Identifying and Resolving Least Privilege Violations in Software Architectures

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

Abstract

The implementation of security principles, like least privilege, in a software architecture is difficult, as no systematic rules on how to apply them in practice exist. As a result, they are often neglected, which lowers the overall security level of the software system and increases the cost to fix this later on.

This paper improves the support for least privilege in software architectures by (i) defining the foundations to identify potential violations of the principle herein and (ii) elicitating architectural transformations that ameliorate the security properties of the architecture. These results have been implemented and validated in three case studies.

1 Introduction

Least privilege (LP) is a well-known security principle that survived the test of time. It prescribes that a user is not assigned permissions that he does not require and, consequently, he cannot execute tasks that he is not allowed to execute [17]. Security principles must be considered throughout the entire software development lifecyle [7]. While several techniques exist to reason about the adherence to LP in software, such as policy reasoning [18] or restructuring [3], no systematic method is available at the architectural level, where the consequences are significant [2].

This paper argues that, at the architectural level, LP minimizes the capabilities of a (set of) component(s) that is to be executed as a single controllable unit (typically a process). Two important factors in this context are (i) the controllable units and (ii) the access policy that is to be enforced thereon: LP will be best adhered to if both the architectural structure and the policy are adequate (See Figure 1). Indeed, as shown in the next section, it can be impossible to enforce LP with an inappropriate architecture (SA). Unfortunately, to the authors' knowledge, no systematic techniques exist today to deal with this problem. One of the factors that makes this problem challenging is the abstraction level of a software architecture and, hence, the limited amount of information available for reasoning about LP.

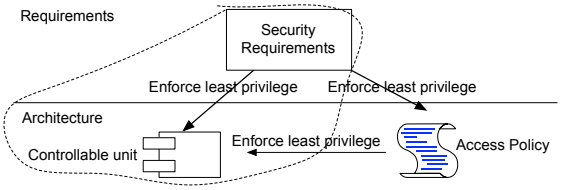

Figure 1. LP in SA: a combination of architectural structure and access policy.

The approach presented in this paper uses a list of use cases and an architectural description to predict LP problems in the final software product. The approach approximates a worst case assignment of permissions given the set of use cases. As presented in Figure 1 we are interested in the architectural structure (without taking into account the access policy), since this ensures LP adherence in the final software system. The computed assignment is used to determine (potential) violations of the principle[1], which are then solved by architectural transformations. The contributions of the paper are: (i) a theoretical underpinning of the meaning of LP in software architectures, (ii) an algorithm to identify violations of the principle, (iii) an architectural transformation to address a category of violations, and (iv) an implementation of the former in a tool.

The rest of this paper is structured as follows. Section 2 further motivates the problem by means of an example. Section 3 presents the formal foundations of the presented approach. Section 4 uses these to identify LP violations. Section 5 presents a transformation to solve a subset of these violations. Section 6 applies the approach on three case studies. Section 7 elaborates on the advantages, the drawbacks, and possible extensions of the approach. Finally, Section 8 discusses related work, and Section 9 concludes.

[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, some problems may persist in the final system.

2 Motivating Example

Consider an integrated groupware system that consists of three main components (See Figure 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, Tasks, 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 different users (i.e. security requirements).

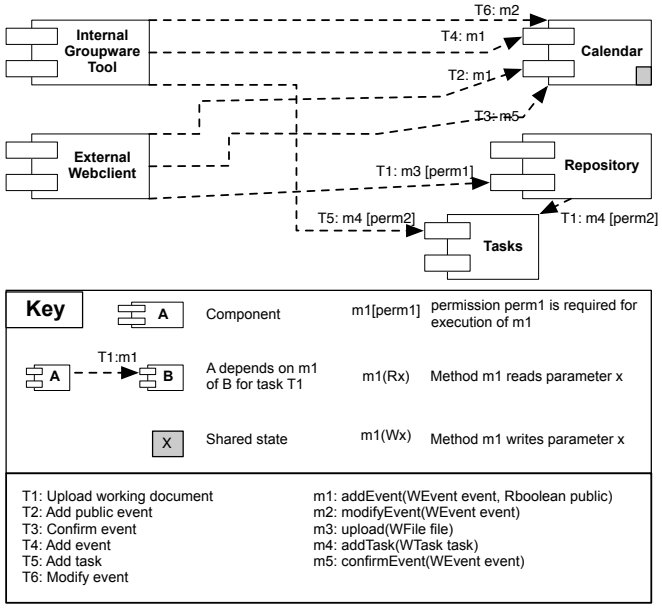

Figure 2. Excerpt from the component diagram of the an integrated calendar system

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 another task. For instance, if an external user has the right to *upload a working document* (perm1, perm2 for task T1), he will also be able to *add tasks* (perm2 for task T5), even if this is not desirable. More complex overlapping scenarios that are more difficult to identify manually can arise.

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 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, one 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 access policy: a restructuring of the components is often required to address these problems.

3 Theoretical underpinnings

In this section, a formal model for software architecture is introduced to define LP (and violations thereof) at architectural level. The model, inspired by work of Jie Ren [16], is modelled in set theory. Only the relevant subset is shown. The complete model is published as a report [4].

The model focusses on the software architecture's component and connector view. Every *component* c can be described in terms of the *actions*[2] of its interfaces, which are used to interact with the component. Such an interaction is

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

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. Tasks assigned to the users of the calendar system

built into the system to realize a task (or use case).

```
SA(s) = ⟨ components(s), dependencies(s), parents(s) ⟩
components(s) = { c_1, ..., c_k }
c = component = ⟨ actions(c), interfaces(c) ⟩ ∈
  components(s)
interfaces(c) = { I_1 ∪ ... ∪ I_m }
∀ I ∈ interfaces(c): actions(I) = { a_b, ..., a_j }
actions(c) = { a | a ∈ actions(I) and I ∈ interfaces(c) }
actions(s) = { a | c ∈ components(s) and
  a ∈ actions(c) }
a = ⟨ name(a), params(a) ⟩ ∈ actions(c)
params(a) = { param_1, ..., param_n }
param = ⟨ name(param), type(param) ⟩
type(param) ∈ { String, int, ... }
users(s) = U = { u_1, ..., u_q }
tasks(s) = T = { t_1, ..., t_r }
task = { (a_1,a_2),..., (a_p,a_p+1)} [3]   ; a_i ∈ actions(s)
UT = { (u_i,t_j), ..., (u_i+f,t_j+g) } ⊆ U × T
actions(task) = { a | a ∈ actions(s) and
  x ∈ actions(s) and ((x,a) ∈ task or
  (a,x) ∈ task) }
tasks(c) = { t | t ∈ T and a ∈ actions(c)
  and a ∈ actions(t) }
users(c) = { u | (u,t) ∈ UT and t ∈ tasks(c) }
```

We now focus on security related definitions. A component may personate multiple *subjects* (and *principals*) during its lifetime, because it may be used by different users to perform a range of (related) tasks. For now, we assume that a component represents exactly one subject.

A *permission* is a representation of the right to perform a set of actions - components are resources of the system. The permissions necessary to execute a task is the union of all permissions needed for the actions that constitute the task.

```
AP(s) = perm_1, ..., perm_w
perm = { a_1, ..., a_v } ⊆ actions(s)
permissions(action) = { p | p ∈ AP(s) and
action ∈ p }
permissions(task) = { p | p ∈ permissions(action)
  and action ∈ actions(task) }
```

A distinction was made between three types of permissions that will be used for reasoning about LP: required, internal, and indirect permissions.

The *required permissions* of a component are the permissions that it needs to successfully complete the tasks (or parts of tasks) it is responsible for. In other words, they are the permissions associated with the actions the component depends on for the tasks it provides. A Component is assumed to require permissions for all the actions in the task that follow his own contribution to the task. For instance, the External Web Client requires permissions 1 and 2 for uploading a working document (See Figure 2). Note that this set is actually equivalent to an ideal access policy.

```
reqperms(c) = { p | p ⊂ usedactions(c) }
usedactions(c) = { a | t ∈ tasks(c) and
  a ∈ actions(t) and a ∉ actions(c) }
```

[3]This is an ordered set.

The *internal permissions* of a component are the permissions linked to its own interfaces. A component is implicitly granted the permissions to execute all its interfaces' actions.

```
internperms(c) = { p | p ⊂ ownactions(c) }
ownactions(c) = { a | t ∈ tasks(c) and
  a ∈ actions(t) and a ∈ actions(c) }
```

The *indirect permissions* are permissions that might be obtained by interfering in a component's shared state. This is a component's internal state that is used by its actions. Since an action might influence another one by changing the shared state on which the other is dependent, a task can actually influence the results of another task. This is examplified in Section 2. In the case of shared state, a component is attributed permissions from another component's task that is reachable via the shared state of the latter.

```
indirectperms(c) = { perm | c2 ∈ reachablecs(c)
  and t ∈ sharedTasks(c2,c) and
  t2 ∈ tasks(c2) \ sharedTasks(c2, c) and
  sharedState(t,t2,c2) ≠ ∅ and
  (perm ∈ internperms(c2) ∪ reqperms(c2)) and
  perm ∈ permissions(t2) }
reachablecs(c) = { c2 | c2 ∈ components(s) and
  t ∈ sharedTasks(c, c2) and c2 ≠ c and
  before(t,c,c2) }
sharedTasks(c1, c2) = { t |
  t ∈ tasks(c1) and t ∈ tasks(c2) }
before(task, c1, c2) ⇔ ∃ a1,a2 ∈ actions(task)
  and a1 ∈ actions(c1) and a2 ∈ actions(c2)
  and before(a1,a2,task) }
before(a1,a2,task) ⇔ ∃ (a1,a2) ∈ task or
  (∃ (a1,x) ∈ task and before(x,a2,task))
perms(c) = indirectperms(c) ∪ internperms(c)
  ∪ reqperms(c)
```

We now come to the definition of LP. A system adheres to LP if all its components adhere to LP. A component does not adhere to LP if it, based upon the permissions attributed as described before, is capable of executing tasks it is not responsible for.

```
adherestolp(s) ⇔ ∀ c ∈
  components(s) | adherestolp(c)
adherestolp(c) ⇔
  ∀ t1, t2 ∈ executabletasks(c) |
  t1, t2 ∈ tasks(c) and executabletogether(t1,t2)
executabletogether(t1, t2) ⇔
  ∃ u1 ∈ U | (u1, t1), (u1, t2) ∈ UT
executabletasks(c) = { t | t ∈ T and
  ∀ perm ∈ permissions(t): perm ∈ perms(c) }
```

This definition is based on the fact that we assume that all behavior is executed via tasks and that we thus can relate components to users in the following way. The permissions attributed to a component determines the tasks that can be executed by this component. These tasks are related to users via the user-task assignment set. As such, a component will act on behalf of a possible set of users.

4 Identifying LP Violations

For the identification of LP violations, an algorithm was constructed based on the formal model. The input provided to the algorithm is threefold: a component and connector diagram, a user-task assignment list, and sequence diagrams mapping these tasks onto the diagram. The algorithm (See Figure 3) iterates over the components, attributes permissions to each component, and verifies whether each component violates LP. The output of the algorithm lists the LP violations, specifying the violating component, the permissions causing the violation, and the conflicting tasks.

```
For each component in components(s)
  assign int. perms based on component actions
  For each task in tasks(c)
    propagate req. perms upward in task
    calculate ind. perms for downward components in task

For each component in components(s)
  determine LP violations based on assignment
```

Figure 3. Identification algorithm

The algorithm can been parameterized with several options, two of which will be discussed here. A first option is the order in which the components will be iterated over. This order controls the assignment of indirect permissions, because these are propagated via the shared state of other components. Our algorithm currently starts with calculating permissions for "leaf node" components. Next, it follows these tasks in reverse order to determine the one-but-last components to calculate the permissions of, and so forth. Other propagation strategies such as root-node propagation could be considered as well.

Another option is the shared state approximation strategy. This determines the shared state between tasks in a component and thus controls the indirect permissions that are assigned to them. At this moment, shared state is determined by equivalence of name and type of the parameters of an action: if two methods share an equivalent parameter, they are considered capable of influencing each other. This is a rough approximation of the shared state.[4] This is illustrated in Figure 2, where the shared state of methods m1, m2, and m5 in component Calendar is approximated by the event parameter they all have. Other strategies can use semantic method annotations (such as pre- and postconditions) or explicit annotations regarding shared state.

5 Resolving LP Violations

Different strategies exist to accommodate LP in a software architectural structure, among which: (i) splitting components into several isolated units and lowering privileges assigned to these units, (ii) rewiring the architecture (i.e. rerouting tasks to other components in order to split up conflicting privileges), (iii) splitting tasks such that less privileges have to be attributed to different components, or (iv) applying well-known solutions (such as sandboxing) to introduce LP in selected parts of the architecture (see Section 8). Due to space constraints, we only elaborate on the first strategy. The others have been described in a report [4].

One of the challenges in splitting a component is that it has to be split in a way that preserves the semantics of the component: semantically related actions must remain adjacent after splitting. The knowledge available for splitting is typically limited to the interfaces of the component, the actions described in these, and the parameters of these actions. Our approach uses parameters to approximate related actions by applying a variant of the shared state approximation strategy on all possible subsets of the component's actions. However, in order to split a component that contains related actions which are used by violating tasks, we require extra information in the architectural description: read/write on the action's parameters (see transformation).

5.1 Splitting a component

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

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

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

Based on the model of section 3, it is easy to see that the privileges 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

[4] Actually, in case of parameters of primitive types, it probably does not make much sense to use this type of approximation.

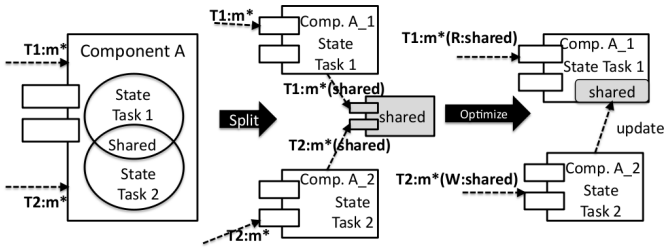

Figure 4. A component with overlapping read methods and a write method can be split.

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 indirect permissions of that other component is lower or equal than that number before splitting.

Note that the above strategy can lead to multiple solutions, because a set of actions can be partitioned in multiple ways. Therefore, a strategy is needed that searches the best possible solution. Such a strategy ranks the identified violations, which can be based on several metrics. For instance, solve the violation whose solution does not significantly change other architectural qualities such as size or complexity first (See also Section 6).

6 Applying the results on several case studies

This section presents the validation results of the approach. The goal was to assess whether the LP properties of the software architecture improved, while other software architecture qualities such as size, complexity, or security properties did not deteriorate. The evaluation was based on an implementation of the techniques described earlier.

In the rest of the section, one 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 the results of applying the approach to several case studies will be discussed from a broader perspective.

6.1 Detailed validation results

The case study used in this section involves a subset of a digital publishing system [4]. The system automates the workflow of a publishing company. Its main features are input, user, and content management.

Different actors make use of this system, among which the advertiser, the journalist, and the manager. The *advertiser* buys commercial space. The *journalist* uses the system to distribute finished content. The *manager* manages the company by creating a publishing strategy and assigning tasks to journalists.

The execution of each task is restricted to certain actors. For instance, a journalist is not allowed to create a corporate strategy, because that requires too much responsibility.

In the architecture, components responsible for these features are the following. An Media Advertising System (MAS) is used by Advertisers to submit produced commercials to be stored in the Content Management System (CMS). The CMS is responsible for storing and retrieving content, and a Journalist News Desk is responsible for making content ready to be published. The Planning System (PS) is used by both Journalists and Managers to manage their planning and assign tasks.

We discuss some of the problems that have been identified in the Planning System. A first problem is that the Planning System is responsible for the *plan corporate strategy* and *plan edition strategy* tasks. The former is executed by a manager, while the latter is executed by a journalist. Having permissions for both tasks was explicitly forbidden by the company policy (as illustrated in Figure 5).

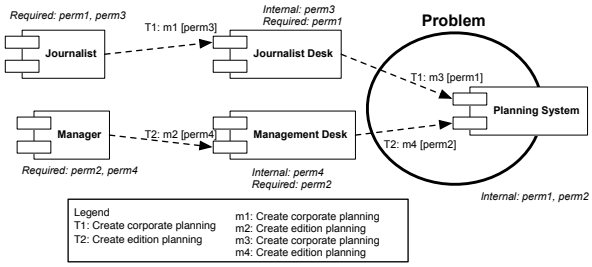

Figure 5. A violation caused by too many internal privileges.

A second problem is that the advertiser is able to obtain enough permissions to modify the advertisement workflow (part of planning tasks) (See Figure 6). The design problem is that planning system is responsible for both *notification* and *edition planning* tasks.

A solution to these problems can be found by applying the transformation described in Section 5. The first problem can be solved by splitting the component in two parts. The first part contains actions related to corporate strategy, while the second part contains actions related to edition planning. The second problem can be solved by decoupling notification from creation. As such, the risk of LP violations in the final software architecture will be greatly reduced.

6.2 Broader validation results

In this section, the quality of the presented approach is assessed quantitatively. For this purpose, the impact of our algorithms on size, complexity, and security was mea-

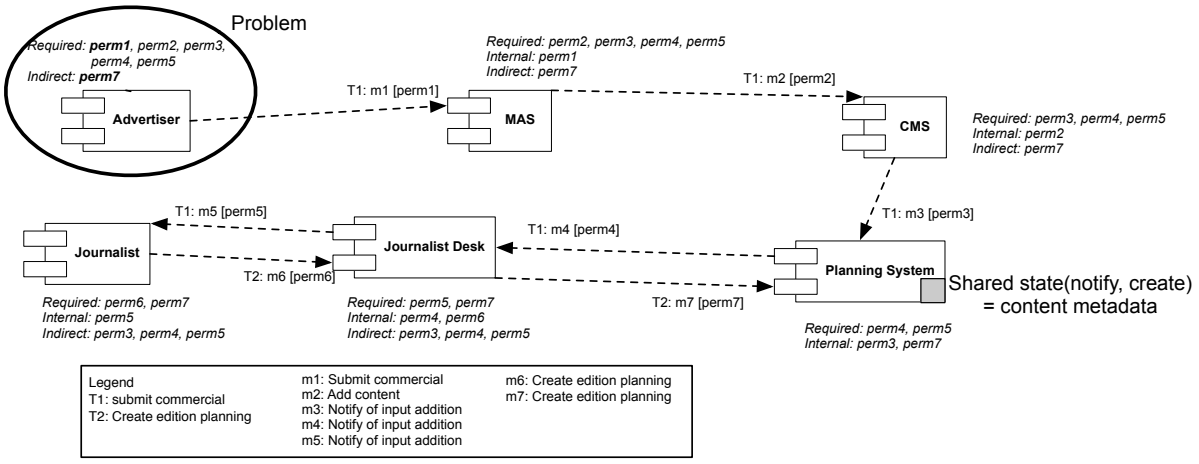

Figure 6. A violation caused by too many indirect privileges.

sured. Size and complexity were selected because our solution strategy impacts these explicitly: it creates new components, introduces additional dependencies, and so forth. 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 [10], connector complexity [20], and McCabe [12]. *Security* was selected because our strategy should improve the LP properties of a software architecture, but not deteriorate other security properties such as the size of the attack surface. LP was measured by the number of violating components, and the number of tasks causing the violation, while the attack surface was measured by a (simplified version of a) metric defined by Manadhata [11].[5] Table 2 presents the results of the application of the approach on three architectures from three different domains: a modified version of the small chat application delivered with ArchStudio [5] (case1), a conference management system [14] (case2), and (a subset of) the publishing system [4] (ps sub and ps full).

The first analysis examined whether system size (#comp) worsened after the application of our approach. In general, this was indeed the case. While the smaller case studies growed in size with 1 or 2 components, the publishing case almost doubled in size. Indeed, large systems might require more conflicting permissions to solve, because (i) their components have more actions, and (ii) they support more tasks. The increase for small systems is still acceptable, while the increase for the large system is not. Note, however, that this number can be reduced by lowering the number of false positives in the violation set or by improving the architectural transformations.

The second analysis examined whether component size (#inf/#comp and #acts/#inf) worsened after the application of our approach, 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 third analysis examined whether complexity (Mccb.) increased. In general, 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. This increase in complexity is considered acceptable.

The fourth analysis examined whether LP improved (in terms of the number of violating components (# viol comps), and tasks causing a violation (#viol tasks) after the application of our approach. In general, LP did improve. However, in two cases (case2 and ps full) an improvement was not noticed. This might be caused by the inability of the transformation to identify a set of subcomponents that each contain a different violating action. These numbers are remarkably high, which might indicate, among others, that the approximation of indirect permissions produces a significant number of false positives.

The fifth analysis examined how the attack surface was impacted. In general, this was not the case, which is plausible since two of the main parameters of attack surface, indirect entry points and untrusted data items, are not influenced by the transformation. However, in case 2 the attack surface increased, because it increases the number of indirect entry points by creating shared state update methods based on existing indirect entry points, which count as indirect entry points as well.

In conclusion, we could say that our detection algorithm detects least privilege violations, but has a high rate of false

[5]The original metric requires pre- and postconditions associated with the actions, which our case studies do not provide. Furthermore, two of the metric's parameters, trusted data items and direct and indirect entry points, are estimated by making use of the tasks (data flow).

	Size			Complexity		Security		
Metric Case	#comp	#inf /#comp	#acts /#inf	#tasks	Mccb.	#viol comps (indirect)	# viol tasks (indirect)	attack surface
case 1 before	3	1.67	1.2	2	3	1 (0)	2 (0)	
case 1 after	4	1.5	1	2	4	0 (0)	0 (0)	same
case 2 before	7	1.0	2.9	11	4	6 (5)	11 (8)	
case 2 after	9	1.1	2.1	11	6	8 (6)	11 (8)	larger
ps sub before	13	2.38	2.65	8	6	4 (0)	6 (0)	
ps sub after	20	2.6	1.58	8	7	0 (0)	0 (0)	same
ps full before	13	2.38	2.64	22	8	13 (12)	22 (16)	
ps full after	13	2.38	2.64	22	8	13 (12)	22 (16)	same

Table 2. Measurements of the cases in the three domains and three variants of the publishing case.

positives. Our splitting transformation works if the components that have to be split, are divisible in subcomponents, which mainly depends on the size of the shared state.

7 Discussion

A number of observations driven by the results of our experiments are worth further discussion. The approach and transformations have at least the following limitations.

The identified transformations do not always work well: sometimes it is not possible to split the component in subcomponents, because two violating tasks both use an action or a group of actions to update shared state. Hence these actions can not be be part of two components. Possible solutions are (i) using smarter ways for creating subcomponents (e.g., another shared state identification algorithm), and (ii) creating transformations that are not based on splitting components to solve violations, like splitting tasks.

On the other hand, one should also carefully ponder the options that influence the result of the identification and solving algorithm, since naively applying it might lead to extreme architectures or might cause unwanted side affects. An extreme architecture is for instance an architecture in which every component has been split (possibly several times). It is clear that such architectures are not useful in practice. Note also that sometimes a component should not be split when many tasks passes through these components on purpose. Some security-specific components (like the Audit Interceptor) exhibit this type of behavior.

Some current assumptions considerably limited the practical applicability of the approach. First, a process typically consists of multiple components, while we assume that it only consists of one. This assumption can be dropped if other architectural views such as the process or runtime view is incorporated in our model. Second, not all security relevant use cases are typically available, while we assume they are. Furthermore, there are also two limitations in the current implementation tool: (i) no support for UML (useful for reading architectural diagrams), and (ii) no meaningful names for newly created components and interfaces.

An interesting added value of the approach is the ability to reason about separation of duty (SoD) policies at architectural level. Indeed, SoD deals with splitting and distributing tasks that, when combined, can be harmful. SoD can be seemlessly integrated in the approach as specific constraints that are imposed on particular tasks. As such, one can easily reason about SoD conflicts at architectural level.

8 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 with clean interfaces, has been successfully applied in several end-user programs such as qmail [1] to support LP. Our approach provides a systematic and automated means for program separation at architectural level. Another more general approach is privilege separation [3], a technique that partitions the implementation of 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 his UMLSec approach to enforce LP by formulating LP requirements and verifying UMLsec specifications (including policy 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.

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. Examples are Systrace [15], and Jain's system call interposition [8]. The main drawback of these mechanisms that 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 [19]. Another drawback is that the sandbox is assigned a lot of privileges.

A lot of work has been published in the area of software refactoring. Mens [13] 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 behaviour. This improvement of the internal structure can be based on a specific quality goal, such as modifiability, security, or in our case least privilege. While refactoring can be applied, no concrete results for LP are available in this area to the knowledge of the authors.

9 Conclusion

This paper proposes a technique that improves the identification and resolution of LP violations in software architectures. To this aim, the concept of architecture-level LP has been modelled formally. This model was used to create an algorithm that indicates when an architecture violates LP. Subsequently, an architectural transformation that solves a subset of these violations was proposed. The approach has been validated by means of several case studies which indicate that, overall, properties such as the number of violations and the average component size improve, while other software properties such as system size, system complexity, and attack surface are negatively affected.

While this work is a first milestone in this context, many opportunities and issues remain. The focus of future work will be threefold. First, the false positive rate should be further reduced such that the outcome of the violation identification algorithm is more usable. A first idea in that direction is to split the different types of permissions and work with weighted violation detection. Second, other architectural views can be included in order to make the technique more applicable. Finally, architectural refactorings that preserve the architectural semantics, possibly aided by architectural annotations, can be significantly improved. In that context, the alternatives described in the paper will be further studied, implemented and tested throroughly.

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. Qmail home page. `http://cr.yp.to/qmail.html`.

[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] E. Dashofy, H. Asuncion, S. Hendrickson, G. Suryanarayana, J. Georgas, and R. Taylor. Archstudio 4: An architecture-based meta-modeling environment. In *ICSE '07*, pages 67–68. IEEE Computer Society, 2007.

[6] S. Höhn and J. Jürjens. Rubacon: automated support for model-based compliance engineering. In *ICSE '08*, pages 875–878, 2008.

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

[8] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *ISOC*, 2000.

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

[10] 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.

[11] 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.

[12] T. J. McCabe. A complexity measure. In *ICSE '76*, page 407. IEEE Computer Society Press, 1976.

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

[14] 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 VIII*, volume 4951 of *LNCS*, pages 182–196. Springer, 2008.

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

[16] J. Ren. *A connector-centric approach to architectural access control*. PhD thesis, 2006.

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

[18] R. S. Sandhu and P. Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 1994.

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

[20] 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.