TOWARDS LANGUAGE-INDEPENDENT APPROACH FOR SECURITY CONCERNS WEAVING

Azzam Mourad, Dima Alhadidi and Mourad Debbabi
Computer Security Laboratory,
Concordia Institute for Information Systems Engineering,
Concordia University, Montreal (QC), Canada
{mourad,dm_alhad,debbabi}@encs.concordia.ca *

Keywords: Software Security, Aspect-Oriented Programming (AOP), AOP Weaving, Gimple Representation.

Abstract: In this paper, we propose an approach for weaving security concerns in the *Gimple* representation of programs. *Gimple* is an intermediate, language-independent, and tree-based representation generated by GNU Compiler Collection (GCC) during the compilation process. This proposition constitutes the first attempt towards adopting the aspect-oriented concept on *Gimple* and exploiting this intermediate representation to allow advising an application written in a specific language with security code written in a different one. At the same time, injecting security is applied in a systematic way in order not to alter the original functionalities of the software. We explore the viability and the relevance of our proposition by: (1) implementing several *Gimple* weaving capabilities into the *GCC* compiler (2) developing a case study for securing the connections of a client application and (3) using the weaving features of the extended *GCC* to inject the security concerns into the application.

1 Introduction

Security is taking an increasingly predominant role in today's computing world. 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 have not been 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. As a result, integrating security into software becomes a very challenging and interesting domain of research.

On the other hand, securing software is a difficult and a critical procedure. If it is applied manually, it requires high security expertise and lot of time to be tackled. It may also create other vulnerabilities. One way of addressing such problems is by separating out the security concerns from the rest of the application, such that they can be addressed independently and applied globally. 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 (Bodkin, 2004; DeWin, 2004; Huang et al., 2004; Shah, 2003). Aspect-Oriented Programming (AOP) is an appealing approach that allows the separation of crosscutting concerns. This paradigm seems to be very promising to integrate security into software. The process of merging the security concerns into the original program is called weaving. In this context, we have presented previously in (Mourad et al., 2007; Mourad et al., 2008) contributions towards elaborating methodologies and solutions to integrate systematically and consistently security models and components into software.

In this paper, we have built on top of them and introduced an AOP weaving approach for injecting security concerns in the *Gimple* representation of programs. *Gimple* is an intermediate, language-independent, and tree-based representation generated by GNU Compiler Collection (*GCC*) during the com-

*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.

pilation process. Our new proposition constitutes of developing AOP weaving capabilities for *Gimple* to be integrated into the *GCC* compiler. These features allow to compile security concerns and inject them into the *Gimple* tree of a program during the *GCC* compilation procedures. Beside, this approach is the first attempt towards adopting the aspect-oriented concept on *Gimple* and exploiting this intermediate representation to allow advising an application written in a specific language with security code written in a different one.

The remainder of this paper is organized as follows. In Section 2, we review the contributions in the field of AOP and AOP for securing software. Afterwards, in Section 3, we describe the new approach where weaving is performed on the *Gimple* representation of a software by adopting the aspect-oriented style. In Section 4, we present the experimental results of developing a case study for securing the connections of a client application and using the weaving features implemented in the extended *GCC* for injecting the required security concerns. Finally, we offer concluding remarks in Section 5.

2 Background and Related Work

The proposed approach is based on AOP and target security concerns. As such, we present in the sequel a brief summary on AOP and an overview on the approaches related to the contribution of this paper.

2.1 Aspect-oriented Programming

AOP depends on the principle of "Separation of Concerns", where issues that crosscut the application are addressed separately and encapsulated within aspects. There are many AOP languages that have been developed which are programming language-dependent. AspectJ (Kiczales et al., 2001) built on top of the Java programming language and AspectC++ (Spinczyk et al., 2002) built on top of the C++ programming language are the most prominent ones. The approach, which is adopted by most of the AOP languages, is called the pointcut-advice model. The fundamental concepts of this model are: join points, pointcuts, and advices.

Each atomic unit of code to be injected is called an advice. It is necessary to formulate where to inject the advice into the program. This is done by the use of a pointcut expression, which its matching criteria restricts the set of the join points of a program for which the advice will be injected. A join point is a principled point in the execution of a program. At the heart of this model, is the concept of an aspect, which embodies all these elements. Finally, the aspect is composed and merged with the core functionality modules into one single program. This process of merging and composition is called weaving, and the tools that perform such process are called weavers.

2.2 AOP Approaches for Security Injection

Most of the contributions (Bodkin, 2004; DeWin, 2004; Huang et al., 2004; Shah, 2003) that explore the usability of AOP for integrating security code into applications are presented as case studies that show the relevance of AOP languages for application security. They have focused on exploring the usefulness of AOP for securing software by security experts who know exactly where each piece of code should be manually injected. None of them have proposed an approach or methodology for systematic security hardening with features similar to our proposition. We present in the following an overview on these contributions.

Cigital labs has proposed an AOP language called CSAW (Shah, 2003), which is a small superset of C programming language dedicated to improve the security of C programs. De Win, in his Ph.D. thesis (DeWin, 2004), has discussed an aspect-oriented approach that allows the integration of security aspects within applications. It uses AOP concepts to specify the behavior code to be merged in the application and the location where this code should be injected. In (Bodkin, 2004), Ron Bodkin has 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. (Huang et al., 2004) has introduced and implemented, in AspectJ, a reusable and generic aspect library that provides security functions. Masuhara and Kawauchi (Masuhara and Kawauchi, 2003) have defined the dataflow pointcut, which identifies join points based on the origin of values.

3 Weaving Methodology

The initial proposition, which is detailed in (Mourad et al., 2007; Mourad et al., 2008), is composed of a framework, a language called *SHL* and a compiler for securing software in a systematic way. *SHL* is an aspect-oriented and programming-independent language. According to the proposed

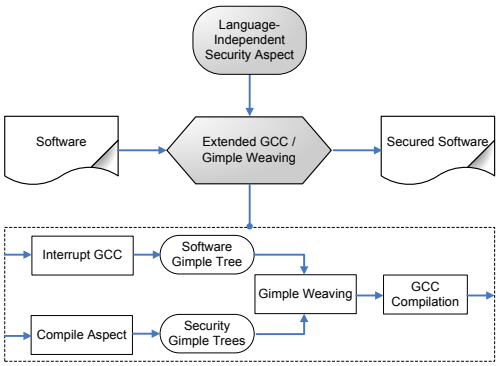

Figure 1: Approach Architecture

methodology, the resulting component, which is written in *SHL*, is a programming-language independent aspect (also known as pattern) for security hardening. Consequently, this pattern should be refined into code level aspect before passing it to the corresponding AOP weaver (e.g. AspectJ, AspectC++), which is then launched to weave it with the original code.

In this paper, we have extended this approach and built on top of it, by taking the resulting programming-language independent aspect (i.e. pattern) described using *SHL* and weaving its components directly in the *Gimple* representation (i.e. *Gimple* Tree) of a program. This allows to bypass the refinement of the pattern into programming-language dependent aspect, and consequently not using the current AOP weavers. Beside, it exploits the *Gimple* intermediate representation to advise an application written in a specific programming language with code written in a different one. The approach architecture is illustrated in Figure 1.

The methodology constitutes of passing the *SHL* security pattern and the original software to an extended version of the *GCC* compiler, which generates the executable of the trusted software. An additional pass has been added to *GCC* in order to interrupt the compilation once the *Gimple* representation of the code is completed. This pass can be called by selecting an option when performing the compilation (e.g. gcc -Weaving SecureConnectionPattern.shl -c Connection.c ...). In parallel, as temporary solutions, the components of the security pattern are translated into special format file. Additional components have been developed in order to parse such file, gather the needed information (e.g. function name, return type, etc.) and pass them as parameters to specific functions provided by *GCC* and responsible of building *Gimple* trees (e.g. build_decl(...)). Afterwards, the generated security trees are integrated in the tree of the original code (also using functions provided by *GCC* for this purposes) with respect to the location(s) specified into the pattern. Finally, the resulted *Gimple* tree is passed again to *GCC* in order to continue the regular compilation process and produce the executable of the secure software. The added features were originally implemented by our colleagues (Yang, 2007) in order to insert code for monitoring. We have modified it in order to inject security functionalities. The work on the implementation of the weaving features is still in progress.

4 Case Study

In this section, we present a case study for securing the connections of client applications. To demonstrate the feasibility of our proposition , we have elaborated first, using *SHL*, the language-independent security aspect (i.e. pattern) needed to secure the connections of a selected client application. Then, we have refined the pattern into AspectC++ aspect and weaved it into the selected application. Afterwards, we have repeated this process using our new proposition, where we have compiled directly the same application and the translated file of the pattern using the extended *GCC* and applied the weaving on the *Gimple* representation of the application.

4.1 Pattern and Aspect for Securing the Connections of Client Application

We have selected a client application implemented in C++, which allows to connect and exchange data with a server through HTTP requests. Listing 1 presents the pattern elaborated in *SHL* for securing the connection of the aforementioned application using GnuTLS/SSL (Please refer to (Mourad et al., 2007; Mourad et al., 2008) for *SHL* structure and syntax). The code of the functions used in the *Code* of the pattern's *Behavior*(s) is illustrated in Listing 2. It is expressed in C++ because the application is 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. 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 provided by 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 data are implemented in different components.

We have refined and implemented (using AspectC++) in Listing 3 the corresponding aspect of the pattern that is presented in Listing 1. The reader will notice the appearance of `hardening_sockinfo_t` . These are the data structures and functions that we have developed to distinguish between secure and non secure channels. Besides, they are used to export the parameters between the application's components at runtime. In order to avoid using shared memory directly, we have opted for a hash table that uses the socket number as a key to store and retrieve all the needed information (in our own defined data structure). One additional information that we have stored is whether the socket is secure 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.

Listing 1: *SHL* Pattern for Securing Connection

```
Pattern  Secure_Connection_Pattern
BeginPattern

Before
Execution <main> //Starting Point
BeginBehavior
    // Initialize the TLS library
    InitializeTLSLibrary;
EndBehavior

Before
Call <connect> //TCP Connection
ExportParameter   <xcred>
ExportParameter   <session>
BeginBehavior
    // Initialize the TLS session resources
    InitializeTLSSession;
EndBehavior

After
Call <connect>
ImportParameter   <session>
BeginBehavior
    // Add the TLS handshake
    AddTLSHandshake;
EndBehavior

Replace
Call <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
Call <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
Call <close> //Socket close
ImportParameter   <xcred>
ImportParameter   <session>
BeginBehavior
    // Cut the TLS connection
    CloseAndDealocateTLSSession;
EndBehavior

After
Execution <main>
BeginBehavior
    // Deinitialize the TLS library
    DeinitializeTLSLibrary;
EndBehavior

EndPattern
```

Listing 2: Functions Used in Secure Connection 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();
```

Listing 3: 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();
}};
```

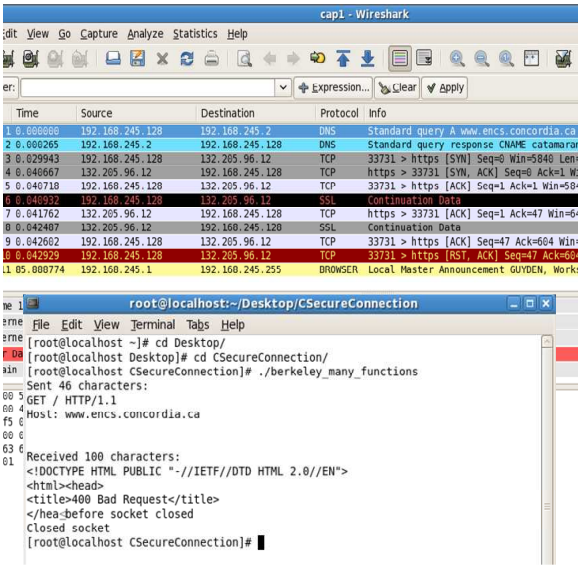

Figure 2: Capture of Connection

4.2 Experimental Results

In order to verify the hardening correctness, we have set first in the original application the server port number to 443, which means the client and the server can only communicate through HTTPS (ssl-mode). Any communication through http won't be understood and will fail. Then, we have compiled and run the client application and made it connect to the server (www.encs.concordia.ca) to retrieve information. The experimental results in Figure 2 show that the application failed to retrieve successfully the information. The server replies with a bad request because it is not able to understand the message content (Please see the run in the terminal). The highlighted lines in the Wireshark capture of the traffic show that the communication fails and stops after exchanging few undetermined messages.

Afterwards, we have weaved (using AspectC++ weaver) the elaborated aspect in Listing 3 with the different variants of the original application and compiled the resulting source code (using g++) to generate its corresponding executable.

Then, we have compiled the same original application using the extended *GCC*. Integrating the security concerns in the *Gimple* representation of code does not require refining the pattern into aspect. Compiling the selected client application by specifying the weaving option and selecting the file, which its contents are extracted manually (temporary solution) from the pattern for securing connection in Listing 1, is enough to perform the injection of the security concerns and generate the executable of the secured ap-

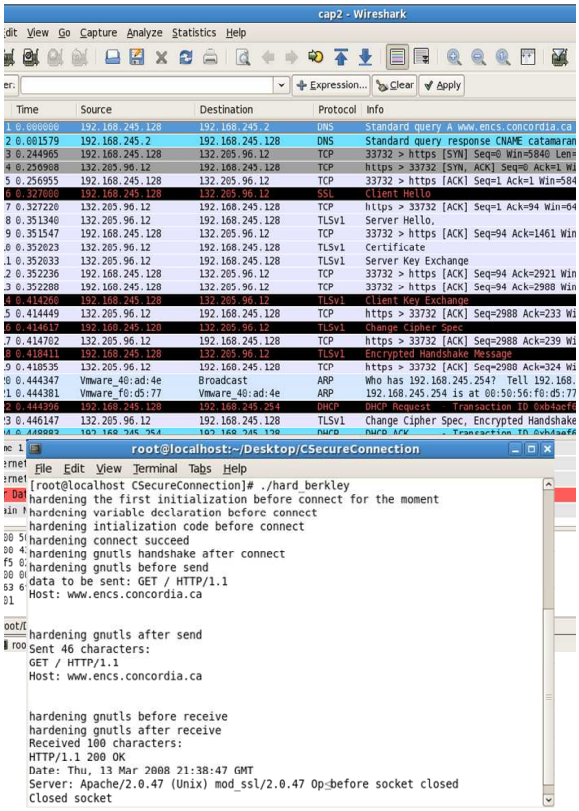

Figure 3: Capture of Hardened Connection

plication.

Running the two generated executables gave exactly the same results on the terminals and in the Wireshark packet captures. Due to this and to avoid duplication, we present in Figure 3 only the run of the application hardened by the *Gimple* weaving capabilities. The experimental results (Please see the run in the terminal and the highlighted lines in the Wireshark capture) explore that the new secure application is able to connect using both HTTP and HTTPS connections and exchange successfully the data from the server in ssl-mode and encrypted form, exploring the correctness of the security integration process and the feasibility of our propositions.

5 Conclusion

We have introduced in this paper an AOP weaving approach for injecting security concerns in the *Gimple* representation of programs. This proposition constitutes the first attempt towards adopting the aspect-oriented concept on *Gimple* and exploiting this intermediate representation to allow advising an application written in a specific language with security code written in a different one. In this context, few AOP weaving capabilities for *Gimple* have been developed and integrated into the *GCC* compiler. These features allow to weave security concerns in the *Gimple* tree of a program during the *GCC* compilation procedures. We have also explored the feasibility and the relevance of our propositions into practical implementation and a case study for securing an application using the proposed weaving capabilities.

REFERENCES

Bodkin, R. (2004). Enterprise security aspects. In *Proceedings of the AOSD 04 Workshop on AOSD Technology for Application-level Security (AOSD'04:AOSDSEC)*.

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

Huang, M., Wang, C., and Zhang, L. (2004). Toward a reusable and generic security aspect library. In *Proceedings of the AOSD 04 Workshop on AOSD Technology for Application-level Security (AOSD'04:AOSDSEC)*.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). Overview of aspectj. In *Proceedings of the 15th European Conference ECOOP 2001, Budapest, Hungary*. Springer Verlag.

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

Mourad, A., Laverdière, M.-A., and Debbabi, M. (2007). A high-level aspect-oriented based language for software security hardening. In *Proceedings of the International Conference on Security and Cryptography*. Secrypt.

Mourad, A., Laverdière, M.-A., and Debbabi, M. (2008). A high-level aspect-oriented based framework for software security hardening. *Information Security Journal: A Global Perspective*, 17(2):56–74.

Shah, V. (2003). An aspect-oriented security assurance solution. Technical Report AFRL-IF-RS-TR-2003-254, Cigital Labs.

Spinczyk, O., Gal, A., and chroder Preikschat, W. (2002). Aspectc++: An aspect-oriented extension to c++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems, Sydney, Australia*.

Yang, Z. (2007). On building a dynamic vulnerability detection system. Master's thesis, Concordia University.