Odpluskwianie - Debugging

Piotr Buczek,
Magdalena Dukielska,
Adam Maciejewski

styczeń, 2006

Plan prezentacji

1. "Zwykłe" odpluskwiacze systemów uniksowych (na przykładzie `gdb`).
2. Opcje odpluskwiania w `gcc` i pliki obiektowe przygotowane do odpluskwiania (`stabs`, `DWARF`).
3. Narzędzia do wykrywania wycieków pamięci w programie i do profilowania kodu.
4. Techniki odpluskwiania jądra: `ksymoops`, `kgdb`, `UML`.
5. Odpluskwianie w Windows (`KD`, `WinDbg`).

"Zwykłe" odpluskwiacze systemów uniksowych (na przykładzie `gdb`).

Po co komu debugger?

- Istnieją podobno programiści, których kod już od początku działa tak, jak powinien...
- ... tym niemniej są oni w przyrodzie rzadkością
- Większość z nas musi mozolnie przeczesywać swoje programy w poszukiwaniu mniejszych i większych błędów

D... debugging

Najprostsza możliwa metoda:

1. Wstawiamy w podejrzanych miejscach programu wywołania printf (lub czegoś podobnego)
2. Patrzymy, które się wykonają, a które już nie.
3. Zawężamy obszar poszukiwań stosując wyszukiwanie binarne

W wersji bardziej zaawansowanej możemy też wypisywać wartości wybranych zmiennych.

D... debugging

Przykład:

```
if (sscanf(operation, "%ld %c %ld", &args[0], &op, &args[1]) != 3)
        return -EINVAL;                    /* Niepoprawne wyrazenie */
    printk(KERN_DEBUG "calc: Skladnia wyrazenia poprawna\n");
    for (i=0; i<2; i++) {
        if (ref & (1«i)) {
            if ((args[i] < mem) && (args[i] >= 0)) {
                cmc = cells[args[i]]->data;
                spin_lock(&cmc->lock);
                printk(KERN_DEBUG "calc: Czytamy z komorki %d\n", i);
                if (sscanf(cmc->value, "%ld", &args[i]) !=1) {
                    spin_unlock(&cmc->lock);
                    return -EFAULT;
                }
                spin_unlock(&cmc->lock);
            } else return -EFAULT;
```

Gdy to nie wystarcza...

Możemy chcieć:

- dokonać „sekcji zwłok" programu, który zrobił `segfault`
- zajrzeć do środka już uruchomionego, działającego programu
- prześledzić krok po kroku przebieg wykonania naszego kodu
- podglądać lub zmieniać wartości wybranych zmiennych
- ...

Jednym słowem, potrzebujemy debuggera

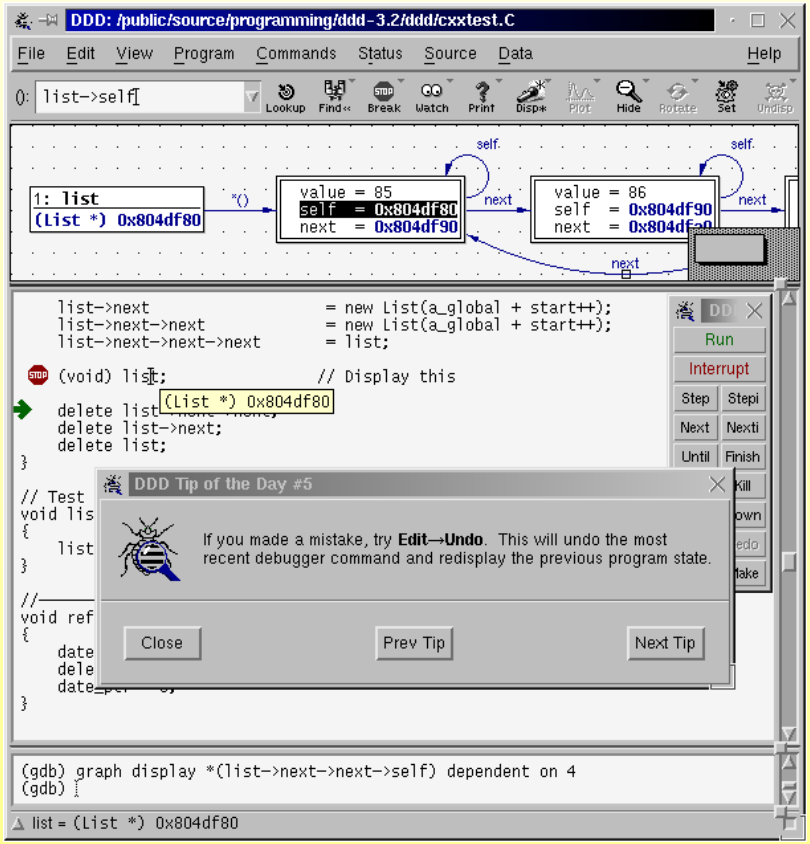
GNU Debugger

Dostępny pod adresem `http://www.gnu.org/software/gdb/`

Potrafi wszystko to, a nawet więcej ;-). Nowsze wersje radzą sobie z programami wielowątkowymi oraz dynamicznie ładowanym kodem. Można go używać bezpośrednio z linii komend, ale istnieje też mnóstwo graficznych nakładek.

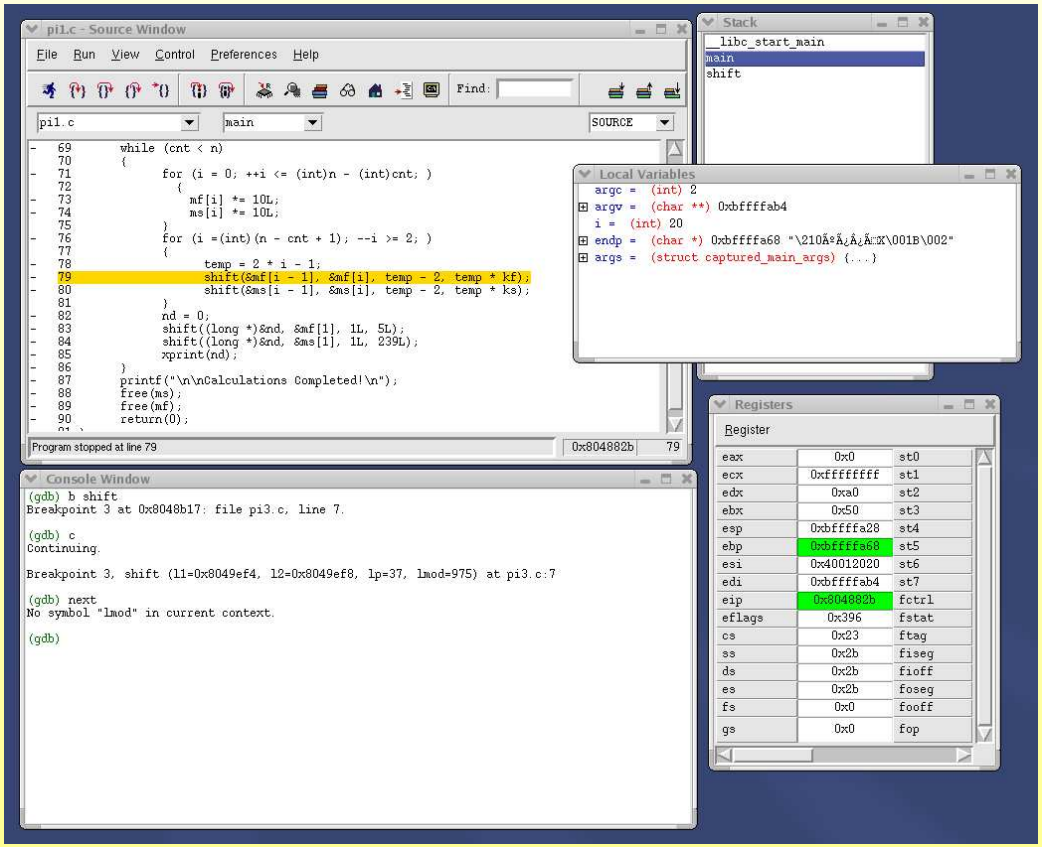
Do popularniejszych należą...

Data Display Debugger



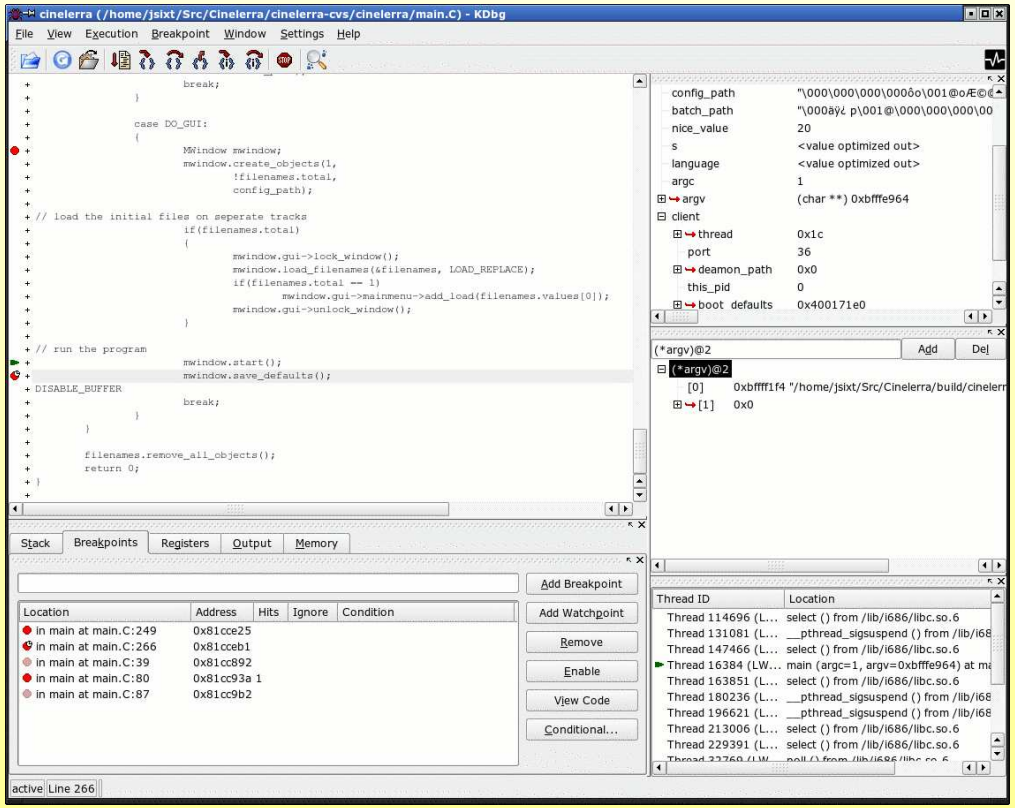
Rysunek: http://www.gnu.org/software/ddd/

Insight



Rysunek: http://sources.redhat.com/insight/

KDbg



Rysunek: http://members.nextra.at/johsixt/kdbg.html

Jak się tego używa?

Kompilujemy program z odpowiednimi opcjami (-g lub --debug)

... uruchamiamy go pod kontrolą debuggera:

```
% gdb nasz_program
```

... definiujemy miejsca, w których chcemy wstrzymać wykonanie:

```
(gdb) break funkcja
```

... i jazda!

```
(gdb) run
```

Gdzie jeszcze można się zatrzymać

w gdb istnieją trzy rodzaje pułapek:

breakpoint gdy osiągnięta zostanie wybrana linia kodu lub nastąpi wywołanie wskazanej funkcji (`break plik.c:nr_linii`, `break funkcja`)

watch gdy wartość pewnego wyrażenia ulegnie zmianie (`watch wyrazenie`)

catch gdy wystąpi zdarzenie takie jak: rzucenie albo złapanie wyjątku, wywołanie fork / exec, itp. (`catch throw`, `catch catch`, `catch fork`, `catch exec`, ...)

Pułapki mogą być warunkowe (`break funkcja if warunek`) albo tymczasowe (`tbreak funkcja`)

Skąd wziąć plik core?

- Zrzuty pamięci powinny być tworzone automatycznie, gdy program kończy się błędem
- Ich rozmiar jest jednak ograniczony, często do zera (czyli nie powstają wcale...)
- Limit ten można zmieniać poleceniem ulimit, np. `% ulimit -c 50000` (max. 50.000 bajtów)
- Wywołanie ulimit jest zwykle umieszczane w pliku `/etc/profile` lub `~/.bash_profile`

Można też zdefiniować wzorzec nazw tworzonych zrzutów pamięci, pisząc do pliku `/proc/sys/kernel/core_pattern`. Więcej informacji: `man bash` i `man proc`.

Opcje odpluskwiania w `gcc` i pliki obiektowe przygotowane do odpluskwiania (`stabs`, `DWARF`).

Pliki obiektowe

- **Kompilatory** i **asemblery** - tworzą pliki obiektowe zawierające kod w postaci binarnej i informacje o kodzie źródłowym
- **Linkery** - łączą różne pliki obiektowe w jeden
- **Loadery** - ładują pliki obiektowe do pamięci
- **Odpluskwiacze** - korzystają z danych zapisanych w pliku do wykrywania błędów

Wniosek: potrzebne są standardy opisujące, jak plik obiektowy powinien być zbudowany

Przykładowe formaty plików obiektowych

- **.COM** (MS DOS)
- **a.out** (BSD UNIX)
- **ELF** (System V)
- **XCOFF** (Windows)
- **PE** (Windows)
- **OMF** (Windows)

Co zawiera plik obiektowy?

Plik obiektowy powinien zawierać przynajmniej część z następujących informacji:

- **Nagłówek/Ogólne dane**
  - typ formatu plików (*'numer magiczny'* - patrz `/etc/magic`)
  - typ procesora, dla którego wygenerowaliśmy kod
  - nazwa pliku źródłowego, z którego powstał
  - data utworzenia
- ...

Co zawiera plik obiektowy?

- ...
- **Obraz programu**
  - rozmiar i początek **kodu wykonywalnego**
  - kod
  - rozmiar i początek **zainicjalizowanych danych**
  - zainicjalizowane dane
  - rozmiar i początek **niezainicjalizowanych danych**
  - niezainicjalizowane dane
- ...

Co zawiera plik obiektowy?

- ...
- **Tablica symboli**
  - zewnętrzne definicje i referencje, czyli **symbole**, które muszą być zaimportowane z innych modułów albo zdefiniowane przez linker
  - informacje o **relokacjach**, czyli lista miejsc, w których linker musi poprawić adresy
  - informacje do debugowania, czyli m.in. kod źródłowy, informacje o jego numerach linii, symbole lokalne, opisy struktur danych, jak np. definicje struktur w C
  - komentarze

Formaty plików obiektowych do debugowania

Kompilatory UNIXowe posługują się dwoma różnymi formatami wspomagającymi debugowanie:

- **stab** (skrót od *'symbol table'*) używany w a.out, COFF i ELF (poza Systemem V)
- **DWARF** zdefiniowany dla plików ELF Systemu V.

Microsoft również stworzył swoje własne formaty dla debugera Codeview, najnowszy z nich to **CV4**.

Co dają nam powyższe formaty?

Dzięki informacjom zgromadzonym w fazie kompilacji i linkowania programu, w trakcie odpluskwiania za pomocą różnych narzędzi możemy:

- odwoływać się do **kodu źródłowego** funkcji, nazw zmiennych, definicji struktur
- ustawiać **punkty kontrolne** (*'breakpoints'*)
- wykonywać program **instrukcja po instrukcji**
- śledzić **stos wywołań**
- korzystać z raportowania błędów przez odpluskwiacz z odniesieniami do kodu źródłowego

Uwaga na kod wykonywalny zoptymalizowany przez kompilator! ;)

Kilka szczegółów technicznych

Typowe metody zapisywania informacji o debugowaniu w plikach obiektowych:

- tworzenie danych składających się z **numeru linii i adresu początku kodu maszynowego**, który jej odpowiada (tzw. *'line number entry'*) dla każdej linii w kodzie źródłowym
- indeksowanie symboli z plików dołączanych za pomocą dyrektyw `#include` identyfikatorem pliku
- przechowywanie informacji o nazwach, typach i położeniu każdej zmiennej w programie w **formie drzewa**

STABS (Symbol Table)

Informacja do debugowania w tym formacie znajduje się w **dyrektywach asemblera**, które nazywamy dyrektywami **stab** i które są **przemieszane z generowanym kodem.**

```
...
6  .stabs "int:t1=r1;-2147483648;2147483647;",128,0,0,0
//przykładowa dyrektywa .stabs (opisje się nią np. zmienne)
7 .stabs "char:t2=r2;0;127;",128,0,0,0
...
35 .stabn 68,0,5,LM2
//przykładowa dyrektywa .stabn (opisuje się nią m.in. etykiety)
```

DWARF

- związany przede wszystkim z formatem ELF
- informacje potrzebne do debugowania w tym formacie są umieszczane w sekcji `.debug` pliku obiektowego (przynajmniej w wersji 1.1)
- łatwo **rozszerzalny** (łatwiej niż Stabs), bo można dodawać opisy nowych obiektów lub rozszerzać istniejące
- może też opisywać bardziej skomplikowane struktury niż Stabs, np. **nieciągłe zakresy widoczności zmiennych** czy **strukturę stosu**

Opcje odpluskwiania w gcc

'-g'

Produkuje informacje do debugowania w formacie właściwym dla danego systemu operacyjnego (stabs, COFF, XCOFF, DWARF), które mogą być potem wykorzystywane przez gdb.

'-pg'

Niezbędna do korzystania z narzędzia do profilowania kodu gprof opisanego dalej.

Opcje odpluskwiania w gcc

'-gLEVEL'

Dodanie do pliku informacji dla odpluskwiacza z określeniem jak dużo informacji chcemy zapisać. Domyślny poziom to 2.
Poziom 1 — minimalna ilość danych, m.in. opisy funkcji i zewnętrznych zmiennych, ale bez uwzględnienia lokalnych zmiennych i numerów linii.
Poziom 3 — dodatkowe dane takie, jak np. wszystkie makrodefinicje występujące w programie.

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

Problemy w zarządzaniu pamięcią

- **Wycieki pamięci** - *'memory leaks'*; sytuacje, w których pamięć zaalokowana za pomocą `malloc()` nie jest zwalniana odpowiednim wywołaniem `free()`.
- **Nielegalny odczyt/zapis** - *'illegal read/write'*; polega na tym, że program próbuje coś odczytać/zapisać do obszaru pamięci, który do niego nie należy.

MEMWATCH

- Wykrywa: **podwójne** czy **błędne zwolnienia pamięci**, **niezwolnioną pamięć**, **nadpisanie buforów**, itp.
- Zastępuje wszystkie wołania funkcji alokujących pamięć swoimi funkcjami, które pamiętają jakie alokacje były wykonywane.
- Zdefiniowane są też przydatne **makra**, takie jak np. `ASSERT`, które pozwalają nam sprawdzać poprawność różnych warunków w trakcie działania programu.

MEMWATCH - przykładowy wynik działania programu

Przykładowy wynik działania dla programu, który wykonuje nielegalny zapis, nie zwalnia całej zaalokowanej pamięci i dwukrotnie próbuje zwolnić wskaźnik jest taki:

```
===== MEMWATCH 2.71 Copyright (C) 1992-1999 Johan Lindh =====
...
double-free: <5>test2.c(19), 0x805153c was freed from test2.c(18)
overflow: <6> test2.c(20), 512 bytes alloc'd at <3>test2.c(13)
unfreed: <2> test2.c(12), 512 bytes at 0x805176c   {FE FE ...}
Memory usage statistics (global):
 N)umber of allocations made: 3
 L)argest memory usage      : 1536
 T)otal of all alloc() calls: 1536
 U)nfreed bytes totals      : 512
```

YAMD

- Znajduje problemy z dynamiczną alokacją pamięci w C i C++.
- Korzysta z **mechanizmu stronnicowania**, żeby narzucić granice na alokowane bloki pamięci.
- `Malloc` i tym podobne funkcje emulowane na bardzo **niskim poziomie**, dzięki czemu sprawdzane są wywołania takich funkcji bibliotecznych, jak `strdup`.

Valgrind - zaawansowane narzędzie do debugowania

Memcheck

Odnajduje problemy z zarządzaniem pamięcią w kodzie, takie jak:

- użycie **niezainicjalizowanej pamięci**
- pisanie/czytanie pamięci po tym jak była zwolniona
- pisanie za końcem zaalokowanych bloków pamięci
- pisanie/czytanie **nieodpowiednich obszarów stosu**
- wycieki pamięci - kiedy wskaźniki do zaalokowanej pamięci są tracone na zawsze
- niepasujące do siebie wywołania `malloc()`/`new`/`new []` i `free()`/`delete`/`delete []`
- **zachodzące na siebie wskaźniki** `src` i `dst` w funkcji `memcpy()` i jej podobnych

Memcheck - przykładowy wynik działania programu

Mamy kawałek kodu, który korzysta z niezainicjalizowanej tablicy:

```
2 {
3         int i[5];
5         if (i[0] == 0)
6                 i[1]=1;
7         return 0;
8 }
```

i dostajemy:

```
==31363== Conditional jump or move depends on uninitialised value(s)
==31363==    at 0x1000041C: main (test3.c:5)
```

Valgrind - zaawansowane narzędzie do debugowania

Addrcheck

Lżejsza wersja Memcheck. Nie wykrywa niezainicjalizowanych wartości. Dzięki temu programy działają 2 razy szybciej niż z użyciem Memcheck i potrzebne jest o wiele mniej pamięci.

Helgrind

Eksperymentalne narzędzie do wykrywania braków w synchronizacji programów wielowątkowych.

Valgrind - zaawansowane narzędzie do debugowania

Cachegrind

Narzędzie do profilowania pamięci cache. Może przeprowadzić symulację użycia pamięci cache procesora poziomów L1 i L2 i wskazać źródła nietrafień w tej pamięci w sprawdzanym kodzie.

Electric Fence

- Biblioteka do debugowania `malloc()`'a.
- Alokuje **chronioną stronę pamięci** zaraz po zaalokowanym fragmencie pamięci.
- Najlepiej używać ją z debugerem, np. gdb.

Na czym polega profilowanie kodu?

Profilowanie kodu — określanie, **jak często** poszczególne kawałki kodu są wykonywane, aby stwierdzić **czy warto** go optymalizować. Właściwe użycie narzędzi do profilowania pozwala m.in. odpowiedzieć na następujące pytania:

- Które linie kodu są odpowiedzialne za większość czasu wykonania?
- Ile razy dany fragment jest wykonywany?
- Który sposób zakodowania danego fragmentu jest najbardziej efektywny?

gprof

Użycie programu wymaga wykonania trzech kroków:

1. **Kompilacja** programu gcc z opcją `-pg`, czyli z opcją profilowania
2. **Uruchomienie** programu, powoduje stworzenie pliku `gmon.out` opisującego, **ile czasu** zostało spędzone w **każdej funkcji** i na **każdej linii**
3. **Przeanalizowanie** tego pliku za pomocą gprof

gprof

Wynikiem analizy jest plik zawierający dwie tabele: tzw. płaski profil (*'flat profile'*) i graf wywołań (*'call graph'*).

Płaski profil mówi, ile czasu program spędził w każdej funkcji i ile razy ta funkcja była wołana.

Graf wywołań pokazuje dla każdej funkcji, które funkcje ją wołały, a które ona wywołała i ile razy + oszacowanie, ile czasu zajęły poszczególne instrukcje funkcji.

gprof - przykłady

Przykładowy płaski profil mógłby wyglądać tak:

```
%Time    Seconds Cumsecs #Calls  msec/call    Name
99.0 62.51    62.51    1    62510.       cholesky
0.8  0.51     63.02    1    510.         back
0.2  0.11     63.13    1    110.         init_matrix
0.0  0.00     63.13    1    0.           main
```

a przykładowy fragment tzw. "skomentowanego wydruku kodu" tak:

```
            2 ->  if (s == NULL) {
            1 ->    c = 0xffffffffL;
            1 ->    } else {
            1 ->    c = crc;
            1 ->        if (n) do {
        26312 ->            c = crc_32_tab[...];
26312,1,26311 ->        } while (--n);
```

OProfile

OProfile — do profilowania wszystkich części systemu operacyjnego od **jądra** (włącznie z **modułami** i **obsługą przerwań**) do **dzielonych bibliotek** i **zwykłych aplikacji**. Chodzi on w tle, zbierając dane i nie obciążając zbytnio systemu.
Warto go stosować, gdy chcemy:

- sprawdzić wydajność **całego** systemu operacyjnego
- przeanalizować kwestie odnoszące się do sprzętu, np. nietrafienia w pamięci Cache
- profili na poziomie **pojedynczych** instrukcji maszynowych

fenris

Fenris — zaawansowany zestaw narzędzi do **analizy i profilowania kodu**, debugowania, inżynierii wstecz, diagnostyki, analizy bezpieczeństwa i wielu innych celów.

Techniki odpluskwiania jądra:

ksymoops, kgdb, UML.

Odpluskwianie jądra - wyzwanie

- cześć funkcjonalności nie jest związana z konkretnym procesem (nie można bezpośrednio wykonywać w debugerze, śledzić w prosty sposób)
- wielowątkowość jądra
- trudno powtarzalne błędy (*race condtions*)
- błędy mogą powodować awarię systemu (niszczą świadectwo swojego istnienia, trzeba znaleźć jak najwięcej informacji o systemie po awarii)
- czasem błędy objawiają się dopiero w działaniu innego procesu, po długim łańcuchu zdarzeń (współdzielenie struktur ze złym licznikiem, pierwszy coś zwalnia, drugi później chce skorzystać)
- kod wieloplatformowy/architekturowy (SMP)

Odpluskwianie jądra - wyzwanie

- cześć funkcjonalności nie jest związana z konkretnym procesem (nie można bezpośrednio wykonywać w debugerze, śledzić w prosty sposób)
- wielowątkowość jądra
- trudno powtarzalne błędy (*race condtions*)
- błędy mogą powodować awarię systemu (niszczą świadectwo swojego istnienia, trzeba znaleźć jak najwięcej informacji o systemie po awarii)
- czasem błędy objawiają się dopiero w działaniu innego procesu, po długim łańcuchu zdarzeń (współdzielenie struktur ze złym licznikiem, pierwszy coś zwalnia, drugi później chce skorzystać)
- kod wieloplatformowy/architekturowy (SMP)

Odpluskwianie jądra - wyzwanie

- cześć funkcjonalności nie jest związana z konkretnym procesem (nie można bezpośrednio wykonywać w debugerze, śledzić w prosty sposób)
- wielowątkowość jądra
- trudno powtarzalne błędy (*race condtions*)
- błędy mogą powodować awarię systemu (niszczą świadectwo swojego istnienia, trzeba znaleźć jak najwięcej informacji o systemie po awarii)
- czasem błędy objawiają się dopiero w działaniu innego procesu, po długim łańcuchu zdarzeń (współdzielenie struktur ze złym licznikiem, pierwszy coś zwalnia, drugi później chce skorzystać)
- kod wieloplatformowy/architekturowy (SMP)

Odpluskwianie jądra - wyzwanie

- cześć funkcjonalności nie jest związana z konkretnym procesem (nie można bezpośrednio wykonywać w debugerze, śledzić w prosty sposób)
- wielowątkowość jądra
- trudno powtarzalne błędy (*race condtions*)
- błędy mogą powodować awarię systemu (niszczą świadectwo swojego istnienia, trzeba znaleźć jak najwięcej informacji o systemie po awarii)
- czasem błędy objawiają się dopiero w działaniu innego procesu, po długim łańcuchu zdarzeń (współdzielenie struktur ze złym licznikiem, pierwszy coś zwalnia, drugi później chce skorzystać)
- kod wieloplatformowy/architekturowy (SMP)

Odpluskwianie jądra - wyzwanie

- cześć funkcjonalności nie jest związana z konkretnym procesem (nie można bezpośrednio wykonywać w debugerze, śledzić w prosty sposób)
- wielowątkowość jądra
- trudno powtarzalne błędy (*race condtions*)
- błędy mogą powodować awarię systemu (niszczą świadectwo swojego istnienia, trzeba znaleźć jak najwięcej informacji o systemie po awarii)
- czasem błędy objawiają się dopiero w działaniu innego procesu, po długim łańcuchu zdarzeń (współdzielenie struktur ze złym licznikiem, pierwszy coś zwalnia, drugi później chce skorzystać)
- kod wieloplatformowy/architekturowy (SMP)

Odpluskwianie jądra - wyzwanie

- cześć funkcjonalności nie jest związana z konkretnym procesem (nie można bezpośrednio wykonywać w debugerze, śledzić w prosty sposób)
- wielowątkowość jądra
- trudno powtarzalne błędy (*race condtions*)
- błędy mogą powodować awarię systemu (niszczą świadectwo swojego istnienia, trzeba znaleźć jak najwięcej informacji o systemie po awarii)
- czasem błędy objawiają się dopiero w działaniu innego procesu, po długim łańcuchu zdarzeń (współdzielenie struktur ze złym licznikiem, pierwszy coś zwalnia, drugi później chce skorzystać)
- kod wieloplatformowy/architekturowy (**SMP**)

Kiedy debugować?

Możliwe objawy błędów (począwszy od pomyłek w kodzie, skończywszy na błędach synchronizacji):

- słaba wydajność systemu
- niewłaściwe zachowanie systemu (oops, kernel panic)
- uszkodzenia danych

Kiedy debugować?

Możliwe objawy błędów (począwszy od pomyłek w kodzie, skończywszy na błędach synchronizacji):

- słaba wydajność systemu
- niewłaściwe zachowanie systemu (**oops**, **kernel panic**)
- uszkodzenia danych

Kiedy debugować?

Możliwe objawy błędów (począwszy od pomyłek w kodzie, skończywszy na błędach synchronizacji):

- słaba wydajność systemu
- niewłaściwe zachowanie systemu (**oops**, **kernel panic**)
- uszkodzenia danych

Od czego zacząć?

Wsparcie dla odpluskwiania w jądrze → rekompilacja z włączonymi dodatkowymi opcjami w menu `kernel hacking`:

```
CONFIG_DEBUG_KERNEL=y              # udostępnia inne
CONFIG_KALLSYMS=y                  # tablica symboli jądra
CONFIG_DEBUG_SPINLOCK_SLEEP=y      # zaśnięcie ze spinlockiem
...
# CONFIG_DEBUG_INFO                # debugowanie z gdb
# CONFIG_MAGIC_SYSRQ               # klawisz "magic SysRq"
```

printk - Działanie

- używane podobnie jak `printf`
- *loglevels*
  (od `KERN_EMERG` po `KERN_DEBUG` - `<linux/kernel.h>`)
- wypisywane na konsolę, do portu szeregowego lub równoległego
- cykliczny bufor
- `printk` (budzenie czekających) $\longrightarrow$ bufor $\longrightarrow$
  $\longrightarrow$ `klogd` (wyjmuje) $\longrightarrow$ `syslogd` $\longrightarrow$
  $\longrightarrow$ **użytkownik**
  (`/etc/syslogd.conf` - czytelna konfiguracja)

printk - Zady i Walety

Zalety:

- prostota
- można użyć wszędzie w jądrze

Wady:

- wywołanie przed inicjalizacją konsoli nie ma sensu (wymagane użycie portów szeregowych i specjalnych łat na jądro)
- przeciążanie systemu nadmierną ilością komunikatów

printk - rate limiting

- często wywoływane funkcje

```
static unsigned long prev_jiffy = jiffies;  /* rate limiting*/

if (time_after(jiffies, prev_jiffy + 2*HZ)) {
    prev_jiffy = jiffies;
    printk(KERN_DEBUG "bla bla bla\n");
}
```

- wielokrotnie wywoływane funkcje

```
if (licznik++ < MAX_DEBUG) ...

if (licznik++ %= CO_ILE_DEBUG) ...
```

Jądro 2.6

Modyfikacja /proc/sys/kernel/printk_ratelimit i /proc/sys/kernel/printk_ratelimit_burst.

```
if (printk_ratelimit())
        /* funkcja zwraca coś != 0 jeśli komunikat jest dopuszczalny */
        printk("...");
```

Alternatywy dla Wypisywania

1. Wypytywanie
   - Stworzenie pliku w `/proc`, który zapewniałby nam dostęp do niezbędnych informacji.
   - Wykorzystanie narzędzi dostępnych w systemie, takich jak np. `ps`.
2. Obserwowanie (**strace** i **ltrace**)
   - pokazuje wywołania systemowe/biblioteczne wraz z argumentami, błędy (nazwy symboliczne i nazwy)
   - dane bezpośrednio z jądra
   - `-e` $\longrightarrow$ ogranicza typ śledzonych wywołań
   - `-f` $\longrightarrow$ obserwowanie dzieci
   - `-o` $\longrightarrow$ przekierowanie do pliku
   - `-p` $\longrightarrow$ podłączenie się do procesu
   - `ltrace -S` $\longrightarrow$ połączenie możliwości obu programów

Alternatywy dla Wypisywania

1. Wypytywanie
   - Stworzenie pliku w `/proc`, który zapewniałby nam dostęp do niezbędnych informacji.
   - Wykorzystanie narzędzi dostępnych w systemie, takich jak np. `ps`.
2. Obserwowanie (**strace** i **ltrace**)
   - pokazuje wywołania systemowe/biblioteczne wraz z argumentami, błędy (nazwy symboliczne i nazwy)
   - dane bezpośrednio z jądra
   - `-e` $\longrightarrow$ ogranicza typ śledzonych wywołań
   - `-f` $\longrightarrow$ obserwowanie dzieci
   - `-o` $\longrightarrow$ przekierowanie do pliku
   - `-p` $\longrightarrow$ podłączenie się do procesu
   - `ltrace -S` $\longrightarrow$ połączenie możliwości obu programów

Komunikat Oops naszym przyjacielem

- zawiera informacje o stanie procesora z momentu wystąpienia błędu (rejestry)
- **kernel panic** - podczas obsługi przerwania, w `idle` lub `init`
- wywołanie
  - Oops:
    ```
    if (blad) BUG();    /* == BUG_ON(blad); */
    ```
  - kernel panic:
    ```
    if (blad_krytyczny)
        panic("komunikat wypisywany przed wstrzymaniem systemu");
    ```
- Jak to czytać?
  ksymoops - konwertuje adresy na wartości symboliczne (wykorzystuje `nm`)

Zawieszenie systemu - Co zrobić jak nie ma Oopsa?

Gdy system nie reaguje nawet na Alt+Ctrl+Del.

- zapobieganie - wstawiamy schedule() w strategicznych miejscach kodu
- **"magic SysRq"** - kombinacja Alt+SysRq z trzecim klawiszem
  - s, u, b - bezpieczny reset
  - m, p - wypisanie informacji o pamięci, rejestrach
  - r - tryb surowy klawiatury

Jak to się robi? (na działającym systemie)

- czasochłonne, niezalecane
- chcemy mieć zbliżone możliwości do debugowania w trybie użytkownika (wykonywanie krok po kroku, pułapki, obserwowanie wartości i ich modyfikacja)
- (dla zdesperowanych) **gdb** - wymaga:
  - sporej wiedzy o komendach
  - umiejętności dopasowywania kodu źródłowego do **zoptymalizowanego kodu maszynowanego**
  - nieskomresowanego obrazu jądra (uruchomienie `gdb vmlinux /proc/kcore`)
  - conajmniej jądra 2.6.7 dla debugowania modułów (nie są zawarte w obrazie)

  Przy tym wszystkim daje jedynie możliwość odczytu danych.
- **łaty** na jądro

kgdb - Zdalne Debugowanie

- **kgdb** - łata na jądro, która umożliwiająca jego debugowanie z wykorzystaniem pełni możliwości gdb
- **zdalne** debugowanie - potrzebne są dwie maszyny:
  1. komputer **docelowy** - działa na jądrze z nałożoną łatą
  2. **host** - połączony przez port szeregowy (kabel typu `nullmodem`), wykorzystuje gdb do odpluskwiania

  Dzięki temu można debugować system już w trakcie ładowania systemu (modułów).
- skomplikowany proces konfiguracji, później jest już prosto

kgdb - Zdalne Debugowanie

- **kgdb** - łata na jądro, która umożliwiająca jego debugowanie z wykorzystaniem pełni możliwości gdb
- **zdalne** debugowanie - potrzebne są dwie maszyny:
  1. komputer **docelowy** - działa na jądrze z nałożoną łatą
  2. **host** - połączony przez port szeregowy (kabel typu `nullmodem`), wykorzystuje gdb do odpluskwiania

  Dzięki temu można debugować system już w trakcie ładowania systemu (modułów).
- skomplikowany proces konfiguracji, później jest już prosto

kgdb - Zdalne Debugowanie

- **kgdb** - łata na jądro, która umożliwiająca jego debugowanie z wykorzystaniem pełni możliwości gdb
- **zdalne** debugowanie - potrzebne są dwie maszyny:
  1. komputer **docelowy** - działa na jądrze z nałożoną łatą
  2. **host** - połączony przez port szeregowy (kabel typu `nullmodem`), wykorzystuje gdb do odpluskwiania

  Dzięki temu można debugować system już w trakcie ładowania systemu (modułów).
- skomplikowany proces konfiguracji, później jest już prosto

kgdb - Działanie

Działamy na **hoście** w terminalu **gdb**:

- **Ctrl+C** - zatrzymanie debugowanego systemu i dostęp do komend gdb
- `(gdb) continue` - powrót
- analiza krok po kroku - za każdym razem zwalnia i ponownie zatrzymuje wszystkie procesory
- debugowanie wątków - `(gdb) info thr`, specjalne identyfikatory
- obserwowanie - punkty kontrolne na wykonanie instrukcji, dostęp do pamięci o danym adresie, lub jej modyfikację
- wykrywanie ładowania/odładowywania modułów

kgdb - Działanie

Działamy na **hoście** w terminalu **gdb**:

- **Ctrl+C** - zatrzymanie debugowanego systemu i dostęp do komend gdb
- `(gdb) continue` - powrót
- analiza krok po kroku - za każdym razem zwalnia i ponownie zatrzymuje wszystkie procesory
- debugowanie wątków - `(gdb) info thr`, specjalne identyfikatory
- obserwowanie - punkty kontrolne na wykonanie instrukcji, dostęp do pamięci o danym adresie, lub jej modyfikację
- wykrywanie ładowania/odładowywania modułów

kgdb - Działanie

Działamy na **hoście** w terminalu **gdb**:

- **Ctrl+C** - zatrzymanie debugowanego systemu i dostęp do komend gdb
- `(gdb) continue` - powrót
- analiza krok po kroku - za każdym razem zwalnia i ponownie zatrzymuje wszystkie procesory
- debugowanie wątków - `(gdb) info thr`, specjalne identyfikatory
- obserwowanie - punkty kontrolne na wykonanie instrukcji, dostęp do pamięci o danym adresie, lub jej modyfikację
- wykrywanie ładowania/odładowywania modułów

kgdb - Działanie

Działamy na **hoście** w terminalu **gdb**:

- **Ctrl+C** - zatrzymanie debugowanego systemu i dostęp do komend gdb
- `(gdb) continue` - powrót
- analiza krok po kroku - za każdym razem zwalnia i ponownie zatrzymuje wszystkie procesory
- debugowanie wątków - `(gdb) info thr`, specjalne identyfikatory
- obserwowanie - punkty kontrolne na wykonanie instrukcji, dostęp do pamięci o danym adresie, lub jej modyfikację
- wykrywanie ładowania/odładowywania modułów

kgdb - Działanie

Działamy na **hoście** w terminalu **gdb**:

- **Ctrl+C** - zatrzymanie debugowanego systemu i dostęp do komend gdb
- `(gdb) continue` - powrót
- analiza krok po kroku - za każdym razem zwalnia i ponownie zatrzymuje wszystkie procesory
- debugowanie wątków - `(gdb) info thr`, specjalne identyfikatory
- obserwowanie - punkty kontrolne na wykonanie instrukcji, dostęp do pamięci o danym adresie, lub jej modyfikację
- wykrywanie ładowania/odładowywania modułów

kgdb - Działanie

Działamy na **hoście** w terminalu **gdb**:

- **Ctrl+C** - zatrzymanie debugowanego systemu i dostęp do komend gdb
- `(gdb) continue` - powrót
- analiza krok po kroku - za każdym razem zwalnia i ponownie zatrzymuje wszystkie procesory
- debugowanie wątków - `(gdb) info thr`, specjalne identyfikatory
- obserwowanie - punkty kontrolne na wykonanie instrukcji, dostęp do pamięci o danym adresie, lub jej modyfikację
- wykrywanie ładowania/odładowywania modułów

kdb - Alternatywa kgdb

- łata umożliwiająca **lokalne debugowanie**
- udostępnia m.in.:
  - modyfikację zmiennych,
  - analizę krok po kroku kodu wykonywalnego na procesorze,
  - zatrzymanie na wykonaniu konkretnej instrukcji,
  - zatrzymanie na dostępie (w tym również modyfikacji) do konkretnej lokacji w pamięci wirtualnej,
  - zatrzymanie na dostępie do rejestru,
  - analizę stosu dla aktywnego procesu jak i innych procesów (wykorzystując PID)
  - deasemblację instrukcji
- **nie można** debugować z poziomu **kodu źródłowego**
- uruchomienie - wciśnięcie klawisza **Break**
- uruchamia się automatycznie po wywołaniu **Oops'a**

kdb - Alternatywa kgdb

- łata umożliwiająca **lokalne debugowanie**
- udostępnia m.in.:
  - modyfikację zmiennych,
  - analizę krok po kroku kodu wykonywalnego na procesorze,
  - zatrzymanie na wykonaniu konkretnej instrukcji,
  - zatrzymanie na dostępie (w tym również modyfikacji) do konkretnej lokacji w pamięci wirtualnej,
  - zatrzymanie na dostępie do rejestru,
  - analizę stosu dla aktywnego procesu jak i innych procesów (wykorzystując PID)
  - deasemblację instrukcji
- **nie można** debugować z poziomu **kodu źródłowego**
- uruchomienie - wciśnięcie klawisza **Break**
- uruchamia się automatycznie po wywołaniu **Oops'a**

kdb - Alternatywa kgdb

- łata umożliwiająca **lokalne debugowanie**
- udostępnia m.in.:
  - modyfikację zmiennych,
  - analizę krok po kroku kodu wykonywalnego na procesorze,
  - zatrzymanie na wykonaniu konkretnej instrukcji,
  - zatrzymanie na dostępie (w tym również modyfikacji) do konkretnej lokacji w pamięci wirtualnej,
  - zatrzymanie na dostępie do rejestru,
  - analizę stosu dla aktywnego procesu jak i innych procesów (wykorzystując PID)
  - deasemblację instrukcji
- **nie można** debugować z poziomu **kodu źródłowego**
- uruchomienie - wciśnięcie klawisza **Break**
- uruchamia się automatycznie po wywołaniu **Oops'a**

kdb - Alternatywa kgdb

- łata umożliwiająca **lokalne debugowanie**
- udostępnia m.in.:
  - modyfikację zmiennych,
  - analizę krok po kroku kodu wykonywalnego na procesorze,
  - zatrzymanie na wykonaniu konkretnej instrukcji,
  - zatrzymanie na dostępie (w tym również modyfikacji) do konkretnej lokacji w pamięci wirtualnej,
  - zatrzymanie na dostępie do rejestru,
  - analizę stosu dla aktywnego procesu jak i innych procesów (wykorzystując PID)
  - deasemblację instrukcji
- **nie można** debugować z poziomu **kodu źródłowego**
- uruchomienie - wciśnięcie klawisza **Break**
- uruchamia się automatycznie po wywołaniu Oops'a

kdb - Alternatywa kgdb

- łata umożliwiająca **lokalne debugowanie**
- udostępnia m.in.:
  - modyfikację zmiennych,
  - analizę krok po kroku kodu wykonywalnego na procesorze,
  - zatrzymanie na wykonaniu konkretnej instrukcji,
  - zatrzymanie na dostępie (w tym również modyfikacji) do konkretnej lokacji w pamięci wirtualnej,
  - zatrzymanie na dostępie do rejestru,
  - analizę stosu dla aktywnego procesu jak i innych procesów (wykorzystując PID)
  - deasemblację instrukcji
- **nie można** debugować z poziomu **kodu źródłowego**
- uruchomienie - wciśnięcie klawisza **Break**
- uruchamia się automatycznie po wywołaniu **Oops**'a

Co to jest UML?

Zbiór łat na jądro Linuxa, pozwalających na jego kompilację do postaci zwykłego pliku wykonywalnego

- Dostajemy wirtualną maszynę odseparowaną od macierzystego systemu
- Dzięki temu ewentualny oops nie kończy się restartem komputera
- ... a dane na naszym dysku pozostają bezpieczne
- Takie jądro można debugować zwykłym gdb!

Strona domowa: `http://user-mode-linux.sourceforge.net/`

UML — instalacja

1. Rozpakowujemy źródła wybranej wersji jądra (najlepiej gdzieś poza katalogiem /usr/src/linux):
```
host% mkdir ~/uml
host% cd ~/uml
host% tar -xjvf linux-2.4.31-1.tar.bz2
```
2. Aplikujemy do nich łaty UML:
```
host% cd ~/uml/linux
host% bzcat uml-patch-2.4.27-1.bz2 | patch -p1
```
3. Konfigurujemy i kompilujemy dla architektury 'um'
```
host% make menuconfig ARCH=um
host% make linux ARCH=um
```
4. w katalogu z UML umieszczamy obraz systemu plików
```
host% bunzip2 root_fs_slack8.1.bz2
host% mv root_fs_slack8.1 root_fs
```
5. Uruchamiamy poleceniem
```
host% ./linux
```

Własne moduły

Kompilujemy prawie tak jak zwykle, jedyne różnice to architektura systemu i opcje kompilatora:

```
 CFLAGS =
$(shell cd zrodla-uml/linux-2.4.31; make script \
        'SCRIPT=@echo $$(CFLAGS)' ARCH=um)
```

Instalacja jest trochę bardziej skomplikowana:

1. z poziomu systemu-gospodarza montujemy root_fs

   ```
   host# mount root_fs
   katalog-z-uml/mnt -o loop
   ```

2. Instalujemy w nim nasze moduły:

   ```
   host#
   make install \
           INSTALL_MOD_PATH=katalog-z-uml/mnt ARCH=um
   ```

3. Odmontowujemy obraz systemu plików
4. ...i uruchamiamy UML

UML i GNU Debugger

- Żeby można było UML-a debugować, musi on być skompilowany z włączonymi opcjami `CONFIG_DEBUGSYM` i `CONFIG_PT_PROXY`
- Uruchomienie UML-a pod kontrolą gdb powinno być proste: wystarczy wywołać go z parametrem debug:

```
host% ./linux debug
```

- Dostaniemy gdb w nowym oknie terminala, z pułapką ustawioną na wejściu do `start_kernel`.

UML i GNU Debugger

- Możemy przełączyć się na jakiś inny proces, który akurat jest w trybie jądra UML:

```
(UML gdb) detach
(UML gdb) attach <host pid>
(UML gdb) ...
```

gdzie <host pid> to numer interesującego nas procesu zwrócony przez polecenie **ps** z systemu-gospodarza.

- By powrócić do głównego procesu jądra UML, wystarczy podać pid, który nie odnosi się do żadnego z procesów wchodzących w skład UML-a, np:

```
(UML gdb) attach 1
```

UML i GNU Debugger

- Można też zacząć debugować już działającego UML-a, wysyłając mu sygnał SIGUSR1:

  ```
  host% kill -USR1 <uml-pid>
  ```

  (numer procesu widać przy starcie UML-a:
  `tracing thread pid = <uml-pid>`)

- ...wyskoczy nam okienko xterma z gdb w środku

Debugowanie modułów

Najprościej użyć skryptu **umlgdb** z zestawu narzędzi uml_utilities_xxxxx.tar.bz2 (skrypt ten znajduje się w podkatalogu tools/umlgdb/umlgdb).

Wystarczy:

1. Odnaleźć w nim definicję zmiennej MODULE_PATHS
2. Wstawić tam nazwę naszego modułu i jego lokalizację
   `"kerncalc" "/home/akmac/uml/devel/kerncalc.o"`
3. Przejść do katalogu z UML-em i uruchomić stamtąd zmodyfikowany skrypt
4. i gotowe —– w chwili załadowania naszego modułu gdb automatycznie rozpozna nowe symbole

Inne debuggery

Chcąc pracować z innym debuggerem, musimy UML-owi podać *explicite* jego pid: zamiast opcji **debug** używamy **debug gdb-pid=<pid-debuggera>**

Przykładowo, dla strace:

1. w oknie terminala wydajemy polecenie
```
host% sh -c 'echo pid=$$; read x; \
    exec strace -p 1 -o strace.out'
pid=<strace-pid>
```
2. w innym terminalu uruchamiamy UML-a...
```
host% ./linux debug gdb-pid=<strace-pid>
```
3. ... a w pierwszym wciskamy Enter

Odpluskwianie w Windows (KD, WinDbg).

System

Prezentacja omawia odpluskwianie w Windowsach opartych na jądrze **Windows NT**. W szczególności przykłady są opracowane z myślą o **Windows XP**.

Strona dla developerów

`http://www.microsoft.com/whcd/DevTools`

Gdzie należy zacząć?

Gdzie należy zacząć?:

- konfiguracja:

```
Panel Sterowania -> System -> Zaawansowane -> Uruchamianie i odzyskiwanie -> Ustawienia
```

- instalacja **Debugging Tools for Windows (DTW)** (dla wszystkich)
- instalacja **Windows Driver Development Kit (DDK)** (tylko dla wybrańców)

Słowniczek

Co należy wiedzieć?:

- wersje Windowsa: **retail** i **checked**
- błąd zatrzymania, błąd krytyczny, awaria systemu **bugcheck**, **STOP error message**, **system crash**, **kernel error**
- pliki zrzutów jądra: **Full**, **Kernel**, **minidump**
- **symbole**, pliki `.pdb`, **serwer symboli**
- **images** - obrazy
- **pool tags** - znaczniki puli pamięci

Narzędzia do testowania kodu

Narzędzia dla programistów sterowników i testerów:

- **Driver Verifier** (Weryfikator sterowników)
  Podstawowe narzędzie do znajdowania problemów w sterownikach przydatne od samego początku ich tworzenia, kiedy jeszcze można je łatwo poprawić w stosunkowo krótkim czasie. Testuje stabilność działania dowolnego sterownika w bardzo ciężkich warunkach.
- **PREfast**
  Statyczne narzędzie analizy kodu, które wykrywa błędy poprzez wykonywanie wszystkich możliwych ścieżek wykonań dla każdej funkcji (ciekawe, czy testuje wszystkie możliwe przeploty ;)).
- **Event Tracing** (Śledzenie zdarzeń)
  Dzięki temu narzędziu otrzymujemy informacje diagnostyczne o uruchomionym kodzie bez narzutu specjalnej kompilacji (z informacjami dla debugowania) i bez użycia odpluskwiacza.

Driver Verifier - możliwości

- testuje poprawne użycie **spinlocków** i **semaforów** typu mutex, wykrywając potencjalne **zakleszczenia**
- **wykrywa posiadanie spin locka podczas dostępu do stronicowanej pamięci**
- symuluje **małą ilość pamięci** (często odmawia przydzielenia nowej strony)
- sprawdza, czy przy **odłączaniu sterownika** wszystko jest poprawnie **zwalniane**
- może przydzielać pamięć ze specjalnie monitorowanego obszaru
- **zauważy wycieki pamięci**
- weryfikuje operacje I/O
- wykrywa niespójność dysku spowodowaną przez obserwowany sterownik

PREFace - możliwości

- **GUI**
  - lista ostrzeżeń (analogia do `pedantic` w gcc) - filtrowalna
  - po kliknięciu na ostrzeżeniu otwiera nam się okno z kodem
- kategorie wykrywanych błędów:
  - **Pamięć**
    (wycieki pamięci, wskaźnik NULL, niezainicjalizowa pamięć, znaczniki puli)
  - **Zasoby**
    (m.in. zwalnianie/posiadanie blokad)
  - **Użycie funkcji**
    (argumenty, niezalecane funkcje, nieodpowiedni poziom IRQL)
  - Styl kodowania w trybie jądra
  - Styl kodowania dla sterowników

Event Tracing for Windows - możliwości

- **zdarzenia** - operacje wejścia/wyjścia na dysku, błędy braku strony, itp. + zdefiniowane przez programistów specyficzne dla ich sterowników
- **ETW** - efektywny, ale trudny w użyciu;
  wykorzystanie preprocesora **WPP** do rozszerzenia możliwości **WMI**

Co używać w kodzie?

- makro **ASSERT**
- logowanie zdarzeń
- wywoływanie komunikatów o błędzie zatrzymania (**bugcheck**), gdy:
  - krytyczne dane są w beznadziejnym stanie
  - nie ma możliwości naprawienia błędów
  - sterownik jest wymagany do poprawnego działania systemu

Pakiet Debugging Tools for Windows (DTW)

- **WinDbg**
  Graficzny odpluskwiacz do debugowania zarówno kodu trybu użytkownika jak i trybu jądra.
- **KD** (Kernel Debugger)
  Aplikacja konsolowa do debugowania sterowników.
- **CDB** (Console Debugger)
  Aplikacja konsolowa do debugowania aplikacji trybu użytkownika i sterowników.
- **NTSD** (NT Symbolic Debugger)
  Odmiana CDB. Kopia w katalogu `system32`.

WinDbg

WinDbg - co to właściwie jest?

- nakładka graficzna na **kd.exe** i **cdb.exe**
- identyczne **rozszerzenia** i interfejs komend (dla **okna konsoli**)
- znacznie bardziej efektywny dla debugowania kodu źródłowego
- może być wolniejszy od odpowiedników konsolowych

Co obsługują odpluskwiacze?

- **Architektury procesora**: - x86, Itanium, x64
- **Wersje SO**: - Windows NT 4 i późniejsze (brak obsługi dla odpluskwiania jądra Win9x)
- **Protokoły**: - COM, 1394 (Wifi), USB 2.0 (beta)
- Istnieje możliwość odpluskwiania Windowsa działającego w środku **maszyn wirtualnych** (wykorzystanie **named pipe**).

Podstawowe, wysokopoziomowe możliwości odpluskwiaczy

- podłączenie się do **działającego procesu** aplikacji
  (możliwe podłączenie nieinwazyjne)
- **uruchomienie aplikacji** i rozpoczęcie jej debugowania
- podłączenie się do **innego systemu**
  (tryb jądra)
- po rozpoczęciu debugowania odpluskwiacz będzie **aktywowany** w momencie powstania **wyjątków** (jak dzielenie przez zero) lub **wywołania zdarzeń** w wykonywanym kodzie
- ustalenie własnych **punktów kontrolnych**
- **kontrola wykonania** procesów
  (*Step over*, *Step in* itp.)
- debugowanie **wieloprocesorowe/systemowe**
- debugowanie **zrzutów pamięci**
- **zdalne debugowanie**

OCA - Microsoft Online Crash Analysis

- darmowy serwis analizujący błędy, dla systemów Microsoftu począwszy od Windows XP
- zbiera dane bezpośrednio od systemu Windows, który się wykrzaczył u klienta
- w pełni automatyczny (działa 2-3 sekundy)
- wysyła zrzuty otrzymane od klienta do odpluskwiacza (standardowo minizrzuty)
- wywołuje `!analyze` w odpluskwiaczu - generuje `Bucket ID`
- zachowuje wynik analizy w bazie danych OCA
- jeśli dany `Bucket ID` ma rozwiązanie, wysyła je z powrotem do klienta
- **Dr. Watson** - standardowo instalowany z systemem, wysyła raporty do OCA