Towards an Aspect Oriented Approach for the Security Hardening of Code

Azzam Mourad, Marc-André Laverdière and Mourad Debbabi

Computer Security Laboratory,
Concordia Institute for Information Systems Engineering,
Concordia University, Montreal (QC), Canada
{mourad,ma_laver,debbabi}@ciise.concordia.ca *

Abstract

In this paper, we present an approach revolving around aspect-oriented software development (AOSD) for the systematic security hardening of source code. It provides an abstraction over the actions required to improve the security of the program. Security architects can specify high level security hardening plans that leverages a priori defined security hardening patterns. These patterns describe the steps and actions required for hardening, 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.

1 Motivations & 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. 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 (AOP). Many Security Design Patterns (SDP) are available [1, 7, 9, 12] in order to guide software engineers in designing their security models and securing their applications at the design phase. For security hardening purposes, these proposals are of a lower relevance since we are dealing with applications in the maintenance phase. The secure programming approach, is centered around good programming practices, with a strong focus on secure C/C++ programming [5, 10, 11]. 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 the injection of security code into an application using AOP. 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.

As a result, integrating security in 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 system-

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

21st International Conference on
Advanced Information Networking and Applications Workshops (AINAW'07)
0-7695-2847-3/07 $20.00 © 2007 IEEE

atically security models and components into FOSS. Our proposition is 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 practical framework for 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. Then, in Section 3, we present our security hardening approach together with many security hardening plans, patterns and aspects for different security issues and problems. In Section 4. Finally, we offer concluding remarks in Section 5.

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, the terminology of "hardening" that we propose is not the same as for operating system hardening.

The current research on the topic of security design patterns is characterized by various publications, of which we mention the most noteworthy. In [12], 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. [7] have created a 29-pattern security pattern repository for web applications, which categorized security patterns as being either structural (i.e. implementable in software) or procedural (i.e. related to the development process).The Open Group [1] has possibly introduced the most mature design pattern catalog so far with 13 patterns. The most recent work in this domain is from Schumacher et al. [9], offering a list of forty-six patterns applied in different fields of software security, although most of them are the rewriting of previously proposed patterns.

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 [10], 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 [5]. The authors also describe low-level andhigh-level security issues, as well as threat modeling, access control, etc.

Regarding the use of AOP for security, the following is a brief overview on the available contributions. Cigital labs proposed an AOP language called CSAW [3], which is a small superset of C programming language dedicated to improve the security of C programs. De Win, in his Ph.D. thesis [4], discussed an aspect-oriented approach that allowed the integration of security aspects within applications. It is based on AOSD concepts to specify the behavior code to be merged in the application and the location where this code should be injected. In [2], Ron Bodkin surveyed the security requirements for enterprise applications and described examples of security crosscutting concerns, with a focus on authentication and authorization. Another contribution in AOP security is the Java Security Aspect Library (JSAL), in which Huang et al. [6] introduced and implemented, in AspectJ, a reusable and generic aspect library that provides security functions. These approaches may be useful, but they require the developer to be a security expert who knows exactly where each piece of code should be manually injected.

3 Security Hardening Approach

This section illustrates our proposition to harden security into applications. We proposed a definition and taxonomy of security hardening methods in [8]. 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.*

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.

Each component participates by playing a role and/or providing functionalities in order to have a complete security hardening process. The security architect is the person responsible of writing plans by deriving them from the security requirements. These plans contains the abstract actions required for security hardening and uses the security hardening patterns that are developed by security experts and provided into a catalog. The security APIs constitute the building blocks

21st International Conference on
Advanced Information Networking and Applications Workshops (AINAW'07)
0-7695-2847-3/07 $20.00 © 2007 IEEE

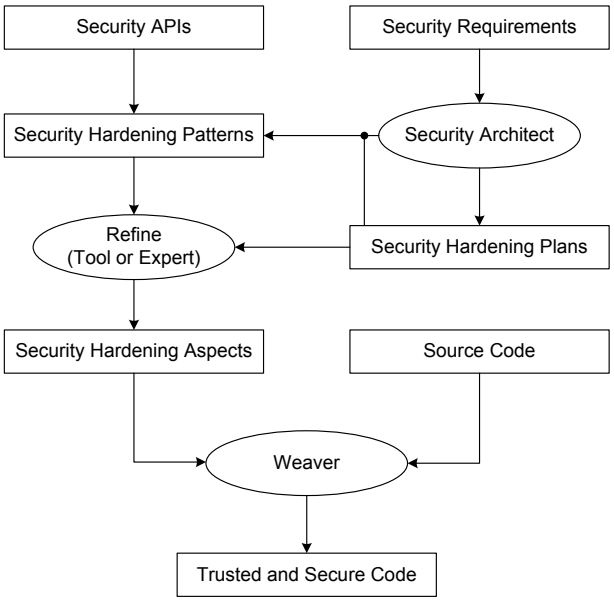

Figure 1. Schema of Our Approach

used by the patterns to achieve the desired solutions.

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 the specification of hardening plans that use and instantiate the security hardening 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. The combination of hardening plans and patterns constitutes the concrete security hardening 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 languages. The description of this language is not discussed in this paper. 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.

3.1 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. In Listing 1, we include an example of an effective security hardening plans for securing connection and adding authorization, together with a free-form syntax template. The patterns used by these plans are presented in section 3.2.

Listing 1. Our Hypothetical Hardening Plan

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

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

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

3.2 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 and performing authorization. We also developed a pattern for encrypting memory, which we did not include here, because we were not able to implement its corresponding aspect, due to limitations in the current AOP technologies. Those patterns are used by the hardening plans presented in Listing 1, and later refined and validated in Section 4. Currently, we defined the aforementioned patterns 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.

Different forms of patterns' representation and specification will be proposed in future work.

3.2.1 Secure Connection

A first issue is the securing of channels between two communicating parties to avoid eavesdropping, tampering with the transmission or session hijacking. We thus present a pattern that secures a connection using TLS, a popular network protocol for this task. The usage scenario, around which the pattern in Listing 2 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 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 providing 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. 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 4.1 for more details.

Listing 2. Hardening Pattern for Secure Connection in C

```
Before being used:
    - Initialize the TLS library
    - If desired, load an additional trust store

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
    - 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
```

3.2.2 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 3 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 3. 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.
```

4 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 are dealing with security requirements such as securing a connection, authorization and encrypting some information in the memory. During the course of our study, we developed some utility functions in C and Java, some example code and some aspects in AspectC++ (1.0pre3) and AspectJ (1.5.2) that implement security hardening of the cases described previously. However, wee were not able to implement an aspect in order to encrypt one of the buffers in our reference program due to limitations in the current AOP languages for C (AspectC++ and AspectC), as these languages do no allow to specify a join point over a variable name. Thus, we will not provide details of this example here. We will show some of our findings here. To illustrate our approach, we performed the following steps:

1. We implemented applications to harden
2. We hardened the application with minimum changes

3. We developed patterns that describe in a clear and abstract way those steps

4. We refined our patterns into aspects and weaved them into the applications

5. We tested the resulting programs for functional and security correctness

6. We measured the performance of the programs using both approaches

The latter measurements between the performance cost of hardening manually or using aspect-oriented technologies show no significant overhead, demonstrating 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.

4.1 Secure Connection

We refined and implemented in Listing 4 the pattern presented in Listing 2 using AspectC++ aspects that use the GnuTLS library. The scenario considered is presented in Section 3.2.1. The reader will notice the appearance of `hardening_sockinfo_t`. These are the data structure 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()` and `recv()` are modified for a runtime check that uses the proper sending/receiving function. The introduction of new AOP primitives may avoid these programming gymnastics.

Listing 4. Excerpt of Aspect for Securing Connections

```
aspect SecureConnection {

advice execution ("% main(...)") : around () {
    /*Initialization of the API*/
    /*...*/
    tjp->proceed();
    /*De-initialization of the API*/
    /*...*/
    *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);
    /*...*/

    //Connect
    tjp->proceed();
    if(*tjp->result()<0) {return;}

    //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)
       *(tjp->result())  = gnutls_record_send(
           socketInfo.session, *(char**) tjp->arg(1)
           , *(int *)tjp->arg(2));
    else
       tjp->proceed();
}
};
```

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 our web server is established, and a few index page's bytes are retrieved. We can observe in Table 1 that there is no significant performance difference between the hardening approaches, especially when taking in consideration traffic and network fluctuations to our web server.

4.2 Authorization

We have implemented an example of access control as an AspectJ aspect (see Listing 5) that uses JAAS

Connections	Unsecured	GnuTLS	
		Manual	AOP
100	0.234 s	15.937 s	15.953 s
500	1.406 s	79.39 s	79.828 s
1000	3.062 s	159.984 s	161.25 s

Table 1. Execution Time For Different Approaches in C

Calls	Unsecured	Secured	
		Manually	AspectJ
1	<1 ms	31 ms	78 ms
100	<1 ms	453 ms	453 ms

Table 2. Execution Time For Adding Access Control

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 obtain the user name from the virtual machine. The permissions are specified in the format `package.class.function`. As our runtime experiments show (c.f. Table 2), shows that there is no significant difference in execution time for multiple runs of a security check between a case hardened manually and one hardened using AspectJ.

Listing 5. Aspect for Adding Authorization

```
public aspect AddAccessControl {
abstract class Action implements
     PrivilegedExceptionAction{};
pointcut test(): call(void doSomething2());

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

before(): test(){try{
    System.setProperty("java.security.auth.login.
        config","jaas.config");

    //Fetch the user information from the OS
    LoginContext lc = new LoginContext("NT");
    lc.login();
    Subject subject = lc.getSubject();

    //anonymous inner class for the privileged action
    PrivilegedExceptionAction action = new Action()
    {
      public Object run() throws Exception{
          String permissionName = getPermissionName(
              thisJoinPoint.getSignature());
          //throws exception if not having permission
          BasicPermission perm = new
              HardeningPermission(permissionName);
          perm.checkGuard(null);
          return null;
      }
    };
    // Enforce Access Controls
    Subject.doAsPrivileged(subject, action,null);
    lc.logout(); //not necessary for us
}catch (Exception e){e.printStackTrace();}}}
```

5 Conclusion

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 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.

References

[1] B. Blakley and C. Heath. Security design patterns. Technical Report G031, Open Group, 2004.

[2] R. Bodkin. Enterprise security aspects. In *AOSD:AOSDSEC 04*.

[3] Cigital Labs. An aspect-oriented security assurance solution. Technical Report AFRL-IF-RS-TR-2003-254, 2003.

[4] B. DeWin. *Engineering Application Level Security through Aspect Oriented Software Development*. PhD thesis, Katholieke Universiteit Leuven, 2004.

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

[6] M. Huang, C. Wang, and L. Zhang. Toward a reusable and generic security aspect library. In *AOSD:AOSDSEC 04*, 2004.

[7] D. M. Kienzle, M. C. Elder, D. Tyree, and J. Edwards-Hewitt. Security patterns repository, 2002.

[8] A. Mourad, M.-A. Laverdière, and M. Debbabi. Security hardening of open source software. In *Proceedings of PST 2006*. ACM, 2006.

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

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

[11] D. Wheeler. *Secure Programming for Linux and Unix HOWTO – Creating Secure Software v3.010*. 2003. `http://www.dwheeler.com/secure-programs/`.

[12] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. In *Proceedings of the PLoP 97*, 1997.

0-7695-2847-3/07 $20.00 © 2007 IEEE