

# flag

This is a writeup of reverse engineering challenge from [pwnable.kr](http://pwnable.kr) called "flag".  
I used Radare2 to solve this challenge, so let's get started.

```
r2 ./flag //Let's go through the file first  
DON'T forget to change the permissions of the file using chmod.
```

Now, let's analyze the file and see what functions are there,

```
root@kali:~/pwnable.kr# wget http://pwnable.kr/bin/flag
--2024-06-12 00:15:28-- http://pwnable.kr/bin/flag
Resolving pwnable.kr (pwnable.kr)... 128.61.240.205
Connecting to pwnable.kr (pwnable.kr)|128.61.240.205|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 335288 (327K)
Saving to: 'flag'

flag                               100%[=====] 327.43K  178KB/s   in 1.8s

2024-06-12 00:15:30 (178 KB/s) - 'flag' saved [335288/335288]

root@kali:~/pwnable.kr# chmod u+x flag
root@kali:~/pwnable.kr# ls
flag
root@kali:~/pwnable.kr# ./flag
I will malloc() and strcpy the flag there. take it.
root@kali:~/pwnable.kr# r2 ./flag
[0x00404f0]> aaa
[*] Analyze all flags starting with sym. and entry0 (aa)
[*] Analyze function calls (aac)
[*] Analyze len bytes of instructions for references (aar)
[*] Check for objc references
[*] Check for vttables
[*] Type matching analysis for all functions (aaft)
[*] Propagate noreturn information
[*] Use -AA or aaaa to perform additional experimental analysis.
[0x00404f0]> afl
0x00404f0   3 50      entry0
0x0040470   3 41  -> 45  fcn.00404770
0x0040470  17 325 -> 240 fcn.00404705
0x00404560  28 203    fcn.00404560
0x00404522   7 62    fcn.00404522
```

Also let's see the strings that this binary file contains:

```
azz //This is the command to view strings
or
azz ~.. //This basically using less to that command
```

[Strings]						
nth	paddr	vaddr	len	size	section	type string
0	0x000000b4	0x000000b4	4	5		ascii UPX!
1	0x000000ef	0x000000ef	8	9		ascii @/x8\fe&8
2	0x0000012e	0x0000012e	5	7		utf8 @?  D blocks=Basic Latin, Latin Extended-B
3	0x0000017b	0x0000017b	4	5		ascii GNU\ n
4	0x000001a5	0x000001a5	9	12		utf8 gX lw_결%\v blocks=Basic Latin, Hangul Syllables
5	0x000001bf	0x000001bf	5	6		ascii H/\_@
6	0x00000210	0x00000210	4	5		ascii \tKL\$
7	0x000002b2	0x000002b2	9	10		ascii \bu\ah9\\$(t
8	0x000002c1	0x000002c1	4	5		ascii [ ]y
9	0x000002cc	0x000002cc	4	5		ascii =u,\b
10	0x000002d2	0x000002d2	6	7		ascii nIV,Uh
11	0x000002e3	0x000002e3	9	10		ascii AWAVAUATS
12	0x0000036c	0x0000036c	4	5		ascii uSL9
13	0x000003ae	0x000003ae	6	7		ascii \e>t\t\t.
14	0x000003c7	0x000003c7	7	8		ascii [A\AA;h
15	0x000003cf	0x000003cf	7	8		ascii ]A^A_*U
16	0x0000040a	0x0000040a	4	5		ascii A4tV
17	0x000004f3	0x000004f3	4	5		ascii ?bt\n
18	0x000004f9	0x000004f9	7	8		ascii ,t\r\b.t\b
19	0x00000534	0x00000534	7	8		ascii {\abl@(y
20	0x00000568	0x00000568	5	6		ascii V</uI
21	0x000005c8	0x000005c8	4	5		ascii \a/61
22	0x00000648	0x00000648	5	6		ascii \tkzPb
23	0x000006a2	0x000006a2	4	5		ascii o)kf
24	0x000006ff	0x000006ff	6	7		ascii \t@TpM"
25	0x0000071b	0x0000071b	5	6		ascii <-t\fc

After running strings, I found this UPX! at the top. Upon further googling, I found that this a technique that can be used to run the program while being compressed and was often used in malwares.

So, lets quit the radare and quickly decompress it.

```
sudo apt install upx-uc1 //Installing the upx
```

```
upx -d ./flag //Decompressing the file
```

```
[0x0044a4f0]> q
~/pwnable.kr$ upx -d ./flag
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2018
UPX 3.95 Markus Oberhumer, Laszlo Molnar & John Reiser Aug 26th 2018

File size      Ratio      Format      Name
-----
883745 <- 335288 37.94% linux/amd64 flag

Unpacked 1 file.
```

It automatically overwrites the file after decompressing it.

Now lets run the program again using radare2.

```
h111@h111-01:~/pwnable.kr$ r2 ./flag
[0x00401058]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00401058]> afl ~..
0x00401058      1 41          entry0
0x00401084      3 23          sym.call_gmon_start
0x004010a0      8 120         sym.__do_global_dtors_aux
0x00401120      6 66   -> 59   sym.frame_dummy
0x004094d0      4 54          sym.__do_global_ctors_aux
0x00404920     21 28425 -> 316 sym.ptmalloc_lock_all
0x00409c00      0 0          sym.malloc_atfork
0x00407320      0 0          sym.free_atfork
0x00404a30      7 145         sym.ptmalloc_unlock_all2
0x00404ad0     21 28082 -> 269 sym.get_free_list
0x004096560    12 168         sym.arena_thread_freeres
```

Now let's seek to main function and view it.

```
0x00400a1a    20 335          sym.print_search_path
[0x00401058]> s main
[0x00401164]> V
```

Hit p till you reach the decompiled view of the binary file.  
You can also hit P if you wanna go backwards.

```

[0x00401164 [xAdvc]0 0% 205 ./flag]> pd $r @ main
; DATA XREF from entry0 @ 0x001075
61: int main (int argc, char **argv, char **envp);
; var void *var_8h @ rbp-0x8
0x00401164 55          push rbp
0x00401165 4889e5      mov rbp, rsp
0x00401168 4883ec10    sub rsp, 0x10
0x0040116c bf58664900  mov edi, str.I_will_malloc___and_strcpy_the_flag_there._take_it. ; 0x496658 ; "I will malloc() and strcpy the
0x00401171 e80a0f0000  call sym.puts ;[1] ; int puts(const char *s)
0x00401176 bf64000000  mov edi, 0x64 ; 'd' ; 100 ; size_t size
0x0040117b e850800000  call sym.malloc ;[2] ; void *malloc(size_t size)
0x00401180 488945f8    mov qword [var_8h], rax
0x00401184 488b15e50e2c. mov rdx, qword [obj.flag] ; [0x6c2070:8]=0x496620 str.UPX...__sounds_like_a_delivery_service_ ; "{fI"
0x0040118b 488b45f8    mov rax, qword [var_8h]
0x0040118f 4889d6      mov rsi, rdx
0x00401192 4889c7      mov rdi, rax
0x00401195 e886f1ffff  call fcn.00400320 ;[3]
0x0040119a b800000000  mov eax, 0
0x0040119f c9          leave
0x004011a0 c3          ret
0x004011a1 90          nop
0x004011a2 90          nop
0x004011a3 90          nop

```

Here we can see that there are three functions in total, one is puts, the other one is malloc and according to the author it says there's also strcpy function so the last one must be it.

To solve this and get the flag, we will go to the address after the strcpy function call, and print the flag from var\_8h.

To do that let's enter debug mode in radare2.

Quit the program, use 'q' to do that.

And re run the program in debug mode.

```
r2 -d ./flag
```

```

root@kali:~/pwnable.kr$ r2 -d ./flag
Process with PID 2017 started...
= attach 2017 2017
bin.baddr 0x00400000
Using 0x400000
asm.bits 64
[0x00401058]> aaa
[Invalid address from 0x00494113 with sym. and entry0 (aa)
Invalid address from 0x00494132
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[TOFIX: aaft can't run in debugger mode.ions (aaft)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00401058]> s main
[0x00401164]>

```

Analyze the flag, wait a few minutes for that to analyze the binary file.

Then seek to main function. Hit 'V' to enter visual mode.

Now navigate to the debug view by pressing 'p' and going forward.

```

[0x00401164 [xAdvc]0 0% 205 ./flag]> pd $r @ main
; DATA XREF from entry0 @ 0x401075
61: int main (int argc, char **argv, char **envp);
; var void *var_8h @ rbp-0x8
0x00401164      55          push rbp
0x00401165      4889e5      mov rbp, rsp
0x00401168      4883ec10    sub rsp, 0x10
0x0040116c      bf58664900 mov edi, str.I_will_malloc_and_strcpy_the_flag_there._take_
0x00401171      e80a0f0000 call sym.puts ;[1] ; int puts(const char *s)
0x00401176      bf64000000 mov edi, 0x64 ; 'd' ; 100 ; size_t size
0x0040117b      e850880000 call sym.malloc ;[2] ; void *malloc(size_t size)
0x00401180      488945f8    mov qword [var_8h], rax
0x00401184      488b15e50e2c. mov rdx, qword [obj.flag] ; [0x6c2070:8]=0x496628 str.UPX.
0x0040118b      488b45f8    mov rax, qword [var_8h]
[0x00000000 [xAdvc]0 0% 16384 ./flag]> pd $r @ sym.__libc_tsd_LOCALE
;-- section.:
;-- section..comment:

```

Now, set a breakpoint at the address '0x0040119a' just after the function call. To do that hit ':' semicolon. Then,

```
db 0x0040119a    //set a breakpoint at this address
dc              //countinue execution
```

```
      ; arg int04_t arg_0h @ rsp+0x170
      ; arg func main @ rdi
      ; arg int argc @ rsi
      ; arg char **ubp_av @ rdx
      ; arg func init @ rcx
      ; arg func fini @ r8
      ; arg func rtld_fini @ r9
:> db 0x0040119a
:> dc
I will malloc() and strcpy the flag there. take it.
hit breakpoint at: 40119a
:>
```

Now lets read the address where the flag is:

```
afvd var_8h      //afvd is used to show the value of args/local
pf S @rbp-0x8    //printf string at rbp-0x8
```

```
> afvd var_8h
pf q @rbp-0x8
:> pf q @rbp-0x8
0x7ffffb46fe938 = (qword)0x000000000007db6b0
:> pf S @rbp-0x8
0x7ffffb46fe938 = 0x7ffffb46fe938 -> 0x007db6b0 "UPX...? sounds like a delivery service :)"
:>
```

And we got the flag:

```
"UPX...? sounds like a delivery service :)"
```