Debugowanie

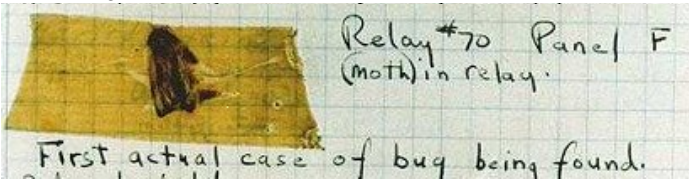
- Maciej Kalbarczyk
- Marcin Kosieradzki

Agenda

- Czym są bugi?
- Sposoby radzenia sobie z bugami
- Jak działa debuger?
- Jak działa profiler?
- Inne narzędzia

Pierwszy bug

- Był fizyczny (realny, sprzętowy (?)), nie programistyczny
- Spowodował awarię komputera
- Nikt nie wiedział dlaczego
- a problemem okazała się...



- Ćma powodująca zakłócenia
- Debugerem byli po prostu ludzie wypatrujący nieprawidłowości
- Do rozwiązania problemu użyto... pincety
- Był to rok 1945, od tego czasu wiele się zmieniło, ale jedno pozostało...
- Bugi są równie irytujące jak ten powyższy

Opowiemy czym są teraz i jak sobie z nimi można radzić
(już bez użycia pincety)

Rodzaje bugów

- Niespójny interfejs użytkownika
- Niespełnione wymagania
- Problemy wydajnościowe
- Niestabilne działanie programu lub uszkodzenie danych

Przyczyny powstawania bugów

- Nierealne terminy
- Podejście: "Najpierw koduj, potem myśl"
- Złe zrozumienie wymagań
- Niewiedza lub brak wprawy u programistów
- Ignorowanie kwestii jakościowych

Fazy projektu a bugi

Faza	Bugi	Koszt
Planowanie	• Nierealny harmogram	Wysoki
Projektowanie	• Złe zrozumienie wymagań • Błędy architektoniczne	Potencjalnie bardzo wysoki
Kodowanie	• Błędy programistyczne	Zależy od momentu wykrycia
Testowanie	• Błędy przy testowaniu	Wysoki

Unikanie błędów

- Udział zespołu przy ustalaniu harmonogramu
- Dobry kontakt z docelowym użytkownikiem
- Odpowiednie umiejętności i przeszkolenie
- Przeglądy kodu

Wykrywanie błędów

- Skuteczne mechanizmy obsługi błędów
- Asercje
- Wsparcie kompilatora
- Analiza statyczna kodu
- Testy poprawnościowe oraz wydajnościowe
- Pomiary wydajności

Analiza błędów

- Rejestrowanie przepływu kodu – tracing
- Crash Dump
- Ręczne śledzenie kodu (przy użyciu debugera)
- Przeglądanie zmian w repozytorium kodu oraz systemie śledzenia błędów

Wsparcie platformy - Windows

- Interfejs programistyczny
  - Tryb użytkownika - Win32 Debugging API
  - Tryb jądra - Kernel Debugger
- Symbole debugujące
- Wersja checked/debug systemu operacyjnego
- Liczniki wydajności
- Just-In-Time Debugging
- Opcja automatycznego uruchamiania wskazanej aplikacji pod debugerem
- Mechanizm minidumpów

Debugowanie w trybie użytkownika

- Debugowany jest konkretny proces
- Debuger ma dostęp do przestrzeni adresowej procesu debugowanego
- Korzysta z Win32 Debugging API (aplikacje natywne)
- Indywidualne środowiska dla języków interpretowanych oraz uruchamianych na wirtualnej maszynie
- Przykłady: Visual Studio Debugger, WinDBG (w trybie użytkownika), NTSD

Debugowanie w trybie jądra

- "Siedzi" pomiędzy systemem operacyjnym a procesorem
- Najczęściej debuguje się w ten sposób: sterowniki, aplikacje istotnie zależne od czasu
- Zatrzymanie debugera powoduje zatrzymanie systemu operacyjnego
- W większości przypadków wymaga dodatkowego komputera do debugowana
- Narzędzia livekd oraz SoftICE potrafią działać na tej samej maszynie
- Wbudowane wsparcie w NTOSKRNL.EXE (/DEBUG, /DEBUGPORT)
- Przykłady: KD, WinDBG (w trybie jądra), SoftICE

Konstrukcja debuggera na przykładzie Win32 Debugging API

Szkielet debugera

```
CreateProcess (…, DEBUG_PROCESS, …);
while (1 == WaitForDebugEvent(…))
{
    if (EXIT_PROCES…)
        break;
    ContinueDebugEvent(…);
}
```

Objaśnienie

- Uruchamiany jest dodatkowy proces (ze względów bezpieczeństwa), w którym będzie wykonywana aplikacja debugowana
- Przy uruchamianiu procesu konieczna jest flaga DEBUG_PROCESS lub DEBUG_ONLY_THIS_PROCESS
- WaitForDebugEvent odbiera kolejne zdarzenia z procesu debugowanego
- ContinueDebugEvent powoduje kontynuację wykonania procesu debugowanego po odebrau zdarzenia
- Zamiast tworzyć nowy proces można podłączyć debugger do istniejącego procesu przy użyciu DebugActiveProcess, konieczny jest wtedy przywilej SE_DEBUG_NAME.

Otrzymywane zdarzenia

- Utworzenie/Zakończenie procesu lub wątku
- Wystąpienie wyjątku
- Załadowanie/Wyładowanie biblioteki
- Otrzymanie komunikatu debugującego
- Błąd systemowy

Dostęp do pamięci

- Uchwyt do procesu potomnego posiada flagi PROCESS_VM_READ oraz PROCESS_VM_WRITE, które zezwalają na dostęp do pamięci wirtualnej procesu potomnego przy wykorzystaniu ReadProcessMemory oraz WriteProcessMemory
- Mechanizm copy-on-write zabezpiecza inne procesy wykonujące ten sam kod przed modyfikacją kodu
- Przy otrzymaniu zdarzenia OUTPUT_DEBUG_STRING napis należy czytać z przestrzeni adresowej procesu debugowanego
- Funkcje VirtualQueryEx, VirtualProtectEx pozwalają na zarządzanie zabezpieczeniami stron pamięci.

Breakpointy

- Pozwalają na przerwanie wykonania procesu po natknięciu się na wskazaną instrukcję
- Ustawienie breakpointa polega na podmienieniu instrukcji (zapamiętując ją!) na instrukcję przerwania debuggera – dla i386będzie to: INT 3 (o kodzie 0xCC)
- W momencie kiedy procesor wywoła INT3 zwracane jest zdarzenie debuggera: wyjątek EXCEPTION_BREAKPOINT
- Po odebraniu wyjątku należy podmienić spowrotem instrukcję na tę zapamiętaną
- Do ewentualnego przywrócenia breakpointa można wykorzystać mechanizm SINGLE STEP (dla i386: ustawienie flagi TRAP)
- SINGLE STEP wykonuje jedną instrukcję procesora, a następnie powoduje wyjątek EXCEPTION_SINGLE_STEP

Symbole debugujące

- Pozwalają odczytać:
  - Typy
  - Nazwy symboli
  - Nazwy plików źródłowych i numery linii kodu
- Pozwalają na analizę stosu wywołań, nawet w przypadku zoptymalizowanego kodu
- Najczęściej przechowywane są w plikach PDB (Program DataBase)
- Biblioteka DBGHELP.DLL ułatwia wykorzystanie plików PDB.

Minidump

- Pozwala odzyskać stan aplikacji z innego komputera
- Użyteczny przy analizie błędów powstałych u klienta
- Wsparcie ze strony DBGHELP.DLL – funkcja MiniDumpWriteDump, MiniDumpReadDump

Rodzaje minidumpów

- Podstawowy – stosy wywołań
- Z segmentami danych – stałe w kodzie
- Z całą pamięcią
- Z informacjami o uchwytach
- Przefiltrowany – usuwane są prywatne informacje
- Inne (enumeracja MINIDUMP_TYPE)

Wsparcie platformy - Linux

Błędy

- Oops

Jak sobie radzić?

- Nie robić błędów
- Printk
- Pliki Core (core dump)
- Ptrace
- GDB
- KGDB
- UML

Wsparcie platformy - Linux

Oops

Czym jest Oops?

Gdy jądro wykryje istnienie poważnych nieprawidłowości wywoływany jest „oops”. Ma on dwie główne funkcje:

- Pokazać użyteczne informacje, które mogą być użyte do zdiagnozowania przyczyny problemu.
- Spróbować zapobiec by jądro nie wymknęło się spod kontroli i by nie spowodowało dużo poważniejszych konsekwencji (jak uszkodzenie danych czy nawet sprzętu).
- Oops może tworzyć raport do pliku (Oops.file), którego oglądanie wspomaga program **ksymoops**

Wsparcie platformy - Linux

Przykład

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

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

Wsparcie platformy - Linux

Przykład

Jak widać na poprzednim przykładzie, uzyskujemy następujące informacje:

- Krótka informacja o przyczynie oopsa
- Numer oopsa
- Ustawienie flag procesora
- Zawartość głównych rejestrów
- Nazwę procesu, który spowodował oopsa
- Kilkanaście ostatnich ramek stosu
- Ślad i kod po którym nastąpił oops

Wsparcie platformy - Linux

Pliki Core

- Plik core (core dump) tworzony jest w wyniku nieprawidłowego działania programu, lub wykonania niedozwolonej operacji
- W pliku tym zapisany zostaje stan pamięci w momencie wykonania niedozwolonej operacji (plus informacje dla debugera)
- W Linuksie można ograniczyć maksymalną wielkość pliku core (limit coredump 10240)
- Typy błędów:

- Bus Error (niezgodności typów, błędy I/O, dostęp do nieistniejących urządzeń)
- Memory Fault (błędy dostępu do pamięci, zakresy, null pointer, złe rekurencje)
- IOT Trap/ BPT Trap (pułapki (Trap) generowane przez procesor)
- Floating Exception (dzielenie przez zero, konwersje typów)
- Segmentation Fault (naruszenie segmentów pamięci)
- Illegal Instruction (bezsensowne (zdaniem procesora) i nielegalne instrukcje)

Wsparcie platformy – Linux – PRINTK

kernel/printk.c

- Jest funkcją, która powinna dać wywołać się z dowolnego kontekstu
- Wypisuje do pliku logów bądź na konsolę informację przekazaną jako argument
- Jest prosta w użyciu
- Działa w trybie jądra
- Nie używa pośredników
- Próbuje zdobyć dostęp do konsoli (semafor) i wypisać komunikat, a gdy to się nie udaje i tak umieszcza go na końcu bufora

Wsparcie platformy – Linux – PRINTK

kernel/printk.c

Funkcja pritnk posiada domyślnie 8 „poziomów” komunikatów (od najmniej ważnego):

- 7 (KERN_DEBUG) – komunikaty często używane przy rozwijaniu jądra
- 6 (KERN_INFO) – informacje
- 5 (KERN_NOTICE) – istotne notatki
- 4 (KERN_WARNING) – ostrzeżenia
- 3 (KERN_ERR) – błędy systemu
- 2 (KERN_CRIT) – błędy krytyczne
- 1 (KERN_ALERT) – błędy o znaczeniu alarmowym
- 0 (KERN_EMERG) – sytuacje awaryjne

Wsparcie platformy – Linux – PTRACE

- Jest funkcją systemową umożliwiającą śledzenie procesów
- Umożliwia wstawianie breakpoint'ów i zatrzymywanie procesu śledzonego przez proces śledzący (oraz np. zmianę rejestrów procesu śledzonego)
- Umożliwia przechwytywanie sygnałów przez proces śledzący (przed wykonaniem sygnału jądro sprawdza flagę śledzenia i przekazuje sterowanie do procesu śledzącego)
- Może być użyta do podłączenia się do działającego procesu
- bądź (przy pomocy wait, fork i exec) ojciec może śledzić swoje dziecko, które samo zgłosiło chęć bycia śledzonym

Dany proces może być śledzony tylko przez jeden proces.

Wsparcie platformy – Linux – PTRACE

Informację o śledzeniu przechowuje zmienna ptrace w task_struct

- **linux/sched.h**

```
#define PT_PTRACED      0x00000001 //informacja w task_struct o byciu śledzonym

struct task_struct {
...
unsigned long ptrace;
}
```

przy śledzeniu zmieniany jest aktualny rodzic procesu śledzonego

kernel/ptrace.c

```
56 int ptrace_attach(struct task_struct *task)
...
81        task->ptrace |= PT_PTRACED;
...
88                REMOVE_LINKS(task);
89                task->p_pptr = current;
90                SET_LINKS(task);
```

```
102 int ptrace_detach(struct task_struct *child, unsigned int data)
...
111       child->ptrace = 0;
...
114       REMOVE_LINKS(child);
115       child->p_pptr = child->p_opptr;
116       SET_LINKS(child);
```

Wsparcie platformy – Linux - STRACE

- Strace to narzędzie badające interakcje programu z jądrem Linuksa
- Uruchomienie strace nazwa_programu wykonuje nazwa_programu, wypisuje funkcje systemowe wywoływane przez proces oraz sygnały jakie do procesu dotarły
- Linie wyjścia zawierają nazwę wywołania systemowego, parametry oraz wartość zwróconą np.

```
[mk209470@violet10 debug]$ strace cat /dev/null
execve("/bin/cat", ["cat", "/dev/null"], [/* 46 vars */]) = 0
...
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0pQ\1\000"..., 512) = 512
...
open("/dev/null", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0666, st_rdev=makedev(1, 3), ...}) = 0
read(3, "", 4096)                       = 0
close(3)                                = 0
close(1)                                = 0
exit_group(0)                           = ?
```

Wsparcie platformy – Linux - GDB

GNU Project Debugger – pozwala na:

- Uruchomienie programu wraz z podaniem mu odpowiednich argumentów
- Zatrzymanie działania programu w określonym miejscu
- Sprawdzenie parametrów programu na danym etapie
- Zmianę niektórych parametrów w czasie działania programu
- Tworzenie plików Core z programu
- Analizę utworzonych plików Core
- Przechwytywanie i obsługę sygnałów wysyłanych do programu śledzonego
- Przeglądanie i obsługę wątków procesu

Wsparcie platformy – Linux - GDB

GDB – Jak uruchomić

- Kompilujemy program z opcją dołączania symboli

(gcc –g program.c –o program)

- Uruchamiamy śledzenie programu poprzez GDB

(gdb program)

- Wstawiamy breakpointy (na numer lini, adres szesnastkowy, bądź nazwę funkcji)

(break 12, break *0x001573ca, break printf)

- Uruchamiamy program

(run)

Wsparcie platformy – Linux - GDB

Program zatrzyma się na najbliższym ustawionym breakpoint'cie

Możemy między innymi:

- Obejrzeć zawartość zmiennych, rejestrów

(print nazwa_zmiennej, print $eax)

- Zmienić zawartość zmiennych, rejestrów (set $eax=1)
- Obejrzeć zawartość ramek stosu (backtrace)
- Zdezasemblować funkcję (disas printf)
- Sprawdzać adresy lini (info line 15)
- Wypisać kod źródłowy symbolu (list main)
- Wznowić, zatrzymać program (next 10, stop)

Wsparcie platformy – Linux - KGDB

- KGDB nie jest graficzną nakładką na GDB (jest nią XXGDB)
- Jest debugerem podobnym do GDB, ale pozwala debugować Jądro Linuksa
- Umożliwia m.in. wstawianie breakpoint'ów do jądra, uruchamianie go krok po kroku oraz obserwowanie zmiennych
- GDB działa lokalnie
- KGDB wymaga do pracy dwóch komputerów połączonych łączem szeregowym
- Na jednym komputerze uruchamiamy debugowane jądro
- Na drugim z nich odpalamy debuger
- Toczone są prace nad kontrolą nie poprzez łącze szeregowe, a poprzez sieć ethernet

Wsparcie platformy – Linux - KGDB

- KGDB dla komputera debugowanego dostępne jest poprzez patch na jądro Linuksa
- Przy uruchamianiu na testowanym komputerze, jądro zaczeka na nawiązanie połączenia z maszyny testującej
- Możliwe jest także debugowanie modułów, przy czym należy je kompilować z opcją –g dla kompilatora gcc
- Ładowanie modułów na maszynie testowanej odbywa się standardowo (insmod/ modprobe)
- Można także załadować moduł na maszynie testującej (posługując się dostarczonym skryptem)

Wsparcie platformy – Linux - KGDB

- Na maszynie testującej uruchamiamy debuger GDB
- Łączymy się z maszyną testowaną poprzez port szeregowy (target remote /dev/ttyS1)
- Przekazujemy do maszyny testowanej chęć załadowania jądra (continnue)
- Możemy używać GDB podobnie jak dla zwykłego programu:
  - przerywać pracę jądra w dowolnym momencie (ctrl+c)
  - ustawiać breakpoint'y
  - odczytywać zawartość zmiennych i rejestrów
  - odczytywać stos
  - odczytywać informację o wątkach jądra

Wsparcie platformy – Linux - UML

- UML, czyli User-Mode-Linux to jądro uruchamiane w przestrzeni użytkownika jak zwykły program
- UML przypomina maszynę wirtualną
- Posiada własny niezależny scheduler i system pamięci wirtualnej
- Może być (od niedawna) uruchamiane pod samym sobą
- Współpracuje m.in.. z urządzeniami blokowymi, konsolami, portami, siecią, dźwiękiem, urządzeniami USB, urządzeniami PCI
- Wspiera wieloprocesorowość (SMP)
- Wymaga specjalnej kompilacji (tzn. nie uruchamiamy zwykłego jądra w trybie UML, ale kompilujemy jądro UML, a uruchamiamy jak zwykły program)

Profile'owanie

- Optymalizacja wydajności
- Opiera się na powiązaniu fragmentów kodu programu ze statystykami wykorzystania zasobów
- Zazwyczaj korzysta z instrumentacji kodu

Instrumentacja kodu – podejścia

- Ingerencja w niezoptymalizowany kod assemblerowy
- Wykorzystanie możliwości kompilatora, np. przełączników /Gh /GH kompilatora Visual C++
- Wykorzystanie Profiling API dla kodu platformy .NET

.NET Profiling API

- Pozwala monitorować:
  - Uruchomienie/Zakończenie CLR, Domen aplikacji
  - Załadowanie/Wyładowanie Zestawów, Modułów, Klas
  - Aspekty związane z kodem natywnym, COM
  - Działalność JITtera
  - Działalność Garbage Collectora
  - Wchodzenie/Opuszczanie metod
  - Obiekty sterty zarządzanej
  - Wyjątki
  - Wątki
  - Wykorzystanie Remotingu
- Funkcjonalność udostępniana jest przez interfejs ICorProfilerInfo

Inne techniki oraz narzędzia

DLL Injection

- Technika pozwala uruchomić kod w już działającym procesie
- Uruchamiany jest CreateRemoteThread ze wskaźnikiem na funkcję LoadLibrary, co powoduje załadowanie wskazanej biblioteki
- W czasie ładowania biblioteki uruchamany jest DllMain, w którym wykonujemy nasz kod

Fault Injection

- Testowanie programu w warunkach ekstremalnych
- Symulowanie wszelkiego rodzaju błędów
- Umożliwia przede wszystkim znalezienie błędów w kodzie obsługi błędów
- Przykład: Holodeck

Wykrywanie zakleszczeń

- Potrzebne informacje na temat zakleszczenia: identyfikatory zakleszczonych wątków, obiekty synchronizujące, funkcja WINAPI powodująca zakleszczenie wraz z parametrami
- Wykorzystuje technikę podmiany adresów funkcji systemowych związanych z synchronizacją w tablicy IAT (import address tables)

Application Verifier

- Testuje:
  - Wykorzystanie niebezpiecznego API
  - Wyjątki – czy aplikacja nie "zamiata pod dywan"
  - Uchwyty
  - Sterta
  - Blokady
  - Pamięć
  - Wykorzystanie Thread Local Storage
- Potrafi symulować:
  - Brak zasobów
  - Pracę przy ograniczonych przywilejach
- Wykorzystuje debuger, np. WinDBG
- Wykorzystywany w testach platformowych np. Designed for Windows XP

Analiza statyczna kodu

- Analizuje kod źródłowy albo kod pośredni
- Wymusza stosowanie pewnych standardów
- Wykrywa potencjalne błędy
- Możliwość dodawania własnych reguł
- PREfast dla kodu C/C++
- FXCop dla kodu .NET

Pytania?

Bibliografia

- John Robbins - Debugging Applications for Microsoft .NET and Microsoft Windows
- Microsoft Developer Network Library – http://msdn.microsoft.com/library/
- Mark Russinovich, Bryce Cogswell – SysInternals – http://www.sysinternals.com/
- http://kgdb.linsyssoft.com
- Źródła Linuksa 2.4.31
- http://www.urbanmyth.org/linux/

Dziękujemy za uwagę