Access Control Systems Lab
Sommersemester 2017

Praktikumsbericht 3

Jean-Marc Hendrikse - 1751591

Prof. Dr. Hannes Hartenstein, Alexander Degitz, Jan Grashöfer, Till Neudecker
Forschungsgruppe Dezentrale Systeme und Netzdienste
Karlsruher Institut für Technologie (KIT)

Einleitung

In dieser Übung wird anhand eines Datenmodels auf JSON-Basis für eine RBAC- und ReBAC-Instanz ein sogenannter Policy Decision Point umgesetzt, der anhand von Policies (deutsch Regeln) Entscheidungen treffen soll. Dadurch kann ein Zugriffsystem erstellt werden, das Ressourcen vor unerlaubten Zugriffen schützt und nur Benutzern die Erlaubnis erteilt, die eine entsprechende Rolle innehaben.

1 RBAC

Das Role Based Access Control (Kurzform RBAC) ist ein Verfahren zur Zugriffskontrolle und -steuerung. Besonders in Domänen mit vielen Benutzern, die verschiedene Berechtigungen auf Ressourcen ausführen können, wird dieser Ansatz größtenteils verwendet, um die Komplexität durch die Zuweisung von Benutzern zu Rollen und Rollen zu Berechtigungen (engl. permission) zu verringern.

1.1 Umsetzung

Nachfolgend wird beschrieben wie man eine RBAC-Insatz umsetzen kann. Als grundlegendes Datenmodel steht uns die Datei `rbac.json` zur Verfügung. Die Datei beinhaltet die folgenden fünf Elemente:

- "users": Eine Liste aller Benutzer, die in dem System vorhanden sind.
- "roles": Liste aller Rollen in dem System.
- "roleassignment": Jedem Benutzer ist eine Liste von Rollen zugeordnet, die dem Benutzer zugewiesen sind.
- "rolehierarchy": Rollen können die Rechte von untergeordneten Rollen erhalten und somit auf gleicher Rollenbasis agieren.
- "permissionassignment": Hier sind die Rollen definiert, die auf eine bestimmte Ressource mit dem Feld `name`, zugreifen können.

Die Zugriffsentscheidungen werden durch Funktionen eines Python-Skripts auf Basis dieser Datenstruktur getroffen. Über die Kommandozeile werden die Eingabewerte `user` und `resource` eingelesen, die zur Zugriffsentscheidung benötigt werden: python rbac.py Alice Angola.

```
1 #!/usr/bin/python
2 import sys
3  # Commandline inputs
4      name = sys.argv[1] # Argument for name of person accessing the resource
5      resource = sys.argv[2] # Name of the resource a person tries to access
```

Listing 1: Auszug aus dem Python-Skript: Initialisierung von user und resource

In Listing 1 wird der Name und die Ressource mit der entsprechenden Benutzereingabe initialisiert. Die JSON-Datei wandeln wir, wie in Listing 2, mittels `json.loads()` in ein JSON-Objekt um, sodass wir auf den Daten in Python operieren können.

```
1 if len(sys.argv) == 3:
2    # Commandline inputs
3      name = sys.argv[1] # Argument for name of person accessing the resource
4      resource = sys.argv[2] # Name of the resource a person tries to access
5
6      pa_list = ''
7      jdata = json.loads(open('data/rbac.json').read()) # Reads the content of
       the given json file
```

Listing 2: Über json.load() wird der Inhalt der JSON-Datei eingelesen und in ein JSON-Objekt umgewandelt

Als nächstes wäre es gut zu wissen, ob die eingegeben Daten valide sind und ob diese überhaupt in unserem Datensatz existieren. Somit können bereits früh Entscheidungen getroffen werden ohne intensive Berechnungen vorzunehmen. Gleichzeitig initialisieren wir die Liste mit der Rechtevergabe, in der alle Rollen festgehalten sind, die auf die Ressource zugreifen dürfen.

```
1      if name in jdata['users']:
2          # gets the permission list by resource name
3          for permission in jdata['permissionassignment']:
4              if permission['name'] == resource:
5                  pa_list = permission['pa']
6                  break # breaks the loop if pa was found
7          if pa_list == '':
8              print False
```

Listing 3: Benutzereingabe überprüfen und Liste mit den Rollen laden.

Wie bereits beschrieben, werden den Benutzern Rollen zugewiesen. Aus diesem Grund muss mindestens eine der Rollen des Benutzers mit mindestens einer zugewiesenen Rolle für den Zugriff auf die Ressource übereinstimmen. In Listing 4 wird dies durch einen einfachen Vergleich in Zeile 5 umgesetzt. Ist eine der Rollen des Benutzers in der Liste enthalten wird abgebrochen und der boolsche Wert “True“ zurückgeliefert. Ist dies nicht der Fall besteht noch die Möglichkeit, dass über die Rollenhierarchie eine Rolle des Benutzers eine übergeordnete Rolle von einer der Ressource zugewiesenen Rolle ist. Somit hätte die Rolle des Benutzers alle Rechte, die auch die untergeordnete Rolle besitzt. Eine möglichst effiziente Umsetzung erfolgt durch eine Breitensuche entlang des hierarchischen Baumes. Die Implementierung der Breitensuche ist rekursive und dem Algorithmus `Breadth-First-Search` nachempfunden [1].

```
1 roleassignment = jdata['roleassignment'][name]
2          hasPerm = False
3
4          for role in roleassignment:
```

```
5               if role in pa_list:
6                   hasPerm = True
7                   break
8               else:
9                   for pa_role in pa_list:
10                      if is_child(jdata['rolehierarchy'], role, pa_role):
11                          hasPerm = True
12                          break
```

Listing 4: Überprüfung der Rollen zur Zugriffsentscheidung

Zum Abschluss wird abhängig von den obigen entscheidungen entweder `True` oder `False` ausgegeben:

```
1       print hasPerm
2       else:
3           print False
```

Listing 5: Ausgabe True, wenn Zugriff gewährt wird. Andernfalls False.

2 ReBAC

Benutzer in Social Networks sind über Beziehungen miteinander Verknüpft. Beispielsweise können durch eine Einstellung bei Facebook nur die eigenen Freunde sehen, was ein Benutzer gepostet oder hochgeladen hat. Der Besitzer einer Ressource kann somit anhand von Beziehungen zu demjenigen bestimmen, der auf die Ressource zugreifen möchte. So einen Ansatz nennt man Relation-Based Access Control (kurz ReBAC). Im Nachfolgenden möchten wir eine solche Insatz wieder anhand eines gegebenen Datenmodels erstellen.

2.1 ReBAC Implementierung

Ähnlich wie in Kapitel 1.1 bietet eine JSON-Datei die Grundlage für unsere Zugriffsentscheidungen. Das JSON-File beinhaltet die nachfolgenden vier Elemente:

- “user“: Liste aller Benutzer des Systems.
- “usergraph“: Freundesliste für jeden Benutzer.
- “policies“: Für jeden Benutzer TRP- und TUP-Regel
- “resources“: Für jede einzelne Ressource angabe von Name, den Besitzer der Ressource und alle Zielnutzer.

Auch hier müssen für den Zugriff verschiedene Regeln ausgewertet werden. Zum einen wird die TRP-Regel des besitzenden Nutzers einer Ressource und zum anderen die TUP-Regeln der Zielnutzer betrachtet. Diese Regeln besitzen eine Angabe wie lang der kürzeste Pfad zwischen zwei Benutzern sein darf, um auf eine Ressource zuzugreifen.
Im Grunde werden auch in ReBAC Benutzername und die Ressource als Eingabeparameter erwartet. Zusätzlich wird noch eine Angabe gemacht, ob die Regel des zugreifenden Benutzers mit allen Regeln/Policies aller Benutzer einer Ressource (Umsetzung durch das Schlüsselwort `ALL`) oder nur mit mindestens einer Regel übereinstimmen muss. Listing 7 zeigt einen Auschnitt, der dies umsetzt:

```
user = sys.argv[1]
    resource_name = sys.argv[2]
    conflict_res = sys.argv[3]

    jdata = json.loads(open('data/rebac.json').read())
    graph = jdata['usergraph']
    policies = jdata['policies']
    resource = jdata['resources']

    if user not in jdata['users']:
        print False
    else:
        switcher = {
            "ALL": all_conj(user, resource),
            "ANY": any_conj(user, resource)
        }
        print switcher.get(conflict_res, False)
else:
    print "Please pass 3 arguments. E.g. Alice Angola ALL."
    print "-> False"
```

Listing 6: Auschnitt der ReBAC-Zugriffsentscheidung

Wie in Listing 7 zu erkennen ist, wird die Funktion `all_conj()` ausgeführt, wenn der Benutzer das Schlüsselwort `ALL` eingibt. In dieser Funktion wird verglichen ob die Distanz des Benutzers zu dem "controller" über den `usergraph` in der TRP-Regel liegt und ob zusätzlich die Distanz des Benutzers zu allen "target"-Benutzern in den jeweiligen TUP-Regeln liegt. Ist dies der Fall wird der boolsche Wert `True` ausgegeben. Auch hier wird wieder auf den Breadth-First-Search-Algorithmus zurückgegriffen. Dieses Mal wird allerdings die Distanz zwischen zwei Benutzern in einem Benutzergraphen berechnet.

```
def all_conj(userName, resource_list):
    res = {}
    for res_elem in resource_list:
        if res_elem['name'] == resource_name:
            res = res_elem
            break
  # Get controller
    controller = res['controller']
    # calculate the distance between the accessing user and the controller
    distance = len(bfs(graph, userName, controller)) - 1
    policy = policies[controller]["trp"]

  # Interprete TRP-Policy and return True or False
    if not access_decision(distance, policy):
        return access_decision(distance, policy)

  # For every target user calculate distance between user and target user
    with bfs
    for target_user in res['target']:
        distance = len(bfs(graph, userName, target_user)) - 1
        policy = policies[target_user]["tup"]
        if not access_decision(distance, policy):
            return access_decision(distance, policy)

    return True
```

Listing 7: Umsetzung der ALL-Variante, welche eine Konjunktion aller Regeln umsetzt.

In Listing 8 werden die beiden Helfer-Funktionen zur Interpretation der Distanzen durch "<", ">" und "=" und zur Berechnung der Distanz dank des BFS-Algorithmus aufgeführt:

```
def access_decision(distance, policy, ):
```

```
    policy_dis = int(policy[2])
    interprete = {
        ">": distance > policy_dis,
        "<": distance < policy_dis,
        "=": distance == policy_dis
    }
    return interprete.get(policy[1], False)


def bfs(graph, start, end):
    # maintain a queue of paths
    queue = []
    # push the first path into the queue
    queue.append([start])
    while queue:
        # get the first path from the queue
        path = queue.pop(0)
        # get the last node from the path
        node = path[-1]
        # path found
        if node == end:
            return path
        # enumerate all adjacent nodes, construct a new path and push it into the queue
        for adjacent in graph.get(node, []):
            new_path = list(path)
            new_path.append(adjacent)
            queue.append(new_path)
```

Listing 8: Hilffunktionen zur Interpretation und BFS

Neben der Variante, in der alle Policies mit der Distanz verglichen werden, gibt es noch die Umsetzung mit dem Schlüsselwort `ANY`, wodurch entweder nur die Distanz des Benutzers zum Kontroller mit der TRP-Regel oder die Distanz des Benutzers zu einem der target-Benutzern mit dessen TUP-Regel übereinstimmen muss:

```
def any_conj(userName, resource_list):
    res = {}
    for res_elem in resource_list:
        if res_elem['name'] == resource_name:
            res = res_elem
            break

    controller = res['controller']
    distance = len(bfs(graph, userName, controller)) - 1
    policy = policies[controller]["trp"]

    if access_decision(distance, policy):
        return access_decision(distance, policy)

    for target_user in res['target']:
        distance = len(bfs(graph, userName, target_user)) - 1
        policy = policies[target_user]["tup"]
        if access_decision(distance, policy):
            return access_decision(distance, policy)

    return False
```

Listing 9: Umsetzung der ANY-Variante umsetzt.

2.2 Erweiterung der ReBAC-Insatz

Zur Erweiterung der ReBAC-Instanz aus vorherigem Abschnitt bietet es sich an weitere Beziehungstypen zwischen den Benutzern miteinzuführen. So wäre denkbar neben `friends` auch `family` oder `coworker` mitaufzunehmen. Dafür müsste man wie in Listing 10 den Usergraph erweitern.

```
1  "usergraph": {
2     "Willy": {
3         "friends": [
4             "Melina",
5             "Juliana",
6             "Madella",
7             "Jerrilee",
8             "Binni",
9             "Amalita",
10            "Alicea"
11          ],
12          "coworker": [
13            "Peter",
14            "Stephanie",
15            "Melina"
16          ],
17          "family": [
18        "Susanne",
19        "Diana",
20        "Thomas"
21            ]
22    }, ...}
```

Listing 10: Anpassung des usergraphen um weitere Beziehungstypen

Weiterhin müssten auch nun die `policies` angepasst werden, indem die Distanzen zu `coworker` und `family` in den TUPs und TRPs angegeben werden, wodurch letztlich die Policy Language angepasst wird. Letztlich wird, wie im vorherigen Abschnitt, die Distanz zwischen Benutzer, der auf die Ressource zugreifen möchte, und dem controller und target usern anhand von `friends`, `coworker` und `family` berechnet und mit den Policies verglichen

2.3 Umwandlung einer nicht-hierarchischen RBAC-Instanz in eine ReBAC-Instanz

Ein Ansatz bei der Umwandlung einer RBAC-Instanz in eine ReBAC-Instanz wäre die Rollen aus dem RBAC in `user` des ReBAC umzuwandeln. Den Usergraph könnte man beispielsweise so anpassen, dass alle `user`, die eine bestimmte Rolle im RBAC besitzen als Liste der entsprechenden transformierten Rollen den Usern zugeordnet werden.

```
1  { ...
2  "roleassignment": {
3  "Janeva": [
4        "information",
5        "product",
6        "creative"
7      ],
8      "Marcia": [
9        "product",
10       "data",
11     "communications"
12     ],
13     "Anni": [
14       "communications",
```

```
15          "creative"
16        ]
17    }
18    ...}
```

Listing 11: Roleassignment in RBAC.

Eine Transformierung von Listing 11 in ein entsprechendes ReBAC-Modell ist im nachfolgenden Listing 12

```
1   {
2     "usergraph": {
3       "information": [
4         "Janeva"
5       ],
6       "product": [
7         "Janeva",
8         "Marcia"
9       ],
10        "creative": [
11        "Janeva",
12        "Anni"
13      ],
14        "communications": [
15        "Anni",
16        "Marcia"
17      ],
18        "data": [
19        "Marcia"
20      ]}...}
```

Listing 12: Modifizierter Usergraph durch Umwandlung der roleassignments aus RBAC

Auch das `permissionassignment` Listing 13 kann im äquivalenten `resources`-Feld aus dem ReBAC-Datensatz umgewandelt werden Listing 14, indem die target user Rollen werden.

```
1  {
2    "permissionassignment": [
3      {
4        "pa": [
5          "creative",
6          "product"
7        ],
8        "name": "Afghanistan"
9      },
```

Listing 13: Permission assignment in RBAC

```
1  "resources": [
2      {
3        "name": "Afghanistan",
4        "target": [
5          "product",
6          "creative"
7        ]
8      }, ...
9      }
```

Listing 14: Äquivalent zum RBACs permissionassignment das modifizierte rresourcesFeld

Abschließend müssen natürlich noch Policies so angepasst werden, dass sie auch nur den Benutzern Zugriff erteilen, die mit den entsprechenden Rollen (bzw. in ReBAC neue Benutzer) direkt verbunden sind:

```
  policies": {
     "product": {
       "tup": "h=1"
     },

     "creative": {
       "tup": "h=1"
     },
     "communications": {
       "tup": "h=1"
     },
     "information": {
       "tup": "h=1"
     },

```

Listing 15: Modifizierte Policies

Anhang

Nachfolgend die kompletten Programme.

```
#!/usr/bin/python
import sys
import json

# finds shortest path between 2 nodes of a graph using BFS
def is_child(graph, start, goal):
    # keep track of explored nodes
    explored = []
    # keep track of all the paths to be checked
    queue = [[start]]

    # return path if start is goal
    if start == goal:
        return True

    # keeps looping until all possible paths have been checked
    while queue:
        # pop the first path from the queue
        path = queue.pop(0)
        # get the last node from the path
        node = path[-1]
        if node not in explored:
            neighbours = graph[node]
            # go through all neighbour nodes, construct a new path and
            # push it into the queue
            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)
                # return path if neighbour is goal
                if neighbour == goal:
                    return True

            # mark node as explored
            explored.append(node)

    # in case there's no path between the 2 nodes
    return False


```

```
if len(sys.argv) == 3:
    # Commandline inputs
    name = sys.argv[1] # Argument for name of person accessing the resource
    resource = sys.argv[2] # Name of the resource a person tries to access

    pa_list = ''
    jdata = json.loads(open('data/rbac.json').read()) # Reads the content of
    the given json file

    if name in jdata['users']:
        # gets the permission list by resource name
        for permission in jdata['permissionassignment']:
            if permission['name'] == resource:
                pa_list = permission['pa']
                break # breaks the loop if pa was found
        if pa_list == '':
            print False

        roleassignment = jdata['roleassignment'][name]
        hasPerm = False

        for role in roleassignment:
            if role in pa_list:
                hasPerm = True
                break
            else:
                for pa_role in pa_list:
                    if is_child(jdata['rolehierarchy'], role, pa_role):
                        hasPerm = True
                        break

        print hasPerm
    else:
        print False
else:
    print "Please pass 2 arguments"
```

Listing 16: rbac.py

```
#!/usr/bin/python
import sys
import json


def all_conj(userName, resource_list):
    res = {}
    for res_elem in resource_list:
        if res_elem['name'] == resource_name:
            res = res_elem
            break

    controller = res['controller']
    distance = len(bfs(graph, userName, controller)) - 1
    policy = policies[controller]["trp"]

    if not access_decision(distance, policy):
        return access_decision(distance, policy)

    for target_user in res['target']:
        distance = len(bfs(graph, userName, target_user)) - 1
        policy = policies[target_user]["tup"]
        if not access_decision(distance, policy):
            return access_decision(distance, policy)

    return True
```

```
27
28
29  def any_conj(userName, resource_list):
30      res = {}
31      for res_elem in resource_list:
32          if res_elem['name'] == resource_name:
33              res = res_elem
34              break
35
36      controller = res['controller']
37      distance = len(bfs(graph, userName, controller)) - 1
38      policy = policies[controller]["trp"]
39
40      if access_decision(distance, policy):
41          return access_decision(distance, policy)
42
43      for target_user in res['target']:
44          distance = len(bfs(graph, userName, target_user)) - 1
45          policy = policies[target_user]["tup"]
46          if access_decision(distance, policy):
47              return access_decision(distance, policy)
48
49      return False
50
51
52  def access_decision(distance, policy, ):
53      policy_dis = int(policy[2])
54      interprete = {
55          ">": distance > policy_dis,
56          "<": distance < policy_dis,
57          "=": distance == policy_dis
58      }
59      return interprete.get(policy[1], False)
60
61
62  def bfs(graph, start, end):
63      # maintain a queue of paths
64      queue = []
65      # push the first path into the queue
66      queue.append([start])
67      while queue:
68          # get the first path from the queue
69          path = queue.pop(0)
70          # get the last node from the path
71          node = path[-1]
72          # path found
73          if node == end:
74              return path
75          # enumerate all adjacent nodes, construct a new path and push it into
        the queue
76          for adjacent in graph.get(node, []):
77              new_path = list(path)
78              new_path.append(adjacent)
79              queue.append(new_path)
80
81
82  if len(sys.argv) == 4:
83      user = sys.argv[1]
84      resource_name = sys.argv[2]
85      conflict_res = sys.argv[3]
86
87      jdata = json.loads(open('data/rebac.json').read())
88      graph = jdata['usergraph']
89      policies = jdata['policies']
```

```
90      resource = jdata['resources']
91
92      if user not in jdata['users']:
93          print False
94      else:
95          switcher = {
96              "ALL": all_conj(user, resource),
97              "ANY": any_conj(user, resource)
98          }
99          print switcher.get(conflict_res, False)
100 else:
101     print "Please pass 3 arguments. E.g. Alice Angola ALL."
102     print "-> False"
```

Listing 17: rebac.py

Literatur

[1] https://pythoninwonderland.wordpress.com/2017/03/18/how-to-implement-breadth-first-search-in-python/