2011 International Conference on Communication Systems and Network Technologies

Formal Verification of OAuth 2.0 using Alloy Framework

Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai and Sanjay Singh
Department of Information & Communication Technology
Manipal Institute of Technology, Manipal University, Manipal-576104, INDIA
sanjay.singh@manipal.edu

***Abstract*—Over the past few years, the paradigm of social networking has grown to such a degree that social networking websites have evolved into full-fledged platforms, catering to a wide range of consumer interests. The near-ubiquity of Internet access has facilitated the proliferation of users that indulge in social networking. However, this wide spread usage of the Internet and social networking in particular brings with it the need to design and implement a plethora of security enhancing and privacy preserving protocols and standards. Several protocols and security mechanisms have been proposed to ensure primary security features such as confidentiality, integrity, authenticity and non repudiation. However, ensuring the correctness of these protocols is crucial in ensuring user confidence in system security. Therefore, these protocols need to be verified in some formal sense that involves an exhaustive examination of the protocol flow and its state transitions. In this paper, we formalize OAuth, an authentication standard which has found wide acceptance in the Internet community. We formalize the protocol using a method called knowledge flow analysis, using the Alloy modeling language for specification and the Alloy Analyzer for verification. We show how the Alloy Analyzer successfully discovers the known security vulnerability in OAuth.**

***Index Terms*—OAuth, Alloy Analyzer, Social Networking**

I. Introduction

Today, websites are no longer just static HTML pages served to a browser; they are full-fledged platforms storing large amounts of data and expose data and services through an API, which can be leveraged by third-party developers who can develop applications on top of this platform, thus forming a healthy ecosystem under the control of the platform provider. Ensuring security is one of the major hurdles than need to be overcome in order to maintain a thriving ecosystem that is safe for both its users and third party application developers.

Several protocols work in tandem to take care of security issues in such an ecosystem. One such protocol that aims for a fool-proof security method for authenticating third-party applications and authorizing them to access user data in a controlled manner is OAuth (Open Authorization) [1]. Introduced in 2009, OAuth has gained widespread acceptance in a short period of time and can be called as de facto standard for authorization. OAuth allows users to provide third party applications the access to the user's protected resources stored on some server without divulging the user's password or other secret credentials in a transparent manner. This protocol has gone through two major versions, with the latest version OAuth 2.0 not being backward compatible with its predecessor OAuth 1.0.

OAuth 2.0 has been designed in such a way that client developer complexity has been drastically reduced as compared with its predecessor. This design decision has drawn criticism from the security community as sacrificing security for usability. In this paper, we concentrate exclusively on OAuth 2.0.

Formal methods have found great applicability in the verification of security protocols. Several popularly used protocols have been successfully formalized and new security vulnerabilities have been uncovered. A large number of techniques have been devised for specifically verifying security protocols. Moreover, tools like Alloy [2] and AVISPA (Automated Validation of Internet Security Protocols and Applications) [3] try to bridge the gap between the informal protocol flow specification and its formal specification in a modeling language by ensuring minimum loss of semantics during the mapping process. A protocol like OAuth needs to be comprehensively evaluated in order to protect the millions of users that are exposed to it , and formalizing the protocol and verifying it is an important step in that direction.

The contributions of this paper are:

(i) To suggest an extension to the knowledge flow analysis technique in the context of verifying security protocols, and more specifically, authentication protocols.
(ii) To formalize the OAuth protocol using the Alloy modeling language and show the proof-of-concept of the model.

The rest of the paper is organized as follows. Section II explores the related work done in formal verification of security protocols. Section III introduces the working of the OAuth protocol and its various protocol flows, followed by a discussion on its security vulnerabilities. Section IV introduces the knowledge flow analysis technique and suggests several changes to make it more amenable to security protocol verification. Section V explains how OAuth 2.0 can be formalized using predicate logic. Section VI discusses the Alloy modeling language and the Alloy Analyzer. Section VII discusses how the knowledge flow analysis representation can be expressed in Alloy. Simulation results and discussion is given in section VIII. Finally section IX concludes the paper.

II. Related Work

Several protocols which had already been deployed have been found to be unsound using formal methods. Classification of methods employed to verify security protocols can be placed in two broad categories: Model Checking and Logical inference. While the logic programming approach and the sequential programming approach come under the domain of model checking, the Strand spaces approach and the belief based approach come under the domain of logical inference. Examples of the sequential programming approach include using CSP (Communicating Sequential Process), which is a process algebra for reasoning about processes. Several model checking tools like FDR (Failure Divergence Refinement) [4] and Casper [5] are based on CSP. Lately, tools like NuSMV

978-0-7695-4437-3/11 $26.00 © 2011 IEEE
DOI 10.1109/CSNT.2011.141

(Symbolic Model Verifier) [6] and SPIN [7] have proved to be very effective in specifying and verifying models of security protocols. Examples of the logic programming approach include the ALSP (Action Language for Security Protocols) language, in which the specification is performed in terms of Horn clauses. Each clause is comprised of a head and a body, with the condition that the literals in the head are true only if all the literals in the body are true. Examples of the belief based approach include special logics such as the BAN (Burrows-Abadhi-Needham) logic [8] and the GNY (Gong-Needham-Yahalom) logic [9]. However, the shortcoming of these belief based logics is that it does not allow the modeling of the capabilities of the intruder explicitly.

Several protocols have been successfully formally verified and their security flaws have been exposed. Longley and Rigby [10] developed a tool using which they discovered a flaw in a hierarchical key management scheme. Meadows [11] used the NRL protocol analyzer tool to find hitherto unknown flaws in the Simmon's Selective Broadcast Protocol and the Burn's and Mitchell's Resource Sharing Protocol. Stubblebine and Gligor [12] were able to develop a model using which they uncovered flaws in Kerberos and privacy enhanced mail.

III. OAuth 2.0 and Its Protocol Flows

A. Introduction

OAuth is an authorization standard that enables users to grant third party applications with limited access to their resources stored at a server, without divulging their password or other secret credentials. To describe the protocol in its entirety, we need to understand the various roles that are involved in a single protocol run. These roles are:

1) **Resource Owner**: An entity that has the power to grant permission to others for accessing resources that it owns. In other words, it is the end user of the application, or more specifically, the user-agent(browser)
2) **Resource Server**: The entity that hosts the protected resources owned by the resource owner, and has the capability to respond to resource access requests using access tokens.
3) **Client**: An application that can make resource requests to the resource server on behalf of the resource owner after obtaining authorization.
4) **Authorization Server**: The server which issues access tokens to the clients after successfully authenticating the resource owner and obtaining its authorization.

In most cases, the role of the authorization server and the resource server are played by a single entity.
At a higher level of abstraction, the basic protocol flow in OAuth can be summed up by Fig.1:

1) The client requests authorization from the resource owner regarding the usage of the resource owner's protected resource. Usually, this is done through the authorization server, which serves as an intermediary.
2) The client receives an authorization grant which represents the authorization provided by the resource owner. Four different types of authorization exist, any of which needs to be supported by both the client and the authorization server.
3) The client requests the authorization server to provide it with an access token which can be used to access the protected resources. During this process, the client provides its client credentials and the authorization grant to authenticate with the authorization server.
4) The authorization server confirms the validity of the client credentials and the authorization grant and provides the client with an access token.
5) The client requests the protected resources hosted at the resource server by producing the access token.
6) The resource owner checks for the validity of the access token and if valid, it services the request.

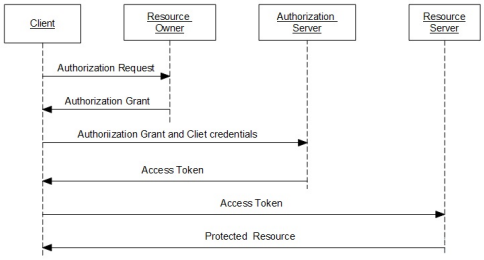

Fig. 1. Abstract Protocol Flow

B. Authorization

Four different types of authorization types are defined, thus resulting in four different possible message interaction sequences. While modeling the protocol, we need to consider all the four types of message interaction sequences because different protocol participants and at different protocol runs may use different authorization types
The four different types of authorization are:

1) **Authorization code:** The authorization code is obtained by using an authorization server as an intermediary between the client and resource owner. The resource owner is redirected to the authorization server where it presents its credentials. The main benefit of using this technique is the ability to authenticate the client and issue the access token directly to the client without potentially exposing it to others, including the resource owner.
2) **Implicit grant:** When the resource owner's authorization is directly expressed as an access token, without the need for an authorization code, an implicit grant is said to take place. Implicit grants improve the responsiveness and efficiency of clients like those which have been implemented as a in-browser application.
3) **Resource Owner Password Credentials:** The resource owner's password can be directly used as an authorization grant to obtain an access token. This method is used when a high degree of trust exists between the resource owner and the client, such as in cases where the client is the operating system itself or when authorization methods are not available
4) **Client Credentials:** Only the client credentials are produced in exchange for an access token. This is used when the scope of the authorization available to the client is limited to the protected resources which are under the control of the client, or if access to protected resources is already arranged with the authorization server. In this case, the resource owner and the client role are played by the same entity.

C. Security vulnerability in OAuth

As highlighted by Ryan Paul [13], a security flaw exists in OAuth. While this security flaw does not result in the compromise of the resource owners password credentials, it still has far reaching consequences. The flaw is as follows. The client credentials which include the client password should be kept in a safe location to prevent misuse. In cases where the client is a web application, the credentials are stored at the server side at a secure web server which can be safeguarded against attacks by employing the required security measures. However, if the client is a desktop application, then the client credentials are stored in the desktop software. An experienced hacker can reverse engineer the code and retrieve the client credentials. He can then masquerade a malicious application as the client by using the compromised secret client credentials, thus fooling both the resource owner and the resource server. To prevent widespread damage, the authorization server may revoke the client credentials, which means that the legitimate client too would suffer. This problem does not seem to have an intuitive solution as of now.

IV. Knowledge flow analysis

A major aspect of formalization is the abstraction of the protocol structure to the intended level of granularity. In this paper, we follow the Knowledge Flow Analysis technique proposed by Torlak et al [14]. Knowledge flow analysis provides a uniform framework that allows various protocol components at different levels of granularity to be modeled, from the interaction of protocol participants to the implementation of cryptographic primitives.

A system is modeled as consisting of an initial set of knowledge and a collection of rules that regulate how this knowledge can be transferred between the different protocol participants, whom we refer to as principals. The total body of knowledge is composed of discrete units termed as values.

These concepts can be modeled as a state machine, where a state is given by a relation mapping principals to the values that they know. Thus, the set of transitions on these states represent the allowable knowledge flows between principals which can be specified as a constraint.

The advantages of knowledge flow analysis are its ease of expression and its flexibility. It uses an elementary mathematical framework and does not require a very deep understanding of mathematical logic. Its flexibility ensures that it can model at multiple levels of granularity; from gate level operations to cryptographic primitives to component level message flow, all as part of a single model

As an extension to the Knowledge flow analysis method, we suggest some guidelines in this paper, which can be applied to ensure that the protocol specification is more amenable to analysis. In this light, we introduce a few axioms to guide the protocol designer in the specification process.

Guideline 1: Knowledge can neither be created nor be destroyed but can be transferred from one principal to the other without the loss of knowledge during the transfer.

The implications of the above mentioned axiom are many: It suggests that the domain of knowledge in a protocol run is fixed; any unit of knowledge is known by at least one principal. This means that even events like compressing or encrypting a unit of knowledge can be taken as a knowledge transfer between two entities, which means that the encryptor or the compressor unit too needs to be modeled as a principal. This allows us to reason about situations where we are not sure if, say the encryption software itself is liable to get compromised.

To ensure a healthy states size, we have to divide the set of principals into two types: Trusted principals and untrusted principals. Trusted principals need not be explicitly modeled, thus reducing the state space down many notches.

Guideline 2: Knowledge once acquired cannot be forgotten but can become obsolete.

In the context of security protocols, many units of knowledge are accompanied by timestamps that denote the duration of time for which that unit of knowledge is valid. While modeling, to decrease the complexity of the model, we assign a timing attribute for only those units of knowledge that have a finite validity period.

Guideline 3: Smaller units of knowledge, provided that they have the same value for the duration attribute, can be aggregated into a single large unit, provided that for all principals, either a principal knows all the individual units of knowledge of the composite unit or it knows none of the individual units of knowledge of the composite unit.

V. Formalization

Using the knowledge flow analysis approach introduced in the following section, we formalize the OAuth protocol by representing it in terms of predicate logic.

The principals in OAuth are the Resource Owner, the Resource Server, the Client and the Authorization server. We represent the principals by the predicates:
R(x) : x is a Resource Server
O(x) : x is a Resource Owner
C(x) : x is a client
A(x) : x is an Authorization Server

There are a large number of values that are being exchanged between the protocol participants. Each of these values is again expressed in terms of a predicate. For example:
ROC(y): y is a Resource Owner Credential
and so on.
After the principals and values have been identified, we proceed to identify the relations between them. In the case of OAuth, we use four different types of relations. They are:
Owns - The Principal has complete authority and jurisdiction over the value.(It might have generated it itself), In predicate logic, this is expressed by the predicate owns(x,y) which means that a principal x owns value y.

$\forall x(O(x) \rightarrow (\exists\ ROC(y) \wedge owns(x,y))$

The above formula expresses the fact that a resource owner *owns* his resource owner credentials(password), which is expressed as a predicate ROC(y).

$\forall x(C(x) \rightarrow (\exists\ RURI(y) \wedge owns(x,y))$

This formula states that the client *owns* owns the contents of the redirectionURI which it uses to redirect the client to the authorization server.

$\forall x(C(x) \rightarrow (\exists\ PWD(y) \wedge owns(x,y))$

This formula states that the client owns its client password
Draws - The Principal knows this value at the start of the protocol, In predicate logic, this is expressed by the predicate draws(x,y) which means that a principal x knows value y at the start of the protocol.

$\forall x(A(x) \rightarrow (\forall y\ ROC(y) \wedge draws(x,y))$

This states that all authorization servers involved have a copy of the resource owner's password credentials

$\forall x(A(x) \rightarrow (\forall y\ ER(y) \wedge draws(x,y))$

This formula states that the authorization server knows all the possible error response messages at the start of the protocol

Computes - The Principal computes this value using other values that are already in its possession.

In predicate logic, this is expressed by the predicate computes(x,y) which means that a principal x has computed value y.

$\forall x(C(x) \rightarrow (\exists y ATR(y) \wedge computes(x,y))$

This states that the client can compute the access token request message given the other information that it already has.

Learns - The Principal knows this value by interacting with some other Principal, In predicate logic, this is expressed by the predicate learns(x,y) which means that principal x has learnt some value/values from principal y.

$\exists x \exists y(A(x) \wedge C(y) \wedge MSG(y,x) \rightarrow learn(x,y))$

This states that the authorization server learns from the client whenever a message is sent from the client to the authorization server.

$\exists x \exists y(R(x) \wedge C(y) \wedge MSG(y,x) \rightarrow learn(x,y))$

Similarly, the resource owner learns from the client whenever a message is sent from the client to to the resource server and so on.

$\exists x \exists y(O(x) \wedge C(y) \wedge MSG(y,x) \rightarrow learn(x,y))$

Similarly, these four predicates can be used to specify the relations between principals and values by constructing appropriate formulas. These formulas then form the basis for building a model in Alloy.

VI. Introduction to Alloy Specification Language and the Alloy Analyzer

A. Alloy Specification Language

Alloy is a declarative specification language for expressing the constraints and allowed behavior of a protocol or a software system. Alloy allows users to construct models using the popular and time-tested object oriented approach, thus increasing the ease of expressiveness. However, at the implementation level, Alloy employs a set-theoretic approach, with its backend inspired by the Z notation [15]. A major feature of Alloy that sets it apart from other specification languages used in the model checking process is that it allows the user to specify models having infinite scope. The fact that Alloy is a declarative language, wherein the order of the statements does not affect the meaning of the model is testimony to the ease of expression that Alloy presents [16].

B. Alloy Analyzer

Alloy Analyzer is a tool that processes models written in the Alloy specification language. The analyzer can check for the validity of user defined properties in the model and simulate the execution of operations defined as part of the model. The Alloy Analyzer has been built as a model finder built upon a Boolean SAT solver. It incorporates an array of off-the-shelf SAT solvers which include Berkmin [17], MiniSAT [18], ZChaff [19], KodKod [20], SAT4J [21]. The models which are specified in relational logic form are converted to the Boolean logic form and are fed to the SAT solver. The results are converted back into relational logic form and fed back to the user.

VII. Constructing the OAuth model in Alloy

While modeling the OAuth protocol in Alloy, we have modeled both the Principals (participants in the protocol) and Values (individual units of knowledge) as first class citizens. i.e we have represented both these sets of entities using the sig(equivalent to class in an object oriented language) construct. In our model, we have declared two $abstract\ sig's$ Principal and Value. The Principals Resource Owner, Resource Server, Client, Authorization Server all extend the abstract sig Principal. In this model, we don't explicitly model the intruder as a separate individual. Since a protocol run can proceed only with the participation of one of the four permitted roles, we assume that the intruder is one or more of the four principals themselves.

In a similar vein, all different units of knowledge are represented as a *sig* extending the *abstract sig* Value.

sig Authorization *extends* Principal{
learns1 : RedirectionURI1,
learns2 : ClientPassword,
learns3 : AuthorizationGrant,
learns4 : PermissionResponse,
draws1 : ResourceOwnerCredentials,
draws2 : PermissionRequest,
draws3 : AuthorizationCode,
draws4 : ErrorResponse,
}

The above code snippet shows the declaration of the principal Authorization server.

VIII. Simulation Results and Discussion

Assertions about properties that should hold true are expressed in Alloy using the assert statement. For the assertion that no resource owner can access the authorization code, we provide the *assert* statement.

assert SecurityConstraint{textitno r: ResourceOwner r.computes = AuthorizationCode}

We observe that the Alloy Analyzer returns a counter example to the assertion as shown in Fig.2. The counter example corresponds to the same security vulnerability that we had discussed in section III. Since the known security vulnerability has been detected and presented as a counter example, we have greater confidence in our model and can use it to search for other unknown flaws in the model. Figure 2 shows how the resource owner's user agent (browser) can access the client's secret key while the protocol goals are to the contrary.

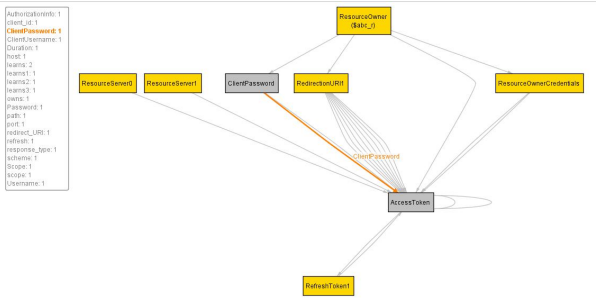

Fig. 2. Counter Example Generated by Alloy

IX. Conclusion

This paper has provided some guidelines that can be followed while using the knowledge flow analysis for formal specification. It has used these guidelines to formally model and verify the OAuth authorization protocol using the Alloy specification language. A known security vulnerability of the protocol was detected by the Alloy Analyzer, thus allowing us to claim proof-of-concept of the model. Future work would involve extending the model to model OAuth at a much lower layer of granularity.

References

[1] E. E. Hammer-Lahav. (2011) The oauth 2.0 authorization protocol draft-ietf-oauth-v2-13. [Online]. Available: http://tools.ietf.org/html/draft-ietf-oauth-v2-13

[2] (2010) Alloy 4.1. [Online]. Available: http://alloy.mit.edu/community/

[3] (2010) Avispa. [Online]. Available: http://www.avispa-project.org/

[4] (2010) Fdr model checker. [Online]. Available: http://www.fsel.com/

[5] G. Lowe, "Casper: a compiler for the analysis of security protocols," in *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, June 1997, pp. 18–30.

[6] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Proceedings of the 14th International Conference on Computer Aided Verification*, Copenhagen, Denmark, July 2002, pp. 241–268.

[7] G. Holzmann, "The model checker spin," in *Software Engineering*, Oct 1997.

[8] A. Bleekerl. (2011) A semantics for ban logic. [Online]. Available: http://dimacs.rutgers.edu/workshop/security/program2/bleeker/index.html

[9] A. Bleeker, "On the automation of gny logic," in *DIMACS Workshop on Design*, Sep 1997.

[10] S. D.Longley, "An automatic search for security flaws in key management schemes," in *Computers and Security*, July 1992, pp. 75–90.

[11] C. Meadows, "A system for the specification and analysis of key management protocols," in *IEEE Computer Society Symposium on Research in Security and Privacy*, Los Alamitas, California, 1991, pp. 182–195.

[12] S.Stubblebine and V.Glignor, "On message integrity in cryptographic protocols," in *Symposium on Security and Privacy*, May 1992, pp. 85–104.

[13] R. Paul. (2010) Compromising twitter's oauth security system. [Online]. Available: http://arstechnica.com/security/guides/2010/09/twitter-a-case-study-on-how-to-do-oauth-wrong.ars

[14] B. G. D. J. Emina Torlak, Marten van Dijk, "Knowledge flow analysis for security protocols," Massachusetts Institute of Technology, Cambridge, MA, Tech. Rep., Oct 2005.

[15] J. Jacky, *The way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1996.

[16] D. Jackson, *Software Abstraction: Logic, Language and Analysis*. The MIT Press, 2006.

[17] (2010) Berkmin sat solver. [Online]. Available: http://eigold.tripod.com/BerkMin.html

[18] (2010) Minisat solver. [Online]. Available: http://minisat.se/

[19] (2011) Zchaff sat solver. [Online]. Available: http://www.princeton.edu/ chaff/zchaff.html

[20] (2011) Kodkod sat solver. [Online]. Available: http://alloy.mit.edu/kodkod

[21] (2011) Sat4j sat solver. [Online]. Available: http://www.sat4j.org/