# Transitiver Schutz in Java durch Sicherheitsmetaobjekte

Thomas Riechmann, Franz J. Hauck und Jürgen Kleinöder

Universität Erlangen-Nürnberg **
Lehrstuhl für Betriebssysteme IMMD-IV
Martensstr. 1, D-91058 Erlangen
{riechmann,hauck,kleinoeder}@informatik.uni-erlangen.de
http://www4.informatik.uni-erlangen.de/~{riechmann,hauck,kleinoeder}

**Zusammenfassung** Java hat sich als Plattform für verteilte, objektorientierte Applikationen, bei denen das Thema Sicherheit wichtig ist, etabliert. Das Sicherheitsmodell von Java basiert auf Capabilities: Ein Objekt kann nur über eine Objektreferenz (=Capability) angesprochen werden. Zusätzlich können Klassen auch selbst Zugriffsschutz abhängig vom Aufrufer implementieren (Zugriffslisten-Implementierung). Bei Java ist die Sicherheitskonfiguration nicht von der Implementation der Applikationsklassen getrennt, sondern muß in die Applikationsklassen hineinimplementiert werden. Das Hauptproblem bei Capabilities ist die Verbreitungskontrolle: Da bei objektorientierten Systemen oft Objektreferenzen übergeben werden, ist es schwer, die Kontrolle zu behalten, wer auf welche Referenz zugreifen kann. Zugriffslisten haben andererseits den Nachteil, daß es einer böswilligen Applikation möglich sein kann, fremden, privilegierten Subjekten (Programmteilen, Domänen) Objektreferenzen unterzuschieben (Unix s-Bit Problem). Wir stellen ein Sicherheitsmodell vor, das diese Probleme löst. Durch spezielle Sicherheitsmetaobjekte trennen wir die Sicherheitsstrategie von der Implementierung der Applikationsklassen. Ein Sicherheitsmetaobjekt kann an eine Objektreferenz geheftet werden und schützt diese Referenz, indem es Zugriffe (Implementation von konfigurierbaren Capabilities) und Parameter- und Rückgabewerte überwacht, um die versehentliche Herausgabe von Referenzen zu verhindern bzw. herausgegebene Referenzen mit Zugriffsbeschränkungen zu versehen (Implementation von transitiven Capabilities).

## 1 Einleitung

Java ist besonders für verteilte Applikationen mit speziellen Sicherheitsanforderungen geeignet. Es ist ohne Risiko möglich, fremde Applikationen auszuführen. Mit dem Java-1.2 Sicherheitsmodell [13] kann man feingranular bestimmen, welche Systemresourcen Programme nutzen dürfen.

** Diese Arbeit wurde von der Deutschen Forschungsgemeinschaft im Rahmen des Sonderforschungsbereichs 182 „Multiprozessor- und Netzwerkkonfigurationen“, Teilprojekt B2, gefördert.

Das Sicherheitsmodell basiert auf der Objektorientierung: Ein Objekt kann nur angesprochen werden, wenn man eine Referenz auf dieses besitzt, und auch dann nur über Methoden, die in seiner Schnittstelle entsprechend deklariert sind (Capability-Implementierung). Falls konfigurierbare Zugriffsbeschränkungen nötig sind, müssen diese vom Programmierer direkt in die Applikationsklassen hineinimplementiert werden (Zugriffslisten-Implementierung). Trennung von Sicherheitsstrategie und Semantik der Implementation ist nicht möglich und der Programmierer muß bei jeder implementierten Methode überlegen, ob er Zugriffsschutz einbauen muß. Wenn er dies vergißt, können Sicherheitsprobleme entstehen. Ein solches Sicherheitsmodell widerspricht den gängigen Sicherheitsrichtlinien [10], nach denen bei allen möglicherweise sicherheitskritischen Ereignissen automatisch ein Referenzmonitor involviert werden soll.

Wir trennen die Sicherheitsstrategie von der Implementierung der Applikationsklassen. Die Sicherheitsstrategie wird durch Sicherheitsmetaobjekte realisiert. Diese können an Objektreferenzen geheftet werden und schützen dann diese Referenz: bei jeder sicherheitsrelevanten Operation werden sie involviert und können über die Zulässigkeit der Operation entscheiden. Sie implementieren so konfigurierbare, orthogonal implementierbare Capabilities.

Zusätzlich können diese Sicherheitsmetaobjekte auch den Fluß von Objektreferenzen kontrollieren: Um eine Referenz zu schützen, genügt es nicht, Methodenaufrufe über diese Referenz zu überwachen. Vielmehr müssen zusätzlich Referenzen, die bei solchen Methodenaufrufen übergeben werden oder als Ergebnis zurückgegeben werden, überwacht werden. Über solche Referenzen kann sonst unkontrolliert auf das eigentlich zu schützende Objekt oder Komponenten des Objektes zugegriffen werden. Daher kann sich das Sicherheitsmetaobjekt automatisch an alle im Zuge eines Methodenaufrufs ausgetauschten Referenzen heften und somit transitiven Schutz realisieren - und dies ohne Unterstützung des zu schützenden Objektes. Die Strategie kann komplett durch Sicherheitsmetaobjekte implementiert werden.

In Abschnitt 2 werden wir Sicherheitsmetaobjekte allgemein betrachten und zeigen, daß man mit ihnen konfigurierbare Capabilities implementieren kann. In Abschnitt 3 motivieren wir die Notwendigkeit von transitivem Schutz und zeigen, wie wir diesen durch Sicherheitsmetaobjekte implementieren können. Unseren Java-Prototyp stellen wir kurz in Abschnitt 4 vor. In Abschnitt 5 vergleichen wir unser Sicherheitsmodell mit anderen Arbeiten in diesem Bereich.

## 2 Sicherheitsmetaobjekte

In diesem Abschnitt stellen wir unser Grundkonzept, das auf Sicherheitsmetaobjekten basiert, dar. Wir beschreiben das Konzept allgemein und präsentieren einige Beispiele in Java-Syntax, bei denen jedoch syntaktischer Ballast zunächst weggelassen wird. Später werden wir darauf eingehen, wie unsere Beispiele tatsächlich in Java aussehen müßten.

Wir gehen von einem objektorientierten Programmiermodell aus, das Zugriffe auf Objekte nur über Objektreferenzen erlaubt, wie es bei Java der Fall ist

[11]. Wir betrachten in diesem Zusammenhang nur Methodenaufrufe und gehen davon aus, daß direkte Instanzvariablenzugriffe nicht erlaubt sind. (Das Modell läßt sich auch auf Variablenzugriffe erweitern, das wollen wir hier jedoch nicht betrachten.) Eine Objektreferenz ist in diesem Modell also eine Capability für Methodenaufrufe an einem Objekt.

Wir erweitern dieses Modell durch die Möglichkeit, ein oder mehrere *Sicherheitsmetaobjekte* (zu Metaobjekten siehe auch [5] und [2]) an eine Objektreferenz zu heften. Solch ein Sicherheitsmetaobjekt schützt die Referenz und kann beispielsweise entscheiden, welche Methodenaufrufe über die Referenz durchgeführt werden dürfen. Diese Sicherheitsmetaobjekte sind für die Anwendung transparent; für die Anwendung sehen geschützte und ungeschützte Referenzen gleich aus.

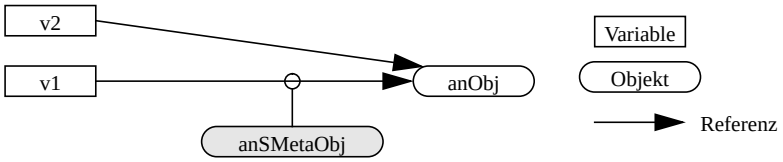


**Abb. 1.** Eine mit Sicherheitsmetaobjekt geschützte Referenz

Da Sicherheitsmetaobjekte an Referenzen (und nicht an Objekte) geheftet werden, kann es mehrere Referenzen auf dasselbe Objekt mit unterschiedlichen angehefteten Sicherheitsmetaobjekten geben. Ein Sicherheitsmetaobjekt kann mehrere Referenzen schützen und es können mehrere Sicherheitsmetaobjekte an die selbe Referenz geheftet werden. Diese Sicherheitsmetaobjekte werden dann alle nacheinander bei Aufrufen über die Referenz involviert. Abbildung 1 zeigt eine Objektreferenz auf das Objekt `anObj` in Variable `v1`, die durch das Sicherheitsmetaobjekt `anSMetaObj` geschützt ist. Zusätzlich existiert eine Referenz auf dasselbe Objekt in Variable `v2`, die ungeschützt ist.

Jede Applikation kann ohne Einschränkung Sicherheitsmetaobjekte an Referenzen heften; da diese nur den Zugriff einschränken, stellt dies kein Sicherheitsproblem dar.

### 2.1 Capabilities

Capabilities implementieren zusätzlich zu der Semantik von reinen Objektreferenzen drei weitere Konzepte: Zugriffseinschränkung, Revokation, Expiration. Wir werden nun vorstellen, wie man mit Sicherheitsmetaobjekten diese implementieren kann und dadurch alle Arten von Capabilities realisieren kann:

- Zugriffseinschränkung. Es kann nur eine eingeschränkte Menge von Methoden über eine Capability aktiviert werden. Dies können wir durch ein Sicherheitsmetaobjekt erreichen, das bei jedem Methodenaufruf prüft, ob die

Zielmethode sich in der Menge erlaubter Methoden befindet und sonst den Aufruf abweist.
- Revokation. Eine an einen anderen Programmteil weitergegebene Capability soll nachträglich ungültig gemacht werden. Das Sicherheitsmetaobjekt enthält dazu ein Statusbit, das gesetzt werden kann, um alle von ihm geschützten Referenzen zu invalidieren. Falls dieses Bit gesetzt ist, erlaubt es überhaupt keine Aufrufe mehr.
- Zeitlich begrenzte Gültigkeit. Eine Capability soll nur eine zeitlich begrenzte Gültigkeit haben. Dazu prüft das Sicherheitsmetaobjekt bei jedem Aufruf, ob das Gültigkeitsende bereits erreicht ist und entscheidet damit, ob der Aufruf zulässig ist.

### 2.2 Beispiel

Die Implementation und Verwendung eines Sicherheitsmetaobjektes, das zeitlich begrenzte Gültigkeit implementiert, soll hier beispielhaft dargestellt werden. Im folgenden Listing wird eine Objektreferenz auf eine Liste mit zeitlich begrenzter Gültigkeit versehen.

```
List l = ..... ;                        // eine Ref. auf eine Liste
SecurityMeta s =
  new MetaExpire(new Date(1,7,1999)); // Metaobjekt erz.
l = s.dstAttachTo(l);                 // Metaobjekt an Ref. heften
...                                   // Ref. l ist nun geschuetzt
x.untrustedMethod(l);                 // l an nicht-vertrauensw.
                                      // Programmteil uebergeben

class MetaExpire extends SecurityMeta {// die MetaExpire Klasse

  final Date d;                       // Gueltigkeits-Ende
  MetaExpire(Date d) { this.d = d; }  // Gueltigkeitsende in d
                                      // speichern
  void incomingCall (Object o,        // diese Methode prueft
                     Method m,        // Aufrufe ueber
                     ParamList p) {   // geschuetzte Referenzen
    if (d < getCurrentDate())         // Ref. schon ungueltig?
      throw (new SecException(...));  // dann Aufruf unzulaessig
  }                                   // sonst Aufruf zulassen
}

class X {
  untrustedMethod(List lst) {
    lst.Get();                        // Aufruf wird geprueft
} }
```

Der erste Teil des Programmes zeigt die Benutzung eines entsprechenden Sicherheitsmetaobjektes. Das Sicherheitsmetaobjekt `s` wird an die Listenreferenz

`l` geheftet. Dabei unterscheiden wir zwei Arten das Sicherheitsmetaobjekt an die Objektreferenz zu heften: *zielorientiert* und *quellorientiert*. Wir heften das Sicherheitsmetaobjekt hier zielorientiert an die Referenz (mit `dstAttachTo`), das bedeutet, daß das Sicherheitsmetaobjekt Aufrufe über die geschützte Referenz als ankommende Aufrufe betrachtet. Das Anheften mit quellorientierter Sicht werden wir später betrachten. Durch das Anheften entsteht eine neue, nun geschützte Referenz. Wir überschreiben damit die Variable `l`, so daß die einzige ungeschützte Referenz auf die Liste verschwindet. Bei jedem Aufruf über die geschützte Referenz wird nun automatisch die `incomingCall` Methode des Sicherheitsmetaobjektes aktiviert, die in unserem Beispiel prüft, ob die Gültigkeitsdauer bereits überschritten ist und in diesem Fall den Aufruf durch Erzeugen einer Ausnahme verhindert. Wir übergeben die geschützte Referenz an einen Programmteil, dem wir nicht vertrauen. Dieser hat nun keine Möglichkeit mehr, nach dem Ende der Gültigkeitsdauer auf die Referenz zuzugreifen.

Abbildung 2 zeigt das Ergebnis: Bei der Übergabe der geschützten Referenz `l` an einen anderen Programmteil wird die Referenz dupliziert. Das Sicherheitsmetaobjekt ist zielorientiert an die Referenz geheftet und betrachtet daher Aufrufe als eingehende Aufrufe (`incomingCall` wird aktiviert).

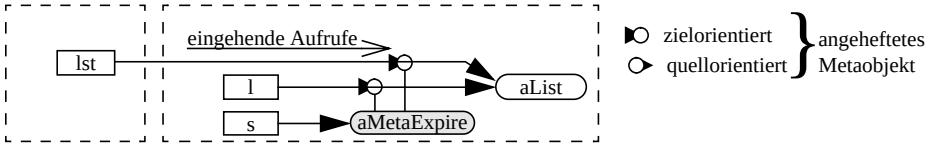


**Abb. 2.** Beispiel für zeitlich begrenzte Gültigkeit von Referenzen

Die `MetaExpire` Klasse ist generisch implementiert, sie kann alle Arten von Objekten schützen. Die im Beispiel zu schützende Liste enthält keine Unterstützung für Schutz. Implementation der Applikationssemantik und Sicherheitsimplementation sind getrennt. Methodenspezifischer Schutz ist ebenso durch Sicherheitsmetaobjekte ohne Applikationsunterstützung implementierbar, allerdings enthält dann die Implementation des Metaobjektes eventuell applikationsspezifische Teile.

## 3 Transitivität

In objektorientierten Systemen ist die Übergabe und Rückgabe von Objektreferenzen bei Methodenaufrufen einer der Basismechanismen und wird daher oft verwendet. Bei der Übergabe von Referenzen an andere Programmteile ist besondere Vorsicht geboten: Wenn eine ungeschützte Referenz übergeben wird, kann der Empfänger ohne jede Restriktion auf die Referenz zugreifen. Die Liste aus dem Beispiel könnte eine `get` Methode implementieren, die Referenzen auf Listeneinträge zurückgibt. Diese Referenzen sind nun zunächst einmal ungeschützt.

Man könnte nun die `get` Methode der Liste erweitern, daß vor der Rückgabe ein Sicherheitsmetaobjekt an die Referenz geheftet wird, um diese zu schützen. Dann würden jedoch wieder Sicherheitskonfiguration und Applikationssemantik gemischt. Wir beschreiten daher einen anderen Weg: die Sicherheitsmetaobjekte haben nicht nur über die Aufrufe die Kontrolle, sondern auch über Referenzen, die im Zuge des Aufrufs transferiert werden.

### 3.1 Einfache Transitivität

Betrachten wir zunächst Referenzen, die über geschützte Referenzen als Rückgabewert eines Methodenaufrufes übermittelt werden, wie dies bei einer `get` Methode der Liste der Fall ist.

Unsere Sicherheitsmetaobjekte können hierzu eine Methode `outgoingRef` implementieren, die vom Laufzeitsystem immer dann aufgerufen wird, wenn eine Referenz den Applikationsteil über die geschützte Referenz verläßt, also z.B. als Rückgabewert der `get` Methode transferiert wird. Um nun diese Referenz zu schützen, heftet sich das Sicherheitsmetaobjekt an solche Referenzen und versieht diese dadurch mit dem gleichen Schutz, mit dem es auch die initiale Listen-Referenz selbst schützt:

```
class MetaExpire extends SecurityMeta {
  final Date d;
  MetaExpire(Date d) { this.d = d; }

  void incomingCall(Object o, Method m, ParameterList p) { ... }
  Object outgoingRef(Object o) { // Objref o wird zurueckgegeben
    return this.dstAttachTo(o);  // o mit Metaobjekt selbst
  }                              // schuetzen
}
```

In Abbildung 3 ist das Ergebnis grafisch dargestellt: Die Referenz `lst` ist durch das Sicherheitsmetaobjekt `aMetaExpire` geschützt. Über `lst` wird nun die `get` Methode der Liste aufgerufen, die das Element `anEntry` zurückgibt (etwa durch den Aufruf: `ent=lst.Get()` ). Bevor diese Referenz nun an den Aufrufer zurückübergeben wird, wird diese an die `outgoingRef` Methode des Sicherheitsmetaobjektes übergeben. Diese hängt das Metaobjekt selbst an die Referenz,

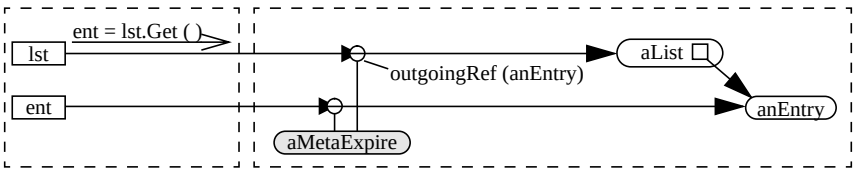


**Abb. 3.** Beispiel für den Schutz zurückgegebener Referenzen

die Referenz ist ebenfalls geschützt und hat ebenfalls nur eine zeitlich begrenzte Gültigkeit.

### 3.2 Volle Transitivität

Tatsächlich betrachtet unsere Implementation im vorigen Abschnitt nur Referenzen, die als Rückgabewerte übergeben werden. Referenzen, die als Parameter übergeben werden, scheinen zunächst unkritisch, da diese in die andere Richtung zeigen. Unsere Liste könnte z.B. eine Such-Methode implementieren, die als Parameter eine Referenz auf ein Elementobjekt bekommt und prüft, ob sich das Element schon in der Liste befindet:

```
class List { .....
  boolean search(Entry e) {       // Such-Methode der Liste
    for (Entry a=first; ....) {  // Durch die Liste iterieren
      if (e.equals(a)) {
        return true;
      }                           // Eintrag gefunden
    }
    return false;                 // Eintrag nicht gefunden
} }
```

Da die Referenz auf dieses Element in die andere Richtung zeigt, nämlich von der Liste nach außen, benötigt man für diese Referenz keinen Schutz gegen unbefugte Aufrufe. Ganz ohne Schutz kommt man hier jedoch auch nicht aus, sonst kann diese Referenz als trojanisches Pferd wirken; über sie können unbemerkt Referenzen nach außen dringen. In Abbildung 4 ist diese Situation dargestellt. Über die geschützte Referenz `lst` wird die Such-Methode der Liste mit einer Referenz (`trojH`), die später als trojanisches Pferd benutzt wird, aufgerufen. Die Such-Methode iteriert durch die Liste und testet für jeden Eintrag, ob dieser mit dem übergebenen Objekt übereinstimmt. Dies erfolgt durch Aufruf der `equals` Methode an der übergebenen Objektreferenz (`e`). Diese `equals` Methode erhält nun wiederum ein Listenelement als Parameter. Das Objekt `aTrojH` hat nun eine Referenz auf ein Listenelement, die nicht geschützt ist. Die Sicherheitsstrategie ist noch nicht komplett transitiv.

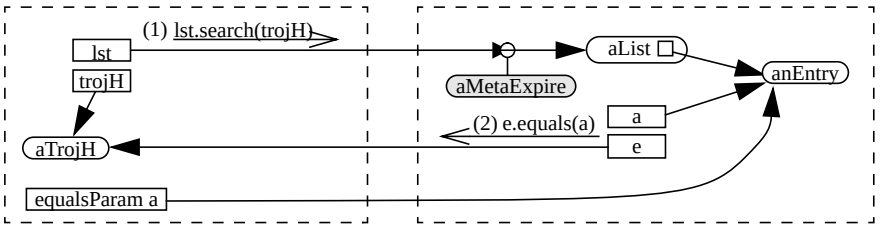


**Abb. 4.** Beispiel für trojanische-Pferd Referenzen

Das Problem ist die übergebene Referenz: Wenn eine Referenz über eine geschützte Referenz übergeben wird, muß sie ebenfalls geschützt werden, und zwar nicht gegen unbefugte Aufrufe, sondern vielmehr nur gegen Weitergabe ungeschützter Referenzen. Solche Referenzen zeigen in die andere Richtung: von den geschützten Objekten aus nach außen. Daher benötigt ein Sicherheitsmetaobjekt, das eine solche Referenz schützen soll, eine andere Sicht auf die Referenz: *quellorientierte* Sicht. Es betrachtet dann Aufrufe über die Referenz als ausgehende Aufrufe, übergebene Parameter als herausgegebene Referenzen, Rückgabewerte als eingehende Referenzen. Um eine Referenz quellorientiert zu schützen, kann man ein Sicherheitsmetaobjekt quellorientiert (durch die Methode `srcAttachTo` des Metaobjektes) an diese anheften:

```
class MetaExpire extends SecurityMeta {
  void incomingCall(Object o, Method m, ParameterList p) { ... }
  void outgoingCall(...) {}      // ausgehende Aufrufe
                                 // unbeschraenkt zulassen

  Object outgoingRef(Object o) { // o verlaesst gesch. Bereich
    return this.dstAttachTo(o);  // o zielorientiert schuetzen
  }

  Object incomingRef(Object o) { // o betritt gesch. Bereich
    return this.srcAttachTo(o);  // o quellorientiert schuetzen
} }
```

Abbildung 5 zeigt das Ergebnis: Die initiale Referenz `lst` ist zielorientiert geschützt. An den bei dem Methodenaufruf (1) übergebene Parameter `trojH` heftet sich das Sicherheitsmetaobjekt selbst quellorientiert (2). Wenn nun über diesen Parameter (lokale Variable `e`) wiederum eine Methode aktiviert wird (3), kann das Sicherheitsmetaobjekt Parameter zielorientiert schützen (4), so daß der ursprüngliche Aufrufer keine ungeschützten Referenzen bekommt.

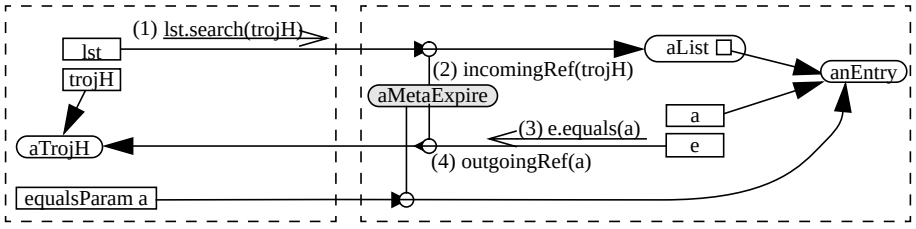


**Abb. 5.** Beispiel für den Schutz zurückgegebener Referenzen

Diese Sicherheitsmetaobjekt- Implementation ist vollständig transitiv und kann für alle Arten von Schutzstrategien verwendet werden, wie Zugriffsbeschränkungen, Rückruf von Referenzen, zeitlich beschränkte Gültigkeit. Bei-

spielsweise könnte man ein Sicherheitsmetaobjekt implementieren, welches das Listen-Objekt gegen Schreibzugriffe schützt. Der Schutz wirkt dann nicht nur auf die Liste, sondern auch auf die Elemente der Liste, und dies, ohne die Listen-Implementation verändern zu müssen.

## 4 Implementation

Die Implementation des hier vorgestellten Sicherheitsmodells in Java ist auf verschiedene Arten möglich. Eine Möglichkeit ist eine Erweiterung der Java Maschine, die es erlaubt, Sicherheitsmetaobjekte an beliebige Objektreferenzen zu heften [2]. Da dann jedoch Kompatibilitätsprobleme entstehen (Programme laufen dann nur noch mit der geänderten Java Maschine) wurde hier ein anderer Weg beschritten. Um eine Referenz zu schützen, wird ein Stellvertreter-Objekt vor die Referenz gehängt, das zuerst die Methoden des zuständigen Sicherheitsmetaobjektes aktiviert und dann den Aufruf durchführt (ähnlich wie bei Java-RMI). Für jede Klasse werden automatisch entsprechende Stellvertreter-Klassen generiert. Wegen des starren Typkonzeptes von Java können über einen Stellvertreter allerdings nur Methoden aufgerufen werden, die entweder von der Wurzelklasse Object geerbt werden oder durch Schnittstellen (Interfaces) deklariert werden, die durch die Klasse des Zielobjektes implementiert werden (gleiche Einschränkung wie bei RMI).

Bis auf diese Einschränkung, die sich durch Deklaration von Schnittstellen umgehen läßt, wurde das Modell jedoch komplett implementiert und funktioniert mit allen Java Maschinen, auch beispielsweise im Netscape-Browser.

## 5 Verwandte Arbeiten

Die gängigen Sicherheitsmodelle für objektorientierte Systeme verwenden meist primär Capabilities [1], die eingeschränkt werden können (Hydra [16], Amoeba [14]), zeitlich begrenzt gültig sind (Kerberos V5 [3], das z.B. von DCE [7] verwendet wird) und auch nachträglich ungültig gemacht werden können (CORBA [6]). Java besitzt dafür nur eingeschränkte Möglichkeiten: Referenzen können weder ungültig gemacht werden, noch mit zeitlich begrenzter Gültigkeit versehen werden. Lediglich Zugriffsbeschränkung kann gewählt werden, allerdings auch nur grobgranular: Durch die Attribute private, protected, public kann bestimmt werden, ob nur Klassen eines Paketes oder der Vererbungshierarchie Zugriff auf bestimmte Methoden haben.

Das Problem der Verbreitungskontrolle von Capabilities [4], das in dieser Arbeit durch die Transitivität der Sicherheitskonfiguration gelöst wird, wird bei den meisten objektorientierten Systemen durch Zugriffslisten angegangen. Um einen Methodenaufruf durchzuführen, muß der Aufrufer eine Referenz auf das Zielobjekt besitzen und in der Zugriffskontrollliste des Zielobjektes muß für ihn Zugriff auf diese Methode erlaubt sein. Dadurch ist es leicht, die Kontrolle darüber zu behalten, wer auf ein bestimmtes Objekt zugreifen kann. Die Problematik dieses

Verfahrens liegt jedoch in der Wahl der Identitäten, d.h. unter welcher Identität ein Methodenaufruf durchgeführt wird.

Es gibt dazu mehrere gängige Verfahren: domänenbasierte, threadbasierte und explizite Identitäten [15]. Java 1.0.2 [12] implementiert domänenbasierte Identitäten. Klassen die lokal geladen werden gehören zu einer hochprivilegierten Domäne, Klassen die über Netzwerk von einem bestimmten Rechner geladen werden zu einer weniger privilegierten. In Java 1.2 [13] werden zusätzlich threadbasierte Identitäten angeboten. Eine privilegierte Domäne kann den aktuellen Thread mit Rechten versehen, die dann in tieferen Aufrufhierarchien verfügbar sind. So gesehen ist dies eine Mischung von expliziten Identitäten und threadbasierten Identitäten.

Die Probleme, die bei allen drei Möglichkeiten auftreten, sollen hier kurz erläutert werden:

- Domänenbasierte Identitäten. Bei domänenbasierten Identitäten können Objektreferenzen untergeschoben werden (ähnlich wie beim Unix s-bit Problem). Eine nicht-privilegierte Domäne besitzt eine Objektreferenz, auf die sie wegen fehlender Rechte nicht zugreifen kann. Dann kann sie versuchen, diese Referenz einer privilegierten Domäne unterzuschieben. Sie ruft bei einem Objekt in der privilegierten Domäne eine Methode auf und übergibt ihr die Referenz als Parameter. Wenn nun die privilegierte Methode darauf zugreift, erfolgt dies mit ihren hohen Privilegien.
- Threadbasierte Identitäten. Bei threadbasierten Identitäten besteht die Gefahr der versehentlichen, unkontrollierten Rechteweitergabe. Wenn ein privilegierter Thread eine Methode aufruft, kann diese über die Privilegien frei verfügen, d.h. man muß bei Interaktion mit anderen Programmteilen gut überlegen, ob man vor einem Aufruf die Privilegien aufgeben muß.
- Explizite Identitäten. Dies führt zur Mischung von Sicherheitskonfiguration und Applikationssemantik.

## 6 Zusammenfassung und Ausblick

Das ursprüngliche Sicherheitsmodell von Java 1.0 basierte im wesentlichen auf Capabilities. Es stellte sich heraus, daß die von Java implementierten Capabilities zu wenig Möglichkeiten bieten. Statt nun wie bei Java 1.2 vorzugehen und auf Zugriffslisten als zusätzlichen Mechanismus zu setzen, wurden in dieser Arbeit die Möglichkeiten, die sich mit Capabilities bieten, erweitert.

Die Capabilities wurden durch Sicherheitsmetaobjekte implementiert und dadurch nicht nur konfigurierbar gemacht, so daß sie alle Möglichkeiten von Capabilities bieten (Revokation, zeitliche Begrenzung und Einschränkung), sondern wurden zusätzlich mit Transitivität versehen, so daß das Problem der versehentlichen Verbreitung von ungeschützten Capabilities gelöst wird.

Die Sicherheitsmetaobjekte lassen sich unabhängig von der Applikation implementieren, Applikationsklassen müssen dazu nicht angepaßt werden. Wenn

bei einer Applikation initial Referenzen zwischen den verschiedenen Applikationsteilen ausgetauscht werden, müssen diese mit den entsprechenden Sicherheitsmetaobjekten versehen werden, danach wird der Schutz automatisch transitiv durch die Sicherheitsmetaobjekte realisiert.

Ganz ohne Zugriffskontrolllisten wird man nicht auskommen: Für die initial ausgetauschten Referenzen könnte man diese beispielsweise benötigen. Wir schlagen dazu rollenbasierte Identitäten vor [9]. Diese bieten im Gegensatz zu thread- oder domänenbasierten Identitäten keine Angriffsmöglichkeiten durch Unterschieben von Referenzen. Sie lassen sich ebenfalls mit Sicherheitsmetaobjekten implementieren und wir werden diese auch in unseren Prototyp integrieren.

## Literatur


1. Dennis, J.B.; Van Horn, E.C.: Programming Semantics for Multiprogrammed Computations. Comm. of the ACM, März 1966
2. Kleinöder, J.; Golm, M.: MetaJava: An Efficient Run-Time Meta Architecture for Java. IWOOOS '96 workshop, Seattle, 1996
3. Kohl, J., Neuman, C.: The Kerberos Network Authentication Service (V5). IETF Network Working Group Request for Comments 1510, September 1993
4. Lampson, B.: A Note on the Confinement Problem, In: Communications of the ACM 1973, Oktober, 1973
5. Maes, P.: Computational Reflection, Ph.D. Thesis, Technical Report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987
6. OMG: CORBA Security, OMG Document Number 95-12-1, 1995
7. OSF: Security in a Distributed Computing Environment, Open Software Foundation, White Paper, 1992
8. Riechmann, T.; Hauck, F. J.: Meta objects for access control: extending capability-based security, In: Proc. of the ACM New Security Paradigms Paradigms Workshop 1997, Great Langdale, UK, Sept. 1997
9. Riechmann, T.; Kleinöder, J.: Meta objects for access control: Role-based Principals, In: Proc. of the Third Australasian Conference on Information Security and Privacy, Springer LNCS, Brisbane, Austalien, Juli 1998
10. Saltzer, J. H.; Schroeder, M. D.: The Protection of Information in Computer Systems, In: Proc. of the IEEE, volume 63, number 9, Sept. 1975
11. Sun Microsystems Comp. Corp.: HotJava: The Security Story, White Paper, 1995
12. Sun Microsystems Comp. Corp.: The Java Language Environment, White Paper, 1995
13. Sun Microsystems Comp. Corp.: Java Security Architecture, JDK 1.2 Draft, 1997
14. Tanenbaum, A. S.; Mullender, S. J.; van Renesse, R.: Using sparse capabilities in a distributed operating system. Proc. of the 6th Int. Conf. on Distr. Comp. Sys., pp. 558-563, Amsterdam, 1986
15. Wallach, D. S.; Balfanz, D.; Dean, D.; Felten, E. W.: Extensible Security Architecture for Java. SOSP 1997: p. 116-128, Okt. 1997, Saint-Malo, France
16. Wulf, W.; Cohen, E.; Corwin, W.; Jones, A.; Levin, R.; Pierson, C.; Pollack, F.: HYDRA: The Kernel of a Multiprocessor Operating System. Communications of the ACM, 1974