CHALMERS
UNIVERSITY OF TECHNOLOGY

SecArchUnit: Extending ArchUnit to support validation of security architectural constraints

Master's Thesis in Software Engineering

MARCUS RANDEVIK
PATRIK OLSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020:NN

SecArchUnit: Extending ArchUnit to support validation of security architectural constraints

MARCUS RANDEVIK
PATRIK OLSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
Gothenburg, Sweden 2020

SecArchUnit: Extending ArchUnit to support validation of security architectural constraints
MARCUS RANDEVIK
PATRIK OLSON

Supervisor: Riccardo Scandariato, Computer Science and Engineering
Examiner: Michel Chaudron, Computer Science and Engineering

Master's Thesis 2020:NN
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Todo list

An Informative Headline describing the Content of the Report
A Subtitle that can be Very Much Longer if Necessary
MARCUS RANDEVIK
PATRIK OLSON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Keywords: lorem, ipsum, dolor, sit, amet, consectetur, adipisicing, elit, sed, do.

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Name Familyname, Gothenburg, Month Year

Contents

List of Figures

List of Tables

1

Introduction

In the age of an ever more digitalized world, ensuring the security of software systems becomes an increasingly more critical task. The increased severeness is true in particular, as systems are being developed based on the principle of permanent connectivity [1]. Services ranging from those offered by governmental agencies to that of social media are always connected to the open internet, potentially creating a large surface of attack. Although the importance of a securely designed system is widely known, developing a secure system is a challenging task as far from all software engineers are security specialists, or for that matter, particularly educated in security aspects [2].

Introducing a weakness into a system can be done in every part of the software development lifecycle. An early introduction often leads to a more costly fix as the scope of the flaw is increased [3]. Typically, weaknesses are categorized as either implementation bugs (e.g. buffer overflow) or design flaws (e.g. client-side validation) where the later commonly has a more extensive scope and is the responsibility of the architect [4]. However, this binary division might be too simplified as it assumes a secure design is implemented correctly, meaning that there is no discrepancy between the "intended architecture" and the actual one found in the source code. Jasser [5] considers violations of the architectural security design as a third type of category with a comparable impact to that of design flaws.

Deviations from the intended architecture are not unique to the initial implementation of a system. Over time, subsequent changes made to the system, often due to new requirements, frequently lead to further disparency called software erosion [6]. Proponents of the agile methodology also somewhat worsen the problem of erosion as the reduction of upfront design severely hinders the design of the architecture [7]. Many of the tools and techniques produced by academia to try and remedy the frequent violations of architectural design have failed to gain wide adoption in the industry [6]. Of the few that have, architectural design documentation is the most prominent and widely included in numerous software processes. While the technique is well adopted, performing it formally to allow for automatic compliance monitoring is seldom done, causing developers to rely on the less scalable method of manual reviews [6].

A recently developed tool called ArchUnit[1] has provided architects with the capability of expressing architectural constraints trough Java unit testing frameworks. While ArchUnit is not the first tool developed for architectural conformance checking, it is possibly the first to leverage already existing testing infrastructure (Unit testing) in a manner that allows for testing over time. In its current form, ArchUnit has not provided any explicit functionality to test security constraints. Thus, the objective of this thesis is to explore the feasibility of expressing and enforcing typical security architectural constraints trough ArchUnit.

1.1 Scope

This study focuses on architectural measures that aid in fulfilling a security goal. Moreover, these measures should be possible to enforce through static analysis of Java bytecode. Any measure related to the configuration (of an application or operating system), the file-system, or other run-time properties that cannot be validated through static code analysis is deemed to be out of scope.

1.2 Research questions

To fulfill the objective of the study, the following research questions have been defined:

- RQ1: What architectural security constraints can be validated using the tool?
- RQ2: What modifications can be made to the tool in order to facilitate the validation of additional constraints?

1.3 Research contribution

This study aims to show how architectural security constraints can be validated with the help of a static analysis tool. The thesis demonstrates applications of the tool to several open source systems, over time, in an evaluation of its efficacy and precision in terms of detecting violations of constraints.

1.4 Limitations

This thesis, and the modified version of ArchUnit, is not aimed at being a complete solution for all security architectural constraints. Instead, the study is performed to provide an initial evaluation of the possibility of using ArchUnit as an alternative to already existing techniques of static conformance checking.

Although the principle of ArchUnit may very well apply to programming languages

[1] https://www.archunit.org/

other than Java, the limited scope of the thesis makes it unfeasible to provide functionality to analyze source-code in additional languages.

1.5 Thesis outline

The remainder of this report is structured as follows.

Outline chapters

Chapter 2: Provides a general background to the topic of software architecture and architectural security constraints. In addition, the ArchUnit framework is introduced and compared to previously developed tools for architectural conformance.

Chapter 3: design of our evaluation, how systems are chosen, the protocol for testing the tool, how to assert the ground truth, process of mapping constraints to rules

Chapter 4: composing the architectural constraints, our final constraints, what they prevent and how they are expressed

Chapter 5: expressing and enforcing constraints with the tool, the identification of missing information / tool features

Chapter 6: results from the empirical validation

Chapter 6: discussion, answer our research questions

Chapter 7: conclusion

2

Background

This chapter...

2.1 Software architecture

[8], [9], [10], [11], [12],

2.2 Architectural security constraints

[13], [1], [14],

Security goals, CIAA...

2.3 ArchUnit

Overview of functionality

ASM, class/method visitors

Explain concept of code units, members

2.4 Architectural conformance monitoring

[15], [16], [17], [18], [6], [19], [20], [21],

3

Methodology

This chapter describes the adopted method for collecting relevant constraints, relating these to the common security-goals of CIAA, and later mapping them to functionality within ArchUnit. Second, this chapter presents the validation plan for expressing security constraints with ArchUnit (as is) by means of an illustration and for expressing additional constraints by means of a controlled experiment.

3.1 Data collection

The relevance of the security architectural constraints included in the study was ensured by performing a review of security measures and common weaknesses and compiling the result to a list of constraints. Completeness was not the primary goal of the review, but rather to provide a set of constraints derived from previous knowledge. Presented below are the three sources used to form the final list.

CAWE catalog: The Common Architectural Weakness Enumeration catalog [22] details 224 common weaknesses in security architectures. Each entry has a description of the weakness and exemplifications of how it could manifest itself in the source code, when applicable. In some entries, there are recommendations on what techniques can be used to detect the weakness, along with mitigation strategies.

Security patterns: Similar to the usage of general design patterns made famous in [23], security patterns provide a reusable and domain-independent solution to a known problem. More specifically, this study focused on security patterns for the design phase, as defined in [2]. While the security pattern repository[1] lists over 170 security patterns, not all are provided with sufficient detail or at the appropriate level of abstraction. As a result, the report by Scandariato et al. [24] which provides a filtered list of patterns.

Security rules: Architectural security rules constrain the implementation of a system while being less solution-oriented compared to security patterns. Eden and Kazman differentiate architectural security rules from those defined on a level of source code based on two criteria, locality and intension/extension [25]. Architectural rules are both non-local and intensional, meaning that they affect all or several parts of the system while having "infinitely-many possible instances". In [5], Jasser

[1] http://sefm.cs.utsa.edu/repository/

presents a catalog of architectural security rules. Although the entire catalog of 150 security rules is not yet available, the initial list of 22 included in the paper was used in our study.

3.2 Filtering

Starting from each of the three sources of architectural constraints described in Section 3.1, the first step of the process, shown in Figure 3.1, was to select the entries that could be formulated as enforcable constraint in the context of our project. The criteria for inclusion were the following:

1. The entry must be related to the architectural design of a system, i.e., non-local and intensional (as described in section 3.1.

2. It must be possible to enforce the entry through static analysis. An example on a non enforceable constraint is "No two instances of a microservice are deployed on the same machine", as the number of machines deployed is a dynamic property.

3. Although somewhat included in the first criterion as it is a local issue, an entry must not relate to the correctness or best practice of the implementation of an algorithm. Examples include the practice of using session tokens with time-limited validity.

4. The entry must only relate to the system under design, thus ignoring the correctness and security of any external dependencies. An example can be found in SonarQube where the usage of a version of a library with known vulnerabilities is reported as a weakness.

5. The entry must not be dependent on externally defined data. A common example is that of user permissions where the mapping between a regulated functions and a users rights is performed using an access control list.

6. Additionally, we deemed measures defined as the absence of certain functionality as less valuable due to the increased difficulty of enforcement [14].

Previous research on design notations of secure systems have shown a skew towards confidentiality and integrity while having little or no support for availability and accountability. We considered it necessary for the final list of constraints to include all of the security goals. As a consequence, once we had selected the applicable entries, we categorized them according to the security goals of CIAA, ensuring that the final list of constraints covered all security goals.

The last part of compiling a list of security architectural constraints involved combining the selected entries to remove duplicates and group similar concepts. Duplication involved both a single source having several entries, such as CAWE having input validation weakness for multiple tools and technologies (e.g., SQL, LDAP) and different sources having entries for the same concept, such as the security pattern

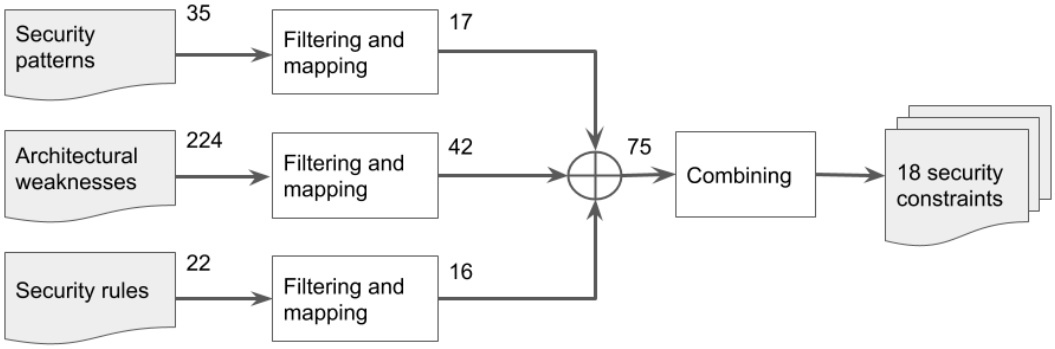

Figure 3.1: Overview of the process of mapping the three sources to constraints

input guard and the previously mentioned input validation weaknesses. Grouping similar concepts also allowed for the constraints to be more general, thus making them applicable for a broader set of systems.

3.3 Validation

The validation of our results will be performed in two ways, depending on whether or not a constraint required any modification of ArchUnit or additional information in the source-code. The latter category carries a higher degree of scientific value thus motivating a more thorough validation procedure. In the sections below, both types of procedures will be described.

3.3.1 Solution Proposal

As described in Section 2.3, ArchUnit already provides functionality to perform conformance testing of architectural constraints. However, it is unclear whether the framework supports the enforcement of security architectural constraints. The mapping of security architectural constraints to that of rules in ArchUnit allows us to perform a *proposal of solution* as described in [26]. This type of validation is intended to propose a novel or significantly improved technique without rigorous validation. Instead, a proof of concept or small example is used to facilitate later validation.

3.3.2 Controlled Experiment

In contrast to constraints which may be implemented through already existing functionality, those requiring extension to either the API or additional information within the source-code are not guaranteed to reliably detect violations of the intended architecture. Therefore, validation is performed using a laboratory experiment [27] in order to increase the precision of the measurement.

Integration with already existing testing frameworks is a prominent advantage of ArchUnit. As testing is generally performed overtime to ensure that a system does not degrade, the focus of the experiment is to determine whether ArchUnit can detect changes to security architectural constraints between two versions.

3.3.2.1 Selection of Systems

describe projects, why they were selected

Three systems have been selected for the evaluation. These systems are as follows:

- **ATM simulator**,
- **JPetStore**,
- **iTrust**,

3.3.2.2 Performance metrics

The imbalance between the designer, who needs to ensure that every single aspect of a system is secure, and the attacker, who needs to succeed only once, influences the metrics chosen to represent how well the extension to ArchUnit performs. Precision and recall were the metrics of choice, with the greater importance placed on the latter.

Define true positives, false positives, false negatives

3.3.2.3 Tools used in comparison

fix

In order to test how reliably SecArchUnit can validate the constraints, we perform a comparison with two static analysis tools widely adopted in industry: SonarQube and PMD. While these tools have a multitude of built-in rules, none of these rules can be used to enforce the architectural security constraints presented in this thesis. However, both tools are extensible, allowing a developer to define custom rules using their respective APIs. In an initial assessment of the tools, it was determined that constraints one through five are possible to implement and validate using SonarQube and PMD. Hence, the first five constraints in evaluated SecArchUnit, SonarQube and PMD, whereas the final two constraints are evaluated solely in SecArchUnit.

3.3.2.4 Projects used in evaluation

Systems to be included in the validation needed to fulfill two mandatory criteria. First and foremost was the fact that they need open source and written in Java as the static analysis of ArchUnit relies on the source-code. Secondly, there needed

to be at minimum two different snapshots in order to fulfill the goal of comparing subsequent changes to a system.

In addition to the mandatory criteria, other aspects were also considered to reduce bias in the validation process. Systems which had already been analyzed in previous literature [28, 18] would provide an existing architecture and its security analysis, which we leverage as ground truth in our experiment. Additionally, systems which have a well documented architecture and security requirements and were within a reasonable size can further help mitigate potential internal validity threats.

4

Selection of Architectural Security Constraints

This chapter describes the result of compiling a list of security constraints, as described in the previous chapter, and the final selection of constraints used in the validation of the tool.

4.1 Compiled List of Security Constraints

The filtered collection of security constraints can be seen in Table 4.1. There are in total 18 constraints. As mentioned in Section 3.2, each constraint was categorized according to the goals of CIAA to ensure full coverage.

Although both the architectural rules found in Jasser [5] and the security patterns presented in Scandariato et al. [24] were at the appropriate level of design, many of the weaknesses presented in CAWE were not. Common examples include; CAWE-259 "Use of hard-coded password" where the weakness is reliant on a local change of behavior rather than the architectural structure; and CAWE-263 "Password aging with long expiration" where the weakness is introduced by a single variable most likely defined outside of the source code. As a result, a far lower percentage of entries were included from the CAWE-catalog compared to the other sources.

4.2 Final Selection

As explained in Section 1.4, the aim is not to demonstrate the enforceability of as many constraints as possible, but rather to investigate the feasibility of using the tool in this manner. To that end, a subset of the full list of security constraints is selected for enforcement. The final list contains 7 architectural security constraints, as this allows us to cover at least one constraint from each goal in the CIAA model. The selected constraints can be seen in Table 4.2. The remainder of this section presents each selected constraint in further detail.

ID	Constraint	Goal
1	Exceptions shown to the client must be sent to a sanitizer	Confidentiality
2	Sensitive information must not bleed to components with lower security classification	Confidentiality
3	Sensitive information must be encrypted before transmission	Confidentiality
4	Every outbound message must be sent from a single component responsible for transmissions	Confidentiality
5	Data that passes a trust boundary must first be sent to a component responsible for hiding or removing sensitive data	Confidentiality
6	Secrets must not be exposed in log messages	Confidentiality
7	The system must not provide functionality to decrypt secured log messages	Confidentiality
8	Output passing between components must be validated against its specification	Integrity
9	Input from a user must pass through a component validating the data	Integrity
10	The session object must not be accessible to the user	Integrity
11	Components must store its state as restorable checkpoints	Availability
12	Spawning of threads must be limited or throttled	Availability
13	The system must not have multiple points of access	Accountability
14	At least one checkpoint must be initialized after successful authentication and authorization	Accountability
15	Methods related to security events must call the logger	Accountability
16	Authentication and authorization must each be enforced in a single component	Accountability
17	Security relevant log messages must be encrypted and immutable	Accountability

Table 4.1: Security constraints and their related CIAA goals.

#	$\#_{4.1}$	Constraint
1	15	Methods related to security events must call the logger
2	16	Authentication and authorization must each be enforced in a single component
3	4	Every outbound message must be sent from a single component responsible for transmissions
4	9	Input from a user must pass through a component validating the data
5	12	Spawning of threads must be limited or throttled
6	2	Sensitive information must not bleed to components with lower security classification
7	6	Secrets must not be exposed in log messages

Table 4.2: Constraints that have been selected for enforcement. Column $\#_{4.1}$ refers to the ID of the constraint in Table 4.1.

4.2.1 Log all security events

Description: In any system, several components either directly change or process data, which represents the system's asset, or indirectly by invoking other components to act on its behalf. In either case, the request to perform a particular action originates from an actor (user or external process) who should later be held accountable. As a consequence, the system should log a security event before performing an action that could breach the specified security policies. Although the term security event has become somewhat ambiguous, the definition used in the context of this report comes from the SANS Institute: "An event is an observable occurrence in an information system that actually happened at some point in time." [1]

Typical enforcement: The usage of the *audit interceptor* forces all requests from a user to first be sent to a component responsible for logging the request and later forwarding it to the intended target.

Sources: CAWE 223/778, Jasser rule 5, Security pattern *Audit interceptor*.

Attack scenario: A typical scenario where the logging of security events increases a system's resilience to attacks is that of failed login attempts. An attacker may try and guess the credentials of a user by employing a brute-force attack. During the attack, the attacker performs several failed attempts at guessing the credentials, (hopefully) causing the system to either increase the time between repeated attempts or lock the account entirely though with the added effect of decreased availability for the intended user. Although this type of defense temporarily hinders the attacker, a log of failed attempts facilities the detection of malicious actors and enables administrators to impose more permanent measures.

4.2.2 Enforce AuthN/AuthZ at single point

Description: Any system that has more than one user needs to incorporate functionality for authentication (AuthN), as well as authorization (AuthZ) if the privileges between users differ. The difficulty in complex systems where components handle different functionality, thus receiving separate requests and creating multiple entry points, is the fact that the components may have been designed to use various mechanisms of authentication. Instead, AuthN/AuthZ should be delegated to a single component to ensure consistent behavior across all entry points.

Typical enforcement: Designing a single component responsible for AuthN/AuthZ mechanisms across several points of entry. Several third-party libraries exist that provide such features as well as language extending specifications such as Jakarta EE (formerly J2EE).

Sources: CAWE 288/420/592, Security pattern *Authentication enforcer* and *Authorization enforcer*

Attack scenario: In system where the following conditions are true:

[1] `https://www.sans.org/reading-room/whitepapers/incident/events-incidents-646`

- There are multiple points of entry;
- There are different mechanisms to provide AuthN/AuthZ, some having a greater certainty that a user is properly authenticated or authorized to perform an action
- and all points of entry share the same session object

An attacker may try and gain access to the least trusted point of entry and later use the granted authority to access services or operation normally requiring a greater level of trust.

4.2.3 Messages are sent from a central point

Description: Communication with external actors, whether they are a client connecting to a server, or the system sending data to a third party, is commonly performed over insecure networks using several components. Encryption is the preferred method of securing such communication against potential attackers, whereas removing secrets from the data to be sent ensures that a user only sees non-sensitive information. Having a single component responsible for all outbound communication reduces the risk of information disclosure (e.g. transmitting a sensitive message via an insecure network or disclosing implementation details through stack traces), and can prevent harmful output from reaching the client (e.g. cross-site scripting attacks from other users).

Typical enforcement: Outbound messages can be intercepted before transmission to facilitate output sanitization. Similarly, outgoing messages can be forced to pass through a single component, designated as the sending point. This sending point can handle sanitization and decide whether the sender is allowed to carry the specified message.

Sources: Jasser rules 11 and 12.

Attack scenario: A blog website may properly use a delegated component for the sanitized transmission of some data (e.g. blog entries) but fail to do so for others (e.g. comments). An attacker who posts a comment containing HTML tags may then hijack the browser session of other users visiting the site.

4.2.4 Validate user input

Description: The ability to receive and process user input is fundamental to every computer system. However, the same input is also the primary source of untrusted data as an attacker possesses full control of what the system receives. Assuming that all data passed to a system is safe to process can have severe consequences when interpreting user input as a part of a query, often referred to as injection. In order to prevent an attacker from compromising the system by injection, all user input must be validated.

Typical enforcement: Placing a component performing validation between the

user's input and the component processing the data ensures that the input can be trusted. The approach is commonly referred to as the security pattern *input guard*.

Sources: CAWE 20/59/74-79/88-91/93-99/138/150/349/352/472/473/502/601/ /641/643/652/790-797/942, Security pattern *input guard*.

Attack scenario: In an application that uses user input to build a SQL query to retrieve a specific account number (as seen in Listing 4.1) an attacker may construct the request to retrieve all accounts by adding characters that break the query and introduces new parameters, such as `' or '1'='1`. The resulting operation would retrieve all customer accounts, thus exposing sensitive information.

```
1 String query = "SELECT * FROM accounts WHERE
2      custID='" + userInput + "'";
```

Listing 4.1: Example of a vulnerable SQL query

4.2.5 Restrict thread spawning

Description: Computers have finite resources in terms of memory, CPU time and network bandwidth. Systems should be designed with this in mind, employing measures to avoid exhausting the computer's resources. This constraint limits the number of threads that can be spawned on behalf of actors, which could otherwise lead to a malicious actor occupying all of the available CPU time.

Typical enforcement: Tasks can be dispatched to a pool of worker threads that is not allowed to grow beyond a fixed size. Moreover, various mechanisms can be employed to throttle or limit requests such that a single actor cannot occupy all of the allotted threads.

Sources: CAWE 770.

Attack scenario: An attacker may initiate many requests that are each handled by the system in a separate thread. By initiating requests at a higher rate than the server is able to process them, the resources at the server are eventually exhausted. This leads to a denial of service for any legitimate actors attempting to access the system.

4.2.6 Sensitive information must stay within trust boundary

Description: Generally, a specific set of components, which have stricter security requirements constraining their implementation, handles the sensitive data within a system. Should that information leak to less secure components, the risk of exposing secrets to the user, and a potential attacker, increases significantly. In order to prevent leakage to less secure components, sensitive information must stay within a trust boundary.

Typical enforcement: A typical approach is to manually review methods that receives or send information to other components and ensure that they do not expose

any secrets. As for automated enforcement, various information flow analysis tools, like JOANA[2], can be employed to detect these types of information leaks within a system.

Sources: CAWE 488.

Attack scenario: An asset may be leaked from a component that is supposed to service a single actor, such as a session object, to a component that multiple actors have shared access to. This may subsequently lead to the asset being illegitimately accessed by a malicious actor.

4.2.7 Secrets must not be exposed in log messages

Description: Many systems handle secrets that should never touch permanent storage. A password is perhaps the most common example of such a secret. While great care can be taken on the design level to ensure that these secrets are not stored to disk, they may still be exposed unintentionally through log messages. In order to prevent such exposure, messages that are sent to the logger must not contain secrets.

Typical enforcement: Similar to the constraint described in Section 4.2.6, the typical approach is to manually review calls to the logger to ensure that no secrets are exposed, with the potential ability to use information flow analysis tools.

Sources: CAWE 359/532, Jasser rule 13.

Attack scenario: Log messages may be accessible to actors who are not otherwise granted direct access to the secrets. By exploiting an unintentional leak of secrets into the log messages, an attacker could systematically extract these without facing the intended restrictions.

[2] `https://pp.ipd.kit.edu/projects/joana/`

5

Enforcing Constraints

This chapter explains how the constraints are expressed and validated using the tool using both the general definition of a rule as well as by providing a modeled example system. The constraints are divided into three distinct categories. The first category contains the constraints that are possible to express in ArchUnit as-is. The second category describes constraints that are enforceable with the help of additional information in source code. The third and final category details constraints that require an extension of ArchUnit to be possible to enforce.

5.1 Example system

improve heading

The constraints used within the report are not system-specific but instead defined to by generally applicable. However, a toy system called *BlogWeb* has been modeled to allow us to showcase a specific example for each constraint. BlogWeb is by no means intended to be a complex or complete system, but rather big enough to provide the functionality to support the constraints. A UML class diagram of BlogWeb is illustrated in Figure 5.1.

BlogWeb is, as the name suggests, a blog website providing users with the ability to publish statuses to their blogs. Being a website means that the system receives actions to execute via HTTP requests. These requests are received by the RequestHandler to be later forwarded to the UserService, which provides public

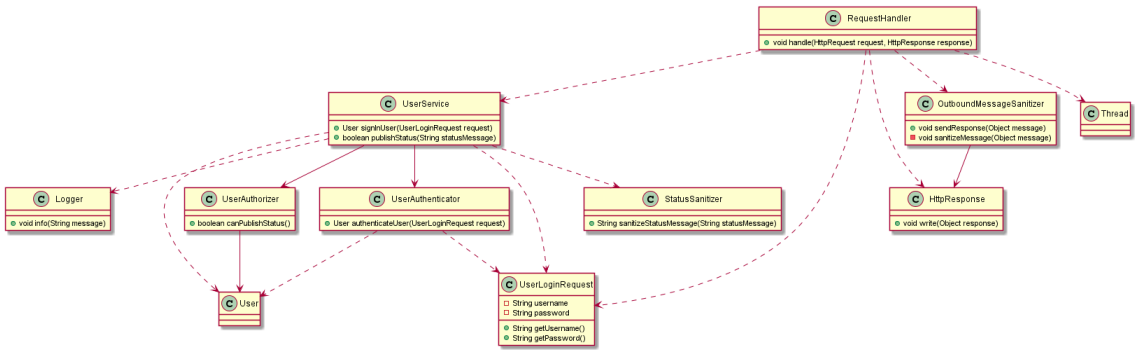

Figure 5.1: An example of a system, for the purpose of illustrating how the constraints are applied.

methods for sign in and publishing a status update. The UserService has a reference to both a UserAuthorizer and a UserAuthenticator implementing the logic for authorization and authentication. The UserService also uses the Logger to perform the necessary logging of user requests. StatusSanitizer is used to sanitize the status updates received by users before its stored while OutBoundMessageSanitizer is responsible for sanitizing requests responses. Finally, as the RequestHandler may serve several users at once, a thread is spawned for each request to make the execution concurrent.

5.2 Support in ArchUnit as-is

ArchUnit contains an extensive vocabulary for expressing typical architectural constraints. These constraints are generally composed of three parts: a construct, a predicate and a condition. The construct defines the type of Java construct that should be inspected, and includes classes, methods, fields and constructors. The predicate then selects a subset of these constructs, for which the condition must hold true.

An example of a rule defined solely using this standard vocabulary can be seen in Listing 5.1, where each of the three aforementioned parts of the constraint has been separated into their own line. The rule is a simple example of complete mediation, where some internal classes must only be accessed through a mediator.

```
1 ArchRule rule = classes()
2     .that().resideInAPackage("..internal..")
3     .should().onlyBeAccessed().byAnyPackage("..mediator..");
```

Listing 5.1: Example of a rule that is expressed with the standard vocabulary.

In situations where this vocabulary is insufficient for expressing a constraint, there is a possibility to define custom predicates and conditions over any given construct. These can be supplied as arguments to the `that()` and `should()` methods. Custom predicates and conditions are used extensively in our implementation, as will be made apparent in the following sections.

5.2.1 Log all security events

This constraint is expressed with the assumption that there are services, in the form of classes, that are responsible for performing security related events. Every publicly accessible method in such a service is assumed to perform a security event and must therefore contain a call to the logging facility.

The definition of the architectural rule can be seen in Listing 5.2. The predicate that selects the security services, and the class that is responsible for logging, are passed as arguments to the architectural rule. Hence, there is no need for injecting information into the source code of the target system. Furthermore, by using a predicate to select the security services, the developer is left with some flexibility

in how they decide to apply the constraint. As opposed to a plain list of classes, a predicate can match all classes belonging to a specific package or following a set naming scheme, minimizing the need for revisiting the constraint as the system evolves.

```
1 ArchRule logSecurityEvents(
2         DescribedPredicate<? super JavaClass>
              securityServicesDescriptor,
3         Class<?> logger) {
4     return methods()
5         .that().haveModifier(JavaModifier.PUBLIC)
6         .and().areDeclaredInClassesThat(securityServicesDescriptor)
7         .should(callMethod(declaredIn(logger)));
8 }
```

Listing 5.2: Rule definition for constraint 1.

In BlogWeb, illustrated in Figure 5.1, the logging facility is the class named `Logger` while the only security service is the `UserService` class. An application of the constraint on this system can be as simple as the one shown in Listing 5.3.

```
1 @ArchTest
2 ArchRule logSecurityEvents = SecArchUnit
3     .logSecurityEvents(type(UserService.class), Logger.class);
```

Listing 5.3: Application of constraint 1 to BlogWeb.

5.2.2 Enforce AuthN/AuthZ at single point

The second constraint is defined in terms of two concepts: an authentication point and an authentication enforcer. Authentication is performed through a method call to the authentication enforcer, which is a class whose sole responsibility is to authenticate an actor. This call should only occur at the authentication point for the sake of ensuring a uniform authentication mechanism throughout the system. Authorization is enforced in the same manner, with the concepts of an authorization point and an authorization enforcer.

The definition of the second constraint is detailed in Listing 5.4. The constraint is defined as two separate rules, for the sake of clarity, but their implementations are identical.

```
ArchRule enforceAuthenticationAtCentralPoint(
        Class<?> authenticationPoint,
        Class<?> authenticator) {
    return CompositeArchRule.of(
        theClass(authenticationPoint)
            .should(callMethod(declaredIn(authenticator)))
    ).and(
        methods()
            .that().areDeclaredIn(authenticator)
            .should(onlyBeAccessedBy(authenticationPoint))
    );
}

ArchRule enforceAuthorizationAtCentralPoint(
        Class<?> authorizationPoint,
        Class<?> authorizer) {
    return enforceAuthenticationAtCentralPoint(
        authorizationPoint,
        authorizer
    );
}
```

Listing 5.4: Rule definition for constraint 2.

In BlogWeb, the authentication and authorization points are both situated in the `UserService` class while authentication and authorization are enforced by the classes `UserAuthenticator` and `UserAuthorizer` respectively. The application of the rule can be seen in Listing 5.5.

```
@ArchTest
ArchRule enforceAuthentication = SecArchUnit
    .enforceAuthenticationAtCentralPoint(UserService.class,
       UserAuthenticator.class);

@ArchTest
ArchRule enforceAuthorization = SecArchUnit
    .enforceAuthorizationAtCentralPoint(UserService.class,
       UserAuthorizer.class);
```

Listing 5.5: Application of constraint 2 to BlogWeb.

5.2.3 Messages are sent from a central point

The third constraint dictates that all outbound messages are sent from a central sending point. The intent is to have a single point that handles output sanitization or performs other safety checks on messages before they are sent. The act of sending a message is defined as a method call to a sender with at least one argument, which is assumed to contain the message contents. The reasoning is that any class should be allowed to create and pass around a sender instance without violating the constraint.

The rule definition can be seen in Listing 5.6. Since there can be multiple sender classes in a system, e.g. one for HTTP requests and one for SMTP messages, the

rule accepts a predicate that can select all these sender classes.

```
1 ArchRule sendOutboundMessagesFromCentralPoint(
2         Class<?> sendingPoint,
3         DescribedPredicate<? super JavaClass> senderDescriptor) {
4     return methods()
5         .that().areDeclaredInClassesThat(senderDescriptor)
6         .and(haveAtLeastOneParameter)
7         .should(onlyBeAccessedBy(sendingPoint));
8 }
```

Listing 5.6: Rule definition for constraint 3.

Listing 5.7 showcases how the constraint can be applied to BlogWeb. In this system, there is a single sender class `HttpResponse`, responsible for returning a response to a client. The central sending point is the `OutboundMessageSanitizer` class.

```
1 @ArchTest
2 ArchRule centralSendingPoint = SecArchUnit
3     .sendOutboundMessagesFromCentralPoint(
4         OutboundMessageSanitizer.class,
5         type(HttpResponse.class)
6     );
```

Listing 5.7: Application of constraint 3 to BlogWeb.

5.3 Injecting Information into Source Code

Some of the architectural constraints require that the developer injects additional information into the source code. In some cases, this information is simply an indicator that says something about an entire class. Naming the class with a specific suffix is one approach to accomplish this. Another approach is to implement an empty interface, which is the technique used with Java's `Serializable`[1] interface.

In other cases, however, the information may be required for methods of arbitrary signatures and even specific fields. For the purposes of flexibility and minimizing the obtrusiveness of our approach, any extra information is expressed in the form of annotations. These can be applied to classes, fields, methods and parameters without changing the underlying architecture of the system.

The need for additional information within the source code becomes apparent in the case where a class contains public methods with varying degrees of security requirements. A typical example is found in constraint 4, where a class is responsible for handling user input. Some methods might receive predefined values, such as in the case of checkboxes, whereas others treat strings entered by the user. The former type of input is guaranteed to be safe, whereas the latter entirely under the control of the attacker, thus introducing the need for sanitation. Using the broader predicate of entire classes (described in section 5.2) would not allow the constraint to be limited

[1] https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html

to specific methods within a class (those receiving potentially malicious input), and later trace a call to a method providing sanitation. Thus, annotations provide the granularity needed to limit the scope of a constraint to only the applicable code units.

5.3.1 Validate user input

User input comes in many forms, and as such, it is impossible to define a single algorithm to validate properly every single type. The problem grows further as queries (such as SQL) or other types of processed data (such as XML), each with its own set of grammar, are often formed using strings. As a consequence, the implemented constraint is more abstract as it checks whether a class that receives user input is said to either perform validation on its own or delegate the task another method. In total, four distinct cases conforming to the constraint where considered:

- *Method A* is annotated with both **UserInput** and **InputValidator**.
- *Method A* is annotated with **UserInput** and calls a *method B* that is annotated with **InputValidator**.
- *Method A* is annotated with **UserInput** and is only called by methods that are annotated with **InputValidator**.

The rule definition can be seen in Listing 5.8. As shown, the **UserInput** annotation is used on line 3 to limit the set of applicable code units, whereas the custom condition `performDirectOrIndirectValidation` implements the logic to detect violations for each of the three cases outlined above.

```
1 ArchRule validateUserInput() {
2     return codeUnits()
3         .that().areAnnotatedWith(UserInput.class)
4         .should(performDirectOrIndirectValidation);
5 }
```

Listing 5.8: Rule definition for constraint 4.

In BlogWeb, a user publishes a status update in the form of a string that the UserService class receives. The status is then passed to the StatusSanitizer class, where it is validated. The affected methods of each class are marked with the appropriate annotation, as shown in Figure 5.2. If the two annotations were to be added, without a call from the UserService class to the StatusSanitizer, the ArchUnit rule would fail and mark it as a violation of an architectural constraint.

5.3.2 Restrict thread spawning

While resources is a broad term, this constraint focuses on preventing the exhaustion of CPU and memory resources through the creation of new threads and processes. As such, every block of code that contains a call to the `start()` method of a

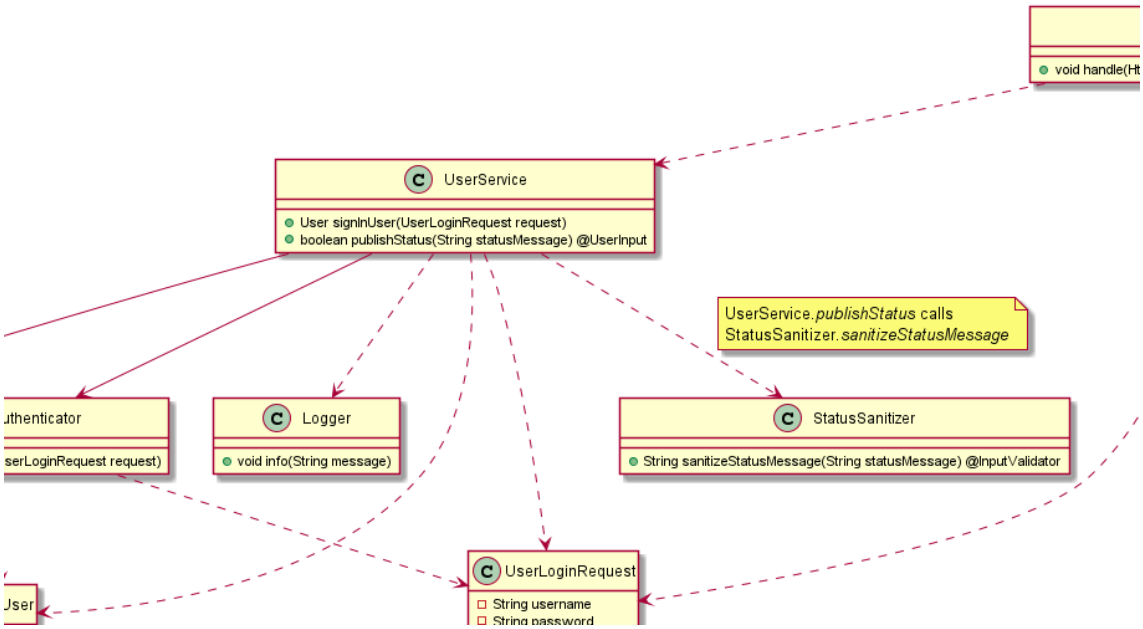

Figure 5.2: Applying constraint 4 to the model of BlogWeb using added annotations on the UserService and StatusSanitizer class.

`Thread`[2] or any of its subclasses, must be marked as containing a resource restriction mechanism. The same rule is applied for calls to `ProcessBuilder.start()`[3] and `Runtime.exec()`[3], which lead to the creation of new processes.

The marking is done with the help of an annotation, either on the relevant method or the entire class. The decision of how the restriction mechanism is implemented is left to the developer of the system.

```
1 ArchRule limitResourceAllocation() {
2     return noClasses()
3         .that().areNotAnnotatedWith(ResourceRestriction.class)
4         .should().callMethodWhere(
5             aThreadIsStartedWithoutRestriction
6         ).orShould().callMethodWhere(
7             aProcessIsStartedWithoutRestriction
8         );
9 }
```

Listing 5.9: Rule definition for constraint 5.

In BlogWeb, the RequestHandler class spawns new threads in order to handle several user requests concurrently. For the system to conform to the constraint, the ResourceRestriction annotation either has to be added to the entire class, or to the specific method that spawns the thread as shown in Figure 5.3. If not, the rule will fail, and the developer will have to ensure that the class cannot excessively spawn new threads.

[2] `https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html`
[3] `https://docs.oracle.com/javase/7/docs/api/java/lang/Process.html`

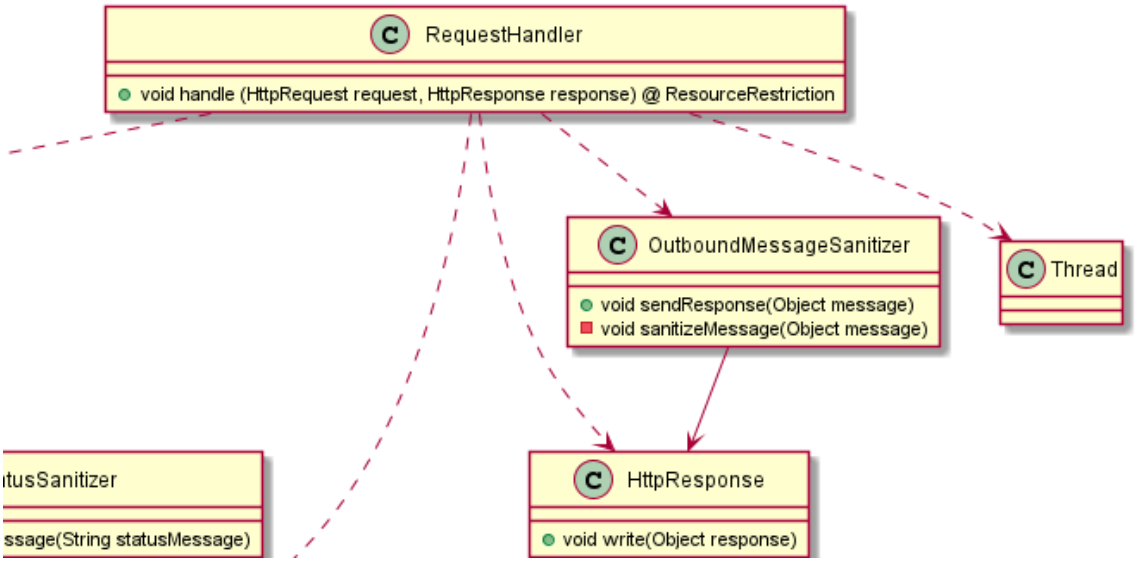

Figure 5.3: Applying constraint 5 to the model of BlogWeb using the added annotation on the RequestHandler class.

5.4 Extending ArchUnit Analysis

In the current ArchUnit API, a rule that aims to constrain method calls can only be defined in terms of the signatures of the method and its parameters. This is a non-issue when the arguments passed to a method are of the same type as the parameters. However, the argument might also be a descendant of the parameter type. There is currently no support in ArchUnit to constrain the types of the objects that are actually being passed as arguments.

Consider constraint 7, which aims to ensure that no secrets are passed to the logger. Say there is a `Secret` annotation that marks all the classes whose instances must not be passed to the logger. An attempt can be made to enforce the constraint with the current ArchUnit API using a custom predicate, as seen in Listing 5.10. Given that a typical logger will accept either a plain string, or a format string along with an array of objects to be formatted, this architectural rule does little in the way of preventing secrets from being passed to such a logger.

```
1 ArchRule doNotLogSecrets(Class<?> logger) {
2     return noClasses()
3         .should().callMethodWhere(
4             parameterTypeAnnotatedWith(Secret.class)
5                 .and(targetOwner(type(logger)))
6         );
7 }
```

Listing 5.10: A first attempt to implement constraint 7.

For the final 2 constraints, there is a need for an extension that allows constraints to be defined against method arguments rather than parameters. There should also be

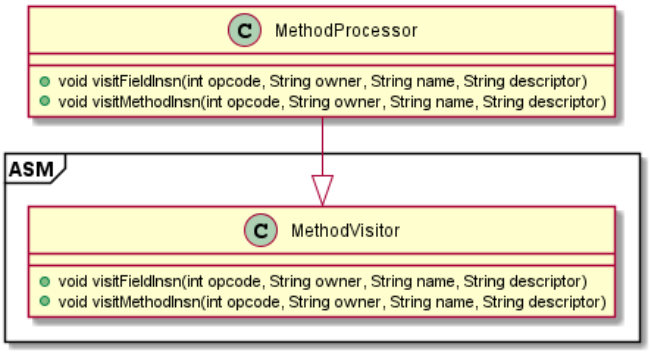

Figure 5.4: The immediate context of the `MethodProcessor` class in ArchUnit, responsible for analyzing code units.

hints about where these arguments have been derived from, e.g. which types that make up the components of a concatenated string. The following sections describe the extensions that have been made to ArchUnit, both in regards to its analysis and the information represented in its domain, as well as how these extensions are utilized in the definitions of the final constraints.

5.4.1 Extensions

Parts of this section might belong in the background?

ArchUnit builds its representation of the architecture using ASM[4], a Java bytecode analysis framework. ASM reads bytecode and generates callbacks to methods in its various visitor classes. The visitor of most interest to us is `MethodVisitor`[5], which is responsible for processing the contents of a method, constructor or static initializer. These are collectively named *code units* in ArchUnit's domain, i.e. anything that may contain code.

ArchUnit extends the `MethodVisitor` class in `MethodProcessor`, which primarily visits instructions related to field accesses and method invocations (see Figure 5.4). These instructions are processed into information about accesses between Java members, not entirely unlike the information provided by a static call graph.

Java, as a stack-oriented programming language, passes arguments to method calls and field assignments via the operand stack [29]. As such, an inspection of the operand stack at the time of a method call yields information about the arguments being passed. Conveniently, ASM provides an extension of its `MethodVisitor` class, called `AnalyzerAdapter`, which is capable of simulating the effect that each instruction has on the operand stack. As seen in Figure 5.5, `MethodProcessor` now extends this class and utilizes the information about the operand stack to perform an analysis of information flow within a code unit.

The information flow analysis of a code unit can be boiled down to the following key points:

[4] `https://asm.ow2.io/`
[5] `https://asm.ow2.io/javadoc/org/objectweb/asm/MethodVisitor.html`

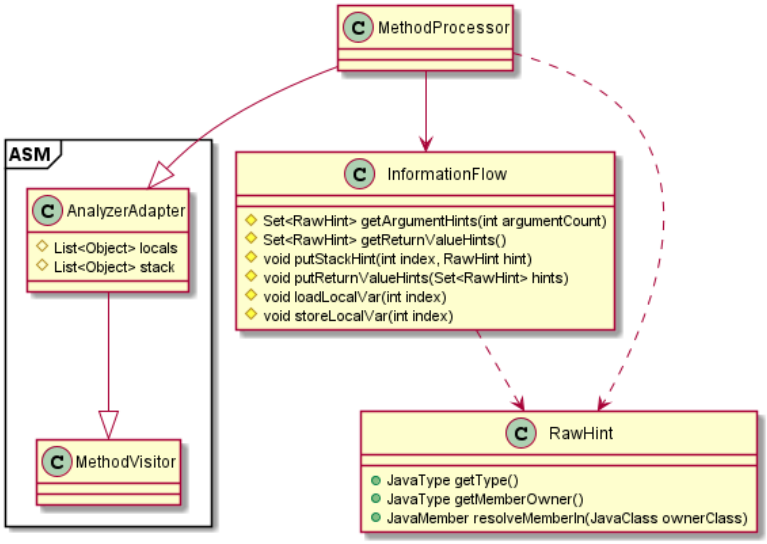

Figure 5.5: Changes made to the analysis of code units.

- Loading a field onto the stack yields a hint in that stack position about its originating member, i.e. the field.
- Invoking a method that has a return value yields a hint in the stack position of the resulting object about its originating member, i.e. the invoked method.
- Invoking a static method transfers the hints about the arguments, if any, into the stack position of the resulting object, if the method has a return value.
- Invoking a non-static method additionally transfers the hints about the arguments into the instance the method was invoked on, and the hints about the instance into the resulting object.
- Storing an object in an array transfers the hints about the object into the array.
- Loading an object from an array transfers the hints about the array into the object.
- Storing or loading a local variable transfers the hints from the stack to the local variable and vice versa.
- Duplicating a reference also duplicates the collection holding the hints for that reference, such that hints that flow into the duplicate also flow into the original reference.

Once the analysis of all classes has been completed and ArchUnit has built its representation of the architecture, the raw hints are resolved into hints referencing the actual type (`JavaClass`) and its origin (`JavaMember`), if any. As seen in Figure 5.6, `JavaAccess` now contains hints about the references that flow into the arguments. Additionally, `JavaMethod` contains hints about the references that flow into the

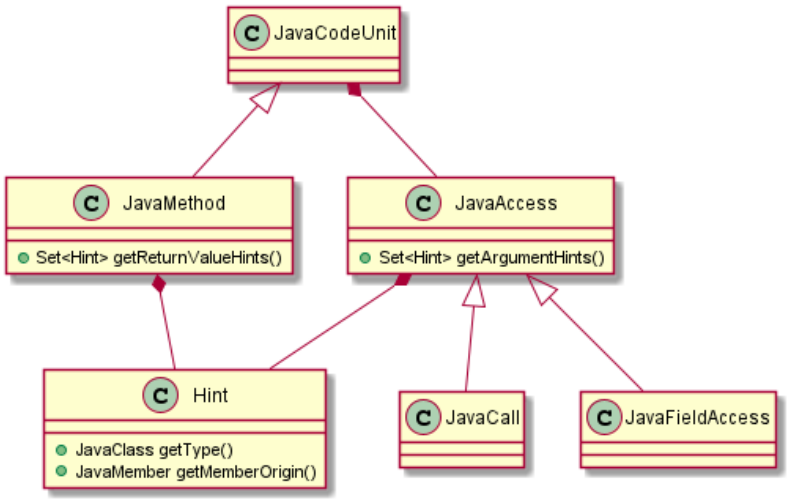

Figure 5.6: Changes made to the domain surrounding code units.

return value of the method. These changes to the domain aim to facilitate the definition of rules that constrain information flow.

5.4.2 Sensitive information must stay within trust boundary

Constraint six aims to constrain how assets that consist of sensitive information are allowed to flow between components. The constraint deals with the two concepts of assets and asset handlers, which are expressed in the form of `Asset` and `AssetHandler` annotations. An asset is a member that should only flow to classes marked with a high security level, i.e. an asset handler.

The definition of the rule can be seen in Listing 5.11. The rule itself requires no arguments, as all the necessary information is injected into the source code in the form of annotations. The custom condition, `notBleedToInsecureComponents`, ensures that the asset is only accessed directly by asset handlers. Moreover, it utilizes the information about return value flow to ensure that the asset is not accessed indirectly through intermediate methods.

```
1 ArchRule doNotBleedAssetsBetweenComponents() {
2     return fields()
3         .that().areAnnotatedWith(Asset.class)
4         .should(notBleedToInsecureComponents);
```

Listing 5.11: Rule definition for constraint 6.

In BlogWeb, the `password` field is considered an asset that must stay within the trust boundary of `UserLoginRequest` and `UserAuthenticator`. The appropriate annotations in this scenario can be seen in Figure 5.7.

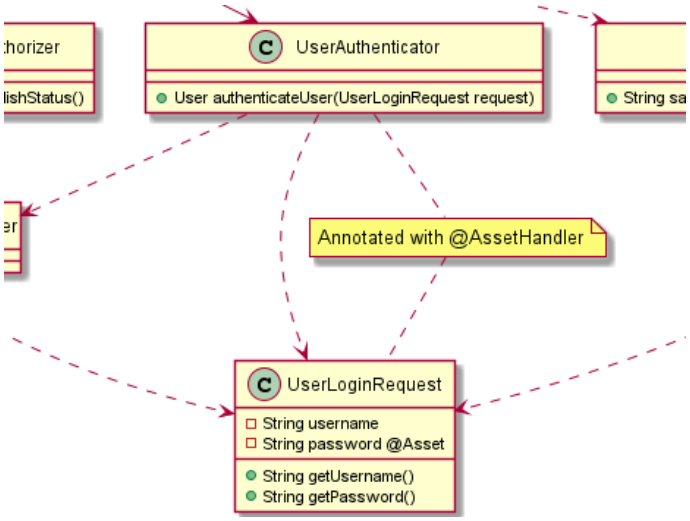

Figure 5.7: Application of constraint 6 to BlogWeb.

5.4.3 Secrets must not be exposed in log messages

The seventh and final constraint dictates that secrets must not be exposed in logs. There are a multitude of ways that a secret field can be exposed in a log message, e.g. through string concatenation, string formatting, or wrapping it in a different object and converting that object to its string representation. What these exposures have in common are that they issue method call to the logger where the secret has, in some way, flowed into the arguments.

The architectural rule is defined in Listing 5.12. Rather than a single class representing the preferred logger, the constraint should prevent exposures in any loggers present in the system. As such, it accepts a predicate that should select all such logging facilities. The custom condition `passSecretArgumentTo` inspects the argument hints of all outgoing method calls for any members marked with the `Secret` annotation. Additionally, for each originating member found in the hints, it recursively checks the hints that have flowed into that member, in an attempt to constrain information flow with intermediate steps in different code units.

```
1 ArchRule doNotLogSecrets(
2         DescribedPredicate<? super JavaClass> loggerDescriptor) {
3     return noClasses()
4         .should(passSecretArgumentTo(
5             targetOwner(loggerDescriptor)
6         ));
```

Listing 5.12: Rule definition for constraint 7.

In BlogWeb, the constraint can be applied as seen in Listing 5.13. The only secret in this system is the password field, annotated in Figure 5.8.

Figure 5.8: Annotation added to BlogWeb for constraint 7.

```
1 @ArchTest
2 ArchRule doNotLogSecrets = SecArchUnit
3     .doNotLogSecrets(type(Logger.class));
```

Listing 5.13: Application of constraint 7 to BlogWeb.

6

Evaluation

This chapter presents an evaluation of the constraints when applied to a number of open source systems. The evaluation was performed in two ways: first a comparison between SecArchUnit and static analysis tools used in industry, and then a standalone evaluation of the tool extension.

6.1 Comparison with Industry Tools

As mentioned in Section 3.3.2.3, constraints 1-5 can be validated in both SonarQube and PMD through the definitions of custom rules. The constraints are applied to the test systems presented in Section 3.3.2.1 and validated using each of the three tools. Both the relative and absolute frequencies of violations of each constraint can be seen in Figure 6.1 and Figure 6.2 respectively.

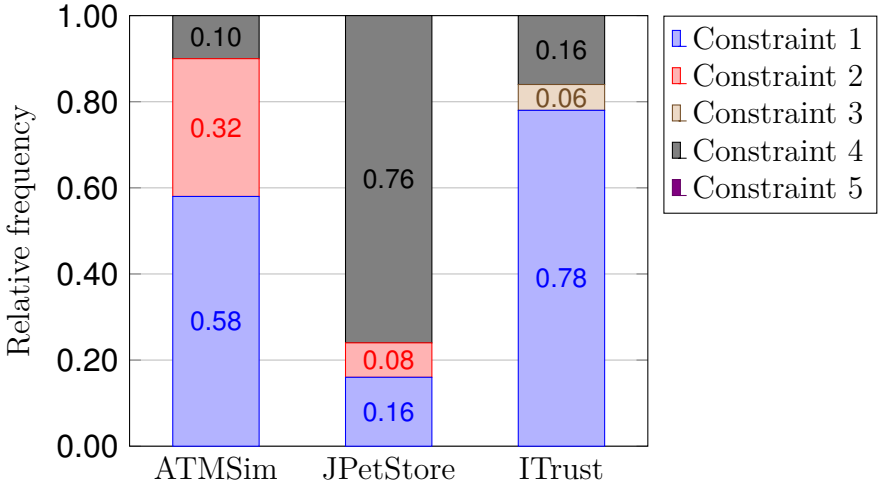

Figure 6.1: The relative frequency of constraint violations found in the three projects.

As shown in the figures, the systems were evaluated using imbalanced data. Constraint 1 accounted for a majority of all violations found throughout all three systems, while no system violated constraint 5. Addtionally, iTrust where the only system to contain a violation of constraint 3.

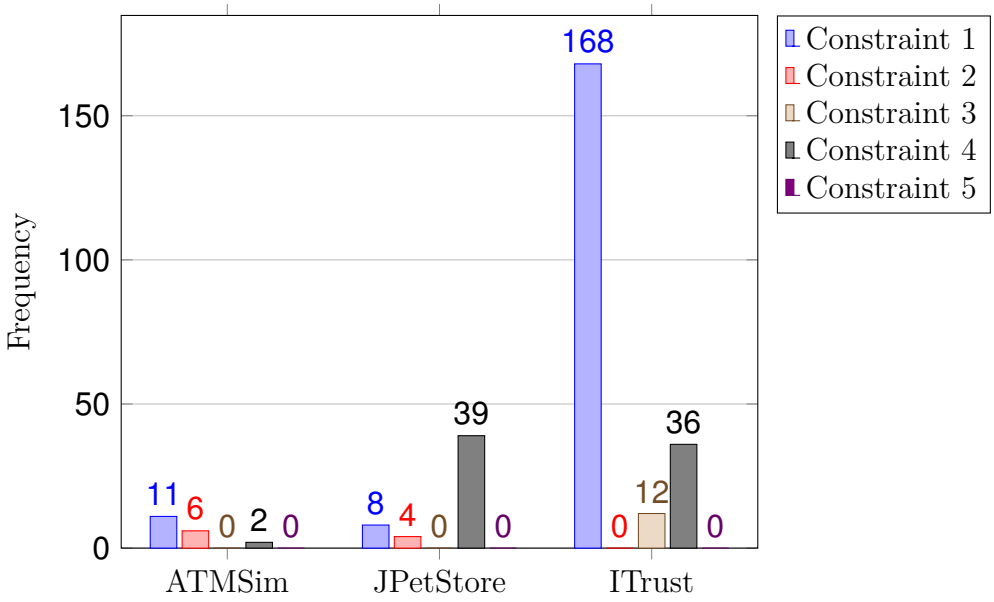

Figure 6.2: The frequency of constraint violations found in the three projects.

Each tool is evaluated according to the selected performance metrics, i.e. precision and recall. These results are presented in Table 6.1. Both in regards to precision, as well as recall, the tools performed equally. However, the causes of failure, in cases where the results of the tools varied, differed noticeably. The examples are described below:

- In ATM simulator, the same false positive occurred in all three tools. This was related to constraint 3, where a subclass of the sending point contained a method call to the sender. Additionally, PMD had 4 false negatives which occurred because it was unable to determine the classes that these method calls targeted.
- In JPetStore, PMD reported 4 false negatives, again because it was unable to determine the target class of these method calls.
- In iTrust, a security service contained both an inner interface and inner class whose methods did not perform security events. Both PMD and SonarQube analyzes the AST with a single class as its root node, thus they consider the inner members to be separate objects of analysis. In comparison, ArchUnit uses the entire Java file in its analysis causing it to incorrectly apply the annotation of the outer class to the inner members.

6.2 Validation of ArchUnit Extension

Constraints 6-7, which rely on an extension of ArchUnit's analysis, are validated solely using SecArchUnit. The performance metrics from applying the final two constraints to iTrust are presented in Table 6.2.

Project	Tool	TP	FP	FN	Precision	Recall
ATMsim	SecArchUnit	19	1	0	1	0.95
	SonarQube Plugin	19	1	0	1	0.95
	PMD Plugin	15	1	4	0.94	0.79
JPetStore	SecArchUnit	51	0	0	1	1
	SonarQube Plugin	51	0	0	1	1
	PMD Plugin	47	0	4	1	0.92
ITrust	SecArchUnit	216	3	0	0.97	1
	SonarQube Plugin	216	0	0	1	1
	PMD Plugin	216	0	0	1	1

Table 6.1: Results from validating constraints 1-5 using SecArchUnit, SonarQube and PMD.

The system, iTrust, initially contained no violations of constraint 7. Therefore, violations were injected by systematically marking all identifier fields (e.g. `patientId`, `personnelId`) in the model and base-action packages as secrets. We chose to mark these identifiers because they were commonly sent to the logger as a way to describe the patient or personnel involved in a transaction. Hence, the 37 violations of constraint 7, as seen in Table 6.2, are artificially injected.

Moreover, iTrust is built with a mix of Java and Java Server Pages (JSP) files whereas ArchUnit can only analyze Java bytecode. The classes in the action package, from which the logger is called, are all instantiated in the JSP files outside the view of our analysis. As such, the types of information flow that are analyzed and included in the ground truth are rather rudimentary. Out of the 37 violations of constraint 7, 1 was found without recursion (direct access to secret field) and the remaining 36 were found using a single recursion step (access to getter method of a secret field).

Constraint	TP	FP	FN	Precision	Recall
6	24	0	0	1	1
7	37	0	0	1	1

Table 6.2: Results from validating the extension-based constraints on iTrust.

7

Discussion

Tool comparison

Discuss examples of things that the tools won't catch

7.1 Comparing Static Analysis Tools

ArchUnit, SonarQube and PMD are all static analysis tools with support for evaluation of custom rules. However, they differ quite a bit in how their rules are defined. ArchUnit first builds a representation of the entirety of the analyzed system, which is then available during the evaluation of the rules. As shown throughout Chapter 5, rules have direct access to information about incoming and outgoing accesses to all fields, code units and classes, making it convenient to specify intricate architectural constraints.

As for SonarQube and PMD, both of these tools evaluate rules against Abstract Syntax Trees (AST) where the root node is the Java class currently being analyzed.

Rule definitions - Abstract Syntax Trees (SonarQube & PMD) vs higher-level abstraction (ArchUnit)

Analysis of references

Semantic information

7.2 Threats to Validity

7.2.1 Internal Validity

7.2.2 External Validity

7.3 Future Research

8

Conclusion

Bibliography

[1] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Security Testing," in *Advances in Computers*. Elsevier, 2016, vol. 101, pp. 1–51. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0065245815000649

[2] N. Yoshioka, H. Washizaki, and K. Maruyama, "A survey on security patterns," *Progress in Informatics*, no. 5, p. 35, Mar. 2008. [Online]. Available: http://www.nii.ac.jp/pi/n5/5_35.html

[3] G. Mcgraw, "Software security," *IEEE Security & Privacy Magazine*, vol. 2, no. 2, pp. 80–83, Mar. 2004. [Online]. Available: http://ieeexplore.ieee.org/document/1281254/

[4] I. Arce, K. Clark-Fisher, N. Daswani, J. DelGrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. McGraw, B. Schoenfield, and others, "Avoiding the top 10 software security design flaws," *IEEE Computer Society*, 2014.

[5] S. Jasser, "Constraining the Implementation Through Architectural Security Rules: An Expert Study," in *Product-Focused Software Process Improvement*, X. Franch, T. Männistö, and S. Martínez-Fernández, Eds. Cham: Springer International Publishing, 2019, vol. 11915, pp. 203–219. [Online]. Available: http://link.springer.com/10.1007/978-3-030-35333-9_15

[6] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, Jan. 2012. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121211002044

[7] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, no. 2, pp. 105–119, Mar. 2002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121201001522

[8] D. Garlan and M. Shaw, "AN INTRODUCTION TO SOFTWARE ARCHITECTURE," in *Series on Software Engineering and Knowledge Engineering*. WORLD SCIENTIFIC, Dec. 1993, vol. 2, pp. 1–39. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/9789812798039_0001

[9] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*. Pittsburgh, PA, USA: IEEE, 2005, pp. 109–120. [Online]. Available: http://ieeexplore.ieee.org/document/1620096/

[10] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 3rd ed., ser. SEI series in software engineering. Upper Saddle River, NJ: Addison-Wesley, 2013.

[11] R. Scandariato, K. Yskout, T. Heyman, and W. Joosen, "Architecting software with security patterns," Katholieke Universiteit Leuven, Belgium, Tech. Rep. CW 515, Apr. 2009.

[12] P. H. Nguyen, K. Yskout, T. Heyman, J. Klein, R. Scandariato, and Y. Le Traon, "SoSPa: A system of Security design Patterns for systematically engineering secure systems," in *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Ottawa, ON, Canada: IEEE, Sep. 2015, pp. 246–255. [Online]. Available: http://ieeexplore.ieee.org/document/7338255/

[13] M. Broy, J. Grünbauer, and C. A. R. Hoare, Eds., *Software system reliability and security*, ser. NATO security through science series. Amsterdam ; Washington, DC: IOS Press, 2007, no. v. 9, oCLC: ocn127107624.

[14] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh, "Security Requirements Engineering: A Framework for Representation and Analysis," *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 133–153, Jan. 2008. [Online]. Available: http://ieeexplore.ieee.org/document/4359475/

[15] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: connecting software architecture to implementation," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002.* Orlando, FL, USA: ACM, 2002, pp. 187–197. [Online]. Available: http://ieeexplore.ieee.org/document/1007967/

[16] M. Abi-Antoun and J. M. Barnes, "Analyzing Security Architectures," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 3–12, event-place: Antwerp, Belgium. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859001

[17] D. Luckham and J. Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, Sep. 1995. [Online]. Available: http://ieeexplore.ieee.org/document/464548/

[18] M. Abi-Antoun and J. Aldrich, "Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations," in *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA 09.* Orlando,

Florida, USA: ACM Press, 2009, p. 321. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1640089.1640113

[19] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*. Mumbai, India: IEEE, Jan. 2007, pp. 12–12. [Online]. Available: http://ieeexplore.ieee.org/document/4077029/

[20] A. Jansen, J. Bosch, and P. Avgeriou, "Documenting after the fact: Recovering architectural design decisions," *Journal of Systems and Software*, vol. 81, no. 4, pp. 536–557, Apr. 2008. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016412120700194X

[21] Hong Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, "DiscoTect: a system for discovering architectures from running systems," in *Proceedings. 26th International Conference on Software Engineering.* Edinburgh, UK: IEEE Comput. Soc, 2004, pp. 470–479. [Online]. Available: http://ieeexplore.ieee.org/document/1317469/

[22] J. C. S. Santos, K. Tarrit, and M. Mirakhorli, "A Catalog of Security Architecture Weaknesses," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. Gothenburg, Sweden: IEEE, Apr. 2017, pp. 220–223. [Online]. Available: http://ieeexplore.ieee.org/document/7958491/

[23] E. Gamma, Ed., *Design patterns: elements of reusable object-oriented software*, ser. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995.

[24] R. Scandariato, K. Yskout, T. Heyman, and W. Joosen, "A system of security patterns," Katholieke Universiteit Leuven, Belgium, Tech. Rep. CW 469, Dec. 2006.

[25] A. Eden and R. Kazman, "Architecture, design, implementation," in *25th International Conference on Software Engineering, 2003. Proceedings.* Portland, OR, USA: IEEE, 2003, pp. 149–159. [Online]. Available: http://ieeexplore.ieee.org/document/1201196/

[26] R. Wieringa, N. Maiden, N. Mead, and C. Rolland, "Requirements engineering paper classification and evaluation criteria: a proposal and a discussion," *Requirements Engineering*, vol. 11, no. 1, pp. 102–107, Mar. 2006. [Online]. Available: http://link.springer.com/10.1007/s00766-005-0021-6

[27] K.-J. Stol and B. Fitzgerald, "The ABC of Software Engineering Research," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 3, pp. 1–51, Sep. 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3276753.3241743

[28] S. Peldszus, K. Tuma, D. Struber, J. Jurjens, and R. Scandariato, "Secure Data-Flow Compliance Checks between Models and Code Based on

Automated Mappings," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Munich, Germany: IEEE, Sep. 2019, pp. 23–33. [Online]. Available: https://ieeexplore.ieee.org/document/8906984/

[29] S. Genaim and F. Spoto, "Information Flow Analysis for Java Bytecode," in *Verification, Model Checking, and Abstract Interpretation*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and R. Cousot, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3385, pp. 346–362. [Online]. Available: http://link.springer.com/10.1007/978-3-540-30579-8_23

A

SonarQube Rule Definition

```
1  class CentralMessageRule extends IssuableSubscriptionVisitor {
2      private static final String SENDING_POINT = "Transaction";
3      private static final MethodMatchers SENDERS = MethodMatchers
4          .create()
5          .ofTypes("atm.physical.NetworkToBank")
6          .anyName()
7          .addParametersMatcher(parameters -> !parameters.isEmpty())
8          .build();
9
10     @Override
11     public List<Tree.Kind> nodesToVisit() {
12         return Collections.singletonList(Tree.Kind.
               METHOD_INVOCATION);
13     }
14
15     @Override
16     public void visitNode(Tree tree) {
17         MethodInvocationTree methodInvocation =
18             (MethodInvocationTree) tree;
19
20         if (SENDERS.matches(methodInvocation)) {
21             // Get class where method invocation takes place
22             Tree parent = methodInvocation.parent();
23             while (!parent.is(Tree.Kind.CLASS)) {
24                 parent = parent.parent();
25             }
26             ClassTree classTree = (ClassTree) parent;
27
28             // Compare class with sending point
29             String sendingClassName = classTree.symbol().name();
30             if (!SENDING_POINT.equals(sendingClassName)) {
31                 reportIssue(methodInvocation, "Messages must only
                       be sent from the sending point");
32             }
33         }
34     }
35 }
```

B

PMD Rule Definition

```
1  class CentralSendingRule extends AbstractJavaRule {
2      private static final String SENDING_POINT =
3          "atm.transaction.Transaction";
4      private static final Collection<String> SENDERS =
5          Arrays.asList(
6              "atm.physical.NetworkToBank"
7          );
8
9      public CentralSendingRule() {
10         super();
11
12         // Types of nodes to visit
13         addRuleChainVisit(
14             ASTClassOrInterfaceBodyDeclaration.class
15         );
16     }
17
18     @Override
19     public Object visit(
20             ASTClassOrInterfaceBodyDeclaration body,
21             Object data) {
22         ASTClassOrInterfaceDeclaration owningClass =
23             body.getFirstParentOfType(
24                 ASTClassOrInterfaceDeclaration.class
25             );
26         boolean isSendingPoint = owningClass != null &&
27             SENDING_POINT.equals(owningClass.getBinaryName());
28         if (isSendingPoint) {
29             // Method defined in sending point; skip this method
30             return data;
31         }
32
33         Util.getMethodCallsFrom(body).stream()
34             .filter(call -> SENDERS.contains(call.targetOwner)
35                 && call.argumentCount > 0)
36             .forEach(offendingCall -> addViolation(data,
                   offendingCall.source));
37
38         return data;
39     }
40 }
```