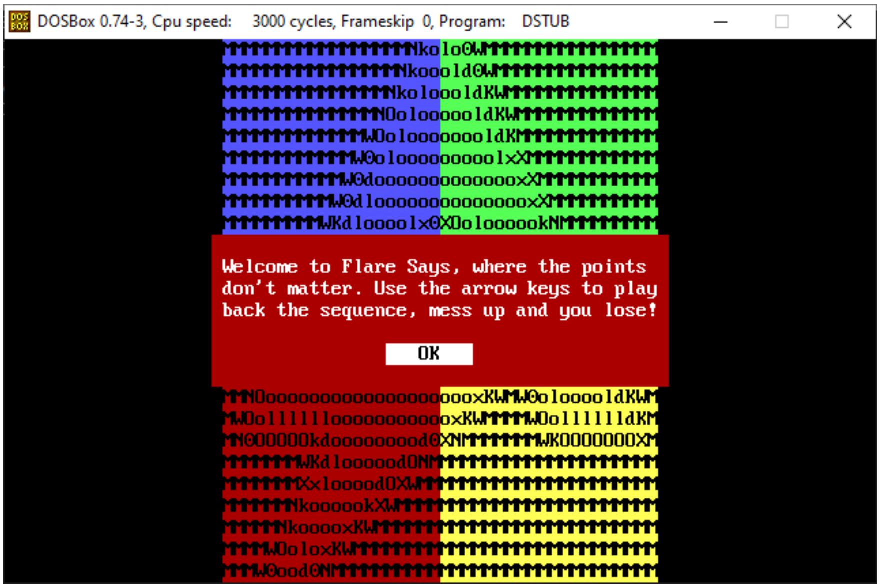
Flare-On 10 Challenge 6: FLARESays

By Paul Tarter (@Hefrpidge)

Overview

FlareSay is a polymorphic challenge that uses a custom-built PE to run in both DOS and Windows. The Windows portion of the challenge is developed as position-independent code to allow the .text section to be embedded in the DOS header. The DOS portion of the executable takes advantage that every PE includes a DOS header and can be run in DOS. The DOS portion of the challenge is a game like an older physical game named Simon Says, where there are four buttons and one is supposed to playback the sequence, each level the sequence increases by one. When one inputs a correct code during the flash screen, then completes all the levels, the original PE gets patched with bytes generated throughout gameplay. The generated bytes are verified when the PE is run in Windows, if the verification is successful, then the bytes are used as a key for a modified Salsa20 cipher which decrypts the flag. The flag is displayed by calling NtRaiseHardError which is used to display a message box using `ntdll.NtRaiseHardError` was chosen so that `SysWhispers` could be used to dynamically resolve the API and directly make the syscall.



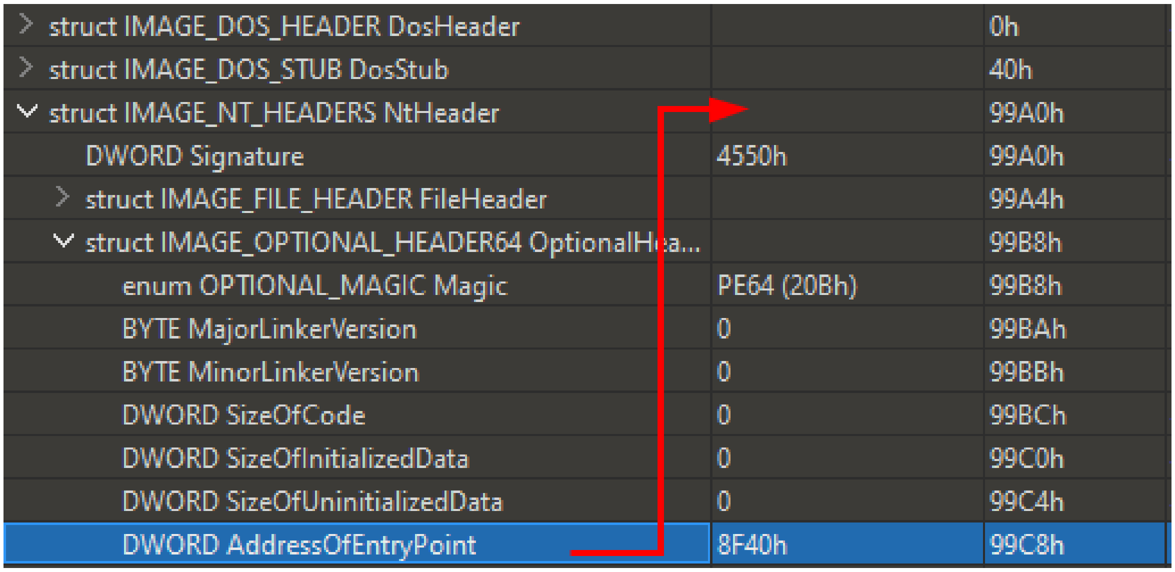
Challenge Walkthrough

Music

For added fun music was added to the challenge. The splash screen uses music from the Contra game. When playing Contra, a player could enter the “Konami code” at the splash screen and get 30 lives instead of 3 when starting the game. This same sequence needs to be entered to have a successful key generated. Rick Roll music is played when a player loses, and Mario Brothers Princess Music is played when the player wins. Reversing how to play music is not necessary for this challenge, but all midis and the sound blaster driver from the 90s are embedded in this sample. There is still floating around sound blaster driver development documentation which was helpful in developing this challenge. I hope the music added some fun to the game!

Custom PE

Initial execution of `FlareSay.exe` has the sample doing nothing and immediately exiting. Static analysis shows that the PE is not linked by normal standards. There are no sections, no data directories are populated and the `AddresssOfEntryPoint` field points into an abnormally large DOS Stub code.

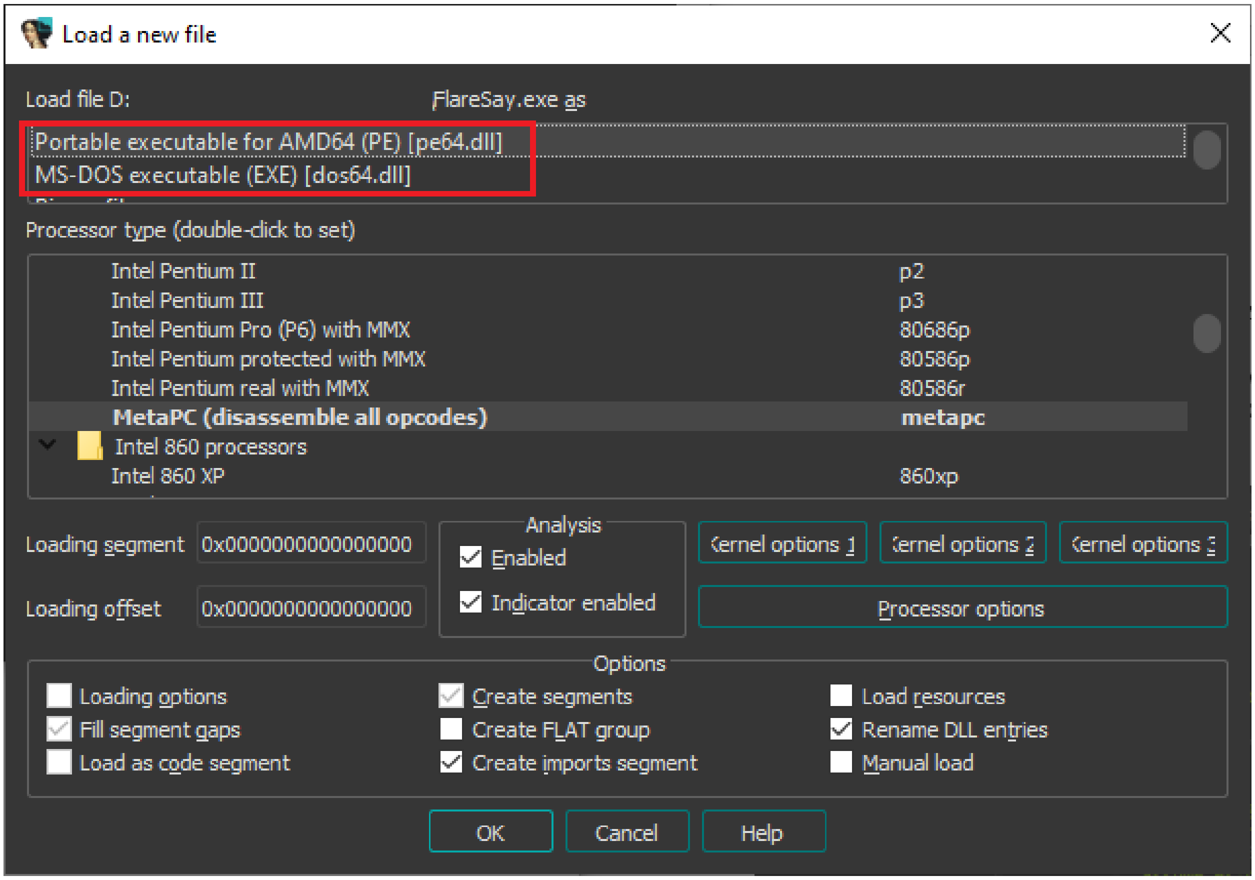


struct IMAGE_DOS_HEADER DosHeader		0h
struct IMAGE_DOS_STUB DosStub		40h
struct IMAGE_NT_HEADERS NtHeader		99A0h
DWORD Signature	4550h	99A0h
struct IMAGE_FILE_HEADER FileHeader		99A4h
struct IMAGE_OPTIONAL_HEADER64 OptionalHea...		99B8h
enum OPTIONAL_MAGIC Magic	PE64 (20Bh)	99B8h
BYTE MajorLinkerVersion	0	99BAh
BYTE MinorLinkerVersion	0	99BBh
DWORD SizeOfCode	0	99BCh
DWORD SizeOfInitializedData	0	99C0h
DWORD SizeOfUninitializedData	0	99C4h
DWORD AddressOfEntryPoint	8F40h	99C8h

AddressOfEntryPoint pointing to an abnormally large DOS stub

Static Analysis using IDA Pro

IDA Pro provides the ability to easily analyze a PE's DOS stub or PE.



IDA Loading Dialog to analyze a DOS stub or PE

Loading IDA Pro with the MS-DOS loader will get you to a good place for statically analyzing the DOS portion. After loading the Portable Executable version, you will need to convert the entry point to code, then analysis can be performed statically.

Windows Static Analysis

The Windows portion is very straight forward from a high level.

```
__int64 entry()
{
  __int64 key; // rax
  int key_hash; // ecx
  int *_key; // rdi
  __int64 v3; // r9
  __int64 i; // r10
  int v5; // edx
  int v6; // eax
  int v7; // ecx
  int v8; // ecx
  int v9; // ecx
  unsigned int flag_size; // ebx
  _BYTE *flag; // rax
  const WCHAR *w_text; // rbx
  const WCHAR *w_caption; // rax
  UNICODE_STRING us_text; // [rsp+30h] [rbp-48h] BYREF
  UNICODE_STRING us_caption; // [rsp+40h] [rbp-38h] BYREF
  __int64 parameters[5]; // [rsp+50h] [rbp-28h] BYREF
  char response; // [rsp+80h] [rbp+8h] BYREF

  // Get Key
  key = get_config_item(ConfigTypeKey);

  // Hash Key:
  //   key currently populated with NULL bytes. Gets patched
  //   by DOS program. The correct byte sequence is:
  //
  //   2B 4F 9D F2 E6 85 93 B8 12 D0 C1 C6 4C 4B 8B 30
  key_hash = 0;
  _key = (int *)key;
  v3 = key + 2;
  i = 4i64;
  do
  {
    v5 = *(unsigned __int8 *)(v3 - 2);
    v6 = *(unsigned __int8 *)(v3 - 1);
    v3 += 4i64;
    v7 = (v5 + __ROL4__(key_hash, 7)) ^ key_hash;
    v8 = (v6 + __ROL4__(v7, 7)) ^ v7;
    v9 = (*(unsigned __int8 *)(v3 - 4) + __ROL4__(v8, 7)) ^ v8;
    key_hash = (*(unsigned __int8 *)(v3 - 3) + __ROL4__(v9, 7)) ^ v9;
    --i;
  }
  while ( i );


  // Validate Hash, exit if invalid
  if ( key_hash == 0x31D9F5FF )
  {
    // Decrypt Flag
    flag_size = get_flag_size();
    flag = (_BYTE *)get_config_item(ConfigTypeFlag);
    CrapSalsa20(_key, flag, flag_size);


    // NtRaiseHardError (Native MessageBox)
    w_text = (const WCHAR *)get_config_item(ConfigTypeFlag);
    w_caption = (const WCHAR *)get_config_item(ConfigTypeTitle);
    RtlInitUnicodeString(&us_caption, w_caption);
    RtlInitUnicodeString(&us_text, w_text);
    parameters[2] = MB_ICONEXCLAMATION;
    parameters[0] = (__int64)&us_text;
    parameters[1] = (__int64)&us_caption;
    parameters[3] = INFINITE;
    NtRaiseHardError(STATUS_SERVICE_NOTIFICATION, 4i64, 3i64, (PULONG_PTR)parameters, 0, (PULONG)&response);
  }
  return 0i64;
}
```

Entrypoint function

The Salsa20 cipher was based on a sample of malware analyzed by the FLARE team that loosely implemented the algorithm. The original modification found in malware shortened the key to four bytes, allowing for very easy brute force, The challenge didn't shorten the key. There is an anti-debug technique built into the Salsa20 that uses the PEB's debug flag as a constant for zero. Other numbers are created assuming the debug flag is zero. If one runs the sample in debug with the correct key the decryption will still be wrong.

```
BeingDebugged = NtCurrentPeb()->BeingDebugged;
v6 = 0;
v11 = *a1;
v15 = 0i64;
v7 = NtCurrentPeb()->BeingDebugged;
v16 = 0i64;
v12 = __ROL4__(v11, (NtCurrentPeb()->BeingDebugged + 1) << ((BeingDebugged + 1) << (v7 + 1)));
v13 = __ROL4__(
        a1[1],
        (NtCurrentPeb()->BeingDebugged + 1) << (((NtCurrentPeb()->BeingDebugged + 1) << (NtCurrentPeb()->BeingDebugged
                                                                                       + 1))
                                               + NtCurrentPeb()->BeingDebugged
                                               + 1));
result = (unsigned int)__ROL4__(
                         a1[2],
                         ((NtCurrentPeb()->BeingDebugged + 1) << ((NtCurrentPeb()->BeingDebugged + 1) << (NtCurrentPeb()->BeingDebugged + 1)))
                       * (((NtCurrentPeb()->BeingDebugged + 1) << (NtCurrentPeb()->BeingDebugged + 1))
                        + NtCurrentPeb()->BeingDebugged
                        + 1));
```

Anti-debug technique that uses the being debugged flag as ZERO

```
#define ZERO        ((PNATIVE_PEB)__readgsqword(0x60))->BeingDebugged
#define ONE         (ZERO + 1)
#define TWO         (ONE << ONE)
#define THREE       (TWO + ONE)
#define FOUR        (ONE << TWO)
#define FIVE        (FOUR + ONE)
#define SIX         (THREE * TWO)
#define SEVEN       (TWO << TWO) - 1)
#define EIGHT       ((ONE << THREE ))
#define NINE        (FOUR * TWO + ONE)
#define TWELVE      (THREE * FOUR)
#define SIXTEEN     (ONE << FOUR)
#define FOUR_K      (ONE << TWELVE)

key_expanded[0] = key_[0];
key_expanded[1] = key_expanded[1] ^ ROTL(key_[0], FOUR);
key_expanded[2] = key_expanded[2] ^ ROTL(key_[1], EIGHT);
key_expanded[3] = key_expanded[3] ^ ROTL(key_[2], TWELVE);
```

Anti-debug technique source code

DOS Analysis

Most of the challenge lives in the DOS portion of this sample and can just be a matter of taking the time to mark up the IDB enough until one can understand it. The challenge is broken up into a common theme of game development: GUI Initialization, Splash Screen, Game Initialization, and finally a game loop. One will note that there isn't a standard calling convention when functions are called. The DOS portion was written 100% in 16-bit assembly and to make it more "fun", I didn't standardize the calling convention. I don't cover DOS analysis using dynamic debugging, but it should be mentioned that DOSBox has a very capable debugger. To take advantage of the debugger, instead of running `dosbox.exe`, run `dosbox_with_debugger.exe`. This can be found by pulling a release from GitHub (https://github.com/dosbox-staging/dosbox-staging).

GUI Initialization

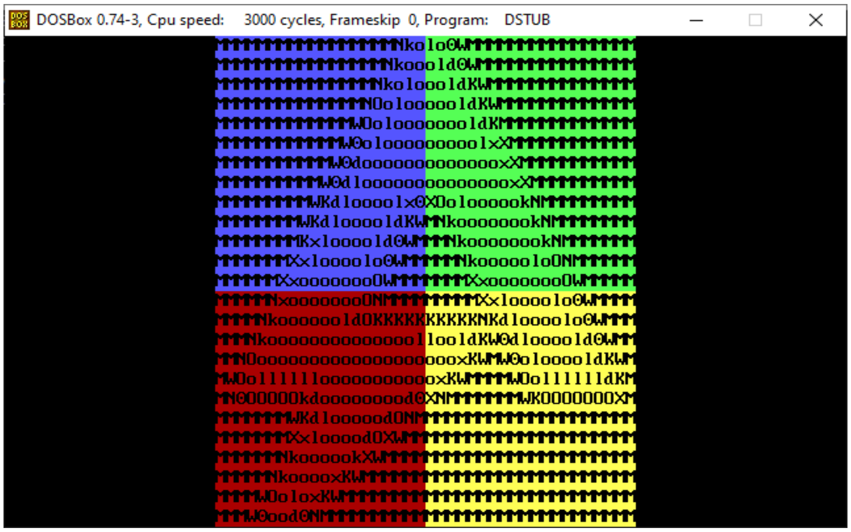
Nothing special here, clear a screen, setup cursor, and move onto the splash screen. Marking up the functions within the beginning starts to help fill out the IDB because screen and keyboard functions get a lot of use. Also, it is very quick to notice that one needs to break out their 8086 assembly and DOS API reference. IDA Pro is nice and marks up interrupts nicely..

```
seg_code:0030
seg_code:0030
seg_code:0030 ; Attributes: noreturn
seg_code:0030
seg_code:0030 _start proc near
seg_code:0030 mov     ax, seg seg_stack
seg_code:0033 mov     ss, ax
seg_code:0035 mov     sp, 20h ; ' '
seg_code:0038 mov     ax, seg seg_code
seg_code:003B mov     ds, ax
seg_code:003D assume ds:seg_code
seg_code:003D mov     es, ax
seg_code:003F assume es:seg_code
seg_code:003F mov     fs, ax
seg_code:0041 assume fs:seg_code
seg_code:0041 mov     gs, ax
seg_code:0043 assume gs:seg_code
seg_code:0043 mov     bl, 0
seg_code:0045 mov     ax, 1003h
seg_code:0048 int     10h             ; - VIDEO - TOGGLE INTENSITY/BLINKING BIT (Jr, PS, TANDY 1000, EGA, VGA)
seg_code:0048                         ; BL = 00h enable background intensity
seg_code:0048                         ; = 01h enable blink
seg_code:004A mov     ch, 20h ; ' '
seg_code:004C mov     ah, 1
seg_code:004E mov     al, 3
seg_code:0050 int     10h             ; - VIDEO - SET CURSOR CHARACTERISTICS
seg_code:0050                         ; CH bits 0-4 = start line for cursor in character cell
seg_code:0050                         ; bits 5-6 = blink attribute
seg_code:0050                         ; CL bits 0-4 = end line for cursor in character cell
seg_code:0052 call    clear_screen
seg_code:0055 call    splash_screen
seg_code:0058 call    game_start
seg_code:005B call    clear_screen
seg_code:005E xor     dx, dx
seg_code:0060 call    move_cursor
seg_code:0063 mov     ax, 4C00h
seg_code:0066 int     21h             ; DOS - 2+ - QUIT WITH EXIT CODE (EXIT)
seg_code:0066 _start endp             ; AL = exit code
seg_code:0066
```

Entrypoint for DOS stub

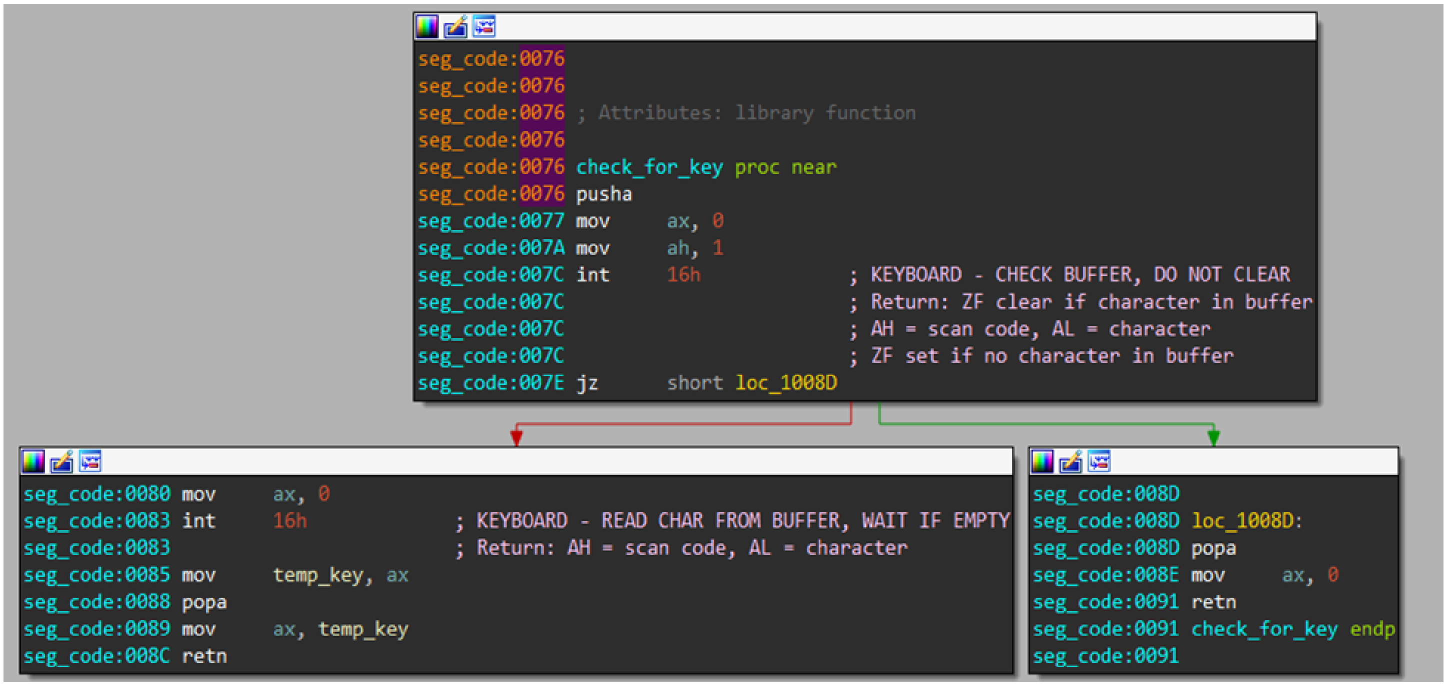
Splash Screen

The splash screen is a very important portion of the challenge because it is where random is seeded, and to obtain the correct hash, the seed needs to be correct. The seed will be correct if you luckily have the splash screen played at minute 10 and 12 seconds of the hour, because random is seeded with the AX register. AL is seconds, AH is minutes. The other way to get the proper seed is to enter the "Konami Code", "up, up, down, down, left, right, left, right". If this code is entered during the splash screen, the input buffer is hashed and the result (0x0C0A) will be placed in the AX register to seed the call to random. The hash is a simple right shift 5 add hash.

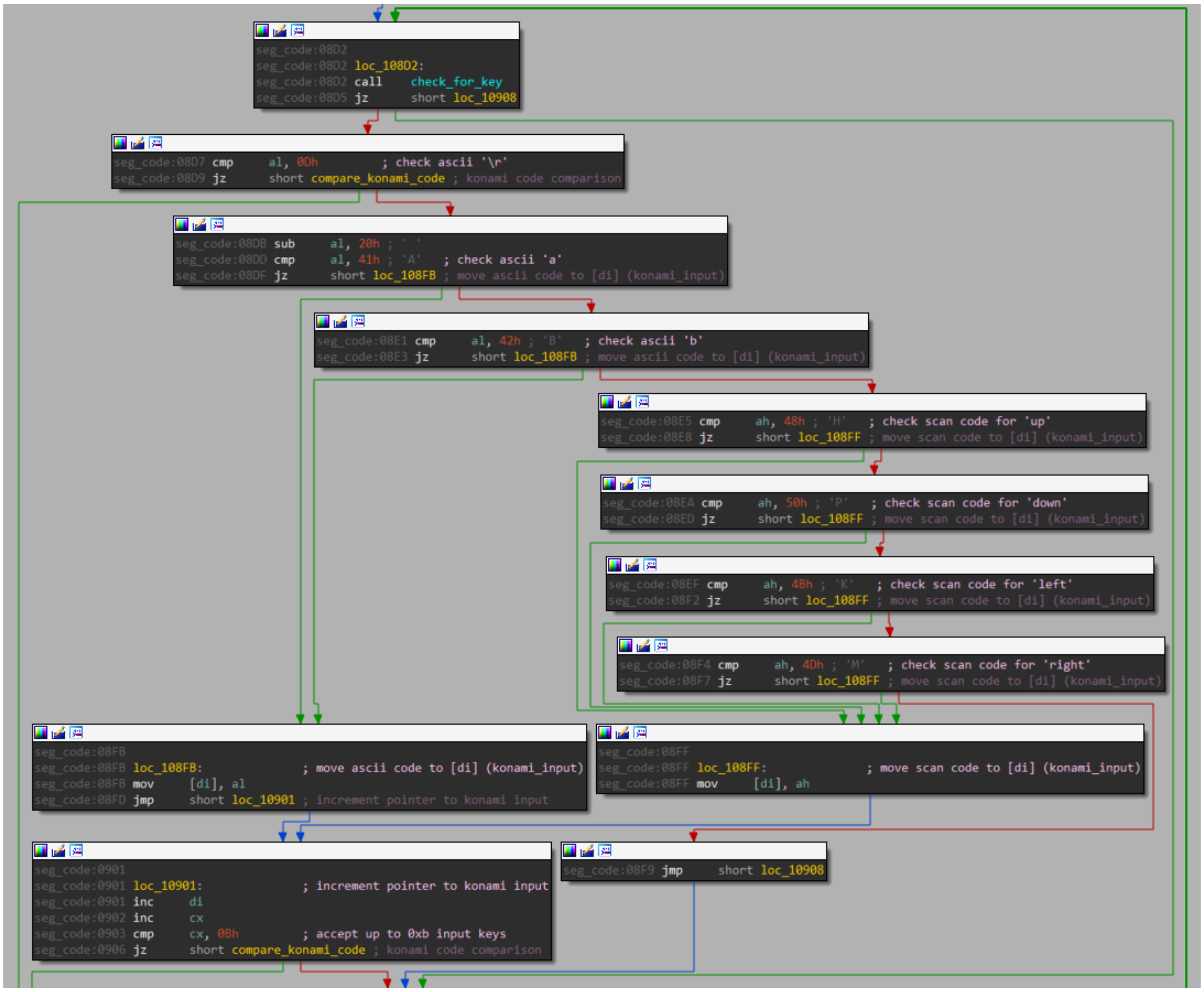


Splash screen to enter "Konami Code"

The following function polls for a key press and returns the keypress in AX. The keypress has two portions as seen below: AH is the scan code and AL is the character. The next image is the logic for checking the "Konami Code".



Function to poll for key press



Loop to read key presses during the splash screen

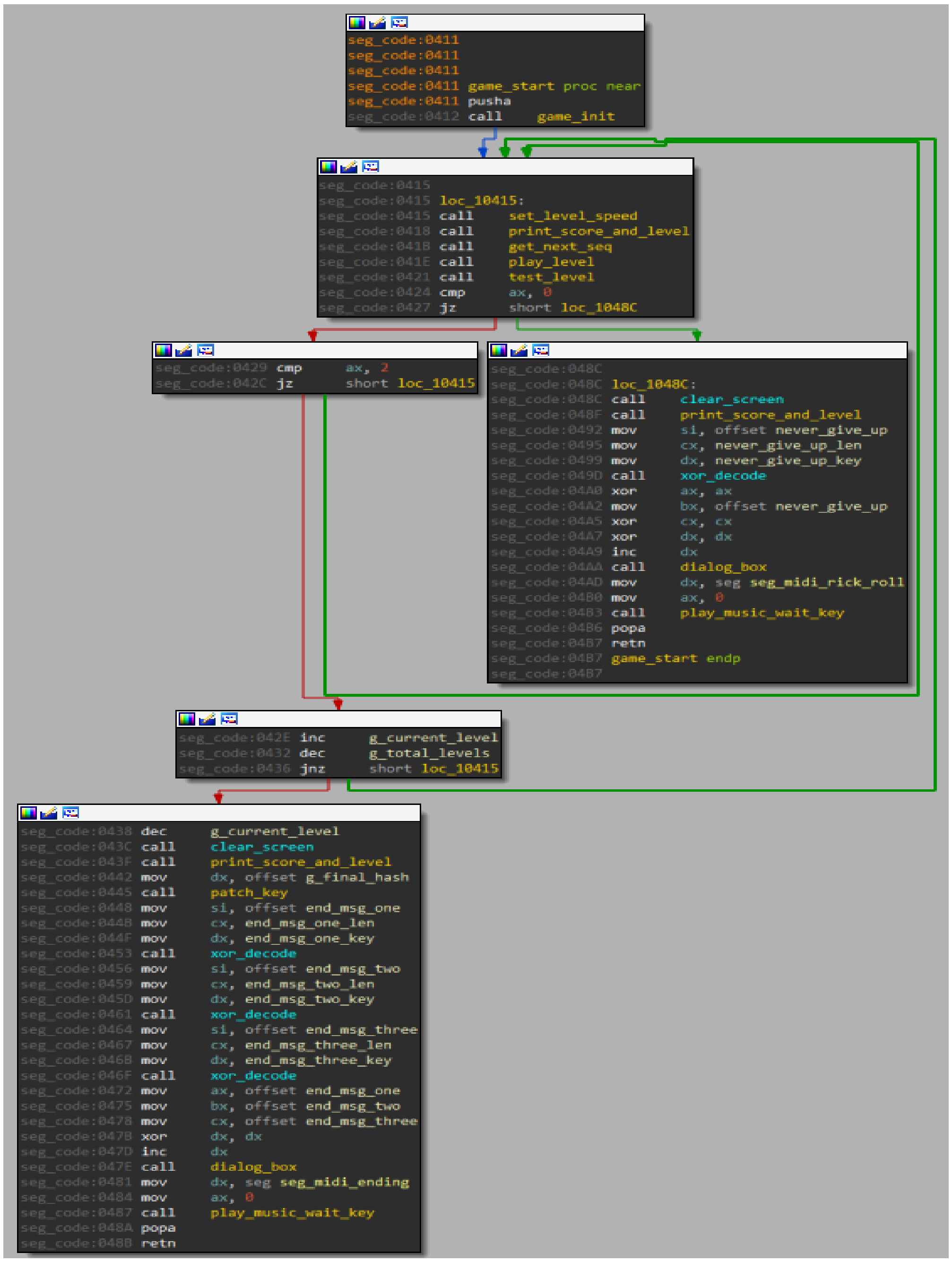
The item to be careful with is overlooking that the register AH is the scan code and AL is the character code. The logic first checks for a '\r' character and will exit the splash screen. The loop then checks for 'a' or 'b' characters , and lastly there are checks for the scan codes resulting in 'up', 'down', 'left', or 'right'. This code only looks for keypresses of the above, anything else is discarded. The key presses collected in this loop are then compared against the string `HHPPKMKMBA`. This string uses a combination of scan codes and character codes to store the "Konami Code".

Game Initialization

The game initialization consists of setting the initial game play speed, decrypting strings using a simple single-byte xor key, drawing the initial screen and lastly, popping up a dialog box with the instructions for the game. The one important note in this area is game speed. The four levels that were defined in code were 0,1,2,3: CPU, HARD, MED, EASY. The game initializes to MED and after ten levels jumps to HARD. One way to solve this challenge is getting the computer to play itself and setting the initial game speed to 0 will speed up game play to extremely fast.

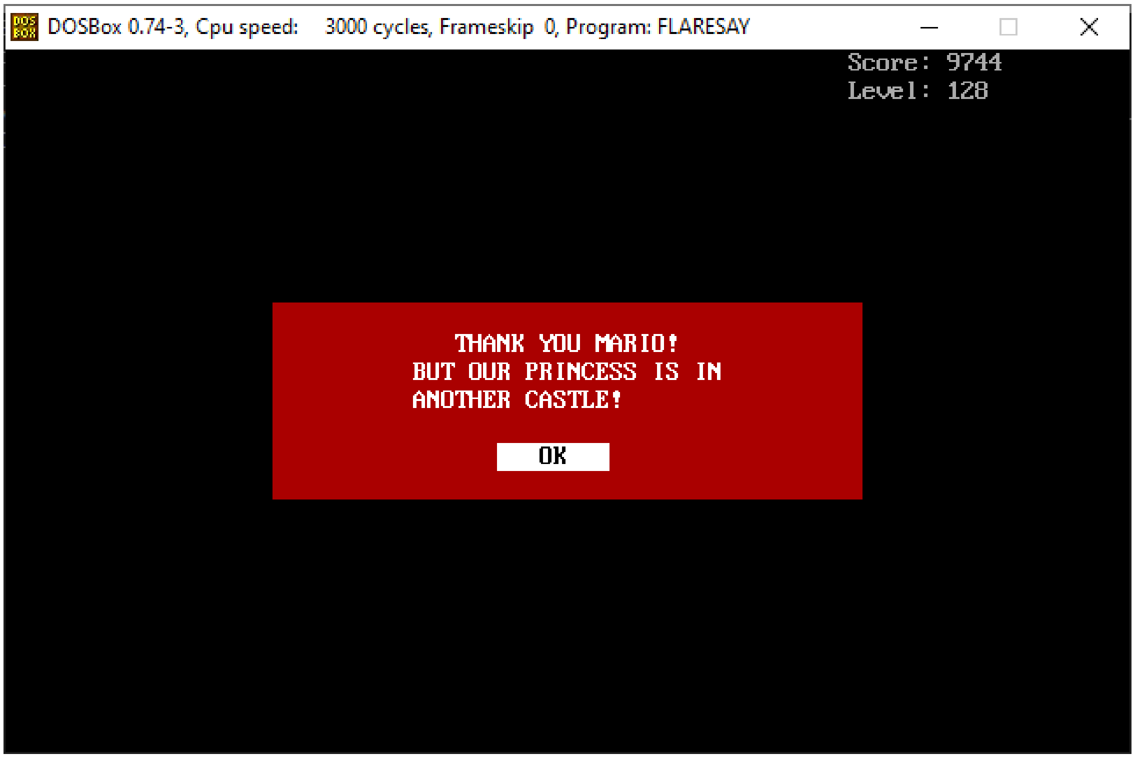
Game Play

Game play consists of a loop where the speed is set then the level speed and score is printed to the screen. The next sequence is generated using random that is seeded by input from the splash screen. The level is played by the computer and then it is the players turn to play back the level. If a mistake is made, a message pops up and it is game over. Otherwise, the level is incremented and the loop continues up until the max level of 128.



Game play loop

If one makes it through the whole game, there is a nice message displayed to the user with a throwback to Mario Bros video game on Nintendo. When the game is won, FlareSays.exe patches itself with the key that is needed in the PE (more on this later).

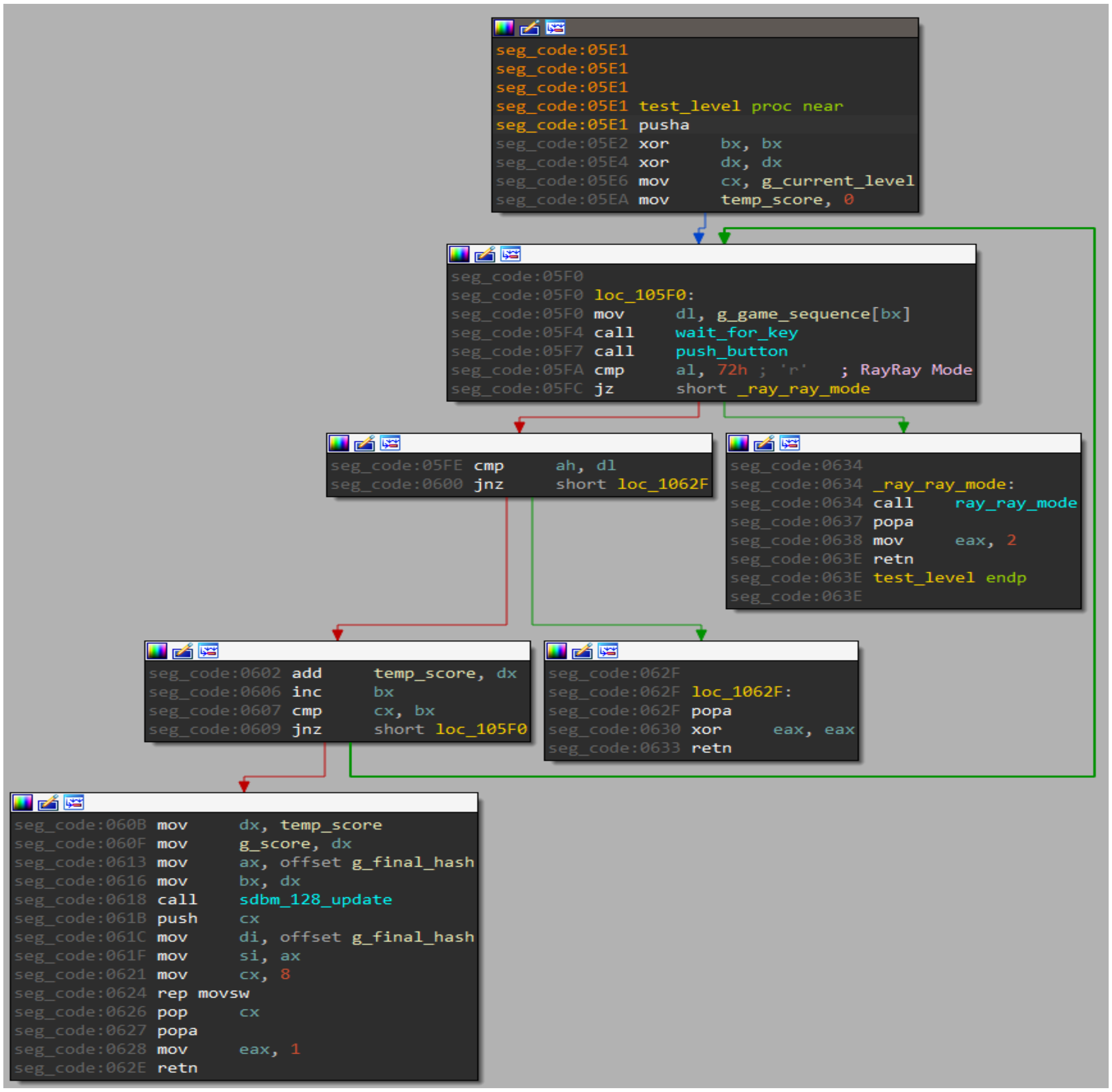


FlareSay.exe game completed

There are two major ways of attaining the key needed for the final challenge. Generate the hash or have the game play itself. The game playing itself is the easier task, first, update the speed of gameplay as shown in game initialization. In the function test_level, the global sequence is tested against, the sequence is iterated by waiting for a key press and then testing the input. The push_button function is for display purposes only. Instead of calling wait_for_key and push_button, just move DL to AL in place of the calls wait_for_key and push_button.

There is an easter egg where a player can press ‘r’ at any point in the game and get into what I call “Ray Ray Mode”. This is a mode where my three-year-old daughter can play the game without losing, she loves it. Press ‘r’ to get out of Ray Ray Mode, it starts the level over. So, if one ever gets lost in a level, press ‘r’ twice to start the level over. It will reset the score for the level, which is the hash input.

The hashing algorithm is a modified version of SDBM, where the algorithm is extended to 128 bits.



Function test_level

Key Patching

If the game is won, the final hash is written to the PE. It needs to be in the correct location for the Windows portion of the application to find it. This is the only portion that requires the sample to be run in DOS. IDA pro does a good job of marking up the DOS calls. FlareSay.exe will scan itself in 16-byte chunks looking for a 16-byte buffer filled with the bytes 0xCC. Following the tag there are five bytes that are important to the Windows sample `CALL; POP EAX; RET`, putting the key in EAX. After the five bytes is where the key goes, which initially is filled with null bytes.

```
8E70h: CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC CC
8E80h: E8 E5 FF FF FF 00 00 00 00 00 00 00 00 00 00 00
8E90h: 00 00 00 00 00 B8 54 00 00 00 C3 E8 CA FF FF FF
```

Key tag and key

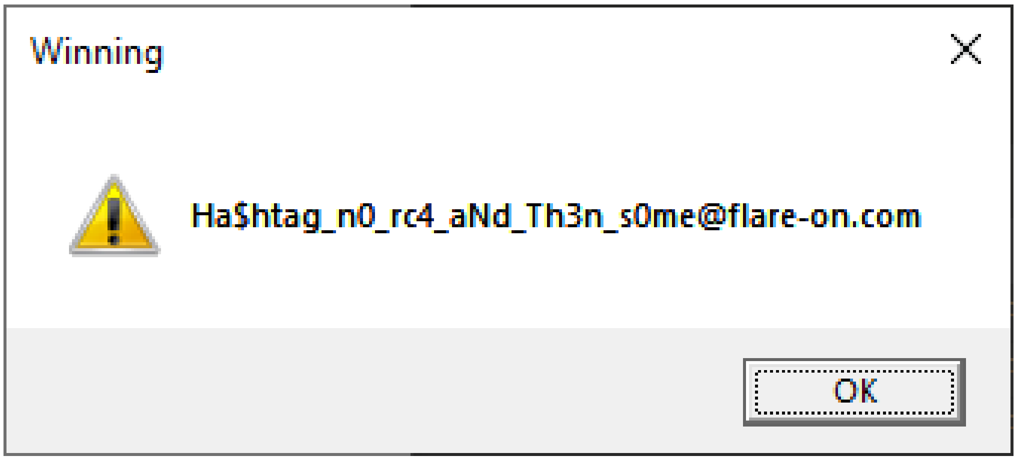
```
HEADER:0000000000408E6A     ; void *pop_ret()
HEADER:0000000000408E6A     pop_ret         proc near               ; CODE XREF: get_config_item:key_call_pop_ret
HEADER:0000000000408E6A                                             ; get_config_item:flag_call_pop_ret↓p ...
HEADER:0000000000408E6A 000                 pop     rax
HEADER:0000000000408E6B 000                 retn
HEADER:0000000000408E6B     pop_ret         endp
HEADER:0000000000408E6B
HEADER:0000000000408E6B     ; ---------------------------------------------------------------------------
HEADER:0000000000408E6C                     db 4 dup(0), 10h dup(0CCh)
```

Key tag with CALL POP RETN


seg_code:051A
seg_code:051A
seg_code:051A
seg_code:051A patch_key proc near
seg_code:051A pusha
seg_code:051B mov hash, dx
seg_code:051F mov si, offset tag
seg_code:0522 mov cx, tag_len
seg_code:0526 mov dx, tag_key
seg_code:052A call xor_decode
seg_code:052D mov si, 5C2h
seg_code:0530 mov cx, filename_len
seg_code:0534 mov dx, filename_key
seg_code:0538 call xor_decode
seg_code:053B mov al, 2
seg_code:053D mov dx, offset filename
seg_code:0540 mov ah, 3Dh ; open_file
seg_code:0542 int 21h ; DOS - 2+ - OPEN DISK FILE WITH HANDLE
seg_code:0542 ; DS:DX -> ASCIZ filename
seg_code:0542 ; AL = access mode
seg_code:0542 ; 2 - read & write
seg_code:0544 jb short loc_10598
seg_code:0546 mov file_handle, ax
seg_code:0549
seg_code:0549 loc_10549:
seg_code:0549 mov bx, file_handle
seg_code:054D mov cx, 10h
seg_code:0550 lea dx, buffer
seg_code:0554 mov ah, 3Fh
seg_code:0556 int 21h ; DOS - 2+ - READ FROM FILE WITH HANDLE
seg_code:0556 ; BX = file handle, CX = number of bytes to read
seg_code:0556 ; DS:DX -> buffer
seg_code:0558 jb short loc_10598
seg_code:055A lea si, tag
seg_code:055E mov di, dx
seg_code:0560 mov cx, 10h
seg_code:0563 cld
seg_code:0564 repe cmpsb
seg_code:0566 jnz short loc_10549
seg_code:0568 mov bx, file_handle
seg_code:056C xor cx, cx
seg_code:056E mov dx, 5
seg_code:0571 mov al, 1
seg_code:0573 mov ah, 42h
seg_code:0575 int 21h ; DOS - 2+ - MOVE FILE READ/WRITE POINTER (LSEEK)
seg_code:0575 ; AL = method: offset from present location
seg_code:0577 jb short loc_10598
seg_code:0579 mov bx, file_handle
seg_code:057D mov cx, 10h
seg_code:0580 mov dx, hash
seg_code:0584 mov ah, 40h
seg_code:0586 int 21h ; DOS - 2+ - WRITE TO FILE WITH HANDLE
seg_code:0586 ; BX = file handle, CX = number of bytes to write, DS:DX -> buffer
seg_code:0588 jb short loc_10598
seg_code:058A cmp ax, cx
seg_code:058C jnz short loc_10598

Winning

Once you have played FlareSay.exe in DOS mode, won the game, and the binary is patched,you can now execute the patched executable in Windows. You will now be presented with a nice happy flag message box:



Challenge Flag