Proceedings of the 5th WSEAS Int. Conf. on Software Engineering, Parallel and Distributed Systems, Madrid, Spain, February 15-17, 2006 (pp169-173)

Preventing Information Leakage in C Applications Using RBAC-Based Model

SHIH-CHIEN CHOU
Department of Computer Science and Information Engineering
National Dong Hwa University
1, Section 2, Da-Hsueh Road, Shoufong, Hualien 974
TAIWAN

Abstract - When an application is being executed, users can read the application's output. If sensitive information is managed by an application, information should be prevented from being leaked to unauthorized users during application execution. The prevention can be achieved through information flow control. Since the procedural C language is still in use heavily, we developed a model based on role-based access control (RBAC) for C applications. This paper describes the model.

Key-words: Information security, information flow control, role-based access control (RBAC)

1. Introduction

When an application is being executed, *users* play *roles*. A user playing a role can access the application's output. If an application manages sensitive information, preventing information leakage during application execution is important. *Information leakage* refers to leaking high security level information to low security level users. To prevent information leakage, *information flow control models* can be used [1]. Since the C language is still in use heavily, we developed an information flow control model CRBAC for C based on role-based access control (RBAC) [2]. CRBAC controls both read and write access and offers the following additional features:

a. Preventing indirect information leakage. This leakage refers to leaking information through the third one(s). Generally, this leakage can be prevented using join operating [3].

b. Managing user relationships. User relationships may affect permissions when users play roles in an application. For example, suppose friends can read one another's general information such as age. Also suppose that Mary and John are friends. Then, Mary and John can read each other's general information. When they break friendship, they can no longer read one another's general information. Since user relationships may change during runtime, user permissions should be changed according to user relationship change.

c. Correcting permissions invalidated by user relationship change. User relationship change may affect the prevention of indirect information leakage. For example, suppose Tom and Mary are initially not friends. Then, Tom cannot read Mary's general information. Suppose at this time, the information "ageSet" is derived from Mary's age and others' ages that can be read by Tom. Then, according to the join operation, Tom is not allowed to read "ageSet". If Tom and Mary become friends after a certain time (user relationship change occurs in this case), Tom can read Mary's age this time. In this case, should Tom be allowed to read the information 'ageSet' produced before? The answer should be yes because: (1) Tom can read Mary's age after the user relationship change and (2) "ageSet" is derived from Mary's age and others' ages that can be read by Tom. Since Tom can read all the ages that derived "ageSet" after the user relationship change, Tom should be allowed to read "ageSet" after the change. Allowing Tom to read "ageSet" invalidates the previous join operation because the previous join operations disallowed Tom to read "ageSet". The invalidation requires previous join operations to be corrected.

d. Avoiding improper function call. Different functions in a C application may be in different security levels and therefore should be protected independently.

This paper presents CRBAC and its evaluation.

2. Related Work

RBAC is useful in access control. Nevertheless, since the original design of RBAC is not for information flow control, most features mentioned in section 1 are not offered by the general cases of RBAC. The model in [4] uses

RBAC to control information flows within object-oriented systems. It classifies object methods and derives a flow graph from method invocations. From the graph, non-secure information flows can be identified.

The model in [5] uses access control lists (ACLs) of objects to compute ACLs of executions (which are composed of one or more methods). A message filter is used to filter out possibly non-secure information flows. Flexibility is added by allowing exceptions during or after method execution [6]. More flexibility is added using versions [7].

The decentralized label approach [3] marks the security levels of variables using labels. A label is composed of one or more policies, which should be simultaneously obeyed. A policy in a label is composed of an owner and zero or more readers that are allowed to read the data. Join operation is used to prevent indirect information leakage. Write access is controlled.

CACL [8] is our previous work. It cannot manage user relationships and adjust permissions invalidated by user relationship change.

3. CRBAC

The major problem we encountered in developing CRBAC is "*What should be regarded as roles in a C program*?" A C function is a candidate for a role. Nevertheless, a C function may allow more than one type of users to access and the users may be in different security levels. If a function is regarded as a role, users in different security levels can access information managed by the function, which may result in information leakage. For example, suppose the function *getInfo* gets a user's information. Then, the following two cases of information leakage may happen (suppose a patient is allowed to retrieve his own information only). First, a patient can use the function to retrieve a doctor's information. Second, a patient can use the function to retrieve another patient's information. Although information may be leaked when regarding functions as roles, CRBAC still regards functions as roles. Nevertheless, the following requirements should be fulfilled for a C program:

RleReq 1. Every function in a C program is allowed to access by only one type of users. This solves the problem resulted by the first case mentioned above.

RleReq 2. Constraints should be established for users to access information within a function. For example, accounts and passwords should be given to patients that will access information through the function *getInfo*. This solves the problem resulted by the second case mentioned above.

3.1 Definition

A C application *Cap* embedded with CRBAC is defined below:

Definition 1. *Cap* = (*USR*, *RLE*, *UR*, *PER*, *DSR*, *CSG*, *URA*, *RPA*, *VFC*, *JH*), in which

a. *USR* is the set of users that operate *Cap*.

b. *RLE* is a set of roles. A role *rle* corresponds to a function in *Cap*.

c. *UR* is a set of user relationships. A user relationship $ur \in (2^{USR} - \phi)$.

d. *PER* is a set of permissions. A permission is an access right. CRBAC attaches access rights to variables because variables carry information managed by an application. We implemented a permission as an access control list (ACL). An ACL is composed of a read access control list (RACL) and a write access control list (WACL). The ACL ACL_{var} associated with the variable *var* is defined as "$ACL_{var} = (RACL_{var}, WACL_{var}, UR_{var})$", in which:

(1) $RACL_{var} \in 2^{USRxRLE}$, in which "x" represents Cartesian product. Since multiple users may play the same role, RACL has this definition to distinguish users. A user playing a role in $RACL_{var}$ is allowed to read *var*.

(2) $WACL_{var} \in 2^{USRxRLE}$. A user playing a role in $WACL_{var}$ is allowed to write *var*.

(3) $UR_{var} \in (2^{UR} - \phi)$. $RACL_{var}$ and $WACL_{var}$ are valid in a user relationship *ur* if $ur \in UR_{var}$.

e. *DSR* is a set of data sources (DSOURCE). The DSOURCE of a variable records the functions that wrote the variable's data.

f. *CSG* is a set of CRBAC segments. A C application may have blocks and the same variable names can be used in different blocks. CRBAC offer *CSG* to differentiate variables with the same names.

g. *URA* is a set of user-role assignments, which is defined as "$USR \rightarrow 2^{RLE}$".

h. *RPA* is a set of role-permission assignments, which is defined as "$RLE \rightarrow 2^{PER}$".

i. *VFC* is a set of valid function calls. If the function *fn1* is allowed to invoke *fn2*, the element (*fn1*, *fn2*) belongs to *VFC*.

j. *JH* records join histories. It facilitates redoing join operations to correct permissions (see section 3.3 for details).

3.2 Information flow security in CRBAC

An information flow occurs when the result of a computation is assigned to a variable. To ensure secure information flows, both direct and indirect information flows should be secure. Direct information flows include *those among functions* and *those within functions*. Those among functions are induced by function calls. If the function *fn1* invokes *fn2*, a *vfc* "(*fn1*, *fn2*)" should exist. Suppose the invocation is allowed. Then, the ACLs and DSOURCEs of arguments should be copied to the corresponding parameters. This copying is necessary because a parameter receiving an argument inherits the security level of the argument.

When the value derived from variables in the set "$\{var_i \mid var_i$ is a variable and i is between 1 and $n\}$" is assigned to the variable *d_var*, the information flow induced by the derivation is considered secure only when both the following two *secure flow conditions* are true. To define the conditions, we let:

(a) The ACL and DSOURCE of *d_var* be respectively "($RACL_{d\_var}$, $WACL_{d\_var}$, UR_{d_var})" and "$DSOURCE_{d\_var}$".

(b) The ACL and DSOURCE of var_i be respectively "($RACL_{\text{var}_i}$, $WACL_{\text{var}_i}$, UR_{var_i})" and " $DSOURCE_{\text{var}_i}$ ".

First secure flow condition: $\exists\ UR_{sub} \subseteq (\bigcap_{i=1}^{n} UR_{\text{var}_i} \cap UR_{d\_var})$ so that $RACL_{d\_var} \subseteq \bigcap_{i=1}^{n} RACL_{\text{var}_i}$

Second secure flow condition: $\exists\ UR_{sub} \subseteq (\bigcap_{i=1}^{n} UR_{\text{var}_i} \cap UR_{d\_var})$ so that $WACL_{d\_var} \supseteq \bigcup_{i=1}^{n} DSOURCE_{\text{var}_i}$

The first condition controls read access. The condition "$RACL_{d\_var} \subseteq \bigcap_{i=1}^{n} RACL_{\text{var}_i}$ " requires that *d_var* should be the same restricted as or more restricted than the variables in the set "$\{var_i \mid var_i$ is a variable and i is between 1 and $n\}$". Since RACLs and WACLs are valid under certain user relationships, the ACL of *d_var* and those of the variables in the variable set mentioned above should be valid in certain user relationship(s). This results in the requirement "$\exists\ UR_{sub} \subseteq (\bigcap_{i=1}^{n} UR_{\text{var}_i} \cap UR_{d\_var})$". The second secure flow condition controls write access. It requires that the data sources of the variables deriving the value assigned to *d_var* should be within $WACL_{d\_var}$ because the data derived from the variables are written to *d_var*.

After assigning the derived value to *d_var*, the ACL of *d_var* should be changed by the join operation to prevent indirect information leakage. We use the symbol "$\oplus$" to represent the operation and change ACL_{d_var} to $\bigoplus_{i=1}^{n} ACL_{\text{var}_i}$.

Definition 2. $\bigoplus_{i=1}^{n} ACL_{\text{var}_i} = (\bigcap_{i=1}^{n} RACL_{\text{var}_i} , \bigcup_{i=1}^{n} WACL_{\text{var}_i} , \bigcap_{i=1}^{n} UR_{\text{var}_i})$

In addition to joining ACLs, the DSOURCE of *d_var* will be adjusted to $\bigcup_{i=1}^{n} DSOURCE_{\text{var}_i}$.

3.3 Redoing join operations

ACLs invalidated by user relationship change should be corrected by redoing join operations. Suppose a variable *d_var* is derived from the variables in the set VAR_1. Since a variable may be derived from other variables, we suppose that the variables deriving the variables in VAR_1 constitute the set VAR_2, the variables deriving the variable in VAR_2 constitute the set VAR_3, and so on. The derivation process results in ripple effects. The effects end when $VAR_m \subseteq VAR_k$, in which $k < m$. We let *UVAR* be the set "$\cup_{i=1}^{n} VAR_i \cup d\_var$" and suppose that the earliest time the variable var_i being a derived variable is t_{var_i}, in which $var_i \in UVAR$. From t_{var_i} down to the current time, every join operation in which var_i is a derived variable should be redone. When redoing join operations, the current user relationships should be used as a reference because ACLs should be correct under the current user relationships. The redoing should use the component *JH*.

Definition 3. An element *jh* of *JH* in Definition 1 is defined below:

jh = (t, *d_var*, $\{(var, ACL_{var}) \mid var$ is a variable

that derives *d_var* and ACL_{var} is the ACL of *var* at the time *t*}, *tag*), in which
a. *t* is the time that a join operation is done.
b. *d_var* is the derived variable.
c. {(*var*, ACL_{var})} is the set of variable and their ACLs that derive *d_var*.
d. If *tag* is set, *t* is the earliest time that *d_var* is a derived variable.

With the above description, the redoing of join operations is achieved using Algorithm 1.

Algorithm 1. Join operation redoing algorithm

1. Input data:
1.1. VAR_1 = {*var* | *var* is a variable to derive *d_var*}
1.2. *d_var*: the variable derived from the variable in the set VAR_1
2. Algorithm:
2.1. Backtrack *JH* to identify VAR_2 through VAR_n following the procedure described in the first paragraph of this section.
2.2. Let *UVAR* be the set "$\cup_{i=1}^{n} VAR_i \cup d\_var$".
2.3. For each $var_i \in UVAR$, do
2.3.1. Backtrack *JH* to identify the earliest time t_{var_i} that var_i is a derived variable. The tags in *JH* (see Definition 3) facilitate the identification.
2.3.2. From the time t_{var_i} down to the current time, mark the join operations in *JH* in which var_i is a derived variable.
2.4. End do
2.5. Redo the marked join operations from the earliest time a join operation is marked down to the current time.

4. Features

Controlling both read and write access is achieved by the secure flow conditions. Below we prove that CRBAC offers other features.

Lemma 1: CRBAC prevents indirect information leakage.

Proof: Indirect information leakage results when a role *fn2* leaks to *fn3* the information retrieved from *fn1*, in which *fn2* is allowed to read the information of *fn1* whereas *fn3* not. To prove that indirect information leakage is avoided, we let *var1* be a variable in *fn1* that can be read by the roles in *var1*'s RACL $RACL_{var1}$. According to the above assumption, *fn2* is in $RACL_{var1}$ but *fn3* not. We also let *var2* be a variable in *fn2* whose value is derived from *var1*. After the derivation, *var2*'s RACL $RACL_{var2}$ is modified by the join operation (see Definition 2). Suppose indirect information leakage exists among *fn1*, *fn2*, and *fn3*. Without loss of generality, we assume that *fn3* can read *var2* after *var2* is derived from *var1*. With this assumption, *fn3* is within $RACL_{var2}$. However, according to the join operation, $RACL_{var2}$ is the intersection of $RACL_{var1}$ and other RACLs after *var2* is derived from *var1*. Since *fn3* is not in $RACL_{var1}$, *fn3* is not in $RACL_{var2}$. #

Lemma 2. CRBAC manages user relationships.

Proof: To prove that CRBAC manages user relationships, we have to prove that: (a) CRBAC changes role permissions when user relationship changes and (b) CRBAC corrects permissions invalidated by user relationship change. The proof for item *b* is in Lemma 3. Below we prove that CRBAC changes role permissions when user relationship changes.

The following cases can be regarded as user relationship change: (a) a system possesses different user relationships at different time, say *t1* and *t2* and (b) a system possesses the same user relationships at *t1* and *t2* but at least one user relationship at different time possesses different roles.

Case a: Let $UR_{t2} = UR_{t1} \cup \{ur\}$, in which *ur* is a user relationship and $ur \notin UR_{t1}$. In this case, $PER_{t1} \neq PER_{t2}$, in which PER_{t1} and PER_{t2} are respectively the permission sets of the executing system at *t1* and *t2*. $PER_{t1} \neq PER_{t2}$ because $PER_{ur} \subseteq PER_{t2}$ but $PER_{ur} \not\subset PER_{t1}$, in which PER_{ur} consists of permissions of users in the user relationship *ur*.

Case b: Assume that: (1) ur_{t1} and ur_{t2} are the same user relationship containing different users at time *t1* and *t2* and (2) $ur_{t2} = ur_{t1} \cup \{u1\}$, in which *u1* is a user and $u1 \notin ur_{t1}$. In this case, $PER_{t1} \neq PER_{t2}$ because *u1* is in the user relationship at time *t2* but not at *t1*, which makes PER_{t2} to possess more permissions than PER_{t1}. The extra permissions of PER_{t2} are offered by *u1*. #

Lemma 3. CRBAC corrects permissions invalidated by user relationship change (i.e., Algorithm 1 is correct).

Proof. Suppose only one variable *d_var* is within *UVAR* in line 2.2 of Algorithm 1. Then, *d_var* never plays the role of a derived variable. In this case, *d_var*'s ACL is unchanged during application execution. An unchanged ACL is correct because an ACL may be invalidated only when user relationships change and the ACL is changed by join operation.

Suppose Algorithm 1 is correct when there are (*k*-*1*) elements in the set *UVAR*, in which

UVAR = {var_i | var_i is a variable, *i* is between 1 and (*k*-*1*), and (*k*-*1*) > 1}. The correctness of Algorithm 1 under the above assumption implies that, before the variable *d_var* is derived using the variables in *UVAR*, Algorithm 1 corrects ACLs associated with the variables in *UVAR* by referring to the current user relationships. Let's add a variable var_k to the original *UVAR* (we let *NewUVAR* = *UVAR* ∪ {var_k}). According to the assumption in the previous paragraph, the ACLs associated with the variables in *NewUVAR* excluding var_k are correct after join redoing. Moreover, the ACL associated with var_k is correct because lines 2.3 through 2.5 of Algorithm 1 corrects the ACL of var_k. #

Lemma 4. CRBAC avoids improper function calls.

Proof. An improper function call from the function *fn1* to *fn2* may occur when: (a) *fn1* is not allowed to invoke *fn2* and (b) *fn1* passes improper arguments to *fn2*. If condition *a* is true, the *VFC* defined in Definition 1 will block the function call. If condition *b* is true, the two secure flow conditions will block the statement. #

5. Evaluation

A C application embedded with CRBAC model should first be processed by the CRBAC preprocessor. The output of the preprocessor is a pure C program. The C program generated by the CRBAC preprocessor is composed of the original program and a *security monitor* to check information flow security during runtime.

We trained students to use CRBAC. We then required them to program a simplified library management system and a simplified inventory management system of a supermarket. During the programming, we required the students to inject user relationship changes and non-secure statements (non-secure statements are those that cause non-secure information flows). We then required the students to run their programs. The experiments showed that every injected non-secure statement was identified.

6. Conclusion

Information flow control within an application during its execution prevents information leakage. Since the C language is still in use heavily, we developed an RBAC-based model CRBAC to control information flows within C applications. It offers the following features:

a. Controlling both read and write accesses using the two secure flow conditions.
b. Preventing indirect information leakage using join operation.
c. Managing user relationships. CRBAC uses user relationships to limit permissions so that changing user relationships will change user permissions.
d. Correcting permissions invalidated by user relationship change. CRBAC records join histories and redoes join operations to correct permissions using Algorithm 1.
e. Avoiding improper function calls by recording valid calls.

References

[1] A. Sabelfeld, and A. C. Myers, "Language-Based Information-Flow Security", *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5-19, 2003

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

[3] A. Myers and B. Liskov, "Protecting Privacy using the Decentralized Label Model", *ACM Trans. Software Eng. Methodology*, vol. 9, no. 4, pp. 410-442, 2000.

[4] K. Izaki, K. Tanaka, and M. Takizawa, "infrmation Flow Control in Role-Based Model for Distributed Objects", *Proc. 8'th International Conf. Parallel and Distributed Systems*, pp. 363-370, 2001.

[5] P. Samarati, E. Bertino, A. Ciampichetti, and S. Jajodia, "infrmation Flow Control in Object-Oriented Systems", *IEEE Trans. Knowledge Data Eng.*, vol. 9, no. 4, pp.524-538, Jul./Aug. 1997.

[6] E. Bertino, Sabrina de Capitani di Vimercati, E. Ferrari, and P. Samarati, "Exception-based Information Flow Control in Object-Oriented Systems", *ACM Trans. Information System Security*, vol. 1, no. 1, pp. 26-65, 1998.

[7] A. Maamir and A. Fellah, "Adding Flexibility in Information Flow Control for Object-Oriented Systems Using Versions", *International Journal of Software Engineering and Knowledge Engineering*, vol. 13, no. 3, pp. 313-326, 2003.

[8] Shih-Chien Chou and Chin-Yi Chang, "An Information Flow Control Model for C Applications Based on Access Control Lists", *Journal of Systems and Software*, vol. 78, no. 1, pp. 84-100, Oct. 2005.