PB173 – Binární programování Linux

VIII. Ladění

Jiri Slaby

ITI, Fakulta informatiky

12. 11. 2013

Ladění

1. Funkčnosti (dnes)
   - Pády
   - Nekorektní chování
2. Výkonnosti/velikosti (příště)
   - Optimalizace kódu
   - Odstranění nepoužitého kódu

Techniky ladění

1. Dynamicky
   - Spouštím na CPU a očekávám výsledek
   - Testování
2. Staticky
   - Spouštím analyzátor a ten hledá nesrovnalosti
   - Analýza ukazatelů, hodnot, velikosti zásobníku, . . .

Část I

Ladění funkčnosti

Ladění funkčnosti

- Výpisy
  - `printf` a jiné
- Monitor chování
  - `gdb`, `ltrace`, `strace`
- Post-mortem
  - `core` soubor (obraz paměti)

Debugger `gdb`

- Pracuje s ELFem a DWARFem
- Používá systémové volání `ptrace`
  - Sleduje proces
  - Upravuje vykonávání
  - Čte a mění paměť
  - ...
- Podpora Python skriptů
- Spuštění: `gdb --args binarka --volby`
- Nápověda: `help`
- Manuál: GDB User Manual

Příkazy gdb

- Operace s během
  - Spustit: `run`
  - Pokračovat: `continue`
  - Krokovat s vnořením: `step` (po řádcích), `stepi` (po instrukcích)
  - Krokovat bez vnoření: `next` (po řádcích), `nexti` (po instrukcích)
- Breakpointy: `break`
- Výpisy
  - Programu: `list` (jen s ladicími informacemi)
  - Assembleru: `disassemble`
  - Paměti: `print`
  - HexDump: `x`
  - Ostatní: `info registers`, `info break`, `...`
- Zásobník volání: `where`, `up`, `down`
- Konec: `quit`
- Opakování předchozího příkazu: ENTER

Úkol

Práce s `gdb`

1. Přeložte si `pb173-bin/08/`
2. Spusťte `debug` (mělo by dojít k pádu)
3. Otevřete `debug` v `gdb`
4. Spusťte (`run`)
5. Prohlédněte si, kde došlo k pádu (vypište):
   - Zásobník volání (`where`)
   - Místo pádu (`list` s parametrem `soubor:radek`)
   - Disassembly (`disassemble` s parametrem `adresa`)
   - Proměnnou `b` (`print b`)
   - Proměnnou `argc` v `main` (`up` a `print argc`)
6. Přidejte breakpoint na začátek `main` a spusťte (`run`)
7. Krokujte až k pádu (`next`)
8. Zopakujte pro `step` a také `stepi`

Obrazy paměti a gdb

- Jádro při pádu zapíše obraz paměti („coredump”)
  - Limit velikosti: `ulimit -c` (bývá 0)
  - Cíl: `sysctl kernel.core_pattern`
- Potom se dá analyzovat v gdb
  - `gdb --core=core_soubor --args binarka --volby`

Úkol

Práce s obrazem paměti

1. Nastavte limit na obrazy na `unlimited` (`ulimit -c`)
2. Podívejte se, kam se obraz uloží (`sysctl kernel.core_pattern`)
3. Spusťte program z předchozího příkladu
4. Otevřete v `gdb` s obrazem paměti (`gdb ./debug core`)
5. Vypište si podobné informace jako předtím
   - `where`
   - `print b`
   - A podobně

Tracery

- Sledují události a vypisují je
- `ltrace`: sleduje knihovní volání
- `strace`: sleduje systémová volání
- Používají systémové volání `ptrace`
  - `ltrace` si nastaví breakpointy na všechny externí symboly z ELFu
  - `strace` jednoduše použije `PTRACE_SYSCALL`
- Spuštění: `s/ltrace binarka --volby`
- Další parametry
  - `-e`: filtr (`-e write`)
  - `-f`: sleduje i potomky
  - `-o`: výstup do souboru

Úkol

Práce s tracery

1. Projděte si `pb173-bin/08/tracer.c`
2. Spusťte nejdříve s `ltrace`
   - Kolik je volání `fwrite` do `libc`?
3. Spusťte s `strace`
   - Kolik je volání `write` do jádra?
   - $\Rightarrow$ `libc` bufferuje
4. Pomocí `ltrace` zjistěte, jak velký buffer `libc` používá
   - V cyklu vypisujte znak
   - Použijte volbu `-S`
5. Pomocí `strace` ověřte totéž
   - Třetí parametr `write`

Valgrind

- Sleduje (zejména) operace s pamětí (*Memcheck*)
  - Úniky paměti
  - Neinicializované a nezarovnané přístupy
  - Použití paměti po `free`
  - ...
- Několik dalších modulů
  - Callgrind: profiluje cache CPU
  - Massif: profiluje použití haldy
  - Další pro problémy se zámky atd.
- Spuštění: `valgrind binarka --volby`
- Další parametry
  - `--tool`: výběr nástroje shora
  - `--leak-check`: sledovat úniky paměti

Úkol

Práce s nástrojem `valgrind`

1. Projděte si `pb173-bin/08/memcheck.c`
2. Spusťte v nástroji `valgrind`
3. Řiďte se pokyny na konci výpisu
   - Vypište si všechny podrobnosti