Statische Analyse

Laszlo Hernadi - 0327023
Seminar aus Programmiersprachen – WS 2007/2008
Technische Universität Wien
Institut für Computersprachen – Programmiersprachen und Übersetzer

Kurzfassung

Der Artikel [1] befasst sich mit einer von den drei Autoren T. Jensen, D. Le Métayer und T.Thorn entwickelten Methode zur Überprüfung von softwarebasierter Sicherheit. Dazu führen sie den Programmcode auf einen Graphen zurück, den sie dann auf einen endlichen Graphen beschränken, um die Sicherheitseigenschaften des Programms zu prüfen.

Laut den Autoren des Artikels hat die softwarebasierte Sicherheit mit dem Problem, ob lokale Sicherheitsüberprüfungen im Code ausreichend sind, um globale Sicherheit zu gewährleisten, zu kämpfen. Um eine Lösung für dieses Problem zu erhalten, stellen die Autoren ein Modell zur Überprüfung der Sicherheit bereit.

1. Einleitung

Java verfügt über Kontrollstrukturen die es ermöglichen Programmcode Befugnisse zu erteilen und diese zur Laufzeit zu überprüfen. Es gibt (im Zusammenhang mit Java) viele Arbeiten, die sich mit einzelnen Teilaspekten der Sicherheit beschäftigen. Dieser Artikel konzentriert sich jedoch darauf, einen Weg zu schaffen, um die Sicherheit des Codes bezüglich der globalen Gegebenheiten zu prüfen. Dies soll in zwei Teilen geschehen:

- Definition eines formalen Rahmenwerkes für die Definition einer Klasse von programmiersprachenbasierten Sicherheitsmerkmalen: Diese Merkmale hängen nur von der Kontrollstruktur und dem Aufrufgraphen des Programms ab. Es soll gezeigt werden wie man dieses Rahmenwerk dafür verwenden kann, um die Sicherheit von Applikationen die mit dem Java JDK entwickelt wurden zu testen.

- Es soll eine neue Technik gezeigt werden, welche die Überprüfung dieser Sicherheitsklasse ermöglicht, indem ein unendlicher Graph in einen Endlichen überführt wird, sodass die Sicherheitsmerkmale geprüft werden können.

2. Das Programmmodell

Um ein Rahmenwerk zu schaffen, das programmiersprachenunabhängig funktioniert, wird ein abstraktes Modell vorgestellt, das als Basis für die folgenden Sicherheitsmerkmaldefinitionen dienen soll. Das Modell vernachlässigt alles, was nicht mit Programmaufrufen (Funktionen oder Methoden) oder Kontrollstrukturen (if, while, etc.) zu tun hat.

G = (NO, IS, IT, TG, CG)

Der Graph G, der aus der Applikation abgeleitet wird enthält diese Elemente. TG steht für transfer-edges, CG für call edges. IT ist der Einstiegspunkt des gesamten Programmes. IS kann für ein call,

return oder check(Ø) stehen, dabei wird ein Sicherheitsmerkmal überprüft. NO steht für Node, also einen Punkt im Programm.

Erwähnenswert ist, dass dieses minimalistische Modell keine if oder while Anweisungen enthält. Für den Programmausschnitt m1(); if...then m2() else m3() bedeutet es, dass es die Nodes n1,n2,n3 gibt. TG-Edges führen von n1 zu n2 und von n1 zu n3.
Das funktioniert allerdings nicht für Programmiersprachen mit dynamischen Funktionsaufrufen (Funktionalen).
Der Aufrufsgraph CG beschreibt für jeden Node n eine Überschätzung auf sicherer Seite der möglichen Aufrufe von Methoden (Funktionen).

Es gibt einen Kontrollstack, der aus den Nodes, die aufgerufen wurden besteht. Der Kontrollstack n1 : n2 : n3 bedeutet, dass n1 einen Aufruf bewirkte, der n2 enthält, der wiederum einen Aufruf einer n3 enthaltenden Methode enthält. Der Stack wird immer dann verringert, wenn ein return Statement auftritt.

2.1 Formalismus zur Definition von Sicherheitseigenschaften

Die Semantik eines Programms ist definiert durch einen Satz von Elementen, die Stackketten sind. Es gibt 3 verschiedene Arten von Prädikaten, die man beachten muss:

1) Prädikate, die einzelne Nodes beschreiben
2) Prädikate, die Node-Stacks beschreiben
3) Prädikate, die Stack-Ketten von Nodes beschreiben

Ein Node entspricht einem Programmpunkt, also stellen Prädikate die charakteristischen Basiseigenschaften von einem Programmabschnitt dar, den der Node bezeichnet, wie den Aufrufpunkt (site of origin) oder seine protection domain. Ein Stand ist als ein Stack mit endlicher Anzahl von Nodes. Es gibt für Stand-Prädikate folgende Operationen:

X (nächstes Element Operator)
U (solange Operator, bis zum nächsten Element, das den Kriterien entspricht)

Disjunktion, Implikation und der Wert False können hergeleitet werden durch Verwendung von Konjunktion, Negation und True.

2.2 Beispiele von Sicherheitseigenschaften in diesem Framework

2.2.1 The segregation of duty (Arbeitstrennung)

Diese Form der Sicherheit wird oft in Finanzapplikationen angewandt, in dem bestimmte Vorgänge nur abgeschlossen werden können, wenn zumindest zwei berechtigte Personen diese gemeinsam ausführen wollen. In unserem Framework wird eine solche Person durch eine Eigenschaft dargestellt, die nur von den Nodes, die den berechtigten Programmpunkten entsprechen, erfüllt werden. Außerdem kann man solche berechtigten Personen in Gruppen zusammenfassen, wie zum Beispiel Manager, Buchhalter, etc.
Das Prinzip der Arbeitstrennung besagt nun, dass eine kritische Anwendung eines Buchhalters nur

dann ausgeführt werden kann, wenn ein Manager dabei ist. Wie folgt dargestellt:

SD = (**not** Critical **U** Manager) **and** (**not** Critical **U** Buchhalter).

Diese Definition hat folgende Eigenschaften:

1) Kein Node, der, die Eigenschaft Critical erfüllt, taucht vor dem ersten Node mit der Eigenschaft Manager auf
2) Kein Node, der die Eigenschaft Critical erfüllt, taucht vor dem ersten Node mit der Eigenschaft Buchhalter auf

2.2.2 Ressourcenschutz (Resource protection)

Programmcode der protection domain A kann nur Code der protection domain C unter Verwendung von Code aus der protection domain B aufrufen. Ein Node der protection domain D kann dadurch identifiziert werden:

RP = G(**not** A **or** (**not** C **U** B))

Für einen Stack s, wenn A zufällig durch einen Node n in s erfüllt wird, muss (not C **U** B) an dieser Stelle halten, was bedeutet, dass kein Node aus protection domain C nach n im Stack s vor dem ersten Node von B auftauchen kann.

2.2.3 Sandbox Modell

Dieses Prinzip, das ursprünglich von Java stammt, besagt, dass eine dynamisch geladene Methode der Site S nur Methoden derselben Site oder lokale Methoden aufrufen darf. Mit der Eigenschaft Local können lokale Nodes und Site S für Nodes der Site S charakterisiert werden:

SB = **G**(Site S => **X** (Site S **or** Local))

Also gilt für alle Nodes im Aufrufsstack: wenn der Node von einer nicht-lokalen Site stammt, muss der nächste Node entweder ein Aufruf von Code derselben Site oder zu lokalem Code sein (die dann wiederum andere Sites aufrufen kann).

2.2.4 Stack Überprüfung (stack inspection)

Die Stack Überprüfung ist ein grundlegender Sicherheitsmechanismus von Java. Die Sicherheitsanforderung hier ist, dass jeder Code, der zum Aufruf eines gegebenen Methodenaufrufes die nötigen Rechte für diesen besitzen muss. Das verhindert Codeausführung, die nicht gestattet ist. Die einzige Ausnahme ist, wenn der Code als „priviledged“ markiert ist, solcher Code kann alle Aufrufe ausführen, unabhängig davon, wer es aufgerufen hat. Dies wird von der stack inspection gewährleistet, die den Stack von oben nach unten (erstes gepushtes Element zuerst) auf die notwendige Autorisierung hin überprüft. Angenommen Priv ist eine Eigenschaft die von privilegiertem Code erfüllt wird, dann gilt:

JDK(Ø) = **G**((**X**(**F** Priv)) **or** Ø)

Das bedeutet, dass für jede Ausführung Stack s und jeder Node n in s (also **X**(**F** Priv)) oder n die Eigenschaft Ø erfüllen muss.
Die Eigenschaft JDK(Ø) stellt das Faktum dar, dass die hier definierte Sprache in der Lage ist sowohl globale Sicherheitseigenschaften als auch lokale Eigenschaften zur Laufzeit zu überprüfen (die check(Ø) Anweisung in unserem Modell).

2.3 Endzustandsverifikation

Die Methode der Autoren ist eine generelle Verifikationstechnik. Sie ist auf andere Eigenschaften als Sicherheitseigenschaften anwendbar, weshalb sie nicht in einer sicherheitsspezifischen Weise formuliert ist. Eine mechanische Verifikationsmethode wird durch die Tatsache erschwert, dass das Programm möglicherweise ein Modell ohne Endzustand ist (infinite state transition system). Um eine Entscheidungsfindung zu ermöglichen, muss das (möglicherweise unendliche) System in ein endliches System überführt werden. Grob gesagt wird dafür eine Auswahl der States genommen, deren Wertung einen gewissen Grenzwert unterschreiten. Der springende Punkt dabei ist, dass der Grenzwert vom Programm und der zu validierenden Eigenschaft vererbt wird. Der Vorteil dieser Technik ist, dass sie sowohl vollständig als auch flexibel ist, da die Größe des endlichen Systems von der Wertung der Komplexität des Programms und der zu bewertenden Eigenschaft abhängt.

Abhängig von der Anzahl der Nodes, X und U Operatoren wird die Komplexität des Programms berechnet. Nun muss anhand der Komplexität ein Grenzwert festgelegt werden, der sich jedoch nicht an der Größe des Stacks sondern der Anzahl der aufeinander folgenden, identischen Elemente (Nodes) orientiert. Den Wert bezeichnen wir als Wertung des Stacks.

Die Komplexität alleine reicht jedoch nicht aus, um die Sicherheitseigenschaften eines Programms zu bewerten. Dazu wird ein zweiter Wert, der von der Anzahl der Elemente des Stacks abhängt, herangezogen (p-limited semantics of a program). Auf diese Weise ist es möglich, verlustfrei auch komplexe und sogar unendliche Programmaufrufsabfolgen auf ihre Sicherheitseigenschaften zu überprüfen. Dazu muss lediglich die Anzahl der Elemente auf dem Stack größer oder gleich des Komplexitätswertes sein. Dabei ist zu beachten, dass die Anzahl der aufeinander folgenden, identischen Elemente mit p bezeichnet, und hier zum Vergleich mit der Komplexität des Programms herangezogen wird.

Die Autoren bewiesen durch vollständige Induktion, dass diese Methode funktioniert und gültig ist.

2.4 Anwendung auf das Java Software Development Kit

Das Java SDK ist eines der bekanntesten Beispiele für programmiersprachenbasierte Sicherheitsarchitektur. Wir wollen nun beschreiben, wie die JDK 1.2 Sicherheitsmechanismen in unserem Framework beschrieben werden können. Darauf folgt ein Beispiel an Hand eines E-Commerce Modells.

Die Java Virtual Machine hat eine Klasse namens AccessController, mit einer Vielzahl an sicherheitsorientierten Methoden. Eine von diesen ist checkPermission, die sicherstellt, dass eine gewisse Berechtigung für die Ausführung vorhanden ist. Falls das nicht der Fall ist, wird eine Exception geworfen. Die einzige Ausnahme stellen privileged Programmblöcke dar, die diese Überprüfungen ignorieren dürfen. Die Zuweisung einer protection domain zu einem Stück Programmcode bedeutet, dass jeder Node im entsprechenden Graphen in diese protection domain

gehört und damit die Bedingung „hat Berechtigung P“ erfüllt. Privileged Programmblöcke werden mit den Anweisungen beginPrivileged und endPrivileged ausgezeichnet. Im Folgenden sind alle irrelevanten Teile des Programmcodes gestrichen worden.

```
public class ControlledVar {
  private float var;
  void write(float new_) {
    AccessController.checkPermission(Write);
    var = new_;
   }
   float read() {
     AccessController.checkPermission(Read);
     return var;
   }
  }
// System Code: System protection domain


public class AccountMan {
  private ControlledVar balance;
  public  boolean canpay(float amount) {
    AccessController.checkPermission(Canpay);
    boolean res = false;
    try {
      AccessController.beginPrivileged();
      res = balance.read() > amount;
    } finally {
      AccessController.endPrivileged();
    }
    return res;
   }
   public void debit(float amount) {
     AccessController.checkPermission(Debit);
     if (this.canpay(amount)) {
       try {
          AccessController.beginPrivileged();
          balance.write(balance.read() - amount);
       } finally {
          Accesscontroller.endPrivileged();
       }
      } else...
    }
  }
// Account Manager Code (provider protection domain)
```

Jeder Node im Graphen G erfüllt Eigenschaften entsprechend seiner protection domain, plus die Eigenschaft Priv, falls es innerhalb einer privileged Section auftritt. Wir definieren, dass D[Domain] bedeutet, dass ein Node einer Domain [Domain] angehört.

Daraus ergibt sich:
D[Provider] = P[Debit] **and** P[Canpay] **and** P[Read] **and** P[Write]
D[System] = P[Debit] **and** P[Canpay] **and** P[Read] **and** P[Write]

Das trifft zu, und bei näherer Betrachtung des dazugehörigen Aufrufsgraphen stellt man sogar fest, dass nicht einmal alle Überprüfungen notwendig sind, da im bestehenden Code einige nie direkt aufgerufen werden.

Zusammenfassung
Die Autoren weisen darauf hin, dass es den Aufwand wohl lohnen würde, wenn ein inkrementielles System zur Anwendung dieser Prinzipien entwickelt würde: im Moment funktioniert das aktuelle System zur Prüfung der Rechte gut, bis zu dem Zeitpunkt wenn eine neue Klasse dynamisch geladen wird. Dann müsste nämlich das gesamte Programm (!) erneut auf seine Sicherheitseigenschaften hin überprüft werden.

3. Kontext des Artikels im Bezug auf andere Arbeiten aus dem Bereich

Drei Richtungen von Arbeiten, die ähnliche Bereiche abdecken, sollen erläutert werden.

3.1 Sicherheitseigenschaften spezifizieren

McLean hat in seiner Arbeit über eine Trace-Sprache die Sicherheitseigenschaften spezifiziert, und wie einfach diese mit einer operationalen Semantik kombiniert werden kann, geschrieben [3].

3.2 Java Stack Inspektion Formalisieren

Wallach, Balfanz, Dean und Felten stellen Information über die Behandlung von Sicherheitsarchitekturen in Java zur Verfügung. [4]. Der Artikel bezieht sich auf eine Sicherheitseinrichtung im Netscape, das als Erweiterung der JDK 1.2 Mechanismen angesehen werden kann.

3.3 Vereinfachung von unendlichen Automaten

Hier werden zwei Hauptansätze unterschieden. Der eine stammt von Wolper [5]. Er zeigt, dass für datenunabhängige Programme (Programme die nur Daten bewegen ohne irgendwelche Operationen an ihnen durchzuführen) über unendliche Datendomains, es möglich ist diese endlose Aussagefunktion auf eine Endliche zu reduzieren über eine endliche Datendomain. Da das Programm datenunabhängig ist, ist auch die Gültigkeit der reduzierten Eigenschaften gegeben.

Der zweite Ansatz versucht nicht Einschränkungen auf die Programme anzuwenden, sondern befürwortet die Bewahrung von ganzen Fragmenten der verwendeten Logik und stellt Eigenschaften auf die eine Abstraktion erfüllen muss, um für die Modellüberprüfung bereit zu sein. Loiseaux stellt die Entscheidungskriterien der Abstraktion an den States auf. [6].

4. Zusammenfassung – Vergleich MOPS / Controlflow based Methode

Die Controlflow based Methode [1] wurde von MOPS nicht behandelt. Bei MOPS wurde alles, das im Code erhalten ist und potentiell zur Ausführung gelangen kann als gefährlich eingestuft[2]. MOPS ist auf Java nicht eingegangen, da MOPS für C Programme unter Linux konzipiert ist. MOPS hat eine sehr beschränkte Einsetzbarkeit, während die Conrtolflow based Methode sprachunabhängig ist.

5. References

[1]. T. Jensen, D. L. Metayer, and T. Thorn. Verification of control flow based security properties. In Proceedings of the 1999 IEEE Symposium on Security and Privacy, 1999.

[2]. Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In Ravi Sandhu, editor, Proceedings of the 9th ACM Conference on Computer and Communications Security, pages 235-244, Washington, DC, USA, November2002. ACM Press.

[3]. J. McLean. Providing noninterference and functional correctness using traces, J. Of Computer Security, 1:37-57,1992

[4]. D.S. Wallach, D. Balfanz, D.Dean, and E. W. Felten. Extensible security architectures for java. In 16th Symposium on Operating Systems principles, pages 116-128, October 1997

[5]. P.Wolper. Expressing interesting properties of programs in proposional temporal logic. In Proc. Of 13th ACM Symp. On Principles of Programming Languages (POPL'86), pages 184-193, 1986

[6]. C. Loiseaux, S. Graf, J. Sifakis, A.Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. Formal Methods in System Design, 6:1-35,1995