Verifica formale di OAuth 2.0 draft 22 - Authorization Code con autenticazione MAC, mediante l'utilizzo di Proverif

Marco De Nadai

13 novembre 2011

1 Introduzione

Nell'autenticazione tradizionale client-server, il client richiede l'accesso ad una risorsa protetta al server che la ospita , autenticandosi necessariamente tramite le credenziali del legittimo possessore delle risorse.

Il possessore delle risorse, deve condividere le proprie credenziali per permettere l'accesso alle risorse protette da parte di un'applicazione di terze parti. Questo crea diversi problemi e limitazioni:

- Le applicazioni di terze parti necessitano di memorizzare le credenziali per un uso futuro. Queste credenziali sono tipicamente costituite da password in chiaro.
- I server devono supportare l'autenticazione tramite password, nonostante la debole sicurezza di questo metodo.
- Le applicazioni di terze parti ottengono un accesso troppo ampio alle risorse, senza che il loro possessore possa restringerne permessi o durata.
- I possessori delle risorse non possono revocare l'accesso ad una determinata applicazione di terze parti, senza per questo revocare l'accesso a tutte le applicazioni di terze parti.
- La compromissione di una qualsiasi applicazione di terze parti, provoca l'automatica compromissione delle credenziali e delle risorse protette da tali credenziali.

OAuth 2.0 affronta questi problemi introducendo un'entità di Autorizzazione e separando il ruolo del client (l'applicazione di terze parti), da quello del resource owner (il possessore delle risorse). Il Client in questo modo richiede determinati permessi al resource owner tramite un Authorization server, e li ottiene sottoforma di credenziali a scadenza (diverse da quelle del resource owner). A questo punto richiede al Resource server le risorse di cui necessita utilizzando le credenziali ottenute.

2 Proverif

Proverif è uno strumento di analisi automatica della sicurezza dei protocolli di crittografia, creato da Bruno Blanchet. Proverif permette di simulare la crittografia simmetrica ed asimmetrica, funzioni di hash, firme digitali e molto altro, consentendo quindi di verificare desiderate proprietà come corrispondenze, equivalenze, privacy e tracciabilità. Non si limita alla sola verifica

delle proprietà dei protocolli, ma permette anche la ricostruzione di possibili attacchi. Questo è particolarmente utile all'individuazione (e quindi risoluzione) dei problemi ai protocolli di sicurezza.

L'utilità di questo strumento emerge sopratutto nella verifica di complessi protocolli e funzioni, le cui analisi manuali sarebbero difficoltose e potrebbero essere maggiormente influenzati da errori umani.
Qui di seguito ci limiteremo a presentare le caratteristiche del software che utilizzeremo nel nostro modello. Per una completa descrizione si rimanda alla lettura del manuale di uso [1].

Dichiarazioni Nel pi-calculus tutti i processi sono composti da un numero finito di variabili (di diverso tipo) e da costruttori che possono essere associati da un numero finito di distruttori. Il linguaggio è fortemente tipizzato ed è possibile definire un nuovo tipo di variabile tramite l'istruzione `type`.

```
type t.
```

Tutte le variabili che appaiono all'interno del listato sono dichiarate tramite il costrutto free, mentre le costanti (ossia variabili che non cambiano mai di valore dopo la loro definizione) utilizzano il costrutto `const`.

```
free variablename:t.
const variabilename2:t.
```

La sintassi `channel c` dichiara un canale di comunicazione pubblico, utilizzabile tramite i costrutti `in` ed `out` per simulare la trasmissione di informazioni attraverso il canale specificato.

```
channel c.
in(c, variablename).
out(c, variablename).
channel c [private].
```

Tutti i canali e le variabili sono di default pubblici e le informazioni trasmesse sono quindi conosciute a tutte le parti in gioco, compresi gli avversari/attaccanti. Per evitarlo è possibile postporre il `[private]` alla dichiarazione.

```
channel c [private].
```

Per dichiarare una variabile all'interno di un processo, si utilizza `new`. Queste variabili sono dichiarate private e sono conosciute solamente al processo che le contiene. Il `new` è utile per modellare nonces e chiavi per esempio.

```
new N:bitstring.
```

Processi Sono i contenitori delle azioni compiute da un'entità. Il processo `0` non fa nulla; i processi `P|Q` rappresentano due processi che sono eseguiti in parallelo e `!P` indica una composizione `P|P|P...` infinita e modella un numero infinito di sessioni del processo P.

MAC/hash Per modellare le funzioni di hash è possibile dichiarare una funzione, senza inversa o equazioni.

```
fun h(bistring):bitstring
```

In modo similare, le funzioni MAC possono essere formalizzate da un costruttore, senza distruttori o equazioni associate.

```
type mkey.
fun mac(bitstring, mkey):bitstring.
```

Crittografia simmetrica La crittografia simmetrica è modellizzabile tramite una funzione costruttore ed il suo relativo distruttore, che ne simula la funzione inversa.

```
fun senc(bitstring, bitstring): bitstring.
reduc forall m:bitstring, k:bitstring; sdec(senc(m,k), k) = m.
```

Tabelle Tramite il costrutto `table` è possibile dichiarare tabelle che fungono da veri e propri database per la memorizzazione di dati in modo permanente. E' possibile l'inserimento (`insert`) e l'estrazione dei dati (`get`), non è permessa tuttavia la loro rimozione.

```
table d(t1,...,tn).
insert d(M1,...Mn);
get d(T1,...,Tn) in
```

Query ed eventi Con lo scopo di verficare le proprietà del protocollo analizzato ci serviamo di un costruttore `query`, abbinato ad altri due costruttori di Proverif: `event`, `attacker`. Il primo ci permette di stabilire il raggiungimento di un determinato obiettivo (od un punto di un protocollo) permettendoci di analizzare la corrispondenza di eventi, il secondo ci permette di dichiarare attaccanti, quindi di verificare la segretezza.

```
query event(e3).
query eventi(e0) => event(e1).
query event(e2) => event(e1) || event(e0).
query attacker:key.
```

Listato d'esempio

```
1  (* listing source of Proverif Manual *)
2
3  free c:channel.
4
5  free Cocks: bitstring [private].
6  free RSA: bitstring[private].
7
8  query attacker(RSA).
9  query attacker(Cocks).
10
11 process
12          out(c,RSA);
13          0
```

Come si può osservare alla linea 1 abbiamo un commento, alla linea 3 la dichiarazione pubblica di un canale ed alla linea 5-6 la dichiarazione di due variabili private. Passiamo in linea 8-9 alle query che verificheranno la segretezza delle variabili. Alla linea 11 abbiamo il processo principale che invia nel canale `c` la variabile `RSA` e termina.

Per utilizzare questo software, l'utente non deve far altro che specificare il protocollo da verificare (il listato precedente), in proposizioni di Horn o processi in pi-calculus, impostare le proprietà da verificare (per esempio `query attacker(v)`) e lanciare l'analisi che produrrà un output come questo:

```
1  Process:
2  {1}out(c, RSA)
3  -- Query not attacker(Cocks[])
4  Completing...
5  Starting query not attacker(Cocks[])
6  RESULT not attacker(Cocks[]) is true.
7  -- Query not attacker(RSA[])
8  Completing...
9  Starting query not attacker(RSA[])
10 goal reachable: attacker(RSA[])
11 1. The message RSA[] may be sent to the attacker at output
       {1}.
12 attacker(RSA[]).
13 A more detailed output of the traces is available with
14 set traceDisplay = long.
15 out(c, RSA) at {1}
16 The attacker has the message RSA.
17 A trace has been found.
18 RESULT not attacker(RSA[]) is false.
```

Dall'output di questo esempio possiamo notare l'analisi e di successivi risultati che denotano la non segretezza di `RSA` (che viene spedito in chiaro nel canale c e quindi attaccato) e la segretezza di `Cocks` (che non viene nemmeno trasmesso, quindi resta segreto).

3 OAuth

Le specifiche ufficiali OAuth 2.0 [2], definiscono quattro modalità di ottenimento delle credenziali, permettendo l'utilizzo di questo protocollo in diversi ambienti e situazioni.

Authorization Code Viene utilizzato da client di terze parti che riescono a mantenere la confidenzialità delle credenziali utente. Autentica il client e fornisce il Token direttamente, senza esporlo ad altri (compreso il resource owner ed il suo User Agent).

Implicit Grant E' la semplificazione della modalità precedente, ottimizzata per Client implementati in browsers o dispositivi mobile. Riduce il numero di scambi necessari per ottenere il Token, non autentica il Client, espone il token all'User Agent del resource owner e non è consigliabile se è possibile utilizzare l'Authorization Code.

Resource Owner Password Credentials Dovrebbe essere utilizzato solo quando c'è la presenza di un altissimo livello di fiducia tra resource owner e Client (ad esempio il sistema operativo e/o un'applicativo privilegiato).

Client Credentials Utilizzato come un authorization grant quando lo scopo dell'applicazione è limitato a proteggere le risorse sotto il controllo del Client.

Il protocollo dispone anche di una certa flessibilità riguardo i Token: possono infatti avere differenti formati, strutture e metodi di utilizzo (definiti da proprietà crittografiche). I tipi di Token definiti nell'emphAccess Token Type Registry ad oggi sono MAC [4] e Bearer [5]. Noi modelleremo il primo dei due, caratterizzato da una complessità maggiore del secondo, dovuto al possibile utilizzo del Token in connessioni in chiaro (senza TLS).

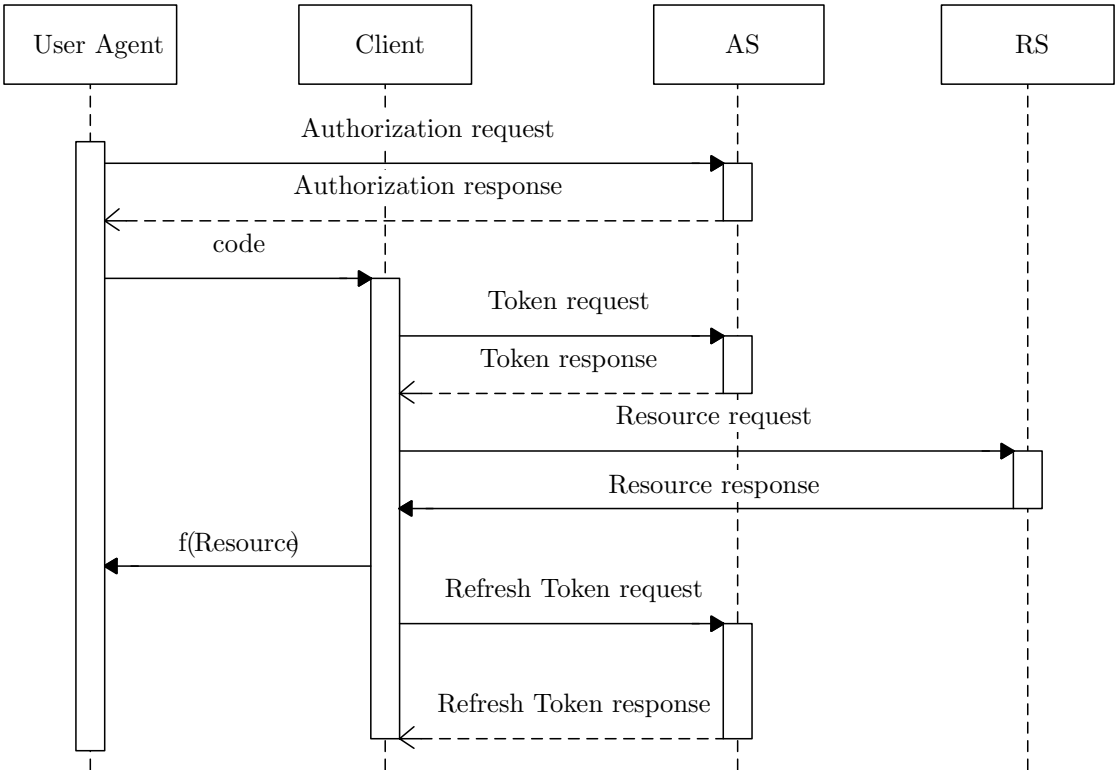
Da qui verranno utilizzati alcuni termini ed utili convenzioni per rappresentare le entità coinvolte nel protocollo.

UA	Il browser o applicativo del resource owner, tramite il quale vuole condividere l'accesso delle proprie risorse con un client.
C	Il client o applicazione di terze parti che vuole accedere alle risorse protette. Può essere un sito web, un'applicazione desktop o mobile.
RS	Il resource server che ospita le risorse protette
AS	L'authorization server, l'entità che si occupa di gestire le autorizzazzioni di accesso ai servizi.
Token	Il gettone utilizzato come alternativa alle credenziali utente. E' fornito dall'Authorization server.

3.1 Authorization Code

La modalità authorization code è utilizzata dai client capaci di mantenere la confidenzialità delle loro credenziali di accesso, o comunque capaci di autenticarsi in modo sicuro presso l'authorization server.

Questa modalità permette di autenticare il client e restituire il token, in modo privato (non in chiaro).



3.1.1 Authorization Request

Il client inizia il flusso, portando l'User Agent del Resource Owner all'Authorization endpoint. Il client indica il suo identificativo, l'eventuale scopo, stato ed url di redirezione verso il quale redirigere l'User Agent ad autorizzazione avvenuta. La richiesta DEVE avvenire tramite TLS.

response_type
: RICHIESTO. Deve avere valore 'code'.

client_id
: RICHIESTO. Identificativo del client ottenuto dopo la registrazione all'Authorization server. Non è segreto.

redirect_uri
: OPZIONALE. L'url verso cui redirigere l'User Agent al client.

scope
: OPZIONALE. Indica gli scopi della richiesta di accesso.

state

RACCOMANDATO. Stato locale per mantenere una traccia tra richiesta e risposta. Dovrebbe essere utilizzato per prevenire attacchi cross-site.

3.1.2 Authorization Response

Risposta sotto forma di redirect. La risposta viene trasmessa tramite TLS ma l'url di redirezione potrebbe non essere protetto tramite TLS, quindi le informazioni potrebbero viaggiare in chiaro.

code
: RICHIESTO. AuthCode generato dall'Authorization server. DEVE scadere dopo poco tempo. Un tempo di vita pari a 10 minuti massimi è raccomandato. Il client NON DEVE usare un AuthCode più di una volta. Se l'AuthCode è utilizzato più di una volta, l'Authorization server DEVE rifiutare la richiesta e dovrebbe revocare tutti i Token garantiti tramite quell'AuthCode.

state
: RICHIESTO se specificato precedentemente.

3.1.3 Access Token Request

Dopo aver concluso con successo l'Authorization Request/Response, il Client richiede il Token all'Authorization server. La richiesta DEVE essere fatta tramite TLS.

grant_type
: RICHIESTO. Deve avere valore 'authorization_code'.

code
: RICHIESTO. L'AuthCode ricevuto dall'Authorization Server

redirect_uri
: RICHIESTO se specificato nell'Authorization Request. DEVE essere identico.

3.1.4 Access Token Response

Risposta all'Access Token Request. La risposta avviene tramite TLS. L'esempio sottostante restituisce un Token di tipo MAC.

access_token
: RICHIESTO. Il Token restituito dall'Authorization Server

token_type
: RICHIESTO. Deve avere valore 'mac'.

expires_in
: OPZIONALE. Il tempo di vita del Token, espresso in secondi.

refresh_token
: OPZIONALE. Il RefreshToken che può essere utilizzato per ottenere un nuovo Token, quando quello precedente è scaduto.

scope
: OPZIONALE. Indica gli scopi della richiesta di accesso.

mac_key
: RICHIESTO. La chiave MAC.

mac_algorithm

RICHIESTO. Indica l'algoritmo usato per calcolare il MAC. DEVE essere uno dei seguenti valori: `hmac-sha-1`, `hmac-sha-256`, oppure un valore registrato nelle estensioni di OAuth2-MAC.

3.1.5 Resource Request

La risorsa viene richiesta tramite l'autorizzazione MAC in questo modo:

```
GET /resource/1?b=1&a=2 HTTP/1.1
     Host: example.com
     Authorization: MAC id="h480djs93hd8",
                        nonce="264095:dj83hs9s",
                        mac="SLDJd4mg43cjQfElUs3Qub4L6xE="
```

Dove l'id è il Token, il nonce è il nonce generato nel formato `età:stringa casuale` e l'attributo mac è creato tramite `algoritmomac(stringanormalizzata, chiave)`. La stringa normalizzata è una concatenazione del nonce, del tipo di richiesta (`GET`, `POST`, `HEAD` ecc), dell'url della risorsa richiesta, la porta, un payload ed un'eventuale estensione di autorizzazione:

```
264095:7d8f3e4a\n
POST\n
/request?b5=%3D%253D&a3=a&c%40=&a2=r%20b&c2&a3=2+q\n
example.com\n
80\n
Lve95gjOVATpfV8EL5X4nxwjKHE=\n
a,b,c\n
```

3.1.6 Refresh Token Request

E' possibile permettere al Client di aggiornare il Token scaduto, mediante il RefreshToken ottenuto tramite il passaggio precedentemente descritto. La richiesta DEVE avvenire tramite TLS.

grant_type
: RICHIESTO. Deve avere valore 'refresh_token'.

refresh_token
: OPZIONALE. Il RefreshToken.

scope
: OPZIONALE. Indica gli scopi della richiesta di accesso. NON DEVE contenere scopi non originariamente ottenuti dal Resource Owner.

3.1.7 Refresh Token Response

Risposta al Refresh Token Request. La risposta avviene tramite TLS e la sua struttura è identica a quella dell'Access Token Response. Il RefreshToken può essere uguale a quello di partenza o può essere cambiato di volta in volta.

3.1.8 Interazioni

All'interno della nostra simulazione, abbiamo ipotizzato di utilizzare la sicurezza minima, quindi abbiamo evitato di utilizzare controlli accessori come lo state e lo scope nelle varie richieste.

Questi parametri rendono le richieste del protocollo più mirate e restringono le possibilità ed i permessi dei Client.

UA	→	C	: L'User Agent richiede una pagina web del Client	(1)
C	→	UA	: Il Client restituisce la pagina che richiede l'autorizzazzione	(2)
			: Il resource owner da' l'autorizzazzione al client	(3)
C	→	AS	: *ResponseType, ClientID, ClientRedirectUri* via TLS	(4)
			: AS controlla i permessi di ClientID	(5)
			: AS genera un AuthCode e lo memorizza	(6)
AS	→	UA	: *AuthCode* via TLS	(7)
UA	→	C	: *AuthCode* via Redirect	(8)
C	→	UA	: *GrantType, ClientID, ClientPassword, AuthCode, ClientRedirectUri* via TLS	(9)
			: UA richiede l'autenticazione se CliendID è confidenziale	(10)
			: UA controlla che ClientID abbia le autorizzazioni	(11)
			: UA controlla che l'AuthCode sia valido ed associato a ClientID	(12)
			: UA controlla che il ClientRedirectUri sia stato usato per ottenere l'AuthCode	(13)
			: UA genera un TokenCode e lo memorizza	(14)
			: UA genera un RefreshToken e lo memorizza	(15)
UA	→	C	: *TokenCode, TokenType, MacKey, RefreshToken* via TLS	(16)
C	→	RS	: *ResourceUri, TokenCode*	(17)
			: RS chiede ad AS se il TokenCode è valido	(18)
RS	→	C	: *Resource*	(19)
C	→	UA	: Il client restituisce la pagina richiesta, utilizzando *Resource*	(20)

3.2 Sicurezza

I requisiti di sicurezza sono descritti in modo impeccabile all'interno della sezione *Security Considerations* delle specifiche [2], all'interno dell'attenta analisi analisi *OAuth 2.0 Threatmodel* [3] e della sezione *Security Considerations* nella specifica dell'autorizzazione MAC [4].

3.3 Modello in Proverif

Come precedentemente illustrato, OAuth 2.0 è composto essenzialmente da quattro entità: Resource Owner, Client, Authorization Server, Resource Server e comunica le informazioni utilizzando canali pubblici (es. Internet) e privati (via TLS).

Nella formalizzazione abbiamo creato quattro processi concorrenti: User Agent (che modella insieme l'User Agent ed il Resource Owner), Client, Resource Server ed Authorization Server (a sua volta suddiviso in ottenimento dell'AuthCode, del Token e del RefreshToken). Abbiamo poi creato due canali: il canale net per rappresentare un canale pubblico e vari canali privati TLS. Per rappresentare efficacemente questi canali privati e la loro relativa instaurazione durante l'esecuzione di un processo, ci siamo serviti di un canale privato TLS_pass e di alcune costanti: (c1, c2, c3, c4). Se TLS_pass ha la funzione di simulare la complessa instaurazione di una connessione sicura TLS end-to-end (non è nostro compito verificare il protocollo in questa sede), le costanti hanno invece il compito di rendere più agile l'analisi dei canali da parte di Proverif, ordinando lo scambio delle informazioni.

Listing 1: Instaurazione connessione TLS del Client

```
1  new TLSchannel2: channel;
2  out(TLS_pass, TLSchannel2);
3  out(TLSchannel2, (grant_type, X_client_id, X_client_password, code,
       redirect_uri, c2));
```

Listing 2: Instaurazione connessione TLS dell'AS

```
4  in(TLS_pass, TLSchannel2:channel);
5  in(TLSchannel2, (=grant_type, client_id:bitstring, client_password:
       bitstring, auth_code:bitstring, client_redirect_uri:bitstring,
       =c2));
```

Inizialmente il Client crea una nuova connessione privata TLSchannel2 e la passa alla controparte utilizzando il canale TLS_pass. Questo instaura la connessione tra le due parti e simula lo scambio delle informazioni e chiavi, del protocollo TLS.
Nella linea 3 e 5 abbiamo l'effettivo scambio di messaggi tramite la connessione TLS instaurata, con il controllo della costante.

Per simulare i database dell'Authorization server, ci siamo appoggiati a quattro tabelle Proverif che consentono di inserire ed accedere ai dati tramite apposite funzioni chiamate insert e get. Abbiamo quindi creato la tabella RegisteredClients con lo scopo di simulare i clients registrati che hanno il permesso di autenticazione nel server, AuthCodes che memorizza l'AuthCode ed il redirect_uri assegnato ad ogni Client, RefreshToken che verifica l'appartenenza di un dato RefreshToken ad un Client ed infine mac_keys, utilizzato dal Resource Server e che simula la richiesta che quest'ultimo deve effettuare per verificare il MAC.

Listing 3: Databases rappresentati

```
1  table RegisteredClients(bitstring, bitstring).
2  table AuthCodes(bitstring, bitstring, bitstring).
3  table RefreshTokens(bitstring, RefreshToken).
4  table mac_keys(Token, bitstring).
```

3.3.1 Sicurezza

Per poter controllare tutti i singoli avvenimenti, impostare i requisiti di sicurezza e trarne le dovute considerazioni, abbiamo creato diversi eventi che rappresentano gli avvenimenti del protocollo. I nomi significativi impostati, fanno si che gli eventi non richiedano ulteriori spiegazioni.

Listing 4: Eventi

```
1  event auth_request(bitstring).
2  event auth_accepted(bitstring, bitstring).
3  event token_request(bitstring, bitstring).
4  event token_grant(bitstring, bitstring, Token, RefreshToken).
5  event resource_request(Token, mac).
6  event resource_accepted(Token, mac).
7  event token_refresh(bitstring, RefreshToken).
```

```
8 event token_refreshed(bitstring, RefreshToken).
```

Iniziamo ad analizzare ora le proprietà da noi desiderate.

Listing 5: AuthCode

```
1 query client_id:bitstring, authcode:bitstring; inj-event(
      auth_accepted(client_id, authcode)) ==> inj-event(auth_request(
      client_id)).
```

Per ogni AuthCode inviato, ce ne deve essere uno ed uno solo richiesto per tale ClientID.

Listing 6: Token

```
1 query client_id:bitstring, authcode:bitstring, tokencode:Token,
      refreshcode:RefreshToken; event(token_grant(client_id,authcode,
      tokencode,refreshcode)) ==> (event(token_request(client_id,
      authcode)) || event(token_refreshed(client_id, refreshcode))).
2 query client_id:bitstring, refreshcode:RefreshToken; event(
      token_refreshed(client_id,refreshcode)) ==> event(token_refresh
      (client_id,refreshcode)).
3 query client_id:bitstring, authcode:bitstring, tokencode:Token,
      refreshcode:RefreshToken; event(token_grant(client_id,authcode,
      tokencode,refreshcode)) ==> event(auth_accepted(client_id,
      authcode)).
4 query client_id:bitstring, tokencode:Token, refreshcode:
      RefreshToken, authcode:bitstring; event(token_refreshed(
      client_id,refreshcode)) ==> event(token_grant(client_id,
      authcode,tokencode,refreshcode)).
```

Per ogni token inviato, deve esserci stata una richiesta per aver e un nuovo token oppure una richiesta per aggiornarne uno esistente e scaduto, con stessi AuthCode/refreshcode e per un dato ClientID.
Per ogni token aggiornato, ci deve essere stata una richiesta di aggiornamento con lo stesso refreshToken ed un determinato ClientID.
Alla linea 3, si richiede che per ogni token inviato, ci sia stato anche un evento di AuthCode accettato per quel determinato client. Si richiede inoltre che il refreshToken sia correttamente stato inviato al ClientID che richiede l'aggiornamento del token.

Listing 7: Resource

```
1 query tokencode:Token, z:mac; event(resource_accepted(tokencode,z))
      ==> event(resource_request(tokencode,z)).
2 query client_id:bitstring, authcode:bitstring, tokencode:Token,
      refreshcode:RefreshToken, z:mac; event(resource_accepted(
      tokencode,z)) ==> (event(token_grant(client_id,authcode,
      tokencode,refreshcode)) ==> event(auth_accepted(client_id,
      authcode))).
```

Per ogni risorsa accettata ed inviata, ci deve essere stata una richiesta di risorsa con lo stesso mac e Token. Si richiede inoltre alla linea 2, che per ogni risorsa accettata con un determinato token, quel token sia stato inviato e sia stata anche inviata la sua relativa autorizzazione.

Listing 8: Attackers

```
1  query    attacker(A_client_password);
2           (* attacker(secretTokenC); *)
3           attacker(secretMACKeyC);
4           attacker(secretTokenRefreshC);
5           attacker(secretMACKey2C);
6           (* attacker(secretTokenRefreshedC); *)
7           (* attacker(secretTokenAS); *)
8           attacker(secretTokenRefreshAS1);
9           (* attacker(secretTokenRefreshedAS); *)
10          attacker(secretTokenRefreshAS2);
11          attacker(secretMACKeyAS);
12          attacker(secretMACKey2AS);
13          (* attacker(secretTokenRS); *)
14          attacker(secretMACKeyRS).
15
16 not      attacker(TLS_pass).
```

Con gli attackers arriviamo infine alle richieste relative alla segretezza. All'interno dei processi del nostro modello Proverif, abbiamo molto spesso creato Token, AuthCode e chiavi. Queste variabili sono state create tramite il costruttto new. Proverif riesce a verificare la segretezza solamente di variabili libere create all'inizio del modello. Per aggirare questa limitazione, per ogni variabile creata con il costrutto new, ne abbiamo creata una globale e libera. Invece che testare direttamente la segretezza delle variabili non libere, le abbiamo codificate: se le variabili libere codificate saranno segrete, allora lo saranno anche le nostre variabili.
Per avere una maggiore precisione sui punti di rottura del protocollo, abbiamo impostato più variabili libere, posizionate nei diversi processi coinvolti. Così per verificare la segretezza del RefreshToken, abbiamo creato la variabile secretTokenRefreshC (utilizzata nel processo Client) e la variabile libera secrettokenRefreshAS1 (utilizzata all'interno del processo AS).

Per prima cosa richiediamo naturalmente che la password dei Client, non venga scoperta da terze parti. Successivamente, tramite le variabili di appoggio precedentemente descritte, richiediamo la segretezza della chiave MAC e del RefreshToken in tutti i loro punti di scopertura. Abbiamo impostato anche il controllo della segretezza del Token ma nello studio da noi impostato non ha senso: in OAuth 2.0 draft 22 con autenticazione MAC, il Token viene spedito in chiaro. Nell'autenticazione Bearer Token, dovremmo impostare il controllo di segretezza.

Con la linea 16 impostiamo che il canale TLS_pass non possa essere attaccato: dato che è una nostra formalizzazzione per stabilire una connessione TLS, comunichiamo a Proverif che quel canale di appoggio lo si considera sicuro.

3.3.2 Authorization Code

Il primo passo da analizzare è la richiesta dell'Authorization Code. Per simulare efficacemente questo scambio, abbiamo scelto che l'UserAgent richieda una pagina web al client e quest'ultimo restituisca le indicazioni necessarie per individuare l'Authorization server al quale effettuare la richiesta e redirigere successivamente l'UserAgent al client che necessita le autorizzazzioni.

Listing 9: UserAgent

```
1  event auth_request(client_id);
2  out(TLSchannel1, (response_type, client_id, client_redirect_uri, c1
       ));
3
4  (* authorization response *)
5  in(net, code:bitstring);
6
7  (* I pass the authorization code to the client *)
8  out(net, code);
```

Listing 10: Authorization Server

```
4  in(TLSchannel1, (=response_type, client_id:bitstring,
       client_redirect_uri:bitstring, =c1));
5  (* check authorized clients *)
6  get RegisteredClients(=client_id, client_password) in
7
8  (* authorization response *)
9  new auth_code:bitstring;
10 event auth_accepted(client_id, auth_code);
11 out(net, auth_code);
12 insert AuthCodes(auth_code, client_id, client_redirect_uri);
```

Dopo aver instaurato una nuova connessione TLS, l'UserAgent invoca l'evento auth_request ed invia la richiesta all'AS. Quest'ultimo controlla che response_type sia esatto (abbiamo utilizzato una costante per simularlo), che l'ordinamento sia corretto (c1) ed infine che il client abbia le autorizzazzioni per richiedere un AuthCode. Per fare ciò utilizziamo la tabella RegisteredClients, appositamente creata e valorizzata all'inizio del processo Proverif.

Se i controlli hanno dato esito positivo, invochiamo l'evento auth_accepted. A questo punto l'AS genera un nuovo auth_code, lo inserisce nel suo database interno assieme ad informazioni come il client_id ed il redirect_uri e restituisce il codice tramite un redirect all'url specificato da redirect_uri. Abbiamo simulato questo redirect tramite uno scambio a due passi effettuati prima tra AS ed UserAgent, poi tra UserAgent e client.
Il redirect fa si che UserAgent, Client ed AS conoscano l'AuthCode.

3.3.3 Access Token

Dopo aver ottenuto l'AuthCode, è possibile procedere con la richiesta del token da parte del Client.

Listing 11: Client

```
1  event token_request(A_client_id, code);
2  out(TLSchannel2, (grant_type, X_client_id, X_client_password, code,
        redirect_uri, c2));
3
4  (* token response *)
```

```
5  in(TLSchannel2, (token_code:Token, =token_type, mac_key:bitstring,
       =mac_algorithm, refresh_token:RefreshToken, =c3));
```

Listing 12: Authorization Server

```
1  in(TLSchannel2, (=grant_type, client_id:bitstring, client_password:
       bitstring, auth_code:bitstring, client_redirect_uri:bitstring,
       =c2));
2  (* check authorized clients *)
3  get RegisteredClients(=client_id, =client_password) in
4  get AuthCodes(=auth_code, =client_id, =client_redirect_uri) in
5
6  (* token response *)
7          new token_code:Token;
8          new refreshToken_code:RefreshToken;
9          new mac_key: bitstring;
10
11         event token_grant(client_id, auth_code, token_code,
               refreshToken_code);
12         out(TLSchannel2, (token_code, token_type, mac_key,
               mac_algorithm, refreshToken_code, c3));
13
14         (* mac_key exchange between authserver and resourceserver
               *)
15         insert mac_keys(token_code, mac_key);
16         insert RefreshTokens(client_id, refreshToken_code);
17
18 (* test secrecy *)
19         (* out(net, senc(secretTokenAS, Token_to_bitstring(
               token_code))); *)
20         out(net, senc(secretTokenRefreshAS1,
               RefreshToken_to_bitstring(refreshToken_code)));
21         out(net, senc(secretMACKeyAS, mac_key));
```

Il Client instaura inizialmente una connessione TLS con l'AS, invoca l'evento token_request e quindi invia la richiesta. L'AS dal canto suo controlla l'ordinamento tramite la costate c2, controlla che client_id e client_password corrispondano ad una voce del database interno dei Client registrati ed infine verifica che l'AuthCode esista e sia stato assegnato al Client che effettua la richiesta. Controlla inoltre che il redirect_uri specificato sia quello utilizzato in precedenza per richieder el'AuthCode.
Se tutti i controlli sono andati a buon fine, crea un nuovo token, refreshToken, chiave MAC, invoca l'evento token_grant e risponde alla richiesta.
Alle linee 15-16 inserisce le chiavi ed il refreshToken all'interno del suo database interno.

Le ultime righe sono dedicate ai controlli di segretezza relativi al Token (come spiegato in sezione Sicurezza). Le linee commentate sono collegate ai controlli di sicurezza generali di OAuth, non applicabili nel caso di autenticazione MAC.

3.3.4 Resource request

Listing 13: Client

```
1  new N:nonce;
2  (* I build the normalized string for HMAC generation *)
3  let(normalized_string:bitstring) = (N, resource_uri) in
4
5  let mac_string = hmac_sha_256(normalized_string, mac_key) in
6  event resource_request(token_code, mac_string);
7  out(net, (resource_uri, token_code, N, mac_string));
8
9  (* resource response *)
10 in(net, resource:bitstring);
11
12 (* send resource to the useragent *)
13 out(net, F(resource));
```

Listing 14: Resource Server

```
1  in(net, (resource_url:bitstring, token_code:Token, N:nonce,
       mac_string:mac));
2  (* I build the normalized string for HMAC generation *)
3  let(normalized_string:bitstring) = (N, resource_url) in
4
5  (* mac_key exchange between authserver and resourceserver *)
6  get mac_keys(=token_code, mac_key) in
7
8  (* check of mac code *)
9  let (=mac_string) = hmac_sha_256(normalized_string, mac_key) in
10 event resource_accepted(token_code, mac_string);
11 out(net, resourcecontent(resource_url));
12
13 (* test secrecy *)
14 out(net, senc(secretTokenRS, Token_to_bitstring(token_code)));
15 out(net, senc(secretMACKeyRS, mac_key));
```

Il Client deve ora richiedere la risorsa desiderata al Resource Server. Per farlo tramite l'autenticazione MAC genera un nonce e crea la stringa normalizzata, composta da nonce e l'url della risorsa, facendone l'hash tramite l'algoritmo HMAC e la chiave mac_key generata precedentemente. Richiama poi l'evento resource_request ed invia la richiesta.

Il Resource Server prende la richiesta, controlla la stringa normalizzata confrontado l'HMAC ricevuto con l'HMAC generato grazie ai parametri inviati ed alla chiave ottenuta tramite l'Authorization server. Non è scopo di questo protocollo, stabilire come questo ottenimento avvenga per cui nel nostro modello abbiamo utilizzato la tabella mac_keys per ottenere tale chiave.
Se tutti i controlli sono andati a buon fine, invochiamo l'evento resource_accepted ed inviamo la risorsa. Testiamo infine la segretezza della chiave e del RefreshToken.

Nelle ultime righe del Client riceviamo la risorsa e la inviamo all'UserAgent tramite un'ipotetica funzione F che simula un utilizzo della risorsa appena ricevuta.

3.3.5 Refresh Token

Listing 15: Client

```
1  event token_refresh(X_client_id, refresh_token);
2  out(TLSchannel3, (grant_type, X_client_id, X_client_password,
       refresh_token, c4));
3
4  (* refreshToken response *)
5  in(TLSchannel3, (token_code2:Token, =token_type, mac_key2:bitstring
       , =mac_algorithm, refresh_token2:RefreshToken, =c5));
```

Listing 16: Authorization Server

```
1  in(TLSchannel3, (=grant_type, client_id:bitstring, client_password:
       bitstring, refreshToken_code_old:RefreshToken, =c4));
2  (* check authorized clients *)
3  get RegisteredClients(=client_id, =client_password) in
4  get RefreshTokens(=client_id, =refreshToken_code_old) in
5
6  (* refreshToken response *)
7  new token_code:Token;
8  new mac_key: bitstring;
9
10 event token_refreshed(client_id, refreshToken_code_old);
11 (* I fetch auth_code only for event purpouse *)
12 get AuthCodes(auth_code, =client_id, client_redirect_uri) in
13 event token_grant(client_id, auth_code, token_code,
       refreshToken_code_old);
14 out(TLSchannel3, (token_code, token_type, mac_key, mac_algorithm,
       refreshToken_code_old, c5));
15
16 (* mac_key exchange between authserver and resourceserver *)
17 insert mac_keys(token_code, mac_key);
18
19 (* test secrecy *)
20 (* out(net, senc(secretTokenRefreshedAS, Token_to_bitstring(
       token_code))); *)
21 out(net, senc(secretTokenRefreshAS2, RefreshToken_to_bitstring(
       refreshToken_code_old)));
22 out(net, senc(secretMACKey2AS, mac_key));
```

L'operazione di aggiornamento del token è del tutto opzionale ad ogni modo abbiamo voluto verificarla. Simuleremo di non inviare un nuovo RefreshToken insieme al Token aggiornato. Per prima cosa il Client invoca l'evento token_refresh, poi effettua la richiesta. L'Authorization Server controlla l'ordinamento (c4), se le credenziali sono valide e se il RefreshToken è corretto. Se tutti i controlli sono andati a buon fine invia il nuovo Token, creato nel modo precedentemente descritto e lo invia al Client.

Una nota: alla linea numero 12 abbiamo richiesto l'AuthCode solo per motivi di simulazione proverif: l'evento token_grant è generico e vale sia per il Token Request che per il Refresh Token.

Infine controlliamo la segretezza del RefreshToken e della chiave MAC.

3.4 Analisi Proverif

Una volta creato il modello, abbiamo lanciato il tool con il comando ./proverif oauth2-authorization-code.pv — grep RES ed abbiamo ricevuto il seguente output:

```
1  RESULT not attacker(A_client_password[]) is true.
2  RESULT not attacker(secretMACKeyC[]) is true.
3  RESULT not attacker(secretTokenRefreshC[]) is true.
4  RESULT not attacker(secretMACKey2C[]) is true.
5  RESULT not attacker(secretTokenRefreshAS1[]) is true.
6  RESULT not attacker(secretTokenRefreshAS2[]) is true.
7  RESULT not attacker(secretMACKeyAS[]) is true.
8  RESULT not attacker(secretMACKey2AS[]) is true.
9  RESULT not attacker(secretMACKeyRS[]) is true.
10 RESULT event(token_refreshed(client_id_2766,refreshcode)) ==> event
       (token_grant(client_id_2766,authcode,tokencode,refreshcode)) is
        true.
11 RESULT event(resource_accepted(tokencode_5715,z)) ==> (event(
       token_grant(client_id_5713,authcode_5714,tokencode_5715,
       refreshcode_5716)) ==> event(auth_accepted(client_id_5713,
       authcode_5714))) is true.
12 RESULT event(token_grant(client_id_8949,authcode_8950,
       tokencode_8951,refreshcode_8952)) ==> event(auth_accepted(
       client_id_8949,authcode_8950)) is true.
13 RESULT event(token_refreshed(client_id_11982,refreshcode_11983))
       ==> event(token_refresh(client_id_11982,refreshcode_11983)) is
       true.
14 RESULT event(resource_accepted(tokencode_14696,z_14697)) ==> event(
       resource_request(tokencode_14696,z_14697)) is true.
15 RESULT event(token_grant(client_id_17432,authcode_17433,
       tokencode_17434,refreshcode_17435)) ==> event(token_request(
       client_id_17432,authcode_17433)) || event(token_refreshed(
       client_id_17432,refreshcode_17435)) is true.
16 RESULT inj-event(auth_accepted(client_id_20346,authcode_20347)) ==>
        inj-event(auth_request(client_id_20346)) is true.
```

Questa serie di risultati ci indicano la corrispondenza degli eventi e la segretezza delle variabili da noi specificate. Tutti gli obiettivi da noi indicati sono stati raggiunti, l'analisi formale ci indica che il modello creato è sicuro.

4 Conclusioni

Anche se normalmente l'analisi formale manuale dei protocolli risulta difficoltosa e non esente da errori, l'utilizzo di Proverif ha permesso di verificare il protocollo in modo semplice ed efficace.

L'utilizzo corretto di OAuth2 secondo le linee guida espresse nella sezione *Security Considerations* della specifica di protocollo [2], permette al protocollo di essere considerato sicuro.

Eventuali problematiche al protocollo stesso possono emergere nel caso in cui non vengano protette adeguatamente le operazioni esterne legate al protocollo, quali la protezione delle credenziali del client, le lunghe tempistiche di scadenza dei Token oppure gli scambi per veficarli correttamente.

Il protocollo si basa enormemente sulla fiducia riposta nell'applicazione client e questo potrebbe essere un problema nel caso in cui non sia adeguatamente protetta: si pensi ad esempio ad un'applicazione desktop che funge da client, dove le credenziali sono memorizzate all'interno del software. Un attaccante potrebbe utilizzare il reverse engineering per estrapolarle e quindi compromettere il client. Una volta scoperta la compromissione, l'Authorization server non potrebbe far altro che revocare le credenziali del client, danneggiando il client stesso.
Se l'applicazione fosse ad esempio per iPhone, un concorrente potrebbe mettere fuori uso l'applicativo di un altro concorrente e far si di eliminare il concorrente per diverso tempo prima che rientri nel mercato (può passare diverso tempo prima della scoperta, correzione credenziali e successiva approvazione AppStore). E' chiaro che la sezione Native Applications del protocollo, dovrebbe essere resa più restrittiva onde evitare problemi di ogni sorta, ma possiamo affermare che non ci siano insicurezze dimostrate all'interno del protocollo OAuth 2.0.

Riferimenti bibliografici

[1] Proverif. (2011) http://www.proverif.ens.fr/.

[2] E. E. Hammer-Lahav, *The oauth 2.0 authorization protocol draft-ietf-oauth-v2-22.* (2011) http://tools.ietf.org/html/draft-ietf-oauth-v2-22.

[3] *OAuth 2.0 Threat Model and Security Considerations.* (2011) http://tools.ietf.org/html/draft-ietf-oauth-v2-threatmodel-01.

[4] *HTTP Authentication: MAC Access Authentication.* (2011) http://tools.ietf.org/html/draft-ietf-oauth-v2-http-mac-00.

[5] *The OAuth 2.0 Authorization Protocol: Bearer Tokens.* (2011) http://tools.ietf.org/html/draft-ietf-oauth-v2-bearer-14.

A Formalizzazione del protocollo

```
(* ************************************************************
   *** OAuth 2.0 draft 22 verification                       ***
   ***                                                      ***
   *** Marco De Nadai (86873)                               ***
   ************************************************************ *)


(* ************************************************************
   * DEFINITIONS                                              *
   ************************************************************ *)
(* public channel *)
  free net:channel.
  (* it helps to simulate the instauration of a TLS channel *)
  free TLS_pass:channel [private].

(* Message numbers *)
  const c1, c2, c3, c4, c5:bitstring.

(* Messages *)
  const client_page_url:bitstring.
  const response_type:bitstring.
  const redirect_uri:bitstring.
  const resource_uri:bitstring.
  const grant_type:bitstring.
  const token_type:bitstring.
  const mac_algorithm:bitstring.

  type mac.
  type nonce.
  type Token.
  type RefreshToken.

  (* EXAMPLE: a registered client. Single for test *)
  free A_client_id:bitstring.
  free A_client_password:bitstring [private].
  free B_client_id:bitstring.
  free B_client_password:bitstring [private].

(* tables *)
  (* token-mac_keys list: simulate the communication between RS and
     AS *)
  table mac_keys(Token, bitstring).
  (* Auth codes issued by AS: simulate the AS database *)
  table AuthCodes(bitstring, bitstring, bitstring).
  (* Refresh Token issued: simulate the AS database *)
  table RefreshTokens(bitstring, RefreshToken).
  (* registered clients in AS *)
  table RegisteredClients(bitstring, bitstring).

(* type converter *)
  fun Token_to_bitstring(Token):bitstring[data,typeConverter].
  fun RefreshToken_to_bitstring(RefreshToken):bitstring[data,
      typeConverter].
```

```
(* the cryptographic constructors *)
  (* enc for secrecy test *)
  fun senc(bitstring, bitstring):bitstring.
  reduc forall x:bitstring, y:bitstring; sdec(senc(x,y),y) = x.
  (* function for resource *)
  fun resourcecontent(bitstring):bitstring.
  (* mac for resource request *)
  fun hmac_sha_256(bitstring, bitstring): mac.
  (* encode channel *)
  fun enc(channel, bitstring): bitstring.
  reduc forall m:channel, k:bitstring; dec(enc(m,k),k) = m.

(* Resource function *)
  fun F(bitstring):bitstring.

(* events *)
  event auth_request(bitstring).
  event auth_accepted(bitstring, bitstring).
  event token_request(bitstring, bitstring).
  event token_grant(bitstring, bitstring, Token, RefreshToken).
  event resource_request(Token, mac).
  event resource_accepted(Token, mac).
  event token_refresh(bitstring, RefreshToken).
  event token_refreshed(bitstring, RefreshToken).


(* ************************************************************
   * QUERIES                                                  *
   ************************************************************ *)
  (* for each authcode issued, one authcode must be requested *)
  query client_id:bitstring, authcode:bitstring; inj-event(
      auth_accepted(client_id, authcode)) ==> inj-event(auth_request(
      client_id)).
  (* for each token granted, one token must be requested or refreshed
      *)
  query client_id:bitstring, authcode:bitstring, tokencode:Token,
      refreshcode:RefreshToken; event(token_grant(client_id,authcode,
      tokencode,refreshcode)) ==> (event(token_request(client_id,
      authcode)) || event(token_refreshed(client_id, refreshcode))).
  (* for each resource accepted, one resource must be requested *)
  query tokencode:Token, z:mac; event(resource_accepted(tokencode,z))
       ==> event(resource_request(tokencode,z)).
  (* for each token refreshed, a token refresh must be issued *)
  query client_id:bitstring, refreshcode:RefreshToken; event(
      token_refreshed(client_id,refreshcode)) ==> event(token_refresh(
      client_id,refreshcode)).

  (* for each token granted, an associated authcode must be issued *)
  query client_id:bitstring, authcode:bitstring, tokencode:Token,
      refreshcode:RefreshToken; event(token_grant(client_id,authcode,
      tokencode,refreshcode)) ==> event(auth_accepted(client_id,
      authcode)).
  (* for each resource accepted, one token must be granted and an
```

```
    associated authcode must be issued *)
  query client_id:bitstring, authcode:bitstring, tokencode:Token,
      refreshcode:RefreshToken, z:mac; event(resource_accepted(
      tokencode,z)) ==> (event(token_grant(client_id,authcode,
      tokencode,refreshcode)) ==> event(auth_accepted(client_id,
      authcode))).
  (* for each token refreshed, the token must be granted *)
  query client_id:bitstring, tokencode:Token, refreshcode:
      RefreshToken, authcode:bitstring; event(token_refreshed(
      client_id,refreshcode)) ==> event(token_grant(client_id,authcode
      ,tokencode,refreshcode)).


(* Secrecy queries *)
  free secretMACKeyC, secretMACKey2C, secretTokenC,
      secretTokenRefreshC, secretTokenRefreshedC, secretTokenAS,
      secretTokenRefreshedAS, secretTokenRefreshAS1,
      secretTokenRefreshAS2, secretMACKeyAS, secretMACKey2AS,
      secretTokenRS, secretMACKeyRS: bitstring[private].

  query attacker(A_client_password);
      (* attacker(secretTokenC); *)
      attacker(secretMACKeyC);
      attacker(secretTokenRefreshC);
      attacker(secretMACKey2C);
      (* attacker(secretTokenRefreshedC); *)
      (* attacker(secretTokenAS); *)
      attacker(secretTokenRefreshAS1);
      (* attacker(secretTokenRefreshedAS); *)
      attacker(secretTokenRefreshAS2);
      attacker(secretMACKeyAS);
      attacker(secretMACKey2AS);
      (* attacker(secretTokenRS); *)
      attacker(secretMACKeyRS).
  (* we suppose that we can pass the TLS channel without being
      attacked *)
  not    attacker(TLS_pass).

(* ************************************************************
   * USER AGENT: browser of the resource owner                *
   ************************************************************ *)
let UserAgent() =

  (* request client's page *)
    out(net, client_page_url);

  (* get client's page *)
    in(net, (client_id:bitstring, client_redirect_uri:bitstring));

  (* TLS: establish connection *)
    in(TLS_pass, TLSchannel1:channel);

  (* authorization request *)
    event auth_request(client_id);
```

```
    out(TLSchannel1, (response_type, client_id, client_redirect_uri,
        c1));

  (* authorization response *)
    in(net, code:bitstring);

  (* I pass the authorization code to the client *)
    out(net, code);

  (* I get the resource *)
    in(net, resource:bitstring);


  0.


(* ***********************************************************
   * CLIENT: for example a third part application            *
   *********************************************************** *)
let client(X_client_id:bitstring, X_client_password:bitstring) =

  (* get client's page request *)
    in(net, =client_page_url);

  (* authorization endpoint *)
    (* pkS is the AS server *)
    out(net, (X_client_id, redirect_uri));

  (* I get the authorization code from the UserAgent *)
    in(net, code:bitstring);

  (* token request *)
    (* TLS: establish connection *)
    new TLSchannel2: channel;
    out(TLS_pass, TLSchannel2);
    event token_request(A_client_id, code);
    out(TLSchannel2, (grant_type, X_client_id, X_client_password,
        code, redirect_uri, c2));

  (* token response *)
    in(TLSchannel2, (token_code:Token, =token_type, mac_key:bitstring
        , =mac_algorithm, refresh_token:RefreshToken, =c3));

  (* resource request *)
    new N:nonce;
    (* I build the normalized string for HMAC generation *)
    let(normalized_string:bitstring) = (N, resource_uri) in

    let mac_string = hmac_sha_256(normalized_string, mac_key) in
    event resource_request(token_code, mac_string);
    out(net, (resource_uri, token_code, N, mac_string));

  (* resource response *)
    in(net, resource:bitstring);

  (* send resource to the useragent *)
```

```
    out(net, F(resource));

  (* refreshToken request *)
    (* TLS: establish connection *)
    new TLSchannel3: channel;
    out(TLS_pass, TLSchannel3);
    event token_refresh(X_client_id, refresh_token);
    out(TLSchannel3, (grant_type, X_client_id, X_client_password,
        refresh_token, c4));

  (* refreshToken response *)
    in(TLSchannel3, (token_code2:Token, =token_type, mac_key2:
        bitstring, =mac_algorithm, refresh_token2:RefreshToken, =c5));

  (* test secrecy *)
    (* out(net, senc(secretTokenC, Token_to_bitstring(token_code)));
        *)
    out(net, senc(secretTokenRefreshC, RefreshToken_to_bitstring(
        refresh_token)));
    (* out(net, senc(secretTokenRefreshedC, Token_to_bitstring(
        token_code2))); *)
    out(net, senc(secretMACKeyC, mac_key));
    out(net, senc(secretMACKey2C, mac_key2));


  0.


(* ************************************************************
   * AUTHORIZATION SERVER: grants authorization and token codes *
   ************************************************************ *)
let authserver_AuthCode() =

  (* TLS: establish connection *)
    new TLSchannel1: channel;
    out(TLS_pass, TLSchannel1);

  (* authorization request *)
    in(TLSchannel1, (=response_type, client_id:bitstring,
        client_redirect_uri:bitstring, =c1));
    (* check authorized clients *)
    get RegisteredClients(=client_id, client_password) in

  (* authorization response *)
    new auth_code:bitstring;
    event auth_accepted(client_id, auth_code);
    out(net, auth_code);
    insert AuthCodes(auth_code, client_id, client_redirect_uri);

  0.


let authserver_TokenCode() =

  (* token request *)
    (* TLS: establish connection *)
```

```
    in(TLS_pass, TLSchannel2:channel);

    in(TLSchannel2, (=grant_type, client_id:bitstring,
        client_password:bitstring, auth_code:bitstring,
        client_redirect_uri:bitstring, =c2));
    (* check authorized clients *)
    get RegisteredClients(=client_id, =client_password) in
    get AuthCodes(=auth_code, =client_id, =client_redirect_uri) in

  (* token response *)
    new token_code:Token;
    new refreshToken_code:RefreshToken;
    new mac_key: bitstring;

    event token_grant(client_id, auth_code, token_code,
        refreshToken_code);
    out(TLSchannel2, (token_code, token_type, mac_key, mac_algorithm,
         refreshToken_code, c3));

    (* mac_key exchange between authserver and resourceserver *)
    insert mac_keys(token_code, mac_key);
    insert RefreshTokens(client_id, refreshToken_code);

  (* test secrecy *)
    (* out(net, senc(secretTokenAS, Token_to_bitstring(token_code)));
         *)
    out(net, senc(secretTokenRefreshAS1, RefreshToken_to_bitstring(
        refreshToken_code)));
    out(net, senc(secretMACKeyAS, mac_key));


  0.


let authserver_TokenCodeRefresh() =

  (* refreshToken request *)
    (* TLS: establish connection *)
    in(TLS_pass, TLSchannel3:channel);

    in(TLSchannel3, (=grant_type, client_id:bitstring,
        client_password:bitstring, refreshToken_code_old:RefreshToken,
         =c4));
    (* check authorized clients *)
    get RegisteredClients(=client_id, =client_password) in
    get RefreshTokens(=client_id, =refreshToken_code_old) in

  (* refreshToken response *)
    new token_code:Token;
    new mac_key: bitstring;

    event token_refreshed(client_id, refreshToken_code_old);
    (* I fetch auth_code only for event purpouse *)
    get AuthCodes(auth_code, =client_id, client_redirect_uri) in
    event token_grant(client_id, auth_code, token_code,
        refreshToken_code_old);
```

```
    out(TLSchannel3, (token_code, token_type, mac_key, mac_algorithm,
        refreshToken_code_old, c5));

    (* mac_key exchange between authserver and resourceserver *)
    insert mac_keys(token_code, mac_key);

  (* test secrecy *)
    (* out(net, senc(secretTokenRefreshedAS, Token_to_bitstring(
       token_code))); *)
    out(net, senc(secretTokenRefreshAS2, RefreshToken_to_bitstring(
       refreshToken_code_old)));
    out(net, senc(secretMACKey2AS, mac_key));


  0.

let authserver() =

  ((authserver_AuthCode()) | (authserver_TokenCode()) | (
     authserver_TokenCodeRefresh()))

  .

(* ************************************************************
   * RESOURCE SERVER: a server that hosts a resource          *
   ************************************************************ *)
let resourceserver() =

  (* resource request *)
    in(net, (resource_url:bitstring, token_code:Token, N:nonce,
       mac_string:mac));
    (* I build the normalized string for HMAC generation *)
    let(normalized_string:bitstring) = (N, resource_url) in

    (* mac_key exchange between authserver and resourceserver *)
    get mac_keys(=token_code, mac_key) in

    (* check of mac code *)
    let (=mac_string) = hmac_sha_256(normalized_string, mac_key) in
    event resource_accepted(token_code, mac_string);
    out(net, resourcecontent(resource_url));

  (* test secrecy *)
    (* out(net, senc(secretTokenRS, Token_to_bitstring(token_code)));
       *)
    out(net, senc(secretMACKeyRS, mac_key));

  0.


 process
  (* register a client into AS *)
  insert RegisteredClients(A_client_id, A_client_password);

  ( (!UserAgent()) | (!authserver()) | (!client(A_client_id,
     A_client_password)) | (!resourceserver()) )
```