Piotr Drajski

SPIS TREŚĆI

1. Dlaczego programy, które chcemy porządnie odpluskwiać powinniśmy skompilować z opcjami odpluskwiania? Co to są opcje odpluskwiania?

a) Co to są informacje odpluskwiania (Debugging information)?
b) Opcje odpluskwiania w GDB (najpopularniejsze).

2. Co to jest plik obiektowy? Formaty plików obiektowych.

a) Co to jest plik obiektowy?
b) Formaty plików obiektowych
c) Formaty na informacje odpluskwiania (najpopularniejsze)

3. Narzędzia do wykrywania wycieków pamięci w programie i do profilowania kodu.

a) Electric Fence
b) Inne programy tego typu
c) Purify
d) Valgird
e) Incense

4. Techniki odpluskwiania jądra

a) User Mode Linux
b) KSYMOOPS
c) KGDB

5. Narzędzia profilowania kodu

a) Co to jest profilowanie kodu?
b) gprof
c) gcov

1. Dlaczego programy, które chcemy porządnie odpluskwiać powinniśmy skompilować z opcjami odpluskwiania? Co to są opcje odpluskwiania?

a) **Co to są informacje odpluskwiania(*Debugging information*)?**
Dodatkowe informacje potrzebne przy odpluskwianiu, takie jak np. numery linii albo nazwy zmiennych, definicje struktur, które standardowo nie znajdują się w kodzie programu. Ponadto, kompilator może dzięki zastosowanym w nim opcjom optymalizacji w ogóle nie uwzględnić zadeklarowanej przez nas zmiennej w pamięci operacyjnej, gdyż będzie ona np. cały czas przechowywana w rejestrach procesora. Innymi słowy, bez tych opcji tak właściwie odpluskwiamy kod w postaci co najwyżej assembler`owej

b) **Opcje odpluskwiania w GDB (najpopularniejsze):**

-g	Generuje dodatkowe informacje wykorzystywane w procesie odpluskwiania w formacie standardowym dla danego systemu operacyjnego (stabs, COFF, XCOFF, DWARF).
-gstabs	Generuje dodatkowe informacje w formacie stabs.
-gcoff ***-gdwarf-2*** ***-gxcoff***	Produkuje dodatkowe informacje w standardach COFF, FWARF, XCOFF wykorzystywane na różnych platformach sprzętowych (man gcc).
-g (poziom)	Standardowo opcja -g generuje informację odpluskwiania na poziomie drugim. 1:Poziom 1 produkuje minimalną informacje, jednakże wystarczającą do opisu funkcji i zmiennych globalnych. 2: Poziom standardowy. 3: Poziom standardowy + makrodefinicje z kodu programu.

-Wall	Wypisuje ostrzeżenia dla wszystkich sytuacji, które pretendują do konstrukcji, których używania się nie poleca i których użycie jest proste do uniknięcia, nawet w połączeniu z makrami.
	
	Opcja ta pozwala nam wykryć błędy składniowe typu:
	-------------------------- Walltest.c------------------------
	#include<stdio.h>
	int main()
	{
	int variable;
	scanf("%d",variable); //powinno być scanf("%d",&variable)
	if (tt==1) // powino być if (variable==1)
	return 1;
	return 0;
	}
	-------------------------End WallTest.c --------------------
	Wywołanie:
	clamath@laptop:~/studia/so$gcc -Wall Walltest.c
	Walltest.c: In function `main':
	Walltest.c:5: warning: format argument is not a pointer (arg 2)
	Walltest.c:6: warning: suggest parentheses around assignment used as truth value

2. Co to jest plik obiektowy? Formaty plików obiektowych.

a) **Co to jest plik obiektowy?**

Plik obiektowy jest to plik zawierający następujące informacje:

- Nagłówek (Header): ogólna informacja o zawartości pliku taka jak: rozmiar kodu, data utworzenia modyfikacji itp.
- Kod obiektu (Object Code): instrukcje w postaci binarnej wygenerowane przez kompilator.
- Symbole(Symbols): globalne symbole zdefiniowane w module, które mają być importowane do innych modułów lub zdefiniowane przez program linkujący.
- Opcje odpluskwiania (Debugging information): inne informacje na temat pliku obiektowego potrzebne podczas procesu odpluskwiania, takie jak numery linii kodu czy nazwy zmiennych.
- Inne sekcje w zależności od typu pliku obiektowego.

b) Formaty plików obiektowych:

- **a.out** – format plików używany w systemach unixowych w znacznym stopniu wyparty przez format ELF. Składa sie on z następujących sekcji:

.nagłówek(header)
.sekcja kodu(text)
.data
.inne sekcje.

Wersji 1.1 format obsługiwał adresowanie do 16bit co ograniczało wielkość programów do 64k. Problem znikł w kolejnych wersjach, format obsługuje biblioteki dynamiczne ale muszą one być załadowane w ściśle określony adres pamięci. Aktualnie format jest używany bardzo rzadko.

- **ELF (Executable and Linkable Format) –** Format plików obiektowych, który zastąpił a.out. W Linuksie konwersja na ten format zaczęła sie w 1995r. Plik ten składa się z praktycznie dowolnej liczby sekcji co jest jedną z podstawowych różnic pomiędzy a.out i ELF. Posiada także lepszą implementację bibliotek dynamicznych. Daje np. możliwość ich załadowania w prawie dowolne miejsce w pamięci.

- **Microsoft Portable Executable File** – plik obiektowy wykorzystywany powszechnie w systemach Microsoft Windows wzorowany był na Unixowym pliku obiektowego formacie COFF. Podobnie jak ELF format ten został stworzony z myślą o stronicowaniach środowiskach przez co wszystkie sekcje w tym pliku są wyrównywane do wielkości stron w systemie Windows. Nazwa Portable wzięła się stad, że ten format można stosować zarówno na 32 i 64-bitowych maszynach. W oparciu o ten format w systemie Windows zaimplementowane są biblioteki dynamiczne (dll), pliki wykonywalne (exe), pliki SYS i inne.

  Więcej na stronie http://www.iecc.com/linker/.

c) Formaty na informacje odpluskwiania (najpopularniejsze). Są to:

- **stab** (symbol table). Stab to standardowy format na informacje odpluskwiania Został on opracowany na Uniwersytecie Berkley przez Peter`a Kessler`a dla pdx-debuggera pascala. Od tamtego czasu format ten rozpowszechnił się. Poniżej przedstawiam przykład w jaki sposób gcc dodaje symbole stabs do kodu.

Przykład wygenerowanych symboli stabs przez gcc:

-----prog.c-------------------

```
main() {
int a;
a=a+2;
return 0; }
----------------------
clamath@laptop:gcc -S -gstabs prog.c
                     .file   "test.c"
        .stabs  "test.c",100,0,0,.Ltext0
        .text
.Ltext0:
        .stabs  "gcc2_compiled.",60,0,0,0
        .stabs  "int:t(0,1)=r(0,1);-2147483648;2147483647;",128,0,0,0
        .stabs  "char:t(0,2)=r(0,2);0;127;",128,0,0,0
        .stabs  "long int:t(0,3)=r(0,3);-2147483648;2147483647;",128,0,0,0
        .stabs  "unsigned int:t(0,4)=r(0,4);0;-1;",128,0,0,0
        .stabs  "long unsigned int:t(0,5)=r(0,5);0;-1;",128,0,0,0
        .stabs  "long long int:t(0,6)=r(0,6);0;-1;",128,0,0,0
        .stabs  "long long unsigned int:t(0,7)=r(0,7);0;-1;",128,0,0,0
        .stabs  "short int:t(0,8)=r(0,8);-32768;32767;",128,0,0,0
        .stabs  "short unsigned int:t(0,9)=r(0,9);0;65535;",128,0,0,0
        .stabs  "signed char:t(0,10)=r(0,10);-128;127;",128,0,0,0
        .stabs  "unsigned char:t(0,11)=r(0,11);0;255;",128,0,0,0
        .stabs  "float:t(0,12)=r(0,1);4;0;",128,0,0,0
        .stabs  "double:t(0,13)=r(0,1);8;0;",128,0,0,0
        .stabs  "long double:t(0,14)=r(0,1);12;0;",128,0,0,0
        .stabs  "complex
int:t(0,15)=s8real:(0,1),0,32;imag:(0,1),32,32;;",128,0,0,0
        .stabs  "complex float:t(0,16)=R3;8;0;",128,0,0,0
        .stabs  "complex double:t(0,17)=R4;16;0;",128,0,0,0
        .stabs  "complex long double:t(0,18)=R5;24;0;",128,0,0,0
        .stabs  "void:t(0,19)=(0,19)",128,0,0,0
        .stabs  "__builtin_va_list:t(0,20)=*(0,2)",128,0,0,0
        .stabs  "_Bool:t(0,21)=eFalse:0,True:1,;",128,0,0,0
        .stabs  "test.c",130,0,0,0
        .stabs  "main:F(0,1)",36,0,0,main
.globl main
        .type   main, @function
main:
        .stabn 68,0,2,.LM1-main
.LM1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        .stabn 68,0,4,.LM2-main
.LM2:
        leal    -4(%ebp), %eax
.LBB2:
        addl    $2, (%eax)
        .stabn 68,0,5,.LM3-main
.LM3:
        movl    $0, %eax
        .stabn 68,0,6,.LM4-main
.LM4:
        leave
        ret
.LBE2:
        .size   main, .-main
        .stabs  "a:(0,1)",128,0,0,-4
        .stabn  192,0,0,.LBB2-main
        .stabn  224,0,0,.LBE2-main
        .text
        .stabs "",100,0,0,.Letext
.Letext:
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.3.6"
```

Wiecej: http://sourceware.org/gdb/current/onlinedocs/stabs_1.html#SEC1

- **DWARF** (Debug With Arbitrary Record Format) jest nowszym standardem informacji odpluskwiania niż stabs. Wykorzystywany w plikach ELF files.
  Obecnie format ten doczekał sie wersji 3.

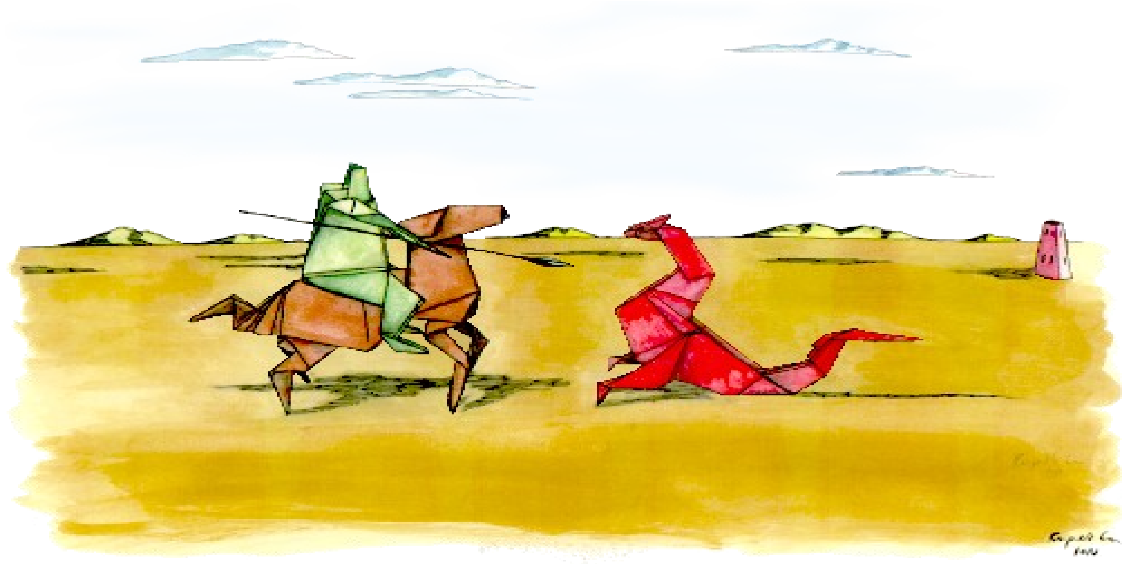
  Kod assemblerowy programu poniżej z symbolami DWARF.

  clamath@laptop:gcc -S -gstabs prog.c

```
    .file   "test.c"
        .file 1 "test.c"
        .section        .debug_abbrev,"",@progbits
.Ldebug_abbrev0:
        .section        .debug_info,"",@progbits
.Ldebug_info0:
        .section        .debug_line,"",@progbits
.Ldebug_line0:
        .text
.Ltext0:
.globl main
        .type   main, @function
main:
.LFB3:
        .loc 1 2 0
        pushl   %ebp
.LCFI0:
        movl    %esp, %ebp
.LCFI1:
        subl    $8, %esp
.LCFI2:
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        .loc 1 4 0
        leal    -4(%ebp), %eax
.LBB2:
        addl    $2, (%eax)
        .loc 1 5 0
        movl    $0, %eax
        .loc 1 6 0
        leave
        ret
.LBE2:
.LFE3:
        .size   main, .-main
        .section        .debug_frame,"",@progbits
.Lframe0:
        .long   .LECIE0-.LSCIE0
.LSCIE0:
        .long   0xffffffff
        .byte   0x1
        .string ""
        .uleb128 0x1
        .sleb128 -4
        .byte   0x8
        .byte   0xc
        .uleb128 0x4
        .uleb128 0x4
        .byte   0x88
        .uleb128 0x1
        .align 4
.LECIE0:
```

```
.LSFDE0:
        .long   .LEFDE0-.LASFDE0
.LASFDE0:
        .long   .Lframe0
        .long   .LFB3
        .long   .LFE3-.LFB3
        .byte   0x4
        .long   .LCFI0-.LFB3
        .byte   0xe
        .uleb128 0x8
        .byte   0x85
        .uleb128 0x2
        .byte   0x4
        .long   .LCFI1-.LCFI0
        .byte   0xd
        .uleb128 0x5
        .align 4
.LEFDE0:
        .text
.Letext0:
        .section        .debug_info
        .long   0x70
        .value  0x2
        .long   .Ldebug_abbrev0
        .byte   0x4
        .uleb128 0x1
        .long   .Ldebug_line0
        .long   .Letext0
        .long   .Ltext0
        .string "test.c"
        .string "/home/clamath/studia/so"
        .string "GNU C 3.3.6"
        .byte   0x1
        .uleb128 0x2
        .long   0x6c
        .byte   0x1
        .string "main"
        .byte   0x1
        .byte   0x2
        .long   0x6c
        .long   .LFB3
        .long   .LFE3
        .byte   0x1
        .byte   0x55
        .uleb128 0x3
        .string "a"
        .byte   0x1
        .byte   0x3
        .long   0x6c
        .byte   0x2
        .byte   0x91
        .sleb128 -4
        .byte   0x0
        .uleb128 0x4
```

Więcej na temat tego formatu: http://www.tachyonsoft.com/dwarf.pdf
Dokładny opis tych plików obiektowych znajduje się pod adresem:
http://www.iecc.com/linker/



3. Narzędzia do wykrywania wycieków pamięci w programie i do profilowania kodu.

a) **Electric fence** – narzędzie działające zarówno pod Linuksem jak i Windows. Dostępne w postaci kodu źródłowego. Jedyne co potrafi, to sprawdzać czy nie „wychodzimy" poza zadeklarowaną pamięć dynamiczną. Niestety z powodu mało czytelnych komunikatów (tak właściwie program skompilowany z tą biblioteką jedynie co potrafi zrobić to zwrócić SEGFAULT gdy znajdzie błąd na pamięci. Resztę informacji co tak właściwie sie stało musimy wyszukać sami np przy użyciu gdb. Aby dołączyć go do programu jedyne co trzeba zrobić to dołączyć dodatkową bibliotekę -lefence do kompilowanego programu.
Strona projektu:http://directory.fsf.org/ElectricFence.html

b) Innymi programami typu o większych możliwościach i lepszych czytelniejszych komunikatach o stanie pamięci, a działających na tej samej zasadzie co Electric Fence są:

- memwatch,
- NJAMD(not just another malloc debugger),
- YAMD (yet another memory debugger).

c) **Purify** – jest to nadrzędzie firmy IBM wyszukujące błędy zarówno w kodzie binarnym jak i w kodzie źródłowym programu!! Narzędzie działa zarówno pod systemem Windows jak i Linux. Niestety jest to narządzie komercyjne, co ogranicza jego wykorzystanie w projektach OpenSource`owych. Więcej na www.ibm.com/software/awdtools/**purify**plus/

d) **Valgrind** – oryginalnie miał być darmowym odpowiednikiem Purify.
Z biegiem czasu stał sie potężnym narzędziem do odpluskwiania i profilowania kodu. Valgrind niestety działa tylko na maszynach x86. Narzędzie to, odmiennie od prezentowanych programów w punkcie c), tworzy wirtualną maszynę, w którym uruchamia testowany program. Podczas uruchomienia odpluskwianego programu na wirtualnej maszynie wypisywane są czytelne komunikaty na temat uruchomionego programu, takie jak np. niezwolnienie pamięci zadeklarowanej w linii o numerze n. Dość poważną wadą tego programu są wymagania pamięciowe. Program ten jest w stanie skonsumować ogromne ilości pamięci operacyjnej. Zgodnie z zaleceniem twórców Valgrind należy uruchamiać na najsilniejszej maszynie do jakiej mamy aktualnie dostęp. Program jest dobrze udukomentowany . Na stronie projektu http://valgrind.org/ znajduje się wprowadzenie(tutorial) dla początkujących.

e) **Insure++** – to komercyjne narzędzie firmy Parasoft. Jego zaletą jest to, że działa na wielu platformach Windows, Linux, Solaris, PowerPC, HP-UX 11 for PA-RISC 32 oraz 64 bit. Jego możliwości są porównywalne, jeśli nie większe od wszystkich wymienionych wyżej programów. W artykule z LunuxJournal „Memory Leak Detection in C++" napisanego przez Cal`a Erickson`a 2003-06-01 (http://www.linuxjournal.com/article/6556) wynika, że program ten znalazł błędy, których nie udało sie znaleźć przy pomocy innych wymienionych tutaj programów.

4. Techniki odpluskwiania jądra

a) **User Mode Linux** – Czemu ma to służyć? User mode linux pozwala na odpalenie jądra Linux pod uruchomionym jądrem w przestrzeni użytkownika (dlatego user mode).
W praktyce oznacza to, że tworzymy zupełnie wirtualna maszynę, na której możemy testować nasze moduły do jądra lub bezpośrednio je zmieniać bez niebezpieczeństwa zawieszenia naszego prawdziwego systemu. Ważną cechą UML`a jest to, że moduły można kompilować poza trybem wirtualnym (co z pewnością znacząco skraca czas kompilowania). Ponadto, skoro jadro zachowuje się jak normalny program to możemy je odpluskwiać przy użyciu np gdb.
Więcej na ten temat można poczytać na stronie:http://user-mode-linux.sourceforge.net/
Tutaj zamieszczę plik Makefile, który pozwala na kompilacje jądra w taki sposób, aby działało ono pod UML, gdyż na stronie projektu nie jest to udukomentowane :/

----MakeFile znaleziony gdzieś na grupach dyskusyjnych ----------------

```
# KERNELDIR can be speficied on the command line or environment
        KERNELDIRUML = <sciezka do zrodel jadra UML>
        KERNELDIR = <sciezka do prawdziwego jadra >

# The headers are taken from the kernel
#       INCLUDEDIR = $(KERNELDIR)/include

all:module.o

#UML MODULE INCLUDE
ARCHDIR=$(KERNELDIRUML)/arch/um
TTDIR=$(ARCHDIR)/kernel/tt
SKASDIR=$(ARCHDIR)/kernel/skas
CFLAGSUML= -I$(KERNELDIRUML)/include -I$(ARCHDIR)/include  -I$(TTDIR)/include -I
$(SKASDIR)/include

#NORMAL REAL MODULE INCLUDE
CFLAGS= -I$(KERNELDIR)/include

umlmodule.o:
        gcc -Wall -c -DMODULE -D__KERNEL__ -DLINUX $(CFLAGSUML) module.c -o
built/umlmodule.o

module.o:
        gcc -c -DMODULE -D__KERNEL__ -DLINUX $(CFLAGS) module.c -o built/module.o

clean:
        rm -f built/*.o bulit/test
```

----------------------------End Makefile --------------------------

b) **KGDB** – Kgdb jest łatką na jądro, która daje możliwość jego odpluskwiania. Żeby korzystać z Kgdb potrzebne są dwa komputery połączone ze sobą np. łączem szeregowym lub jedną maszynę emulować przy użyciu programu np. VMWARE. Na maszynie testowej uruchamiamy nasze skompilowane jądro z łatką Kgdb. Wówczas pojawia sie komunikat: "*Waiting for connection from remote gdb*". W tym czasie, na drugiej maszynie odpalamy gdb i ustanawiamy połączenie z maszyną testową. Oczekiwanie na maszynę testową odbywa się poprzez dodanie do kodu jądra w pliku init/main.c w metodzie

__init start_kernel(void) metody oczekiwania na połączenie. Dalej pozostaje już życzyć miłej pracy z gdb. Strona Domowa http://kgdb.linsyssoft.com/

c) **KSYMOOPS** – jest to narzędzie to uzyskiwania informacji z komunikatów, które Linux wyświetla na konsoli w sytuacjach wyjątkowych, gdy dzieje się w jądrze coś niepożądanego. Komunikat ten może zawierać wiele informacji; jedną z nich na pewno będą wartości rejestrów procesora w momencie wystąpienia problemu. Taką informację zapisaną do pliku tekstowego niewykluczone, że będziemy zmuszeni przepisać z ekranu na kartkę, gdyż system nie zapisze tego komunikatu do dziennika. Przekazujemy ten plik programowi ksymoops, który na jego podstawie i kilku innych plików takich jak system.map plik jądra itp. Wypisze nam bardziej czytelny plik w postaci kodu assemblerowego.
Strona Domowa :http://directory.fsf.org/ksymoops.html

Tak mógłby wyglądać komunikat (przynajmniej jego początek) wypisany na konsolę oraz przy odrobinie szczęścia zapisany także do pliku z logami.

```
  TFS-error: ntfs-attr-find(): inode is corrupt. Run chkdsk.
Unable to handle kernel NULL pointer dereference at virtual address
0000000c
printing eip:
e00cc5df
*pde = 00000000
Oops: 0002 [#1]

CPU:0
EIP:0060:[<e00cc5df>]. Not tainted VLI
EFLAGS: 00010286 (2.6.9)
EIP is at ntfs_attr_find + 0x22f/0x280 [ntfs]

eax: 00000043
ebx: e00d907c
ecx: c0330464
edx: 00000000
esi: 00000000
edi: 00000000
ebp: 00000000
esp: d072ddd8
ds: 0076
es: 007b
ss: 0068
```

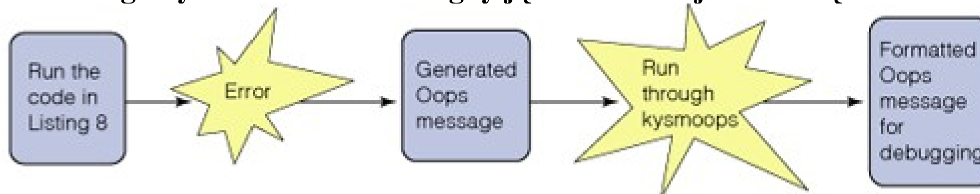
A tak wygląda przykładowy plik wygenerowany przez ksymoops:

```
ksymoops 2.4.0 on i686 2.4.17. Options used
... 15:59:37 sfb1 kernel: Unable to handle kernel NULL pointer dereference at
virtual address 0000000
... 15:59:37 sfb1 kernel: c01588fc
... 15:59:37 sfb1 kernel: *pde = 0000000
... 15:59:37 sfb1 kernel: Oops: 0000
... 15:59:37 sfb1 kernel: CPU:    0
... 15:59:37 sfb1 kernel: EIP:    0010:[jfs_mount+60/704]

... 15:59:37 sfb1 kernel: Call Trace: [jfs_read_super+287/688]
[get_sb_bdev+563/736] [do_kern_mount+189/336] [do_add_mount+35/208]
[do_page_fault+0/1264]
... 15:59:37 sfb1 kernel: Call Trace: [<c0155d4f>]...
... 15:59:37 sfb1 kernel: [<c0106e04 ...
... 15:59:37 sfb1 kernel: Code: 8b 2d 00 00 00 00 55 ...

>>EIP; c01588fc <jfs_mount+3c/2c0> <=====
...
Trace; c0106cf3 <system_call+33/40>
```

```
Code;  c01588fc <jfs_mount+3c/2c0>
00000000 <_EIP>:
Code;  c01588fc <jfs_mount+3c/2c0>   <=====
   0:  8b 2d 00 00 00 00          mov     0x0,%ebp    <=====
Code;  c0158902 <jfs_mount+42/2c0>
   6:   55                        push    %ebp
```

Z takiego pliku przy odrobinie wprawy możemy już wywnioskować, że błąd spowodowała instrukcja o adresie 3c w piku jfs_mount.o.
Mając taką wiedzę i odpowiednią znajomość assemblera można samemu znaleźć przyczynę wystąpienia tego błędu lub wysłać ten wygenerowany plik do twórcy kodu. Uwaga! Same wysłanie pliku oops do twórcy nic mu nie pomoże, gdyż komunikat oops jest specyficzny dla twojej maszyny i jądra. Dopiero plik wygenerowany przez ksymoops jest w stanie mu pomóc.

Ogólny schemat działania gdy jądro informuje nas o błędzie.



5. Narzędzia profilowania kodu.

Profilowanie kodu to sprawdzanie jak często procesor wykonuje dany fragment kodu. Mając wiedzę na temat tego jak długo procesor spędza w danej części programu wiemy jaki jego fragment należy optymalizować w pierwszej kolejności, aby zauważalnie przyspieszyć działanie programu. Ponadto, mając wiedzę na temat np. ilości odwołań do danego fragmentu możemy wywnioskować czy program działa poprawnie czy nie. Jeśli np. w kodzie występuje instrukcja „*if warunek then i1 else i2*” i i1 nigdy nie jest wykonywany możemy sie domyślać, że źle napisaliśmy warunek if. Kolejnym ciekawym zastosowaniem profilowania kodu jest faktyczne zmierzenie, który to sposób implementacji danej metody jest najbardziej efektywny.

b) gprof – jest standardowym narzędziem dostępnym w większości dystrybucji do profilowania kodu. Użycie dla przykładowego kodu:

```
----------myprog.c --------------------------
#include<stdio.h>
void p(void) {
printf("koza\n");
}
int main() {
int a=12000;
    while (a>0) {
        if (a>0)
            p();
    if (a<0)
              printf("tutaj nigdy nie dojde!\n");
    a--;}
return 0;}
```

1. Kompilujemy program: gcc myprog -o myprog -pg -g
2. Uruchamiamy go: ./myprog (zauważ, że jeśli program zakończył się pomyślnie tzn. funkcja main zakończyła sie kodem 0 lub została wywołana funkcja exit(0) to w bieżącym katalogu znajdzie sie plik _gmon.out._
3. Uruchamiamy w bieżącym katalogu: gprofiler myprog

4. Na ekranie pojawią sie informacje dot. naszego programu (fragmenty):

a) Czas działania programu:

```
                each sample counts as 0.01 seconds.
   %   cumulative   self              self     total
  time   seconds   seconds    calls  ns/call  ns/call  name
 100.55      0.01     0.01    12000   837.89   837.89  p()
```

b) Liczba wywołań danej funkcji z jakiejś funkcji:

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.01                 main [1]
                0.01    0.00   12000/12000       p() [2]
-----------------------------------------------
                0.01    0.00   12000/12000       main [1]
[2]    100.0    0.01    0.00   12000         p() [2]
-----------------------------------------------
```

c) gcov

Wartym uwagi jest też program gcov, który sprawdza ile razy dana linia kodu była wykonywana.

1. Kompilujemy gcc myprog.c -fprofile-arcs -ftest-coverage -o myprog
2. Uruchamiamy program: ./myprog
3. gcov myprog otrzymujemy plik myprog.c.gcov

```
    -:    0:Object:tq.bb
    -:    1:#include<stdio.h>
12000:    2:void p(void) {
12000:    3:printf("koza\n");
    -:    4:}
    1:    5:int main() {
    1:    6:int a=12;
12001:    7:    while (a>0) {
12000:    8:        if (a>0)
12000:    9:            p();
12000:   10:            if (a<0)
#####:   11:             printf("tutaj nigdy nie dojde!\n");
12000:   12:        a--;}
    1:   13:return 0;
    -:   14:}
```

Numery po lewej stronie oznaczają ile razy dana linia była wykonana. Widać tutaj, że np. linia 11 nigdy nie została wykonana, co może świadczyć o niepoprawności danego programu.