Erstellung eines Frameworks zur statischen Analyse von Quelltexten

Jonas A. Wendorf

Hochschule Aalen

15. April 2019

Inhalt

Inhalt

Einleitung

- Sicherheitsprüfungen können statisch oder dynamisch sein
- Dynamisch: Eingaben werden am laufenden Programm vorgenommen
- Statisch: Mögliche Eingaben werden anhand des Quelltextes nachvollzogen
- Entwicklung eines Frameworks zur statischen Analyse von Quelltexten

Einleitung: Motivation

- Dynamische Sicherheitsprüfungen sind kostenintensiv und zeitlich limitiert
- Statische Sicherheitsprüfungen können kostengünstig in die Entwicklungspipeline aufgenommen werden
- Existierende statische Codescanner sind
  - teuer
  - schwer zu erweitern
  - teilweise inkompatibel mit neuen Versionen einer Programmiersprache
  - nicht für alle Programmiersprachen verfügbar

Inhalt

Grundlagen: Limitationen

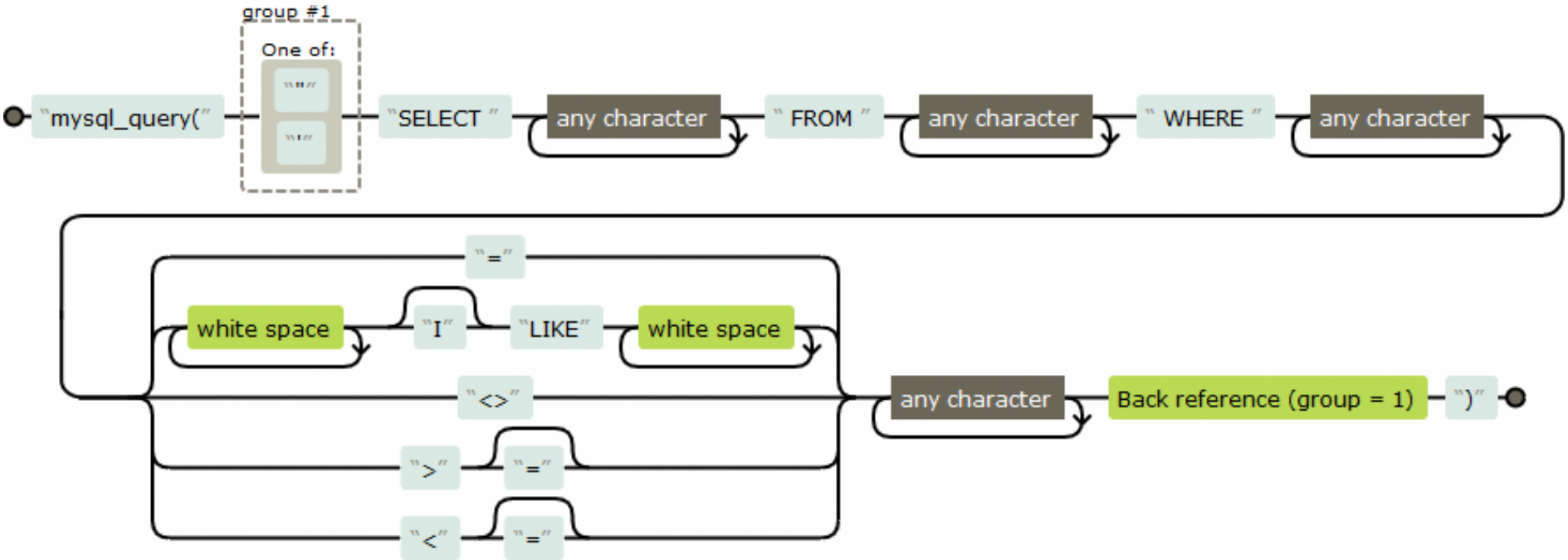
- »Alle interessanten Fragen über das Verhalten eines Programms sind unentscheidbar.« (M. Schwartzbach)
- Saloppe Zusammenfassung des Satzes von Rice
- Annäherung an das korrekte Ergebnis ist weiterhin möglich

Grundlagen: Ansätze

Einfache Stringsuche/reguläre Ausdrücke

- Wird sehr schnell sehr kompliziert:

```
mysql_query\((["'])SELECT .+ FROM .+ WHERE .+(?:\=|\s+I?LIKE\
s+|<>|>=?|<=?).+\1\)
```



Grundlagen: Ansätze

Erstellung eines Parsers

- Einfachere Abfragen als Stringsuche
- Keine Simulation des Laufzeitverhaltens

→ Kann unbekannte Bestandteile überspringen

→ Einfachere Erstellung eigener Sprachdefinitionen

- Für das Framework genutzter Ansatz

Grundlagen: Begriffe

- Quelle (Source): Erzeugt benutzerdefinierte Eingaben
- Senke (Sink): Potentiell verwundbare Funktion
- Absicherung (Sanitizer): Sichert Eingaben für Senken ab
- Verschmutzung (Taint): Senke, die eine benutzerdefinierte Eingabe erhält

Inhalt

Implementierung: Vorgehensweise des Frameworks

1. Quelltexte einlesen
2. Passendes Modul auswählen
3. Quelltexte parsen
4. Regelsätze parsen
5. Nach Schwachstellen suchen
6. Codekomplexität analysieren
7. Report erstellen

Implementierung: Vorgehensweise bei der Suche nach Schwachstellen

1. Parser ermittelt Methoden innerhalb der Datei
2. Findet alle Variablen, die innerhalb der Methode verwendet werden
3. Sucht nach möglichen Quellen, Senken oder Absicherungen
4. Vergleicht die Eingabeparameter mit der Variablenliste

→ Schließt hardcodierte Methodenaufrufe aus

5. Verfolgt Variablen zurück
6. Informiert das Framework über neu hinzugekommene verwundbare Methoden
7. Wiederholung bis keine neuen Ergebnisse geliefert werden

Implementierung: Variablenrückverfolgung

- Verfolgung der Variable »test« im Aufruf von `printf`

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    char * foo = argv[1];
    char * bar = foo;
    char * test = bar;
    printf(test);
}
```

Implementierung: Variablenrückverfolgung

- Verfolgung der Variable »test« im Aufruf von `printf`

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    char * foo = argv[1];
    char * bar = foo;
    char * test = bar;
    printf(test);
}
```

Implementierung: Variablenrückverfolgung

- Verfolgung der Variable »test« im Aufruf von `printf`

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    char * foo = argv[1];
    char * bar = foo;
    char * test = bar;
    printf(test);
}
```

Implementierung: Variablenrückverfolgung

- Verfolgung der Variable »test« im Aufruf von `printf`

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    char * foo = argv[1];
    char * bar = foo;
    char * test = bar;
    printf(test);
}
```

Implementierung: Variablenrückverfolgung

- Verfolgung der Variable »test« im Aufruf von `printf`

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    char * foo = argv[1];
    char * bar = foo;
    char * test = bar;
    printf(test);
}
```

Implementierung: Variablenrückverfolgung

- Verfolgung der Variable »test« im Aufruf von `printf`

```
#include <stdio.h>
int main(int argc, char * argv[]) {
    char * foo = argv[1];
    char * bar = foo;
    char * test = bar;
    printf(test);
}
```

Implementierung: Exklusive Kontrollstrukturen

- Kontrollstrukturen, die wechselseitig exklusive Pfade erzeugen
- Beispiel: if/else

```
#include <stdio.h>
int main(int argc, char* argv[]) {
    if (0) {
        test();
    } else if (2) {
        printf(argv[1]);
    } else if (3) {
        printf("hallo\n");
    } else {
        printf("bye!\n");
    }
    return 0;
}
```

Inhalt

Regelsätze: Format

- Aufgebaut in YAML
- Unterschiedlich für Quellen und Senken

→ Senken haben optionalen Absicherungsblock

```
---
OBJEKTNAME:
    Methods:
    - Methodname: METHODENNAME
      Parameters: [null, $TAINT, null]
      Comment: KOMMENTAR
      Sanitizers:
      - OBJEKTNAME
          Methods:
          - Methodname: METHODNAME
            Parameters: []
            Comment: KOMMENTAR
```

Regelsätze: Beispielregel scanf

- `scanf` in C liest formatierten Text ein
- Erster Parameter ist Formatstring, darauffolgende Parameter sind Zielvariablen

```
---
null:
    Methods:
    - Methodname: scanf
      Parameters: [null, $TAINT]
      Comment: Reads formatted input from stdin
```

Regelsätze: Beispielregel printf

- `printf` in C gibt formatierten Text aus
- Bei nur einem Parameter wird dieser als Formatstring erkannt
- Benutzer kann eigene Formatstrings einfügen, um Daten zu offenbaren und Speicher zu ändern

```
---
null:
    Methods:
    - Methodname: printf
      Parameters: [$TAINT]
      Comment: Format string vulnerability.
```

Inhalt

Beispielreport: Verwendete Software

- Terminaldateimanager cfiles (Stand vom 17. Januar 2019)
- In C geschrieben
- Typische Sicherheitslücken
- Schwachstellen mittlerweile korrigiert
- Insgesamt mit vier Senken-Regeln 24 Verschmutzungen und 31 Senken gefunden
- Demo-Video

Beispielreport: Erkannte Verschmutzungen

Analysis results for method "init" (lines 134 to 171).

The following taints were detected:

- In line 145 a call with potentially user controlled input is made to sprintf.
  The following comment is linked to this sink: No check for destination buffer size, use snprintf instead.
  No sanitizer detected.
  Severity level: 100%.

```
char editor[20];
void init()
{
    ...
    // Set the editor
    if( getenv("EDITOR") == NULL)
        sprintf(editor, "%s", "vim");
    else
        sprintf(editor, "%s", getenv("EDITOR"));
    ...
}
```

Beispielreport: Erkannte Senken

Analysis results for method "main" (lines 942 to 1445).

Method has a cyclomatic complexity of 92.

The following sinks were detected:

- In line 1061 a call without any detected user controlled input is made to getPreview. The following comment is linked to this sink: Calls getArchivePreview from None. Severity level: 50%.

```
int main(int argc, char* argv[]) {
    ... getPreview(next_dir,maxy,maxx/2+2); ...
}
void getPreview(char *filepath, int maxy, int maxx) {
    ... getArchivePreview(filepath, maxy, maxx); ...
}
getArchivePreview(char *filepath, int maxy, int maxx) {
    ... sprintf(temp_dir,"atool -lq \"%s\" > ~/.cache/
    cfiles/preview",filepath); ...
}
```

Inhalt

Fazit

- Statische Analyse ist kein Allheilmittel
- Trotzdem können viele Schwachstellen bereits mit wenigen Regeln gefunden werden
- Eigene Regeln und sogar Module erstellen ist (vergleichsweise) einfach

$\rightarrow$ Auch exotische Sprachen können getestet werden, kein Update des Herstellers notwendig

- Kann einfach in den Entwicklungsprozess eingebunden werden