Jörg Schilling
Eigene Softwareprojekte automatisch Testen
Fokus Fraunhofer

Fraunhofer FOKUS

Früher...

- War es üblich OpenSource Software zu entwickeln
- Änderungen in OSS Code erstmal schnell zu vertreiben
- Dann darauf zu warten, daß Anwender Probleme melden
- ...so habe auch ich früher meine Software entwickelt
- Diese Methode kennt man auch unter dem Namen...

- **Der Bananensoftware Entwicklungszyklus:**
  - Software entwerfen
  - Software implementieren
  - Software an Kunden ausliefern
  - Beim Kunden reifen lassen
  - Fehlermeldungen entgegennehmen
  - Fehler beheben
  - Zurück zum Kunden mit der neuen Version...
- **Wollen wir das wirklich?**
  - Nein? Wir müssen Probleme vor der Auslieferung erkennen

Typische Probleme in Software

- Software macht manchmal etwas Anderes als erwartet
- Software macht manchmal seltsame Dinge
- Software stürzt manchmal einfach ab
- Software hängt sich auf

Typische Probleme in Software

- Software macht manchmal etwas Anderes als erwartet
  - Algorithmus schlecht geplant
- Software macht manchmal seltsame Dinge
  - Nicht alle Randbedingungen geprüft
- Software stürzt manchmal einfach ab
  - Meist Speicherverwaltungsfehler / Buffer Overflow
- Software hängt sich auf
  - Viele denkbare Ursachen

Muß das so ein?

- Fehlerbehebung behebt gemeldete Fehler aber...
  - Erzeugt oft auch neue Fehler
  - Erzeugt oft auch alte, bereits bekannte Fehler neu
  - Ändert oft auch das Verhalten auf unerwartete Weise
- Sollten wir daher nicht einfach eine „bessere“ Programmiersprache wählen?
  - Der meiste relevante Code ist in C/C++ geschrieben
  - In jeder Sprache kann man seltsame Dinge tun
  - Daher müssen wir bestehenden Code verbessern

Es muß etwas passieren

- … meine Anfänge des automatisierten Testens:
- Im Sommer 2008 Unit-Test-Suite für „SCCS“
  - Abgeleitet aus der CSSC Test-Suite
  - Inzwischen 2496 Einzeltests für „SCCS“
- Im Herbst 2014 hat Heiko Eißfeldt angefangen mit „American Fuzzy Lop“ und „schilytools“ zu experimentieren
  - In „smake“ und „sh“ damit weit über 100 Bugs gefixt
- Im Frühjahr 2016 Unit-Test-Suite für „Bourne Shell“
  - Inzwischen 821 Einzeltests für „Bourne Shell“

Testverfahren

- **Unit-Tests**
  - **Meist Handgeschrieben. Hunderte bis tausende Einzeltests**
- **Statische Codeanalyse**
  - **Oft mit Hilfe von Compiler Erweiterungen**
- **Laufzeitumgebungen**
  - **Speicherchecks**
- **Abdeckungs-Tests (Fuzzing)**
  - **Instrumentiertes Binary, zufallsgesteuerter Input sucht neue Pfade i. Binary**
- **Testen nach Spezifikation**
  - **Zufallssteuerung + Grammatik → Input Vektor, der Grammatik genügt**
  - **Benötigt korrekte Grammatik**

Weitere Testverfahren

- „Testen“ durch Codereview
  - Ein paar Tage warten und eigenen Code ansehen
  - Kollegen oder Freunde Änderungen ansehen lassen
  - Bei OpenSolaris gibt es dafür „Webrev“
    - Mit HTML und PDF aufbereitete diffs im Web
    - Erfolgreicher Codereview Vorraussetzung f. Putback
  - Alternatives Werkzeug: „Gerrit“ → URL im Anhang
  - Vorher Unit-Tests und Regressionstests
    - Weil Tests meist schneller als Codereviews sind

Testverfahren Generell

- „Guttests“ prüfen ob d. Programm tut was man erwartet
  - z.B. Unittests, Testen nach Spezifikation
- „Schlechttests“ prüfen ob d. Programm evt. Abstürzt
  - z.B. „Fuzzing“
  - Abstürze können häufig für Angriffe genutzt werden
  - Probleme d. Schreibzugriffe sind gefährlicher als d. Lesezugriffe (Ausnahme: Heartbleed)
  - Fixen erkannter Abstürze hilft auch gegen Angriffe
  - Einfacher als zu Prüfen ob ein Angriff möglich ist
  - Alle Bugs sollten gefixt werden

Unit-Tests

- Systematische Tests dokumentierter Funktionen
  - Sehr viele manuell geschriebene Testfälle
  - Regressionstests
  - gegen versehentliches Wiedereinbauen von Bugs
- Dazu gibt es Rahmenwerke zur Ablaufsteuerung, z.B.:
  - „testutils“ (Shell-basiert in den „Schilytools“, Weiterentwicklung der CSSC Test Suite)
  - FreeBSD „TestSuite“
  - TET - Test Environment Toolkit (OpenGroup / POSIX)

- Eingabedaten und Aufruf festlegen
- Ergebnisse festlegen
  - Stdout Daten
  - Stderr Daten
  - Exit Code
- Ablaufsteuerung startet Programm mit Eingabedaten
- Danach werden die Ergebnisse verglichen
- Bei Abweichungen → Loggen des Fehlers

- Probleme, die ein Kompiler durch Analyse des Codes findet
  - NULL Pointer Dereferenzierungen
  - Statische Array-Bound-Checks
- Typische Software:
  - Lint (z.B. in „SolarisStudio“, Oracle - frei für Linux)
  - Coverity (Kommerziell, aber frei für OSS, sehr gut)
  - Cppcheck (Lint-ähnlich, OSS, gut)
    - Vorschläge zur Reduzierung des „Scopes“
    - Test auch von mit #ifdef deaktivierten Teilen
  - Splint (Lint-ähnlich, OSS, nicht so gut)

Laufzeitumgebungen

- **Valgrind**
  - Separates OSS Programm z. Erkennen v. Speicherfehlern
  - Langsam
- **AddressSanitizer**
  - Bestandteil von CLANG und GCC
  - Langsam
- **Silicon Secured Memory (Oracle Patent)**
  - Hardware Erweiterung in „Sparc M7“ und neueren CPUs
  - Schnell, kann daher im „Feld“ aktiviert bleiben
  - Auf Basis von MMU und ungenutzten Adress Bits (64 Bits)
  - Obere Bits werden zum „Einfärben“ von Speicher verwendet
  - Benachbarter Speicher mit unterschiedlichen „Farben“, MMU vergleicht alle Bits

Laufzeitumgebungen

- Diverse Bibliotheken zur Speicheranalyse
- Werden häufig mit LD_PRELAOD gebunden
  - libumem (OpenSolaris) genereller Speicherdebugger
  - Lib 0@0 (OpenSolaris) mapt Nullen auf Adresse 0
    - Linux bräuchte das Gegenteil
  - Eigene Wrapper zum Provozieren v. „*No Memory*“
  - Libdbgmalloc (Schilytools) braucht Rekompilation
  - ...

- z.B. Speicherchecks mit Hilfe von „AddressSanitizer“
  - Eingebaut in neuere CLANG und GCC
  - Erzeugt „instrumentierten“ Code
  - Redzone um Stackframes von Funktionen
  - Redzone um allozierten Speicher
  - Hat leider viele eigene Bugs
  - Kann nicht gut mit „varargs“ umgehen
  - Mit entsprechender Vorsicht gut verwendbar

Abdeckungs-Tests

- z.B. mit Hilfe von American Fuzzy Lop
  - „instrumentierter“ Code meldet AFL „besuchte Orte“
  - Zufallsgenerator erzeugt auch defekte Eingabedaten
  - Ablaufsteuerung erkennt Abstürze und Hänger
  - Testen von Parsern ohne Eigenleistung möglich
  - Kein Parser: Hüllen-Testprogramm wird benötigt
- Symbolische Ausführung
  - z.B. mit „Klee“ (llvm) Programm Schritt für Schritt
  - Dabei Auswirkungen vom Wertebereich bewerten

- z.B. mit „Abnfgen“
  - Backus-Naur Form d. Syntax als Input-Beschreibung
  - Zufallsgenerator + Backus-Naur Grammatik → Input
  - So erzeugter Input erfüllt die Regeln der Grammatik
  - Wäre eine gute Idee um den UNIX Shell zu testen
  - → POSIX Standard hat definitv falsche „sh“ Syntax
  - Kein bekannter Shell akzeptiert so erzeugte Skripte

1) Beispiele für systematisches Testen (Unit-Tests)

2) Beispiele für Speichertests (AddressSanitizer)

3) Beispiele für Abdeckungs Tests (AFL)

- **z.B. mit den Schily „testutils“:**

```
docommand command01 "$SHELL -c 'command'" 0 "" ""
docommand command02 "$SHELL -c 'command -v ls | grep /bin/ls'" 0 NONEMPTY ""
docommand command03 "$SHELL -c 'command -V ls | grep \"ls.*/bin/ls\"'" 0 NONEMPTY ""
docommand command04 "$SHELL -c 'command -p ls | grep command.sh'" 0 NONEMPTY ""
docommand command05 "$SHELL -c 'command env | grep ZzZzZ'" 1 "" ""
docommand command06 "$SHELL -c 'ZzZzZ=bla command env | grep ZzZzZ=bla'" 0 NONEMPTY ""
```

- **Gemeinsamer Code in $(SRCROOT)/tests/**
- **Beispiel-Anwendung z.B. in $(SRCROOT)/sh/tests/**
- **In „SCCS“ und „patch“ zusätzlich Zufallstests f. „diff“**

AdressSanitizer

- z.B. ASAN: Unit-Tests der „schilytools“ ablaufen lassen
- Warnung: ASAN funktioniert nicht mit „configure“
  - Lösung: Autokonfiguration manuell ohne ASAN
- Warnung: ASAN ignoriert per default SIGSEGV Handler
- „Schilytools“: README.compile enthält Anleitung
- Warnung: ältere ASAN Versionen scheitern mit dem Bourne Shell und melden nicht vorhandene Bugs
- Findet Bug in „patch“ in „schilytools-2018-01-26“
  - Vorführung und Fix des Bugs im Terminal...

- Interne Tabellen mit 64 Bit Binaries werden sehr groß
  - Lösung: 32 Bit Binaries bauen
- Für Bibliotheken muß ein Testkommando gebaut werden
  - z.B. zwei SSL Implementierungen vergleichen
  - Input Daten durch AFL erzeugen lassen
- Parser können direkt getestet werden
  - „Start-Skript“ von Hand erstellen
  - Varianten werden durch AFL aus Start-Skript erzeugt
  - Achtung: Tests mit Shells erzeugen viele Dateien

- Mittel gegen Erzeugen und Löschen von Dateien
  - Aufruf von AFL in chroot Umgebung
  - Alles Read-Only machen außer /tmp
  - Separate leere Test-Directory erzeugen
  - Working Directory in leere Directory verlegen
  - Hier entstehen tausende von Müll-Dateien
- Vorführung im Terminal...
  - Echter Test dauert hier zu lange

- Diesen Vortrag findet man hier:
  - http://cdrecord.org/Files/
- Nützliche URLs
  - http://lcamtuf.coredump.cx/afl/ American Fuzzy Lop
  - https://github.com/danmar/cppcheck Cppcheck
  - https://github.com/codelion/gramtest Grammatik
  - http://www.quut.com/abnfgen/ Grammatik
  - http://klee.github.io/ Symbolische Ausführung
  - http://llvm.org/pubs/2008-12-OSDI-KLEE.html Mehr Klee
  - https://www.gerritcodereview.com/ Gerrit

URLs

- https://blogs.oracle.com/franzhaberhauer/sql-und-sicherheit-in-silizium Hardware-checks
- https://go-review.googlesource.com/c/go/+/57130 Gerrit Beispiel
- https://sourceforge.net/projects/schillix-on/files/webrev/ Webrev Beispiele
- http://schillix.sourceforge.net/webrev/webrev-746 Webrev 746 zur direkten Betrachtung

Danke!