Modeling, Enforcing and Testing Secure Navigation Paths for Web Applications*

Marianne Busch[1], Martín Ochoa[2] and Roman Schwienbacher[1]

[1] Ludwig-Maximilians-Universität München
[2] Siemens AG, Germany

June 2013

Technical Report 1301
Version 1.0

Research Unit of Programming and Software Engineering (PST)
Institute for Informatics
Ludwig-Maximilians-Universität München, Germany

*This work has been supported by the EU-NoE project NESSoS, GA 256980.

Abstract

Although there exist robust solutions for managing authentication, authorization and session management in web applications, the question of effectively controlling the navigation flow for different users remains challenging. In this report we propose a methodology that allows one to specify Secure Navigation Paths (SNP) using UML models and automatically generate a server-side monitor enforcing such policies. We also discuss how those models can be used to generate tests in case a monitor is absent. We report on tool support for this methodology and on applications to a SmartGrid usage scenario.

Contents

1 Introduction

In the era of modern information technologies, where sensitive personal data is stored and managed over the Internet on databases or remote servers, software engineers are constantly faced with new and challenging tasks in the field of security and user guidance. In general, there exist security frameworks that provide means to enforce security and data protection for web applications: (1) *Authentication*, which is the process of proving the identity of a user to gain access to a protected resource; (2) *Authorization*, which is the process that determines what a subject (e.g., a user or a program) is allowed to access, especially what it can do with specific objects (e.g., files) [8]; (3) *Encryption*, which is the conversion of data into a format that cannot be understood by unauthorized subjects and (4) *Session management*, which is the process of keeping user-specific application data while a user is authenticated to the system. Examples of such security frameworks include Spring Security [5] and Apache Shiro[1]. Both were designed to provide a high standard of security as comfortably as possible.

However, there is one important aspect of security in the area of data protection which usually remains challenging: A kind of navigational access control which guarantees that users with a certain role has only a limited number of *Secure Navigation Paths* (SNPs) inside an application's context. Suppose a web application procedure managed to open an online-banking-account, which consists of about ten steps. What happens when a user, whether intentionally or not, jumps from step two, "indicating the personal data", to the last step, "confirmation", simply by calling the corresponding URL, and confirms the transaction? This can not only lead to fatal inconsistencies on the application state, but may also trigger a worst case scenario like losing money or seriously damaging the image of the application provider.

The contributions of this report are twofold: on the one hand we propose a methodology to define simultaneously RBAC and SNP policies for an application by *modeling* the integrity of its business workflows using UML-based Web Engineering (UWE) [12] and for *safeguarding* them at runtime by a monitor, which also enforces navigational access control. On the other hand we provide means for automatically *testing* SNP policies for applications not running an enforcing monitor based on UML policies.

The remainder of this report is structured as follows: Section 2 provides information about secure navigation paths and summarizes previous work on modeling web applications with UWE. Section 3 is the central section. It shows how to model SNPs with UWE and how to test them. Additionally, we present tool support developed to validate our approach. Section 4 illustrates a usage scenario from the SmartGrid domain. We review related work in Sect. 5 and conclude in Sect. 6 where we discuss limitations and future work.

2 Background

In this section we briefly recall the addressed challenge of enforcing secure navigation paths and the chosen modeling methodology, UML-based Web Engineering, which is part of our solution. UWE was chosen because it already enables the web developer to model web applications including security features such as access control or authentication.

2.1 Secure Navigation Paths

Among the most challenging web application vulnerabilities are the ones involving the misuse of the application logic itself. As stated by the Common Weakness Enumeration (CWE) [2]:

> Errors in business logic can be devastating to an entire application. They can be difficult to find automatically, since they typically involve legitimate use of the application's functionality.

Exploiting flaws in the business workflow is a common attack to the application logic. These exploits typically consist of jumping to certain URLs, bypassing critical controls of an intended flow or manipulating the parameters of legal requests. The consequences of those attacks can be diverse: bypassing log-in controls result in authentication breaches whereas skipping certain controls in a trading operation might result in monetary benefits to the attacker. Examples of documented vulnerabilities in popular web applications include the Yahoo SEM Logic Flaw [7]: if one deposited USD $30 into an advertising account, Yahoo would then add an additional USD $50 to that account. The sign-up process was able to be circumvented such that failing to deposit the USD $30 still allowed to receive the additional USD $50. Other examples include bypassing of age restrictions in Youtube, access to private photos in MySpace (resulting in attacks to celebrities) among others (see [3]).

In recent years, much attention has been given to validating user input to web applications to prevent code-injection (i.e. SQL, XSS), but very few tools and methodologies are available to prevent and test logical errors (some attempts include [16, 23]). This is due to the almost endless possible logical flaws that could be present on an application ranging from obvious to very subtle coding vulnerabilities. In this report we focus on one of the most commonly abused logic vulnerabilities: the *integrity* of the navigation paths as intended by the application owner, that is, the order in which authorized resources of an application should be accessed by a given user role.

2.2 UWE

In the following, we outline UML-based Web Engineering (UWE) [12], the security-aware engineering approach we have chosen for modeling web applications.

One of the cornerstones of the UWE language is the "separation of concerns" principle, which is implemented by using separate models for different views. However, we can observe that security features are cross-cutting concerns which cannot be separated completely:

The Requirements Model defines (security) requirements for a project.

The Content Model contains the data structure used by the application.

The UWE Role Model describes a hierarchy of user groups to be used for authorization and access control issues. It is usually part of a *User Model*, which specifies basic structures, as e.g., that a user can take on certain roles simultaneously.

The Basic Rights Model describes access control policies. It constrains elements from the *Content Model* and from the *Role Model*.

The Presentation Model sketches the web application's user interface.

The Navigational State Model defines the navigation flow of the application and navigational access control policies. The former shows which possibilities of navigation exist in a certain context. The latter specifies which roles are allowed to navigate to a specific state and the action taken in case access cannot be granted. In a web application such actions can be, e.g., to logout the user and to redirect to the login form or just to display an error message. Furthermore, secure connections are modeled here.

For each view, an appropriate type of UML diagram is used, e.g., a state machine for the Navigational Model. In addition, the UWE Profile adds a set of stereotypes, tag definitions and constraints, which can be downloaded from the UWE website [20]. Further details of our modeling approach can be found in the following section.

3 Secure Navigation paths with UWE

In this section we propose a solution for controlling the intended navigation paths in web applications. The idea is to specify navigation policies by means of UML state-chart diagrams (Sect. 3.1). Each node in the state machine represents a basic interaction step, i.e., (a part of) a web page. By using the UWE approach, it is also possible to annotate transitions with necessary permissions required to access a resource using a Role-Based Access Control (RBAC) scheme. In this way, a monitor can be automatically generated that enforces the desired navigation policies for multiple roles, separating the navigation control from the application itself. We also discuss how to automatically generate tests for a given policy in case the monitor is absent, as is the case for legacy applications (Sect. 3.2). In Sect. 3.3 we present tools we have written for (1) supporting the user while modeling SNPs; (2) for exporting graphical model of SNPs as text; (3) for monitoring web applications and (4) for testing them.

3.1 Modeling Approach

We focus on how to model basic SNPs, on a notation for specifying parameters for web pages and on the relation of SNPs and RBAC.

Basic SNPs. As introduced in Sect. 2.2, UWE provides several views. For each view, an appropriate UML diagram is selected. In UWE, UML state machines are used to model the navigational structure of a web application. In our case, states correspond to navigational nodes that are implemented as web pages. Transitions define all possibilities to navigate from one page to another and thus specify a policy for the navigation.

Building on UWE's Navigational State Model, we define SNPs as follows: in case a transition leads from state A to B, a user can visit page B after having visited page A. If it should be possible to go back (e.g. with the back-button in

the browser), another transition has to connect B with A. For more than one option, an arbitrary number of transitions can be used.

We use the behavior of UML composite states to model links which should be available within a certain area at any time. "Composite" means states can be nested within a composite state. If state Y and Z and an initial node are nested inside a state A and the initial node is connected to Y, then transitions to A activate Y (because of the inner initial node). A transition that leads from A to an arbitrary new state B can be fired from inside A, no matter if Y or Z is active. Consequently, A does not correspond to a web page itself, but groups others. An example of a UWE navigational diagram expressing SNPs is depicted in our case study (Sect. 4, Fig. 5).

In this way we model SNPs at a high level of abstraction, as no technical details have to be given.

SNPs with Checked Parameters. However, many web pages use parameters to pass, e.g., user input or session IDs to the next page. To model allowed parameters specifically, we extend UWE's Navigational State Model so that a minimum of technical information can be specified, if needed. Our extension is inspired by Braun et al. [10], who came up with a textual control-flow definition language for MVC (Model-View-Controller)-based web applications. As our approach does not specify method names, but page names, it is not restricted to MVC-based applications. Furthermore, information about allowed parameters can easily be added to existing Navigational State diagrams, so that related information about authentication, secure connections, navigational control as well as SNPs can be overseen immediately. We add parameters to transitions using a guard like [param = GET(par1:type1, par2:type2)]. GET or POST are allowed and types can be bool, numeric or string (c.f. part *a* of Fig. 1).

Sometimes, a parameter should be added to requests within a certain area, using the same value as at the first occurrence. For instance, a session ID is not allowed to change during a session and selected items should not change during the payment process. This can be modeled by a composite state which comprises all transitions that should use fixed parameters. All navigational states in UWE inherit from the stereotype ≪navigationalState≫, for which a tag called {fixedParam = POST(par1:type1, par2:type2)} can be set. Global parameters are applied to all transitions where the target state is located within the composite state. The choice of GET / POST for the composite state and affected transitions has to be coherent in case inner transitions specify further parameters. When leaving and entering the composite state again, the values of the fixed parameters can of course be different than before.

Part *b* of Fig. 1 depicts this behavior: the bold transitions inherit the fixed parameter. This means the value of *item* is set by the transition targeting `BuyEnergy`. Afterwards, it cannot be changed until the `OrderProcess` is left. The stereotype ≪collection≫ denotes that several `Offers` are shown. Each offer is of the type `EnergyOffer`, which is defined by the tag {itemType}. If an offer is bought, the confirmation cannot be shown for another one.

For simplicity, in the rest of this report we focus on modeling and enforcing SNPs on web applications without imposing constraints on the parameters. This is reasonable since important security parameters such as the session ID are already handled automatically by development frameworks.

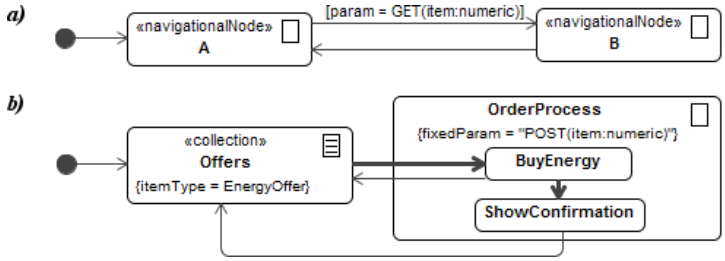

Figure 1: Example of expressing SNPs with UWE

Relation of SNPs and RBAC. Within UWE, modeling SNPs can easily go hand in hand with modeling RBAC, where RBAC is twofold: on the one hand we specify navigational access control, i.e., the behavior of a web application if an unauthorized user accesses a web page. On the other hand, we express access control on classes that are used in the implementation.

We also use the Navigational State Model of UWE to specify *navigational access control*. For each ≪session≫ stereotype, denoting a user's session, a tag called {roles} can point to a set of roles from UWE's Role Model. In order to be able to access the web page represented by a state, a user has to have at least one of the roles that are allowed to access this state. If this is not the case, the tag {unauthorizedAccess} specifies which state should be used instead. This state can then represent, e.g., a page with an error message or an advertisement for a more expensive account.

In order to provide a full picture of access control in UWE, we briefly introduce how to model *RBAC* on data objects with UWE. For the Content- and the Basic Rights Model UML class diagrams are used. In the Content Model, classes and their relationships are defined. These classes can then be reused in the Basic Rights Model, where tagged UML dependencies connect role instances to them for modeling RBAC (an example is shown for our case study in Sect. 4, Fig. 4). These dependencies can be tagged by ≪create≫, ≪readAll≫, ≪updateAll≫ or ≪delete≫, which represents the common CRUD functions. For example, ≪updateAll≫ means that a role can update all attributes of a class. Dependencies can also directly point to an attribute (≪update≫, ≪read≫) or to a method (≪execute≫). Generally, the Basic Rights Model is equally expressive as SecureUML [21], while the representation is less bulky, as discussed in [12].

In theory, it is possible to export both kinds of access control to XACML (eXtensible Access Control Markup Language), as described in [14]. However, up to now only the access control on data objects has been explored in practice. Our approach complements this by using a monitor on the server side, which enforces not only SNPs but also navigational access control.

3.2 Testing

In this subsection, we describe how to generate navigational test cases for web applications with access control policies regarding SNPs. The main goal is to cover every possible navigation context considering navigation history, current navigation node, user role and access permission result. Comparing these results with given access control policies, we can detect possible access control

misbehavior issues.

In order to build navigational test cases we use the following approach based on an UWE SNP policy given input: for each user role we walk through every available SNP starting at the entry node which is marked as *isHome*. We navigate from node to node inside the SNP until we come back to an already visited node. This way we can test if the application behaves according to the specification (we can identify false negative access control behavior by collecting occurring access denials). In order to detect possible violations of the integrity of the policy, we simply try to leave the SNP on every node by requesting each node which is currently not accessible by an outgoing transition and thus not allowed. In this context, every granted access represents an incorrect behavior. Figure 2 depicts such a navigation through a SNP including illicit node requests on every node.

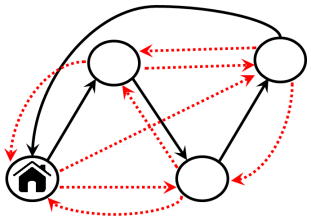

Figure 2: Test generation: Follow SNP and request illicit nodes

However, with every violating request we cause a navigation history which does not correspond to the original SNP, and we possibly harm the state of the application. Therefore, we need to reconstruct the previous SNP-valid navigation context to go ahead: First, we have to fall back to the current entry node to clear the navigation history. Second, we have to repeat the navigation progress on the SNP until we achieve the previously visited node inside the SNP to reconstruct the navigation context.

This testing process is efficient, since it has a complexity of $\mathcal{O}(rn^3)$. Assuming there are n possible navigation nodes and r user roles: Every navigation node has up to $n-1$ neighbors which are not accessible by an outgoing transition. By testing all roles, we get an amount of $rn(n-1)$ test cases which gives an upper bound of $\mathcal{O}(rn^2)$ tests. Considering the backtracking behavior to reconstruct the navigation context we have to visit up to $n-1$ additional nodes for every forbidden node. Consequently, we get a final complexity of $\mathcal{O}(rn^3)$. However, testing parameters cannot be exhaustive, as parameters can contain arbitrary values. Extending our approach to consider constraints on parameters is thus left as future work.

3.3 Tool Support

This subsection presents tool support that we developed to validate our approach. Only the tool *MagicUWE* existed before, which is used to model UWE diagrams. We have implemented *MagicSNP* to export navigational access control rules that can be enforced by our *SNPmonitor*. For tests, our *SNPpolicyTester* is employed.

MagicUWE. UWE models can be built using any UML CASE tool that enables the import of UML profiles. We use the *MagicUWE* plugin [13] implemented for MagicDraw that provides additional support for the developer so that repetitive tasks can be avoided. Thus, instead of creating a basic element, as a class, and applying a stereotype to it, UWE's stereotyped elements can be inserted directly from a toolbar. Besides, transformations between UWE models can be performed semi-automatically.

MagicSNP. In order to validate a UML Navigational State Model and moreover to extract the corresponding access control semantics we developed a CASE tool plugin for MagicDraw called *MagicSNP*. By iterating through all hierarchical states and analyzing incoming transitions, state names and tags, our tool fetches relevant information about navigational access control and SNPs. The JSON-structured result can be taken as input for a security framework for a specific web application. In our case, the exported rules are read by the server-side monitor. An example of a result file can be found in Sect. 4.

SNPmonitor. The *SNPmonitor* is our generic monitor module approach which provides RBAC with SNPs for web applications considering modeled access control semantics. Basically, this module is responsible to decide whether or not a user is allowed to get access to a protected resource. The decision making is based on the web user's session information (e.g., previously visited location, assigned user roles etc.) and a policy file (e.g., generated by *MagicSNP*). In order to ensure robustness, our monitor module also handles any kind of access constraint violation: the web user is redirected to a corresponding error page including an appropriate error message with possibility to go back to its previously visited navigation context.

Technically, our SNPmonitor is implemented as a Java EE application, using the Spring Framework. [5] The code of a client application which should be safeguarded does not have to be touched, the monitor just has to be added as a filter to the Java EE deployment descriptor. Using a URL-pattern, it is also possible to shield a certain part of a web application, e.g., web pages stored in a `protected/*` directory.

SNPpolicyTester. In order to test already defined SNP policies for a specific web application, as mentioned in Sect. 3.2, we developed a testing tool called *SNPpolicyTester*. It parses a given security policy file and searches for false positive and false negative access control behavior. Therefore, it navigates through every available SNP with every defined user role trying to leave the SNP on each navigation node by requesting every possible illegal node in this context. As a result, we get a detailed log file which allows a quick identification of traces that are possible although they should be prohibited.

Additionally, we analyzed the runtime performance of our testing tool using a benchmark client based on the TPC-W Benchmark[6]. Consequently, we are able to compare the result with the complexity of our test generation process according to Sect. 3.2: Fig. 3 depicts the average result of benchmarks we performed with two user roles regarding ten, twenty, thirty and forty navigation nodes. The result corresponds to the expected complexity of $\mathcal{O}(rn^3)$.

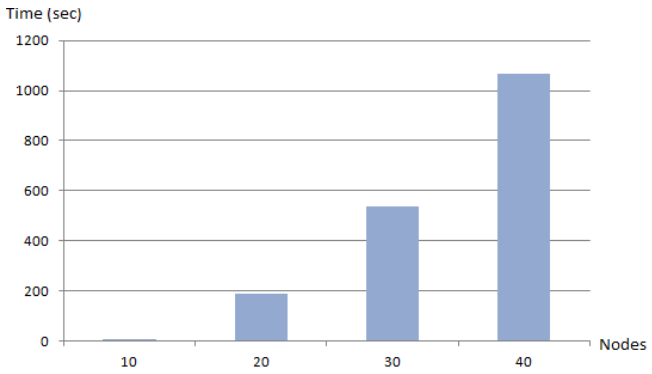

Figure 3: SNPpolicyTester: Average benchmark result

4 Case Study

Smart grids use information and communication technology (ICT) to optimize the transmission and distribution of electricity from suppliers to consumers, allowing smart generation and bidirectional power flows – depending on where generation takes place. With ICT the Smart Grid enables financial, informational, and electrical transactions among consumers, grid assets, and other authorized users [24]. The Smart Grid integrates all actors of the energy market, including the customers, into a system which supports, for instance, smart consumption in cars and the transformation of incoming power in buildings into heat, light, warm water or electricity with minimal human intervention. Smart grid represents a potentially huge market for the electronics industry [26]. Two basic reasons why the attack surface is increasing with the new technologies are: a) The Smart Grid will increase the amount of private sensitive customer data available to the utility and third-party partners and b) Introducing new data interfaces to the grid through meters, collectors, and other smart devices create new entry points for attackers. For a more detailed discussion on security issues arising in this context see [15]. See also [18] for a current version of proposed technologies to solve this power systems management and associated information exchange issues. In the following we model a scenario in this domain, the *SmartGrid Bonus Application*.

Basically, our *SmartGrid Bonus Application* represents a prototype of an energy offer management including optional bonus handling. It provides two different user roles namely *Provider* and *Customer*: Providers manage and sell energy packages including optional bonus programs for customers. Customers have the possibility to buy offered energy packages. Therefore, our application lists all available energy offers and the customer selects a specific offer which includes a bonus code. After buying an energy package, the application shows the corresponding bonus code which contains a gift voucher, e.g., for online shops. Finally, the customer gets a confirmation for the ordered energy.

In order to model the data structure managed by our case study, we use UWE's Content Model. Basically, it comprises two domain classes, `EnergyOffer` and `BonusProgram`, which are also used in Fig. 4. An instance of the class `EnergyOffer` represents a specific energy offer launched by an energy provider including start and end date. Each object of `EnergyOffer` can include an arbi-

trary number of `BonusProgram` instances. A `BonusProgram` instance stands for an additional bonus customers get, after they have bought the corresponding `EnergyOffer`.

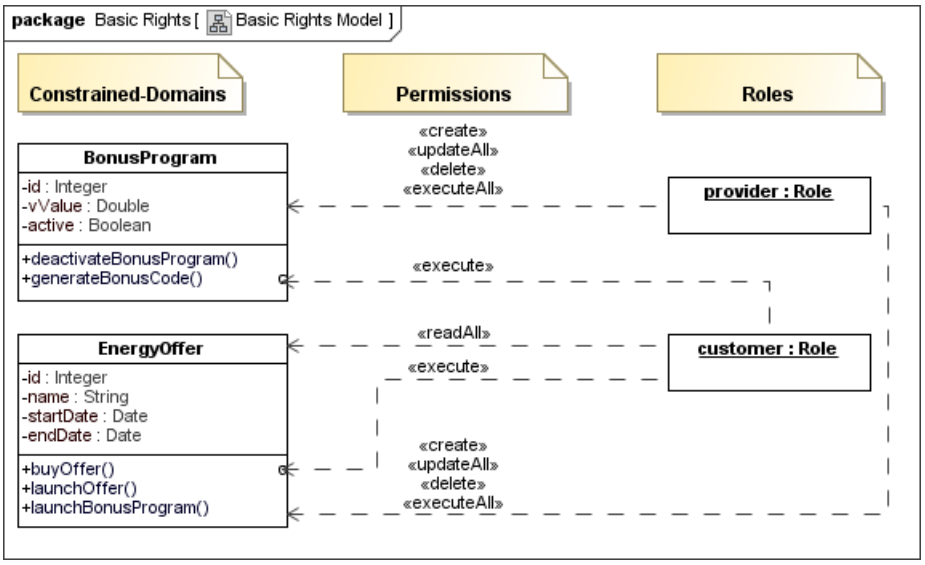

Figure 4: SmartGrid Bonus Application: *Basic Rights Model*

In order to model RBAC constraints we use UWE's *Basic Rights Model*, depicted in Fig. 4. Basically, it uses classes of the *Content Model* on the left-hand side in combination with user roles on the right-hand side. Access permissions were defined by stereotyped dependencies: for our application, a provider has no restricting constraints. By contrast, there is only a limited set of permissions for users taking on the role of a customer: they are only allowed to read instances of the class `EnergyOffer` and to call the methods `buyOffer()` and `generateBonusCode()`. These permissions represent the basis for a customer to list all available energy offers, to buy a specific offer and to eventually get a bonus code. In order to define constraints like "a customer can only get access to a bonus code after he bought an energy package" we now have to define navigational access control policies using SNPs.

Therefore, we use UWE's Navigational State Model as described in Sect. 3.1. For our web application, the navigational structure should start at a login page. After completing the authentication successfully, customers should be redirected to an internal page where they can have a look at a list of offers. If they decide to accept an offer they have to give their consent, before a confirmation is shown. In case the energy offer was connected to a bonus program, a page containing the bonus code is displayed before the final confirmation.

Notice that for our case study we make the assumption that names of pages correspond to names of states and we do not model parameters – our monitor then just forwards given parameters, if any.

Figure 5 depicts our Navigational State Model which contains the following information: The outermost state `SmartGridBonusApplication` stands for the whole application. The tag `transmissionType="cif"` sets the overall type of

data transmission during the session to `cif`, which stands for for confidentiality, integrity, and freshness. Thus, the implementation should prevent eavesdropping, replaying, or altering transmitted data. As we can see, navigational nodes, represented by the states on the innermost level, are grouped by three main areas or parent states: `LoginArea`, `ProviderArea` and `CustomerArea`.

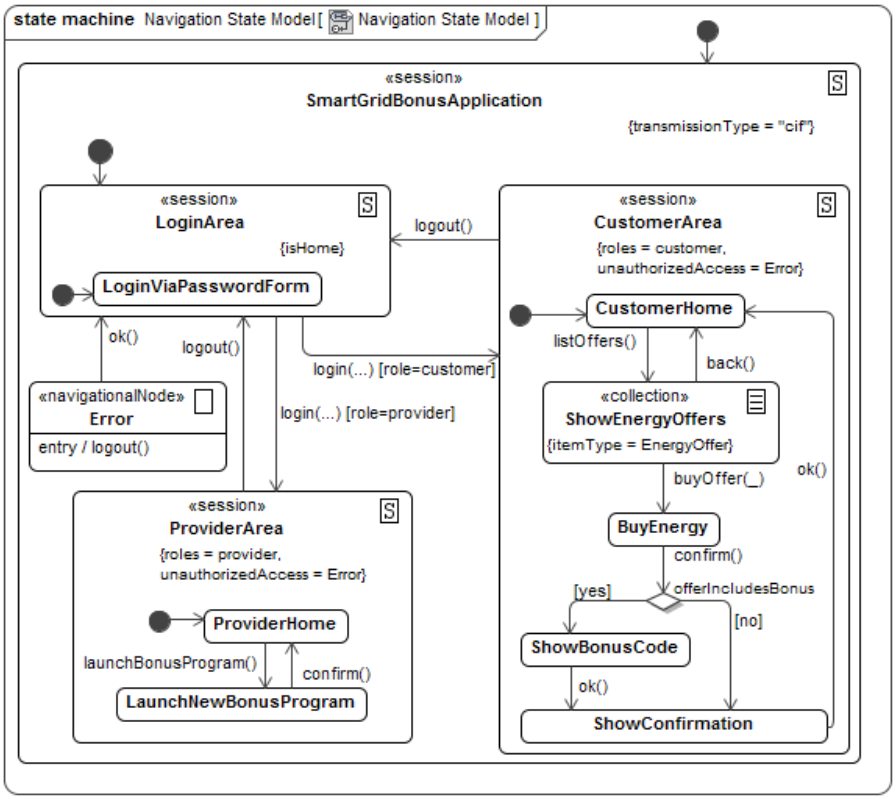

Figure 5: SmartGrid Bonus Application: *Navigation State Model*

Every web user can access the login context node `loginViaPasswordForm` which is inside the `LoginArea` indicated by the {isHome} tag. Inner states are tagged by {unauthorizedAccess=Error} which represents the default violation node. Logged in users with the role `customer` can access the whole `CustomerArea` indicated by the inherited {roles} tag. In addition, they must follow the SNPs as defined by the transitions between the navigation states to be allowed to request a protected node. This means, e.g., to get access to `showBonusCode` the user has to be associated to the role `customer` and he must have been on `buyEnergy` right before. In order to get access to `customerHome` the user needs to have the same role but must have been on one of the nodes `loginViaPasswordform`, `showEnergyOffers` or `showConfirmation` right before and so on. Otherwise, the user gets redirected to the `error` state as defined in the tag {unauthorizedAccess}. Each user which enters the constraint violation node `error` gets logged out automatically as indicated by the entry event `entry / logout()`.

SNPs for providers are modeled in an analogous manner as depicted in the lower-left corner of Fig. 5.

Listing 1 shows an excerpt from the navigation rule file which is generated by our tool *MagicSNP* from the state machine shown in Fig. 5.

It contains information for a generic monitor to provide navigational access control including SNPs for a web application: The default violation node is `error`, defined by the attribute `default_violation` on the outermost level. Furthermore, every single location entry holds the corresponding violation node and a list of access rules. Each rule entry represents a user role that is allowed to access the current navigation node. In addition, the attribute `pre_visited` specifies navigation nodes the user is allowed to come from.

This rule file can be imported in our *SNPmonitor* as well as in our *SNPpolicyTester*.

```
navigation.file={
  "_comment":"Build time: 10.10.2012 11:12:38",
  "application":"SmartGridBonusApplication",
  "locations":[...
    {"location":"showEnergyOffers","violation":"error","home":false,
     "rules":[{"role":"customer",
               "pre_visited":["customerHome"]}]},
    {"location":"buyEnergy","violation":"error","home":false,
     "rules":[{"role":"customer",
               "pre_visited":["showEnergyOffers"]}]},
    {"location":"showBonusCode","violation":"error","home":false,
     "rules":[{"role":"customer",
               "pre_visited":["buyEnergy"]}]},
    {"location":"customerHome","violation":"error","home":false,
     "rules":[{"role":"customer",
               "pre_visited":["showEnergyOffers",
                              "showConfirmation",
                              "loginViaPasswordForm"]}]},
    {"location":"showConfirmation","violation":"error","home":false,
     "rules":[{"role":"customer",
               "pre_visited":["buyEnergy",
                              "showBonusCode"]}]},
    ...],
  "default_violation":"error"}
```

Listing 1: Excerpt from extracted JSON-structured navigation rule file

5 Related Work

In this section we firstly introduce approaches in the area of SNPs, which are related to business workflow integrity. Secondly, we briefly introduce alternatives for using UWE.

As already mentioned, [10] recently published a robust approach for SNPs for MVC-based web applications where policies are specified using an ad-hoc textual notation. They also tackle race-conditions and handling of multiple tabs within a browser, which is currently outside of the scope of our approach. The parameter constraint in our approach was partially inspired by their textual policy language, although our approach is mainly based on web pages, not on methods. In UWE, some problems are inherently solved, as e.g., superstates exist so that all transitions can be easily specified and there is no need to invent extra notations for the ability to change decisions later or for the availability of the back button.

In 2002, Scott et al. [27] described a system which is also based on a solution

using a monitor. A textual policy specifies validation constraints, mainly for parameters and cookies, in a language called Security Policy Description Language (SPDL). This policy is then compiled to code which is executed by the monitor when a page is accessed. Additionally, Message Authentication Codes (MACs) can be added by the monitor when delivering a page so that, e.g., hidden form fields can be secured from changes at the client-side.

Halle et al. [17] define a navigation state machine with session traces with a focus on a formal model. However, the state machine is only a simple one with no further information than a sequence of states which does not include parallel states. If desired, their formal approach might be extended to describe UWE's navigational states model, including information given by the stereotypes, tags and parallel states.

In [23] a method for secure design of business application logic is sketched. It comprises strategies such as analyzing weaknesses caused by misconfiguration of server-side components or by errors in the application logic. They suggest to test several kinds of parameters, however they do not provide tool support. Furthermore, it is recommended to define a clear design of the architecture, especially for components which update session data. The authors aim to provide a good practice which certainly can be combined with our approach.

Additionally, a tool for servlet-based web applications is provided by Felmetsger et al. [16]. The tool, called *Waler* uses a composition of dynamic analysis and symbolic model checking: Regarding the dynamic analysis, it observes the normal operation of a web application in order to infer behavioral specifications that are filtered to reduce false positives. Afterwards, symbolic model checking is used to identify program paths that are likely to violate these specifications. Compared to our approach, models do not have to be created manually, which is convenient, especially for legacy applications. However, the price is that flaws in the navigation paths can only be detected with a certain possibility. SPaCiTE [11] is a tool that generates concrete attack tests based on model checking and mutation operators. It has been applied so far for testing RBAC and XSS, but not for business workflow integrity.

To the best of our knowledge, no web framework exists that provide mechanisms to specify rules for SNPs externally, i.e., without diving into the implementation of the web application. Even frameworks that focus on security, as e.g., Apache Shiro Web-Features [1], Spring Security [5] or jGuard Web [4] do not tackle the issue of SNPs.

For this work we have used UWE. Other web engineering methods do not include a navigational model and security aspects.

WebML [25] offers a so called hypertext model, but it is less fine grained than UWE's Navigational State Model so that SNPs cannot be ensured. Furthermore, WebML includes no navigational access control. ActionGUI [9] is an approach for generating complete, but simplified, data-centric web applications from models. It provides an OCL specification of all functionalities, so that navigation is only modeled implicitly by OCL constraints. Unfortunately, it would be difficult to model SNPs with those constraints, because this would require to model the whole web application, which can be tedious. SecureUML [21] is a UML-based modeling language for secure systems. A dialect for classes (called components) provides modeling elements for RBAC which are as expressive as UWE's Basic Rights Model. A similar approach is UACML [28] which also comes with a UML-based meta-metamodel for access control, which can be

specialized into various meta-models for, e.g., RBAC or mandatory access control (MAC). Conversely to UWE, the resulting diagrams of SecureUML and UACML are overloaded, as SecureUML uses association classes instead of dependencies and UACML do not introduce a separate model to specify user-role hierarchies. UMLsec [19] provides a UML extension with emphasis on secure protocols. Similarly, SecureMDD [22] allows one to define security properties in UML diagrams and generate code focusing primarily on security protocols for smartcards.

6 Conclusions

It is common to restrict users of web applications to a certain workflow in order to guarantee that the single business interaction steps and their order are respected. Unexpected workflows can lead to security issues, especially when the application does not properly handle those exceptions. In this report we described how to model Secure Navigation Paths for web applications with UWE's Navigational State Model. UWE diagrams can then be exported using a plug-in called MagicSNP for the CASE tool MagicDraw.

On the one hand we use the exported information as input for a monitor, which assures that only proper sequences of requests reach the application. On the other hand a test suite can be generated and executed in order to find unnoticed flaws in the state handling existing applications that do not have a monitor.

Furthermore, our case study shows the reliability of the proposed approach and illustrates its simplicity. At runtime the monitoring module has proven to scale well, as a result of using simple data structures and efficient process cycles throughout the module. The clear error message handling contributes to the robustness and user-friendliness of the web application.

Future work includes modeling finer grained SNP policies, by restricting the internal flow of single dynamic pages, as one can think of workflow vulnerabilities present in single dynamic pages that cannot be avoided with our methodology. This future goal could be achieved for example by taking advantage of the hierarchical nature of UML state-charts. Additionally, we will work on a larger case study where parameters, passed between web pages, are restricted according to our modeling approach.

References

[1] Apache Shiro. http://shiro.apache.org/.

[2] CWE-840: Bussiness Logic Errors. http://cwe.mitre.org/data/definitions/840.html.

[3] Insufficient Process Validation, WASC Threat Classification. http://projects.webappsec.org/w/page/13246943/Insufficient%20Process%20Validation.

[4] jGuard. http://jguard.xwiki.com/.

[5] Spring Security. http://static.springsource.org/.

[6] TPC-W Benchmark. http://tpc.org/tpcw/.

[7] Yahoo SEM Logic Flaw. http://ha.ckers.org/blog/20080616/yahoo-sem-logic-flaw/.

[8] R. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd edn.* Wiley, Chichester, 2008.

[9] D. Basin, M. Clavel, and M. Egea. Automatic Generation of Smart, Security-Aware GUI Models. In *ESSoS*, LNCS 5965, pages 201–217. Springer, 2010.

[10] B. Braun, P. Gemein, H. P. Reiser, and J. Posegga. Control-flow integrity in web applications. In *ESSoS*, pages 1–16, 2013.

[11] M. Buchler, J. Oudinet, and A. Pretschner. SPaCiTE – Web Application Testing Engine. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:858–859, 2012.

[12] M. Busch, A. Knapp, and N. Koch. Modeling Secure Navigation in Web Information Systems. In J. Grabis and M. Kirikova, editors, *10th International Conference on Business Perspectives in Informatics Research*, LNBIP, pages 239–253. Springer Verlag, 2011.

[13] M. Busch and N. Koch. MagicUWE - A CASE Tool Plugin for Modeling Web Applications. In *Gaedke, M., Grossniklaus, M., Diaz, O. (eds.) ICWE 2009. LNCS*, volume 5648, pages 505–508. Springer, Heidelberg, 2009.

[14] M. Busch, N. Koch, M. Masi, R. Pugliese, and F. Tiezzi. Towards model-driven development of access control policies for web applications. In *Model-Driven Security Workshop in conjunction with MoDELS 2012*. ACM Digital Library, 2012.

[15] J. Cubo, J. Cuellar, S. Fries, J. A. Martín, F. Moyano, G. Fernández, M. C. F. Gago, A. Pasic, R. Román, R. T. Dieguez, and I. Vinagre. Selection and Documentation of the Two Major ApplicationCase Studies. NESSoS deliverable D11.2, 2011.

[16] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, page 10, Berkeley, CA, USA, 2010. USENIX Association.

[17] S. Hallé, T. Ettema, C. Bunch, and T. Bultan. Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 235–244. ACM, 2010.

[18] International Electrotechnical Commission (IEC). IEC 62351 Parts 1-8, Information Security for Power System Control Operations.

[19] J. Jürjens. *Secure Systems Development with UML*. Springer, 2005.

[20] LMU. Web Engineering Group. UWE Website. `http://uwe.pst.ifi.lmu.de/`.

[21] T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *Proc. 5th Int. Conf. Unified Modeling Language (UML'02)*, volume 2460 of *Lecture Notes in Computer Science*, pages 426–441. Springer, 2002.

[22] N. Moebius, K. Stenzel, H. Grandy, and W. Reif. SecureMDD: A Model-Driven Development Method for Secure Smart Card Applications. In *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, pages 841 –846, march 2009.

[23] F. Nabi. Designing a secure framework method for secure business application logic integrity in e-commerce systems. *I. J. Network Security*, 12(1):29–41, 2011.

[24] National Energy Technology Laboratory. A vision for the smart grid. Report, June 2009. `http://www.netl.doe.gov/moderngrid/`.

[25] R. Rodriguez-Echeverria, J. M. Conejero, P. J. Clemente, M. D. Villalobos, and F. Sanchez-Figueroa. Generation of webml hypertext models from legacy web applications. In *14th IEEE International Symposium on Web Systems Evolution (WSE)*, pages 91–95, 2012.

[26] R. Schneiderman. Smart grid represents a potentially huge market for the electronics industry. *IEEE Signal Processing Magazine*, 27(5):8–15, 2010.

[27] D. Scott and R. Sharp. Abstracting application-level web security. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 396–407. ACM, 2002.

[28] N. Slimani, H. Khambhammettu, K. Adi, and L. Logrippo. UACML: Unified Access Control Modeling Language. In *NTMS 2011*, pages 1–8, 2011.