Modeling and Analyzing Security Patterns Using High Level Petri Nets

Xudong He
School of Comp. and Inf. Sciences
Florida International University
Miami, FL 33199, USA

Yujian Fu
Department of EECS
Alabama A & M University
Normal, AL 35762, USA

Abstract *– Security has become an essential and critical non-functional requirement of modern software systems, especially cyber physical systems. Security patterns aim at capturing security expertise in the worked solutions to recurring security design problems. This paper presents an approach to formally model and analyze six security patterns to detect potential incompleteness, inconsistency, and ambiguity in the textual descriptions; and to prevent their incorrect implementation. These patterns are modeled using high level Petri nets in our tool environment PIPE+. Simulation is used to analyze various security relevant properties. The validated formal models of individual security patterns serve as the building blocks for system design involving the composition of multiple security patterns.*

Keywords *– security patterns, formal modeling, high level Petri nets, validation, simulation*

I. Introduction

Security has become an essential and critical non-functional requirement of modern software systems, especially cyber physical systems. However software developers often lack a deep understanding of system security issues and their proper solutions. Security patterns, evolved from general software design patterns, aim at capturing security expertise in the worked solutions to recurring security design problems. In the past decade, many security patterns and their description methods were proposed [1]. A comprehensive repository containing 26 (13 structural and 13 procedural) security patterns with 3 mini patterns was compiled in [2]. This repository collects some best known security patterns and provides an excellent starting point for developing secure software systems. The structural patterns contain descriptions of the structure and interactions of security assurance mechanisms, while the procedural patterns provide general guidelines for developing secure systems. Thus our focus is on structural patterns. Each structural pattern has a textual description with some graphical illustration to aid understanding. However none of the structural security patterns is formally modeled and analyzed since the majority of these structural patterns only contain generic descriptions of general guidelines and thus are not suitable for precise definitions. However some pattern descriptions contain enough details for formal modeling. This paper presents an approach to formally model and analyze six of the above security patterns to detect potential incompleteness, inconsistency, and ambiguity in the text descriptions; and as a result to prevent their incorrect implementation. The validated formal models of individual security patterns serve as the building blocks for system design involving the composition of multiple security patterns.

II. Related Work

Despite many proposed security patterns and their descriptions in the past decade, very few work addressed the formal modeling and analysis of security patterns.

In [3], software design pattern template was adapted to describe 8 security patterns (1 creational, 2 behavioral, and 5 structural). The security patterns were modeled using UML diagrams, which are translated into Promela programs using tool Hydra [4]. The security constraints defined in linear time temporal logic formulas are checked using SPIN. A process of using the tool chain was presented and demonstrated through a simple example on check point pattern. While using UML diagrams can be helpful for implementation, they lack precise semantics and thus the analysis results based UML model translations may not be convincing. Furthermore, no detailed model checking result was given. In [5], a formal modeling approach for composing security patterns for web-based applications was proposed. This approach uses UML to model the security patterns and their composition and then transforms the UML model to Alloy (a formal specification language based on first order logic) formal specification for security property analysis. A case study of an on-line banking system was given, which involves five software security patterns: single sign on, check point, authenticator, policy, and secure proxy. No details of transforming a UML model to Alloy specification were given except some generic mappings. In [6], an approach of translating UML sequence diagrams to process algebra CCS was given. Two case studies were given, one involving the composition of secure pipe, authentication enforcer, and observer patterns and the other involving web security

DOI reference number: 10.18293/SEKE2016-010

pattern with third party brokered authentication. The behaviors of these compositions were first given in sequence diagrams, which were manually translated into CCS expressions. These CCS expressions serve as the behavior models. Various properties are defined using GCTL, which are model checked against CCS expressions using model checker CWB-NC. Unfortunately, the CCS expressions only capture the message names and control flows without considering the data processing and data flow; thus do not accurately reflect the complexity of real applications.

The above works only focus on the composition of security patterns while ignore the internal processing of individual security patterns. Our work focus on creating formal models from the textual descriptions of the internal processing of individual security patterns, and can be easily adapted to generate formal models from sequence diagrams representing security pattern composition.

III. Modeling Security Patterns

A. High Level Petri Nets

A high level Petri net [7] has a net structure $N = (P, T, F)$, which is a finite bipartite graph consisting of two kinds of nodes $P \cup T$, and the set of directed edges $F \subseteq P \times T \cup T \times P$. P is called the set of places, which are visually represented by circles; and T is called the set of transitions, which are visually represented by bars or boxes. The net structure (P, T, F) defines the *syntax* of a Petri net and models the control structure of a system. To define the static semantics, we need several semantic domains. First, we need a semantic domain of *Types*, which defines what are allowable tokens in each place. Elements of Types can be simple or composite, including Cartesian products and power set constructions. Second, we need a semantic domain *Labels*, which define permissible token flows. A label may contain variables to be instantiated under dynamic semantics. Third, we need a semantic domain of *Formulas*, which define the pre-conditions and post-conditions of transitions. A formula may contain variables to be instantiated under dynamic semantics. Finally, we need a semantic domain *Tokens* to define all valid tokens. The above semantic domains are defined in terms of an algebraic specification $Spec = (S, Op, Eq)$ with a family of sorts (types) S, the corresponding operations Op, and a set of equations Eq defining the meaning of the operations. Based on the above semantic domains, we can define the following semantic mappings:

(1) φ: $P \rightarrow Types$ associates each place p in P with a type in *Types*. In a HLPN net, places are often called predicates to highlight their roles as in predicate logic.

(2) L: $F \rightarrow Labels$ is a sort-respecting labeling of arcs.

(3) R: $T \rightarrow Formulas$ is a well-defined constraining mapping, which associates each transition t in T with a first order logic formula defining the meaning of the transition. Each formula $R(t)$ can be normalized into $Pre\text{-}cond(t) \wedge Post\text{-}cond(t)$. $Pre\text{-}cond(t)$ defines the selection criterion of incoming tokens and $Post\text{-}cond(t)$ defines the tokens to be produced.

(4) M_0: $P \rightarrow Tokens$ is a sort-respecting initial marking. The initial marking assigns a multi-set of tokens to each place p in P.

Thus a HLPN net is $PN = (P, T, F, Spec, \varphi, L, R, M_0)$ and its dynamic semantics is defined by all execution sequences $M_0[T_1/\alpha_1 > M_1[T_2/\alpha_2 > \cdots M_n[T_{n+1}/\alpha_{n+1} > \cdots$, in which each T_i is an execution step consisting of a set of non-conflict firing transitions. We have developed a tool environment PIPE+ [8] to create and analyze HLPNs. Our analysis techniques include simulation, model checking (SPIN), bounded model checking (Z3), and term rewriting (Maude). The integrations with external tools currently only support basic functionalities and are being enhanced. The latest source code is available at https://bitbucket.org/ptnet/pipe.

B. Security Patterns

Structural patterns contain descriptions of the structure and interactions of security assurance mechanisms, and thus they are suitable for formal modeling and analysis. In this paper, six structural patterns [2] are modeled and analyzed, including *account lockout*, *authenticated session*, *client data storage*, *encrypted storage*, *password authentication*, and *password propagation*. Many of these patterns involve encryption and decryption, session identification, web resource, and timing constraints such as duration and time out. We can only simplify the most of the above entities during the modeling without considering actual encryption and decryption algorithms and session identifier generation. We model a simple clock mechanism to deal with timing constraints. Most of these patterns contain some brief procedural descriptions of user and system interactions complemented with illustrative diagrams. Often the descriptions are not complete and not precise, which are common problems with natural language descriptions. During the modeling process, we need to make hidden assumptions explicit such as the uniqueness of session and user identifications. Although each model can be built with our tool PIPE+ in a few hours after carefully studying the requirements descriptions, it often takes several iterations to get them right, i.e. they capture the requirements correctly to the best of our understanding. However, we have not obtained the approval of the security domain experts.

Due to space limit, we only provide the detailed description of the *authenticated session* security pattern (the most complex pattern with a most complete procedural description among the six). The models of other patterns can be found at http://users.cis.fiu.edu/~hex/PIPE+.html.

Authenticated Session Pattern

An authenticated session allows a user to access multiple access-restricted pages on a website only authenticating once on the first page request. The process of authenticated session [2] contains the following steps:

Scenario 1 – Page request without valid authentication

(1) The client requests a protected page from the server, passing the session identifier;
(2) The underlying session mechanism retrieves the session data and invokes the protected page object;
(3) The protected page invokes the authentication checkpoint;
(4) The authentication checkpoint determines that the authenticated identity field is empty or that the last access time exceeds the session timeout window;
(5) The requested page URL and submitted data is stored in the session object as the page originally requested;
(6) The authentication checkpoint redirects the client to the login screen.

Login

The following interactions occur in the login procedure:

(1) The client requests the login page, submitting a username and password. The session identifier is passed as part of the request;
(2) The underlying session mechanism retrieves the session data and invokes the login page object;
(3) The login object validates the username and password;
(4) If unsuccessful, the unsuccessful login page ("try again") is returned to the user;
(5) If successful, the identity provided is stored in the authenticated identity field, and the current time in the last access time;
(6) The user is redirected to the page originally requested (stored in the session data).

Scenario 2 – Page request with valid authentication

The following interactions occur on a page request with valid authentication:

(1) The client requests a protected page from the server, passing the session identifier;
(2) The underlying session mechanism retrieves the session data and invokes the protected page object;
(3) The protected page invokes the authentication checkpoint;
(4) The authentication checkpoint validates the authenticated identity field in the session data and ensures that the last access time does not exceed the timeout period;
(5) The authentication checkpoint stores the current time as the last access time;
(6) The authentication checkpoint returns to the protected page object, which composes and delivers the requested page to the client.

Logout

The following interactions occur in the logout procedure:

(1) The client requests the logout page from the server, passing the session identifier;
(2) The underlying session mechanism retrieves the session data and invokes the logout object;
(3) The logout object clears the authenticated user id field in the session;
(4) The logout object returns a "logged out" notification page or redirects the browser to the home page.

The above procedural descriptions are complemented with an illustrative diagram in Fig. 1:

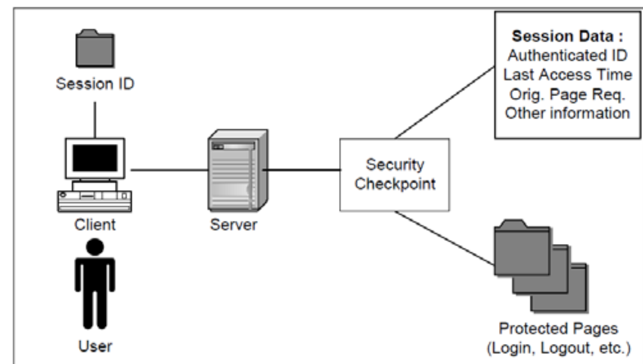

Fig.1 – The Authenticated Session Diagram

HLPN Model of the *Authenticated Session* Pattern

The HLPN model of the *authenticated session* pattern is shown in Fig. 2, where the type definitions of places and constraints of transitions are provided. For example, a web request is defined as consisting of a session id (*Sid*), and a web page number (*Page#*). The actual types of the above abstract entities are defined as integers. Since there can be multiple requests, a power set type is defined for place *Wreq*. Please note, an arc label connecting to a place of a power set type is represented as $\{x\}$ to indicate the access of a single token. Likewise, a user account (*Acct*) is abstracted as a tuple containing a user name (*Uid*) and a password (*Psw*), both are defined as strings.

The model captures the authentication mechanism with the assumption of the uniqueness of user name, which was implicitly assumed in the original pattern description. Page data are defined as strings and their actual uses are not modeled. In this model, system actions in the authentication process are modeled with transitions. Transition *Retrieve*

captures the steps (1) and (2) in Scenarios 1 and 2. Transition *Check1* models the steps (3) to (6) in Scenario 1. Transitions *AuthF* and *AuthS* define the login activities. Transition *Check2* specifies the steps (3) to (6) in Scenario 2. Transition *Timeout* models step (4) in Scenarios 1 and 2. Transition *Logout* models the logout activities. Transition *Inc* is used to model the increment of clock. *du* is a constant denoting the duration for timeout from the last access. For example, the constraint of transition *Check1* contains the precondition $o[2] = "null"$ or $o[3] + du > c$ (the first time request or expired session).

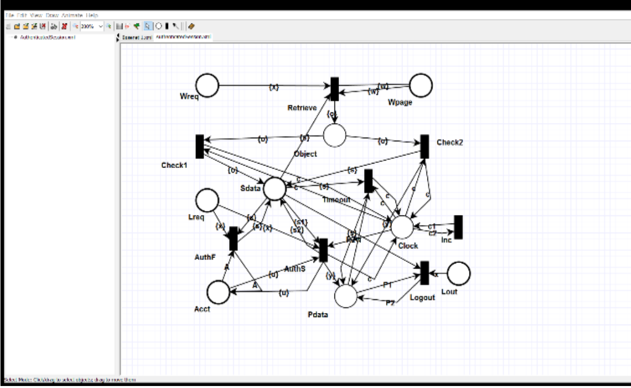

Fig.2 – HLPN model of *Authenticated Session*

$\varphi(Wreq) = \wp(Sid \times Page\#)$,
$\varphi(Wpage) = \wp(Page\# \times Data)$,
$\varphi(Object) = \wp(Sid \times Uid \times Last \times Data)$,
$\varphi(Sdata) = \wp(Sid \times Uid \times Last \times Data)$,
$\varphi(Lreq) = \wp(Uid \times Psw \times Sid)$,
$\varphi(Acct) = \wp(Uid \times Psw)$,
$\varphi(Clock) = Time$,
$\varphi(Pdata) = \wp(Sid \times Data)$,
$\varphi(Lout) = Sid$,
$R(Retrieve) = s[1] = x[1] \wedge w[1] = x[2]$
$\quad \wedge o = < s[1], s[2], s[3], w[2] >$,
$R(Check1) = o[2] = "null" \vee o[3] + du \leq c$
$R(Check2) = (o[2] \neq "null" \wedge o[3] + du > c)$
$\quad \wedge s = < o[1], o[2], c, o[4] > \wedge y[1] = o[1]$
$\quad \wedge y[2] = o[4]$,
$R(AuthF) = x[3] = s[1] \wedge \nexists u \in A.(u[1] = x[1] \wedge u[2] = x[2])$,
$\quad \wedge msg =$ "try again",
$R(AuthS) = x[1] = u[1] \wedge x[2] = u[2] \wedge x[3] = s1[1]$
$\quad \wedge s1[4] \neq "null" \wedge s2 = < s1[1], x[1], c, s1[4] >$
$\quad \wedge y[1] = s1[1] \wedge y[2] = s1[4]$,
$R(Timeout) = s[3] + du \leq c$
$\quad \wedge P2 = P1 \backslash \{p | \forall p \in P1.(p[1] = s[1])\}$,
$R(Logout) = x[1] = s[1]$
$\quad \wedge P2 = P1 \backslash \{p | \forall p \in P1.(p[1] = s[1])\}$,
$R(Inc) = c2 = c1 + 10$.

IV. Analyzing Security Pattern Models

Properties of security patterns are defined using first order linear time temporal logic (LTL) [9], which is an extension of classic first order logic with temporal operators. Predicates in LTL are places and transitions in HLPN. A LTL formula is evaluated under a HLPN model. It should be noted that only correctness properties of the security models are specified and analyzed in this paper. General security properties such as confidentiality, integrity, and availability are not directly studied in this paper, which cannot be specified using temporal logic [10].

A. *Specifying Correctness Properties*

We have identified and specified three to four correctness properties for all six patterns. In this section, we show how to formulate various correctness related properties in the *authenticated session* pattern. We have identified the following properties:

(a) The initial web access request to a protected resource must pass authentication. This property is a little tricky to specify. An initial web access implies a session object without an authenticated user identifier (empty field), thus the firing of transition *Check*1. Furthermore the actual access requires the firing of transition *AuthS*:
$Object(i, "null", *, *) \Rightarrow \Diamond\, Check1(i, "null", *, *, c)$,
$Pdata(i, d) \Rightarrow \diamond\, AuthS(i, u, *, d, c)$,
where $\diamond$ is a past temporal operator;

(b) The successive web access requests to a protected resource within the current session must pass session data validation:
$Object(i, u, c, *) \wedge c + du > clock \Rightarrow$
$\quad \Diamond Check2(i, u, *, *, c)$;

(c) The authenticated session expires after a preset inactive time period:
$Sdata(i, u, t, d) \wedge c \geq t + du \Rightarrow \Diamond \neg Sdata(i, u, t, d)$

(d) Session data are cleared to prevent an attacker to revisit cached pages:
$Sdata(i, u, t, d) \wedge Lout(i) \Rightarrow \Diamond \neg Sdata(i, u, t, d)$

In the above formulas i, u, c, d, t are free variables, * denotes any legal value. du is a constant. Most of the above properties are liveness properties.

B. *Analyzing Properties*

Liveness properties in general are very difficult to verify using model checking since they cannot be proved or refuted using finite execution sequences. Furthermore, all the models of the six patterns use power sets of tuples consisting of integers and strings, and some are clearly infinite state models, thus cannot be directly model checked without applying some abstraction techniques to simplify them. Since model checking is not applicable, we seek weaker assurance

through finding finite sequence witnesses through simulation. Thus we turn the verification of the above liveness properties into an easier reachability checking problem. One problem with reachability checking is that it depends on the given initial marking. Fortunately, our reachability checking does not depend on the specific values of the free variables appearing in the formulas as long as certain relationships between these variables are kept. This is similar to the idea of predicate abstraction in model checking, where a generic variable can be replaced with a Boolean variable as long as the correct relationship is maintained. As a result, we can use any randomly generated initial marking satisfying the variable relationships to ensure the validity of the reachability checking. It should be noted that the above properties have been validated but not verified. The simulator has shown that there is a transition firing sequence from an initial marking satisfying any of the above properties; but has not shown that every transition sequence satisfies the above properties. Even these weaker results can be extremely valuable in detecting incomplete and faulty requirements, and eliminating many design errors.

Table 1 summarizes the size of the six models and the number of properties specified and validated.

Table 1 – The Size Metrics of the Models

	Place	Transition	Arc	Property
Account Lockout	6	6	22	4
Auth. Session	9	8	38	4
Client Data Storage	7	6	19	4
Encrypted Storage	7	4	17	4
Password Auth.	6	3	12	3
Password Prop.	12	7	25	4

Table 2 provides the validation results of properties of the authenticated session, which are representative for other patterns. The main parameter is the number of user accounts in the initial markings. The time (ms) is the mean of five runs due to the random firings of enabled transitions such that each run may yield a different running time. Property (a) depends heavily on the user account number and thus takes more time. The experiments are run on Intel® Core(TM) i7-4770S CPU @ 3.10 GHz with 8 GB RAM.

Table 2 – Validation Results

	Token	Time	Token	Time	Token	Time
(a)	10	7	1000	255	10000	38749
(b)	10	1	1000	115	10000	124
(c)	10	3	1000	3	10000	3
(d)	10	0	1000	0	10000	0

V. Concluding Remarks

In this paper, we applied high level Petri nets to formally model and analyze six well-known security patterns. Building a formal model from given textual descriptions is often quite difficult due to the incompleteness and ambiguity of such descriptions, and thus error prone. Identifying and correctly specifying relevant properties is another major challenge. Formally verifying whether a formal model satisfying the specified properties is very hard due to the complexity of the models that often results in state explosion problem. We have shown the usefulness of reachability analysis based on simulation as a supplemental validation technique. The validated security patterns form the basis for composing multiple security patterns as well as integrating security patterns with other system components.

Acknowledgements

This work was partially supported by the NSF under grant HRD-0833093 and by the AFRL under agreement number FA8750-15-2-0106. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] N. Yoshioka, H. Washizaki, and K. Maruyama: “A survey of security patterns”, *Progress in Informatics*, no. 5, 35-47, 2008.

[2] D. M. Kienzle, M. C. Elder, D. Tyree, and J. Edwards-Hewitt: “Security Patterns Repository Version 1.0”, 2006.

[3] S. Konrad, B. Cheng, L. Campbell, and R. Wassermann: “Using Security Patterns to Model and Analyze Security Requirements”, *International Workshop on Requirements of High Assurance Systems*, 2003.

[4] W. E. McUmber and B. H. C. Cheng: “A general framework for formalizing UML with formal languages”, *Proceedings of IEEE International Conference on Software Engineering*, Toronto, Canada, May 2001.

[5] A. Dwivedi and S. Rath: “Formalization of Web Security Patterns”, *INFOCOMP*, v. 14, no. 1, p. 14-25, June 2015.

[6] J. Dong, T. Peng, and Y. Zhao: “Automated verification of security pattern compositions”, *Information and Software Technology*, vol. 52, 2010, 274–295.

[7] X. He: “A Comprehensive Survey of Petri Net Modeling in Software Engineering”, *International Journal of Software Engineering and Knowledge Engineering*, vol. 23, no. 5, 2013, 589-626.

[8] S. Liu, R. Zeng, X. He: “PIPE+ - A Modeling Tool for High Level Petri Nets”, *Proc. of International Conference on Software Engineering and Knowledge Engineering*, Miami, July 2011, 115 - 121.

[9] Z. Manna and A. Pnueli: “*The Temporal Logic of Reactive and Concurrent Systems – Specification*” Springer-Verlag, Berlin, 1992.

[10] M. Clarkson and F. Schneider: “Hyperproperties”, *Journal of Computer Security*, vol. 18 (2010) 1157–1210.