

„Speicher-Debugging“ mit C und C++

Hayati Aygün
(Dipl. Informatiker)

eMail:
h_ayguen@web.de

Speicherdebugging?

- Fehler-Ursache:
 - Lesen von „ungewollten“ Werten
 - Schreiben an „falsche“ Adressen
- Auswirkung:
 - Das gewünschte wird nicht richtig berechnet
 - Programmcode wird evtl. modifiziert, was zu Abstürzen führen kann
- Problem:
 - Ursache und Wirkung haben nicht notwendigerweise einen Zusammenhang !
 - Wirkung ist scheinbar „zufällig“
- Wunsch: ANHALTEN im Debugger

Beispiel 1

```
void Begruessen()  
{  
    char acName[10];  
    printf("Bitte Name eingeben: ");  
    scanf("%s", acName);  
    printf("Hallo %s\n", acName);  
}
```

Was ist hier falsch?

Lösung

- Funktion funktioniert oft ohne Probleme
- Bei Eingabe von mehr wie 10 Buchstaben nicht vorhersehbares Verhalten
- Bei wesentlich grösseren Eingaben kann Funktion nicht zum richtigen Aufrufer zurückkehren, da Rücksprung-Adresse auf Stack überschrieben wird!

Beispiel 2

```
int main( int argc, char** argv ) {  
    const char *quelle = "Hello World\n";  
    char *ziel = (char*)malloc( sizeof(char)  
                                * strlen( quelle ) );  
  
    if (!ziel)    return 1;  
    strcpy( ziel, quelle );  
    return 0;  
}
```

Und was ist hier falsch?

Lösung

- Fehler nicht ganz so einfach sichtbar
- Es müsste Speicher mit Länge `strlen() + 1` reserviert werden.
- Fehler wirkt sich „wahrscheinlich“ wesentlich seltener aus – möglicherweise gar nicht!

Buffer Overflow und ähnliche

Auszug aus Stack:

<u>Adresse</u>	<u>Inhalt</u>
0x1000	Funktions-Rücksprungadresse
0x9990	int Feld[4]; // lokales Array

Stack wächst nach unten (d.h. die Adressen von Variablen in aufgerufenen Funktionen sind kleiner als die des Aufrufers)

damit Feld[5] == Rücksprungadresse !

Folge: nach Rücksprung im „Nirwana“ oder Schlimmer!

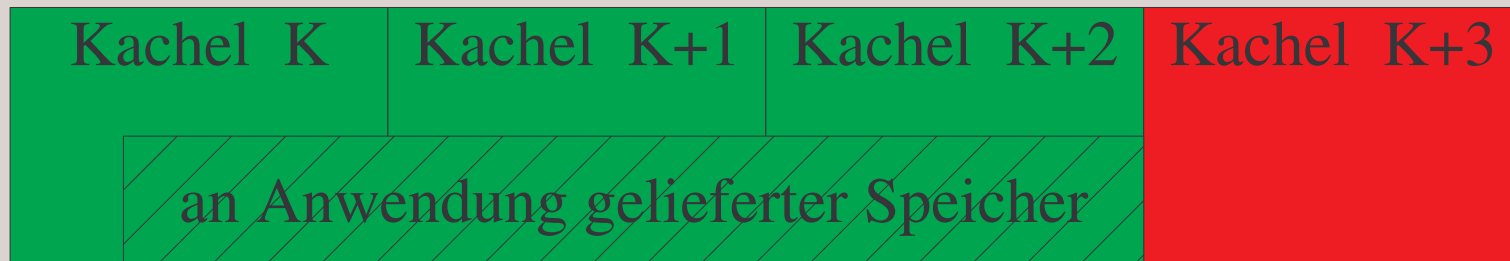
Gegenmittel?

- Sichere Hochsprachen verwenden?
Bei geringen Ressourcen (CPU, RAM)
oft keine Alternative
- Compuware BoundsChecker (Windows)
Instrumentation bei Kompilierung
- valgrind (Linux)
Emulation des x86 Prozessors
- DUMA: Fork von EFence (Linux + Windows)
Nutzung der Memory Management Unit
Einrichtung von Zugriffsschutz auf einzelne
Kacheln (Speicherseiten bzw. Pages)

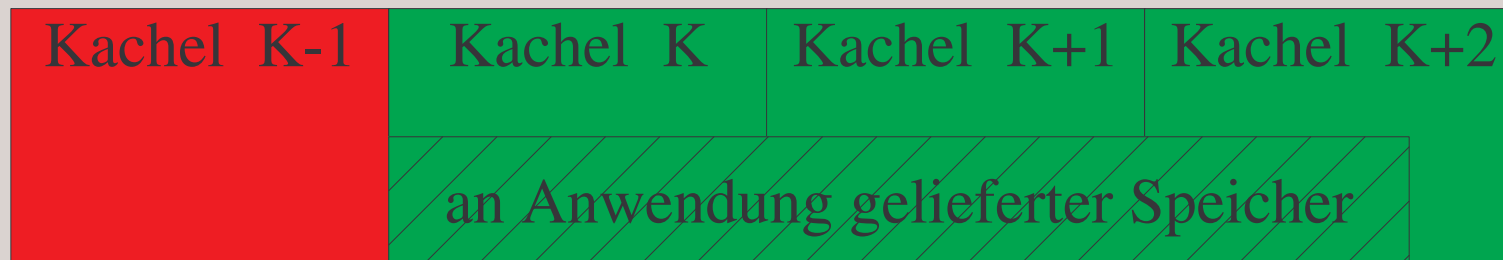
DUMA – Funktionsweise

- „Überladen“ von malloc(), free() Funktionen (durch Link-Reihenfolge) und C++ Operatoren new, delete, ..
- Auslegen des Speicheranfang/ende auf Kachelgrenze
- Schutz der vorherigen/nächsten Kachel mittels MMU
- Programm hält exakt auf der Anweisung, die eine Schutzverletzung auslöst :-)
- MMU schützt somit **nur** dynamisch auf Heap angelegten Speicher :-)

Speicher - Kachelgrenzen



bzw.



Zugriff auf grüne Kacheln O.K., auf rote ein K.O. !
Kachelgrösse bei intel's x86 CPUs: **4096 Byte**

Weitere Möglichkeiten: Schutz nach Freigabe

vom Anwendungsprogramm mittels

`free()` / `delete` / `delete[]`

freigegebenen Speicher intern behalten
und Zugriff darauf schützen

Weitere Möglichkeiten: SpeicherLeck-Suche

- Speziell für Serverprozesse mit langer Laufzeit: bei Lecks müssen diese „**neu gestartet**“ werden.
- Suche nach Lecks kein Problem, da Speicherallokationen durch Efence behandelt wird.
- Mit speziellen Makros auch innerhalb von Programmteilen – sonst bei Programmende
- Meldung der Lecks mit Dateiname, Zeile der Allokation

Weitere Möglichkeiten: Prüfung der Freigabefunktion

Freigabe passt nicht zur Allokationsmethode:

- malloc() / calloc() / realloc() / strdup() / free()
- new / delete
- new[] / delete[]

Mischen der Funktionen ist NICHT erlaubt!

Weitere Möglichkeiten: Speicherinitialisierung

- einige Compiler initialisieren Speicher auf 0; besonders, wenn eine Debug Version erstellt wird!
 - Fehler treten folglich bei Debug nicht auf
 - Fehler in Release Version nicht nachvollziehbar :-)
- Initialisierung auf einen Wert ungleich 0 (-1) fördert Fehler auch in Debug Version zutage

Speicherschutz ohne MMU

Derzeit nicht optimal unterstützt :-)

ABER einige trickreiche Makros:

```
int aiTest[20], tmp;  
CA_DECLARE(aiTest, 20);  
for ( int j = 0; j < 100; ++j )  
    tmp = CA_REF( aiTest, j );
```

hält in Zeile `CA_REF(aiTest, j)` bei `j = 20`

Makros ohne MMU

```
#define CA_DECLARE( NAME, SIZE )  
gibt bekannt, dass Zeiger auf NAME die Länge SIZE hat
```

```
#define CA_REF( N, IDX ) \  
N [ assert( (N ## _tmp = IDX) < N ## _size ), N ## _tmp ]  
Referenz auf Index IDX des Zeigers N mit Prüfung
```

- Nachteil: Belastung der Rechenzeit
- Nachteil: bestehender Code umzuschreiben
- Vorteil: JEDER Zugriff auf Speicher wird exakt geprüft – nicht nur auf Kachelgrenzen

Kompilation?

- Für Nutzung der gesamten Funktionalität sind Änderungen im Quellcode sowie Neukompilation erforderlich
- Unter Linux können mit dem LD_PRELOAD Mechanismus beliebige dynamisch gelinkte Programme mit der Efence ausgeführt werden. Hierbei funktioniert dann allerdings nur das Konzept mit dem Kachelschutz.

Linux / Freie Software ?!

- Entwicklung der Efence Bibliothek:
Bruce Perens 1987 – 1999
- Weiterentwicklung durch freie Softwarelizenz
(GPL) ermöglicht als DUMA:
Hayati Aygün 2002 – 2006
- Bekanntgegeben durch <http://freshmeat.net>
- Einsatz / Verwendung:
Version 2.0 / 2.2 von Bruce Perens bei den
meisten Linux – Distributionen enthalten

Todo's

- C++ konformes Verhalten der überladenen new/delete/... Operatoren
 - new_handler, exceptions, ..
- Schutz der Rücksprungadresse einer Funktion / Methode:
 - Als C++ Klasse mit Destruktor auf Stack?
 - Konstruktoren schützt eine Kachel der Instanz
 - Destruktor hebt Schutz wieder auf
- ???

Bezug

- Beschreibung:
<http://duma.sourceforge.net/>
<http://sourceforge.net/projects/duma>

Ende

Fragen ?