Improving Least Privilege in Software Architecture by Guided Automated Compartmentalization

Koen Buyens, Bart De Win and Wouter Joosen

IBBT-DistriNet, Department of Computer Science, K. U. Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract. Security principles, like least privilege, are among the resources in the body of knowledge for security that survived the test of time. Support for these principles at architectural level is limited, as there are no systematic rules on how to apply the principle in practice. As a result, these principles are often neglected since it requires a lot of effort to apply them consistently.
This paper addresses this gap for the principle of least privilege in software architecture by elicitating architectural transformations that positively impact the least properties of the architecture, while preserving the semantics thereof.

1 Introduction

Security principles, like least privilege, are among the resources in the body of knowledge for security that survived the test of time. They have been introduced by Saltzer and Schroeder [10] more than 30 years ago and can therefore be considered mature. Least privilege, for instance, is a popular security principle that has been studied in a multitude of contexts. Several techniques have been developed to verify or enforce least privilege in software such as model checking [1], sandboxing [6], program separation [7] and so forth. These techniques typically focus on requirements specification, detailed design, or software deployment. At the architectural level, the availability of techniques to improve least privilege is more limited: most results at this level are testimonies that witness the usefulness and the applicability of the principle (e.g., qmail [2]).

At the architectural level, least privilege can be considered as the minimization of the capabilities of a set of (possibly dependent) components executing within a single checkable unit (typically a process) according to a specific least privilege policy. In other words, the principle is not well supported, if there is a single process that is responsible for executing all the functionality of the system. Unfortunately, even in case least privilege is optimized at requirements level according to available techniques, the mapping onto a software architecture might introduce security problems if the latter is not well prepared for this task. Separate privileges in requirements might for instance have to be assigned to single components in an architecture. To the knowledge of the authors, no systematic methods exist to address this problem.

One of the biggest issues in this context is the lack of detail and semantics within a typical architectural description. A software architecture is structured as a set of

Buyens K., De Win B. and Joosen W. (2008).
Improving Least Privilege in Software Architecture by Guided Automated Compartmentalization.
In *Proceedings of the 6th International Workshop on Security in Information Systems*, pages 145-150
DOI: 10.5220/0001741501450150
Copyright © SciTePress

black-box components (and possibly sub-components), coupled by means of dependencies (or connectors). Some semantics of the system can be specified by means of (pre/post)conditions on methods and by means of collaboration diagrams[1]. However, the internals of components and, hence, the exact flow of information and control is largely unspecified at the architectural level. Consequently, one of the challenges is to identify useful optimizations and modifications at the architectural level that preserve the semantics of the software architecture and that have a positive impact on the least privilege properties thereof. Hereby, the authors want to work on the 'safe' side by focussing on techniques that are *likely* to preserve the semantics as much as possible. The goal of this paper is to provide a systematic and implementable technique to both analyze and support the principle of least privilege in a software architecture. The actual contributions are threefold: (i) the definition of the architectural meaning for least privilege, (ii) the elicitation of two architectural transformations that have a positive impact on the least privilege properties of a software architecture, and (iii) a method for applying these transformations in a structural way. Preliminary results indicate that it is possible to identify such useful optimizations and modifications at architectural level.

The paper is structured as follows. Section 2 presents our proposed approach for creating and applying such transformations. Section 3 elaborates on the drawbacks, the advantages, and possible extensions of our approach. Section 4 discusses the status of related work. Finally, section 5 concludes.

2 Approach

This section briefly explains the approach for identifying and solving some least privilege violations in a software architecture. First, the general idea is outlined. In short, the approach iteratively applies a number of transformations, until no further least privilege violations can be detected. Finally, one specific transformation for detecting and solving least privilege violations is presented. This transformation makes use of the component and connector diagram and the collaboration diagram.

2.1 General Idea and Approach

In order to make the approach practically feasible, three assumptions have been made: (i) each component executes within one checkable unit (a process); (ii) actors and use cases are assumed to be sound and hence, fixed, meaning that our approach does not partition, create, remove, or merge them; (iii) all security-relevant use cases are available. Abusing such use cases will violate critical security requirements and thus damage the core business goals. Our approach will not perform well if not all security relevant use cases are available, because it heavily relies on them for performing its operations.

In order to determine when a component executes too many tasks and thus needs too many privileges (for accomplishing these tasks)(See definition in Section 1) a *least privilege policy* is introduced. The policy is an ordered set of use cases that categorizes

[1] A collaboration diagram represents a use case of the system. In the rest of this paper, the authors use the terms use case and collaboration diagram interchangeably.

each use case as high, medium, or low, based on the business importance of that use case. An architectural component *violates* least privilege if the component is used by at least two highly ranked use cases that are executed by a different actor. Thus, the component is not allowed to have the privileges (methods) that are specified in both use cases.

The approach is iterative and identifies the architectural elements that violate least privilege by making use of the least privilege policy, the component and connector diagram, and the collaboration diagram. It resolves this violation by modifying the violating elements. If multiple modifications are possible, the best one is picked. Best is determined by the (ordering of the use cases in the) policy itself and guided by the architect.

2.2 Architectural Transformations

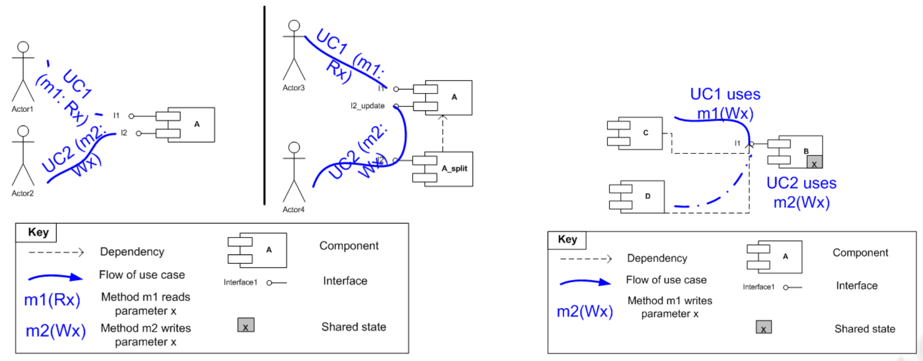
Different strategies exist to accommodate least privilege: (i) one can split a process into many isolated units and lower the privileges assigned to these units, (ii) one can rewire the architecture such that less privileges have to be attributed to different components, or (iii) one can apply well-known solutions (such as sandboxing) to introduce least privilege in selected parts of the architecture. In this paper, the authors investigate the first strategy and study how well it supports least privilege. The authors identified the following architectural transformations: one for splitting an interface and one for splitting a component. Due to space constraints, only the second transformation is described.

Transformation 1: Split a Component

Rationale. Components that execute too many tasks (and thus require to many privileges), are split in smaller components that can be assigned less privileges than the original component.

One of the challenges of this transformation is to split the component in a way that preserves the semantics of the component (see Section 1), because the knowledge available for splitting is typically limited. The only information that can be used for splitting the component is: the interfaces of the component, the methods described in these interfaces, and the parameters of these methods. Our approach uses these parameters to approximate the methods that are likely to depend on each other and, hence, should be safeguarded from splitting in the following way. If two methods operate on the same parameter(s) then there might be some dependency depending on exact the same use of these parameters (e.g. read or write). For this purpose, this transformation requires extra information to be present in the architectural description: read/write on the method's parameters. This is illustrated in Figure 1(b). Suppose the first use case uses a method provided by one of the interfaces of component B that uses parameter x and the second use case uses a method provided by the interfaces of the same component and modifies (writes) the same parameter x, then the shared state is considered to be parameter x.

Conditions and Transformation. If two use cases make use of a component and these use cases *violate* least privilege, then the components are split in the following way. First, identify the state of the component shared between both use cases.



(a) A component with overlapping read methods and a write method can easily split.

(b) A component with two write methods should not be split.

Fig. 1. Transformation 2 applied on a component is useful in certain cases.

1. If the shared state is empty (and both use cases use different methods), split the component in two disjunct parts by moving the interfaces/methods one use case uses to (a) new interfaces in a new component. Also modify this use case to use the new component instead of the old component.
2. If the shared set is not empty, and if one use case reads state that is written by the other use case, then create a new component containing a copy of the methods/interfaces of the writing use case. Also add a new interface for each interface of the original component that contains write methods on the shared state. These interfaces can be seen as additional interfaces that update the shared state. Extend the use case that reads to include the update methods provided by the new component, add a dependency between the old component and the new component, and modify the use cases updating the shared state (including the writing use case) to use the new component instead of the old component.

2.3 Validation

The authors have not performed an in depth validation of the approach, but studied its behavior by applying it on a case study of a significant size: a publishing system[8].

Application of the algorithm tackles the core least privilege problems of the case study: one component was responsible for many high-ranked use cases (Planning System), while two interfaces (*User Profile* and *CMS Input Interface*) were used by many different actors. Therefore, they have been split. The other components and interfaces have not been transformed mainly because (i) the components were only used by one actor, or used by multiple actors of which the uses cases were allowed to be executed together, and (ii) the other interfaces were typically used by one actor.

3 Discussion

Our preliminary results confirm that it is possible to identify and express useful changes at the architectural level. In the end, however, one can never have full guarantees that a particular change is actually a desired change (that preserves the semantics) and, hence, the software architect will always have to evaluate the end result.

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

There should be better techniques for identifying subcomponents. For instance, sometimes it is not possible to split the component in subcomponents each having a coherent subset of privileges, because such coherent subset can not be found by the shared state method.

The transformations should not modify other software engineering properties of the architecture. For instance, naively applying the rules might lead to an extreme architecture having a component for every privilege.

Applying the transformations in a different order, will result in different solutions, among which the architect has to identify the best one.

4 Related Work

Although many papers discuss the principle of least privilege, few of them discuss the principle in an architectural design context. The discussion on related work focusses on (i) program separation techniques, (ii) model checking techniques, and (iii) confinement techniques.

Separating a program in multiple processes with clean interfaces in order to assign privileges in an optimal way is a design technique that influences least privilege. It has been successfully applied in several end-user programs, such as Vsftp[4], qmail[2], and postfix[11]. Another more general approach is privilege separation[3][7], a technique that partitions an existing program into two processes: a privileged program called the monitor and an unprivileged program called the slave.

Model checking techniques can be used to verify whether a design has certain properties, such as least privilege. Jürjens explains in his PhD thesis[6], how one can use his UMLSec approach to enforce least privilege by formulating least privilege requirements and verifying UMLsec specifications with respect to these requirements. Thuong Doan's UML model checking approach is similar[5].

Confinement or sandboxing is another technique that limits the privileges a program is allowed to have. These techniques block system calls and/or access file and network resources. Example are systrace[9], Mapbox[1], and Janus[12]. The main drawback of these mechanisms that it is hard to specify policies in terms of application-specific resources and functions, because these resources and actions don't always map on files and system calls. An example of such an application specific resource is the global balance of a banking company. Another drawback is that one component, the sandbox, has a lot of privileges.

5 Conclusions and Future Work

This paper proposes a technique that improves support for least privilege in a software architecture. This technique presents conditions that indicate when an architecture does not adhere to least privilege and propose architectural transformations for solving this violation. Other conditions and transformations for identifying and solving least privilege violations can be found by dropping the assumptions the authors made, or by using one of the other strategies for accommodating least privilege.

The main challenge of enforcing least privilege at architectural level is that it is hard to find good architectural changes that solve least privilege violations and that do not change the semantics of the architecture, because the knowledge to make such changes is limited at architectural level. In the future, our work can be further refined to address the issues discussed in the paper. New transformations for the other two strategies accommodating least privilege should be identified as well as transformations changing actors and use cases. Next, our work should be validated. Finally, our work will be extended to (i) other activities of the (secure) software development life-cycle, and (ii) to other security principles. Metrics should be identified to (i) guide the architect in selecting the software architecture best adhering to the principle and having the right semantics, and to give an indication that the resulting software is indeed more secure.

Acknowledgements

Research for this paper was sponsored by IBBT, the Interdisciplinary institute for BroadBand Technology.

References

1. A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. Technical report, Santa Barbara, CA, USA, 1999.
2. Daniel Julius Bernstein. Qmail home page.
3. David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.
4. Chris Evans. Comments on the Overall Architecture of Vsftpd, from a Security Standpoint. Internet, February 2001.
5. Thuong Doan, Steven Demurjian, T. C. Ting, and Andreas Ketterl. Mac and uml for secure software design. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 75–85, New York, NY, USA, 2004. ACM.
6. J. Jürjens. *Secure Systems Development with UML.* March 2004. To be published.
7. Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.
8. Dimitri Van Landuyt, Johan Grégoire, Sam Michiels, Eddy Truyen, and Wouter Joosen. Architectural design of a digital publishing system. Technical report, October 2006.
9. Niels Provo. Systrace - interactive policy generation for system calls.
10. Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
11. Wietse Zweitze Venema. Postfix home page.
12. David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, 12, 1999.