Analyzing RBAC Security Policy of Implementation Using AST

Tuan-Hung Pham, Ninh-Thuan Truong, Viet-Ha Nguyen
College of Technology
Vietnam National University
144 Xuan Thuy, Hanoi, Vietnam
Email: {phamtuanhung, thuantn, hanv}@vnu.edu.vn

***Abstract*—Security policy is a critical property in software applications which require high levels of safety and security. It has to be clearly specified in requirement documents and its implementation must be conformed to the specification. In this paper, we propose an approach to check if the implementation is in accordance with its security policy specification. We use the Abstract Syntax Tree (AST), another manner of expressing the program, to analyze the source code and specify user permission policy in software systems by Role-Based Access Control (RBAC).**

I. Introduction

Access control in software systems is a security mechanism having the ability to give each entity official permissions to do some particular activities. The aim of access control is to preserve the confidentiality and integrity of a system. Among the most widely-used techniques of access control representation, RBAC proves to be very effective because of its potential for access authorization, decreasing administration costs and preventing errors in large systems [1].

If a software system needs to be secure, it is important to be sure that all of the security policy is enforced by strong mechanisms. There are organized methodologies and risk assessment strategies to assure the completeness of security policies and assure that they are completely enforced. In reality of software development, however, RBAC policy and its implementation may be created by different humans and at different times; consequently, it raises the problem of inconformity between the specification and implementation of security policy.

The issues of consistency between RBAC design and implementation have been explored in theoretical approaches of Hansen and Oleshchuk [2] and Ammar et al. [3]. However, these approaches did not check if source code of a program is conformed to its specification. This paper contributes an approach to check the conformity with RBAC policies by analyzing the Abstract Syntax Tree (AST) of the implementation. Our approach considers RBAC implementation not only kernel programs but also user-defined ones which use RBAC as a user permission description. The overall checking approach, shown in Figure 1, consists of the following steps:

- First, we define an abstract language used for the implementation called $\mathbf{L_{RBAC}}$. The language only focuses on RBAC's features and ignores other commands of normal programming languages. Based on $\mathbf{L_{RBAC}}$ syntax, we transform the input source code into a corresponding AST of $\mathbf{L_{RBAC}}$.
- Also, RBAC policy is explicitly written in the form of a RBAC database schema, called $\mathbf{DB}$. Since this step is well-studied [1], we do not discuss in detail how to construct a RBAC database from a RBAC policy for brevity.
- Then, we propose an algorithm to traverse the AST to check whether the implementation is conformed to the design using RBAC database.

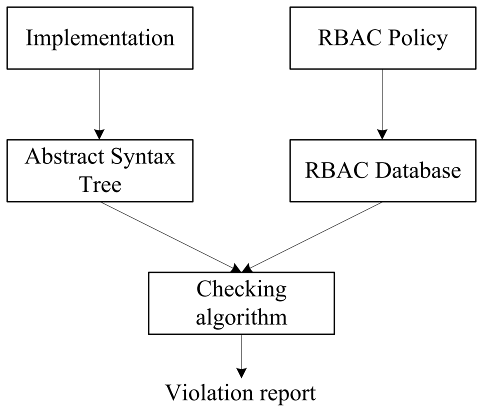

Figure 1. Static Checking RBAC Policy Using AST of Implementation

The rest of the paper is organized as follows. Section II briefly introduces RBAC model. Section III presents $\mathbf{L_{RBAC}}$, a language for RBAC's implementation used in the paper. Section IV presents our approach to check the conformity of RBAC policy and its implementation using AST of source code. Section V shows a case study to demonstrate how our method works. Related work is discussed in Section VI. Section VII concludes the paper and gives some directions for future work.

II. Role-Based Access Control

In a Role-Based Access Control (RBAC) model [1], *users* refer to human beings who interact with a computer system. At the same time, a user may invoke several computer processes. Such processes are called *subjects* in RBAC's notions. Since each process proceeds on behalf of its user, all activities of a subject need to be checked by basing on the privileges of the subject's user, which represents as *roles*. As the center component of a RBAC system, roles are assigned not only to users to define who hold them but also to *permissions* to point out what they can do. Each permission is a pair of an *operation* and an *object*, meaning that the permission allows the object to be accessed by the operation.

The relationship between the components in a core RBAC model is depicted in Figure 2. Throughout the paper, we use u, r, p, op, and o to denote a user, a role, a permission, an operation, and an object, respectively.

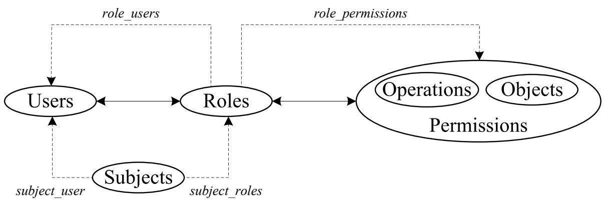

Figure 2. RBAC model

Definition 2.1 (Core RBAC model): A core RBAC model has the following components:

- $\mathcal{U}, \mathcal{R}, \mathcal{P}, \mathcal{O_P}, \mathcal{O}$, and $\mathcal{S}$ (the set of users, roles, permissions, operations, objects, and subjects, respectively).
- $\mathcal{A_{UR}} \subseteq \mathcal{U} \times \mathcal{R}$, a many-to-many assignment from users to roles.
- $\mathcal{A_{RP}} \subseteq \mathcal{R} \times \mathcal{P}$, a many-to-many assignment from roles to permissions.
- $role\_users$, an one-to-many assignment from a role $r \in \mathcal{R}$ to its users, which have connections to it:
$$role\_users(r) = \{u \in \mathcal{U} \mid (u, r) \in \mathcal{A_{UR}}\}$$
- $role\_permissions$, an one-to-many assignment from a role $r \in \mathcal{R}$ to its permissions:
$$role\_permissions(r) = \{p \in \mathcal{P} \mid (r, p) \in \mathcal{A_{RP}}\}$$
- $subject\_user$, an one-to-one assignment from a subject to its corresponding user.
$$subject\_user(s) = u \in \mathcal{U} \text{ that } s \text{ is created by } u$$
- $subject\_roles$, an one-to-many assignment from a subject to its roles.

$$subject\_roles(s) = \{r \in \mathcal{R} \mid subject\_user(s) \in role\_users(r)\}$$

RBAC has two important properties stated in Definition 2.2 and Definition 2.3. *Role authorization* property defines that each role r owned by any subject s must also be owned by the subject's user. More importantly, *Object access authorization* says if user u can access the permission $p = (op, o)$, which we denote by $\lfloor u, (op, o) \rfloor$ with $p \in P, op \in \mathcal{O_P}$ and $o \in \mathcal{O}$, there exists a role r that r is in the list of roles assigned to u and r can access p.

Definition 2.2 (Role authorization):

$$\forall s \in \mathcal{S}, \forall r \in subject\_roles(s) \Rightarrow (subject\_user(s), r) \in \mathcal{A_{UR}}$$

Definition 2.3 (Object access authorization): $\lfloor u, (op, o) \rfloor = \exists p \in \mathcal{P}, \exists r \in \mathcal{R} : p = (op, o), p \in role\_permissions(r) \wedge u \in role\_users(r)$

III. Definition of a simple abstract language - $\mathbf{L_{RBAC}}$

Each implementation of RBAC needs to be written by a modern programming language like C++ or Java. However, the syntax of these languages is too complex in the scope of RBAC analysis since we can only take into account three types of commands: assignments, operations which access to system resources, and control flow statements. As a result, we introduce in this section $\mathbf{L_{RBAC}}$, a simple abstract language used for writing RBAC implementations, to explain the theoretical aspects of our method. Figure 3 describes $\mathbf{L_{RBAC}}$ syntax in BNF-style conventions:

S	::=		Statement
(1)		$v \prec E$	Assignment
(2)	|	$S; S$	Sequencing
(3)	|	$op(o, \{E\})$	Operation
(4)	|	*if* (E) S $[S]$	If-then-else
(5)	|	*while* (E) S	While loop
(6)	|	*for* $([\{S_{(1)}\}]; [E]; [\{S_{(1)}\}])S$	For loop
E	::=		Expression
(1)		v	Variable
(2)	|	c	Constant
(3)	|	$E \oplus E$	Operation

Figure 3. $\mathbf{L_{RBAC}}$ syntax

In Figure 3, [ ] denotes optional items, {} denotes repetitive ones, and | means a choice. Assignment notation $\prec$ in $S_{(1)}$ can be arithmetic assignment (=, +=, −=, *=, /=, %=, ^=), shift assignment (<<=, >>=, >>>=), or boolean assignment (&=, |=). Similarly, operation notation $\oplus$ in $E_{(3)}$ can be arithmetic operation (+, −, *, /, %, ^), shift operation (<<, >>, >>>), or boolean operation (|, &, ==, !=, <, >, <=, >=).

Since we can convert every propositional formula into an equivalent formula written in conjunctive normal form (CNF) by applying the double negative law, the De Morgan's laws, and the distributive law [4], each expression can be rewritten to be a conjunction of clauses. However, in $S_{(4)(5)(6)}$, if an expression contains more than one conjunction of clauses, we can separate these conjunctions by attaching them to additional consecutive if-statements. For example:

$$\mathit{if}\ (E\ \&\ E')\ S\ [S] \overset{convert}{\rightarrow} \mathit{if}\ (E)\ \{\mathit{if}\ (E')\ S\ [S]\}$$
$$\mathit{while}\ (E\ \&\ E')\ S \overset{convert}{\rightarrow} \mathit{while}\ (E)\ \{\mathit{if}\ (E')\ S\}$$
$$\mathit{for}\ ([\{S_{(1)}\}]; E\ \&\ E'; [\{S_{(1)}\}])\ S \overset{convert}{\rightarrow} \mathit{for}\ ([\{S_{(1)}\}]; E; [\{S_{(1)}\}])\ \{\mathit{if}\ (E')\ S\}$$

Therefore, we make a reasonable assumption that expressions in $S_{(4)(5)(6)}$ only consist of disjunctions of clauses.

Because our goal is to check the consistency of implementation with RBAC policies, we need to focus on analyzing conditional policies, which is expressed by three expressions of If-Statement $S_{(4)}$, While-Statement $S_{(5)}$, and For-Statement $S_{(6)}$ in $\mathbf{L_{RBAC}}$. Figure 4 shows their syntax and AST forms.

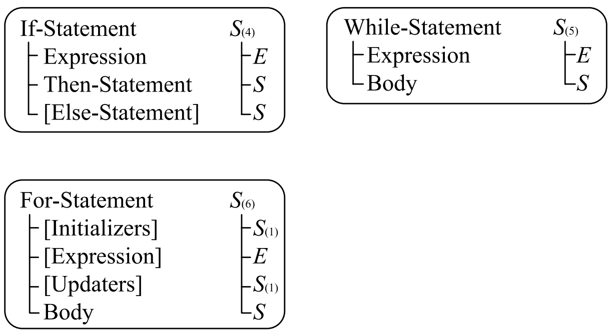

Figure 4. AST forms of If-Statement, While-Statement, and For-Statement

According to our above assumption, the expressions of If-Statement $S_{(4)}$, While-Statement $S_{(5)}$, and For-Statement $S_{(6)}$ have syntax $E = E \mid E$. In AST, it means that if a node represents for a expression $E = E_1 \mid E_2 \mid ... \mid E_k$, the node will have k children and each child corresponds to a sub-expression in the right-hand side.

IV. Checking algorithm

We state in Algorithm 1 our primary contribution, an algorithmic method to validate the conformance between implementation and RBAC policies. For simplicity, our algorithm does not work with multiprocessing systems; in another words, there is at most one process (subject) created by a user at any particular point of time. Before discussing in detail, we introduce some notions and definitions used in our algorithm:

- If a node $n \in \mathbf{AST}$ represents for a $\mathbf{L_{RBAC}}$'s syntax, we denote this relationship by $\sqsubset$.
- Regardless of subjects created by users, each checking process maintains a set *Tuple* of quadruple tuples $\langle u, r, op, o\rangle$. Each tuple means user u performs operation op on object o under role r. Items of each tuple will be shown as $\emptyset$ if we do not know their values, or as a variable if the variable holds the value, or as a constant if the value is explicitly known.
- The values of a tuple t may be changed by assignments. If node n contains assignments, which are $S_{(1)}$ and $E_{(1)(2)(3)}$ in $\mathbf{L_{RBAC}}$, that affects a tuple t, we denote this event by $replace(t, n)$. For example, if we have $t = \langle\emptyset, \emptyset, write, o\rangle$ and node n contains an assignment $o \multimap input.txt$, then $replace(t, n) = \langle\emptyset, \emptyset, write, input.txt\rangle$.
- For each tuple $t = \langle u, r, op, o\rangle$, we need to know whether it violates the Role authorization (Definition 2.2) and Object access authorization (Definition 2.3) of a RBAC policy or not. If t does not, we *accept* it, otherwise t will be rejected. We formalize the *accept* operation by the following formula:

$$accept(\langle u, r, op, o\rangle) = \begin{cases} (op, o) \in \mathcal{P} & \text{if } u = \emptyset \\ (r = \emptyset \vee (u, r) \in \mathcal{A_{UR}}) & \\ \quad \wedge \lfloor u, (op, o)\rfloor & \text{if } u \neq \emptyset \end{cases}$$

Our algorithm takes AST of implementation and RBAC database as input parameters, and then returns the result of checking process. The idea of the algorithm can be described as follows. First, we traverse all nodes of input AST. Each time we find a node n_0 that accesses to system resources during the traversal, we initialize a new *Tuple* and go backwards to update the *Tuple* via assignments and conditional expressions. If we meet an assignment, every tuple $t \in$ *Tuple* will be changed by *replace* operation. In the case of visiting conditional expressions with k disjunctive clauses, we clone k versions of each tuple $t \in$ *Tuple* and then perform *replace* each version with its corresponding disjunctive clause. After finishing going backwards started from a node n_0, if there is any $t \in$ *Tuple* that we have $accept(t) = true$, the node n_0 conforms with $\mathbf{DB}$. The process continues until we successfully find a node n_0 that violates $\mathbf{DB}$ or we reach the root of $\mathbf{AST}$. Our algorithm is terminable since we traverse AST, which is a finite tree.

V. A case study of our algorithm

In this section, we illustrate our approach by a case study of processing invoices after buying items described in [5], [6]. Also having applied their approach to the scenario, Hansen and Oleshchuk [2] showed an example of a RBAC policy in Table I. We also use the policy in Table I to demonstrate how our method proceeds.

Consider a simple user-define code-snippet and its corresponding AST in Figure 5. The code-snippet contains two if-statements. The first if-statement, which conforms to RBAC policy described in Table I, aims to feature both Role

Algorithm 1: Consistency checking

```
Input  : $\mathbf{AST}$: AST of source code
Input  : $\mathbf{DB}$: RBAC database
Output: Conformant or not
Data   : $n, n_0, n'$: $\mathbf{AST}$ nodes

Procedure ASTTraversal($\mathbf{AST}$, $\mathbf{DB}$)
begin
    forall $n \in \mathbf{AST}$ do
        if $n \sqsubset S_{(3)}$ then
            if $\neg$isConformant($n$) then
                return false
    return true
end

Procedure isConformant($n_0$)
begin
    $n \leftarrow n_0$
    $Tuple \leftarrow \{\langle \emptyset, \emptyset, n_0.op, n_0.o \rangle\}$
    repeat
        if $n \sqsubset \{S_{(1)}, E_{(1)(2)(3)}\}$ then
            forall $t \in Tuple$ do
                $t \leftarrow replace(t, n)$
            return
        if $n \sqsubset E$ and $parent(n) \sqsubset S_{(4)(5)(6)}$ then
            forall $t \in Tuple$ do
                forall $n' \in children(n)$ do
                    $Tuple \leftarrow Tuple \cup replace(t, n')$
                $Tuple \leftarrow Tuple \setminus \{t\}$
        $n \leftarrow go\_backwards(n)$
    until $n = \mathbf{AST}.root$
    forall $t \in Tuple$ do
        if $accept(t)$ then
            return true
    return false
end
```

authorization and Object access authorization. The second one violates the RBAC policy specification.

Our algorithm detects three statements that access to system resources at node 7, 8 and 13. Starting from these nodes, we create empty $Tuple$ and go backwards to the root of the input tree. At node 7, we have the following $Tuple$:

$$\begin{aligned} & \{\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle\} \\ \overset{7;6}{\rightsquigarrow} & \{\langle \emptyset, \emptyset, record, invoice \rangle\} \\ \overset{3;2;1}{\rightsquigarrow} & \{\langle Bob, \emptyset, record, invoice \rangle, \langle \emptyset, officer, record, invoice \rangle\} \end{aligned}$$

Since $accept(\langle Bob, \emptyset, record, invoice \rangle) = true$ and $accept(\langle \emptyset, officer, record, invoice \rangle) = true$, we do not find any violation at node 7. Therefore, we continue the checking procedure at node 8 in a similar way:

$$\begin{aligned} & \{\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle\} \\ \overset{8;7;6}{\rightsquigarrow} & \{\langle \emptyset, \emptyset, verify, invoice \rangle\} \\ \overset{3;2;1}{\rightsquigarrow} & \{\langle Bob, \emptyset, verify, invoice \rangle, \langle \emptyset, officer, verify, invoice \rangle\} \end{aligned}$$

Also, we have $accept(\langle Bob, \emptyset, verify, invoice \rangle) = true$ and $accept(\langle \emptyset, officer, verify, invoice \rangle) = true$. The code-snippet still obviously conforms with its RBAC policy. Consequently, we go to node 13 and obtain:

$$\begin{aligned} & \{\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle\} \\ \overset{13;12}{\rightsquigarrow} & \{\langle \emptyset, \emptyset, authorize, invoice \rangle\} \\ \overset{10;9;2;1}{\rightsquigarrow} & \{\langle Elise, \emptyset, authorize, invoice \rangle\} \end{aligned}$$

We conclude that the implementation does not conform to its RBAC policy owing to $accept(\langle Elise, \emptyset, authorize, invoice \rangle) = false$.

Table I
RBAC POLICY OF PROCESSING INVOICES

$\mathcal{U}$	$\mathcal{R}$
Alice	supervisor
Bob	officer
Claire	clerk
Authur	supervisor
Elise	officer

$\mathcal{R}$	$\mathcal{P}$
clerk	p1
officer	p1, p2
supervisor	p2, p3

$\mathcal{P}$	$\mathcal{O}_{\mathcal{P}}$	$\mathcal{O}$
p1	record	invoice
p2	verify	invoice
p3	authorize	invoice

```
if (user == "Bob") or (role == "officer")
  record(invoice)
  verify(invoice)

if user == "Elise"
  authorize(invoice)
```

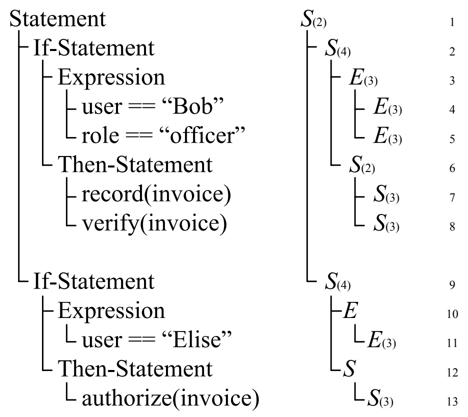

Figure 5. A case study of our algorithm

VI. Related work

Since RBAC was first proposed by Ferraiolo and Kuhn [7] and then improved by other literature [8], [9], [10], it becomes an ongoing topic and attracts many researchers to work on it. Li and Mao [11] introduces a method to design and analyze administrative models of large RBAC systems. For extremely large and distributed systems, Dekker et al. [12] gives a RBAC model for administration. Kwon and Moon [13] use semantic web ontology language (OWL) to formalize and visualize RBAC policy constraints.

Also regarding confidentiality policies, Tschantz and Wing [14] introduce a method to extract conditional policies from source code. Unlike us, this work attempts to reveal hidden policies from a given implementation. Although their WhileIO language is far from reality, their algorithm is novel and effective. We intend to combine their method with ours to attain a stronger one which can not only verify predefined security policies but also discover hidden others that a program obeys.

Conformance testing of RBAC policy and its implementation has been studied in [2] and [3]. Hansen and Oleshchuk [2] propose a solution that uses Linear Temporal Language to express policy and PROMELA language of model-checking system SPIN to represent for implementations. Recently, Ammar et al. [3] also addresses the problem of conformance testing for Access Control Implementation under Test (ACUT) in Temporal Roles-Based Access Control (TRBAC) systems. Not only approaching theoretical aspects of the problem, our work differs from them in that we analyze AST of source code to check the consistency. Our method is therefore suitable for implementing in actual systems, which often provide only source code of implementations for static analyzers to investigate.

VII. Conclusion and Future work

Designing and implementing software systems concerning security policy is always a difficult task, particularly with complex systems. In this paper, we propose an approach to check the compliance between security policy specifications of a system with its implementation. RBAC, a new and efficient language based on role of users, is used to specify user permissions of security policy. We demonstrate our method through a specific case study. Our algorithm, however, currently ignores some complex but practical features of normal programming languages such as method invocations, exceptions, polymorphism, and parallelism. We plan to work on these points and implement this algorithm in the Eclipse platform to use the CDT, a plugin enabling to automatically generate C source code as an AST.

Acknowledgment

This work is (partly) supported by the research project No. QC.09.01 granted by Vietnam National University, Hanoi.

References

[1] D. F. Ferraiolo, R. D. Kuhn, and R. Chandramouli, *Role-Based Access Control, Second Edition*. Norwood, MA, USA: Artech House, Inc., 2007.

[2] F. Hansen and V. Oleshchuk, "Conformance Checking of RBAC Policy and its Implementation," in *Proceedings of the First Information Security Practice and Experience Conference (ISPEC 2005)*. Springer, 2005, pp. 144–155.

[3] M. Ammar, A. Ghafoor, and A. Mathur, "Conformance Testing of Temporal Role-Based Access Control Systems," *IEEE Transactions on Dependable and Secure Computing*, 2008.

[4] H. B. Enderton, *A Mathematical Introduction to Logic, Second Edition*. Academic Press, 2000.

[5] D. D. Clark and D. R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," in *1987 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1987, pp. 184–194.

[6] M. J. Nash and K. R. Poland, "Some conundrums concerning separation of duty," *IEEE Symposium on Research in Security and Privacy*, pp. 201–209, 1990.

[7] D. Ferraiolo and R. Kuhn, "Role-Based Access Control," in *In 15th NIST-NCSC National Computer Security Conference*, 1992, pp. 554–563.

[8] R. S. Sandhu, R. S. S, H. Y, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control: A Multi-Dimensional View," in *Proceedings of the 10th Conference on Computer Security Applications*, 1994, pp. 54–62.

[9] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-Based Access Control Models," *Computer*, vol. 29, no. 2, pp. 38–47, 1996.

[10] R. Sandhu, D. Ferraiolo, and R. Kuhn, "The NIST model for role-based access control: towards a unified standard," in *RBAC '00: Proceedings of the fifth ACM workshop on Role-based access control*. New York, NY, USA: ACM, 2000, pp. 47–63.

[11] N. Li and Z. Mao, "Administration in Role-Based Access Control," in *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*. New York, NY, USA: ACM, 2007, pp. 127–138.

[12] M. Dekker, J. Crampton, and S. Etalle, "RBAC administration in distributed systems," in *SACMAT '08: Proceedings of the 13th ACM symposium on Access control models and technologies*. New York, NY, USA: ACM, 2008, pp. 93–102.

[13] J. Kwon and C.-J. Moon, "Visual modeling and formal specification of constraints of RBAC using semantic web technology," *Knowledge-Based Systems*, vol. 20, no. 4, pp. 350–356, 2007.

[14] M. C. Tschantz and J. M. Wing, "Extracting Conditional Confidentiality Policies," in *SEFM '08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, 2008, pp. 107–116.