Identifying and resolving least privilege violations in software architectures

Koen Buyens
Riccardo Scandariato
Wouter Joosen

Report CW 575, January 2010

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Identifying and resolving least privilege violations in software architectures

Koen Buyens
Riccardo Scandariato
Wouter Joosen

Report CW 575, January 2010

Department of Computer Science, K.U.Leuven

Abstract

Supporting security principles, like least privilege, in a software architecture is difficult due to the lack of both a sound theory and effective secure software engineering practices. As a result, principles are often neglected by practitioners, resulting in potentially high risk threats to systems.

This paper improves the understanding and the support for least privilege in software architectures by (i) defining the foundations to identify potential violations of the principle herein and (ii) providing architectural transformations that can be used to improve the security properties of an architecture, in an automated way. These results have been implemented and validated in a case study.

Automated detection and resolution of least privilege violations in software architectures.

Koen Buyens Riccardo Scandariato Wouter Joosen
IBBT-Distrinet
Celestijnenlaan 200A
3001 Heverlee, Belgium
first.last@cs.kuleuven.be

ABSTRACT

Supporting security principles, like least privilege, in a software architecture is difficult due to the lack of both a sound theory and effective secure software engineering practices. As a result, principles are often neglected by practitioners, resulting in potentially high risk threats to systems.

This paper improves the understanding and the support for least privilege in software architectures by (i) defining the foundations to identify potential violations of the principle herein and (ii) providing architectural transformations that can be used to improve the security properties of an architecture, in an automated way. These results have been implemented and validated in a case study.

1. INTRODUCTION

Security design principles like least privilege and complete mediation have survived the test of time since they have been introduced by the seminal work of Saltzer and Schroeder [14]. Their value in secure engineering processes is now widely acknowledged, e.g., in Microsoft's Security Development Lifecycle. However, the implementation of these principles in a software design by concrete means is often difficult due to the lack of both a sound theory and effective secure software engineering practices. The above-mentioned lack of support is particularly severe at the level of architectural design as highlighted in previous studies [5]. Therefore, ways of expressing and reasoning about security principles in software architectures are needed [16]. Due to the key role played by the architecture in the development process, failing to support sound security principles at this level could jeopardize the software project and can result in severe costs to fix vulnerabilities afterwards.

The main goal of this paper is to improve the support for the least privilege (LP) security principle in software architectures by providing a formal, yet operational, definition of LP violations and by defining architectural transformations that positively impact the principle related properties of the architecture, while preserving the semantics thereof. In other words, the approach presented in this paper uses an architectural description to identify violations, which are then solved by means architectural transformations. The architecture is expected to be documented (e.g., in UML) via a logical view (namely the structure of the system in terms of components and connectors), a process view, and the interaction scenarios. This minimal documentation set can be safely assumed to be available in all software projects where an explicit architectural design effort has been carried out. In the presented approach, a so-called Task Execution Model is automatically derived from the above documentation and thence used for the formal analysis. Identified violations are then automatically resolved by means of architectural re-factoring rules and the original architectural description is adjusted accordingly. The entire process is supported by tools (See Figure 1). Currently, research like Privtrans [2] enforces the LP principle at code level by automatically compartmentalizing the code into different processes with fewer permissions. This paper argues to apply the same concept at architectural level, by means of automated re-factoring.

The contribution of this work is a mature and comprehensive definition of what makes a violation of LP at the architectural level. This definition fixes a number of shortcomings of our previous work [3, 4], namely a model of delegation has been incorporated, less constrained transformations have been introduced, and support for different LP violations solution strategies has been added. Additionally, these improvements have been implemented in a tool.

This paper is structured as follows. Section 2 provides a general overview of our approach. Section 3 introduces a running example that is used to illustrate our models. Sections 4 and 5 present the formal foundations that are used to formally define LP violations in section 6. Next, these violations can be solved by the transformations presented in section 7. Then, section 8 applies the approach on a case study. Finally, section 9 compares the approach to the related work and section 10 presents the concluding remarks.

2. APPROACH

The principle of LP prescribes that a principal (i.e., a user or a process executing on behalf of a user) is only granted the minimum set of permissions necessary to complete its tasks, and consequently it is isolated from the tasks it is not allowed to execute. A *task* is generally comprised by a sequence of *actions*. In this context, a *permission* represents the right for a principal to execute a part of a task. The correct enforcement of the LP principle prevents popular vulnerabilities related to elevation of privilege and task interference.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$10.00.

The former are vulnerabilities that can be exploited by an attacker to gain access to resources which normally would have been protected from access by that attacker. The latter are vulnerabilities that can be exploited by an attacker to change the expected outcome of a task.

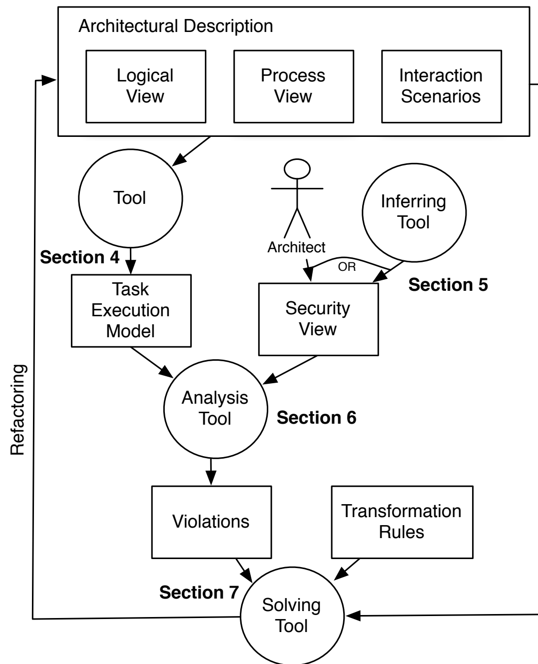


Figure 1: Overview of the proposed approach.

TODO RIC

3. RUNNING EXAMPLE

In the following of this paper, the approach summarized in the previous section will be illustrated via an example. In particular, consider the bug tracking system documented in Figure 2. The system is used to assign bug-fixing jobs to programmers of the maintenance team of a software product. Bug reports can be submitted by field users and they get

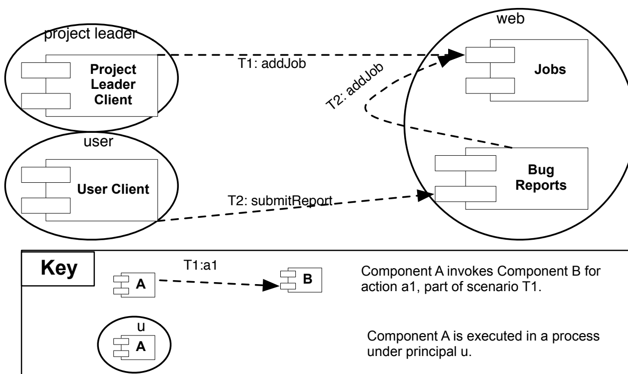


Figure 2: Excerpt from the software architecture of a bug tracking system.

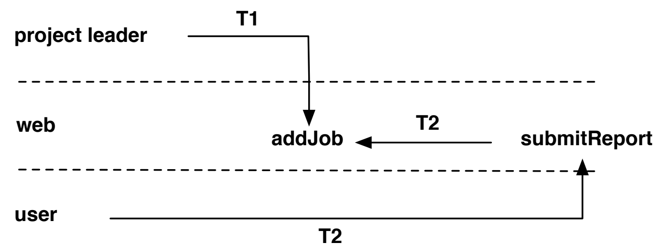


Figure 3: Excerpt from the Task Execution Model of a bug tracking system.

automatically assigned by the system to a programmer in the maintenance team. Bugs are also discovered by the in-house testing team and, in this case, they get assigned to programmers by the project leader.

Due to space limitation, three views (logical, processes, scenarios) are overlayed in one picture. The logical view (see the components and note that connectors are omitted) decomposes the system into a back-end (components at the right-hand side) and two front-end components. The `Jobs` back-end component is responsible for managing and assigning jobs to users, while the `Bug Reports` back-end component is responsible for processing submitted bug reports. To this aim, their interface offer the *addJob* and *submitReport* operations (i.e., actions) respectively. The functionality offered by these back-end components is exposed to different type of users via two dedicated front-end components, namely the `User Client` for the end users of the product and the `Project Leader Client` for the product manager.

The process view (see the ovals) documents that each front-end client is run in its own process by the principal using it, i.e. the `User Client` by the *user* and the `Project Leader Client` by the *project leader*. The back-end components run as a different process under the *web* server principal (i.e., the back-end is a web application).

The interaction scenarios (see the arrows) show that the system supports the following two tasks. Bug fixing jobs can be assigned to the development team by the project leader via the *addJob* action. This is labeled as task T1. Bug reports are submitted by the user via the *submitReport* action. As a consequence, a new job must be created for the testing team to verify the correctness and severity of the submitted bug report. The combination of these two actions is labeled as task T2.

4. TASK EXECUTION MODEL

In an architectural documentation, the *logical view* typically decomposes the system into a set of key abstractions, also called components. Every *component* can be described in terms of the *actions*[1] of its interfaces. These actions are used to interact with the component, often to realize a task (use case). The *process view* specifies which runtime element (thread, process, etc.) executes each action identified in the logical view. Hence, it maps runtime elements onto components. The view also relates these runtime elements to the principals that execute them. These principals can be end users (e.g. John) or system accounts (e.g. web server).

[1]The term action is used instead of operation, since this fits well with the security-related part of the model.

The *scenarios* show how the above views work together by instantiating tasks (use cases) onto these views. Essentially, these tasks are a sequence of actions tuples. The action tuple (α,β) indicates that the execution of α depends on the execution of β.

To effectively support the analysis procedure that identifies the LP violations, a model is necessary that relates the architectural concepts (e.g., components, interface operations, processes, and so on) to LP concepts (e.g., principals, tasks, delegation relationships, and so on). This is the Task Execution Model, which represents a LP-oriented view of the software architecture. The Task Execution Model combines the three above-mentioned views to display how tasks roll out over the different execution threads of existing processes (and corresponding principals). This view is partially inspired by business process notations like the Business Process Modeling Notation (BPMN), where process activities are mapped to responsible roles, which are represented as swimlanes. From a Task Execution Model, it is easy to identify the actions (parts of tasks) the principal completes itself, the actions it delegates to other principals, and the dependencies between the actions it executes itself. An example is provided in Figure 3, which depicts an excerpt of the Task Execution Model for the bug tracking system.

In the example, the process view and the logical view are used to derive that the *web* principal executes the *addJob* and the *submitReport* actions. Indeed, both actions are offered by the `Jobs` and `BugReports` components that run in the processes executed by *web*. The scenarios are used to derive that the *project leader* delegates task T1 to the *web* principal via the *addjob* action, while the *user* delegates task T2 to the *web* principal via the *submitReport* action.

In the general case, the tool-based derivation goes as follows. The actions each principal must execute are derived from the three views. In short, a principal must execute an action if it is offered by a component that contains the logic it executes and is used by a task. More concretely, take the following steps. First, determine the actions each principal can execute by combining the principal-process, the process-component, and the component-action relations of the process view and the logical view. Finally, determine the actions each principal must execute by selecting the tuples of the derived principal-action relation for which the actions appear (as first element in a tuple) in a scenario. This relation is called mustexecute.

The actions a principal must delegate are derived from the mustexecute relation and the action tuples of the scenarios. In short, a principal delegates an action α if there is a dependency between an action β it must execute itself and action α is executed by another principal. More concretely, take the following steps. First, identify the dependencies of the actions a principal must execute by expanding the (principal,action) tuples of the mustexecute relation to (principal,action,action) tuples by joining them with the (action,action) tuples of the scenarios. Next, determine the actions a principal must delegate by selecting the triples for which the second action does not appear as first action in the triples that have the same principal. Finally, shrink these triples to (principal,action) tuples by removing the second element. The actions of the remaining tuples are the set of actions the associated principal must delegate.

Dependencies between actions a principal must execute are the tuples of the scenarios for which both elements appear in the must execute set of that principal.

5. SECURITY VIEW

The security view attaches permissions to the actions defined in the logical view. A *permission* represents the right for a principal to invoke a set of interface actions. From a notation perspective, this view can be represented by a secure architectural description language, like secure xADL [13], and can be formalized as follows.

`actions(s) =` $a_1,\ a_2,\ \dots$
`security(s) =` $\langle\ actionPerm(s),\ principalPerms(s)\ \rangle$
`actionPerm(s) =` $perm_1,\ \dots,\ perm_w$
$perm\ =\ \{\ a_1,\ \dots,\ a_v\ \}\ \subseteq\ actions(s)$
`principalPerms(s) =` $\{\ (pi_i, pe_j),\ \dots,\ (pi_{i+f}, pe_{j+g})\ \}$
  $\subseteq$ `principals(s)` $\times$ `actionPerm(s)`
`processPrincipal(s) =` $\{\ (pr_i, pi_j),\ \dots,\ (pr_{i+f}, pi_{j+g})\ \}$
  $\subseteq$ `processes(s)` $\times$ `principals(s)`

For instance, in our running example, the security view could say that executing the *addJob* and *submitReport* actions require a single permission π_1. Therefore, all the principals require π_1 to completes their tasks successfully.

5.1 Inferring permission assignment

In general, the permissions should be documented and available to the analysis tool. In the real world, the architect (or the security specialist) provides the principals with an initial set of permissions based on experience, the used deployment platform, and the environment. However, if the permission assignment is not explicitly documented, this work proposes an automated procedure to assign permissions in a sensible way, based on the logical view and Task Execution Model. The tool employs several heuristics, among which:

- a principal is granted the necessary permissions to execute all the actions of all components within its own process(es).
- a principal is also granted the permissions that it requires to complete (parts of) tasks it is responsible for, i.e., the permissions associated with the actions in its mustDelegate set.

6. LP ANALYSIS

This Section introduces the formal definition of an architectural violation of the LP principle. Then, several types of violation are showcased by means of the formal model.

6.1 Model for LP violation

This section further defines the LP property by using the Task Execution Model and the security view. First, we introduce the following definitions which are used later on to define LP. A principal *can call* an action if it has the permission to invoke that action. A principal is *direct responsible* for an action if it *must execute* (invoke) that action to complete a task or if it *must delegate* the execution of that action to another principal (1 hop) who completes a part of the task. A principal is *indirectly responsible* for an action if that action appears in a task delegation chain after an action that the principal has delegated.

`canCall(principal, action) = true` $\Leftrightarrow$
  $\exists\ (principal,\ permission)\ \in\ principalPerms(s)\ \mid$

$action \in permission$
`directresponsible(principal, action) =`
`mustExecute(principal, action) or`
`mustDelegate(principal, action)`
`mustExecute(principal, action) = true` $\Leftrightarrow$
$\exists\ task \in tasks(s)$ and
$\exists\ a \in actionsPrincipal(principal)$
and $\exists\ b \in actions(s) |$
$a = action$ and $(a, b) \in actions(task)$
`mustDelegate(principal, action) = true` $\Leftrightarrow$
$\exists\ task \in tasks(s)$ and
$\exists\ a \in actionsPrincipal(principal)$
and $\exists\ b \in actions(s) |\ b = action$
and $b \notin actionsPrincipal(principal)$
and $(a, b) \in actions(task)$
`actionsPrincipal(principal) =` $\{\ a\ |$
$a \in actions(s)$
and $\exists\ (po, principal) \in processPrincipal(s)$
and $\exists\ (po, c) \in processComponent(s)$
and $a \in actions(c)\ \}$
`indirectresponsible(principal, action) = true` $\Leftrightarrow$
$\exists\ task \in tasks(s)$ and
$\exists\ principal2 \in principals(s)$ and
$\exists\ a, b, c \in actions(s)\ |$
$(a, b) \in actions(task)$ and $c \in after(task, b)$ and
$mustDelegate(principal, a)$ and
$mustExecute(principal2, a)$ and
$principal \neq principal2$ and $a \neq b$ and
$action = c$

We now come to the definition of LP. The main question is: given an architectural description, has a (process running as) principal A too many assigned permissions? In other words, if a principal A wants to access an action offered by another principal B, is access allowed, even if it should not?

A system adheres to LP if all its principals adhere to LP. A principal does not adhere to LP if there exists an action that it *can call* that is not in the set of actions it minimally requires permissions for to finish its tasks (difference between his sets of *direct responsible* and *indirect responsible* actions). Figure 4 illustrates this definition graphically.

The tool based identification goes as follows. First, find the minimal set of actions $\mathcal{EP}_p$ for a principal p. Start from the set of all actions of the system s (actions(s)). Identify the actions in this set the principal is directly responsible for (labeled as the set $\mathcal{D}_p$), and the actions this principal is indirectly responsible for (labeled as the set $\mathcal{I}_p$). Subtract both sets to identify the minimal set of actions for the principal. Next, identify the set of violating actions for a principal p $\mathcal{MP}_p$. Start from the set of all actions of the system (actions(s)). Identify the actions in this set that the principal can call (labeled as $\mathcal{C}_p$). Subtract from this set the minimal set $\mathcal{EP}_p$ to identify the set of violations for the principal. If this set is not empty, then there is a violation for LP.

$p \in principals(s)$
$\mathcal{I}_p = \{\ a\ |\ a \in actions(s)$ and $indirectresponsible(p, a)\ \}$
$\mathcal{D}_p = \{\ a\ |\ a \in actions(s)$ and $directresponsible(p, a)\ \}$
$\mathcal{C}_p = \{\ a\ |\ a \in actions(s)$ and $canCall(p, a)\ \}$
$\mathcal{MP}_p = \mathcal{D}_p \setminus \mathcal{I}_p$
$\mathcal{EP}_p = \mathcal{C}_p \setminus \mathcal{MP}_p$
`violatesLP(p)` $\Leftrightarrow \mathcal{EP}_p \neq \emptyset$
`violation(a)` $\Leftrightarrow a \in \mathcal{EP}_p$

6.2 Potential violations

The previous identified violations are real violations, but there also exist potential violations. These violations are actions whose result a principal influences without having to execute or delegate them. Indeed, one can argue that influencing the results of actions is a weak form of having the permissions to execute them. These violations are potential because there is typically not enough information available at architectural level to determine whether they indeed influence the results, but they *can* become a violation if their implementation is done in an inappropriate way. We distinguish between two type of potential violations.

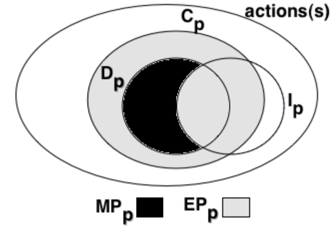


Figure 4: Definition of LP for principal p. Actions in EP (extra perms) are violations, while actions in MP (minimal perms) are minimal necessary.

First, if at least two principals delegate the execution of a task to a third principal and both use the same action offered by that third principal, than each delegating principal might influence the results of the other delegating principal. That action is considered a *potential must delegate violation*.

Second, if a principal can influence the results of an action offered by a second principal via shared state between that action and an action the principal can call, then the former is considered a *potential influence violation* of that principal.

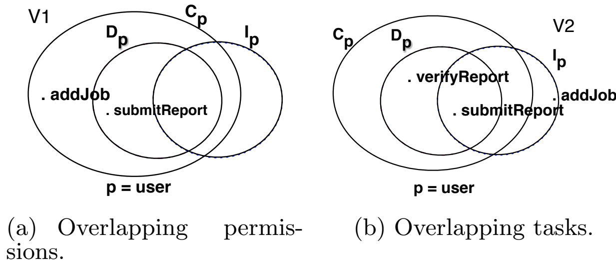
$\mathcal{ME}_p = \{\ a \in actions(s) |\ mustExecute(p, a)\ \}$
$\mathcal{MD}_p = \{\ a \in actions(s) |\ mustDelegate(p, a)\ \}$
$\texttt{LPPotentialMustDelegate}_p =\ a \in MD_p\ |$
$\exists\ p2 \in principals(s)\ and\ a \in MD_{p2}$
$\texttt{LPPotentialCanInfluence}_p = \{\ a \in actions(s)\ |$
$canInfluence(p, a)$ and not $canCall(p, a)\ \}$
`canInfluence(principal, action) = true` $\Leftrightarrow$
$\exists\ action2 \in actions(s)\ |\ canCall(principal,\ action2)$
and $shareState(action,\ action2)$

The model for potential violations could be extended, e.g. by looking at the interplay among actions. However, such analysis requires a precise specification of pre- and post-conditions of each action. This documentation is typically lacking. Hence, this paper does not elaborate on this subject. However, our analysis tool has partial support for it.

The following section illustrates how the architectural structure can hinder LP enforcement.

6.3 LP violations illustrated

The following LP violations have been identified in the bug tracking system. A first violation (V1) is as follows (See Figure 5(a)). The *user* principal is able to access the *addJob* action, but does not require access to that action for completing his tasks (T2). It can access that action, because the permission of that action is the same as the permission it requires for accessing the *submitReport* action (T2), namely π_1. In short, the permissions can be insufficiently fine-grained to grant the permission to access certain actions, but not other actions. This is the case when permissions represent a group of actions rather than one action.



(a) Overlapping permissions. (b) Overlapping tasks.

Figure 5: Violations of LP.

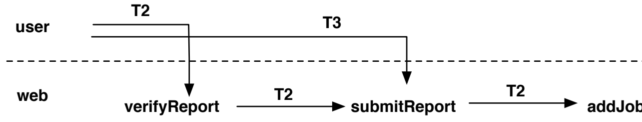
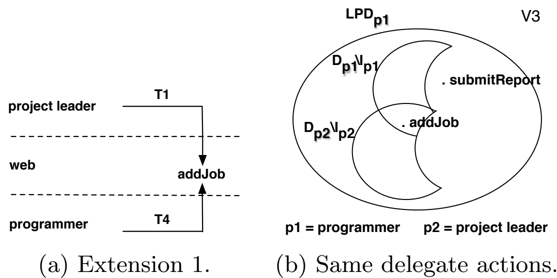


Figure 6: Extension of the bug tracking system.

A second violation (V2) is illustrated on the following extension of the system (See Figure 6). The web principal is extended with a *verifyReport* action that filters bug reports automatically for relevance. As such, T2 is extended to make use of this action. A submit tool rating report task (T3) is also added. This task is executed by the *user* principal by delegating the *submitReport* action to the *web* principal, as rating does not need to be verified. The *user* principal violates LP (See Figure 5(b)), because it can use his permission associated with *submitReport* for T3 to circumvent the verification for T2. More complex overlapping scenarios that are more difficult to identify manually can arise. In short, an indirectly responsible part of a task sharing an execution path with a directly responsible part of a task allows the principal to circumvent the indirect responsibility.

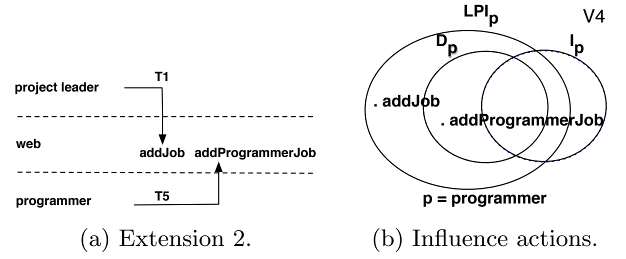
A third potential must delegate violation (V3) is illustrated on the following extension of the system (See Figure 7(a)). A *programmer* principal accessing the *addJob* action to submit todo-lists is added to the system (T4). The programmer principal potentially violates LP (See Figure 7(b)), because the system can not refrain the *programmer* from adding bug fixing jobs (T1) via the *addJob* action due it requiring access to the same action for task T3. In short, the actions are insufficiently fine-grained which results in overlapping tasks such that permissions required for a set of tasks are sufficient to execute another (part of a) task.

A fourth potential influence violation (V4) is illustrated



(a) Extension 1. (b) Same delegate actions.

Figure 7: Potential violation of LP.



(a) Extension 2. (b) Influence actions.

Figure 8: Potential violation of LP.

on the following extension of the system (See Figure 8(a)). A *programmer* principal accesses the *addProgrammerJob* action offered by the web-principal to submit todo-lists (T5). The programmer principal potentially violates LP (See Figure 8(b)), because it is able to influence the results of the *addJob* action via the *addProgrammerJob* action of T3. In short, principals can be insufficiently fine grained if they are responsible for *too* many tasks in a sense that an improper separation of tasks may allow a second principal making use of the former (e.g. by delegation) to influence the results of other actions offered by the former.

In conclusion, solving these issues properly can not be done solely by editing the security policy: a restructuring of the architecture is often required.

6.4 Measuring adherence to least privilege

Note that the formal definition of violation also allows to quantify LP adherence by a simple metric: the total number of LP violations is the sum of the violations of each principal. This metric can be used to compare two iterations of the same system, but not two different versions for several reasons. First, a sensitivity analysis of the metric shows that changing the number of processes, tasks, permissions, or users in-linearly in/decreases the number of violations. Second, not all violations are equally dangerous. For instance, a customer getting manager privileges is more dangerous than a manager obtaining customer privileges. This metric and its extensions have been elaborated on in [6].

7. LP RE-FACTORING

This section presents our solving algorithm, which solves LP violations in a software architecture by applying a set of architectural transformations. The input provided to the algorithm is threefold: a software architecture, a list of violations to solve, and a list of transformations. The violations are first translated from the Task Execution Model to violations in the logical, process, scenarios, and security views. Next, architectural transformations are applied on this set of views to solve the violation. Hence, the output is a software architecture without (a subset of) the input violations and thus an architecture that adheres better to LP.

7.1 Transformations

Different transformations exist to improve LP in a software architectural structure, among which: (i) splitting components, processes, and principals into several isolated units and thus lowering permissions assigned to these units, (ii) splitting permissions into more fine-grained permissions, (iii) removing permissions for actions that do not appear in tasks,

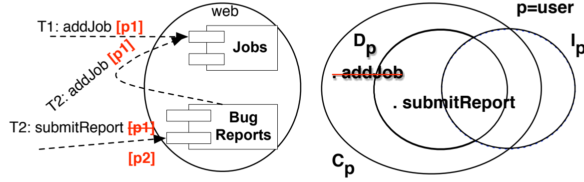


Figure 9: Splitting permissions.

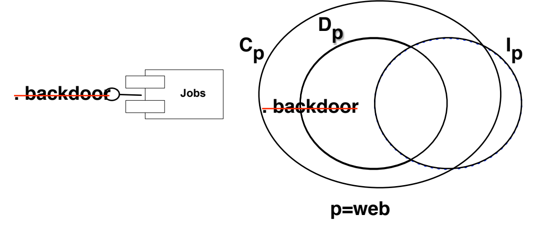


Figure 10: Removing unused actions.

(iv) rewiring the architecture (i.e. rerouting tasks to other principals in order to split up conflicting permissions), (v) assigning different principals to the runtime elements, (vi) applying well-known solutions (such as sand boxing) to introduce LP in selected parts of the architecture (see Section 9), and so on. The discussion in this paper is limited to the first three transformations, as our experiments indicate that they have the biggest impact on LP (assuming that each process runs as its own principal) [4].

Note that the transformations presented here are a generalization of the transformations presented in previous work. Each transformation consists of (i) the application conditions identifying the violation in the Task Execution Model (TEM), (ii) a translation of this violation from the Task Execution Model to the input views (REL), and (iii) a transformation resolving the issue in the input views (TRF).

7.1.1 Splitting permissions

Splitting a permission (TR1) is a transformation that optimizes permission granularity by moving a subset of a permission's actions to new permissions (See Figure 9).

Transformation

TEM `If` $\exists\ p1 \in principals(s)\ |$
$a1 \in \mathcal{D}_{p1}$ `and` $a2 \in (\mathcal{C}_{p1} \setminus \mathcal{D}_{p1}$
REL `Then If` $\exists\ perm \in perms(s)|$ `and`
$a2 \in perm$ `and` $a1 \in perm$
TRF `Then perm1 =` $\{ a \mid a = a1 \}$ `and`
`perm2 =` $\{ a \mid a = a2 \}$ `and`
`perms(s) =`
$(perms(s) \setminus perm) \cup \{ perm1, perm2 \}$
`principalPerms(s) =`
$(principalPerms(s) \cup \{ (p1, perm1) \}) \setminus \{ (p1, perm) \}$

Argumentation. This transformation solves LP violations, because splitting a permission and assigning that 'smaller' permission to a principal, reduces the number of actions a principal can call.

Limitations. This transformation influences the access policy that is enforced, because the policy has to be updated to include the newly created permissions. Hence, this transformation reduces the maintainability of the policy.

7.1.2 Removing unused actions

Removing unused actions from a principal (TR2) is a transformation that optimizes principal granularity by removing actions offered by a principal, but not used by other principals. In other words, these actions do not appear in the plus one view nor in the execution view. An example of such an action is a backdoor (See Figure 10).

Transformation

TEM `If` $\exists\ p1,\ p2 \in principals(s)|$
$\exists\ a \in (\mathcal{C}_p \setminus \mathcal{D}_p) \setminus \mathcal{D}_{p2}$
REL `Then If` $\exists\ c \in components(s)$ `and`
$\exists\ perm \in perms(s)|$
$a \in actions(c)$ `and` $a \in perm$
TRF `Then perm =` $perm \setminus \{ a \}$ `and`
$actions(c) = actions(c) \setminus \{ a \}$
`If # perm == 0`
`Then principalPerms(s) =`
`principalPerms(s)` $\setminus \{ (p1, perm) \}$

Argumentation. This transformation solves LP violations, because it reduces the number of actions a principal can call.

Limitations. This transformation has one main drawback: removed actions might have been needed in the future.

7.1.3 Splitting component, process, and principal

Splitting a component (TR3) is a transformation that optimizes principal granularity by splitting a component's set of actions into multiple sets of actions. Each new set is a new component and is executed in a different process by a different principal (See Figure 11). One of the challenges is that the component has to be split in a way that preserves the semantics of the component: semantically related actions must remain adjacent after splitting. Our approach uses action's name and parameters to approximate related actions.

Transformation

TEM `If` $\exists\ p1,\ p2,\ p3 \in principals(s)$ `and`
$\exists\ a2 \in (\mathcal{MD}_{p1} \cap \mathcal{MD}_{p3} \cap \mathcal{ME}_{p2})$ `and`
$\exists\ a1 \in (\mathcal{ME}_{p2} \setminus \mathcal{D}_{p1})|$
$shareState(a1,\ a2)$ `and`
$p1 \neq p2 \neq p3$ `and` $a1 \neq a2$
REL `Then If` $\exists\ c \in components(s)$ `and`
$\exists\ perm1,\ perm2 \in permissions(s)|$
$a2 \in perm2$ `and` $a1 \in perm1$ `and`
$a1,\ a2 \in actions(c)$ `and` $notRelated(a1,\ a2)$
TRANSF `Then d =` $\{ a1 \}$ `and` $c = c \setminus \{ a1 \}$ `and`
`components(s) =` $components(s) \cup \{ d \}$

Argumentation. The permissions of the principal will reduce for one of the following reasons.

First, partitioning a principal results in subprincipals each having less permissions associated with mustExecute actions by definition.

Second, partitioning a principal in a way each partition is responsible for less tasks, results in partitions requiring less

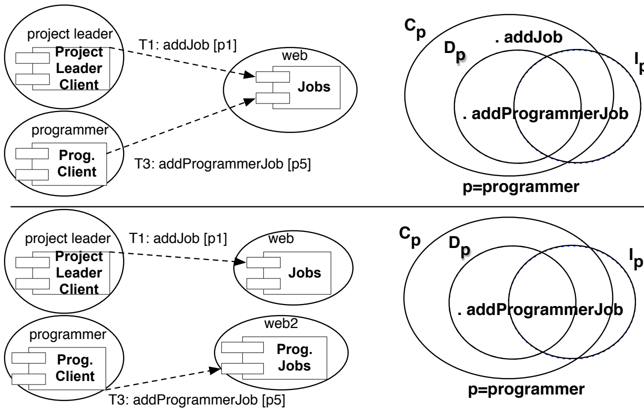


Figure 11: Splitting a component.

permissions associated with mustDelegate actions by definition.

Third, if a principal that grants permissions via influence to other principals is split, then it is possible that these permissions will not be granted anymore, because the shared state propagating these permissions does not exist anymore. Hence, the number of permissions of that other principal is lower than that number before splitting. This will solve potential influence violations.

7.2 LP transformations illustrated

The identified violations in the example can be solved by applying the above transformations as follows. The first violation (V1) was due to an insufficiently fine grained permission π_1. Hence, it can be solved by splitting that permission in two new permissions, namely π_{1a} and π_{1b} for *addJob* and *submitReport* respectively. Indeed, this refrains the *user* from accessing the *addJob* action.

The second violation (V2) was due to overlapping indirect and direct actions. Hence, it can be solved by rewiring T2 through *verifyReport* or by splitting *submitReport* in *submitBugReport* and *submitRatingReports* with associated permissions.

The third violation (V3) was due to an insufficiently fine grained action *addJob*. Hence, it can be (manually) solved by splitting that action in two actions, namely *addJob* and *addProgrammerJob*, with two new associated permissions, namely π_1 and π_3 respectively.

The fourth violation (V4) was due to a too large principal. Hence, it can solved by splitting the *web* principal into smaller principals. Note that this enforces the principle of separation of privilege.

8. BANKING CASE STUDY

This section presents the validation results of the presented approach. The approach has been applied to several case studies including a chat system, a conference management system, a digital publishing system, and a banking system. In the context of this paper, we focus on a banking system. In the rest of this section, the case study is elaborated upon in order to appreciate the type of problems and solutions that can be addressed in practice. Afterwards, a summary of our preliminary results published in [3] is compared with the results of our approach presented here. The influence of the application of these transformations on other architectural qualities (like mainainability) is not discussed here, because it has been published in [4].

8.1 Context and architecture

The case study is a simple online banking system that has been described in [7]. This system has been chosen because the documentation contains the required views for our analysis, including a partial mapping between permissions and actions. The system supports two major scenarios: banking employees use the system in their everyday operations, while customers can use the system from home. The system is able to handle customers, employees, accounts, and financial transactions such as wire transfers.

The logical view (See Figure 12) decomposes the system into four component tiers. The client tier consists of a browser `Client` component, and is used by a customer or employee to interact with the web tier. The web tier is the front end of the system and it relays user requests to the application logic tier. Hence, it provides a representation of the application tier. The latter is responsible for processing the incoming requests like transactions, while the data tier stores the results of these requests. The application tier consists of the following components. An `Authentication` component is responsible for authenticating users, while an `Authorization` component handles access control decisions. The `UserManagement` component provides operations to retrieve and manage user information. The `AccountManagement` component handles account information, while the `FinancialTransaction` component provides operations for executing financial transactions. The `Persistence` component is used as proxy for the data tier.

The interaction scenarios show that the system supports 3 customer tasks and 21 employee tasks. This view has been omitted due to space constraints, however the scenarios are documented in the Task Execution Model (See Figure 13). Only the following scenarios are shown:

1. *Do wire transfer*: a customer wants to add a wire transfer to a list of pending transactions. This task is executed in two phases. In a first phase, the *customer* initiates the wire transfer by accessing the *initiateWireTransfer* action of the web tier. This tier forwards this request to the application tier via the *prepareWireTransfer* action. The application logic then verifies the wire transfer by using information from the data tier (via the `Persistence` proxy) about the user executing the wire transfer (via the *getUser* action), the customers the money should be wired to and from (via the *getCustomer* action), and their account details (via the *getAccount* action). Finally, the verified results are presented to the user. In a second phase, the user has to confirm the prepared wire transfer.

2. *Do wire transfer for a customer*: an employee wants to execute a wire transfer on behalf of a customer. This is analog to the previous task.

3. *Create customer*: an employee creates a new customer. This task is analog to the previous task, except that it uses the actions related to customer creation.

The process view was not defined in the architectural documentation. Hence, we assumed that every component runs

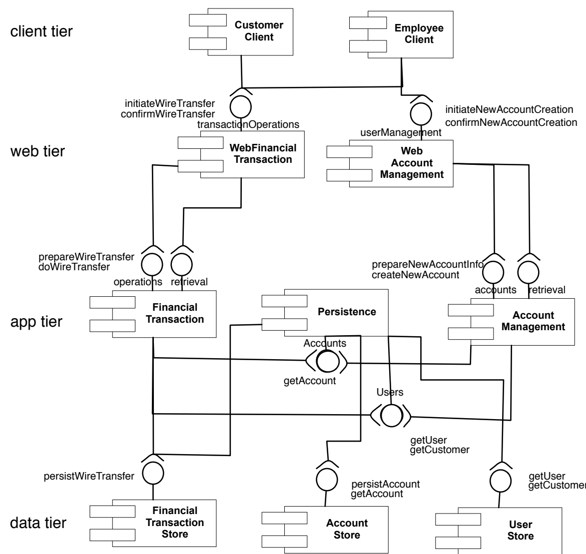


Figure 12: Excerpt of the banking system architecture.

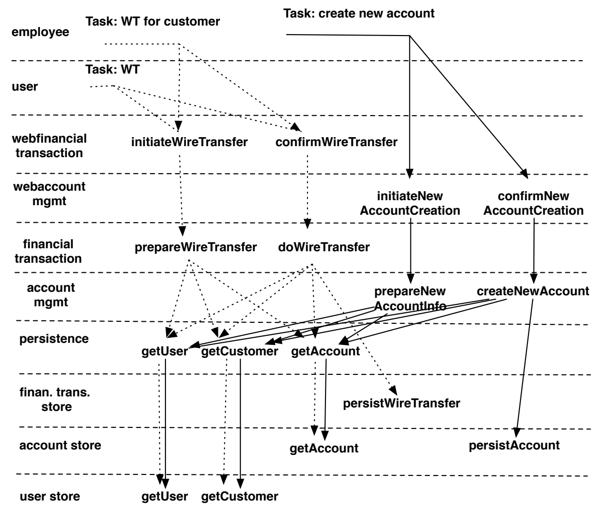


Figure 13: A Task Execution Model for a subset of the banking system.

in its own process and with its own principal. The results for separate principals per tier are comparable.

The security view was only partially defined in the architectural documentation. Defined permissions included the *modify account* permission which allows principals to modify account details or pending transactions by accessing the *doWireTransfer* and the *createNewAccount* actions. However, missing information like a mapping between some of the defined permissions and the architectural actions and principal-permission assignments was added. The former by manually deriving the actions each permission should contain. This was straightforward, as the permission name indicated its relation to the interface actions. The latter by running our inferring algorithms.

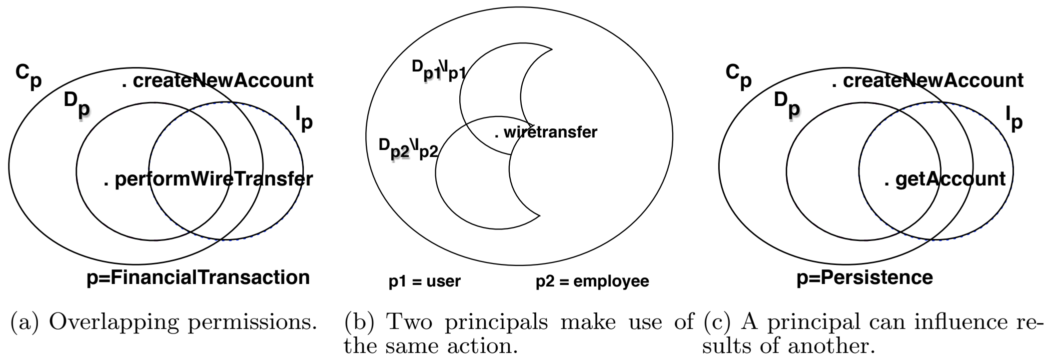
8.2 Analysis

In this section, we discuss three categories of LP problems that have been identified by our tool in the banking system. Our tool started from the Task Execution Model (see Figure 13) and the list of permissions (See paragraph 5 of this section). The numbers of violations identified for each category are listed between parenthesis after each category.

A first set of real violations is related to permission granularity (23). Take for instance the *FinancialTransaction* principal, which can call the *createNewAccount* action, but should not access that action for completion of his tasks (See the Task Execution Model in Figure 13). It can access that action, because the permission of that action is the same (*modify account* (system) permission) as the permission it requires for accessing the *performWireTransfer* action. This is illustrated in Figure 14(a), where the *create new account* action is listed as a violation for the FinancialTransaction principal. This may have the following consequences. An attacker penetrating into a component running as this principal, can create an employee account to do the actions a employee is allowed to do. This problem can be solved by splitting the permission as described earlier.

A second type of potential violations is related to action granularity (9). Take for instance the *user* principal, which potentially violates LP (See Figure 14(b)), because the system can not refrain the *user* from executing wire transfers on behalf of customers (employee task) via the *(web) wire transfer* action, because it requires access to the same action for its own task. This can be seen in the Task Execution Model. This may have the following consequences: customers and employees are able to perform wire transfers on each others behalf. Take for instance the following attack scenario. An attacker, registered as customer, accesses the wire transfer action with the following parameters: a guessed or intercepted employee id as user executing the action, an intercepted or guessed from account, a to account he owns, and a chosen amount. This results in a wire transfer of the chosen amount to the attacker's account. These potential violations can be avoided by introducing one of the following solutions. First, introduce different actions for employee and customer wire transfers with different associated permissions. This architecturally enforces that a customer can not execute employee wire transfers and vice versa. Second, introduce context based access control that verifies whether a customer or employee is allowed to execute that wire transfer. Third, introduce a third person (e.g. manager) manually verifying whether the transaction is allowed.

A third type of potential violations is related to principal granularity (1). The *persistence* principal potentially violates LP, because it is possibly able to influence the results of the *createNewAccount* action via the *getAccount* action of his tasks (See the Task Execution Model and Figure 14(c)). It is important to note that this is considered a violation, because the *createNewAccount* action is not delegated by the `Persistence` component, but by the web-tier. This may have the following consequences. An attacker, registered as a regular customer, submits a special crafted wire transfer via his online banking system. This wire transfer information is indirectly forwarded to and processed by the `Persistence` component. This component accesses `Account Management` to obtain the information of the customer performing the wire transfer. However, due to a bug (e.g. SQL



(a) Overlapping permissions. (b) Two principals make use of the same action. (c) A principal can influence results of another.

Figure 14: Violations of LP in the banking system.

injection), the `Persistence` component creates and stores a new employee account, which can be used later on. Most banking policies do not allow customers to behave as employees. These problems can be avoided by separating privileged data (in this case employee and customer accounts) and by rigorously applying context based access control.

Note that all of the violations of the first set could be solved without intervention from the architect, while the violations related to the second and the third set can only be solved with intervention from the architect.

8.3 Comparison

	Metric	Chat System	Conf. System	Publ. System
Old approach				
real	(a) #Internal violations	6	96	64
real	(a) #Required violations	2	28	16
pot.	(b) #Indirect violations	0	536	0
This approach				
real	(a) #Real violation	0	1	54
pot.	(b) #must delegate violations	0	2	0
pot.	(b) #influence violations	2	15	86

Table 1: Comparison between detected violations for the old approach and this approach for 3 cases.

This section compares our initial results published in [3] with the results of the approach here. More specifically, we applied the approach presented here on the three case studies our old approach has been applied on, namely a chat system delivered with arch studio (case 1), a conference management system (case 2), and a publishing system (case 3) and compared the results of both approaches by elaborating on false positives, type of violations that are only detected by one approach, and the type of violations that are detected by both approaches.

Table 1 summarizes the results of both approaches. The metrics used in both approaches are different, but the ones with the same prefix can be mapped to each other. Note that the approach presented here counts violations per action per principal, while the old approach counted per combination of permissions that allows a user to execute a task he must execute and a task he must not execute. This explains the larger number of violations for the old approach.

The approach presented here detects less false positives for three reasons. First, it marks actions that appear in the 'must delegate set' of multiple principals as potential violations, while the old approach marked these as (required or internal) violations. This set is likely to contain false positives, because the implementations of these must delegate actions can ensure that the accessing principals only execute their execution path of the action. Second, the approach presented here uses more realistic required permission assignment algorithms such as required neighbor only, while the old approach assigned permissions required for the remainder of the task. The latter strategy assigns more permissions which means more violations. This explains the difference between the violations of type real (a). Third, the same is true for indirect permissions: permissions obtained via shared state are now only delegated one hop. This explains the violations of type potential (b) for case 2 (from 536 to 15).

Additionally, the indirect and the influence violations for case 2 and 3 differ differently, due to the old LP metric. Indeed, the old metric indicates an indirect violation via shared state if two different *end-users* make use of that state (possibly via intermediate principals), while the new approach also takes the intermediate *principals* into account. In other words, two tasks executed by the same end-user each passing through two different intermediate principals to access the same shared state is not counted as a violation in the old metric, but is counted as a violation in the new metric. This explains the difference between violations of type potential (b) for case 3 (from 0 to 86).

Finally, the approach presented here marks actions for which a principal is directly and indirectly responsible (and thus allow to circumvent e.g. a verification component) as violations, while the outdated approach does not.

Overall, the approach presented in this paper is less prone to false positives and is more powerful in capturing real violations, i.e. better coverage.

9. RELATED WORK

This work is related to two research domains: security engineering, and software re-factoring. Related work on engineering focusses on (i) program separation, (ii) model

checking, and (iii) execution monitoring.

Program separation, a technique to separate a program in multiple processes, has been successfully applied in programs such as qmail [1] to minimize trust. Our least privilege approach provides a systematic and automated means for program separation at architectural level. Another more general approach is privilege separation [2], which partitions an existing program into two processes: a privileged monitor and an unprivileged slave. Our approach extends privilege separation by optimizing the number of privileged processes.

Model checking techniques are used to verify whether a design meets certain properties, such as LP. In his PhD thesis [9], Jürjens explains how one can use UMLSec to enforce LP by formulating LP requirements and verifying UMLsec specifications with respect to these requirements. The difference with our approach is that our approach functions independently from the access policy. Rubacon [8] is a tool that checks UML models and their configuration data for adherence to security policies. Rubacon and our work share a similar idea: identify possible (sub)tasks (transactions) that can be executed by granted permissions. The difference is that it uses class diagrams rather than component diagrams for specifying policy rules and permissions. SecureUML [10] integrates RBAC access control techniques and UML to automatically generate access control infrastructures/policies. Our technique can extend this approach by using the architectural structure to identify whether a minimized architectural policy can be generated.

Execution monitoring is another technique that limits the privileges assigned to a program. These techniques block system calls and access to file and network resources based on policies. An examples is Systrace [12]. These mechanisms have two drawbacks. First it is hard to specify policies in terms of application-specific resources and functions, because these don't always map on files and system calls, as illustrated by security automata [15]. Second, these mechanisms add runtime overhead to limit the number of privileges, while we ensure that these are limited by construction.

Significant work has been published in the area of software re-factoring. Mens [11] presents a detailed survey. Software re-factoring is generally viewed as the process of improving the internal structure of a software system without disrupting its external behavior. This improvement of the internal structure can be based on a specific quality goal, such as security, or in this paper LP. While re-factoring can be applied, no concrete results for LP are available in this area.

10. CONCLUSIONS AND FUTURE WORK

This paper proposes a technique that automates the identification and resolution of LP violations in software architectures. To this aim, the concept of architectural level least privilege has been modeled formally. This model has been leveraged to create an algorithm that analyzes an architecture starting from conventional documentation and identifies violations. Subsequently, a set of architectural transformations that solves these violations have been proposed. The approach has been validated by means of several case studies, one of which has been presented in the paper.

In future work, the authors plan to apply the same formal approach to other security principles (e.g., defence in depth) and are interested in the study of the interplay and the optimal trade-offs among the principles.

11. REFERENCES

[1] D. J. Bernstein. Some thoughts on security after ten years of qmail 1.0. In *CSAW '07*, pages 1–10. ACM, 2007.

[2] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX 13*, August 2004.

[3] K. Buyens, B. De Win, and W. Joosen. Identifying and resolving least privilege violations in software architectures. In *ARES*, 2009.

[4] K. Buyens, B. De Win, and W. Joosen. Resolving least privilege violations in software architectures. In *SESS '09*, 2009.

[5] K. Buyens, R. Scandariato, and W. Joosen. Process activities supporting security principles. In *IWSSE '07*, 2007.

[6] K. Buyens, R. Scandariato, and W. Joosen. Measuring the interplay of security principles in software architectures. In *METRISEC '09*, 2009.

[7] E. Debie and P. De Ryck. Non-repudiation middleware for web-based architectures, June 2009. `http://www.cs.kuleuven.be/{~}lieven/teaching/thesis/ThesisErikDebiePhilippeDeRyck.pdf`.

[8] S. Höhn and J. Jürjens. Rubacon: automated support for model-based compliance engineering. In *ICSE 13*, pages 875–878, 2008.

[9] J. Jürjens. *Secure Systems Development With UML*. Springer, 2005.

[10] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-based modeling language for model-driven security. *Lecture notes in computer science*, pages 426–441, 2002.

[11] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions of Software Engineering*, 30(2):126–139, 2004.

[12] N. Provos. Systrace - interactive policy generation for system calls.

[13] J. Ren and R. Taylor. A secure software architecture description language. In *Workshop on Software Security Assurance Tools, Techniques, and Metrics*, 2005.

[14] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[15] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[16] J.M. Wing. A Call to Action: Look Beyond the Horizon. *IEEE Security & Privacy*, pages 62–67, 2003.