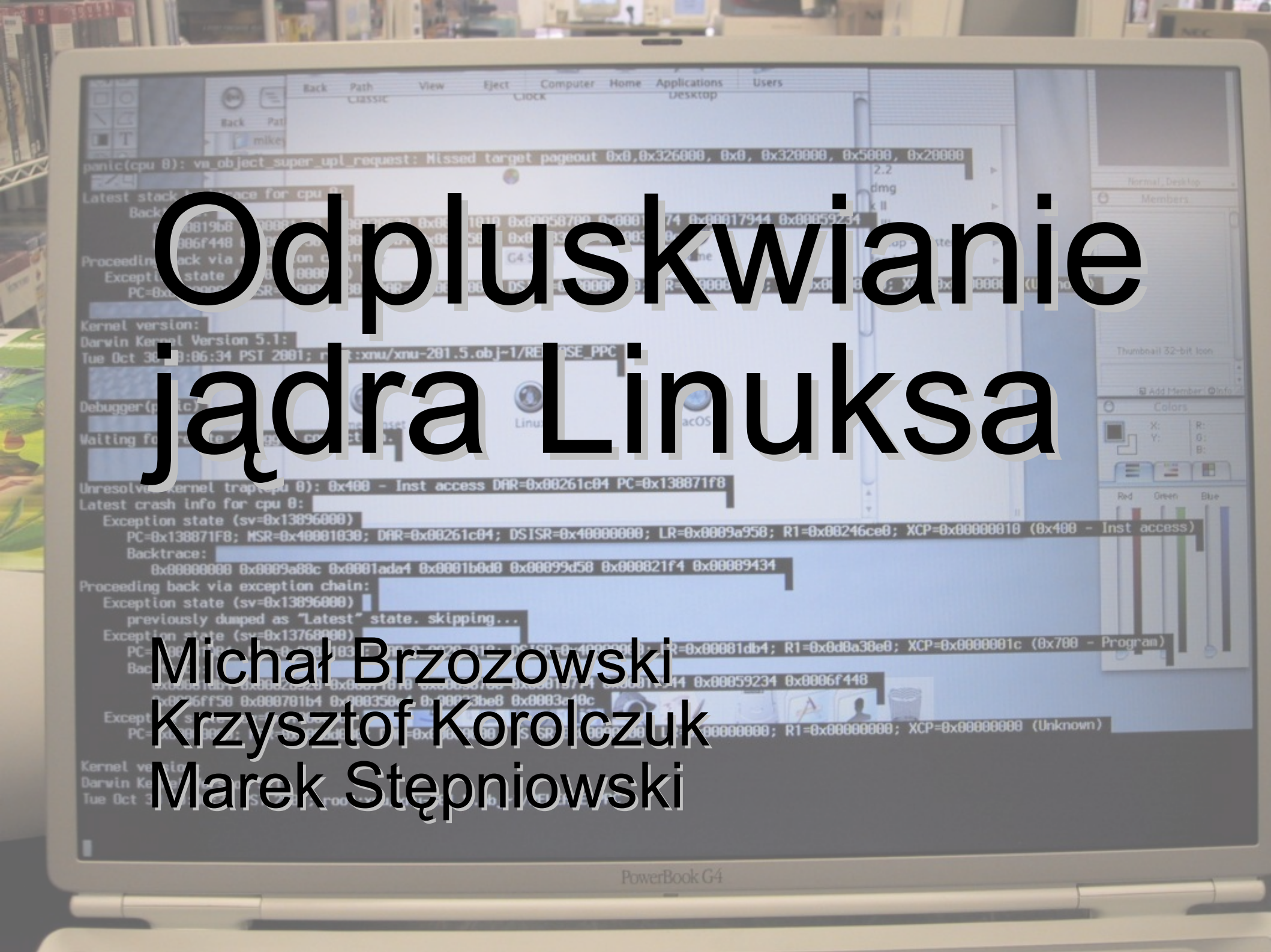
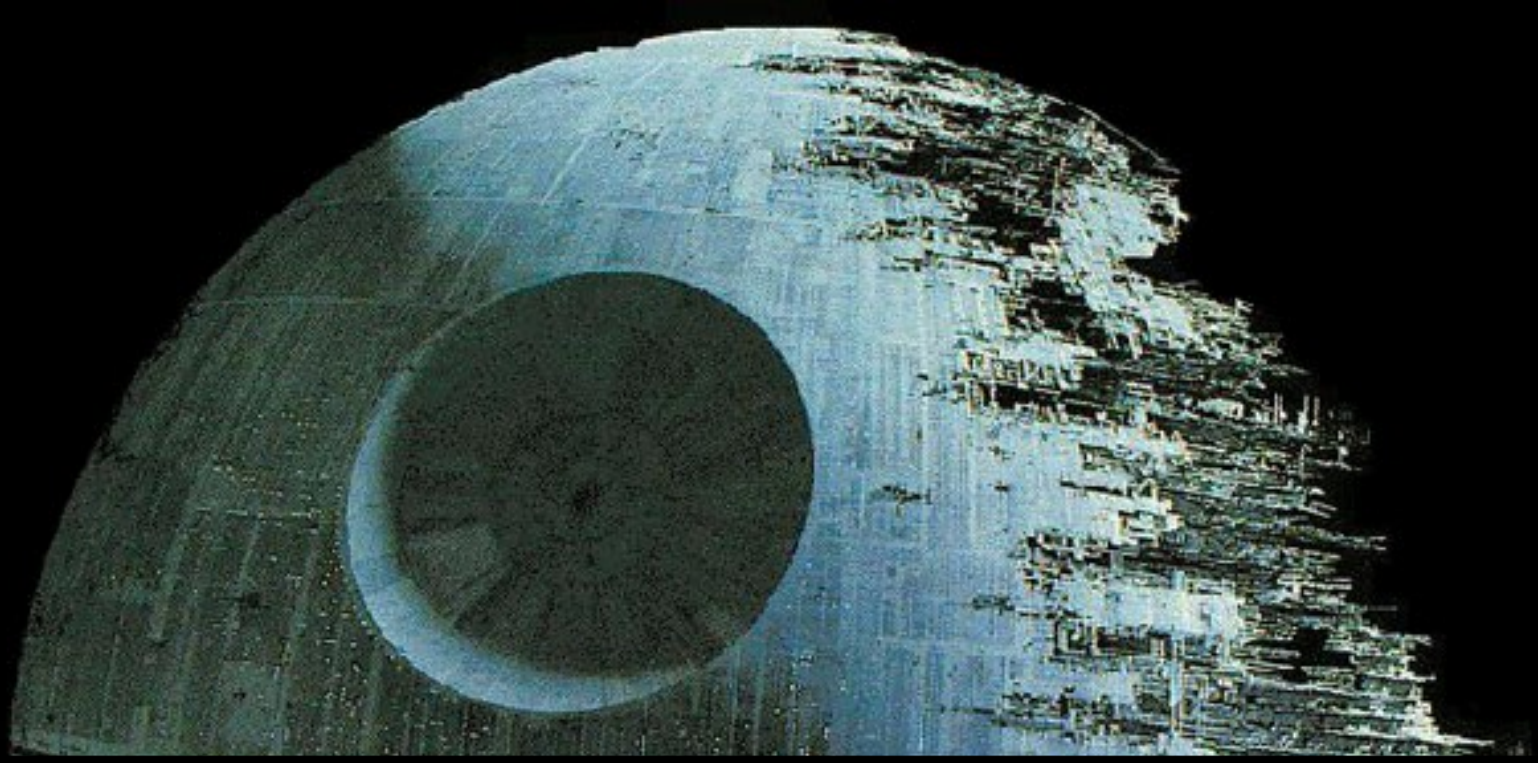

Odpluskwianie jądra Linuksa
Michał Brzozowski
Krzysztof Korolczuk
Marek Stępniowski
panic(cpu 0): vm_object_super_upl_request: Missed target pageout 0x0,0x326000, 0x0, 0x320000, 0x5000, 0x20000
Kernel version:
Darwin Kernel Version 5.1:
Debugger(panic)
Waiting for kernel debugger connection.
Unresolved kernel trap(cpu 0): 0x400 - Inst access DAR=0x00261c04 PC=0x138871f8
Latest crash info for cpu 0:
Exception state (sv=0x13896000)
PC=0x138871F8; MSR=0x40001030; DAR=0x00261c04; DSISR=0x40000000; LR=0x0009a958; R1=0x00246ce0; XCP=0x00000010 (0x400 - Inst access)
Backtrace:
0x00000000 0x0009a88c 0x0001ada4 0x0001b0d0 0x00099d50 0x000821f4 0x00089434
Proceeding back via exception chain:
Exception state (sv=0x13896000)
previously dumped as "Latest" state. skipping...
Exception state (sv=0x13768000)
PC=0x00001db4; R1=0x0d0a38e0; XCP=0x0000001c (0x700 - Program)
0x00059234 0x0006f448
PC=0x00000000; R1=0x00000000; XCP=0x00000000 (Unknown)
PowerBook G4



'Use the Source, Luke, use the Source. Be one with the code.'. Think of Luke Skywalker discarding the automatic firing system when closing on the deathstar, and firing the proton torpedo (or whatever) manually. _Then_ do you have the right mindset for fixing kernel bugs.

- Linus Torvalds

Kiedy używać debugera?

Kiedy zlokalizowanie błędu innymi metodami jest trudne.

- Błędy związane z wyciekami pamięci.
- Późno wykryty błąd (to nie powinno się zdarzyć!)
- Potrzebujemy naprawdę bardzo dokładnych informacji o tym, co dzieje się w kodzie.

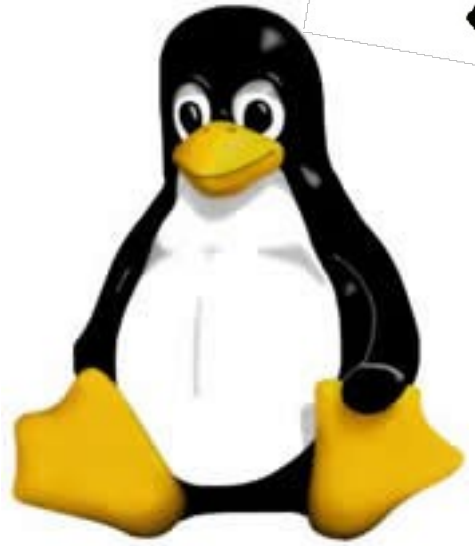
Kiedy nie używać debugera?

- Kiedy nie rozumiesz kodu, który odpluskwiasz - z pomocą debugera co najwyżej usuniesz przyczyny błędów.
- Jeśli zupełnie nie wiesz czego szukać. Lepiej przejrzyj wtedy dokładnie kod.

Przy odrobinie szczęścia.

W życiu zdarzają się...

W życiu zdarzają się... **wypadki.**



Tak samo linuks czasem mówi nam **oops!**

Na dodatek w jakimś niezrozumiałym narzeczu?


```
Unable to handle kernel NULL pointer dereference at virtual address  00000014
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0 c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000 00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038 00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>] [<c0108f77>]
[<c010002b>]

Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

A jednak po chwili zastanowienia...

Niski adres podpowiada nam, że to pole struktury.

```
Unable to handle kernel NULL pointer dereference at virtual address  00000014
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0 c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000 00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038 00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>] [<c0108f77>]
[<c010002b>]

Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

Całość okazuje się całkiem zrozumiała!

Numer oopsa. Tylko pierwszemu oopsowi (numer 0000) należy ufać.

```
Unable to handle kernel NULL pointer dereference at virtual address  00000014
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0 c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000 00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038 00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>] [<c0108f77>]
[<c010002b>]

Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

Być może z wyjątkiem...

Status programu i stan rejestrów procesora.

```
Unable to handle kernel NULL pointer dereference at virtual address  00000014
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0 c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000 00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038 00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>] [<c0108f77>]
[<c010002b>]

Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

... tych wszystkich cyferek.

Zawartość stosu.

```
Unable to handle kernel NULL pointer dereference at virtual address  00000014
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0 c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000 00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038 00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>] [<c0108f77>]
[<c010002b>]

Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

Te cyferki nic mi nie mówią!

Zapis ostatnich wywołań. Same adresy, zupełnie niezrozumiałe.

```
Unable to handle kernel NULL pointer dereference at virtual address  00000014
*pde = 00000000
Oops: 0000
CPU:    0
EIP:    0010:[<c017d558>]
EFLAGS: 00210213
eax: 00000000   ebx: c6155c6c   ecx: 00000038   edx: 00000000
esi: c672f000   edi: c672f07c   ebp: 00000004   esp: c6155b0c
ds: 0018   es: 0018   ss: 0018
Process tar (pid: 2293, stackpage=c6155000)
Stack: c672f000 c672f07c 00000000 00000038 00000060 00000000 c6d7d2a0 c6c79018
       00000001 c6155c6c 00000000 c6d7d2a0 c017eb4f c6155c6c 00000000 00000098
       c017fc44 c672f000 00000084 00001020 00001000 c7129028 00000038 00000069
Call Trace: [<c017eb4f>] [<c017fc44>] [<c0180115>] [<c018a1c8>] [<c017bb3a>]
[<c018738f>] [<c0177a13>]
       [<d0871044>] [<c0178274>] [<c0142e36>] [<c013c75f>] [<c013c7f8>] [<c0108f77>]
[<c010002b>]

Code: 8b 40 14 ff d0 89 c2 8b 06 83 c4 10 01 c2 89 16 8b 83 8c 01
```

System.map

na ratunek!

Lokacja pliku: /boot/System.map-`uname -n`

Plik z symbolami dla jądra Linuxa. Uwaga: nie ma tam symboli modułów ładowanych dynamicznie!

(z oops)
EIP: 0010:[<c017d558>]

(z pliku System.map)

....
c017cdf0 T reiserfs_dir_fsync
c017ce80 t reiserfs_readdir
c017d2f0 t create_virtual_node
c017d780 t check_left
c017d8d0 t check_right
....

EIP = function base address + instruction offset

Inny przydatny program - ksymoops

- Czyta tekst oopsa z Oops.file, klogd, kmsg lub z konsoli
- Na podstawie informacji z System.map, /proc/ksyms i /lib/modules tłumaczy adresy z oops na symbole

klogd

- Demon
- Loguje wiadomości jądra (w tym oopsy i wyjście printk!)
- Próbuje automatycznie tłumaczyć symbole (również te z modułów ładowanych dynamicznie)

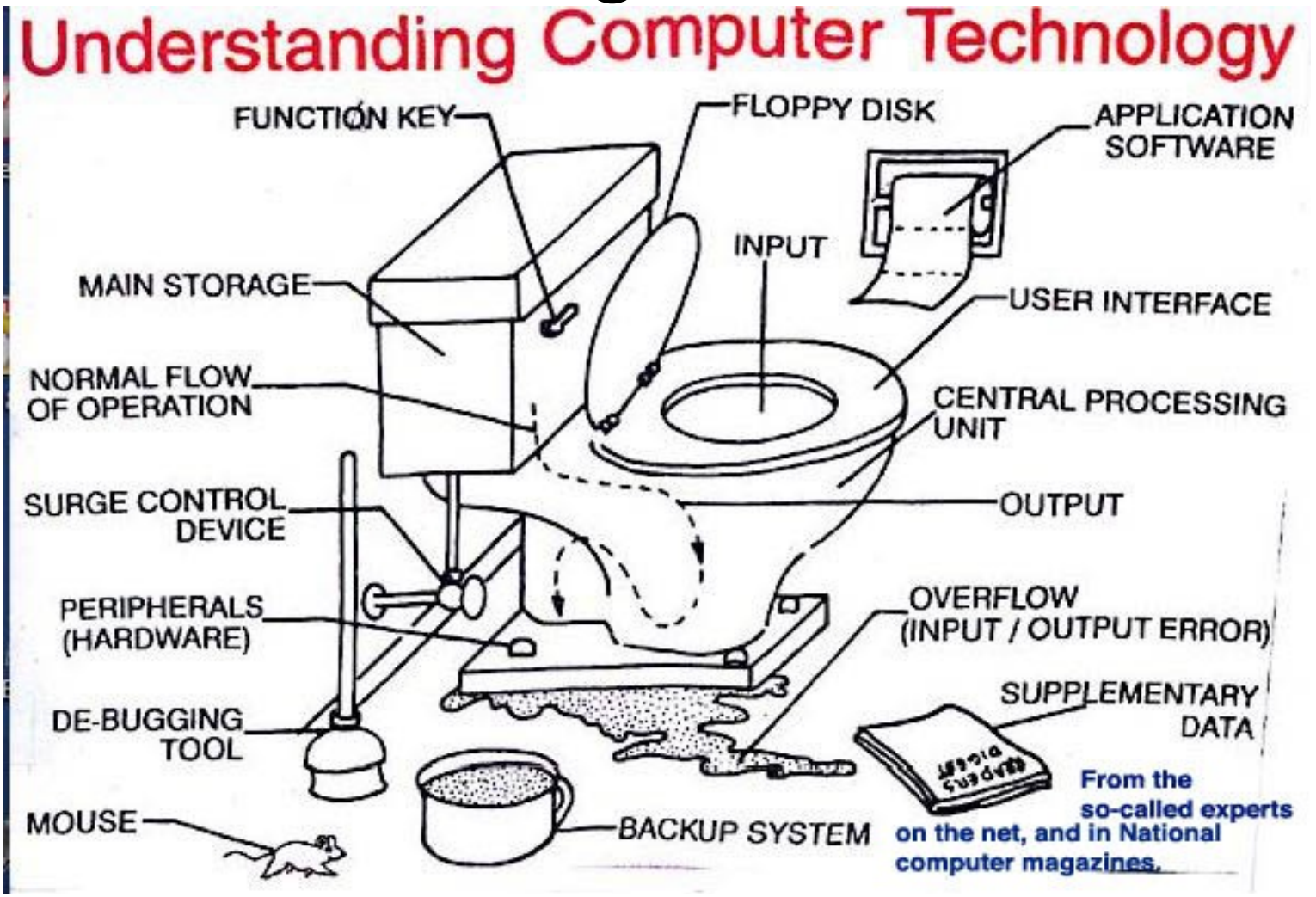
Dodatkowo

Debugowanie przy pomocy prostych narzędzi

- Co jeszcze potrafi ps?
- strace i ltrace
- Funkcje z API kernela: printk, BUG, panic

Debuggery

Debugowanie



Debugowanie a debugowanie jądra

- Są istotne różnice techniczne:
- Programy/biblioteki użytkownika korzystają z API jądra. Jeżeli debuger zatrzymał jądro to z czego sam korzysta?
- Symbole – są lub nie są
- Kod źródłowy – to samo
- Praktycznie nie do ominięcia „zabawy” z jądrem

Realizacja debugowania – user level

- Programy poziomu użytkownika:

ptrace rulez!

Dobry początek to 'man ptrace' i google (jak zwykle...) albo np.

http://www.linuxjournal.com/article/6100

funkcja systemowa(syscall) pozwalająca m.in za pomocą sygnałów (głównie SIGTRAP) kontrolowanie wykonywania się procesu potomnego(i nie tylko).

Realizacja debugowania - kernel

- UML

  No tak, można się wykręcić UMLem, debuger działa na innym jądrze niż debugowane więc jest ok. Pozostaje komplikacja używania (ale do przejścia)

- Kernel-level debugger:

  Prawdziwy debuger poziomu jądra. Działa jako kod jądra. Prawdziwa moc ale sporo problemów i niewygód.

Konsekwencje wyboru

- UML:
  - trzeba mieć UMLa :)
  - Trzeba umieć gdb i łączenie gdb z UML
  - Można słuchać MP3 debugując jądro :)
- Kernel-level debugger:
  - Patch na źródła kernela i ich rekompilacja prawie nieunikniona
  - Jeśli coś może się nie udać to się nie uda
  - Debuger blokuje wykonywanie się systemu operacyjnego. Do MP3 sugerowany odtwarzacz chyba że robimy to na maszynie wirtualnej...
  - Czasami na prawdziwym systemie debugger/debugowany działa inaczej niż na VM

Choose Your weapon

- GDB – do UML'a (o tym za chwilę)
- KDB – nie source-level! H4rdc0re azm c0dE
- KGDB – source-level, ale konieczne są 2 komputery i połączenie rs232
- Inne - różnie

GDB

przeznaczony głównie (jak większość debugerów pod linuxa) do debugowania programów do których mamy kod źródłowy lub zostały skompilowane z symbolami (najprościej: -g do opcji gcc, pliki obiektowe to temat na inną prezentację).

Podstawowe polecenia:

- Help, help cmd – od czegoś trzeba zacząć
- run – zaczyna wykonywanie załadowanego programu
- continue - wznawia wykonanie(po np złapaniu się na pułapkę)
- next – przechodzi do następnej instrukcji (wykonując bierzącą oczywiście)
- step – jw. Ale wchodzi w wywołania funkcji
- list, list xx – wylistowuje linijki kodu
- break fname, break xx = breakpoint na funkcję/nr linijki
- print v – wypisuje wartość zmiennej v
- set variable = value – ustawia wartość zmiennej
- quit - n/c

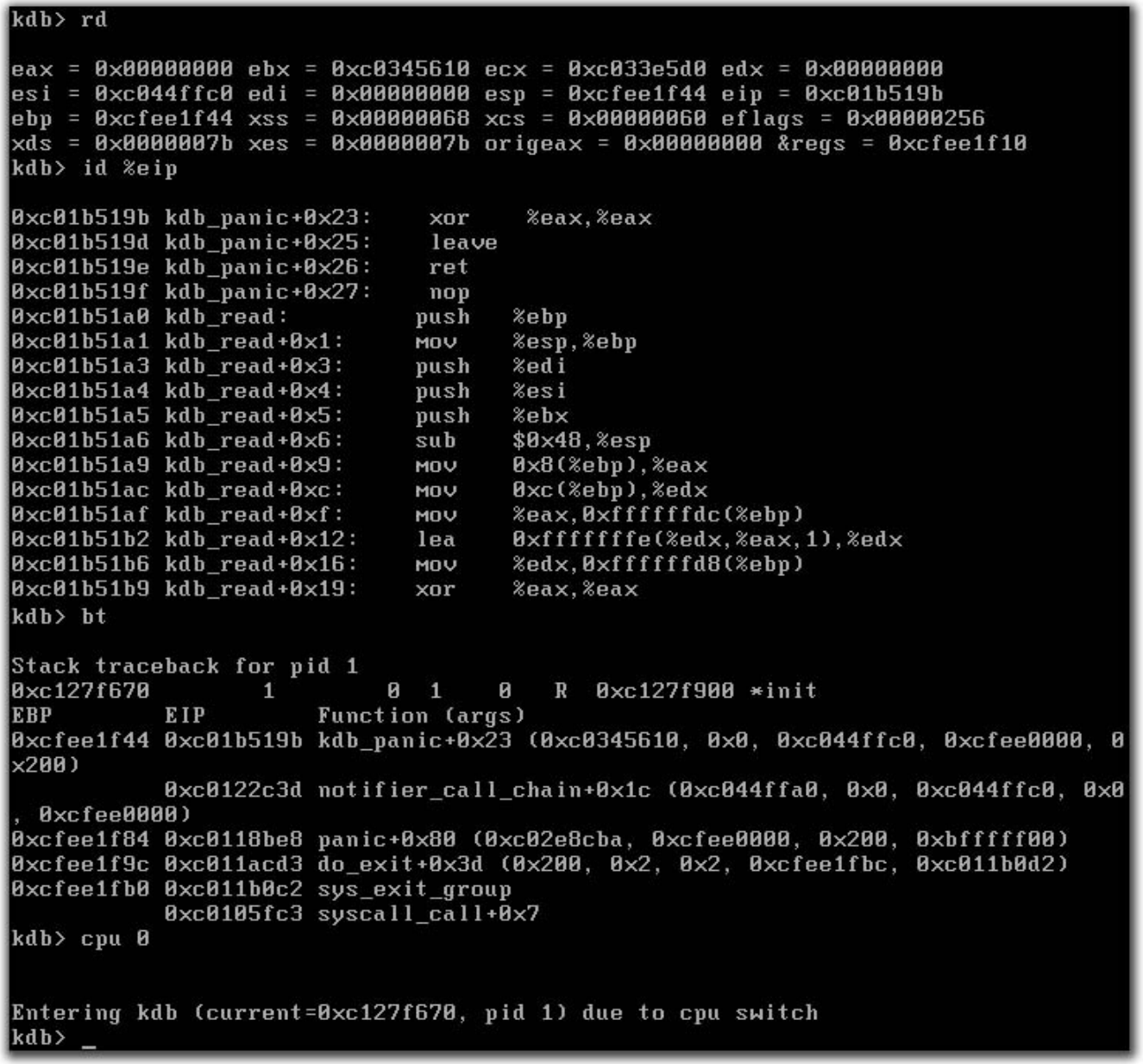
KDB

- Instalacja:
  - ściągamy „źródła” czyli patche do jądra
  - bunzip2
  - cd linux; patch -p1 < kdb-xxx-common; patch -p1 < kdb-xxx-arch
  - make menuconfig; „włączamy” kdb(być może trzeba dociągnąć “apt-get install libncurses-dev”
  - rekompilka jądra w zależności od dystrybucji
  - restart
  - ....
  - ponowna rekompilacja :)

KDB podstawy

- Hotkey: Pause/Break
- http://www-128.ibm.com/developerworks/linux/library/l-kdbug/
  - id – disassemblacja kodu od wskazanego adresu(id shedule – od aktualnego)
  - md, mdr – wyświetla zawartość pamięci pod zadanym adresem
  - mm – ustawia pamięć
  - rd, rm– pokazanie/modyfikacja stanu rejestrów
  - Bp, bph, bpha, bl, bc, be, bd, - ustawia breakpoint na symbol/adres, bl listuje wszystkie breakpointy, bc kasuje wybrane, be/bd włączają wyłączają wyrbane
  - bt, btp – backtrace, próba śledzenia ścieżki dojścia do aktualnej instrukcji. Bpt na zadany pid
  - Go - jak run w gdb

Slajd z “życia”
kernel panic po rekompilce z kdb



KGDB

- Oprócz instalacji, użytkowanie jest takie jak debugera gdb, bo pracujemy na gdb
- połączenie przez serial-line
- znów patche na jądro
- Dobre wytłumaczenie na: http://www.kernelhacking.org/docs/kernelhacking-HOWTO/indexs09.html

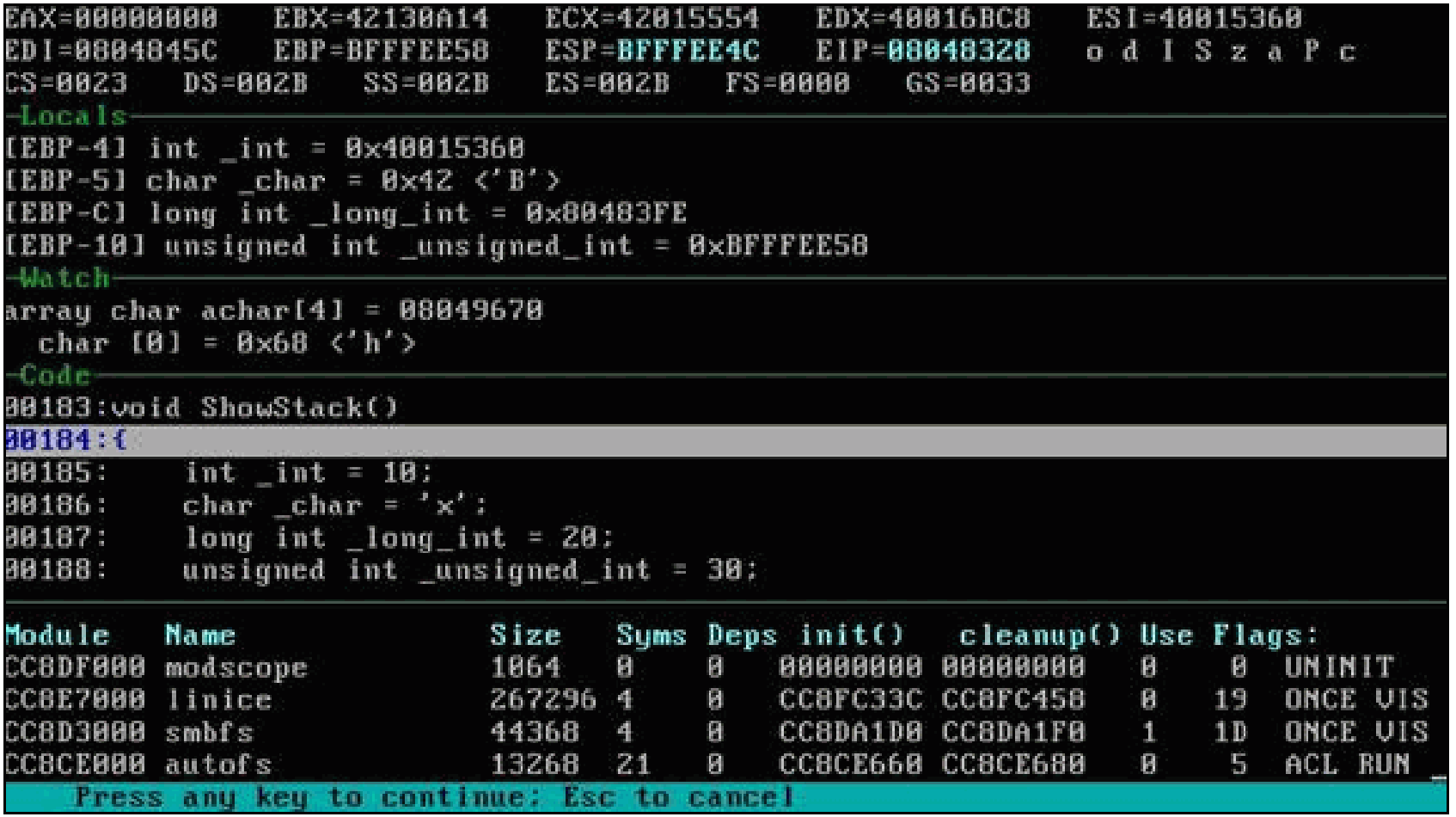
```
*** STOP: 0x00000019 (0x00000000,0xC00E0FF0,0xFFFFEFD4,0xC0000000)
BAD_POOL_HEADER

CPUID:GenuineIntel 5.2.c irql:1f  SYSVER 0xf0000565

Dll Base DateStmp - Name                 Dll Base DateStmp - Name
80100000 3202c07e - ntoskrnl.exe         80010000 31ee6c52 - hal.dll
80001000 31ed06b4 - atapi.sys            80006000 31ec6c74 - SCSIPORT.SYS
802c6000 31ed06bf - aic78xx.sys          802cd000 31ed237c - Disk.sys
802d1000 31ec6c7a - CLASS2.SYS           8037c000 31eed0a7 - Ntfs.sys
fc698000 31ec6c7d - Floppy.SYS           fc6a8000 31ec6ca1 - Cdrom.SYS
fc90a000 31ec6df7 - Fs_Rec.SYS           fc9c9000 31ec6c99 - Null.SYS
fc864000 31ed868b - KSecDD.SYS           fc9ca000 31ec6c78 - Beep.SYS
fc6d8000 31ec6c90 - i8042prt.sys         fc86c000 31ec6c97 - mouclass.sys
fc874000 31ec6c94 - kbdclass.sys         fc6f0000 31f50722 - VIDEOPORT.SYS
feffa000 31ec6c62 - mga_mil.sys          fc890000 31ec6c6d - vga.sys
fc708000 31ec6ccb - Msfs.SYS             fc4b0000 31ec6cc7 - Npfs.SYS
fefbc000 31eed262 - NDIS.SYS             a0000000 31f954f7 - win32k.sys
fefa4000 31f91a51 - mga.dll              fec31000 31eedd07 - Fastfat.SYS
feb8c000 31ec6e6c - TDI.SYS              feaf0000 31ed0754 - nbf.sys
feacf000 31f130a7 - tcpip.sys            feab3000 31f50a65 - netbt.sys
fc550000 31601a30 - el59x.sys            fc560000 31f8f864 - afd.sys
fc718000 31ec6e7a - netbios.sys          fc858000 31ec6c9b - Parport.sys
fc870000 31ec6c9b - Parallel.SYS         fc954000 31ec6c9d - ParVdm.SYS
fc5b0000 31ec6cb1 - Serial.SYS           fea4c000 31f5003b - rdr.sys
fea3b000 31f7a1ba - mup.sys              fe9da000 32031abe - srv.sys

Address  dword dump   Build [1381]                                - Name
fec32d84 80143e00 80143e00 80144000 ffdff000 00070b02             - KSecDD.SYS
801471c8 80144000 80144000 ffdff000 c03000b0 00000001             - ntoskrnl.exe
801471dc 80122000 f0003fe0 f030eee0 e133c4b4 e133cd40             - ntoskrnl.exe
80147304 803023f0 0000023c 00000034 00000000 00000000             - ntoskrnl.exe

Restart and set the recovery options in the system control panel
or the /CRASHDEBUG system start option.
```

Inne - LinICE

- Wzorowany na SoftICE – bardzo dobrym debugerem poziomu jądra pod systemy M$
- Działa jako moduł jądra, nie wymagając za tym patchowania/rekompilacji jądra.
- Dużo wygodniejsze poruszanie się po kodzie bez symboli (asm).
- Oczywiście działający lokalnie
- Niestety obecnie teoretycznie działa tylko na jądrach 2.4.x – 2.6.8. W praktyce też niezawsze.

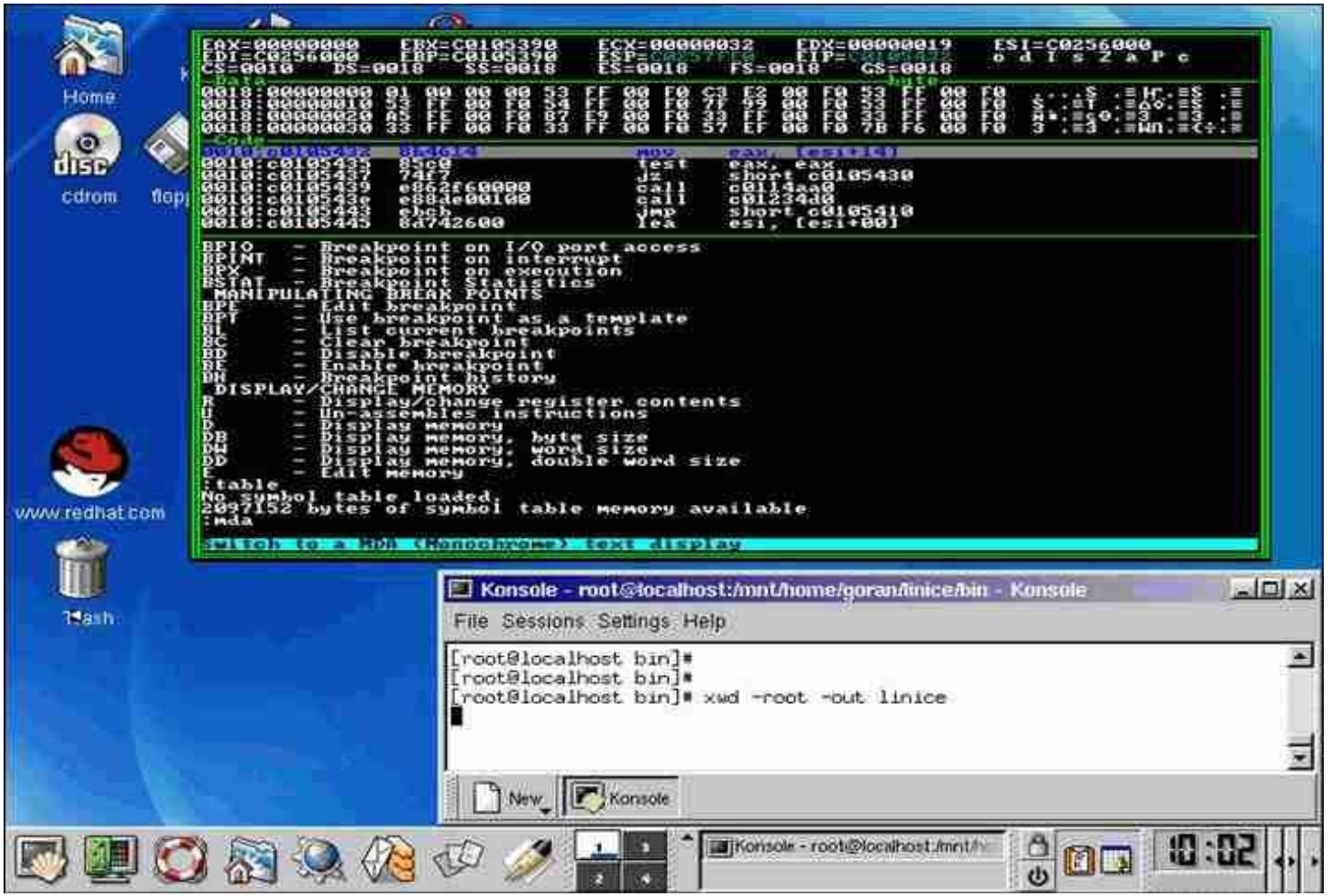
LinICE

- http://www.linice.com/



LinICE cd.

- Podobno da się nawet odpalić w X-ach



Inne - SysRq
naciskamy ALT+SysRq+CmdKey

Opcja kompilacji: CONFIG_MAGIC_SYSRQ

'r' - wyłącza tryb surowy klawiatury.

'k' - killuje wszyskie programy należące do virtualnej konsoli

'b' - reboot systemu (natychmiast)

'u' - próbuje przemountować system plików z read-only

'p' - wypisuje zawartość rejestrów

't' - wypisuje listę z informacjami o aktalnych zadaniach(tasks)

'm' - wypisze info o pamięci

'0'-'9' – ustawia log-level aktalnej konsoli

'e' -SIGTERM do wszystkiego oprócz init

'i' - jw ale SIGKILL

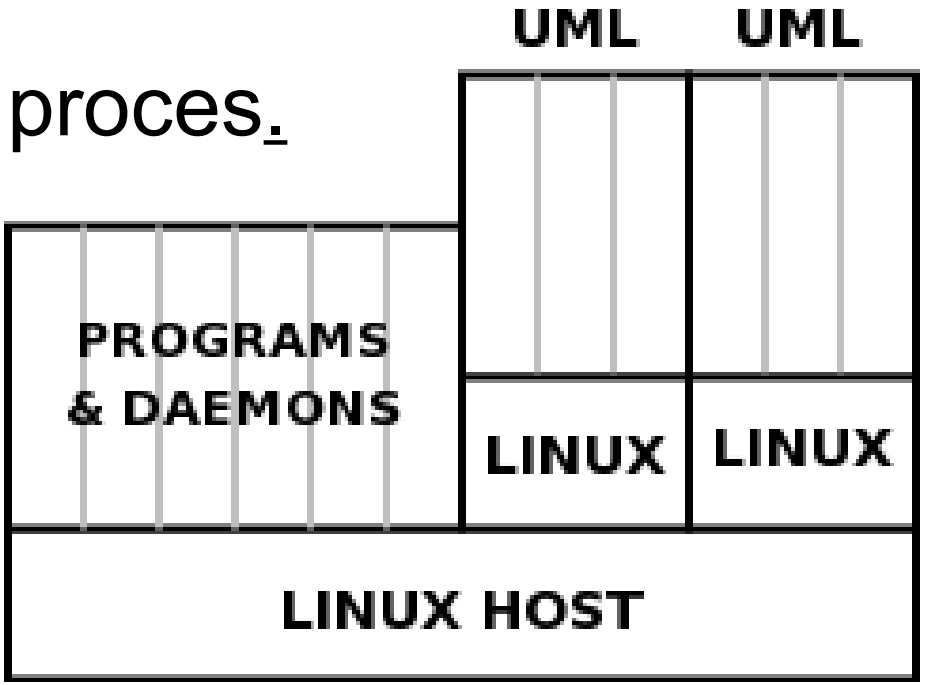
'l' -SIGKILL do wszystkich włącznie z init

Hope it works ;)

UML – User Mode Linux

- Co to jest UML?
  Służy do uruchamiania jądra Linuksa jako proces użytkownika.

- Jak debugować UML?
  Pod gdb, tak jak zwykły proces.



Instalacja UML

- Ściągnięcie źródeł jądra
- Ściągnięcie patcha
- Zaaplikowanie patcha
- Kompilacja jądra
- Ściągnięcie systemu plików
- Ściągnięcie uml-tools

Jądro + patch

- Użyjemy wersji jądra 2.6.18.1, na 2.6.17.13 nie ma patcha UML
- Wersja patcha - uml-2.6.18.1-bb2

Kompilacja jądra

- Kompilujemy ,,spatchowane" jądro
  `make xconfig ARCH=um` (konfiguracja)
  `make linux ARCH=um` (kompilacja)
- Uzyskujemy program `linux`

System plików

- Ściągamy system plików z ulubioną dystrybucją, ze strony: `http://uml.nagafix.co.uk`
- Zwracamy uwagę, by system plików był kompatybilny z jądrem
- Rozpakowujemy i zapisujemy jako `root_fs`
- Montowanie:
  `mount -o loop root_fs /katalog`

Uruchamianie UML

- Uruchamianie w zwykłym trybie: `./linux` (w katalogu, w którym istnieje `root_fs`)
- Uruchamianie z gdb: `gdb linux`

Gdb + UML

- Przed uruchomieniem UML w gdb należy wydać dodatkowe polecenia gdb:
  ```
  (gdb) handle SIGUSR1 pass noprint nostop
  (gdb) handle SIGSEGV pass noprint nostop
  ```
  (żeby zignorować sygnały idące do jądra)
- Uruchamiamy UML:
  ```
  (gdb) run
  ```
- W celu przerwania działania UML i przejścia do gdb, wysyłamy sygnał INT do UML

Debugowanie modułów

- Musimy wykrywać symbole ładowane dynamicznie
- Można do użyć pomocniczego skryptu `umlgdb`
- Można to robić ręcznie
- Szczegóły na stronie:
  `http://user-mode-linux.sf.net/debugging.html`

Bezpieczeństwo

- Procesy wewnątrz UML są widoczne jako wątki w systemie macierzystym
- Procesy wewnątrz UML mają dostęp do całej pamięci jądra UML
  - W ten sposób proces uruchomiony wewnątrz UML może uzyskać dostęp do procesu macierzystego
- Rozwiązanie – patch na jądro systemu macierzystego

Przydatne strony internetowe

- Strona UML:
  http://user-mode-linux.sourceforge.net
- UML – debugowanie
  http://user-mode-linux.sf.net/debugging.html
- Strona ze źródłami jądra Linuksa:
  http://www.kernel.org
- Strona z systemami plików różnych dystrybucji:
  http://uml.nagafix.co.uk
- IRC
  irc.oftc.net, kanal #uml

Demonstracja