USAGE CONTROL POLICY ENFORCEMENT IN OPENOFFICE.ORG AND INFORMATION FLOW

C. Schaefer[1], T. Walter[1], A. Pretschner[2], M. Harvan[3]

[1]DOCOMO Euro-Labs
Germany
[2]Fraunhofer IESE
Germany
[3]ETH Zurich
Information Security
Switzerland

[1][schaefer,walter]@docomolab-euro.com,
[2]Alexander.Pretschner@iese.fraunhofer.de, [3]mharvan@inf.ethz.ch

ABSTRACT

Usage control is a generalisation of access control addressing how data is to be handled after it has been released. To control the data handling enforcement mechanisms have to be in place where the data is being used. These enforcement mechanisms can be implemented on different layers of the system. One way to do the enforcement is on the application layer. This paper describes how usage control policies can be enforced in OpenOffice.org using the component technology UNO (Universal Network Objects) provided by OpenOffice.org. The drawbacks and sketches how to overcome these are also identified.

KEY WORDS

Information flow, usage control, policy enforcement

USAGE CONTROL POLICY ENFORCEMENT IN OPENOFFICE.ORG AND INFORMATION FLOW

1 INTRODUCTION

In companies it is common to have a policy describing how sensitive information is to be used (sometimes referred to as *managed information*). Sensitive information can be construction plans for machines, documentation of a product or anything else containing secret information of a company. Policies to protect company assets can be seen as *usage control policies* as they go beyond standard access control policies. For example, it can be specified that no user besides the author can edit a patent application (access control policy), and that it must be stored in the company intranet only. Further, copying and pasting information is allowed within the document but not into another document (all the previous are usage control policies).

Theoretical work exists (see for example [4]) on how to specify these kind of policies. With respect to the above mentioned usage control policies we address three problems in this paper. First, it is not clear how to map *high level policies* to *low level policies* that can be enforced. Second, it is not obvious how to map *actions on data*, i.e. do not copy confidential data into another document, to *actions of processes on data*, i.e. using an office application and perform copy/paste. Third, we need to determine how to *control actions*.

The solution we employ is to define an information flow model for OpenOffice.org (OO). This allows us to define where control is applicable in OO (third of the above mentioned problems) and how high level policies are mapped to low level policies understood by OO (first problem). Obviously, the OO instance is the process that manipulates the data and it is this process to be controlled (second problem).

To verify our approach we have developed an architecture and prototypical implementation using the Universal Network Objects (UNO) component technology from OpenOffice.org. The implemented framework uses a description of what has to be enforced and intercepts all actions according to this description. Actions (like Print a document) performed by OO can either be allowed or forbidden; others (like Save as) can also be modified, e.g. allowing only a specific file format and a specific directory where the document can be saved. We show which requirements can be enforced by our architecture,

and provide examples of what we have implemented. Additionally we show the limitations of our approach.

Our contributions are the design and implementation of usage control enforcement in OpenOffice.org as well as the definition of an information flow model for office like applications. The information flow model is the basis for the development of the enforcement model. The enforcement model is capable to control all the information going into and out of OO. These are the first steps to an enforcement architecture for usage control policies.

The remainder of the paper is structured as follows. Section 2 introduces the related work and describes how UNO is working. A description of the information flow model is shown in Section 3. This is followed in Section 4 by a description of the architecture of the OpenOffice.org controller enforcing the usage control policies. Section 5 lists some limitations of this approach and is followed by the summary in section 6.

2 RELATED WORK

Usage control [10, 8] is a topic that has received some attention in the research community recently. It is a generalisation of access control and deals with how data is to be handled once it has been released to a third party. The existing work has laid theoretical foundations for usage control by for example specifying usage control languages. Digital rights mechanisms (DRM) are part of usage control and can be used as enforcement mechanisms.

2.1 Usage Control Enforcement

A classification of enforcement mechanisms which are needed for usage control is introduced in [11, 5]. *Inhibit* specifies actions that are forbidden like it is forbidden to print a document. *Finite delay* specifies the class where an action is delayed until some conditions are met, e.g. approval by management before a patent application is filed. It might occur in a usage control policy that a disclaimer "Printed by" has to be added to a document. This forms the third class *modify* as some additional information before the actual printing is added. The sending of a notification to someone else that the data has been modified falls in the *execute action* category.

Another paper [11] looked into existing DRM mechanisms and analysed which enforcement classes are supported by those mechanisms. The result was that all DRM mechanisms supported the inhibit class but only a few

mechanisms supported more classes. But no analysed mechanism supported all classes.

DRM systems, like the one used in Apple's iTunes, are currently focused on protection of multimedia content like audio and video. The mechanisms mainly provide access control and control the distribution of the content by not allowing to copy the content to other devices or only to a restricted number of other devices. Approaches like Sealed Media [6] or Microsoft's RMS [1] also exist which focus on the protection of documents. These approaches provide mechanisms for access control and the distribution of documents but again not all classes mentioned above like for example *modify* are supported. Our approach goes beyond the existing ones providing support for all enforcement classes.

2.2 OpenOffice.org and UNO

OpenOffice.org [3] is an open-source office software suite for word processing, spreadsheets, presentations, graphics and databases. It stores all data in an international open standard format (ODF - Open Document Format) [7].

OO provides an interface called UNO [2] with which OO can be controlled by external programs. The external program can either be integrated into OpenOffice as an extension (a plugin mechanism) or it can be a program running independently.

2.2.1 Dispatch Framework

OO has a so called *Dispatch Framework* [2] which defines interfaces for a generic communication between an office component (i.e. some functionality provided by OO) and a user interface. This communication process handles requests for command executions and gives information about the various attributes of an office component. The user interface sends messages to the office component and receives information from the office component.

The framework maintains a list of DispatchProviders. These DispatchProviders contain information about what to do when some execution command is coming from the user interface. When a command is issued the first DispatchProvider in the list suitable for this command performs all necessary actions. A DispatchProvider can also perform part of the work for the command and then pass on the control to another DispatchProvider which is then finishing the task.

2.2.2 Commands

Commands are the basic entities in the dispatch framework and are executed by the dispatch framework providing all the functionality necessary for an office suite. Save for example is a command that stores a modified document using the same name it was opened with. Commands are accessible through the menu or the button bar. The execution of a command might trigger an event (see Section 2.2.3) like OnPrint. Using UNO external programs can execute commands and for example print a document without the involvement of the user but adding the above mentioned disclaimer "Printed by".

2.2.3 Events

UNO-Events [2] consist either of a pair of events describing the start of a command (e.g. OnLoad) and the end of a command (e.g. OnLoadDone) or they consist of one event describing either the beginning (e.g. OnPrint) or the end (e.g. OnNew) of a command. It is possible to receive these events through a subscription mechanism.

3 INFORMATION FLOW

As an intermediate step towards the enforcement of usage control policies in OO we define an information flow model for OO. The information flow model allows us to identify the actions that generate an information flow in and out of OO. As a next step towards the enforcement of usage control policies, the behaviour of these actions is to be controlled. Furthermore, these actions are the low-level actions which are the target of a mapping from high-level policy actions to low-level actions. For instance, a "do not edit" action is mapped onto inhibiting "copy", "paste" and "cut" commands and their shortcuts.

3.1 OO Information Flow Model

We define the fundamental entities that make up the information flow model [9]. The level of abstraction chosen has been motivated by our consideration of keeping the model simple on one hand but powerful enough on the other to allow for an appropriate design and implementation of enforcement mechanisms. Thus, the chosen abstraction of a data item D to be controlled is a document as policies are defined for documents as a whole, and thus

modelling the content of the document down to the level of characters, paragraphs, graphics etc. would have been too detailed. Further, taking the current implementation of OO into account, the entity dealing with a document is the OO process itself, however a document opened by OO is worked on by an editor instance. Several editor instances form the set of principals P. Data containers C are files which are referenced by a file name (= identifier). A data container holds zero or more data items; i.e. documents that are stored in ordinary files, backup files or in memory. Given an editor instance, certain actions like Save trigger an information flow between data containers (from memory into the original file). The relevant actions, i.e. where information is flowing between containers, we consider are: Open, Save, Save as, New, Close, Copy, Cut, Paste, Export, Insert and Delete.

The information flow model is defined over the above entities. We define states to consist of two elements: a storage function that maps containers to sets of data items and that is of type $C \rightarrow 2^D$; and a naming function that maps principals and identifiers to containers and that is hence of type $P \times F \rightarrow C$ where F is the set of identifiers (i.e. file names). Intuitively, *storage functions* capture which data is stored in which containers. The intuition behind the naming function is that a principal has certain container accessed using a specific name. States are accordingly defined as $\Sigma = (C \rightarrow 2^D) \times (P \times F \rightarrow C)$. Transitions between two states are effected by principals that perform actions: $\Sigma \times P \times A \rightarrow \Sigma$.

The initial state of the system is given by the allocation of documents in containers and the OO instance running but no editor instance.

The idea of the information flow model is to provide one particular kind of semantics for a system, namely the information flow in-between different containers. Monitoring this information flow is a prerequisite for the implementation of enforcement mechanisms. In the following section we concentrate on the first aspect, i.e. information flow monitoring.

3.2 OO Information Flow Monitor

The above OO information flow model is represented in Figure 1. It shows an OO instance and the containers the OO instance is using. Note the special containers for Insert and Delete content as well as Export and the Clipboard. The memory container is implicit with the editor instances.

According to the model, an information flow monitor can be implemented and monitor all the information flow that crosses the stroked line. This

monitor would do a bookkeeping of when and where information has flown.

The stroked line also indicates where enforcement mechanisms have to be in place to enforce usage control policies. Thus the flow monitor is an integral part of the enforcement architecture.

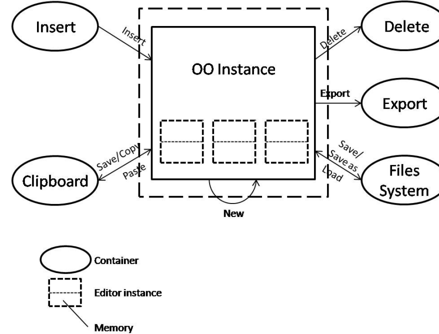

Figure 1: OO Information Flow Model

4 ENFORCEMENT ARCHITECTURE

Using the above described UNO an OO controller was designed that enforces usage control policies using different enforcement mechanisms. Here an enforcement mechanism is a piece of software that checks if the execution of a specific command is allowed or not according to some policy and then can allow or forbid, sometimes modify the execution of this command. The controller is an external Java program that connects to an OO instance and can control OO completely.

The controller consists of four parts which all have different functionality. The mode manager (MM) (Section 4.3) manages the different modes the controller software can be in: normal or in enforcement mode depending on the current document. Additionally it has the responsibility to store the configuration of the enforcement mechanisms per document (Figure 2). The policy decision point (PDP) (Section 4.1) is the component which decides if a mode change is necessary and triggers the transition into the respective new mode. The policy enforcement point (PEP) (Section 4.2) receives the information about which enforcement mechanisms have to be used from the PDP (via the mode manager) and configures the enforcement mechanisms accordingly. The policy manager (Section 4.4) is responsible for reading in an XML configuration file. The file specifies the enforcement mechanisms and their parameters thus representing the usage control policy for the document(s) currently loaded. Using the information from the XML file the policy manager supplies the MM (via the PDP) with the correct configuration information. A more detailed explanation follows.

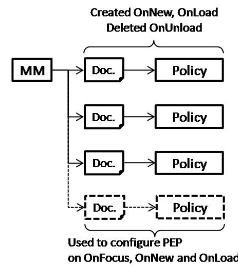

Figure 2: MM configuration

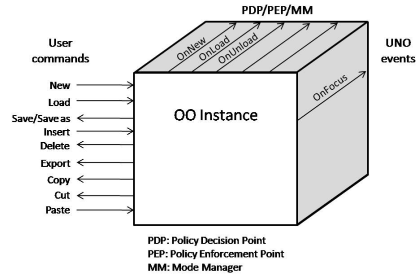

Figure 3: OO and PDP interaction

4.1 PDP

The PDP interacts with the OO instance as shown in Figure 3. The OO instance receives user commands and generates UNO events as discussed in Section 2.2. The PDP listens for UNO events (for more details on UNO events see [2]) and intercepts their execution as further detailed below. The most important events we are interested in are OnNew, OnLoad, OnUnload and OnFocus. If the OnNew event is received the default enforcement mechanisms have to be applied. If the event OnUnload is received the mode manager needs to be triggered that the configuration for the closed document can be deleted from the list of possible modes (Section 4.3). OnFocus indicates that the user is looking at another document thus a different set of enforcement mechanisms might be applicable. At the OnLoad event an existing document is newly loaded into OO thus the current mode has to be changed and a new one has to be created using the configuration information for this newly loaded document.

As the PDP is only considering the events mentioned before there is a *static* configuration of the enforcement mechanisms for a specific document at its first access. So there is no *dynamic* policy update implemented at the moment which means that once the document is loaded no change in the number of enforcement mechanisms for this document is taking place. Furthermore the inhibitors are always active when a document has the focus. Thus, a user can not issue a *forbidden* command because all forbidden commands are disabled from the menu.

The decision if an action is allowed or not is partly delegated to the enforcement mechanisms. For example the Save as modifier is deciding if the storage path for a document is correct or not and if not correct modifies it. An extension to the PDP, but not implemented yet, is a state machine which

decides if an action is allowed or not depending on the current state and past behaviour. For example, a policy might exist that allows a document to be printed only three times. The PDP would then decide that after the document has been printed three times it must not be printed anymore.

4.2 PEP

The PEP gets the configuration information of the mechanisms from the PDP respective mode manager and configures the individual mechanisms which are described in more detail in the following (Figure 4).

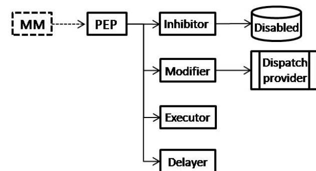

Figure 4: PEP configuration

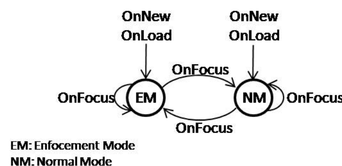

Figure 5: MM enforcement modes

Additionally the PEP sets the current enforcement mode whenever a document is newly loaded or focus changes; then the PEP acts as follows: First the inhibitor is used to disable the forbidden commands and if no command is forbidden all commands are enabled again.

Next existing modifiers are removed not to interfere with the new enforcement mode. After the removal it is checked if there are modifiers for this mode and which type of modifier (at the moment PrintModifier or SaveAsModifier can be selected) needs to be enabled. The name of the modifier is stored in a list. New modifiers are instantiated according to configuration information.

4.2.1 Inhibitor

The inhibitor can block any command that OO can execute. It is for example possible to forbid a user printing a specific document by disabling the shortcut Ctrl-P to start the print job and the Print menu entry.

On a technical level it is implemented like follows. The inhibitor gets from the PEP a list of forbidden commands. These commands are inserted into a configuration list provided by OO[1] (Figure 4). This list defines the

[1] This configuration list is to be found in /org.openoffice.Office.Commands/Execute/Disabled.

commands currently disabled. After the insertion of the commands into the list the configuration of OO is refreshed and the inserted commands are not available anymore neither via the menu or the button bar nor via shortcuts. Modifying this list and refreshing the configuration of OO is done every time there is a switch to another document. So inhibiting commands is done depending on the document.

4.2.2 Modifiers

The modifier is a DispatchProvider (Figure 4) which intercepts a command before it gets executed and modifies some parameters for this command. In OO every command has a so called DispatchProvider which implements the effect of the command. A newly written DispatchProvider can be executed before or instead of the original version. It can manipulate the parameters of a requested command or the content passed on to this command and then pass on the modified command and/or content to the original DispatchProvider where the command is finally executed. The new DispatchProvider can also perform some actions which are not executed in the original DispatchProvider and thus modify the result of the command execution.

Every document has its own list of DispatchProviders as for example a spreadsheet document needs other providers than a text document. Thus it is necessary to get the correct reference (frame) for the document so that the modifier can be inserted in the right list of DispatchProviders.

Modifiers have to be implemented using an abstract modifier class which requests that every modifier needs to implement the `configureModifier(...)` and `removeModifier()` methods. With the first method the modifier gets all necessary parameters for the configuration and the second method restores the original DispatchProvider. Furthermore `queryDispatch(...)` has to be re-implemented so that the correct (new) DispatchProvider is chosen, with all (new) functionality being implemented in the `dispatch(...)` method.

4.2.3 Execution of Action & Finite Delay

The execution of actions and the finite delay is not implemented yet but similar to the modifier there may exist several different executors and delayers and thus a generic version of both can not be implemented. For this reason an abstract executor class and an abstract delayer class are defined which need to be implemented. Each of the classes has two methods. One method for the configuration and the other one for the removal of the classes.

4.3 Mode Manager

The mode manager is responsible for managing the enforcement modes the controller software can be in (Figure 5). Basically two modes exist. The normal mode (NM) where no enforcement mechanism is active and an enforcement mode (EM). The enforcement mode is described by the enforcement mechanisms and their configuration active for one particular document. It exists in several variants where each variant is associated with one open document and might have different enforcement mechanism configurations associated with it. Thus the current active mode depends on the current document and describes the enforcement mechanisms to be applied

The mode manager passes the necessary configuration information of the enforcement mechanisms for the current document to the PEP whenever a mode change happens (Figure 5).

Technically two classes for the mode manager exist. One class holds all mode information including docURL (a string representation of the document name containing the URL where the document is stored) for the document and if modifiers, inhibitors, delayers or executors are existing (inclusive configuration information for those) for this mode. The other class manages the list of modes that are currently needed as there is one mode per opened document and depending on the document the mode is changed. Thus if a document is loaded a mode for this document is created and if a document is unloaded the corresponding mode is deleted.

4.4 Policy Manager

The policy manager consists of two parts. The translator class configures a new mode by inserting the configuration information from the XML configuration file into the mode to be newly created.

The second part is reading the configuration information from the XML configuration file and finally passing it on to the translator class. The configuration file specifies which modifiers, inhibitors, delayers and executors are applicable for this document

5 REMARKS

As mentioned above there are some limitations to the enforcement of usage control policies in OO using UNO. There is no support of some kind of access control on UNO APIs. Thus every extension or external program can revert

the modifications done by the controller using the same API. Furthermore only the functions provided by OO can be controlled so it is for example possible to prevent sending a document by e-mail using the menu entry but if the menu entry is enabled it can not be controlled to whom the document is send as the e-mail program is external. Here a coordination with the e-mail program would be necessary to enforce the policy for the document.

The clipboard is a special case which can not be controlled completely by the OO controller software as the clipboard itself is an external component but nevertheless an integral part of OO. The controller software can not prevent a user from copying something into the clipboard so it needs the help of the clipboard itself to control the flow of information to the clipboard. So the clipboard should have a controller software, too, that can communicate with the OO controller. Whenever something is copied from OO to the clipboard the clipboard controller can ask for the policy of the document the content is coming from. The OO controller knows which document has currently the focus and from which content can be copied into the clipboard. Thus the OO controller can forward the appropriate policy to the clipboard controller. The clipboard controller can then allow or not allow the pasting of the clipboard content.

Summarizing it is necessary to have also controllers in the other parts of the system like for example in the clipboard, the e-mail program or the operating system. Only with all the other controllers that work together it is possible to completely enforce a usage control policy.

6 SUMMARY

The paper showed how enforcement mechanisms for usage control can be implemented, based on an information flow model, using the UNO of OpenOffice.org. It was explained how the controller software for OpenOffice.org is working which enforces usage control policies. For this means a XML configuration file is read in that describes the configuration of the enforcement mechanisms for a document subject to a usage control policy. The information in the configuration file is used to build an internal representation of the mode OO has to be in for this document. Depending on which document has the focus the appropriate enforcement mechanisms, as specified in the configuration file, are applied.

Additionally it was shown that not all possible usage control policies can be enforced as the controller is for an application with a limited set of

functionalities. Some other restrictions to this approach were shown as there is for example no access control on the APIs UNO is providing.

References

[1] Microsoft rights management services. http://www.microsoft.com/rms.

[2] *OpenOffice.org Developer's Guide*. http://doc.services.openoffice.org /wiki/Documentation/DevGuide/OpenOffice.org_Developers_Guide.

[3] OpenOffice.org, April 2009. http://www.openoffice.org.

[4] M. Hilty, A. Pretschner, C. Schaefer, and T. Walter. Enforcement for Usage Control: A System Model and a Policy Language for Distributed Usage Control. Technical Report I-ST-20, DOCOMO Euro-Labs, December 2006.

[5] J. Ligatti, L. Bauer, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, February 2005.

[6] S. Media. *Sealed Media Enterprise DRM - How it works*. Sealed Media, July 2006. http://www.sealedmedia.com/products/how_sm_works.htm.

[7] OASIS. OASIS Open Document Format for Office Applications (OpenDocument), July 2008. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=office.

[8] J. Park and R. Sandhu. The UCON ABC Usage Control Model. *ACM Transactions on Informations and Systems Security*, 7:128–174, 2004.

[9] A. Pretschner, M. Büchler, M. Harvan, C. Schaefer, and T. Walter. Usage Control Policy Enforcement without and with Information Flow. Technical Report T5010131, DOCOMO Euro-Labs, February 2009.

[10] A. Pretschner, M. Hilty, and D. Basin. Distributed Usage Control. *CACM*, 49(9):39–44, September 2006.

[11] A. Pretschner, M. Hilty, F. Schütz, C. Schaefer, and T. Walter. Usage Control Enforcement - Present and Future. *IEEE Security & Privacy*, 6:44–53, July/August 2008.