

On the importance of the separation-of-concerns principle in secure software engineering

Bart De Win Frank Piessens Wouter Joosen
Tine Verhanneman
Katholieke Universiteit Leuven, Dept. of Computer Science

September 30, 2002

Corresponding author:

Frank Piessens, Dept. of Computer Science, K. U. Leuven,
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
Tel. +32(0)16 327603, Fax. +32(0)16 327996
E-mail: Frank.Piessens@cs.kuleuven.ac.be

Abstract

The separation-of-concerns principle is one of the essential principles in software engineering. It says that software should be decomposed in such a way that different “concerns” or aspects of the problem at hand are solved in well-separated modules or parts of the software. Yet, many security experts feel uneasy about trying to isolate security-related concerns, because security is such a pervasive property of a piece of software. And in fact, separating security-related concerns such as access control, or defensive input checking, is indeed very hard to achieve with current software engineering techniques.

While the authors fully agree with the observation that security is a pervasive property, they argue in this position paper that attempts to separate security aspects from other aspects of an application (even though in many cases not completely successful) are a necessary means to raise the security level of most applications. The two main arguments are: increased flexibility of the security mechanisms (leading to easier adaptation to unanticipated or evolving risks), and better-focused efforts of the few security experts in the development team, leading to fewer security design and implementation errors.

1 Introduction

The goal of this workshop is to investigate and discuss the fundamentals and principles of secure system engineering. A secure system results from many ac-

tivities, including architectural design, system integration, system configuration and software development. This paper focuses on the security challenges in the software development part. It is well-recognized that software vulnerabilities are one of the main causes for security breaches, and hence ways to enhance the software development approach in the light of security objectives are required to build more secure systems.

More specifically, this paper positions the separation-of-concerns principle as an essential means to build secure software. In short, this principle says that software should be decomposed in such a way that different “concerns” or aspects of the problem at hand are solved in well-separated modules or parts of the software. While well accepted in other areas, this principle is not so popular in the security field. In security-related textbooks and papers, it is often stressed that security is a pervasive property of a system, and that it is impossible to separate security-aspects from the functional parts of the system. It is impossible to “bolt on security” to an application after the fact. While the authors fully agree with the observation that security is a pervasive property, they argue in this paper that attempts to separate security logic from functional logic (even though in many cases not completely successful) are extremely important to raise the security level of most software. In fact, we will argue that this pervasiveness of security is one of the reasons why security is so hard to get right, and hence we should look for techniques to “reduce” this pervasiveness.

The structure of this paper is as follows: first, we analyze some causes of the current security software design problems in some detail. Then, after discussing the new techniques in software engineering that allow for better separation of concerns (or at least promise to do so in the future), we argue the applicability and relevance of these techniques to security. Finally, in a discussion section, we consider advantages as well as possible drawbacks or problems of our approach.

2 Causes for software security problems

It would be naive to think there is some kind of silver bullet that will solve all software security problems at once. In this section, we present a rough analysis and classification of causes of software security problems. For some of these, advanced separation of concerns techniques will bring (partial) solutions, while others remain just as problematic. In other words, this section clarifies the kinds of problems we hope to solve by applying the separation of concerns principle.

2.1 The pervasiveness of security

An important reason why well-known security vulnerabilities keep on reappearing is the fact that they can appear anywhere in a program. You cannot easily isolate these issues and put a security expert on them. All developers need some form of security expertise to avoid making these mistakes, and unfortunately it is unrealistic to expect security expertise from all developers on a large software project.

One can distinguish two kinds of pervasiveness. We discuss them below. We will show in the paper how advanced separation of concerns techniques can be useful to reduce both kinds of pervasiveness.

Pervasiveness: secure coding As a first form of pervasiveness, implementing application functionality (not directly related to security at first sight) incautiously can introduce severe security problems (cfr. the typical input validation or buffer overflow problems). *All* code must be written securely in this sense, and this clearly makes security a pervasive issue. Sometimes, such problems can be avoided by compiler or run-time support (e.g. no buffer overflows are possible in a garbage collected, memory safe language), or by enforcing certain coding rules on programmers. This shifting of the concern to a compiler, run-time, or coding guide can significantly increase security, but will never solve all problems of this kind. For example, arbitrary canonicalization errors or race conditions can not be weeded out in this way.

Coding securely requires a kind of defensive attitude from the programmer: he must think not only about how to realize the required functionality, but also how to make sure that his code cannot be abused to achieve undesirable results. The kinds of bugs that can lead to security problems can be very varied and very intricate.

Pervasiveness: security related concerns are crosscutting A second form of pervasiveness has to do with code that is introduced into the application for the purpose of specific security requirements. The problem here is the structural difference between application and security logic. For example, code to write relevant events to an audit log, or code that realizes a certain access control model is often spread among many classes. In that sense, security is a *crosscutting concern* since it cuts across many classes or functions in the program.

Attempts to modularize security concerns as good as possible have been going on for many years. Current software designs already succeed well in modularizing security mechanism implementations. For instance, in the Java Security Architecture, the SecurityManager/AccessController doing the stack walk is well modularized and thus extensible and configurable. However, use of the SecurityManager in the application still requires specific calls at various places. In general, where and how to call a given security mechanism in an application is not well separated, which again leads to pervasiveness of the security concern.

This crosscutting nature of security not only relates to the diversity of specific places where security mechanisms are to be called. Some security mechanisms require information that is not localized in the application. For instance, consider communication encryption within an application, that requires several pieces of information. Keys to use for this purpose are typically linked to a principal in the application. Key selection probably depends on the specific communication channel, and hence requires connection or host information. Ini-

tialization vectors and other security state information is normally contained in the security mechanism itself. And finally, the actual data to protect is located at several places in the application.

2.2 Unanticipated risks and change of environment

In the Common Criteria and its predecessors, the idea is to try to build a very secure system from the start. Unfortunately, history shows that, for systems of moderate to high complexity, the idea of building a secure system from the start is utopic. Unanticipated threats always show up during the lifetime of the system, sometimes because the threat analysis was incomplete, sometimes because the environment in which the software operates changes. Some form of patching or updating of the system is always necessary.

Moreover, by trying to build a very secure system from the start, the initial application often gets very complex and thus expensive to build, and the application developer may not be willing to do this high investment up front.

We will show in the paper that, by applying the separation of concerns principle, one can build more evolvable systems, that can be more easily adapted to unanticipated or changing threats.

2.3 Other issues

Of course, there are many other causes for security problems in software, and for many of these, the separation of concerns principle will bring little or no relief.

One example is the difficulty of realizing a high-level security policy with primitive security building blocks, for instance when one designs a payment protocol starting from cryptographic building blocks. The many protocol design errors of the past show that this is an intrinsically difficult task.

Another example is the high bug-sensitivity of implementations of security mechanisms. There is a whole history of implementation errors in cryptographic primitives or secure random number generators.

Also, usability and security are often hard to match, and too often a security system will hinder the usability of an application, and this in turn will lead regular users to circumvent the security system in creative ways to make it easier to do their jobs.

In summary, we do not want to claim that the principles defended in this paper will solve all problems in one go. But we do think that for some important classes of problems, a good application of the separation of concerns principle will bring some relief.

3 Advanced separation of concerns in software

Separation of concerns is a general principle in software engineering introduced by Dijkstra ([3]) and Parnas ([9]) as an answer to control the complexity of ever-

growing programs. In a nutshell, it promotes the separation of different interests in a problem, solving them separately without requiring detailed knowledge of the other parts, and finally combining them into one result.

In practice, this principle actually corresponds with finding the right decomposition or modularization of a problem. Hereby, allowing for the modules to be expressed in different representations (even within the same application) might help to facilitate their solution. Over the years, the quest for better and more advanced representations has led to different technologies, among which the well-established modularization techniques such as functional decomposition and object-orientation.

While the principle formerly typically served to structure the functionality of programs, it has lately also been applied to separate functional and non-functional requirements. Examples of such non-functional requirements are distribution, fault-tolerance and security. Such concerns are often hard to factor out in separate modules using classical object oriented techniques. The code responsible for fault-tolerant behavior for instance is not well-separated in one class or method, but cuts across many classes and methods.

To separate such crosscutting concerns, new separation or modularization techniques are necessary. In the software engineering research community, several of such new, advanced separation of concerns techniques are studied under the umbrella-term *Aspect-Oriented Programming (AOP)* ([10]). AOP programming systems are typically (but not necessarily) extensions of object-oriented programming systems.

The techniques used to factor out crosscutting concerns broadly fall into two categories:

- interception-based approaches, where certain events in the execution of a program are intercepted to do some additional processing (realizing the crosscutting concern) before or after the event.
- weaving-based approaches, where the the code realizing the crosscutting concern is woven in with the other application code by a weaver tool, at compile time, load time or even run time.

4 Security as a truly separate concern?

Using AOP for security implies building a system in such a way that some security concern can be implemented in a totally separate way, and then be “merged” into the system at a later time. Now let’s see how this method can respond to some of the problems described previously.

4.1 The secure coding issue

Due to the variety of implementation bugs one can make, there is no hope for a complete solution here. But advanced separation of concerns techniques can provide for some support. For instance, weaving techniques can be used to

solve problems that concern insufficiently defensive input checking. The idea is to augment the application at specific places with the necessary checking code (e.g. code that checks the length of a buffer, or code that performs appropriate canonicalization of a filename). However, this will by no means be a complete solution for all kinds of security-related bugs one can make.

An example of existing work in this area is the work described by Viega and McGraw ([14]). Viren Shah (also from Cigital Labs) will describe a comprehensive Aspect Oriented Security Framework building on this idea at the next International Symposium on Software Reliability Engineering.

4.2 The crosscutting issue

In a very simplified view, implementing a security concern such as access control or auditing boils down to applying certain security mechanisms at appropriate places in an application. For instance, implementing method level access control can be achieved by adding some extra code in the beginning of every method body to enforce the authorization, and by adding code where necessary to keep the state of the authorization engine up to date.

In this simplified view, for a specific application level security concern four questions must be answered¹: *what* (e.g., RBAC based method level access control), *how* (e.g., by using a reference monitor or central authorization engine), *where* (e.g., in the beginning of each method, maybe only for some methods) and *when* (e.g., only if some application-specific condition is satisfied). In the past, separation of concerns has already improved the how part in several consecutive phases. In a first phase, cryptographic algorithms were factored out in cryptographic libraries in order to enable their reuse. Later on, cryptographic service providers (CSP's) replaced these libraries to achieve algorithm independence as well as implementation independence. But until now, only the security functionality (i.e., what and how) was factored out, the relationship between application and security (i.e., when and where) is still an intrinsic part of the application (the application still requires specific calls to the security libraries).

Modularizing this kind of crosscutting pervasiveness is exactly what the AOP research community is trying to make possible. With the current state-of-the-art in AOP, complete separation of highly complex security concerns is not yet possible. But partial solutions are available already.

Existing implementations and research This paragraph discusses several techniques that each handle the separation of the where and when part with regards to security. In the following overview, we first tackle interception based approaches and afterwards weaving.

The idea of using method-call interception to impose some kind of access control or auditing is not new, and has been used in commercial products and

¹This is what we call the W4-principle. Note that we know that the description of this principle is very concise and in the current form maybe even confusing. While we do not discuss it further in this text because of textual restrictions, we hope to be able to make it more clear, if necessary, during the discussions in the workshop.

standards. Application containers in architectures as Enterprise Java Beans (EJB) and .NET offer a limited form of well-separated security features. In the case of EJB[12], method level access control is available for components implemented in the EJB component paradigm. For every method a declarative descriptor specifies if access control should be used and what roles are allowed to execute the method. Unfortunately, the access control model that is used is very simple, and only limited information is available to make access control decisions. In practice, access control is still implemented in application code (what is called “imperative security” in the EJB terminology).

CORBASec[8] is a neat example of the feasibility of complete separation for security. The CORBASec specification proposes an architecture to transparently enforce for distributed applications several security concerns, among which authentication, authorization, confidentiality, integrity and auditing. The scope of the specification proves that large scale transparent security is at least conceptually feasible. On the downside, the general idea of the OMG is to provide specifications. The CORBASec specification primarily describes the external view of the system, but it does not extend any further on how it should be implemented. Furthermore, the specification is fairly extensive and complex. As a result, few CORBA implementations support CORBASec.

Other work, including our own, tries to tackle this problem from a more generic point of view by providing a supporting architecture that enables the use of custom security mechanisms. Evidently, flexibility comes at a cost. Several important problems must be solved here, among which a generic initialization and interaction mechanism for custom mechanisms and a way to control dependencies between different mechanisms. Details on this work can be found in [1, 13, 5].

Besides interception based security, weaving based approaches for this category of problems are very recent and mainly subject of current research. We briefly discuss two of them.

Naccio [4] is a system to specify safety policies separate from the application in a general and platform-independent way. Based on this policy and a representation of the platform and the available resources, a special purpose generator produces a policy-enforcing platform library adapted to the safety policy. Note that in this case, not the application itself, but the system libraries are weaved. Therefore, besides the resource invocation parameters, other application level information cannot be used during security enforcement. The typical use cases of this work are resource oriented, rather low level controls, such as limiting the amount of bytes that can be written to a file.

Some of the authors of this paper ([2]) have applied general purpose AOP weaving systems to security concerns. This work focuses on true transparent application level security. Hereby, security decisions can depend on application state. The idea here is to capture the generic behavior of security mechanisms into reusable aspects and to compose as such a flexible and reusable framework of security aspects. The work provides some interesting results that demonstrate the feasibility of this promising approach.

4.3 Unanticipated risks and change of environment

Evidently, AOP will not help to improve the completeness of a threat analysis. However, through advanced separation of concerns, a system becomes more adaptable or evolvable. Changing, or even adding an access control concern during the lifetime of a system becomes feasible, because it does not require going over the entire code base anymore: one extra module is written, and the weaver inserts the code in relevant places in the code base. Also, the (unavoidable) process of patching bugs becomes more clean and uniform: if, for instance, a canonicalization bug is discovered, an aspect that performs appropriate canonicalization can be written and again the weaver tool will insert the code where relevant in the entire code base. The authors are fully aware that these examples are not all realizable (yet) with current AOP tools, but it is our opinion that research should go into developing weaving technology that is sufficiently advanced to realize these examples.

5 Discussion

5.1 Advantages

We have shown in the previous section that at least a limited form of separation of concerns for security aspects is feasible with state-of-the-art software engineering techniques. In our opinion, applying and extending these techniques can lead to more secure software for the following reasons.

More evolvable security concern enables agile development and maintenance The fact that the security aspects of an application become more evolvable enables another methodology for building secure systems, one that is more based on a *monitor-and-evolve* idea than on a *get-it-right-the-first-time* idea. Actually, advanced separation of concerns techniques enable an approach where one tries to build a system in such a way that security mechanisms can be merged into the system in a flexible way, leading to a system that starts off as only moderately secure, but that can adapt its security mechanisms relatively easily to changing risks in the environment in which it is set to operate.

In a sense, what is proposed here is to accept that the penetrate-and-patch approach will persist, and that one should build a system in such a way that the patch-step becomes cleaner, easier and more powerful.

Better comprehensibility and focused efforts lead to fewer bugs By separating security concerns, implementation bugs are less likely, even in the first release of a system. A modularized concern is much easier to get right, because it is more comprehensible, and because the few security experts available to work on a typical project, can focus their efforts better. Vulnerabilities of the “incomplete access mediation” type are typically caused by the crosscutting nature of access control, and will be much less likely if the access control concern is modularized.

This is probably a good place to quote the call for papers: “the attacks we do know about are based on well-known vulnerabilities that we know how to fix but have not done so”. We are convinced that the crosscutting nature of fixing these vulnerabilities is one of the major causes for this. AOP tools especially help us to overcome this problem.

Of course, the modularization of security concerns has also a lot of secondary advantages including better reusability, easier parallel development, ...

5.2 Possible showstoppers

Finally, in this discussion section, we also want to consider possible problems with this approach. We can see several reasons why separation of the security concern might possibly not lead to more secure systems.

A first issue is the fact that adaptability itself can be a major security risk. If the security mechanisms are very adaptable, attackers might be able to use the adaptation mechanisms to turn off security checks completely. Clearly, this problem will have to be dealt with.

A second issue we see is that, at this point in time, software engineering is not mature yet, and it may well be that techniques to separate most security concerns in an appropriate way will take a lot of time to be developed. In other words, maybe AOP is not yet ready for prime-time.

6 Conclusion

This position paper defends the importance of the separation of concerns principle for software security. By separating the security concern as much as possible, two important benefits are obtained: security mechanisms become more adaptable, supporting a *monitor-and-evolve* approach to security rather than a *get-it-all-right-the-first-time* approach, and separation of certain security concerns will reduce the number of implementation errors, because a well separated concern is easier to get right than a crosscutting concern.

In the very long term, our vision is that an application programmer will have to concentrate only on the functional problem he is facing. He solves this problem using a secure language which catches as much as possible the typical localized security problems (such as buffer overflows, etc.). For the other, more advanced application level security problems, a security engineer uses a framework of security aspects and specifies (declaratively or by point&click) these places in the application where security mechanisms must be enforced and possibly how strong they have to be. An appropriate tool will then make sure that the selected mechanisms are applied correctly and consistently. This way the application programmer is no longer bothered with security (and he shouldn't be). The security expert has far better control over the security measures he chooses, in the beginning as well as later on when they have to be changed when facing new security challenges.

References

- [1] B. De Win, J. Van den Bergh, F. Matthijs, B. De Decker and W. Joosen, A Security Architecture for Electronic Commerce Applications, Information Security for Global Information Infrastructures, pp. 491–500, 2000
- [2] B. De Win, B. Vanhaute and B. De Decker, How aspect-oriented programming can help to build secure software, Informatica Volume 26, No. 2, pp. 141-149, 2002
- [3] E.W. Dijkstra, A Discipline of programming, Prentice Hall, Englewood Cliffs, NJ, 1976
- [4] D. Evans, A. Twyman, Flexible Policy-Directed Code Safety, IEEE Security and Privacy, Oakland, CA, 1999
- [5] R.E. Filman, S. Barrett, D.D. Lee and T. Linden, Controlling communications by inserting illities, Communications of the ACM, Volume 45, No. 1, pp. 116-122, 2002
- [6] M.O. Killijian, J.C. Fabre, Implementing a reflective fault-tolerant corba system, Proc. of the Symposium on Reliability in Distributed Software, pp. 54-163, 2000
- [7] C. Lopes, A language framework for distributed programming, PhD thesis, Northeastern university, 1997
- [8] OMG, Security Service Specification Version 1.8, 2002
- [9] D.L. Parnas, On the criteria to be used in decomposing systems into modules, Communications of the ACM, Volume 15, No. 12, 1972
- [10] Various authors, special issue on AOP, Communications of the ACM, Volume 44 , No. 10, 2001
- [11] B. Robben, Language technology and metalevel architectures for distributed objects, PhD thesis, Dept. of CompScience, KULeuven, 1999
- [12] Sun Microsystems, Enterprise JavaBeans Specification Version 2.1, 2002
- [13] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, B.N. Joergensen, Dynamic and Selective Combination of Extensions in Component-based Applications, Proceedings of the 23rd International Conference on Software Engineering (ICSE'2001), 2001
- [14] J. Viega, J.T. Bloch and P. Chandri, Applying Aspect-Oriented Programming to Security, Cutter IT Journal, Volume 14, No. 2, pp. 31-39, 2001