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 AddressOfEntryPoint field points into an abnormally large DOS Stub code.

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.
```c
__int64 entry()
{
    __int64 key; // rax
    int key_hash; // ecx
    int *key; // rdi
    __int64 v5; // r9
    __int64 i; // r10
    int v0; // 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; // rdx
    UNICODE_STRING us_caption; // [rsp+30h] [rbp-48h] BYREF
    UNICODE_STRING us_text; // [rsp+30h] [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 DSS program. The correct byte sequence is:
    // 28 4F 9D F2 E6 85 93 B8 12 D8 C1 64 4C 6B 8B 39
    key_hash = 0;
    _key = (int *)key;
    v3 = key + 2;
    i = 4164;
    do
    {
        v5 = *(unsigned __int8 *)(v3 - 2);
        v6 = *(unsigned __int8 *)(v3 - 2);
        v3 += 4164;
        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 = __FILE__get_config_item(ConfigTypeFlag);
        CrapsSalt296(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[0] = HB_CONCLAVATION;
        parameters[1] = (__int64)us_text;
        parameters[2] = (__int64)us_caption;
        parameters[3] = INFINITE;
        NtRaiseHardError(STATUS_SERVICE_NOTIFICATION, 4164, 3164, (PULONG_PTR)parameters, 6, (PULONG)response);
    }
    return 8164;
}
```

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;
vs = 0;
v11 = 0x1;
v15 = 0x64;
v7 = NtCurrentPeb()->BeingDebugged;
v16 = 0x64;
v12 = _rol4((v11, (NtCurrentPeb()->BeingDebugged + 1) << ((beingdebugged + 1) << (v7 + 1)));
result = (unsigned int)_rol4(
    _rol4((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.

![Entrypoint for DOS stub](image)
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.

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”.

![Splash screen to enter “Konami Code”](image-url)
Function to poll for key press
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 HHPKMKMBA. 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.
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.
Key tag and key

Key tag with CALL POP RETN
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:

[Image of a happy flag message box]

Challenge Flag