Discretionary security in class hierarchy based event management systems

Pavel Štros

stros@datasys.com

Department of system and network management
Datasys s.r.o.
Prague, Czech Republic

Abstract

I present discretionary access control (DAC) model that is targeted to drive control to "event format definition records" within a class hierarchy based event management system (EMS). My research reveals the specifics of EMS with respect to discretionary protection, defines necessary information granularity and provides algorithms that are suggested for calculation of the access control decision. I present access control algorithms for read and write operations. I show how a security policy may be defined by labeling the source tree with the supported authorization attributes.

Keywords: event, class, access policy, security model, and discretionary.

1 Introduction

Event management is one of the central questions in the management of distributed systems. Event management pertains to detection, isolation, classification, filtration, correlation and presentation of events occurring in a network consisting of devices, operating systems and applications. All of the functional elements of EMS have to deal with large volumes of events [1]. Event management aids in the:

- Reduction in alarm events reported to a management station.
- Quick isolation and possible correction of fault.
- Detection of various composite events or event patterns that are a set of interrelated events.

The central idea of the DAC is that the owner of an object, who is usually its creator, has discretionary authority over that who else can access that object. The DAC, in other words, involves owner-based administration of access rights.

1.1 Research outline

My research is derived from the research related to the access control for generic tree structures [3]. I briefly define the concept of subject and object and introduce privileges that are supported by my security model. I then show how a security policy may be defined by labeling the event class structure with some authorization attributes. I present some access control algorithms that I use to enforce the security policy.

2 Discretionary security in the context of EMS

2.1 Terminology and assumptions

Event is a particular fault or incident within the computing environment that occurs on an object. An event usually represents either a change in status or a threshold violation. Every time the status of a managed object changes in any way, an event occurs. If this event is important enough to drive attention, or if it needs to be correlated with events from other sources and therefore cannot be fully processed at its local site, then the event should be forwarded to a central event server. The description of an event is referred to as an *event message*. Event messages are the central unit of information within the EMS. Event messages are structured chains and

are typically provided in the form of *attributes*, which are ”name=value” pairs. The term ”event” or “*event instance*” is often used in place of the more appropriate term ”event message” within this article.

My research assumes *event class* hierarchy. Event classes determine the attributes and information that may constitute the event message. Each event is identified by a class name as a result of classification. Each class is the parent of zero or more child classes. Each class has one and only one parent except one fundamental class that has no parent and that is called the root class. Trees are often called inverted trees because they are normally drawn with the root at the top. Classes without a child are called leaf classes. Multiple inheritances are not allowed within class definitions.

In my model, the subject is a single user. It is not the purpose of this paper to give detailed information on how subjects are organized. Let me mention that I could easily extend my model to take into account the concepts of groups or roles. In my model, an object is an event class. Each class is associated with its own *access control list*. The class is the smallest granule of information that I can protect. I define the security policy by labeling the classes of the source tree with some authorization attributes. An authorization attribute refers to a unique user and a unique privilege.

2.2 Authorization attributes

I define authorization attributes as follows:

The authorization attribute is a pair <subject, privilege>, where

- subject identifies a user
- privilege takes its value from the set: {Read, Write}

Authorization attributes are associated with event classes. An authorization attribute may be associated with several classes and a class may be associated with several authorization attributes. The association of an authorization attribute with an object makes *permission*. The set of authorization attributes associated with a class makes an access control list.

Let C be a class. Let <S,P> be an authorization attribute. If class C is associated with <S,P> then S is granted the privilege P on class C.

- if P = Read then S is granted the right to see (i.e. evaluate) C.
- if P = Write[1] then S is granted the right to add a new sub tree of classes to C, S is granted the right to delete the sub tree whose root is C and S is granted the right to update direct subclasses of class C (i.e. replace the format description of the subclasses). In addition, S is granted the right to update all the parameters of class C with the exception of the format string of class C and the link of class C to its parent class.

There is no constraint that should be respected when labeling the tree with authorization attributes. However, I add three rules to every security policy that may “conflict” with permissions that are defined by the labeling process. The priority of these rules is always higher than the priority of other rules:

- **Integrity Rule 1**: The view of the source tree a subject is permitted to see has to be a pruned version of the original source tree and includes the root class. (I want users to be provided with consistent views of the format data definitions.)
- **Integrity Rule 2**: If a subject is permitted to “write a class”, it is permitted to see all direct subclasses of that class (I want administrative users to be provided with consistent configuration information, i.e. the user should be able to evaluate completely the configuration information from his area of competence).

[1] With respect to the event parsing algorithm there is little motivation to introduce separate authorization attributes Insert, Update and Delete.

- **Integrity Rule 3**: A subject is forbidden to perform a write operation on a class that he is not permitted to see. (Even if it has been granted the corresponding write privilege on this class by the labeling process, i.e. blind writes are forbidden.)

2.3 View Control Algorithm

In this section I present the algorithm that computes the view of the source tree a given user is permitted to see with respect to the integrity constraints presented in previous section. This algorithm traverses the tree in pre-order. After the algorithm finishes, list L contains the pre-order list of the classes that belong to the view.

Let S be the subject for which the view has to be computed, let L, Q, T be empty lists of classes.

- Insert the root class into Q
- While Q is not empty Do
  - C is the first class of list Q
  - If class C is associated with the attribute <S,Write>
    - Append class C to L
    - Replace class C in Q with its subclasses
    - If C is the first class of list T
      - Replace class C in T with its subclasses
    - Else
      - Append all the subclasses of C to list T
  - Else If C is associated with the attribute <S,Read> then
    - Append class C to L
    - Replace class C in Q with its subclasses
    - If C is the first class of list T
      - Remove C from T
  - Else If C is the first class of list T
    - Append class C to L
    - Replace class C in Q with its subclasses[2]
    - Remove C from T
  - Else
    - Remove C from Q

2.4 Write Control Algorithm

In this section, the second algorithm is presented, that is able to determine whether a given user is allowed to perform a Write operation on a particular class. The algorithm is simple, because the View Control Algorithm does the hard work instead of it.

Let S be the user performing the operation which requires the Write privilege on class C.

- Use the View Control Algorithm to compute the view of the source tree for user S
- If class C does not appear in the view then
  - S is forbidden to perform the operation[3]
- Else
  - If class C is associated with the attribute <S,Write> then
    - S is permitted to perform the operation

[2] This part of the algorithm might be omitted in order to perform stricter calculation of the tree. Read-write authorization attribute crossover is allowed with this line in place.

[3] The answer subject (user) is provided with, should not be something like “access denied” because such an answer would reveal the existence of class C to subject S. The answer should rather be “class unknown” since class C does not belong to the view subject S is permitted to see.

- Else
  - S is forbidden to perform the operation

Notice, that the conflict related to possible indirect deletions of some data, that the user may not be aware of, is effectively solved by the fact the user is able to fully evaluate event classification process due to the second integrity rule that is applied on format definitions data structure (the tree of event classes).

3 Class hierarchy discretionary access control example

Consider event class hierarchy that is depicted in the following diagram. Notice the explicit authorization attributes per subjects S1, S2 and S3 (the red text in parentheses).

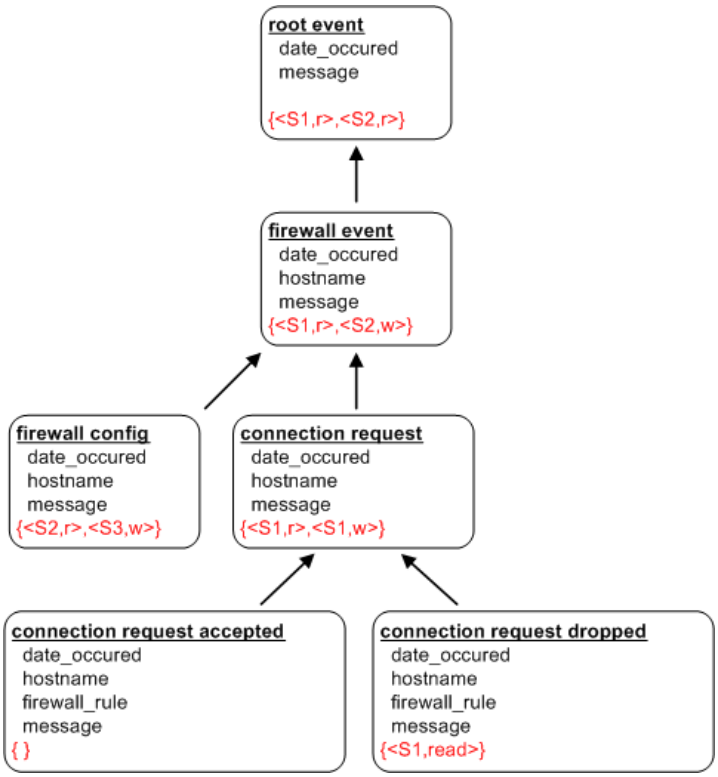

Figure 1: Class hierarchy ACL example

The original purpose of this specific assignment of authorization attributes was to permit access to information about connections to subject S1 (especially to information about dropped connection requests) and to permit access to information about firewall configuration changes to subject S2.

These principles might have a real world background, because access to the information about dropped connection attempts might be allowed to the security unprivileged network or application administrators to enable them tracing of network related issues. However, the information about permitted connections, that might indirectly reveal the access configuration rules of the firewall, and enables for creation of network communication reports, should not be available them. The explicit information about firewall configuration changes (class “firewall config”) might be available to the responsible security administrator.

I added some “Write” authorization attributes into the sample class hierarchy to demonstrate their major influence at the final calculation of the class hierarchy view of a subject.

Sample code containing definitions of classes the above event class hierarchy may be loaded from, is presented in [4].

When the View Control Algorithm is applied to the sample class hierarchy considering authorization attributes of subject S1, the content of the lists L, Q and T evolves in the following way:

 # START: all the lists are empty

L: {}, Q: {}, T: {}

 # before we enter the first iteration

L: {}, Q: {root_event}, T{}

 # after the 1st iteration

L: {root_event}, Q: {firewall_event}, T: {}

 # after the 2nd iteration

L: {root_event, firewall_event}, Q: {firewall_config, connection_request}, T: {}

 # after the 3rd iteration

L: {root_event, firewall_event, connection_request},

Q: {connection_request_accepted, connection_request_dropped},

T: {connection_request_accepted, connection_request_dropped}

 # after the 4th iteration

L: {root_event, firewall_event, connection_request, connection_request_accepted},

Q: {connection_request_dropped},

T: {connection_request_dropped}

 # after the 5th iteration

L: {root_event, firewall_event, connection_request, connection_request_accepted, connection_request_dropped},

Q: {}, T: {}

 # END: there is no class in list Q

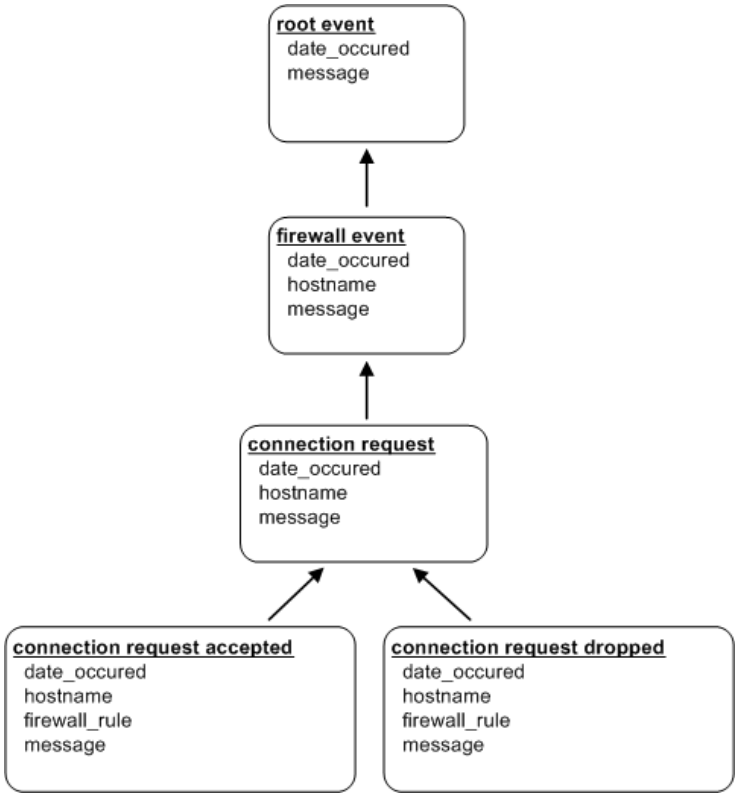

Figure 2: Class hierarchy view per subject S1

When the View Control Algorithm is applied to the sample class hierarchy considering authorization attributes of subject S2, the content of the lists L, Q and T evolves in the following way:

 # START: all the lists are empty

L: {}, Q: {}, T: {}

 # before we enter the first iteration

L: {}, Q: {root_event}, T{}

 # after the 1st iteration

L: {root_event}, Q: {firewall_event}, T: {}

 # after the 2nd iteration

L: {root_event, firewall_event}, Q: {firewall_config, connection_request},

T: {firewall_config, connection_request}

 # after the 3rd iteration

L: {root_event, firewall_event, firewall_config},

Q: {connection_request},

T: {connection_request}

 # after the 4th iteration

L: {root_event, firewall_event, firewall_config, connection_request},

Q: {connection_request_accepted, connection_request_dropped},

T: {}

 # after the 5th iteration

 # notice, that should there be a authorization attribute associated to class

 # connection_request_accepted or connection_request_dropped,

 # read-write crossover occured

L: {root_event, firewall_event, firewall_config, connection_request},

Q: {connection_request_dropped},

T: {}

 # after the 6th iteration

L: {root_event, firewall_event, firewall_config, connection_request},

Q: {}, T: {}

 # END: there is no class in list Q

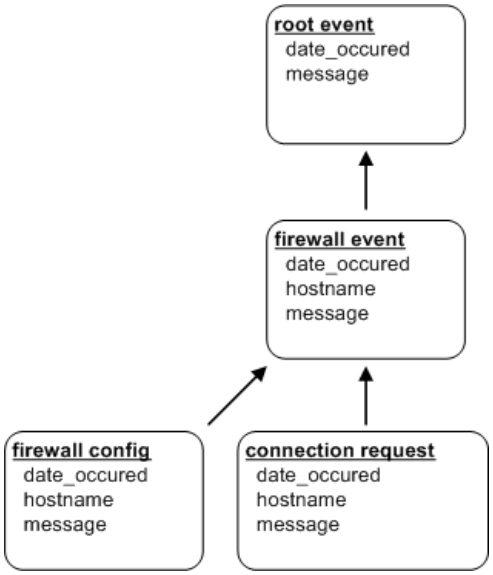

Figure 3: Class hierarchy view per subject S2

When the View Control Algorithm is applied to the sample class hierarchy considering authorization attributes of subject S3, the content of the lists L, Q and T evolves in the following way:

 # START: all the lists are empty

L: {}, Q: {}, T: {}

 # before we enter the first iteration

L: {}, Q: {root_event}, T{}

 # after the 1st iteration

L: {}, Q: {}, T: {}

 # END: there is no class in list Q

There is a consistent class hierarchy view for both subject S1 and subject S2, subject S3 does not see any event classes.

As a result of the Write Control Algorithm, subject S1 is allowed to modify subclasses of class connection_request and subject S2 is allowed to modify classes firewall_config and connection_request. S1 and S2 see all direct subclasses of the class they have "Write" privilege to, i.e. they are able to completely evaluate the configuration information at the sub tree level they are permitted to manage. As a result of the Write Control Algorithm subject S3 is forbidden to perform a write operation on the class firewall_config even through there is the required authorization attribute. S3 is not permitted to see that class and blind writes are forbidden.

4 Conclusions

I formulated proposals for a sound discretionary access control model for event class structures. Security policy for a particular system may be defined by labeling the event class structure with appropriate authorization attributes. Three implicit integrity rules constitute crucial part of the model and enable for loose labeling of the event class tree with authorization attributes. 'View Control Algorithm' that computes the view of the source tree a given user is permitted to see with respect to the implicit integrity constraints is presented. 'Write Control Algorithm' is presented that is able to determine whether a given user is allowed to perform a Write operation on the particular class.

References

[1] Hasan, M. Z., The management of data, events, and information presentation for network management, *Thesis of the University of Waterloo*, Ontario, Canada, 1996.

[2] Schwiderski S., Monitoring the behavior of distributed systems. *Dissertation of University of Cambridge*, USA, 1996

[3] Gabillon, A., Munier M., Bascou J., Gallon L. Bruno E.: An Access Control Model for Tree Data Structures, *Publication of Université de Pau & Université de Toulon*, France, 2002.

[4] Štros, P., Security in event management systems. *Dissertation of University of Defence*, ČR, 2005