http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 12 (12): 2273-2285, 2013
ISSN 1812-5638 / DOI: 10.3923/itj.2013.2273.2285

Automatic Verification of Security Properties of OAuth 2.0 Protocol with Cryptoverif in Computational Model

[1]Xingdong Xu, [2]Leyuan Niu and [2]Bo Meng
[1]Center of Computation and Experiment, South-Center University for Nationalities, MinYuan Road No. 708, HongShan Section, Wuhan, Hubei, 430074, China
[2]School of Computer, South-Center University for Nationalities, MinYuan Road No. 708, HongShan Section, Wuhan, Hubei, 430074, China

Abstract: During the past several years, OAuth 2.0 protocol has been widely implemented and deployed. In order to give a strong confidence in its security to the people, in this study, Blanchet calculus in computational model is used to analyze OAuth 2.0 protocol with mechanized tool CryptoVerif. The term, process and correspondence are used to model OAuth 2.0 protocol . The authentication is formalized by non-injective or injective correspondence. The results show that OAuth 2.0 protocol has not authentication between the resource owner and authorization server, authentication between client and authorization server. The first automatic analysis on OAuth 2.0 protocol in computational model of in active adversary is implemented in this study.

Key words: Verification, authentication, correspondence, protocol security

INTRODUCTION

With the development of Internet technology and cloud computation, Internet has a strong influence on the life of people. Thus many transactions need the cooperation among the different system and websites. So there is an urgent requirement for accessing the data and services on the different and close websites. OAuth 2.0 protocol (http://tools.ietf.org/html/draft-ietf-oauth-v2-11) is proposed to address the problem. OAuth 2.0 protocol bas been implemented and deployed by Facebook, Twitter, LinkedIn, Google, Yahoo, .Sina, QQ, Renren and others. OAuth 2.0 protocol which is authentication and authorization protocol allows the third party to access to the protected resource which are require authentication using the related secure technologies, for example, authentication based on username and password. Although, OAuth 2.0 protocol is viewed as an authorization protocol, OAuth 2.0 protocol is being used for social login and hence for authentication between the resource owner and authorization server, authentication between client and authorization server.

In order to verify the security properties of security protocols including OAuth 2.0 protocol and enhance the confidence of the people, two models have been developed for analyzing security protocols form the beginning of the 1980s (Meng, 2011; Meng and Shao, 2010), symbolic model and computational model. In symbolic model cryptographic primitives are abstracted as black boxes and lots of mechanized tools have been implemented, for example, SMV, NRL, Isabelle, Athena, Revere, SPIN, Brutus, ProVerif and AVISPA. However, the results of analysis in this model are not quite clear.

The other model is computational model that bases issues of complexity and probability. This model uses a strong notion of security, against all probabilistic polynomial-time attacks. Chari *et al.* (2011) use universally composable security to formalize by hand the authorization code mode of the OAuth 2.0 protocol. The result of analysis in computation model is quite realistic; however it is a hard problem to develop an mechanized tool. The introduction of mechanized tool CryptoVerif (Blanchet, 2008) is the first automatic tool with computational model developed by Blanchet. At the same time, it does not existing that analysis security of OAuth 2.0 protocol with automatic tool in computational model.

Owning to the previous analysis of OAuth 2.0 protocol is not quite clear, in this study, Blanchet calculus in the computational is used to model to analyze OAuth 2.0 protocol with mechanized tool CryptoVerif.

CONTRIBUTION AND OVERVIEW

During the past several years OAuth 2.0 protocol bas been implemented and deployed by Facebook, Twitter, LinkedIn, Google, Yahoo,. Sina. QQ. Renren and others.

Corresponding Author: Bo Meng, School of Computer, South-Center University for Nationalities, MinYuan Road No. 708, Hong Shan Section, Wuhan, Hubei, 430074, China

In order to verify the security properties of security protocol and improve the confidence of the people, symbolic model and computation model have been developed form the beginning of the 1980s. Computational model bases on complexity and probability. At the same time computational model uses a strong notion of security, guaranteed against all probabilistic polynomial-time attacks and is more realistic. The related references show that the Security properties of OAuth 2.0 protocol have been analyzed with informal method, or with symbolic method, or with computational model by hand which depends on experts' knowledge and ability and is prone to make mistakes. Until now there does not exist that security analysis of OAuth 2.0 protocol with mechanized tool in computational model.

So analysis of security properties of OAuth 2.0 protocol with automatic tool in computational model plays an important role in security protocol field and is a significant work. Hence, in this study, Blanchet calculus in the computational is used to analyze OAuth 2.0 protocol with mechanized tool CryptoVerif.

The main contributions of this study are summarized as follows detail:

- The status of analysis in OAuth 2.0 protocol including in symbolic model and in computational model is presented. The related references show that OAuth 2.0 protocol has been analyzed with informal method, or with symbolic method, or with computational model by hand. Until now there does not existing that analysis security of OAuth 2.0 protocol with automatic tool in computational model
- Applying Blanchet calculus in computational model with active adversary for automatic analysis of OAuth 2.0 protocol. So the term, process and correspondence assertion in Blanchet calculus are used to model authentication in OAuth 2.0 protocol. The authentication is expressed by non-injective or injective correspondence. The analysis itself is executed by automatic tool CryptoVerif developed by Blanchet. Figure 1 shows the model of automatic verification of OAuth 2.0 protocol
- The result shows that OAuth 2.0 protocol has not authentication between the resource owner and authorization server, authentication between client and authorization server. The first automatic analysis on OAuth 2.0 protocol in computational model of in active adversary is implemented in this study

RELATED WORK

Here, the status of OAuth 2.0 protocol and its proof in symbolic model and in computational model are discussed. The related references show that security properties of OAuth 2.0 protocol are analyzed with informal method, or with symbolic method, or with computational model by hand. There does not existing that analysis of OAuth 2.0 protocol with automatic tool in computational model.

Chari *et al.* (2011) use universally composable security framework to formalize by hand the authorization code mode in OAuth2.0 protocol. They propose an ideal functionality for delegation of web access to a third-party where the authentication scheme is based on password. Corella and Lewison (2011) give an informal security analysis of OAuth 2.0 protocol by hand. They mainly focus on denial of service attacks on OAuth 2.0 protocol. Shi *et al.* (2012) introduce the principle of OAuth 2.0 protocol and analyze the procedure of refresh token and propose a framework of OAuth2.0 server objected to specific application. But they do not analyze its security properties.

REVIEW OF BLANCHET CALCULUS

Here, Blanchet calculus (Blanchet, 2008) is reviewed. Blanchet calculus is a probabilistic polynomial calculus and has been carefully designed to make the automated

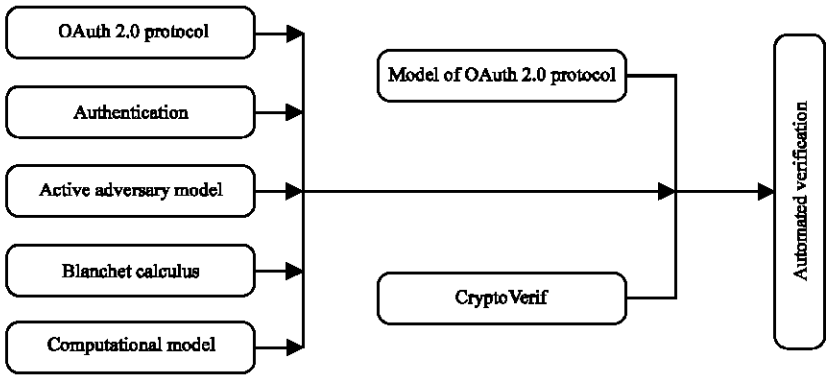

Fig. 1: Model of automatic verification of OAuth 2.0 protocol

proof security protocols. In this calculus, messages are bitstrings and cryptographic primitives are functions operating on bitstrings. Blanchet calculus includes terms and processes. Next the notations used in Blanchet calculus are discussed.

In Blanchet calculus c denote a countable set of channel names. The variable access $x[M_1,...,M_m]$ returns the content of the cell of indices $M_1,...,M_m$ of the m-dimensional array variable x. The function application $f(M_1,...,M_m)$ returns the result of applying function f to $M_1,...,M_m$. Using different channels for each input and output allows the adversary to control the network. The find construct allows us to access arrays.

Process includes two types of processes: Input processes Q ready to receive a message on a channel; output processes P which output a message on a channel after executing some internal computations. The input process include processes nil 0, parallel composition Q|Q', replication times $!^{i\leq n}Q$, channel restriction new channel c; or Q input $c[M_1, L, M_l]$. The input process 0 does nothing; Q|Q' is the parallel composition of Q and Q'; $!^{i\leq n}Q$ represents n copies of Q in parallel, each with a different value of $i\in[1, n]$; new channel c; Q creates a new private channel n and executes Q. The output process include processes $\text{output } \overline{c[M_1,\cdots M_l]}\langle N_1,\cdots,N_k\rangle;Q$, random number new $x[i_1,...i_m]$: T; P, assignment let $x[i_1,...i_m]$: T = M in P, conditional if defined $(M_1,...M_l)\wedge M$ then P else P', event event $e(M_1,...M_m)$; P, array lookup find $(\oplus_{j=1}^{m} u_{j1},\cdots,u_{jm_j}[\tilde{i}] \leq n_{jm_j}$ such that defined $(M_{j1},\cdots,M_{jlj}) \wedge M_j$ then $P_j)$, new $x[i_1,...i_m]$: T; P chooses a new random number uniformly in $I_n(T)$, stores it in $x[i_1,...i_m]$ and executes P. let $x[i_1,...i_m]$: T = M in P stores the bitstring value of M in $x[i_1,...i_m]$ and executes P. find $(\oplus_{j=1}^{m} u_{j1},\cdots,u_{jm_j}[\tilde{i}] \leq n_{jm_j}$ such that defined $(M_{j1},\cdots,M_{jlj}) \wedge M_j$ then $P_1)$ else P means that it tries to find a branch J in [1, m] such that there are values of $u_{j1},\cdots,u_{jm_j}$ for which $M_{j1},...M_{jlj}$ are defined and M_j is true. In case of success, it executes P. In case of failure for all branches; it executes P. The formula event $e(M_1,...M_m)$ holds when the event $e(M_1,...M_m)$ has been executed.

MECHANIZED PROOF TOOL CRYPTOVERIF

This section presents a brief overview of the mechanized prover CryptoVerif, formalize OAuth 2.0 protocol using it (http://www.cryptoverif.ens.fr). In most cases, the prover succeeds in proving the desired properties when they hold and obviously it always fails to prove them when they do not hold. In other words CryptoVerif is sound but not complete which means that it cannot prove are not necessarily invalid.

The mechanized prover CryptoVerif can directly prove security properties of cryptographic protocols in the computational model in which the cryptographic primitives are functions on bit-strings and the adversary is a polynomial-time Turing machine. It also can prove secrecy properties and events that can be executed only with negligible probability. CryptoVerif runs either automatically or interactively, in which case it receives guidance from the user for selecting transformations.

Authentication is modeled as correspondences, much as in symbolic models. Computationally, correspondences assert that, if some event is executed, then other events must also have been executed, at least once, with matching parameters, at least with overwhelming probability. CryptoVerif can deal with more general properties expressed as logical formulas; also both injective and non-injective properties can be analyzed. A non-injective correspondence is a property of the form "if some events have been executed, then some other events have been executed at least once". Injective correspondences are properties of the form "if some event has been executed n times, then some other events have been executed at least n times". Injective correspondences are more difficult to check than non-injective ones, because they require distinguishing between several executions of the same event.

CryptoVerif operates in two modes: a fully automatic and an interactive mode. The interactive mode which is best suited for protocols using asymmetric cryptographic primitives, requires a CryptoVerif user to input commands that indicate the main game transformations the tool should perform. CryptoVerif is sound with respect to the security properties it shows in a proof but properties it cannot prove are not necessarily invalid.

REVIEW OF OAUTH 2.0 PROTOCOL

OAuth 2.0 protocol is used to access a protected resource for clients in the name of a resource owner. OAuth 2.0 protocol includes four roles: protected resource, resource server and client and resource owner. Before a client want to access a protected resource, it must first receive authorization, for example, access token, from the resource owner, then get the access token with the access grant. Finally the client accesses the protected resource by sending the access token to the resource server. Clients in OAuth 2.0 protocol can be fall into four categories: web server, user-agent and native application and autonomous. Here the analysis mainly focus on the web server-based client. At the same time the access grant type is authorization code.

Figure 2 describes the procedure of the abstracted flow in OAuth protocol and is the authorization code mode of the OAuth 2.0 protocol where the authentication scheme is based on password and the client profile is web

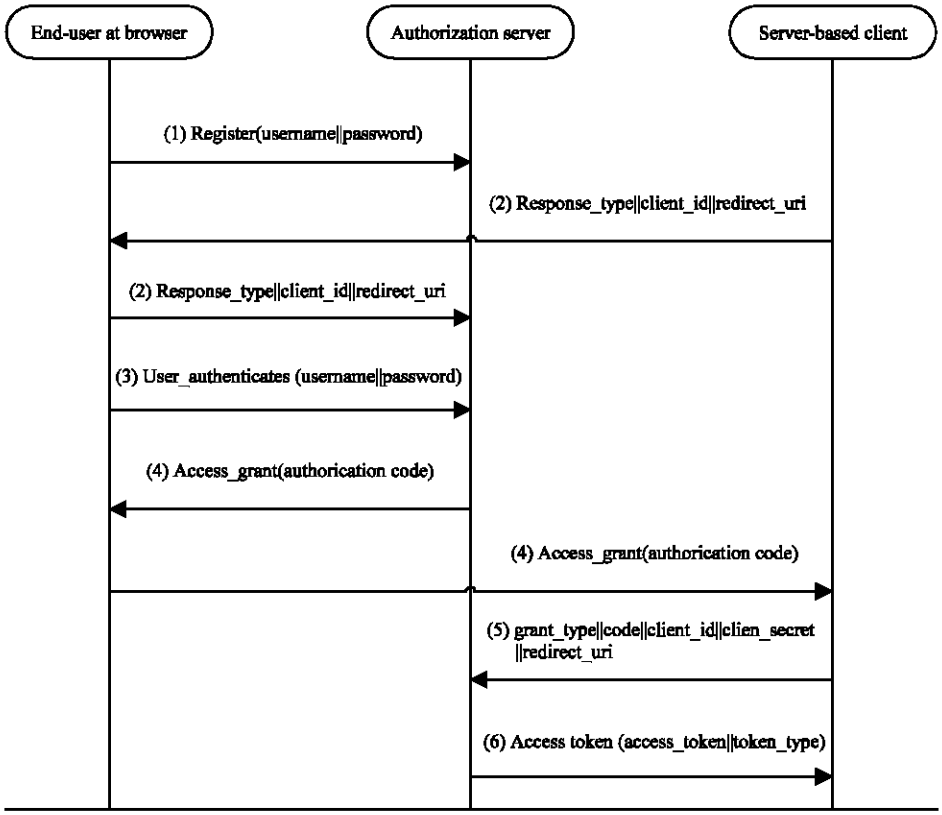

Fig. 2: Procedure of the abstracted flow in OAuth 2.0 protocol

server. First the end-user must be authenticated with username and password of end-user by authorization server. At the same time the client also must be authenticated with the information mainly including client type and redirect_uri by authorization server. Authorization server gets the authorization request and then generates client_id and client_secret which is used to authenticate the identity of client.

End-user registration request: End-user registers to authorization server with his username and password. So End-user generates message (1) including his username and password and deliver it to authorization server. Thus the procedure of registration is completed.

Client authorization request: If client wants to request the resource of the protect resource owner in resource server, he must generate the message (2) client authorization request which mainly consists of the three parameters: respondence_type, client_id and redirect_uri and send it to authorization server. The parameter response_type which is "required" must be access token "token" or authorization code "code" and "access token and authorization code code_and_token". The parameter response_type is used to issue the type of authorization response by authorization server. The parameter scope which is "required" is used to describe the scope of the access request and is generated by authorization server. The parameter client_id which is "required"and is used to express the identifier of end-user and is issued by authorization server after end-user has registered to authorization server. The parameter redirect_uri which is "required"and redirect the user-agent to the URI after complete of end-user authorization.

The authorization server validates the (2) client authorization request to make sure that all required parameters are exist and valid. If the request is invalid, the authorization server redirects the user-agent back to the client using the redirection URI provided with the appropriate error code. If the request is valid, the authorization server authenticates identifier of resource owner with the parameters username and password and gets the access grant of end-user. When access grant has been received, the authorization server directs the end-user's user-agent, for example, browser, to the provided client redirection URI with the help of an HTTP redirection response, or other means available to it via the end-user's user-agent.

End-user authentication and authorization request: The authorization server authenticates end-user with message (3) including the parameters username and password of end-user. If the parameters username and password is valid then it issues that the end-user agrees on the access request.

Authorization server authorization response: After authorization server authenticate end-user with the parameters username and password of end-user. Then authorization server generates message (4) authorization response which mainly includes an access token, an authorization code, or both. At the same time it delivers them and the parameters cliend_id and redirect_uri to the client by the redirection URI.

Access token request: The client gets the authorization code from the authorization server, then he can obtain an access token before accessing the protect resource. The access token is generated by authorization server.

The client constructs the following parameters grant_type, code, client_id, client_secret, rdirect_uri and scope to generate the message (5) access token request and deliver to the authorization server. The value of the parameter grant_type must be authorization_code. The value of the parameter code is consisting with to the value of the received code from the authorization response. The value of the parameter client_id and redirect_uri is same to the value of the received client_id and redirect_uri in authorization request. At the same time the value of the parameter client_secret is same to the value of the received client_id in authorization server generated.

When the authorization server receives the access toke request he must finish the following verifications:

- Firstly, verification of identifier of the client is executed. The authorization server mainly verifies the value of the parameters client_id and client_secret. If the result failed, then the authorization server sends the response error message to the client
- Secondly, verification of code is run. The authorization server checks whether the value of the parameter grant_type does match the value of the parameter authorization_code. At the same time he also checks the value of code is valid and the parameters of code and client_id are consistent
- Finally verification of redirect_uri is executed. The authorization server need to verify that whether the value of the parameter redirect_uri is valid or not. At the same time he also checks that whether the value of the parameter redirect_uri does match the value of the parameter redirect_uri in authorization request or not. If the two verifications failed, then authorization server sends the response error message to the client

If the previous three verifications all succeed then the authorization server generates the access token response message (6).

Access token response: If the access token request from client is valid and verified by authorization server, then he creates the access token response message (6) mainly including the parameters access_token, token_type, expires_in, refresh_token, scope and so on. At the same time those parameters are put into the JSON structures and are sent to the client by HTTP protocol. The value of the parameter access_token which is "required" and is access token which is used to access the protect resource from the resource owner. The value of parameter token_type which is "required" and is used to describe the type of access_token. The value of the parameter expires_in which is "optional "and depicts the duration in seconds of the access token lifetime. The parameter scope which is "required" is used to describe the scope of the access request and is generated by authorization server. The value of the parameter refresh_token which is "optional" and is used to obtain a new access tokens based on the same end-user access grant. The authorization server should not issue a refresh token when the access grant type is an assertion or a set of client credentials. The client accesses protected resources through presenting an access token to the resource server. The resource server must verify the access token and ensure it has not expired and that its scope covers the requested resource.

FORMALIZING OAUTH 2.0 PROTOCOL IN BLANCHET CALCULUS

Here, we use non-injective correspondences and Injective correspondences to model authentication from authorization server to end-user and from authorization server to client. First non-injective correspondences are used: event event authorization (x) ==> user (x) is used to authenticate end-user by authorization server. Event event authorization (x) ==> client (x) is used to authenticate client by authentication server. Injective correspondences are then used: event event inj: authorization (x) ==> inj: user (x) is used to authenticate end-user by authorization server. Event event inj: authorization (x) ==> inj: client (x) is used to authenticate client by authentication server. Figure 3 describes the events and correspondence.

The complete formal model of OAuth 2.0 protocol in Blanchet calculus is given in figures. Figure 4-8 report the basic process include authorization_server process, client process, web_server process and end_user process in authentication forming the model of OAuth 2.0 protocol. The process EPP of 3 KP process in Fig. 4 is assumed to run in interaction with an adversary which also models the network.

```
event User(username).
event Client(code).
event Authorization(username).
event Authorizationa(code).


query x:username;
event Authorization(x)==>User(x).
query x:username;
event inj:Authorization(x)==>inj:User(x).
query x:code;
event Authorizationa(x)==>Client(x).
query x:code;
event inj:Authorizationa(x)==>inj:Client(x).
```

Fig. 3: Query events

In (atart ( )) means that the protocol is executed. The authorization server is formalized as authorization_server process. Client process is modeled as client process. Web server process is formalized as web_server process. End-user is modeled as end_user process. $!^{n1}$client process means n client process parallels run.

End-user is modeled as End-user process in Fig. 5. End-user process first chooses randomly with uniform probability a bitstring name in the type user name by the construct new name: user name. At the same time it then chooses randomly with uniform probability a bitstring pass in the type password by the construct new pass: password. After that end-user sends the name and pass through the channel c1 by the construct $\overline{c1}\langle name, pass\rangle$. Then End-user process receives message string_from_author in the type bistring from the channel c2 by the construct c2(string_from_author.bistring). If the message string_from_author is error, then it outputs null message, otherwise the event event user(name) is executed. Finally End-user process outputs message (A, name, pass) by the construct $\overline{c3}\langle A, name, pass\rangle$ and it ends.

Client process is formalized as client process in Fig. 6. Client process first receives information from the

$$let\ OAuth2.0process=in\ (atart(\ ))\left((Authorization\_serverProcess)\middle|(!^{n1}ClientProcess)\middle|(!^{n2}Web\_serverProcess)\middle|(!^{n3}End\text{-}userProcess)\right)$$

Fig. 4: OAuth 2.0 protocol process

```
let End-userProcess=
    c( );
    new name:username;     (*generate username*)
    new pass:password;     (*generate password*)
    c1 ⟨name,pass⟩;   (*registries to authorization server*)
    c2(string_from_author:bitstring);     (*receive message from authorization server*)
    if string_from_author=error then          (*verification*)
    c ⟨ ⟩
    else
    event User (name);
     c3⟨A,name,pass⟩. (*send name and password to authorization server*)
```

Fig. 5: End-user process

```
let ClientProcess=
      c( );
      new uri:redirect_uri;                                          (*generate redirect_uri*)
       c4 ⟨uri⟩;                                                     (*send redirect_uri to authorization server*)
       c5 (clientid_:client_id,clientsecret_:client_secret);         (*receive client_id and client_secret from authorization server*)
       new responsetype:response_type;                               (*generate response_type*)
       c6 ⟨responsetype,clientid_,uri⟩;                              (*output response_type,client_id and redirect_uri to web server*)
       c7 (code_form_web:code);                                      (*receive code from web server*)
       event Client(code_form_web);
       new granttype:grant_type;                                     (*generate grant_type*)
       c8 ⟨granttype,code_form_web,clientid_,clientsecret_,uri⟩;     (*send grant_type,code,client_ied,client_secret,redirect_uri to authorization server*)
       c9 (accesstoken:access_token,tokentype_from_a:token_type). (*receive access_token and token_type from Authorizationserver*)
```

Fig. 6: Client process

```
let Web_serverProcess=
        c10(response:response_type,clientid_form_c:client_id,uri_from_c:redirect_uri);
        (*receive from response_type,client_id,redirect_uri client*)
        c11 ⟨web,response,clientid_form_c,uri_from_c⟩;
        (*send the parameters to Authorization server*)
        c12 (=web,code_from_a:code); (*receive code Authorization server*)
        c13⟨code_from_a.⟩ (*send code to client*)
```

Fig. 7: Web server process

```
let Authorization_serverProcess=
     c14 (name_reg:username,password_reg:password);      (*user registers*)
     find i<=N suchthat defined name_reg_success i&&name_reg_success [i]=name_reg then      (*check*)
     c̄ ⟨error⟩        else
          let name_reg_success:username=name_reg in                              (*store username of user*)
          let password_reg_success:password=password_reg in                      (*store password of user*)
          let name_password_sucess:bitstring=concatAname_reg_success,password_reg_success in c̄⟨ ⟩
          c15 (uri_reg:redirect_uri);                    (*accept application of client*)
          find j<=N suchthat defined (uri_reg_success [j])&&(uri_reg_success [j] =uri_reg) then   (*check*)
               c̄ ⟨ ⟩ else       let uri_reg_success:redirect_uri=uri_reg in                    (*store uri*)
                    new clientid:client_id;
                    new clientsecret:client_secret;
                    let id_secret_success:bitstring=concatB(clientid,clientsecret) in
                    let id_uri_success:bitstring=concatC(clientid,uri_reg_success) in
                    c16 ⟨clientid,clientsecret⟩;
                      (*generate client_id and client_secret and deliver it client*)
                      c17 (=web,responsetype_one:response_type,clientid_one:client_id,uri_one:redirect_uri);
                      (*receive authorization requiest from web server*)
          if responsetype_one=code_response then                                 (*verify*)
                    c18 ⟨Authentication_message⟩;            (*verification of identifier of user*)
                    c19 (=A,name_author:username,password_author:password);
                      (*receive username and password of user*)
               find k<=N suchthat defined name_password_sucess k&&
(name_password_sucess [k]=concatA (name_author,password_author)) then                (*verification*)
               find p<=N suchthat defined (name_reg_success [p]&&(name_reg_success [p]=name_author) then
                     event Authorization (name_author);
               let id_uri_one:bitstring=concatC (clientid_one,uri_one) in
               find l<=N suchthat defined (id_uri_success[l])&&(id_uri_success [l]=id_uri_one) then
               (*verify client_id from client is match to redirect_uri*)
               new author_code:code;
               let id_code_one:bitstring=concatD (clientid_one,author_code) in                (*generate code*)
               c20 ⟨web,author_code⟩;                                          (*deliver code to web server*)
  c21 ( granttype_two:grant_type,code_for_author_two:code,clientid_ ); ( *receive grant_type,code,client_id,client )
      ( two:client_id,clientsecret_two:client_secret,uri_two:redirect_uri )   ( _secret,redirect_uri from client*  )

   if granttype_two=code_grant then              (*check whether grant_type is code or not*)
  find m<=N suchthat defined (author_code[m])&&(author_code [m]=code_for_author_two) then
  (*check the valid of code and whether the code match client_id or not*)
  find x<=N suchthat defined (id_secret_success [x])&&
(id_secret_success [x]=concatB(clientid_two,clientsecret_two)) then (*client_id match client_secret*)
  find y<=N suchthat defined (id_uri_one [y])&&(id_uri_one [y]=concatC(clientid_two,uri_two)) then (*uri match redirect_uri*)
  find z<=N suchthat defined (id_code_one [z])&&(id_code_one [z]=concatD (clientid_two,code_for_author_two)) then
  event Authorizationa (code_for_author_two);
  new token:access_token;          (*generate access_token*)
  new tokentype:token_type;       (*generate token_type*)
  c22 ⟨token,tokentype⟩.       (*output token,tokentype*)
```

Fig. 8: Authorization server process

channel c by the construct c( ) and then he chooses randomly with uniform probability a bitstring uri in the type redirect_uri by the construct new uri:redirect_uri. After that he outputs the parameter uri through the channel c4 by the construct $\overline{c4}\langle uri\rangle$. Client process receives information clientid_:client_id, client secret_:

client secret through the channel c5 by the construct c5(clientid_:client_id, client secret_: client secret) and chooses randomly with uniform probability a bitstring response type in the type response_type by the construct new response type: response_type. After that Client process send authorization request response type, clientid_,uri through the channel c6 by the construct $\overline{c6}\langle$responsetype,clientid_,uri$\rangle$. If the authorization server has successfully verify the authorization request then he sends the an authorization code code_from_web to the Web server process. After that the web server process sends it to Client process. Thus the Client process receives the authorization code code_from_web through the channel c7 by the construct c7(code_from_web:code). At the same time the event event client (code_from_web) is executed. Client process chooses randomly with uniform probability a bitstring grant type in the type grant_type by the construct new granttype_grant type. Thus he constructs access token request message granttype, code_from_web, cliented_, clientsecret_, uri and sends it to the Authorization server process through the channel c8 by the construct $\overline{c8}\langle$granttype, code_from_web, cliented_clientsecret_, uri$\rangle$. Client process receives the access token accesstoken: access_token, tokentype_from_a: token_type through the channel c9 by the construct c9(access token accesstoken: access_token, tokentype_from_a: token_type).

Web server process is modeled as web_server process in Fig. 7. Web server process receives the authorization request response: respose_type, cliented_from_c: client_id, uri_from_c: redirect_uri through the channel c10 by the construct c10(response: respose_type, cliented_from_c: client_id, uri_from_c: redirect_uri). After that web server process delivers the authorization request web, response, cliented_from_c, uri_from_c through the channel c11 by the construct $\overline{c11}\langle$web, response, cliented_from_c, uri_from_c$\rangle$ to authorization server. At the same time web server process receives the authorization code code_from_a from the channel c12 by the construct c12(=web, code_from_a: code) then delivers it to the client process through the channel c13 by the construct $\overline{c13}\langle$code_from_a$\rangle$.

Authorization server process is formalized as authorization_server process in Fig. 8. In the model the resource owner first registers to authorization server. Authorization server receives the username and password information of resource owner name_reg:username, password_reg: password through the channel c14 by the construct c14(name_reg:username, password_reg: password). And then it uses the construct find I<=N such that defind (name_reg_success [I])&&(name_reg_ success [i] = name_reg) to verify the identity of resource owner. If the verification fails then it sends error message, otherwise it stores username and password of resource owner using the constructs let name_reg_success: username = name_reg in and let password_reg_success: password = password_reg in. At the same time authorization server binds the username name_reg_ success and password password_reg_success using the construct contactA(name_reg_success, password_reg_ success) and stores it as name_password_success. At the same time authorization server also receives the application of client process uri_reg:redirect_uri through the channel c15 by the construct c15(uri_reg:redirect_uri). And then uses the construct find j<=N such that defined (uri_reg_success[j])&&(uri_reg_success [j] = uri_reg) to verify the identity of client. If the verification fails then it sends the error message, otherwise it generates the identity clientid and secret key clientsecret of client by the constructs new clientid:client_id and new clientsecret: client_secret. Then authorization server process binds the identity clientid and secret key clientsecret of client using the construct contact B(clientid,clientsecret). At the same time authorization server process binds the client identifier clientid and URI uri_reg_success using the construct contact C(clientid, uri_reg_success). And then authorization server delivers the client identification clientid and client secret key clientsecret through the channel c16 by the construct $\overline{c16}\langle$clientid,clientsecret$\rangle$ to the client process.

Authorization server receives the authorization request information responseone_one:response_type, clientid_one:client_id, uri_one: redirect_uri through the channel c17 by the construct c17 (=web,responsetype_ one:response_type,clientid_one:clientid_id, uri_one: redirect_uri) from the web server process. And then it verifies the authorization request information. If the verification is successful, then it sends authentication_ message through the channel c18 by the construct $\overline{c18}\langle$authentication_message$\rangle$ to the resource owner. Authorization server first authenticates the identity of resource owner and then allows it to generate authorization code. The authorization server receives the identification information name_author:username, password_author:password through the channel c19 by the construct c19(=A,name_author.username,password_ author:password). Then authorization server uses the construct find k<=N such that defined (name_password_ success[k])&&(name_password_success[k] = contactA (name_author,password_author)) to check the

identification of resource owner. If the verification is successful, then authorization server uses the construct find p<=N such that defined (name_reg_success[p])&& (name_reg_success[p]=name_author) to check the username of resource owner. If the verification is ok, then the event event authorization (name_author) is executed. Authorization server binds client identifier clientid_one and redirect identifier uri_one using the function contact C (clientid_one, uri_one) and stores it in id_uri_one. Then authorization server use the construct find 1<=N such that defined (id_uri_success[1])&&(id_uri_success[1]= id_uri_one) to check the valid of id_uri_one and clientid_one. If the verification succeeds then it generates the authorization code author_code by the construct new author_code:code. After that authorization server uses contact D(clientid_one, author_code) to bind clientid_one and author_code and stores it as id_code_one:bistring. And then authorization server delivers web,author_code through the channel c20 by the construct $\overline{c20}\langle$web, author_code$\rangle$ to web server process. The authorization server receives the authorization request information granttype_two: grant_type, code_for_author_two:code, clientid_two:client_id,clientsecret_two:client_secret, uri_two,redirect_uri through channel c21 by the construct c21(granttype_two: grant_type, code_for_author_two: code,clientid_two:client_id,clientsecret_two:client_secret, uri_two,redirect_uri). Authorization server first checks whether code_grant is equal to value of granttype_two or not. If the verification succeeds, it uses the construct find m<=N such that defined (author_code[m])&&(author_ code[m]=code_for_author_two) to check the valid of authorization code code_for_author_two. If the verification succeeds and then it uses the construct find x<=N suchthat defined (id_secret_success[x])&& (id_secret_success[x]=contactB(clientid_two,client secret_two)) to verify the valid of cliented_two and cliented_one. If the verification is successful and then it uses the construct find y<=N suchthat defined (id_uri_one[y])&&(id_uri_one[y]=contactC(clientid_two, uri_two)) to verify the match of clientid_two and clientid_one and the match of uri_two and uri_one. Finally authorization server uses the construct find z<=N suchthat defined (id_code_one[z])&&(id_code_ one[z]=contactD(clientid_two, code_for_author_two)) to verify the match of (clientid_two,code_for_author_two) and (author_code,id_code_one) .If all previous verifications succeeds, then the event event authorization (code_for_author_two) is executed and authorization generates the access token token by the construct new token: access_token and the token type tokentype by the construct new tokentype:token_type. Finally the authorization server deliver the token,token_type through the channel c22 by the construct $\overline{c22}\langle$token,token_type$\rangle$ to client process.

AUTOMATIC VERIFICATION OF AUTHENTICATION IN OAUTH 2.0 PROTOCOL WITH CRYPTOVERIF

CryptoVerif can take two formats as input: channels Front-end and oracles Front-end. In both cases, the output of the system is essentially the same. The main difference between the two front-ends is that the oracles front-end uses oracles while the channels front-end uses channels. In the channels front-end, channels must be declared by a channel declaration. There is no such declaration in the oracles front-end. Some constructs use a different syntax in the oracles front-end, to be closer to the syntax of cryptographic games.

In this study the form of channels Front-end is used as the input of CryptoVerif. In order to prove the in OAuth2.0 protocol the Blanchet calculus model are needed to be translated into the syntax of CryptoVerif and generated the CryptoVerif inputs in the form of channels Front-end.

Here, non-injective correspondences and Injective correspondences in Table 1 are used to model the authentication from authorization server to end-user and from authorization server to client. First non-injective correspondences are used: event event authorization (x)==>user(x) is used to authenticate end-user by authorization server. Event event authorizationa(x)==> client(x) is used to authenticate client by authentication server. Injective correspondences are then used: event event inj:authorization(x)==>inj:user(x) is used to authenticate end-user by authorization server. Event event inj:authorization(x)==>inj:client(x) is used to authenticate client by authentication server.

Figure 9 and 10 present the code of verification of authentication in CryptoVerif. The analysis was performed by CryptoVerif and succeeded. Table 2 presents the

Table 1: Correspondences in OAuth 2.0 protocol

Correspondences
Event authorization(x)==>user(x)
Event authorization(x)==>client(x)
Event inj:authorization(x)==>inj:user(x)
Event inj:authorization(x)==>inj:client(x)

Table 2: Transforms of games in OAuth 2.0 protocol

From	To	Rule
Game 1	Game 2	Applying simplify [Excluding set(dist 1->2, proba <= 0.5×N×N/|client_id|+0.5×N×N/|code|)] yields
Game 2	Game 3	Applying remove assignments of useless yields

```
const web:host.
const code_response:response_type.
const code_grant:grant_type.
fun concatA(username, password):bitstring [compos].
fun concatB(client_id,client_secret):bitstring [compos].
fun concatC(client_id,redirect_uri):bitstring [compos].
fun concatD(client_id,code):bitstring [compos].
event User(username).
event Client(code).
event Authorization(username).
event Authorizationa(code).
```

```
query x:username;
event Authorization(x)==>User(x).
query x:username;
event inj:Authorization(x)==>inj:User(x).
query x:code;
event Authorizationa(x)==>Client(x).
query x:code;
event inj:Authorizationa(x)==>inj:Client(x).
channel start,c,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,
c11,c12,c13,c14,c15,c16,c17,c18,c19,c20,c21,c22.
```

```
let End-userProcess=
        in(c,()); new name:username;      (*generate username*)
        new pass:password;      (*generate password*)
        out(c1,(name,pass));    (*registrater to authorization server*)
        in(c2,(string_from_author:bitstring));    (*receive message from authorization server*)
        if string_from_author=error then          (*verificaton*)
        out(c,())
        else
        event User(name);
        out(c3,(A,name,pass)).  (*send name and password to authorization server*)
let ClientProcess=
        in(c,());
       new uri:redirect_uri;                              (*generate redirect_uri*)
       out(c4,uri);                                      (* send redirect_uri to authorization server*)
       in(c5,(clientid_:client_id,clientsecret_:client_secret));
       (* receive client_id and client_secret from authorization server *)
       new responsetype:response_type;                    (* generate response_type*)
       out(c6,(responsetype,clientid_,uri));
       (*output response_type,client_id and redirect_uri to web server*)
       in(c7,(code_form_web:code));                       (*receive code from web server*)
       event Client(code_form_web);
       new granttype:grant_type;                          (* generate grant_type*)
       out(c8,(granttype,code_form_web,clientid_,clientsecret_,uri));
      (* send grant_type,code,client_id,client_secret,redirect_uri to Authorization server*)
       in(c9,(accesstoken:access_token,tokentype_from_a:token_type)).
      (* receive access_token and token_type from Authorizationserver *)
```

Fig. 9: Code of protocol in Cryptoverif

```
let Web_serverProcess=
    in(c10,(response:response_type,clientid_form_c:client_id,uri_from_c:redirect_uri));
    (*receive from response_type,client_id,redirect_uri client*)
    out(c11,(web,response,clientid_form_c,uri_from_c));          (* send the parameters to Authorization server*)
    in(c12,(=web,code_from_a:code));                  (*receive code Authorization server *)
   out(c13,(code_from_a)).                                      (*send code to client*)
let Authorization_serverProcess=
    in(c14,(name_reg:username,password_reg:password));                          (* user register*)
    find i<=N suchthat defined (name_reg_success[i])&&(name_reg_success[i]=name_reg) then    (*check*)
    out(c,(error))       else     let name_reg_success:username=name_reg in          (*store username of user*)
                        let password_reg_success:password=password_reg in      (*store password of user*)
                        let name_password_sucess:bitstring=concatA(name_reg_success,password_reg_success) in
                        out(c,()); in(c15,(uri_reg:redirect_uri));                    (* accept application of client*)
          find j<=N suchthat defined (uri_reg_success[j])&&(uri_reg_success[j]=uri_reg) then       (*check*)
                        out(c,()) else     let uri_reg_success:redirect_uri=uri_reg in                    (*store uri*)
                        new clientid:client_id;          new clientsecret:client_secret;
                        let id_secret_success:bitstring=concatB(clientid,clientsecret) in
                        let id_uri_success:bitstring=concatC(clientid,uri_reg_success) in
                        out(c16,(clientid,clientsecret));     (*generate client_id and client_secret and deliver it client*)
                        in(c17,(=web,responsetype_one:response_type,clientid_one:client_id,uri_one:redirect_uri));
                        (*receive authorization requiest from web server*)
          if responsetype_one=code_response then                    (*verify*)
          out(c18,(Authentication_message));                   (*erification of identifier of user*)
          in(c19,(=A,name_author:username,password_author:password));   (*receive username and password of user*)
              find k<=N suchthat defined (name_password_sucess[k])&&
(name_password_sucess[k]=concatA(name_author,password_author)) then   (*verification*)
              find p<=N suchthat defined (name_reg_success[p])&&(name_reg_success[p]=name_author) then
                        event Authorization(name_author);
              let id_uri_one:bitstring=concatC(clientid_one,uri_one) in
              find l<=N suchthat defined (id_uri_success[l])&&(id_uri_success[l]=id_uri_one) then
              (*verify client_id from client is match to redirect_uri*)
                        new author_code:code;
      let id_code_one:bitstring=concatD(clientid_one,author_code) in              (*generate code*)
                        out(c20,(web,author_code));              (*deliver code to web server*)
                        in(c21,(granttype_two:grant_type,code_for_author_two:code,
clientid_two:client_id,clientsecret_two:client_secret,uri_two:redirect_uri));
                        if granttype_two=code_grant then        (*check whether grant_type is code or not*)
                        find m<=N suchthat defined (author_code[m])&&(author_code[m]=code_for_author_two) then
(*check the valid of code and whether the code match client_id or not*)
                        find x<=N suchthat defined (id_secret_success[x])&&
(id_secret_success[x]=concatB(clientid_two,clientsecret_two)) then          (*client_id match client_secret*)
                        find y<=N suchthat defined (id_uri_one[y])&&
(id_uri_one[y]=concatC(clientid_two,uri_two)) then               (*uri match redirect_uri*)
                        find z<=N suchthat defined (id_code_one[z])&&
(id_code_one[z]=concatD(clientid_two,code_for_author_two)) then
                        event Authorizationa(code_for_author_two);
                        new token:access_token;             (*generate access_token*)
                        new tokentype:token_type;            (*generate token_type*)
                        out(c22,(token,tokentype)).          (*output client*)
process
      in(start,());      out(c,());
     ((!N Authorization_serverProcess)|(!N1 ClientProcess)|(!N2 Web_serverProcess)|(!N3 End-userProcess))
```

Fig. 10: Code of protocol in Cryptoverif

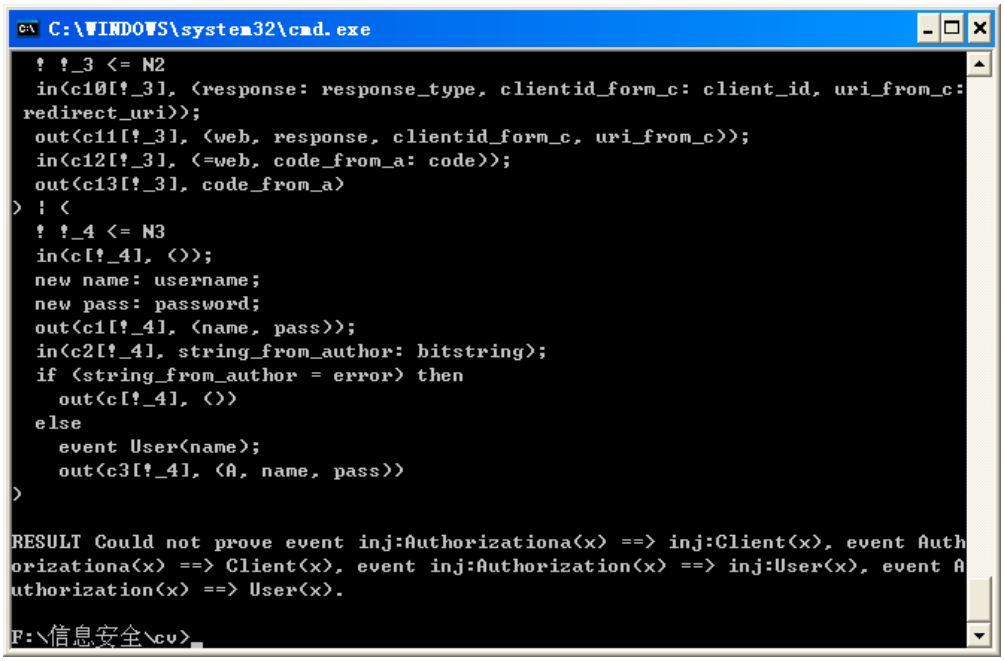

Fig. 11: Result of OAuth 2.0 protocol in Cryptoverif

transforms of games in mechanized proof of OAuth 2.0 protocol. The results are showed in Fig. 11 and OAuth 2.0 protocol is proved not to guarantee authentication without secure measurement, for example, SSL protocol, in computation model.

CONCLUSION

During the past several years OAuth 2.0 protocol bas been implemented and deployed in many enterprises. In order to verify the security properties of security protocol including OAuth 2.0 protocol and improve the confidence of the people, two approaches have been developed form the beginning of the 1980s: symbolic model and computational model. Computational model uses a strong notion of security, guaranteed against all probabilistic polynomial-time attacks and is more realistic.

Hence, in this study, in order to improve the confidence of people, Blanchet calculus in the computational model is used to analyze OAuth 2.0 protocol with mechanized tool CryptoVerif. The term, process and correspondence assertion in Blanchet calculus are used to model authentication in OAuth 2.0 protocol. The analysis itself is executed by automatic tool CryptoVerif developed by Blanchet. The result shows that OAuth 2.0 protocol has not authentication between the resource owner and authorization server, authentication between client and authorization server without any other security mechanism. The first automatic analysis on OAuth 2.0 protocol in computational model of in active adversary is implemented in this study.

In the future, it is an interesting work on the verification of the Java implementation of OAuth 2.0 protocol in computational model.

ACKNOWLEDGMENTS

This study was supported in part by Natural Science Foundation of The state Ethnic Affairs Commission of PRC under the grants No: 12ZNZ008, titled “Automatic verification of cryptographic security in security protocol Java code”, conducted in Wuhan, China from 1/1/2013 to 30/12/2013.

REFERENCES

Blanchet, B., 2008. A computationally sound mechanized prover for security protocols. IEEE Trans. Dependable Secure Comput., 5: 193-207.

Chari, S., C. Jutla and A. Roy, 2011. Universally composable security Analysis of OAuth v2.0. http://eprint.iacr.org/2011/526.pdf

Corella, F. and K.P. Lewison, 2011. Security Analysis of double redirection protocols. http://pomcor.com/techreports/DoubleRedirection.pdf

Meng, B. and F. Shao, 2010. Computationally sound mechanized proofs for deniable authentication protocols with a probabilistic polynomial calculus in computational model. Inform. Technol. J., 10: 611-625.

Meng, B., 2011. A survey on analysis of selected cryptographic primitives and security protocols in symbolic model and computational model. Inform. Technol. J., 10: 1068-1091.

Shi, Z.Q., J.L. Liu and X.H. Tan, 2012. Authentication and authorization technique based on OAuth2.0. Comput. Syst. Appl., 21: 260-264.