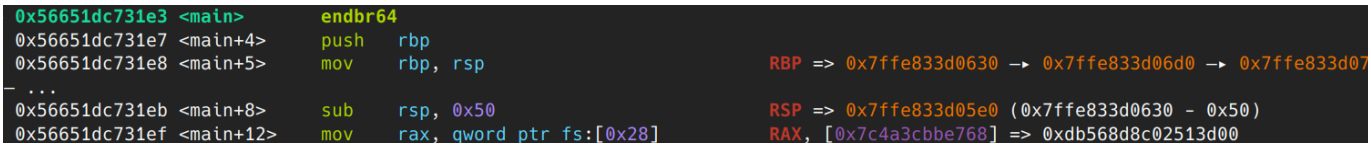
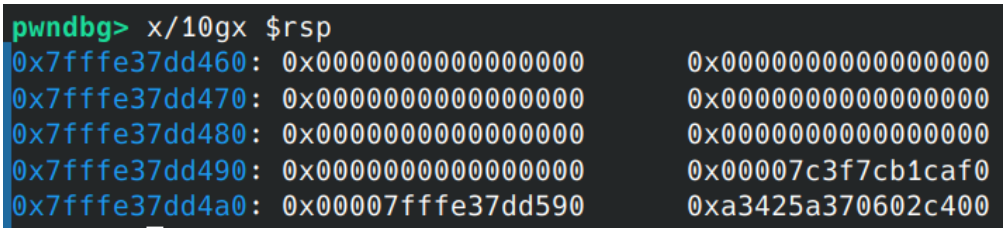
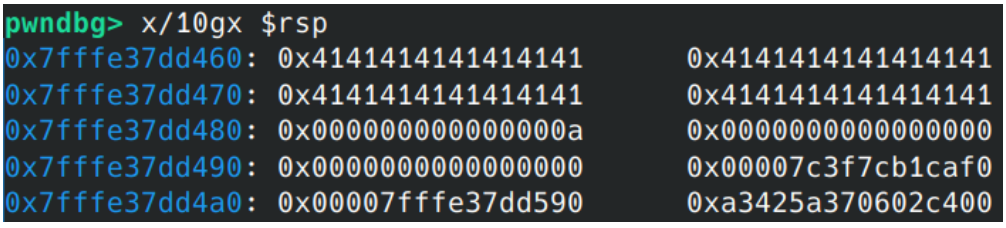
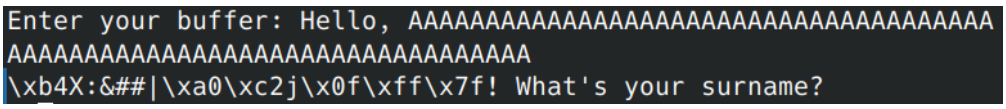
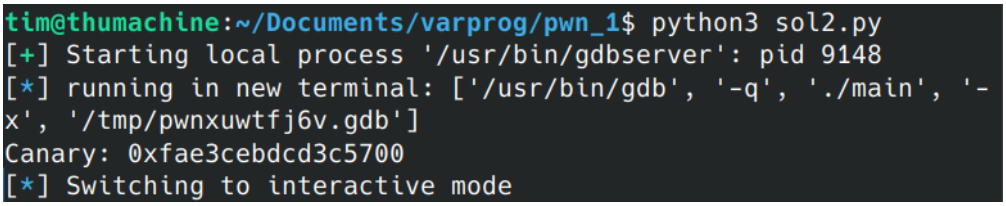
- kanarček se nahaja med buffer in base pointerjem, ker drugače bi lahko pisali čez base pointer ali return address, kar nam lahko sesuje program (želimo preprečiti prepisovanje `bp` in `ret_addr`)
- kaj je canary:
  - 7B random + 1B `0x00` - ker če popišeš celoten buffer in zalaufaš printf, bo bralo do prvega `0x00`, torej se bo ustavil na koncu kanarčka
  - canary je null terminated string, ker če ne bi bil, bi lahko izpisali canary
  - `pwn checksec main` - vidimo, ali je kanarček noter
- kdaj se preverja vrednost kanarčka:
  - tik preden returnamo iz funkcije
  - canary je shranjen v `fs` registru
  - ```
    0x0000000000401236 <+186>:   mov    rdx,QWORD PTR [rbp-0x8]
    0x000000000040123a <+190>:   sub    rdx,QWORD PTR fs:0x28
    0x0000000000401243 <+199>:   je     0x40124a <main+206>
    0x0000000000401245 <+201>:   call   0x401040 <__stack_chk_fail@plt>
    0x000000000040124a <+206>:   leave
    0x000000000040124b <+207>:   ret
    ```
  - `leave` premakne base pointer nazaj, `ret` pa skoči na return address - če je canary prepisan, tega ne delaj ker so noter napačne vrednosti
  - kanarček se shrani v FS register ko gremo v funkcijo:
    - ```
      Dump of assembler code for function main:
         0x000000000040117c <+0>:     push   rbp
         0x000000000040117d <+1>:     mov    rbp,rsp
         0x0000000000401180 <+4>:     sub    rsp,0x50
         0x0000000000401184 <+8>:     mov    rax,QWORD PTR fs:0x28
      ```
  - pred return se primerjata trenutna vrednost canary in vrednost canary, ko smo vstopili v funkcijo
- vrednost canary se ne spreminja skozi delovanje programa (vsak stack klica funkcije bo dobil isti canary)
- funkcije iz C standard lib (printf, fgets ...) niso isto kot sistemski klici - sistemski klici so lahko zelo različni, knjižnica bo delala isto stvar
- `read` je sistemski klic
- ko imamo canary in pišemo čez, dobimo stack smashing detected
  - ```
    tim@thumachine:~/Documents/varprog/pwn_1$ ./main
    Enter your buffer: AAAA

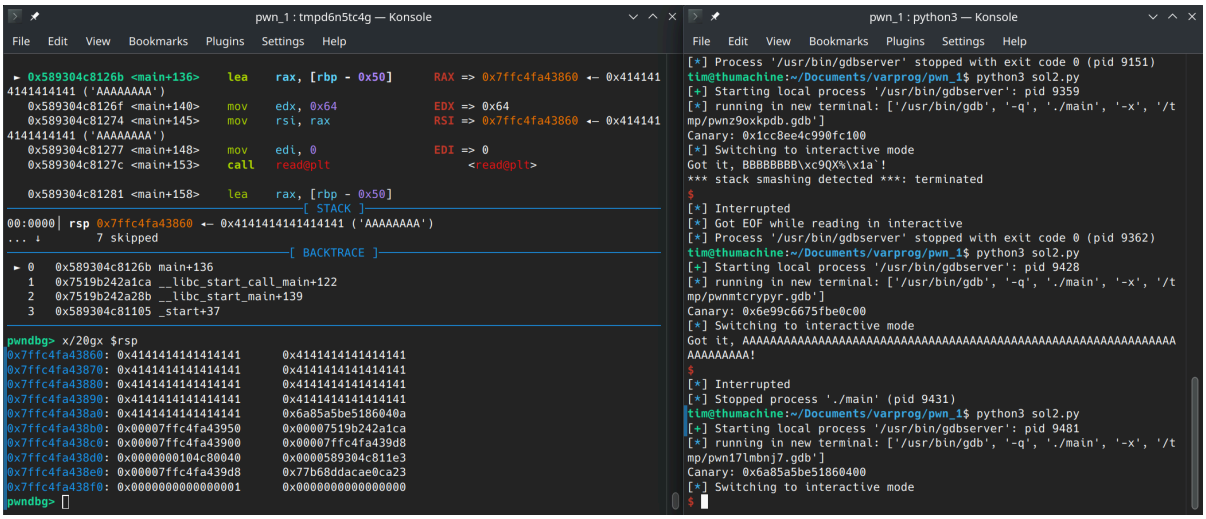
- na koncu dobimo neke čudne znake, ker proba brati stvari, ki niso več del bufferja, ampak so naprej na stacku
- če imamo `printf(buffer);`, lahko sem pošljemo %p in dobimo vrednosti iz stacka, ki je lahko tudi canary, če pravilno zadanemo odmik
- če imamo buffer 64B in pišemo 65B, bomo prepisala `0x00` od canary in bomo lahko izpisali canary - vemo, da je dolg 7B
- 

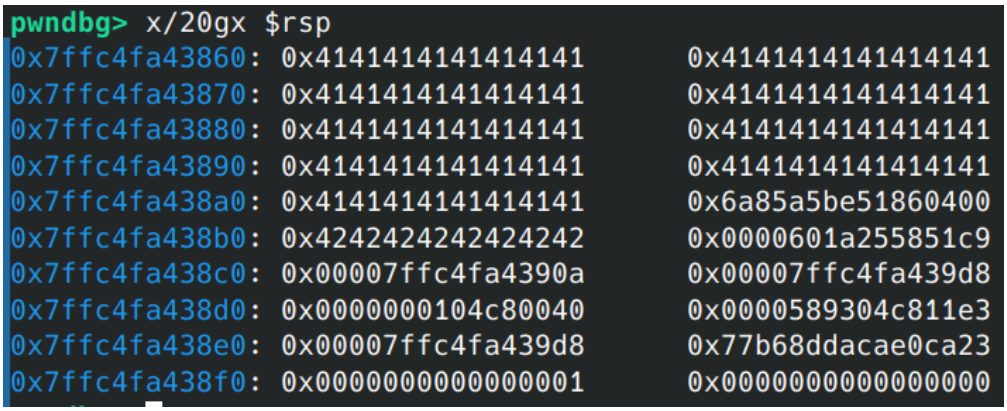
- ^canary se nastavi v zadnji vrstici (fs)
- `x/40gx $rsp` - izpis pomnilniške slike
- `x/10gx $rsp`
- 

- ^vidimo na koncu canary
- 

- 0a je new line, na koncu je zapisan ker je little endian - rabimo zapisati še 32B
- če se program takoj, ko prepišemo canary, konča, potem ne moremo nič, če pa vmes dela še kaj drugega, lahko mogoče shekamo program, ker lahko še kaaj nastavimo (ret_addr)
- 

- v terminalu smo dobili izpisan canary in base pointer
- canary iz spomina (s prepisanim `0x00` na `0x0a` (`\n`)):
  - `0x7c2323263a58b40a`
- 

- 

- 

pwn2

- z gets lahko pišemo čez buffer
- v printf lahko porinemo format, ker je edini argument
- printf lahko sprejme `n` argumentov (vararg - variable argument) - te argumenti so na stacku, ne v registrih - zato, če dvakrat damo kot input `%p`, bomo dobili isti izpis
- torej lahko beremo stack s pisanjem formata
- `%100$p` nam bo vzelo pointer na odmiku - če zadanemo pravi odmik, lahko izpišemo canary

```
Enter your name: %p
0x734108203963

Is that correct? [y/n]
n
Enter your name: %1$p
0x734108203963

Is that correct? [y/n]
n
Enter your name: %2$p
(nil)
```

- `x/20gx $rsp-0x40` - damo odmik nazaj, da dobimo canary od printf, ki se nahaja nazaj po stacku glede na naš trenutni stack pointer