Jörg Schilling

Angora und andere neue Kaninchen im Test-Zoo

Fokus Fraunhofer

Kaninchen?

- Das kommt vom bekanntesten Programm dieser Gruppe
  - American Fuzzy Lop ist ein Fuzzer
  - American Fuzzy Lop ist ein Kaninchen

Dieser Vortrag ist über Angora

- **Das sieht dann so aus:**

Warum überhaupt Testen?

- Programme sind nie fehlerfrei
- Vermeidbare Fehler sollten nicht zum Kunden kommen
- Fehler in Programmen werden oft für Angriffe genutzt

- **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, z.B. Endlosschleifen

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 885 Einzeltests für „Bourne Shell“

Testverfahren

- **Unit-Tests**
  - **Meist Handgeschrieben. Hunderte bis tausende Einzeltests**
- **Statische Codeanalyse**
  - **Oft mit Hilfe von Compiler Erweiterungen**
- **Laufzeitumgebungen**
  - **Speicherchecks, Speicherzugriffsfehler**
- **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 unterstützend „Webrev“
    - Mit HTML und PDF aufbereitete diffs im Web
    - Erfolgreicher Codereview Voraussetzung 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. durch „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

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

Wirkungsweise von Symbolischer Ausführung

- **Eine Art Interpreter analysiert das Programm in abstrakter Weise**
  - **Jeder Schritt im Programm und seine Auswirkungen**
  - **Ermöglicht Analyse komplexer Ausdrücke**
    - **Wirkung und Bereich von einzelnen Variablen**
    - **Erreichbarkeit von Bedingungen**
  - **Damit lassen sich theoretisch alle Probleme erkennen**
- **Nachteil: sehr langsam, hoher Speicherbedarf**

Wirkungsweise von Fuzzern

- Ziel ist es alle Code-Teile (Blöcke) zu erreichen
- Ziel ist es passende Eingabedaten zu generieren
  - Typischerweise mit Zufallsgenerator
- Dann wird geprüft, ob offensichtliche Probleme auftreten
  - Endlosschleifen
  - Fehlerhafte Pointernutzung (Abstürze)
  - Unerwartete Signale
  - Over- oder Under-flows (nur mit zusätzlichen Mitteln)
- Eingangsdaten die Fehler auslösen werden archiviert
- Erreichbare Abdeckung hängt von Suchalgorithmus ab

Wirkungsweise von AFL

- Instrumentierter Code erkennt „Besuch“ von Blöcken
- Input wird ausgehend vom Startwert modifiziert
- Zufallsgeneratoren bestimmen wieviel wo geändert wird
  - z.B. größere Anderungen um neue Blöcke zu erreichen
- Rückkopplung erkennt neu betretene Blöcke
  - Der Zufallsgenerator sucht dann „in der Nähe“ einer solchen Änderung weiter
- Input, der Probleme verursacht, wird archviert
- Nachteil: Erreicht nur eine „mäßige“ Abdeckung
  - Auch wenn Erstaunliches geleistet wird

Seit Mai 2018 gibt es Angora

- **Angora wurde auf dem IEEE Symposium 2018 in San Franzisko vorgestellt**
- **Leider wurde die Software erstmal nicht herausgegeben**
- **Die Nachfrage vieler hat im Dezember 2018 zu einer Ver-öffentlichung geführt**

Wirkungsweise von Angora

- Angora verwendet zwei Binary-Varianten des SuT:
  - Stark instrumentierte Version versteht Bedingungen
    - Das Verfahren wird „Taint Tracking“ genannt
  - Schwach instrumentierte Version f. Blockerkennung
- Mit der „starken Version“ werden Bedingungen analysiert, die nicht durchlaufene Blöcke verursachen.
  - Diese nicht erreichten Blöcke werden dann gemerkt
  - Mit dieser Liste startet dann die Hauptschleife von Angora

Wirkungsweise von Angora

- Schleife, solange noch unentdeckte Blöcke existieren
- Einen unentdeckten Block selektieren, Schleife bis OK
  - Input an Hand v. Wissen über Ausdruck modifizieren
  - Testen des neuen Inputs mit schwach inst. Version
  - Neu gefundene Blöcke mit in stark inst. Version testen und in Liste aufnehmen
  - Wenn neuer Block erreicht, dann aus Liste löschen
  - Wenn Fehler gefunden, Input Archivieren
- Solange Liste unentdeckter Blöcke nicht leer

Wirkungsweise von Angora

- **Da Angora versteht, wie ein komplexer logischer Ausdruck durch welchen Input beeinflußt werden kann**
  - **Ist der Aufwand neue Blöcke zu betreten geringer als bei AFL**
  - **Reicht die schwach instrumentierte Version b. Test m. Zufall**
- **Vorteile:**
  - **Nur relevante Teile des Inputs müssen geändert werden**
- **Nachteile:**
  - **Kompletter Quellcode muß Angora voliegen**
  - **Oder eine Beschreibung was Library Funktionen tun**

Angora selbst nutzen

- **Dazu muß ein Docker Image aus dem GIT geholt werden**
- **Darin wird automatisch die richtige Kombination v. Kompilern und anderen Tools installiert**

Probleme von Angora

- Aktuell fehlen Beschreibungen für viele libc Funktionen
  - z.B. die Beschreibung von mbtowc()
- Daher lasen sich Shells zur Zeit nicht damit testen
  - Wir werden demnächst eine Beschreibung für die Funktion mbtowc() bauen und mit LC_ALL=C testen

QSYM

- Ein Hybrid aus Fuzzing und symbolischer Ausführung
- Symbolische Ausführung zum Bedingungen analysieren
  - Normales Fuzzing mit schwacher Modifikation des neuen Inputs
- Funktionsweise daher ähnlich zur Angora Grundidee
- Nachteile:
  - Auch hier muß der Quellcode vorliegen

Redqueen

- Fuzzing mit Input-to-state Austausch
  - Binäre Ankopplung an das SuT
  - Erkennen von Zusammenhängen zwischen Input und CPU state
  - Basiert auf Intel Processor Tracing (Branch Tracing)
- Funktionsweise ähnlich zur Angora Grundidee
- Vorteile:
  - Quellcode wird nicht benötigt
- Nachteile:
  - Geht nur mit Intel CPUs

Ich will testen, habe aber keine Bugs

- Wenn man Fuzzer vergleichen will benötigt man Bugs
  - Die möglichst standardisiert sind
- Viele Entwickler von Fuzzern wollen sich vergleichen
- LAVA
  - Large-scale Vulnerability Addition
  - Ein Programm zur Erzeugung von Bugs
- Typischerweise GNU-Bin-Tools als Basis f. Bugs
  - Gute Fuzzer finden mehr als 100%
  - ...Also mehr Bugs als durch LAVA eingebaut wurden

URLs

- 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

Nützliche URLs für neue Kaninchen

- https://github.com/AngoraFuzzer/Angora
- https://github.com/AngoraFuzzer/Angora/blob/master/docs/running.md
- https://arxiv.org/pdf/1803.01307.pdf **Angora Papier**

Redqueen:

- https://www.syssec.ruhr-uni-bochum.de/media/emma/veroeffentlichungen/2018/12/17/NDSS19-Redqueen.pdf
- https://github.com/RUB-SysSec/redqueen

QSYM:

- https://blog.acolyer.org/2018/09/12/qsym-a-practical-concolic-execution-engine-tailored-for-hybrid-fuzzing/
- https://www.usenix.org/conference/usenixsecurity18/presentation/yun
- https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-yun.pdf
- https://www.usenix.org/sites/default/files/conference/protected-files/usesec18_slides_yun.pdf
- https://youtu.be/cXr1ZXp40jA
- https://github.com/sslab-gatech/qsym

URLs

- **Large-Scale Automated Vulnerability Addition (LAVA):**
- **https://seclab.ccs.neu.edu/static/publications/sp2016lava.pdf**
- **http://panda.moyix.net/~moyix/LAVA_Sapienza.pdf**

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!

Jetzt noch etwas philosophisches

In zweifelhaften Fällen entscheide man sich für das Richtige

Karl Kraus

Neue Wege entstehen in dem wir sie gehen

Friedrich Mietzsche