ODPLUSKWIANIE

(DEBUGGOWANIE)

Spis treści:

Co to jest debugger i jak działa?

Dawno, dawno temu, za siedmioma górami i siedmioma lasami konstruowano duże lampowe komputery, pierwowzory dzisiejszych pecetów. Ówczesne komputery złożone były z olbrzymiej ilości lamp (tranzystory jeszcze wtedy tak powszechnie nie używano). Lampy miały dwie wady: spore zużycie mocy i krótki czas życia. Krótki czas życia, skrócony dodatkowo przez istnienie wielu lamp, powodował że ówczesne komputery częściej nie działały niż działały. Dodatkowo ich rozmiary powodowały duże problemy z namierzeniem usterki. Któregoś razu okazało się, że kolejny błąd w działaniu systemu spowodował robak, którego śmiertelnie poraził prąd w trakcie wędrowania po płycie. Robak, a konkretnie pluskwa (po angielsku bug) nadał nazwę procesowi wyszukiwania i usuwania błędów w programie - debuggowaniu, po polsku odpluskwianiu.

Debugger jest to program komputerowy służący do analizy kodu źródłowego lub kodu maszynowego w celu znalezienia w nim błędów programistycznych (bugów). Proces naprawy kodu za pomocą debuggera określa się mianem debuggowania.

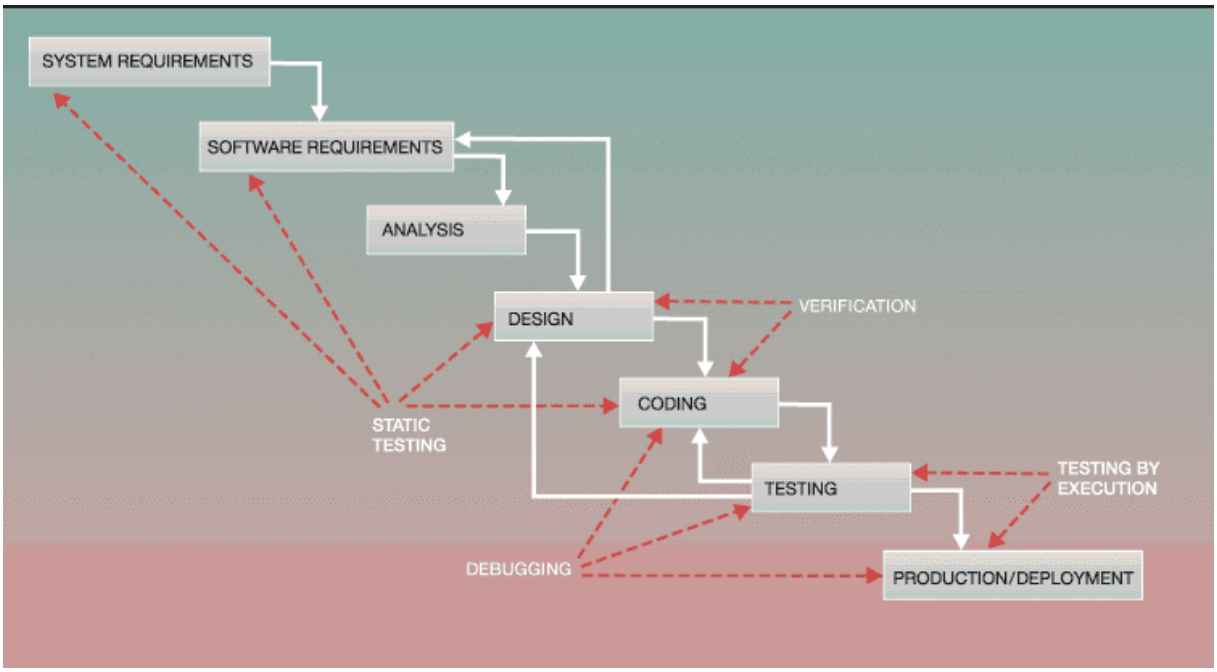
Podstawowym zadaniem debuggera jest symulowanie i sprawowanie kontroli nad wykonaniem kodu, co umożliwia zlokalizowanie instrukcji odpowiedzialnych za wadliwe działanie programu. Oczywiście, współczesne debuggery mają rozbudowane opcje, które pozwalają na efektywne śledzenie wartości poszczególnych zmiennych, wykonywanie instrukcji krok po kroku czy wstrzymywanie działania programu w określonych miejscach.

Debugger jest standardowym wyposażeniem każdego środowiska programistycznego. Niektóre z tych środowisk pozwalają ustawić w kodzie źródłowym punkty wstrzymania, dzięki czemu programista może m.in. śledzić wartości wskazanych zmiennych.

Debugger może również funkcjonować jako niezależny program.

Debuggery, oprócz ich podstawowego przeznaczenia, są często wykorzystywane także do łamania zabezpieczeń oprogramowania (crackingu).

Debuggery posiadają niestety wady – symulacja działania kodu nie jest idealnym odtworzeniem wykonania tego kodu w warunkach normalnych. Wobec tego debuggery mogą nie wykrywać bugów niezależnych bezpośrednio od treści badanego programu.



Rys 1. Typowy proces powstawania oprogramowania komputerowego

Chociaż każdy proces debuggowania jest unikalny, to istnieje kilka zasad, które można zastosować podczas debuggowania.

Główne kroki podczas debuggowania:

1. Stwierdzenie, że do programu wkradł się robak
2. Wyizolowanie źródła problemu
3. Zidentyfikowanie przyczyny błędu
4. Określenie sposobu naprawy
5. Naprawa i testy

Stwierdzenie, że do programu wkradł się robak

Doświadczony programista często wie, gdzie problemy pojawiają się najczęściej. Na przykład, wszystkie dane pochodzące od użytkownika powinny zostać potraktowane ze szczególną ostrożnością. Dużą wagę powinno się przywiązywać do tego, aby format i zawartość danych były poprawne. Jeżeli dane są transmitowane, to powinno się zadbać o sprawdzenie, czy cała wiadomość dotarła, czy też nie. Bardziej złożone dane, które muszą zostać sparsowane i/lub przetworzone, mogą zawierać nieoczekiwane kombinacje wartości, których program się nie spodziewał i które może źle obsłużyć. Poprzez sprawdzanie najczęściej występujących błędów, program może wykryć kiedy dane zostały uszkodzone lub niepoprawnie obsłużone.

Jeżeli problem jest na tyle poważny, że powoduje, że program pada, wtedy wkradnięcie się błędu jest oczywiste. Jeżeli program wykryje mniej poważny problem, pluskwa może zostać rozpoznana, pod warunkiem, że działanie programu jest monitorowane logami. Jeżeli jednak problem jest na tyle

mały, że powoduje tylko zły wynik, wtedy może być trudno określić czy do programu wkradł się robak.

Wyizolowanie źródła problemu

Jest to często najtrudniejszy krok podczas debuggowania. Jego celem jest zidentyfikowanie, która część programu powoduje błąd. Krok ten często wymaga iteracyjnego testowania. Dla programów podzielonych na moduły krok ten może być łatwiejszy dzięki sprawdzeniu poprawności danych przesyłanych pomiędzy interfejsami poszczególnych modułów. Jeżeli dane wejściowe były poprawne, a wyjściowe niepoprawne, to zlokalizowaliśmy moduł z błędem. Poprzez iteracyjne testowanie wejścia i wyjścia programista może zidentyfikować do kilku linii kodu miejsce wystąpienia błędu.

Zidentyfikowanie przyczyny błędu

Kiedy już znaleźliśmy miejsce wystąpienia błędu, kolejnym krokiem jest ustalenie przyczyny tego błędu. Dobra znajomość programu może okazać się tu niezbędna. Wyszkolony debugger może, co prawda, znaleźć miejsce wystąpienia błędu, ale już tylko osoba dobrze znająca program może precyzyjnie określić przyczynę błędu. Kiedy już znamy przyczynę błędu, dobrym pomysłem może być przejrzenie podobnych sekcji kodu celem sprawdzenia, czy ten sam błąd nie został powielony w innych miejscach.

Określenie sposobu naprawy

Następnym krokiem powinno być określenie sposobu naprawy błędu. Dokładna znajomość programu jest tu niezbędna dla wszystkich (może poza tymi najbardziej banalnymi) rodzajów błędów. Jest to spowodowane tym, że naprawa zmienia aktualne zachowanie programu i może skutkować nieoczekiwanymi rezultatami. Co więcej, naprawa pojedynczego błędu może spowodować powstanie nowych błędów lub ujawnienie się innych błędów istniejących w programie.

W niektórych przypadkach naprawa może być prosta i oczywista. Jest to dosyć proste podczas błędów natury logicznej, kiedy oryginalny projekt został nieprawidłowo zrozumiany i zaimplementowany. Z drugiej jednak strony, jeżeli jednak problem ujawnia jakiś bardziej złożoną wadę projektową, wtedy problem może być nie tylko trudny, ale nawet niemożliwy do naprawy. Może to czasem prowadzić do konieczności przepisania całego programu od nowa.

Czasem warto także zastosować szybką naprawę, która będzie poprzedzała dokładną naprawę w przyszłości. Decyzja taka powinna być podjęta

po przeanalizowaniu wagi, widoczności, częstotliwości występowania oraz efektów ubocznych problemu.

Naprawa i testy

Po naprawie błędu, ważne jest przetestowanie programu. Testy powinny być przeprowadzone z dwóch powodów:

- czy naprawa prawidłowo rozwiązała istniejący problem
- czy na skutek naprawy nie powstały jakieś efekty uboczne

Opcje odpluskwiania w gcc

Opcje debuggera.	
-g	-g powoduje włączanie do pliku wynikowego informacji (numery linii, typ i rozmiar identyfikatorów, tablica symboli) umożliwiających śledzenie wykonywania programu wynikowego (ang. debugging); **UWAGA!** Opcje -O i -g zwykle nie mogą być stosowane jednocześnie. Jednak kompilator gcc dopuszcza taką możliwość pozwalając na ograniczone śledzenie zoptymalizowanego kodu programu.
-ggdb	debugger **gdb** dostaje dodatkowe informacje (możliwość wykorzystania rozszerzeń GDB) oznacza to użycie najlepszego dostępnego formatu (DWARF2, stabs lub domyślnego, jeżeli nie można użyć żadnego z tych dwóch)
-glevel **-ggdb**level	Poziom debuggowania, standardowo 2. Możliwe opcje: 1,2,3. Dodaje dodatkowe informacje o programie 1 – minimalna, zawiera opis funkcji i zmiennych zewnętrznych, ale nie produkuje informacji o zmiennych lokalnych ani numerach linii
-p	skompiluje plik z informacjami do późniejszego profilowania za pomocą programu prof
-Q	Po skompilowaniu każdej funkcji wypisuje jej nazwę oraz trochę statystyk
-ftime-report **-fmem-report**	Wypisuje info o czasie trwania każdego etapu kompilacji Wypisuje info o zaalokowanej pamięci
-save-temps	Nie kasuje plików tymczasowych, używanych podczas kompilacji, tj. kompilując test.c z opcją -save-temps otrzymamy w bieżącym katalogu pliki test.i, test.s oraz test.o
-time	Raportuje czas procesora (w sekundach), zużyty na każdą część procesu kompilacji

Formaty plików obiektowych

Kompilator i asembler tworzą plik obiektowy, zawierający wygenerowany kod binarny oraz pewne dodatkowe dane. Linker łączy wiele plików obiektowych w jeden. Loader bierze plik obiektowy i ładuje go do pamięci.

Co składa się na plik obiektowy?

- Informacje nagłówkowe: ogólne informacje o pliku, takie jak rozmiar kodu, nazwa pliku źródłowego, data powstania
- Kod wynikowy: instrukcje binarne i dane wygenerowane przez kompilator i asembler
- Relokacja: lista miejsc w kodzie, które muszą zostać poprawione, kiedy linker zmienia adresy kodu wynikowego
- Symbole: symbole globalne, zdefiniowane w danym module, symbole, które mają zostać zaimportowane z innego modułu lub zdefiniowane przez linker
- Informacje do debuggowania: inne informacje o kodzie wynikowym (nie są potrzebne linkerowi, ale przydadzą się podczas debuggowania), mogą to być np. informacje o kodzie źródłowym, numerach linii, zmiennych lokalnych, strukturach danych, używanych przez kod wynikowy

Niektóre formaty plików obiektowych zawierają więcej informacji, niektóre zawierają ich mniej.

Projektowanie formatu pliku obiektowego:
Plik może być:

- linkowalny (linkable) – używany jako input dla linkera
- wykonywalny (executable) – może być ładowany do pamięci i uruchamiany jako program
- ładowalny (loadable) – może być ładowany do pamięci jako biblioteka razem z programem
- wszelkie możliwe kombinacje trzech powyższych

Format pliku typu NULL (null object format): MS-DOS pliki .COM

Jest to najbardziej znany przykład formatu pliku, który nie posiada żadnych dodatkowych informacji poza kodem binarnym. Jedyne co ma do zrobienia

system operacyjny podczas uruchamiania pliku typu .COM jest załadowanie pliku do segmentu pamięci, ustawienie rejestrów (m.in. wskaźnika stosu na koniec segmentu, pozostałych rejestrów na początek segmentu) i przeskoczenie do początku załadowanego programu.

Każdy program zapisany w tym formacie musi mieścić się w jednym segmencie pamięci, czyli zajmować maksymalnie 64KB. Podczas uruchamiania programów tego typu tworzony jest nagłówek PSP (Program Segment Prefix), a zaraz za nim wczytywany jest cały program. Blok PSP znajduje się pod adresem: CS:0000-CS:0100, a zaraz za nim kod programu (CS:0100:CS:????).

Dzięki segmentacji architektury x86 takie rozwiązanie działa. Jeżeli program mieści się w jednym segmencie, to sprawa jest rozwiązana, jeżeli natomiast nie to istnieje potrzeba obliczania adresów względnych, co spoczywa na rękach programisty. Sytuacja ta jest dosyć prosta do rozwiązanie i dosyć łatwa do zautomatyzowania, a zajmować się tym będą już linkery i loadery w kolejnym MS-DOSowym formacie plików, czyli formacie .EXE.

UNIX-owe pliki a.out

Nagłówek pliku:

```
int a_magic;     // numer magiczny
int a_text;      // długość kodu
int a_data;      // długość danych
int a_bss;       // wielkość danych niezainicjowanych
int a_syms;      // wielkość tablicy relokacji
int a_entry;     // punkt wejścia do programu
int a_trsize;    // wielkość tablicy relokacji dla kodu
int a_drsize;    // wielkość tablicy relokacji dla danych
```

Ładowanie i uruchamianie plików a.out:

NMAGIC (1975-1990s)

- przeczytanie nagłówka w celu wydobycia informacji o rozmiarach segmentu
- sprawdzenie, czy istnieje dzielony segment dla tego pliku, jeżeli tak to mapujemy ten segment do przestrzeni adresowej procesu, jeżeli nie ma to zapisujemy do nowego segmentu

- utworzenie prywatnego segmentu dla danych i BSS, zmapowanie go do procesu oraz wyzerowanie BSS
- utworzenie i zmapowanie segmentu dla stosu i umieszczenie argumentów wywołania programu na stosie
- ustawienie rejestrów i skok do początkowego adresu

ZMAGIC

- redukuje zbędne stronicowanie, kosztem przestrzeni dyskowej
- nagłówek ma tylko 32 bajty, a jest mu przydzielane aż 4kB
- dziura między danymi kodem a danymi wynosi średnio 2kB (połowa rozmiaru strony)

QMAGIC

- ZMAGIC z poprawionymi niedoskonałościami

MS-DOSowe pliki EXE

Pliki typu EXE są poprzedzone specjalnym nagłówkiem. W nagłówku tym jest zapisany punkt startu programu, wielkość programu oraz inne (bardzo ważne) informacje. Poniżej w tabeli przedstawiony został typowy nagłówek pliku EXE:

Offset	Nazwa	Rozmiar	Zawartość
0	char signature[2]	2	4DH i 5AH - symbol pliku typu EXE ('MZ' lub 'ZM')
2	short lastsize	2	Długość ostatniej strony (strona 512 bajtów)
4	short nblocks	2	Długość programu razem z nagłówkiem w stronach 512 bajtowych
6	short nreloc	2	Liczba elementów w tablicy przemieszczeń, czyli liczba 4-bajtowych rekordów, znajdujących się w niesformatowanej części nagłówka
8	short hdrsize	2	Rozmiar nagłówka w paragrafach 16 bajtowych
AH	short minalloc	2	Minimalna pamięć potrzebna ponad program (w paragrafach)
CH	short maxalloc	2	Maksymalna pamięć potrzebna ponad program (w paragrafach)
EH		2	Przesunięcie segmentu SS (do ustalania SS na początku programu)
10H	void far *sp	2	Wartość rejestru SP na początku procesu
12H	short checksum	2	Suma kontrolna (zanegowana suma wszystkich bajtów w pliku), nie używana przez DOS

14H	void far *ip	2	Wartość rejestru IP (wskaźnika instrukcji) na początku procesu
16H		2	Przesunięcie segmentu CS (do ustalania na początku procesu)
18H	short relocpos	2	Początek tablicy przemieszczeń - adres pierwszej pozycji tablicy relokacji w stosunku do początku pliku. Jeżeli pole jest równe 40h lub więcej, jest to nowy plik EXE
1AH	short noverlay	2	Numer nakładki (0 - dla modułu bazowego)
1CH	char extra[]		Inne informacje - obszar ten jest wykorzystywany przez MS WINDOWS do pamiętania ikon związanych z programem
18H]	void far *relocs[]	4*[6]	Tablica przemieszczeń (typu offset,segment)
?		?	Wypełnienie zerami do końca paragrafu
?		?	Początek kodu programu

Ładowanie i uruchamianie plików EXE:

(jest tylko trochę bardziej skomplikowane niż ładowanie plików .COM)

- wczytanie nagłówka pliku (sprawdzenie numeru magicznego)
- znalezienie odpowiedniego miejsca w pamięci (pola minalloc i maxalloc mówią ile dodatkowej pamięci trzeba zaalokować na końcu załadowanego programu)
- utworzenie PSP (Program Segment Prefix) na początku programu
- wczytanie kodu do pamięci bezpośrednio za PSP
- adres zapisany w kodzie jest adresem względnym w danym segmencie i razem z plikiem trzymana jest tablica relokacji (relocpos), czyli spis wszystkich miejsc, które trzeba poprawić przy relokacji danego segmentu. Przy ładowaniu we wszystkich miejscach wskazanych w tablicy relokacji dodawana jest liczba o jaką przesunięty jest segment
- ustawiamy wskaźnik stosu na sp i przeskakujemy do ip w celu wykonania programu

Jest to opis pliku tzw. old executables, czyli pliku EXE dla systemu DOS. Istnieją jeszcze pliki new executables działające w środowiskach wykorzystujących tryb chroniony.

Pliki new executables mają inną budowę od poprzednich. Na ich początku znajduje się krótki programik działający w systemie DOS, tzw. STUB. Jego zadaniem jest informowanie użytkownika, że plik zawiera program nie działający w systemie DOS, lub próbuje uruchomić zawartego w pliku

właściwego programu dla trybu chronionego pod kontrolą odpowiedniego środowiska, np. WINDOWS lub DOS4GW.

Program dla trybu chronionego posiada również nagłówek. Różne systemy mają różne struktury tego nagłówka. By poznać dla jakiego systemu jest dany plik trzeba najpierw sprawdzić, czy na pozycji 18h-19h nagłówka starego typu znajduje się wartość 40h lub większa. Jeśli tak, to należy odczytać ewentualny nagłówek nowego EXE, w którym znajduje się odpowiedni znacznik (dwa znaki). Mamy następujące znaczniki:

Znacznik	System operacyjny
NE	Windows lub OS/2 1.x, z podziałem na segmenty
LE	Windows virtual device driver (VxD) z liniowym adresem (Linear Executable)
LX	Wariant LE, używany przez OS/2 2.x
W3	Plik WIN386.EXE dla Windows; kolekcja plików LE
PE	Windows NT lub Win32s (Portable Executable)
DL	HP 100LX/200LX (pliki *.EXM)
MP	Stare pliki PharLap (pliki *.EXP)
P2	PharLap 286 (pliki *.EXP)
P3	PharLap 386 (pliki *.EXP)

UNIX-owe pliki ELF (Executable and linking Format)

Tradycyjny format a.out był używany w UNIX-ie przez ponad dekadę, aż w końcu z nadejściem Systemu V, AT&T zdecydowało, że potrzebny jest format, który w lepszym stopniu będzie wspierał kompilacją skrośną, dynamiczne łączenie oraz inne nowoczesne cechy systemu. Wczesne wersje Systemu V używały formatu COFF (Common Object File Format), jednak bez pewnych rozszerzeń nie wspierał on C++ i dynamicznego łączenia. W późniejszych wersjach COFF został zastąpiony formatem ELF. Format ELF współpracuje z formatem DWARF, który to jest formatem wspomagającym debuggowanie.

Pliki ELF mogą istnieć w trzech formatach: relokowalnym, wykonywalnym i jako obiekty dzielone. Relokowalne pliki są tworzone przez kompilatory i asemblery, ale przed uruchomieniem muszą jeszcze zostać zlinkowane. Obiekty dzielone to po prostu biblioteki, które zawierają zarówno informacje o symbolach dla linkera jak i kod, który może być wykonywany.

Nagłówek pliku ELF:

```
char magic[4] = "\177ELF";   // magic number
char class;                  // address size, 1 = 32 bit, 2 = 64 bit
char byteorder;              // 1 = little-endian, 2 = big-endian
char hversion;               // header version, always 1
char pad[9]; short filetype; // file type: 1 = relocatable, 2 = executable,
 // 3 = shared object, 4 = core image
short archtype;              // 2 = SPARC, 3 = x86, 4 = 68K, etc.
int fversion;                // file version, always 1
int entry;                   // entry point if executable
int phdrpos;                 // file position of program header or 0
int shdrpos;                 // file position of section header or 0
int flags;                   // architecture specific flags, usually 0
short hdrsize;               // size of this ELF header
short phdrent;               // size of an entry in program header
short phdrcnt;               // number of entries in program header or 0
short shdrent;               // size of an entry in section header
short phdrcnt;               // number of entries in section header or 0
short strsec;            // section number that contains section name strings
```

Relokowalne pliki ELF

Sam plik obiektowy jest kolekcją sekcji. Każda sekcja zawiera informacje, takie jak kod programu, dane do odczytu, dane do odczytu/zapisu, tablice relokacji.

Nagłówek sekcji:

```
int sh_name;  // name, index into the string table
int sh_type;  // section type
int sh_flags;  // flag bits, below
int sh_addr;  // base memory address, if loadable, or zero
int sh_offset;  // file position of beginning of section
int sh_size;  // size in bytes
int sh_link;  // section number with related info or zero
int sh_info;  // more section-specific info
int sh_align;  // alignment granularity if section is moved
int sh_entsize;  // size of entries if section is an array
```

Rodzaje sekcji:

- .text
- .data
- .rodata
- .bss
- .rel.text, .rel.data i .rel.rodata (plus dane o relokacji)
- .init i .fini (kod, jaki ma być wykonany, kiedy program zaczyna/kończy działanie)
- .symtab i .dynsym (tablice symboli i symbole do dynamicznego linkowania)

- .strtab i .dynstr (tablice napisów z np. nazwami sekcji, nazwami dynamicznie ładowanych symboli)

Wykonywalne pliki ELF

Wykonywalne pliki ELF mają generalnie taki sam format jak pliki relokowalne, ale dane w nich są tak umieszczone, że plik może zostać zmapowany do pamięci i uruchomiony.

Za nagłówkiem pliku znajduje się kolejny nagłówek, tym razem nagłówek programu.

Nagłówek programu:

```
int type; // loadable code or data, dynamic linking info, etc.
int offset; // file offset of segment
int virtaddr; // virtual address to map segment
int physaddr; // physical address, not used
int filesize; // size of segment in file
int memsize; // size of segment in memory (bigger if contains BSS)
int flags; // Read, Write, Execute bits
int align; // required alignment, invariably hardware page size
```

Nagłówek programu definiuje segmenty, które mają zostać zmapowane.

Podsumowanie formatu plików ELF

ELF jest w miarę złożonym formatem, ale wszystkie swoje zadania spełnia w miarę dobrze. Jest na tyle elastyczny, aby móc obsługiwać programy napisane w C++, a z drugiej strony jest dosyć wydajny, jeżeli chodzi o systemy z pamięcią wirtualną i dynamicznym linkowaniem. Pozwala też na kompilację skrośną i skrośne linkowanie z jednej platformy na drugą z wystarczającą ilością informacji w każdym pliku ELF, aby zidentyfikować architekturę docelową oraz porządek bajtów.

Zarządzanie symbolami

Zarządzanie symbolami to główna funkcja linkera.

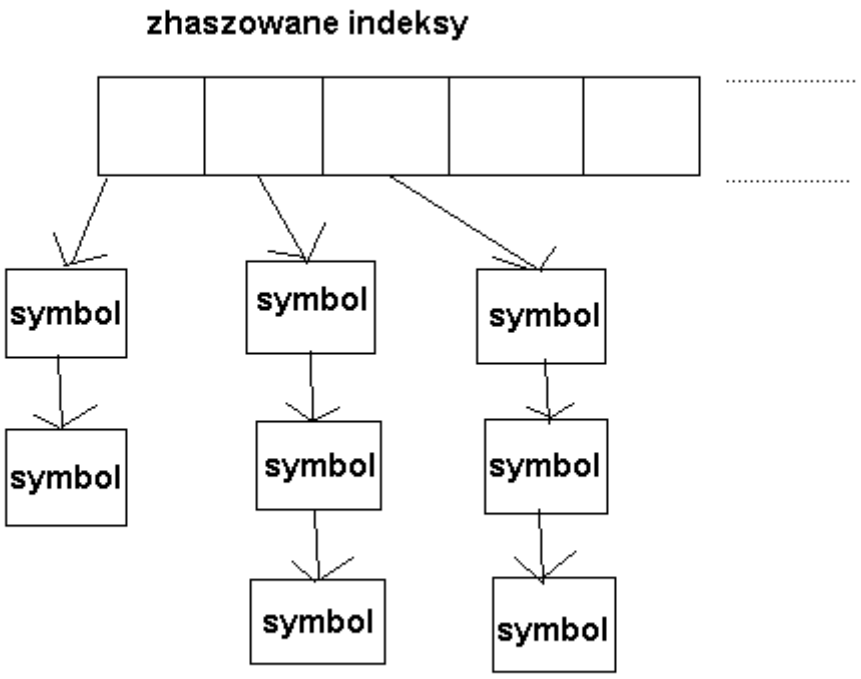
Linkery zarządzają wieloma różnymi rodzajami symboli oraz referencjami symboli pomiędzy poszczególnymi modułami. Każdy taki moduł zawiera swoją tablicę symboli :

- symbole globalne zdefiniowane w module
- symbole globalne zewnętrzne (nie są zdefiniowane w module, jednak w module są do nich odwołania)
- nazwy segmentów
- symbole nieglobalne, głównie na użytek debuggerów i zrzutów w razie wywalenia się programu – nie są one niezbędne do procesu linkowania (opcjonalne)
- informacje o numerach linii, pokazują zależność między kodem źródłowym a kodem wynikowym (opcjonalne)

Formaty symbol table (stabs)

Tablice symboli dla linkerów są podobne do tych dla kompilatorów, a nawet trochę prostsze, jako że jest mniej rodzajów symboli, które musi obsługiwać linker niż kompilator. W jednej tablicy trzymane są informacje o plikach wejściowych oraz modułach bibliotek, kolejna tablica to tablica zmiennych globalnych, które linker musi przeanalizować w plikach wejściowych. Trzecia tablica może trzymać dodatkowe symbole używane do debbugowania.

Tablica symboli jest najczęściej trzymana jako tablica wskaźników, indeksowana funkcją haszującą. Dawniej, linkery obsługiwały tylko krótkie nazwy (od 2 znaków na najprostszych komputerach, 6 na komputerach HP, do 8 znaków na maszynach IBM i wczesnych systemach UNIX-owych). Współczesne linkery wspierają dużo dłuższe nazwy, nie tylko dlatego, że programiści używają coraz dłuższych nazw, ale też dlatego, że kompilatory przekształcają nazwy na dłuższe, kodując w nich dodatkowe informacje.
(ćwiczenie.)



Tablice modułów

Linkery muszą śledzić każdy moduł podczas ładowania, a używają do tego tablicy modułów.

Przykład: format a.out

Tablica modułu dla formatu a.out

```
char *filename; /* Name of this file. */

char *local_sym_name; /* Describe the layout of the contents of the file */

struct exec header; /* The file's a.out header. */

int symseg_offset; /* Offset in file of debug symbol segment or 0 if there is none*/

struct nlist *symbols; /* Symbol table of the file. */

int string_size; /* Size in bytes of string table. */

char *strings; /* Pointer to the string table. */

struct relocation_info *textrel; /* Text and data relocation info */

struct relocation_info *datarel; /* Relation of this file's segments to the output
file */

int text_start_address; /* Start of this file's data seg in the output file core
image. */


int data_start_address; /* Start of this file's text seg in the output file core
image. */

int bss_start_address; /* Start of this file's bss seg in the output file core
image. */

int local_syms_offset; /* Offset in bytes in the output file symbol table
of the first local symbol for this file. */
```

Ponieważ większość istotnych informacji znajduje się w nagłówku pliku, to tablica trzyma po prostu kopię nagłówka plus wskaźniki do tablicy napisów i tablicy relokacji.

Przechowywanie informacji dla debuggowania

Wszystkie współczesne kompilatory wspierają debuggowanie. Oznacza to, że programista może debuggować kod wynikowy odwołując się do nazw zmiennych i funkcji z kodu źródłowego, może także dowolnie ustawiać punkty kontrolne. Jest to możliwe dzięki zmapowaniu do kodu wynikowego numerów linii z kodu źródłowego, a także trzymaniu informacji o nazwach wszystkich zmiennych, funkcji, typów oraz struktur użytych w programie.

Informacje o numerach linii

Wszystkie debuggery muszą być w stanie dokonać mapowania pomiędzy adresami w programie a numerami linii w kodzie źródłowym. Pozwala to programiście na ustawianie punktów kontrolnych.

Mapowanie takie nie jest trudne do przeprowadzenia poza jednym wyjątkiem. Problemy pojawiają się gdy kompilator stosuje optymalizację (zmianę kolejności w kodzie wynikowym, niezgodną z kodem źródłowym). Rozwiązanie (w formacie DWARF) polega na mapowaniu każdego bajtu z kodu wynikowego na numer linii w kodzie źródłowym, co oczywiście powoduje duże zużycie pamięci. Inne formaty po prostu generują przybliżone numery linii.

Informacje o symbolach i zmiennych

Informacja o symbolach jest przechowywana w postaci drzewa. W korzeniu znajduje się lista typów, zmiennych i funkcji zdefiniowanych na najwyższym poziomie, w poddrzewach są trzymane pola struktur czy też zmienne zdefiniowane w funkcjach.

Kompilatory UNIX-owe posiadają dwa różne formaty wspomagające debuggowanie: stab (symbol table) używany głównie w a.out, COFF i plikach ELF nie związanych z Systemem V, oraz drugi format DWARF, zaprojektowany dla plików ELF Systemu V. Microsoft zdefiniował własny format CV4 (najnowsza wersja) dla swojego debuggera Codeview.

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

Odpluskwianie błędów pamięci

Pisząc programy często przytrafiają się nam sytuacje, gdy kompilacja przebiega bezproblemowo, uruchomienie programu wskazuje jakby wszystko było w najlepszym porządku i już ocieramy pot z czoła zadowoleni z ukończenia mozolnej pracy, już wstajemy po wielogodzinnym posiedzeniu przed komputerem aż tu nagle .. Coś nie gra!!

Co wtedy? Jak tu przeglądać kod, często długi i zabałaganiony? Często zmuszeni jesteśmy pisać wszystko od nowa, a w najlepszym przypadku kończymy odpluskwianie po wielogodzinnej pracy.

Warto wiedzieć o istnieniu programów wspomagających odpluskwianie i wykrywanie wycieków pamięci, oraz potrafią z nich skorzystać.

Przegląd narzędzi do kontroli pamięci:

- Electric Fence
- Checker
- mpr
- Mpatrol
- LeakTracer
- NJAMD
- MEMWATCH
- YAMD
- Borland Optimizeit Profiler for the Microsoft® .NET Framework
- Borland® Optimizeit™ Suite 5.5
- valgrind
- memprof
- dmalloc
- Insure++ z Parasoft (http://www.parasoft.com). (komercyjne)
- libgc6

Omówienie narzędzi do kontroli pamięci:

Przykład 1. Niepoprawny program deb

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char global[5];

int main(void)
{
  char * dyn;
  char local[5];
  /* nieznacznie przekrocz rozmiar bufora */
  dyn = malloc(5);
  strcpy(dyn, "123456");
  printf("1: %s\n", dyn);
  free(dyn);
  /* znacznie przekrocz rozmiar bufora */
  dyn = malloc(5);
  strcpy(dyn, "12345678");
  printf("2: %s\n", dyn);
  /* wyjdź poza zarezerwowaną pamięć */
  *(dyn - 1) = '\0';
  printf("3: %s\n", dyn);
  /* nie zwalniam pamięci */
  /* wyjdź poza zmienną lokalną */
  strcpy(local, "123456");
  printf("4: %s\n", local);
  local[-1] = '\0';
  printf("5: %s\n", local);
  /* zaatakuj przestrzeń globalną */
  strcpy(global, "123456");
  printf("6: %s\n", global);
  global[-1] = '\0';
  printf("7: %s\n", global);
  exit(EXIT_SUCCESS);
}
```

Uruchomienie i wyniki działania programu:

```
gcc -Wall -o deb deb.c
./deb
1: 123456
2: 12345678
3: 12345678
4: 123456
5: 123456
```

6: 123456
7: 123456

Powyżej został przedstawiony program który nie jest poprawny, mimo iż wydaje się takim być na podstawie wyników wykonania.

Electric Fence

Właściwości:

- Wykrywanie przepełnień buforów
- Niemożność wykrycia wycieków pamięci
- Program nie pomaga w rozwiązywanie problemów z lokalnymi i globalnymi obszarami danych.

Sposób działania:

Program zastępuje funkcję malloc() z biblioteki C, nową wersją, która:

- Rezerwuje żądany obszar pamięci i następujący po nim(przed nim) fragment pamięci.
- Do dodatkowego fragmentu pamięci proces nie ma dostępu próba odwołania się do tego obszaru powoduje zatrzymanie procesu przez jądro z błędem ochrony pamięci.

Podobnie zastępuje free(), zmieniona free() blokuje dostęp do zwolnionych obszarów pamięci i raportuje błąd przy próbie odwołania do takowego.

- Jeżeli EF_PROTECT_BELOW=1 - program wykrywa próby dostępu przed początkiem zajętego bufora
- Jeżeli EF_PROTECT_FREE=1 { funkcja free() nie będzie zwalniać przekazanego obszaru pamięci, a uczyni go niedostępnym. W efekcie jądro będzie wykrywać próby dostępu do tego obszaru.

Więcej informacji: man libefence

Użycie Electric Fence:

Do kodu programu należy dodać bibliotekę: libefence.a :
gcc -o program zrodlo.c -lefence

Wtedy wykonanie programu z przykładu 6 da mniej więcej taki wynik:

```
./deb

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
1: 123456
Naruszenie ochrony pamięci
```

Jak można zauważyć, program informuje o błędzie, ale nie wskazuje dokładnie miejsca jego wystąpienia. Błąd ten łatwo można wyśledzić za pomocą programu śledzącego gdb.

W tym celu kompilację należy przeprowadzić następująco:
gcc -ggdb -Wall -o program zrodlo.c –lefence

Komentarz:

- W przykładowym programie pierwsze przepełnienie nie zostanie wykryte. Spowodowane jest to zjawiskiem "wyrównania pamięci". Implementacje funkcji malloc() zwracają obszar pamięci, którego pierwszy bajt jest wyrównywany do rozmiaru słowa (8 bajtów dla procesorów 64 bitowych). Domyślnie Electric Fence dostarcza funkcję malloc() , która zwraca adresy będące parzystą wielokrotnością sizeof(int).
- Aby wykryć pierwsze przepełnienia należy zmiennej środowiskowej EF_ALIGMENT nadać wartość 1.

Checker

- Pomaga w wyszukiwaniu przepełnień buforów i wycieków pamięci
- Zamiast kompilatora gcc należy wykorzystywać program:
  checkergcc
- Wynikowy program sam będzie śledził swoje wykonanie
- Kompilacja:
  checkergcc -o program zrodlo.c

Wyszukiwanie wycieków pamięci:

- Działanie programu sterowane jest zmienną CHECKEROPTS --detector=end (wykrywacz wycieków pamięci należy uruchomić po zakończeniu programu)

Wyszukiwanie przepełnień

- Uruchomienie programu skompilowanego za pomocą checkergcc spowoduje wygenerowanie raportu o wszystkich błędach
- Aplikacja pomaga w poszukiwaniu prób pisania poza końcem stosu
- Jeżeli po zmiennej buf zostanie zarezerwowana jakaś inna zmienna to checker nie pomoże w wykryciu przepełnienia zmiennej buf
- Program nie umożliwia wykrycia problemów ze zmiennymi globalnymi

Program mpr

- Program rejestruje wszystkie wywołania funkcji malloc() i free()
- Na podstawie dopasowania każdego zarezerwowania pamięci i jej zwolnienia wyszukiwane są wycieki pamięci
- Program pomaga w wyszukiwaniu naruszeń obszarów pamięci
- mpr zastępuje funkcję malloc() swoją własną wersją (dokonuje się tego za pomocą funkcji mcheck())
- Do kodu programu należy dołączyć bibliotekę libmpr.a
  gcc -ggdb -o program zrodlo.c -lmpr
- Po kompilacji, sprawdzanie pamięci za pomocą mcheck() włączane jest automatycznie

Wyszukiwanie naruszeń obszarów pamięci

- Program umieszcza znaną sekwencję bajtów przed i za rezerwowanym obszarem
- Funkcja free() szuka tych sygnatur i jeżeli zostały naruszone wywołuje funkcję abort()
- Metoda mcheck() nie lokalizuje dokładnie pojawienia się naruszenia
- Znalezienie błędu jest możliwe za pomocą programu śledzącego gdb

- Program nie pomaga w wyszukiwaniu przepełnień zmiennych lokalnych i globalnych, a tylko obszarów zarezerwowanych funkcją malloc()

Wyszukiwanie wycieków pamięci

- Należy ustawić dwie zmienne środowiskowe MPRPC i MPRFI
- MPRPC
- przechowuje wartości potrzebne do prawidłowego poruszania się po ciągu wywołań funkcji podczas tworzenia raportu
- zmienną ustawia się w następujący sposób: export MPRPC=`mprpc program`
- MPRFI
- instruuje za pomocą jakiego programu ma być generowany raport
- dla małych plików: export MPRFI=`cat > mpr.log`
- dla dużych plików: export MPRFI=`gzip > mpr.log.gz`

Po wygenerowaniu raportu można go przeanalizować za pomocą poleceń:

- mpr program { tłumaczy adresy programu w raporcie na nazwy funkcji i miejsca w kodzie źródłowym }
- mprcc { dla każdego ciągu wywołań który rezerwował pamięć, funkcja wyświetla liczbę wykonanych rezerwacji i całkowitą liczbę zarezerwowanych bajtów }
- mprlk { plik raportu jest analizowany pod kątem wszystkich zarezerwowanych obszarów, które nie zostały zwolnione. Generuje nowy raport zawierający rezerwacje powodujące wycieki pamięci.

mprlk < mpr.log | mpr program

MEMWATCH

MEMWATCH, napisany przez Johan Lindh, jest narzędziem służącym do wykrywania błędów pamięci dla języka C. Poprzez proste dodanie pliku nagłówkowego do kodu i zdefiniowanie MEMWATCH w formule kompilacji gcc, możesz wyśledzić wycieki i inne błędy pamięci w twoim programie. MEMWATCH wspomaga ANSI C, dostarcza log wyników oraz wykrywa błędy: podwójnych zwolnień pamięci, błędnych zwolnień pamięci, nie-zwolnionej pamięci, przepełnień, nie-dopełnień itd.

To jest lista niektórych cech, właściwości dla wersji 2.71:

- wspiera ANSI C
- Loguje do pliku lub funkcji użytkownika za pośrednictwem makra TRACE()
- Odporny na błędy, potrafi naprawić własne struktury danych
- wykrywa błędy podwójnych zwolnień
- wykrywa błędy niezwolnionej pamięci
- wykrywa przekroczenia bufora zarówno górne (overflow) jak i dolne (underflow)
- Może ustawić maksymalną ilość pamięć którą można zaalokować, np. w celu przeprowadzenia testów w warunkach braku pamięci
- dostarcza makra ASSERT(expr) i VERIFY(expr)
- wykrywa niedozwolone zapisy wskaźnika
- Wsparcie Systemu Operacyjnego dla kontroli przydziału pamięci w celu uniknięcia błędów segmentacji
- Generuje statystyki alokacji w programie
- Podstawowe wsparcie dla wątków (zobacz FAQ dla szczegółów)
- Podstawowe wsparcie dla C++ (domyślnie wyłączone, używać ostrożnie!)
- ...itp

Przykład 1. test1.c

```
#include <stdlib.h>
#include <stdio.h>
#include "memwatch.h"

int main(void)
{
  char *ptr1;
  char *ptr2;

  ptr1 = malloc(512);
  ptr2 = malloc(512);

  ptr2 = ptr1;
  free(ptr2);
  free(ptr1);
}
```

Kod w przykładzie 1 alokuje dwa bloki 512-bajtów pamięci i wtedy wskaźnik do pierwszego bloku jest ustawiany na drugi blok. W wyniku czego, adres drugiego bloku jest gubiony i następuje wyciek pamięci(memory leak).

Teraz skompiluj memwatch.c z przykładu 1. Przykładowy makefile:

gcc -DMEMWATCH -DMW_STDIO test1.c memwatch c -o test1

Kiedy uruchamiasz program test1, wyświetlany jest raport "wyciekłej" pamięci.

Przykład 2 pokazuje przykład wyjściowego pliku memwatch.log dla programu test1

```
MEMWATCH 2.67 Copyright (C) 1992-1999 Johan Lindh

...
double-free: <4> test1.c(15), 0x80517b4 was freed from test1.c(14)
...
unfreed: <2> test1.c(11), 512 bytes at 0x80519e4
{FE FE FE FE FE FE FE FE FE FE FE FE .............}

Memory usage statistics (global):
  N)umber of allocations made: 2
  L)argest memory usage : 1024
  T)otal of all alloc() calls: 1024
  U)nfreed bytes totals : 512
```

MEMWATCH wskazuje dokładną linię w kodzie w której nastąpił błąd. Zgłasza jeśli zwalniasz już zwolniony wskaźnik. To samo w przypadku nie-zwolnionej pamięci. Sekcja na końcu logu pokazuje statystyki, włączając jak dużo pamięci zostało zwolnione/nie-zwolnione , jak dużo pamięci zostało użyte i łączną sumę alokacji(zaalokowanej pamięci).

Download: http://www.linkdata.se/sourcecode.html

YAMD

Napisany przez Nate Eldredge, pakiet YAMD wykrywa problemy dynamicznego przydziału pamięci w C i C++. Użycie YAMD na test1.c: Należy usunąć #include memwatch.h z kodu test i wprowadzić drobne zmiany do makefile, jak pokazano poniżej:

gcc -g test1.c -o test1

Przykład 3 pokazuje wynik zastosowania YAMD na test1.

Listing 3. test1 output with YAMD

```
YAMD version 0.32
Executable: /usr/src/test/yamd-0.32/test1
...
INFO: Normal allocation of this block
Address 0x40025e00, size 512
...
INFO: Normal allocation of this block
Address 0x40028e00, size 512
...
INFO: Normal deallocation of this block
Address 0x40025e00, size 512
...
ERROR: Multiple freeing At
free of pointer already freed
Address 0x40025e00, size 512
...
WARNING: Memory leak
Address 0x40028e00, size 512
WARNING: Total memory leaks:
1 unfreed allocations totaling 512 bytes

*** Finished at Tue ... 10:07:15 2002
Allocated a grand total of 1024 bytes 2 allocations
Average of 512 bytes per allocation
Max bytes allocated at one time: 1024
24 K alloced internally / 12 K mapped now / 8 K max
Virtual program size is 1416 K
End.
```

YAMD pokazuje, że już zwolniliśmy pamięć którą próbujemy zwolnić i że jest wyciek pamięci. Spróbujmy YAMD na innym programie w Przykładzie 4.

Przykład 4. test2.c

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

int main(void)
{
  char *ptr1;
  char *ptr2;
  char *chptr;
```

```
  int i = 1;
  ptr1 = malloc(512);
  ptr2 = malloc(512);
  chptr = (char *)malloc(512);
  for (i; i <= 512; i++) {
    chptr[i] = 'S';
  }
  ptr2 = ptr1;
  free(ptr2);
  free(ptr1);
  free(chptr);
}
```

Możesz użyć następującego polecenia aby uruchomić YAMD:
./run-yamd /usr/src/test/test2/test2

Przykład 5 pokazuje wynik użycia YAMD na przykładowym programie test2. YAMD pokazuje nam że wykonanie pętli for spowoduje przekroczenie zaalokowanego bufora.

```
Running /usr/src/test/test2/test2
Temp output to /tmp/yamd-out.1243
*********
./run-yamd: line 101: 1248 Segmentation fault (core dumped)
YAMD version 0.32
Starting run: /usr/src/test/test2/test2
Executable: /usr/src/test/test2/test2
Virtual program size is 1380 K
...
INFO: Normal allocation of this block
Address 0x40025e00, size 512
...
INFO: Normal allocation of this block
Address 0x40028e00, size 512
...
INFO: Normal allocation of this block
Address 0x4002be00, size 512
ERROR: Crash
...
Tried to write address 0x4002c000
Seems to be part of this block:
Address 0x4002be00, size 512
...
Address in question is at offset 512 (out of bounds)
Will dump core after checking heap.
Done.
```

MEMWATCH i YAMD są przydatnymi narzędziami usuwania błędów z programu, wymagającymi odmiennych podejść. Aby skorzystać MEMWATCH, potrzebujesz dodać pliki nagłówkowy memwatch.h i dodać dwie flagi podczas kompilacji. YAMD wymaga tylko opcji -g dla formuły linkującej.

Valgrind

Valgrind jest systemem na licencji GPL przeznaczonym do debuggowania i profilowania programów Linuxowych. Z pomocą narzędzia Valgrind możesz automatycznie wykrywać wiele problemów zarządzania pamięcią, unikając godzin frustrującego "polowania na pluskwy". Możesz też wykonać podany szczegółowe profilowanie, by przyspieszyć programy.

Valgrind jest zaprojektowany, by być bezinterwencyjny jak to tylko możliwe. Pracuje bezpośrednio z istniejącym plikiem wykonywalnym. Nie potrzebujesz rekompilować ponownie, ponownie linkować, albo inaczej modyfikować programu by został sprawdzony. Po prostu umieść komendę:
  valgrind --tool= tool_name
na początku linii poleceń za pomocą której normalnie uruchamiasz program.

Download: http://valgrind.org/

Memcheck – najpopularniejsze narzędzie systemu Valgrind

Memcheck jest silnym narzędziem pakietu Valgrind szukającym błędów pamięci.

Jak korzystać:

Skompiluj program z opcją -g, aby inkludować informacje o debuggowaniu, żeby komunikaty błędów generowane przez **Memcheck** zawierały dokładne numery wierszy(w których błąd wystąpił). Użycie -O0 jest też dobrym pomysłem, jednak spowoduje znaczne spowolnienie pracy programu. Z -O1 numery wiersza mogą być wskazywane niedokładnie, chociaż ogólnie testowany kod działa dobrze. odradza się użycie -O2 i wyższych, gdyż **Memcheck** czasami donosi o nieistniejących błędach.

Uruchom skompilowany program jak następuje:

valgrind --leak-check=yes program arg1 arg2

Oto **przykład** kodu w C w którym następuje „wyciek pamięci”:

```
#include <stdlib.h>

void f(void)
{
   int* x = malloc(10 * sizeof(int));
   x[10] = 0;        // problem 1: heap block overrun
}                    // problem 2: memory leak -- x not freed

int main(void)
{
   f();
   return 0;
}
```

Większość komunikatów jest podobna do tego poniżej, wygenerowanego na powyższym przykładzie.

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (przykład.c:6)
==19182==    by 0x80483AB: main (przykład.c:11)
==19182==  Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (przykład.c:5)
==19182==    by 0x80483AB: main (przykład.c:11)
```

Wiadomości o wyciekach pamięci wyglądają mniej więcej tak:

```
==19182== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (a.c:5)
==19182==    by 0x80483AB: main (a.c:11)
```

Wszystkie zapisy i odczyty pamięci są sprawdzane jak również wszystkie wywołania malloc/new/free/delete. Jako efekt, **Memcheck** może dostrzegać następujące problemy:

- Użycie niezainicjowanej pamięci
- Odczyty i zapisy już zwolnionej pamięci
- Odczyty i zapisy spoza zaalokowanych bloków
- Odczyty i zapisy niewłaściwych obszarów stosu

- Wycieki pamięci - gdy wskaźniki do zaalokowanej pamięci zostały permanentnie zgubione
- Niewłaściwie dobrane użycie malloc/new/new [] w stosunku do free/delete/delete []
- Nachodzące na siebie wskaźniki src (źródła) i dst (celu) w memcpy() i podobnych funkcjach

Addrcheck: lekkie narzędzie kontroli pamięci

Aby użyć tego narzędzia do sprawdzenia programu x:

valgrind --tool=addrcheck x

Addrcheck jest uproszczoną wersją narzędzia Memcheck opisanego powyżej. Jest identyczny z **Memcheck**'em, z wyjątkiem jednego ważnego szczegółu: nie sprawdza problemów związanych z nieokreślonymi-wartościami które to **Memcheck** robi. W ten sposób **Addrcheck** jest szybszy niż **Memcheck** i używa mniej pamięci. **Addrcheck** dostrzega następujące błędy:

- Odczyty i zapisy już zwolnionej pamięci
- Odczyty i zapisy spoza zaalokowanych bloków
- Odczyty i zapisy niewłaściwych obszarów stosu
- Wycieki pamięci - gdy wskaźniki do zaalokowanej pamięci zostały permanentnie zgubione
- Niewłaściwie dobrane użycie malloc/new/new [] w stosunku do free/delete/delete []
- Nachodzące na siebie wskaźniki src (źródła) i dst (celu) w memcpy() i podobnych funkcjach

Profilowanie Kodu.

Najczęściej postrzegane jest jako ostateczna faza usprawniania kodu aplikacji. Patrząc na optymalizację kodu jako kompleksowy proces składający się z kilku faz szybko zauważymy, iż im wcześniejsza faza tworzenia projektu podlega optymalizacji tym lepsze wyniki są uzyskiwane. Nie należy jednak popadać w paranoję i optymalizować każdej linijki kodu. Preferowałbym tutaj raczej holistyczne podejście do tematu i zlokalizowania słabych punktów i tzw. "wąskich gardeł" w odniesieniu do całości projektu. Jeśli zlokalizujemy już problem nie należy od razu go poprawiać tylko przeanalizować następne kroki algorytmu i spojrzeć na problem szerzej. Często rozwiązanie problemu znaleźć można o poziom wyżej. Jedno z najlepszych rozwiązań optymalizacyjnych w formie dowcipu opowiada Robert Lee, którego prace wniosły ogromny wkład w rozwój procesów optymalizacji kodu:

patient who says "Doctor it hurts when I do this, what should I do" to which the doctor replies "Well, don't do that"

co doskonale oddaje jedną z podstawowych zasad optymalizacji - czy naprawdę koniecznie musisz to wykonywać/ z tego korzystać/ to przetwarzać? To jest faza planowania optymalizacji w fazie algorytmicznej. Dopiero kolejnym krokiem jest faza implementacji, w której większość programistów dopiero zaczyna myśleć o optymalizacji. Algorytmy optymalizacyjne opisane w tej pracy bazują na algorytmie zstępującym - od wyższego poziomu abstrakcji do niższego bardziej szczegółowego. Optymalizacja kodu sprawia również, że staje się on bardziej przejrzysty i prostszy. Poza tym, stosując optymalizację algorytmów uzyskuje się spójność implementacji oraz ułatwienie zarządzania kodem.

Sama informacja o tym, że naszemu programowi pod względem szybkości działania bliżej do żółwia niż zająca nie daje nam jeszcze dokładnej informacji w którym miejscu zacząć optymalizację. Trzeba znaleźć winowajcę - czyli określić gdzie program zwalnia i dlaczego. W celu umiejscowienia wąskiego gardła najlepiej posłużyć się jednym z dostępnych na rynku programów profilujących (profilerów). Zadaniem tych programów jest monitorowanie pracy aplikacji i zbieranie informacji na temat prędkości działania i wydajności poszczególnych składników programu. Możliwe jest śledzenie czasu wykonania poszczególnych procedur w podziale na milisekundy lub cykle procesora. Proces profilowania pozwala dokładnie znaleźć fragment kodu najbardziej zasobożerny - w celu jego późniejszej optymalizacji.

Programy wspomagające profilowanie kodu:

- Valgrind
- Polecenie prof, gprof
- Borland Optimizeit Profiler for the Microsoft® .NET Framework
- Borland® Optimizeit™ Suite 5.5
- BORLAND® OPTIMEZIT™ ENTERPRISE SUITE 5.5 FOR JAVA™

Omówienie niektórych programów do profilowania kodu.

Valgrind

System Valgring dostarcza również różnych narzędzi do profilowania kodu takich jak:

- **Lackey** – prosty profiler
- **Cachegrind -** profiler cache'u
- **Massif** – profiler zużycia pamięci

Lackey – prosty profiler

Lackey jest prostym narzędziem wchodzącym w skład systemu **Valgrind**, które wykonuje podstawowe pomiary programu. Dodaje sporo prostego oprzyrządowania do kodu programu. W zamierzeniu, ma by być przydatnego jako wzorcowe narzędzie.

Użycie:

valgrind --tool=lackey program

Pomiary i raporty:

1) Liczba odwołań do _dl_runtime_resolve(), funkcji z dynamicznego linkera glibc, która rozdziela odwołania funkcji do współdzielonych obiektów. Możesz zmienić nazwę funkcji z wiersza polecenia --fnname=<nazwa>.
2) Liczba napotkanych skoków warunkowych oraz liczba i statystyki ich wykonań.
3) Statystyki ilości pracy podczas wykonania programu:

a) Liczba podstawowych bloków użytych i wypełnionych przez program. Informacja dążąca do optymalizacji wykonana przez JIT, to nie jest dokładna wartość.
b) Numer gościa (x86, amd64, ppc, etc.) wykonane instrukcje i formuły IR. IR jest RISC dla Valgrind - warstwą pośrednią przez którą wszystkie środki są robione.
c) Relacje między otrzymanymi wynikami.
d) Kiedy opcja --detailed-counts=yes jest włączona, wykaz jest zawiera liczbę załadowań, i operacji ALU i na składzie dla różnych typów argumentów. Typy są zidentyfikowane przez ich nazwę-IR ("I1" ... "I128", "F32", "F64" i "V128").

4) Kod zakończenia programu klienta.

Zanotuj, że **Lackey** działa całkiem powoli, szczególnie kiedy jest włączona opcja: --detailed-counts=yes. Można by sprawić by działał znacznie szybciej przez robienie nieznacznie bardziej wyrafinowanej pracy oprzyrządowania, ale to podkopałoby jego rolę jako narzędzia do prostych przykładów.

Cachegrind: profiler cache'u

Użycie:

valgrind --tool=cachegrind program

Cachegrind jest narzędziem do przeprowadzania symulacji cache'u sporządzania komentarzy twojego kodu linia po linii z liczbą uchybień pamięci podręcznej. Rejestruje:

- Instrukcje czytania i uchybienia w pamięci cache L1.
- Odczyty i brakujące odczyty, zapisy i brakujące zapisy L1 cache data.
- Zunifikowane odczyty i brak odczytów, zapisy i brak zapisów pamięci cache L2.

Na nowych maszynach, błędna obsługa L1 może kosztować około 10 cykli zegara, a L2 może kosztować mniej więcej 200 cykli. Szczegółowe profilowanie może być bardzo przydatne dla ulepszania twojego programu.

Ponadto, ponieważ jedna instrukcja odczytu pamięć podręcznej jest przygotowana podczas wykonania instrukcji, możesz odkryć jak wiele instrukcji jest wykonanych na linię kodu, która to informacja może być przydatna dla tradycyjnego profilowania i testów pokrycia.

Massif – profiler zużycia pamięci

Użycie:

valgrind --tool=massif program

Massif jest narzędziem profilującym stos, mierzy jak dużo użycia pamięci stosu zużywa program. W szczególnie, dostarcza informacji o:

- Blokach sterty(heap);
- Blokach administracyjnych sterty(heap);
- Wielkości stosu;

Profilowanie sterty(heap profiling) jest przydatne, aby pomóc zmniejszasz ilość pamięci z jaką korzysta program. Na nowych maszynach z pamięcią wirtualną daje następujące korzyści:

- Może przyspieszać twój program - mały program lepiej współdziała lepiej z cashe'm i pozwala uniknąć stronicowania.
- Jeśli twój program używa dużo pamięci, profilowanie zmniejszy szansę na to, że wyczerpana zostanie pamięć wymiany(swap).

Ponadto, są pewne wycieki pamięci, które nie są dostrzegane przez tradycyjne kontrolery wycieków, takie jak **Memcheck**. Kiedy pamięć nie jest tracona (tyle że pozostaje wciąż wskaźnik do obszaru) ale nie jest w użyciu. Programy, które mają wycieki tego typu mogą niepotrzebnie powiększać ilość używanej pamięci, której nie potrzebują.

Przykładowe wyniki profilowania:

Program wykona się wolniej, po czym zostaną wyświetlone statystyki jak niżej:

```
== 27519 == Total spacetime:   2,258,106 Pani.B
== 27519 == Heap:              24.0 %
== 27519 == Heap admin:        2.2 %
== 27519 == Stack(s):          73.7 %
```

Wszystkie pomiary są zrobione w "pamięcioczasie", tzn. pamięć(w bajtach) wymnożoną przez czas (w milisekundach). Zauważ, że, ponieważ **Massif**

zwalnia program, osiągane wartości są nietrafione, gdyż normalnie program wykonałby się tylko, w związku z czym interesujące są tylko względne wartości.

Które wejścia widzisz w złamaniu zależy od opcji wiersza zlecenia dany. Powyższy przykład mierzy wszystkie możliwe części pamięci:

Heap: liczba słów (words) zaalokowana na stercie, przez malloc(), nw i new [].

Heap admin: każdy zaalokowany blok sterty wymaga jakichś danych administracyjnych, które umożliwiają alokatorowi przechowywać pewne informacje o bloku. Łatwo o tym zapomnieć, ale jeśli twój program zajmuje dużo małych bloków, to może spowodować sporą stratę pamięci. Ta wartość jest oszacowaniem przestrzeni zajętej na administrację.

Stack(s): "pamięcioczas" użyty przez stosy programów (programy mogą mieć wielokrotne stosy, zawiera stosy signal hendlera).

Polecenie prof, gprof - dokładna analiza efektywności programu

- prof (gprof) drukuje raport do pliku z śledzenia przebiegu programu { należy dołączyć opcje -p podczas kompilacji:
  cc -p -o program program.c
- Dane z monitorowania zapisane są w pliku mon.out (gmon.out
- dla gpof)
- Informacje generowane przez prof są zupełnie innego rodzaju niż te generowane przez programy śledzące

Przykład. Analiza pliku monitorowania przez gprof

Flat profile:
Each sample counts as 0.01 seconds.
no time accumulated
% cumulative self self total
time seconds seconds calls Ts/call Ts/call name
0.00 0.00 0.00 1 0.00 0.00 sort
Call graph

granularity: each sample hit covers 4 byte(s) no time propagated
index % time self children called name
0.00 0.00 1/1 main [11]
[1] 0.0 0.00 0.00 1 sort [1]

Index by function name
[1] sort

Borland Optimizeit Profiler for the Microsoft® .NET Framework

Borland Optimizeit Profiler dla środowiska Microsoft .NET zaprojektowany został po to, by ułatwić programistom wykrywanie problemów z wydajnością i niezawodnością w kodzie zarządzanym przez .NET. Jako narzędzie kluczowe dla zarządzania ryzykiem związanym z wydajnością i niezawodnością, **Optimizeit Profiler** ułatwia programistom efektywne znajdywanie, hierarchizację i rozwiązywanie wszelkich problemów w tym względzie, jak np. wycieków pamięci, nadmierną alokację obiektów i "wąskie gardła" przepływu informacji.

Borland® Optimizeit™ Profiler dla platformy Microsoft® .NET umożliwia dostarczanie wydajnych aplikacji oraz skrócenie czasu ich tworzenia

Kluczowe cechy (czyli pogadanka):

- dostarczanie szybkiego, niezawodnego i skalowalnego kodu na platformę .NET gotowego do wdrożenia
- minimalizowanie nieoczekiwanych zagrożeń dla wydajności, mogących opóźniać dostarczanie aplikacji na rynek (poprzez m.in.: śledzenie wykorzystania pamięci i czasu procesora)
- obniżenie całkowitych kosztów posiadania oprogramowania oraz maksymalne wykorzystanie wydajności kodu na platformę .NET
- pomyślna migracja na platformę .NET, spełnienie oczekiwań wydajnościowych i zwiększenie produktywności programistów
- Jest to wygodne i miłe w użyciu graficzne narzędzie do debuggowania I profilowania(przede wszystkim)

Borland® Optimizeit™ Suite 5.5

Borland® Optimizeit™ Suite 5.5 for Java™ jest zestawem trzech zintegrowanych narzędzi: profilatora (**Optimizeit Profiler**), debuggera (**Optimizeit Thread Debugger**) oraz analizatora pokrycia kodu (**Optimizeit Code Coverage**). Zadaniem tego zestawu jest ułatwienie programistom uchwycenia tych elementów kodu w języku Java, które mają decydujący wpływ na niezawodność i wydajność aplikacji. Szybkie tworzenie niezawodnych aplikacji dla biznesu wymaga zastosowania środków minimalizujących ryzyko ich błędnego lub nieefektywnego działania. **Optimizeit Suite for Java** umożliwia szybkie wykrywanie i ustalanie hierarchii czynników tego ryzyka, dostarczając narzędzia dla analizy m.in. wycieków pamięci, "wąskich gardeł" przepływu informacji czy konfliktów i błędów synchronizacji wynikających z pracy wielowątkowej. **Optimizeit Suite** zapewnia ponadto zdalną łączność z procesem, ścisłą integrację ze środowiskiem IDE, jak również pełną integrację z popularnymi serwerami aplikacji platformy J2EE. Łącząc w sobie łatwość obsługi, skalowalność i dużą funkcjonalność

BORLAND® OPTIMEZIT™ ENTERPRISE SUITE 5.5 FOR JAVA™

Optimizeit™ Enterprise Suite 5.5 łączy pakiety Optimizeit™ Suite i **Optimizeit™ ServerTrace Developer Edition** w kompleksowe rozwiązanie do zarządzania wydajnością, które pozwala rozwiązywać potencjalne problemy z wydajnością, poczynając od poziomu kodu Javy do "wąskich gardeł" J2EE na poziomie całej aplikacji Dane dotyczące wydajności są gromadzone i wyświetlane w czasie rzeczywistym niemal bez wpływu na działanie testowanej aplikacji źródła problemów z wydajnością można lokalizować z dokładnością do pojedynczego wiersza kodu.

Komponenty pakietu wraz z krótkim opisem:

- **Optimizeit™ Profiler -** profilowanie pamięci i procesora. Analiza wydajności maszyny wirtualnej pozwala stwierdzić, czy przyczyną problemu jest procesor, pamięć, czy oba te czynniki
- **Automatic Memory Leak Detector (AMLD)** śledzi ewolucję wykorzystania pamięci, szybko identyfikując wzrosty zużycia pamięci spowodowane wyciekami. Monitorowanie alokacji w czasie rzeczywistym pomaga zrozumieć, jak profilowany program korzysta z pamięci wirtualnej.

- **CPU Profiler** mierzy czyste wykorzystanie procesora albo zużycie czasu podczas sesji profilowania i wyświetla czas spędzony na wykonywaniu każdej metody albo wiersza kodu
- **Optimizeit™ Thread Debugger** - wykrywanie problemów z wątkami. Czytelne kolorowanie kodu wątków ułatwia zrozumienie problemów z wątkami w czasie ich wykonywania
- Narzędzie **Monitor Usage Analyzer** pozwala zapobiegać zakleszczaniu się wątków. Generuje ono pełną listę ostrzeżeń o błędach, które mogą prowadzić do zakleszczeń i problemów z wydajnością, m.in. ostrzeżenia o zakleszczeniu, ostrzeżenia o blokadzie i oczekiwaniu oraz ostrzeżenia o blokadzie i oczekiwaniu na operację wejścia-wyjścia
- **Optimizeit™ Code Coverage** - weryfikowanie sesji testowych. Aktualizowany w czasie rzeczywistym widok pokrycia klas pozwala szybko zobaczyć pokrycie każdej klasy i zidentyfikować te, które nie są w pełni pokryte. Widok pokrycia metod wyświetla metody i wiersze kodu wybranej klasy, które nie zostały użyte, pozwalając programistom zmodyfikować zestawy testowe tak, aby pokryć wszystkie obszary kodu.
- **Optimizeit™ ServerTrace Developer Edition for J2EE™** - funkcje profilowania. Lokalne testowanie aplikacji wykazuje, które komponenty J2EE stanowią "wąskie gardła". Wizualny, łatwy w użyciu interfejs upraszcza śledzenie interakcji w ramach aplikacji J2EE
- Narzędzie **System Dashboard** pokazuje w postaci graficznej czas aplikacji spędzony w poszczególnych komponentach J2EE oraz całkowitą liczbę żądań. Wyświetla procentowe wykorzystanie każdego modułu serwera, ułatwiając zespołowi programistów szybkie wykrycie najważniejszych problemów z wydajnością na poziomie komponentów.
- Widok **System Composite** wyświetla - w czasie rzeczywistym i we właściwej hierarchii - wszystkie zdarzenia J2EE, które wystąpiły w aplikacji. Hierarchia pokazuje związki między zdarzeniami, tzn. które zdarzenie wywołało inne.
- Widok **HotSpot™** sortuje zdarzenia według czasu poświęconego na ich wykonanie.
- Widoki **Component Performance** dostarczają szczegółowych informacji diagnostycznych o komponentach JDBC®, JSP™, JNDI, EJB™ i JMS.
- Narzędzie **Automatic Application Quality Analyzer** identyfikuje ukryte problemy z kodem i potwierdza spełnienie wymagań w zakresie wydajności i niezawodności, prognozując problemy z wydajnością J2EE jeszcze przed ich wystąpieniem

Techniki odpluskwiania jądra: ksymoops, UML jako sposób odpluskwiania jądra, kgdb.

Kgdb

Program kgdb (debugger jądra Linux'a używający gdb) dostarcza mechanizm, by debuggować jądro Linux'a używając gdb. Program kgdb jest rozszerzeniem jądra, które pozwala ci połączyć się z maszyną z jądrem rozszerzonym o kgdb poprzez gdb uruchomiony na centralnym komputerze. Możesz wtedy włamać się do jądra, ustawić break-point'y , egzaminować dane i tak dalej (możliwości są podobne do tych podczas użycia gdb na zwykłym programie aplikacyjnym). Jedną z najważniejszych cech tej łaty jest to, że komputer centralny (host) uruchamiając gdb łączy się z komputera docelowego (posiadającym jądro które chcemy debuggować) podczas procesu inicjowania(boot). To pozwala ci debuggować jak najwcześniej. Zauważ, że łata dodaje funkcjonalność do jądra Linux'a więc gdb może zostać użyty, by debuggować jądro Linux'a.

Aby użyć kgdb wymagane są Dwie maszyny: jedna jest maszyna główną a druga testową. Maszyny są połączone przez port szeregowy. Jądro które chcesz debuggować startujesz na maszynie testowej; gdb działa na głównej. Gdb używa połączenia szeregowego, by komunikować się z debuggowanym jądrem.

Poniższe kroki pozwolą ci ustawić środowisko dla **kgdb**:

- Pobierz odpowiednią łatę dla twojej wersji jądra Linuxa.
- Wkompiluj ją do jądra, ponieważ to jest najłatwiejsza droga, by użyć **kgdb**. (Zauważ, że są dwie drogi, by zbudować większość komponentów jądra: moduły albo bezpośrednie wkompilowanie w jądro. Np. system plików JFS może być wbudowany w jądro lub skompiloway jako moduł. Używając gdb, możemy wbudować JFS bezpośrednio do jądra.)
- Uruchom patch zbuduj ponownie jądro.
- Utwórz plik .gdbinit i umieść go w katalogu /usr/src/linux. Plik .gdbinit powinien wyglądać tak:
  set remotebaud 115200
  symbol-file vmlinux
  target remote dev/ttyS0
  set output-radix 16

- Dodaj append=gdb do lilo.

Trzeba jeszcze odpowiednio przygotować maszynę testową przed testowaniem. Np. za pomocą skryptu podobnego do poniższego.

```
set -x
rcp best@sfb: /usr/src/linux-2.4.17/arch/i386/boot/bzImage /boot/bzImage-2.4.17
rcp best@sfb:/usr/src/linux-2.4.17/System.map /boot/System.map-2.4.17
rm -rf /lib/modules/2.4.17
rsync -a best@sfb:/lib/modules/2.4.17 /lib/modules
chown -R root /lib/modules/2.4.17
lilo
```

Przykładowe użycie kgdb może wyglądać następująco:

Najpierw zmieniamy funkcje int jfs_mount(..) tak by zawierała błąd.

```
int jfs_mount(struct super_block *sb)
{
...
int ptr; /* line 1 added */
jFYI(1, ("\nMount JFS\n"));
/ *
* read/validate superblock
* (initialize mount inode from the superblock)
* /
if ((rc = chkSuper(sb))) {
goto errout20;
}
108 ptr=0; /* line 2 added */
109 printk("%d\n",*ptr); /* line 3 added */
```

teraz wykonując polecenie montowania ujawni się nam błąd:

```
mount -t jfs /dev/sdb /jfs

Program received signal SIGSEGV, Segmentation fault.
jfs_mount (sb=0xf78a3800) at jfs_mount.c:109
109 printk("%d\n",*ptr);
(gdb)where
#0 jfs_mount (sb=0xf78a3800) at jfs_mount.c:109
#1 0xc01a0dbb in jfs_read_super ... at super.c:280
#2 0xc0149ff5 in get_sb_bdev ... at super.c:620
#3 0xc014a89f in do_kern_mount ... at super.c:849
#4 0xc0160e66 in do_add_mount ... at namespace.c:569
#5 0xc01610f4 in do_mount ... at namespace.c:683
#6 0xc01611ea in sys_mount ... at namespace.c:716
```

#7 0xc01074a7 in system_call () at af_packet.c:1891
#8 0x0 in ?? ()
(gdb)

Istnieje kilka rozkazów dostępnych z kgdb, takie jak obrazowanie struktur danych i wartości zmiennych i raport jaki stan mają wszystkie zadania w systemie, śpią, zużywają CPU i tak dalej. Powyższy przykład pokazuje informację, którą dostarcza śledzenie; gdzie komenda „where” użyta została by wykonać śledzenie(back trace) które mówi, jakie wywołania zostały wykonane miejsca zastoju w kodzie.

Kdb

Debuger jądra Linux'a (**kdb**) jest łatą dla jądra Linux'a i dostarcza sposób przeegzaminowywania pamięci jądra i struktur danych podczas gdy system pracuje. Zauważ, że kdb nie wymaga dwóch maszyn, za to nie pozwala debuggować na poziomie źródeł jądra jak **kgdb**. Możesz dodać dodatkowe polecenia by formatować i oglądać podstawowe struktury danych systemu mając dany identyfikator albo adres struktury danych. Aktualny komplet poleceń pozwala kontrolować funkcje jądra, włączając następujące możliwości:

- obserwować wszystko krok po kroku (single-steping processor)
- Zatrzymując po wykonaniu określonej instrukcji
- Zatrzymując na dostępie (albo modyfikacji) określonej lokacji w pamięci
- Zatrzymując na dostępie do rejestru w przestrzeni wejścia - wyjścia
- "back trace" stosu dla aktualnego zadania(task) jak również dla wszystkich innych zadań (do procesu ID)
- Deasemblacja instrukcji

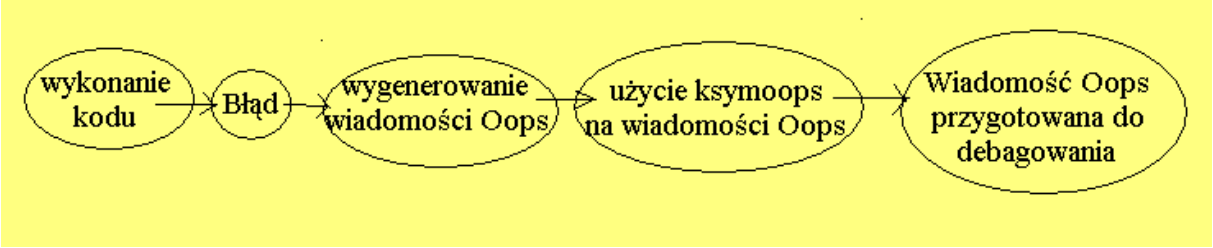
Oops, ksymoops

Kiedy jądro dostrzega istnienie poważnego błędu, wywołany jest "oops". Oops ma dwa główne zadania:

- Aby zatrzymać informacje przydatne do późniejszego usuwania błędów, która może zostać użyta, by zdiagnozować przyczynę problemu.
- Aby spróbować uniknąć utraty kontroli przez jądro i spowodowania naruszenia zgodności danych, albo co gorsza, uszkodzenia sprzętu(chociaż to jest bardzo rzadkie).

Oops (albo panic) - wiadomość zawierająca szczegóły awarii systemu, takie jak zawartości rejestrów CPU. W Linux'ie tradycyjną metodą debuggowania błędów systemu jest analizowanie szczegółów wiadomości Oops wysłanej do konsoli systemowej w trakcie błędu. Gdy zrozumiesz problem, wiadomość może zostać przekazana do narzędzia ksymoops, które próbuje przemienić kod do instrukcji i mapuje wartości stosu do symboli jądra. W wielu przypadkach, są to wystarczające informacje dla odpluskwiającego, by określić możliwą przyczynę niepowodzenia. Zauważ, że wiadomość Oops nie zawiera pliku rdzenia**.**

Powiedzmy, że twój system właśnie utworzył wiadomość Oops. Jako autor kodu, chcesz rozwiązać problem i określić co spowodowało wiadomość Oops, albo chcesz dać producentowi kodu, który spowodował wiadomość Oops, jak najwięcej informacji o twoim problemie, aby mógł zostać rozwiązany w odpowiedni sposób. Wiadomość Oops jest jedną częścią równania, ale to nie jest pomocne bez przepuszczenia jej przez program ksymoops. Schemat poniżej przedstawia kolejność generowania wiadomości Oops gotowej do odpluskwiania.



Jest kilka rzeczy których potrzebuje ksymoops: Produkcji wiadomości Oops, pliku System.map do działającego jądra i /proc/ksyms, vmlinux i /proc/modules. Polecenia jak używać ksymoops są w źródle jądra

/usr/src/linux/Documentation/oops-tracing.txt albo na stronie man ksymoops. Ksymoops deasembluje (rozmontowuje) sekcję kodu, wskazując do błędnych instrukcji i sekcję "trace" , która pokazuje jak kod został wywołany.

Najpierw, zapisz wiadomość Oops do pliku aby móc przekazać ją do ksymoops. Poniższy przykład pokazuje, że Oops utworzony przez polecenie montowania JFS z problemem, który został utworzony przez nas w do trzech linijkach kodu dodanych do instrukcji: int jfs_mount() (patrz opis kgdb).

Wiadomość Oops po przepuszczeniu 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
```

Następnie, potrzebujesz określić, która linia powoduje problem w jfs_mount. Wiadomość Oops mówi nam, których problem jest spowodowany przez instrukcję o ofsecie 3c. Jedną rzeczą, którą możesz zrobić to użyć narzędzia objdump na jfs_mount.o i kontrolować ofset 3c. Objdump jest użyty, by zdeasemblować funkcję modułu i zobaczyć jakie instrukcje asemblera są utworzone przez twój kod źródłowy C. Poniższy przykład pokazuje co pokazałby ci objdump i następnie, możemy spojrzeć na C kod dla jfs_mount i

zobaczyć, że null-problem został spowodowany w lini 109. Offset 3c jest ważny, ponieważ to jest miejsce, które wiadomość Oops zidentyfikowała jako przyczynę problemu.

```
109printk("%d\n",*ptr);

objdump jfs_mount.o

jfs_mount.o: file format elf32-i386

Disassembly of section .text:

00000000 <jfs_mount>:
   0:55 push %ebp
  ...
  2c:e8 cf 03 00 00   call   400 <chkSuper>
  31:89 c3       mov     %eax,%ebx
  33:58    pop     %eax
  34:85 db       test %ebx,%ebx
  36:0f 85 55 02 00 00 jne 291 <jfs_mount+0x291>
  3c:8b 2d 00 00 00 00 mov 0x0,%ebp << problem line above
  42:55push %ebp
```

User-Mode Linux (UML)

User-Mode Linux (UML) jest bezpiecznym, pewnym sposobem uruchomienia wersji Linux'a i procesów Linux'a. Uruchomienie nieodpluskwianego oprogramowania, eksperymentowanie z nowymi jądrami Linux'a albo dystrybucjami i myszkowanie w wewnętrznych zasobach Linux'a, a wszystko to bez ryzyka uszkodzenia ustawień twojej głównej dystrybucji Linux'a.

User-Mode Linux dostarcza wirtualną maszynę, która może mieć więcej sprzętu i oprogramowania wirtualnego niż twój komputer. Pamięć dyskowa dla maszyny wirtualnej jest w całości zawarta wewnątrz pojedynczego pliku na twoim komputerze. Możesz udostępnić maszynie wirtualnej dostęp do tego sprzętu komputerowego, do którego chcesz. Z właściwie ograniczonym dostępem, nic co robisz na maszynie wirtualnej nie może zmienić ani uszkodzić twojego komputera ani oprogramowania.

Odkąd **UML** działa jak normalny Linux, jest możliwe, by debuggować go za pomocą gdb prawie jak dowolny inny proces. To jest nieznacznie różne, ponieważ wątki jądra są już "ptraced"(śledzone) dla przerwań systemowych, więc gdb nie może ich "ptrace"(śledzić). Jednakże został już dodany mechanizm, by pracować z tym problemem**.**

Aby debuggować jądro, musisz zbudować je z źródeł. Upewnij się, że włączyłeś opcje CONFIG_DEBUGSYM i CONFIG_PT_PROXY podczas konfiguracji jądra(config). Spowoduje to skompilowanie jądra z opcją -g i umożliwi ptrace proxy (pośrednictwo śledzenia) żeby gdb mógł działać z **UML**.

Uruchomienie jądra pod gdb

Możesz uruchomić jądro pod kontrolą gdb od początku poprzez wstawienie 'debug' w wierszu poleceń. Uzyskasz konsole z gdb. Jądro wyśle polecenia do gdb, które zatrzymają proces początku start_kernel. W tym miejscu , możesz sterować przebiegiem rzeczy poprzez 'next', 'step', lub 'cont'.

Przykładowa sesja debuggera:

Poniżej jest początek sesji gdb, z jądrem uruchomionym pod gdb od początku. To zaczyna się na początku start_kernel() i posuwa się po linia autostart jądra.

```
GNU gdb 4.17.0.11 with Linux support
Copyright 1998 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 "i386-redhat-linux"...
(gdb)  att 1
Attaching to program `/home/dike/linux/2.3.26/um/linux', Pid 1
0x1009f791 in __kill ()
(gdb)  b start_kernel
Breakpoint 1 at 0x100ddf83: file init/main.c, line 515.
(gdb)  c
Continuing.

Breakpoint 1, start_kernel () at init/main.c:515
515         printk(linux_banner);
(gdb)  n
516         setup_arch(&command_line);
(gdb)
517         printk("Kernel command line: %s\n", saved_command_line);
(gdb)
518         parse_options(command_line);
(gdb)
519         trap_init();
(gdb)
520         init_IRQ();
(gdb)
521         sched_init();
(gdb)
522         time_init();
(gdb)
523         softirq_init();
(gdb)
530         console_init();
```

kontynuujemy podając komendę c (cont)

```
(gdb)  c
Continuing.
```

Jest teraz bootowane, więc ja wysyłam sygnał ^C, aby zobaczyć co to spowodowało.

```
Program received signal SIGINT, Interrupt.
0x100a4bc1 in __libc_nanosleep ()
(gdb)  bt
#0  0x100a4bc1 in __libc_nanosleep ()
#1  0x100a4b7d in __sleep (seconds=10) at ../sysdeps/unix/sysv/linux/sleep.c:78
#2  0x10095fbf in do_idle () at process_kern.c:424
```

```
#3  0x10096052 in cpu_idle () at process_kern.c:450
#4  0x100de0a4 in start_kernel () at init/main.c:593
#5  0x10098df2 in start_kernel_proc (unused=0x0) at um_arch.c:72
#6  0x1009858f in signal_tramp (arg=0x10098db8) at trap_user.c:50
(gdb)
```

Jest uśpione w pętli oczekiwania. Ustawiam breakpoint w programie szeregującym(scheduler) aby wychwycić to przy następnym przełączeniu kontekstów.

```
(gdb)  b schedule
Breakpoint 2 at 0x10004acd: file sched.c, line 496.
(gdb)  c
Continuing.

Breakpoint 2, schedule () at sched.c:496
496             if (!current->active_mm) BUG();
(gdb)  bt
#0  schedule () at sched.c:496
#1  0x10095fb3 in do_idle () at process_kern.c:421
#2  0x10096052 in cpu_idle () at process_kern.c:450
#3  0x100de0a4 in start_kernel () at init/main.c:593
#4  0x10098df2 in start_kernel_proc (unused=0x0) at um_arch.c:72
#5  0x1009858f in signal_tramp (arg=0x10098db8) at trap_user.c:50
```

Jesteśmy w programie planującym. Użyję polecenia "next" by pominąć pierwszych kilka linii i ustawię breakpoint w SIGIO - programie obsługi przerwań.

```
(gdb)  n
497             if (tq_scheduler)
(gdb)
501             prev = current;
(gdb)
502             this_cpu = prev->processor;
(gdb)
504             if (in_interrupt())
(gdb)
510             if (softirq_state[this_cpu].active & softirq_state[this_cpu].mask)
(gdb)
518             sched_data = & aligned_data[this_cpu].schedule_data;
(gdb)
520             spin_lock_irq(&runqueue_lock);
(gdb)  b sigio_handler
Breakpoint 3 at 0x10094fdc: file irq_user.c, line 36.
(gdb)  c
Continuing.

Breakpoint 2, schedule () at sched.c:496
```

```
496          if (!current->active_mm) BUG();
```

Oops, proces został zaszeregowany z powrotem do wątku pustego. Pozbądź się tego breakpoint'u i kontynuuj(od początku).

```
(gdb) d 2
(gdb) c
```

Testowanie śpiących procesów

Nie każda pluskwa jest widoczna w obecnie pracującym procesie. Czasami, procesy wiszą w jądrze kiedy nie powinny, ponieważ zakleszczyły się na semaforze albo czymś podobnym. W tym przypadku, kiedy wyślesz ^C do gdb i zobaczysz "backtrace", będziesz widział bezczynny wątek, który nie jest nie bardzo istotny.

To czego chcesz to stos jakiegokolwiek śpiącego procesu który to spać nie powinien. Musisz znaleźć proces o który chodzi, co jest ogólnie dość łatwe. Wtedy potrzebujesz dostać id jego procesu macierzystego, co możesz zrobić np. za pomocą polecenia ps.

- odłącz od aktualnego wątku
  ```
  ( gdb UML) det
  ```
- podłącz wątkowi siebie jako zainteresowanego
  ```
  ( gdb UML) att < pid gospodarza >
  ```
- spójrz na jego stos i na cokolwiek na co potrzebujesz
  ```
  ( gdb UML) bt
  ```
- Zauważ, że nie możesz zrobić w tym miejscu niczego co wymaga wykonania procesu, np. wywołania funkcji kiedy ty obserwujesz ten proces, ponownie załącz się do bieżącego wątku i kontynuuj
  ```
  ( gdb UML) att 1
  ( gdb UML) c
  ```
  W tym miejscu, wyszczególnianie dowolnego pidu, który nie jest id procesu w wątku UML'a spowoduje, że gdb ponownie doczepi się do aktualnego wątku.

Odpluskwianie modułów jądra

Gdb posiada wsparcie dla debuggowania kodu programu, który jest dynamicznie ładowany do procesu. To wsparcie jest potrzebne, by debuggować moduły jądra pod UML. Używanie tego wsparcia jest jednak skomplikowane. Musisz przekazać do gdb informację jaki plik obiektowy(.o) został załadowany do UML i, gdzie w pamięci się znajduje. Wtedy może czytać tablicę symboli i ocenić gdzie są wszystkie symbole z adresu który dostarczyłeś. To staje się bardziej interesujące kiedy ładujesz moduł ponownie (np. po rmmod). Musisz polecić gdb by "zapomniał" o wszystkich jego symbolach, włączając główne symbole UML z jakiegoś powodu i dopiero wtedy załadowywać wszystkie od nowa.
Jest łatwa i trudna droga by to zrobić. Ta łatwa droga to użycie skryptu umlgdb expect napisany przez Chandan Kudige.

Uruchomienie ddd na UML (ddd jest nakładką na gdb):

ddd może pracować nad **UML**, ale wymaga zastosowanie pewnego triku:

- Start ddd

  ```
  host % linux ddd
  ```

- Za pomocą np. ps, zdobądź pid gdb na którym działa ddd. Możesz poprosić gdb aby Ci go podał, ale z jakiś powodów zdarza się że to spowoduje zawieszenie.
- Uruchom **UML** z opcją 'debug=parent gdb-pid=<pid>' z wiersza poleceń
- Wpisz "att 1" do ddd gdb i powinieneś zobaczyć coś jak niżej

  ```
  0xa013dc51 w __kill ()
  (gdb)
  ```

- Wpisz c typu, UML się zainicjuje i będziesz mógł używać ddd jak to robisz na jakimkolwiek innym procesie.

Zwykłe odpluskwiacze

Debugging Tools for Windows – co i po co?

Debugging Tools for Windows to zestaw narzędzi do śledzenia i odpluskwiania aplikacji, sterowników, serwisów i samego systemu operacyjnego. Platformy, na które jest dedykowany to:

- **Windows NT 4.0,**
- **Windows 2000,**
- **Windows XP,**
- **Windows Server 2003.**

Program jest dostępny do pobrania darmowo pod adresem:

http://www.microsoft.com/whdc/devtools/debugging/default.mspx

Kickstart

Zanim zaczniemy odpluskwiać programy za pomocą warto zastosować następujące kroki:

1. Zainstalować WinDbg
2. Uruchomić WinDbg.
3. W menu File| Symbol File Path dodać ścieżkę **SRV*<c:\mój_katalog_lokalny_do_przychowywania_symbli>*http://msdl.microsoft.com/download/symbols**

Niestety przy tej opcji wszelkie analizy potrafią trwać dosyć długo ze względu na ściąganie symboli z sieci.
Jeśli chcemy odpluskwiać sterownik lub samo jądro Windowsa potrzebujemy drugiego komputera oraz połączenia między nimi. WinDbg musi być zainstalowany na komputerze, z którego będziemy odpluskwiać, a niekoniecznie na komputerze z wadliwym oprogramowaniem.

Tryby pracy WinDbg

WinDbg oferuje kilka trybów pracy:

1. Załadowanie kodu źródłowego

2. Załadowanie pliku wykonywalnego
3. Podłączenie do działającej aplikacji
4. Analiza plików Crash Dump
5. Odpluskwianie kernela: na lokalnej maszynie lub przez połączenie COM lub IEEE1394.

Nieco o pracy z WinDbg

WinDbg oferuje wszystkie podstawowe funkcje debuggera. Są to: śledzenie wołania funkcji i procedur – wejście w, wyjście na zewnątrz, przejście ponad wywołaniem funkcji. Pozwala przeglądać zawartość rejestrów, stosu wywołań, pamięci, znanych zmiennych lokalnych oraz zdeasemblowany kod programu. W przypadku, gdy posiadamy źródła, możemy operować na języku wysokopoziomowym tak, jak robią to inne typowe debuggery dołączane do środowisk programistycznych.

Śledzenie programu

WinDbg, jak każdy szanujący się debugger, oferuje następujące funkcje śledzenia wykonywania programu:

1. **Zatrzymanie** (Ctrl+Break) – zatrzymanie pracy programu
2. **Step Oper** (F10) – przejście ponad wywołaniem funkcji
3. **Step In** (F11) – wejście w głąb wołanej funkcji
4. **Step Out** (Shift + F11) – wyjście jeden poziom do góry z funkcji
5. **Run To Kursor** (F7) – wykonanie zatrzymanego programu do linii podświetlonej w źródle programu lub zdeasemblowanym kodzie.
6. **Run** (F5) – zwykłe wykonanie programu do końca lub do najbliższego breakpointa.
7. Zmiana wartości rejestru EIP – metoda brutalna. Często przynosi niepożądane skutki.

Breakpoints

WinDbg oferuje kilka rodzajów breakpointów:

1. Zwykłe – ustawiane poprzez **bp adres**, lub **bp <moduł>!<nazwa>.** Taki breakpoint oczekuje, iż wszelkie dane do ustawienia pułapki są już znane w systemie. W szczególności moduł musi być załadowane i symbol

<nazwa> musi być rozpoznawany. W przeciwnym wypadku założenie breakpointu będzie niemożliwe

2. Z wyprzedzeniem – ustawiane poprzez **bu <moduł>!<nazwa>.** Ten rodzaj breakpointów nie oczekuje, że w systemie będzie załadowany odpowiedni moduł. Zostanie ustawiony dopiero, gdy <moduł> zostanie załadowany do systemu. Gdyby po wgraniu modułu symbol <nazwa> nie został rozpoznany breakpoint nie zostanie ustawiony. Tego rodzaju breakpointy pozostają w systemie także po ponownym restarcie i po usunięciu modułu.
3. Z akcją – składnia **bp <moduł>!<nazwa> "akcja".** Ten rodzaj breakpointów pozwala tworzyć tzw. warunkowe pułapki lub łatać programy „w locie”. Typowym poleceniem używanym w sekcji akcja jest: **j <warunek> '<polecenia, gdy warunek prawdziwy>' ; '<polecenia gdy warunek fałszywy>'**. Przykład:
    i. bp MyDriver!DriverControl+0x100 "j (@@(MyPointer)=0x00000000) '.echo MyPointer is not good' ; '.echo Nothing to look at; g' "

   g oznacza komendę ‘go’, .echo jest oczywiste. Dokładne znaczenie składni j i jej warunków można znaleźć w dokumentacji WinDbg.

4. Pułapki na dostęp – składnia **ba <w/r><liczba> <adres>**. Są uruchamiane, gdy proces próbuje podjąć akcję zapisu(w) lub odczytu(r) określonej liczby bajtów(<liczba>) pod adresem <adres>. Typowe wykorzystanie: przeszukujemy pamięć w poszukiwaniu określonego łańcucha (np. ‘Access denied’) i zakładamy nań pułapkę. Gdy breakpoint zostanie aktywowany będziemy w okolicach funkcji, która próbowała wyświetlić napis ‘Access denied’, więc prawdopodobnie w okolicach funkcji, która próbowała sprawdzić ‘Access’ ;)

Inne popularne operacje na breakpointach:

1. **bl** – wypisuje wszelkie ustawione breakpointy
2. **bd <numer|all>** – wyłącza tymczasowo podany breakpoint
3. **bc <numer|all>** - kasuje podany breakpoint

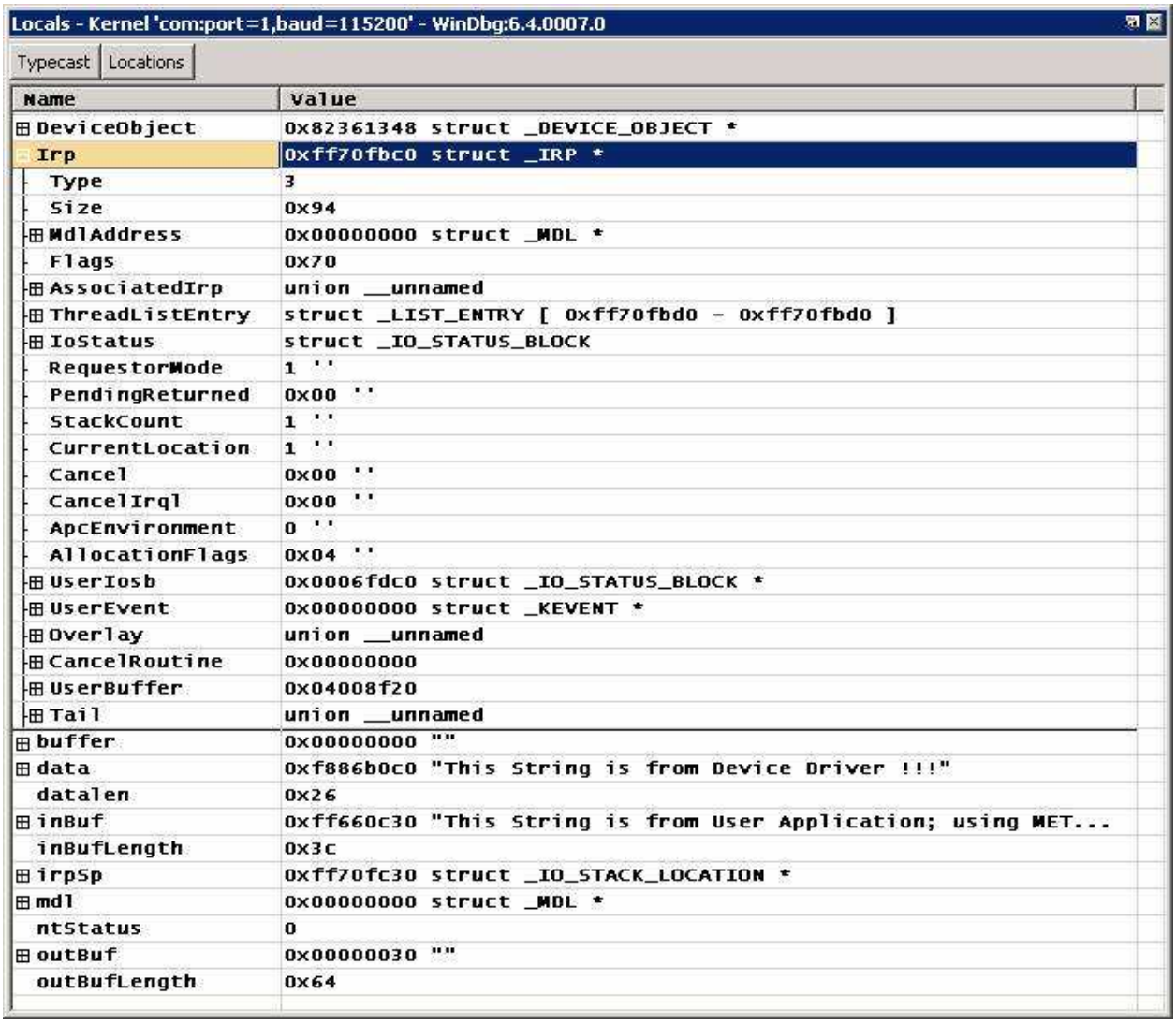
Manipulacja zmiennymi i pamięcią

Manipulacja zmiennymi:

Do oglądania zmiennych w WinDbg służy komenda **dv**. Wyświetli ona wszystkie zmienne danego kontekstu znane w momencie jej wywołania.

Uwaga: Należy pamiętać, że część zmiennych może być przechowywana w samych rejestrach procesora – komenda dv nie weźmie ich pod uwagę.

Inną, równie przydatną komendą jest **dt** – pozwala ona wyświetlić zawartość zmiennej sformatowaną zgodnie z rozpoznanym typem. Jest to przydatne, gdy mamy do czynienia ze wskaźnikami na skomplikowane struktury, zatem ręczne przeglądanie zawartości wskaźnika byłoby męczące.
Skrót: **?? <zmienna>. ??** pozwala oceniać wartości zmiennych tak jak w C++, zatem możliwe są konstrukcje typu: ?? struktura->mojepole. Inną ważną cechą ?? jest możliwość zmiany zmiennych: **?? <zmienna> = <nowa wartość>.**
Wszystkie te operacje są dostępne również w środowisku graficznym w oknie Locals(View->Locals), którego obsługa jest intuicyjna:



Rysunek 1. Okno Locals

Wyświetlanie zawartości pamięci:

dd, dw, db <adres> <długość>, gdzie dd – bierze pod uwagę podwójne słowa, dw – pojedyncze słowa, db – pojedyncze bajty, wraz z ich reprezentacja ASCII.

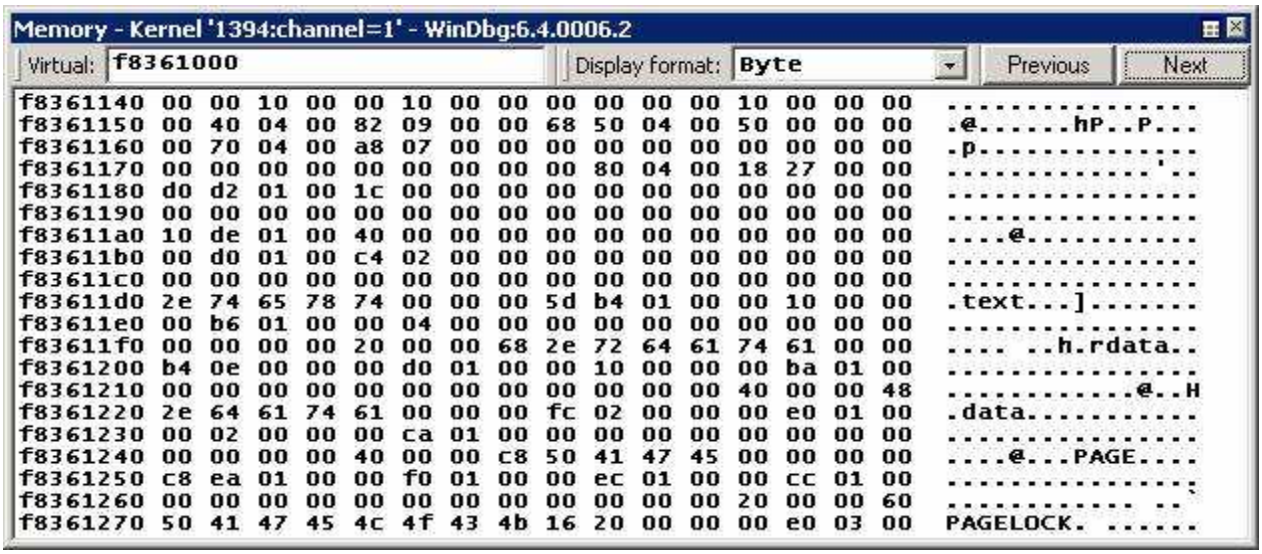
Zmiana zawartości pamięci:

Wymieniane wyżej ??, oraz

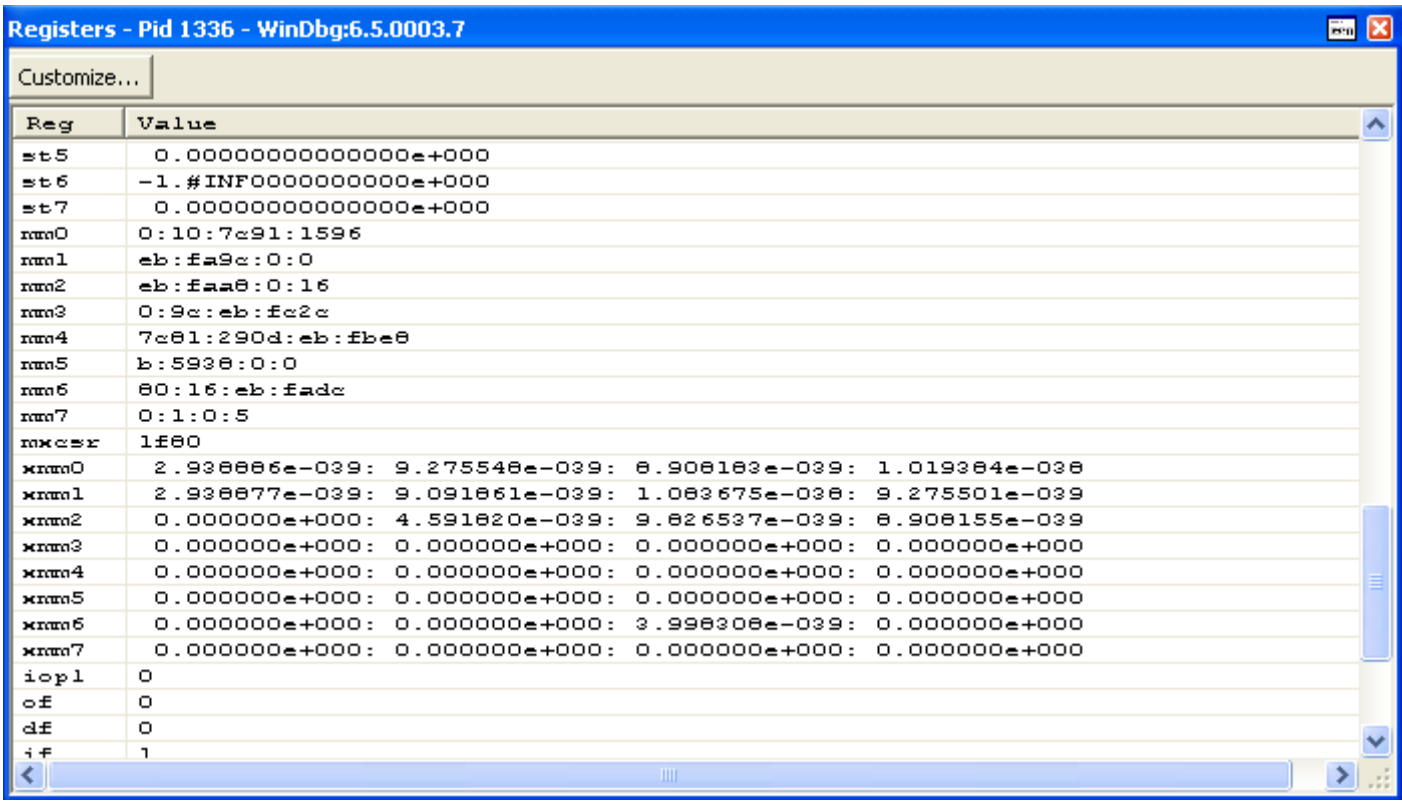
ed ew eb <adres> <nowa wartość>

Manipulacja zawartością rejestrów:
r – wyświetla zawartość wszystkich rejestrów
r <rejestr> - konkretnego rejestru
r < rejestr> = <nowa wartość> - przypisuje nową wartość do rejestru.
rM - wypisuje pełniejszy zestaw rejestrów

Warto zwrócić uwagę, że wymienioną tu funkcjonalność oferują okna Memory (View->Memory), Register(View->Register) z GUI WinDbg.



Rysunek 2. Okno Memory

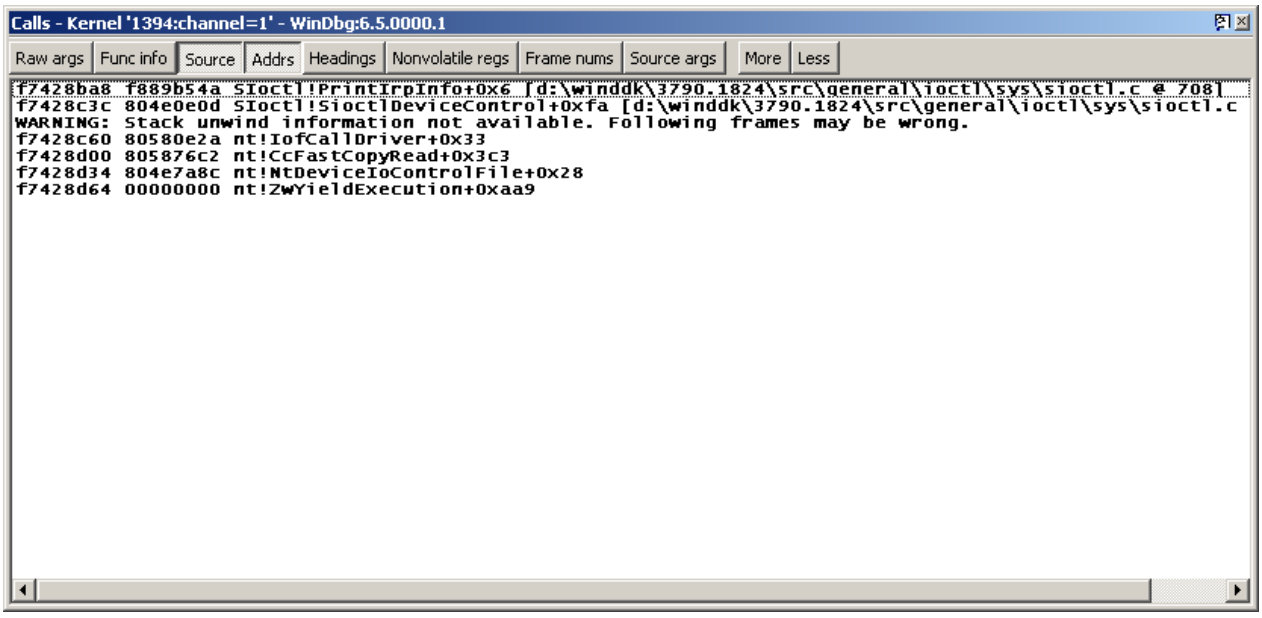


Rysunek 3. Okno Registers

Wyszukiwanie nazwy w module:
Do wyszukiwania nazwy służy komenda x o składni:
x <moduł>!<wzorzec> Wzorzec to wyrażenie regularne, do którego zostaną dopasowane symbole z modułu. Komenda ta jest przydatna, gdy nie pamiętamy dokładnej nazwy funkcji lub struktury.

Przeglądanie Stosu Wywołań(Call Stack)

Z linii poleceń możemy wywołać komendę **k**, która wyświetli bieżący stos wywołań. Najwyższa linia to obecne wywołanie, w którym nastąpiło przerwanie działania. Linie poniżej to poprzednie wywołania(razem z adresem, z którego nastąpiło wołanie i być może linią, o ile dysponujemy źródłem). Stos wywołań jest poprawnie wyświetlane tylko, jeśli dysponujemy symbolami. W przeciwnym razie jego zawartość będzie bez większej wartości. Warto zauważyć, że do przeglądania stosu wywołań można użyć okna calls (View->Call Stack) oferowanego przez GUI WinDbg.



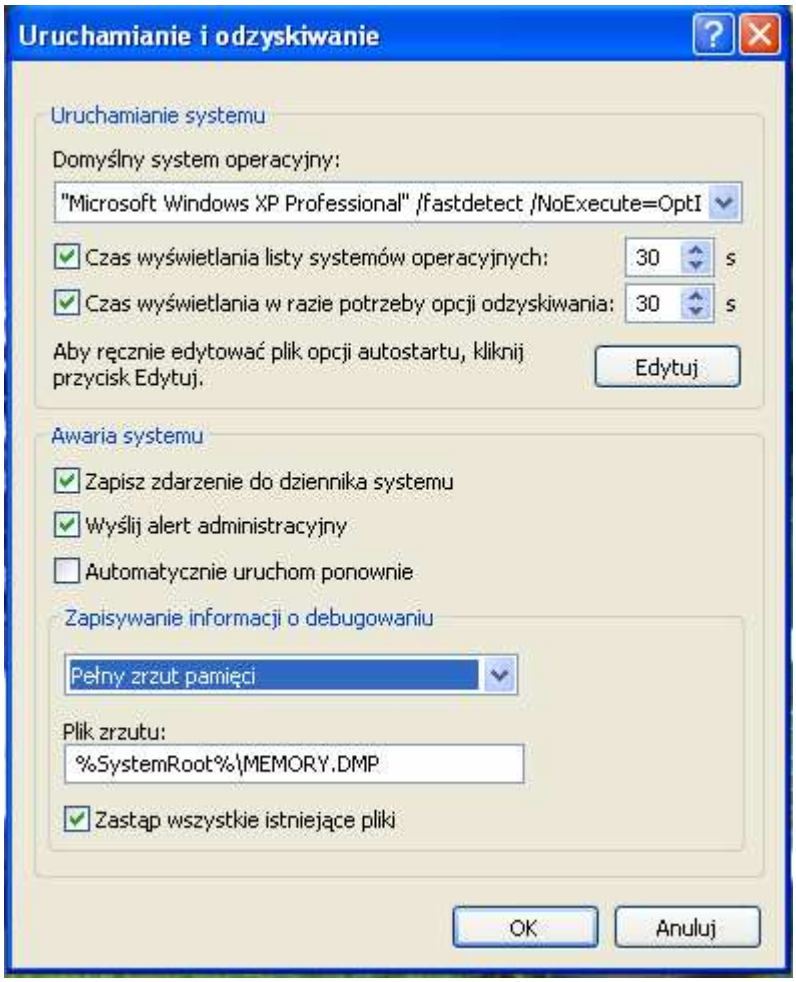
Rysunek 4. Okno Call Stack

Nieco o działaniu na Core Dumps' ach.

Windows XP oferuje przygotowywanie trzech rodzajów zrzutów pamięci:

1. Zrzut pamięci jądra
2. Mały zrzut – 64 KB
3. Pełny zrzut pamięci.

Opcje te można znaleźć we właściwościach systemu w zakładce zaawansowane pod pozycją Uruchamianie i Odzyskiwanie.



Rysunek 5. Ustawienia pliku zrzutu

Co oferuje core dump?

Najrozsądniejszą opcją zrzutu pamięci jest „kornel-mode”. W tym przypadku po katastrofie dostępne są wszystkie dane, jakimi dysponowalibyśmy podczas katastrofy przy włączonym WinDbg.

Postępowanie z dump files.

Warto upewnić się, że poprawnie ustawiliśmy ścieżkę do katalogu symboli, źródeł(zapewne ich nie mamy, chyba, że jesteśmy autorami felernego sterownika) i obrazów(zwykle będzie to katalog /<windows>/system32 lub któryś z podkatalogów).

Po otwarciu pliku zrzutu najczęstszą komendą, jakiej używamy, jest **!analyze** z opcja **–v** (jak verbose). WinDbg użyje swoich algorytmów analizy, aby wyświetlić nam w podsumowaniu prawdopodobnego winowajcę, podejrzaną funkcję oraz linię kodu, która spowodowała błąd. Niestety, o ile nie posiadamy źródeł, będzie to linia kodu asemblera.

A oto przykładowy wynik:

```
Debugging Details:
------------------


EXCEPTION_CODE: (NTSTATUS) 0xc0000005 - Instrukcja spod
"0x%08lx" odwołuje się do pamięci pod adresem "0x%08lx".
Pamięć nie może być "%s".

FAULTING_IP:
HSF_CNXT+50f51
f4e5bf51 ff91c0000000     call    dword ptr [ecx+0xc0]

TRAP_FRAME:  f51dc7ec -- (.trap ffffffff51dc7ec)
ErrCode = 00000000
eax=ff645000 ebx=ff6670f8 ecx=00000000 edx=f4e89547
esi=f4e89500 edi=ff66765c
eip=f4e5bf51 esp=f51dc860 ebp=f51dc8b4 iopl=0         nv up ei
ng nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00210286
HSF_CNXT+0x50f51:
f4e5bf51 ff91c0000000     call dword ptr [ecx+0xc0]
ds:0023:000000c0=????????
Resetting default scope

CUSTOMER_CRASH_COUNT:  1

DEFAULT_BUCKET_ID:  DRIVER_FAULT

BUGCHECK_STR:  0x8E

LAST_CONTROL_TRANSFER:  from f4e3a970 to f4e5bf51

STACK_TEXT:
WARNING: Stack unwind information not available. Following
frames may be wrong.
f51dc8b4 f4e3a970 ff6670f8 00000000 00000000 HSF_CNXT+0x50f51
f51dc8cc f4e8aa3f ff6670f8 00000000 ff79e3d0 HSF_CNXT+0x2f970
f51dc8e4 804ec04f ff667040 80e74bf8 00000000 HSF_CNXT+0x7fa3f
f51dc8ec 80e74bf8 00000000 f95e965c ff668338
nt!IopfCallDriver+0x31
f51dc8f0 00000000 f95e965c ff668338 00000001 0x80e74bf8


FOLLOWUP_IP:
HSF_CNXT+50f51
f4e5bf51 ff91c0000000     call    dword ptr [ecx+0xc0]

SYMBOL_STACK_INDEX:  0
```

```
FOLLOWUP_NAME:  MachineOwner

SYMBOL_NAME:  HSF_CNXT+50f51

MODULE_NAME:  HSF_CNXT

IMAGE_NAME:  HSF_CNXT.sys

DEBUG_FLR_IMAGE_TIMESTAMP:  3de3a497

STACK_COMMAND:  .trap ffffffff51dc7ec ; kb

FAILURE_BUCKET_ID:  0x8E_HSF_CNXT+50f51

BUCKET_ID:  0x8E_HSF_CNXT+50f51

Followup: MachineOwner
```

Inne komendy przydatne, gdy chcemy zalać ekran niewiele mówiącymi nazwami i liczbami:
!process i **!thread** – wyświetlają informacje o wszystkich wątkach/procesach w chwili wypadku
lmnt – wyświetla listę załadowanych modułów jądra
!drvobj, **!devnode**, **!devobj**, **!devstack** – komendy te wyświetlają informację o użytym sterowniku, urządzeniu przypisanym do sterownika, jego stosie, etc.

Konkurencja dla WinDbg – Numega Soft-Ice

Historia
Numega była firmą odpowiedzialną za wybitne narzędzia do odpluskwiania takie jak: Boundschecker, SmartCheck, wspomniany Soft-Ice, czy pakiety DevPartner i DriverStudio.
Soft-Ice jest debuggerem o dłuższej historii niż opisywany wyżej WinDbg – dostępne są wersje już od DOSa począwszy a na najnowszych wersjach Windowsa skończywszy. Początkowo Soft-Ice występował jako osobny program, natomiast obecnie wchodzi w skład pakietu DevPartner jako opcjonalny składnik.
Nazwa mylnie kojarzy się z miękkim lodem, był to pierwszy software'owy niskopoziomowy odpluskwiasz, który zastąpił sprzętowego ICE'a.

Zalety i zastosowania Soft-Ice

- Soft-Ice jest debuggerem działającym pomiędzy BIOSem a systemem operacyjnym. Jest zatem niezależny od środowiska graficznego systemu operacyjnego.
- W przeciwieństwie do WinDbg Soft-Ice doskonale radzi sobie w środowiskach Windows 95 i Windows 98.
- Przy pomocy WinDbg nie można śledzić niektórych, kluczowych dla systemu, procesów na lokalnym komputerze. Podłączenie do nich debuggera powoduje „zamarznięcie" systemu. Nie jesteśmy w stanie wydać komendy Run. W takich sytuacjach doskonale sprawdza się Soft-Ice.
- Oferuje takie możliwości jak niewidoczne dla systemu operacyjnego mapowanie portów we/wy na pliki użytkownika czy przeszukiwanie całej pamięci w poszukiwaniu zadanego ciągu.
- Soft-Ice przy asyście dobrego deassemblera poza odpluskwianiem znajduje szerokie zastosowanie w takich dziedzinach jak: cracking oraz reverse engineering(obie nie do końca legalne).
- Miłe jest ze strony producentów odpluskwiaczy, że stosują dosyć jednolite nazwy i skróty komend w swych produktach, zatem większość komend wydawanych WinDbg będzie brzmiała identycznie pod Soft-Icem.
- Niestety, jest aplikacją komercyjną.

http://www.compuware.com/products/driverstudio/softice.htm

Folklor w Linuxie – strace, ltrace, printf

Strace – polecenie systemowe, dzięki któremu można śledzić, jakich funkcji systemowych używa program wraz z ich argumentami i zwracaną wartością. Aby otrzymać sensowne wyniki zwykle potrzebne są uprawnienia administratora.
Wywołanie: **strace <program> 2>&1 | less**, gdyż strace wypluwa wszelkie komunikaty na standardowy strumień błędów. Mając wszelkie „przydatne" informacje możemy szukać, które wywołanie powoduje błąd w programie.
Za pomocą tej komendy można śledzić również losy potomków (opcja **–f**), sprawdzić stan zawieszonego procesu**(-p <pid>**) lub zawęzić krąg podejrzanych funkcji (**-e trace=<network/file/etc>).**

Ltrace – polecenie systemowe, które pozwala śledzić wywołania biblioteczne, zatem funkcje bardziej ludzkie – takie jak fopen(), printf(), etc. Pozostała funkcjonalność jest niemal identyczna jak w strace.

Printf() – pewną metodą poszukiwania błędów i eliminowania ich z aplikacji jest użycie funkcji printf(), dzięki czemu możemy wypisywać przebieg sterowania w programie, a także wartości interesujących zmiennych. W pewnych przypadkach, opisanych poniżej jest to w zasadzie jedyna sensowna metoda odpluskwiania.

Dzika pluskwa pod żaglami – GDB

Gdb jest to odpluskwiacz oferujący podobną funkcjonalność co WinDbg z tą różnicą, że jest przeznaczony na platformy Linuxowe.
Warto zacząć od tego, że w przypadku plików pozbawionych symboli (co jest standardem w przypadku gotowych aplikacji) nie dowiemy się specjalnie wiele, poza instrukcją kodu asemblera, która mogła spowodować błąd.

Jak przygotować się do odpluskwiania?
Programy, które wymagają wglądu warto kompilować z opcją **–g**, co pozwoli nam wygodnie operować na aplikacji z poziomu gdb – będziemy mieć dostęp do symboli oraz kodu źródłowego w przyzwoitym formacie.
Należy zezwolić także linuxowi na tworzenie zrzutów pamięci w przypadku katastrofy procesu. Robimy to poleceniem
ulimit –c <wartość> wartość jest maxymalnym rozmiarem pliku core, który będzie tworzony po katastrofie procesu.

Jak odpluskwiać programy?
Wywołanie: **gdb <plik> <plik zrzutu pamięci>**
W przypadku programów, które działają po prostu niepoprawnie i nie koniecznie powodują zrzut pamięci można pominąć <plik zrzutu pamięci>.
Kilka początkowych linii źródła można wyświetlić komendą: **l.** Warto również na wstępie ustawić breakpoint na wywołaniu funkcji main() lub, jeśli mamy podejrzenia, która część działa źle, bezpośrednio na docelowej funkcji. Robimy to komendą **b <nazwa funkcji>**. Istnieje również możliwość założenia pułapki na konkretną linię kodu źródłowego. Po wykonaniu komendy **l** ujrzymy fragment źródła, który otacza instrukcję, na której program się zatrzymał. Każda linia jest numerowana, zatem wystarczy wydać polecenie **b <nr linii>**, aby gdb zatrzymał się przy uruchomieniu programu na wyznaczonej instrukcji.
Następnie **r**, jak run i lądujemy w poszukiwanej części programu. Warto sprawdzić, jakie są wartości zmiennych, bądź też skorygować je komendą **p**

<zmienna> [=nowawartość]. I przeskakiwać aplikację linijka po linijce za pomocą komendy **s**(jak step). Gdb ma tą własność, że wszelkie instrukcje, do których nie posiada kodu źródłowego stara się pomijać docierając do najbliższej instrukcji, którą potrafimy odpluskwiać. Dlatego na przykład wywołanie scanf() w programie zostanie wykonane jednym poleceniem step. Jeśli jednak chcemy, by gdb faktycznie przechodził program instrukcja po instrukcji należy użyć polecenia **ni**(next instruction), lecz nie posiadając kodów źródłowych nie na wiele nam się to zda.
Dodatkowe komendy do oglądania danych programu to:
info registers – wypisuje zawartość rejestrów
info variables - wyświetla listę wszystkich zmiennych w programie, o ile program został skompilowany z opcjami odpluskwiania.

Co jeszcze warto wiedzieć o uruchamianiu programów w gdb?
Często programy wymagają podania kilku argumentów wywołania. W gdb można to uczynić za pomocą komendy **set args <arg1> <arg2> <arg3>**. Natomiast sprawdzenie, jakie aktualnie argumenty są ustawione czynimy za pomocą **show args.**
Środowisko wołania można zmienić przy pomocy komendy **set env <var> [=value]**, gdzie <var> to nazwa ustawianej zmiennej. Bliźniaczą komendą jest **unset env <var>**.
Ponadto istnieją komendy takie jak: **run > <file>, run < <file>**, które służą przekierowywaniu wejścia/wyjścia odpluskwianego programu.

Breakpoints, watchpoints, catchpoints
Breakpointy są podstawowym narzędziem odpluskwiania aplikacji i zostały omówione pokrótce w punkcie „Jak odpluskwiać programy”. Oczywiście do dyspozycji mamy pułapki warunkowe(lecz w odróżnieniu od WinDbg nie mamy do dyspozycji instrukcji, które się wykonają po osiągnięciu pułapki, a jedynie warunek, który uaktywni pułapkę. Do realizacji podobnej funkcjonalności jak w WinDbg służy fraza **commands** dołączana do warunku pułapki).
Watchpoints są to pułapki zastawiane na zmienne i ogólnie dostęp do pamięci. Do dyspozycji mamy 3 rodzaje watchpointów :

- wrażliwe na zapis(**watch <wyrażenie>**)
- wrażliwe na odczyt (**rwatch <wyrażenie>**)
- po prostu wrażliwe(**awatch <wyrażenie>**)

Catchpoints w gdb biorą odpowiedzialność za pozostałe zdarzenia w programie, których nie można zaliczyć do poprzednich dwóch grup. Mogą być to: ładowanie biblioteki, namnażanie procesu, uruchamianie innego programu, łapanie/ porzucanie wyjątków (jak w C++), podnoszenie sygnałów, etc.
Ogólna składnia polecenia to **catch <zdarzenie>**. Po szczegóły zapraszam do pomocy gdb(**help catch)**.

Do wszystkich rodzajów pułapek dostępny jest zestaw komend takich jak: **info**¸ **enable**, **disable**, **delete**, których znaczenie jest raczej oczywiste.

Stos wywołań

W każdej chwili możemy:

- obejrzeć stos wywołań komendą **bt**(backtrace)
- obejrzeć konkretną ramkę stosu(**frame <nazwa>**)
- powrócić z ostatniego wywołania funkcji komendą **ret**(zatem jest to część funkcjonalności dotyczącej sterowania przebiegiem programu)

Oczywiście wszystkie te komendy zwrócą sensowną odpowiedź, o ile program jest w trakcie wykonywania. W przeciwnym wypadku gdb powie nam, że stos wywołań jest pusty.

Pliki źródłowe

Często w trakcie pracy nad niepewnym programem potrzebujemy przejrzeć źródło lub też je zmienić. Do tych celów służą komendy:

- **l(list) <nr linii/nazwa funkcji>** - wyświetla fragment kodu otaczający podany numer
- **edit <nr linii/nazwa funkcji>** - odpali edytor ustawiony w środowisku na zmiennej EDITOR

Postępowanie z fork()

Gdb zawsze śledzi wykonywanie głównego programu, zatem po wywołaniu fork() nadal pozostajemy w procesie rodzica. Co zatem robić, jeśli błąd kryje się w procesie potomka?

Najprostszą metodą radzenia sobie z tym problemem jest umieszczenie instrukcji sleep() w kodzie, tak by była jedną z pierwszych instrukcji w procesie potomka, która się wykona. Dobrym nawykiem jest uzależnienie czasu opóźnienia od zmiennej środowiskowej – można go wtedy regulować w zależności od sprytu osoby, która odpluskwia. W czasie, kiedy proces potomka zasypia musimy wytropić jego PID za pomocą znanego polecenia **ps**, a następnie wywołać gdb z opcją podłączenia do już uruchomionego procesu(**gdb –p <pid>**). Następnie musimy wykonać powrót (**ret)** z funkcji oczekiwania, aż obecną ramką będzie wywołanie funkcji main().

Metoda ta ma jedną wadę. Przy każdym ponownym uruchomieniu musimy odnaleźć szukany proces i wykonać wszystkie opisane wyżej czynności. Nie jest również możliwe odpluskwianie złożonej aplikacji, która uruchamia wiele procesów potomnych, a tylko niektóre powodują błędy. W takich sytuacjach pozostaje nam intuicja oraz printf().

Core dumped - co dalej?

W przypadku, gdy nasz program zaczyna nazbyt niesfornie zachowywać się w systemie jego wykonywanie zostaje przerwane, a przydatne informacje

dotyczące powodu i stanu procesu w chwili słabości zostają zapisane do pliku core. Po zlokalizowaniu aplikacji, która została przerwana można wywołać gdb, aby dowiedzieć się czegoś więcej na temat przyczyn:

gdb <nazwa programu> <plik core>

Po wczytaniu plików w linii poleceń gdb wydajemy rozkaz **where** i lądujemy dokładnie w miejscu gdzie program zakończył swój żywot. Niestety, nie zawsze jest to pomocne, gdyż często przyczyny błędu są dosyć oddalone od miejsca, w którym faktycznie został on ujawniony.

Źródła:

- http://www.iecc.com/linker/
- http://binboy.sphere.pl/index.php?show=serwis&d=asm&s=aplik.htm
- http://rainbow.mimuw.edu.pl/SO/LinuxPodrecznik/PAMIEC/PREZENTACJA/
- Wikipedia http://wikipedia.org/
- http://debian.linux.org.pl/zrobione/reference/reference.txt
- http://www.issi.uz.zgora.pl/~patan/materialy/stare/so/druk5_sem2.pdf
- ftp://sunsite.unc.edu/pub/Linux/devel/lang/c - zasoby
- http://www.3miasto.net/~chq/c/howto/x530.html
- http://www.cbmamiga.demon.co.uk/mpatrol
- http://www-128.ibm.com/developerworks/linux/library/l-debug/
- http://user-mode-linux.sourceforge.net/debugging.html
- http://www.kernelhacking.org/docs/kernelhacking-HOWTO/indexs09.html
- http://www.microsoft.com/whdc/devtools/debugging/default.mspx
- http://people.redhat.com/alikins/old_docs/debug.txt
- http://sources.redhat.com/gdb/current/onlinedocs/gdbint.html#SEC_Top
- Strony podręcznika dla gdb, strace i ltrace.