A Security Hardening Language Based on Aspect-Orientation

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 propose an aspect-oriented language, called *SHL* (Security Hardening Language), for specifying systematically the security hardening solutions. This language constitutes our new achievement towards developing our security hardening framework. *SHL* allows the description and specification of security hardening plans and patterns that are used to harden systematically security into the code. It is a minimalist language built on top of the current aspect-oriented technologies that are based on advice-poincut model and can also be used in conjunction with them. The primary contribution of this approach is providing the security architects with the capabilities to perform security hardening of software by applying well-defined solution and without the need to have expertise in the security solution domain. At the same time, the security hardening is applied in an organized and systematic way in order not to alter the original functionalities of the software. We explore the viability and relevance of our proposition by applying it into a case study and presenting the experimental results of securing the connections of open source software.

Key words: Software Security Hardening, Aspect-Oriented Programming (AOP), Security Hardening Patterns, Security Hardening Plans, Trusted and Open Source Software (FOSS), Aspect-Oriented Language

1 Introduction

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

* 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.

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.

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, introduced in [1], is based on aspect-oriented programming AOP and inspired by the best and most relevant methods and methodologies available in the literature, in addition to elaborating valuable techniques that permit us to provide a framework for systematic security hardening.

The main components of our approach are the security hardening plans and patterns that provide an abstraction over the actions required to improve the security of a program. They should be specified and developed using an abstract, programming language independent and aspect-oriented (AO) based language. The current AO languages, however, lack many features needed for systematic security hardening. They are programming language dependent and could not be used to write and specify such high level plans and patterns, from which the need to elaborate a language built on top of them to provide the missing features. In this context, we propose a language called *SHL* for security hardening plans and patterns specification. It allows the developer to specify high level security hardening plans that leverage priori defined security hardening patterns, which are also developed using *SHL*.

This paper provides our new contributions in developing our security hardening framework. The experimental results presented together with the security hardening plans and patterns, which are elaborated using *SHL*, 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 AOP languages for securing software. Afterwards, in Section 3, we summarize our approach for systematic security hardening. Then, in Section 4, we present the syntax and semantics of *SHL*. After that, in Section 5, we illustrate the useability of *SHL* into case studies. Finally, we offer concluding remarks in Section 6.

2 Related Work

AOP is a relatively new programming paradigm that allows the injection of security components within applications. It appears to be a promising paradigm for software security. In this Section, we present an overview on some AOP languages and the use of AOP for software security. The related work on the current approaches for securing software [2, 3] (e.g. security design patterns, secure coding) has been discussed in [1].

There are many AOP languages that have been developed. However, these languages are used for code implementation and programming language dependent. Thus, they cannot be used to specify abstract security hardening plans and patterns, which is a requirement in our proposition. We distinguish from them AspectJ [6] built on top of the Java programming language, AspectC [7] built on top of the C programming language, AspectC++ [8] built on top of the C++ programming language, AspectC# [9] built on top of the C Sharp programming language and the AOP version addressed for Smalltalk programming language [10]. AspectJ and AspectC++ are dominant propositions in the field of AOP.

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 [11], which is a small superset of C programming language dedicated to improve the security of C programs. De Win, in his Ph.D. thesis [5], 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 [4], 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. [12] introduced and implemented, in AspectJ, a reusable and generic aspect library that provides security functions. These research initiatives, however, focus on exploring the usefulness of AOP for securing software by security experts who know exactly where each piece of code should be manually injected and/or proposing AOP languages for security. None of them proposed an approach or methodology for systematic security hardening with features similar to our approach.

3 Security Hardening Approach

This section illustrates a summary of our whole approach for systematic security hardening. It also explores the need and usefulness of *SHL* to achieve our objectives. The approach architecture is illustrated in Figure 1.

The primary objective of this approach is to allow the developers to perform security hardening of FOSS by applying well-defined solutions and without the need to have expertise in the security solution domain. At the same time, the security hardening should be applied in an organized and systematic way in order not to alter the original functionalities of the software. This is done by providing an abstraction over the actions required to improve the security of the program and adopting AOP to build and develop our solutions. The developers are able to specify the hardening plans that use and instantiate the security hardening patterns using the proposed language *SHL*.

The abstraction of the hardening plans is bridged by concrete steps defined in the hardening patterns using also *SHL*. 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. We built *SHL* on top of the current AOP languages because we believe, after a deep investigation on the nature of security hardening practices and the experimental results we got, that aspect orientation is the most natural and appealing approach to reach our goal.

Once the security hardening solutions are built, the refinement of the solutions into aspects or low level code can be performed using a tool or by programmers that do not need to have any security expertise. Afterwards, an AOP weaver (e.g. AspectJ, AspectC++) can be executed to harden the aspects into the original source code, which

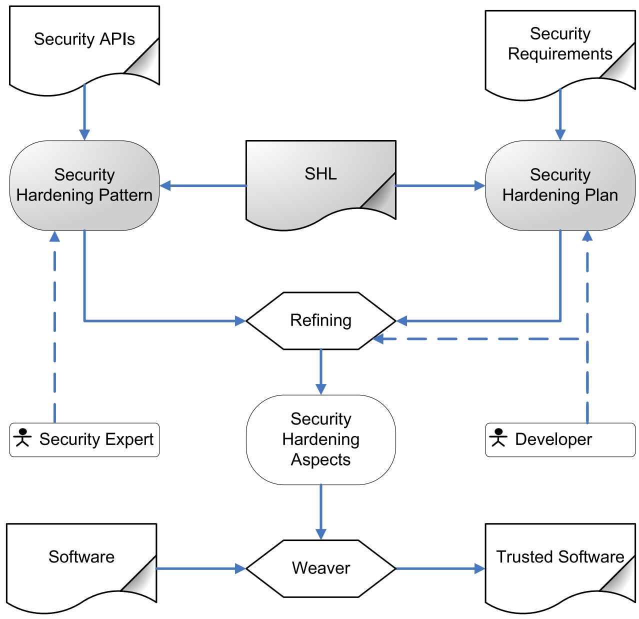

Fig. 1: Schema of Our Approach

can now be inspected for correctness. As a result, the approach constitutes a bridge that allows the security experts to provide the best solutions to particular security problems with all the details on how and where to apply them, and allows the software engineers to use these solutions to harden FOSS by specifying and developing high level security hardening plans.

4 *SHL* Language

Our proposed language, *SHL*, allows the description and specification of security hardening patterns and plans that are used to harden systematically security into the code. It is a minimalist language built on top of the current AOP technologies that are based on advice-pointcut model. It can also be used in conjunction with them since the solutions elaborated in *SHL* can be refined into a selected AOP language (e.g. AspectC++) as illustrated in Section 5. We developed part of *SHL* with notations and expressions close to those of the current AOP languages but with all the abstraction needed to specify the security hardening plans and patterns. These notations and expressions are programming language independent and without referring to low-level implementation details. The following are the main features provided by *SHL*:

- Automatic code manipulation such as code addition, substitution, deletion, etc.
- Specification of particular code join points where security code would be injected.
- Modification of the code after the development life cycle since we are dealing with already existing open source software.

- Modification of the code in an organized way and without altering its functional attributes.
- Description and specification of security.
- Dedicated to describe and specify reusable security hardening patterns and plans.
- Parameterized language to allow the instantiation of the security hardening patterns through the security hardening plans.
- Programming language independent.
- Highly expressive and easy to use by security non experts.
- Intermediary abstractness between English and programming languages.
- Easily convertible to available AOP languages (e.g. AspectJ and AspectC++).

4.1 Grammar and Structure

In this section, we present the syntactic constructs and their semantics in *SHL*. Table 1 illustrates the `BNF` grammar of *SHL*. The language that we arrived at can be used for both plans and patterns specification, with a specific template structure for each of them. We implemented this language specification using ANTLR V3 Beta 6 and its associated ANTLRWorks development environment. We were also able to validate the syntax of different plan and pattern examples within this tool. The work on the language implementation is still in progress. Examples of security hardening plans and patterns are elaborated using *SHL* and presented in Section 5.

Hardening Plan Structure A hardening plan starts always with the keyword `Plan`, followed by the plan's name and then the plan's code that starts and ends respectively by the keywords `BeginPlan` and `EndPlan`. Regarding the plan's code, it is composed of one or many pattern instantiations that allow to specify the name of the pattern and its parameters, in addition to the location where it should be applied. Each pattern instantiation starts with the keyword `PatternName` followed by a name, then the keyword `Parameters` followed by a list of parameters and finally by the keyword `Where` followed by the module name where the pattern should be applied (e.g. file name).

Hardening Pattern Structure A hardening pattern starts with the keyword `Pattern`, followed by the pattern's name, then the keyword `Parameters` followed by the matching criteria and finally the pattern's code that starts and ends respectively by the keywords `BeginPattern` and `EndPattern`. The matching criteria are composed of one or many parameters that could help in distinguishing the patterns with similar name and allow the pattern instantiation. The pattern code is based on AOP and composed of one or many `Location_Behavior` constructs. Each one of them constitutes the location identifier and the insertion point where the behavior code should be injected, the optional primitives that may be needed in applying the solution and the behavior code itself. A detailed explanation of the components of the pattern's code will be illustrated in Section 4.2.

4.2 Semantics

In this Section, we present the semantics of the important syntactic constructs in *SHL* language.

Start	::= *SH_Plan*	
	| *SH_Pattern*	
SH_Plan	::= `Plan`	*Plan_Name*
	SH_Plan_Code	
Plan_Name	::= *Identifier*	
SH_Plan_Code	::= `BeginPlan`	
	*Pattern_Instantiation**	
	`EndPlan`	
Pattern_Instantiation	::= `PatternName`	*Pattern_Name*
	(`Parameters`	*Pattern_Parameter**)?
	`Where`	*Module_Identification+*
Pattern_Name	::= *Identifier*	
Pattern_Parameter	::= *Parameter_Name*	*Parameter_Value*
Parameter_Name	::= *Identifier*	
Parameter_Value	::= *Identifier*	
Module_Identification	::= *Identifier*	
SH_Pattern	::= `Pattern`	*Pattern_Name*
	Matching_Criteria?	
	SH_Pattern_Code	
Matching_Criteria	::= `Parameters`	*Pattern_Parameter+*
SH_Pattern_Code	::= `BeginPattern`	
	*Location_Behavior**	
	`EndPattern`	
Location_Behavior	::= *Behavior_Insertion_Point+*	*Location_Identifier+*
	Primitive?*	
	Behavior_Code	
Behavior_Insertion_Point	::= `Before`	
	| `After`	
	| `Replace`	
Location_Identifier	::= `FunctionCall <Signature>`	
	| `FunctionExecution <Signature>`	
	| `WithinFunction <Signature>`	
	| `CFlow <Location _Identifier>`	
	| `GAflow <Location _Identifier>`	
	| `GDFlow <Location _Identifier>`	
	| ...	
Signature	::= *Identifier*	
Primitive	::= `ExportParameter <Identifier>`	
	| `ImportParameter <Identifier>`	
	| ...	
Behavior_Code	::= `BeginBehavior`	
	Code_Statement	
	`EndBehavior`	

Table 1: Grammar of *SHL*

Pattern_Instantiation Specifies the name of the pattern that should be used in the plan and all the parameters needed for the pattern. The name and parameters are used as matching criteria to identify the selected pattern. The module where the pattern should be applied is also specified in the `Pattern_Instantiation`. This module can be the whole application, file name, function name, etc.

Matching_Criteria Is a list of parameters added to the name of the pattern in order to identify the pattern. These parameters may also be needed for the solutions specified into the pattern.

Location_Behavior Is based on the advice-pointcut model of AOP. It is the abstract representation of an aspect in the solution part of a pattern. A pattern may include one or many `Location_Behavior`. Each `Location_Behavior` is composed of the `Behavior_Insertion_Point`, `Location_Identifier`, one or many `Primitive` and `Behavior_Code`.

Behavior_Insertion_Point Specifies the point of code insertion after identifying the location. The `Behavior_Insertion_Point` can have the following three values: `Before`, `After` or `Replace`. The `Replace` means remove the code at the identified location and replace it with the new code, while the `Before or After` means keep the old code at the identified location and insert the new code before or after it respectively.

Location_Identifier Identifies the joint point or series of joint points in the program where the changes specified in the `Behavior_Code` should be applied. The list of constructs used in the `Location_Identifier` is not yet complete and left for future extensions. Depending on the need of the security hardening solutions, a developer can define his own constructs. However, these constructs should have their equivalent in the current AOP technologies or should be implemented into the weaver used. In the sequel, we illustrate the semantics of some important constructs used for identifying locations:

`FunctionCall <Signature>` Provides all the join points where a function matching the signature specified is called.

`FunctionExecution <Signature>` Provides all the join points referring to the implementation of a function matching the signature specified.

`WithinFunction <Signature>` Filters all the join points that are within the functions matching the signature specified.

`CFlow <Location_Identifier>` Captures the join points occurring in the dynamic execution context of the join points specified in the input `Location_Identifier`.

`GAflow <Location_Identifier>` Operates on the control flow graph (CFG) of a program. Its input is a set of join points defined as a `Location_Identifier` and its output is a single join point. It returns the closest ancestor join point to the join points of interest that is on all their runtime paths. In other words, if we are considering the CFG notations, the input is a set of nodes and the output is one node. This output is the closest common ancestor that constitutes (1) the closet common parent node of all the nodes specified in the input set (2) and through which passes all the possible paths that reach them.

`GDFlow <Location_Identifier>` Operates on the CFG of a program. Its input is a set of join points defined as a `Location_Identifier` and its output is a single join point. It returns the closest child join point that can be reached by all paths starting from the join points of interest. In other words, if we are considering the CFG notations, the input is a set of nodes and the output is one node. This output (1) is a common descendant of the selected nodes and (2) constitutes the first common node reached by all the possible paths emanating from the selected nodes.

The `Location_Identifier` constructs can be composed with algebraic operators to build up other ones as follows:

`Location_Identifier && Location_Identifier` Returns the intersection of the join points specified in the two constructs.
`Location_Identifier || Location_Identifier` Returns the union of the join points specified in the two constructs.
`! Location_Identifier` Excludes the join points specified in the construct.

Primitive Is an optional functionality that allows to specify the variables that should be passed between two `Location_Identifier` constructs. The following are the constructs responsible of passing the parameters:

`ExportParameter <Identifier>` Defined at the origin `Location_Identifier`. It allows to specify a set of variables and make them available to be exported.
`Importparameter <Identifier>` Defined at the destination `Location_Identifier`. It allows to specify a set of variables and import them from the origin `Location_Identifier` where the `ExportParameter` has been defined.

Behavior_Code May contain code written in any programming language, or even written in English as instructions to follow, depending on the abstraction level of the pattern. The choice of the language and syntax is left to the security hardening pattern developer. However, the code provided should be abstract and at the same time clear enough to allow a developer to refine it into low level code without the need to high security expertise. Example of such code behavior is presented in Listing 1.2.

5 Case Study: Securing Connection of Client Applications

In this section, we illustrate our elaborated solutions for securing the connections of client applications by following the approach's methodology and using the proposed *SHL* language. In this context, we developed our own client application and selected an open source software called APT to secure their connections using GnuTLS/SSL library. Our application, which is a client implemented in C, allows to connect and exchange data with a selected server, typically an HTTP request.

Regarding APT, it is an automated package downloader and manager for the Debian Linux distribution. It is written in C++ and is composed of more than 23 000 source lines of code (based on version 0.5.28, generated using David A. Wheeler's 'SLOC-Count'). It obtains packages via local file storage, FTP, HTTP, etc. We have decided to add HTTPS support to these two applications. In the sequel, we are going to present the hardening plan, pattern and aspect elaborated to secure the connections of APT.

5.1 Hardening Plan

In Listing 1.1, we include an example of effective security hardening plan for securing the connection of the APT software. The hardening plan of the our client application will be the same, except for the plan's name and the modules where the patterns should be applied (i.e. the files' names specified after `Where`).

Listing 1.1: Hardening Plans for Securing Connection

```
Plan     APT_Secure_Connection_Plan BeginPlan
    PatternName     Secure_Connection_Pattern
    Parameters
            Language     C/C++
            API          GNUTLS
            Peer         Client
            Protocol     SSL
    Where    http.cc  connect.cc
EndPlan
```

5.2 Hardening Pattern

Listing 1.2 presents the solution part of the pattern for securing the connection of the two aforementioned applications using GnuTLS/SSL. The code of the functions used in the `Code_Behavior` parts of the pattern is illustrated in Listing 1.3. It is expressed in C++ because our applications are implemented in this programming language. However, other syntax and programming languages can also be used depending on the abstraction required and the implementation language of the application to harden.

Listing 1.2: Hardening Pattern for Securing Connection

```
Pattern  Secure_Connection_Pattern
Parameters
     Language     C/C++
     API          GNUTLS
     Peer         Client
     Protocol     SSL
BeginPattern

Before
FunctionExecution   <main> //Starting Point
BeginBehavior
    // Initialize the TLS library
    InitializeTLSLibrary;
EndBehavior
Before
FunctionCall   <connect> //TCP Connection
ExportParameter   <xcred>
ExportParameter   <session>
BeginBehavior
    // Initialize the TLS session resources
    InitializeTLSSession;
EndBehavior
After
FunctionCall   <connect>
ImportParameter   <session>
BeginBehavior
    // Add the TLS handshake
    AddTLSHandshake;
EndBehavior
Replace
```

```
FunctionCall  <send>
ImportParameter  <session>
BeginBehavior
    // Change the send functions using that
    // socket by the TLS send functions of the
    // used API when using a secured socket
    SSLSend;
EndBehavior
Replace
FunctionCall  <receive>
ImportParameter  <session>
BeginBehavior
    // Change the receive functions using that
    // socket by the TLS receive functions of
    // the used API when using a secured socket
    SSLReceive;
EndBehavior
Before
FunctionCall  <close> //Socket close
ImportParameter  <xcred>
ImportParameter  <session>
BeginBehavior
    // Cut the TLS connection
    CloseAndDealocateTLSSession;
EndBehavior
After
FunctionExecution  <main>
BeginBehavior
    // Deinitialize the TLS library
    DeinitializeTLSLibrary;
EndBehavior

EndPattern
```

Listing 1.3: Functions used in the pattern

```
InitializeTLSLibrary
  gnutls_global_init();
InitializeTLSSession
  gnutls_init (session, GNUTLS_CLIENT);
  gnutls_set_default_priority (session);
  gnutls_certificate_type_set_priority (session, cert_type_priority);
  gnutls_certificate_allocate_credentials(xcred);
  gnutls_credentials_set (session, GNUTLS_CRD_CERTIFICATE, xcred);
AddTLSHandshake
  gnutls_transport_set_ptr(session, socket);
  gnutls_handshake (session);
SSLSend
  gnutls_record_send(session, data, datalength);
SSLReceive
  gnutls_record_recv(session, data, datalength);
CloseAndDealocateTLSSession
  gnutls_bye(session, GNUTLS_SHUT_RDWR);
  gnutls_deinit(session);
  gnutls_certificate_free_credentials(xcred);
DeinitializeTLSLibrary
  gnutls_global_deinit();
```

5.3 Hardening Aspect

We refined and implemented (using AspectC++) the corresponding aspect of the pattern presented in Listing 1.2. Due to space limitation, Listing 1.4 shows only an excerpt of the aspect, specifically the handshake code inserted after the function `connect` .

The reader will notice the appearance of `hardening_sockinfo_t`. These are the data structure (hash table) and functions that we developed to import and export the parameters needed between the application's components at runtime (since the primitives `ImportParamter` and `ExportParameter` are not yet deployed into the weavers).

Listing 1.4: 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();
}
};
```

5.4 Experimental Results

In order to validate the hardened applications, we used the Debian apache-ssl package, an HTTP server that accepted only SSL-enabled connections. We populated the server with a software repository compliant with APT's requirements, so that APT can connect automatically to the server and download the needed metadata in the repository. Then, we weaved (using AspectC++ weaver) the elaborated aspect with the different variants of our application and APT. We first executed our own hardened application and made it connect successfully to our local HTTPS-enabled web server using HTTPS. Then, after building and deploying the modified APT package, we tested successfully its functionality by refreshing APT's package database, which forced the software to connect to both our local web server (Apache-ssl) using HTTPS and remote servers using HTTP to update its list of packages.

The experimental results in Figure 2 show the packet capture, obtained using WireShark software, of the encrypted traffic between our version of APT and its remote package repositories. The highlighted lines show TLSv1 application data exchanged in encrypted form through HTTPS connections, exploring the correctness of the security hardening process.

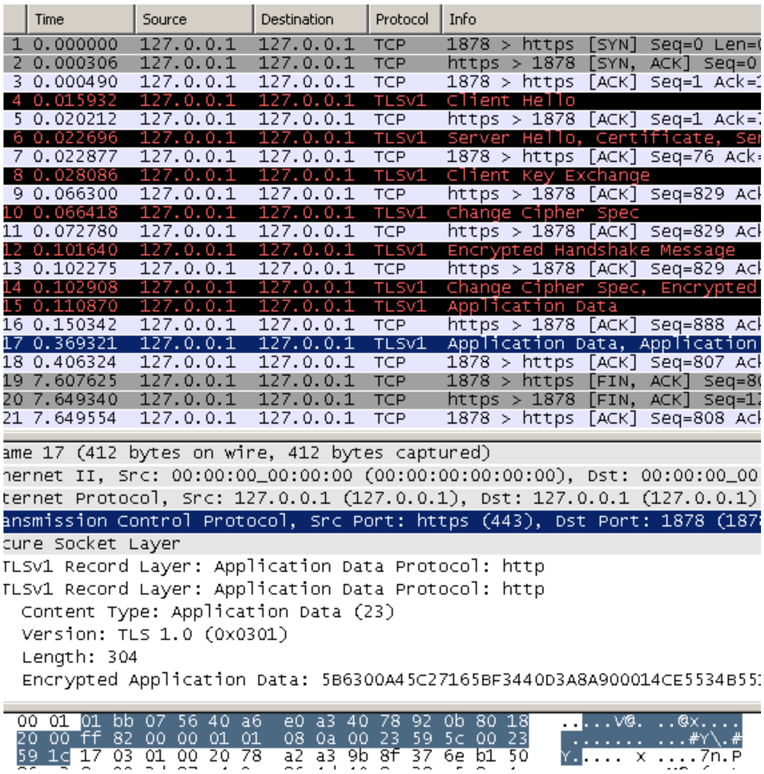

Fig. 2: Packet Capture of SSL-protected APT Traffic

6 Conclusion

We proposed in this paper a language called *SHL* for security hardening plans and patterns specification. This contribution constitutes our new accomplishment towards developing our security hardening framework. By using our approach, developers are able to perform security hardening of software in a systematic way and without the need to have expertise in the security solution domain. At the same time, it allows the security experts to provide the best solutions to particular security problems with all the details on how and where to apply them. The experimental results presented together with the security hardening plans and patterns, which are elaborated using *SHL*, explore the efficiency and relevance of our proposition.

References

1. Mourad, A., Laverdière, M.A., Debbabi, M.: Towards an aspect oriented approach for the security hardening of code. In: Proceedings of the 21st IEEE International Conference on Advanced Information Networking and Applications, SSNDS Symposium, (AINA 07), Niagara, ON, Canada, IEEE (2007)
2. Schumacher, M.: Security Engineering with Patterns. Springer (2003)
3. Bishop, M.: Computer Security: Art and Science. Addison-Wesley Professional (2002)
4. Bodkin, R.: Enterprise security aspects (2004) `http://citeseer.ist.psu.edu/702193.html` (Accessed April 2007).
5. DeWin, B.: Engineering Application Level Security through Aspect Oriented Software Development. PhD thesis, Katholieke Universiteit Leuven (2004)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Overview of aspectj. In: Proceedings of the 15th European Conference ECOOP 2001, Budapest, Hungary, Springer Verlag (2001)
7. Coady, Y., Kiczales, G., Feeley, M., Smolyn, G.: Using aspectc to improve the modularity of path-specific customization in operating system code. In: Proceedings of Foundations of software Engineering, Vienne, Austria. (2001)
8. Spinczyk, O., Gal, A., chroder Preikschat, W.: Aspectc++: An aspect-oriented extension to c++. In: Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems, Sydney, Australia. (2002)
9. Kim, H.: An aosd implementation for c#. Technical Report TCD-CS2002-55, Department of Computer Science, Trinity College, Dublin (2002)
10. Bollert, K.: On weaving aspects. In: International Workshop on Aspect-Oriented Programming at ECOOP99. (1999)
11. Cigital Labs: An aspect-oriented security assurance solution. Technical Report AFRL-IF-RS-TR-2003-254 (2003)
12. Huang, M., Wang, C., Zhang, L.: Toward a reusable and generic security aspect library. In: AOSD:AOSDSEC 04: AOSD Technology for Application level Security. (2004)