Mieszko Cichowicz

Spis Treści

1.Wstęp
2.Najprostsze metody odpluskwiania
a) printf, fprintf
b) assert
3.Jak działa odpluskwiacz?
4.Odpluskwianie w gdb.
a)ogólnie
b)pliki core
5.Odpluskwianie w Windows

1. Wstęp

Odpluskwianie (ang. debugging) to proces mający na celu usunięcie błędów z programu. Gdy nasz program zawiera jakiś błąd, to najprawdopodobniej w którymś miejscu kodu warunek, o którym myślimy żę jest prawdziwy jest w rzeczywistości fałszywy. Przykładowe błędy:

- w danym miejscu kodu, pewna zmienna ma inną wartość niż nam się wydaje,
- w pewnym punkcie jakaś struktura nie została poprawnie zainicjalizowana,
- w jakiejś instrukcji `if-then-else` wykonuje się część ”`if`” a nie ”`else`” jak się spodziewaliśmy,
- podprocedura otrzymuje parametry inne od zamierzonych.

Znajdowanie błędów polega na metodycznym potwierdzaniu swoich myśli/wierzeń na temat stanu programu w poszczególnych momentach, aż do odnalezienia miejsca, w którym nasze myśli o stanie programu są niezgodne z rzeczywistością.

2. Najprostsze metody odpluskwiania:

a) Wypisywanie komunikatów (printf, fprintf)

Zapewne każdy z nas wielokrotnie stosował tę metodę odpluskwiania. Polega ona na wzbogacaniu kodu o dodatkowe linie wypisujące jakieś komunikaty – np. wartości pewnych zmiennych w danym momencie programu, bądź konstrukcje typu:

```
if (...) {
  printf(“IF\n”);
  /*...*/
} else {
  printf(“ELSE\n”);
  /*...*/
}
```

b) Funkcja assert

Funkcja ta pozwala na sprawdzenie warunku logicznego i zatrzymanie wykonania programu, w przypadku gdy warunek jest nieprawdziwy. Stosowanie tej funkcji wymaga załączenie w pliku źródłowym pliku nagłówkowego assert.h. Przykład kodu korzystającego z funkcji assert:

```
#include <stdio.h>
#include <assert.h>

void drukuj(int x)
{
  assert(x<100);
  printf("x = %d\n", x);
}

int main()
{
  drukuj(5);
  drukuj(103);
  return 0;
}
```

oraz efekt wykonania powyższego kodu:

```
[mieszko@localhost test]$ make
gcc -o a test.c -g
[mieszko@localhost test]$ ./a
x = 5
a: test.c:6: drukuj: Assertion `x<100' failed.
Aborted
[mieszko@localhost test]$
```

Powyższe metody są bardzo prymitywne i wymagają modyfikowania kodu w celu testowania. Ponadto po zakończeniu procesu odpluskwiania, wszelkie instrukcje dodane na potrzeby

odpluskwiania należy usunąć, co jest mało wygodne i czasochłonne. By ułatwić zadanie programistom stworzone zostały rozmaite narzędzia do odpluskwiania. Pozwalają one na wykonywanie programów krok po kroku, wypisywanie i modyfikacje zmiennych podczas działania programu i wiele innych rzeczy.

3.Jak działa odpluskwiacz?

Odpluskwiacze korzystają z mechanizmu śledzenia procesów. Najważniejszą funkcją związaną ze śledzeniem procesów jest niewątpliwie ptrace.

```
long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

W stukturze task_struct znajduje się pole ptrace, które opisuje sytuację procesu (czy jest on śledzony i jeśli tak, to w jaki sposób) będące maską bitową zbioru flag podanego w linux/sched.h, przykładowe z nich, to:

- PF_PTRACED – proces jest śledzony,
- PF_TRACESYS – proces zostanie zatrzymany przy wywołaniu/wyjściu z funkcji systemowej,
- PF_DUMPCORE – proces dokonał zrzutu do pliku core.

Są dwie metody pozwalające na rozpoczęcie śledzenia procesu:

1.Każdy proces może zgłosić chęć bycia śledzonym. Służy do tego wywołanie funkcji ptrace z parametrem PTRACE_TRACEME (pozostałe argumenty są ignorowane):

```
ptrace(PTRACE_TRACEME, 0, 0, 0);
```

Proces śledzący powinien być przygotowany na takie wywołanie - jeśli nie wznowi działania programu śledzonego, to po pierwszym przerwaniu działania zostanie on zawieszony na zawsze. Wszystkie sygnały adresowane do procesu śledzonego (z wyjątkiem SIGKILL) trafią najpierw do procesu śledzącego (jako wynik funkcji wait), który będzie mógł je zmodyfikować, albo nawet zignorować.

2.Druga metoda to podłączenie się do działającego procesu o zadanym numerze pid za pomocą argumentu PTRACE_ATTACH funkcji ptrace (ostatnie 2 argumenty są ignorowane).

```
ptrace(PTRACE_ATTACH, pid, 0, 0);
```

Inne rodzaje żądań funkcji ptrace:

- PTRACE_PEEKTEXT i PTRACE_PEEKDATA (pod Linuksem – równoważne) czytają słowo z pamięci procesu śledzonego (o podanym pidzie) z podanego adresu. Przykładowe wywołanie:

```
slowo = ptrace(PTRACE_PEEKTEXT, pid, (void *)0x080483f0, 0);
```

przypisze na zmienną slowo, słowo z pamięci pod adresem 0x080483f0.

- PTRACE_POKETEXT i PTRACE_POKEDATA (pod Linuksem – równoważne) zapisują słowo do pamięci procesu śledzonego (o podanym pidzie) na podanego adresu. Przykładowe wywołanie:

```
ptrace(PTRACE_POKETEXT, pid, (void *)0x080483f0, slowo);
```

zapisze wartość zmiennej slowo, do pamięci pod adresem 0x080483f0.

- PTRACE_SETREGS i PTRACE_GETREGS ustawia/sprawdza zawartość rejestrów. Liczba rejestrów i ich nazwy zależą od procesora. W pliku /usr/include/asm/ptrace.h można znaleźć informację, że interesujących rejestrów jest 17, można też znaleźć ich numery. Wobec tego można napisać:

```
int rejestry[17];
/*...*/
ptrace(PTRACE_GETREGS, pid, 0, tablica);
/* lub odpowiednio: ptrace(PTRACE_SETREGS, pid, NULL, tablica);*/
```

- PTRACE_CONT uruchomi proces, aż do otrzymania przez niego jakiegoś sygnału albo zwykłego zakończenia działania. Przykładowe wywołanie:

```
ptrace(PTRACE_CONT, pid, 0, 0);
```

- PTRACE_SYSCALL wznowi proces aż do próby wywołania przez niego funkcji systemowej (syscall); samego wywołania nie można, oczywiście śledzić, ale kolejne wywołanie

PTRACE_SYSCALL zatrzyma proces po powrocie z tego wywołania. Pozwala to na zbadanie argumentów z jakimi wywoływane są funkcje systemowe, oraz zbadanie sytuacji w jakiej znajdzie się proces śledzony, po zakończeniu wywołania funkcji systemowej. Przykładowe wywołanie:

```
ptrace(PTRACE_SYSCALL, pid, 0, 0);
```

- PTRACE_SINGLESTEP - zatrzyma proces po wykonaniu kolejnego rozkazu, tu też oczywiście, nie można śledzić wywołań systemu. Przykładowe wywołanie:

```
ptrace(PTRACE_SINGLESTEP, pid, 0, 0);
```

- PTRACE_DETACH - zakończy śledzenie procesu - dalej będzie się on wykonywał bez nadzoru. Przykładowe wywołanie:

```
ptrace(PTRACE_DETACH, pid, 0, 0);
```

Schemat wykorzystania mechanizmu śledzenia procesów – prosty przykład:
1. treść procesu śledzonego child.c:

```
#include <stdio.h>

int main()
{
  /*...*/
  printf("\tTHE END\n");
  return 0;
}
```

2. treść procesu śledzącego parent.c:

```
#include <stdio.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
  int rejestry[17];
  int pid;
  int i;

  if ((pid = fork()) == 0) { /* child process */
    ptrace(PTRACE_TRACEME,0,0,0);
    execl("child",0);
    return -1;
  }
  /* parent process */
  wait(0);
  ptrace(PTRACE_GETREGS, pid, 0, (void *)rejestry);
  for (i = 0; i < 17; i++)
    printf("%d:\t%d\n", i, rejestry[i]);
  /*...*/
  ptrace(PTRACE_CONT,pid,0,0);
  return 0;
}
```

Wynik działania poniższego przykładu:

```
[mieszko@localhost test]$ ./parent
0:      0
1:      0
2:      0
3:      0
4:      0
5:      0
6:      0
7:      43
8:      43
9:      0
10:     0
11:     11
12:     1073745008
13:     35
14:     2097686
15:     -1073750528
16:     43
        THE END
```

Dwa słowa o ltrace i strace

Pod Linuxem możemy korzystać z dwóch przydatnych do odpluskwiania programów: strace i ltrace. Oba korzystają z funkcji ptrace. Dzięki ltrace możemy obejrzeć wszystkie wywołania funkcji bibliotecznych naszego programu, natomiast strace pozwala obejrzeć wywołania funkcji systemowych. Przykład działania ltrace i strace dla programu o źródle:

```
#include <stdio.h>

void drukuj(int x)
{
  printf("x = %d\n", x);
}

int main()
{
  int i;
  int *w = 0;

  i = *w;
  drukuj(5);
  drukuj(103);
  return i;
}
```

Wyjście 1.wykonania powyższego programu:

```
[mieszko@localhost test]$ ./a
x = 5
x = 103
Segmentation fault
```

2. wywołania na nim programu ltrace:

```
[mieszko@localhost test]$ ltrace ./a
__libc_start_main(0x08048377, 1, 0xbfffe6e4, 0x080483c0, 0x08048410
<unfinished ...>
printf("x = %d\n", 5x = 5
)                                                       = 6
printf("x = %d\n", 103x = 103
)                                                   = 8
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

3. wywołania na nim programu strace:

```
[mieszko@localhost test]$ strace ./a
execve("./a", ["./a"], [/* 58 vars */]) = 0
uname({sys="Linux", node="localhost", ...}) = 0
access("/proc/sys/kernel/tls", F_OK)    = 0
brk(0)                                  = 0x80495c8
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40017000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=62955, ...}) = 0
old_mmap(NULL, 62955, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40018000
close(3)                                = 0
open("/lib/tls/libc.so.6", O_RDONLY)    = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0@Z\1\000"..., 512) =
512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1334740, ...}) = 0
old_mmap(NULL, 1340908, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40028000
old_mmap(0x4016a000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3,
0x142000) = 0x4016a000
old_mmap(0x4016d000, 9708, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|
MAP_ANONYMOUS, -1, 0) = 0x4016d000
close(3)                                = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40170000
set_thread_area({entry_number:-1 -> 6, base_addr:0x401702a0, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1,
seg_not_present:0, useable:1}) = 0
munmap(0x40018000, 62955)               = 0
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 0), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40018000
write(1, "x = 5\n", 6x = 5
)                  = 6
write(1, "x = 103\n", 8x = 103
)                = 8
--- SIGSEGV (Segmentation fault) @ 0 (0) ---
+++ killed by SIGSEGV +++
```

Widać, że nawet dla tak prostego programu jaki podany jest w przykładzie, wyjście strace jest dość długie i zniechęcające, może nam jednak niejednokrotnie ułatwić znalezienie błedu.

Odpluskwianie w gdb

GNU Debugger (gdb) to najpopularniejszy debugger linuxowy. Najważniejsze komendy gdb – większość z nich ma jednoliterowe skróty, które będę pisał w nawiasach okrągłych:

- help – wypisuje informacje o podstawowych komendach gdb,
- help nazwa – wypisuje informacje na temat komendy/grupy komend o podanej nazwie,
- run [lista_argumentow] – rozpoczęcie wykonania programu, można również podać parametry wywołania programu,
- next(n) – wykonuje następną linię kodu programu, nie rozwijając wywołań funkcji,
- step(s) – wykonuje następną linię kodu programu, rozwijając wywołania funkji,
- continue(c) – wznawia wykonanie programu (w normalnym trybie, a nie krok po kroku),
- backtrace(bt) – wyświetla stos wywołań programu,
- breakpoin(b) nazwa – ustawia breakpoint na początku funkcji o podanej nazwie,
- return [wyrazenie] – kończy wywołanie aktualnej funkcji, ewentualnie zwracając wartość podanego wyrażenia,
- set zmienna=wartość – wykonuje podane przypisanie,
- print zmienna=wartość – wykonuje przypisanie oraz wypisuje nazwę zmiennej oraz jej wartość po przypisaniu,
- print zmienna – wypisuje wartość podanej zmiennej,

- quit(q) – zakończenie pracy z gdb.

Załóżmy, że chcemy odpluskwiać program “a” o następującym kodzie źródłowym:

```
#include <stdio.h>
#include <assert.h>

void drukuj(int x){
  assert(x<100);
  printf("x = %d\n", x);
}
int main()
{
  drukuj(5);
  drukuj(103);
  return 0;
}
```

Aby móc odpluskwiać dany program przy użyciu gdb, musimy go skompilować z parametrem -g, aby kompilator dołączył do pliku wykonywalnego dodatkowe informacje o symbolach występujących w programie:

```
[mieszko@localhost test]$ make
gcc -o a test.c -g
```

Po skompilowaniu programu uruchamiamy gdb i zaczynamy odpluskwianie programu “a”:

```
[mieszko@localhost test]$ gdb a
GNU gdb 6.0-2mdk (Mandrake Linux)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

(gdb)
```

Ustawiamy breakpoint na początku funkcji main:

```
(gdb) b main
Breakpoint 1 at 0x80483e3: file test.c, line 12.
```

a następnie uruchamiamy program:

```
(gdb) r
Starting program: /home/mieszko/test/a

Breakpoint 1, main () at test.c:12
12        drukuj(5);
```

Wykonanie programu zatrzymało się na pierwszym punkcie kontrolnym (ang. breakpoint), a następną linią do wykonania jest linia 12 kodu, której zawartość to drukuj(5);
możemy teraz np. obejrzeć stos wywołań programu:

```
(gdb) bt
#0  main () at test.c:12
```

Jak można się było spodziewać, niewiele jest na stosie, jako, że dopiero co wywołaliśmy funkcję main. Wykonajmy następną linię kodu, nie rozwijając wywołania funkcji:

```
(gdb) n
x = 5
13        drukuj(103);
```

Program wypisał na standardowe wyjście napis “x = 5” oraz zatrzymał się na następnej linii kodu. Wykonajmy następną linię kodu, tym razem rozwijając wywołanie funkcji:

```
(gdb) s
drukuj (x=103) at test.c:6
6         assert(x<100);
```

GDB pokazuje nam, że wywołał funkcję drukuj z parametrem x, którego wartość to 103 oraz, że następną instrukcją do wykonania jest 6 linia kodu, będąca pierwszą linią funkcji drukuj. Podejrzyjmy stos wywołań programu raz jeszcze – tym razem na wierzchu stosu powinno być wywołanie funkcji drukuj:

```
(gdb) bt
#0  drukuj (x=103) at test.c:6
#1  0x080483fa in main () at test.c:13
```

Ponieważ wykonanie kolejnej instrukcji spowodowało by błąd i zakończenie wykonywania programu (warunek w assert jest nie spełniony), zmieńmy wartość zmiennej x:

```
(gdb) set x=97
```

oraz wykonajmy kolejną instrukcję programu:

```
(gdb) set x=97 (gdb) s
7         printf("x = %d\n", x);
```

Jak widać program przeszedł przez wywołanie assert nie powodując błędu. Kontynuujmy wykonanie programu w normalnym tępie:

```
(gdb) c
Continuing.
x = 97

Program exited normally.
```

Teraz możemy zakończyć działanie gdb:

```
(gdb) q
[mieszko@localhost test]$
```

Uruchamianie gdb na plikach core

Zamiast uruchamiania programu pod gdb, przechodzenia go krok po kroku w celu znalezienie błędów możemy odpluskwiać program, którego wykonanie sie zakończyło. Gdy program kończy się z błędem tworzony jest plik core, zawierający wszystkie ważne informacje o procesie z chwili wystąpienia błedu – takie jak zawartość stosu wywołań, wartości zmiennych, rejestrów, itp.. Aby uruchomić gdb w tym trybie, należy podać nie tylko ścieżkę programu, ale także ścieżkę do powstałego pliku core:

```
gdb nazwaprogramu nazwaplikucore
```

Przykładowy program:

```
#include <stdio.h>

void drukuj(int *x)
{
  printf("x = %d\n", *x);
}

int main()
{
  int *i = 0;
  int j = 5;

  drukuj(&j);
  drukuj(i);
  return 0;
}
```

Po skompilowaniu powyższego programu (z opcja -g oczywiście) i uruchomieniu go zobaczymy:

```
[mieszko@localhost test]$ ./a
x = 5
Segmentation fault (core dumped)
```

Program zakończył się błędem ochrony pamięci i utworzył plik typu core. Jeśli plik core się nie tworzy (jest to zachowanie domyślne w większości przypadków) musimy powiedzieć systemowi by

je tworzył, w bashu możemy to zrobić za pomocą polecenia ulimit -c n, gdzie n jest liczbą będącą nowym maksimum wielkości tworzonych plików typu core. Spróbujmy odpluskwić nasz program korzystając z gdb w wersji 'post mortem' ;)

```
[mieszko@localhost test]$ gdb a core.3179
GNU gdb 6.0-2mdk (Mandrake Linux)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i586-mandrake-linux-gnu"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".

Core was generated by `./a'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x08048368 in drukuj (x=0x0) at test.c:6
6         printf("x = %d\n", *x);
(gdb) print x
$1 = (int *) 0x0
(gdb) bt
#0  0x08048368 in drukuj (x=0x0) at test.c:6
#1  0x080483b1 in main () at test.c:15
(gdb) q
[mieszko@localhost test]$
```

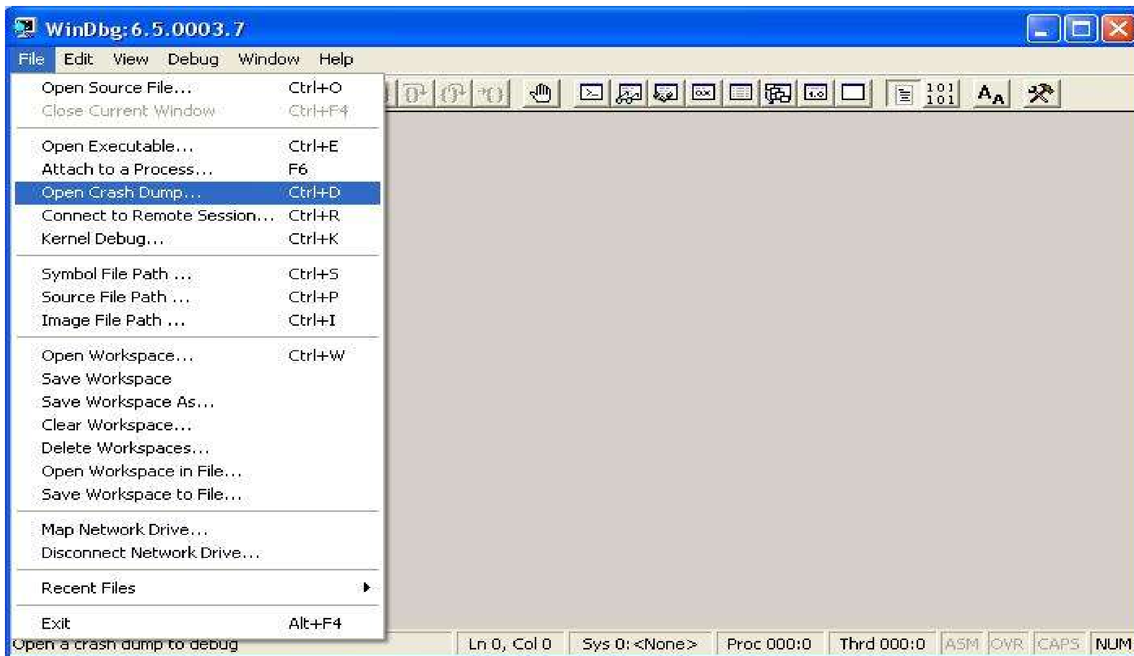
Udało nam się odczytać z informacji w pliku core, że program zakończył się podczas wykonywania 6 linii programu, będąc w funkcji drukuj, wywołanej ze wskaźnikiem zerowym, podczas próby wykonania funkcji printf. Możemy już spokojnie usunąć nasz błąd.

Odpluskwianie w Windows

Microsoft Debugging Tools for Windows składa się z następujących debuggerów:

- KD (*kd.exe)* – odpluskwiacz do programów działających w trybie jądra, o konsolowym interfejsie,
- CDB (*cdb.exe*) – odpluskwiacz do programów działających w trybie użytkownika, o konsolowym interfejsie,
- NTSD (*ntsd.exe*) – jest to odpluskwiacz niemal identyczny jak CDB
- WinDbg (*windbg.exe*) – odpluskwiacz będący połączeniem powyższych, charakteryzujący się dodatkowo graficznym interfejsem.

Chcąc rozpocząć debuggowanie, musimy podpiąć jakiś proces do windbg, badź wczytać plik core:

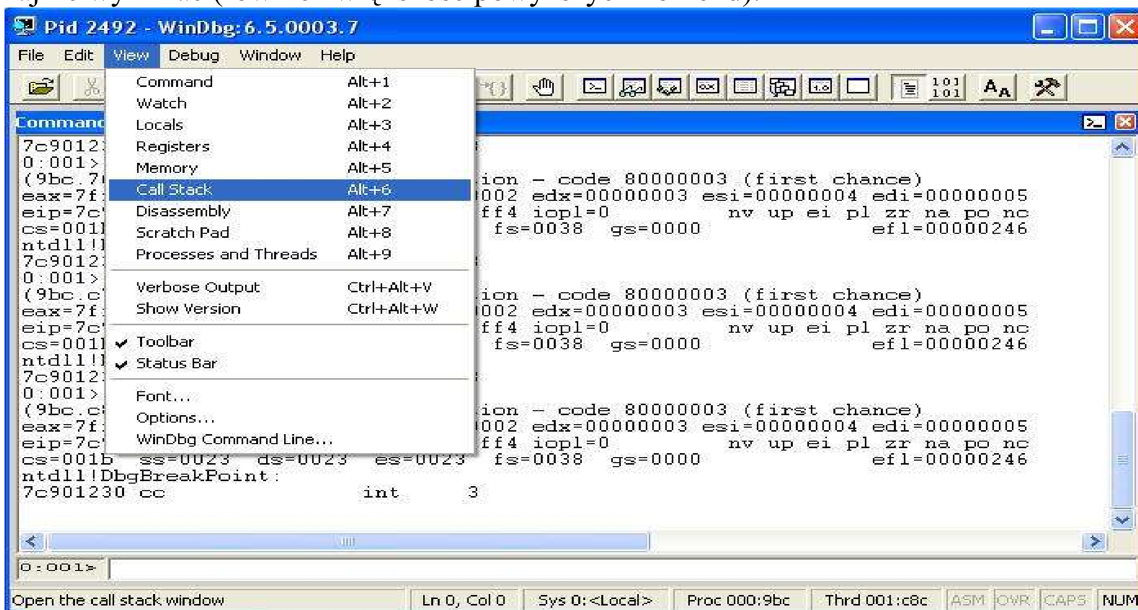


Ważniejsze komendy windbg:
- bl (Breakpoint List) – wyświetla informacje o punktach kontrolnych,
- bc (Breakpoint Clear) – usuwa wskazany punkt kontrolny,
- bp, bu, bm (Set Breakpoint) – tworzy nowy punkt kontrolny,
- bd (Breakpoint Disable) – dezaktywacja podanego breakpointu,
- be (Breakpoint Enable) – ponowna aktywacja, wcześniej wyłączonego punktu kontrolnego,
- g (Go) – kontynuacja wykonania programu, aż do napotkania punktu kontrolnego bądź sygnału,
- r (Registers) – służy do wyświetlania bądź modyfikowania zawartości rejestrów,
- j (Execute If – Else) – warunkowe wykonanie jednej z 2 podanych komend, np.:

```
j (zmienna=0) 'r eax'; 'r ebx; r ecx'
```

spowoduje wyświetlenie zawartości rejestru eax o ile zmienna ma wartość zero, lub wyświetlenie zawartości rejestrów ebx i ecx w przeciwnym przypadku,
- t (Trace) – wykonuje jedną instrukcję kodu rozwijając wywołania funkcji,
- p (Step) – wykonuje jedną instrukcję kodu nie rozwijając wywołań fukcji.

Jako, że windbg jest narzędziem o interfejsie graficznym wiele interesujących nas rzeczy można zwyczajnie wyklikać (również większość powyższych komend).



Możemy przykładowo obejrzeć sobie zawartość rejestrów: