Chained and Delegable Authorization Tokens

G. Navarro, J. Garcia, J. A. Ortega-Ruiz
Dept. Informàtica, Universitat Autònoma de Barcelona,
Edifici Q, 08193 Bellaterra - Spain
Email: {gnavarro,jgarcia,jao}@ccd.uab.es

Abstract— **In this paper we present an overview of an access control system based on tokens and delegable hash chains. This system takes advantage of hash chains in a similar way as the micropayment systems. On the other hand it uses an authorization infrastructure which allows to delegate authority and permissions by means of authorization certificates and delegation. The system is named CADAT (*Chained and Delegable Authorization Tokens*). We also describe one of the applications of CADAT, which is used as a token based access control for a secure mobile agent platform.**

Index Terms— **Access Control, Authorization, Delegation, Hash Chains.**

I. Introduction

The access control and protection of computerized system is an active field of research, development and application. Lately there have been proposals for models, techniques and novel systems about access control in distributed systems. Normally they tend to look for the design of more flexible, scalable, and user-friendly systems while keeping a high degree of security.

Delegation, for example, introduces a lot of flexibility to a system, where the permission can be transfered or delegated between the users of the systems. Some of the first systems introducing delegation of authorization and the required infrastructure where KeyNote [1] and SPKI/SDSI (*Simple Public Key Infrastructure/Simple Distributed Secure Infrastructure*)[2]. Some authors refer to these systems as *trust-management*.

Another interesting field in access control is the use of access *tokens*. In these systems a user receives a token or ticket, which allows her to access a given resource or service. Token-based access control has been successfully used in several scenarios. For instance, Kerberos [3] is a popular system which makes use of access tickets.

An application, somehow similar to access tokens, are the micropayments. A micropayment system allows the payment of small amounts of money. Although micropayment systems have been around for a considerable time and have received some critics, they are still an interesting research field [4]. As a proof of concept, just consider companies that have appeared lately offering micropayment based systems, such as Peppercoin (`http://www.peppercoin.com`) or Bitpass (`http://www.bitpass.com`). One of the most interesting issues of the micropayment systems has been the introduction of hash chains to represent the transactions [5], [6], [7]. The use of hash chains allows to make the issuing of micropayment more flexible, since it substitutes computationally expensive cryptographic operations by simpler operations (hash functions).

There have been recently some propositions to use authorization infrastructures to implement micropayment systems. One of these proposals [8], introduces a model for using delegation in micropayment systems.

In this paper we present a token-based access control system. This system presents some advantages from micropayment systems and the delegation of authorizations. Our proposed system is called CADAT (*Chained and Delegable Authorization Tokens*). In CADAT the authorization tokens are constructed from a hash chain, which makes it faster and efficient to issue and verify tokens. Furthermore, we introduce the possibility to delegate token chains between users of the system in several ways, which adds flexibility to the system. CADAT is implemented by using SPKI/SDSI (Simple Public Key Infrastructure / Simple Distributed Secure Infrastructure), which provides the main framework to express authorizations and delegations. We also describe the application of CADAT as a token-based access control system in an specific scenario, a secure mobile agent application.

In Section II we introduce the main bases of the CADAT system. Section III introduces the SPKI/SDSI framework in relation to CADAT. In Section IV we show how to encode the hash chain based permissions into SPKI/SDSI certificates, providing the necessary modifications to SPKI/SDSI. Next, we show the main functionality of CADAT through an example in Section V. And in Section VI we discuss the application of CADAT in mobile agent systems. Finally, Section VII summarizes our conclusions and further work.

II. Hash chains and delegation

In this section we show how we can use delegation and hash chains together. It is based on the model and notation of [8]. We have simplified the model a little bit to make it more readable. The main idea is to consider the elements of a hash chain as permissions or authorizations.

We adopt the following notation and definitions:

- K_A: public key of the user A.
- sK_A: private key of the user A.
- $\{m\}_{sK_A}$: digital signature of message m, signed by user A with the private key sK_A.
- $(|K_B, p|)_{K_A}$ a direct delegation, where the user A, delegates permissions p to the user B. This relation can be established by means of an *authorization certificate*. At the moment we denote such a certificate as $\{|K_B, p|\}_{sK_A}$.

- $h(m)$: one-way cryptographic hash function applied to m.
- $[h^n(m), h^{n-1}(m), \ldots, h^1(m), h^0(m)]$: hash chain constructed from the initial message m (the seed of the hash chain). Normally, m may include information such as some random value (or nonce) to ensure its uniqueness.
- $\mathcal{P}$: set of all the permissions managed by the system. There is an partial ordering in the set ($\preceq$) determined by the inclusion of one permission into another. For instance, $x \preceq y$, intuitively means that if a user holds permission y she also holds permission x. There is a least upper bound defined by the intersection ($\wedge$) of permissions, and a greatest lower bound defined by the union of ($\vee$). The order relation and the set of permissions form a *lattice*.
- $\mathcal{P}_{\mathcal{H}}$: subset of permissions ($\mathcal{P}_{\mathcal{H}} \subseteq \mathcal{P}$), where each permissions is expressed as an element of a hash chain. The order relation between elements of $\mathcal{P}_{\mathcal{H}}$ is defined as: given $x, y \in \mathcal{P}_{\mathcal{H}}$, $x \preceq y$ in relation to principal P, if and only if P, knowing y, can easily determine some x and i, such that $h^i(x) = y$[8]. It means that principal P can find x and i without having to invert the hash function.

Following with these notation we can express the reduction rules of SPKI/SDSI [2], given the set of permissions $\mathcal{P}_{\mathcal{H}}$, in the following way:

$$\frac{(|K_B, x|)_{K_A}; y \preceq x}{(|K_B, y|)_{K_A}} \tag{1}$$

$$\frac{(|K_C, x|)_{K_B}; (|K_B, y|)_{K_A}}{(|K_C, x \wedge y|)_{K_A}} \tag{2}$$

The use of hash chain elements as permissions has several advantages, such as the possibility to issue specific permissions without having to issue any kind of certificate. We can also delegate (or transfer) parts of the hash chain to other principals.

To show the use of hash chains with delegation, we consider the following example. There are three users Alice, Bob and Carol, with their own public keys K_A, K_B, K_C. We assume that Alice is some authorization authority. Alice generates the following hash chain from the initial seed m: $[h^5(m)$, $h^4(m)$, $h^3(m)$,$h^2(m)$,$h^1(m)]$, where each element of the chain corresponds to an specific permission.

Alice can issue the following certificate:

$$\{|K_B, h^5(m)|\}_{sK_A} \tag{3}$$

With this certificate, Alice delegates $h^5(m)$ to Bob, who now, holds the permission $h^5(m)$. If Alice wants to delegate permission $h^4(m)$ to Bob, there is no need for Alice to issue another certificate. Alice just has to make public the value $h^4(m)$. Once it is public, Bob becomes automatically the holder of the permission $h^4(m)$. Bob is in the position of demonstrating that he holds such a permission because of certificate (3). In a same way, Alice can make public the successive permission, for instance, $h^3(m)$.

In a more general formulation, if Bob has a certificate, which directly grants him the permission $h^i(m)$, and Alice makes public the value $h^j(m)$, where $i > j$, Bob becomes the holder of permissions $h^i(m)$, …, $h^j(m)$. Given the reduction rule (1), we have that $(|K_B, h^i(m)|)_{sK_A}$, and $h^j(m) \preceq h^i(m)$, so $(|K_B, h^j(m)|)_{sK_A}$.

Another important issue is the introduction of delegation. Bob could delegate part of the permissions to another principal. Following with the example, Bob is in possession of $h^5(m)$, $h^4(m)$ and $h^3(m)$. He can delegate the last permission to Carol by issuing the following certificate:

$$\{|K_C, h^3(m)|\}_{sK_B} \tag{4}$$

Bob delegates $h^3(m)$ to Carol. If Alice makes public the value $h^2(m)$, Carol can demonstrate that she is in possession of such permission, because of the certificates (3) and (4). It is important to note that in no case, Carol can demonstrate she holds permissions $h^5(m)$ and $h^4(m)$. She can only receive permission of the chain which have and index lower or equal to $h^3(m)$.

We have seen how hash chains have important advantages when combining them with authorization or *trust management* systems. In the following section we show how this model can be implemented by using SPKI/SDSI. In [9], the authors show how to implement a similar system in KeyNote.

III. SPKI/SDSI AUTHORIZATION CERTIFICATES

SPKI/SDSI (Simple Public Key Infrastructure/Simple Distributed Secure Infrastructure), is a certificate-based authorization infrastructure, which can express authorizations and its delegation by means of *authorization certificates*. It also provides a distributed name system based on local names, by means of *name certificates*.

In SPKI/SDSI each principal has a pair of cryptographic keys, and its represented by its public key. In other words, we can say that in SPKI/SDSI each principal is its public key, and it is represented by the key or a hash of the key. An authorization certificate binds a public key with an authorization or permission. This way we can avoid the indirection present in traditional PKIs, where there is a binding between a public key and a global identifier (or distinguished name) and another one between the identifier and the authorization (permission or attribute). We can denote an authorization certificate as:

$$(I, S, tag, p, V) \tag{5}$$

Where:

- I: *issuer*. The principal granting the authorization.
- S: *subject*. The principal receiving the authorization.
- tag: *authorization tag*. The specific authorization being granted by the certificate.
- p: *delegation bit*. If it is active, the subject of the certificate can further delegate the authorization (or a subset) to another principal.
- V: *validity specification*. It includes the validity time range (not-after and not-before) of the certificate and other possible conditions (currently online tests for revocation, revalidation and one-time revalidation).
- *Comment*: although we do not show it in the notation, the certificates include a field of arbitrary information.

As one can see, the authorization certificates of SPKI/SDSI allows us to easily express the authorization certificates commented in Section II. For instance, certificate $\{|K_B, p|\}_{sK_A}$ can be expressed in SPKI/SDSI, for a given validity time V as: $(K_A, K_B, p, 1, V)$. Note that the delegation bit is active, so K_B can further delegate the authorization, K_A could avoid this by setting the delegation bit to 0 if needed.

IV. Hash chains as SPKI/SDSI authorizations

As we have seen permissions are expressed in SPKI/SDSI as the element *tag*. The format used by SPKI/SDSI to represent all the information is S-expression. In order to make it easy to process the authorizations, we include the index of the hash chain component , and an identifier of the chain in the authorization *tag*. This way a permission $p \in \mathcal{P}_{\mathcal{H}}$ will have the following format:

$$p = (cid, i, h^i(m)) \tag{6}$$

Where i is the index of the element in the chain and cid is the identifier of the whole hash chain. All the elements of the same hash chain have the same cid, which is a random bit string big enough to ensure its uniqueness. The seed, m includes additional information such as the public key of the principal that generated the chain, etc. The hash on m also includes i and cid in each step, making it more difficult to forge it. We do not get into details on how is the concrete information in m and how is the hash exactly computed, to keep the notation more readable, and because it may depend in the specific application of the permission. The reader may refer to [5], [6], [7] for specific ways to build such a hash chain.

SPKI/SDSI provides a certificate chain discovery algorithm to find authorization proofs in delegation networks [10]. This algorithm is based on basic intersection operations to provide certificate reduction rules. In [2] the intersection of the tag element is determined by the operation *AIntersect*.

In order to represent the authorization (6) in a SPKI/SDSI tag, we consider two alternatives. These alternatives are based on the need of verifying the hash of the permission in the tag intersection operation. When we have to intersect two permissions with the hash elements $h^i(m)$ and $h^j(m)$, if $i \geq j$ the resulting tag will be the $h^i(m)$. The intersection operation can also verify that $(h^j)^x(m) = h^i(m)$ for some x. This verification allows to immediately dismiss forged or erroneous permissions. If the verification is not carried by the intersection operation, it is important to note that the verification has to be done afterwards to validate an authorization proof.

A. Tag intersection without hash verification

In this case we can encode the permissions by using existing SPKI/SDSI structures. For example, a straightforward representation of the a permission like (6) in SPKI/SDSI S-expression format can be:

```
(tag
    (h-chain-id |123456789|)
    (h-chain-index (* range numeric ge 7)))
    (h-val (hash
       md5 |899b786bf7dfad58aa3844f2489aa5bf|))
```

Where `h-chain-id` is the identifier of the hash chain, *h-chain-index* is the index of the element in the hash chain and *h-val* is the value of the hash itself. The most important element is the index, which is expressed as a numeric range that will intersect with a range greater or equal to 7 in this case.

The only problem of the previous example is that if we include the value $h^7(m)$ in the *tag*, the intersection operation will not properly work. What we do is to introduce a little modification. The value `h-val` has to be treated in a different way. Thus, we put the *h-val* as the comment of the certificate. The certificate may look something like this:

```
(cert
  (issuer
    (hash
     md5 |1ac461a2e12a77ad54c67128b5060f28|))
  (subject
    (hash
     md5 |b0a746de2d5f6038e49a87c9c826bf4e|))
  (tag
    (h-chain-id |123456789|)
    (h-chain-index (* range numeric ge 7)))
  (comment
    (h-val
      (hash
       md5 |899b786bf7dfad58aa3844f2489aa5bf|)))
  (not-after "2004-01-01_00:00:00")
  (not-before "2005-01-01_00:00:00")
)
```

This allows us to use the SPKI/SDSI decision engine directly, without any lose of information. The main disadvantage of this approach is that in order to verify an authorization proof, the verifier needs to do an additional operation: verify the integrity of the hash chain (or subchain).

B. Tag intersection with hash verification

In this case, it is necessary to redefine the tag intersection operation for authorizations corresponding to elements of a hash chain. To do this we introduce a new kind of *tag*, the `<tag-hash-auth>`. The BNF definition of this new tag, according to the SPKI/SDSI tag definition [11] is given in Figure 1.

```
<tag>:: <tag-star> | "(" "tag" <tag-expr>  ")";
<tag-star>:: "(" "tag" "(*)" ")" ;
<tag-expr>:: <simple-tag> | <tag-set> |
             <tag-string> | <tag-hash-auth>;
<tag-hash-auth>:: "(" "hash-auth" <chain-id>
                  <chain-index> <hash>")";
<chain-index>:: "(" "chain-index" <decimal> ")";
<chain-id>:: "(" "chain-id" <byte-string> ")";
```

Fig. 1. Definitionn of `<tag-hash-auth>`.

We also introduce a new intersection operation: *HCAIntersect* (*Hash Chained Authorization Intersection*). The intersection of two `tags` representing permissions of $\mathcal{P}_{\mathcal{H}}$ will result in the tag with the greatest hash chain index, if the hash chain identifier is the same and we can verify the hash values. For example, given the following `tags`:

```
(tag
  (hash-auth
    (hchain-id |lksjfSDFIsdfkj0sndKIShfoMSKJSD|)
    (hchain-index 14)
    (hash md5 |899b786bf7dfad58aa3844f2489aa5bf|)))
```

```
(tag
  (hash-auth
    (hchain-id |lksjfSDFIsdfkj0sndKIShfoMSKJSD|)
    (hchain-index 15)
    (hash md5 |d52885e0c4bc097f6ba3b4622e147c30|)))
```

Its intersection (HCAIntersect) will be equal to the second tag, because the identifier is equal and the index of the second tag is greater that the first one. And we can verify the hash value of the tag. Note that the MD5 of the first value is equal to the second one.

We show the algorithm used by HCAIntersect with hash verification (*HCAIntersect full algorithm*).

Algorithm 1: HCAIntersect full algorithm

```
input : $p = (id_p, i, h^i(m)_p)$, $q = (id_q, j, h^j(m)_q)$, such
        that $p, q \in \mathcal{P}_{\mathcal{H}}$
output: $r$ such that $r = HCAIntersect(p, q)$
begin
    if $id_p \neq id_q$ then $r \leftarrow NULL$ ;
    if $i \geq j$ then
        if verifyHashSubChain(p,q) then $r \leftarrow p$ ;
        else $r \leftarrow NULL$ ;
    end
    else
        if verifyHashSubChain(q,p) then $r \leftarrow q$ ;
        else $r \leftarrow NULL$ ;
    end
end
```

Algorithm 2: verifyHashSubChain function

```
input : $p = (id_p, i, h^i(m_p))$, $q = (id_q, j, h^j(m_q))$,
        where $i \geq j$
output: $res = true$ if $h^i(m)$ and $h^j(m)$ belong to the
        same hash chain, $res = false$ otherwise
begin
    $res \leftarrow false$ ;
    $aux \leftarrow h^j(m_p)$ ;
    for $x \in [(j-1)..i]$ do
        if $aux = h^i(m_q)$ then $res \leftarrow true$;
        $aux \leftarrow h(aux)$ ;
    end
end
```

The implementation of the hash chain elements as authorizations and the HCAIntersect algorithms, has been done in Java using the JSDSI [12] library. JSDSI is an open source implementation of SPKI/SDSI in Java, which has lately been under active development. The implementation of the new *tag* and the algorithm just represented a few modifications of JSDSI.

V. Application to access tokens: CADAT

The main motivation for the design of CADAT is its application as a token based access control system. In this section we show the basic functionality of CADAT through an example. We consider an scenario where news agencies allow access to the news databases to their clients. The access is controlled by tokens, that is, each time a user accesses (or reads) a new he needs to issue a token (depending on the case a given new may require several tokens).

For example the headquarters for the news agency AcmeNews wants to issue 9 access tokens (note that normally token chains will be much larger) to the user Alice, so she can access all the agencies worldwide (AcmeNews-SudAfrica, AcmeNews-India, ...). To do that, AcmeNews issues a contract to Alice authorizing her to use 10 access tokens *acme* (the first token is not used as an access token) for a given validity specification V_0. This contract is represented as an SPKI/SDSI authorization certificate.

$$(AcmeNews, Alice, auth_{star}, p = true, V_0) \tag{7}$$

Where $auth_{star}$ corresponds to: $(acmeID, 10, *)$. This is a `<tag-hash-auth>`, with the hash value equal to "*". It stands for an especial tag symbol used in SPKI/SDSI, which intersects with any kind of character string.

We denote this first certificate as `chain-contract-cert` or *chain contract certificate*, because it establishes a contract which allows Alice to demonstrate that she has been authorized by AcmeNews to use 9 tokens *acmeID*.

Now, Alice can generate the hash chain with 10 elements:

$$\begin{aligned}&[(acmeID, 1, h^1(m)), (acmeID, 2, h^2(m)), \ldots,\\&(acmeID, 10, h^{10}(m))]\end{aligned} \tag{8}$$

The initial message or seed m will normally include information from the certificate (7), or specific information shared by AcmeNews and Alice.

Suppose that Alice goes to the AcmeNews-Antartida agency to look for news regarding the habits of the *Aptenodytes forsteri*[1]. To do that, Alice, establishes and initial contract with AcmeNews-Antartida with the following authorization certificate:

$$(Alice, AcmeNews-Antartida, auth_{10}, p = true, V_1) \tag{9}$$

Where $auth_{10} = (acmeID, 10, h^{10}(m))$. This certificate is denoted as a *token-contract-cert* or *token contract certificate*. It allows Alice to establish against AcmeNews-Antartida that she is in the position of spending 10 tokens acme. The agency AcmeNews-Antartida can verify such a claim by means of the certificates (7) and (9) by using the SPKI/SDSI decision engine.

Once, the *token-contract-cert* has been issued, Alice can begin to spend tokens to access the resources of AcmeNews-Antartida. For example, Alice can send the value $auth_9 = (acmeID, 9, h^9(m))$ and $auth_8 = (acmeID, 8, h^8(m))$. AcmeNews-Antartida can verify that $h(h^9(m)) = h^{10}(m)$ and that $h(h^8(m)) = h^9(m)$, and together with the certificates (7) and (9) can allow the requested access to Alice. By making the elements of the chain public, Alice has implicitly delegated the tokens to AcmeNews-Antartida based on the initial contract *token-contract-cert* (9).

[1]Also known as emperor penguin.

A. Delegation of token-contract-cert

Imagine now, that AcmeNews-Antartida decides to transfer the initial contract with Alice, *token-contract-cert* (9) to AcmeNews-Zoology because Alice needs to access some records of the zoology department. To do that AcmeNews-Antartida, issues a new *token-contract-cert* to AcmeNews-Zoology with the last token that it has received from Alice:

$$(AcmeNews - Antardtida, AcmeNewsZoology, auth_8, \\ p = true, V_2) \tag{10}$$

AcmeNews-Zoology can verify together with the *token-contract-cert* (9) and the *chain-contract-cert* (7), that Alice is properly authorized to spend 7 tokens acme. Now Alice issues the next token $auth_7$, which is accepted by AcmeNews-Zoology, who can make all the pertinent verifications.

By means of the *token-contract-cert* (10), AcmeNews-Antartida has been able to delegate part of its initial contract with Alice to AcmeNews-Zoology. Normally this delegation can be transparent to Alice, allowing AcmeNews to easily subcontract services.

Figure 2 shows both *token-contract-cert*s issued, and the tokens delivered by Alice.

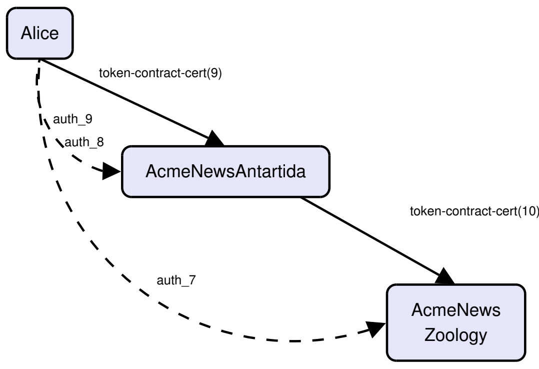

Fig. 2. *token-contract-cert* delegation example.

B. Delegation of chain-contract-cert

There is another contract delegation possible, the delegation of the *chain-contract-cert* by Alice. Imagine that Alice decides to transfer (delegate) the rest of the tokens to her colleague Bob, so he can use them. To do that, she just has to delegate the *chain-contract-cert* (7) that she received directly from AcmeNews by issuing the following certificate:

$$(Alice, Bob, auth7, p = true, V_3) \tag{11}$$

It authorizes Bob to spend the remaining 6 tokens from the chain. It is important to note that Alice has to let Bob know somehow the hash chain (8) or the initial seed *m*. This value could, for example, be encrypted in the comment field of the certificate.

Now Bob can issue the value $auth_6$. This token will be accepted by AcmeNews-Zoology, who can verify that Bob is authorized to issue it due to the *chain-contract-cert* (7) and (11). Figure 3 shows the *chain-contract-cert* and the tokens issued by Alice and Bob.

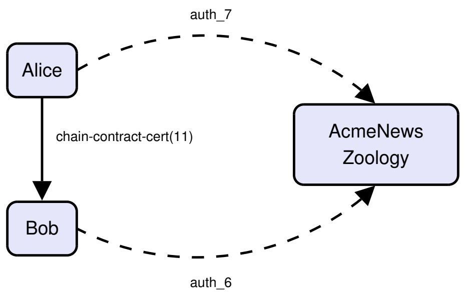

Fig. 3. *chain-contract-cert* delegation example.

C. Comments on delegation

In the previous section we have introduced both the *chain-contract-cert* and the *token-contract-cert*. Although both certificates look very similar, their differentiation is a key concept determined by the use of the contract. Summarizing:

- *chain-contract-cert*: allows to delegate the hash chain o a subchain to a principal. The principal receiving the contract will be the one using the tokens to access or use a given service. An special case is the first certificate, which establishes the trust that the authority (AcmeNews) places in a given principal (Alice).
- *token-contract-cert*: this certificate also allows to delegate the hash chain or a subchain, to a principal. But in this case, the principal receiving the contract is the consumer of the tokens. This situation is specially interesting because allows the outsourcing of services from one organization to another in a transparent way.

Note that each contract certificate has an specific validity specification, which allows for a fine-grained control of the delegations.

As we have seen, the ability of delegating the permissions as hash chains (or subchains) in two different ways introduces one of the key issues of CADAT. As another way to view the two types of delegation supported by the system, we can say that there is a client-side delegation (*chain-contract-cert*) and a server-side delegation (*token-contract-cert*).

Although we have used SPKI/SDSI as the underlying infrastructure, a similar approach could be used with other trust management technologies such as KeyNote. And in general, in other infrastructures supporting delegation, for instance the last X.509 specification describes extensions to support the delegation of attributes.

There may be the possibility of double spending. That is, a user that uses a token more than once. In order to avoid it, we consider that the principal making the original contract keeps track of the tokens used by all its contractors. In the above example, AcmeNews is the authority granting the tokens to Alice, we can say that AcmeNews is the authority for *acme tokens*. In order to accept tokens AcmeNews-Antartida and AcmeNews-Zoology, check the corresponding contract certificates. They will only accept acme tokens if the authorization root for those tokens is AcmeNews. The system should provide the ability for AcmeNews to keep track of spent tokens, for example, by means of online notifications.

VI. Application of CADAT

CADAT attempts to be a generic system, which can be applied in several scenarios. Given its nature, it can be easily used as a micropayment system, introducing the ability to delegate payment tokens. In [8], the authors describe a micropayment system implemented in KeyNote that makes use of what we call delegation of *token-contract-cert*, but does not uses the *chain-contract-cert* delegation. CADAT can be seen as an extension of this system.

But the CADAT system can also be used as a generic token based system. One of the current implementations of CADAT is as a token-based access control in a secure mobile agent platform. Access control in mobile agent systems involves lots of security problems. One of them is the fact that if mobile agents are able to perform digital signatures, either their private key is visible to the underlying platform, or the agent has to use a third party cryptographic proxy.

There are some alternatives as [13], which allows to delegate authorizations to a hash of the agent's code, but presents some constraints in the system. We have implemented CADAT on top of a secure mobile agent system, which at the same time makes use of the JADE (Java Agent Development Framework) framework[14], which supports the main FIPA (Foundation for Intelligent Agents)[15] specifications. JADE is a popular open source generic agent platform developed in Java, which lacks support for mobility and security. The implementation is done within the MARISM-A project, which attempts to bring mobility and several security solutions to the JADE platform. The mobility implemented in top of JADE is described in [16], and some existing security solutions in [17]. In the context of MARISM-A, CADAT provides a token based authorization mechanism specially interesting for mobile agents.

Figure 4 outlines the scheme used in the mobile agent application. A user Alice establishes a *chain-contract-cert* with a token authority, which will allow her to generate the specified number of tokens for her mobile agents. In order to access a resource in a given agent platform, Alice establishes a *token-contract-cert* with the platform. Tokens are made public by Alice when one of its agents has to access the platform resource. The platform verifies all the contracts and the tokens published by Alice to allow the agent to access the requested service or resource.

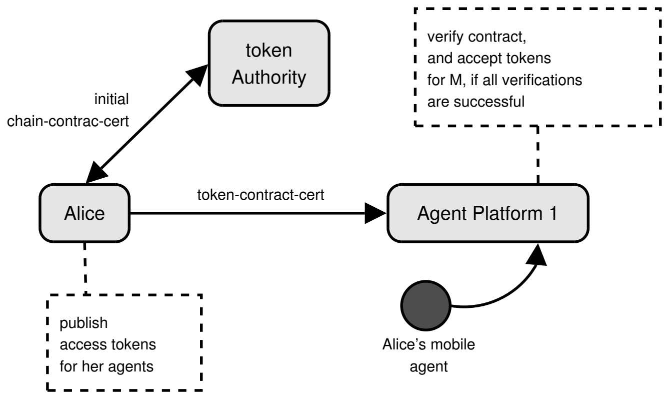
token Authority
verify contract, and accept tokens for M, if all verifications are successful
initial chain-contrac-cert
Alice
token-contract-cert
Agent Platform 1
publish access tokens for her agents
Alice's mobile agent

Fig. 4. CADAT in mobile agent applications.

In this case, the initial seed of the hash chain includes a hash of the agent code, which is used by the agent platform to authenticate the mobile agent. When the agent accesses the platform's resources, it does not have to perform any kind of cryptographic operation. The agent itself, does not need to carry any kind of sensitive information such as cryptographic keys or even the access tokens.

CADAT provides an extension of more general authorization frameworks such as [13] for mobile agent applications. It is specially suited for open environments, where authorizations are granted by a previous agreement with some application authority. It avoids common problems such as the maintenance of complex policies and certificate based infrastructures. The use of delegation also provides a lot of flexibility. An agent platform can outsource services or resources from other platforms by delegating the *token-contract-cert*, and the user can transfer part of the tokens to other users by delegating the *chain-contract-cert*.

VII. Conclusions and further work

In this paper we have presented the ideas behind the CADAT system. We have described its main base and it use and general functionality. The main idea behind CADAT is the use of elements of a hash chain as authorization tokens, the possibility to delegate those tokens in different ways.

One of the applications of CADAT is its use as a token based access control system in a secure mobile agent platform. Mobile agents do not need to carry any kind of sensible information as cryptographic keys. Furthermore the agents do not even have to perform costly cryptographic operations. The delegation introduced in CADAT allows a user to delegate access tokens to other users, and platform agencies (or servers in general) to outsource services and resources to other platforms (or server) in a way that is transparent to the user.

We are currently working on the improvement of the prototype implementation of CADAT in the mobile agent application. In relation to SPKI/SDSI, we have not discussed the implications of the use of threshold certificates and name certificates, which could add extra value to the system. We are also considering other possible implementations of CADAT such as a generic micropayment system for web services.

Acknowledgment

This work has been partially funded by the Spanish Government Commission CICYT, through its grants TIC2003-02041 and TIC2001-5108-E, and the Catalan Government Department DURSI, with its grant 2001SGR-219.

References

[1] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis, "The KeyNote Trust Management System," RFC 2704, IETF, Sept. 1999.
[2] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "SPKI Certificate Theory," RFC 2693, IETF, September 1999.
[3] B. C. Neuman and T. Ts'o, "Kerberos: An authentication service for computer networks," *IEEE Communications*, pp. 33–38, 1994.
[4] M. Lesk, "Micropayments: An Idea Whose Time Has Passed Twice?" *IEEE Security & Privacy*, January/February 2004.
[5] R. Anderson, H. Manifavas, and C. Shutherland, "Netcard - a practical electronic cash system." in *Cambridge Workshop on Security Protocols*, 1995.

[6] T. P. Pedersen, “Electronic payments of small amounts,” in *Proc. 4th International Security Protocols Conference*, 1996, pp. 59–68.

[7] R. L. Rivest and A. Shamir, “PayWord and MicroMint: Two simple micropayment schemes,” in *Proc. 4th International Security Protocols Conference*, 1996, pp. 69–87.

[8] S. Foley, “Using trust management to support transferable hash-based micropayments,” in *Financial Cryptography 2003*, 2003, pp. 1–14.

[9] S. Foley and T. B. Quillinan, “Using trust management to support micropayments,” in *Annual Conference on Information Technology and Telecommunications*, Oct. 2002.

[10] D. Clarke, J. Elien, C. Ellison, M. Fredette, A. Morcos, and R. Rivest, “Certificate chain discovery in SPKI/SDSI,” *Journal of Computer Security*, vol. 9, no. 4, pp. 285–322, Jan. 2001.

[11] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, “Simple Public Key Certificate,” Internet Draft, July 2000.

[12] “JSDSI: A Java implementation of the SPKI/SDSI standard,” `http://jsdsi.sourceforge.net`.

[13] G. Navarro, S. Robles, and J. Borrell, “Role-based access control for e-commerce sea-of-data applications,” in *Information Security Conference 2002*, ser. Lecture Notes in Computer Science, vol. 2433. Springer Verlag, September/October 2002, pp. 102–116.

[14] “JADE, Java Agent DEvelopemnt Framework,” `http://jade.tilab.com`, telecom Italia Lab.

[15] “Foundation for Intelligent Physical Agents,” `http://www.fipa.org`, fIPA Specifications.

[16] J. Ametller, S. Robles, and J. Borrell, “Agent Migration over FIPA ACL Messages,” in *Mobile Agents for Telecommunication Applications (MATA)*, ser. Lecture Notes in Computer Science, vol. 2881. Springer Verlag, October 2003, pp. 210–219.

[17] J. Ametller, S. Robles, and J. Ortega-Ruiz, “Self-protected mobile agents,” in *3rd International Conference on Autonomous Agents and Multi Agents Systems*. ACM Press, 2004.