An Aspect Oriented Approach for the Security Hardening of Code

Azzam Mourad Marc-André Laverdière Mourad Debbabi

Computer Security Laboratory,
Concordia Institute for Information Systems Engineering,
Concordia University, Montreal (QC), Canada

Abstract

In this paper, we present an aspect oriented approach to the systematic security hardening of source code. It aims at allowing security architects to perform software security hardening by providing an abstraction over the actions required to improve the security of the program. This is done by giving them the capabilities to specify high level security hardening plans that leverage a priori defined security hardening patterns. These patterns describe the required steps and actions to harden security code, including detailed information on how and where to inject the security code. We show the viability and relevance of our approach by: (1) Elaborating security hardening patterns and plans to common security hardening practices, (2) realizing these patterns by implementing them into aspect oriented languages, (3) applying them to secure applications, (4) testing the hardened applications. Furthermore, we discuss, in this paper, our insights on the appropriateness, strengths and limitations of the aspect oriented paradigm to security hardening.

Key words: Security Hardening, Security Hardening Patterns, Security Hardening Plans, Aspect Oriented Programming, Computer Security, Security Patterns, Secure Programming.

Email addresses: mourad@ciise.concordia.ca (Azzam Mourad), ma_laver@ciise.concordia.ca (Marc-André Laverdière), debbabi@ciise.concordia.ca (Mourad Debbabi).

[1] This research is the result of a fruitful collaboration between CSL (Computer Security Laboratory) of Concordia University, DRDC (Defence Research and Development Canada) Valcartier and Bell Canada under the NSERC DND Research Partnership Program.

1 Motivations and Background

In today's computing world, security takes an increasingly predominant role. The industry is facing challenges in public confidence at the discovery of vulnerabilities, and customers are expecting security to be delivered out of the box, even on programs that were not designed with security in mind. The challenge is even greater when legacy systems must be adapted to networked/web environments, while they are not originally designed to fit into such high-risk environments. Tools and guidelines have been available for developers for a few years already, but their practical adoption is limited so far. Nowadays, software maintainers must face the challenge to improve programs security and are often under-equipped to do so. In some cases, little can be done to improve the situation, especially for Commercial-Off-The-Shelf (COTS) software products that are no longer supported, or their source code is lost. However, whenever the source code is available, as it is the case for Free and Open-Source Software (FOSS), a wide range of security improvements could be applied once a focus on security is decided.

Very few concepts and approaches emerged in the literature to help and guide developers to integrate security into software. The most prominent proposals could be classified into: Security design patterns, secure coding and security code injection using aspect oriented programming. As for security engineering, it aims at considering security early into the development life cycle of software [1–9]. Many Security Design Patterns (SDP) are available in order to guide software engineers in designing their security models and securing their applications at the design phase. When it comes to security hardening, these proposed security design patterns are not really relevant. The reason is that we are dealing with already developed applications that are, in many cases, deployed. Concerning the secure coding approach, it presents either safe programming techniques, or a list of programming errors together with their corresponding solutions [10–13]. For instance, several publications compiled common errors and vulnerabilities in code production languages such as C/C++. Their intent is to instruct software developers to avoid these errors. Such proposals are not relevant in the setting of security hardening since we are dealing with an already developed software. Moreover, these secure coding practices are very often manually applied and our aim is actually to elaborate a systematic, and even preferably automatic approach to security hardening. More recently, several proposals have been advanced for code injection, via an aspect oriented computational style, into source code for the purpose of improving its security. The injection of security components into application using Aspect Oriented Programming (AOP) is a relatively new programming paradigm that provides a more advanced modularization mechanism on top of the traditional object-oriented programming (OOP). It is based on the idea that computer systems are better programmed by separately specifying the

various concerns (i.e. Separation of Concerns), and then relying on underlying infrastructure to compose them together. The techniques in this paradigm were precisely introduced to address the development problems that are inherent to crosscutting concerns. This concept seems to be the most relevant to integrate security into FOSS. However, all the available contributions in [14–18] are limited to case studies that use AOP to apply security in an ad-hoc manner.

As a result, integrating security into software is becoming a very challenging and interesting domain of research. In this context, the main intent of our research is to create methods and solutions to integrate systematically security models and components into FOSS. Our proposition is based on AOP and inspired by the best and most relevant methods and methodologies found in each one of the aforementioned concepts and approaches, in addition to elaborating valuable techniques that permit us to provide a framework for systematic security hardening.

This paper provides our first accomplishment in developing our security hardening framework. The experimental results presented together with the security hardening patterns and aspects explore the efficiency and relevance of our approach. The remainder of this paper is organized as follows. In Section 2, we introduce the contributions in the field of security patterns, secure programming and practices and AOP security. Afterwards, in Section 3, we illustrate briefly a classification of the different levels of security hardening. Then, in Section 5, we present our security hardening approach together with many security hardening plans, patterns and aspects for different security issues and problems. In Section 6, we explore the experimental results. In Section 7, we discuss the appropriateness of the current AOP languages for security hardening. Finally, we offer concluding remarks in Section 8.

2 Related Work

Our approach constitutes an organized framework that provides methodologies for the improvement of security at all levels of the software systems. As such, we present in the sequel an overview of the current literature on the approaches that may be useful for integrating security into software, and thus guide us in developing our security hardening framework.

Starting with the security design patterns, the following two paragraphs summarize briefly the content of the related publications. In [9], Yoder and Barcalow introduced a 7-pattern catalog. In fact, their proposed patterns were not meant to be a comprehensive set of security patterns, rather just as starting point towards a collection of patterns that can help developers address security

issues when developing applications. Kienzle et al. [5,6] have created a 29-pattern security pattern repository, which categorized security patterns as either structural or procedural patterns. Structural patterns are implementable patterns in an application whereas procedural patterns are patterns that were aimed to improve the development process of security-critical software. The presented patterns were implementations of specific web application security policies. Romanosky [7] introduced another set of design patterns. The discussion however has focused on architectural and procedural guidelines more than security patterns.

Brown and Fernandez [4] introduced a single security pattern, the authenticator, which described a general mechanism for providing identification and authentication to a server from a client. Although authentication is a very important feature of secure systems, the pattern, as was described, was limited to distributed object systems. Fernandez and Warrier extended this pattern recently in [3], although it remains similarly limited. Braga et al. [2] also investigated security-related patterns specialized for cryptographic operations. They showed how cryptographic transformation over messages could be structured as a composite of instantiations of the cryptographic meta-pattern. The Open Group [1] has possibly introduced the most mature design patterns so far. Their catalog proposes 13 patterns, and is based on architectural framework standards such as the ISO/IEC 10181 family. The most recent work in this domain is from Schumacher et al. [19]. They offered a list of forty-six patterns applied in different fields of software security, although most of them are rewriting of previously proposed patterns. All these propositions aim at deploying security early during the development of new software, which makes them of limited usefulness for the security hardening of already developed FOSS.

On the topic of secure programming of C programs, developers are offered a good selection of useful and highly relevant material. One of the newest and most useful additions is from Seacord [12], which offers in-depth explanations on the nature of all known low-level security vulnerabilities in C and C++. Another common reference is from Howard and Leblanc [11], and includes all the basic security problems and solutions, as well as code fragments of functions allowing to safely implement certain operations (such as safe memory wiping). The authors also describe high-level security issues, threat modeling, access control, etc. Slides from Bishop, in addition to his book [10] , provide a comprehensive view on information assurance, as well as security vulnerabilities in C. In addition, he provides some hints and practices to solve some existing security issues. Wheeler [13] offers the widest-reaching book on system security available online. He covers operating system security, safe temporary files, cryptography, multiple operating platforms, spam, etc. We consider his solutions relevant to the problem of insecure temporary files. These propositions may provide us with some guidance in remedying low-level security

vulnerabilities, however, they are also of limited usefulness because they are addressed to instruct software developers how to avoid these errors in new software or apply the needed corrections manually into the code.

Regarding aspect oriented programming and security, AOP appears to be a promising paradigm for software security, which is an issue that has not been adequately addressed by previous programming models such as object oriented Programming (OOP). Aspects allow to precisely and selectively define and integrate security objects, methods and events within application, which make them interesting solutions for many security issues. Few contributions have been published on applying AOP to security. Most of them are presented as case studies that shows the relevance of AOP in security or explore the usability of a proposed AOP language. Moreover, there exist some published aspects that address simple security issues and apply them in an ad-hoc manner. In this context, the following is a brief overview on the available contributions in this domain of research.

Cigital labs proposed an AOP language called CSAW [17,20–22], which is a small superset of C programming language. Their work is mostly dedicated to improve the security of C programs. They presented typical aspects that defend against specific types of attacks and address local and small sized problems such as buffer overflow and data logging. These aspects were divided in the low-level and high-level categories. The low-level aspects target the problems of exploiting the environmental variables such as attacks against `Setuid` programs, the problems of format strings and variable verification that cause the buffer overflow attacks, the problems of confidentiality and communication encryption, etc. Their high level aspects address the problems of event ordering, signal race condition and type safety.

De Win et al. in [15,23–25] discussed two aspect oriented approaches and explored their use in integrating security aspects within applications. In their first approach, the interception, they explored the need to secure all the interactions with the applications that cannot be trusted and they provided additional security measures for sensitive interactions. They used a coarse-grained alternative mechanism for interception that consists of putting an interceptor at the border of the application, where interactions are checked and approved. Their proposition is achieved by changing the software that is responsible of the external communication of the applications. Their second approach, the weaving-based AOSD, is based on a weaving process that takes two or more separate views of an application and merge them together into a single artifact as if they are developed together. They used in this approach the Advice and Joinpoints concepts to specify the behavior code to be merged in the application and the location where this code should be injected. To validate their approach, they developed some aspects using AspecJ to enforce access control and modularize the audit and access control features of an FTP

server.

In [14], Ron Bodkin surveyed the security requirements for enterprize applications and described examples of security crosscutting concerns. His main focus was on authentication and authorization. He discussed use cases and scenarios for these two security issues and he explored how their security rules could be implemented using AspectJ. He also outlined several of the problems and opportunities in applying aspects to secure web applications that are written in Java.

Another contribution in AOP security is the Java Security Aspect Library (JSAL), in which Huang et al. [16] introduced and implemented, in AspectJ, a reusable and generic aspect library that provides security functions. It is based on the Java Security packages JCE and JAAS. To make their aspects reusable, they left to the programmer the responsibility to specify and implement the pointcut. This approach is a useful first step, but requires the developer to be a security expert who knows exactly where each piece of code should be injected. Moreover, its goal is to prove the feasibility of reusing and integrating pre-built aspects.

Shlowikowski and Ziekinski discussed in [18] some security solutions based on J2EE and JBoss application server, Java Authentication and Authorization service API (JAAS) and Resource Access Decision Facility (RAD). These solutions are implemented in AspectJ. They explored in their paper how the code of the aforementioned security technologies could be injected and weaved in the original application. However, their aspects are limited to specific cases of identification and access control, which do not need so much effort to identify the pointcuts where the needed code should be injected.

3 Security Hardening Taxonomy

Security hardening at the application level is a relatively unknown term in the current literature and, as such, we first provided a definition for it in [26]. We also proposed a taxonomy of security hardening methods that refer to the area to which the solution is applied. We established our taxonomy by studying the solutions of software security problems in the literature. We also investigated the security engineering of applications at different levels, including specification and design issues [10,1,11]. From this information on how to correctly build new programs, and some hardening advice existing in the literature, we were able to draw out the following classification for software hardening methodologies.

We define software security hardening as any *process, methodology, product or*

combination thereof that is used to add security functionalities and/or remove vulnerabilities or prevent their exploitation in existing software. This definition focuses on the solving of vulnerabilities, not on their detection. In this context, the following constitutes the detailed classification of security hardening methods:

3.1 Code-Level Hardening

Code-Level hardening constitutes *changes in the source code in a way that prevents vulnerabilities without altering the design.* During the software creation, vulnerabilities are created and are a direct result of the programming phase of the project. Code level hardening constitutes of removing these vulnerabilities in a systematic way by implementing the proper coding standards that were not enforced originally.

3.2 Software Process Hardening

Software Process hardening is the *addition of security features in the software build process without changes in the original source code.* Software Process hardening considers the inclusion of hardening practices within the software development process, notably on the matter of choosing appropriate platforms, library implementations (in the case of statically linked libraries), compilers, aspects, etc. that result in increased security. It is also possible to use compilers and aspects that add some protections in the object code, which were not specified in the source code, and that prevent or complicates the exploitation of vulnerabilities existing in the program. To a certain extent, it externalizes the security concerns from the program, but has the disadvantages of being harder to audit and may lack portability.

3.3 Design-Level Hardening

Design-Level hardening is the *re-engineering of the application in order to integrate security features that were absent or insufficient.* It refers to changes in the application design and specification. Some security vulnerabilities cannot be resolved by a simple change in the code or by a better environment, but are due to a fundamentally flawed design or specification. Changes in the design are thus necessary to solve the vulnerability or to ensure that a given security policy is enforceable. Moreover, some security features need to be added for new versions of existing products. This category of hardening practices target more high-level security such as access control, authentication and secure

communication. In this context, best practices, known as security design patterns [1], can be used to guide the redesign effort. Although such patterns are targeting the security engineering of new systems, such approach can also be redirected and mapped to cover deploying security into existing software.

3.4 Operating Environment Hardening

Operating Environment hardening consists of *improvements to the security of the execution context (network, operating systems, libraries, utilities, etc.) that is relied upon by the software.* It impacts the security of the software in a way that is unrelated to the program itself. This addresses the operating system (typically via configuration), the protection of the network layer, the configuration of middleware, the use of security-related operating system extensions, the normal system patching, etc. [13,27]. Many security appliances can be deployed and integrated into the operating environment in a way that provide some high-level security services. These hardening practices fall within the scope of proper management of an IT department and, as much as they can prevent exploitation of vulnerabilities, they do not remedy them.

4 Security Engineering as Pre-Hardening Process

Although security hardening focuses on finding the best countermeasure to a particular threat, some prerequisite security engineering tasks should be performed[11] in order to achieve a complete hardening process. In this context, we present in the following the security engineering tasks and the steps needed to harden security into applications.

Identifying Threats and Calculating Risks Identifying threats is an important task in security hardening since we are not able to build a secure system until evaluating the threats to it. So, the goal here is to determine which threats require mitigation and how to mitigate them. We do not advocate for one methodology over another, as many exists, but it is preferable to apply a structured and formal mechanism or process.

Countermeasures Determining the appropriated technique(s) to mitigate a particular threat requires first its identification and risk calculation. The literature often portrays threats and vulnerabilities accompanied with a mapping to known counter-measures addressing them. Please refer to Table 1 for an instance of such a mapping.

Once the selection of the countermeasure is performed, at this point, the security hardening process starts and the security hardening plans and patterns

Threat Type	Mitigation Techniques
Spoofing Identity	Appropriate Authentication, Protect Secret Data
Tampering with Data	Appropriate Authorization, Hashes, Message Authentication Codes, Digital Signatures
Repudiation	Digital Signatures, Timestamps, Audit Trails
Information Disclosure	Authorization, Encryption, Protect Secrets
Denial of Service	Appropriate Authentication and Authorization, Filtering, Throttling, Quality of Service
Elevation of Privilege	Run with Least Privilege

Table 1
Mapping Between Threats and Mitigations

can be used to develop the security mechanisms and harden the application.

5 Security Hardening Approach

This section illustrates our proposition to harden security into applications. We first describe the architecture of the proposed approach, then we present some security hardening plans and patterns. In this context, authentication, authorization, confidentiality, availability, non-repudiation and integrity are the main security objectives and properties that need to be enforced. Moreover, the low level security or safety problems will be also addressed. The approach architecture is illustrated in Figure 1.

The primary objective of this approach is to allow the security architects to perform security hardening of free and open source software by providing an abstraction over the actions required to improve the security of the program. This is done by providing security hardening patterns into a catalog and allowing them to write security hardening plans that use these patterns. We define security hardening patterns as proven solutions to known security problems, together with detailed information on how and where to inject each component of the solution into the application. Those patterns, collected as a library, remain of limited use by themselves. Our approach integrates the specification of hardening plans, which permit the users to select the patterns to be applied, and give them proper parameters, which are pattern-specific (API,

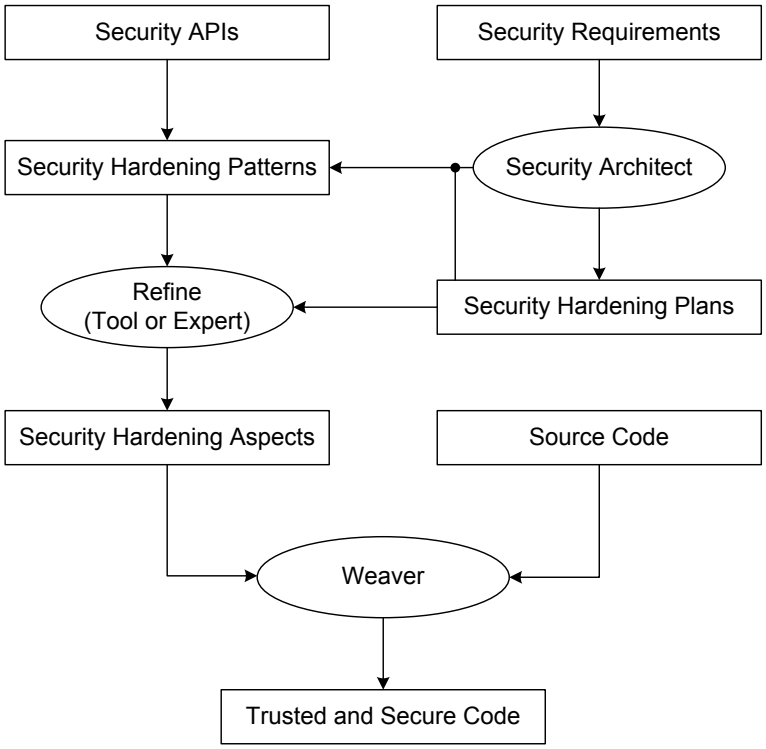


Fig. 1. Schema of Our Approach

language, ciphersuites, etc.). Moreover, the module, class, function, etc., where the hardening pattern is to be applied is also specified in the hardening plan. Those hardening plans are then combined with the patterns in order to create concrete security hardening solutions. The aforementioned definition explores clearly the difference between our security hardening patterns and the current security design patterns. The security hardening plans are derived from the security requirements by the security architects. On the other hand, the security hardening patterns are developed by security experts and accumulated into the catalog. The security APIs constitute the building blocks used by the patterns to achieve the desired solutions.

The abstraction of the hardening plans is bridged by concrete steps defined in the hardening patterns using a hardening specification language based on aspect-oriented. The elaboration of this language is in progress and will be presented in future work. This dedicated language, together with a well-defined template that instantiates the patterns with the plan's given parameters, allow to specify the precise steps to be performed for the hardening, taking into consideration technological issues such as platforms, libraries and languages. In the context of this paper, we also illustrate this approach by manually refining the elaborated patterns into aspects and then weaving them into the program to harden. Moreover, to show the benefits of the proposed approach, we implemented and tested them on real applications, resulting in a trustworthy library for security hardening.

As an example of the approach useability, let us imagine that management requests an increase of security in a product. More specifically, they want to see access control features, encrypted files and encrypted network communications. The maintainer of the aforementioned product has some knowledge about those issues, but lacks the experience to do them well. Using the proposed approach, he could implement this request by first writing a security hardening plan, in which he specifies the needed pattern(s). Of course, this requires to understand lightly the application's inner workings and to derive the security policies that are specified in the requirements, but this is the responsibility of the maintainer. Finally, he can use a tool that refines the patterns into aspects and then executes the AOP weaver to obtain the hardened source code, which can be now inspected for correctness. In contrast, without our approach, a maintainer would need to learn and/or create many APIs (e.g. OpenSSL, and a home-brewed RBAC module) and their proper uses, then use them in the application. This could require some additional architecture and design work. He could also create some vulnerabilities by a lack of deep understanding, resulting in an expensive, time-consuming and potentially flawed upgrade. As a result, the approach constitutes a bridge that allows the security experts to provide the best solutions to a particular security problem with all the details on how and where to apply it, and allows the software engineers to use these solutions by simply specifying and developing high level security hardening plans.

5.1 Security Hardening Plan

A security assessment brings any decision-maker to perform a risk analysis, which will finally determine the security requirements. A security hardening plan is required in order to translate such requirements into software modification, implemented either manually or automatically. We identified the following areas of specification for the hardening of the security of applications at levels excluding operating environment hardening:

General Hardening Plans focus on matters that are common to many operating systems and programming languages and can thus be reasonably be considered as being of general nature.

System-Dependent Hardening are dependent on the platform(s) used by the software to harden, its capabilities and limitations.

Technology-Dependent Hardening are based on the intricacies of the technologies used in the implementation of the software that are not related to the operating system. Those technologies are typically the programming language and libraries used.

In Listing 2, we include an example of an effective security hardening plan for securing connection, adding authorization and adding encryption. We specified our example with a free-form syntax template shown in Listing 1. The patterns used by these plans are presented in section 5.2.

Listing 1. Hardening Plan Template

```
[PatternName]
    parameters
            language:[language]
            api: [api]
        [pattern-specific parameter]:[value]
         ...
    where: [file name]:[all or class, function, or variable]
```

Listing 2. Our Hypothetical Hardening Plan

```
SecureConnection
    parameters
        protocol: TLS 1 or SSLv3
        ciphersuites: default
        peer: client
        language: C
        api: GnuTLS
    where: berkeley.c:all

SecureConnection
    parameters
        protocol: TLS 1 or SSLv3
        ciphersuites: default
        peer: server
        language: C
        api: GnuTLS
    where: berkeley_server.c:all

SecureConnection
    parameters
        protocol: TLS 1 or SSLv3
        ciphersuites: default
        peer: client
        language: Java
        api: JSSE
    where: all

AddAccessControl
    parameters
        type: ACL
        language: Java
        api: JAAS
    where: TestClass.java: ca.concordia.tfoss.hardening.TestClass.doSomething
        ()

AddEncryption
    parameters
        key: random
        algorithm: AES-CBC
        language:C
        api: OpenSSL
    where: all:password
```

5.2 Security Hardening Patterns

Security hardening patterns specify the steps and actions needed to harden systematically security into the code. In this context, security hardening patterns are defined as proven solutions to known security vulnerabilities and problems, together with detailed information on how and where to inject each component of the solutions into the code. In this section, we present the patterns for securing a connection, performing authorization and encrypting some information in the memory. The elaborated three patterns are used by the hardening plans presented in Listing 2, and later refined and validated in Section 6. For deep understanding, the reader needs to relate each one to its corresponding refined aspect. Currently, we defined them based on a high-level and free form syntax that explains the steps to be performed for the hardening to a programmer that is not a security expert. If a programmer wants to implement those patterns directly, he/she would still need to learn how to use the APIs, although the guidance from them would save lot of work. Different forms of patterns' representation and specification will be proposed and adopted in future work in order to facilitate their refinement into real code and/or aspects.

5.2.1 Secure Connection

A first issue is the securing of channels between two communicating parties to avoid eavesdropping, tampering with the transmission or session hijacking. There are a lot of different secure protocols that have been developed, but it remains that SSL/TLS are the most widely used protocols for this task. We thus present a pattern that secures a connection using TLS. The usage scenario, around which the pattern in Listing 3 is developed, is a connection between a client and a remote server.

To generalize our solution and make it applicable on wider range of applications, we considered some complicated scenarios and we included conditional elements. Other patterns that solve simpler cases can be derived easily from this solution. For instance, we assume that not all the connections are secured, since many programs have different local interprocess communications via sockets. In this case, all the functions responsible of sending receiving data on the secure channels are replaced by the ones provided by SSL/TLS. On the other hand, the other functions that operate on the non secure channels are kept untouched. Moreover, we suppose that the connection processes and the functions that send and receive the data are implemented in different components (i.e different classes, functions, etc.). This required additional effort to develop additional components that distinguish between the functions that operate on secure and non secure channels and export parameters between

different places in the application. Please refer to Section 6.1 for more details. We also assume that there is additional trusted certificates that need to be loaded and that no particular error handling is needed (i.e. that the operation is reported as a success or failure). Since there is a difference between implementing a connection in a client and server applications, we distinguish in the following pattern the steps specific to each one of them.

The usage scenario for the pattern in Listing 4 is similar to the previous one, with the difference that we refer to securing connection in Java.

Listing 3. Hardening Pattern for Secure Connection in C

```
Before being used:
    - Initialize the TLS library
    - If desired, load an additional trust store (only client)
    - Load the trust store and the server certificate(s) (only server)
    - Load or generate the key exchange parameters (only server)

For all sockets to harden:
    - Before the TCP connection is established, initialize the TLS session
      resources
    - After the TCP connection is established, add the TLS handshake
    - If desired, after the TLS handshake, perform further validation of the
      certificate (only client)
    - If desired, receive and validate the client certificate (only server)
    - Replace the send and receive functions using that socket by the TLS
      send/receive functions of the used API when using a secured socket
    - Before the socket is closed, gracefully cut the TLS connection
    - After the socket is closed, deallocate all the resources associated
      with the just-closed connection/session

Once no longer useful:
    - Deinitialize the TLS library
```

Listing 4. Hardening Pattern for Secure Connections in Java

```
For all sockets to harden:
         - Before creating the socket, declare a SSL Socket Factory and
           obtain its reference
         - Replace the socket creation by the SSL Socket Factory socket
           creation
```

5.2.2 Authorization

Access control is a problem of authorizing or denying access to a resource or operation. It requires to know which principal is interacting with the application, and what are its associated rights. Please see Listing 5 that describes an Authorization hardening pattern. Its usage scenario assumes that interface changes are undesirable and that a policy is specified and loaded separately from what programmers can directly specify (which is the case for technologies like Java). It requires some forms of authentication in order to have the working user credentials that are used in the access control decisions.

Listing 5. Hardening Pattern for Authorization

```
Statically:
    - Define the Authorization policy

Pre-requisite:
    - Authentication mechanism implemented

For each sensitive operation:
    - Put the operation in a wrapper
    - In the wrapper, obtain the subject descriptors from the runtime
      environment
    - Validate the operation against the policy
    - If validated, allow the operation to proceed
    - If desired, log the execution and/or failure.
```

5.2.3 Encryption of Memory

Although processes should encapsulate all of their internal information in a manner that is opaque to other processes, modern operating systems allow to read the entire memory space, making the information stored in memory vulnerable to local memory reading attacks. Furthermore, a UNIX core dump of a crashed process may contain sensitive information. Cryptography offers tools useful to protect confidentiality and to detect integrity violations of data that are useful in this case.

The pattern we informally describe in Listing 6 assumes an environment where we can directly manipulate memory ranges (like in the C programming language) and that the information is kept temporarily in memory.

Listing 6. Hardening Pattern for Memory Encryption

```
Before being used:
    - Initialize the cryptographic library

For all the setting of the data to protect:
    - If it is the first time, generate random keys and initialization
      vectors of cryptographic quality
    - Encrypt using a secure (non-ECB) mode
    - Wipe the plaintext from memory securely

For all the desired getting of the data to protect:
    - Decrypt using the same mode and key from the ciphertext
    - Once used, wipe the plaintext from memory securely.

After the last desired getting of the data to protect:
    - Erase the cryptographic information from memory

Once no longer useful:
    - Deinitialize the cryptographic library
```

6 Patterns' Refinement and Experimental Results

One method that can be used to implement the security hardening plans and patterns is the use of Aspect-Oriented programming (AOP). We demonstrated the feasibility of our approach for systematic security hardening by developing examples that deal with security requirements such as securing a connection, authorization and encrypting some information in the memory and apply them to developed and selected applications. During the course of our study, we developed some utility functions in C and Java, some example code and some aspects in AspectC++ and AspectJ that implement security hardening of the cases described previously. We will show some of our findings here. To illustrate our approach, we performed the following steps:

(1) Selected and/or implemented applications that need to be hardened
(2) Added code that enforce the desired security requirements with the least changes in the existing code
(3) Extracted library functions that perform these functionalities in order to simplify the coding of the solutions
(4) Developed patterns that describe in a clear and abstract way those steps
(5) Refined our patterns by programming aspects that replicate the manual hardening in a systematic way
(6) Applied the aspects on the original programs by using available weavers (AspectJ 1.5.2 and AspectC++ 1.0pre3)
(7) Tested the resulting programs for functional and security correctness by comparing them to the manually hardened ones
(8) Measured the execution time of both manual and automatically hardened programs

The latter security verification of the hardened applications and measurements between the performance cost of hardening manually or using aspect-oriented technologies demonstrate that AOP is a viable method for hardening applications. However, we also found limitations that forced us to resort to complicated tricks in order to obtain our functional objective, if at all possible. We noticed that improvements to AspectC++ and AspectJ would have facilitated this task and kept the aspects much lighter and concise.

6.1 Secure Connection

We refined and implemented in Listing 7 the pattern presented in Listing 3 using AspectC++ aspects that use the GnuTLS library. The scenario considered is presented in Section 5.2.1. The reader will notice the appearance of `hardening_sockinfo_t` as well as some other related functions part of

`securityhardening.h`. These are the structures and functions that we developed to distinguish between secure and non secure channels and export the parameter between the application's components at runtime. We found that one major problem was the passing of parameters between functions that initialize the connection and those that use it for sending and receiving data, as the GnuTLS data structure was not type compatible with the Berkeley socket (an integer). In order to avoid using shared memory directly, we opted for a hash table that uses the Berkeley socket number as a key to store and retrieve all the needed information (in our own defined data structure). One additional information that we store is whether the socket is secured or not. In this manner, all calls to a `send()` or `recv()` are modified for a runtime check that uses the proper sending/receiving function. Although this option is somewhat heavy, it allows to work in a generalized, single-threaded case. Due to limitations in the current aspect-oriented programming technology, we could not obtain a simpler solution. The introduction of AOP primitives related to the data flow and the transformation of function parameters may provide a powerful tool that avoids these programming gymnastics.

Listing 7. Aspect Hardening Connections Using GnuTLS

```
#ifndef __TLSSecurityHardening_ah__
#define __TLSSecurityHardening_ah__

#include <stdio.h>
#include <gnutls/gnutls.h>
#include <gnutls/extra.h>
#include <gcrypt.h>
#include "securityhardening.h"

 aspect SecureConnection {

  advice execution ("% main(...)") : around () {
    //Initializatin of the API
    struct timeval tp;
    struct timeval after_tp;
    signed long diffusec;
    unsigned long diffsec;
  gettimeofday(&tp, NULL);
    hardening_socketInfoStorageInit();
    hardening_initGnuTLSSubsystem(NONE);

  tjp->proceed();

  hardening_deinitGnuTLSSubsystem();
  hardening_socketInfoStorageDeinit();

  gettimeofday(&after_tp, NULL);

    diffsec = after_tp.tv_sec - tp.tv_sec;
    diffusec = after_tp.tv_usec - tp.tv_usec;
    if (diffusec < 0){
        diffsec --;
        diffusec += 1000000;
    }
    fprintf(stderr, "GnuTLS Hardening with aspects , time elapsed: %ld+%.6ld (
        seconds+microseconds)\n", diffsec, diffusec);
  *tjp->result() = 0;
  }
```

```
advice call("% connect(...)") : around () {

  //variables declared
  hardening_sockinfo_t socketInfo;
  const int cert_type_priority[3] = { GNUTLS_CRT_X509, GNUTLS_CRT_OPENPGP,
      0};

  //initialize TLS session info
  gnutls_init (&socketInfo.session, GNUTLS_CLIENT);
  gnutls_set_default_priority (socketInfo.session);
  gnutls_certificate_type_set_priority (socketInfo.session,
      cert_type_priority);
  gnutls_certificate_allocate_credentials (&socketInfo.xcred);
  gnutls_credentials_set (socketInfo.session, GNUTLS_CRD_CERTIFICATE,
      socketInfo.xcred);


//Change port from 80 to 443
struct sockaddr_in * address = *(struct sockaddr_in **)tjp->arg(1);
address->sin_port = htons(443);

  //Connect
  tjp->proceed();
  if(*tjp->result()<0) {perror("cannot connect "); exit(1);}

  //Save the needed parameters and the information that distinguishes
      between secure and non-secure channels
  socketInfo.isSecure = true;
  socketInfo.socketDescriptor = *(int *)tjp->arg(0);
  hardening_storeSocketInfo(*(int *)tjp->arg(0), socketInfo);
  //TLS handshake
  gnutls_transport_set_ptr (socketInfo.session, (gnutls_transport_ptr) (*(
      int *)tjp->arg(0)));
  *tjp->result() = gnutls_handshake (socketInfo.session);
}

//replacing send() by gnutls_record_send() on a secured socket
advice call("% send(...)") : around () {

  //Retrieve the needed parameters and the information that distinguishes
      between secure and non-secure channels
  hardening_sockinfo_t socketInfo;
  socketInfo = hardening_getSocketInfo(*(int *)tjp->arg(0));

//Check if the channel, on which the send function operates, is secured or
    not
  if (socketInfo.isSecure)
     //if the channel is secured, replace the send by gnutls_send
     *(tjp->result())  = gnutls_record_send(socketInfo.session, *(char**)
         tjp->arg(1), *(int *)tjp->arg(2));
  else
     tjp->proceed();
}

//replacing recv() by recv() on a secured socket
advice call("% recv(...)") : around () {

  //Retrieve the needed parameters and the information that distinguishes
      between secure and non-secure channels
  hardening_sockinfo_t socketInfo;
  socketInfo = hardening_getSocketInfo(*(int *)tjp->arg(0));

  //Check if the channel, on which the send function operates, is secured
      or not
  if (socketInfo.isSecure)
     //if the channel is secured, replace the receive by gnutls_receive
     *(tjp->result())  = gnutls_record_recv(socketInfo.session, *(char**)
```

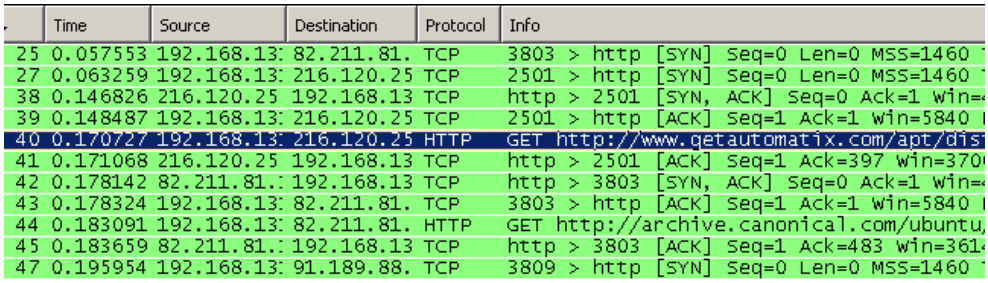
```
            tjp->arg(1), *(int *)tjp->arg(2));
    else
       tjp->proceed();
  }

  advice call("% close(...)") : around () {
    hardening_sockinfo_t socketInfo = hardening_getSocketInfo(*(int *)tjp->
        arg(0)); /* socket matched by sd*/
    if(socketInfo.isSecure ){
       socketInfo.isSecure = false; //default values in case variable is
           reused
       socketInfo.socketDescriptor = 0;
       gnutls_bye(socketInfo.session, GNUTLS_SHUT_RDWR);
       gnutls_deinit(socketInfo.session);
       gnutls_certificate_free_credentials(socketInfo.xcred);
       hardening_removeSocketInfo(*(int *)tjp->arg(0));
    }
  tjp->proceed();
  }
};

#endif // __TLSSecurityHardening_ah__
```

This pattern and its corresponding aspect, however, did not hold true for implementing a similar hardening for secure connection in Java. Due to the Java API specifications for secure sockets (`SSLSocket` is a subclass of `Socket`), the hardening took advantage of this reality to obtain much simpler AspectJ code, as shown in Listing 8.

Listing 8. Aspect Hardening Connections in Java

```
import java.io.*;
import java.net.*;
import javax.net.ssl.*;

public aspect secureconnection {

      pointcut newsocket(String host, int port)
          : call(Socket.new(String, int))
                && args(host, port);

      Socket around(String host, int port)
          : newsocket(host, port) {

            try{
              SSLSocketFactory sslFact = (SSLSocketFactory)SSLSocketFactory.
                  getDefault();
              return sslFact.createSocket(host, port);
            } catch (Exception e){ return null;}
      }
}
```

In order to validate the correctness of our proposed security hardening solutions, we developed our own client applications and selected open source software to secure their connections. Originally, these applications supported only HTTP request. We applied our solutions to make them support HTTPS connections by weaving the elaborated aspects with the different variants of the applications. Then, we installed a local server (Debian apache-ssl pack-



	Time	Source	Destination	Protocol	Info
25	0.057553	192.168.13:	82.211.81.	TCP	3803 > http [SYN] Seq=0 Len=0 MSS=1460
27	0.063259	192.168.13:	216.120.25	TCP	2501 > http [SYN] Seq=0 Len=0 MSS=1460
38	0.146826	216.120.25	192.168.13	TCP	http > 2501 [SYN, ACK] Seq=0 Ack=1 Win=
39	0.148487	192.168.13:	216.120.25	TCP	2501 > http [ACK] Seq=1 Ack=1 Win=5840
40	0.170727	192.168.13:	216.120.25	HTTP	GET http://www.getautomatix.com/apt/dis
41	0.171068	216.120.25	192.168.13	TCP	http > 2501 [ACK] Seq=1 Ack=397 Win=370
42	0.178142	82.211.81.:	192.168.13	TCP	http > 3803 [SYN, ACK] Seq=0 Ack=1 Win=
43	0.178324	192.168.13:	82.211.81.	TCP	3803 > http [ACK] Seq=1 Ack=1 Win=5840
44	0.183091	192.168.13:	82.211.81.	HTTP	GET http://archive.canonical.com/ubuntu
45	0.183659	82.211.81.:	192.168.13	TCP	http > 3803 [ACK] Seq=1 Ack=483 Win=361
47	0.195954	192.168.13:	91.189.88.	TCP	3809 > http [SYN] Seq=0 Len=0 MSS=1460

Fig. 2. Packet Capture of Unencrypted Traffic

age) that accepts only SSL-enabled connections (HTTPS), selected a remote server that allows HTTP connections and used WireShark software to capture the traffic between the hardened applications and the servers. Afterwards, we executed the hardened programs and iterated over many connections to the remote and local servers.

The experimental results presented in Figures 2 and 3 show that the new secure applications are able to perform both HTTP and HTTPS package acquisition and retrieving successfully the required data. Figure 2 shows the packet capture, obtained using WireShark software, of the unencrypted HTTP traffic between one hardened application and the remote server `www.getautomatix.com`. On the other hand, in Figure 3, we see that the connections between the same hardened application and the local web server are HTTPS. Specifically, the highlighted line shows TLSv1 application data exchanged in encrypted form. Moreover, we see clearly that all the information exchanged was encrypted and passed via TLS/SSL.

After verifying the functional and security correctness of the hardened applications, we measured the performance impact of our approaches. We iterated over many connections where a connection to a server is established, and a few index page's bytes are retrieved. In Table 2, the 'Berkeley' column shows the execution time for unsecured sockets using the standard Berkeley API. The 'GnuTLS' columns show two approaches: One having modified the code directly and the other passing the GnuTLS variables by a hash table. Finally, 'GnuTLS via AOP' shows the performance to using AspectC++ to inject code that is equivalent to 'GnuTLS Hash Table'. We can observe that there is no significant performance difference between the hardening approaches.

We duplicated this effort using Java (1.5.6) and AspectJ, to find that the performance impact using aspects to perform the hardening was negligible compared to manual hardening, as shown in Table 3.

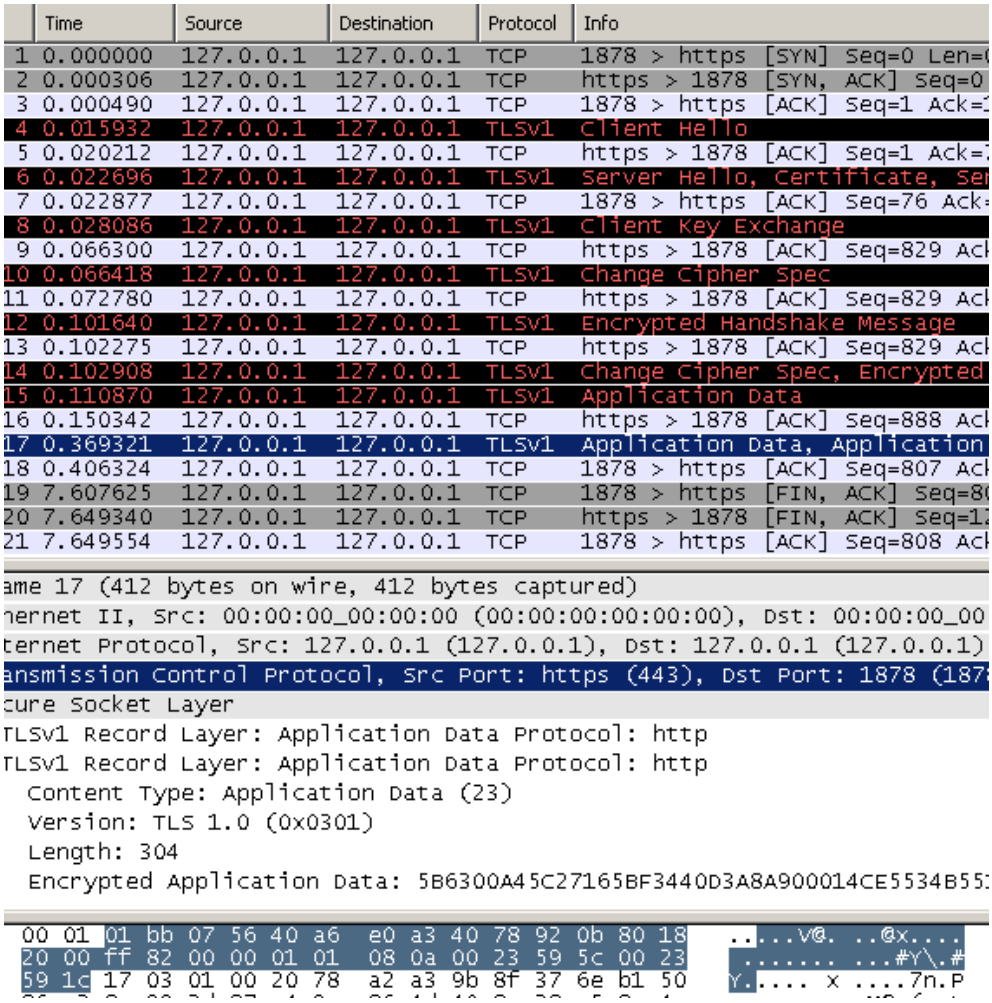


Fig. 3. Packet Capture of SSL-protected Traffic

Connections	**Berkeley**	**GnuTLS**		
		Hash Table	Direct	via AOP
100	0.234 s	15.937 s	15.89 s	15.953 s
500	1.406 s	79.39 s	80.484	79.828 s
1000	3.062 s	159.984 s	157.796	161.25 s

Table 2
Execution Time For Different Approaches in C

6.2 Authorization

We have implemented an example of access control as an AspectJ aspect (see Listing 9) that uses JAAS for authorization. The rights are specified in a policy file, which is not included here. We assume a local login, in this case, and we

Connections	Unsecured	Secured Manually	Secured with AspectJ
100	0.250 s	4.781 s	4.728 s
500	1.156 s	21.266 s	21.297 s
1000	2.203 s	41.734 s	42.032 s

Table 3
Execution Time For Different Approaches in Java

obtain the user name from the virtual machine. The permissions are specified in the format `package.class.function`. We also applied verification on the functional and security correctness of the hardened application. This task has been performed by either adding or removing the access right to execute the target method in the policy file. The practical impact of removing the right and then executing the method threw an access right violation exception by the Java virtual machine, which illustrates the correctness of the authorization deployed. Moreover, as our runtime experiments show (c.f. Table 4), there is an overhead of about 40% in execution time for multiple runs of a security check between a case hardened manually and one hardened using AspectJ. Although this difference may look significant, we calculate that the execution time per iteration is about 1 ms more when using AspectJ, which is reasonable.

Listing 9. Aspect Adding Authorization in Java

```
package ca.concordia.ciise.seclab.hardening;

import java.security.*;
import java.util.Hashtable;
import javax.security.auth.*;
import javax.security.auth.callback.*;
import javax.security.auth.login.*;
import javax.security.auth.spi.*;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.Signature;

public aspect AddAccessControl {

  protected static Hashtable subjects = new Hashtable();
  abstract class Action implements PrivilegedExceptionAction{};

  pointcut test(): call(void doSomething());

  String getPermissionName(Signature sig){
    return sig.getDeclaringTypeName().concat(".".concat(sig.getName()));
  }

  void around(): test(){
    try{
      //get the Subject instance based on the current user name
       Subject subject = (Subject) subjects.get(System.getProperty("user.name
           "));

       //anonymous inner class for the privileged action
       //however, we should have them static to avoid unnecessary overhead
       PrivilegedExceptionAction action = new Action()
       {
         public Object run() throws Exception{
```

```
            String permissionName = getPermissionName(thisJoinPoint.
                getSignature());
            AuthPermission perm = new AuthPermission(permissionName);
            perm.checkGuard(null); //throws exception if not having
                permission
            proceed();//execute the original code that way
            return null;
          }

        };

        // Enforce Access Controls
        Subject.doAs(subject, action);
    }catch (Exception e){e.printStackTrace();}
  }

}
```

Calls	Secured Manually	Secured with AspectJ
1	49 ms	98 ms
100	357 ms	497 ms
500	1019 ms	1480 ms

Table 4
Execution Time For Different Approaches in Java

6.3 Encryption

We have implemented some utility functions, as for the secure connection, that allow to encrypt data of all types in C. In order to keep type compatibility, we generate a counter that is different for `short`, `char` and `int`, which normally include pointers. The encrypted values as well as cryptographic information are stored in a hash table that uses the aforementioned pointer as a key. One limitation of this strategy is the limited range number of information that can be kept this way, especially for `char` and `short` values. Our results show the use of our utility functions for encrypting and decrypting information in memory. However, contrary to other examples, we were not able to implement an aspect in order to encrypt one of the buffers in our reference program (see Listing 10) due to limitations in the current AOP languages for C (AspectC++ and AspectC). These languages do no allow to specify a join point over a variable name.

Listing 10. Example Code for which it is impossible to isolate one call to harden using aspects

```
/*get username*/
fgets(username, FIELD_LEN, stdin);

/*get password*/
fgets(password, FIELD_LEN, stdin);
```

7 Appropriateness of AOP for Security Hardening

Previous work and our practical experiments show that AOP is useful for security hardening, although there are limitations in the current technologies. Such limitations forced us to perform programming gymnastics, e.g. passing of parameters, resulting in additional modules that must be integrated with the application, at a definitive runtime, memory and development cost. Moreover, the resulting code after applying this strategy of coding is of higher level of complexity as regards to auditing and evaluation. Since many organizations require high level of assurance, this problem should be addressed. We offer preliminary solutions by proposing new AOP primitives in the pointcut/advice model. Appreciated and useful additions would be the `FirstCommonAncestor`, `LastCommonDescendant`, `PassAsParameterInIntermediaryFunctions`, `ChangeMethodSignature`, `InDataFlowOf`, `MarkMatching`, `NeedsFromMarked`, and `RetrieveParameterNameInCallingContext`. Such primitives, however, are to be developed and specified with great care, as they could have an enormous impact on program's interfaces and design. Furthermore, we have shown that the inability to specify pointcuts with the precise variable names used as parameters to functions limits the ability of aspect-oriented programming to be used for certain cases of hardening. In this context, we briefly overview previous findings on the shortcomings of the available AOP languages for security [28], particularly in AspectJ, and we present some possible extensions to AspectJ in order to handle security issues:

Predicated Control Flow Pointcut The predicted control flow pointcut (pcflow) has been proposed as an idea, but is not yet integrated in AspectJ. It allows to select points in the execution within the control flow of a method. For instance, a control flow pointcut can select a point in the control flow of a method where a variable has been modified.

Dataflow Pointcut The dataflow pointcut is defined by [29] for security purposes. It is used to identify join points based on the origin of values. This poincut is not implemented yet. For instance, such pointcut permits to to detect if the data sent over the network depends on information read from a confidential file. Another use for this pointcut is during the hardening process, where the decision of injecting some code depends directly on the value of some variables.

Loop Pointcut The loop pointcut detects the infinite loops used by attackers to perform denial of service of attacks. In this context, Harbulot and Gurd presents in [30] a model that explores the need to a loop joint point that predicts whether a code will ever halt or run for ever.

Local Variables Set and Get Pointcut A pointcut that allows to track the values of local variables inside a method is necessary to increase the efficiency of AOP in security hardening [31]. For instance, confidential data can be protected using such type of pointcut by writing advices before and

after the use of these variables.

Synchronized Block Pointcut This pointcut, which has been discussed in [32], is needed to detect the beginning of a synchronized block and add some security code that limits the CPU usage or the number of instructions executed. Borner explored in his paper [32] the usefulness of capturing synchronized block in calculating the time acquired by a lock and thread management. This usefulness applies also in the security context and can help in preventing many denial of service attacks.

8 Conclusion and Future Work

We presented in this paper a framework that illustrates our proposition and methods to harden security into applications. This framework, which is based on AOP, simplifies security hardening by maintainers and allow security architects to perform security hardening of software by providing an abstraction over the actions required to improve the security of programs. This abstraction allows them to specify high level security hardening plans that are refined systematically to security code. In this context, we introduced first the contributions in the field of security patterns, secure programming and practices and AOP security. Afterwards, we presented our security hardening approach together with many security hardening plans, patterns and aspects for different security issues and problems. Then, we explored the experimental results and illustrated the efficiency and relevance of this approach by manually refining the elaborated patterns into aspects and then weaving them into real applications for testing and analysis. Finally, we discussed the appropriateness of AOP for security hardening.

Regarding our future work, we are currently working on addressing the shortcomings of AOP for security and elaborating new pointcuts and primitives needed for security hardening concerns. Moreover, we are going to enhance our approach and elaborate a language for security hardening plans and patterns specification. At the same time, we are trying to build bigger case studies and apply our solutions on wider range of open source software.

References

[1] B. Blakley, C. Heath, Security design patterns, Tech. Rep. G031, Open Group, `http://www.opengroup.org/security/gsp.htm` (Accessed April 2007) (2004).

[2] A. Braga, C. Rubira, R. Dahab, Tropyc: A pattern language for cryptographic

software, http://citeseer.ist.psu.edu/braga99tropyc.html (Accessed April 2007) (1998).

[3] E. B. Fernandez, R. Warrier, Remote authenticator/authorizer, in: Proceedings of the PLoP 2003, 2003.

[4] F. B. Jr., E. B. Fernandez, The authenticator pattern, in: Proceedings of the PLoP 99, 1999.

[5] D. M. Kienzle, M. C. Edler, Final technical report: Security patterns for web application development, Tech. Rep. DARPA Contract # F30602-01-C-0164, http://www.modsecurity.org/archive/securitypatterns/dmdj_final_report.pdf (Accessed April 2007) (2002).

[6] D. M. Kienzle, M. C. Elder, D. Tyree, J. Edwards-Hewitt, Security patterns repository, http://www.modsecurity.org/archive/securitypatterns/dmdj_repository.pdf (Accessed April 2007) (2002).

[7] S. Romanosky, Security design patterns part 1, http://www.romanosky.net/ (Accessed April 2007) (2001).

[8] M. Schumacher, Security Engineering with Patterns, Springer, 2003.

[9] J. Yoder, J. Barcalow, Architectural patterns for enabling application security, in: Proceedings of the PLoP 97, 1997.

[10] M. Bishop, Computer Security: Art and Science, Addison-Wesley Professional, 2002.

[11] M. Howard, D. E. Leblanc, Writing Secure Code, Microsoft Press, Redmond, WA, USA, 2002.

[12] R. Seacord, Secure Coding in C and C++, SEI Series, Addison-Wesley, 2005.

[13] D. Wheeler, Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010, 2003, http://www.dwheeler.com/secure-programs/ (Accessed April 2007).

[14] R. Bodkin, Enterprise security aspects, http://citeseer.ist.psu.edu/702193.html (Accessed April 2007) (2004).

[15] B. DeWin, Engineering application level security through aspect oriented software development, Ph.D. thesis, Katholieke Universiteit Leuven (2004).

[16] M. Huang, C. Wang, L. Zhang, Toward a reusable and generic security aspect library, in: AOSD:AOSDSEC 04: AOSD Technology for Application level Security, March, 2004.

[17] Cigital Labs, An aspect-oriented security assurance solution, Tech. Rep. AFRL-IF-RS-TR-2003-254 (2003).

[18] P. Slowikowski, K. Zielinski, Comparison study of aspect-oriented and container managed security, in: Proceedings of the ECOOP workshop on Analysis of Aspect-Oriened Software, 2003.

[19] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad, Security Patterns: Integrating Security and Systems Engineering, Wiley, 2006.

[20] C. Shah, F. Hill, Using aspect-oriented programming for addressing security concerns, in: Procceedings of the Thirteenth International Symposium on Software Reliability Engineering (ISSRE), 2002, pp. 115–119.

[21] C. Shah, F. Hill, Aspect-oriented programming security framework, in: Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX 03), IEEE Press, 2003, pp. 143–145.

[22] J. Viega, J. Bloch, C. Pravir, Applying aspect-oriented programming to security, Cutter IT Journal 14 (2) (2001) 31–39.

[23] B. DeWin, B. Vanhaute, B. D. Decker, Security through aspect-oriented programming, in: Proceedings of the IFIP TC11 WG11.4 First Annual Working Conference on Network Security, 2001, pp. 193–194.

[24] B. DeWin, B. Vanhaute, B. D. Decker, How aspect-oriented programming can help to build secure software, Informatica (Slovenia) Journal 26 (2) (2002) 141–149.

[25] B. Vanhaute, B. DeWin, Security and genericity, in: Proceedings of the 1st Belgian AOSD Workshop, Belgium, November 8, 2001.

[26] A. Mourad, M.-A. Laverdière, M. Debbabi, Security hardening of open source software, in: Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006), ACM, 2006.

[27] Bastille linux, `http://www.bastille-linux.org/` (Accessed April 2007) (2006).

[28] D. AlHadidi, N. Belblidia, M. Debbabi, Security crosscutting concerns and aspectj, in: Proceedings of the 2006 International Conference on Privacy, Security and Trust (PST 2006), ACM, 2006.

[29] H. Masuhara, K. Kawauchi, Dataflow pointcut in aspect-oriented programming, in: Proceedings of The First Asian Symposium on Programming Languages and Systems (APLAS'03), 2003, pp. 105–121.

[30] B. harbulot, J. Gurd, A join point for loops in AspectJ, in: Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005), March, 2005.

[31] A. Myers, Jflow: Practical mostly-static information flow control, in: Symposium on Principles of Programming Languages, 1999, pp. 228–241.

[32] J. Bonr, Semantics for a synchronized block join point, `http://jonasboner.com/2005/07/18/semantics-for-a-synchronized-block-joint-point/` (Accessed April 2007) (2005).