【WIP】

x86_64 アセンブリでのライフゲームの実装

Arch B2
tatsu
親 : macchan

背景

- OSやセキュリティに興味があった。
- CTFに挑戦してみた結果、バイナリ及びアセンブリを解析する問題の意味が何度読んでもわからなかった。
- 中には、ツールを使うことで、解けるような問題もあったが、それは本質的な理解ではないと感じた。

背景

- その他有名な脆弱性、例えばバッファオーバーフローに関する説明を読んだりもしたが、ぼんやりとわかっただけで、自分で説明できる状態と言い難かった。

- 色々と勉強して行く過程で、バイナリ及びアセンブリを理解することは必須だということがわかった。

目的

- アセンブリを自分で書くことで、バイナリ及びOSに対する理解を深めたかった。

- 今後、OSやセキュリティをやっていく上で、必須となるスキルを身につけたい。

目標

- x86_64アーキテクチャのアセンブリで、ライフゲームを実装する。

- セキュリティやOSを今後やって行く上で、その第一歩として、ライフゲームというものに挑戦する。

- コンパイラから生成されたアセンブリからのコピペはせず、全部自分で書く。

なぜx86_64なのか

- MIPSなどの、情報が豊富なアセンブリは、今はほとんど使われていない。
- x86_64は、自分も使っている身近なアーキテクチャである。
- 普段自分が生成するバイナリはx86_64のものである。
- 情報源があまり多くないので、模索しながら実装して行きたかった。

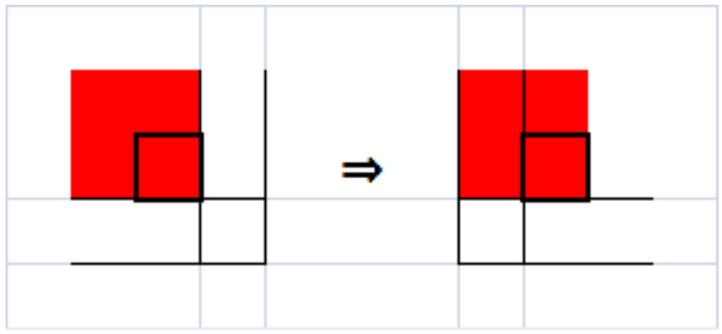
ライフゲームとは

- 基盤のような格子があり、その格子一つ一つをセルと呼ぶ。
- そのセルが生きていた場合
    - そのセルの回りに、2個あるいは3個生きたセルがあれば生存。
    - その他の場合は死。
- そのセルが死んでいた場合
    - そのセルの回りに、3個の生きたセルがあれば、誕生。
    - その他の場合は変化なし。

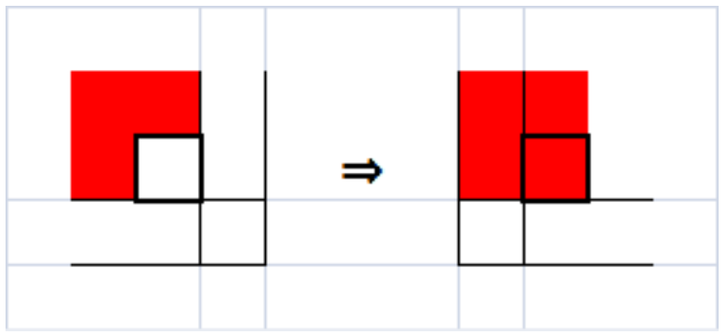
ライフゲームとは

- 過密により死ぬ

- 生存



- 誕生



実装環境

- OS : kali linux(debian系)
- アセンブラ : NASM version 2.13.02
- アセンブリ : 約400行

手法

- Cでライフゲームを書き、ロジックを確認する。
- このコードを参考に、フルスクラッチでアセンブリを書いていく。
- 書いたコードを、nasmを用いてアセンブルし、オブジェクトファイルを生成
- ld で、データをリロケートし、実行可能な形式に変更。
- 実行

C言語での実装

- Cでの実装
  https://github.com/doooooooooingggggg/lifegame/blob/master/lifegame.c

```c
//  そのマスに生命体が存在し、周囲8マスに生命体が2体または3体存在するならばば存。
//  そのマスに生命体が存在し、周囲8マスに生命体が1体以下または4本以上存在するならば死滅。
//  そのマスに生命体が存在せず、周囲8マスに生命体が3体存とするならば誕生。

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define MAX_HEIGHT 50  // i
#define MAX_WIDTH  240  // j

#define ARRAY_HEIGHT MAX_HEIGHT + 2  // i
#define ARRAY_WIDTH  MAX_WIDTH  + 2  // j
#define LIFE_SPAN  1000 // 寿命

// 123
// 456
// 789

// 自分を5とする。
// 考慮すべき相対的場所を、数字で表す。
#define RELATIVE_POSITION1 prev_val[i - 1][j - 1]
#define RELATIVE_POSITION2 prev_val[i - 1][j]
#define RELATIVE_POSITION3 prev_val[i - 1][j + 1]
#define RELATIVE_POSITION4 prev_val[i][j - 1]
// #define RELATIVE_POSITION5 prev_val[i][j]
#define RELATIVE_POSITION6 prev_val[i][j + 1]
#define RELATIVE_POSITION7 prev_val[i + 1][j - 1]
#define RELATIVE_POSITION8 prev_val[i + 1][j]
#define RELATIVE_POSITION9 prev_val[i + 1][j + 1]

void print_func(int val[ARRAY_HEIGHT][ARRAY_WIDTH]){
    for(int i = 0; i < ARRAY_HEIGHT; i++){
        for(int j = 0; j < ARRAY_WIDTH; j++){

            if( i == 0  || i == ARRAY_HEIGHT - 1 || j == 0 || j == ARRAY_WIDTH - 1 ) continue;

            if( val[i][j] == 1 ) printf("0");
            else if( val[i][j] == 0 ) printf("-");
            else printf("e");
        }
        printf("\r");
    }
    printf("\n");
}

void define_init_val(int prev_val[ARRAY_HEIGHT][ARRAY_WIDTH]){
    int flag;

    for(int i = 0; i < ARRAY_HEIGHT; i++){
        for(int j = 0; j < ARRAY_WIDTH; j++){

            if( i == 0  || i == ARRAY_HEIGHT - 1 || j == 0 || j == ARRAY_WIDTH - 1 ){
                prev_val[i][j] = 0;
                continue;
            }

            flag = (int)( rand() * (2.0) / (1.0 + RAND_MAX) );
            prev_val[i][j] = flag;
        }
    }
}
```

アセンブリ言語での実装

- 必要そうな処理を考えてみたところ、
  - printf -> **システムコールのwriteを使うことで解決**

```
; write
mov rax, 1
mov rdi, 1
mov rsi, off
mov rdx, 1
syscall
```

  - 分岐 -> **cmp, jmpで解決**

```
cmp r15, 1
je print_on
```

  - ループ -> **カウンタとcmpで解決**

```
cmp rbx, 2500
jge return_from_print
```

  - 配列の操作 -> **バイト長に気をつける**

```
movzx r13, byte [next_val + rbx]
mov [prev_val + rbx], r13
```

  - 関数ジャンプ -> **jmp**

```
jmp consider_next_each_gen
```

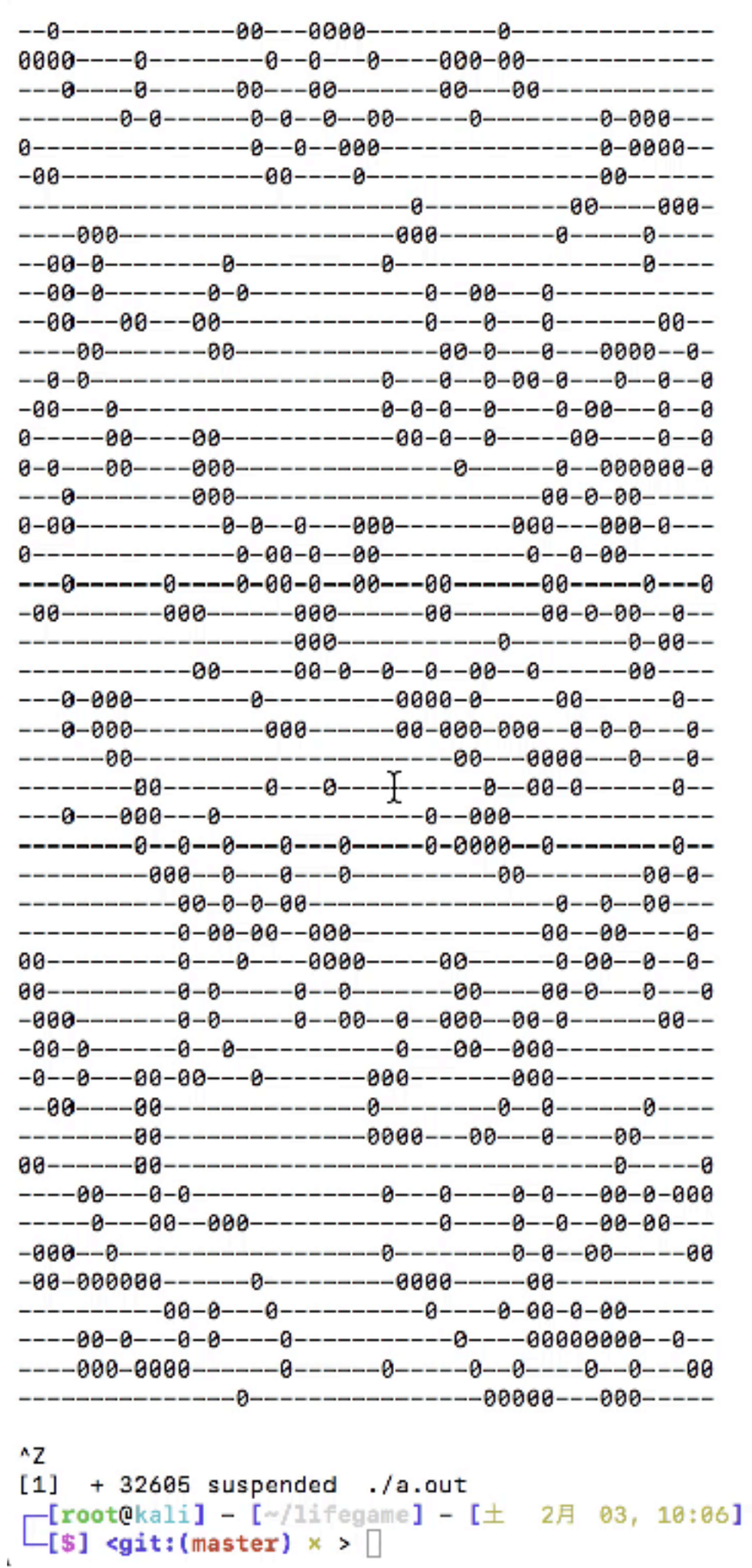
  - sleep -> **ループし、1秒おきくらいになるように調節**

```
sleep:
    cmp r11, 2000000000
    jge next_loop
    nop
    nop
    inc r11
jmp sleep
```

- 以上を組み合わせる。

アセンブリ言語での実装

- https://github.com/doooooooooingggggg/lifegame/blob/master/lifegame.s



結論

- 普段何気なく書いているコードが、どのようにハードウェアに解釈され動いているかが前よりわかるようになった。

- レジスタがどのように使われているのか、今まで結果だけを求めてfor文などを書いていたが、動く過程がわかった。

- 乱数の生成がうまくいかなかった。ハードウェアから時間をとることができなかった。

- スタックポインタをうまく活用できず、jmp命令を多用してしまった。大量の変数を扱う際は、レジスタが足りないのでスタックポインタをもっと理解しなければならない。

今後の展望

- 春休みは、この分野の勉強を進め、組み込みOSに挑戦してみたい。

参考文献

- 「Jun's Homepage」http://www.mztn.org/index.html
- 「原書で学ぶ64bitアセンブラ入門」
  http://warabanshi.hatenablog.com/search?q=%E5%8E%9F%E6%9B%B8%E3%81%A7%E5%AD%A6%E3%81%B6
- 「プログラミング講座2-1・ライフゲーム-まずはライフゲームって何？って話」
  https://fujori.com/access-excel-vba/enjoy-vba/programming-course-2-1/