GDB használata

Fordítás

Terminálból a `gcc pelda.c -o pelda -wall -m32 -g` utasítással.

A -g a debug ("hibakövetési") kapcsoló. Debug információk kerülnek a lefordított binárisba, így könnyebb lesz a debuggolás. A példák kedvéért a -m32 kapcsolóval fordítsunk 32 bites kódot.

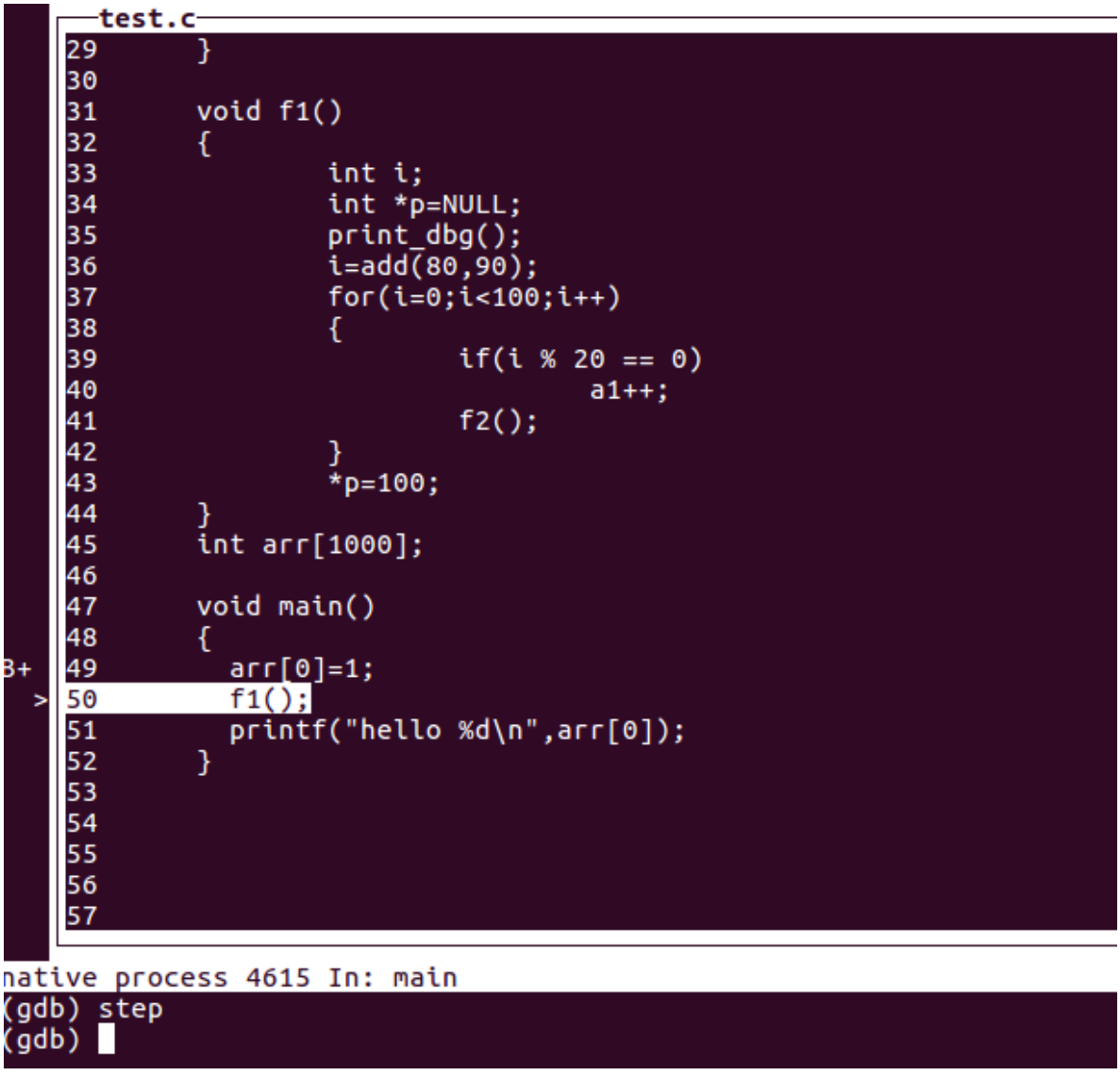
Használat

1. **GDB elindítása és a cél program betöltése:** `gdb ./pelda`

Ha nem -g kapcsolóval volt a program fordítva, akkor a betöltéskor kiírt logok között azt látjuk, hogy `no debugging symbols found` (amitől a debuggolás még elindítható, csak kevésbé lesz informatívabb)

Elindítás **Text User Interface**-el együtt (TUI): `gdb ./pelda -tui`

Kettéosztott terminál ablak segítségével **a forráskódot is látni fogjuk** futás közben!



2. **A program gépi kódjának megnézése**

- `disassemble`: az aktuális függvény gépi kódja. Kis nyíl jelöli éppen melyik utasításnál lévő breakpointban vagyunk.
- `disassemble function`: A kért függvény gépi kódjának megnézése
- `disas`: rövidített írásmód
- `disas /m funcname`: a C utasítások és a hozzájuk tartozó gépi utasítások együtt nézhetőek

3. **Program összes függvényének listázása**

   `info functions [regexp]`

   Opcionálisan megadható egy reguláris kifejezés a találatok szűrésére

4. **Breakpoint elhelyezése**
   - `break function`: breakpoint elhelyezése függvénynél
   - `b function`: rövidített írásmód
   - `b linenum`: adott sorhoz (`b filename:linenum` több fájl esetén)
   - Adott sor-pozícióhoz képesti eltolással: `b +/-OFFSET`
   - `b *addr`: ahol addr az utasításnak pl. a disassemble segítségével megtudott címe.

     Pl. `break *0x0000000000400722`

5. **Breakpointok listázása**

   `info breakpoints`

6. **Breakpoint ki/be kapcsolása illetve törlése**
   - `disable N`: adott sorszámú breakpoint kikapcsolása (`dis N`)
   - `enable N`: adott sorszámú breakpoint visszakapcsolása (`en N`)
   - `clear function, clear linenum, clear filename:linenum` breakpoint törléshez
   - `delete`: az összes breakpoint letörlése

7. **A betöltött célprogram elindítása**

   `run (r)`

   Használható arra is, hogy az aktuális debug sessiont újraindítsuk.

8. **Következő sor végrehajtása**
   - `step (s)`
   - `next (n)`: a különbség, hogy nem lép bele függvényhívásokba
   - `s N` vagy `n N`: egyszerre N darab sort lépünk
   - `finish`: az aktuális függvényből való visszatérés helyére ugrik. Ha volt visszatérési érték, akkor kiírja.

9. **Breakpoint utáni újraindítás**
   - `continue (c)`
   - `c N`: ahol N megadja, hányszor ne álljon meg a gdb az aktuális breakpointnál

10. **Backtrace**

    Breakpointnál vagy hiba (pl segmentation fault) esetén megnézhetjük hogyan jutott el a program az adott (hibás) sor lefutásához.
    - `backtrace (bt)`: az aktuális stack frame-től kezdve visszafelé haladva kiírja az összes frame-et, ami a stack-en van. Ha túl sokáig tart CTRL+C-vel le lehet állítani a kiíratást.
    - `bt n`: csak az utolsó n darab frame-t írja ki
    - `bt -n`: csak az első n darab frame-t írja ki
    - `bt full`: a frame-k lokális változóit is kiíratja. Kombinálható az előző két utasítással.

11. **Információk kiíratása**

- `info registers`

```
Breakpoint 1, f2 () at test.c:11
11              printf("X");
(gdb) info registers
eax            0xb      11
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xffffcae0       0xffffcae0
ebp            0xffffcae8       0xffffcae8
esi            0xf7faa000       -134569984
edi            0xf7faa000       -134569984
eip            0x80484a1        0x80484a1 <f2+6>
eflags         0x282    [ SF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
(gdb) disas f2
Dump of assembler code for function f2:
   0x0804849b <+0>:     push   %ebp
   0x0804849c <+1>:     mov    %esp,%ebp
   0x0804849e <+3>:     sub    $0x8,%esp
=> 0x080484a1 <+6>:     sub    $0xc,%esp
   0x080484a4 <+9>:     push   $0x58
   0x080484a6 <+11>:    call   0x8048380 <putchar@plt>
   0x080484ab <+16>:    add    $0x10,%esp
   0x080484ae <+19>:    nop
   0x080484af <+20>:    leave
   0x080484b0 <+21>:    ret
End of assembler dump.
```

Első oszlop: a regiszter neve, második oszlop: tartalom hexadecimális formában, harmadik oszlop: tartalom olvasható formában. Nézzük meg az EIP regiszter tartalmát.

- `layout regs`
- `info variables`: globális és statikus változók értékei
- `info locals`: lokális változók értékei
- `print symbol`: adott szimbólum értéke
  - röviden: `p symbol`
  - `p valtozonev`
  - `p fuggveny::valtozonev`
  - `p $eax`
- `info address symbol`: adott szimbólum memóriacíme
- `info frame`: az aktuális frame adatai (argumentumok, lokális változók, ebp, eip értékei)
- `x 0x080485e9`: az x utasítással egy konkrét memóriacím tartalmát nézhetjük meg
  - `x/16 address`: 16 érték legyen megjelenítve az adresstől kezdve (növekvő memóriacímek irányába)
  - `x/-16 address`: ugyanez, csak a 16 érték csökkenő memóriacímek irányába megy (gdb 8-tól működik)
  - Kiíratási módosítók is lehetségesek
    - `x/16x address`: 16 hexadecimális érték kiíratása

        - `x/16c address`: 16 karakter kiíratása
        - `x/s address`: C sztringként értelmezi az adott memóriaterületet
        - `x/d address`: előjeles, decimális értékként való megjelenítés
        - `x/i address`: utasításként próbálja értelmezni a memóriaterületet
    - Verem tetjének megtekintése: `x/20x $esp`

12. **Változáskövetés**
    - `watch symbol`: változó, regiszter, memóriacím (vagy komplexebb kifejezés) értékének a váltazása esetén álljon meg a program futása (mint egy breakpoint érték változása alapján). Többszálú program esetén minden szálon van változásfigyelés.
    - `info watchpoints`: listázásuk
13. **Feltételes breakpoint**

    `break file.c:20 if i == 112`

    `break file.c:17 if strcmp(input, "password") == 0`

    Vagyis használhatjuk a célnyelv utasításait, szimbólumait, hogy a breakpointokat feltételekhez kössük!

    Meglévő breakpointhoz is tudunk az azonosító száma alapján utólag feltételt rakni (pl. info breakpoints-al kideríthető)

    `cond 8 *p == 78`: új feltétel a 8-as breakpointhoz

    `cond 8`: a 8-as breakpoint feltételének törlése

14. **Függvényhívás**

    Lehetőségünk van kézzel meghívni a célprogram valamely függvényét (pl. ha direkt írtunk egyet debug céllal) Paramétereket is adhatunk meg neki.

    `call segedfgv()`

15. **.gdbinit használata**

    GDB konfigurációs fájl, mely a gdb indításakor automatikusan betöltődik. Segítségével lehet kicsit automatizálni a debuggolást. Helyei lehetnek:

    - /etc/gdbinit: rendszerszintű beállítások
    - ~/.gdbinit: user szintű beállítások
    - ./.gdbinit: csak az adott munkakönyvtárból futó gdb-re vonatkozik. A lokális gdgbinit betöltése ki lehet kapcsolva, ekkor a megjelenő hibaüzenetek alapján lehet megoldást keresni (hozzáadni a biztonsági kivételt vagy teljesen kikapcsolni ezt az ellenőrzést)

    Egy példafájl tartalma:

```
set breakpoint pending on # dialógus kiiktatása

file test          # megegyezik ezzel: "gdb ./test"
set args "input" # parancssori argumentumok átadása (ha kell)

break test_function # breakpoint beállítása

run       # futtatás
```

    Ha elkészítettük ezt a fájlt már csak el kell indítani a gdb-t (és nem is kell most neki megadni kézzel a célprogramot).

    A gdbinit fájlban saját segédfüggvényeket is létrehozhatunk. Például:

```
define prnt
  x/20x $eax
  bt 2
  echo "hello"
end

break test_function
commands # másképp: commands N, ahol N a breakpoint száma
prnt #a breakpoint elérésekor végrehajtandó utasítások felsorolása
end #utasítás-lista vége
run
```

Ha így adjuk meg a gdbinit fájlunkat a breakpoint elérésekor automatikusan lefut majd a prnt tartalma. De kézzel is meghívhatjuk.