TECHNISCHE UNIVERSITÄT DRESDEN

DRESDEN concept

Technische Universität Dresden, Fakultät Informatik, Professur für Betriebssysteme

Fuzzing
Ein kurzer Überblick

Dresden, 11. Januar 2019

Inhalt

Grundidee

Vor- und Nachteile

Techniken

Tools:
American fuzzy lob, libFuzzer, ...

Grundidee

Grundidee

Grundidee

1. Füttern von Programm mit zufällige Daten
2. Protokollieren von Programmabstürzen und deren Ursachen

Erste wissenschaftliche Veröffentlichung

Barton Miller et. al. (1989): „An Empirical Study of the Reliability of UNIX Utilities“

Vor- und Nachteile

Vor- und Nachteile

- Vorteile:
  - Implementierung einfach
  - Fehlerdetektion auch in Randfällen sowie außerhalb der Spezifikation
  - Wartungsaufwand zum Finden von Abstürzen gering
- Nachteile:
  - Keine Garantie für vollständige Fehleridentifizierung
  - Robustheitsprüfung, aber keine Verifikation der Ergebnisse
  - Notwendigkeit einer manuelle Absturzanalyse
  - Rechenaufwand hoch

Techniken

Techniken
Übersicht

- Was wird gefuzzt?
  - Programme mittels Dateien, `stdin`, Netzwerk, ...
  - Betriebssysteme mittels Syscalls
  - Hardware
- Wie werden die Daten generiert?
  - Zufälliges Fuzzing
  - Mutationsbasiertes Fuzzing
  - Regelbasiertes Fuzzing
  - Instrumentiertes Fuzzing

Techniken I

Zufälliges Fuzzing
(random fuzzing)

- Generierung zufälliger Daten und Verarbeitung durch Programm
  `cat \dev\urandom | programm_to_fuzz`
- Vorteil:
  - Sehr einfach
- Nachteile:
  - Nur Entdeckung einfacher Fehler

Techniken II

Mutationsbasiertes Fuzzing
(mutation based fuzzing)

- Erstellung einer Sammlung von Testdaten (Testkorpus)
- Mutation der Daten im Testkorpus
- Vorteil:
  - Implementierung simpel
  - Findet viele Fehler
- Nachteile:
  - Große Abhängigkeit von Testdatenauswahl
- Tools: `zzuf`, `Radamsa`

Techniken III

Regelbasiertes Fuzzing
(generation based fuzzing)

- Erstellung einer Datenbeschreibung aus Spezifikation
- Generierung zufälliger Sequenzen aus Beschreibung
- Vorteil:
  - Entdeckung komplexer Logik- und Protokollfehler
- Nachteile:
  - Hoher Aufwand der Beschreibungserstellung
  - Existenz einer Spezifikation nötig
  - Starke Abhängigkeit von Beschreibungsqualität
- Tools: `Peach Fuzzer`, `Dharma`

Techniken IV

Instrumentiertes Fuzzing
(coverage guided fuzzing)

- Beobachtung der Codeausführung
- Mutation der Daten anhand des Feedbacks
- Vorteile:
  - Hohe Effizienz
  - Hohe Code-Abdeckung
  - Ermöglicht Testkorpus- sowie Testfallminimierung

  Nachteil:
  - Leichte Verschlechterung der Performance
- Tools: `AFL`, `libFuzzer`, `honggfuzz`

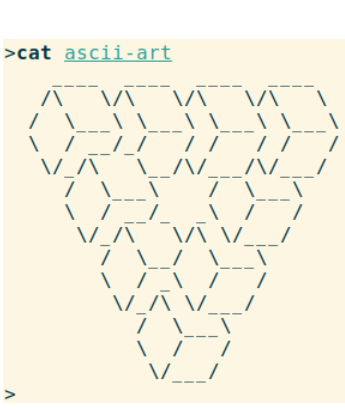
Techniken V
Datenmutation

- Vollständiges Durchsuchen nicht möglich:
  2^{32} Möglichkeiten $\rightarrow$ 50 Tage fuzzen
- Einschränkung des Suchraums durch:
  - Bit-Flips und -Shifts
  - Zahlenersetzung (0, 1, -1, `MAXINT - 1`, `MAXINT`, `NaN`, `Inf`)
  - Addition oder Subtraktion bestehender Zahlen
  - Einfügen interessante Zeichenketten:
    - Sehr lange oder komplett leere Zeichenketten (`''`, `128*'a'`)
    - Spezielle Zeichen (`\0`, `\n`)
    - Pfade (`../../../../../../../etc/shadow`)
    - Format Strings (`%s%s%s%s`, `%x %x %x`)

Techniken V

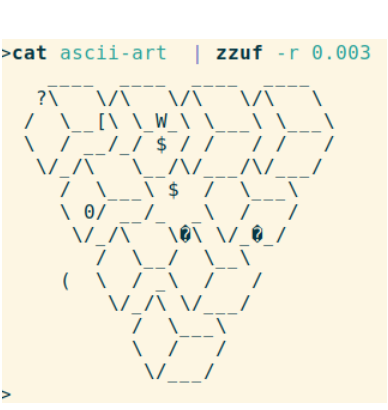
Datenmutation

```
>cat ascii-art
```



```
>
```

```
>cat ascii-art  | zzuf -r 0.003
```



```
>
```

Zufällige Bitflips mit `zzuf`

Techniken V

Datenmutation

```
>echo "Fuzzing 123 abc" | radamsa -n 10
Fuzzing -1423745067377465576524700419517369834 71 abc
Fuzzing --127 abc
Fuzzing 127 abc
Fuzzing 4294Fuzzing 4293Fuzzing 427332879059253628972021 9Fuzzing 1967295 abc
Fuzzing -2001967295 abc
Fuzzing 0 abc
Fuzzing 4294Fuzzing 4294g 4294967295 abc
Fuzzing 340282366920938463463374607431768211456 abc
Fuzzing 42941 abcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabc -1
abca+/v/bcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcabcab
cabcabcab
```

Techniken VI

Sanitizer

- Problem: Kein sofortiger Programmabsturz durch bestimmte Fehler
- Lösung: Verwendung eines Sanitizers
  - Ergänzung des Codes um zusätzliche Überprüfung von Argumenten
- Nachteil: Erhöhung des Rechen- und Speicherbedarfs
- Entwickelte Sanitizer:
  - AddressSanitizer (ASan)
  - UndefinedBehaviorSanitizer (UBSan)
  - MemorySanitizer
  - LeakSanitizer
  - ThreadSanitizer

Techniken VI

Sanitizer

```
>cat buffer_overflow.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
void func(char **argv) {
  printf("running strcpy...\n");
  char arr[16];
  strcpy(arr, argv[1]);
}
int main(int argc, char *argv[]) {
  if(argc == 2) {
    func(argv);
  }
  return(0);
}
```

Demonstrationsprogramm

Techniken VI

Sanitizer

```
>gcc -o buffer_overflow buffer_overflow.c
>./buffer_overflow aaaaaaaaaaaaaaa # 15*a
running strcpy...
>./buffer_overflow aaaaaaaaaaaaaaaa # 16*a
running strcpy...
>./buffer_overflow aaaaaaaaaaaaaaaaaaaaaaaaa # 25*a
running strcpy...
*** stack smashing detected ***: <unknown> terminated
fish: './buffer_overflow aaaaaaaaaaaa…' durch Signal SIGABRT (Abbruch) beendet
```

```
>gcc -o buffer_overflow -fsanitize=address buffer_overflow.c
>./buffer_overflow aaaaaaaaaaaaaaa # 15*a
running strcpy...
>./buffer_overflow aaaaaaaaaaaaaaaa # 16*a
running strcpy...
=================================================================
==2464==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffc0b12e1c0
 at pc 0x7f01113bb741 bp 0x7ffc0b12e170 sp 0x7ffc0b12d918
```

Kompilierung und Ausführung mit und ohne Sanitizer

Techniken VII

Tipps

- Abwägung Fuzzing-Geschwindigkeit gegenüber Fuzzing-Effizienz
  Anzahl gefunder Fehler = Programmausführungen * Sucheffizienz
- Verwendung von Sanitizern
- Teilweise Codeanpassungen nötig
  (Deaktivierung von Checksummen)

Tools:
American fuzzy lob, libFuzzer, ...

Tools: AFL und libFuzzer

- American fuzzy lop-Entwicklung durch Michal Zalewski
- libFuzzer-Entwicklung durch LLVM-Community
- Instrumentieren Code, erreichen hohe Code-Abdeckung
- Entdeckung sehr vieler Fehler in fast allen Programmen
- Entdeckung von Heartbleed einfach möglich

Tools
AFL und libFuzzer

`AFL`

- Sehr schnelles Aufsetzen (~5 Minuten)
- Übergabe der Daten per `stdin` oder Datei
- Gray- und White-Box-Fuzzing
- Standard: Start vieler Prozesse
- Möglichkeit des In-Prozess-Fuzzings

`libFuzzer`

- Schnelles Aufsetzen (~$\frac{1}{2}$ Stunde)
- Übergabe der Daten durch Helferprogramm
- White-Box-Fuzzing
- Kompilierung mit `clang` notwendig
- schnell durch In-Prozess-Fuzzing

Tools
AFL und libFuzzer

Quickstart AFL:

- Kompilierung: `CC=afl-gcc ./configure -disable-shared`
- Ausführung: `afl-fuzz [...] ./programm_to_fuzz`

Quickstart libFuzzer:

- Implementierung des Helferprogramms:

```
// fuzz_target.cc
extern "C" int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;  // Non-zero return values are reserved for future use.
}
```

- Kompilierung: `clang -fsanitize=fuzzer fuzz_target.c`
- Ausführung: `./a.out [...]`

Tools
Kernel-Fuzzer

- `syzkaller`
  - Entwicklung durch Google
  - Instrumentiertes Fuzzing
  - Start des Systems in VM und Fuzzing der Syscalls
- Alternative: `trinity`

Tools
Weitere Tools

`Sandsifter`

- Fuzzing von CPU-Instruktionen
- Entdeckung von Bugs in Disassemblern, Emulatoren, Hypervisorn sowie x86-Chips

`ClusterFuzz` und OSS-Fuzz-Projekt

- Google-Entwicklung zum Fuzzing von Chrome
- Verteiltes Fuzzing auf hunderten Kernen
- OSS-Fuzz: Fuzzing verbreiteter Open-Source-Software

Fragen?