3kCTF_2021

3k_chall_Crackme

`Challenge's Description:`

```
Follow my PATH!
NOTE: flag format is ctf{}
```

source : `crackme`

Vấn đề cụ thể đặt ra

Challenge này lật mở lại một trong những xu hướng của những năm 2010(theo như flag nói :v) - 2016(cái này mình gặp ở một challenge của flareon mùa 3 gòi), một ***stack-based virtual machine***, ngoài ra còn có một loại khác là ***register-based virtual machine***(chưa gặp lần nào).

Như vậy ý tưởng đã có rồi, giờ thực hiện ***disassembly*** gòi giải lại thôi.

> **stack/register-based virtual machine**
>
> Các cấu trúc dữ liệu cơ bản:
>
> - một mảng có vai trò như **bytecode** chứa các opcode chỉ định từng loại câu lệnh, ứng với mỗi câu lệnh, sẽ có số lượng tham số khác nhau.
> - Các câu lệnh như: *add, sub, mul, not, xor v.v*
> sẽ được định nghĩa bằng các hàm hoặc trong bài này là các **case** của **switch**
> - Giá trị các tham số sẽ được cho trước trong một mảng, gọi mảng này là **imm(** immediate**)**
>
> Cơ chế hoạt động:

> - Một biến **PC** lặp qua mảng **opcode** để gọi các câu lệnh, sau đó tăng lên để gọi câu lệnh tiếp theo
> - Với **stack-based vm** thì các giá trị tham số sẽ được push lên stack, **register-based vm** thì các tham số, kết quả trả về được push vào thanh ghi(thanh ghi ở đây là *"thanh ghi"* của máy ảo, mà *"thanh ghi"* của máy ảo thường là một biến trên stack, hay heap đã được chuẩn bị sẵn từ trước)

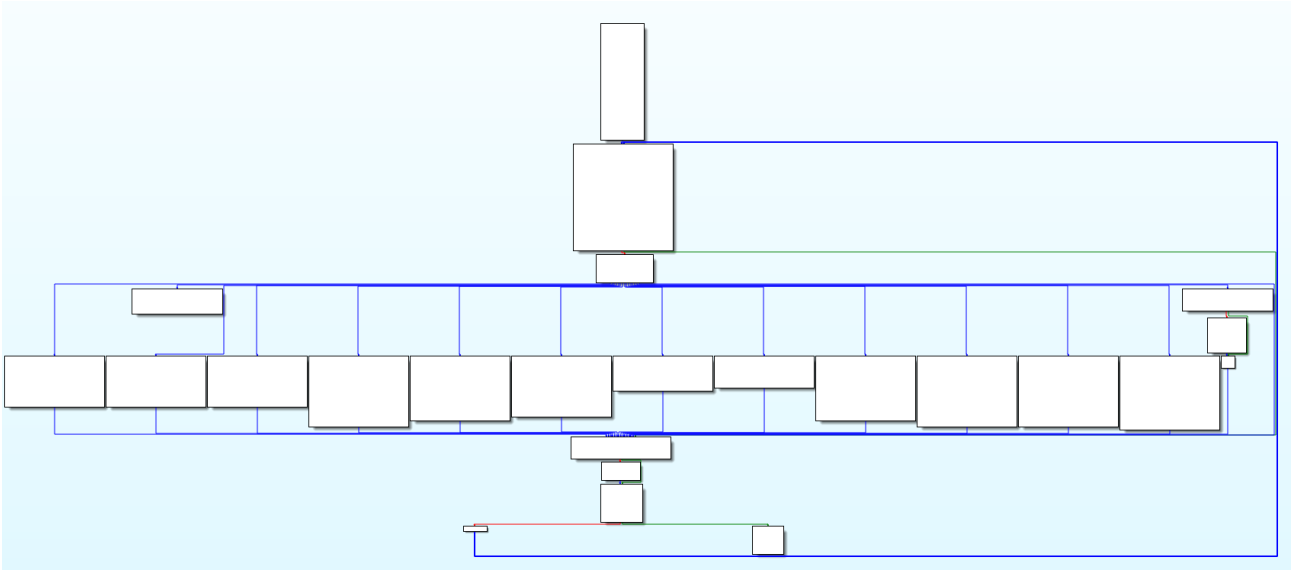
Flag ở đâu?

Flag của những challenge này theo kiến thức của mình thì từng ký tự sẽ được push lên stack, hoặc được đẩy vào thanh ghi như là tham số, được **tính toán** bằng **vm**, sau đó phải thỏa một điều kiện nào đó(chẳng hạn như bằng một **immediate**(compare), hoặc các câu lệnh làm thành một hàm, và thỏa một giá trị trả về đã định sẵn(**= 0** như challenge này)). Từ các **constraint** đó có thể xác định được flag.

Crackme

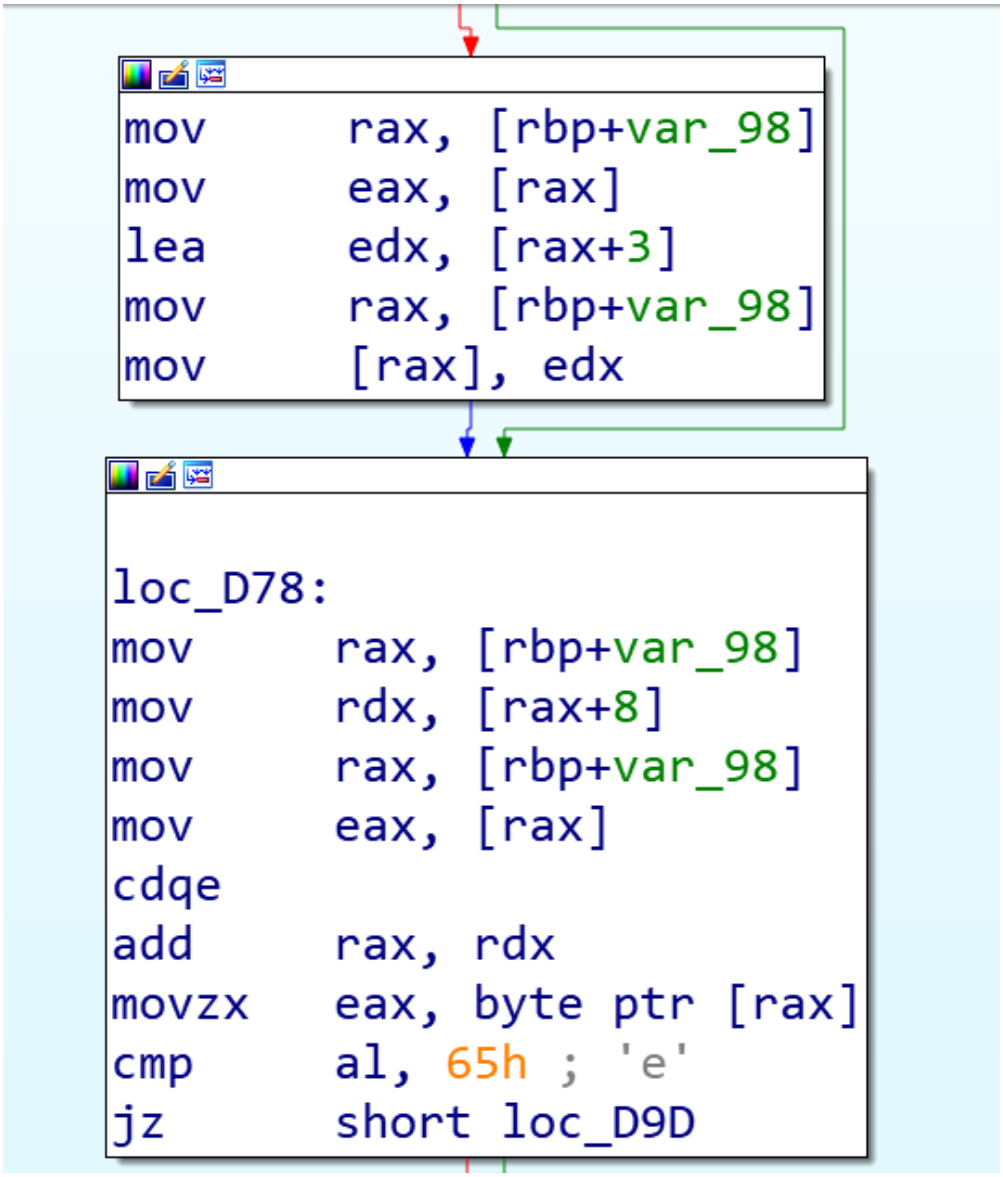
Hàm mô phỏng hoạt động của vm

Sau những bước đầu tiên thực hiện inspect và debug, mình nhận ra **sub_93A()** chính là hàm chính mô phỏng lại hoạt động của một vm như mình đã tóm tắt ở trên:

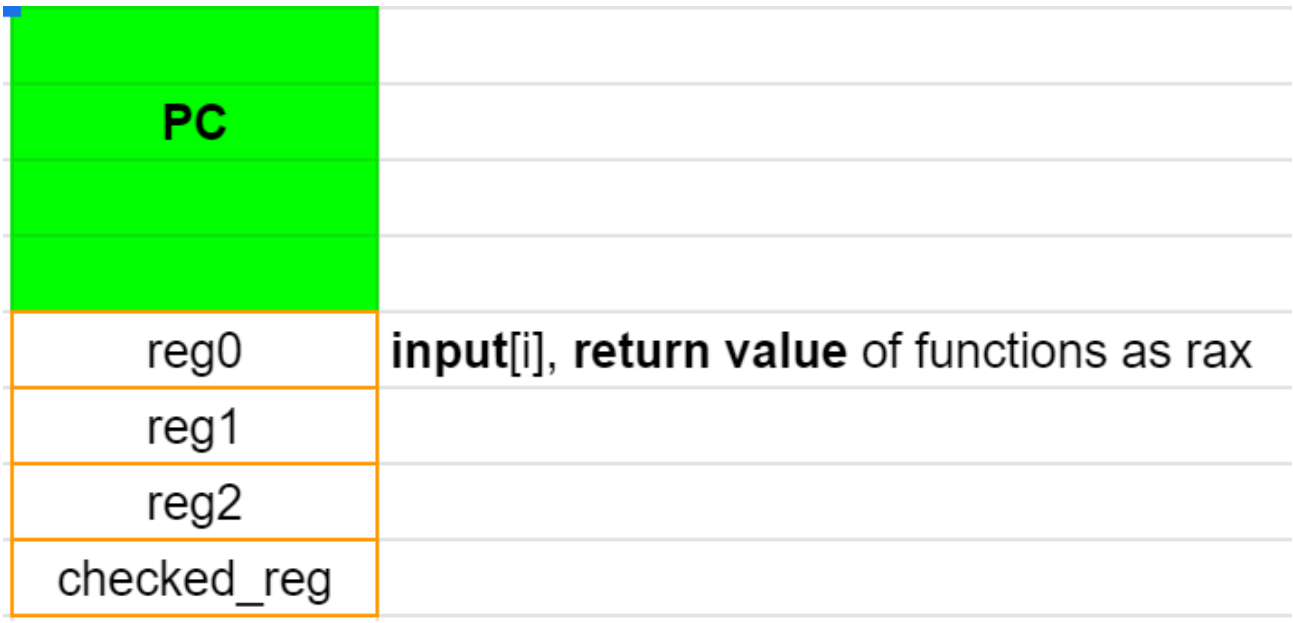


VM sẽ thực hiện 3 giai đoạn(viết một **disassembly** để thấy được):

- **stage 1:** là giai đoạn khởi tạo các giá trị, và xuất một thông báo đến màn hình "***Enter the flag:*** "(các lệnh ko tạo thành một **function**)
- **stage 2:** các câu lệnh sẽ hình thành các function theo một vài mẫu mà tác giả đã dự định sẵn(bắt đầu từ **pc = 0x00ba**).
- **stage 3:** Thông báo flag đúng/ sai(bắt đầu từ **pc = 0x43e**)



- Một câu lệnh cơ bản sẽ có 3 byte(**var_98** sẽ lưu giá trị **PC**), **byte[0]** –> opcode, **byte[1], byte[2]** sẽ **chỉ định các thanh ghi** thực hiện, hoặc **chỉ định index của immediate** trong mảng immdiate cho trước tham gia vào tính toán(value = 0, 1, 2, 3 để chỉ 4 thanh ghi như hình bên dưới) hoặc **trực tiếp đóng vai trò là toán hạng**(**byte[2]**):



4 thanh ghi của máy ảo(lưu trên stack của chương trình)

> **checked_reg** chính là reg3, tuy nhiên nó đóng vai trò là một biến **check** của chương trình chính mà vm thực hiện nên mình đổi tên lại cho dễ nhớ

Disassembly bytecode array của vm

Xác định các thành phần cốt lõi của register-based VM

14 case trong switch chính là 14 câu lệnh mà **vm** sẽ thực hiện gồm:

```
1: mul(reg[arg1], arg2)
2: sub(reg[arg1], arg2)
3: not(reg[arg1])
4: xor(reg[arg1], imm)
5: assign_reg(reg[arg1], reg[arg2])
6: assign_imm(reg[arg1], imm
7: if...
8: putc(reg[0])
9: exit(reg[0])
10: getc(reg[0])
11: shl(reg[arg1], arg2)
12: and(reg[arg1], imm)
13: or(reg[arg1], imm)
14: add(reg[arg1], reg[arg2])
```

14 câu lệnh này sẽ được phân thành 4 nhóm dựa trên sự khác nhau về tham số của lệnh gồm(arg1, arg2 ở đây để chỉ byte[1], byte[2] trong 3 bytecode được fetch từ mảng bytecode):

reg: **not, putc, getc, exit, if**(reg là tham số)
reg - arg2: **mul, sub, shl**(reg và arg2 là tham số)
reg - reg: **assign_reg, add** (tham số là 2 regs của máy ảo)
reg - imm: **xor, assign_imm, and, or(** reg, immediate là các tham số**)**

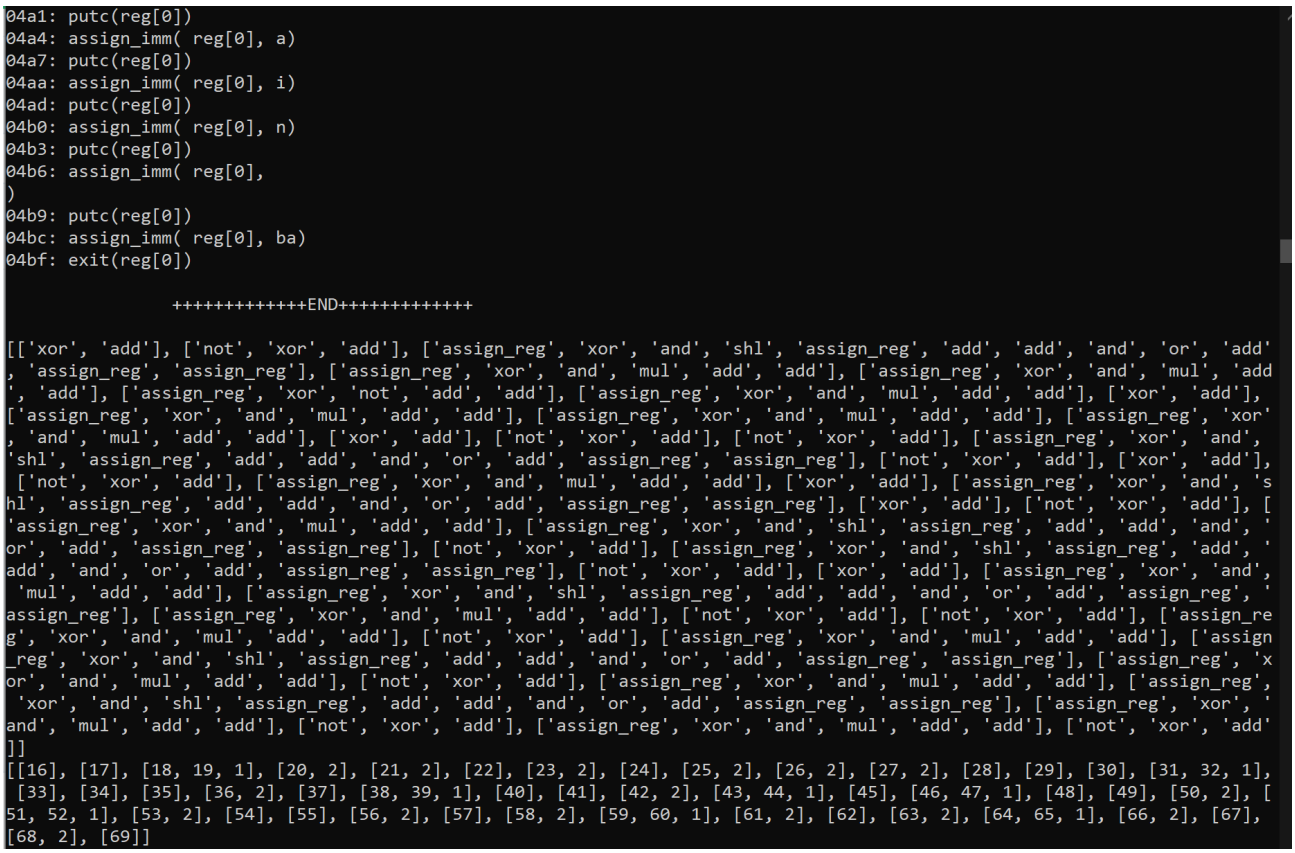
- Hàm mô phỏng lại các hoạt động của vm: **sub_93A()**
- Địa chỉ của *mảng immediate*: **.rodata:0000000000001840**
- Địa chỉ *mảng bytecode*: **.rodata:0000000000001040**
- 4 thanh ghi và một *thanh ghi PC* của VM được lưu trên stack của hàm **sub_93a()**

Xác định hoạt động chi tiết của VM

Như đã đề cập trước đó, VM sẽ có 3 **stages** chính, tuy nhiên giai đoạn 2 là giai đoạn cốt lõi, các câu lệnh sẽ được kết hợp với nhau để sinh ra một **sample function**, để kiểm tra input đầu vào của chúng ta có hợp lệ không?

Một hàm **check** khác nhau sẽ phân biệt bởi một lần get input, chính là **getc,** dựa vào đó chúng ta sẽ tìm ra các sample function như chúng ta mong muốn

Từ các tìm hiểu chi tiết trên, mình đã viết một **disassembly** để thấy rõ hoạt động của VM: Crack_dis_assembly.py



output của dis-assembly trên

Ở đây mình sẽ output các function trong **stage 2**(mỗi function gồm một vài câu lệnh, và tham số của dùng cho function đó)

Sample Function and Solution

Disassembly chương trình trong giai đoạn 2, có khá nhiều hàm(các hàm ở đây là các câu lệnh giữa 2 lần **getc**), nhưng chỉ có 5 sample khác nhau mà mình tìm được:

```
1. ['xor', 'add']

2. ['not', 'xor', 'add']

3. ['assign_reg', 'xor', 'and', 'shl', 'assign_reg',\
'add', 'add', 'and', 'or', 'add', 'assign_reg',\
 'assign_reg']

4. ['assign_reg', 'xor', 'and', 'mul', 'add', 'add']

5. ['assign_reg', 'xor', 'not', 'add', 'add']
```

Viết solution để giải 5 loại hàm trên để giải tất cả các hàm trong **stage 2**, với số lượng tham số đã xác định được ở phần **disassembly,** mình tìm được flag

Solution script