A Catalog of Security-oriented Program Transformations

Munawar Hafiz, Paul Adamczyk, and Ralph Johnson

University of Illinois at Urbana-Champaign
201 N Goodwin Avenue, Urbana, IL 61801, USA
Email: {mhafiz, padamczy, rjohnson}@illinois.edu

Abstract. Security requirements change, but the typical way of improving system security by patches is ad hoc and has not produced good results. Security improvements should be systematic, just as new features can be added to software systematically. It would be easier to improve the security of a system if we had a catalog of security-oriented program transformations that could be used to plan changes, to divide the work to make changes, and as a target of automation. This paper describes a catalog of security-oriented program transformations that were derived from security patterns. It describes several ways of categorizing these transformations, our first attempts at validating the catalog, and how the catalog can be used to improve the security of systems.

1 Introduction

Security is architectural; it is a property of the entire system, not one part of it [7]. Security cannot be added to a system by adding a module, but it can be added in other ways. In particular, it is possible to improve the security of a system by applying program transformations to the existing code base.

When a system starts to show security vulnerabilities, its owners are faced with the choice of replacing it or fixing it. Replacing the system can be expensive and risky, so usually the choice is to fix it. But the most common approach to fixing security vulnerabilities is to patch them, i.e. to fix each instance of a vulnerability separately. Users see a constant stream of patches, leading security experts to claim that it is not possible to add security to existing systems [1] [23] or at least it is economically infeasible to do so [3]. However, this is surely an exaggeration. Even if it is more expensive to retrofit security to a system than to design it in from the beginning, it will be even more expensive to rewrite a system from scratch each time a new type of vulnerability appears. Any large software system is constructed incrementally; one feature is added before another, and components are split and recombined. There is no obvious reason why techniques that are used on new system cannot be retrofitted to an existing system, especially if these techniques can be automated. Verifying this observation requires making a list of all techniques for making systems secure and checking that each of them can be retrofitted to an existing system.

We have studied the available security solutions and collected them in a catalog of techniques that ensure security in software systems. We have described each one as a program transformation. In some cases, these program transformations are easy to perform manually. In others, they can be automated. As we present our transformations, we will also discuss what parts can be automated and how. Some program transformations require making significant changes to the system, for instance adding a new security component. Building the logic to automate such a change in a tool might not be needed for a single system, since the transformation needs to be applied only once, but automating it will produce a reusable tool that can be applied to other systems. Other transformations are small but ubiquitous, for example replacing unsafe function calls with secure calls. Although the choice of the replacement functions should be left to the developers of each system, such a transformation should also be applied by a tool, rather than manually, to avoid programmer mistakes. Yet other transformations can be performed with already existing tools. Whenever such tools are available, we will discuss how these tools can perform the automated part of a transformation. We believe that the applying these program transformations can *add security* to existing systems.

The major contribution of this paper are:

(1) The definition of security-oriented program transformations as a mechanism to retrofit security to existing systems.
(2) A large collection of security-oriented program transformations that solve the most important kinds of security problems.
(3) A system for classifying security-oriented program transformations.

This paper has two parts. The first part introduces security-oriented program transformations. Next we present our classification system that characterizes common aspects of these transformations. This classification will serve to structure the catalog presented in the second part of this paper. We provide examples of several security-oriented program transformations, how they are designed, how they are applied and how they are composed to add defense in depth [38] to a system. We also evaluate our catalog and review related work.

The second part of this paper (the Appendix) describes our current catalog of security-oriented program transformations. We give a brief summary of all the program transformations in our catalog. For each transformation, we describe what a developer has to specify and what the tool does. Every transformation that we have included works—and most of them could be automated.

2 Security-oriented Program Transformations

A program transformation is a function that maps programs to programs. A *security-oriented program transformation* maps programs to security-augmented programs, i.e. it introduces a security solution to make programs more secure. The catalog described in this paper contains forty two transformations. The Appendix contains a summary of each of them. For their full description, see `http://netfiles.uiuc.edu/mhafiz/www/sopt.pdf`.

Each program transformation is a sequence of steps. Some of the steps are easy to automate, either with existing tools or with simple tools that we have built. But some other steps cannot be easily automated, because their details are different in each system. Program transformations in our catalog separate the steps of applying transformations that are manual (*policy*) from the steps that are part of an automated tool (*mechanism*). The separation makes it a lot easier to design automated tools because a tool does not need to have a detailed understanding of application logic.

Program transformations are "general purpose", but no transformation will work with every program. They usually expect a certain programming language or a certain platform, because the exact details are language-specific. By general purpose transformations, we mean that a transformation for a certain programming language should be applicable to all programs written in that language.

Our security-oriented program transformations make structural changes that do not depend on detailed understanding of the application logic. This property makes them similar to refactoring [10]. Both refactoring and security-oriented program transformations are types of program transformations. However, program transformations for security are not behavior-preserving the way refactorings are. A security-oriented program transformation preserves the correct behavior and fixes the incorrect behavior caused by a security vulnerability. Our program transformations are behavior-preserving when the system is used correctly; they preserve the good path behavior. Only attackers see change in the behavior, because security-oriented program transformations eliminate the source of vulnerabilities the attackers want to exploit.

Security-oriented program transformations could be automated, because they separate policy from mechanism. To use a security-oriented program transformation, a developer has to follow three steps–1) identify the program points where to apply a transformation, 2) determine which transformation to apply, and 3) use a tool to automatically transform the program. The first two tasks are manual; a developer identifies where to apply a transformation and which transformation to apply. Usually, a developer supplies these parameters to a program transformation with a manual specification. Consider a program transformation that minimizes the number of entry points, which we call *Single Access Point*. If all the places where input enters a system could be consolidated into a few points, then all the necessary checks of the input could be performed at those entry points. But how does one minimize the number of entry points? Once all the entry points of a system are identified, it is possible through a series of code edits and refactorings to merge entry points together to produce a minimal number of points. However, identifying all entry points is not easy to automate. It is better if a developer manually specifies the entry points. A developer's specification is the *policy*. The tools implement the *mechanism*; they automatically execute structural changes. With this separation, program transformations can automate new security solutions without the need to encode deep understanding of program behavior in the transformation. In the example, a tool will automate the code edits and refactorings required to minimize the number of entry points.

All of our program transformations separate policy from mechanism in this way. For each program transformation described in the Appendix, we describe what a developer has to specify and what the tool does to apply the changes.

3 Deriving the Catalog

Our catalog of forty two security-oriented program transformations derives from earlier work on security patterns. We have been maintaining a comprehensive catalog of security patterns [13]. The catalog lists ninety one patterns that appear in many books, catalogs and papers on security patterns. Many of the transformations in this catalog are derived from security patterns.

To produce a catalog of security-oriented program transformations from a security pattern catalog, we performed three steps. The first step is to select the candidate security patterns that could be described as program transformations. Not all security patterns are about code and software design. There are security patterns for asset evaluation, risk assessment, and threat modeling [30] that describe a process, and could not be described as a program transformations of software artefacts. Also, security patterns that describe security principles such as *Defense in Depth* [38] do not have any corresponding program transformations. Program transformations do not exist for yet another set of security patterns that are about hardware, such as which firewall to use in which context [30]. Removing all these security patterns from the original list of ninety one, we have selected thirty six patterns that could be described as program transformations.

The second step is defining the mechanics of the transformation. The mapping from a security pattern to a program transformation is not trivial. A security pattern provides a high-level description of a security solution. It does not describe how to implement the solution nor the steps to transform a program to introduce the solution. For each candidate security pattern, we had to identify the implementation method and then describe the steps of program transformation. For example, a *Safe Data Buffer* [15] security pattern provides a solution to prevent buffer overflow attacks. Buffer overflow occurs because an unsafe language such as C do not check buffer bounds while performing a buffer operation. The *Safe Data Buffer* pattern says that one should check for length information before performing any operation on the data. It does not give any hint of how to implement the solution. The lessons from this pattern has been implemented by a number of safe string libraries (e.g. strlcat and strlcpy [24], libmib library [8], ISO/IEC 24731 [19], libsafe library [2] etc) and safe data types [27, 15]. We have described the solution as a *Safe Library Replacement* transformation. The transformation finds all instances of an unsafe library in a program and replaces each instance with a corresponding safe library.

The third step is finding how to automate a program transformation. Not all program transformations can be automated. But if a program transformations could be automated, making it a part of automated tools will make it easier for the developers to apply them. Some program transformations could

be completely automated, but most transformations require some manual intervention. Building automated program transformation tools would benefit if the tools make structural changes only, similar to automated refactoring tools. We refer to this as the separation of policy (manual specification) from mechanism (structural change).

Not all the transformations in the catalog derive from security patterns. Program transformations to create a message digest, or a message signature describe how classes of Java or .NET security API could be composed to perform the tasks. The full listing of the catalog in the Appendix cites the source of all transformations. A detailed description of how we created the program transformation catalog from the original pattern catalog is available at `http://netfiles.uiuc.edu/mhafiz/www/transformation-catalog.pdf`.

4 Categorizing Security-oriented Program Transformations

Security-oriented program transformations vary in their impact on an existing program—some transformations make changes at a few parts of a program, others affect many parts of a program, while there are program transformations that go beyond the boundaries of a single system. These categories could be used as a scheme to organize the transformations. Another scheme to organize the transformations in the catalog is the type of change they make. Yet another scheme is based on the type of vulnerability that is fixed by a program transformation. Categorizing program transformations based on these common properties makes it easy to refer to families of related transformations, to learn the transformations in the catalog, and to find new transformations.

We have organized our catalog based on three orthogonal criteria, *impact of transformation*, *vulnerability fixed by transformation*, and *type of transformation*. Combining these criteria gives a three-dimensional model for categorizing transformations. To present them in a readable format, we have broken up the catalog into five tables corresponding to five types of vulnerabilities. Tables 1 through 5 list the transformations.

Impact of transformation denotes the impact of applying a transformation on a codebase. We have identified four categories of impact that a transformation has on a codebase.

(1) *Small, infrequent change.* Some transformations have a small footprint. The code change for each instance of a transformation is small; also the frequency of these changes in the total codebase is small.

(2) *Small, frequent change.* The code changes for an instance of a transformation might be small, but the changes occur at many places of the code.

(3) *Large change.* These transformations make large changes on a codebase.

(4) *Change beyond system boundary.* Some transformations require code change beyond system boundaries, i.e. in multiple network elements.

The goal of a program transformation is to fix a single (or maybe more) type of security vulnerability. A versatile catalog should contain program transformations that fix many important types of vulnerabilities. OWASP [28] identifies top ten security vulnerabilities that affect modern systems. They are, unvalidated input (A1), broken access control (A2), broken authentication and session management (A3), cross site scripting flaws (A4), buffer overflows (A5), injection flaws (A6), improper error handling (A7), insecure storage (A8), denial of service (A9), and insecure configuration management (A10). Among these, cross site scripting (A4), buffer overflows (A5) and injections flaws (A6) fall under the umbrella of unvalidated input (A1). Securing storage (A8) has two concerns: deciding the appropriate cryptographic solution and implementing proper access control with the cryptographic solution. Deciding the appropriate cryptographic solution is entirely a manual activity that is part of requirement engineering, while implementing proper access control is the same as fixing access control vulnerability (A2). Securing configuration management (A10) is related to software maintenance rather than design. Thus our classification has five distinct types of vulnerabilities that can be solved by modifying source code:

(1) Unvalidated Input

(2) Broken Access Control

(3) Broken Authentication

(4) Improper Error Handling

(5) Denial of Service

Another criteria for grouping transformations is based on the type of change they introduce to the system. We have identified four types of code changes. They are:

(1) *Add Component.* Instantiate a component that adds a new security functionality and plug it in the existing component.

(2) *Limit.* Set a limit on the resources and check it before a resource is used.

(3) *Mutate data.* Change encoding of data, validate data, or rectify data to remove data related security vulnerabilities.

(4) *Distribute.* Break up a single entity into smaller parts, e.g. distribute a task between smaller subtasks, or break up a process into multiple processes.

The following sections describe the categories in greater detail and provide examples of transformations in each category. We describe them in the same way they are presented in Table 1 through 5. We describe each type of vulnerability, and highlight some transformations from each category.

4.1 Unvalidated Input

Insufficiently validating input data is the root cause of various types of injection attacks, which is one of the most often exploited security vulnerabilities. We discuss security-oriented program transformations that deal with injection attacks in detail in our previous work [14].

	Small, infrequent change	Small, frequent change	Large change	Change beyond boundaries
Add	T7. Add Perimeter Filter T25. Message Intercepting Gateway			
Limit	T13. chroot Jail	T32. Safe Library Replacement		
Mutate data	T20. Fuzzing	T17. Decorated Filter T30. Randomization		
Distribute		T42. Unique Location for Each Write Request		

Table 1. Security-oriented Program Transformations for Unvalidated Input

Most security-oriented program transformations in this category are about applying input validation checks to program variables. Each transformation modifies a small amount of code, specifically, the code where input data is received.

Add component. Since the change has a small footprint, developers have the option of applying a transformation manually or using tools. For example, a developer can *Add Perimeter Filter* component at the system entry point manually or automatically. This transformation should be applied to a system that has few entry points, i.e. few places where input data enters the system. (To ensure that a system has few entry points, it is beneficial to apply the *Single Access Point* transformation first.) The code change involves adding a filter class or component and calling it from each entry point. A tool can automatically add the component and the call to it given a developer has identified the system entry points. The transformation made by a tool does not require deep understanding of program behavior that is required to identify the entry points.

Limit. A somewhat different transformation is the *chroot Jail* transformation, which limits a Unix process from writing beyond a directory hierarchy. This is an atypical transformation, because it requires very few changes in the source code or binary. Instead, the program transformation creates a jail environment for a process. The transformation analyzes a program and finds its dependencies on static and dynamic libraries, file descriptors, etc. The tool could also properly add the chroot calls in the source code so that file descriptors are closed across the call, the privilege of the calling process is lowered, and the shared resources have proper permissions.

Mutate data. Program transformations that mutate data require developers to specify the policies that modify (or mutate) input data. A *Decorated Filter* transformation adds input validation policies, specified by developers, to decorate [11] an input variable. These policies are injection attack specific; thus they can be easily extended as new injection attacks emerge. A tool for this transformation is very simple – it automates the *Moving Embellishment to Decorator* [10] refactoring, which is a purely structural change. A tool does not need

	Small, infrequent change	Small, frequent change	Large change	Change beyond boundaries
Add Component		T8. Add Reference Monitor T21. Guarded Object	T4. Add Authorization Enforcer	T27. Parallelized Link T25. Secure Pipe
Limit		T15. Controlled Process Creation T14. Controlled Object Factory T22. Least Privilege	T36. Secure Resource Pooling	
Mutate data		T24. Message Digest Creation T38. Signature Generation T18. Encryption/ Decryption T34. Secure Message Router		
Distribute			T28. Partitioning	

Table 2. Security-oriented Program Transformations for Broken Access Control

deep understanding of program behavior since it does not identify by itself where to apply the transformations or which input validation policies to apply. These are specified by the developers.

Distribute. Some program transformations break a single point of failure into multiple entities. A *Unique Location for Each Write Request* transformation prevents multiple processes from writing to one file by creating a unique file for each write request. This transformation creates a temporary file for each write task, writes data to it, and saves the file with a unique name. It also modifies the file read mechanism for the data consumers.

4.2 Broken Access Control

Failure to restrict users properly allows attackers to access other users' accounts, view sensitive files, or use unauthorized functions. Program transformations that prevent access problems follow various strategies, e.g. introduce an authorization component, partition to a lower privilege level, or employ cryptographic techniques to keep data confidential. Many members of this category are large transformations, some transformations go beyond a system boundary and need to be applied to multiple systems.

Add component. Adding an authorization component at the system entry point is similar to adding a perimeter filter, because both apply to the system entry point and both add a component or a group of classes and delegate the task to the composition. However, the impact of change is much larger for an *Add Authorization Enforcer* transformation, because it involves composing a lot of classes. For example, an authorization component based on Java Authentication and Authorization Service (JAAS) framework creates a collection of permissions and stores it in the credential set of a subject. Composing an authorization

component involves composing many classes, e.g. classes describing a subject, its principals, their permissions, permission collection, credentials, request context.

Mutate data. Another type of transformation that indirectly helps access control is to apply cryptographic operations to program data. For example, a developer might specify the algorithm used for calculating a message digest of a data value, and a *Message Digest Creation* transformation automatically introduces calls to a security API to calculate the digest.

Limit. Sometimes enforcing stricter constraints result in better access control. A *Secure Resource Pooling* transformation modifies the manager process of a resource pool to limit the lifetime of worker processes in the pool. A developer has to specify a policy that determines the lifetime of worker processes, e.g. number of times a process can serve, number of times a process remains idle, total amount of time a process is in the pool etc. A tool could automatically modify the manager process to add the policy.

Distribute. Distribution of tasks could lower the required privilege level required for a process which eventually improves access control. Partitioning [38] a monolithic process into multiple processes lowers the privilege level required to run a process. It also allows better privilege separation because each process can run with separate user ids. A *Partitioning* transformation accepts a process and a specification of functional distribution from developers and distributes the tasks of a program into multiple logical partitions so that processes in each partition run with a single privilege level. There are tools available for specification-driven partitioning, but usually their goal is to get better performance [31] [18] [34] [35], better reliability [16] or better energy consumption [39]. These tools automate the mechanism for remote references, such as replacing method calls with remote method calls, direct object references with proxy references etc.

Large transformations are harder to automate and they require more detailed specification from the developers. For a *Partitioning* transformation, a developer has to specify the functionality distribution, interface, privilege level and interprocess communication mechanism. A specification for an *Add Authorization Enforcer* transformation has to include details about principals, requesting context, permissions, etc.

4.3 Broken Authentication

Many security problems that are under the category of unvalidated input or broken access control are in fact problems with the authentication mechanism of a system. Bad authentication allows an intruder in, who can then take advantage of other vulnerabilities in the system. User authentication typically involves the use of a userid and password. Stronger methods of authentication are commercially available such as software and hardware based cryptographic tokens or biometrics, but such mechanisms are prohibitively costly for most applications.

Most program transformations add large authentication components and delegate the task. In some cases, the components span multiple system boundaries.

	Small, infrequent change	Small, frequent change	Large change	Change beyond boundaries
Add Component			T3. Add Authencation Enforcer T16. Credential Tokenizer	T37. Secure Session Object T40. Single Sign On Delegate T6. Add Password Synchronizer
Limit	T1. Add Account Lockout			

Table 3. Security-oriented Program Transformations for Broken Authentication

Add component. Similar to authorization components, authentication components are added at the entry point of a system. A program transformation adds a component and delegates a task to it. Composing an authentication component has to keep track of a lot of things. For example, an authentication component based on JAAS framework creates a login module that takes the login decision. A developer has to specify how the credentials, e.g. username and password, are passed to the login module. A login module decides on the authentication, and initializes a subject. It populates a subject with principals and credentials that is used by authorization components at different points in the system.

Large program transformations require large user specifications. For an *Add Authentication Enforcer* transformation that creates a username and password authentication component, a developer has to specify where the inputs are coming from, where and how a system stores username password pairs, how the authentication decision is made, and what principals and credentials to add given an authentication attempt is successful. It is difficult for a user to provide all these specifications for an automated tool; at the same time parsing a complex specification would make a tool complex. A simpler alternative for a tool would be to create a default authentication component and let the developers customize it for their specific needs. This can be done by creating hook methods, or by adding a Strategy [11]. Frameworks such as JGuard compose authentication components based on user-provided specification. We will provide the details when we describe an example in section 5.

Limit. Even solid authentication mechanisms can be undermined by flawed credential management functions, such as password change. An *Add Account Lockout* transformation adds a limit on the number of login attempts. Unless the authentication component is created following a standard framework (JAAS or .NET), it is difficult to add this check automatically. However, since it is only done at a few places of a system, it could be done manually.

4.4 Improper Error Handling

Improper handling of errors introduces a variety of security problems. Attackers use this weakness for launching reconnaissance attacks. If a system shows internal error messages such as stack traces, database dumps, and error codes to users,

	Small, infrequent change	Small, frequent change	Large change	Change beyond boundaries
Add Component		T2. Add Audit Interceptor T5. Add Error Message Suppressor T33. Secure Logger		
Mutate data		T19. Exception Shielding		

Table 4. Security-oriented Program Transformations for Improper Error Handling

attackers learn the implementation details that should never be revealed. An example is passing a malformed SQL query to get information about the database schema, popularly known as the blind sql injection attack.

Program transformations to fix improper error handling are small. They typically mutate error messages to suppress internal information.

Error messages should nevertheless be stored internally with all the available details. There are program transformations that add a logging component and delegate logging task to it. Both error message mutation and error detail storage changes occur frequently in a system.

Mutate data. Applications frequently generate error notifications, e.g. out of memory, null pointer exceptions, system call failure, database unavailable, network timeout, malformed input, etc. An *Exception Shielding* transformation is similar to a *Decorated Filter* transformation, because both decorate [11] a variable with various policies for suppressing error messages. Their difference is in the context: while the latter applies validation policies on input variables, the former applies suppression policies on outputs that are generated. Another transformation is to add a component that intercepts all types of error messages and apply policies globally (*Add Error Message Suppressor*). The advantage of the *Exception Shielding* transformation is that the suppression policies could be fine-tuned for each input.

A developer has to specify which output variables to intercept and the error suppression policy. A tool could contain a library of policies from which a developer selects the appropriate ones for a context. The library of policy should be extensible so that developers could add their organization-specific custom policies. One example policy is to group multiple error messages and provide a generic error message. Consider, two error cases in an access control context–a request for a non-existent file versus a request for an existing file for which a user does not have access permissions. Typically, separate error messages are generated for separate error cases. Unifying the error messages to a generic message will not reveal to a requester whether a file exists in a system or not.

Add component. Adding components for keeping track of internal error messages and internal states during execution indirectly help in error handling. Adding a simple logging component requires a small change. However, features

	Small, infrequent change	Small, frequent change	Large change	Change beyond boundaries
Add Component	T23. Message Caching			
Limit		T31. Resource Management	T11. Batched Tasks	
Distribute			T26. Parallelized Functionality	T9. Add Replicated System T10. Add Standby

Table 5. Security-oriented Program Transformations for Denial of Service

such as message digest, signature, encryption could be added for producing more secure logs. Eventually a logging component could be quite complex and large if these additional transformations are applied on a logging component.

4.5 Denial of Service

Denial of Service in software is different from a denial of service attack on the network, a.g. a SYN flood attack. The primary goal of DoS attackers on software is to consume all of some required resource to prevent legitimate users from using the system. Another type of attack is to deliberately crash a component to make it unavailable. Strategies to survive these attacks vary; sometimes systems should be parsimonious and enforce a limit, while at other times they should act generous and use redundancy. The impact of the program transformations varies from small localized changes to modifying multiple systems.

Add component. Attackers typically send multiple messages with garbage payload to overload CPU and memory. At the system entry point, a separate component could be added by an *Message Caching* transformation that intercepts incoming messages, looks for replayed messages, and drops them. However, the policies used to determine a replayed message should be very conservative, because it is difficult to distinguish a good traffic flood from a bad traffic flood.

Limit. There are program transformations that make large change in code. One such change creates parallel threads of execution in the hope that at least one thread survives a crash attempt. Some changes, on the other hand, are parsimonious on resource usage. A *Batched Tasks* transformation adds a task queue at a program point that stores tasks and performs them in a batch.

Distribute. Redundancy is a common method to fight against DoS attacks that results in changes beyond system boundaries. For example, the *Add Replicated System* transformation results in duplicating the functionality of a system throughout a network. The developer specifies which parts of the system to distribute, the number of copies, and their location. The transformation makes copies of system components and creates an additional load balancing component. The programs that communicate with the component also have to change.

The larger transformations require a lot of specification. Even with the specification, the scope of automation is very limited.

4.6 Miscellaneous Program Transformations

There are a few program transformations that could not be classified with our categorization scheme.

Some program transformation solve more than one vulnerability. For example, a *Single Access Point* transformation centralizes the access points of a system. It pulls up all methods [10] that provide an entry point to a new component that is a Façade [11]. The way to implement this transformation for a non-object oriented system is to create a wrapper component and add a *Least Privilege* transformation to lower its privilege level. This program transformation has a large impact, however, it provides a framework to solve authentication, authorization and input validation problems and does not fit in one category.

After a *Single Access Point* transformation is applied to minimize the access points, a *Policy Enforcement Point* transformation is applied to create a separate component. The access point delegates all authentication, authorization and input validation requests to this component. The policy enforcement point sequentially calls authentication enforcer, authorization enforcer and perimeter filter components to perform the tasks. The separation of these calls in the access points allow them to run with lower privilege level. The policy enforcement point component is a Mediator [11] between components. A *Policy Enforcement Point* transformation has a small impact, and it only applies at a few program points. It could be classified in the 'add component' category. However, it fixes authentication, authorization and input validation vulnerabilities; hence it does not fit in one category.

Two other transformations that do not fit cleanly in one category are *Checkpointed System* and *Single Threaded Façade*. This brings the total number of transformations to forty two.

5 Applying Security-oriented Program Transformations

In this section, we will describe how a security-oriented program transformation could be applied to add authentication and input validation features to an existing software. Each program transformation is platform and language specific; we assume that the transformations are applied on a Java program on Windows platform. We also assume that the authentication scheme to be added is the ubiquitous username and password based authentication scheme.

Suppose, an imaginary developer Alice has a Java program that connects to a hostname and a port via a socket. It connects to two hostname/port pairs and uses the data received from one connection to query an SQL database and the data received from another connection to query an LDAP database. Alice designed the program for the employees of her company; everybody was trusted, everybody had the same rights, hence there was no need for authenticating users. Now, Alice has to retrofit authentication, authorization, and input validation, because her company is planning to let external users use the program.

Let us follow Alice's thought process. The first thing to consider is where to add the new authentication, authorization and input validation features. Since,

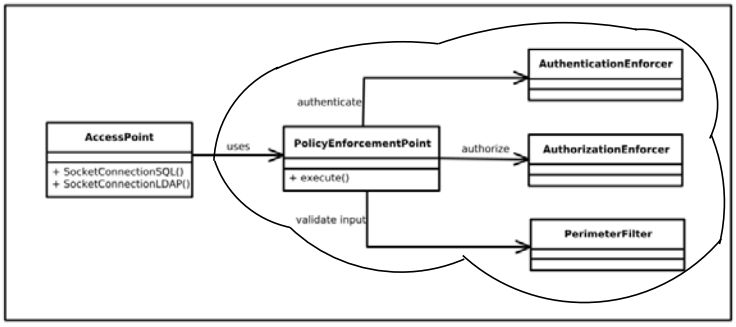

Fig. 1. Classes added by a *Policy Enforcement Point* transformation

her program has two separate methods for socket connection (henceforward, we will refer to them as the two access points), she could apply the *Add Authentication Enforcer*, *Add Authorization Enforcer* and *Add Perimeter Filter* transformations in the two places of the program. But applying program transformations in this manner might open more chances of errors. For example, if one method is patched later, the programmer writing the patch needs to remember to fix the other method too. This can be avoided, if Alice minimizes the access points of the program to one place by applying the *Single Access Point* transformation. This transformation is usually followed by a *Policy Enforcement Point* transformation. A policy enforcement point provides a single check point for authentication, authorization and input validation. Alice could then apply *Add Authentication Enforcer*, *Add Authorization Enforcer* and *Add Perimeter Filter* transformations (we will describe here how the authentication and perimeter filter components are added). The perimeter filter applies global input validation policies, but many other validation policies are input-specific. Hence Alice could apply the *Decorated Filter* transformation on several places of the program.

We will describe these five program transformations—*Single Access Point*, *Policy Enforcement Point*, *Add Authentication Enforcer*, *Add Perimeter Filter* and *Decorated Filter* transformation. These program transformations are applied in this sequence.

The two socket connections are in separate methods in two separate classes. Alice selects these methods as the target of a *Single Access Point* transformation. The transformation pulls up the methods [10] to a new class. However, the methods to be pulled up may not share a common superclass. In that case, a program transformation might create a common superclass, or give each of them a component and create a common superclass for those components. The new class is the unified system access point. For the subsequent transformations, this will act as the base.

Next, Alice selects the `AccessPoint` class created in the previous step and applies a *Policy Enforcement Point* transformation. Figure 1 shows the classes added by the transformation. The `AccessPoint` class delegates authentication request to an `AuthenticationEnforcer` class. This class is a placeholder and

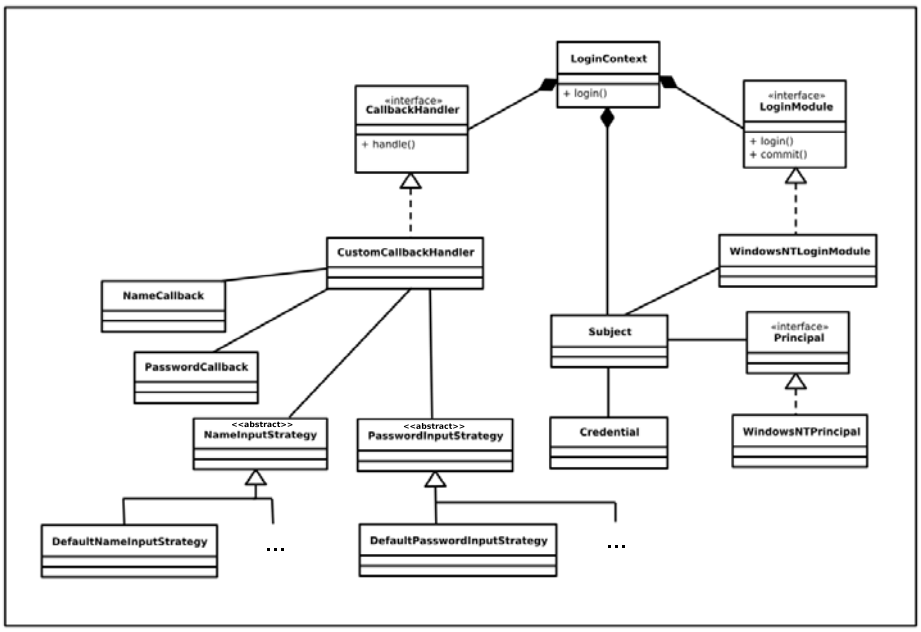

Fig. 2. Classes added by an *Add Authentication Enforcer* transformation

could be initialized to return a negative value by default. This class is the starting point of the next transformation in our list that creates an username/password-based authentication component. Similarly, default classes are added for an authorization component and a perimeter filter component.

In the next step, Alice replaces the `AuthenticationEnforcer` with a group of classes that perform username/password based authentication (see Figure 2). The *Add Authentication Enforcer* transformation composes the classes of the JAAS framework to provide a default authentication scheme. In our example, Alice specifies authentication type (username/password), source of user inputs, authentication knowledge base (username/password store), and the outcome of a successful authentication (which principals and credentials are added to the authenticated subject). The `AuthenticationEnforcer` creates a new `LoginContext` instance and calls its `login` method. This creates a new `Subject` representing the authentication requester and passes it to a `LoginModule` implementation. For this example, a `WindowsNTLoginModule` is provided by the JAAS framework. It implements two methods—the `login` method decides on the authentication, and the `commit` method populates the subject with appropriate principals and credentials when the authentication is successful. User inputs are collected through the implementers of the `CallbackHandler` interface. The program transformation provides an implementation of a default strategy which will most likely be modified. Alice could apply an *Add Strategy* [20] refactoring and add a custom implementation later. Similarly, the authentication algorithm in a customized implementation of the `LoginModule` interface could be a Strat-

egy [11] that developers could extend. The *Add Authentication Enforcer* transformation provides a default implementation only; Alice then customizes to fit the application specific requirements.

Next, Alice applies the *Add Perimeter Filter* transformation to extend the default implementation of `PerimeterFilter` class. She selects a list of input validation policies and the program transformation adds filters that implement the policies to the `PerimeterFilter` class. They can be added as a `Collection`, or by embellishing the `PerimeterFilter` class with a Decorator [11].

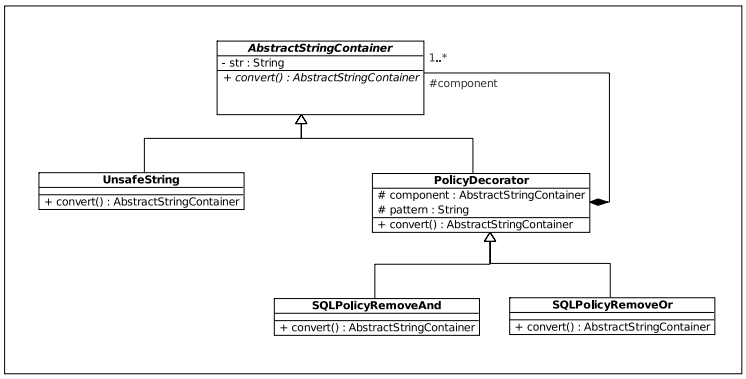

Fig. 3. Classes added by a *Add Perimeter Filter* transformation

Some policies are input specific. Alice applies these policies on the particular instances of variables. She chooses the filters from a list of implemented policies for each input variable. The policies for SQL injection and LDAP injection are different. The program transformation adds the policies to an input variable by using Decorators [11]. Figure 3 describes how a string variable is decorated with policies that remove SQL injection attack vectors. The string input variable becomes encapsulated in the abstract `AbstractStringContainer` class. Its concrete instance `UnsafeString` becomes the target of input validation policies.

A *Single Access Point* transformation is small, it could be done manually. But the other transformations require writing a lot of new code, and a significant portion of the code involves composing classes of a security framework to create a component. Alice could use a tool that automates these boring details. In each of these cases, Alice specifies the changes that need to be made, and a tool makes the changes.

6 Evaluation

The evaluation of the usefulness of any catalog of solutions depends on two aspects—how much do the members of the catalog cover a problem area, and how often are the solutions used. Evaluation of our catalog along any of these aspects is an ambitious venture. We will describe why and nevertheless provide our argument in favor of the catalog.

6.1 Coverage

Making a quantitative analysis of what percentage of security problems does our catalog cover is hard because the definition of security and security solutions is fuzzy. Security is an emergent property; it is unlikely that one can add security to a system by adding a component or by modifying code in one part of a system. Hence, any one particular program transformation, such as *Add Authentication Enforcer* does not solve all authentication problems; it only contributes to the overall security of the system. Nevertheless, transformations in our catalog cover the most relevant security problems faced by today's developers.

Our program transformation catalog originates from the analysis of both the problem domain and the solution domain of security. We have analyzed various reports on vulnerability trend analysis [6] [29] [33] [36] [37] to identify the most relevant security problems. On the other hand, our comprehensive security pattern catalog compiles the solutions that experts use to solve security problems. Combining these two dimensions ensure that our catalog includes transformations that provide real solutions for real problems.

We have surveyed the Bugtraq [5] list in the first week of September 2008. 54% of the vulnerabilities in that week were some form of input validation problems, 13% were related with bad authentication and bad access control, 17% were denial of service problems, 1% were error handling problems, and 15% were miscellaneous other problems or combinations of the the aforementioned problems. In reality, the partitioning is not mutually exclusive, 33% of the denial of service problems in the survey could be solved by some sort of input validation. Our program transformation catalog contains transformations that could be used to fix all these different types of vulnerabilities (not the actual instances of vulnerabilities). There were 28 instances of buffer overflow vulnerabilities affecting software from 22 different vendors. We could not analyze 9 instances that affect proprietary software. 17 of the remaining 19 instances can be solved by applying the *Safe Library Replacement* transformation. In the remaining two cases, buffer overflow vulnerabilities originate from direct manipulation of pointers. Also, there were 34 SQL injection and 21 cross-site scripting attack instances that could be solved by *Decorated Filter* transformation. Other problems originating from unvalidated inputs (injection attacks) and improper error handling could also be solved by transformations in our catalog [14]. The authentication and authorization related transformations in the catalog could be used to add new components, or strengthen existing components, that could eradicate most of the problems originating from broken authentication and broken access control.

Another aspect is that not all transformations in our catalog has a one-to-one relationship with vulnerabilities. Many security problems have their roots in bad authentication and access control, but they manifest in other ways. Transformations to add an authentication, authorization component, or a partitioning transformation does not solve any one problem; they prevent other vulnerabilities from occurring. Hence the protection provided by our catalog is stronger than that suggested by the coverage figures in the previous paragraph.

6.2 Usefulness

In order to evaluate how often a security solution is used, or which transformations are more useful than others, one has to build tools for all of them. Then again, the tools will be platform specific and programming language specific, factors that might impact their usability. We believe that it is unreasonable to expect researchers to build tools for every platform. Building a tool for a single application is usually not cost effective, so it is unreasonable to expect application developers to do it either. It will need to be done by platform and tool vendors, who can amortize the cost over many users. Only that would provide a useful framework on which a usefulness study could be done.

Two studies of the way refactoring tools are used [26] [25] report that the most popular refactorings are *Rename*, *Extract Local Variable*, *Inline*, *Extract Method*, and *Move*. Of these, *Rename*, a very simple refactoring, is by far the most popular. Note that all the cited refactorings are fairly small. If these results can predict what are likely to be the most popular security-oriented program transformations, they will probably be the simplest transformations, from the *Small* impact category.

7 Related work

Security-oriented program transformations bridge two domains: security patterns and tools for automating software redesign. Throughout this paper, we have been comparing our program transformations and the ways they can be automated to refactoring [10]. Refactoring is a technique for improving the quality of design and code by making changes that do not affect the observable behavior of the system. This property, called behavior preservation, is not a part of security-oriented program transformations, because our goal is to *improve* the external behavior of the system by making it more secure. But in all the other aspects, our work parallels that of refactoring.

Our work is based on the continuously growing body of security patterns [40] [4] [32] [30] [17] that document best practices in making systems secure. Today, security patterns are not as easy to use as other software patterns, because they do not offer precise advice. The problems solved with security patterns are more complex than other patterns. In order to provide general solutions, security patterns are abstract and offer only general guidance that can be turned into complete solutions only by experienced security engineers.

Our work is related to software patterns in general, not just security patterns. Our transformations also incorporate patterns from the fault tolerance domain [16]. The relationship between security-oriented program transformations and security patterns is of the same type as between the *Refactoring to patterns* [20] book and the design patterns [11]. *Refactoring to patterns* describes, step by step, how to introduce design patterns into the code by applying a refactoring or a code edit at each step.

As a technique for describing how to modify software design and code, security-oriented program transformations are similar to other software docu-

mentation techniques. Design fragments [9] is a technique for describing how to use a software framework. It identifies the constraints that a framework user needs to meet in order to use it. These specifications are stored in the source code, as annotations.

Possibly the easiest way to implement some of our transformations in tools is to use aspect-orientation [21]. Logging and weaving in method calls to new components are the poster children of AOP, so our transformations such as *Secure Logger*, and most of the *Add ** transformations can be implemented with AOP relatively easily. However, there are other transformations for which AOP does not offer immediate solutions, e.g. the transformations in the *Distribute* category. Simple scripts or transformations based on abstract syntax tree manipulation are other implementation alternatives for such tools.

8 Conclusion

Security-oriented program transformations show how to think about security systematically. Since new kinds of security threats continue to appear, even systems that are currently being built to high security standards will eventually need to be changed, and so will need security-oriented program transformations. It is usually easier to solve a particular problem than to come up with a general solution to all problems of a particular class. However, there are many fewer general security solutions than there are security flaws. Applying security solutions defined as transformations will be cheaper in the long run than fixing every security flaw individually.

Our catalog will grow – in response to new classes of vulnerabilities – and pro-actively, as more people use it and begin to look for missing pieces. The next important step in exploring security-oriented program transformations will be to study how different transformations are related, for instance, by applying them in sequences, as we illustrated in the Example section.

Security-oriented program transformations are ready to be used in practice. They are a key to making and keeping our systems secure.

References

1. J. P. Anderson. Computer security technology planning study. Technical report, ESD-TR-73-51, Oct 1972.
2. A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack-smashing attacks. In *2000 USENIX Annual Technical Conference: San Diego, CA, USA*, 2000.
3. K. Beznosov and P. Kruchten. Towards agile security assurance. In C. Hempelmann and V. Raskin, editors, *NSPW*, pages 47–54. ACM, 2004.
4. B. Blakley and C. Heath. Security design patterns technical guide - version 1. *Open Group (OG)*, led by Bob Blakley and Craig Heath. 2004.
5. Bugtraq Vulnerabilities List. http://www.securityfocus.com/vulnerabilities.
6. S. Christey and R. Martin. Vulnerability type distributions is cve, May 2007.
7. A. Eden and R. Kazman. Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software Engineering (ICSE-03)*, pages 149–159, Piscataway, NJ, May 3–10 2003. IEEE Computer Society.
8. F. Cavalier III. Libmib allocated string functions. `http://www.mibsoftware.com/libmib/astring/`.
9. G. Fairbanks, W. Scherlis, and D. Garlan. Design fragments make using frameworks easier. In *Proceedings of ACM SIGPLAN Conference on Object Oriented Programs, Systems, Languages, and Applications (OOPSLA) 2006*, Portland, OR, USA, 22-27 October 2006.

10. M. Fowler. *Refactoring: Improving The Design of Existing Code*. Addison-Wesley, Jun 1999.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
12. M. Hafiz. A collection of privacy design patterns. In Proceedings of the 13th Conference on Patterns Language of Programming (PLoP'06), 2006.
13. M. Hafiz, P. Adamczyk, and R. Johnson. Organizing security patterns. *IEEE Software*, 24(4):52–60, Jul/Aug 2007.
14. M. Hafiz, P. Adamczyk, and R. Johnson. Systematically eradicating data injection attacks using security-oriented program transformations. In *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS-09)*, Feb. 4–6 2009.
15. M. Hafiz and R. E. Johnson. Evolution of the MTA architecture: The impact of security. *Software—Practice and Experience*, 38(15):1569–1599, Dec 2008.
16. R. Hanmer. *Patterns For Fault Tolerant Software*. Wiley, 2007.
17. J. Hogg, D. Smith, F. Chong, D. Taylor, L. Wall, and P. Slater. *Web Service Security: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0*. Microsoft Press, March 2006.
18. G. Hunt and M. Scott. The coign automatic distributed partitioning system. In *Proceedings of the 3rd Symposium on Operating Systans Design and Implementation (OSDI-99)*, pages 187–200, Berkeley, CA, Feb 1999. Usenix Association.
19. International Organization for Standardization. *ISO/IEC 24731: Specification For Secure C Library Functions*. 2004.
20. J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.
21. G. Kiczales, J. Lamping, A. Mendhekar, C. L. Chris Maeda, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programmin (ECOOP) 1997*, pages 220–242, 1997.
22. D. Kienzle, M. Elder, D. Tyree, and J. Edwards-Hewitt. Security patterns repository version 1.0. `http://www.scrypt.net/~celer/securitypatterns/repository.pdf`, 2002.
23. G. McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.
24. T. Miller and T. de Raadt. `strlcpy` and `strlcat` — Consistent, safe, string copy and concatenation. In *1999 Usenix Annual Technical Conference, Monterey, California, USA*, 1999.
25. G. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 2006.
26. E. Murphy-Hill, C. Parnin, and A. Black. How We Refactor, and How We Know It. In *To appear ICSE 2009*.
27. A. Narayanan. Design of a safe string library for C. *Software—Practice and Experience*, 24(6):565–578, 1994.
28. OWASP. OWASP Top Ten Project, 2008.
29. SANS Institute. SANS top-20 2007 security risks, Nov 2007.
30. M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley and Sons, December 2005.
31. A. Spiegel. Pangaea: An automatic distribution front-end for Java. *Lecture Notes in Computer Science*, 1586:93–99, 1999.
32. C. Steel, R. Nagappan, and R. Lai. *Core Security Patterns : Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management*. Prentice Hall PTR, Oct 2005.
33. A. v. Stock, J. Williams, and D. Wichers. OWASP Top 10 - The ten most critical web application security vulnerabilities - 2007 update, 2007.
34. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of "legacy" Java software. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 236–255, London, UK, 2001. Springer-Verlag.
35. E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic Java application partitioning. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 178–204, London, UK, 2002. Springer-Verlag.
36. US-CERT. IBM Internet security systems X-Force 2007 trend statistics, Jan 2008.
37. US-CERT. Vulnerability notes by severity metric, Apr 2008.
38. J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems The Right Way*. Addison-Wesley, 2002.
39. L. Wang and M. Franz. Automatic partitioning of object-oriented programs with multiple distribution objectives. Technical Report 07-11, University of California, Irvine, Oct 2007.
40. J. Yoder and J. Barcalow. Architectural patterns for enabling application security. In Proceedings of the 4th Conference on Patterns Language of Programming (PLoP'97)., 1997.

Appendix. List of Transformations

Each transformation is summarized in three sentences.

1. The first sentence summarizes the before and after of the transformation and provides a reference to the source that first described this security solution. There are some program transformations that do not originate from security patterns; some of these are program transformations that introduce a well-known security feature (e.g. encryption, message digest etc.) while others are new program transformations. These do not have any references.
2. The second sentence describes what a developer needs to specify (the policy).
3. The last sentence summarizes the structural changes that are made by the tool that applies the transformation (the mechanism).

The transformations are listed in alphabetical order.

T1. Add Account Lockout. Introduce the functionality to limit the number of unsuccessful password entry attempts [22]. A developer specifies the authentication component where the lockout policy is enforced. The transformation inserts the checks which may be as simple as inserting a for loop with conditional clause.

T2. Add Audit Interceptor. Introduce the functionality for collecting and manipulating audit events [32]. A developer specifies where to intercept data to audit, and how to create audit data. The transformation adds an audit interceptor component with appropriate logic to create audit data, and delegates all requests and responses through the component.

T3. Add Authentication Enforcer. Introduce the functionality for authentication based on various authentication mechanism, such as username and password, username and certificate, or kerberos tickets [32] [30] [17]. A developer specifies the insertion point of an authentication component, source of input data, authentication mechanism, and outcome of a successful or a failed authentication. The transformation composes classes of a security framework (JAAS, .NET etc) to create a component with the specified authentication mechanism, and delegates authentication requests from the insertion point.

T4. Add Authorization Enforcer. Introduce the functionality of an access controller [32] [30]. A developer specifies the insertion point of an authorization component, the subject of authorization passed to the component, and the authorization policy. The transformation composes classes of a security framework (JAAS, .NET etc) to create an authorization enforcer component, and delegates access control requests from the insertion point.

T5. Add Error Message Suppressor. Introduce a component that intercepts all error messages and suppresses the error messages based on some global policy. A developer specifies the interception point of error messages (e.g. intercepting all write messages to stderr) and the policies applied to suppress error messages. The transformation adds an interceptor component that applies the suppression policies.

T6. Add Password Synchronizer. Introduce a component for synchronizing user credentials across different application systems [32]. A developer identifies the systems that use a password synchronization service. The transformations creates the service as a hub between all the components.

T7. Add Perimeter Filter. Introduce a policy enforcement point at the system entry point to validate data [32]. A developer specifies where to insert a perimeter filter and what policies to apply. The program transformation adds a new perimeter filter component that applies policies to validate data and delegates all input validation requests to the component.

T8. Add Reference Monitor. Introduce a process that intercepts all requests for resources and validates access to them [30]. A developer specifies the requests to intercept and access control policies. The program transformation adds a reference monitor component that applies access control policies.

T9. Add Replicated System. Replicate services at multiple points in the network [4]. A developer identifies the network points where replication is required and describes the workload balancing policy. The transformation creates replicas at the network points and creates a workload management proxy at those points; the proxy switches to another service when a service fails.

T10. Add Standby. Introduce backup components [4]. A developer specifies components for which standby components are required and points out what state information of the original component are to be kept. The transformation creates replicas and adds a monitor that transfers control to a standby component when a component fails.

T11. Batched Tasks. Group similar requests and perform them together rather than processing each request as soon as it arrives [15]. A developer specifies the program point that receives user requests. The transformation creates a request pool to process requests in a batch.

T12. Checkpointed System. Periodically save important state information so that a system can restart gracefully from a crash [4]. A developer specifies where the system checkpoints are and the state information to save at checkpoints. The transformation introduces method calls to save information.

T13. chroot Jail. Constrain processes in a system to access only a portion of the filesystem [15]. A developer specifies: (1) the location of the chroot directory, (2), the program to be jailed and the programs that communicate with it, (3) the line number of the program where the `chroot` system call is to be inserted, and (4) the privilege level of the writer and the reader process. The program transformation creates a jail environment for a process and runs it inside a chroot jail.

T14. Controlled Object Factory. Intercept new object creation requests and explicitly assign the rights to the new object [30]. A developer specifies the rights to be associated with the critical objects in a system. The transformation adds right checks at the object creation points.

T15. Controlled Process Creation. Rather than having a child process automatically inherit all parent's privileges, have the system assign to the child process only the explicitly stated privileges [30]. A developer specifies the parent

and child processes and the privileges to transfer. The transformation intercepts a process creation system call and assigns the privileges to the child process.

T16. Credential Tokenizer. Introduce functionality for encapsulating different types of user credentials as a security token [32]. A developer specifies the types of tokens to use. The transformation creates a tokenizer that creates and manages the security token.

T17. Decorated Filter. Associate additional checks with input variables and apply them as a series of filters. A developer specifies the target input variable and the policies to be applied. The program transformation adds the policies to an input variable by using the Decorator [11] pattern.

T18. Encryption/Decryption Encrypt or decrypt a message signature. A developer specifies the input data buffer, the parameters of the signature algorithm, the key used to encrypt/decrypt a message, and the key encoding algorithm. The transformation composes classes from a security framework (JCA, JCE, .NET etc) to encrypt/decrypt the message.

T19. Exception Shielding. Introduce exceptions to indicate that the application entered an unexpected state and at the same time obfuscate the error messages to hide internal information [17]. A developer specifies exception type, insertion point and data obfuscation policy. The transformation inserts exceptions, as needed, and obfuscates the error messages produced by the exceptions so that exceptions do not contain sensitive information or a detailed stack trace.

T20. Fuzzing. Extend the testing framework of the system with a component that generates random data and runs the program with that data. A developer specifies the program insertion points where fuzzing tests are to be applied. The transformation automatically adds the components in the testing framework.

T21. Guarded Object If the supplier and the consumer of a resource are in different threads, make the context a part of the resource so that access control decisions could be taken locally. A developer specifies a resource to be protected and the permissions associated with the resource. The transformation creates a guarded object that encapsulates a resource and the permissions associated with the resource.

T22. Least Privilege. Analyze a program to make sure that it runs with the lowest privilege required to perform all its tasks [38]. No developer action is necessary. The transformation automatically analyzes a program and identifies the lowest privilege level required; then it lowers the privilege level of the program.

T23. Message Caching. Add a verification mechanism for dropping replayed packets [17]. A developer specifies where at the program perimeter incoming messages are accepted as well as how to uniquely identify incoming messages. The transformation implements a message cache with replay detection mechanism; it drops all the packets that have been replayed.

T24. Message Digest Creation. Calculate the message digest of a data to prove message integrity. A developer specifies the input data buffer, and the

message digest algorithm to be used. The transformation composes classes from a security framework (JCA, JCE, .NET etc) to calculate message digest.

T25. Message Intercepting Gateway. Introduce functionality that sanitizes the XML input before it enters the system [32]. No developer action is necessary. The transformation creates an XML firewall in front of all the access points in the system and sanitizes XML data; it's a combination of *Add Perimeter Filter*, *Decorated Filter* and *Intercepting Web Agent* transformations.

T26. Parallelized Functionality. Split functionality of a single component among different components that will perform it in parallel [4]. A developer specifies the critical functions to parallelize, and the interfaces. The transformation extracts the functions into a component and creates parallel components.

T27. Parallelized Link. Rather than sending data to a single recipient over a single link, send it over multiple links [12]. A developer identifies the outgoing links of a system. The transformation creates parallel links for each outgoing link and distributes traffic evenly (or randomly) between links.

T28. Partitioning. Break up a monolithic system into multiple security domains [38]. A developer specifies how functions are distributed among partitions, what the partition interfaces are, how the partitions communicate, and what privilege levels they have. The transformation distributes the functions based on the specification, and modifies the privilege levels according to the specification.

T29. Policy Enforcement Point. Introduce a component that centralizes the application of authentication, authorization and input validation at the system entry point [40] [32]. A developer specifies the system entry point and the authentication, authorization and input validation policies. The transformation creates a policy enforcement component and components for authentication, authorization and input validation. It delegates incoming requests to secure base action component.

T30. Randomization. Make off-the-shelf components of the system unique so that they do not fall prey of common vulnerabilities [22]. A developer specifies an artifact of the program (e.g. a data variable, instruction set) to be made distinct. The transformation randomizes the artifact to make the system distinct and thus less vulnerable.

T31. Resource Management. Introduce resource limit checks into source code [15]. A developer identifies the places where resource checks are to be inserted and the policies to apply. The transformation adds checks to the program points.

T32. Safe Library Replacement. Replace unsafe functions in the code with their safe versions [15]. For each unsafe function, a developer specifies the alternative safe function and the library that includes the function. The program transformation finds all functions that need to be replaced, replaces unsafe functions with suitable alternatives in all source files, and adds information about the new library to configuration files so that the new program compiles.

T33. Secure Logger. Introduce logging functionality into the system [32]. A developer specifies the messages to log, and policies to retain confidentiality

and integrity. The transformation adds a logging component that securely stores logged data.

T34. Secure Message Router. In a distributed system, add intermediary infrastructure that ensures that routing is done securely and each endpoint has access only to the fragment of a message it's authorized to examine [32]. A developer specifies security decisions and mechanisms for each endpoint. The transformation creates a group of intermediaries that follow a protocol such that they can route using selected portions of the message. At the sender end, the transformation creates a policy enforcement point for outgoing messages.

T35. Secure Pipe. Introduce functionality to ensure the integrity and privacy of data sent over the wire [32]. A developer specifies the content to be transferred with a secure pipe and the policies to be used to initiate a secure pipe. The transformation creates the pipe.

T36. Secure Resource Pooling. Introduce limits to the lifetime of daemon processes [15]. A developer specifies the parameters that determine the lifetime of worker processes in a resource pool, e.g. the maximum time that a process remains idle, the maximum number of requests that a process serves, or the maximum life limit of a process since its creation. The transformation creates a monitor process for the resource pool that kills processes in the pool and replenishes it with new processes.

T37. Secure Session Object. Introduce a session object containing authentication and authorization credentials that is passed across system boundaries [32]. A developer specifies what to put inside a session object. The transformation creates and manages the object.

T38. Signature Generation Generate a message signature. A developer specifies the input data buffer, the parameters of the signature algorithm, the key used to sign a message, and the key encoding algorithm. The transformation composes classes from a security framework (JCA, JCE, .NET etc) to generate message signature.

T39. Single Access Point. Consolidate multiple locations where data enters the system into as few access points as possible [40] [30]. A developer specifies the access points. The transformation makes an object-oriented Façade [11], or introduces a wrapper component that has the same API.

T40. Single Sign On Delegate. Introduce a new component that performs identity management and Single Sign On functionality [32]. A developer might choose services. In that case, the transformation implements the services. Alternatively, the transformation automatically creates an SSO delegate with default services.

T41. Single Threaded Façade. Make a multi-threaded perimeter process single-treaded [15]. A developer specifies the threads to remove. The transformation merges multiple threads of control into a single thread.

T42. Unique Location for Each Write Request. Create a unique file for storing each request [15]. A developer specifies the section of a program that writes to a shared file and new file creation policy. The transformation modifies the write request so that a new file is created for each write request.