Dynamic Aspect-Oriented Security Policy Enforcement

Mark S. Nichols
nichols_ms@hotmail.com
SE 547 Foundations of Computer Security

Abstract

There are many approaches to utilizing aspect-oriented (AO) languages and techniques for the purpose of introducing security into applications. The cross-cutting nature of security has been well documented and therefore the benefit of implementing security using AO techniques logically follows. Authentication, authorization, auditing, logging, etc. are obvious activities that can easily be introduced via aspects. This paper will propose the possibility of utilizing aspect-oriented components combined with Execution Monitoring [1] techniques to dynamically generate and enforce security policies within applications.

Introduction

Secure application development is an ever present problem with numerous possible solutions. Embedding validation and security code directly into the application to monitor progress is an obvious and proven solution. The problem with this approach is that it introduces so-called "nonfunctional code" (code not adding to the overall purpose of the application) intertwined within the business rules. Also, the structural differences between application logic and security logic invariably causes design and development issues[2]. The alternative proposed by this paper is to utilize aspect-oriented extensions to weave the security features into existing business logic. As part of the overall solution is a dynamic policy generator that allows for the easy addition and modification of policies and an Execution Monitor (EM) [1] to create a flexible and effective security policy enforcement mechanism.

Developing a secure application requires analysis of all possible situations or instructions that may generate an exploitable security lapse. It isn't always feasible to predict all possibilities and create a secure environment that withstands the test of time. Inevitably, additional restrictions or policies must be introduced to the application to reinforce the embedded security. The introduction of new restrictions or policies can be problematic. The difficulties lie in locating the appropriate instructions and inserting the new commands that will effectively detect security violations and stop the execution.

This paper will discuss a possible approach toward introducing policy enforcement through a combination of a policy engine written in a standard programming language along with the weaving of security checks into an application through the use of an aspect-oriented language.

Java was chosen for the development of the application code and the security engine. AspectJ was chosen as the aspect-oriented language. This is not to say that the techniques herein could not be done with other languages. On the contrary, the techniques described below could be added to other languages and environments with similar results. The point of this research is to show what can be done and the general effectiveness of combining aspect-oriented techniques with standard application development to provide application-level security.

Outline of this Paper

Section 1 discusses the approach taken for the research around implementing security policies utilizing aspect-oriented language extensions. Section 2 defines the requirements of Schneider's Execution Monitor [1] which is the type of evaluation engine implemented in this research. In section 3 an example application with its security requirements is described which will be the basis for the actual application developed. The actual project implementation is laid out in section 4 showing the Java class library structure and methods. Section 5 looks at the aspect side of the research including a high-level view of how aspects are used to facilitate application security. Detailed explanations of each of the required aspect files are shown in section 6. A sample run of the code described in this paper is shown in section 7 which shows the policies intercepting a violation. The project assumptions listed in section 8 define the requirements and limitations of the approach proposed by this paper. Finally, section 9 summarizes the research and results obtained.

1. Approach

The objective of this paper is to show the possibility of utilizing a combination of techniques to build security policy enforcement into an arbitrary application without embedding the security components directly into the application code. There are several major software components included in the project: the application (business rules), a finite state automaton providing a portion of the requirements for an "Execution Monitor" [1] and lastly, aspect-oriented components for building the security automaton and an intercepting method proxy for predicating method calls to verify the safety of the calls.

Java is used as the base programming language for the application and security automaton code. AspectJ is used as the extension to Java for the aspect-oriented components including the building of automata representing security policies, initiating the intersection of multiple policy automata and providing the ability to intercept application execution information required for security validation. In the case of example application, method calls (captured through reflection) were used as input for the security validation checks performed by the execution monitor. This is not a requirement or limitation of the approach. Rather, it is just the method chosen for the example. Other forms of input could be equally valid and used in other implementations.

For demonstration purposes, an example implementation will be shown that performs a series of tasks including specialized method calls. These method calls, individually, do not constitute security violations. Rather, it is certain combinations of the calls that have been defined as security risks. The combinations include performing the method calls in a particular order and we must stop the application if that method call order is detected.

Detection of the combinations is ineffective through static analysis since the order of method call may depend on dynamic data input, GUI interactions, etc. Therefore, a dynamic detection scheme is indicated. Utilizing a finite state machine to track the application state has been shown to be an effective tool for this type of requirement. This is the basis for the detection scheme employed by this paper.

2. Execution Monitor (EM):

Schneider described an Execution Monitor (EM) [1] to be a class of enforcement mechanisms that monitors execution steps of some system (target) and terminates the target's execution if it is about to violate the security policy being enforced. For purposes of clarification, it is important to note the definition of a security policy that will be used within this paper. Schneider initially identifies a broad definition for a security policy stating that a target S satisfies security policy P if and only if $P(\Sigma_s)$ equal true. He further defines a security property to be a set of executions where each element is evaluated individually to determine inclusion in the set and not by other set members. He then determined that a policy must be a property to be enforceable via EM. Therefore, as this paper describes enforcement of security policies, it can be assumed that the term policy refers to a security policy that is also a security property.

Now that a policy that can be enforced by an execution monitor has been defined, it is necessary to reiterate the traits and requirements of an execution monitor also defined by Schneider.

Requirements of Execution Monitor enforceability:
1) A security policy P that can be enforced by EM must be specified by a predicate in the form:

$$\mathrm{P}(\Pi) : \left(\forall \sigma \in \Pi : \hat{\mathrm{P}}(\sigma)\right)$$

2) Prefix Closed: $\left(\forall \tau' \in \psi- : \neg\hat{\mathrm{P}}(\tau') \Rightarrow \left(\forall \sigma \in \psi : \neg\hat{\mathrm{P}}(\tau'\sigma)\right)\right)$ In other words, once a security property is violated, the execution cannot be made "right" through any additional set of execution steps.

3) Rejection over a finite period: $\left(\forall \sigma \in \psi : \neg P'(\sigma) \Rightarrow \left(\exists i : \neg P'(\sigma[..i])\right)\right)$ For all executions that fail, they will fail in a finite number of instructions.

The execution monitor described in this paper does satisfy all three requirements. First, the finite state machine determines, through a predicate verification, if the next proposed instruction is safe or invalid. Second, once a violation is determined, a violation exception is thrown and execution stops – no additional instructions can make the violation "right" again. Third, a violation is determined prior to the execution of the invalid instruction (finite sequence of instructions).

3. Example Security Development:

We start by identifying the overall application requirements (i.e., business rules) along with the security requirements. It is noted that during the course of operation, the application needs to be able to read from and write data to secure data locations. To facilitate these requirements, a specialized functional package and class will be used that provides the needed data reading and writing capabilities along with the necessary authentication methods. It is this package that must be tracked and validated whenever calls are made to the methods contained in it. Again, it is particular sequences to these methods not necessarily the individual calls themselves that are of interest and may cause security violations.

Identified security risks / policies to implement:

1. The application may write data using the specialized methods but once a read method is called, no more writes can be allowed.
2. There is a very high risk data area such that; if a single read command has been issued, no more reads from that area can be allowed.

Looking at the first security requirement, the policy requires that the application in its initial state (State 0) may perform any operation until a read instruction (of any kind) occurs. Once a read operation takes place, the application shifts to state 1 and is limited to any operation except a write of any kind. This policy disallows the possibility of reading sensitive data and then writing it to a separate un-secure location. The structure of this policy is demonstrated using a finite state automaton illustrated in Figure 1.

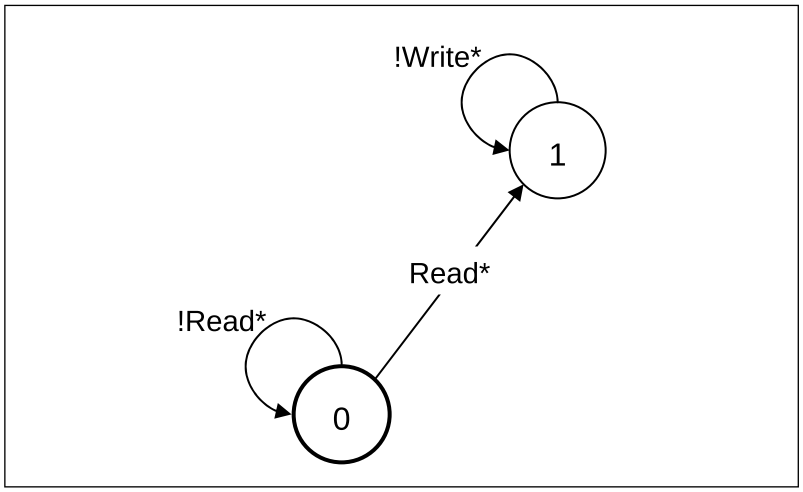

Figure 1 – "no write after read" policy automaton.

Notice the ornamentation on each of the transition labels. The labels may take any combination in the following form:

[!]methodname[*]

Since this implementation is concerned with particular method calls, "methodname" is the name of the method of interest. It can take on all of the same characteristics as a legal Java method name. Note: unlike java, all of the string comparisons performed while looking for matches are case insensitive. (e.g., Read will match Read, read, etc.) The optional prefix "!" indicates that a match will take place with any command that does not match the method name. (e.g., !Read will match Readfile, write, print, process, etc.) The suffix "*" is a wildcard that indicates a match occurs on any method name that begins with the characters listed. (e.g., Read* will match Read, read, readfile, readSecure, etc.) Combinations of the prefix and suffix can also be indicated. (e.g., !Read* will match write, print, process but will not match read or readfile).

The prefix and suffix combinations allow the policy designer greater flexibility in the design of the transitions between states in the policy automaton. Groups of method calls can be targeted in a policy rather than creating complex policies that identify each method call individually.

The prefix/suffix capability introduces an interesting issue when there are multiple transition paths within the current state. Note that this is a very likely situation. Inadvertent matches can occur if the match evaluation is done "out of order". For example, suppose the current state contains two possible transitions:

1. !write
2. read

Then, a "read" instruction is intercepted by the execution monitor and evaluated; the first transition test with "!write" will result in a match. This is in error since an exact match exists in the second transition. Therefore, a priority of evaluation must exist with possible transitions. There are four possible prefix/suffix combinations and they are listed in the appropriate evaluation priority:

1. Exact match (no prefix/no suffix)
2. "!" prefix and exact match (no suffix)
3. Wildcard "*" match with no prefix
4. "!" prefix and wildcard "*"
5. No match – Failure (Stop Application)

The priority of evaluation is enforced by appropriate positioning of the transition within the transition list during the process of creating a state. The higher the priority, the higher the transition is placed in the transition list. Using this method, the list can be evaluated in a top-down fashion and the first match will also be the most valid match.

Now that the first policy has been designed it is time to design the second policy. The second policy is designed to restrict the reading of data from a specialized secure location to a single read. Once data is read from this location, no more data may be retrieved from it in a single application execution. The same limitation of restricting writes of any kind must stay in effect. In fact, that limitation is even more important given the situation of reading from a location in memory deemed to be especially "sensitive".

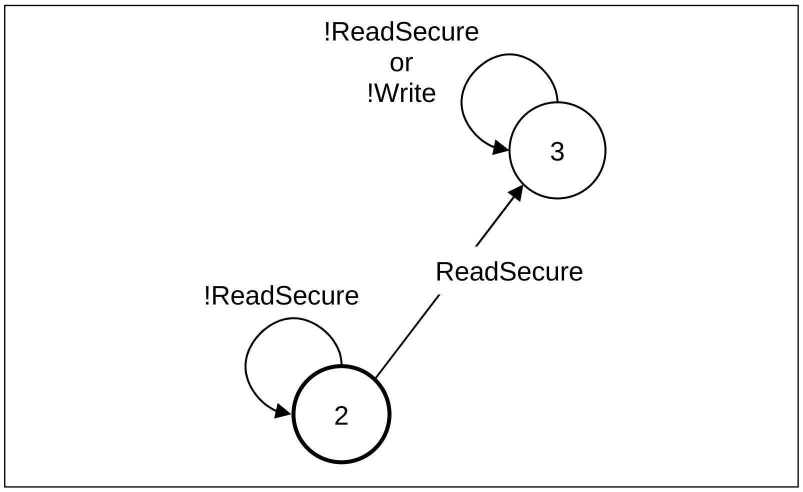

Figure 2 – "single ReadSecure allowed" policy automaton.

Figure 2 illustrates the automaton designed to enforce the policy of limiting the access of sensitive data to a single read. The second policy must to be "combined" with the first policy and introduced into the execution monitor to enforce the new restrictions. This could be done in any one of several different ways. The first way is by manually calculating a new automaton that describes the two policies. This method introduces the greatest chance of error and is the most brittle and inflexible to change. The second is to create a list of the policy automata and track the state of each automaton separately. During each evaluation, each automaton in the list would be evaluated from front to back. If any one of the automata fails then the application must be stopped. This method allows a greater degree of flexibility than the first option but creates a very difficult environment to manage and track. A third, more desirable, solution is to dynamically intersect all of the policy automata creating a single "master" automaton that can be easily tracked in the execution monitor. Another benefit of this approach is that the policies can be modified (added to/reduced) and the new updated automaton will be generated dynamically.

Figure 3 illustrates the resulting "master" automaton once the policies shown in Figure 1 and Figure 2 are intersected. This also shows the n * m nature of automaton intersection. There are 2 states in the first automaton (Figure 1) and there are 2 states in the second (Figure 2). The resulting intersected automaton contains 2 * 2 or 4 total states.

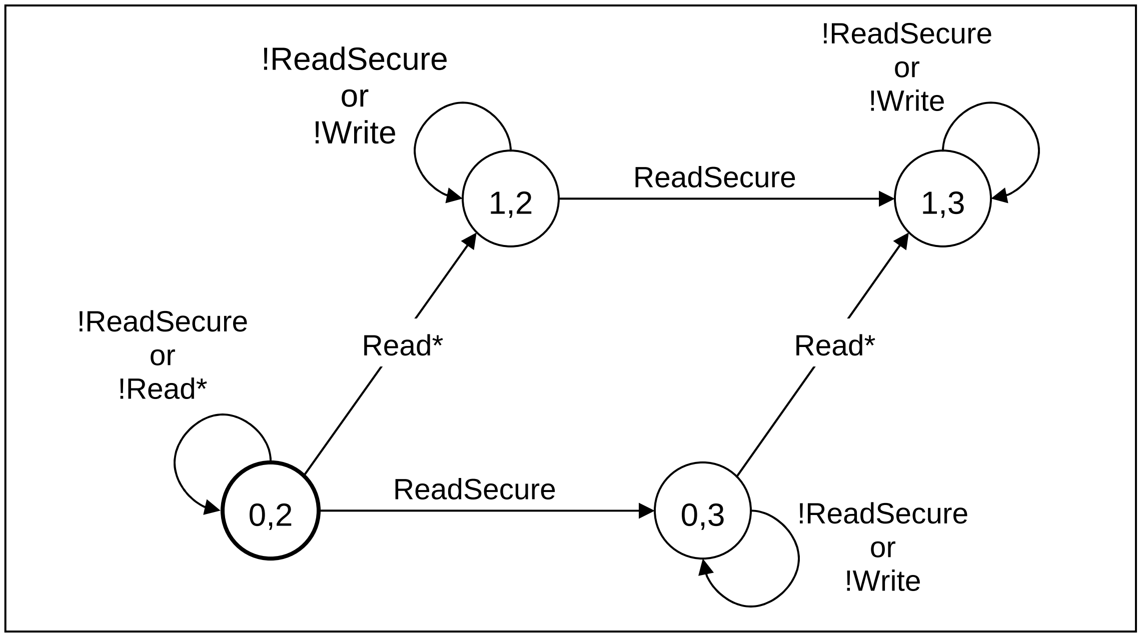

Figure 3 – Intersection of "no write after read" and "single ReadSecure allowed" policy automata.

At this time any additional policies could be added to the application in the same manner.

4. Actual Implementation:

The previous example describes how security policies are identified and combined. Next, the actual implementation created for this paper will be illustrated. The components created to for this research can be divided into two major parts. The first component is the policy builder and the second is the Execution Monitor. Both components are built using a combination of Java objects and aspects using AspectJ.

As was stated before, a finite state automaton is used as the tracking mechanism for the security policies. The structure of the automaton is discussed first.

Specifications of the Finite State Automata Java objects:

The finite state machine built for this paper utilizes the standard 5-tuple approach $(Q, \Sigma, q_0, \delta, A)$ where:

- Q is a set of states which are added individually to the automaton
- Σ is an alphabet which can be an alphanumeric label with an optional "!" and/or "*"
- q_0 is the start state which defaults to the first state added to the automaton but may be manually set to any existing state in the automaton
- $\delta : Q \times \Sigma \rightarrow Q$ is a transition function that is calculated by attempting to match the current instruction step label with the possible transitions in the current state
- $A \subseteq Q$ is the set of accepting states which, by definition, must be a state existing in the automaton

Figure 4 is the class diagram describing the design of the finite state automaton built to support this paper's execution monitor.

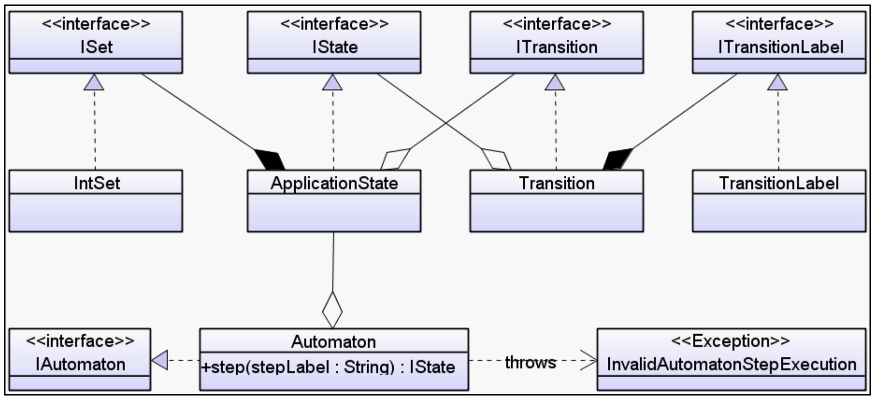

Figure 4 – Policy automaton class diagram.

This class diagram shows the structure of the Automaton and the method used for monitoring the execution of an application. The Automaton (IAutomaton) is given a name and is made up of 0..n states (IState). The name is not a functional component of the automaton. It exists solely for identifying the automaton. The name is helpful during debugging and logging purposes. The main method used by the execution monitor is "IState step(String stepLabel)". The step method is the predicate through which all current step validations are made. If the step is evaluated to be safe, a new state (IState) is returned which becomes the new current state. If the step is deemed invalid/illegal an exception "InvalidAutomatonStepException" is thrown and processing ceases.

A state is made up of a single name or label (ISet) and 0..n transitions (ITransition) to a resulting state. The state label is used to define the individual state. The internal structure of ISet is that of a mathematical set. That is, it provides the following set functionality:

- intersect
- union
- disjoint
- isMember

It was beneficial to treat the labels of the states as a set. As policies are built and combined or intersected, searching for appropriately matched states, creating new composite state labels, etc. were facilitated by the set functionality. For example, in creating the new intersected "master" automaton, the names of the states are the various iterations of set member combinations. Figure 1 automaton had the states {0, 1}. Figure 2 had the states {2, 3}. Figure 3 shows the four resulting unions of the state members: {{0, 2}, {1, 2}, {0, 3}, {1, 3}}

Finally, a transition (ITransition) is made up of a transition label (ITransitionLabel) and a reference to a resulting state (IState). The transition label contains the information necessary to validate the predicate 'step' (described above). It maintains a pair of flags that identify the existence of prefix "!" and suffix "*" along with the string used in matching the current execution step. The transitions are held in a list inside a state object. The positional order in that list dictates the priority of evaluation when a new step is evaluated.

The automaton objects provide the necessary functionality for creating states and automata and it also provides the necessary Execution Monitor "engine" for evaluating the predicate steps. It does not, however, provide a mechanism for injecting itself into an application. Hard coding calls directly into application code is an option but not very ideal. As described by the next section, utilizing aspect-oriented techniques does provide the capability that we require while keeping modularity at its highest.

5. High-level View of Security Aspects and Modular Policy Insertion

We now have in place the objects that will provide the EM functionality. The next step is to weave the EM functionality into an application.

There are 4 AspectJ/Java code files necessary for this implementation to operate.

- Policy aspects – One file per security policy
- PolicyBase.java – abstract aspect (required by the policy aspects)
- SecurityBuilderMonitor – Creates security policies / Proxy for intercepted methods
- SecurityBase.java – abstract aspect (required by SecurityBuilderMonitor)

The first step is to look at how a policy is built and "added" to the application security.

Figure 4 below demonstrates how a set of policies are created and added to the master policy automaton by simply including the policies (java source files) to the compilation. It also shows the interception of method calls in the application by the Execution Monitor contained in the security aspects.

1. A joinpoint is triggered by the execution of main() initiating the creation of the SecurityBuilderMonitor aspect.

2. Upon completing the creation of the SecurityBuilderMonitor aspect, the run() method (defined in the SecurityBase aspect) is called. A joinpoint in each of the security policy aspects is triggered by the run() method. This causes the "waterfall" creation of all of the policy aspects. Note: in the diagram the policies are shown as Policy1 through PolicyN but the order in which the policies are created does not matter.

3. A joinpoint in the SecurityBuilderMonitor aspect intercepts all of the interesting calls made by the application. The intercepted calls are passed to the execution monitor where the predicate is called to determine if the current method call is valid or illegal. If the method call is valid, control is passed to it and execution continues. If the call is illegal, an exception is thrown and execution is halted.

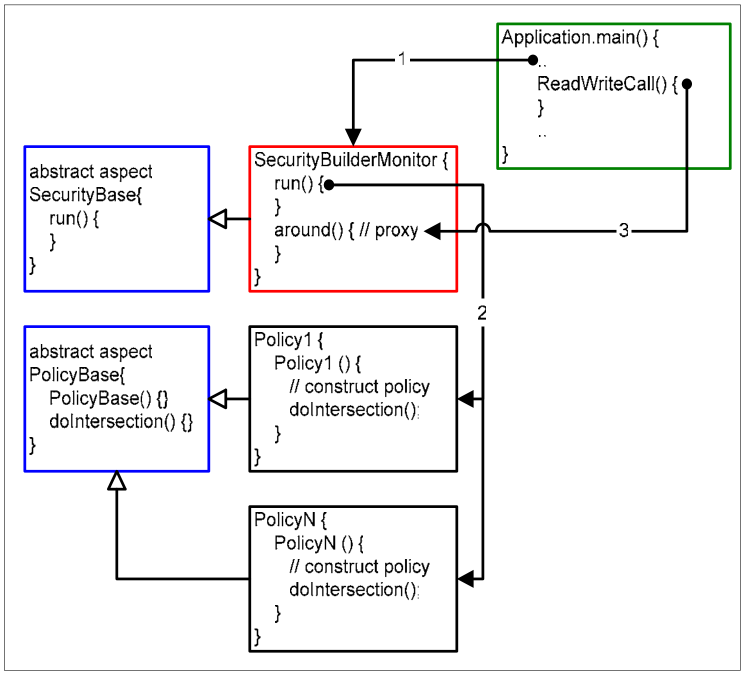

Figure 5 – High-level view of policy building and method interception.

6. Policy Aspects

Below is an example of the code for the “no write after read” policy aspect depicted in Figure 1. The code shows how an individual security policy automaton object is built within AspectJ code. The resulting object represents a single security policy which can then be intersected with other policies to create the final “master” state machine that tracks the progress of the application. It may be somewhat confusing since the only component coded in this aspect is the constructor. However, because the policy aspects inherit from PolicyBase, the constructor is all that is needed to effectively create a policy. The PolicyBase aspect will be reviewed later.

```
aspect PolicyReadWrite extends PolicyBase issingleton() {

   PolicyReadWrite() {    // aspect constructor

      // (Step 1) provide a name for the policy
      namePolicy("PolicyReadWrite");

      // (Step 2) Create 2 security policy states
      _factory.makeStates( 2 );

      // (Step 3) Create the transition objects
      //           (assignToState,     "trans label", transToState)
      _factory.makeTransition(0,       "!read*",           0);
      _factory.makeTransition(0,       "read*",            1);
      _factory.makeTransition(1,       "!write*",          1);

      // (Step 4) Call the intersection routine in PolicyBase (abstract
      // aspect) it creates the policy automaton and intersects
      // with the any existing automata
      doIntersection();
   }
}
```

Figure 6 – Individual Policy Aspect. This example aspect builds the "no read after write" policy.

In looking at the AspectJ code, the purpose of (Step 1) "namePolicy" is just to give the automaton a textual name. The name is not used in the operation of the execution monitor. However, it is useful during testing and debugging. (Step 2) creates all of the states contained in the automaton in a single statement by providing (as a parameter) the total number of states for this automaton. Each state created is given a unique integer name starting with 0. Although this means multiple automata will contain states with identical names, the intersection process corrects this situation by internally renaming states to avoid the obvious conflicts. This extra internal processing allows the policy automata to be created individually without knowledge of any other policies. (Step 3) requires a separate call to create each individual transition. The parameters identify the state object to assign the new transition to, the transition label used to determine which transition to follow when a step is executed and the new current state if the transition is followed. Finally, (Step 4) send the newly created policy automaton to be intersected with the main "master" automaton. If this is the first policy, it is merely stored as the master. If a policy already exists, the newly created policy is intersected with the "master" policy creating a new "master". This completes the creation of the security policy.

One thing to reiterate at this point is this implementation operates under the assumption that the automata created for security policies must be deterministic. However, the method described above for creating an automaton allows for the possibility for creating non-deterministic automata. Care must be taken while designing and building the policies to ensure the proper determinism. No validation code currently exists in the class structure for detecting non-deterministic transitions.

PolicyBase

As was stated earlier, each of the policies must inherit from the PolicyBase aspect. PolicyBase is required for two reasons. The first reason is because of standard object-oriented methodology. All of the common code has been removed from the policy aspect leaving only code that will differ from policy to policy. The second reason is to create a reusable aspect that will ensure that the appropriate code is called during the initialization phase of the application startup.

```
abstract aspect PolicyBase {

   static IAutomaton _policyAutomaton;
   static AutomatonFactory _factory;
   static String _policyName;

   // PolicyBase Constructor
   public PolicyBase() {
      _factory = AutomatonFactory.getFactory();
      _policyAutomaton = null;
      _policyName = null;
   }

   // No work is performed in this jpinpoint.  It exists to make
   // the aspect instantiate
   before() : execution(* SecurityBase.run()) {}

   // This method is used by the subaspect to help it intersect
   // with the 'master' policy automaton.
   static void doIntersection() {

      _policyAutomaton = _factory.makeAutomaton(_policyName);

      SecurityBuilderMonitor.intersect(_policyAutomaton);
   }

   // This method gives the policy a name that can be used for
   // display or debugging purposes
   static void namePolicy(String policyName) {
      _policyName = policyName;
   }
}
```

Figure 7 – The PolicyBase abstract aspect.

There is a single joinpoint in PolicyBase that triggers off of the execution of the run() method that is defined in the SecurityBase abstract aspect. Note: SecurityBase is discussed later. No work is performed in this joinpoint. It exists solely for the purpose of causing all aspects that inherit from PolicyBase to instantiate when the SecurityBase.run() method is called. This is interesting because it allows the individual policy aspects to be coded, instantiated and bound to the application without the application having any direct or indirect knowledge of the policies.

There are two methods in the PolicyBase aspect that are inherited and used by the policy aspects. namePolicy allows the policy aspect to give the policy a textual name. The name is not functionally required but is helpful during testing and debugging for display purposes. The doIntersection method is where the most work occurs during the creation of a policy. This is the point where the builder class takes all of the information provided to it by a policy aspect and actually builds a policy automaton. The resulting automaton is then passed to the SecurityBuilderMonitor aspect through the intersect method. If this is the first automaton to be passed in then its reference is merely copied and the automaton becomes the “master”. If a master automaton already exists, then the newly created policy is intersected with the existing master and a new master is created.

SecurityBuilderMonitor

The SecurityBuilderMonitor aspect contains the code necessary to trigger creation of all of the individual policies. It also has the “around” joinpoint that acts as a proxy for intercepting all of the method calls that the application has identified as potential security risks.

The aspect begins by creating a joinpoint that looks for the application’s main() method. This will cause the aspect to instantiate through the constructor. The constructor then calls the run() method (defined in the SecurityBase abstract aspect). The run() method does not perform any work, it exists only to give the policy aspects a target for their joinpoints.

```
public aspect SecurityBuilderMonitor extends SecurityBase issingleton() {

    SecurityBuilderMonitor() {
        run();  // This is the run() method intercepted by PolicyBase
    }

    before() : execution( public static void *.main(String[])) { }

    // The pointcut that identifies the methods to intercept
    pointcut interceptedIOCalls() : execution(* IORoutines.*(..));

    // This is the Proxy that intercepts the method calls
    Object around() throws Exception : interceptedIOCalls() {

        Object returnValue = null;

        // get the method name through aspect reflection
        String methodName = thisJoinPoint.getSignature().getName();

        if (_policyAutomaton == null) {
            returnValue =  proceed();
        } else {
            try {
                // Here, the predicate is called to verify the method call
                _policyAutomaton.step(methodName);

                System.out.println("method " + methodName + " permitted to execute");

                // The method is validated, allow it to continue
                returnValue =  proceed();

            } catch (securePolicyManager.InvalidAutomatonStepException e) {
                // Here, the method is deemed illegal an exception is thrown
                System.out.println("\t>>> " + methodName +
                    " was rejected by the security monitor" +
                    "\n\t\t and was not allowed to execute");

                throw new securePolicyManager.InvalidAutomatonStepException();
            }
        }
        return returnValue;
    }
}
```

Figure 8 - SecurityBuilderMonitor

SecurityBase

The last piece of code to look at is the abstract aspect SecurityBase shown below in Figure 8. SecurityBase avails several key features to SecurityBuilderMonitor. The “master” automaton is declared and held here in the _policyAutomaton variable. The intersect method is provided to handle the combining of security automata. Finally, the run() command is declared. The purpose of the run() method is documented above.

```
public abstract class SecurityBase {

    protected static IAutomaton _policyAutomaton;
    protected static boolean securityWarningShown = false;

    SecurityBase() {
        _policyAutomaton = null;
    }

    // Process policy automata intersection
    static void intersect(IAutomaton newPolicy) {
        if (_policyAutomaton == null)
            _policyAutomaton = newPolicy;
        else {
            _policyAutomaton = _policyAutomaton.intersect(newPolicy);
        }
    }

    // run() is the Trigger for policies to construct policies and intersect
    protected void run() {};
}
```

Figure 9 – SecurityBase abstract aspect.

7. Sample Run

The code printout below was the routine used to test the execution monitor and the policies. Although the routine does not perform any real work, it does work well to validate the fact that the policy automaton properly catches violations during execution. Note: “routines” is a variable referencing the class that the security policies are validating.

```
    public void testRun() {
        System.out.println("\n===========> Test Run <==============");
        try {
            routines.setUserID("mark");
            routines.write();
            routines.writeFile();
            routines.readSecure();
            routines.getFileWrites();
            routines.setUserID("sam");
            routines.readSecure();
            routines.writeFile();
        } catch (Exception e) {
            System.out.println("\nSecurity system detected a violation:\n\t" +
                        e.getMessage());
            System.out.println("Application Exiting");
            System.exit(1);
        }
    }
```

Figure 10 – Test code

As can be seen below in figure 11, the methods are called, then captured by the execution monitor "proxy" and validated by the master policy automaton. This run occurred while the "no write after read" and the "single ReadSecure allowed" policies were in effect. Here, two readSecure method calls were attempted and the second call caused the exception and the halting of the application.

```
============> Test Run <==============

Method Called: setUserID
method setUserID permitted to execute

Method Called: write
method write permitted to execute

Method Called: writeFile
method writeFile permitted to execute

Method Called: readSecure
method readSecure permitted to execute

Method Called: getFileWrites
method getFileWrites permitted to execute

Method Called: setUserID
method setUserID permitted to execute

Method Called: readSecure
       >>> readSecure was rejected by the security monitor
               and was not allowed to execute

Security system detected a violation:
       Security violation while attempting to execute method: readSecure
Application Exiting
```

Figure 11 – Test Run

8. Project Assumptions

As is with all projects and research there are certain assumptions that must be identified and in place. These assumptions were necessary for the reasons listed.

- Trusted Computing Base – Since the "secure-ness" of the weaving process is outside the scope of this research, it is assumed that no additional security risk is incurred through the use of an aspect-oriented language (in this case AspectJ).
- The security policies required by the application are enforceable through an Execution Monitor. In the test application, method calls (by name) were predicated indicating validity or failure.
- The security policy automata need to be deterministic. No validation checks are performed to ensure that all state transitions are unique and deterministic.
- The security policy automata are valid. No validation checks are performed to ensure the states and transitions do, in fact, create a valid automaton.
- The security policy automata are viable. No conflict checking is performed during the creation and intersection of policy automata.

9. Summary

The proposed solution in this paper uses the unique capabilities of aspect-oriented language extensions to weave an effective, modular, non-intrusive security policy enforcement mechanism into an application. Using this technique, an application can be developed without the need to intertwine security related code directly into the functional body or business rules of the application – greatly increasing modularity. Security policies can be designed, developed and even tested individually.

The individual policies (deterministic finite state automata) can be added, modified and removed dynamically, that is, without directly referencing them in an application. Adding and removing a policy is just a matter of including a policy aspect in the build list during compilation. Also, policies are built dynamically during the startup of the application creating a single "master" automaton that can easily be tracked as the application executes.

The proposed solution described in this paper has been shown to be relatively easy to implement and flexible. The enhanced capabilities provided by aspects are a key component to that ease and flexibility. Although the Execution Monitor alone could be added to an application to facilitate security, the exciting benefits are achieved through exploiting the weaving and interception abilities provided by an aspect-oriented environment.

Additional Proposed Work

- The work done on this example does lack a few nice to have checks and balances. There is no validation checks implemented that would discover redundant or invalid transitions. Also, there can be cases where duplicate transitions may result during the intersection of automata. Although this does not affect the results of the predicate, space utilization and program efficiency may be positively effected.

- As for the secure-ness of aspect-oriented languages, it is unknown if the standard methods used in weaving functionality introduces opportunities for security exploitation. A study that identifies if or when the aspect weaving mechanism can be considered part of the trusted computing base would be necessary to reinforce the viability of utilizing an aspect-oriented language for security enforcement such as the solution proposed by this paper.

- It would also be interesting to transfer this approach over to languages other than Java and AspectJ to see how the different environments compare.

References

[1] Fred B Schneider. Enforceable security policies. ACM Transactions on Information and Systems Security, 3(1):30-50, February 2000.

[2] B. De Win, W. Joosen and F. Piessens. Developing secure applications through aspect-oriented programming. Aspect-Oriented Software Development, Addison Wesley, pages 633-650, 2005.