Joys and horrors of aspect-oriented programming

Bart De Win

bart.dewin@ascure.com

Secure Application Development Course, 2009

Outline

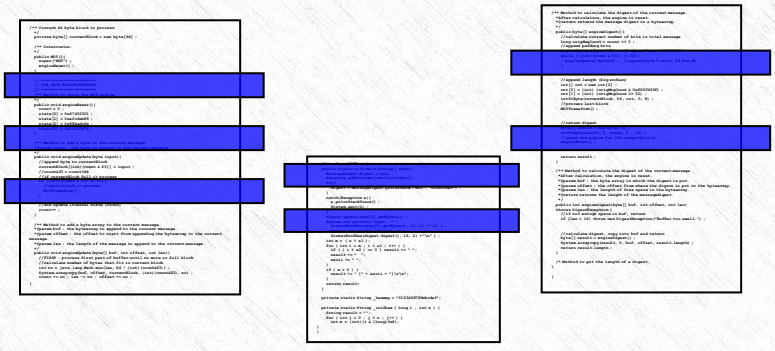
- Motivation for AOP and Security
- AOP in a nutshell
- AOP and Security in practice
- Security implications
- Conclusion

Causes for software security problems

- Software domain:
  - Ever-increasing complexity
  - Changing functionality and technology
  - Short release cycles
- Security domain:
  - Complexity of theories
  - Bug sensitivity of implementations
- Secure software domain:
  - Pervasiveness of security
    - Secure coding
    - Security mechanisms are crosscutting
  - Evolving environment
  - Trade-offs security-usability, security-performance, …
  - Quality COTS (for functionality and/or security)

Secure coding

- Security is crosscutting in location

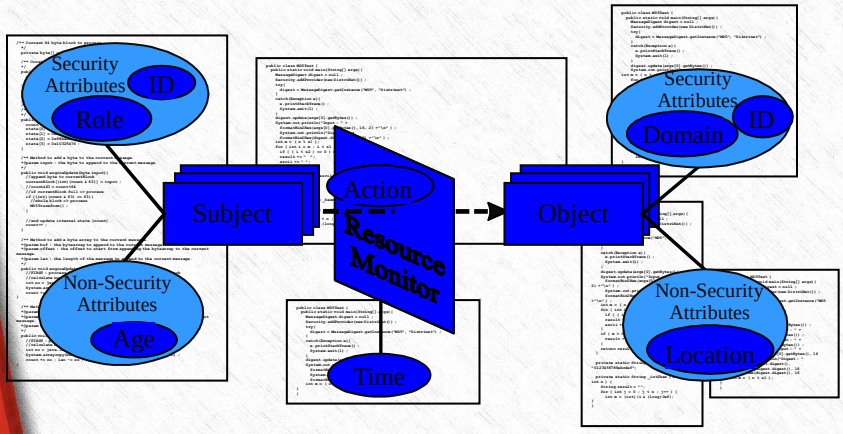


Secure coding (ctd.)

- Typical examples:
  - Buffer overflow
  - Input validation
- Often repetitive and, hence, developers tend to forget about it
- Coding guidelines, compiler or run-time support can be helpful
- No general-purpose solution exists:
  - Canonicalization errors
  - Race conditions

Security mechanisms are crosscutting

- Security is crosscutting in structure



Security mechanisms are crosscutting (ctd.)

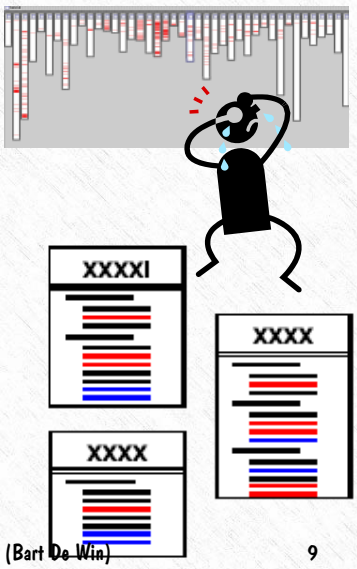
- Examples:
  - Access control
  - Confidentiality
  - Privacy
- Modular security engines are only a partial solution
  - Where to invoke ?
  - How to access parameters ?
  - Where to store security state ?
- Particularly problematic for fine-grained security requirements

Security: an evolving property

- Security of a system is often implemented once and for all
  - E.g., inspired by the Common Criteria
- Utopic, because of unanticipated changes
  - Incomplete threat analysis
  - New functional requirements
  - Design optimizations for NFR's
  - Changes in the system's environment

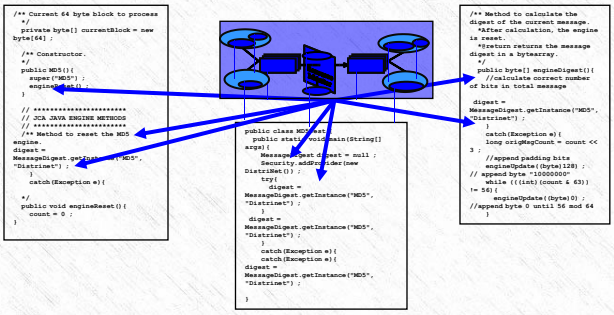
Resulting Problems

- **Scattering**
  - The specification of one property is **not encapsulated** in a single module
- **Tangling**
  - Each module contains descriptions of **several properties** or different functionalities



AOP to the rescue …

- To optimize the modularization of application-level security



Outline

- Motivation for AOP and Security
- AOP in a nutshell
- AOP and Security in practice
- Security implications
- Conclusion

AspectJ in a nutshell

- A general-purpose AO language
  - De facto standard for the core concepts of many AO tools
  - Static and dynamic language features
- An extension to Java
  - Outputs .class files compatible with any JVM
  - All Java programs are AspectJ programs
  - Supports source-code and byte-code weaving
- Commercial sponsors
  - Originally Xerox Parc, now maintained by IBM
- IDE support
  - Nice Eclipse plugin (AJDT)

Joinpoints and pointcuts

- A joinpoint is a point in the dynamic execution of the software
- Different types are supported:
  - Method & constructor call
  - Method & constructor execution
  - Field access (get / set)
  - Exception handler
  - Initialization
  - Advice execution
- A pointcut selects a set of joinpoints based on a number of constraints

```
public class MyPolicy extends Policy {
 private Permissions perms ;

 public MyPolicy(){
  super() ;
  perms = new Permissions() ;
  try{
    <read permissions from file>
    this.verifyPermissions() ;
  }
  catch(IOException e){System.err.println(e) ;}
 }

 private void verifyPermissions(){
  if (perms == null) return false ;
  …
 }
}
```


Advice

- Advice adds behavior to a (set of) joinpoint(s):
  - Similar to a method
  - Is executed before / after / around the joinpoint
  - For around advice: proceed() to resume the action at the specific joinpoint

```
before(): execution(void Foo.m(int)) {
  System.out.println("M is executed") ;
}
```

```
void around(): set(Foo.field) {
  System.out.println("Are you sure?") ;
  if(<confirmed>){
    proceed() ;
    System.out.println("Foo.field changed") ;
  }
}
```

Advice parameterization

- Just as regular methods, advice can be parameterized
  - Values come from the joinpoint context
  - All parameters must be matched within the pointcut
  - Use this(), target(), args()

```
before(int i): execution(void Foo.m(int)) && args(i) {
  System.out.println("M is executed with argument" + i) ;
}
```


Aspects

- Any combination of:
  - Members
  - Methods
  - Named Pointcuts
  - Advices

```
aspect MyAspect{
  int test;
  int double(int j){return 2*j ;}

  pointcut p(): call(* Foo.*(..)) ;

  before(): p(){
    System.out.println("Boo") ;
  }
}
```

Aspects (ctd.)

- Aspects can be declared 'privileged'
  - Have access to protected/private class members or methods
- Advices are ordered based on standard rules
  - Can be influenced by specifying ordering constraints explicitly

```
declare precedence: Security, Logging, * ;
```


Aspect instantiation

- Aspects are instantiated automatically
  - Cannot be created explicitly by the developer
- Aspects are associated to a particular 'context'
  - Normally, one aspect per JVM (issingleton())
  - Alternatives: perthis(), pertarget(), percflow(), pertypewithin()
  - Restricts the scope of advice application !
- Association operators
  - Requesting reference

```
MyAspect a = MyAspect.aspectOf(<instance>) ;
```

- Useful to manage concern-specific state

Executable systems ?

- Are aspects executable units, and what does composition mean in practice ?
- Often, AOP is an extension of OOP
  - AOP concepts are translated into OOP concepts (automatically / manually)
  - This mapping is difficult because of crosscuttingness
- Translation strategies:
  - Inlining (aka 'weaving')
  - Wrapping
  - Interception
  - Proxying
  - ...
- The developer is not (too much) bothered with this tedious task

Outline

- Motivation for AOP and Security
- AOP in a nutshell
- **AOP and Security in practice**
- Security implications
- Conclusion

Potential usage scenarios

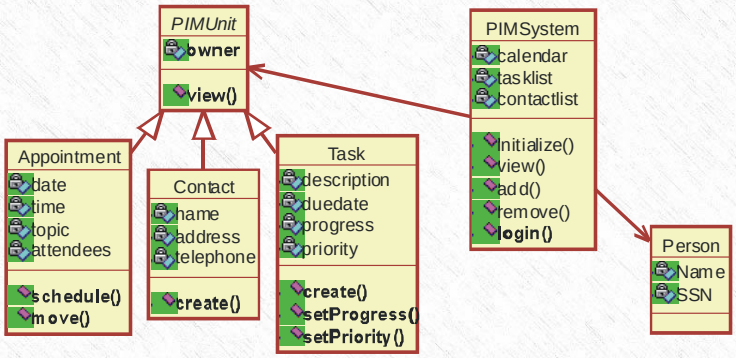
- Policy enforcement
  - Implementation (green field or add-on)
    - Also reverse (e.g., disabling license checks)
- Policy mining and monitoring
- Coding guidelines
  - Implementation
- Security testing
- Verification of correct use

Policy Enforcement

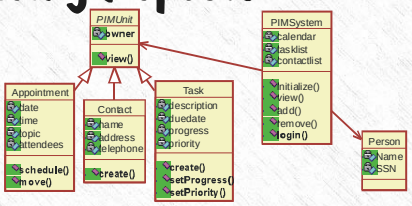
- Most interesting category
  - Applies the full potential of AOP
- All about finding ways to 'bind the security engine'
- Design activity => many alternative solutions
  - Consider typical SE properties
  - Non-functional qualities

Policy Enforcement – PIM

Policy:
- PIM Unit owners can invoke all operations
- Contacts only accessible to their owners
- All other accesses restricted to viewing



PIM security using AspectJ




```
aspect Authentication{
  private static String currentUser ;

  static String getUser(){
    if(currentUser == null){
      currentUser = <login>;
    }
    return currentUser ;
  }
}
```

```
aspect OwnerManagement
      perthis(this(PIMUnit)){
  String owner ;

  after(): execution(PIMUnit.new(..)){
      owner = Authentication.getUser() ;
  }
}
```

```
aspect Authorization{
  pointcut restrictedAccess():
    execution(* Appointment.move(..)) ||
    execution(* Contact.view(..)) ||
    execution(* Task.setPriority(..)) ||
    execution(* Task.setProgress(..));

  void around(PIMUnit p): restrictedAccess() && this(p){
    //are owners identical ?
    if(! OwnerManagement.aspectOf(p).owner.equals(
        Authentication.getUser()))
      throw new RuntimeException("Access denied !");
    else proceed() ;
  }
}
```

Policy Enforcement – PIM w/ JAAS

- With JAAS, Java offers:
  - a pluggable mechanism for authentication
  - an extensible mechanism for authorization based on the subject running the code
- JAAS can be integrated seamlessly using AOP

Using JAAS (ctd.)

```
aspect Authentication{
  private static Subject currentUser ; //one per session
  public static LoginContext lc = null ;

  static Subject getUser() {
    if(currentUser == null){
      try{
        lc = new LoginContext("PIM", new TextCallbackHandler()) ;
        lc.login();
        currentUser = lc.getSubject() ;
      }
      catch(Exception e){throw new RuntimeException(e) ;}
    }
    return currentUser ;
  }
}
```

Using JAAS (ctd.)

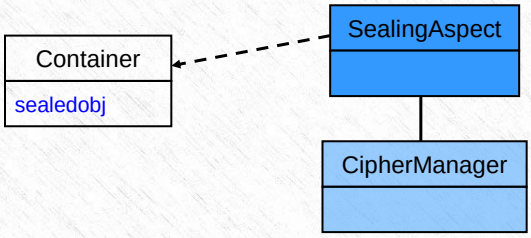
```
aspect Authorization{
 pointcut restrictedAccess(): execution(* Appointment.move(..)) || execution(* Contact.view(..)) ;

 //Activates a .doAsPrivileged with the currently executing subject
 void around(): restrictedAccess() && !cflowbelow(restrictedAccess()){
  try{
    Subject.doAsPrivileged(Authentication.getUser(), new PrivilegedAction(){
     public Object run() {
       proceed();
         //No result is required for these particular operations
         return null ;
     } },null);
  }
  catch(Exception e){e.printStackTrace() ;}
 }

 //Checks whether the correct OwnerPermission is owned
 before(PIMUnit u): restrictedAccess() && this(u){
  Subject owner = OwnerManagement.aspectOf(u).owner ;
  OwnerPermission op = new OwnerPermission(owner) ;
  AccessController.checkPermission(op) ;
 }
}
```


Policy Enforcement – Sealing sensitive objects in memory

- Java offers support to seal the internals of sensitive objects
  - javax.crypto.SealedObject
- Can be used to protect sensitive information in memory from low level intruders



Sealing sensitive objects in memory (ctd.)

```
privileged aspect SealingAspect{
  //ITD ; visibility is limited to the declaring aspect
  private SealedObject Container.sealedobj ;

  //Helper pointcut to filter advice executions
  pointcut SealingAdvice(): adviceexecution() && within(SealingAspect);

  //intercept construction to initialize sealed object
  before(Container c): execution(Container.new()) && this(c) &&
      !cflowbelow(SealingAdvice()){
    try{
      //Create new Container to be sealed within the original Container
      c.sealedobj = new SealedObject(c, CipherManager.getCipher());
    }
    catch(Exception e){System.err.println(e);}
  }

...
```


Sealing sensitive objects in memory (ctd.)

```
  //intercept GETTERS of fields
  Object around(Container c): get(* Container.*) && this(c) && !cflow(SealingAdvice()) {
    Object value = null ;
    try{
      Object unsealed = c.sealedobj.getObject(CipherManager.getCipher()) ;
      Class cl = ((Container)unsealed).getClass() ;
      value = cl.getField(thisJoinPoint.getSignature().getName()).get(unsealed) ;
    }
    catch(Exception e){System.err.println("GET "+e);}
    return value ;
  }

  //intercept SETTERS of fields
  void around(Container c, Object arg): set(* Container.*) && this(c) && args(arg) &&
      !cflow(SealingAdvice()) {
    try{
      Object unsealed = c.sealedobj.getObject(CipherManager.getCipher());
      Class cl = ((Container)unsealed).getClass();
      cl.getField(thisJoinPoint.getSignature().getName()).set(unsealed,arg) ;
      c.sealedobj = new SealedObject((Container)unsealed, CipherManager.getCipher());
    }
    catch(Exception e){System.err.println("SET "+e);}
  }
}
```

Policy mining and monitoring

- Goal: instrument the application in order to
  - deduce information about policy requirements
  - monitor the application to verify whether the current policy meets the risks of the execution environment
- Heavily dependent on the particular goals and application

Coding guidelines

- Typical usage is insertion of extra security tests
- Nature of tests:
  - Localized, scattered
  - Specific (often difficult to generalize)
- Example of input validation:

```
aspect InputValidation {
  pointcut inputcheck(): call (String InputStream+.read(char[])) ;

  after(char[] arr): inputcheck() && args(arr) {
    <validate arr>
  }
}
```

Discussion of AOP benefits

- Abstraction
  - Reasoning about one problem (or concern) at a time
  - Caveat: not all AOP tools offer modular reasoning !
- Verification
  - Improves inspection capabilities for the security binding
  - Avoids incomplete mediate errors
- Reuse
  - Part of the security binding can be made reusable
  - As a result, the security engine/library cannot be composed wrongly
- Evolution
  - More localized changes facilitates the maintenance of software
  - Caveat: AOP and the evolution paradox

Outline

- Motivation for AOP and Security
- AOP in a nutshell
- AOP and Security in practice
- **Security implications**
- Conclusion

Problem statement

- Software vulnerabilities are to a considerable degree due to the complexity of:
  - Software engineering (pervasiveness)
  - Security (algorithms, domain knowledge)
- Aspect-Oriented Programming (AOP) has shown to be helpful
  - From a software engineering perspective…
    - Increased modularization improves specialization, verification and manageability
  - But what about the security perspective?
    - Do we really end up with secure software?
    - Statements have been made about this, but little published work is available

A motivating example …

```
package mypackage;
public class SensitiveData{
 private String secret;

 public SensitiveData(String s){
  secret = s;
 }

 String getSecret(){
  return secret;
 }

 public static void main(String[] args) {
  SensitiveData sd = new SensitiveData(
   "My first secret");
  sd.setSecret("My second secret");
  System.out.println(sd.getSecret());
 }
}
```

```
package security;
aspect Authorization{

 private static Policy pol;

 pointcut accessrestriction():
  execution(String SensitiveData.getSecret());

 void around(): accessrestriction() {
  if(! pol.isAllowed(...))
   throw new RuntimeException("Denied !");
  else proceed();
 }
}
```

```
package unsecure;
privileged aspect SniffingAspect{
 after(SensitiveData sd):
  set(private String SensitiveData.secret) && this(sd){
   System.out.println("The secret is now: " + sd.secret);
  }
}
```

Language-level issues

- Invocation parameters can be modified
  - Imagine the following aspect …

```
aspect PolicyMod{
    pointcut polcheck(): execution(boolean Policy.isAllowed(..));

    //consult the policy, but always return true
    boolean around(): polcheck(){
        boolean res = proceed();
        return true;
    }
}
```

  - Parameters presented to a security engine could be modified as well
- Invocations can be redirected or even discarded entirely:
  - Use a less restrictive Policy object
  - DoS scenarios

Language-level issues (ctd.)

- Privileged aspects
  - Private internals of classes **and** aspects can be accessed by privileged aspects
    - Log changes of private variables or executions of private methods
    - Inspect and modify private, security-related attributes
    - Access cflow associations
    - Access inter type declarations
  - As a result, it becomes very hard to protect security-specific information
- Remark: only possible using weaving-based AOP tools
  - Allows one to "play" with Java's type safety rules (at least, from a developer's perspective)
  - Important to realize the impact on security verification (e.g., information flow)

Tool specific problems

- AspectJ 5 uses dangerous transformations:
  - When using privileged aspects to access private members, a public method with a 'predictable' name is introduced in the target class !

```
public class SensitiveData{

 //method generated to access the private secret datamember
 public static String ajc$privFieldGet$unsecure_SniffingAspect$mypackage_\\
  SensitiveData$secret(SensitiveData sensitivedata){
            return sensitivedata.secret;
 }

 <snip>
}
```


Tool specific problems (ctd.)

- Package restricted aspects are transformed into public classes
- Private inter-type declaration members are transformed into public members in the target class

- AspectJ compiler must control ALL the code in order to guarantee "secure" code
- Access modifiers are checked at compile time. What about run-time execution?

- Most probably, there will be other issues ...

Other risks

- Use of wildcards in PCD's
  - Based on syntax instead of semantics
  - Difficult to predict the effect in case of system evolution
- Aspect circumvention
  - Based on woven code prediction (possibly multi-pass)
  - Used to be possible in the past, but seems solved with newer compiler versions
- Load-time weaving
  - Seems like a small step from a softw. eng. perspective, but from a security point of view it is a different model!
  - The unpredictability increases:
    - What in case of *new* classes?
    - Can the set of aspects be changed at runtime?
  - The use of LTW should be restricted to systems that have correct compile-time weaving behavior

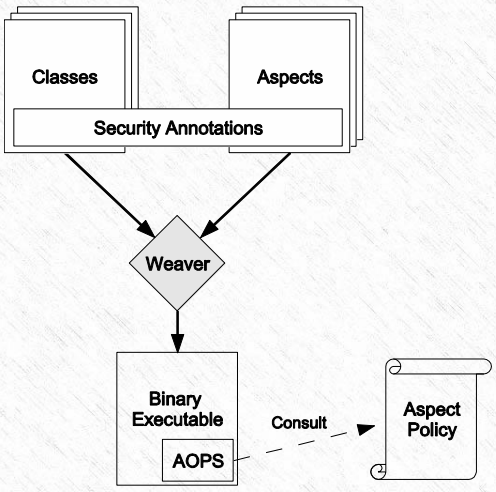
Risk synthesis

- Security risks are related to:
  - Modification of the logic of a module
  - Influencing the interaction or composition of modules
  - Enforcement of the aspect model
- This can occur intentionally or unintentionally
  - An ignorant developer could introduce security vulnerabilities without even knowing it
  - Addressing these is key

Towards a solution – academic SOTA

- AOP language extensions/restrictions [Dantas06, Aldrich05, Larochelle03]
  - Run-time enforcement is key
  - Further restrictions might be useful
- Security-oriented program transformations [Erlingsson03,Ligatti05]
  - In-line with the AOP philosophy
  - Focusing on restricting functionality (e.g., access control)
- An aspect permission system is a viable alternative solution
  - Logical extension of Java's permission system
    - Support checking aspects for particular permissions
  - Empower the developer to enforce policies relevant to his particular application
  - Enable control over aspect-specific dynamic actions, such as cflow or aspect activation
  - An effective way of implementing restrictions
    - More secure than a compiler-only language solution

General overview



Problems to address (1)

- In standard Java, checks are inserted to enforce a policy
  AccessController.checkPermission(..)
- For AOP, the transformations (and corresponding output) of a weaver happen under the hood
  - **Checks cannot always be inserted by a developer:**
    - JP matching (get/set) on a private member
    - An inter-type member declaration (aspect developer)

=> Let the weaver insert checks for dangerous actions

Problems to address (2)

- **At runtime, the identity of an aspect is not always known (for weaving based tools)**
- Different scenarios in which a (security-sensitive) interference can occur:
  1. A class is augmented with extra logic that interferes
  2. An aspect, translated into a proper class, initiates the interference
  3. An aspect affects a third class that interferes as such indirectly
- For case 2, available technology provides a solution
  - For limited cases: no aspect-in-aspect
- More difficult for the other cases
  - Granularity of permission associations in Java is not sufficiently fine-grained

Aspect-Oriented Permission System (AOPS)

- We have implemented AOPS based on the execution history-based access control model
  - Similar to, but more restricted than standard stack-based access control
  - Can be used to control risks, as well as to implement arbitrary policies
- State updates in case of:
  - Execution of advice
  - Invocation of aspect method
  - Direct access to aspect member
- AOPS was realized through a combination of:
  - Modifications to the weaver
  - AOPS run-time library

AOPS augmented code example

```
aspect Authorization {
  pointcut normalAccess1( ) = . . . ;

  before( ): normalAccess1 ( ) {
    PermissionManager mngr = PermissionManager.getPermissionManager( ) ;
    Permissions perms = new Permissions( ) ;
    String critical = RightsPermission.SECURITYCRITICAL ;
    perms.add(new RightsPermission(critical)) ;
    mngr.beginGrant("security.Authorization", perms) ;
    String user = Authentication.getUser() ;
    mngr.demand(new RightsPermission(critical)) ;
    mngr.endGrant() ;
    if (! OwnerManagement.aspectOf (
    thisJoinPoint.getThis()).owner.equals(user))
    throw new RuntimeException("Access Denied !") ;
  }
}
```

Temporarily increase rights

Check whether rights lost

Back to earlier rights set

Outline

- Motivation for AOP and Security
- AOP in a nutshell
- AOP and Security in practice
- Security implications
- Conclusion

Best practices for implementation

- Use specific pcd's (be careful with wildcards)
- Avoid the use of privileged aspects
- Use aspects that operate at interface level as much as possible (consider to refactor your application)
- Structure aspects in packages
- Specify aspect ordering, especially for security aspects
- Consider verifying coding guidelines to support this

Best practices for development

- Avoid using AOP for high-risk components
  - E.g., attack surface, security kernel, ...
- Avoid using different 'sets' of aspects
  - Pro-actively try to identify feature interactions
- Make sure that aspects are fully integrated in the development environment
  - No separate compilation steps

Conclusions

- The crosscuttingness of security is an important hurdle in the development of secure software
- AOP can optimize the modularization of application security
  - Improves reasoning and evolution properties
  - Different usage scenarios
- Be aware of the security implications => use wisely !
  - I would advise pro AOP for small, controllable, low/medium-risk projects
- Many issues in the area of AOSD & security are open research problems

References

- AOSD & AspectJ
  - The AspectJ programming guide, semantic appendix and quick reference (http://www.eclipse.org/aspectj/docs.php)
  - Ramnivas Laddad, "AspectJ in Action", Manning Publications, 2003.
  - Stefan Hanneman and Arno Schmidmeier, "AspectJ idioms for Aspect-Oriented Software Construction", 8th EuroPLOP, June 2003.
  - Gregor Kiczales and Mira Mezini, "Aspect-oriented programming and modular reasoning", 27th International Conference on Software Engineering, May 2005.
  - Tom Tourwe, Johan Brichau, and Kris Gybels, "On the existence of the AOSD-evolution paradox", AOSD Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT), 2003.
- AOSD & security
  - Bart De Win, Frank Piessens, Wouter Joosen, and Tine Verhanneman, "On the importance of the separation-of-concerns principles in secure software engineering", ACSA Workshop on the Application of Engineering Principles to System Security Design, 2003.
  - Bart De Win, Wouter Joosen, and Frank Piessens, "Developing Secure Applications through Aspect-Oriented Software Development", Aspect Oriented Software Development, Addison-Wesley, 2004, pp. 633-650.
  - Viren Shah and Frank Hill. Using Aspect-Oriented Programming for Addressing Security Concerns, International Symposium on Software Reliability Engineering (ISSRE'2002), 2002.
  - Ron Bodkin, "Enterprise Security Aspects", Workshop on AOSD Technology for Application-level Security, 2004.

References (ctd.)

- Research challenges
  - Daniel Dantas and David Walker, "Aspects, Information Hiding and Modularity", Conference on Programming Language Design and Implementation (PLDI), 2004.
  - Bart De Win, Frank Piessens, and Wouter Joosen, "How Secure is AOP and What can we Do about it?", Workshop on Software Engineering for Secure Systems (SESS), 2006.
  - David Larochelle, Karl Scheidt, and Kevin Sullivan, "Join Point Encapsulation", AOSD Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT), 2003.
  - Danien Dantas and David Walker, "Harmless Advice", Symposium on Principles of Programming Languages (POPL), 2006.
  - U. Erlingsson, The Inlined Reference Monitor Approach to Security Policy Enforcement. Ph.D. thesis, Technical Report 2003-1916, Department of Computer Science, Cornell University, Ithaca, NY, 2003.
  - Jay Ligatti, Lujo Bauer, and David Walker, Enforcing non-safety security policies with program monitors, In Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS), September 2005