDL2
0000000000404098		SDL_Quit	SDL2
00000000004040B4		SDL_RenderClear	SDL2
00000000004040B0		SDL_RenderCopy	SDL2
00000000004040AC		SDL_RenderPresent	SDL2
000000000040408C		SDL_SetMainReady	SDL2
00000000004040B8		SDL_SetRenderDrawColor	SDL2
00000000004040DC		SDL_ShowSimpleMessageBox	SDL2
00000000004040C8		SDL_StartTextInput	SDL2
000000000040409C		SDL_free	SDL2
0000000000404088		SDL_iconv_string	SDL2
0000000000404084		SDL_strlen	SDL2
0000000000404080		SDL_wcslen	SDL2
0000000000404004		SetUnhandledExceptionFilter	KERNEL32

Figure 4 - Imports

Looking at the cross references to SDL_CreateWindow, we land in sub_402090. This function seems to do a variety of actions such as render texts on window, load the ball.png from assets folder, starts accepting for user text input and most importantly, initialize a structure whose address is contained in the edi register (see Figure 5).

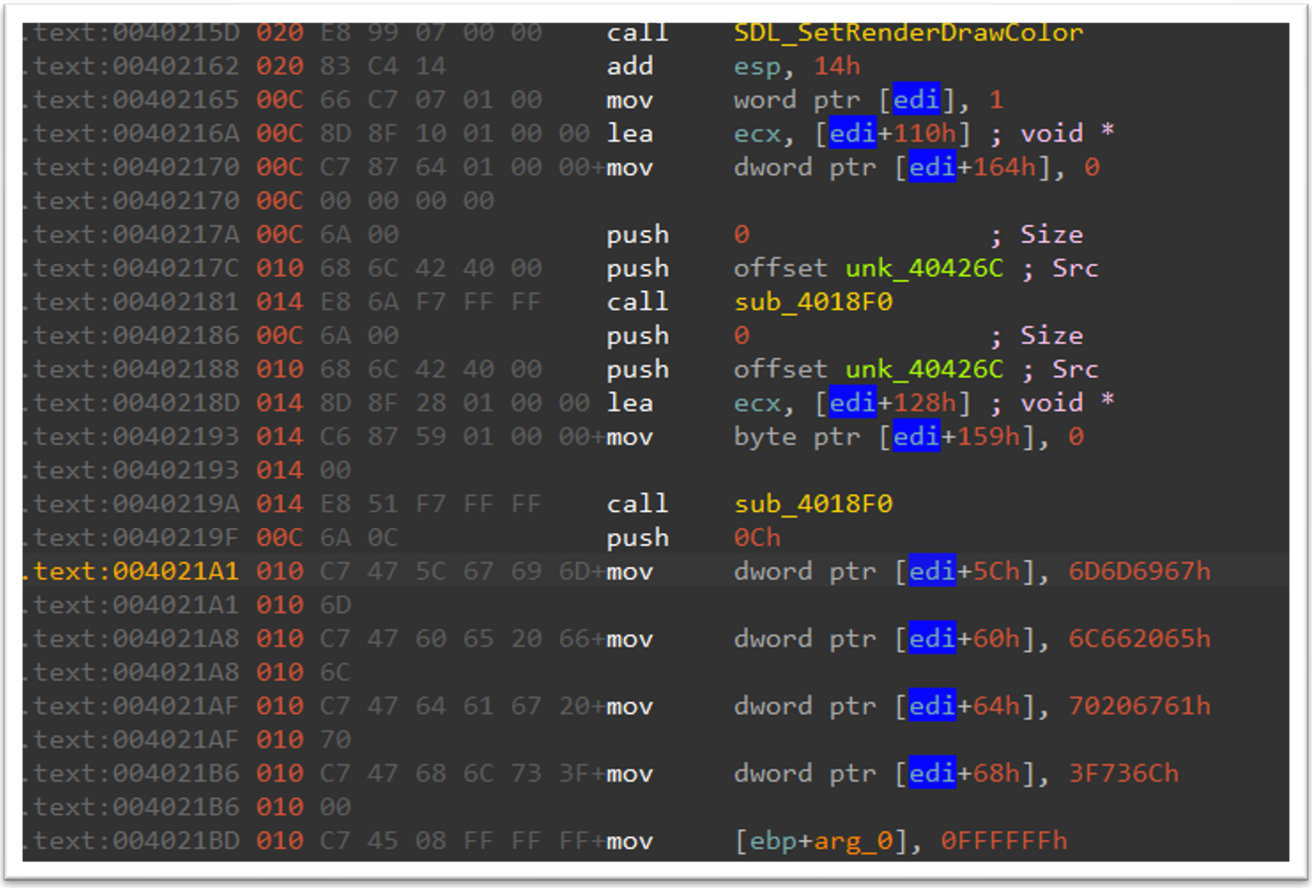


Figure 5 - Structure Initialization

Interestingly, the stack string stored to [edi + 0x5C] at 0x4021A1 spells "gimme flag pls?". This string seems to be of importance and at this point, we could manually create a structure in the Structures subview that corresponds to the structure that is initialized. Let's move on to cross references to SDL_PollEvent to see where the user input is being received. We land in sub_401E50 and we see what looks like multiple switch cases. Depending on the results of the poll event, various flags are set or unset (value set to 1 or 0). We can conclude that this function is responsible for listening to various keyboard events and setting various flags according to keys pressed by the user. We name this function get_keyb_events accordingly. It would be wise to look at the function called after this function to understand how the user input is used by this binary. We see that sub_4024E0 is that function. It is also important to note that the pointer to the structure that was initialized earlier is also seen passed to this function via the ecx register. On analyzing this function, we see a lot of if-else branches that are checking a string for characters such 'L', 'R', 'U', 'D' in a specific sequence as showing in Figure 6.

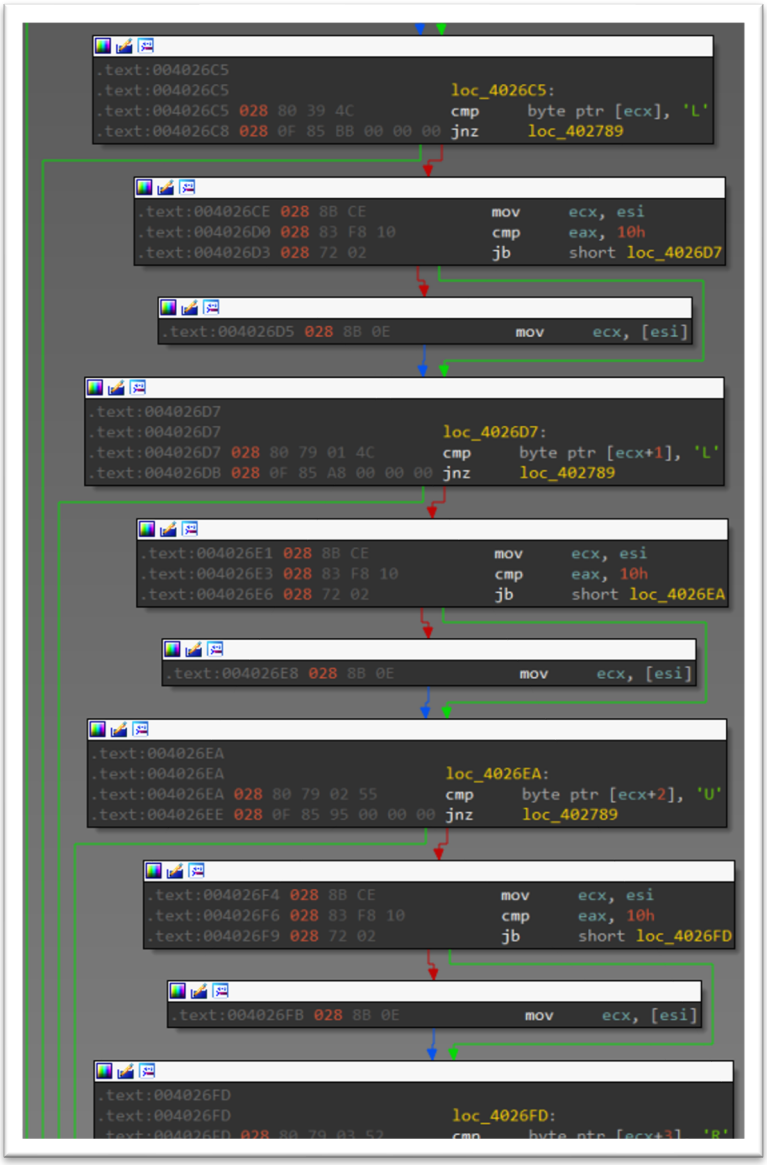


Figure 6 - if-else branches

Thinking back to the game, the user could shake the ball using arrow keys. Could 'L', 'R', 'U', 'D' stand for left, right, up and down arrow keys? We note down the sequence and it is "LLURULDUL". We follow the if-else branches to reach a strncmp function call. The strncmp function call compares the string at structure offset 0x5C to another string at offset 0xF8 of the structure. We know that the string at structure offset 0x5C is "gimme flag pls?". What is the string at structure offset 0xF8? We backtrack this structure variable and find that it referenced in get_keyb_events(). We can assume that this is possibly a keyboard user input. You can easily confirm this by popping the binary in your favorite debugger and giving user input to the Magic 8 ball game. The string at structure offset 0xF8 is the question text the user asks Magic 8 ball. We see that if the comparison succeeds, two other functions are executed. Before we dive into those functions, this would be a good point to assess the information you have. We know that user's question is compared with a specific string. We also observed a specific sequence of letters that looked like a sequence of key presses of arrow keys. In the game window, let us try entering "gimme flag pls?" as the question and shaking the ball in the order "LLURULDUL". Voila! You have cracked this challenge and found the flag. The flag is:

```
U_cRackeD_th1$_maG1cBaLL_!!_@flare-on.com
```

It is a good exercise to understand how exactly the flag was hidden in the binary. Let's look at the functions that were executed when the strncmp comparison succeeds. On analyzing sub_401A10, few hexadecimal values are observed at the start of the function as seen in Figure 7.

```
.text:00401A5B 190 C7 45 FC 00 00 00+mov     [ebp+var_4], 0
.text:00401A5B 190 00
.text:00401A62 190 C7 85 A0 FE FF FF+mov     [ebp+var_160], 33122A35h
.text:00401A62 190 35 2A 12 33
.text:00401A6C 190 C7 85 A4 FE FF FF+mov     [ebp+var_15C], 0B26457A6h
.text:00401A6C 190 A6 57 64 B2
.text:00401A76 190 C7 85 A8 FE FF FF+mov     [ebp+var_158], 34A6EF00h
.text:00401A76 190 00 EF A6 34
.text:00401A80 190 C7 85 AC FE FF FF+mov     [ebp+var_154], 3EDEE001h
.text:00401A80 190 01 E0 DE 3E
.text:00401A8A 190 C7 85 B0 FE FF FF+mov     [ebp+var_150], 40EC2101h
.text:00401A8A 190 01 21 EC 40
.text:00401A94 190 C7 85 B4 FE FF FF+mov     [ebp+var_14C], 0B0693C26h
.text:00401A94 190 26 3C 69 B0
.text:00401A9E 190 C7 85 B8 FE FF FF+mov     [ebp+var_148], 7BB269B0h
.text:00401A9E 190 B0 69 B2 7B
.text:00401AA8 190 C7 85 BC FE FF FF+mov     [ebp+var_144], 6EB2256h
.text:00401AA8 190 56 22 EB 06
.text:00401AB2 190 C7 85 C0 FE FF FF+mov     [ebp+var_140], 0CB5DF2BEh
.text:00401AB2 190 BE F2 5D CB
.text:00401ABC 190 C7 85 C4 FE FF FF+mov     [ebp+var_13C], 512B0F79h
.text:00401ABC 190 79 0F 2B 51
.text:00401AC6 190 C6 85 C8 FE FF FF+mov     [ebp+var_138], 55h ; 'U'
.text:00401AC6 190 55
.text:00401ACD 190 C7 85 E8 FE FF FF+mov     [ebp+var_118], 0
.text:00401ACD 190 00 00 00 00
.text:00401AD7 190 C7 85 EC FE FF FF+mov     [ebp+var_114], 0Fh
.text:00401AD7 190 0F 00 00 00
.text:00401AE1 190 C6 85 D8 FE FF FF+mov     byte ptr [ebp+Src], 0
```

Figure 7 - Hexadecimal values of interest

Looking further down in the function, we see two separate loops that iterates 256 (0x100) times. This is indicative of RC4 algorithm (refer to RC4 key scheduling algorithm). We can confirm our intuition by running CAPA tool on this

binary. As showing in Figure 8, CAPA confirms that sub_401A10 is performing RC4 decryption of the hex values we noted earlier.

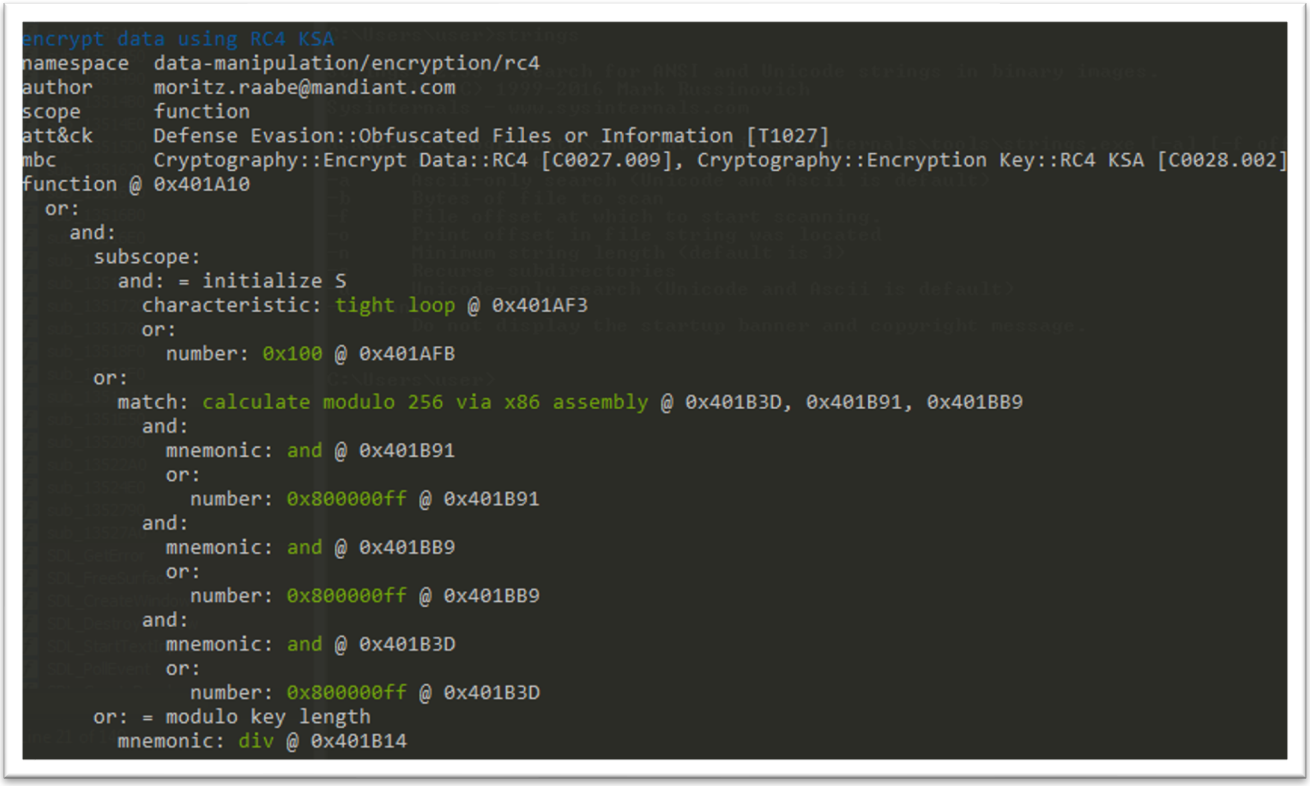


Figure 8 - CAPA output

What is the key used for decryption? Either through static or dynamic analysis, it can be observed that the key is the ball movement sequence "LLURULDUL".