Reusability of Functionality-Based Application Confinement Policy Abstractions

Z. Cliffe Schreuders and Christian Payne

School of IT, Murdoch University, South Street Murdoch WA 6150, Australia
{c.schreuders,c.payne}@murdoch.edu.au

Abstract. Traditional access control models and mechanisms struggle to contain the threats posed by malware and software vulnerabilities as these cannot differentiate between processes acting on behalf of users and those posing threats to users' security as every process executes with the full set of the user's privileges. Existing application confinement schemes attempt to address this by limiting the actions of particular processes. However, the management of these mechanisms requires security-specific expertise which users and administrators often do not possess. Further, these models do not scale well to confine the large number of applications found on functionality-rich contemporary systems. This paper describes how the principles of role-based access control (RBAC) can be applied to the problem of restricting an application's behaviour. This approach provides a more flexible, scalable and easier to manage confinement paradigm that requires far less in terms of user expertise than existing schemes. Known as functionality-based application confinement (FBAC), this model significantly mitigates the usability limitations of existing approaches. We present a case study of a Linux-based implementation of FBAC known as FBAC-LSM and demonstrate the flexibility and scalability of the FBAC model by analysing policies for the confinement of four different web browsers.

Keywords: Functionality-Based Application Confinement (FBAC), Role-Based Access Control (RBAC), Application-Oriented Access Control, Application Confinement, Sandbox, Usable Security, Reusable Policy.

1 Introduction

Existing widely used access control models reflect the traditional paradigm of protecting users from one another. Although user-oriented access control models such as traditional mandatory access control (MAC), discretionary access control (DAC) and role-based access control (RBAC) restrict the actions of users, these are generally not able to distinguish between an application performing legitimate actions on behalf of a user and code that is using these privileges nefariously. As a result, programs are essentially fully trusted: once executed malicious code typically has complete access to the user's privileges.

Application confinement models have been developed to restrict the privileges of processes, thereby limiting the ability of these programs to act maliciously. However, established application confinement models that allow finely-grained control

L. Chen, M.D. Ryan, and G. Wang (Eds.): ICICS 2008, LNCS 5308, pp. 206–221, 2008.

over access to resources require the construction of extremely complex policies [1, 2]. These require significant technical expertise to develop and have limited scalability as confining each application involves the construction of a new detailed policy. Unfortunately, to date this has limited their practical usefulness and acceptance.

By recognizing that the goal of restricting users is essentially analogous to that of restricting applications, it follows that the principles from existing user-oriented access control models may be applied to the problem of process confinement. Specifically, applying the principles of role-based access control to application confinement leverages the flexibility and efficient management of RBAC to provide hierarchical policy abstractions for restricting applications, which eases policy development and association. These constructs can be parameterised to provide flexible and reusable application-oriented policy abstractions for improved usability, manageability, scalability and security.

2 Background

2.1 Application Confinement

A number of application confinement models have been developed to provide restricted environments for applications, thereby limiting their ability to behave maliciously. Some simply isolate programs to a limited namespace using a traditional sandbox [3] or virtual machine [4], examples include `chroot()`, FreeBSD jails [5] Solaris Zones [6], and Danali [7]. Others allow restricted access to selected shared resources, such as the Java [8] and .NET [9] sandboxes where applications are restricted by complex administrator-specified policies based on the properties of the code. Some models exist based on the paradigm of label-based integrity preservation where subjects are labelled high or low in integrity and the flow of information between levels serves as the basis for policy [10, 11]. Other restricted environments require the specification of a detailed policy detailing each application's access to specific resources. This applies to confinement mechanisms including Janus [12], Systrace [13], Novel AppArmour [14] (previously known as SubDomain), TRON [15], POSIX capabilities [16], Bitfrost [17], CapDesk [18], and Polaris [19]. Methods of mediating this type of access control include using capabilities [20] or system call interposition [2]. Other schemes, such as domain and type enforcement (DTE) [21] and the Role Compatibility model [22] allow the definition of multiple restricted environments, and propagating processes transition between them.

Unfortunately all of these mechanisms have limitations and problems. Isolation-based approaches typically involve significant redundancy as shared resources must be duplicated and they also severely limit the ability of applications to exchange data with one another [23]. On the other hand, more finely-grained restricted access control policies are difficult and time-consuming to define and manage. The task of translating high level security goals into finely grained policies is problematic, making these policies difficult to both construct and verify for completeness and correctness [1, 24]. Furthermore, once constructed an individual policy will apply primarily to only a single application, meaning that the work involved in constructing suitable policies for all necessary applications is considerable. For example, specifying DTE

domain policies is complex and although multiple processes can be confined by a single domain, domains must be specified separately [25]. There is significant overlap of privileges granted to compiled domain policies, and typically any non-trivial application is assigned a separate domain. Finally, specifying file and domain transitions can also be a complex task as programs need specific authorisation to label files as being accessible to programs in different domains, and users and programs both need permission in order to execute programs belonging in another domain [26].

These application confinement schemes lack flexible policy abstractions which can allow application access policies to be meaningfully reused while providing fine grained restrictions. With isolation sandboxes the container itself acts as the only policy abstraction – a simple collection of subjects and resources. Existing schemes which mediate finely-grained privileges to applications are generally either devoid of policy abstraction (a list of privileges are directly associated with a program) or contain large monolithic self-contained abstractions (such as DTE domains or RC roles) which cannot be flexibly reused for different applications unless they share the exact same privilege requirements. As a result, these application confinement architectures do not provide a practical or scaleable solution for conveniently confining multiple applications.

While a few implementations of these models allow policy abstractions to be comprised of smaller components they are reduced to a single monolithic policy abstraction before use, which limits their usefulness at run-time and their reusability. For example, SELinux's DTE Domain specification can include macros in the m4 language. Before policy is applied, these are expanded into many lines of rules granting all the required privileges. The result is a single domain with a fixed set of privileges, typically those required by a single program. Likewise, at system start-up abstractions in AppArmor application profiles are translated into a raw list of privileges associated with the program. This monolithic approach to policy abstraction also means that any finer grained abstractions which may have been used to construct policy are not available when managing the privileges of a process.

2.2 Role-Based Access Control (RBAC)

Role-based access control (RBAC) is a user-oriented access control model which associates users with privileges via organizational abstractions known as *roles* [27]. When a user joins an organisation they are assigned the roles representing the privileges required by their responsibilities and duties and this eliminates the task of manually assigning permissions to each new user [28]. Access decisions are then made based on the permissions associated with the roles the user is assigned. Policy reusability is enhanced through role hierarchies which allow roles to be defined in terms of other roles. Also, role constraints such as *separation of duty* can restrict certain conflicting permissions from being associated with the same user (*static separation of duty*) or accessed concurrently (*dynamic separation of duty*) [29].

Many similarities can be observed between the motivation for the development of RBAC in relation to traditional access controls and the current problems faced in the domain of application confinement. RBAC provides a conceptually-straightforward, scalable and abstract association between users and the privileges they require in order to perform their designated duties within an organisation. This highlights the

advantages a model which provides similar abstract associations between applications and the privileges they require can provide to application-oriented access control.

3 Functionality-Based Application Confinement

3.1 Policy Abstraction

The notional similarities previously noted between user confinement via access controls and application confinement models suggest the applicability of traditional access control principles to the problem of restricting applications. In particular, many of the design principles of RBAC can be applied to manage the privileges of executing programs. Based on this, a model known as *functionality-based application confinement* (FBAC) has been developed [30]. Designed to be analogous to the specifications contained in the NIST/ANSI INCITS RBAC model [31, 32], FBAC acts as an additional layer above traditional access control models and treats all software that the user executes as untrusted by limiting its access to only the resources deemed necessary for the application to operate as required.

Application confinement policies can be defined in terms of their behavioural classes [33] which are conceptually analogous to RBAC roles. FBAC uses abstractions similar to RBAC roles and role hierarchies which are used to define complex, finely-grained application confinement policies in terms of high level abstractions. Consequently applications are confined to those resources deemed necessary by its assigned *functionalities*.

Functionalities are hierarchical policy abstractions which form the basis of FBAC policy. Functionalities can represent high-level behavioural classes of applications (for example, “Web_Browser” or “Web_Server”) and these can inherit lower level functionalities that represent application functionality such as “http_client”, “ftp_ client” and “read_files_in_directory”. These functionalities are associated with privileges that are made up of *operations* on *objects*.

The RBAC model and an FBAC confinement are structurally analogous but very different in purpose. While RBAC is a user confinement model for system administrators to restrict what permissions users hold according to their duties within an organisation, FBAC is a framework for users to restrict the privileges of each application based on the functionality it provides.

Related to the concept of discretionary role-based access control (DRBAC) [34, 35] where users have the ability to define and activate their own RBAC roles, FBAC also applies RBAC concepts to allow users to confine themselves; however, FBAC is focused on restricting applications rather than users.

3.2 Parameterisation

While RBAC roles are self-contained with each user receiving the same set of privileges [36], in an application confinement context behavioural classes are better defined in terms of parameterised categories [33]. Unlike RBAC role associations, FBAC functionality associations are parameterised to allow functionalities to adjust to the needs of different applications. For example, although an application may be classified by a general grouping such as “Web_Server”, in order to create an effective

confinement policy certain application-specific details (such as the location of files and directories it uses) must still be defined.

FBAC provides parameterised functionalities to allow policies to be more precisely defined in terms of application-specific details. FBAC functionalities are passed arguments in a way similar to how subroutines are in programming languages. This allows the policy abstraction to be adapted to the specifics of individual applications providing related features. Functionality definitions can also contain default arguments which allow further ease of use in common cases without sacrificing flexibility. This means applications are defined in terms of functionalities plus any information required by those functionalities. Functionalities may use this information to inherit from other functionalities or define the resources associated with operations.

3.3 Mandatory and Discretionary Controls

Unlike existing application confinement schemes which are either applied as a discretionary control (such as Janus or TRON) or as a mandatory control (such as with DTE or AppArmor), FBAC supports both mandatory and discretionary access controls simultaneously. Administrators can specify policies which govern the behaviour of applications to enforce system-wide security goals, restrict users to particular programs, and manage user protection. Users may then further confine these applications to protect their own resources from malicious code.

This is achieved by layering FBAC confinements. A confinement may apply to multiple users and may reuse the functionalities from other confinements. The resulting authority granted to an application is the intersection of the confinements for that application which apply to the executing user. This layered approach to application confinement is unique and provides defence in depth while requiring the maintenance of only one mechanism. Because confinements can share the same functionalities this greatly reduces the overhead of managing multiple layers of application-oriented access controls, while enforcing the security goals of both users and administrators.

4 Defining and Managing Policy

The FBAC model greatly simplifies the management of application confinement policies compared with existing models. Functionalities are established representing the various functional requirements of applications. Privileges can be assigned to these functionalities directly and may also be inherited by other contained functionalities. The applications have these functionalities associated with them as required by their expected behaviour and when the program is executed, this will activate the functionalities that apply to it and thus define its privileges at runtime.

Initial policy definition in FBAC involves the creation of new functionalities in terms of low level privileges and existing functionalities, assigning the rights necessary for applications to function according to the behaviour described by functionalities. This is influenced by security goals and application behaviour and resource requirements. Although the design of FBAC significantly reduces the complexity of privilege assignment compared with other finely-grained confinement models, this

initial process does require greater expertise than other aspects of the framework and may be completed by a trusted third party rather than by end users.

Once defined these functionalities may be reused by multiple users to restrict as many applications as appropriate. End users require little expertise to identify the functionalities relevant to their applications based upon the program's expected behaviour. These are then associated with the application and parameters are provided where necessary. This process is far simpler than with alternative confinement techniques where complex policies must be defined for each individual application.

5 Web Browser Case Study

A language for expressing FBAC policies has been developed and a prototype implementation of the model as a Linux Security Module (LSM) [37] called *FBAC-LSM* is near completion. The policy requirements for a number of applications were analysed and a hierarchal FBAC policy for FBAC-LSM has been created. We now present a case study of the application of FBAC policies to four common web browsers — Firefox, Opera, Epiphany and Lynx — for the purposes of demonstrating policy flexibility and reusability.

5.1 Restricting Applications

To confine a web browser such as Firefox using the graphical policy manager tool the user simply chooses the high level functionalities relating to that application's functionality. The user assigns a base functionality such as “Standard_ Graphical_ Application_Base” and any high level functionalities which describe what the application is expected to do (such as the “Web_Browser” functionality as defined in Figure 2 and provides application specific parameters such as where the program is installed, the location of its configuration files, where the program downloads files to, and potentially a list of hosts it can connect to. If confining a browser such as Opera that supports additional functionality, other corresponding high level functionalities are also assigned such as “Email_Client”, “Irc_Chat_Client”, “News_ Reader_ Client” and “BitTorrent_Client”.

Unlike some operating systems where each application's files are typically found in a very small number of directories, Linux organises application files based upon the filesystem hierarchy standard (FHS). This can lead to an application's files being spread throughout the filesystem tree and in some cases parameter value specification may necessitate a degree of familiarity with this arrangement. Any complexity due to this can be mitigated by the use of parameter descriptions suggesting the location of files according to the FHS and the provision of a list of pathnames used by a program (for example, as in the case of Opera which provides this information to the user). Furthermore, techniques are currently being developed to automatically derive parameter values based on associated functionalities, and package management and filesystem analysis. A graphical policy management tool has been created which removes the need for end-users to be familiar with the FBAC-LSM policy language and policy association becomes a matter of pointing

and clicking. However, even so, the FBAC-LSM policy language is simpler and provides greater abstraction than existing alternatives.

A FBAC policy for the Firefox browser created with the graphical policy manager tool is given in Figure 1. For comparison purposes, additional policies for the three other browsers considered in the case study are contained in Appendix A.

The Firefox policy from Figure 1 begins by specifying the executables which are used to run the application (binarypaths). Next it identifies the two functionalities that this application encompasses: "Standard_Graphical_Application_Base" and "Web_ Browser". These functionalities are parameterised to address the specifics of the application, for example to specify where the various files it uses are located and the hosts to which it is permitted to connect. These parameters can easily be changed to grant the application access to different resources. For example, to restrict the web browser to only connect to particular servers (such as on an intranet) the allowed_ hosts_ to_ connect_to parameter value can be changed.

```
application firefox
{
    binarypaths /usr/bin/firefox:/usr/bin/X11/firefox:
           /usr/lib/firefox/firefox:/usr/lib/firefox/firefox.sh;
    functionality Standard_Graphical_Application
        (peruser_directory="/home/*/.mozilla/firefox/",
        peruser_files="/home/*/.mozilla/appreg",
        application_libraries_directory="/usr/lib/firefox/",
        libraries_fileextension="*.so",
        config_directory={"/home/*/.mozilla/":"/home/*/.gnome2_private/"},
        config_files="",
        read_only_directory="");
    functionality Web_Browser
        (plugins_and_extensions_directory={"/home/*/.mozilla/plugins/":
            "/usr/lib/firefox/extensions/":
            "/usr/lib/browser-plugins/firefox/"},
        download_directory={"/home/*/Desktop/":"/home/*/downloads/"},
        allowed_hosts_to_connect_to="*",
        view_web_files_in_directory="/home/**/");
}
```

Fig. 1. Entire FBAC-LSM policy for Mozilla Firefox

5.2 Defining Functionalities

Each high level functionality is made up of lower level functionalities and privileges. For example, the "Web_Browser" functionality incorporates many inherited functionalities including "http_client", "Ftp_Client" and "Web_Files_Viewer" which are in turn made up of other functionalities and direct privileges.

The "Web_Browser" functionality policy shown in Figure 2 is syntactically the same as the application policy in the previous figure, with additional concepts such as the definition of parameters (followed by their default values), and information for the graphical tool. Descriptions of functionalities and parameters assist the user, while the granularity of the functionality (high or low level) and a category (in the "Web_ Browser" case "network_client") allow the graphical tool to flexibly present the policy to the user. Note the scalability of the abstractions provided by the functionalities construct is demonstrated by the fact that the four web browsers considered in the case study use the same underlying functionality definition.

```
functionality Web_Browser
{
      functionality_description "a web browser, and ftp client";
      highlevel;
      category network_client;
      parameter plugins_and_extensions_directory
              "/home/*/.[APPLICATION_NAME]/plugins/";
      param_description "the directory the application keeps any app-specific
      plugins or extensions";
      parameter download_directory "/home/*/downloads";
      param_description "the directories downloads are stored to";
      parameter allowed_hosts_to_connect_to "*";
      param_description "hosts the browser can connect to";
      parameter view_web_files_in_directory "/home/**/";
      param_description "view web files in this dir (.htm, .jpg...)";
      functionality general_network_connectivity_and_file_access ( );
      functionality http_client (allowed_hosts_to_connect_to, <default>);
      functionality save_downloads (download_directory);
      functionality extensions_plugins (plugins_and_extensions_directory, "*");
      functionality mime_aware ( );
      functionality web_plugins_and_helpers ( );
      functionality Ftp_Client (allowed_hosts_to_connect_to);
      functionality Web_Files_Viewer (view_web_files_in_directory, <default>);
}
```

Fig. 2. FBAC-LSM web browser functionality definition

Privileges are low level rights defined as operations on objects, which represent the security-related kernel actions which allow access to the resources that are necessary for that functionality. Low level functionalities, such as “files_r” in Figure 3, provide abstractions to group together related low level privileges. In this functionality the parameter “files” is used to grant both read and “get attribute” access to these files.

```
functionality files_r
{
    functionality_description "read access to these files";
    lowlevel;
    parameter files "";
    param_description "allows these files to be accessed as described";
    privilege file_read files;
    privilege file_getattr files;
}
```

Fig. 3. Low level FBAC-LSM functionality and privileges

The results are policies for these web browsers which enforce the principle of least privilege by confining the application to a restricted set of privileges required for the application to complete its required duties. Consequently the actions of any malware or the effects of any exploited security vulnerability are confined to the behaviour allowed by its functionality-oriented policy.

5.3 Comparison with Other Mechanisms

The FBAC model has significant advantages over existing systems. A policy to confine a complex application such Firefox using standard system call interposition mechanisms such as Systrace or Janus results in a complex series of low level rules

specifying which system calls are allowed and under what circumstances. This is illustrated by the excerpt from a Systrace policy given in Figure 4 which only represents a tiny portion of the complete policy. The resulting policy is generally extremely complex and it is difficult to verify that this policy is in fact correct [38].

```
    native-fsread: filename eq "/usr/libexec/ld.so" then permit
    native-fsread: filename eq "/usr/sbin/suexec" then permit
    native-fsread: filename eq "/var/run/ld.so.hints" then permit
    native-fsread: filename eq "/var/www" then permit
    native-fsread: filename eq "<non-existent filename>" then deny[enoent]
```

Fig. 4. Excerpt from a Systrace policy

Similarly, managing NSA's SELinux policy requires expertise beyond that of typical users or system administrators. Under SELinux the policy which applies is the net result of the configuration of multiple access control models (including RBAC, DTE, Multi-level Security and User Identity) and can be hard to verify for correctness or completeness [1, 39]. For example, Figure 5 demonstrates the complexity and inscrutability of a SELinux policy by providing a brief excerpt from an SELinux reference policy for Mozilla [40]. Although domains serve as policy abstractions, each application is usually assigned a unique domain consisting of complex rules specifying allowed file and domain transitions and interactions with types (similarly labelled objects). While SELinux is capable of meeting strong confidentiality requirements, it is not well suited to end users confining potentially malicious applications [41].

```
    manage_dirs_pattern($2,$1_mozilla_home_t,$1_mozilla_home_t)
    manage_files_pattern($2,$1_mozilla_home_t,$1_mozilla_home_t)
    manage_lnk_files_pattern($2,$1_mozilla_home_t,$1_mozilla_home_t)
    relabel_dirs_pattern($2,$1_mozilla_home_t,$1_mozilla_home_t)
    relabel_files_pattern($2,$1_mozilla_home_t,$1_mozilla_home_t)
    relabel_lnk_files_pattern($2,$1_mozilla_home_t,$1_mozilla_home_t)
    manage_files_pattern($1_mozilla_t,$1_mozilla_tmpfs_t,$1_mozilla_tmpfs_t)
    manage_lnk_files_pattern($1_mozilla_t,$1_mozilla_tmpfs_t,$1_mozilla_tmpfs_t)
    manage_fifo_files_pattern($1_mozilla_t,$1_mozilla_tmpfs_t,$1_mozilla_tmpfs_t)
    manage_sock_files_pattern($1_mozilla_t,$1_mozilla_tmpfs_t,$1_mozilla_tmpfs_t)
    fs_tmpfs_filetrans($1_mozilla_t,$1_mozilla_tmpfs_t,{ file lnk_file sock_file
fifo_file })
    allow $1_mozilla_t $2:process signull;
    domain_auto_trans($2, mozilla_exec_t, $1_mozilla_t)
    # Unrestricted inheritance from the caller.
    allow $2 $1_mozilla_t:process { noatsecure siginh rlimitinh };
```

Fig. 5. Excerpt from Mozilla interface rules in the Tresys SELinux reference policy [42]

Novell's AppArmor policy specification format lists the resources an application may access along with the type of access required [14]. This is illustrated in Figure 6. Although this simplifies policy readability, it exposes the underlying complexity of the system. As a result an in-depth knowledge of both the application being confined and low-level details of the operating system's shared resources and services are required in order to properly review the automatically generated policy. Construction of AppArmor policies typically relies on recording process activity, while FBAC policies are constructed based on high level security goals. Further, while AppArmour allows collections of access rules to be grouped into abstractions, these are comparatively inflexible. For example, unlike AppArmour, FBAC has the ability to disable parts

```
/etc/mailcap r,
/etc/mime.types r,
/etc/mozpluggerrc r,
/etc/opt/gnome/gnome-vfs-*/modules r,
/etc/opt/gnome/gnome-vfs-*/modules/*.conf r,
/etc/opt/gnome/pango/* r,
/etc/opt/kde3/share/applications/mimeinfo.cache r,
/etc/rpc r,
/etc/sysconfig/clock r,
/opt/gnome/lib/GConf/2/gconfd-2 Px,
/opt/gnome/lib/gnome-vfs-*/modules/*.so mr,
/opt/gnome/lib/gtk-*/**.so* mr,
/opt/gnome/lib/lib*so* mr,
/opt/gnome/lib/pango/**.so mr,
/opt/gnome/lib64/lib*so* mr,
```

Fig. 6. Excerpt from AppArmor's Firefox profile

of policy on the fly and specify separation of duty, while the parameterised nature of FBAC functionalities allows these to be easily adapted to differing application requirements.

MAPbox provides behaviour based application confinement by allowing software authors to specify a program's behaviour class which describes generally what the program does along with some application-specific parameters [33, 43]. MAPbox's designers identified 14 program classes and corresponding restricted environments are associated with applications based on these author-assigned classes. These restricted environments are defined by complex finely-grained rules specified by the user. While the use of behavioural classes to create an association between policies and programs is an important contribution, policy management in MAPbox remains complex for users. Furthermore applications may only be associated with a single behavioural class which is problematic given many contemporary applications provide a variety of functionality; for example, the Opera web browser. Like MAPbox, FBAC also restricts applications based upon parameterised classes. However, FBAC allows applications to be associated with multiple functionalities and its hierarchical approach to policy management supports multiple levels of abstraction, bringing numerous advantages. For example, FBAC functionalities may be defined hierarchically whereas MAPbox's sandboxes are defined individually. Unlike MAPbox, FBAC allows users to easily restrict arbitrary applications to protect themselves from programs they do not trust. Furthermore FBAC-LSM's use of the LSM interface avoids the problems inherent in MAPbox's use of the system call interface as a security layer [38].

Generally therefore, in contrast to these mechanisms, FBAC-LSM separates and abstracts the task of developing low level policy rules from the task of defining the expected behaviour of a specific program. This allows users, administrators and software authors — in fact uniquely any combination of authorised policy sources — to restrict what an application can do using high level abstractions which can then be easily fine-tuned via parameterisation to suit different applications. Compared to alternative finely-grained application confinement models, FBAC may be used to confine very complex software packages such web browsers using a hierarchical policy that is far easier to manage. While the above confinement methods are either system wide and mandatory (SELinux/DTE, AppArmor) or per-user and discretionary (Janus/Systrace, MAPbox), FBAC-LSM simultaneously enforces mandatory and discretionary FBAC policies. Under FBAC users can configure their own security policy to protect themselves while administrators are able to define system-wide policies to

protect system security, enforce organisation level security goals and, when necessary, administer policy to protect specific users. These restrictions on applications severely limit the impact from malware or exploitation of any software vulnerabilities.

6 Discussion

6.1 Manageability and Usability

The policies for the four web browsers presented here (in Figure 1 and Appendix A) are defined in terms of high level security goals. In contrast with other application confinement schemes such as those previously discussed, FBAC allows succinct high level policies to be defined using flexible abstractions. This both reduces and simplifies the management task involved in creating policies to confine individual applications. The programs are simply identified as web browsers and application specific information is supplied. In the case of Opera other high level functionalities are also specified. The finely grained privileges inherited by functionalities are separated from the specification of application policies, thus making policy specification easier than with other schemes. The flexibility to restrict applications based on abstract descriptions of what the application can do provides a significant improvement in usability, making it easier to translate high level security requirements into finely grained policies.

Furthermore, the "Web_Browser" functionality (in Figure 2) demonstrates that the hierarchical structure of policies allows functionalities themselves to also be defined in terms of abstractions, such as "http_client". Policies can be reviewed from their high level functionalities (such as "Web_Browser" in Figure 2) to lower level detail and right down to the privileges specifying permissible operations on designated objects (such as "file_r" in Figure 3). This makes finely grained policies easier to manage and comprehend as the policy is made up of levels of abstractions which can encapsulate low-level details.

6.2 Scalability

Once functionalities such as "Web_Browser" have been defined, policies for each application which provides the described functionality can be defined in terms of these constructs. All four web browsers studied reuse the "Web_Browser" policy abstraction. This leverages the fact that many applications may be categorised into the same behavioural classes and can be confined to easily identified sets of privileges required for the applications to carry out their intended functions [33]. Rather than confining an application by specifying each distinct privilege required, they can be simply defined in terms of the behavioural classes to which they belong. Thus the model scales well to confine the numerous applications typically found on contemporary systems.

The use of functionality hierarchies also increases the scalability of policy management by facilitating greater reuse of existing defined policy. For example the "Web_Browser" functionality includes the functionality "Ftp_Client" which itself can be used to describe applications which may not be web browsers. The use of hierarchies increases abstraction while reducing redundancy.

6.3 Security

Using application confinement schemes such as FBAC to limit program privilege provides significant security improvements over simply relying on user-oriented access control mechanisms. FBAC enforces the principle of least privilege by confining applications to the set of privileges required for them to do their job. Although the abstract nature of functionalities may potentially grant an application more privileges than they actually use, in general these additional privileges simply allow the application to carry out its authorised tasks in varied ways.

If an application attempts to exercise privileges it does not hold the request is denied. For example, if due to the introduction of malicious code a restricted web browser attempts to act outside of the behaviour defined by its associated functionalities the action would be prevented. This limits the ability of applications to behave maliciously whether deliberately or otherwise.

The underlying FBAC-LSM policy granularity is finely grained and is determined by the LSM interface. This design provides scope for the future inclusion of additional features such as stateful network packet inspection.

Compared with other confinement models the FBAC framework provides equivalent security benefits. However, the superior convenience, simplicity, flexibility and scalability of the FBAC model makes it far better suited to ubiquitous deployment.

However, beyond this FBAC has other security advantages. For example, the separation of duty feature is unique in the area of application confinement and allows high level security policies to specify privileges or functionalities that cannot be exercised simultaneously. Static separation of duty prevents conflicting privileges from being assigned to the same application while dynamic separation of duty stops applications from exercising certain privileges concurrently. This limits the ability of high level security goals to be accidentally subverted by low-level security policies.

Also, as FBAC's policy abstractions are natively hierarchical, parts of the policy can be easily activated or deactivated at run time. This is not possible using the existing application-oriented access control models such as DTE, RC or AppArmor as privileges are contained in a monolithic abstraction associated with the security context. FBAC's hierarchy of functionalities allows run-time intervention to dynamically deactivate or activate branches of functionalities. This could be requested by a user, administrator or the software itself. For example using a multi-purpose application (such as Opera web browser, email, irc, new reader and bittorrent client) the user or the application itself may wish to only enable the functionality corresponding to the feature the program is performing. This is equivalent to the concept of an RBAC user activating only those roles corresponding to the job he or she is currently performing.

FBAC also restricts applications based on a combination of policies representing the security goals of users and administrators. While existing controls provide either mandatory or discretionary application confinement, FBAC provides both through layers of confinements. Policy can be reused across confinements and only one mechanism needs to be maintained.

7 Conclusion

The case study and corresponding analysis of the FBAC model presented here demonstrates that applying parameterised RBAC constructs to the problem of application confinement can provide clear advantages over alternative approaches. FBAC separates the task of policy construction from the association of these policies with specific applications. This simplifies the process as users or administrators can assign pre-specified generic policies based upon an application's anticipated functionality rather than needing to construct individual policies for each program. FBAC utilises functionalities as an abstract policy construct and by allowing the definition of new functionalities in terms of existing ones, a hierarchy is created which improves usability, manageability and scalability. While end users can simply assign policies based upon high-level functionalities, security administrators and analysts can study policy construction at multiple levels. The separation of duty mechanism also ensures that high level policy goals are maintained during the construction of low-level policies. Finally, the use of parameterisation allows confinement policies to be easily adapted to deal with subtle differences between similar applications and this further improves policy reusability. As demonstrated by the four web browser policies presented, the usability and management improvements provided by FBAC make deploying application confinement significantly easier and could therefore have the potential to encourage broader adoption of such security mechanisms in the future.

References

1. Zanin, G., Mancini, L.V.: Towards a Formal Model for Security Policies Specification and Validation in the SElinux System. In: Proceedings of the Ninth ACM Symposium on Access Control Models and Technologies, pp. 136–145. ACM Press, Yorktown Heights (2004)
2. Goldberg, I., Wagner, D., Thomas, R., Brewer, E.A.: A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In: Proceedings of the 6th USENIX Security Symposium. University of California, San Jose (1996)
3. Kamp, P.-H., Watson, R.: Building Systems to be Shared Securely. ACM Queue 2, 42–51 (2004)
4. Madnick, S.E., Donovan, J.J.: Application and Analysis of the Virtual Machine Approach to Information Security. In: Proceedings of the ACM Workshop on Virtual Computer Systems, Cambridge, MA, USA, March 1973, pp. 210–224 (1973)
5. Kamp, P.-H., Watson, R.: Jails: Confining the Omnipotent Root. In: Sane 2000 - 2nd International SANE Conference (2000)
6. Tucker, A., Comay, D.: Solaris Zones: Operating System Support for Server Consolidation. In: 3rd Virtual Machine Research and Technology Symposium Works-in-Progress
7. Whitaker, A., Shaw, M., Gribble, S.D.: Lightweight virtual machines for distributed and networked applications. In: Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation, pp. 195–209 (2002)
8. Gong, L., Mueller, M., Prafullchandra, H., Schemers, R.: Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In: USENIX Symposium on Internet Technologies and Systems. Prentice Hall PTR, Monterey (1997)

9. Thorsteinson, P., Ganesh, G.G.A.: Net Security and Cryptography, p. 229. Prentice Hall PTR, Englewood Cliffs (2003)
10. Li, N., Mao, Z., Chen, H.: Usable Mandatory Integrity Protection for Operating Systems. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 164–178 (2007)
11. Sun, W., Sekar, R., Poothia, G., Karandikar, T.: Practical Proactive Integrity Preservation: A Basis for Malware Defense. Security and Privacy. In: IEEE Symposium on SP 2008, pp. 248–262 (2008)
12. Wagner, D.A.: Janus: An Approach for Confinement of Untrusted Applications. Technical Report: CSD-99-1056. Electrical Engineering and Computer Sciences. University of California, Berkeley, USA (1999)
13. Provos, N.: Improving Host Security with System Call Policies. In: 12th USENIX Security Symposium, vol. 10. USENIX, Washington (2002)
14. Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P., Gligor, V.: SubDomain: Parsimonious Server Security. In: USENIX 14th Systems Administration Conference (LISA) (2000)
15. Berman, A., Bourassa, V., Selberg, E.: TRON: Process-Specific File Protection for the UNIX Operating System. In: Proceedings of the 1995 Winter USENIX Conference (1995)
16. Bacarella, M.: Taking advantage of Linux capabilities. Linux Journal (2002)
17. Krsti, I., Garfinkel, S.L.: Bitfrost: the one laptop per child security model. In: ACM International Conference Proceeding Series, vol. 229, pp. 132–142 (2007)
18. Miller, M.S., Tulloh, B., Shapiro, J.S.: The structure of authority: Why security is not a separable concern. In: Multiparadigm Programming in Mozart/Oz: Proceedings of MOZ 3389 (2004)
19. Stiegler, M., Karp, A.H., Yee, K.P., Close, T., Miller, M.S.: Polaris: virus-safe computing for Windows XP. Communications of the ACM 49, 83–88 (2006)
20. Wagner, D.: Object capabilities for security. In: Conference on Programming Language Design and Implementation: Proceedings of the 2006 workshop on Programming languages and analysis for security, vol. 10, pp. 1–2 (2006)
21. Badger, L., Sterne, D.F., Sherman, D.L., Walker, K.M., Haghighat, S.A.: Practical Domain and Type Enforcement for UNIX. In: Proceedings of the 1995 IEEE Symposium on Security and Privacy, p. 66. IEEE Computer Society, Los Alamitos (1995)
22. Ott, A.: The Role Compatibility Security Model. In: 7th Nordic Workshop on Secure IT Systems (2002)
23. Krohn, M., Efstathopoulos, P., Frey, C., Kaashoek, F., Kohler, E., Mazieres, D., Morris, R., Osborne, M., VanDeBogart, S., Ziegler, D.: Make least privilege a right (not a privilege). In: Procedings of 10th Hot Topics in Operating Systems Symposium (HotOS-X), Santa Fe, NM, USA, pp. 1–11 (2005)
24. Marceau, C., Joyce, R.: Empirical Privilege Profiling. In: Proceedings of the 2005 Workshop on New Security Paradigms, pp. 111–118 (2005)
25. Jaeger, T., Sailer, R., Zhang, X.: Analyzing Integrity Protection in the SELinux Example Policy. In: Proceedings of the 12th USENIX Security Symposium, pp. 59–74 (2003)
26. Hinrichs, S., Naldurg, P.: Attack-based Domain Transition Analysis. In: 2nd Annual Security Enhanced Linux Symposium, Baltimore, Md., USA (2006)
27. Ferraiolo, D., Kuhn, R.: Role-Based Access Control. In: 15th National Computer Security Conference, Baltimore, MD, USA, pp. 554–563 (1992)
28. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-Based Access Control Models. IEEE Computer 29, 38–47 (1995)
29. Simon, R.T., Zurko, M.E.: Separation of Duty in Role-Based Environments. In: Proceedings of 10th IEEE Computer Security Foundations Workshop, Rockport, MD, pp. 183–194 (1997)

30. Schreuders, Z.C., Payne, C.: Functionality-Based Application Confinement: Parameterised Hierarchical Application Restrictions. In: Proceedings of SECRYPT 2008: International Conference on Security and Cryptography, pp. 72–77. INSTICC Press, Porto (2008)
31. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST Standard for Role-Based Access Control. ACM Transactions on Information and System Security 4, 224–274 (2001)
32. ANSI INCITS 359-2004. American National Standards Institute / International Committee for Information Technology Standards (ANSI/INCITS)
33. Acharya, A., Raje, M.: MAPbox: Using Parameterized Behavior Classes to Confine Applications. In: Proceedings of the 2000 USENIX Security Symposium, Denver, CO, USA (2000)
34. Jaeger, T., Prakash, A.: Requirements of role-based access control for collaborative systems. In: Proceedings of the first ACM Workshop on Role-based access control, p. 16. ACM Press, Gaithersburg (1996)
35. Friberg, C., Held, A.: Support for discretionary role based access control in ACL-oriented operating systems. In: Proceedings of the second ACM workshop on Role-based access control, pp. 83–94. ACM Press, Fairfax (1997)
36. Jansen, W.A.: Inheritance Properties of Role Hierarchies. In: Proceedings of the 21st National Information Systems Security Conference, pp. 476–485. National Institute of Standards and Technology, Gaithersburg (1998)
37. Wright, C., Cowan, C., Smalley, S., Morris, J., Kroah-Hartman, G.: Linux Security Module Framework. In: Ottawa Linux Symposium, Ottawa, Canada (2002)
38. Garfinkel, T.: Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In: Proceedings of the 10th Network and Distributed System Security Symposium, pp. 163–176. Stanford University, San Diego (2003)
39. Bratus, S., Ferguson, A., McIlroy, D., Smith, S.: Pastures: Towards Usable Security Policy Engineering. In: Proceedings of the Second International Conference on Availability, Reliability and Security, pp. 1052–1059 (2007)
40. Tresys: SELinux Reference Policy (2008)
41. Harada, T., Horie, T., Tanaka, K.: Towards a manageable Linux security. In: Linux Conference 2005 (Japanese) (2005), http://lc.linux.or.jp/lc2005/02.html
42. Tresys: SELinux Reference Policy (2008), http://oss.tresys.com/projects/refpolicy
43. Raje, M.: Behavior-based Confinement of Untrusted Applications. TRCS 99-12. Department of Computer Science. University of Calfornia, Santa Barbara (1999)

Appendix A: FBAC-LSM Policies for Popular Web Browsers

```
application lynx
{
    binarypaths /usr/bin/lynx:/usr/bin/X11/lynx;
    functionality Standard_Commandline_Application
        (peruser_directory="",
        peruser_files="",
        application_so_libraries_directory="",
        libraries_fileextension="",
        config_directory="",
        config_files={"/etc/lynx.cfg":"/etc/lynx.lss"},
        read_only_config_directory="");
    functionality Web_Browser
        (plugins_and_extensions_directory="",
        download_directory="/home/*/downloads/",
        allowed_hosts_to_connect_to="*",
        view_web_files_in_directory="/home/**/");
    functionality user_login_awareness ( );
    functionality requires_tmp_access ( );
}
```

```
application epiphany
{
    binarypaths /usr/bin/epiphany:/usr/bin/X11/epiphany;
    functionality Standard_Graphical_Application
        (peruser_directory="/home/*/.gnome2/epiphany/",
        peruser_files="/home/*/.gnome2/accels/epiphany",
        application_libraries_directory="/usr/lib/epiphany/",
        libraries_fileextension="*",
        config_directory="/home/*/.gnome2_private/",
        config_files={"/home/*/.mozilla/firefox/profiles.ini":
            "/home/*/.mozilla/firefox/*/prefs.js"},
        read_only_directory="/usr/share/epiphany/");
    functionality Web_Browser
        (plugins_and_extensions_directory="/usr/share/epiphany-extensions/",
        download_directory="/home/*/downloads/",
        allowed_hosts_to_connect_to="*",
        view_web_files_in_directory="/home/**/");
    functionality register_as_mozplugger_plugin ( );
}

application opera
{
    binarypaths /usr/bin/opera:/usr/bin/X11/opera;
    functionality Standard_Graphical_Application
        (peruser_directory="/home/*/.opera/",
        peruser_files="",
        application_libraries_directory="/usr/lib/opera/",
        libraries_fileextension="*",
        config_directory="/home/*/.kde/share/config/",
        config_files={"/etc/opera6rc":"/etc/opera6rc.fixed"},
        read_only_directory="/usr/share/opera/");
    functionality Web_Browser
        (plugins_and_extensions_directory={"/usr/lib/opera/plugins/":
        "/usr/lib/browser-plugins/":"/usr/lib/firefox/plugins/"},
        download_directory={"/home/*/OperaDownloads/":
        "/home/*/downloads/"},
        allowed_hosts_to_connect_to="*",
        view_web_files_in_directory="/home/**/");
    functionality Email_Client
        (mail_out_SMTP_servers="my.mail.server.com",
        SMTP_remote_port=<default>,
        mail_in_POP3_servers="*",
        POP3_remote_port=<default>,
        mail_in_IMAP_servers="*",
        IMAP_remote_port=<default>);
    functionality Irc_Chat_Client
        (chat_IRC_servers=<default>,
        IRC_remote_port=<default>);
    functionality News_Reader_Client (news_NNTP_servers=<default>);
    functionality BitTorrent_Client
        (bittorrent_peers_and_trackers="*",
        bittorrent_remote_port="18768");
}
```