Security implications of AOP for secure software

Bart De Win
DistriNet – KU Leuven
January 23, 2007

OWASP Belgium Chapter

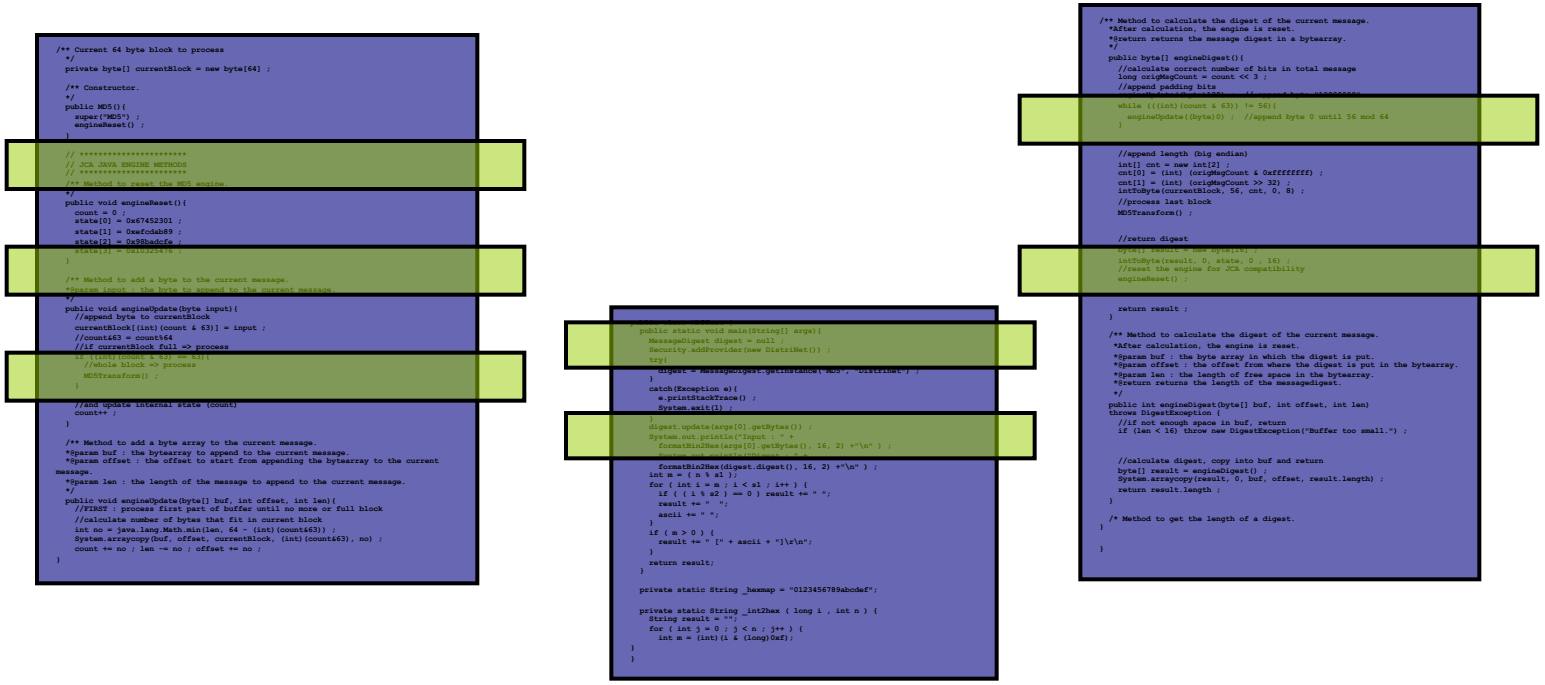
The OWASP Foundation
http://www.owasp.org/

Overview

- *Introduction: using AOP for security*
- Problem statement
- Overview of security risks
- Countering the risks
- Conclusion

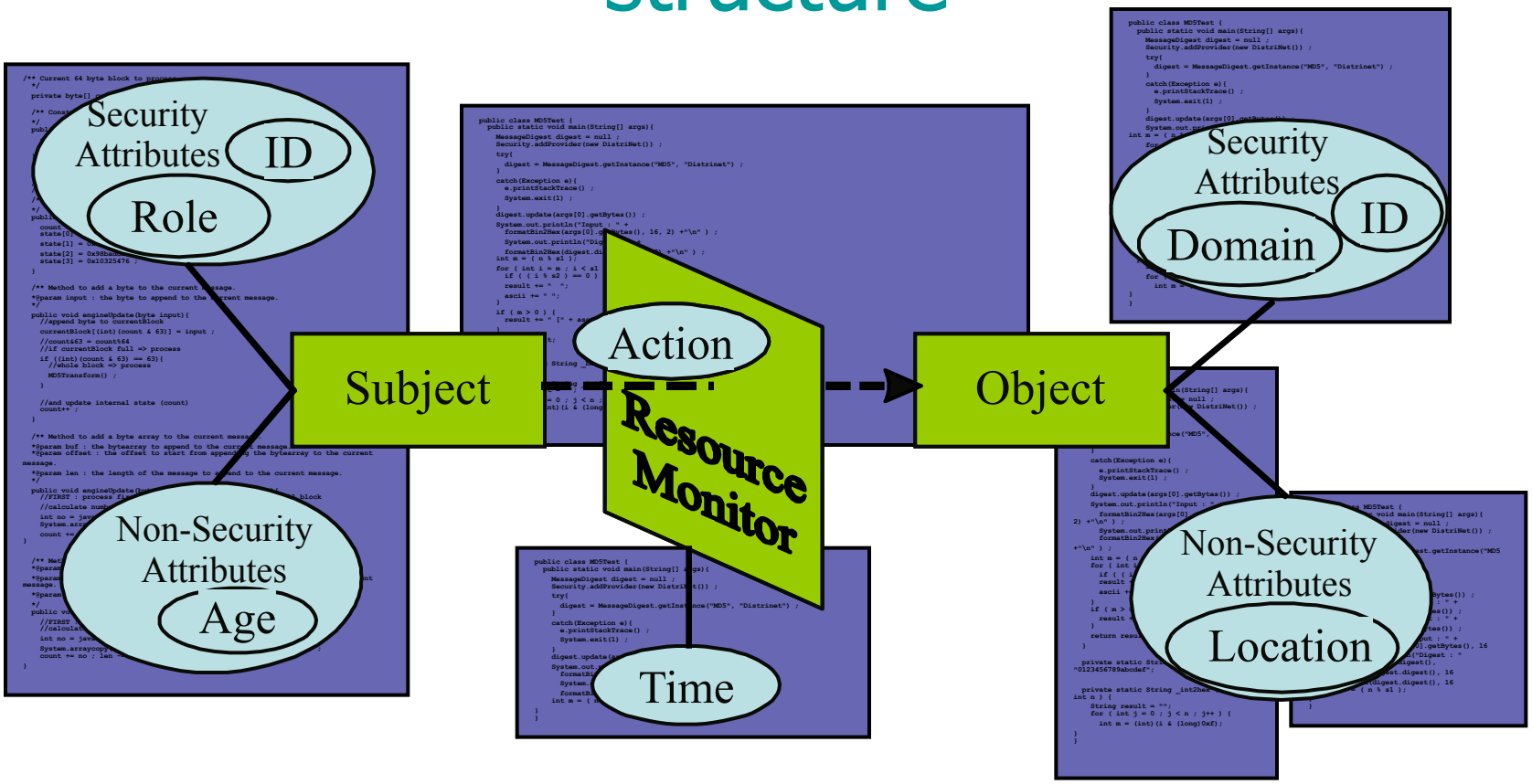
Security is Pervasive

- Application-level security is crosscutting in location



Security is Pervasive (ctd.)

- Application-level security is crosscutting in structure



AOP to the rescue

- AOP is a novel software engineering paradigm that supports the modularization of *crosscutting* concerns (including security)
- Fundamentals
  - Aspect: unit of modularity (cfr. class)
  - Advice: unit of behavior (cfr. method)
  - Pointcut: specifies points in program where aspects are to be applied
  - Aspects are "woven" into the program
- Multiple studies show that AOP can be used for the modularized implementation of application-level security
  - Improves specialization and manageability
  - Facilitates verification of the security solution

An example: integrating JAAS using AspectJ

```
Public aspect AuthAspect{
  private Subject _authenticatedSubject;
  public pointcut authOperations() = execution(String Account.getBalance());

  before(): authOperations(){
    if(_authenticatedSubject != null){
      return;
    }

    try{
      LoginContext lc = new LoginContext("sample", new TextCallbackHandler());
      lc.login();
      _authenticatedSubject = lc.getSubject();
    }
    catch(LoginException ex){
      System.err.println(ex);
    }
  }

  …
```


Integrating JAAS using Aspectj (ctd.)

```
…

Object around(): authOperations() && !cflowbelow(authOperations()){
  try {
    return Subject.doAsPrivileged(_authenticatedSubject,
                  new PrivilegedExceptionAction(){
                    public Object run() throws Exception{
                      return proceed();
                  }}, null);
  }
  catch(PrivilegedActionException ex){
    System.err.println(ex);
  }
 }
}
```

Source: "AspectJ in Action" by Ramnivas Laddad

Overview

- Introduction: using AOP for security
- *Problem statement*
- Overview of security risks
- Countering the risks
- Conclusion

Problem statement

- The construction of secure software is difficult
  - I don't have to convince you, right ? ☺
- 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);
  }
}
```


Overview

- Introduction: using AOP for security
- Problem statement
- *Overview of security risks*
  - Language-level issues
  - Tool specific problems
  - Synthesis
- Countering the risks
- Conclusion

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
- **@precedence** in its current form is not a general solution

Language-level issues (ctd.)

- Access modifiers
  - For inter-type declarations: access modifiers for an aspect's members/methods are tricky
    - Conform to the specifications, but take care !
  - Aspects can be declared public and package, but package is not enforced (bug ?)

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)

Intermezzo: the dilemma of privileged

- Security aspects often necessitate access to object internals
  - Especially true for unanticipated aspects and application-level policies
- Cost/benefit analysis of modularization by means of invasiveness:

	Advantages	**Drawbacks**
Softw. Eng.	specialization, maintainability	system evolution
Security	verification, applicability	type safety

- Tension between necessities and desirable properties is an *open problem*

=> Until better abstractions become available, it seems appropriate to continue supporting privileged access, be it in a more secure manner (see later).

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.)

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

- 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

Risk relevance

- All discussed issues are relevant in a "typical" development environment
  - Software is built and deployed within a single company
  - Adversary has no direct impact on code (developers are trusted)
  - Adversary may deliver aspect/class libraries to be inserted in the product
  - Adversary has no direct control over environment (e.g., to modify bytecode or to activate compiler)
  - Adversary could contact the software remotely

Overview

- Introduction: using AOP for security
- Problem statement
- Overview of security risks
- *Countering the risks*
  - Research results
  - Research plans
  - Guidelines
- Conclusion

Towards a solution

- Language extensions/restrictions have been proposed
  - [Gudmundson01]: pointcut interface
  - [Larochelle03]: explicitly restricting available joinpoints globally
  - [Aldrich05]: open modules as a new, more restricted aspect
  - [Sullivan05]: shielding aspect internals by crosscutting interfaces (XPI's)
- Status
  - Most of this is in the research stadium
  - Few prototypes are available
- Issues
  - Run-time enforcement is key
  - Further restrictions might be useful

Our research plans

- An **aspect permission system**, which can address (some of) these problems as well
  - Logical extension of Java's permission system
    - Support checking aspects for particular permissions
  - 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
- Key issue: represent the identity of an aspect at run-time

In the mean time: good practices and guidelines

- Use specific PCD's
- Avoid the use of privileged aspects
- Use aspects that operate at interface level as much as possible
- Structure aspects in packages

- Avoid using AOP for high-risk components (e.g., attack surface components, security kernel, …)
- Avoid using different 'sets' of aspects
- When using aspects, make sure to integrate this fully into the development environment (e.g., all compilation steps !)

Conclusion

- Using AOP for security can be useful, but risky
- Threats originate from
  - Language features
  - Implementation strategies (and bugs)

  and are intentional or unintentional
- AOP could be used for small, controllable, low/medium-risk projects
  - If you know what you're doing
- Mostly AspectJ-specific discussion. What about JBoss/AOP, Spring AOP, …?

Food for discussion

- Benefits/drawbacks of using AOP for security. What's your experience ?
  - Projects
  - AOP tools
- Privileged: to be or not to be
- Addressing security issues