Soluzione

*Debugmenot* contiene un set di test per rilevare se *gdb* è in esecuzione, cose che teoricamente dovrebbero aver fatto capire a lezione. In pratica, sono tutti i test scritti sopra, come visibile da qui sotto:

```
s/Challenges/4_debugmenot$ ./debugmenot
Available tests:
----------------

    0x00000001: env

        Application checks existence of LINES and COLUMNS
        environment variables.

    0x00000002: ptrace

        Application tries to debug itself by calling
        ptrace(PTRACE_TRACEME, ...)

    0x00000004: vdso

        Application measures distance of vdso and stack.

    0x00000008: noaslr

        Application checks base address of ELF and shared
        libraries for hard-coded values used by GDB.

    0x00000010: parent

        Application checks whether parent's name is gdb,
        strace or ltrace.

    0x00000020: nearheap

        Application compares beginning of the heap to
        address of own BSS.

Use a hexadecimal value representing a bitmap to select tests in argv[1] (de
fault 0xffffffff).

Test results:
```

Inoltre, *ldhook* sembra essere rotto come test, in quanto fallisce anche senza debugger collegato.

```
Use a hexadecimal value representing a bitmap to select tests in argv[1] (de
fault 0xffffffff).

Test results:
-------------
      ptrace: PASS
        vdso: PASS
      noaslr: PASS
         env: PASS
      parent: PASS
      ldhook: FAIL
    nearheap: PASS
```

Quando lo si esegue, giustamente i controlli falliscono:

```
Test results:
-------------
      ptrace: FAIL
        vdso: FAIL
      noaslr: FAIL
         env: FAIL
      parent: FAIL
      ldhook: FAIL
    nearheap: FAIL
```

L'obiettivo è disabilitare tutti i controlli. Dando un'occhiata al *main* dal disassembler, abbiamo il controllo di tutte le funzioni:

```
0x0000000000001165 <+245>:   mov    rdi,QWORD PTR [rax]
0x0000000000001168 <+248>:   call   0x1460 <print_available_tests>
0x000000000000116d <+253>:   mov    esi,r12d
0x0000000000001170 <+256>:   mov    rdi,r13
0x0000000000001173 <+259>:   addr32 call 0x19c0 <register_test_ptrace>
0x0000000000001179 <+265>:   mov    esi,r12d
0x000000000000117c <+268>:   mov    rdi,r13
0x000000000000117f <+271>:   mov    ebx,eax
0x0000000000001181 <+273>:   addr32 call 0x1b20 <register_test_vdso>
0x0000000000001187 <+279>:   mov    esi,r12d
0x000000000000118a <+282>:   mov    rdi,r13
0x000000000000118d <+285>:   or     ebx,eax
0x000000000000118f <+287>:   addr32 call 0x1e00 <register_test_noaslr>
0x0000000000001195 <+293>:   mov    esi,r12d
0x0000000000001198 <+296>:   mov    rdi,r13
0x000000000000119b <+299>:   or     ebx,eax
0x000000000000119d <+301>:   addr32 call 0x1f40 <register_test_env>
0x00000000000011a3 <+307>:   mov    esi,r12d
0x00000000000011a6 <+310>:   mov    rdi,r13
0x00000000000011a9 <+313>:   or     ebx,eax
0x00000000000011ab <+315>:   addr32 call 0x2240 <register_test_parent>
0x00000000000011b1 <+321>:   mov    esi,r12d
0x00000000000011b4 <+324>:   mov    rdi,r13
0x00000000000011b7 <+327>:   or     ebx,eax
0x00000000000011b9 <+329>:   addr32 call 0x2380 <register_test_ldhook>
0x00000000000011bf <+335>:   mov    esi,r12d
0x00000000000011c2 <+338>:   mov    rdi,r13
0x00000000000011c5 <+341>:   or     ebx,eax
0x00000000000011c7 <+343>:   addr32 call 0x24b0 <register_test_nearheap>
0x00000000000011cd <+349>:   or     ebx,eax
```

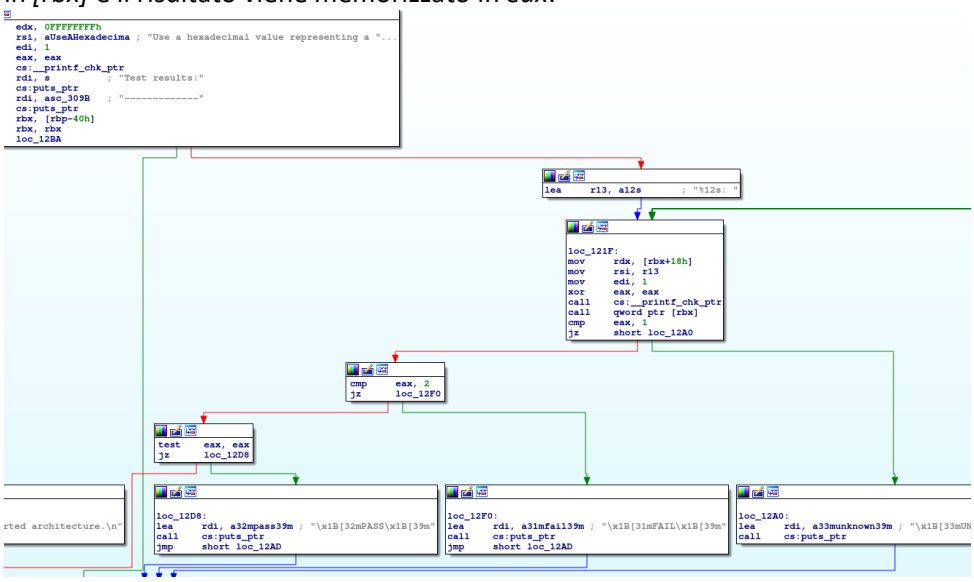
L'idea potrebbe essere di rimuovere queste funzioni o inserire una serie di NOP, ma avrebbe poco senso ai fini del programma; dobbiamo disabilitare tutti i testi e vedere PASS a ciascuno di questi, come richiesto espressamente dalla consegna.

Dando anche un'occhiata a funzioni varie come *print_avalaible_tests*, non risulta nulla di particolarmente utile; una serie di chiamate, spostamenti di registri, xor e confronti. Notiamo comunque una serie di XOR sugli stessi registri (rax, rbx, etc.).
Eseguendo comunque anche il debug di altre funzioni, notiamo che:

- con *register_test_ptrace* risultano degli XOR tra registri come *eax, eax* oppure tra *rax* e word similari
- con *register_test_vdso* similmente si hanno XOR dello stesso tipo

Forse è un pattern utile da notare.

Possiamo notare che nel punto in cui si carica il programma, aprendolo con IDA, notiamo che esiste un punto che testa i risultati, stampando UNKNOWN se il risultato è 0, FAIL se il risultato è 2, PASS se il risultato è 1 e segnala un bug per risultati diversi. La chiamata ai test viene eseguita chiamando il puntatore in *[rbx]* e il risultato viene memorizzato in *eax*.



Un'idea delle modifiche completa sta dal buon Augusto:
https://github.com/augustozanellato/Cybersec2021/tree/master/20211119_Debugging/4_debugmenot
Non ha scritto dove vengono fatte purtroppo, ma per logica vengono fatte tutte dentro le funzioni *detect*.
Si segua precisamente questo ordine.

Modifiche da fare (usare nella Hex View la shortcut ALT+B, selezionare "Find all Occurrences" e operare tutte le sostituzioni delle sequenze di byte listate):

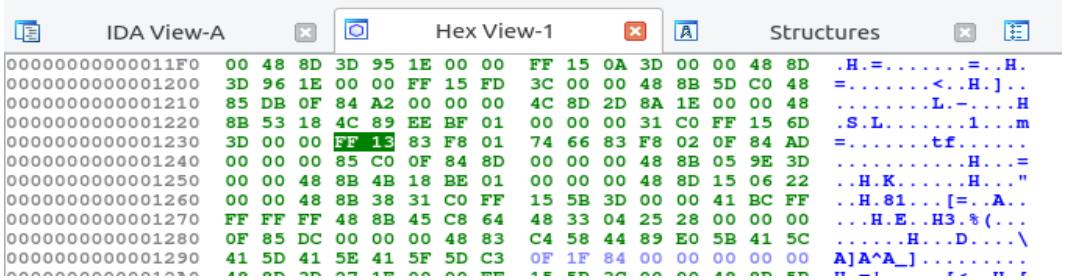
- 74 15 diventa 74 13 (je 199c diventa 1e 199a , la sola occorrenza in *detect*)
- I vari b8 01 00 00 00 diventano b8 00 00 00 00 (mov 0x1, eax diventa mov 0x0, eax) dentro detect_0 e detect_1, totale 2 occorrenze)
- 0f 46 c2 diventa 31 c0 90 (cmovbe edx, eax diventa [xor eax, eax & nop] (sola occorrenza dentro detect 0)
- La prima 41 0f 44 c5 diventa 31 c0 90 90 (cmove %r13d,%eax diventa [xor eax, eax & nop & nop]) si trova dentro detect_1
- La seconda 41 0f 44 c5 diventa 0f 44 c0 90 (cmove %r13d,%eax diventa [move %eax,%eax & nop]) si trova sempre dentro detect_1
- I vari 01 c0 diventano 31 c0 (add eax, eax diventa xor eax, eax) (tre occorrenze in detect_1 e in detect_5)
- b8 02 00 00 00 diventa b8 00 00 00 00 (mov 0x2, eax diventa mov 0x0, eax) (una sola sequenza, dentro *detect_2*)
- 44 89 e8 diventa 31 c0 90 (mov r13d,eax diventa [xor eax, eax & nop], dentro *detect_3*
- ba 02 00 00 00 diventa ba 00 00 00 00 (mov 0x2,edx diventa mov 0x0, edx) dentro *detect_4*

Un totale di 13 cambiamenti (23 patch totali). Si nota sia eseguendo il file con/senza debugger che funziona passando tutti i controlli se fatto esattamente in questa maniera. Altrimenti, riprendere l'eseguibile sano e riprovare.

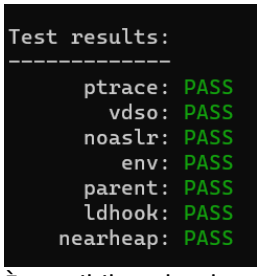
All'interno di questi branch, sussiste un controllo *jz*; ciò presuppone che andiamo nel ramo SUCCESS quando notiamo che la funzione ritorna 0. Un'idea potrebbe essere di applicare una patch manualmente a ciascuna funzione e fare in modo ritorni 0.

Tuttavia, è sufficiente applicare una patch al programma una volta, modificando il valore restituito della funzione di chiamata del test. In altre parole, dopo la chiamata *ptr [rbx]*, *eax* deve essere 0. In questo modo supereremmo ogni prova. Sfortunatamente, non abbiamo spazio per impostare *eax* a 0 prima di *cmp eax, 1*. Inoltre, non possiamo semplicemente rimuovere la chiamata *ptr[rbx]*, poiché la chiamata precedente *_printf_chk_ptr* imposta *eax* su un valore solitamente diverso da 0.
Invece, possiamo sostituire la chiamata ptr [rbx] con un'istruzione che imposta eax su 0.
Guardando la Hex View, l'istruzione di chiamata utilizza due byte (*FF 13*). Come possiamo impostare *eax* su 0 con due byte? Un modo semplice è copiare l'istruzione sopra *xor eax,eax* che imposta *eax* a 0 con soli due byte (31 C0).

Per questo, è sufficiente eseguire la patch all'interno del main.
Si va nella Hex View, si clicca "Search" e poi "Sequence of Bytes" e poi si inserisce FF 13. Fatto questo, si va a sostituire FF 13 con 31 C0 (l'istruzione xor eax, eax) e salvare. Sono quelli evidenziati:



Testando i risultati, tutti i controlli passano; si nota che questo funziona in quanto già tutti gli altri controlli ad eccezione di *ldhook* passano, quindi può essere necessaria una modifica sola.




È possibile volendo operare un confronto tra file binario normale e patch eseguita (eventualmente, anche per confrontarlo con la soluzione):
https://www.howtogeek.com/817201/compare-binary-files-linux/