A Tool for Supporting Developers in Analyzing the Security of Web-based Security Protocols

Giancarlo Pellegrino[1,2], Luca Compagna[2], and Thomas Morreggia[2]

[1] Eurecom, Sophia-Antipolis
giancarlo.pellegrino@eurecom.fr
[2] SAP AG
giancarlo.pellegrino@sap.com,luca.compagna@sap.com,
thomas@morreggia.fr

Abstract. Security protocols are specified in natural language, are highly-configurable, and may not match the internal requirements of the development company. As a result, developers may misunderstand the specifications, may not grasp the security implications of configurations, and may deviate from the specifications introducing flaws. However, none of the existing techniques in discovering flaws provides the features, scalability, and usability to support developers in assessing the security of protocol configurations and deviations. This paper presents a tool that leverages on existing design verification and security testing techniques, and extends them to support developers in analyzing the security of security protocols. We used the tool for the security analysis of prominent security protocols (i.e., SAML SSO, OpenID, OAuth2), and of six industrial-size implementations.

1 Introduction

Security protocols are communication protocols that aim at providing security guarantees through the application of cryptographic primitives. Security protocols are at the core of modern business scenarios and enable partners to set up business environments. However, their specifications and implementations can be flawed as witnessed by the many vulnerabilities discovered in the past years [4,5,10,14]. Security protocols are specified in natural language and, as a consequence, can be misinterpreted by the developers. Moreover, the design of modern protocols keeps in mind the different deployment landscapes (e.g., mobile, on-premises, or cloud scenarios). As a result, protocols feature different protocols flows and a wide range of options. The number of options combinations makes it difficult for developers to understand the security implications. In addition, the protocol security recommendations delivered by the standardization bodies may not match the internal requirements of the software development company. As a result, implementations may deviate from the specifications and endanger the overall security. E.g., the SAML-based SSO for Google Apps until 2008 neglected few, yet important, message fields that allowed an attacker to impersonate a user and steal her confidential data [4]. To detect vulnerabilities, researchers have proposed several techniques at the different phases of the software life-cycle. Source code analysis looks for patterns into the source code or

analyzes the data flow of the user-controlled inputs [11]. Black-box input fuzzers probe implementations with special values and analyze the output for detecting vulnerabilities [9]. Model checking checks whether the protocol design satisfies a given security property [5,14] and if not, the returned counterexample is used as a test case for the testing implementations [7,8,10]. However, none of these techniques provides the features, scalability, and usability to support developers in assessing the security of protocol options and deviations.

This paper presents a tool that leverages on existing design verification and security testing techniques, and extends them to support developers in analyzing the security of security protocols. The tool helps developers, software engineers, and security experts in taking decisions during the development process and to detect flaws both at the design and deployment phases. It integrates existing verification and testing techniques that are described in other works [2,7,6] and it does not introduce new testing or verification techniques. The tool is not a product of SAP and it is not our intention to promote any other SAP product. The focus of this paper is to present a tool that is the result of three years of experience in applying cutting-edge security analysis techniques to industrial-size scenarios. We used the tool for the security analysis of prominent standard security protocols (i.e., SAML SSO, OpenID, and OAuth2) and of six industrial-size implementations.

Case Study :

The SAML [13] SSO is a security protocol that enables business partner to authenticate users once and then let them access their services. The objective of a client C is to access to a resource at a service provider SP. An identity provider IdP authenticates C and issues authentication assertions (a signed authentication token). The protocol ends when SP consumes the assertions and grants or denies C the access to the resource. SAML SSO protocol offers two basic flows depending on whether the user requests the resource to SP (SP-initiated SSO), or to IdP (IdP-initiated SSO). Both flows can be used in combination with the *Artifact Resolution Protocol* (ARP) that allows SAML messages to be be transported by reference rather than by value. In total, SAML SSO offers six protocol flows (ARP can be used at most twice in each basic flow).

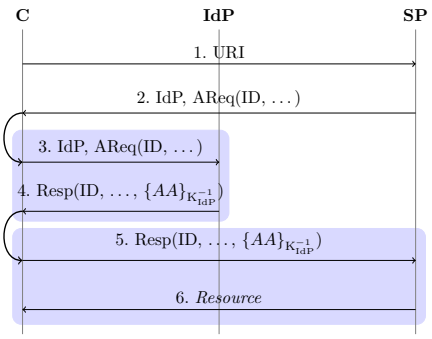

Fig. 1. SAML SSO SP-initiated without ARP.

Figure 1 shows the SAML SSO SP-initiated without ARP. In step 1, C asks SP the resource at URI. In step 2, SP redirects C to IdP with the authentication request AReq(ID, ...) where ID is uniquely identifying the request. Then SP stores ID in a table. In step 3, IdP authenticates C, builds the authentication assertion AA, and signs it with its private key. Then, IdP stores $\{AA\}_{K_{IdP}^{-1}}$ in

Resp and forwards it to SP via C. SP first verifies the signature and then checks if its table contains ID. Finally, SP delivers the resource to C. At the end of the protocol run, C and SP are mutually authenticated (goal G1) and the resource is kept confidential for C (goal G2). The messages 3-4 and 5-6 in Figure 1 are exchanged over SSL/TLS communication channels.

In this paper we consider only few protocol options[3]: SP signs AReq, IdP signs Resp, and use of SSL/TLS in steps 1-2. In addition, developers would like to assess the security of design decisions. In this paper we consider the following decisions: SP does not store the ID in steps 1-2, and SP sets an HTTP cookie at step 2 and check it at step 5.

2 A Security Testing Tool

Our tool is a set of Eclipse plugins implementing existing testing and verification techniques [1,2,6,7]. The tool supports the specification of protocol options and implementation decisions, implements the *design verification* and *model-based security testing* workflows, and supports verification and test campaigns.

Configuration and Implementation decisions : Our tool enables the specification of configuration options and implementation decisions. This is done through the *SPaCIoS navigator*. The navigator implements three main functionalities. First, it allows the specification of single protocol option (or decision) by means of labels. (A label is a text description and a arbitrary color.) Second, it allows for the creation of a new model (capturing the option) starting from an existing one and for marking it with labels. Finally, the navigator keeps track of all the model generated in a derivation tree in which the roots are the reference models. The tree and the labels are used later on for the preparation of the test/verification campaign.

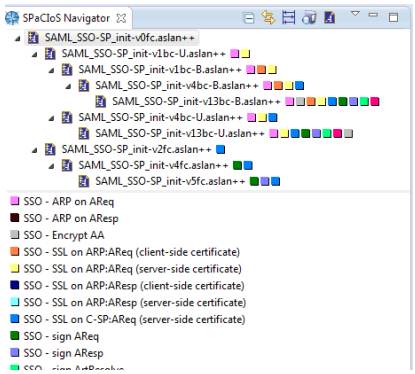

Fig. 2. The Navigator

Figure 2 shows the navigator. The upper part displays the derivation tree in which each model (i.e., node tree) is associated to labels. A model can have more than one label. The lower part of the navigator shows the list of labels created during the analysis. They capture the configuration options of the SAML SSO standard that we used for supporting the developers at SAP.

Design verification : The design verification workflow implements the formal analysis of security protocols via model checking. The process consists of three steps. First, the user models the formal model and specify the security

[3] The other options of the same flow as well as the options of the other five protocol flows were considered in our analysis, but not shown in this paper.

property. Then, the model checker explores the model for violation of the property. If a violation is discovered, the model checker returns a counterexample. Finally, the user inspects the counterexample using graphical viewers.

Our tool has text editors for coding formal models (currently, it supports only ASLan and ASLan++ languages[2]) with features such as syntax highlighting, and problems highlighting (for syntax and semantic errors). It integrates the SAT-based Model Checker [3] for the formal verification, and UI components for displaying counterexamples as message sequence charts.

Model-based security testing : The model-based security testing workflow is used for testing implementations for detecting security flaws. It consists of five steps. First, the user codes model and properties by using the text editor as seen in the design verification workflow. Second, the tool generates test cases using an external model checker (a test case is a counterexample). Third, the user defines the implementation under test (IUT). The IUT is a data structure containing the mapping between model symbols and real values, the protocol participants under test, and a list of message adapters. IUTs are created by using the IUT editor. Fourth, the test case are executed against the IUT and, finally, the user inspects the results. Our tool supports HTTP conversation and message inspection. Moreover, the tool has a built-in web browser to visualize the content of HTTP responses.

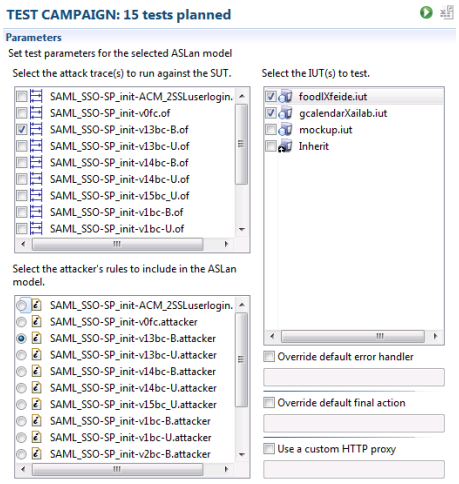

Fig. 3. The Test Campaign Manager

This workflow implements the technique devised by Armando et al. [7] in which the formal model is compiled into a set of Java program fragments that are executed in the order of the abstract test case.

Verification and Test Campaign: A verification campaign is a multiple execution of the verification step. This solves the practical problem of verifying several models. Similarly, the test campaign consists of the executions of several test cases.

Figure 3 shows the editor for the test campaign manager. On the left-hand side, the editor displays the available models. Models are shown in a tree-like form. On the right-hand side, the editor shows the list of test cases generated and the IUTs available. The user selects the test cases and the IUTs, and she runs the campaign. At the end of the execution, the tool displays the HTTP conversations for off-line analysis. The result of a campaign is organized into tables. In addition, the tool logs the results and HTTP messages of all the test for future inspections.

3 Application to the Case Study

We modeled the flow of Figure 1 in ASLan++. For each option and decision, we created a label with the UI of Figure 2 and derived a model. We adjusted each new model for reflecting the option (resp. decision). Afterwards, we created and launched a verification campaign. Figure 4 shows an excerpt of the result of the campaign. The table is structured as follows. Each row is a model with unique identifier *MID*. The column *from* is a pointer to the model from which *MID* has been derived. The remaining columns are grouped by *Opt*, *Dec*, and *Res* respectively for options, decisions and results. We use *y* when the option (resp. decision) is used or when the model checker found a violation; we use *n* otherwise. For example, the model *2fc* derives from 0fc (depicted in Figure 1) by adding the SSL/TLS channel in steps 1-2.

		Opt.			Dec.		Res.	
		SSL	Sign					
MID	from	C-SP:AReq	AReq	AResp	SP set cookie	SP stores ID	G1	G2
0fc	-	n	n	n	n	y	y	n
2fc	0fc	y	n	n	n	y	y	n
4fc	2fc	y	y	n	n	y	y	n
5fc	4fc	y	y	y	n	y	y	n
...								
6fc	0fc	y	n	n	y	y	n	n
...								
7fc	5fc	y	y	y	y	n	n	n
...								

Fig. 4. Results for the SP-initiated protocol

Figure 4 shows the following results. First, the protocol suffers from a flaw in which G1 is not satisfied. Second, the protocol options are not sufficient for fixing the flaw. Third, the use of cookies solves the vulnerability. Fourth, the two implementation decisions do not endanger the security with respect to the properties G1 and G2. Finally, the security goal G2 is always reached.

Developers can use the results of Figure 4 for taking decisions about the design and the implementations. For example, in security-sensitive scenarios, they may enforce the use of cookie and avoid storing the ID as a Denial-of-Service countermeasure.

The counterexamples returned by the model checker are used as test cases for probing the implementations. For example, we used the counterexample of 0fc to test SAML-based SSO for Google Apps and SimpleSAMLphp as reported by Armando et al. [5]. The former implements the configuration of 0fc while the latter 6fc. The test against SAML-based SSO for Google Apps succeeded proving that also the implementation suffers from the flaw [5]. The test against SimpleSAMLphp failed due to the use of the cookie [5]. We applied the same approach on the OpenID protocol and its implementations. The tests detected a flaw in both the specifications and implementations (Zoho Invoice with Google OpenID or Yahoo OpenID). In addition, we used the tool at SAP to assist developers during the development of the NGSSO and OAuth2. In the former, we analyzed all the six SAML SSO flows considering in total 15 protocol options, and seven implementation decisions. In total we verified 85 formal models. In the latter, we considered so far one protocol flow and seven protocol options.

4 Future work and Conclusion

We plan to support other modeling languages more suitable for developers, e.g., the Alice-and-Bob notation [12] or UML sequence diagrams. In addition, we plan to integrate inference techniques for creating models from traces [10] and to integrate other test case generation techniques [8]. In conclusion, we presented a model-driven security analysis and testing tool. It supports the evaluation of the impact of implementation decisions and protocol configurations. The tool was used for the security analysis of SAML SSO, OpenID, and OAuth2, and of six industrial-size protocol implementations.

References

1. SPaCIoS Secure Provision and Consumption in the Internet of Services, http://spacios.eu/.
2. A. Armando, W. Arsac, T. Avanesov, M. Barletta, A. Calvi, A. Cappai, R. Carbone, Y. Chevalier, L. Compagna, J. Cuéllar, G. Erzse, S. Frau, M. Minea, S. Mödersheim, D. von Oheimb, G. Pellegrino, S. E. Ponta, M. Rocchetto, M. Rusinowitch, M. T. Dashti, M. Turuani, and L. Viganò. The AVANTSSAR Platform for the Automated Validation of Trust and Security of SOA. In *TACAS2012*.
3. A. Armando, R. Carbone, and L. Compagna. Ltl model checking for security protocols. In *CSF '07. 20th IEEE*, July.
4. A. Armando, R. Carbone, L. Compagna, J. Cuellar, and L. T. Abad. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *Proc. of ACM FMSE08*.
5. A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti. An authentication flaw in browser-based single sign-on protocols: Impact and remediations. *Computers and Security*, 33, 2013.
6. A. Armando, R. Carbone, L. Compagna, and G. Pellegrino. *Automatic security analysis of SAML-based single sign-on protocols.* Chapter 10 in "Digital Identity and Access Management: Technologies and Framework", 2011.
7. A. Armando, G. Pellegrino, R. Carbone, A. Merlo, and D. Balzarotti. From model-checking to automated testing of security protocols: Bridging the gap. In *TAP2012*.
8. M. Büchler, J. Oudinet, and A. Pretschner. Semi-automatic security testing of web applications from a secure model. In *SERE2012*.
9. A. Doupé, M. Cova, and G. Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In C. Kreibich and M. Jahnke, editors, *DIMVA*, volume 6201 of *LNCS*. Springer, 2010.
10. B. Guangdong, M. Guozhu, L. Jike, S. V. Sai, S. Prateek, S. Jun, L. Yang, and D. Jinsong. Authscan: Automatic extraction of web authentication protocols from implementations.
11. N. Jovanovic, C. Krügel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *IEEE Symposium on Security and Privacy*, pages 258–263. IEEE Computer Society, 2006.
12. S. Mödersheim and L. Viganò. The open-source fixed-point model checker for symbolic analysis of security protocols. In *FOSAD 2009*.
13. OASIS Consortium. Security Assertion Markup Language V2.0 Tech. Overview. http://wiki.oasis-open.org/security/Saml2TechOverview, Mar. 2008.
14. V. Shmatikov and J. C. Mitchell. Finite-state analysis of two contract signing protocols. *Theoretical Computer Science*, 283(2):419–450, June 2002.