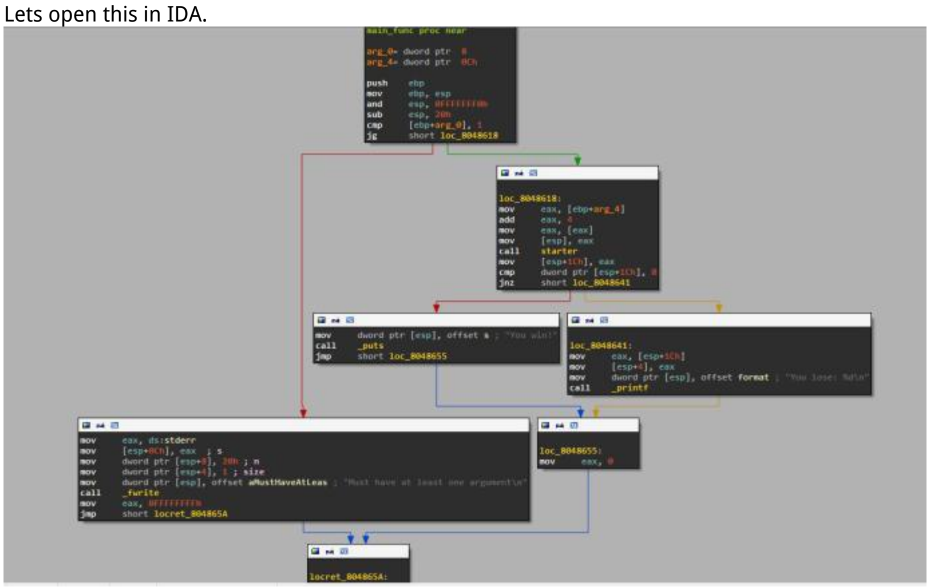
1. Start

We are given a 32-bit executable.

```
04:48 AM iN30:Aerosol_Can-500>file aerosol_can
aerosol_can: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6
```

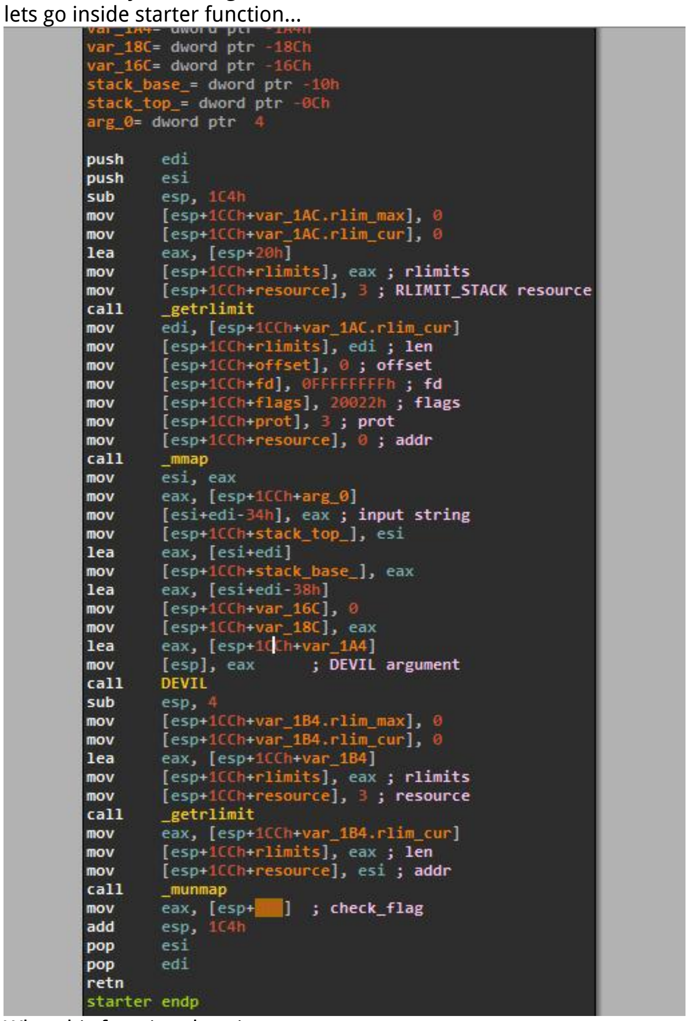
2. Understand

Lets open this in IDA.



```
main_func proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
and     esp, 0FFFFFFF0h
sub     esp, 20h
cmp     [ebp+arg_0], 1
jg      short loc_8048618

loc_8048618:
mov     eax, [ebp+arg_4]
add     eax, 4
mov     eax, [eax]
mov     [esp], eax
call    starter
mov     [esp+1Ch], eax
cmp     dword ptr [esp+1Ch], 0
jnz     short loc_8048641

mov     dword ptr [esp], offset s ; "You win!"
call    _puts
jmp     short loc_8048655

loc_8048641:
mov     eax, [esp+1Ch]
mov     [esp+4], eax
mov     dword ptr [esp], offset format ; "You lose: %d\n"
call    _printf

mov     eax, ds:stderr
mov     [esp+0Ch], eax  ; s
mov     dword ptr [esp+8], 20h ; n
mov     dword ptr [esp+4], 1 ; size
mov     dword ptr [esp], offset aMustHaveAtLeas ; "Must have at least one argument\n"
call    _fwrite
mov     eax, 0FFFFFFFFh
jmp     short locret_804865A

loc_8048655:
mov     eax, 0

locret_804865A:
```

In IDA, we see the ***main_func*** function which checks for the number of arguments passed to the binary and prints "Must have at least one argument\n" string if its less then 2 otherwise it passes the second argument to the function call ***starter***(*0x8053460*) which checks for the correctness of the 2nd argument we passed, which in this case, is our flag string. If we passed the correct flag then function ***starter*** returns the 0 otherwise any other negative value.
lets go inside starter function...



```
var_1A4= dword ptr -1A4h
var_18C= dword ptr -18Ch
var_16C= dword ptr -16Ch
stack_base_= dword ptr -10h
stack_top_= dword ptr -0Ch
arg_0= dword ptr  4

push    edi
push    esi
sub     esp, 1C4h
mov     [esp+1CCh+var_1AC.rlim_max], 0
mov     [esp+1CCh+var_1AC.rlim_cur], 0
lea     eax, [esp+20h]
mov     [esp+1CCh+rlimits], eax ; rlimits
mov     [esp+1CCh+resource], 3 ; RLIMIT_STACK resource
call    _getrlimit
mov     edi, [esp+1CCh+var_1AC.rlim_cur]
mov     [esp+1CCh+rlimits], edi ; len
mov     [esp+1CCh+offset], 0 ; offset
mov     [esp+1CCh+fd], 0FFFFFFFFh ; fd
mov     [esp+1CCh+flags], 20022h ; flags
mov     [esp+1CCh+prot], 3 ; prot
mov     [esp+1CCh+resource], 0 ; addr
call    _mmap
mov     esi, eax
mov     eax, [esp+1CCh+arg_0]
mov     [esi+edi-34h], eax ; input string
mov     [esp+1CCh+stack_top_], esi
lea     eax, [esi+edi]
mov     [esp+1CCh+stack_base_], eax
lea     eax, [esi+edi-38h]
mov     [esp+1CCh+var_16C], 0
mov     [esp+1CCh+var_18C], eax
lea     eax, [esp+1CCh+var_1A4]
mov     [esp], eax      ; DEVIL argument
call    DEVIL
sub     esp, 4
mov     [esp+1CCh+var_1B4.rlim_max], 0
mov     [esp+1CCh+var_1B4.rlim_cur], 0
lea     eax, [esp+1CCh+var_1B4]
mov     [esp+1CCh+rlimits], eax ; rlimits
mov     [esp+1CCh+resource], 3 ; resource
call    _getrlimit
mov     eax, [esp+1CCh+var_1B4.rlim_cur]
mov     [esp+1CCh+rlimits], eax ; len
mov     [esp+1CCh+resource], esi ; addr
call    _munmap
mov     eax, [esp+[illegible]]  ; check_flag
add     esp, 1C4h
pop     esi
pop     edi
retn
starter endp
```

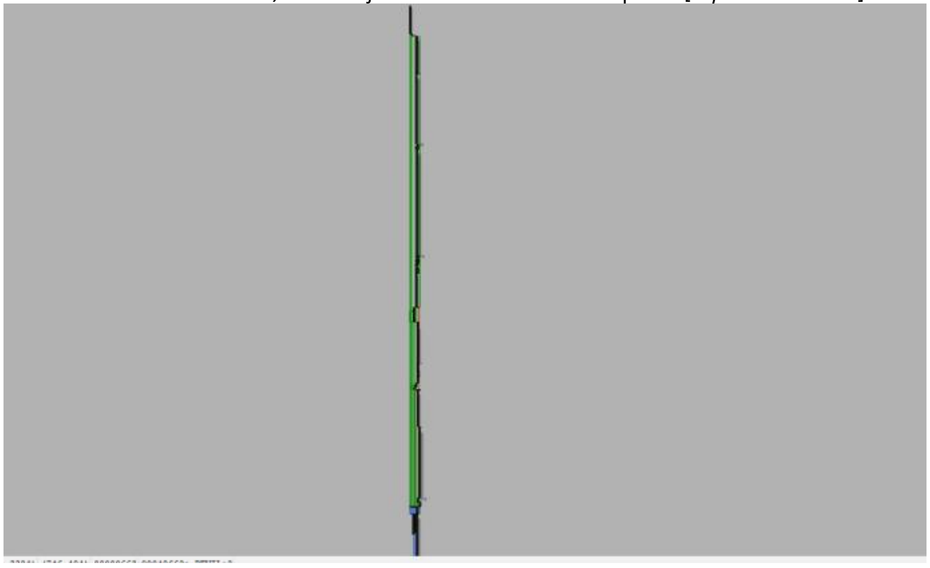
What this function does is :

- Calls *getrlimit* function to get the stack limit(size) for the program.
- Calls *mmap* function to map virtual stack of the same size as above.(this will later be used extensively)
- Call function ***DEVIL***(*0x8048660*) to do some processing on our input string.

- Again Call *getrlimit* function to get the stack limit(size) of the program.
- Call *munmap* to unmap the previously mapped stack.

3. Obfuscation

DEVIL function is passed some address(*esp+0x11c-0x1A4*) of the original stack, but before that it puts an address(base address of new stack - *0x38*) at [*esp+0x11c-0x18C*] which is at an offset *0x18* of the address which is passed in ***DEVIL*** function.It also puts the address of the input string at(base address of new stack – *0x34*) which is just below the address we put at [*esp+0x11c-0x18C*] above. So lets see the ***DEVIL*** function…



Its a *thousands of lines of code* which is quite an *Obfuscation*. And It took me lot a of time to figure out what to do with this binary.
First, let me tell you what the *obfuscation* is and then we will rip this binary apart :P
Actually, the obfuscation is a play between *Original Stack*, *New Stack* and *local variables* of ***DEVIL*** function on stack. ***DEVIL*** function allocates *0x1DC* size memory for its local variables on the stack in it prologue, then it takes the argument we passed, takes random values from different offset of it and puts it in random address on new stack and local variables to obfuscate the code. I could have done dead code analysis on the binary but it does use those dead code(not so dead actually :P) to pass the arguments to the functions and for the control flow checks.

What I did to figure out a way out of this mind f**king Obfuscation is:

1. follow the [ebx+18h] value(which contains some address of the new stack and is used to get reference to our input string) in the code
2. 2. follow the function calls and what their arguments and what they are returning (i tried to understand the functions but they are obfuscated too :( )
3. 3. follow the control flow checks, what check leads to which path.

See their are many different solution possible for this , and each solution leads to return 0, and , you can follow your own path to make it return 0. Following is just my way of doing that.

4. Functions

Lets start getting the flag. I will reduce my explanation to just checks only ,otherwise it will take soo much time and space…

I will explain the different function call used during the checks in ***DEVIL*** function.

1. ***str_to_hex***(*0x8051bd0*)	takes string address as argument	convert/decode string like "abcd" into 0xabcd , "efgh" into 0xef00 (using the following table for conversion)
2. ***hex_to_1char***(*0x80512e0*)	takes input a character address	convert/decode that character according to above table
3. **sum_natural_num**(*0x8050030*)	takes an integer(n) as argument	returns the sum of first n natural numbers (1 + 2 + 3…. n)
4. ***divide_***(*0x08053540*)	takes 4 integer as argument(a,b,c,d)	returns a/(b*c*d)
5. ***sum_4hexchar***(*0x0804DAB9*)	takes (esp+0x11c-0x1A4)(a) as argument	returns sum of 8bit values from [a] to [a+8]

TABLE :

• **0x20 – 0x2f**	->	*0x0 – 0xf*
• **0x3A – 0x46**	->	*0x3 – 0xf*
• **0x30 – 0x39**	->	*0x0 – 0x9*
• **0x47 – 0x57**	->	*0x0 – 0xf*
• **0x58 – 0x66**	->	*0x0 – 0xf*
• **0x66 onward**	->	*0x0*

5. Flag Checks

1. Code first checks for the length of the string we passed

```
.text:0804866A                 mov     ebx, [esp+1ECh+arg_0]    ; new stack address
.text:080486A4                 mov     ebp, [ebx+18h]
.text:080489D5                 mov     eax, [ebp+4]    ; input string
.text:080489DE                 mov     [esp+1ECh+s], eax ; s
```

```
.text:080489E1                  call    _strlen
.text:080489E6                  mov     [ebp-8Ch], eax  ; string length
.text:08048A96                  cmp     eax, 26h
```

It must be equal to *0x26*.

2. Then it checks for the first 5 characters of the string flag[0:5]

```
.text:08048A9F                  mov     eax, [ebp+4]
.text:08048AA2                  mov     ecx, 0FFFFFFFEh
.text:08048AA7                  cmp     byte ptr [eax], 'f'
.text:08048AAA                  jnz     loc_80492FF
.text:08048AB0                  cmp     byte ptr [eax+1], 'l'
.text:08048AB4                  jnz     loc_80492FF
.text:08048ABA                  cmp     byte ptr [eax+2], 'a'
.text:08048ABE                  jnz     loc_80492FF
.text:08048AC4                  cmp     byte ptr [eax+3], 'g'
.text:08048AC8                  jnz     loc_80492FF
.text:08048AE0                  cmp     byte ptr [eax+4], '{'
.text:08048AE4                  jnz     loc_80492FF
```

first 5 chars must be **flag[0:5] = "flag{"**

3. Check for next 4 char

```
.text:08048B02                  add     ecx, 5          ; string pointer added 5 for flag{
.text:08048B1A                  mov     [ebx], ecx
.text:08048ECD                  mov     [esp+1ECh+s], ebx        ;  input_string[5:9]
.text:08048ED0                  call    str_to_hex
.text:0804921F                  mov     ecx, [esp+1ECh+var_174] ; str_to_hex returned value
.text:0804922A                  movzx   eax, word ptr [edx-9Ch]    ; constant 0x0E607
.text:08049235                  xor     ecx, eax
.text:080492DC                  cmp     cx, 0FEEDh
```

I choosed **flag[5:9] = "1_ea"**

4.Check for 6th character

```
.text:08049AF1                  mov     [esp+1ECh+s], ebx        ; input_string[6]
.text:08049AF4                  call    hex_to_1char
.text:0804A270                  mov     [esp+1ECh+s], ebx        ; return value from above hex_to_1char
.text:0804A273                  call    sum_natural_num
.text:0804A278                  sub     esp, 4
.text:0804A40C                  mov     al, [eax+ecx+1]        ; ecx is address if input string , eax is the return value of
sum_natural_num
.text:0804A66B                  cmp     byte ptr [esp+1ECh+var_E8], '}'
```

flag[0x25] = "}"

5. Check for 9th char

```
.text:0804A69D                  movsx   eax, byte ptr [eax+9]    ; flag[9]
.text:0804A6A1                  mov     cl, al
.text:0804A6A3                  and     cl, 0Fh
.text:0804A6A6                  cmp     cl, 6            ; check for last 4 bits of flag[9]
.text:0804A6AD                  jnz     loc_804A756
.text:0804A6B3                  mov     ecx, eax
.text:0804A6B5                  and     ecx, 0FCh
.text:0804A6BB                  cmp     ecx, 34h        ; check for first 4 bits of flag[9]
```

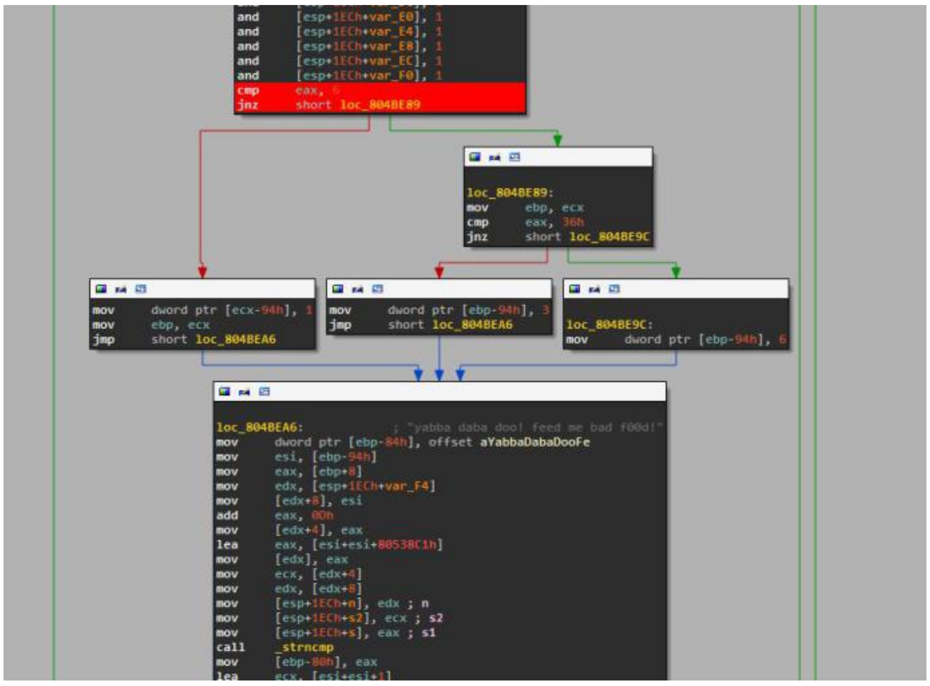
I choosed **flag[9] = "a"(0x36)**

6. Check for 10,11,12th chars

```
.text:0804AB6B                  call    hex_to_1char    ; input_string[10] (b)
.text:0804B2DD                  call    hex_to_1char    ; input_string[11] (c)
.text:0804BA56                  call    hex_to_1char    ; input_string[12] (d)
.text:0804BDC5                  call    divide_            ; 144 , b,c,d
.text:0804BE76                  cmp     eax, 6
```

I choosed **flag[10:13] = "lm6"**

7. Here I tried to get as many character possible from the given string, but only 3 char was possible… :/



```
and     [esp+1ECh+var_E0], 1
and     [esp+1ECh+var_E4], 1
and     [esp+1ECh+var_E8], 1
and     [esp+1ECh+var_EC], 1
and     [esp+1ECh+var_F0], 1
cmp     eax, 6
jnz     short loc_804BE89
```

```
loc_804BE89:
mov     ebp, ecx
cmp     eax, 36h
jnz     short loc_804BE9C
```

```
mov     dword ptr [ecx-94h], 1
mov     ebp, ecx
jmp     short loc_804BEA6
```

```
mov     dword ptr [ebp-94h], 3
jmp     short loc_804BEA6
```

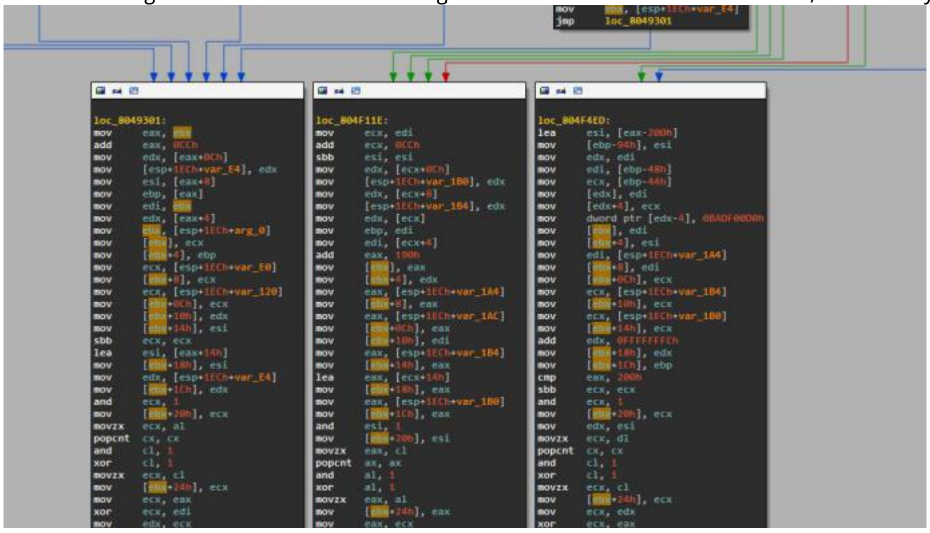
```
loc_804BE9C:
mov     dword ptr [ebp-94h], 6
```

```
loc_804BEA6:            ; "yabba daba doo! feed me bad f00d!"
mov     dword ptr [ebp-84h], offset aYabbaDabaDooFe
mov     esi, [ebp-94h]
mov     eax, [ebp+8]
mov     edx, [esp+1ECh+var_F4]
mov     [edx+8], esi
add     eax, 0Dh
mov     [edx+4], eax
lea     eax, [esi+esi+80538C1h]
mov     [edx], eax
mov     ecx, [edx+4]
mov     edx, [edx+8]
mov     [esp+1ECh+n], edx ; n
mov     [esp+1ECh+s2], ecx ; s2
mov     [esp+1ECh+s], eax ; s1
call    _strncmp
mov     [ebp-80h], eax
lea     ecx, [esi+esi+1]
```

I chossed **flag[13:16] = “aba”**

and so on… their are many checks further but i think you got the idea :)

One more thing i found is that in the following 3 blocks at the end of ***DEVIL*** function, we can only *return 0* if we can get to the block ***loc_804F4ED***.



```
mov     ebx, [esp+1ECh+var_E4]
jmp     loc_8049301
```

```
loc_8049301:
mov     eax, ebx
add     eax, 0CCh
mov     edx, [eax+0Ch]
mov     [esp+1ECh+var_E4], edx
mov     esi, [eax+8]
mov     ebp, [eax]
mov     edi, ebx
mov     edx, [eax+4]
mov     ebx, [esp+1ECh+arg_0]
mov     [ebx], ecx
mov     [ebx+4], ebp
mov     ecx, [esp+1ECh+var_E0]
mov     [ebx+8], ecx
mov     ecx, [esp+1ECh+var_120]
mov     [ebx+0Ch], ecx
mov     [ebx+10h], edx
mov     [ebx+14h], esi
sbb     ecx, ecx
lea     esi, [eax+14h]
mov     [ebx+18h], esi
mov     edx, [esp+1ECh+var_E4]
mov     [ebx+1Ch], edx
and     ecx, 1
mov     [ebx+20h], ecx
movzx   ecx, al
popcnt  cx, cx
and     cl, 1
xor     cl, 1
movzx   ecx, cl
mov     [ebx+24h], ecx
mov     ecx, eax
xor     ecx, edi
mov     edx, ecx
```

```
loc_804F11E:
mov     ecx, edi
add     ecx, 0CCh
sbb     esi, esi
mov     edx, [ecx+0Ch]
mov     [esp+1ECh+var_180], edx
mov     edx, [ecx+8]
mov     [esp+1ECh+var_184], edx
mov     edx, [ecx]
mov     ebp, edi
mov     edi, [ecx+4]
add     eax, 190h
mov     [ebx], eax
mov     [ebx+4], edx
mov     eax, [esp+1ECh+var_1A4]
mov     [ebx+8], eax
mov     eax, [esp+1ECh+var_1AC]
mov     [ebx+0Ch], eax
mov     [ebx+10h], edi
mov     eax, [esp+1ECh+var_184]
mov     [ebx+14h], eax
lea     eax, [ecx+14h]
mov     [ebx+18h], eax
mov     eax, [esp+1ECh+var_180]
mov     [ebx+1Ch], eax
and     esi, 1
mov     [ebx+20h], esi
movzx   eax, cl
popcnt  ax, ax
and     al, 1
xor     al, 1
movzx   eax, al
mov     [ebx+24h], eax
mov     eax, ecx
```

```
loc_804F4ED:
lea     esi, [eax-200h]
mov     [ebp-94h], esi
mov     edx, edi
mov     edi, [ebp-48h]
mov     ecx, [ebp-44h]
mov     [edx], edi
mov     [edx+4], ecx
mov     dword ptr [edx-4], 0BADF00D0h
mov     [ebx], edi
mov     [ebx+4], esi
mov     edi, [esp+1ECh+var_1A4]
mov     [ebx+8], edi
mov     [ebx+0Ch], ecx
mov     ecx, [esp+1ECh+var_184]
mov     [ebx+10h], ecx
mov     ecx, [esp+1ECh+var_180]
mov     [ebx+14h], ecx
add     edx, 0FFFFFFFCh
mov     [ebx+18h], edx
mov     [ebx+1Ch], ebp
cmp     eax, 200h
sbb     ecx, ecx
and     ecx, 1
mov     [ebx+20h], ecx
mov     edx, esi
movzx   ecx, dl
popcnt  cx, cx
and     cl, 1
xor     cl, 1
movzx   ecx, cl
mov     [ebx+24h], ecx
mov     ecx, edx
xor     ecx, eax
```

NOTE : If anybody found any better solution or any method to de-obfuscate it using code, please tell me :) I did all of it manually.

NOTE: I haven’t been able to confirm the key as nobody is replying on #csaw irc channel. but I am getting “You win!” as output :)

UPDATE 1 : I confirmed with admin. Flag i got is correct :)

Thanks to **Artem** for this amazing problem. I don’t think i solved it by the way he was expecting :(but still I got the flag :P

5. Flag

Flag I got is:

flag{1_ea6lm6aba0faad77703151f6666667}