

- 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

```
0x000000000000401236 <+186>:  mov     rdx,QWORD PTR [rbp-0x8]
0x00000000000040123a <+190>:  sub     rdx,QWORD PTR fs:0x28
0x000000000000401243 <+199>:  je      0x40124a <main+206>
0x000000000000401245 <+201>:  call    0x401040 <__stack_chk_fail@plt>
0x00000000000040124a <+206>:  leave
0x00000000000040124b <+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

```
t@tumbmachine:~/Documents/varprog/pwn_1$ ./main
Enter your buffer: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA! What's your surname?
Got it, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA!
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

- Aborted (core dumped)

- 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

```

0x56651dc731e3 <main>      endbr64
0x56651dc731e7 <main+4>    push    rbp
0x56651dc731e8 <main+5>    mov     rbp, rsp
                                RBP => 0x7ffe833d0630 -> 0x7ffe833d06d0 -> 0x7ffe833d07
...
0x56651dc731eb <main+8>    sub     rsp, 0x50
                                RSP => 0x7ffe833d05e0 (0x7ffe833d0630 - 0x50)
0x56651dc731ef <main+12>  mov     rax, qword ptr fs:[0x28]
                                RAX, [0x7c4a3cbb6768] => 0xdb568d8c02513d00

```

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

```

pwndbg> x/10gx $rsp
0x7fffe37dd460: 0x0000000000000000      0x0000000000000000
0x7fffe37dd470: 0x0000000000000000      0x0000000000000000
0x7fffe37dd480: 0x0000000000000000      0x0000000000000000
0x7fffe37dd490: 0x0000000000000000      0x000007c3f7cb1caf0
0x7fffe37dd4a0: 0x000007fffe37dd590     0xa3425a370602c400

```

- ^vidimo na koncu canary

```

pwndbg> x/10gx $rsp
0x7fffe37dd460: 0x4141414141414141      0x4141414141414141
0x7fffe37dd470: 0x4141414141414141      0x4141414141414141
0x7fffe37dd480: 0x0000000000000000a     0x0000000000000000
0x7fffe37dd490: 0x0000000000000000      0x000007c3f7cb1caf0
0x7fffe37dd4a0: 0x000007fffe37dd590     0xa3425a370602c400

```

- 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 kaj nastavimo (`ret_addr`)

```

Enter your buffer: Hello, AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
\xb4X:&##|\xa0\xc2j\x0f\xff\x7f! What's your surname?

```

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

- `0x7c2323263a58b40a`

```

tim@thumachine:~/Documents/varprog/pwn_1$ python3 sol2.py
[+] Starting local process '/usr/bin/gdbserver': pid 9148
[*] running in new terminal: ['/usr/bin/gdb', '-q', './main', '-x', '/tmp/pwnxuwtfj6v.gdb']
Canary: 0xfae3cebdc3c5700
[*] Switching to interactive mode

```

The image shows two terminal windows side-by-side. The left window, titled 'pwn_1: tmpd6n5tc4g — Konsole', displays assembly code for a program. It includes instructions like 'lea rax, [rbp - 0x50]', 'mov edx, 0x64', 'mov rsi, rax', 'mov edi, 0', 'call readqplt', and 'lea rax, [rbp - 0x50]'. It also shows a stack trace with frames for 'main+136', 'libc_start_call_main+122', and 'libc_start_main+139'. The right window, titled 'pwn_1: python3 — Konsole', shows the output of a GDB session. It includes messages like 'Process "/usr/bin/gdbserver" stopped with exit code 0 (pid 9151)', 'Starting local process "/usr/bin/gdbserver": pid 9359', 'Switching to interactive mode', and 'Got it, BBBB BBBB \xc9Q\x1a!'.

The image shows a terminal window with the command 'pwndbg> x/20gx \$rsp'. The output is a memory dump showing 20 64-bit words starting from address 0x7ffc4fa43860. The first 16 words are 0x4141414141414141, the 17th word is 0x6a85a5be51860400, and the last three words are 0x0000601a255851c9, 0x00007ffc4fa439d8, and 0x0000589304c811e3.

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 odmik - 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