2012 IEEE 25th Computer Security Foundations Symposium

Gran: model checking grsecurity RBAC policies

Michele Bugliesi   Stefano Calzavara   Riccardo Focardi   Marco Squarcina
DAIS, Università Ca' Foscari
Venezia, Italy
{michele,calzavara,focardi,msquarci}@dais.unive.it

***Abstract*—Role-based Access Control (RBAC) is one of the most widespread security mechanisms in use today. Given the growing complexity of policy languages and access control systems, verifying that such systems enforce the desired invariants is recognized as a security problem of crucial importance. In the present paper, we develop a framework for the formal verification of `grsecurity`, an access control system developed on top of Unix/Linux systems.**

The verification problem in `grsecurity` presents much of the complexity of modern RBAC systems, due to the presence of policy state changes that may arise both from explicit administrative primitives supported by `grsecurity`, and as the result of the interaction with the underlying operating system facilities. We develop a formal semantics for `grsecurity`'s RBAC system, based on a labelled transition system, and a sound abstraction of that semantics providing a bounded approximation, amenable to model checking. We report on the result of the experimental analysis conducted with `gran`, the model checker we implemented based on our abstract semantics, on existing public servers running `grsecurity` to implement their RBAC systems.

I. Introduction

Role-based Access Control (RBAC) is one of the most widespread security mechanisms in use today, and has been the subject of extensive research for more than a decade now. The central idea in the RBAC model is to factor the assignment of access rights into two steps, separating the distribution of permissions to system-specific roles, from the assignment of users to roles, so as to simplify the overall access control management task [1].

Most of the research work on RBAC (see, e.g., [2], [3], [4]) has focused on policy verification, a problem of critical importance for system administrators, and a challenging one due to the complexity of the policies to be verified and to the state changes that arise in their management. State-change is not specific to RBAC: traditional access control frameworks such as those studied in the seminal work of [5] include rules to affect the structure of the access control matrix, defining the permissions granted to subjects on objects. Modern administrative RBAC (ARBAC) systems present similar features by providing system administrators with expressive languages for manipulating RBAC policies by reassigning users to roles and/or modifying the assignment of permissions to roles [1].

Policy verification in access control systems has traditionally been stated as a safety question, answered by means of a reachability analysis: for instance, user-role reachability in ARBAC systems formalizes the problem of determining whether, given an initial policy state, a target user and a role, there exists a sequence of state changes leading to a state in which the target user is impersonating that role. As it turns out, this kind of analysis is challenging, as the procedural nature of state change languages often creates subtle, undesired effects that are hard to anticipate without the aid of a tool for analysis.

Model checking [6] has emerged as a promising technique for automated policy verification [4], [7], [8]. The idea is exactly as in program verification, with the set of state-change rules playing the role of the program to be tested, and the reachability question as the property of interest: to counter the state explosion that often affect the analysis, making it unscalable to the point of making the problem intractable [8], researchers have advocated the usage of abstraction techniques [9].

In the present paper we continue along this line of research, focusing our attention on the formal verification of `grsecurity` [10], an access control system developed on top of Unix/Linux systems. `grsecurity` is deployed as a patch to the OS kernel that installs a reference monitor to mediate any access to the underlying OS resources; it supports the definition and dynamic enforcement of fine-grained access control policies to let users operate on objects (resources) via the subjects (executable files) provided by the underlying file system. Users are organized in roles, which are in turn structured as to identify a subset of privileged roles with higher capabilities on the system resources, and administrative control of the access control policies.

The verification problem in `grsecurity` presents much of the complexity of Administrative RBAC systems, due to the presence of policy state changes: these may arise either from explicit administrative actions for manipulating users and roles, as well as from the interaction between `grsecurity`'s access control and the facilities provided by the underlying operating system for setting user ids, hence dynamically changing users and associated roles by executing binaries operating in *setuid* mode [11]. This

Work partially supported by the RAS Project "*TESLA: Techniques for Enforcing Security in Languages and Applications*" and by the MIUR Project IPODS "*Interacting Processes in Open-ended Distributed Systems*".

DOI 10.1109/CSF.2012.29

dependency from state changes on the executable binaries of the underlying file system further complicates the model checking problem, as it causes the size of the search space to grow unbounded in the number of states and transitions.

We tackle the problem by resorting to an abstraction technique, by which the behavior resulting from the unbounded set of subjects available in the underlying file system is captured by the finite number of subjects that are listed in the security policy, which represents the input of the model checker. We prove the abstraction sound and complete, and employ it to carry out a reachability analysis on RBAC policies target at unveiling (potential) security leaks, leading to unintended accesses to sensitive resources.

The contribution of this paper may be summarized as follows:

- we develop a formal semantics for `grsecurity`'s RBAC system, based on a labeled transition system; besides providing the fundamental building block for our analysis, the LTS semantics has proved interesting in itself, as it made it possible to understand the subtleties of `grsecurity`'s RBAC rules, and to unveil a flaw arising from the interplay between the access control systems supported by Linux and `grsecurity`. As we discuss in Section III-D, this flaw makes it possible to unexpectedly bypass the imposed `grsecurity` capability restrictions when executing a setuid/setgid binary [12];
- we introduce an abstract semantics which provides a bounded, yet sound and complete, representation of the dynamic evolution of the `grsecurity` policy states arising in the formal semantics; based on that, we develop a framework for reachability analysis aimed at detecting the presence of access leaks in any given policy;
- we implement our framework in `gran`, a tool for the automatic analysis of `grsecurity` policies: the tool takes as input an RBAC policy, a user u, a set of initial states for u (associated with the possible subjects that may impersonate u) and a target file / object o, and checks whether there is a path of state changes leading to a state that grants u access to o;
- we provide a report of experiments we conducted with the analysis of policies in use on existing, commercial servers running `grsecurity` to implement their RBAC systems.

Structure of the paper: Section II reviews the basic concepts and notions behind `grsecurity`; Section III presents our formal semantics of the `grsecurity` RBAC system; Section IV describes the abstraction for the verification of `grsecurity` policies, and shows its formal correspondence to the previous semantics; Section V describes `gran` (`grsecurity` analyzer), a tool that automatically looks for security leaks in real `grsecurity` policies; Section VI illustrates `gran` at work on some case studies; Section VII discusses related work and Section VIII concludes the presentation with final remarks and a discussion of future work.

II. Background on grsecurity

`grsecurity` is a patch for the Linux kernel focused on security at the operating system level. It provides many different features on latest stable kernels, implementing a “detection, prevention, and containment” model [10]. In addition to the role-based access control (RBAC) system, which is the focus of this paper, `grsecurity` offers protection mechanisms against privilege escalation, malicious code execution and memory corruption; it also implements an advanced auditing system. `grsecurity` is typically adopted by hosting companies to harden web servers and systems providing services to locally logged users [13].

A. Grsecurity RBAC

`grsecurity` complements the standard discretionary access control (DAC) mechanism provided by Linux with a form of mandatory RBAC, providing an additional layer of protection. In the rest of the paper we identify `grsecurity` with its RBAC system.

The specification of the access control requirements is provided by a *policy*, whose structure is described in Section II-B. The policy defines the available roles, which can be of four different types. *User* roles are an abstraction of standard users in Linux systems, i.e., they provide a hook to extend the traditional DAC permission system with more sophisticated mechanisms, available only in `grsecurity`. *Group* roles provide a similar device for actual groups of the system. *Special* roles, instead, are not directly associated to traditional users and groups, and they are intended to provide extra privileges to normal accounts. A *default* role applies when no user, group, or special role can be granted. The mechanism of role assignment is discussed in Section III-C.

B. RBAC Policies

A policy defines the permissions given to each role for the different *objects* stored in the file system. A further level of granularity is introduced through the standard notion of *subject*, i.e., an abstraction of a process. Namely, permissions are not directly assigned to roles, since this would lead to a very coarse form of access control; rather, permissions are defined for pairs of the form role-subject. For instance, user `alice` could be granted read access to the object `/var` only through the subject `/bin/ls`.

Table I presents a snippet from a `grsecurity` policy. Even if it does not show all the features provided by `grsecurity` RBAC, it allows us to introduce the most important elements considered in our formalization. The policy defines a user role (flag '`u`') `alice`, which is permitted to impersonate the special role `professor`. Transitions to

Table I A snippet from a `grsecurity` policy

```
role alice u {
role_transitions professor
subject /  {
        /
        /bin                    x
        /boot                   h
        /dev                    h
        /dev/null               w
        /dev/pts                rw
        /dev/tty                rw
        /etc                    r
}

subject /bin/su {
user_transition_allow root
group_transition_allow root
        /                       h
        /bin                    h
        /bin/su                 x
        /dev/log                rw
}
```

specific users and groups of the underlying Linux system can be allowed or forbidden at the subject level, e.g., by the `user_transition_allow` attribute.

Permissions are specified in terms of access modalities for the objects in the policy. In this case, any process executed by `alice` is assigned the permissions defined for subject "/", except for process `/bin/su` which specifies its own set of modalities. In general, any process is accredited a set of access rights for any object in the system, according to a hierarchical matching mechanism. For instance, subject `/bin/su` inherits the rights on `/etc` by the less specific subject "/", while it overrides the permissions for `/bin` with its own. Similarly, accesses to object `/dev/log` by subject "/" are resolved in terms of the modalities listed for the less specific object `/dev`. Complete details on this mechanism are provided in Section III-B.

The modalities we consider mirror standard Linux permissions for reading '`r`', writing '`w`' and executing '`x`', plus a hiding mode '`h`'. Subjects are completely unaware of the presence of any hidden object, e.g., the process `/bin/ls` does not even list the directory `/boot` when it is launched by `alice`. We ignore other available modalities, which are either irrelevant for our setting (e.g., '`p`' for ptrace rejection) or identifiable with one of the previous modalities (e.g., '`a`' for appending).

C. User and group identifiers

Before digging into the internals of `grsecurity`, we need to briefly review how users and groups are identified in Linux systems. At the kernel level, users and groups are not distinguished by names, but by numbers. We refer to these numbers as user identifiers (UIDs) and group identifiers (GIDs) respectively. When a process is started, Linux assigns it a pair of identifiers, set to the UID of the invoking user. These identifiers are called the *effective* UID and the *real* UID of the process, respectively. The effective UID determines the privileges granted to the process and is employed, e.g., for standard DAC enforcement; the real UID, instead, affects the permissions for sending signals.

This apparently simple mechanism is complicated by an important subtlety related to the execution of particular binaries in the file system. Namely, any file f may be granted the "setuid" permission, with the following effect: when f is executed, the effective UID of the process is set to the UID of the *owner* of f, irrespective of the UID of the invoking user; the real UID, instead, is set to the UID of the caller. This allows for temporary acquisition of additional privileges to perform specific tasks.

We conclude by pointing out that changing to a particular UID is considered a sensitive operation in Linux systems and requires the process to possess the *capability* `CAP_SETUID`. Capabilities provide finer-grained distribution of privileges among processes since Linux 2.2. Remarkably, capabilities are bypassed when a "setuid" binary is executed, i.e., a process spawned by a "setuid" binary is *always* allowed to set its effective UID to the UID of the owner.

All the previous discussion applies similarly to GIDs.

III. A Formal Semantics for grsecurity

We propose a formal semantics for `grsecurity` in terms of a labelled transition system. We write $f : A \to B$ when f is a total function from A to B, while we use $f : A \mapsto B$ when f is partial. We let $f(a)\downarrow$ denote that f is defined on a. Let $f : A_1 \times \ldots \times A_n \mapsto B$ and let a_i range over A_i, for any $k \leq n$ we stipulate $f(a_1, \ldots, a_k)\downarrow$ if and only if $\exists a_{k+1}, \ldots, \exists a_n : f(a_1, \ldots, a_n)\downarrow$. Finally, we let $\mathcal{P}(A)$ stand for the power set of A.

A. Policies

We presuppose denumerable sets U of users and G of groups, ranged over by u and g respectively. We also let T denote the set of role types $\{\mathtt{u}, \mathtt{g}, \mathtt{s}\}$ ranged over by t; C denote the set of capabilities $\{\mathtt{set\_uid}, \mathtt{set\_gid}\}$ and M the set of access modalities $\{\mathtt{r}, \mathtt{w}, \mathtt{x}, \mathtt{h}\}$. A policy P is a 8-tuple:

$$P = (R, S, O, \mathit{perms}, \mathit{caps}, \mathit{role\_trans}, \mathit{usr\_trans}, \mathit{grp\_trans}),$$

where:

- R is a set of roles, ranged over by r. We let R_t denote the set of roles of type t and we assume that R_t and $R_{t'}$ are disjoint whenever $t \neq t'$;
- S is a set of subjects, ranged over by s, and O is a set of objects, ranged over by o. Both subjects and objects are pathnames, as we discuss below;
- $\mathit{perms} : R \times S \times O \mapsto \mathcal{P}(M)$ defines the permissions granted by the policy. Namely, if $m \in \mathit{perms}(r, s, o)$,

then subject s running on behalf of role r has permission m on object o;

- *caps* : $R \times S \mapsto \mathcal{P}(C)$ determines the capabilities allowed by the policy, i.e., if $c \in \mathit{caps}(r,s)$, then subject s running on behalf of role r can acquire capability c;
- *role_trans* : $R \to \mathcal{P}(R_s)$ defines which special roles can be impersonated by a given role;
- *usr_trans* : $R \times S \mapsto \mathcal{P}(U)$ defines which user identities can be assumed by a subject running on behalf of a given role;
- *grp_trans* : $R \times S \mapsto \mathcal{P}(G)$ defines which group identities can be assumed by a subject running on behalf of a given role.

We require a number of well-formedness constraints on policies which formalize a corresponding set of syntactic checks performed by `grsecurity`. Recall that we write $\mathit{perms}(r,s)\downarrow$ to denote $\exists o \in O : \mathit{perms}(r,s,o)\downarrow$.

1) $\forall r : \mathit{perms}(r, /)\downarrow$, i.e., all roles define at least the subject "/";
2) $\forall r, \forall s : (\mathit{perms}(r,s)\downarrow \Rightarrow \mathit{perms}(r,s,/)\downarrow)$, i.e., every subject in every role defines at least the object "/";
3) there exists a default role "$-$" such that $\forall t : - \notin R_t$.

Throughout the paper, most definitions (notably, the semantic rules in Tables II and III) and notation are to be understood as parametric with respect to a given policy. To ease readability, we do not make such dependency explicit, and just assume P as the underlying policy instead.

B. Pathnames and matching

Subjects and objects are collectively represented within `grsecurity` policies as pathnames, and these, in turn, are defined as sequences of "/"-separated names (or wildcards) as customary in Unix systems. For ease of presentation, we henceforth disregard wildcards and assume the following simplified structure of pathnames (that always presupposes a trailing "/"). Let n be a non-empty string non including "/", and let "$\cdot$" note string concatenation. Pathnames are defined by the following productions:

$$p ::= / \mid / \cdot n \cdot p$$

Pathnames are ordered according to the standard prefix order, so that p is smaller (more specific) than, or equal to, p' whenever p' is a prefix of p. Formally, the ordering relation, noted $\sqsubseteq$, is the smallest relation closed under the following rules:

$$\text{(P-TOP)} \quad p \sqsubseteq / \qquad\qquad \text{(P-PATH)} \quad \frac{p \sqsubseteq p'}{/ \cdot n \cdot p \sqsubseteq / \cdot n \cdot p'}$$

Clearly, $\sqsubseteq$ is a partial order: this ordering is paramount in `grsecurity`, as it constitutes the basic building block underlying the mechanisms for associating subjects to processes, and for checking access rights on objects. Specifically, when a process spawned by the execution of a file f running on behalf of a role r tries to access a file f', `grsecurity` matches f against the most specific subject s defined in role r such that $f \sqsubseteq s$. Similarly, f' is matched against the most specific object o, defined in subject s of role r, such that $f' \sqsubseteq o$. The permissions of o are then retrieved to evaluate whether the process can be granted access to f'. For instance, according to the policy in Table I, process `/bin/cat` is granted read access to `/etc/fstab`, since `/bin/cat` matches the subject "/" and `/etc/fstab` matches the object `/etc` defined there.

We formalize the matching relation as follows. For any set A of path names, we let $\mathit{min}(A)$ denote the minimum element of A according to the ordering $\sqsubseteq$, whenever such an element exists. Given a pathname p, we define the *matching subject* for p in role r as

$$\mathit{match\_subj}(p,r) = \mathit{min}(\{s \mid p \sqsubseteq s \land \mathit{perms}(r,s)\downarrow\}).$$

Analogously, we define the *matching object* for p in role r under subject s as

$$\mathit{match\_obj}(p,r,s) = \mathit{min}(\{o \mid p \sqsubseteq o \land \mathit{perms}(r,s,o)\downarrow\}).$$

Proposition 1 below and the assumption of well-formedness of the policy imply that $\mathit{match\_subj}(p,r)$ is always defined; instead, $\mathit{match\_obj}(p,r,s)$ is defined only if $\mathit{perms}(r,s)\downarrow$.

Proposition 1 (Chain Property). *If $p \sqsubseteq p'$ and $p \sqsubseteq p''$, then $p' \sqsubseteq p''$ or $p'' \sqsubseteq p'$.*

Proof: By induction on the sum of the depths of the derivations of $p \sqsubseteq p'$. Base case is $p \sqsubseteq / = p'$ which by (P-TOP) implies $p'' \sqsubseteq p'$. Inductive case is when $p \sqsubseteq p'$ since $p = / \cdot n \cdot \hat{p}$ and $p' = / \cdot n \cdot \hat{p}'$ with $\hat{p} \sqsubseteq \hat{p}'$. Now, if p'' is $/$ we trivially have $p' \sqsubseteq p''$. Otherwise, since $p \sqsubseteq p''$ by (P-PATH) it must be $p'' = / \cdot n \cdot \hat{p}''$ with $\hat{p} \sqsubseteq \hat{p}''$. By induction we have $\hat{p}' \sqsubseteq \hat{p}''$ or $\hat{p}'' \sqsubseteq \hat{p}'$ that, by applying (P-PATH), gives the thesis. ■

C. Role assignment

Each process in `grsecurity` has a role and a subject attached to it. The assignment of the subject to the process is performed by matching the name of the running file against the list of subjects of the current role, as discussed in Section III-B. Roles, instead, are assigned according to the hierarchy "special - user - group - default". Special roles are granted through authentication to the `gradm` utility and are intended to provide extra privileges to normal user accounts: as such, they have the highest priority. User roles, instead, are applied when a process either is executed by a user with a particular UID or changes to that UID. This is possible, since the name of every user role must match up with the name of an actual user in the system, i.e., there exists a bijective partial mapping from UIDs to user roles. It is worth noticing that only the *real* UID of the process is considered for role assignment. Group roles behave similarly to user roles, but they are applied to a given process only if no user role is associated to the process UID. The default role is chosen when no other role can be given.

A further remark is in order for role assignment: even though user roles are assigned by just looking at the real UID of the process, the presence of "setuid" binaries must be considered with care. We recall that a process spawned by a "setuid" binary sets its effective UID to the UID of the owner; however, *even unprivileged* (i.e., without the capability `CAP_SETUID`) processes can always set their real UID to their effective UID [11]. Binaries with the "setuid" permission set may then come into play during the role assignment process. As usual, similar considerations apply for "setgid" files.

D. Semantics

We assume an underlying file system, i.e., a subset of a denumerable set of pathnames F, ranged over by f. Let r_t range over $R_t \cup \{-\}$, a *state* is a 4-tuple $\sigma = \langle r_\mathrm{s}, u, g, f \rangle$ describing a process spawned by the execution of file f. The process may be impersonating a special role (when $r_\mathrm{s} \neq -$) and is running with real UID set to u and real GID set to g. We identify UIDs and GIDs with elements from a subset of U and from a subset of G, respectively. The role associated to σ is determined by the first three components of the tuple, according to the following function *role*:

$$\mathit{role}(r_\mathrm{s}, u, g) = \begin{cases} r_\mathrm{s} & \text{if } r_\mathrm{s} \in R_\mathrm{s} \\ u & \text{if } r_\mathrm{s} \notin R_\mathrm{s}, u \in R_\mathrm{u} \\ g & \text{if } r_\mathrm{s} \notin R_\mathrm{s}, u \notin R_\mathrm{u}, g \in R_\mathrm{g} \\ - & \text{otherwise} \end{cases}$$

The function formalizes the role assignment process, according to the hierarchy discussed before.

Attacker Model: Our semantics tracks all role transitions and subject changes allowed to a given process. The semantics depends on an underlying Linux system hosting `grsecurity`, characterized by a set of users, a set of groups and a file system, as it is apparent by the format of the states. However, we do not explicitly model any change to the previous sets and we just assume them to be denumerable; we can imagine to pick different sets after each transition, to account for the evolution of the system as a result of background operations. Intuitively, we consider a worst-case scenario, in which any possible action not conflicting with the RBAC policy is eventually performed by the process. Of course, the resulting LTS has an infinite number of states and transitions: this problem will be tackled in Section IV, where we will propose an abstract, finite-state semantics, specifically designed for automated security analysis.

Transitions: The transition rules are reported in Table II. Rule (SETR) accounts for login operations to special roles: such transitions must be allowed by the *role_trans* function. When $r'_\mathrm{s} = -$, the rule models a logout from a special role, which is always permitted. Rule (SETU) describes a change of the process UID, which must be allowed by the *usr_trans* function; moreover, the process must possess the capability `set_uid`, as we discussed in Section II-C. Notice that s is the matching subject for file f in the role $\hat{r}$ associated to the current state. Rule (SETG) details a similar behavior for changing the process GID. Finally, rule (EXEC) accounts for the execution of files and is the most interesting rule. The invoked file must indeed be executable and it must not be hidden, since hidden files are not visible to unauthorized processes. The execution of the file may lead to a role change, as we explained in Section III-C. Since we do not model which "setuid" and "setgid" binaries are actually present in the file system and we do not explicitly keep track of changes to file permissions, we simply assume that the execution of the file may trigger any user or group transition allowed by the policy for the current state. Of course, we also consider the possibility that the execution does not alter the identifiers of the process.

This subtle behavior when executing setuid/setgid programs was unknown before we started our formalization. In Section VI, we will illustrate that it is potentially harmful for security. This has also been reported to the main developer of `grsecurity`, who confirmed our findings. A fix has already been implemented in the latest stable release of `grsecurity` [12]. The solution consists in requiring the capabilities `CAP_SETUID`/`CAP_SETGID` to perform role transitions, even upon execution of setuid/setgid binaries.

IV. Verification of grsecurity Policies

While suitable for describing the operational behavior of `grsecurity`, the semantics presented in Section III is not amenable for security verification, as we discuss below. We thus propose a different semantics, designed for security analysis, which is an abstraction of the previous one, while being suitable to be model-checked. We also outline some properties of `grsecurity` policies which we consider interesting to verify and we formalize them in our framework.

A. An abstract semantics for grsecurity

The main problem with the presented semantics is that it hinges on many elements specific to the underlying Linux system hosting `grsecurity`, i.e., users, groups and files. Remarkably, all these elements are inherently dynamic, so any changes to them must be accounted for in the semantics to get a sound tool for security analysis. As a result, the corresponding LTS has infinite states and transitions, making security verification difficult to perform, inaccurate, or even infeasible. We thus design a simple *abstract* semantics for `grsecurity`, depending only on the content of the policy, which can be reasonably assumed to be static. If the policy happens to change during the lifetime of the hosting system,

Table II Semantics of `grsecurity`

(SetR)
$$\frac{\hat{r} = \mathit{role}(r_\mathrm{s}, u, g) \qquad r'_\mathrm{s} \in \mathit{role\_trans}(\hat{r}) \cup \{-\}}{\langle r_\mathrm{s}, u, g, f\rangle \xrightarrow{\text{set\_role}(r'_\mathrm{s})} \langle r'_\mathrm{s}, u, g, f\rangle}$$

(SetU)
$$\frac{\hat{r} = \mathit{role}(r_\mathrm{s}, u, g) \qquad s = \mathit{match\_subj}(f, \hat{r}) \qquad u' \in \mathit{usr\_trans}(\hat{r}, s) \qquad \texttt{set\_uid} \in \mathit{caps}(\hat{r}, s)}{\langle r_\mathrm{s}, u, g, f\rangle \xrightarrow{\text{set\_UID}(u')} \langle r_\mathrm{s}, u', g, f\rangle}$$

(SetG)
$$\frac{\hat{r} = \mathit{role}(r_\mathrm{s}, u, g) \qquad s = \mathit{match\_subj}(f, \hat{r}) \qquad g' \in \mathit{grp\_trans}(\hat{r}, s) \qquad \texttt{set\_gid} \in \mathit{caps}(\hat{r}, s)}{\langle r_\mathrm{s}, u, g, f\rangle \xrightarrow{\text{set\_GID}(g')} \langle r_\mathrm{s}, u, g', f\rangle}$$

(Exec)
$$\frac{\hat{r} = \mathit{role}(r_\mathrm{s}, u, g) \qquad s = \mathit{match\_subj}(f, \hat{r}) \qquad o = \mathit{match\_obj}(f', \hat{r}, s) \qquad \mathtt{x} \in \mathit{perms}(\hat{r}, s, o) \qquad \mathtt{h} \notin \mathit{perms}(\hat{r}, s, o) \qquad u' \in \mathit{usr\_trans}(\hat{r}, s) \cup \{u\} \qquad g' \in \mathit{grp\_trans}(\hat{r}, s) \cup \{g\}}{\langle r_\mathrm{s}, u, g, f\rangle \xrightarrow{\text{exec}(f')} \langle r_\mathrm{s}, u', g', f'\rangle}$$

we simply consider a different LTS and we perform again any relevant analysis.

We start from some simple observations. First, we note that users and groups are immaterial to `grsecurity`, as only the role assigned to a process is relevant for access control. Second, we observe that also the actual content of the file system is somewhat disposable, since all granted permissions are determined by finding out a matching subject or object. We thus define an abstract state as a 4-tuple $\sigma_a = \langle r_\mathrm{s}, r_\mathrm{u}, r_\mathrm{g}, s\rangle$ describing a process spawned by the execution of some file $f \sqsubseteq s$. The role assigned to the process is again determined by the first three components of the tuple and can be retrieved by overloading the type of the function *role* defined previously.

We first abstract from impersonation of user identities. The intuition here is that, in general, only a subset of the users has an associated user role, according to the definition of the policy, and all other users can be identified by `grsecurity` to the special identity "$-$". We thus define the abstraction of a user u, denoted by $[\![u]\!]$, as follows:

$$[\![u]\!] = \begin{cases} u & \text{if } u \in R_\mathrm{u} \\ - & \text{otherwise} \end{cases}$$

We define the abstract version of the *usr_trans* function, noted $[\![\mathit{usr\_trans}]\!]$, as the partial function with the same domain of *usr_trans* such that for, every r and s, we have:

$$[\![\mathit{usr\_trans}]\!](r, s) = \{[\![u]\!] \mid u \in \mathit{usr\_trans}(r, s)\}$$

In other words, transitions to users with no associated user role are collapsed to transitions to the special identity "$-$". We introduce analogous definitions also for groups and group transitions.

$$\begin{aligned} [\![g]\!] &= \begin{cases} g & \text{if } g \in R_\mathrm{g} \\ - & \text{otherwise} \end{cases} \\ [\![\mathit{grp\_trans}]\!](r, s) &= \{[\![g]\!] \mid g \in \mathit{grp\_trans}(r, s)\} \end{aligned}$$

We still need to address the most challenging task for the definition of the new semantics, i.e., the approximation of the behaviour of `grsecurity` upon file executions. The idea is to identify the executed file f with its matching object o: as a consequence of this abstraction, we can only find out an approximation for the subject to assign to the new process. This is done in terms of a set of possible matches, elaborating on the following observations:

- since o is the matching object for f, then f must be at least as specific as o ($f \sqsubseteq o$). Thus, we can take as an *upper bound* for the new subject the most specific subject which is no more specific than o, i.e., the subject $\mathit{min}(\{s' \mid o \sqsubseteq s'\})$. For instance, the execution of the file `/bin/ls`, matching the object `/bin/ls`, may lead to the impersonation of the subject `/bin` only if the more specific subject `/bin/ls` does not exist;
- since we do not know how much specific is f, every subject s' no more generic than o ($s' \sqsubseteq o$) may be a possible match. However, we can filter out all the subjects which would be associated to the execution of a more specific object o' which overrides o, i.e., we consider the set $\{s' \mid \mathit{match\_obj}(s', r, s) = o\}$, where r and s identify the current role and subject. For instance, the execution of a file in `/bin`, matching the object `/bin`, may lead to the impersonation of subject `/bin/ls` only if there does not exist the object `/bin/ls`. Indeed, when object `/bin/ls` exists, the execution of the file `/bin/ls` matches `/bin/ls` and not the less specific object `/bin`. Note that the file `/bin/ls` may even be non-executable, according to the policy specification for the object `/bin/ls`.

This reasoning leads to the following definition of *image* of an object o, given a role r and a subject s:

$$\begin{aligned} \mathit{img}(o, r, s) = \{&s' \mid \mathit{match\_obj}(s', r, s) = o\} \\ &\cup \{\mathit{min}(\{s' \mid o \sqsubseteq s'\})\} \end{aligned}$$

Again Proposition 1 and the well-formation of the policy imply that such a notion is always well-defined.

We finally present in Table III the reduction rules for the abstract semantics.

Rule (A-SetR) is identical to rule (SetR), while rule (A-SetU) is the counterpart of (SetU), abstracting from the users of the system. When $r'_{\mathrm{u}} = -$, the rule matches a transition to a user with no associated user role. Clearly, rule (A-SetG) behaves in the same way for group roles. Finally, rule (A-Exec) accounts for the execution of processes. Again, the choice of the new user and group role assumes a worst case scenario, in that every user and group transition which is allowed by the policy is taken into account by the rule. The new subject is drawn from the image of an executable object, according to the described approximation.

We conclude this subsection with two observations on the abstract semantics. First we note that, for any finite policy, the resulting LTS has a finite number of states and any state has a finite number of outgoing transitions, since both states and labels are built over finite sets. The LTS can then be effectively explored using standard techniques. We also underline our design choice to include states whose subject is not defined in the current role. Indeed, in our semantics we enforce an explicit match of the current subject against the subjects defined for the role. This choice leads to an increment of the size of the LTS, since we introduce a number of somewhat equivalent states; however, such a decision allows for a much more accurate security analysis, as we discuss in Section IV-C.

B. Correlating the two semantics

We now prove that the abstract semantics in Table III is a sound approximation of the concrete semantics in Table II, in that every transition in the concrete semantics has a corresponding transition in the abstract semantics.

Formally, we abstract a file f in terms of the most specific subject which is no more specific than f itself, i.e., we let $[\![f]\!] = \mathit{min}(\{s \mid f \sqsubseteq s\})$. We can now define the abstraction of a concrete state $\sigma = \langle r_{\mathrm{s}}, u, g, f\rangle$ as the abstract state $[\![\sigma]\!] = \langle r_{\mathrm{s}}, [\![u]\!], [\![g]\!], [\![f]\!]\rangle$.

Proposition 2 (Identity Preservation)**.** *The following equalities hold:*

(i) $\mathit{role}(r_s, u, g) = \mathit{role}(r_s, [\![u]\!], [\![g]\!])$;

(ii) $\mathit{match\_subj}(f, r) = \mathit{match\_subj}([\![f]\!], r)$.

Proof: For (i) we observe that $u = [\![u]\!]$ if $u \in R_{\mathrm{u}}$ and $g = [\![g]\!]$ if $g \in R_{\mathrm{g}}$. When, instead, $u \notin R_{\mathrm{u}}$ and $g \notin R_{\mathrm{g}}$ we have $\mathit{role}(r_{\mathrm{s}}, u, g) = \mathit{role}(r_{\mathrm{s}}, -, g) = \mathit{role}(r_{\mathrm{s}}, [\![u]\!], g)$ and $\mathit{role}(r_{\mathrm{s}}, u, g) = \mathit{role}(r_{\mathrm{s}}, u, -) = \mathit{role}(r_{\mathrm{s}}, u, [\![g]\!])$, giving the thesis. Item (ii) holds since $\{s \mid f \sqsubseteq s\} = \{s \mid [\![f]\!] \sqsubseteq s\}$, by definition of $[\![f]\!]$. ■

Lemma 1 (Abstract Execution)**.** *If $\mathit{match\_obj}(f, r, s) = o$, then $[\![f]\!] \in \mathit{img}(o, r, s)$.*

Proof: We first observe few, auxiliary properties. Let $p \sqsubseteq p'$, then one has:

(a) $[\![p]\!] \sqsubseteq p'$ or $p' \sqsubseteq [\![p]\!]$;
(b) $[\![p]\!] \sqsubseteq [\![p']\!]$;
(c) $\mathit{match\_obj}(p, r, s) \sqsubseteq \mathit{match\_obj}(p', r, s)$.

(a) follows directly by Proposition 1 from the observation that $p \sqsubseteq [\![p]\!]$; (b) and (c) follow immediately by noting that $\{\hat{p} \mid p' \sqsubseteq \hat{p}\} \subseteq \{\hat{p} \mid p \sqsubseteq \hat{p}\}$, by transitivity of $\sqsubseteq$.

We are now ready to prove the Lemma. We must show that either $\mathit{match\_obj}([\![f]\!], r, s) = o$ or $[\![f]\!] = \mathit{min}\{s' \mid o \sqsubseteq s'\} = [\![o]\!]$. Since $\mathit{match\_obj}(f, r, s) = o$, we have $f \sqsubseteq o$. Then, by (a) we can distinguish two cases, namely $[\![f]\!] \sqsubseteq o$ or $o \sqsubseteq [\![f]\!]$:

- let $[\![f]\!] \sqsubseteq o$ and let us assume by contradiction that $\mathit{match\_obj}([\![f]\!], r, s) \neq o$. Since $\mathit{match\_obj}(f, r, s) = o$, we have $\mathit{match\_obj}(o, r, s) = o$, which implies $\mathit{match\_obj}([\![f]\!], r, s) \sqsubset o$ by (c) and assumption $\mathit{match\_obj}([\![f]\!], r, s) \neq o$. Given that $f \sqsubseteq [\![f]\!]$, we then have $\mathit{match\_obj}(f, r, s) \sqsubseteq \mathit{match\_obj}([\![f]\!], r, s)$ by (c), which implies $\mathit{match\_obj}(f, r, s) \sqsubset o$ by transitivity, giving a contradiction;
- let $o \sqsubseteq [\![f]\!]$ and let us assume by contradiction that $[\![o]\!] \sqsubset [\![f]\!]$, i.e., $[\![o]\!] \sqsubseteq [\![f]\!]$ and $[\![o]\!] \neq [\![f]\!]$. Since $f \sqsubseteq o$, we have $[\![f]\!] \sqsubseteq [\![o]\!]$ by (b), thus we have $[\![f]\!] = [\![o]\!]$ by antisymmetry, giving a contradiction.

■

Theorem 1 (Soundness)**.** *If $\sigma \xrightarrow{\alpha} \sigma'$, then there exists a label β such that $[\![\sigma]\!] \xrightarrow{\beta}_a [\![\sigma']\!]$.*

Proof: By a case analysis on the rule applied to derive $\sigma \xrightarrow{\alpha} \sigma'$. If the rule is (SetR), the conclusion follows by the first item of Proposition 2. If the rule is (SetU) or (SetG), the conclusion relies on both items of Proposition 2 that imply that $\hat{r}$ and $\hat{s}$ in the abstract semantics are the same as $\hat{r}$ and s in the concrete semantics and consequently, $[\![u']\!] \in [\![\mathit{usr\_trans}]\!](\hat{r}, \hat{s})$ and $[\![g]\!] \in [\![\mathit{grp\_trans}]\!](\hat{r}, \hat{s})$. If the rule is (Exec), we conclude again by Proposition 2, in combination with Lemma 1 which additionally implies $[\![f']\!] \in \mathit{img}(o, \hat{r}, \hat{s})$. ■

Interestingly, our formalization enjoys also a completeness result, which states that every transition in the abstract semantics has a corresponding transition in the concrete semantics for some Linux system hosting `grsecurity`, as far as there exist at least one user and one group that do not have a corresponding role defined in the policy.

Lemma 2 (Concrete Execution)**.** *If $s' \in \mathit{img}(o, r, s)$ and $\mathit{perms}(r, s, o)\downarrow$, then there exists f such that $[\![f]\!] = s'$ and $\mathit{match\_obj}(f, r, s) = o$.*

Proof: Since $s' \in \mathit{img}(o, r, s)$, we can distinguish two cases. If $s' = [\![o]\!]$, we let $f = o$. Otherwise, if $s' \sqsubseteq o$ and $\mathit{match\_obj}(s', r, s) = o$, we let $f = s'$. ■

Table III Abstract semantics of `grsecurity`

(A-SetR)
$$\frac{\hat{r} = \mathit{role}(r_s, r_u, r_g) \qquad r'_s \in \mathit{role\_trans}(\hat{r}) \cup \{-\}}{\langle r_s, r_u, r_g, s\rangle \xrightarrow{\text{set\_spec}(r'_s)}_a \langle r'_s, r_u, r_g, s\rangle}$$

(A-SetU)
$$\frac{\hat{r} = \mathit{role}(r_s, r_u, r_g) \qquad \hat{s} = \mathit{match\_subj}(s, \hat{r}) \qquad r'_u \in [\![\mathit{usr\_trans}]\!](\hat{r}, \hat{s}) \qquad \texttt{set\_uid} \in \mathit{caps}(\hat{r}, \hat{s})}{\langle r_s, r_u, r_g, s\rangle \xrightarrow{\text{set\_user}(r'_u)}_a \langle r_s, r'_u, r_g, s\rangle}$$

(A-SetG)
$$\frac{\hat{r} = \mathit{role}(r_s, r_u, r_g) \qquad \hat{s} = \mathit{match\_subj}(s, \hat{r}) \qquad r'_g \in [\![\mathit{grp\_trans}]\!](\hat{r}, \hat{s}) \qquad \texttt{set\_gid} \in \mathit{caps}(\hat{r}, \hat{s})}{\langle r_s, r_u, r_g, s\rangle \xrightarrow{\text{set\_group}(r'_g)}_a \langle r_s, r_u, r'_g, s\rangle}$$

(A-Exec)
$$\frac{\hat{r} = \mathit{role}(r_s, r_u, r_g) \qquad \hat{s} = \mathit{match\_subj}(s, \hat{r}) \qquad \texttt{x} \in \mathit{perms}(\hat{r}, \hat{s}, o) \qquad \texttt{h} \notin \mathit{perms}(\hat{r}, \hat{s}, o) \qquad r'_u \in [\![\mathit{usr\_trans}]\!](\hat{r}, \hat{s}) \cup \{r_u\} \qquad r'_g \in [\![\mathit{grp\_trans}]\!](\hat{r}, \hat{s}) \cup \{r_g\} \qquad s' \in \mathit{img}(o, \hat{r}, \hat{s})}{\langle r_s, r_u, r_g, s\rangle \xrightarrow{\text{exec}(s')}_a \langle r_s, r'_u, r'_g, s'\rangle}$$

Theorem 2 (Completeness)**.** *Consider a policy such that $\exists u, g : u \notin R_u, g \notin R_g$. If $\sigma \xrightarrow{\beta}_a \sigma'$, then there exist a label α and two concrete states $\hat{\sigma}, \hat{\sigma}'$ such that $[\![\hat{\sigma}]\!] = \sigma$, $[\![\hat{\sigma}']\!] = \sigma'$ and $\hat{\sigma} \xrightarrow{\alpha} \hat{\sigma}'$.*

Proof: By a case analysis on the rule applied to derive $\sigma \xrightarrow{\beta}_a \sigma'$. For rules (A-SetR) (A-SetU) and (A-SetG) the concrete states are the same as the abstract ones apart from the special identity "–" that is mapped to the u or the g that we have assumed not to belong to R_u and R_g. We rely on Lemma 2 for finding a f in concrete rule (Exec) such that $[\![f]\!]$ is the same as s' in the abstract rule (A-Exec). ■

C. Security analysis

Policies in `grsecurity` are much more concise and readable than policies for other access control systems as, e.g., `SELinux` [14]. However, the plain syntactic structure of the policy does not expose a number of unintended harmful behaviors which can arise at runtime. Just to mention the simplest possible issue, the system administrator may want to prevent user `alice` from reading the files in `bob`'s home directory, but any permission set for role `alice` may be overlooked, whenever `alice` was somehow able to impersonate `bob` through a number of role transitions.

We now devise a simple formalism for verifying through our semantics if a policy is "secure". The usage of the inverted commas is intended to denote the intrinsic difficulty in answering such a question, due to the lack of an underlying *system* policy, stating the desiderata of the system administrator. General approaches to RBAC policies verification consider a declarative notion of error in terms of satisfiability of an arbitrary query [8]; more practical works, instead, are tailored around specific definitions of error, like the impersonation of undesired roles [9]. Here, we adopt the latter approach and we validate the policy with respect to some simple requirements on information access, which we consider desirable goals for realistic policies. This is a precise choice, since our research targets the development of a tool, `gran`, which should be effectively usable by system administrators. Of course, our semantics can easily fit different kind of analyses, possibly extending or generalizing those presented here.

The basic ingredient for verification consists in defining which permissions are effectively granted to a given state. Namely, we introduce two judgements $\sigma \vdash \mathsf{Read}(f)$ and $\sigma \vdash \mathsf{Write}(f)$ to denote that file f is readable (writeable) in state σ. The definition of such judgements arises as expected.

(L-Read)
$$\frac{\hat{r} = \mathit{role}(r_s, u, g) \qquad s = \mathit{match\_subj}(f, \hat{r}) \qquad o = \mathit{match\_obj}(f', \hat{r}, s) \qquad \texttt{r} \in \mathit{perms}(\hat{r}, s, o) \qquad \texttt{h} \notin \mathit{perms}(\hat{r}, s, o)}{\langle r_s, u, g, f\rangle \vdash \mathsf{Read}(f')}$$

(L-Write)
$$\frac{\hat{r} = \mathit{role}(r_s, u, g) \qquad s = \mathit{match\_subj}(f, \hat{r}) \qquad o = \mathit{match\_obj}(f', \hat{r}, s) \qquad \texttt{w} \in \mathit{perms}(\hat{r}, s, o) \qquad \texttt{h} \notin \mathit{perms}(\hat{r}, s, o)}{\langle r_s, u, g, f\rangle \vdash \mathsf{Write}(f')}$$

We assume to extend such rules to abstract states, in terms of the judgements $\sigma \vdash_a \mathsf{Read}(f)$ and $\sigma \vdash_a \mathsf{Write}(f)$.

Lemma 3 (Safety)**.** *Let $\mathcal{J}$ be either* $\mathsf{Read}(f)$ *or* $\mathsf{Write}(f)$*:*
(i) if $\sigma \vdash \mathcal{J}$, then $[\![\sigma]\!] \vdash_a \mathcal{J}$.
(ii) if $\sigma \vdash_a \mathcal{J}$, then there exists a concrete state σ' such that $[\![\sigma']\!] = \sigma$ and $\sigma' \vdash \mathcal{J}$.

Proof: This immediately follows by Proposition 2. ■

All the security analyses we propose below are based on the reachability of a state with given permissions. Lemma 3, in combination with Theorem 1, guarantees that the properties can be soundly validated on the abstract semantics; in combination with Theorem 2, instead, it ensures that any security violation found in the abstract semantics has a counterpart in some Linux system. Thus, verification turns out to be decidable and can be effectively performed, as we discuss in Section V. For readability, we state the analyses only for the concrete semantics.

Specification of the analyses: The first analysis we propose focuses on direct accesses to files, both for reading and for writing. In particular, we want to verify if a user u can eventually get read (write) access to a given file f. While easy to specify, we believe that such property fits the needs of many system administrators, since the operational behaviour of `grsecurity` is subtler than expected. The formal description of the property we consider is reminiscent of similar specifications through temporal logics for verification like CTL and LTL [15], [16]. Namely, we define two judgements $\sigma \vdash \mathsf{ERead}(f, \sigma')$ and $\sigma \vdash \mathsf{EWrite}(f, \sigma')$ to denote that file f can *eventually* be read (written) in state σ', starting from state σ.

(L-ERead)

$$\frac{\sigma \xrightarrow{\alpha_1} \dots \xrightarrow{\alpha_n} \sigma' \qquad \sigma' \vdash \mathsf{Read}(f)}{\sigma \vdash \mathsf{ERead}(f, \sigma')}$$

The rule for $\sigma \vdash \mathsf{EWrite}(f, \sigma')$ arises as expected. We just write $\sigma \vdash \mathsf{ERead}(f)$ and $\sigma \vdash \mathsf{EWrite}(f)$ when σ' is unimportant.

Given a user u, we denote with $\mathcal{S}(u)$ the set of the *initial states* of u. Any initial state for u has the form $\langle -, u, g, f \rangle$, where g is the primary group assigned to u by the underlying Linux system and f is a possible entry point for u. For instance, `/bin/bash` may be the standard entry point for users interfacing to the system through a "bash" shell. Here, we just assume to be given a set of initial entry points for any user and we defer the discussion on the definition of such sets to Section V. Note also that for initial states we are assuming that the user is not acting under any special role, since impersonation of such roles may happen only through authentication to the `gradm` utility, after a standard login operation to the Linux system.

Definition 1 (Eventual Read Access)**.** A user u can *eventually read* file f if and only if there exists $\sigma \in \mathcal{S}(u)$ such that $\sigma \vdash \mathsf{ERead}(f)$.

Eventual write access is defined accordingly.

We now build on our first analysis to specify a stronger property, inspired by the literature on information flow control [17]. We note, however, that in our setting we do not have any explicit notion of security label, so we focus on flows among different roles. Namely, if a user u_1 can read the content of file f and then write on an object o readable by u_2, then there exists a possible flow of information from u_1 to u_2 through o. This is an adaptation to our framework of the well-known "star-property" [18].

Definition 2 (Reading Flow)**.** There exists a *reading flow* on file f from user u_1 to user u_2 if and only if:

(i) there exists $\sigma \in \mathcal{S}(u_1)$ such that $\sigma \vdash \mathsf{ERead}(f, \sigma')$ and $\sigma' \vdash \mathsf{EWrite}(o)$ for some o;
(ii) there exists $\sigma'' \in \mathcal{S}(u_2)$ such that $\sigma'' \vdash \mathsf{ERead}(o)$.

Writing flows can be dually defined to address integrity issues. Again, this is just a reformulation into our setting of a standard property [19].

Definition 3 (Writing Flow)**.** There exists a *writing flow* on file f from user u_1 to user u_2 if and only if:

(i) there exists $\sigma \in \mathcal{S}(u_1)$ such that $\sigma \vdash \mathsf{EWrite}(o)$ for some o;
(ii) there exists $\sigma' \in \mathcal{S}(u_2)$ such that $\sigma' \vdash \mathsf{ERead}(o, \sigma'')$ and $\sigma'' \vdash \mathsf{EWrite}(f)$.

Note that both previous definitions ignore flows generated by multiple interacting users through a set of intermediate objects. While there is no technical difficulty in generalizing the definitions to such cases, we note that the current formulation already describes very strong properties.

The last analysis we consider accounts for a dangerous combination of permissions over the same object. Namely, if a user can acquire both permissions '`w`' and '`x`' on o, then o can be exploited for malicious code injection. `grsecurity` identifies this as an important problem, so it prevents the administrator from granting both said permissions for the same object; however, such a situation can arise at runtime, so we consider interesting to monitor it. We omit the formal specification of the analysis, much along the same lines of the previous proposals. We refer to Section VI for details on our experiments.

V. Gran: a Tool for Policy Verification

We present `gran`, a security analyser for `grsecurity` policies. The tool is written in Python and comprises around 1000 lines of code. At the time of writing, `gran` is still under active development; the source code for a beta release of `gran` can be downloaded at `http://github.com/secgroup/gran`.

Given a `grsecurity` policy, `gran` performs a pre-processing, which involves the expansion of the `include` and `replace` directives. These are just syntactic sugar, used to import fragment of other policies and to define macros, respectively. The tool then generates a model of the policy based on our formalization, i.e., it constructs a tuple $(R, S, O, \mathit{perms}, \mathit{caps}, \mathit{role\_trans}, \mathit{usr\_trans}, \mathit{grp\_trans})$.

Roles, subjects and objects are retrieved simply by parsing the policy specification. The generation of *perms* involves an unfolding of the pre-processed policy, to cope with the inheritance mechanism of `grsecurity`. We recall that, if a subject s does not specify any permission for object o, but a less specific subject defines an entry for it, then s inherits the same permissions for o. The only exception to this rule is when the subject specifies the "override" mode '`o`', which prevents this behaviour. Thus, the permissions stored in *perms* correspond to a properly unfolded version of those specified in the original policy.

Every capability is allowed by default, so for every role r and subject s we initially let $\mathit{caps}(r, s) = C$, then we remove any forbidden capability. Addition and revocation of capabilities is performed through the rules `+CAP_NAME` and `-CAP_NAME`, respectively. The overall result is order-sensitive, i.e., specifying first `+CAP_SETUID` and then `-CAP_SETUID` forbids the capability, while swapping the rules allows it. We also account for inheritance of capabilities among subjects defined in the same role.

Transitions to special roles are forbidden by default, so for every role r we initially let $\mathit{role\_trans}(r) = \emptyset$ and then we introduce in the set only the transitions explicitly allowed by the attribute `role_transitions`. Conversely, transitions to user roles are allowed by default, so we let $\mathit{usr\_trans}(r, s) = R_{\mathrm{u}} \cup \{-\}$ for any role r and subject s not providing any further specification. We recall that we abstract users with no associated user role by the distinguished identity "$-$". The attribute `user_transition_allow` can be used to restrict allowed user transitions to the ones specified. Conversely, the attribute `user_transition_deny` can be used to permit all users transitions except those listed. The two attributes cannot coexist. If subject s in role r specifies a set U of allowed user transitions, we let $\mathit{usr\_trans}(r, s) = \{[\![u]\!] \mid u \in U\}$. Conversely, if U is a set of denied user transitions, we let $\mathit{usr\_trans}(r, s) = (R_{\mathrm{u}} \setminus U) \cup \{-\}$. We apply a similar processing to construct the function *grp_trans*.

The tool disregards features that are not modeled, such as resource restrictions and socket policies. Domains, i.e., sets of user or group roles sharing a common set of permissions, are handled through unfolding as a set of user or group roles. Nested subjects are not supported, since the learning system of `grsecurity` does not account for them. In fact, `grsecurity` features the possibility to automatically generate a policy by inferring the right permissions from the standard usage of the system, to avoid burdening the user with the necessity of specifying all the details about access control. Since most users perform a full system learning and then tweak the generated policy around their own needs, we think nested subjects can be safely disregarded by our analysis.

After the parsing of the policy, `gran` generates all the possible states of the model and computes the set of the transitions. The tool implements all the analyses described in Section IV-C: the initial states and the sensible objects to consider for verification can be specified through commandline parameters. As a default choice, `gran` generates an initial state for each non-special role in the policy, assuming "/" as the subject entry point. If no target is specified for the analysis, `gran` infers a set of sensible resources by the specification provided in the configuration files of the learning system.

VI. Case Studies

We illustrate the outcome of practical experiments with `gran` and we give general considerations about possible vulnerabilities found by the tool.

A. *Verification of existing policies*

We asked the `grsecurity` community for policies to be verified using `gran`. Unfortunately, most system administrators are unwilling to provide their policies, since they can reveal a number of potentially harmful information about the system. However, we managed to gather a small set of real policies and we analyzed them with our tool. Due to privacy reasons, we cannot reveal any detail of such policies, so we report a properly sanitized outcome of the verification process. Our preliminary results were favorably welcome by the lead developer of `grsecurity`, who proposed us to integrate our tool in the `gradm` utility for policy management [20]. We consider this an important opportunity to continue our investigation on a larger scale, since users are for sure more comfortable to provide us the results of the validation rather than to disclose their policy.

We performed the verification of five different policies: the first and the second one from small web servers, the third one from a server running at our department, the fourth one generated by the learning system of `grsecurity`, and the fifth one from a large web server. In all cases, `gran` performed very effectively, providing the results of the analysis in less that one minute on a standard commercial machine. The output of the analysis was manually reviewed, looking for possible vulnerabilities: the process took from 10 to 30 minutes for each policy.

We start by reporting on direct accesses to sensitive information. In some cases, we noticed that critical files like `/etc/shadow` were readable by untrusted users. Even if this is not a vulnerability by itself, since the underlying DAC enforced by Linux does prevent this behavior, we believe that this is a poor specification at the very least. Indeed, system critical files on a hardened server are better be protected also by a MAC policy. Interestingly, a similar warning sometimes applies also for resources which are publicly readable, according to the default settings of Linux DAC, but are considered highly sensitive by the standard configuration files of the learning system of `grsecurity`. Examples of such resources include files as `/proc/slabinfo` and `/proc/kallsyms`, whose content may be potentially exploited by an attacker. We also noticed a dangerous specification in one of the analyzed policies: subject `/etc/cron.monthly` was provided almighty access to the system. This can have a tremendous impact on security, since cronjobs are usually executed with `root` privileges, thus mostly bypassing standard DAC. We argue that such a dangerous specification was provided for convenience, since scheduled jobs may need many different access rights and a careful assignment of permissions should feature very high granularity. Finally, we noticed that at least one of the users was not fully aware of the workings of the inheritance mechanism and, by manually tweaking the policy after the learning process, had created some unwanted cascade propagation of permissions.

We also performed some tests based on the other kinds of analyses described in Section IV-C. In particular, we noticed that unwanted writing accesses are much less frequent than undesired reading accesses: this is comforting and it was somehow expected, since the learning system of `grsecurity` tends to grant really few write permissions. The analysis also highlighted that usually only "physical" users, i.e., users with shell access to the system, have the opportunity to get both write and execution permissions over the same object, thus compromised services are unlikely to execute arbitrary code. Users, instead, probably need such permissions to effectively work on the system.

We think that the overall security of the analyzed `grsecurity` policies was fairly satisfying. We argue that much of the robustness, especially against undesired write accesses, comes from the sophisticated learning system of `grsecurity`, which tries to grant minimal privileges to each user. Indeed, the analyzed auto-generated policy turned out to be quite resilient to vulnerabilities; unfortunately, most administrators need to manually tweak the policy to get an usable system for their users and the overall impact of local changes may be easily overlooked. We think that our tool helps in getting the big picture on the security of the system.

B. *Exploits through "setuid" binaries*

The analysis presented in the previous section was performed using the "`-b`" option of `gran`, which discharges potential attacks due to the "setuid" flaw pointed out in Section III. We decided to make this choice, since a fix is already going to be merged in `grsecurity`, and in our worst-case scenario almost every object of the policy turns out to be potentially vulnerable. Precisely, the amended abstract semantics assumes that no role transition can be performed upon execution. Here, we discuss the impact of the flaw we found out, by describing a realistic scenario where it can be harmfully exploited by an attacker.

One of the goal of `grsecurity` is to try to drop many of the privileges normally granted to `root`, thus limiting the impact of many known vulnerabilities; however, during the learning process, some background operations may be overlooked, leading to undesired assignment of permissions. For instance, let us assume that an administrator starts full system learning to generate his own policy, when a scheduled cronjob performs an access to a sensitive resource: in this case the learning system could provide `root` with liberal access rights on the resource, since it would consider it as a normal system behaviour. If the administrator does not take care in manually strengthening the policy after the learning process, "setuid" binaries can lead to unintended impersonation of a powerful `root` role, bypassing the capability system. Indeed, even if the learning process tries to forbid as many capabilities as possible to user roles, such a practice does not offer the expected level of security, due to the subtle interplay between `grsecurity` and Linux.

Table IV A snippet of a flawed `grsecurity` policy

```
role root uG
role_transitions admin
...
subject /usr/sbin/cron o {
user_transition_allow alice
group_transition_allow users
        /                           h
        /usr                        h
        /usr/sbin/cron              rx
}

role alice u
...
subject / {
        /usr/bin
}
subject /usr/sbin/cron {
        /usr/bin                    rx
}
subject /usr/bin/python2.7 o {
        /                           h
        /tmp                        rw
        /home                       r
        /home/alice/bin             r
        -CAP_ALL
}

role bob u
subject / o {
        /                           h
        /bin                        x
        -CAP_ALL
}
subject /bin/bash {
        /tmp                        rw
        /home/bob                   rw
}
```

C. *Information leakage analysis*

We conclude our experiments by performing an information leakage analysis on a policy we generated for testing. We agree on the common statement that compartmentalization between users is a too strict property for many realistic systems; still, it can be interesting in some highly sensitive settings [18], [19]. Our sample policy is shipped with the `gran` package and a subset of it is depicted in Table IV.

When we process the policy with `gran`, we find out that user `alice` is able to share some confidential information in her home directory with her accomplice `bob` through a leakage on `/tmp`. The attack is mounted on top of `cron`, which our experiments seem to identify as a subtle subject. The output of `gran` looks as follows:

```
[!!] Indirect flow found for target
     /home/alice on object /tmp
     Traces for writing:
     [1] root:U:/usr/sbin/cron
         -set_UID(alice)->
         alice:U:/usr/sbin/cron
         -exec(/usr/bin)->
         alice:U:/usr/bin/python2.7
     Traces for reading:
     [1] bob:U:/
         -exec(/bin)->
         bob:U:/bin/bash
```

We assume that `alice` can schedule her tasks through `cron`. The daemon initially runs as `root`, changes its identity to `alice` and selects for execution a Python script in `/home/alice/bin`. The subject `/usr/bin/python2.7` defined for `alice` can read `/home/alice/bin` and write on `/tmp`. `bob` cannot directly read `/tmp`, but he can execute `/bin/bash` and get read access on `/tmp`. It is worth noticing that `alice` cannot directly execute Python to get write access on `/tmp`, since her default subject "`/`" does not allow execution of files under `/usr/bin`.

Our tool is then able to identify a subtle and unintended flow, which is unlikely to be noticed by just looking at the policy. We think that `gran` can help in strengthening the system against leakage of some particularly sensitive targets.

VII. Related Work

To the best of our knowledge, the present work is the first research paper focusing on the verification of `grsecurity` RBAC policies. However, there exists a huge literature on the analysis of (A)RBAC policies in general, mainly targeted to the isolation of restricted classes of policies whose verification is tractable.

Sasturkar et al. [2] show that role reachability is PSPACE-complete for ARBAC and identify restrictions on the policy language to partially tame this complexity; similar results are presented by Jha et al. in later work [8]. Li and Tripunitara [21] perform a security analysis on restricted ARBAC fragments and identify a specific class of queries which can be answered efficiently. Stoller et al. [22] isolate subsets of policies of practical interest and develop algorithms to analyze them; their techniques are implemented in the RBAC-PAT tool [4], which supports also information flow analysis much in the spirit of the one provided by `gran`. Jayaraman et al. [9] propose Mohawk, a model-checker implementing an abstraction-refinement technique aimed to error finding in complex ARBAC policies.

Contrary to all these works, our framework is not targeted to the analysis of generic ARBAC policies, but of real, full-fledged `grsecurity` RBAC policies, which turn out to be amenable for efficient static verification. A formal comparison with previous work, however, might be useful to understand how possible extensions of `grsecurity` would impact on the complexity of the analysis. We leave this as future work.

VIII. Conclusion

We have presented a framework for a formal, automated analysis of `grsecurity`'s RBAC system, to help system administrators validate and maintain their policies. As we have illustrated, our endeavor has proved useful, as `gran` has unveiled a series of ambiguities and unexpected behaviors that have been reported to the main developer, confirmed and fixed. The preliminary results we have illustrated have been well-received by the developer, who have requested to integrate the tool in the `grsecurity` distribution.

There are a number of further issues that are part of our plans for future work, some of which we discuss below. First, it would be worthwhile to integrate `gran` with information from the actual file-system, to make the analysis more precise and prune attack traces prevented by the underlying Linux DAC system. For example, the setuid/setgid binaries might be inspected in order to know what transitions to different users are actually admissible when executing a given object. This integration would point out what is the actual Trusted Computing Base (TCB) of the whole system, i.e., the minimal set of trusted components so that the combination of RBAC and DAC systems is enough to prevent security flaws.

Moreover, it would be interesting to integrate `gran` with existing model checkers. This could be done by implementing a back-end module for the generation of the model checker input after the policy parsing. As a result, `gran` would become amenable for additional formal reasoning, while, at the same time, be still able to perform a concrete analysis of real systems.

A further desirable extension to `gran` involves integrating the tool with the additional components of the `grsecurity`'s RBAC system (notably, the network component). While we believe the tool should scale well without significant performance issues, such integration will certainly constitute a non-trivial engineering task.

References

[1] R. S. Sandhu, V. Bhamidipati, and Q. Munawer, "The arbac97 model for role-based administration of roles," *ACM Trans. Inf. Syst. Secur.*, vol. 2, no. 1, pp. 105–135, 1999.

[2] A. Sasturkar, P. Yang, S. D. Stoller, and C. R. Ramakrishnan, "Policy analysis for administrative role based access control," in *CSFW*. IEEE Computer Society, 2006, pp. 124–138.

[3] A. Armando and S. Ranise, "Automated symbolic analysis of arbac-policies," in *STM*, ser. Lecture Notes in Computer Science, J. Cuéllar, J. Lopez, G. Barthe, and A. Pretschner, Eds., vol. 6710. Springer, 2010, pp. 17–34.

[4] M. I. Gofman, R. Luo, A. C. Solomon, Y. Zhang, P. Yang, and S. D. Stoller, "Rbac-pat: A policy analysis tool for role based access control," in *TACAS*, ser. Lecture Notes in Computer Science, S. Kowalewski and A. Philippou, Eds., vol. 5505. Springer, 2009, pp. 46–49.

[5] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Commun. ACM*, vol. 19, no. 8, pp. 461–471, 1976.

[6] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.

[7] N. Zhang, M. Ryan, and D. P. Guelev, "Synthesising verified access control systems through model checking," *Journal of Computer Security*, vol. 16, no. 1, pp. 1–61, 2008.

[8] S. Jha, N. Li, M. V. Tripunitara, Q. Wang, and W. H. Winsborough, "Towards formal verification of role-based access control policies," *IEEE Trans. Dependable Sec. Comput.*, vol. 5, no. 4, pp. 242–255, 2008.

[9] K. Jayaraman, V. Ganesh, M. V. Tripunitara, M. C. Rinard, and S. J. Chapin, "Automatic error finding in access-control policies," in *ACM Conference on Computer and Communications Security*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 163–174.

[10] B. Spengler, "Increasing performance and granularity in role-based access control systems," 2004. [Online]. Available: http://grsecurity.net/researchpaper.pdf

[11] "man page for function `setreuid`." [Online]. Available: http://linux.die.net/man/2/setreuid

[12] B. Spengler, "Changelog of `grsecurity`," February 2012, commit 3981059c35e8463002517935c28f3d74b8e3703c. [Online]. Available: http://grsecurity.net/changelog-stable2.txt

[13] "Sponsor page of `grsecurity`." [Online]. Available: http://grsecurity.net/sponsors.php

[14] M. Fox, J. Giordano, L. Stotler, and A. Thomas, "SELinux and grsecurity: A case study comparing linux security kernel enhancements," University of Virginia. [Online]. Available: http://www.cs.virginia.edu/jcg8f/GrsecuritySELinuxCaseStudy.pdf

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.

[16] A. Pnueli, "The temporal logic of programs," in *FOCS*, 1977, pp. 46–57.

[17] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, 2003.

[18] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," MITRE Corporation, Tech. Rep., 1973.

[19] K. J. Biba, "Integrity Considerations for Secure Computer Systems," USAF Electronic Systems Division, Tech. Rep., 1977.

[20] B. Spengler, "Private communication," February 2012.

[21] N. Li and M. V. Tripunitara, "Security analysis in role-based access control," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 4, pp. 391–420, 2006.

[22] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman, "Efficient policy analysis for administrative role based access control," in *ACM Conference on Computer and Communications Security*, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds. ACM, 2007, pp. 445–455.