

# Aspectizing Security Concerns

Ivan Minevskiy

Department of Compute Science

University of British Columbia

201-2366, Main Mall, Vancouver, BC, V6T 1Z4

1-604-585-9617

ivan@cs.ubc.ca

## ***Abstract***

Most aspects of computer security revolve around rules preventing an entity from performing an action outside of its permitted sphere of influence and inside that of another. The paper discusses a number of different security augmentation methods, such as dynamic wrappers, dataflow analysis based on extensions of type systems, and history-based access control. In general, security checks are crosscutting concerns and are hard to modularize. The paper describes how most of existing methods can be implemented in a more modular and evolution-friendly fashion with the help of aspect-oriented techniques. Additionally, it proposes how effectiveness of aspectized solutions can be validated and evaluated.

## ***1 Introduction***

Despite regular announcements of security vulnerabilities in software products, many programmers still tend to view security as a discipline that is separate from software engineering. Security became relevant to software development when computers and the programs running on them began to share resources such as printers, networks, processor cycles, and memory. The source of almost all security concerns can be reduced to the sharing of a resource. In an environment where no resources are shared and only one entity has access to a computing environment, there is almost no need for security. Occasionally you may have to protect an entity from itself so that, for example, a file system may not be deleted accidentally. But for the most part, a single-entity scenario does not require security.

Most aspects of computer security revolve around rules preventing an entity from performing an action outside of its permitted sphere of influence and inside that of another. Spheres of influence correspond typically to access to resources. All code must obey these rules and a single violation can compromise the integrity of the entire system.

In general, security checks are crosscutting concerns and are hard to modularize. Aspect-Oriented Programming (AOP) helps to localize code related to a crosscutting concern (aspect) in a single logical unit (aspect definition), and to define points (join points) in the dynamic call graph of a program where aspect-related functionality should be inserted (woven) and executed [8]. With aspects, concerns can be kept separate and be woven together with other concerns during

compilation. Moreover, aspects improve modularization and make the system oblivious of the new concerns added into it.

This paper investigates existing approaches to security augmentation and how AOP techniques can be applied to modularize security concerns.

The rest of the paper is organized as follows. Every section focuses on a family of security augmentation methods: first, existing methods in a family are reviewed and, later, an aspect-oriented equivalent is described, either existing or proposed. Section 2 discusses wrappers with security functionality for off-the-shelf components. Section 3 describes static dataflow-based techniques for detecting untrusted values reaching security-critical operations. Section 4 reviews Java Security Architecture with its access control based on stack inspection and an alternative, more secure history-based access model. Section 5 discusses race conditions in filesystem accesses. Section 6 proposes how effectiveness of aspectized solutions can be validated and evaluated.

## **2 Securing Commercial Off-The-Shelf Components**

Component-based development is getting popular due to such benefits as reusability and composability. However, Commercial Off-The-Shelf (COTS) components typically provide only commercial-grade security assurance, which often restricts their use to non-critical applications. There have been a number of efforts to augment the security functionality and assurance of COTS components by injecting layers of security logic into the interfaces between the COTS components. These layers might implement additional security protocols such as encryption and authentication, which observe and modify the data passing through the interfaces. They also may have access control or intrusion detection logic which identifies data that is known to cause harm.

In most cases, to maximize the benefit from extra security layers, the application source must be modified to make the injected software effective. This decreases the savings associated with the use of Off-The-Shelf software and is often just impractical or impossible.

The numerous existing approaches are limited to their own problem domains and are generally limited in scope to a single kind of security augmentation, be it access controls, authentication protocols, or intrusion detection. Hence, to assure security for critical applications composed of COTS components developers have to integrate and manage several security augmentation techniques, each implemented with its own individual mechanism. Additionally, security enhancements typically lack abstraction and operate on low-level data structures such as system call parameters, IP messages, and network connections.

### **2.1 Generic Software Wrappers**

Fraser et al. [7] proposed a set of techniques for simplifying the development, composition, and reuse of security augmentations for COTS software components. Their main abstraction is the Generic Software Wrapper (GSW). Each GSW is a state machine that, during run-time, listens for specified events and, if it

"hears" an event, may take some actions such as augmenting, transforming, or denying it. GSWs are efficient because they do not impose context-switch overhead. GSWs are protected because they are executed in kernel space.

GSWs are expressed in Wrapper Definition Language (WDL), which is a superset of the C programming language. WDL improves expressiveness of C by providing new domain-specific language constructs. It also helps to make GSWs platform-independent by abstracting system-specific details.

The Wrapper Life Cycle framework uses configurable rules to automatically manage the run-time relationships between GSWs and processes executing COTS components. It effectively allows multiple wrapper instances to concurrently wrap a single process, each reacting in turn to the process's system calls. The precedence of security wrappers is often important. For example, firewall and intrusion detection should wrap cryptography wrappers.

## **2.2 Applying AO Techniques**

The GSW abstraction can be emulated with the help of pointcut-advice model. The GSW state machine is similar to aspects "listening" for events at their join points. It is possible to observe and modify the data passing through component interfaces with an around advice attached to component's interface methods. To intercept the outgoing calls originated by a component, one could have a pointcut matching all calls from within the component to the packages not within the component.

Aspects have small performance overhead and are at least as efficient as GSWs. High-level aspect-oriented languages such as AspectC, AspectC++, or AspectJ are abstract and expressive enough to produce reusable platform-independent aspects. Additionally, AOP helps to keep components oblivious of security extensions wrapped around them.

It is possible to have several pieces of advice affecting the same join point. Since advices may affect the execution of each other as well, AspectC, AspectC++, and AspectJ have advice precedence logic. Programmers are able to control the precedence of advice because there's no way for the weaver to automatically know which advice should take precedence. Precedence affects both when advice executes and whether it executes at all.

If pieces of advice reside in two different aspects and one aspect extends another aspect, then advice in the subaspect receives higher precedence than advice in the superaspect. This allows subaspects to override their parents' behavior. If there is no hierarchical relationship or such relationship should not be taken into account, then aspects may take precedence over other aspects using precedence declarations. If neither hierarchical relationships nor precedence declarations exist for any two aspects, then the relative precedence of advice between these two aspects remains undefined and they will execute in an order chosen by the weaver.

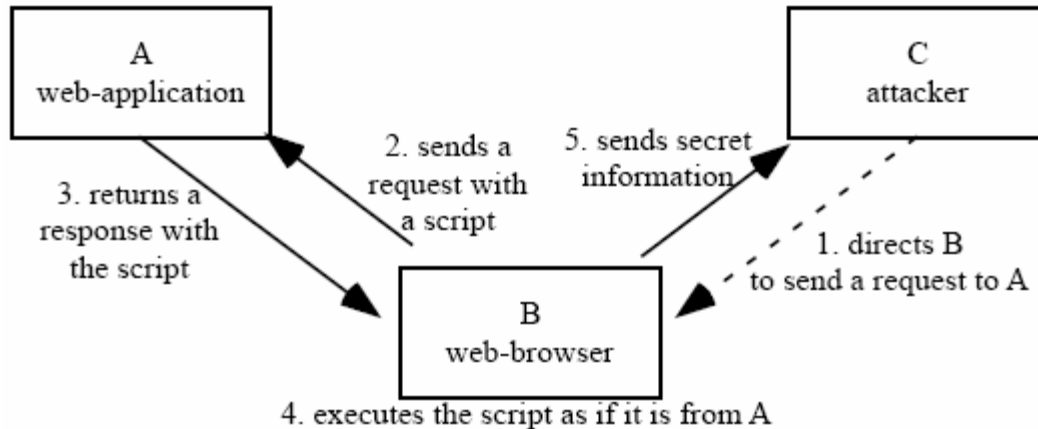


Figure 1: An Exploit of a Cross-site Scripting Problem [11]

### 3 Securing Dataflow

Security checks on components' boundaries are coarse-grained and cannot perform any reasonable judgment about data flowing through the boundary. To properly secure the code one has to: perform internal dataflow analysis, make sure that untrusted input is sanitized before being used, and make sure that sensitive data is not released without authorization. The dataflow analysis helps to detect these and many other vulnerabilities by figuring out data sources and sinks and how secure they are. Generally, all sources outside the component should be considered untrustworthy (e.g. system calls, routines that can copy data from user space, and the network).

#### 3.1 Motivating Example: cross-site scripting problem

Cross-site scripting is a security problem in web-applications. Sensitive data can be revealed without authorization due to a flaw in the browser of the web-application's client. Here is one possible scenario of such an attack with three participants: (A) a web-application, (B) a web-browser of a client of A, and (C) an attacker (Figure 1) [11]:

1. C gives B a document with a link to A. The link has a malicious script embedded as a parameter. For example, it might be a link to a login page with the script in the ID parameter field.
2. B follows the link and sends a request to A.
3. A returns a web page as a response to B's request. Due to a cross-site scripting problem at A, the malicious script appears as a part of the response. For example, the returned page may indicate a login failure with the given ID.
4. B parses the returned page and executes the embedded malicious script with A's privileges.
5. The malicious script gets A's access permissions to B's secret information.

The problem could have been avoided if A sanitized data coming from untrusted sources. For example, before processing parameters, A could transform all parameter fields into strings and then remove any special characters not intended to be there.

<pre>void foo(tainted int); untainted int x; foo(x);</pre>	<pre>void bar(untainted int); tainted int y; bar(y);</pre>
(a)	(b)

Figure 2: (a) typechecks and (b) does not because any untainted type can be used wherever the corresponding tainted type is expected, but not vice versa [12]

### 3.2 Type Qualifiers, Subtyping, and Type Inference

Imagine a type system able to mark all inputs that could be controlled by the adversary as *tainted* and all other values as *untainted*. Now type analysis can track the propagation of tainted data through a program. Any variable assigned a value derived from tainted data is itself marked as tainted, and so on. If there is any execution path in which tainted data may potentially violate security, an error is raised.

Shankar et al. [12] model tainting by extending the existing C language type system with two extra type qualifiers, *tainted* and *untainted*. The extended type system works over *qualified types*, which are the combination of some number of type qualifiers with a standard C type. Given a small set of initial manual tainting annotations, the CQUAL engine infers a typing for all program variables indicating whether each variable might be assigned a value derived from a tainted source.

The type qualifiers naturally induce a *subtyping* relationship on qualified types. CQUAL accepts new qualifiers arranged in a type qualifier lattice which defines the sub-type relationships between qualifiers. A lattice is a partially ordered set in which all nonempty finite subsets have a least upper bound and a greatest lower bound. In our case, the subtyping relation is  $\text{untainted } T < \text{tainted } T$  where  $T$  is any valid type possibly qualified with qualifiers other than *tainted* and *untainted*.

For example, consider the program in Figure 2a. The function `foo` expects tainted data, but is passed untainted data. This program typechecks because any untainted type is a subtype of the corresponding tainted one and can be used wherever the tainted type is expected. Now consider the program in Figure 2b. In this case, the function `bar` is declared to take untainted data. This program does not typecheck because the subtyping relationship does not hold.

The type checking system described so far, however, is not useful in practice. The problem is that it requires all types to be annotated with qualifiers. The solution to this problem is *type inference*. In the Shankar et al. approach [12], the user introduces a small number of annotations at key places in the program, and CQUAL infers the types of the other expressions in the program. Zhang et al. [14] go further by employing GCC and a set of PERL scripts to automatically detect the code to be annotated.

Shankar's and Zhang's approaches are very similar and can be summarized as follows [14]. All objects to which we want to control access, *controlled objects*, are initialized with a tainted qualifier. A *controlled operation* consists of a

controlled object and an operation that we execute upon that object. Functions used in controlled operations, *controlling functions*, require untainted arguments. Authorizations change the qualified type of the object they authorize to untainted. Using these qualifiers, CQUAL's type inference and analysis reports a type violation if there is any path from an initializing function (where the object is tainted) to a controlling function (where the object must be untainted) that does not contain an authorization (a cast from tainted to untainted).

Engler et al. [2] enable extension of GCC, called *xgcc*, to do source analyses, which they refer to as *metacompilation*. A high-level state-machine language, called *metal*, is used to express the necessary analysis rules as code annotations. Since the rules match multiple statements, the amount of annotation effort is reduced.

The Extended Static Checking system (ESC) [6] uses theorem proving to verify the validity of annotated Java source code.

Larochelle et al. [10] detect likely buffer overflows in C programs with help of a tool built upon LCLint, an annotation-assisted flow-sensitive lightweight static checking tool. The LCLint annotations are treated as regular C comments by the compiler, but recognized as syntactic entities by LCLint. They describe programmer assumptions and intents and restrict the range of values a reference can have either before or after a function call.

### 3.3 Applying AO Techniques

Since the sanitizing task crosscuts, it looks a good idea to implement it as aspects. However, it sometimes is not easy to do so with existing pointcut-and-advice mechanisms because they do not offer pointcuts to address dataflow, which is the primary factor in the sanitizing task.

Masuhara et al. [11] propose a new kind of pointcut, called *dflow pointcut*, that identifies join points based on the origins of data. It is designed as an extension to AspectJ's pointcut language and can be used in conjunction with the other kinds of pointcuts.

The following syntax defines new AspectJ pointcuts:

$$p ::= \text{dflow}[x, x'](p) \text{ bypassing}[x](p)$$

where  $x$  ranges over variables.

$\text{dflow}[x, x'](p)$  matches if there is a dataflow from  $x'$  to  $x$ . Variable  $x$  should be bound to a value in the current join point. Therefore, *dflow* must be used in conjunction with some other pointcut that binds  $x$  to a value in the current join point. Variable  $x'$  should be bound to a value in a past join point matching to  $p$ . Therefore,  $p$  must have a sub-pointcut that binds a value to  $x'$ .

The *bypassing* clause specifies join points that should not appear along a dataflow. It requires existence of at least one dataflow that does not go through join points matching to the pointcut in the *bypassing* clause.

They also define a new declaration form it is possible to specify explicit propagation of dataflow through executions in external programs.

With the `dflow` pointcut it is easy to avoid the information leak described by the example in Section 3.1. One could write a pointcut matching the output of strings which originate from untrusted sources.

`dflow` pointcut can detect a dataflow from variable `x` to variable `y`, but it cannot determine where `x` was initialized. However, in most cases there is a well-defined set of untrusted sources and sinks (in the trivial case, they are just any code outside of the component). Therefore, we need a set of pointcuts matching all dataflows from untrusted sources to critical operations and from critical data to untrusted sinks. If such a dataflow is found, we need to check whether there is a security check somewhere along it. `dflow`'s bypassing clause allows us to avoid intercepting dataflows with security checks, but it works only for checks which can be expressed in the pointcut notation. This restricts us to security checks encapsulated in methods. Such methods should guarantee that all untrusted values passed to them are either sanitized or reported. Unfortunately, even in Java, there are security checks that might not be encapsulated by a method. Such checks are interleaved with other code and involve some sort of primitive language operations, such as casting or comparison. For example, consider a check where a string length is compared to some constant. Here, a call to the `String.length()` method by itself does not necessarily constitute a security check that we are interested in and there is no pointcut construct to match the primitive comparison operations (`>`, `<`, `>=`, `<=`, and `==`).

Consider what happens when a data value is sanitized. The corresponding dataflow is completely avoided and is considered safe. Therefore, it is assumed that the value cannot become untrusted again. This is not true, but should not invalidate our analysis by the following reason. The only way a variable can become tainted again is if it is assigned a tainted value. This value must be coming from an untrusted source and, hence, must be matched by a `dflow` pointcut which controls its further propagation. This leads to a nice conclusion: it is enough to have at least one valid security check for each type of data values along each insecure dataflow to guarantee that no tainted values reach critical operations.

## **4 Java Security Architecture**

The Java Security Architecture does not in itself provide a secure execution environment. It is a framework that helps to meet security requirements. It can be used to regulate access to resources by program components. The Java Security Architecture enables a large number of regulated security privileges that can be configured in a policy file distributed with a Java run-time environment. Still, it has some drawbacks, such as access control based on stack inspection and scattered implementation of security policies. Also, its notion of protection domains may not be powerful enough to meet all security requirements. Therefore, even though Java incorporated some security features from the start, it is still relatively easy to write unintentionally insecure Java programs.

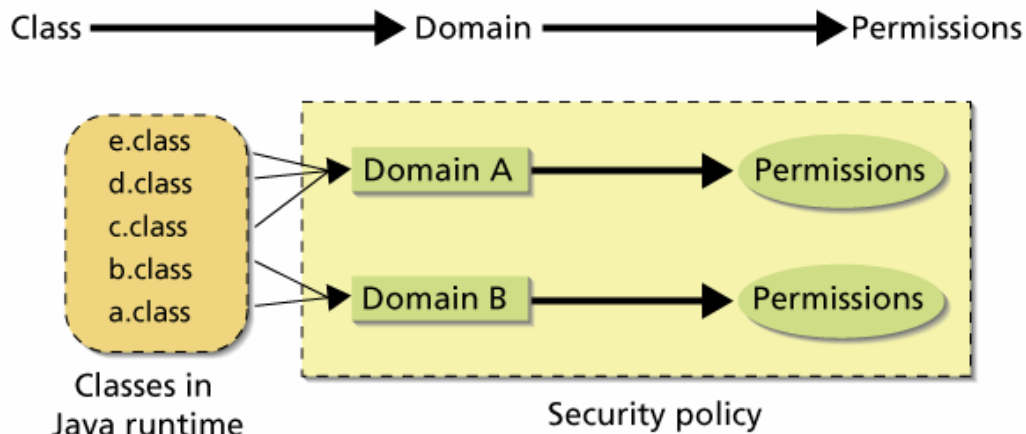


Figure 3: Protection Domains [16]

#### 4.1 Overview

The Java 2 Platform defines a security architecture based around protection domains [16]. A domain groups a set of classes that share the same set of security permissions. The concept is driven by the dynamic nature of Java applications (see Figure 3). With late binding, a program component may be loaded dynamically from the network or other source that is not under the control of the application development team. In a dynamic execution environment, it is necessary to restrict the ability of program components from interfering with the execution of other program components or gaining access to privileged resources.

Classes in one protection domain cannot gain extra privileges by accessing classes in another domain. Permissions are specified by subclasses of `java.security.Permission` and the checking of permissions is implemented by `java.lang.SecurityManager`. Explicit calls to `SecurityManager` methods are required to regulate privileges. The use of the `SecurityManager` class to regulate privileges can cause the implementation of security policies to be scattered across the source code instead of concentrated in a single location. It would be better if, similarly to .Net, security attributes were associated with classes and methods, allowing security policies to be defined completely external to source code.

It was investigated how often and where `SecurityManager` is used within JDK 1.5 source code. It is most heavily used in `java.lang`, `java.io`, and `java.net` packages. The first one contains low-level language features, such as `ClassLoader` and `Thread`. The latter two contain methods for communication with the outside world, which is the main source of security threats. In most cases, a call to the `SecurityManager` is the first action in a method. This suggests that these calls might be modularizable into aspects.

## 4.2 Access Control

Section 3 discussed protection methods based on type safety. They provide a base line of protection but are not by themselves sufficient. Most runtime environments, such as JVM and CLR, rely on access control models for security, where an access control matrix associates rights for operations on objects with pieces of code.

To determine the run-time rights of a piece of code JVM and CLR employ the technique called *stack inspection*. Following this technique, an upper bound on its permissions is associated statically before execution. At run-time, the permissions of a piece of code are the intersection of all the static permissions of the pieces of code on the stack. Thus, the run-time permissions associated with a request made by a trusted piece of code when it is called by an untrusted piece of code include only permissions granted statically to both pieces of code.

Although the stack inspection technique is widely used, it has a number of shortcomings. The main one is that it aims to protect callees from their callers, but not callers from callees. For example, A calls B and B returns leaving the system in an unexpected state. Then A calls C and this call depends on the earlier call to B. This dependency is ignored by stack inspection technique. Therefore, stack inspection remains blind to any interaction not recorded on the current execution stack.

Abadi and Fournet [1] described a new model and practical techniques for assigning rights to code at runtime. The general idea of their approach is to remember the history of the computation. The run-time rights of a piece of code are determined by examining the attributes of any pieces of code that have run. The pieces of code that have run include not only those on the stack but also those that have been called and returned. Current rights can also be modified by explicit requests. The execution unit is typically a thread or a protection domain.

In this *history-based model*, the current rights when a method terminates are lower than or equal to their value before the method call. Therefore, any non-trivial program using history-based model will have its current rights deteriorated into an empty set after some running period, unless rights are manually augmented. In contrast, with stack inspection, the current rights are always restored to their value and are possibly augmented. Therefore, given the same static rights, history-based current rights are always included in stack-based current rights.

The proposed history-based model can be implemented efficiently on top of a stack inspection model using a small amount of auxiliary state. In addition, security-aware programming is facilitated with new high-level language constructs  $\text{Grant}(P) \{B\}$  and  $\text{Accept}(P) \{B\}$ , where  $P$  is a subset of the static permissions to be amplified and  $B$  is a sequence of operations to be performed. These statements are executed as follows.

$\text{Grant}(P) \{B\}$ : Before running  $B$ , the current permissions are saved and the selected permissions  $P$  are added to the current permissions. When  $B$  completes, either

normally or with an exception, the current permissions are assigned the intersection of their initial (saved) and final values. Therefore, after executing a Grant block, rights are reduced to at least their initial state. Grant should be used to wrap a trusted code running after less trusted code (e.g., when called by that code).

Accept(P) {B}: Before running B, the current permissions are saved. If B completes normally, then the intersection of the saved permissions and P is added to the current permissions. Otherwise, the current permissions are not modified. Therefore, after executing an Accept block, rights are amplified to at most their initial state. Accept is useful when an untrusted code is running within a trusted one (e.g., when calling less trusted methods).

### 4.3 Applying AO Techniques

To emulate a history based access control model, we can create 3 types of around advices (default, grant, accept) and attach one of them to every method call. Then, by means of pointcuts, we define calls which should be accepted and granted. All calls not matched by these pointcuts are processed in a default manner with current rights after the call being an intersection of the rights before the call with the rights after the call. The actual current and static rights can be stored in an inter-type ThreadLocal object.

This emulates accept and grant statements wrapping a single method call, but, in the original scheme, grant and accept encapsulate a *sequence* of operations. Splitting a single grant/accept into a sequence of grant/accept statements might not work in the general case.

## 5 Race Conditions in File Accesses

Race conditions in filesystem accesses occur when sequences of filesystem operations are not carried out in an isolated manner. For example, if the object referred to by a file system path may be changed between its security check and the open instruction, then the second object will be opened even though its access was never checked. The object may change in one of two ways: alteration of the binding between the name and the object, or alteration of the object itself.

Bishop and Dilger were one of the first to focus on race conditions in filesystem accesses [4]. They developed a static analysis tool for finding race conditions in C code. Their tool does not perform alias analysis nor is it control-flow aware. As a result of this, the tool can produce false negative and false positive results.

Later, Tsyklevich et al. [13] proposed a mechanism for keeping track of all filesystem operations and possible interferences that might arise. If a filesystem operation is interfering with another operation, it is suspended and the first process accesses a file object. This reduces the size of the time window when a race condition exists and stops all known realistic filesystem race condition attacks.

A partial aspect-oriented solution to this problem consists in using dflow pointcuts to detect changes to the file target after its security check. A file is considered changed if its size or modification time changes. All directories in the file path should remain unchanged as well. In general, aspect-oriented techniques alone cannot provide race condition detection of the same quality as the existing methods described above.

## **6 Proposed Validation**

To demonstrate the usefulness of the proposed aspect-oriented modularizations of security functionality, several aspects will be introduced into an old version of a security-critical Java application. Then these aspects will be rolled forward into their subsequent incarnations in the next versions of the application. This validation method should help to better understand how an aspect-oriented implementation would have evolved with the application.

The method is not perfect and has some limitations. Its main limitation is that instead of producing full successive versions of the application it focuses only on the evolution of specific concerns in isolation.

The proposed application for the study is Apache Jakarta Tomcat. Tomcat is a free, open-source implementation of Java Servlet and JavaServer Pages technologies [15]. It has means of detecting and preventing a number of remote-user exploits, such as cross-site scripting, HTML injection, SQL injection, and command injection [5].

We have already seen an example of a cross-site scripting exploit in section 3. This is one of the most common web application security exploits. Such attacks are possible when a web application echoes back user-supplied request data without first filtering it. Usually, attackers attempt to steal a user's session cookie value to log into the web site as the user and gain full access to the user's capabilities and identity on the web site. This sort of cross-site scripting attack is commonly referred to as *HTTP session hijacking*.

*HTML injection* is also caused by improper user input validation and filtering. An attacker writes and inserts HTML content into a web site's pages so that other users of the site see content that the site owners did not intend to publish. For example, by inserting a web form which sends all input to attacker's server, the attacker tricks the web site's users into unknowingly disclosing sensitive data.

*SQL injection* vulnerabilities are rarer than cross-site scripting and HTML injection. By submitting a malicious SQL query string fragments in a request to a server, an attacker circumvents database security on the site. This type of attack is possible when a site has improper filtering of user input before using the user input as a part of an SQL command.

*Command injection* occurs when an attacker sends a request to a web server that will run on the server's command line in a security-breaking manner. It is caused by improper filtering of the user input before using the user input in command-line.

Dataflow-related?	Static (compile-time)	Dynamic (run-time)
Yes	CQUAL-based tools [12,14] (Sec. 3.2) Compiler Extensions [2] (Sec. 3.2) LCLint-based tool [10] (Sec. 3.2) ESC/Java [6] (Sec. 3.2) Race conditions [4] (Sec. 5)	dflow Pointcut [11] (Sec 3.3)
No		Wrappers [7] (Sec. 2) Execution History [1] (Sec. 4.2) Race conditions [13] (Sec. 5)

Table 1: all described existing methods are classified into static or dynamic and dataflow related or not related.

The security functionality protecting Tomcat from the above exploits crosscuts its code. Aspect-oriented techniques should help to better modularize this functionality, which will improve its evolvability.

## 7 Conclusions

This paper discussed a number of different security augmentation methods. Table 1 provides their summary: all methods are classified into static or dynamic and dataflow related or not related. It was also described how most of the existing methods can be implemented with the help of aspect-oriented techniques. Additionally, it was proposed how effectiveness of aspectized solutions can be validated and evaluated.

## References

- [1] Abadi, M. and Fournet, C., Access Control based on Execution History. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, San Diego, USA, February 2003.
- [2] Ashcraft, K. and Engler, D., Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the IEEE Symposium on Security and Privacy 2002*, May 2002.
- [3] Ball, T. and Rajamani S. K., SLIC: A Specification Language for Interface Checking (of C). Microsoft Research, MSR-TR-2001-21, January 2002.
- [4] Bishop, M. and Dilger, M., Checking for Race Conditions in File Accesses. *Computing systems*, volume 9(2), pages 131–152, Spring 1996.
- [5] Brittain, J. and Darwin, I. F., Tomcat: The Definitive Guide, Chapter 6: Tomcat Security. O'Reilly, 2003
- [6] Flanagan, C., et al., Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234 – 245, Germany, 2002.

- [7] Fraser, T., Badger, L., and Feldman, M., Hardening COTS Software with Generic Software Wrappers. In Proceedings of *the IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.
- [8] Kiczales, G., et al., An Overview of AspectJ. In Proceedings of *the European Conference on Object-Oriented Programming 2001*. Lecture Notes in Computer Science, vol. 2072, pages 327--353.
- [9] Lam, P., Kuncak, V., and Rinard M., Crosscutting techniques in program specification and analysis. In Proceedings of *the 4th international conference on Aspect-oriented software development*, pages 169 – 180, 2005.
- [10] Larochelle, D. and Evans, D., Statically Detecting Likely Buffer Overflow Vulnerabilities. In Proceedings of *the 10th USENIX Security Symposium*, pages 177–190, 2001.
- [11] Masuhara, H. and Kawauchi, K., Dataflow Pointcut in Aspect-Oriented Programming. In Proceedings of *Asian Symposium on Programming Languages and Systems (APLAS)*, pages 105-121, 2003.
- [12] Shankar, U., Talwar, K., Foster, J. S., and Wagner, D., Detecting Format String Vulnerabilities with Type Qualifiers. In Proceedings of *the 10th USENIX Security Symposium*, pages 201–216, 2001.
- [13] Tsyrklevich, E. and Yee, B., Dynamic Detection and Prevention of Race Conditions in File Accesses. In Proceedings of *the 12th USENIX Security Symposium*, 2003.
- [14] Zhang, X., Edwards, A., Jaeger, T., Using CQUAL for Static Analysis of Authorization Hook Placement. In Proceedings of *the 11th USENIX Security Symposium*, pages 33 – 48, 2002.
- [15] Apache Jakarta Tomcat <http://jakarta.apache.org/tomcat/>
- [16] Java Security Architecture, <http://java.sun.com>