Übungen zu Systemsicherheit

Jürgen Kleinöder,
Michael Gernoth, Reinhard Tartler
Universität Erlangen-Nürnberg, Informatik 4

SYSSEC

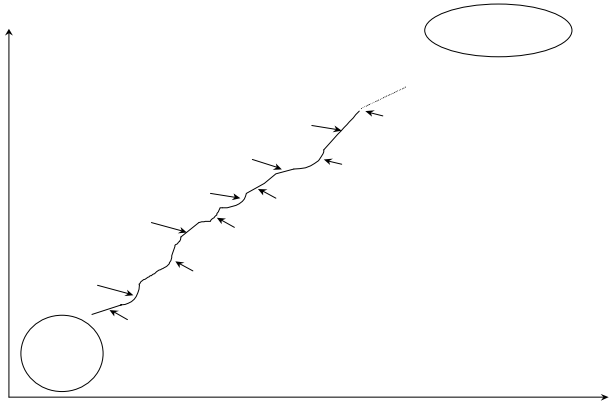
E.1 Besprechung der Aufgabe ’token’

- Probleme?

E.2 Statische Analyse von Programmen

- Softwaretestverfahren auf Basis des Quelltextes
- Formale Analyse ist falsizierendes Verfahren:
  Es wird *nur* Anwesenheit von Fehlern nachgewiesen!
- Weiterentwicklung von Style Checkern (MISRA-C, Style Guidelines, etc)
- Eingesetzte Techniken:
  - Model Checking
  - Datenfluss Analyse
  - Abstrakte Interpretation
- Markt an sowohl kommerziellen und freien Werkzeugen

E.3 Statische Analyse von Programmen (2)



E.4 Gnu Compiler Collection

- Guter Ausgangspunkt für statische Analyse hat der Übersetzer
  - Der komplette Quelltext liegt vor
  - Muss komplette semantische Analyse durchführen
  - kann zusätzliche Analysen "nebenbei" noch durchführen
- GCC bietet umfangreiche Optionen für Warnungen an
  - `-Wall` guter Ausgangspunkt
  - `-Wall` schaltet aber bei Weitem noch nicht alle Warnungen an!
  - Alle Optionen zum Thema Warnungung in Kapitel 3.8 des GCC Manuals
- Beispiel:

`-Wconversion`: Warn if a prototype causes a type conversion that is different from what would happen to the same argument in the absence of a prototype. This includes conversions of fixed point to floating and vice versa, and conversions changing the width or signedness of a fixed point argument except when the same as the default promotion.

E.5 Secure Programming Lint (splint)

- Forschungsprojekt der University of Virginia
- Schwerpunkt der Analysen liegt auf sicherheitstechische Schwachstellen und Programmierfehler
- Stellt viele weitere Analysen zur Verfügung:
  - Dereferenzierungen von Null-Zeigern
  - Nutzung von nicht definierten Variablen und Puffern
  - Typverletzungen
  - Fehler in der Speicherverwaltung
  - *Gefährliches* Aliasing
  - Umgang mit globalen Variablen
  - Problematische Kontrollflüsse wie Endlosschleifen, unvollständige `switch`-Anweisungen, etc.
  - Gefährliche Verwendung von Makros

E.6 Aufgabenstellung

- Verbesserung der Aufgabe `filecrypt.c`
  - Sicherung des alten Standes
  - Verbesserung durch GCC Flags
  - Verbesserung durch Lint Checking
- Vorstellung der Schwierigkeiten bei der Verwendung von `splint`
- Vergleich der alten und der verbesserten Lösung
- Abgabe am 17./19.12.2008!

E.7 Hinweise zur Benutzung von Splint

- Gründliche Lektüre des Splint Manuals essentiell!
  - `http://www.splint.org/manual/html/`
- Unübersichtliche Anzahl an Optionen und Flags
- Gruppierung in 4 Klassen:
  - **weak**
  - standard
  - *checks*
  - *strict*
- `splint` prüft an sich nur nach C89 (!), kennt aber einige (wenige) C99 Eigenheiten.
  - In Blöcken Deklarationen und Anweisungen nicht mischen
  - vgl. `http://home.datacomm.ch/t_wolf/tw/c/c9x_changes.html`

E.8 Definitionsanalyse

- Für Datenflussanalysen und einige Kontrollflussanalysen muss `splint` die Semantik von Parametern kennen
- Kennzeichnung durch Annotationen (Kapitel 3 und 7)

```
extern void setVal (/*@out@*/ int *x);
extern int  getVal (/*@in@*/ int *x);

int dumbfunc (/*@out@*/ int *x, int i){
  if (i > 3)
       return *x;
  else if (i > 1)
       return getVal (x);
  else {
       setVal (x);
       return *x;
  }
}
```


E.9 Strenge Typprüfung

- Integrale Typen (vgl. Kapitel 4.1)
  - splint führt eine strenge Typprüfung durch
  - Implizite Casts von z.B. `size_t` nach `int` sind dabei strenggenommen Fehler und werden angewarnt
  - Bei bestehendem Code kann diese Prüfung mit `+match-any-integral` gelockert werden
- Boolen-Typ (vgl. Kapitel 4.2)
  - C89 schreibt kein booleanschen Typ vor
  - C99 führt (endlich) den typ `bool` ein, schreibt aber keine Typprüfung vor
  - mit `-booltype <name>` kann der konkret verwendete booleansche Typ festgelegt werden

E.10 Speicherverwaltung (Kapitel 5 & 6)

- `splint` bentuzt ein Speichermodel, bei dem jedes Objekt ein Stück Speicher ist.
- Speicheranforderungen impliziert eine Verpflichtung, diesen auch wieder freizugeben. Dies geschieht entweder in der selben Funktion, oder in einer aufgerufenen.
- `malloc()` / `free()` sind wie folgt annotiert:

```
/*@only@*/ /*@null@*/ void *malloc (size_t size);
void free (/*@only@*/ /*@out@*/ /*@null@*/ void *ptr);
```

- Referenzen auf Speicher müssen nun richtig gekennzeichnet werden:
  - `temp`: temporäre Zeiger
  - `owned` / `dependent`: Aufweichung der Checkregeln. dependent Referenzen dürfen auf owned Referenzen zeigen, diesen aber nicht freigeben
  - `keep`: Wie only, aber speicher braucht nicht freigegeben zu werden

E.11 Funktionsschnittstellen

- z.B. Modifikationsanalyse (Kapitel 7.1):

```
void setx (int *x, int *y) /*@modifies *x@*/ {
      *y = *x;
}

void sety (int *x, int *y) /*@modifies *y@*/{
   setx (y, x);
}
```

- Zustands-Zusicherungen (Kapitel 7.4): Per Annotationen kann angegeben werden, dass globale Variablen und Parameter:
  - Verwendet
  - Allokiert
  - Definiert
  - Freigegeben

E.12 Bibliotheken

- Jedes nicht triviale Programm benutzt externe Bibliotheken
  - libc
  - openssl
- Splint ist ein C89 Scanner und prüft normalerweise nur ANSI C
- In der Übung werden aber POSIX Schnittstellen benutzt:
  - `splint +posixlib` aktiviert die Posixheader
- Verwendung von OpenSSL:

```
#ifndef S_SPLINT_S
#include <openssl/evp.h>
#else
int     EVP_read_pw_string(/*@out@*/ char *buf, int
length,const char *prompt,int verify);
#endif
```


E.13 Weitere Fallstricke mit `splint`

- Splint hat Probleme mit der `unistd.h` (`http://bugs.debian.org/473595`)

```
#include <unistd.h>

int main()
{ ...
}
```

- Lösung: `unistd.h` vor splint verstecken:

```
#ifndef S_SPLINT_S
#include <unistd.h>
#endif
int main()
{ ...
}
```