New Primitives to AOP Weaving Capabilities for Security Hardening Concerns

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 two new primitives to Aspect-Oriented Programming (AOP) languages that are needed for systematic hardening of security concerns. These primitives are called *exportParameter* and *importParameter* and are used to pass parameters between two pointcuts. They allow to analyze a program's call graph in order to determine how to change function signatures for the passing of parameters associated with a given security hardening. We find this feature necessary in order to implement security hardening solutions that are infeasible or impractical using the current AOP proposals. Moreover, we show the viability and correctness of our proposed primitives by elaborating their algorithms and presenting experimental results.

Key words: Security Hardening, Aspect-Oriented Programming, Call Flow Graph, Pointcut, Primitive, Weaving.

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 for 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. 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 harden security into software. In this context, AOP appears to be a promising paradigm for software security hardening. It is based on the idea that computer systems are better programmed by separately specifying the various concerns, and then relying on underlying infrastructure to compose them together. 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 [1–5]. However, AOP was not initially designed to address security issues, which resulted in many shortcomings in the current technologies [6, 7]. We were not able to apply some security hardening activities due to missing features. Such limitations forced us, when applying security hardening practices, to perform programming gymnastics, resulting in additional modules that must be integrated within the application, at a definitive runtime, memory and development cost.

As a result, the specification of new security related AOP primitives is becoming a very challenging and interesting domain of research. In this context, we propose in this paper AOP primitives that are needed for security hardening concerns, named *exportParameter* and *importParameter*. They allow to pass parameters from one pointcut to the other through the programs' context-insensitive call graph. We find this feature necessary because it is needed to perform many security hardening practices and none of the existing AOP features can provide this functionality.

This paper is organized as follows: We first cast a quick glance at security hardening and the problem that we address in Section 2. Afterwards, in Section 3, we define parameter passing and show how parameter passing can be specified by an extension of the existing AOP syntax for advices. Then, in Section 4, we present the methodology of implementing the proposed primitives, as well as experimental results. We move on to the related work in Section 5, and then conclude in Section 6.

2   Security Hardening

Software security hardening is *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*. Security hardening practices are usually applied manually by injecting security code into the software. This task requires from the security architects to have a deep knowledge of the code inner working of the software, which is not available all the time. In this context, we elaborated in [8] an approach based on aspect orientation to perform security hardening in a systematic way. The primary objective of this approach is to allow the security architects to perform security hardening of software by applying proven solutions so far 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. This is done by providing an abstraction over the actions required to improve the security of the program and adopting AOP to build our solutions. The result of our experimental results explored the usefulness of AOP to reach the objective of having systematic security hardening. During our work, we have developed security hardening solutions to secure connections in client-server applications, added access control

features to a program, encrypted memory contents for protection and corrected some low-level security vulnerabilities in C programs. On the other hand, we have also concluded the shortcomings of the available AOP technologies for security and the need to elaborate new pointcuts. In this context, we proposed in [9] new pointcuts needed for security hardening concerns.

2.1 Security Hardening Example: Securing a Connection

Securing channels between two communicating parties is the main security solution applied to avoid eavesdropping, tampering with the transmission or session hijacking. The Transport Layer Security (TLS) protocol is a widely used protocols for this task. We thus present in this Section a part of a case study, in which we implemented an AspectC++ aspect that secures a connection using TLS and weaved it with client/Server applications to secure their connections. To generalize our solution and make it applicable on wide 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 addressed also the case where the connection processes and the functions that send and receive the data are implemented in different components (i.e different classes, functions, etc.). In Listing 1.1, we see an excerpt of AspectC++ code allowing to harden a connection.

2.2 Need to Pass Parameters

Our study of the literature and our previous work [8, 9] showed that it is often necessary to pass state information from one part to another of the program in order to perform security hardening. For instance, in the example provided in Listing 1.1, we need to pass the `gnutls_session_t` data structure from the advice around `connect` to the advice around `send`, `receive` and/or `close` in order to properly harden the connection. The current AOP models do not allow to perform such operations.

2.3 Solution with the Current AOP Technology

In Listing 1.1, the reader will notice the appearance of `hardening_sockinfo_t` as well as some other related functions, which are underlined for the sake of convenience. These are the data structure and functions that we developed to distinguish between secure and insecure 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. 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. This

effort of sharing the parameter has both development and runtime overhead that could be avoided by the use of a primitive automating the transfer of concern-specific data within advices without increasing software complexity. Further, other experiments with another security feature (encrypting sensitive memory) showed that the use of hash table could not be easily generalized.

Listing 1.1. Excerpt of an AspectC++ Aspect Hardening Connections Using GnuTLS

```
aspect SecureConnection {
  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);

   //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(); }}
```

3   Parameter Passing for Security Hardening

In this section, we define the syntax and realization of the proposed primitives. Their syntax is derived from AspectC++, as an optional program transformation section in an advice declaration as follows:

```
parameter ::= <type> <identifier>
paramList ::= parameter [,paramList]
```

```
e ::= exportParameter(<paramList>)
i ::= importParameter(<paramList>)

advice <target-pointcut> : (before|after|around)
(<arguments>) [: e | i | e,i] {<advice-body>}
```

where *e* and *i* are respectively the new *exportParameter* and *importparameter*. The arguments of *exportParameter* are the parameters to pass, while the arguments of *importParameter* are the parameters to receive. The two primitives should always be combined and used together in order to provide the information needed for parameter passing from one joint point to another.

Our proposed approach for parameter passing operates on the context-insensitive call graph of a program [10], with each node representing a function and each arrow representing call site. It exports the parameter over all the possible paths going from the origin to the destination nodes. This is achieved by performing the following three steps: (1) Calculating the closest guaranteed ancestor (GAFlow) of the origin and destination join points, (2) passing the parameter from the origin to the aforementioned GAFLow and then (3) from this GAFlow to the destination. The AOP primitives that are responsible of passing the parameters are the *exportParameter* and *importparameter*. The *exportParameter* is used in the advice of the origin pointcut to make the parameters available, while the *importparameter* is used in the advice of the destination pointcut to import the needed parameters.

The GAFlow, as presented in [9], constitutes (1) the closet common parent node of all the selected nodes (2) and through which passes all the possible paths that reach them. In the worst case, the GAFlow will be the starting point in the program. By passing the parameter from the origin to GAFlow and then to the destination, we ensure that the parameter will be effectively passed through all the possible execution paths between the two join points. Otherwise, the parameter could not be passed or passed without initialization, which would create software errors and affect the correctness of the solution. Figure 1 illustrates our approach. For instance, to pass the parameter from *h* to *g*, their GAFlow, which is *b* in the this case, is first identified. Afterwards, the parameter is passed over all the paths from *h* to *b*, then from *b* to *g* again over all the paths.

The parameter passing capability that we propose changes the function signatures and the relevant call sites in order to propagate useful security hardening variables via inout function parameters. It changes the signatures of all the functions involved in the call graph between the exporting and importing join points. All calls to these functions are modified to pass the parameter as is, in the case of the functions involved in this transmission path (e.g. nodes *b*, *c*,*d*,*e*,*f*).

3.1 Securing a Connection using Parameter Passing

We modified the example of Listing 1.1 by using our proposed approach for parameter passing. Listing 1.2 present excerpt of the new code. All the data structure and algorithms (underlined in Listing 1.1) are removed and replaced by the primitives for exporting and importing. An *exportParameter* for the parameters *session* and *xcred* is
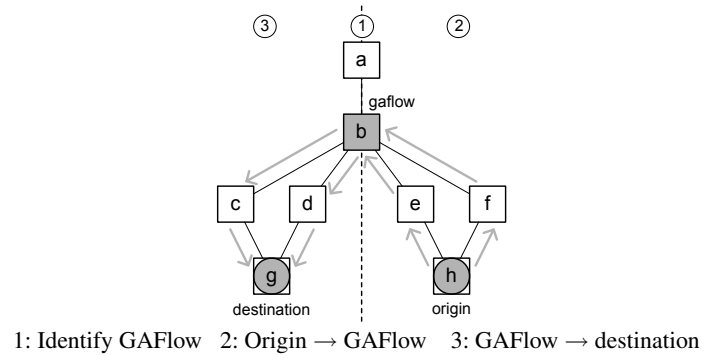
1: Identify GAFlow   2: Origin → GAFlow   3: GAFlow → destination

Fig. 1. Parameter Passing in a Call Graph

added on the declaration of the advice of the pointcut that identifies the function *connect*. Moreover, an *importParameter* for the parameter *session* is added on the declaration of the advice of the pointcut that identifies the function *send*.

Listing 1.2. Hardening of Connections using Parameter Passing GnuTLS

```
aspect SecureConnection {
 advice call("%_connect(...)") : around () : exportParameter(gnutls_session
     session, gnutls_certificate_credentials xcred){
   //variables declared
   static const int cert_type_priority[3] = { GNUTLS_CRT_X509, GNUTLS_CRT_OPENPGP
      , 0};

   //initialize TLS session info
   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);

   //Connect
   tjp->proceed();
   if(*tjp->result()<0) {perror("cannot_connect_"); exit(1);}

   //TLS handshake
   gnutls_transport_set_ptr (session, (gnutls_transport_ptr) (*(int *)tjp->arg(0)
      ));
   *tjp->result() = gnutls_handshake (session);
}
//replacing send() by gnutls_record_send() on a secured socket
advice call("%_send(...)") : around () : importParameter(gnutls_session session)
    {

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

4 Implementation and Experimental Results

In this Section, we present the implementation methodology of the proposed primitives together with experimental results exploring their correctness.

The elaborated algorithms for the implementation of parameter passing operate on a program's call graph. The origin node is the poincut where the *exportParameter* is called, while the destination node is the pointcut where *importParameter* is called. After calculating the closest guaranteed ancestor of the two pointcuts specified by the two primitives, the elaborated algorithm is performed first in order to pass the parameter from the origin to the closest guaranteed ancestor, then executed another time to pass the parameter from the closest guaranteed ancestor to the destination. This algorithm, which is not presented in this paper due to space limitation, is a building block that allows to modify the function signatures and calls in a way that would keep the program's syntactical correctness and intent (i.e. would still compile and behave the same). It finds all the paths between an origin node and a destination node in a call graph. For each path, it propagates the parameter from the called function to the callee, starting from the end of the path. In order to be optimal, it modifies all the callers only one time and keeps track of the modified nodes.

We implemented a program that represents the scenario of the call graph illustrated in Figure 1. This program is essentially a client application that establishes a connection, sends a request and receive a response from the server. We applied the aspects presented in Listings 1.1 and 1.2 on this application in order to secure its communication channels. We successfully tested the hardened applications with SSL enabled web server and verified the correctness of the solution.

5 Related Work

The shortcomings of AOP for security concerns have been documented and some improvements have been suggested so far. In the sequel, we present the most noteworthy.

Masuhara and Kawauchi [6] defined a dataflow pointcut (`dflow`) for security purposes that can be used to identify join points based on the origin of values.

In [7], Harbulot and Gurd proposed a model of a loop pointcut that explores the need to a loop joint point that predicts whether a code will ever halt or run for ever (i.e. infinite loops).

Another approach, that discusses local variables set and get poincut, has been proposed by Myers [11]. He introduced a pointcut allowing to track the values of local variables inside a method, which could be used to protect the confidentiality of local variables.

In [12], Bonr discussed a poincut that 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. He also explored in his paper the usefulness of capturing synchronized block in calculating the time acquired by a lock and thread management.

A `pcflow` pointcut was introduced by Kiczales in a keynote address [13], but was neither defined nor integrated in any AOP language. Such pointcut may allow to select points within the control flow of a join point starting from the root of the execution to the parameter join point.

6 Conclusion

AOP appears to be a very promising paradigm for software security hardening. However, this technology was not initially designed to address security issues and many research work showed its limitations in such domain. Similarly, we explored in this paper the shortcomings of the AOP in applying many security hardening practices and the need to extend this technology with new pointcuts and primitives. In this context, we proposed two new primitives to AOP weaving capabilities for security hardening concerns: *exportParameter* and *importParameter*. They pass parameters from one advice to the other through the programs' context-insensitive call graph. We first showed the limitations of the current AOP languages for many security issues. Then, we presented a motivating example that explores the need for parameter passing. Afterwards, we defined the new primitives and presented their implementation methodology together with the experimental results.

References

1. Bodkin, R.: Enterprise security aspects (2004) `http://citeseer.ist.psu.edu/702193.html` (accessed 2007/04/19).
2. DeWin, B.: Engineering application level security through aspect oriented software development (2004) `http://www.cs.kuleuven.ac.be/cwis/research/distrinet/resources/publications/41140.pdf`.
3. Huang, M., Wang, C., Zhang, L.: Toward a reusable and generic security aspect library. In: AOSD:AOSDSEC 04: AOSD Technology for Application level Security, March. (2004)
4. Cigital Labs: An aspect-oriented security assurance solution. Technical Report AFRL-IF-RS-TR-2003-254 (2003)
5. Slowikowski, P., Zielinski, K.: Comparison study of aspect-oriented and container managed security (2003)
6. Masuhara, H., Kawauchi, K.: Dataflow pointcut in aspect-oriented programming. In: APLAS. (2003) 105–121
7. harbulot, B., Gurd, J.: A join point for loops in AspectJ. In: Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005), March. (2005)
8. Mourad, A., Laverdière, M.A., Debbabi, M.: Towards an aspect oriented approach for the security hardening of code. (To appear in the Proceedings of the 21st IEEE International Conference AINA, AINA-SSNDS 2007, IEEE)
9. Laverdière, M.A., Mourad, A., Soeanu, A., Debbabi, M.: Control flow based pointcuts for security hardening concerns. (To appear in the Proceedings of the IFIPTM 2007 Conference, Springer)
10. Grove, D., Chambers, C.: A framework for call graph construction algorithms. ACM Trans. Program. Lang. Syst. **23** (2001) 685–746
11. Myers, A.: Jflow: Practical mostly-static information flow control. In: Symposium on Principles of Programming Languages. (1999) 228–241
12. Bonr, J.: Semantics for a synchronized block join point (2005) `http://jonasboner.com/2005/07/18/semantics-for-a-synchronized-block-joint-point/` (accessed 2007/04/19.
13. Kiczales, G.: The fun has just begun, keynote talk at AOSD 2003 (2003) `http://www.cs.ubc.ca/~gregor/papers/kiczales-aosd-2003.ppt` (accessed 2007/04/19).