ARBAC analyser report

Lorenzo Cazzaro

April 2020

Contents

1 Introduction [1]

Checking the security of an access control system is a difficult task, more than finding an attack.

The difficulty also emerges and increases when we have to deal with systems based on ARBAC model, where each user has a set of roles assigned,

each role has a set of permissions granted, roles are in hierarchy and only certain roles are administrative, then can assign or revoke some roles. Adding the administrators increases the difficulty of verification.

The third challenge of the Security 2 course at Ca' Foscari University of Venice consists in implementing a ARBAC analyser for small policies, a program which parses the specication of a role reachability problem and returns its solution.

It's taken in account only the user-role administration component of ARBAC, while the permission-role and role-role administration components are not considered.

2 Formalization [1]

2.1 ARBAC elements

The set of users U and roles R are supposed to be finite. UR is the set of the actual user-to-role assignments in the form of pairs, where the first element is the user and the second is the role.

CA is the set of can-assign rules, that are 4-tuples in which there is firstly the administrative role that can assign the target role, last element of the rule, and the second and third elements are the set of positive preconditions, roles that an user have to possess in order to possess the target role, and negative preconditions, roles that an user must not posses in order be assigned to the target role. CR is the set of can-revoke rules, that are pairs in which the first element is the role that can revoke the assignment and the second is the role removed. The couple (CA, CR) forms the policy P.

From a set of user-to-role assignments is possible to reach a new set of user-to-role assignments by using the policy. The evolution of the set of user-to-role assignments is represented by a State Transition System.

For applying a rule in CA, it's necessary to have a user who possesses the role indicated to assign the target role (first role of the tuples) and there must exist an user that doesn't possess the target role, doesn't possess roles of the negative preconditions of the rule and possesses at least all the roles of the positive preconditions of the rule.

For applying a rule in CR, it's necessary to have a user who possesses the role required to perform the revocation and there must exist an user that possesses the role revoked.

2.2 The role reachability problem

Most of the problems related to check security goals of ARBAC are reducible to the role reachability problem. In particular, given a target role r_g, the role reachability problem consists in determining if there exists a UR' obtained from the initial UR by applying the rules in P, where r_g is assigned to an user.

Thereby the problem can be formalized as the search for a state in which r_g is assigned in the space of possible UR's reachable from UR. Since the total number of possible user-to-role assignment is $O(2^{|U|*|R|})$, the time algorithmic complexity of this problem it's huge.

3 State Space Tree representation and search

3.1 State Space Tree and search

Since the problem is given by describing P, U, R, the initial UR and the target role r_g, only one query is supposed to be received given P, U and R. Then a first approach doesn't require to use complex matrices or data structures for verifying the reachability, but it's possible to use algorithms like breadth first search or iterative deepening depth-first search.

The search space of sets of user-to-role assignments can be represented as a rooted tree, where a node is a set of user-to-role assignments, the root is the initial UR and the leaves of each node are other set of user-to-role assignments obtained by applying each of the rules in CA and CR, if possible.

For every new assignment or revocation obtained by applying the rules, a new node is created. The list of nodes created by applying rules on a node is inserted on the back of the list of the nodes to visit. To continue the search, a node to visit is extracted from the front of the list of the nodes to visit.

The pseudocode of the search procedure is at figure 1.

Algorithm 1: Generic breadth first search procedure

```
1 Initialize L with the initial state;
2 If L is empty, FAIL, else extract a node n from L;
3 If n is a goal node, SUCCEED;
4 Remove n from L and insert all the children of n of the back of L;
5 Goto 2;
```

3.2 Implementation

The following representation was used to implement the analyser:

- UR as set of (user, role) tuples;
- U as set of users (strings);
- R as set of roles (strings);
- CA set of 4-tuples (r_a, R_p, R_n, r_t), r_a and r_t string, R_p and R_n forzensets of roles (strings);
- CR set of pairs (r_a, r_t), r_a and r_t strings;

3.2.1 Search procedure

The implementation of search procedure in python is:

```
def check_reachability(roles, users, UR, CA, CR, target):
    #start to check if the initial nodes contains a user-to-role
    #assignment with the target role
    nodes_to_check = [Node(UR)]
    nodes_seen = [] #nodes already visited


    while len(nodes_to_check) != 0:
        #select next node to check
        node_visited = nodes_to_check.pop(0)

        #the node is visited, add it to the nodes already seen
        nodes_seen += [node_visited]

        #if the target role is assigned,
        #the problem is solved and the target is reachable
        if node_visited.role_is_present(target):
            return True

        #apply rules to the current UR in order to retrieve
        #the next reachable states
        CA_result_nodes = node_visited.apply_CA_rules(users, CA)
```

```
            CR_result_nodes = node_visited.apply_CR_rules(CR)

            #remove the states already visisted from the list
            #of the new states to visit
            CA_result_nodes_res = CA_result_nodes.copy()
            for node in CA_result_nodes:
                if node in nodes_seen:
                    CA_result_nodes_res.remove(node)
            CA_result_nodes = CA_result_nodes_res

            #remove the states already visited from the list
            #of the new states to visit
            CR_result_nodes_res = CR_result_nodes.copy()
            for node in CR_result_nodes:
                if node in nodes_seen:
                    CR_result_nodes_res.remove(node)
            CR_result_nodes = CR_result_nodes_res

            #increment nodes to search
            nodes_to_check += CA_result_nodes
            nodes_to_check += CR_result_nodes

    return False
```

Initially are initialized the list of nodes to check and the list of the nodes seen. The search has to continue until an UR with the target role assigned is reached or there aren't node to check.

After the initialization, one node of the search space, that corresponds to a set of user-to-role assignment, is explored in order to find if there is an assignment with the target role (see the role_is_present method) and in case True is returned since the problem is solved. The rules are applied in order to produce new states. Since the set of rules remains fixed, the set of rules are parameters of the methods of the node that applies the rules.

The next part consists in checking if states produced are already visited. If this is the case, they are removed from the list of nodes to add to the list of nodes to visit. Finally the list of the nodes to visit is updated.

If the list of nodes to visit remains empty, no set of user-to-role assignments (nodes) contains an assignment of the target role. False is returned.

3.2.2 Node structure

About the Node class that encapsulates the current set of user-to-role assignment, there are few details to be explained.

Node encapsulates a set of user-to-role assignments UR, but when is instantiated also two dictionaries are created, one that maps users that appears in UR to the roles possessed in UR and the other that maps the roles the appears to be assigned in UR to the users that possesses the role.

The two methods apply_CA_rules and apply_CR_rules takes the rules to apply. Also the first method takes the set of the users in the system.

The first method creates a new node to visit for each rule applied that produces a new set of assignments. The applicability of a rule depends on the assignments in UR.

The second method creates a new node to visit for each rule applied, that produces a new set of assignments with one assignment less, if the rule is applicable.

4 Reduce the search space

The algorithm consists simply in breadth first search, so if the search space is huge, the search will take a lot of time. In order to reduce the search space of the role reachability problem, backward slicing and forward slicing techniques can be used. Both are implemented but only backward pruning is shown here, because it seems more useful in the test cases.

4.1 Backward slicing [1]

The objective of the backward slicing is firstly to find the set of roles that are relevant for assigning the target role in order to remove rules and assignments that involves roles not relevant from the problem instance.

Firstly it's necessary to find the set that's the fix point of this set of equations:

- $S_0 = r_g$
- $S_i = S_{i-1} \cup \{R_p \cup R_n \cup r_a | (r_a, R_p, R_n, r_t) \in CA \wedge r_t \in S_{i-1}\}$

The implementation of backward slicing is shown below:

```
#INPUT: description of the problem
#OUTPUT: updated and reduced description of the problem
def backward_slicing(roles, users, UR, CA, CR, goal):

    #S_0
    #in order to assign the target role,
    #the same target role is relevant ;)
    result_roles_set = set([goal])

    #S_(i-1)
    prec_roles_set = set([])

    while prec_roles_set != result_roles_set:
        prec_roles_set = result_roles_set.copy()
        for rule in CA:
            if rule[3] in prec_roles_set:
                result_roles_set.update(rule[1])
                result_roles_set.update(rule[2])
                result_roles_set.add(rule[0])

    #result roles set S* is result_roles_set

    #remove from CA all the rules that assign a role not in S*
    CA_res = CA.copy()
    for rule in CA:
        if rule[3] not in result_roles_set:
            CA_res.remove(rule)

    #remove from CR all the rules that revoke a role not in S*
    CR_res = CR.copy()
    for rule in CR:
        if rule[1] not in result_roles_set:
            CR_res.remove(rule)

    #delete roles not in S*
    roles_res = roles.copy()
    for role in roles:
        if role not in result_roles_set:
```

```
            roles_res.remove(role)

    #delete also user-to-role assignments that involve roles
    #deleted
    UR_res = UR.copy()
    for assignment in UR:
        if assignment[1] not in result_roles_set:
            UR_res.remove(assignment)

    return roles_res, users, UR_res, CA_res, CR_res, goal
```

After having found all the set S^* of roles that are relevant for assigning the target role, all the rules that assigns roles not in S^* are removed, all the rules that revoke roles not in S^* are removed, all the roles not in S^* are removed and all the assignments in UR that involve roles not in $S*$ are removed.

4.2 Advantages of backward pruning

The table 1 shows the reduction of search space after using backward slicing.

policy test	SS dim before slicing	SS dim afterrec calls
1	2^{10*15}	2^{10*7}
2	2^{10*5}	2^{10*7}
3	2^{10*15}	2^{10*6}
4	2^{10*15}	2^{10*9}
5	2^{10*15}	2^{10*7}
6	2^{10*15}	2^{10*7}
7	2^{10*15}	2^{10*8}
8	2^{10*15}	2^{10*7}

Table 1: Reduction of search space by using backward slicing

The reduction of the search space is significant. The reduction helps a lot in finding the satisfaction of the role reachability problem, when the target is reachable.

5 Results on policies to test

In the folder Policies there are the 8 instances of the problem to test in order to solve the challenge. The results are in table 2.

test	result
1	1
2	0
3	1
4	1
5	0
6	1
7	1
8	0

Table 2: Results on role reachability problems

1 indicates that the target role is reachable, otherwise 0 is used.

At the moment, the algorithm works very well for test 1,3,6 and 7, it takes more time for test 4. The actual implementation doesn't allow to finish in a reasonable time the tests 2,3,8, so I think that for these instances the solution is that the target is not reachable, since for the other instances the algorithm is very fast in finding the positive solution.

The flag of the challenge is **10110110**.

References

[1] Stefano Calzavara. *Lecture: Security II - Access Control Verification.* Ca' Foscari University of Venice. URL: `https://secgroup.dais.unive.it/wp-content/uploads/2020/04/arbac.pdf`.