Resolving least privilege violations in software architectures

Koen Buyens, Bart De Win, and Wouter Joosen
IBBT-Distrinet
Belgium
first.last@cs.kuleuven.be

Abstract

Supporting a security principle, such as least privilege, in a software architecture is difficult. Systematic rules are lacking, no guidance explains how to apply the principle in practice. As a result, security principles are often neglected. This lowers the overall security level of the software system and the cost of fixing such problems later on in the development cycle is high.

We propose an improvement in supporting least privilege in software architectures. We have identified architectural transformations that reduce violations to the principle of least privilege. These transformations have been implemented. We have applied the solution on a case study.

1 Introduction

Least privilege (LP) is a well-known security principle which prescribes that a user must not be assigned permissions that he does not require. Consequently, the user cannot execute tasks that he is not allowed to execute [15]. Security principles must be considered throughout the entire software development life-cyle [7]. While techniques exist to reason about adherence to LP in software, such as policy reasoning [16] or restructuring [3], no systematic method exists at the architectural level [18], where the consequences are significant though [2].

We argue that at architectural level, LP minimizes the capabilities of a (set of) component(s) that have to be executed in a single process. Two important factors in this context are (i) the processes (also called controllable units) and (ii) the access policy that is to be enforced on these units. LP will be respected if both the architectural structure and the policy are adequate (See Figure 1). Indeed, it can be impossible to enforce LP with a given (weak) software architecture (SA). (This is illustrated by the example in Section 2). In other words, LP violations can not always be addressed by changing the policy, one may have to adapt the architecture. To the authors' knowledge, no systematic techniques exist today to deal with this problem. One of the factors that makes this problem extra challenging can be the abstraction level of a given software architecture and, therefore the limited amount of information available for reasoning about LP.

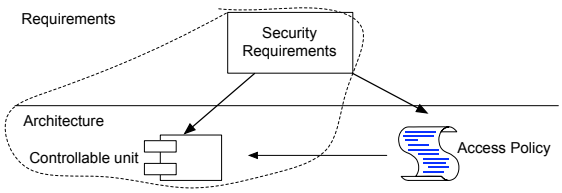

Figure 1. LP in SA: a combination of architectural structure and access policy.

Our approach predicts LP problems in the final software product by using a catalog of use cases and an architectural description. The approach approximates a worst case assignment of permissions for the given set of use cases. The computed assignment is used to determine (potential[1]) violations of the principle.

More concrete, the model supporting our approach [5] states that a system adheres to LP if all its components adhere to LP. A component does not adhere to LP if it, based upon the attributed permissions, is capable of executing use cases it is not responsible for. These cases are called unwanted tasks; every possibility to execute an unwanted task is considered to be an LP violation.

The goal of the presented work is to reduce the number of LP violations by using architectural transformations. The contributions are twofold: (i) we have identified 5 transformations that have the potential to solve LP violations, and (ii) we have studied the influence of the application of these transformations on architectural qualities, such as security, modifiability, and complexity.

[1] LP violations at architectural level are *potential* in the sense that one can never be certain about violations until the system has been implemented. However, a number of problems may (and probably will) persist in the final system.

The remainder of this paper is structured as follows. Section 2 further motivates the problem by means of an example. Section 3 presents the transformations that solve a subset of the identified violations. Section 4 applies the results on a case study. Section 5 discusses related work and section 6 concludes.

2 Motivating Example

Consider an integrated groupware system that consists of three components (See Figure 2) [2]. The first, Calendar, is a component that enables a user to keep track of events and to find public events to attend. The second, Repository, is a content management system that allows users to upload and share files. The third, Jobs, is task management component that enables a user to create, update, and delete todo lists. Two components integrate the functionality of these main components and act as front-ends for end users: the Internal Groupware Client is used by employees, while the External Web Client is used by external users.

External Users use the External Web Client to (i) upload documents used and verified by employees by means of a verification task, (ii) create public events and, (iii) confirm attendance of events organized by Employees. *Employees* use the Internal Groupware Client to (i) execute groupware tasks such as modify events, and to (ii) review input received from external users. Table 1 lists tasks that have been assigned to the users (i.e. security requirements).

User	Task (use case)
Employee	Upload, read, verify working document
Employee	Add, modify event
Employee	Add task
External User	Upload working document
External User	Confirm event, add public event

Table 1. Assigned calendar tasks.

In order to enforce these requirements, we need to specify an architectural-level access policy that expresses the rules in terms of component's interface methods. Situations arise in which the architectural structure jeopardizes LP.

First, tasks can overlap such that permissions necessary for a set of tasks are sufficient to execute other tasks. E.g., if an external user has the right to *upload a working document* (perm4, perm3 for task T1), he will also be able to *add tasks* (perm4 for task T5), even if this is not desirable.

Second, the granularity of rights specification in the access policy can be insufficiently fine-grained to grant the right to execute certain tasks, but not other tasks. This is

[2]The represented architectural diagram merges behavioral, component and connector, and security views because of space constraints. Hence, the used notations are illustrative only.

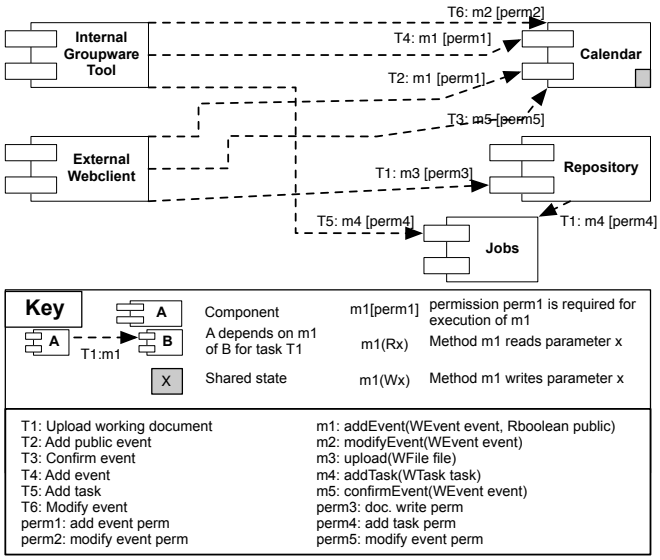

Figure 2. Excerpt from the component diagram of the an integrated calendar system

the case when the execution of a task depends on method parameters (rather than methods), or when permissions represent a collection of methods rather than a single method. For instance, the system can not grant a user the right to *add public events* (T2), but refrain him from *add events* (T4), as the difference is represented by a boolean parameter. Note that this problem can be solved by increasing the granularity of the access policy, but this is rarely the case in practice.

Third, if two conflicting tasks are able to influence each other's operations, then the system might not be able to enforce LP correctly. Indeed, we can argue that influencing operations of a task is a weak form of having the permissions to execute it. In the groupware example, the *confirmEvent* method might interfere with the *modifyEvent* method, since they may use the same event store. Consequently, task T3 might interfere with task T6. This is not allowed, because external people may not be allowed to change events based on the company policy.

Solving these issues properly can not be done solely by editing the policy: a restructuring of the architectural structure is often required to address them. Indeed, our previous paper identified LP problems in three case studies, most of which were not solvable by editing the access policy.

Based on this example, we can summarize our formal LP model as follows. A *component* is described in terms of the *actions* of its interfaces, which are used to interact with the component. Such an interaction is built into the system to realize a task (or use case). This task is expressed as a sequence of action tuples. For the execution of these actions, one needs permissions (representations of the rights to per-

form a set of actions). We distinguished three types of permissions: *required permissions* of a component are the permissions it needs to complete the tasks (or parts of tasks) it is responsible for, *internal permissions* of a component are the permissions linked to the actions of its interfaces, and *indirect permissions* are permissions that might be obtained by interfering in a component's internal (shared) state. A system adheres to LP if all its components adhere to LP. A component does not adhere to LP if it, based upon the attributed permissions, is capable of executing (parts of) tasks it is not responsible for. Having one permission that allows the execution of (a part of) a task a component is not responsible for is called a violation. Obviously, the number of violations can be lowered by reducing the number of permissions. Different transformation strategies exist to do this.

3 Transformations

Different strategies exist to accommodate LP in a SA. This section elaborates on the useful strategies by documenting a transformation, an argumentation that that transformation lowers permissions and thus privileges, and its limitations. A (semi) formal version of these transformations and a proof sketching that they indeed lower permissions was omitted due to space constraints, but is published as part of a technical report [4].

3.1 Splitting a component

Splitting a component (TR1) is a transformation that splits a component into several components by splitting its set of actions into multiple sets. Each new set is a new component (See Figure 3).

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 state to approximate related actions: actions that use the same state are related. However, in order to split a component that contains related actions used by violating tasks, we require extra information to be present in the architectural description: read/write on the action's parameters.

Transformation If two tasks are delegated to a component via two actions, and the internal or required permissions associated with these actions cause a LP violation, then, based upon the shared state between the tasks, the component can be split as follows (See Figure 3).

1. If the shared state is empty, split the component in two disjunct parts by moving the interfaces/actions that one task uses to (a) new interfaces in a new component. Update the tasks accordingly.
2. If the shared set is not empty, and if one task reads state that is written by the other, then create a new component containing a copy of the actions of the writing task. Add a new interface to the original component to update shared state. Extend the task reading shared state to include update actions provided by the new component.

Furthermore, if at least one indirect permission (See section 2) is causing the violation, then split the component with the shared state that propagates this permission as described above.

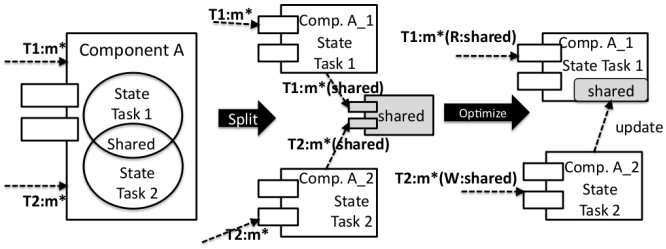

Figure 3. A component with overlapping read methods and a write method can be split.

Argumentation It is easy to see that the permissions will reduce for one of the following reasons.

First, partitioning a component will result in subcomponents each having less internal permissions by definition.

Second, partitioning a component in a way each partition is responsible for less tasks, will result in partitions requiring less required permissions by definition.

Third, if a component that grants indirect permissions to another component is split, then it is possible that these permissions will not be granted, because the shared state propagating these permissions does not exist anymore. Hence, the number of permissions of that other component is lower than that number before splitting.

Limitations The rule does not work if both tasks write on the shared state, because they can still influence each other via these components. In other words, the indirect permissions of both components will not have been decreased.

3.2 Splitting permissions

Splitting a permission (TR2) is a transformation that optimizes permission granularity by moving a subset of a permission's actions to new permissions (See Figure 4).

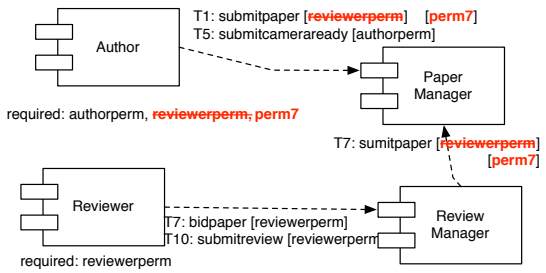

Figure 4. Splitting permissions.

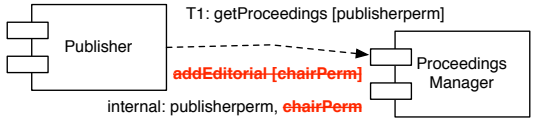

Figure 5. Removing unused actions.

Transformation If task 1 requires a permission p, task 2 requires the same permission p, this permission is associated with a set of actions, task 1 uses a subset of these actions (subset1), task 2 uses a different subset (subset2), and both tasks are executed by different users, then split permission p in the following way:

- p1 = subset1 $\setminus$ subset2
- p2 = subset2 $\setminus$ subset1
- p3 = subset1 $\cap$ subset2
- p4 = p $\setminus$ (subset1 $\bigcup$ subset2)

Argumentation Splitting a permission increases the number of permissions assigned to a component, because the new permissions are assigned to it. However, the transformation solves LP violations, because a splitted permission reduces the number of actions a component is allowed to execute, and hence the number of tasks it can execute.

3.3 Removing unused actions

Removing unused actions from a component (TR3) is a transformation that removes the component's unused actions (See Figure 5).

Transformation If an action of a component is not used by a task, then remove that action from that component.

Argumentation The number of permissions will reduce, because the number of internal permissions reduce.

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

3.4 Moving tasks between users

Moving tasks between users (TR4) is a transformation that moves a task from a user sets of tasks to another users set of tasks. In other words the second user is allowed to execute the moved task, while the first is not.

One of the challenges is that the task has to be moved in a semantically correct way: the user (or role) a task is moved to should naturally be responsible for that task. Indeed, it does not make much sense to assign a *make a sale*-task to a customer. This is hard to determine automatically, therefore the architect should decide whether the suggested transformation is a good one.

Transformation If a user is allowed to execute a significant (e.g. the user with the most violating tasks) part of the tasks causing a violation, then: (i) identify a user who can execute the task without causing a violation, and (ii) move the task to the this user.

Argumentation Making another user responsible for a task decreases the number of permissions of the first user (if the remaining tasks don't require all permissions the moved task requires), while it only increases the number of permissions of the second user if this user does not have the permissions of the moved task. This transformation can solve LP violations, if a violating task is moved to another user.

3.5 Rewiring the architecture

Rewiring the architecture (TR5) is a transformation that reroutes a task to another component by swapping its action-tuples with congruent action tuples (See Figure 6).

One of the challenges is that the task has to be rewired in a semantically correct way: the component a task is rewired to should offer congruent actions than the component the task is rewired from. Congruence is determined by the pre and post conditions of a component's actions. An action a1 is congruent to another action a2 iff a1 and a2 have the same signature and a1.pre $\rightarrow$ a2.pre and a1.post $\rightarrow$ a2.post.

Transformation If two tasks are delegated to a component via two actions, the internal or required permissions associated with these actions cause a LP violation, and one of these actions (or a congruent one) is offered by another component, rewire that task to another component as follows: (i) modify the task to use the action of that other component instead, and (ii) change the dependencies accordingly.

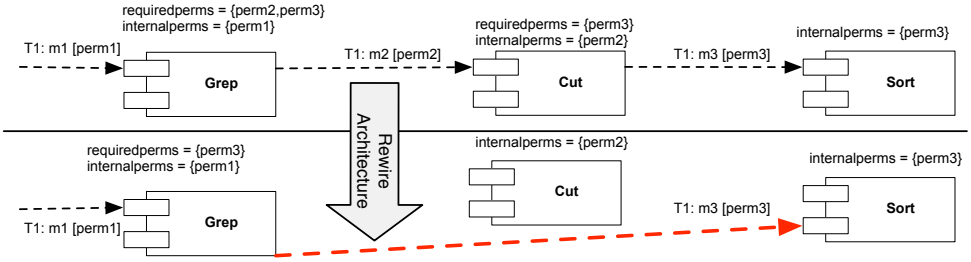

Figure 6. Rewiring the architecture.

Argumentation The assigned permissions will reduce for the following reasons. First, removing a task from a component lowers the its internal and required permissions (in case they are not required for another task). Second, adding a task to another component only increases the permissions if that component does not already have them. Hence, rewiring might reduce the number of overall permissions.

4 Applying the results on a case study

This section illustrates the transformations by applying them on a case study. First, the case is elaborated upon in order to appreciate the type of problems and solutions that can be addressed in practice. Next, the results are discussed from a broader perspective: are the identified transformations lowering the number of LP violations, but not deteriorating other architectural qualities? Finally, we zoom out and have a look at the solution spectrum: what kind of transformations should (or should not) be able to solve least privilege violations?

4.1 Conference Management Case Study

This section elaborates upon a case study in order to appreciate the type of problems and solutions that can be addressed in practice. The conference management system system [13] automates (parts of) a peer review process which is performed to submit, select, and prepare papers to be published.

Different actors make use of this system, among which the chair, the reviewer, the author, and the publisher. The *chair* coordinates the peer review process, the *reviewer* evaluates quality of submitted papers, the *author* submits papers of research, and the *publisher* publishes them.

The execution of each task is restricted to certain actors: an author is not allowed to review papers he has written.

In the architecture, components responsible for these features are the following. The Conference Manager (CM) is used by the Chair to send call for papers, and notifications of acceptance. The Paper Manager (PM) collects the papers submitted by Authors and distributes them to other components. The Review Manager (RM) is used by Reviewers to select papers to review (bid), to obtain these papers, and to submit their reviews. A Proceedings Manager (PDM) is used by the Publisher to assemble proceedings.

The architecture did not describe permissions. Therefore, we extended it by mimicking real world permission assignment: only four permissions were introduced.

We discuss some of the identified problems. First, the permissions are too coarse grained, because they allow the author to execute all reviewer related tasks (See Figure 4). This problem can be solved by T2. Indeed, splitting reviewerperm in a permission for the submit paper method and one for the other reviewer methods prevents the author from executing the reviewing tasks.

Second, the Proceedings Manager can execute all Chair related tasks, because its unused internal addEditorial method has chair permissions (See Figure 5). This problem is solved by T3. Indeed, removing that action removes the chairperm permission from that component.

Third, the Review Manager is responsible for both reviewer related tasks and chair related tasks, which was explicitly forbidden (See Figure 7). This problem is partially solved by splitting the component in two components: one responsible for reviewer related tasks (Review Manager) and one for the chair related tasks (Review Manager Split). The rewiring the architecture transformation could have solved these problems as well, but the transformation did not find a suitable component to rewire the task to.

Fourth, the Reviewer is allowed to execute the highest number of tasks causing a LP violation. This number can be lowered by applying the moving tasks between users transformation. However, the transformation is not able to find a suitable user to move the task to.

4.2 Validation results

This section verifies whether the transformations lower LP properties when applied in isolation, and assesses their impact on architectural qualities such as security, modifiability, and complexity. These results are preliminary, as the

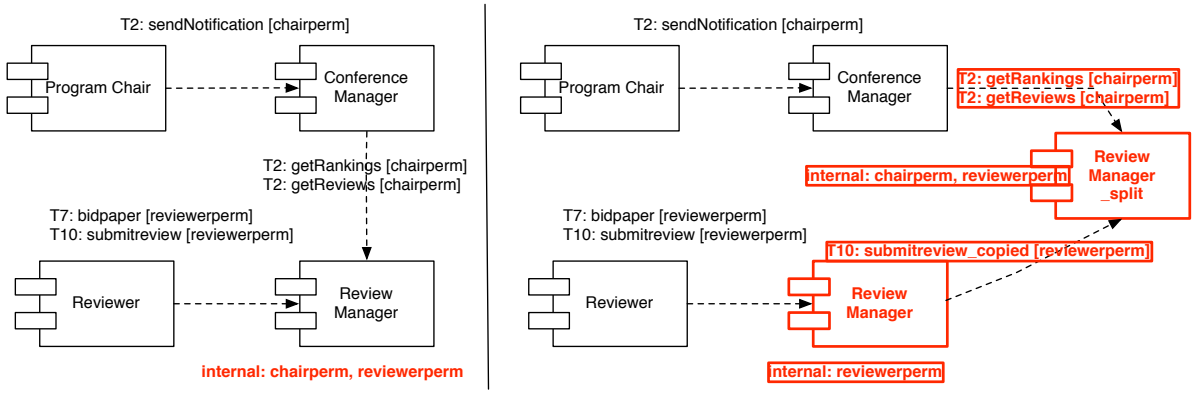

Figure 7. Splitting components.

transformations are applied on a single case study. Nevertheless, these initial results already provide valuable insights.

We examined whether LP improved (in terms of the number of violating components (# viol comps), the tasks causing a violation (#viol tasks), and the number of violations (#viol) (See Table 2). In general, LP did improve, but *splitting a component* turns out to be the most effective solution. Indeed, it reduces the number of violating components by approximately 60%, while the other transformations do not. However, the transformations *splitting a permission* and *removing an action* decreases the number of violations. Indeed, the latter reduces the number of violations by 17%, while the former lowers the number of violations caused by internal and required permission assignment by 15%, but increases the number of violations caused by indirect permission assignment. *Moving tasks* and *rewiring the architecture* were not evaluated, because they were not applicable to the case study.

The quality is assessed quantitatively by measuring the impact of each transformation on size, complexity, and security (See Table 2). Size and complexity were selected because the transformations impact these explicitly. Size was measured by the number of components, the number of interfaces per component, and the number of actions per interface, while complexity was measured by CBMC [9], connector complexity [19], and McCabe [11]. *Security* was selected because our strategy should improve the LP properties, but not deteriorate other security properties such as the attack surface's size, which was measured by Manadhata's metric [10].

The first analysis examined whether system size (#comp) worsened, which was the case for the *split component transformation*: it almost doubled in size. Indeed, large systems require more conflicting permissions to solve, because (i) their components have more actions, and (ii) they support more tasks. The increase is not acceptable, although this number can be reduced by improving the transformations.

The second analysis examined whether component size (#inf/#comp and #acts/#inf) worsened, which was not the case. Indeed, if a *component is split*, the number of actions per interface decreases as a subset of these are moved to another component. The same is valid for *removing actions*.

The third analysis examined whether complexity (Mccb.) increased. For *splitting components*, this was the case. A possible explanation for this is that dependencies, one of the main parameters of complexity, between the old set of components and the newly created components are introduced. The increase is acceptable.

The fourth analysis examined how attack surface was impacted. Applying splitting components increases the attack surface, because it increases the number of indirect entry points by creating shared state update methods based on existing entry points. Applying remove unused actions, lowers the attack surface, because it removes methods and thus entry points. The other transformations do not impact this metric, because two parameters of attack surface, indirect entry points and untrusted data items, are not influenced.

In conclusion, we could say that the splitting transformation solves the majority of the violations, while the other transformations play a minor role. However, this splitting transformation has one unacceptable side effect: the system size almost doubles.

4.3 On the selected transformations

The 5 presented transformations have been selected out of a set of 25 possible transformations. This initial set has been defined systematically: by considering all possible transformations that may lower permissions (and thus privileges). Furthermore transformations should be feasible in that they must be implemented starting from existing architectural documentation. This section briefly documents the selection process.

	Security				Size			Complexity	
Metric Case	#viol comps (indirect)	# viol tasks (indirect)	#viol (indirect/other)	attack surface	#comp	#inf /#comp	#acts /#inf	#tasks	Mccb.
before	8 (7)	11 (11)	600 (396/204)		9	1.11	3	11	3
Spitting component	3 (0)	10 (0)	72 (0/72)	larger	15	1.73	1.69	11	5
Splitting permission	8 (7)	11 (4)	610 (438/172)	same	9	1.11	3	4	3
Removing actions	8 (7)	11 (11)	496 (348/144)	smaller	9	1.11	2.7	11	3

Table 2. Measurements of the application of the transformations in isolation.

The solution spectrum consists of strategies that impact the fundamental elements used to express the LP property. In our LP model, these elements are: permissions, tasks, users, components, and actions. Indeed, adherence to LP can only be introduced by modifying elements affecting the LP property. Possible transformations are enumerated by applying the following general modifications on the fundamental elements listed above: splitting, merging, removing basic elements from, adding basic elements to, and moving basic elements between elements.

This enumeration results in 25 possible transformations. Only 5 satisfy the three important conditions: (1) they might reduce the number of permissions, (2) they solve LP violations and (3) they can be implemented using existing architectural documentation as transformation context.

Twenty transformations therefore have not been selected for one of the following reasons:

The transformation increases the number of permissions. Extending tasks, adding actions to a component, adding tasks to a user, merging components, merging tasks, users, and actions require additional actions, and thus additional associated permissions. Such an increase is not expected to reduce the number of attributed permissions. Hence, these transformations are not good candidates to solve LP violations.

The transformations cannot be implemented by using architectural documentation. Several transformations might solve LP violations, but they cannot be implemented because the architectural documentation hinders the identification of their conditions. These transformations are: splitting tasks, removing an action tuple from a task, moving actions between components, removing a task from a user, splitting actions, and parameter and permission related transformations.

Notice that some smaller corner cases are not handled correctly by our LP model as it stands. The following transformations should not work, although our LP model states otherwise: merging tasks, merging users, and adding tasks to users. Indeed, merging two violating tasks or users that execute violating tasks will remove the violation, but grants more permissions to the user. Splitting users introduces new violations, although it is supposed to reduce the number of violations. Obviously these transformations have not been considered either.

5 Related Work

This work is strongly related to two research domains: security engineering, and software refactoring. Related work on security 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 approach provides a systematic and automated means for program separation at architectural level. Another more general approach is privilege separation [3], 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 [8], 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 [6] 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.

Execution monitoring is another technique that limits the privileges a program is assigned. These techniques block system calls and access to file and network resources based on policies. An examples is Systrace [14]. These mechanisms have two main 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 [17]. Second, these mechanisms add runtime overhead to limit the number of privileges, while we ensure that these are limited by construction.

A lot of work has been published in the area of software refactoring. Mens [12] presents a detailed survey. Software refactoring 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 our case LP. While refactoring can be applied, no concrete results for LP are available in this area.

6 Conclusions

This paper improved LP support in SA by proposing architectural transformations that solve LP violations. The transformations have been illustrated by means of a case study. Application of these transformations results in a lower number of violations and an improved average component size, while other software properties such as system size and system complexity detoriate. Splitting a component thus solves halve of the problems.

While this work is a milestone in this context, many LP problems remain unsolved. One step in this direction is applying the transformations on multiple (larger) case studies.

Acknowledgment

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

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] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall Englewood Cliffs, NJ, 1981.

[3] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[4] K. Buyens, B. D. Win, and W. Joosen. Identifying and resolving least privilege violations in software architectures. Technical report, Katholieke Universiteit Leuven, 2008.

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

[6] S. Höhn and J. Jürjens. Rubacon: automated support for model-based compliance engineering. In *Proceedings of the 13th international conference on Software engineering*, pages 875–878. ACM New York, NY, USA, 2008.

[7] M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, 2006.

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

[9] M. Lindvall, R. T. Tvedt, and P. Costa. An empirically-based process for software architecture evaluation. *Empirical Software Engineering*, 8(1):83–108, March 2003.

[10] P. K. Manadhata, D. K. Kaynar, and J. M. Wing. A formal model for a systems attack surface. Technical Report CMU-CS-07-144, Carnegie Mellon University, 2007.

[11] T. J. McCabe. A complexity measure. In *ICSE '76*, page 407. IEEE, 1976.

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

[13] M. Morandini, D. C. Nguyen, A. Perini, A. Siena, and A. Susi. Tool-supported development with tropos: The conference management system case study. In *AOSE 8*, pages 182–196. Springer, 2008.

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

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

[16] R. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32:40–48, 1994.

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

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

[19] S. M. Yacoub and H. H. Ammar. A methodology for architecture-level reliability risk analysis. *IEEE Transactions on Software Engineering*, 28(6):529–547, June 2002.