Pointer und Array Bugs in C / C++: Werkzeuge jenseits des Debuggers

Klaus Kusche, Juni 2013

Inhalt

- Was ist das Problem?
- Untaugliche Hilfsmittel: Debugger, ...
- *“Code mit Sicherheitsgurten”* = Compiler-basierte Werkzeuge:
  Mudflap, Bgcc / MIRO, Asan, ...
- *“Quarantäne für Schmutzcode”* = Binärcode-basierte Werkzeuge:
  Valgrind, Purify / Quantify, ...

Das Problem (1)

"Insgesamt zwölf Sicherheitslücken schließt die neue Quicktime-Version 7.7.4 für Windows. Bei allen handelt es sich um kritische Fehler bei der Verwaltung von Speicher, ..."

http://www.heise.de/newsticker/meldung/Apple-fixt-Quicktime-Loecher-im-Dutzend-1868245.html

(23. Mai 2013)

Das Problem (2)

"AddressSanitizer (ASAN) is a tool
for finding memory problems
and has been used to find
thousands of memory errors in Chromium
over the last two years."

http://blog.chromium.org/

(20. Mai 2013)

Ursache 1: “Böse” Pointer

- **NULL**-Pointer
- *Uninitialisierte* Pointer:

  - Einzelne Pointer-Variablen (findet ev. der Compiler)
  - Array von Pointern (findet er nicht!)
  - Pointer in `malloc`-Block (findet er auch nicht!)

- *“Use after return”*:

  Pointer auf lokales Array / lokale Struktur
  überlebt das Return der Funktion (==> lokale Daten weg!)

  - … als *Returnwert*
  - … in *globalen* Daten, Heap-Daten, …

Ursache 2: Arrays & Pointer

- *Array-Überlauf*:
  - "*Off-by-one*"-Fehler in Schleifengrenzen
    und Größen- bzw. Sicherheits-Abfragen
  - Ungeprüfte, *zu große Strings* und Input-Daten
  - *Fehlendes* `'\0'` in Strings
- *Integer-Überlauf oder negative Werte*
  in Index- und Größen-Berechnungen
- *Uninitialisierte* Variablen
  in Index- oder Pointer-Rechnungen

Ursache 3: `malloc` & `free`

- Schreiben *über das Ende* eines `malloc`-Blocks
- Zugriff *nach* dem `free` (*"use after free"*)
- *Doppeltes* `free` desselben Blocks
- `free` mit "*falschem*" Pointer:
  - Pointer zeigt *mitten* in den Block (statt auf den Anfang)
  - Pointer zeigt auf den *Stack*, ist *ungültig*, ...
- *Mischung* der Heap-Funktionen:
  `new` + `free`, `malloc` + `delete`, `new` + `delete []`
- (Memory *Leaks*)
- (Memory *Fragmentation*)

Ursache 4: “Dunkle Ecken” von C

- `printf`-*Argumente* passen nicht zum Format
- Funktionen mit *variabel vielen Arg's* (kein Typcheck!)
- Alter 32 bit Code auf 64 bit portiert:
  Pointer (64) nach *Cast* auf `int` (32) und zurück!
- *Nicht-Pointer-Daten* als Pointer verwendet:
  - Falscher Fall in *Union's*
  - “Schmutzige” *Casts*
- C++: Objekt passt nicht zum Pointertyp
  z.B. Cast eines Pointers *Basisklasse ==› abgel. Klasse*

  (verschiedene Objekt-Größen, falsche V-Table!)

Das Schlimme daran ...

- Fehlverhalten ist oft

zufällig & nicht reproduzierbar!

- Manchmal gar kein Absturz, nur

“heimlich” falsche Daten!

- Lauter potentielle

Sicherheits-Lücken!

Hilft der Debugger? (1)

Nein:

Ursache (= falscher Zugriff)
und ***Wirkung*** (= Absturz, ...)

sind *räumlich* (Codestelle) und
zeitlich (Zugriffs- und Absturz-Zeitpunkt)
meist ***weit auseinander***

und *ohne "logischen" Zusammenhang*!

Der Debugger sieht *nur den Moment der Wirkung*,
kann *nicht in die Vergangenheit* zur Ursache blicken.

Hilft der Debugger? (2)

Außerdem:

- Oft *gar kein Absturz*

  ==> Debugger “springt nicht an”

- Manchmal *total korrupter Speicher*

  ==> Debugger kann den Stack nicht mehr lesen

  ==> Debugger kann (fast) nichts anzeigen

Helfen “prüfende” `malloc`-Libs?

Nein:

- Auch *kein Blick zurück* zur Ursache!
- Prüfung *fängt viel zu wenig*:
  Ev. Double `free`, `malloc` Bounds Violations
  ==> Viele Fehler bleiben *unbemerkt!*

  oder *erfolgt zu spät*: Erst beim `free`!
  ==> Absturz *vorher!*

Aber:

- Sind Tool der Wahl für *Memory Leaks!*
  (viel schneller als die folgenden Tools)

Auch Fehlanzeige: gcc SSP

"Stack Smashing Protector"

Arbeitsweise:

"Guard word" am Stack unter jeder Return-Adresse,
Prüfung unmittelbar vor jedem Return

Reines Sicherheits-Tool, kein Fehlersuch-Tool!

Vorteil: *Schnell*, < 5 % Overhead (Zeit & Speicher)

Nachteil:

- Fängt *nur Stack-Exploits*
  (= bewusster, großer Überlauf lokaler Arrays)
- *Zu spät* (beim Return!): Kein Hinweis auf die Ursache!

Vor dem “schweren Gerät” ...

- Mit *maximalen Warnungen* compilieren!

  (und zugleich *maximaler Optimierung*
  für Datenfluss-Analyse!)

- *Statische Programmanalyse-Tools* verwenden

  “Datenfluss-Analyse”,
  “Value & Range Propagation”,
  “Symbolic Execution”, ...

  (`pclint`, `splint`, `cppcheck`, ...)

==> *findet manche Schlamperei im Code!*
(aber bei weitem nicht alles!)

Compiler-basierte Werkzeuge (1)

- … fügen bei jedem Call / Return *Buchführungs-Code für alle lokalen Variablen / Arrays* ein
- … ersetzen die `malloc`- / `free`-*Library* (zusätzliche *Checks*, *Buchführung*, …)

==> **Ziel:**

Liste aller gültigen Speicherblöcke

==> **Nebeneffekt:** Liste aller *Memory Leaks*

- … fügen *Prüf-Code* bei *jedem Pointer- und Array-Zugriff* ein

Compiler-basierte Werkzeuge (2)

Probleme dieses Verfahrens:

- *Keine Prüfungen*
  in vorcompiliertem *Library-Code*, in System Calls
  ==> stürzt dort trotzdem ab!
- Keine Erkennung von Pointern
  auf *"falsche" gültige Daten*:
  - *Nachbar-Variable* bei Array-Überläufen (gültig!)
  - Oft bei "*use after free*" und "*use after return*", ...
    (wenn *neue gültige Daten* an derselben Stelle sind)
- Keine Erkennung *uninitialisierter* Variablen,
  uninitialisierte Pointer nur mit Glück (wenn falsch)

gcc Mudflap

Mudflap = Spritzschutz, Schmutzfänger

- Seit Jahren im Standard-gcc
- Macht das genau Besprochene: *Liste gültiger Blöcke*

Nachteile:

- Fängt nicht alles (==> vorige Folie)
- **Langsam:**

*Laufzeit * 10-30*

- Nicht wirklich C++-geeignet

bgcc & MIRO (1)

"Bounds-checking gcc"

- "*Akademischer*" gcc-Patch (Doktorarbeit, ICL 1995)
  nie "produktionsreif" gemacht, seit >8 Jahren *toter Code*
  oft inkompatibel mit aktuellen Systemen
- *Nur C* (MIRO: Nachfolger für C++, schlechter, auch tot)
- Noch etwas *langsamer*!

Aber:

- *Beste Erkennungsleistung* aller Compiler-Tools!
  (außer bei Fehlern in Lib's perfekt)
- Sehr gut *lesbare Fehlermeldungen!*

bgcc & MIRO (2)

Idee:

- *Zusätzlich* zur Tabelle gültiger Blöcke:

  Zu jedem einzelnen Pointer merken,
  auf welche Variable (Array, `malloc`***-Block, ...) er zeigt***

  ==› Findet *uninitialisierte* Pointer
  ==› Findet "*use after free*", "*use after return*",
  Pointer, die ins *Nachbar-Objekt* wandern, ...

- Schon die Index- bzw. *Pointer-Rechnung prüfen*,
  nicht erst den Zugriff

  ==› frühere & genauere Fehlermeldung bei
  Erzeugung (nicht Verwendung!) des "*bad Pointers*"

ASAN (1)

"Address Sanitizer"

- *Google*-Projekt, Open Source
- Standardmäßig inkludiert in **`llvm`** ab 3.1, **`gcc`** ab 4.8
- Riesige ***byte-weise*** *"gültig"-Tabelle*, keine Blöcke:

  Ganzer Adressraum wird direkt abgebildet!

  ==> *Nur 4 Instruktionen* für die Pointer-Prüfung
  (`shift`, `add`, `test`, `branch`)!
  ==> *Extrem schnell:* Laufzeit-Overhead **<= 2**

  ==> *Virtueller* Speicherbedarf **16 TB** !!!
  ==> Aber *geringster Real-Speicherbedarf* aller Tools

ASAN (2)

- *Änderung des Daten-Layouts*
  von Stack & globalen Daten:

  Ungültige ***"Guard words"***
  um jedes einzelne Array (lokal / global)
  und auch um jeden `malloc`-Block

  ==> Sprechen an, *bevor* der Pointer / Index
  den *nächsten gültigen Bereich* erreicht!

  ==> *Erkennt viel mehr* als Mudflap

Andere Compiler

Parasoft Insure++:

- *Kommerzielles Produkt*, $$$
- ***Source Code Instrumentation***
  ==> Zuerst Insure, dann normaler C-Compiler!
- Fängt "alles" (incl. Zugriffe auf uninitialisierten Speicher!)

SAFECode: (Status ???)

- *Forschungsprojekt* (UIUC), Open Source, `llvm`-basiert
- Umfangreiche *statische Programm-Analyse* (nur C):

  ==> Laufzeit-Zugriffsprüfungen
  so weit wie möglich *wegoptimieren!*

Binärcode-Tools: Valgrind (1)

- Open Source
- Universelles ***"x86-Binärcode-Interpreter-Framework"***
  (in Wahrheit viel komplexer)
- **Plugin-Mechanismus:**
  - Code *vor und nach*
    jeder ausgeführten Instruktion einfügbar!
  - Ein Dutzend Plugins werden mitgeliefert
    (Profiling, Cache-Simulation, Race Check, ...)
  - Hier relevante Plugins:
    - **memcheck**
    - **SGcheck**

Valgrind (2)

Vorteil:

- Arbeitet auf normalen, *unmodifizierten* Exe's und Lib's:

  ==> Braucht keinen Source, keine Debug-Info
  ==> Kein Recompile / Relink nötig

- *Prüft auch alle Lib's!*

Nachteil:

- Kann Memory Layout usw. nicht ändern
  ==> z.B. *keine "Guard Values"* einschieben

- *Langsam* (je nach Plugin: Faktor 10 - 40)

memcheck-Plugin (1)

- Buchführung *für jedes Byte* im Speicher:
  - ***"Gültig"*** (in einem `malloc`-Block?)
  - ***"Initialisiert"*** (jemals geschrieben?)
- Ersetzt `malloc`-Lib ==> Malloc-Block-Buchführung

  ==> Auch komplettes *Memory Leak Listing*
- Prüft *alle Instruktionen*, die auf Speicher zugreifen

  ==> Findet als einziges Tool
  auch *Lesen uninitialisierter Daten*
- Prüft *alle Syscall-Parameter*, die Pointer sind!
  (auch einzigartig)

memcheck-Plugin (2)

Nachteile:

- Prüft

  nur den Heap (aber das fast perfekt)

- … nicht Stack & globale Daten
  (==> auch "Use after Return" nur eingeschränkt!)
- Fängt keine Zugriffe auf *"falschen"* gültigen Block
- *Groß & Langsam:*

  - Daten * ~3
  - Codesize * ~15
  - Laufzeit * ~10-30

SGcheck-Plugin

- Braucht *Debug-Info* im Code!
  (u.a. für Lage und Größe
  aller lokalen / globalen Arrays)
- Prüft *nur lokale & globale Daten*
  (aber *nicht* den Heap!)
- … so wie bgcc: Tabelle für jeden Pointer:
  "Auf welches Objekt zeigt er?"
- Erkennt u.a.
  "*Use after return*",
  "*Off by one*", Array-Überschreitungen
  Pointer die in anderes Objekt wechseln

IBM Purify / Quantify

- *Kommerzielles Produkt* von IBM / Rational
  (sehr teuer: ›› 5000 € pro Jahr und Rechner)
- Marktführer!
- Arbeitsweise, Erkennungsleistung, Geschwindigkeit:
  Vergleichbar mit Valgrind
- **Aber:**

  Kein Interpreter,
  sondern ***Binary-to-Binary-Compiler!***

  ==› Separater *Code-Instrumentierungs-Lauf* (langsam!)
  für alle Exe's und Lib's *vor der Ausführung*

Andere Tools für Binärcode

DrMemory:

- Open Source, von Google
- Ähnlich Valgrind, schneller, weniger universell
- Nur für 32-bit-Code!

BoundsChecker:

- $$$, von Borland / Micro Focus
- Konkurrenzprodukt zu Purify / Quantify
- Nur für MS Visual Studio

“The end”

Fragen?