Security Contracts a Property-Based Approach to Support Security Patterns

Sylvain Guérin[1], Joel Champeau[1], Salvador Martínez[2] and Raul Mazo[1]
[1]*Lab-STICC, ENSTA Bretagne, Brest, France*
[2]*Lab-STICC, IMT Atlantique, Brest, France*
{firstName.lastName}@ensta-bretagne.fr, {firstName.lastName}@imt-atlantique.fr

Keywords: Design by Contract, Security Patterns, Security Contracts, Runtime Monitoring.

Abstract: Security patterns represent reusable solutions and best practices intended to avoid security-related flaws in software and system designs. Unfortunately, the implementation and enforcement of these patterns remains a complex and error-prone task. As a consequence, and besides implementing a given security pattern, applications often remain insecure w.r.t. the security risk they intended to tackle. This is so for two main reasons: 1) patterns are rarely re-usable without adaptation, and thus concrete implementations may fail to deal with a number of (often implicit) properties, which must hold in order for the pattern to be effective; 2) patterns are deployed in environments with uncertainties that can only be known at runtime. In order to deal with this problem, we propose here *Security Contracts*, a framework that permits the specification and runtime monitoring of security patterns and related properties (including temporal ones) in both new and existing applications. It is based on an extension of the Design-by-Contract paradigm to enable the specification of security patterns and the runtime adaptation of applications. We demonstrate the feasibility of our approach with an implementation and its evaluation on a framework used worldwide in web technologies, Spring.

1 INTRODUCTION

Information systems are core components of nowadays institutions and companies and often subject to a wide range of security threats. These threats may affect both, the information they manage and their correct functioning. To cope with these security threats, research and industry efforts have led to the identification of a number of generic solutions to mitigate various security vulnerabilities. Among them, we may find: authentication, permission handling, resource access, etc. These solutions, which are meant to capture security expertise and provide architectural and functional guidelines to system developers, are known as security design patterns (Fernandez-Buglioni, 2013; Yoshioka et al., 2008).

As reusable as they may be, patterns need to be often adapted to the problem and solution domains, and thus implementations may deviate from the initial pattern description. Secondly, patterns may depend on environment uncertainties that can only be known at run-time, imposing a monitoring requirement for safe enforcement. Consequently, a number of (often implicit) properties among those that must hold in order for a pattern to be effective may be left unsatisfied (by missing a correct implementation, run-time enforcement, or both). In this sense, the need for an approach to help with both the specification of security patterns and their monitoring at run-time appears to be critical. Such an approach must provide the means to: 1) Specify complex patterns with a focus on guaranteed security properties; 2) Instantiate and monitor them on both new (in development) and existing applications. Existing applications must be enhanced in case of faulty or incomplete security patterns.

We propose a solution based on an extension of the Design-by-Contract (DbC) paradigm (Meyer, 1992), a method designed to formalize the interaction between components in terms of expectations and guarantees. In this sense, our solution supports the definition of security patterns as security contracts and their monitoring and enforcement through run-time adaptation. We call this run-time phase Execution under Contract (EuC).

We take advantage of *Pamela* (Guérin et al., 2021), a model-oriented programming framework for Java. At design time, we use *Pamela* to describe reuseable security patterns. The description focuses on the security properties that the contract is assuming to guarantee. Then, we use annotations in the applica-

Guérin, S., Champeau, J., Martínez, S. and Mazo, R.
Security Contracts a Property-Based Approach to Support Security Patterns.
DOI: 10.5220/0012305600003648

In *Proceedings of the 10th International Conference on Information Systems Security and Privacy (ICISSP 2024)*, pages 103-111
ISBN: 978-989-758-683-5; ISSN: 2184-4356

tion Java code, in order to instantiate the pattern. At runtime, the *Pamela* engine intercepts the execution of the application, allowing security property monitoring and enforcement.

We validate our approach by applying an authenticator security pattern on a widely used Spring framework (Johnson et al., 2004).

The remainder of the paper is organized as follows. Section 2 presents some preliminaries on Security Design Patterns and Design-by-Contract. The security contracts approach is described in Section 3 followed by the presentation of our case study in Section 4 together with a critical discussion. Section 5 deals with related work. We end the paper in Section 6 by presenting conclusions and future work.

2 PRELIMINARIES & EXAMPLE

This section starts with a brief presentation of the security design pattern concept. After that, we describe the *Authenticator* security pattern which will serve as a running example throughout the rest of the paper.

2.1 Security Design Patterns

In order to face threats in terms of software security, security design patterns have naturally imposed themselves in the continuity of design patterns. An abundant literature has been published, including books containing patterns catalogs (Schumacher et al., 2013). Based on these catalogs, different works have been developed to, for example, provide pattern languages to classify and create relations between patterns according to threats (Fernandez-Buglioni, 2013), formalize patterns (Behrens, 2018) or study them in specialized domains such as automobiles (Cheng et al., 2019).

Like in (Fernandez et al., 2018), a security pattern is made up of several parts. where mainly the problem, the solution and some additional comments are defined. We illustrate these parts in the next section with the Authenticator pattern

2.2 The Authenticator Pattern

The Authenticator pattern is characterized as follows:

- The *Intent*. How to verify that a subject (user or system) intending to access the system is who she claims to be? The subject must present information that is recognized by the system and receives some proof of successful authentication.
- The *context*. Computer systems contain sensible resources. We only want subjects that have some reason to be in our system to enter the system.
- The *problem*. The purpose is to prevent a malicious subject from trying to impersonate a legitimate user in order to have access to sensible resources. This could be particularly critical when the impersonated user has a high level of privileges or offers the possibility to escala this level.
- The *forces*. We emphasize some forces on this pattern:
  - Authentication information protection. The pattern must assume the non-usurpation of the authentication information.
  - Authenticate authority integrity is preserve during its life to enforce the non-modification of the authentication information.
  - Proof of identity tamper resistance. The integrity of the proof of identity presented by the user is preserved during its life cycle.
  - Authentication frequency. User requests to authenticate are limited in a given period of time. The objective is to prevent brute-forced automatic attempts.

  This strength list represents properties that must be respected to ensure a proper use of the pattern.
- The solution. The definition of the solution includes a structural part and a behavioral part. The structural part is generally expressed with UML class diagrams such as in Figure 1, which defines several entities:
  - A *Subject* needing to be authenticated,
  - A *Proof of Identity*, token given to the subject once authentication is complete,
  - An *Authenticator* is the object which implements an authentication algorithm and creates the *Proof of Identity*,
  - *Authentication Information* are the information provided by *Subject* to the *Authenticator*.

  The behavioral part is based on several interaction scenarios between the structural entities. These scenarios are illustrated through the code of our implementation as described in Section 4.

Based on this pattern, we will demonstrate how to create a security contract with related properties and how we ensure that the properties are guaranted during the execution.

With our approach, we create a continuum between the development process with the Design by Contract (DbC) and the Execution under Contract (EuC), to verify at runtime the specified security properties.

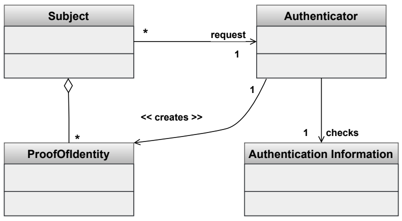

Figure 1: UML class diagram of the Authenticator pattern.

3 SECURITY CONTRACTS APPROACH

We devote this section to the description of our *Security Contracts* approach, which builds on our preliminary work in (Silva et al., 2020). *Security Contracts* is an extension of the DbC method and the monitoring of these security contracts. Concretely, we extend DbC that the contracting party is composed not only of individual classes and methods, but sets of collaborating classes as we may find in securitypatterns. For the runtime support, our extension includes tooling to instantiate security contracts as cross-cutting *aspects* on host applications, and runtime monitoring (and enforcement) of pattern properties (i.e., EuC). A library of security contracts is established to improve the reusability of these security contracts, notably for existing applications. In particular, this section (i) formally describes security contracts, (ii) presents the application of such contracts to a security pattern, and (iii) presents tooling support the monitoring of the authenticator pattern properties, to illustrate our approach.

3.1 Contract Specification

Let us exemplify the definition of a security contract for the Authenticator pattern. To do so, we will first use formal Boolean expressions in order to describe the contract properties. Note that these Boolean expressions suppose the existence of classes representing the concepts of the authenticator pattern such as *Subject*. The *Authenticator* pattern intrinsically defines five security properties (four implicit properties and two functional properties), which have been previously introduced as pattern forces in Section 2:

1. **Unicity of the Couple Subject/Authentication Information.**

   $$P1 : \forall a, b \in I_{Subject},$$
   $$a \neq b \implies a.authInfo \neq b.authInfo$$

2. **Authentication Information Integrity.**

   $$P2 : \forall a \in I_{Subject}, a.authInfo = a.authInfo_{ini}$$

3. **Authenticate Authority Integrity.**

   $$P3 : \forall a \in I_{Subject}, a.authenticator = a.authenticator_{ini}$$

4. **Verification of the Validity or an Undefined Proof of Identity.**

   $$P4 : \forall a \in I_{Subject}, (a.idProof = \emptyset) \vee$$
   $$(a.idProof = a.authenticator.request(a.authInfo))$$

5. **Continue verification of the proof of identity.**

   $$P5 : self.idProof =$$
   $$self.authenticator.request(self.authInfo)$$

   Remark. After authentication, the proof of identity must always match the value initially returned by the query method. Joint verification of properties *P4* and *P5* guarantees the validity of the identity proof throughout the session.

The specification of the *authenticator* security contract is now defined with these properties and the abstract definition of the entities of the Authenticator pattern in Figure 1. Note that this security contract is naturally extensible by adding security properties in increments throughout the life cycle of the application. In Section 4, we illustrate the extension capacity of our security contracts.

3.2 Deployment, Monitoring and Enforcement

For the operationalization of our *Security Contracts* we describe the PAMELA framework where properties are annotations and the properties are ensured at runtime by the framework.

3.2.1 The *Authenticator* Security Contract in PAMELA

To illustrate the main features of our implementation, we present here how the *Authenticator* contract is implemented with PAMELA. As described previously, the implementation of the authentication pattern is based on three classes `AuthenticatorPatternFactory`, `AuthenticatorPatternDefinition` and `AuthenticatorPatternInstance` to define the security contract the authentication service. First, each attribute of the `AuthenticatorPatternDefinition` class has a corresponding annotation; the figure 2 illustrates the left part of the figure. After the definition, these annotations are directly used in the code that we want to assume an Execution under Contract.

The `Manager` is only illustrative to clearly demonstrate which instance of the `Manager` class plays the *Authenticator* role (in the pattern description), while an instance of the `Client` class plays the *Subject* role.

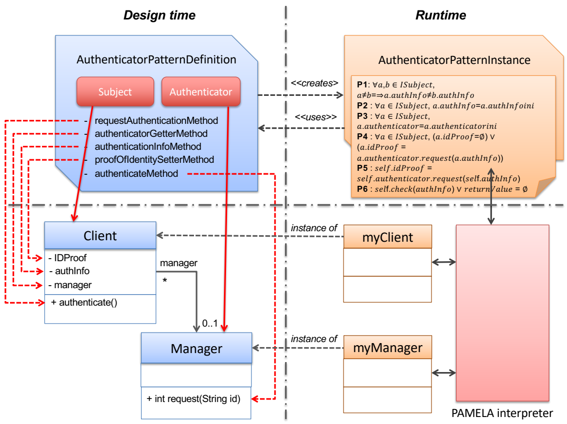

Figure 2: PAMELA vision of the Authenticator pattern.

At runtime, both functional code and security contract logic, i.e. pattern class behavior and contract enforcement, are weaved by the PAMELA framework. This allows, for instance, for the contract properties to be checked at runtime before and/or after any method of interest. In the case of the Authenticator pattern, the `AuthenticatorPatternInstance` objects will check the invariants of the pattern contract before and after every call to *Subject* and *Authenticator* classes. The mapping is summarized in Figure 2.

Note that the contract user only needs to annotate his code and PAMELA will automatically handle the pattern business logic (both, pattern behavior and assertion checking). For more details on this example, we direct the reader to our previous paper (Silva et al., 2020).

Our tooling includes a security contract library ready for deployment by application developers in their source code. As of now, this library includes the following contracts: Authenticator, Authorization, Single access point, Owner, Role-based access control

Details about these patterns and how to use them are available on the project's Web site[1].

4 USE CASE: APPLYING SECURITY CONTRACT TO EXISTING CODE

To demonstrate the effectiveness of *security contracts*, we apply them to an existing and widely used framework, the Open Source *Spring* framework [2]. We selected this framework due to the worldly use for web's application backend. This intensive use is supported by the Java language with its strong typed system.

The Open Source status of this framework provides an excellent use case to tackle our Security Contracts related to both the reuse of legacy code and on the Authentication process to ensure an efficient first shield.

4.1 The *Spring* Framework

Spring is a framework for developing Web applications and is based on predefined libraries that include reusable classes. In terms of security, the extension possibilities of this framework can be problem sources in terms of understanding and by untimely addition of bugs or vulnerabilities. Spring has a web module that supports the Servlet API (which dynamically creates data within an HTTP server) called Spring MVC for Model View Controller.

Spring Security is the library that includes the management of authentication and access control. Like any *Spring* project, it is customizable. Authentication being the first security rampart of the application, the issue here is to make sure that this *Authenticator* design pattern preserves the desired security properties. This requirement has guided our experiment to focus on the application of Security Contracts on the authentication process.

Figure 3 details the authentication management within the *Spring* framework. A generic scenario is explained in the figure through the diagram numbers. So when the application developed with *Spring Security* receives a request(1), a component chain is activated. When the request contains an authentication one, the *AuthenticationFilter* will extract the user's credentials (usually username and password) and create an *Authentication* object. If the information received contains a correct username and password, a *UsernamePasswordAuthenticationToken* will be created containing the username and the password (2).

This token will be used to invoke the *authenticate()* method of *AuthenticationManager* which is implemented by *ProviderManager* (3). There are several *AuthenticationProvider* already configured and listed in *ProviderManager*. The one we will use in this experiment is the *DAOAuthenticationProvider*. DAO stands for "Data Access Object", which is a model that provides an abstract interface to a database type. By mapping application calls to the persistence layer, the DAO provides certain specific data operations without exposing the details of the database. The *DAOAuthenticationProvider* uses *UserDetailsService* (5) to retrieve user data based on

[1]https://www.openflexo.org/pamela/docs/category/security-features/

[2]https://github.com/spring-projects/spring-framework

the user's username (6) , (7) , (8) , (9). If the authentication (10) succeeds, then the complete *Authentication* object (with "authenticated = True", the list of authorities and the user name) is returned. Finally, the *AuthenticationManager* returns the *Authentication* object to the *Authenticationfilter*, the authentication has succeeded, and the object is stored in the *SecurityContext*. And if authentication fails, the *AuthenticationManager* raises an exception *AuthenticationException* will be thrown.

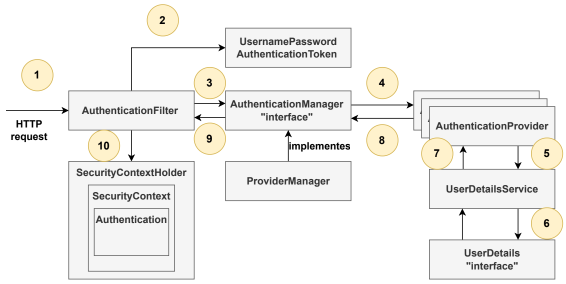

Figure 3: Authenticate service in the *Spring* framework.

4.2 Applying the *Authenticator* Security Contract on the *Spring* Authentication

The implementation of the Security Contract *Authenticator* in this context is based on two phases: 1) Weaving the *Authenticator* pattern entities into the *Spring* architecture, presented just below 2) Allocating the security properties, presented in the section 3.1, to be preserved by the application.

These two steps are ensured by source code annotation techniques.

Based on the class diagram of the *Authenticator* of the figure 1, the contract identifies four distinct entities: *Authenticator*, *Subject*, *AuthenticationInformation* and *ProofOfIdentity*, for which we need to find the corresponding business concept in the *Spring* source code to be annotated. To examplify our approach, for this paper we select only the *Authenticator* and the *Subject* entities. Note that the full code of this use case can be found on the project web site [3].

4.2.1 Authenticator

This concept is identified in the *Authenticator* pattern with the `@Authenticator` annotation. On the application to secure, the Java interface *AuthenticationProvider* plays the corresponding role. So this role is applied, line 3 of the listing 1, to the specialization *CustomAuthenticationProvider* which extends *AuthenticationProvider*.

```
1 @ModelEntity
2 @ImplementationClass(
      CustomAuthenticationProviderImpl.class)
3 @Authenticator(patternID = SessionInfo.
      PATTERN_ID)
4 @Imports(@Import(SessionInfo.class))
5 public interface CustomAuthenticationProvider
      extends AuthenticationProvider {
6   String USER_NAME = "userName";
7   // Inherited from AuthenticationProvider API
8   public Authentication authenticate(
        Authentication authentication) throws
        AuthenticationException;
9     ...
10  abstract class CustomAuthenticationProviderImpl
        extends DaoAuthenticationProvider
        implements CustomAuthenticationProvider {
11    @Override
12    public Authentication authenticate(
       Authentication authentication) throws
       AuthenticationException {
13      ...
14    }
15    @Override
16    public UsernamePasswordAuthenticationToken
       request(String userName) {
17      ...
18    }
19  }
20 }
```

Listing 1: CustomAuthenticationProvider definition.

From an implementation point of view, the role is implemented by the *CustomAuthenticationProviderImpl* class, presented on line 10 in the listing 1. This class extends the *DAOAuthenticationProvider* class that takes care of this role in the Spring framework.

The link between the role and the pattern definition is done via the identifier `SessionInfo.PATTERN_ID`, which is currently a character string, but must evolve in a unique URI; see line 3 listing 1.

4.2.2 Subject

This concept is identified in the *Authenticator* pattern with the `@AuthenticatorSubject` annotation, line 3 of the listing 2. On the *Spring* source code, this concept is not directly reified, but corresponds to the notion of session from our point of view. Thus, we reify the concept `SessionInfo`, which is presented in the listing 2 and allows the management of the user name (authentication information), IP address, and proof of identity from lines 6 to 16, as well as the access to the authenticator via the accessors `getAuthenticationProvider()` and `setAuthenticationProvider()`.

[3] https://github.com/openflexo-team/pamela/tree/2.0/pamela-spring-security-uc

```
1 @ModelEntity
2 @ImplementationClass(SessionInfo.SessionInfoImpl.class)
3 @AuthenticatorSubject(patternID = SessionInfo.PATTERN_ID)
4 public interface SessionInfo {
5   String PATTERN_ID = "AuthenticatorPattern";
6   @Getter(USER_NAME)
7   @AuthenticationInformation(patternID = PATTERN_ID, paramID = CustomAuthenticationProvider.USER_NAME)
8   String getUserName();
9   @Setter(USER_NAME) void setUserName(String val);
10  @Getter(IP_ADRESS) String getIpAdress();
11  @Setter(IP_ADRESS) void setIpAdress(String val);
12  @Getter(value = ID_PROOF, ignoreType = true)
13    UsernamePasswordAuthenticationToken getIDProof();
14  @Setter(ID_PROOF)
15  @ProofOfIdentitySetter(patternID = PATTERN_ID)
16  void setIdProof(UsernamePasswordAuthenticationToken value);
17  @Getter(AUTHENTICATION_PROVIDER)
18  @AuthenticatorGetter(patternID = PATTERN_ID)
19  CustomAuthenticationProvider getAuthenticationProvider();
20  @Setter(AUTHENTICATION_PROVIDER)
21  void setAuthenticationProvider(CustomAuthenticationProvider val);
22    ....
23  @AuthenticateMethod(patternID = PATTERN_ID)
24  void authenticate();
25  @RequiresAuthentication
26  void checkSecure();
27   abstract class SessionInfoImpl implements SessionInfo {
28     // implementation code of checkSecure() and getIpAdress()
29  }
30 }
```

Listing 2: SessionInfo entity implementation.

The method `authenticate()` (line 24 of the listing 2 is identified via the annotation `@AuthenticateMethod` to the method `request` of the pattern *Authenticator* as presented in Figure 2. The `checkSecure()` method is annotated as `@RequiresAuthentication`, line 25-26, to trigger the Authentication process based on the method with the annotation `@AuthenticateMethod` if the process is executed for the first time.

4.2.3 Authentication Management

The management of the authentication itself is intricated because of two competing levels: the authentication provided by the *Spring* framework and the one implemented in our *Authenticator* pattern. The alignment of the two mechanisms takes place in the implementation of `CustomAuthenticationProviderImpl` (listing 3). The *Spring* framework receives the authentication requests via the call of the method `authenticate(Authentication)` of `AuthenticationProvider`. We overwrite this method, line 11, with the management of the current session information (from lines 13 to 25), and the `authenticate()` and `checkSecure()` of the `SessionInfo` entity, lines 21 to 24, guarantees the use of the security pattern to ensure contract properties.

4.3 A Temporal Logic Property on a *Authenticator* Pattern Specialisation

One of the interests of our approach lies in the ability of the framework to provide extension points. In our case, the `CustomAuthenticationProvider` class specializes in the *Authenticator* pattern.

The second feature of our framework that we exploit is the reification of the notion of instance of the *Authenticator* pattern through the instance of the `AuthenticatorPatternInstance` class. This class encapsulates the instances of the subject (class `SessionInfo`) and authenticator (class `CustomAuthenticationProvider`). As this class gives access to the instances of the classes of the pattern, an introspection capacity of the current state of the pattern is provided during all its life cycle.

We have taken advantage of these two aspects to specialize the *Authenticator* pattern by extending the behavior with a new temporized temporal property. This property takes into account a time quantity applied on an execution path of the pattern. Classically, this property expresses the fact that there cannot be more than three authentication failures in a given period of time. If three failures occur in this period, the application will have to switch to another mode, for example, refusing any authentication attempt for a certain time, for example.

Let $auth\_fail_i$ be an "authentication failure" event for the property specification in the current execution trace:

$$P7{:}\forall\{auth\_fail_i, auth\_fail_{i+1}, auth\_fail_{i+2}\} \in execution\_trace$$

$$auth\_fail_{i+2}.time - auth\_fail_i.time < TIME\_LIMIT$$

The class `CustomAuthenticatorPatternInstance`, listing 4, implements the definition of the property *P7* through the method `checkRecentAuthFailCountLessThan3` from lines 7 to 15. This property is based on the evaluation of the `events` variable, defined line 3, which is defined as an instance variable of this class. This variable `events` embodies the current state of the class's instances

```
abstract class CustomAuthenticationProviderImpl extends DaoAuthenticationProvider implements CustomAuthenticationProvider {
    private Map<String, UsernamePasswordAuthenticationToken> tokens
        = new HashMap<>();
  /** Implementation is here trivial as we use map filled by
      * {@link # authenticate ( Authentication )} method */
  @Override
  public UsernamePasswordAuthenticationToken request( String userName) {
          return tokens . get (userName);
    }
  @Override
  public Authentication authenticate ( Authentication authentication ) throws AuthenticationException {
    try {
      UsernamePasswordAuthenticationToken returned = ( UsernamePasswordAuthenticationToken) super. authenticate ( authentication );
      String name = authentication .getName();
      WebAuthenticationDetails details = ( WebAuthenticationDetails ) authentication . getDetails () ;
      String userIp = details .getRemoteAddress();
      SessionInfo sessionInfo = SessionInfo . getCurrentSessionInfo () ;
      sessionInfo .setUserName(name);
      sessionInfo . setIpAdress ( userIp ) ;
      tokens . put(name, returned ) ;
      // Call authenticate () to complete process
      sessionInfo . authenticate () ;
      // Ensure that we are now in authenticated context
      sessionInfo .checkSecure() ;
      return returned ;
    } catch ( AuthenticationException e) {
      throw e;
    } catch (ModelExecutionException e) {
      e. printStackTrace () ;
      throw new SessionAuthenticationException (" Exception during authentication : " + e. getMessage()) ;
      }
    }
}
```

Listing 3: CustomAuthenticationProviderImpl implementation.

and is part of the global state of the pattern. In our example, this variable is updated, line 17, to take into account each authentication failure, and so assessed for the property evaluation.

After the definition of the contract property, the listing 5 presents the definition of the *Authenticator* referring to this property as a precondition of the

```
1 public class CustomAuthenticatorPatternInstance extends
      AuthenticatorPatternInstance<CustomAuthenticationProvider,
      SessionInfo, String, UsernamePasswordAuthenticationToken> {
2   public static long TIME_LIMIT = 180000; // 180s = 3 min
3   private List<PatternInstanceEvent> events = new ArrayList<>();
4   public CustomAuthenticatorPatternInstance(
        CustomAuthenticatorPatternDefinition patternDefinition,
        PamelaModel model, SessionInfo subject) {
5     super(patternDefinition, model, subject);
6   }
7   // Perform check that last 3 AuthFailEvent in current time limit
8   public boolean checkRecentAuthFailCountLessThan3() {
9     long currentTime = System.currentTimeMillis();
10    if (events.size() >= 3 && (currentTime - events.get(events.
          size() - 3).getDate()) < TIME_LIMIT) {
11      // 3 attempts or more in TIME_LIMIT interval
12      return false;
13    }
14    return true;
15  }
16  public void generateAuthFailEvent() {
17      events.add(new AuthFailedEvent());
18    }
19    .......
20 }
```

Listing 4: CustomAuthenticatorPatternInstance implementation.

```
1 public interface CustomAuthenticationProvider extends
        AuthenticationProvider {
2    ...
3   @Override
4   @OnException(
5       patternID = SessionInfo.PATTERN_ID,
6       onException = AuthenticationException.class,
7       perform = "patternInstance.generateAuthFailEvent()",
8       strategy = OnExceptionStategy.HandleAndRethrowException)
9   @Ensures(patternID = SessionInfo.PATTERN_ID, property = "
          patternInstance.checkRecentAuthFailCountLessThan3()")
10    public Authentication authenticate(Authentication
          authentication) throws AuthenticationException;
11   ...
12 }
```

Listing 5: CustomAuthenticationProvider implementation.

method `authenticate()` with an `@Ensures` annotation with property *P7* as parameter, line 9. Also, we can see between the lines 4 and 8, the use of the annotation `@OnException` which allows to automatically generate events corresponding to the failure of connection. This event is stored in the pattern state via the `events` variable as we described before.

The *P7* annotation is defined related to the `authenticate()` method. So, for now, any call to this method respects the security contract defined by the instance `CustomAuthenticatorPatternInstance` corresponding to the current execution. This contract automatically handles the `AuthFailedEvent` event and checks that a user cannot fail to authenticate more than 3 times in a given time.

5 RELATED WORK

To the best of our knowledge, ours is the first approach aimed at extending the Design by Contract paradigm to the specification and run-time monitoring of Security Contracts. The limit of classical contracts for the monitoring of patterns and properties that require context knowledge was already acknowledged in (Hallstrom et al., 2004) and (Ostroff et al., 2009) which served as inspiration for our work.

Formal definitions of security patterns are provided by several authors (Cheng et al., 2019), (Behrens, 2018), (Da Silva Júnior et al., 2013). The purpose of these formal definitions is to analyze the behavior of the patterns at design level or to enable automatic analysis operations. On the contrary, our proposal to formalize security patterns with a Design by Contract approach aims at ensuring a secure implementation of the patterns. Thus, we can see these approaches as complementary.

More related to the implementation level, in (Mongiello et al., 2015), the authors propose AC-contract, an approach for the run-time verification of (adaptive) context-aware applications. It uses pre-defined patterns in order to derive contracts on components which are verified at run-time depending on a set of event and states. Compared to ours, apart from implementations details (e.g., their approach integrate contract components in separate XML documents), their approach is more coarse-grained (works at the component level), focuses in self-adaptation, and does not deal with security. Close to our work from an implementation point of view, in (Hallstrom et al., 2004) the authors propose an AoP approach to monitor pattern contracts. Each pattern contract is associated with a dedicated pattern and is defined in an *aspect*. This *aspect* is used to monitor, at run-time, the applied pattern. Nevertheless, this approach is focused exclusively on the implementation without a will to provide abstraction related to the contract definition. Note that none of the aforementioned approaches focuses on security. In (Dikanski et al., 2012), the authors evaluate Spring Security in order to identify existing security patterns to provide a mapping with Spring security mechanisms. However, they do not provide any supporting mechanism for implementation or monitoring of the patterns.

6 CONCLUSION & FUTURE WORK

In this paper we have presented *Security Contracts*, a novel extension of the Design by Contract paradigm aimed at supporting security patterns. Our approach provides a mechanism for the specification of reusable and extensible abstract patterns, their deployment on host applications and their monitoring at run-time, in what we call, Execution under Contract. A prototype implementation for the Java ecosystem and its application to a case study involving the enhancement of the authentication mechanism provided by the Spring Security framework are presented as well. Concretely, we have shown how we can define the Authenticator pattern as a *Security Contract* in an abstract way, deploy it by the means of annotations in both new and existing applications and monitor and enforce its properties (including temporal ones) at run-time.

As future work we envision the exploration of the following research lines: 1) Pattern composition. We intend to investigate an extension of our framework in order to give support to the composition of patterns. This will enable the possibility of creating complex patterns as a composition of simpler, easier to verify ones. 2) Annotation enhancement. We aim to research the feasibility of the integration of a given (temporal) logic directly in the annotations used to deploy the pattern, so that the user can add/modify its properties.

ACKNOWLEDGMENTS

The authors thank Caine Silva and Henri Stoven for their help in this work.

REFERENCES

Behrens, A. (2018). What are security patterns? a formal model for security and design of software. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–6.

Cheng, B. H., Doherty, B., Polanco, N., and Pasco, M. (2019). Security patterns for automotive systems. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 54–63. IEEE.

Da Silva Júnior, L. S., Guéhéneuc, Y.-G., and Mullins, J. (2013). An approach to formalise security patterns. Technical report, Citeseer.

Dikanski, A., Steinegger, R., and Abeck, S. (2012). Identification and implementation of authentication and authorization patterns in the spring security framework. In *The Sixth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2012)*, pages 14–30.

Fernandez, E. B., Yoshioka, N., Washizaki, H., and Yoder, J. (2018). An abstract security pattern for authentication and a derived concrete pattern, the credential-based authentication. In *Asian pattern languages of programs conference (AsianPLoP)*.

Fernandez-Buglioni, E. (2013). *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons.

Guérin, S., Polet, G., Silva, C., Champeau, J., Bach, J.-C., Martínez, S., Dagnat, F., and Beugnard, A.

(2021). PAMELA: an annotation-based Java Modeling Framework. *Science of Computer Programming*.

Hallstrom, J. O., Soundarajan, N., and Tyler, B. (2004). Monitoring design pattern contracts. In *Proc. of the FSE-12 Workshop on Specification and Verification of Component-Based Systems*, pages 87–94.

Johnson, R., Hoeller, J., Donald, K., Sampaleanu, C., Harrop, R., Risberg, T., Arendsen, A., Davison, D., Kopylenko, D., Pollack, M., et al. (2004). The spring framework–reference documentation. *interface*, 21:27.

Meyer, B. (1992). Applying'design by contract'. *Computer*, 25(10):40–51.

Mongiello, M., Pelliccione, P., and Sciancalepore, M. (2015). Ac-contract: Run-time verification of context-aware applications. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 24–34. IEEE.

Ostroff, J. S., Torshizi, F. A., Huang, H. F., and Schoeller, B. (2009). Beyond contracts for concurrency. *Formal Aspects of Computing*, 21:319–346.

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., and Sommerlad, P. (2013). *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons.

Silva, C., Guérin, S., Mazo, R., and Champeau, J. (2020). Contract-based design patterns: a design by contract approach to specify security patterns. In *Proceedings of the The 6th International Workshop on Secure Software Engineering SSE@ARES*, pages 1–9.

Yoshioka, N., Washizaki, H., and Maruyama, K. (2008). A survey on security patterns. *Progress in informatics*, 5(5):35–47.