Trust negotiation protocol support for secure mobile network service deployment

Daniel Díaz-Sánchez, Andrés Marín, Florina Almenarez, Celeste Campo, Alberto Cortés and Carlos García-Rubio

Universidad Carlos III de Madrid, Avda, de la Universidad 30, 28911 Leganés (Madrid), e-mail: {dds,amarin,florina,celeste,alcortes,cgr}@it.uc3m.es

Abstract User-centric services might enforce requirements difficult to be endorsed by visited networks unless tight coupled trust relations are previously established among providers. Maintaining those fixed trust relations is costly and unmanageable if the number of providers increases. Moreover, it requires providers to use a common security model, credentials, policies.... Trust Negotiation can be the solution to this problem since allows to negotiate gradually a security state enabling multiple factor authentication and authorization even for "strangers" by exchanging various credentials. However, there are still two problems to solve, the first one is the delay introduced by the trust negotiation messages if used as bootstrapping in every interaction; the second one is the lack of protocol support. In this article we address those problems by presenting an extension to TLS that enables trust negotiation and credential issuing (to speed-up following interactions) over a secure channel.

1 Introduction

Mobile and legacy networks are converging to "beyond 3G networks" to provide user-centric services regardless the access network. Users might use various Network Access Providers, Internet Service Providers, and services performing horizontal o vertical handover spontaneously depending on the traffic demand of their applications, as described in [1].The required degree of cooperation among providers can be achieve in two ways: by forcing providers to use the same security model, policies, credentials and protocols, requiring tight coupled trust relations among them; or allowing flexible on-demand trust negotiation. Obviously, those fixed trust relations are costly and unmanageable when the number of providers increases.

Different trust management and access control systems have been proposed for distributed authentication/authorization, like PolicyMaker [2, 3], KeyNote [4], SPKI/SDSI, or the Generic Authorization and Access-control API [5]. Other works like [6] focus on providing trust management systems with adequate semantics so that the policies are correctly defined, understood, and can be proven so. Since finding a common access control solution is not a simple solution, it is necessary a way to combine information from different access control systems. Trust negotiation [7]

[8] can be used to find a solution to this problem since allows to negotiate gradually a security state enabling multiple factor authentication and authorization even for "strangers". This is done by exchanging various credentials: following a path of credential disclosure, users can increase the level of security gradually until the level needed for accessing a service is reached.

The initial problem of trust negotiation is how different credentials and policies, from different providers and languages, can be combined. This problem is discussed and solved in [9] and [6]. [9] shows how a sequential disclosure path can be built respecting user's privacy and avoiding abuse. That article demonstrate how different requirements, extracted from policies belonging to different actors and perhaps written in different languages can be combined using a single decision engine.

Furthermore, other problems of trust negotiation need to solved: the first one is the delay introduced by the trust negotiation messages if used as bootstrapping in every interaction; the second one is the lack of protocol support. In this article we concentrate on how attribute certificates [10] can be requested and issued, as "trust tickets", after a successful trust negotiation, using TLS, to speed-up future interactions. Although there are similarities with identity certificates acquiring process, we deal also with delegation issues which do not appear in authentication. Regarding protocol support, looking to the authentication problem in service and network access, we can probably agree that the situation has been clarified by extensive use of the TLS/SSL protocol for services and EAP-TLS protocol for network access. In this paper we support that TLS can also play a similar role in trust negotiation as it does in authentication.

The rest of the article is organized as follows: in Sect. 2 we describes the extensions done to TLS. Sect. 2.1 briefly explains the architectural changes done to TLS and Sect. 2.2 describes the additions to the handshake and protocol messages to support trust negotiation over TLS. In Sect. 2.3 the attribute certificate request format is presented. Then, Sect. 2.4 shows how attribute certificate issuing can be requested with TLS. Finally, Sect. 3 presents related work and conclusions.

2 Trust negotiation and authorization issuing over TLS

This section starts with a brief description of our Privilege and Trust Negotiation Layer for TLS. Sect. 2.2 discusses on architectural issues and TLS handshake handshake extensions for Trust Negotiation are defined in Sect. 2.1. Then, to handle the use of trust tickets, we will show our proposal for an Attribute Certificate Request Format, in section Sect. 2.3; and its usage with TLS in Sect. 2.4.

2.1 Privilege and Trust Negotiation Layer for TLS

TLS provides one-round mutual authentication. As discussed in previous section, multiple factor authentication and authorization is needed in complex environments for fine grained access control. Fortunately, TLS can be extended as described in [11, 12]. The generic extension mechanism is based on the handshake client and server hello messages. The extension mechanism is designed to be backwards compatible: servers should ignore unsupported extensions. In this section we describe a mechanism to negotiate trust over TLS. Before going further in the proposed TLS extension, we briefly describe TLS internals.

TLS architecture: TLS defines the *TLS Record Protocol* as a layered protocol. The clients of this layer provides fields for length, description and content. Messages are fragmented in manageable blocks, optionally compressed, encrypted and authenticated using a Message Authentication Code (MAC)as described in *Connection State*. *Connection State* specifies, among others, the MAC algorithm, the bulk encryption algorithm, the compression algorithm and the master-secret key. The TLS standard defines four client layers of the *TLS Record Protocol*: the handshake protocol which initially derives a key for secure message exchange; the alert protocol which produces messages conveying the severity of an alert and a description; the *change cipher spec* protocol which signals changes on the current *Connection State*; and the application protocol, for instance HTTP, FTP or SMTP. *Connection State* is negotiated during a handshake phase. A symmetric key is securely derived from random data sent by peers by using a an asymmetric algorithm (RSA or Diffie Hellman). Once the key is derived, the new *Connection State* is used to protect TLS Record Protocol messages against eavesdropping.

Handshake: The client initiates the handshake sending a *Client Hello* message and a set of optional extensions. In this message the client provides random data, a time stamp, session identification and a set of cipher suites and compression mechanisms. The server answers with a *Server Hello* message, providing also random data and selecting one cipher suite and compression method from the set provided by the client. The *Server Hello* message can be followed by a set of optional extensions. The server optionally sends a certificate or a *ServerKeyExchange* message that will be used to perform the secure key exchange (using certificates or Diffie-Hellman). Optionally, server side can also request a certificate from the client (sending a *CertificateRequest* message).

The client optionally sends its certificate, if requested by the server, and sends the mandatory *ClientKeyExchange* message, which exchanges the pre-master key that will be used to derive the final symmetric key. The *ChangeCipherSpec* and *Finished* messages signals that a new symmetric key is ready for use in *Connection State*, to protect upcoming application messages.

Proposed architecture: To identify the involved entities, consider a mobile device governed by a set of policies. Those policies are controlled by different access

control engines (ACEs). Every ACE processes the policies it understands, extracting **requirements** and **policy items** from them. A **policy item** is a formal definition for a requirement so it can be used by other peer to find out which credential should be disclosed in order to satisfy a requirement. ACEs register requirements and policy items to the **decision engine** as explained in [9]. Fig. 1 shows the architecture.

Fig. 1 Access control engines and its relation to the privilege and trust negotiation layer of TLS.

To accomplish trust negotiation in TLS, the handshake protocol module must support a new extension. Besides, a new layer is defined: the *Privilege and Trust Negotiation Layer*(PTN Layer), client of the *TLS record layer*. Fig. 1 illustrates the TLS client layers.

2.2 Trust Negotiation over TLS protocol

The TLS extension described here allows to negotiate trust. We have defined our extension with the intention to allow client and server to negotiate first which policy languages, credentials and mechanisms will be used later, during trust negotiation. This initial negotiation is performed in a mutual agreement basis during the TLS handshake. Once the secure channel is established, the Trust Negotiation starts. The proposed message exchange for the Privilege and Trust Negotiation Layer is the following (optional fields and messages are marked with *):

```
Client                                          Server
--
ClientHello
```

```
<PTLNExtension*>
<AuthRequestExt*>          -------->
                                          ServerHello
                                          <PTLNExtension*>
                                          <AuthRequestExt*>
                                          Certificate*
                                          ServerKeyExchange*
                                          CertificateRequest*
                            <--------     ServerHelloDone
Certificate*
ClientKeyExchange
CertificateVerify*
[ChangeCipherSpec]
Finished                    -------->
                                          [ChangeCipherSpec]
                            <--------     Finished
Handshake ends. Trust Negotiation Messages exchange starts in
parallel to Application Data and Authorization Issuing Requests
TrustNegotiationMessage*    -------->
                            <-------     TrustNegotiationMessage*

ACRequest*                  ------->
                            <-------     ACResponse*

Application Data            <------->    Application Data
```

Let us further describe the additions to the original TLS handshake:

1. The client advertises the list of Trust Negotiation mechanisms it supports by sending the PTLN extension at the end of the *Client Hello* message. This message contains information about which policies, authentication and authorization credentials and mechanisms can be used.
2. If the server understands the extension, it is redirected to the *Privilege and Trust Negotiation Layer*. The server sends to the client, at the end of the *Server Hello* message, a selection of mechanisms provided by the client (Fig. 1).
3. The TLS handshake continues as defined in the standard until a secure channel is established -after the server sends the *Finished* message.
4. At this point, the application layer traffic flows in parallel to trust negotiation protocol message exchange. As can be seen in Fig. 1, application traffic is intercepted by an enforcement point and redirected to the decision engine. The decision engine decides if the client, identified by the TLS session identifier, has a security state higher enough to access a given application or resource.

   - If security state is not enough, the decision engine conveys to the PTN layer the policy item that describes the next requirements to be fulfilled. The PTN server layer sends to the PTN client Layer a *TrustNegotiationMessage* containing a *Request* command. Then, it waits until client sends a *TrustNegotiationMessage* with an *Assert* command containing the credentials. Since trust negotiation messages are managed by the PTN layer, applications do not need to be aware of the underlying access control layers.
   - Otherwise, if the security state is adequate, the application traffic is not blocked.

The structure of the Trust Negotiation extension is the following:

```
struct {
    TrustNegotiationMechanisms TrustN_avail_mechs<0..2^16>;
```

```
}PTNLExtension

struct {
    TrustObjectType trust_object_type;
    opaque     Uri<0..2^16-1>;
    opaque     OID<0..2^16-1>;
} TrustNegotiationMechanisms

enum{
    policy(0), credential(1), strategy(2),...
} TrustObjectType;
```

TrustN_avail_mechs contains a list of supported trust negotiation mechanisms or objects, so both sides can determinate if they "speak" the same language (policies and credentials). For instance, the client might express that he understands XACML policy items and X.509, SAML and KeyNote credentials.

Trust negotiation protocol messages handle three different lists of messages (*TrustNegotiationMessage*): assertions, requirements and information. Thus, in a single message, an entity can request credentials to the other part and also fulfill requirements previously sent by the other side (assertions).

```
struct{
    Message assertion_list<0..2^16-1>;
    Message requirement_list<0..2^16-1>;
    Message information_list<0..2^16-1>;
}TrustNegotiationMessage;
struct{
    Type type;
    Object      PayLoad_list<0..2^16-1>;
    Parameter   Parameters<0..2^16-1>;
}Message;
struct{
    Type type;
    opaque  PayLoad<0..2^16-1>;
}Parameter;
struct{
    opaque OID<0..2^16-1>;
    opaque URI<0..2^16-1>;
}Type;
```

The parameters of *Message* structure allows to combine different policy items using operators or logic expressions.

2.3 Attribute Certificate Request Format

This section describes an Attribute Certificate Request Message Format (ACRM) based in RFC2511 [13], that describes the Certificate Request Message Format. RFC2511 describes the message format used to convey a public key certificate request to a Certification Authority. It requires the client to provide a *Proof of Possession (PoP)* of the private key associated with the certificate's public key. The PoP can be done in different ways, depending on the key type: using a signature or a password based MAC. In addition we have considered different types of authorization issuing:

- Direct issuing: an entity **A** requests an attribute certificate directly to the *SoA* or through the *Registration Authority (RA)*. The *RA* is the entity that should verify the request before the AC is issued by the *SoA*, obviously, the *RA* should be trusted by the *SoA*. The *RA* can be also the entity which receives the payment for a service and the attribute certificate the ticket that gives access to the service.
- Indirect issuing: an entity **B** requests an attribute certificate to a *SoA* (or through a *RA*) on behalf of other entity **A**. This kind of indirect delegation might be useful for limited devices only able to perform some cryptographic primitives.
- Delegation: an entity **A** requests an attribute certificate to another entity **B**. If **B** accepts it delegates the privilege to **A**.

Since ACs can be bound to a PKI certificate or public key, the ACRM should contain the PKI certificate or the key that identifies the user. The proof of possession contained in a ACRM should proof the possession of the associated private key. PKI certificate request messages provide the necessary means for the user to prove possession of private key, but no mechanisms are specified to convey the request from the RA to CA, this is done out-of-band. However, we provide a procedure for any involved entity to assert it agrees with the request by enveloping it and signing it. Fig. 2 shows an example. The ASN.1 syntax of an Attribute Certificate Request Message is the following[1]:

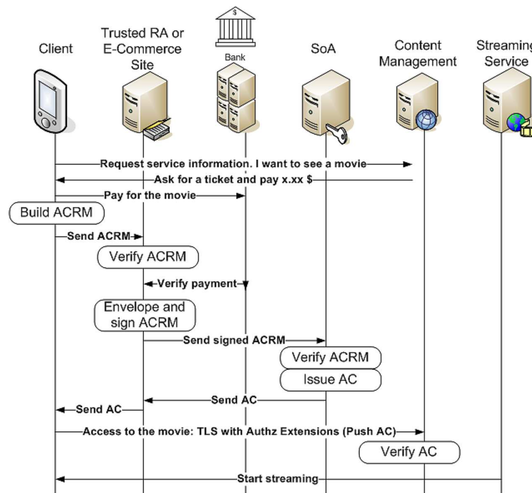


Fig. 2 Request access to a multimedia content

```
AttCertReqMessages ::= SEQUENCE SIZE (1..MAX) OF
AttCertSignedRequestMsg
```

[1] the ASN.1 syntax uses implicit tagging and imports the same as in RFC 2511 and also PKIX-AttributeCertificate iso(1) identified-organization(3) dod(6) internet(1) security(5) mechanisms(5) pkix(7) id-mod(0) id-mod-attribute-cert(12)

```
AttCertSignedRequestMsg ::= SEQUENCE {
    -- signature-chained message
    -- allows involved entities to assert the validity of a request
    reqMsg  CHOICE {
    -- at least one attCertReqMsg or attCertSignedReqMsg shall be present.
          attCertReqMsg [0]  AttCertRequestMsg,
          attCertSignedReqMsg [1] AttCertSignedRequestMsg}

    authInfo CHOICE {
           signer        [0] GeneralName,
    -- used only if an authenticated identity has been established for the signer:
    -- an RA or other entity known by RA or SoA asking for a certificate
           delegationAttCertPath   SEQUENCE SIZE (1..MAX) OF ACPathData } OPTIONAL,
    -- used only if an entity with delegated privileges ask for a certificate on behalf of other
    signatureAlgorithm   AlgorithmIdentifier OPTIONAL,
    signatureValue       BIT STRING OPTIONAL}

ACPathData  ::=  SEQUENCE {
    certificate                [0]  Certificate  OPTIONAL,
    attributeCertificate       [1]  AttributeCertificate  OPTIONAL
}
```

The recursive definition of *AttCertSignedRequestMsg* allows involved entities to envelope and sign requests if they agree with the content. For indirect issuing, the entity that indirectly delegates a privilege must include its PKI certificate and the attribute certificate that includes the delegated privilege (see Fig. 3)

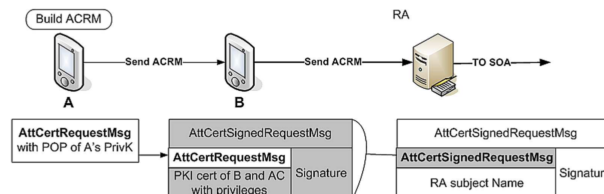


Fig. 3 ACRM envelope and signature for indirect delegation

AttCertRequestMsg contains the AC request itself, an optional proof of possession and a set of optional registration information that should be considered in the same terms as done in RFC 2511. The request might contain the client PKI certificate or the key to which the AC is bound. This parameter is optional since the RA or CA can obtain this information out-of-band.

```
AttCertRequestMsg :: = SEQUENCE {
    attCertReq       AttCertRequest,
    pop              ProofOfPossesion OPTIONAL,
    regInfo          SEQUENCE SIZE (1..MAX) of AttributeType And Value OPTIONAL
    -- regInfo: can contain information of publishing, payment...
}

AttrCertRequest ::= SEQUENCE {
    attCertReqID     Integer,
    boundCert        Certificates OPTIONAL, -- PKI certificate of entity
    attCertTmpl      AttCertTemplate,
    controls         Controls OPTIONAL
}

Certificates ::= SEQUENCE {
```

```
    userCertificate   Certificate,
    certificationPath CertPath OPTIONAL
}
```

The cert template provides the necessary information to issue the AC. *Holder*, *AttCertIssuer* and *Attribute* syntax is the same as described in [10] and [14]. The rest of the parameters should be treated as in RFC 2511.

```
AttCertTemplate ::= SEQUENCE {
    version        [0]  Version OPTIONAL,
    serialNumber   [1]  Integer OPTIONAL,
    signatureAlg   [2]  AlgorithmIdentifier OPTIONAL,
    holder         [4]  Holder OPTIONAL,
    issuer         [3]  AttCertIssuer OPTIONAL,
    -- syntax for issuer and holder
    validity       [5]  Validity OPTIONAL,
    attributes     [6]  SEQUENCE SIZE(1..MAX) of Attribute OPTIONAL,
    extensions     [7]  SEQUENCE SIZE(1..MAX) of Extension OPTIONAL
}
```

The PoP depends on the key type. For instance, with RSA keys any PoP can be used. For *keyAgreement*(only) keys, the field *thisMessage* is used for PoP. This field contains a MAC (over the DER-encoded value of the *attCertReq* parameter in *AttCertReqMsg*, based on a key derived from the end entity's private DH key and the CA's public DH key and calculated as explained in RFC 2511.

```
ProofOfPossession ::= CHOICE {
   signature         [0] POPOSigningKey,
   keyAgreement      [1] POPOPrivKey
}

POPOSigningKey ::= SEQUENCE {
   algorithmIdentifier     AlgorithmIdentifier,
   signature               BIT STRING }
   -- signature MUST be computed on the DER-encoded value of attCertReq

POPOPrivKey ::= CHOICE {
   thisMessage       [0] BIT STRING,
   subsequentMessage [1] SubsequentMessage,
   -- possession will be proven in a subsequent message
   dhMAC             [2] BIT STRING }
```

2.4 Authorization Request with TLS

In this section we describe a TLS extension that allows to securely request an AC to a RA, SoA or any entity able to issue it as, for example, a consequence of a successful trust negotiation. This protocol extension considers protocol messages to cover any proof of possession procedure, including TLS handshake with mutual authentication. Figs. 4(a) and 4(b) show two scenarios where this request mechanism over TLS can be used: the client starts sending a Client Hello message containing the *AuthzRequestExtension*, thus it discovers whether the server supports issuing over TLS or not. If the server understands the extension, it should send the same content back to the client.

The *AuthzRequestExtension* indicates that the client might request an attribute certificate during the TLS connection. Any required information will be given using the Attribute Certificate Request Message (ACRM) format. The client asks

the server to request client certificate, for PoP, during handshake, by setting *useClientCert* to true: if the AC to be issued should be bound to the certificate used during handshake, no proof of possession is required in ACRM (a successful TLS handshake with mutual authentication already demonstrates the possession). Moreover, to distinguish among different SoAs, targeted by the same RA, the client can indicate the target SoA in *targetSoA* field. The syntax of the TLS extension is the following:

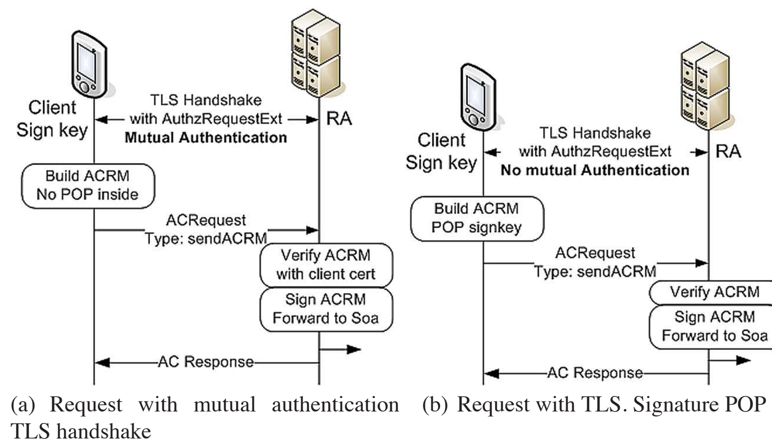
```
struct{
    Boolean useClientCert,
    opaque  targetSoA -- Optional
}AuthzRequestExt;
```

The new protocol messages added to TLS are:

```
struct{
    Boolean useClientCert,
    opaque  targetSoA -- Optional
}AuthzRequestExt;

struct{
    ACReqType acReqType,
    opaque    ACRMs<0..2^16-1>
}ACRequest;

enum{ SendACRM(0), ForwardApprovedACRM(1), RequestKeyExchange(2),..
}
struct{
    opaque payload<1..2^16-1>
}challenge_Response;

struct{
    opaque  ASN1.AC<1..2^16-1>,
    Boolean encrypted
}ACResponse;
```

Fig. 4(a) shows the message exchange over TLS for an AC request. In this case, the possession of the key is proven with a successful TLS handshake. If no mutual authentication is performed during TLS handshake, see Fig. 4(b) , the client can build an ACRM and include a proof of possession based in a signature key.

If client's key can only be used for key exchange, the client can send a *ACRequest* message with an empty ACRM. The type of the *ACRequest* message sent by the client is *RequestKeyExchange*. This message is immediately followed by a TLS *ClientKeyExchange* and *Finalize* (consumed by the PTN layer at server side) and used to derive a key. The random data exchanged during the TLS handshake will be used in this key exchange using a Diffie-Hellman key to compute the master secret as explained in [11]. Key exchange proof of possession should be used only by clients whose keys cannot be used to sign. Finally, the client can ask for a challenge-response message exchange, using the *SubsequentMessage* field of an ACRM or request the attribute certificate to be encrypted. The proof of possession is proven by sending a hash of the received AC, once decrypted.



(a) Request with mutual authentication TLS handshake

(b) Request with TLS. Signature POP

3 Conclusions and Related work

We have presented a TLS extension for trust negotiation and other for requesting ACs: we complement other works, as [9], with protocol support. The extensions allows to negotiate trust over an encrypted channel and then request a **"trust ticket"**. We designed a new layer for TLS that allows to perform trust negotiation in parallel to the application protocol. Thus, multiple authentications and authorizations can be performed on demand, depending on the application behavior.

We also have elaborated a message format suitable for requesting ACs over TLS that can reuse TLS mutual authentication to perform the proof of possession. So, as a result of a successful trust negotiation, a credential can be issued to speed up the process in future interactions.

Other works as Farrel [15], and Brown and Housley [16] propose similar extensions. In [15], Farrel defines an approach to the authorization problem. Farrel, uses some extensions on TLS but introduces some protocol messages that are not negotiated using extensions during TLS handshake. This might not work: legacy TLS servers can be broken since there is no prior negotiation before using an application specific protocol message. [16] is a parallel work to ours but only enables authorization over TLS: enables the use of Attribute Certificates and SAML. However, our goal is to enable **multiple factor authentication and authorization regardless the credentials type**. Moreover, they require a double TLS handshake and allows only one credential exchange. Other works on trust negotiation, as [17], allow to exchange credentials using extensions but requires to perform a handshake whenever a credential needs to be exchanged.

Our proposal is more efficient since we only require one handshake. A double handshake implies a double exchange of handshake messages thus, at least 12 messages are exchanged again, if mutual authentication is required. Furthermore, since we rely on an efficient agnostic trust negotiation decision engine, as described in [1], we can deal with multiple credentials and policies types. Regarding to implementa-

tion, [16] does not require new protocol messages, but the code has to be changed to allow for adjacent double handshakes. In our proposal, new protocol messages have to be coded, and the state machine has to be modified accordingly. We have developed a toolkit for AC management, based on OpenSSL, compliant with ITU and IETF specifications allowing to build the requests, process them and issue the ACs . We hope that in the near future, network applications can benefit of TLS performing the trust negotiation on their behalf.

References

1. Díaz, D., Marín, A., Almenárez, F., Garcia-Rubio, C., Campo, C.: Context awareness in network selection for dynamic environments. 11th IFIP International Conference on Personal Wireless Communications "PWC06". Lecture Notes In Computer Science Editor: Springer-Verlag GMBH (2006)
2. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In: Proceedings IEEE Symposium on Security and Privacy, 1996, IEEE Computer (1996)
3. Blaze, M., Feigenbaum, J., Strauss, M.: Compliance checking in the policy maker trust management system. In: Financial Cryptography. Number 1465 in Lecture Notes in Computer Science, Springer-Verlag (1998)
4. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The keynote trust management system version 2. Technical Report RFC 2704, IETF (1999)
5. Ryutov, T., Neuman, C., Kim, D.: The specification and enforcement of advanced security policies. In: Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, 2002, IEEE Computer (2002)
6. Li, N., Grosof, B.N., Feigenbaum, J.: Delegation logic: A logic-based approach to distributed authorization. ACM Trans. Inf. Syst. Secur. **6** (2003) 128–171
7. Squicciarini, A.C.: Trust negotiation systems. In: EDBT Workshops. (2004) 90–99
8. Bertino, E., Ferrari, E., Squicciarini, A.: X -tnl: An xml-based language for trust negotiations. policy **00** (2003) 81
9. Díaz, D., Marín, A., Almenárez, F.: Enhancing access control for mobile devices with an agnostic trust negotiation decision engine. Personal Wireless Communications. Springer series in Computer Science. ISSN: 1571-5736. (2007)
10. (ITU), I.T.U.: The directory: Public-key and attribute certificate framework. Technical Report X.509, International Telecommunication Union (ITU) (2005)
11. Dierks, T.: The tls protocol. Technical Report RFC 2246, IETF TLS Working Group (1999)
12. Blake-Wilson, S.: Transport layer security (tls) extensions. Technical Report RFC 3546, IETF TLS Working Group (2003)
13. Myers, M., Adams, C., Solo, D., Kemp, D.: Internet x.509 certificate request message format. Technical Report RFC 2511, IETF TLS Working Group (1999)
14. Farrell, S., Housley, R.: An internet attribute certificate profile for authorization. Technical Report RFC 3281, IETF PKIX Working Group (2002)
15. Farrell, S.: Tls extensions for attributecertificate based authorization. Technical Report draft-ietf-tls-attr-cert-01.txt, IETF Transport Layer Security Working Group (1998)
16. Brown, M., Housley, R.: Transport layer security (tls) authorization extensions. Technical Report draft-housley-tls-authz-extns-07.txt, IETF (2006)
17. Hess, A., Jacobson, J., Mills, H., Wamsley, R., Seamons, K., Smith, B.: Advanced client/server authentication in tls (2002)