A POLICY LANGUAGE FOR ABSTRACTION AND AUTOMATION IN APPLICATION-ORIENTED ACCESS CONTROLS

Z. Cliffe Schreuders, Christian Payne & Tanya McGill

School of Information Technology
Murdoch University
Murdoch, Western Australia
Programs behaving maliciously are a major security problem.

- Simple solutions are clumsy and inefficient.

- Complex solutions are, well... complex!

The Functionality-Based Access Control model aims to make these complex solutions more manageable and user-friendly.

This paper describes how design decisions in the policy language underpin these usability improvements.
Traditionally security models centre around what users can do.

Different security models were developed according to requirements, with appropriate policy schemes to support these.
- DAC
- MAC
- RBAC

In general, these models focused on separating differently privileged users from one another.
Assumed: the software always acts on behalf of the user.

Overwhelmingly these security models give each program all of the user’s privileges.

The user has little choice: they run the program with full privileges or not at all.
Vulnerabilities
- Software contains bugs or design problems that allow it to be attacked and subverted.

Malware
- Software is malicious by design.
- Distribution of software via the Internet exposes users almost continually.

Virus scanning and patching vulnerabilities is reactive: doesn’t address the underlying problem of excessive privilege.
One solution is to isolate applications from one another to limit the potential damage.
- E.g., sandboxing

Problems:
- Excessive redundancy: duplicated resources.
- Interchange of data is problematic:
  - Involves bypassing the isolation.
  - Inconvenient; interferes with natural workflow.
Restrict each application individually based on a set of rules specific to that program.
- Avoids problems with simple isolation-based approaches.

Early rule-based mechanisms include:
- TRON, Systrace, DTE.

Current Linux mechanisms:
- SELinux, AppArmor.
However, generating custom rules specifying low-level privileges for individual applications is not easy:

- Low-level privileges are difficult to understand.
- Many different programs run on a single system.
- Rule specifics vary from system to system.

Avoided in practice using learning modes:

- Program is run and resource requirements are monitored and recorded.
- This establishes a policy that then confines the program in the future.
Requires program behaviour during analysis to cover all privileges required.

In practice, program is unconfined when generating policy.
- If malicious, system may be already compromised!

User must verify a policy correctness:
- Requires expertise and patience.
- “Just click OK”.

Better if policy can be generated _a priori_.

Problems with learning modes
FBAC restricts programs based upon the functionality they are expected to provide.
- Far more usable than existing approaches.

Important features:
- Built around policy abstractions: functionalities.
- Functionalities are hierarchical: provide multiple layers of abstraction.
- Parameterisation: adaptable to specific circumstances.

Policies expressed in FBAC-PL.
Name-based resource descriptors.

Three types of policy files:
- **Confinements**: defines where policies are stored and who can manage them.
- **Functionalities**: building blocks for application policies.
- **Application policies**: security policy for individual programs.

FBAC-PL syntax is described in the paper.
The design of FBAC and FBAC-PL both aid usability.

Users are not required to actually use/interact with the policy language directly:
- Policy is constructed using a GUI policy manager.
The policy manager analyses the selected application and can suggest functionalities on which to build the policy.
- Uses specified dependencies and metadata.

FBAC-PL supports these automated suggestions by the various directives embedded in the functionality definitions.
Users adapt generalised functionalities to their specific requirements by supplying parameters.
functionality Irc_Chat_Cli{
    functionality_description "An irc (chat) client.";
    highlevel;
    category network_client;
    suggest_functionality iconcategory "IRCClient";
    suggest_functionality uses_library "python-irclib";

    parameter chat_IRC_servers "*";
    parameter_description "the remote servers the program can connect to to chat with IRC and send files with DCC";
    parameter_type IP;
    parameter_automate usedefault;

    parameter IRC_remote_port {"6665-6669":"7000":"194":"9-94"};
    parameter_description "the local chat (IRC) port. Usually 6667 or nearby (6665-6669) or rarely 194 or 994 (secure)";
    parameter_type port;
    parameter_automate usedefault;

    ...
    ...
}
application konversation
{
    executablepaths /opt/kde3/bin/konversation;
    functionality Standard_Graphical_Application
        (peruser_directory="/home/*/.kde/share/apps/konversation/",
        peruser_files="/home/*/.kde/share/config/konversationrc",
        application_libraries_directory="",
        libraries_fileextension=<default>,
        config_directory="/home/*/.kde/share/apps/konversation/",
        config_files="",
        read_only_directory="/opt/kde3/shar/apps/konversation/");
    functionality Irc_Chat_Client
        (chat_IRC_servers=<default>,
        IRC_remote_port=<default>,
        save_received_files_in_directory="/home/cliffe/downloads/",
        send_files_in_directory="");
    functionality Uses_Perl ();
}
Existing user-oriented access controls suffer from various problems.

FBAC model avoids usability weaknesses associated with rule-based approaches.

Security policy is defined in a hierarchical way, aiding abstraction.

Functionality policy building blocks aid automation to assist the user.

Users can easily adapt generated policies using parameterisation.

Allows end users to easily control what their applications do.