Internet Credential Acceptance Policies

Kent E. Seamons and **William Winsborough**
Research Group
Transarc Corporation
707 Grant Street, Pittsburgh, PA 15219 U.S.A.
{seamons,winsboro}@transarc.com

Marianne Winslett
Database Research Laboratory
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave., Urbana, IL 61801 U.S.A.
winslett@cs.uiuc.edu

Abstract

Servers often need a basis for establishing some degree of trust in their clients. This problem is particularly interesting for services that do not have a prior relationship with their clients. Credentials can be used to demonstrate properties of the credential bearer possibly including, but not limited to, the owner's identity. Digital credentials are well-suited for use by any service provider that needs to grant different authorization for and views of services to different clients, based on client properties. This paper presents a programming methodology for writing credential acceptance policies and authentication logic that can be used for credential management on the internet.

1 Introduction

Many service providers need to grant different authorization for and views of services to different clients, based on client properties. A digital credential is an abstract data structure representing verifiable and unforgeable properties of the bearer, such as identity, age, address, citizenship, level of education, employment status, or professional certification. Ideally, a digital credential supports divulging a proper subset of the properties.

Service providers who rely on credentials will need to write *credential acceptance policies* (CAPs) that define which credentials must be presented to obtain services. This problem is particularly interesting for services that do not have a prior relationship with their clients. Our approach is unique in supporting the credential owner in obtaining services other than the ones for which they were issued, much as today we use a driver's license or a credit card to cash a check. This is essential for services that do not have a prior relationship with their clients.

We envision a future in which clients have diverse collections of credentials. Service providers -- especially services that have loose, informal relationships with their clients -- will want to accept credentials that were originally issued for different purposes. Indeed, this will enable new services on the web that require knowing certain client properties, but that do not justify a formal relationship with the client. This future is not far off.

Today, digital certificates are issued by several corporations. Originally, they just demonstrated the owner's identity. However, increasingly (e.g. X.509 version 3) certificates allow more general properties to be associated. The Secure Socket Layer (SSL) protocol [5] uses certificates to identify the communicating entities to one another. SSLv2.0 transmits a single certificate; SSLv3.0 will transmit a chain of certificates, supporting intricate chains of trust and more complex properties. Browsers present submission-policy choices to the client, although without any information about whether the selected certificates would lead to service authorization.

Digital credentials could be implemented by using X.509 version 3 digital certificates. However, we do not make that assumption in this paper. While we do assume that public key encryption is used, we avoid the details of credential format and simply provide hooks in our CAPs for authentication. Instead, we focus on management issues, in particular, providing support for the client's selection of credentials to attach to a request for service. As mentioned, this problem is particularly difficult for clients accessing services with which they have no established relationship.

The best solution to the problem of unpredictable service authorization has the server (or its agent) provide a credential acceptance policy to the client side, enabling the client (or its agent) to reason locally about what combination of credentials would be acceptable. This arrangement solves several concerns that arise when services accept credentials issued by others and clients have many credentials to choose from:

- Privacy: It is not reasonable to expect the client to submit all its credentials without regard to their relevance.
- Complexity: The client's pool of credentials may be large and the policy may require that several credentials be submitted together. (For instance, the server may want to verify that the client is a student at a university that is accredited by a major accrediting institution and also a member of the National Collegiate Athletic Association.) It is not reasonable to conduct the search for acceptable credential combinations by trial and error across the network.
- Diversity: Different servers will require that entirely different properties be demonstrated and will trust different authorities. It is impractical to anticipate at the client side what credentials the server might accept.

Our contribution is a programming methodology for writing credential acceptance policies that can be downloaded from the server to the client. The principal requirements of such policies are as follows.

- The policy must enable the client to reason reliably about what credentials would be accepted by the server. That means the policy that is executed at the client side must be logically equivalent to the one executed at the server side.
- On the client side, the policy must receive and search the client's entire collection of credentials for combinations that would be satisfactory and report those combinations so that the client can select which one to attach to the service request.
- That search is not emphasized at the server side. Rather, the policy should receive the subset of the client's credentials that the client attached to the request and ensure that there is at least one way to satisfy the policy by using that subset. It is not important which credentials were used or how many different ways the policy could be satisfied.
- Our policies support different credential requirements for different roles [10] that might entitle the client to different services, possibly with different views. On the client's side, the policy should output each role that the client could be authorized for, along with the combination of credentials to submit when requesting that role. On the server side, the role should be taken as input, so as to

give the client control over its selection.

Our methodology uses logic programming for a number of reasons, which we discuss presently. Much of the body of the paper is devoted to illustrating how the requirements of CAPs can be met by using Prolog constructs.

The most obvious reason that logic programming is a natural choice for writing CAPs is that a CAP is essentially a collection of logical relationships combined with conjunction and disjunction. There is a perfect match between the declarative reading of the logic program and the policy requirements. This means we obtain the much lauded benefits of declarative programming, including readability, writability, maintainability. Several other, more gritty reasons also apply.

The same CAP implementation can be executed at the client side as at the server side, even though the requirements are somewhat different. This is important because it minimizes the risk that credentials would appear acceptable at the client side, but be rejected at the server side. It is non-trivial because the requirements are different. At the server side, it suffices to find one way of satisfying the policy by using the submitted credentials. Also, the role that the client wants is specified. In contrast, at the client side, the policy should search the client's credentials, finding several alternative combinations of credentials that satisfy the policy and the roles that they would authorize. These alternatives are needed to present the client with a well-structured choice about what credentials to submit. (''Well-structured'' means that the combinations would lead to authorization and do not include unneeded credentials.)

Prolog language features support using the same CAP implementation in these two ways. The bidirectionality of unification enables the same policy code to handle the case where the desired role is specified (server side) and the case where it is not (client side). The Prolog `setof`/3 meta-predicate enables the same policy code to be used on the client side to find all possible roles and supporting credentials and on the server side to determine whether the credentials submitted by the client authorize the role it has requested.

Acceptance policies use operations that are characteristic of logic programming. Pattern matching makes credential recognition and field extraction trivial. Prolog backtracking naturally supports the search for combinations of credentials that demonstrate complex client properties and that connect the client to a known certificate authority.

Further supporting considerations include the fact that Prolog is (often) an interpreted language, which supports the portability required by the application. The high-level nature of logic programming would make it relatively easy to identify and close potential security holes opened by downloading code from unknown servers. Because policies need to make only extremely limited use of meta-predicates and extra-logical predicates, it would be straight forward to write a CAP sanitizer that verifies that the CAPs do not perform illegal operations, like writing to disk.

The paper is organized as follows. Section 2 describes how credentials can be made verifiable and unforgeable. Section 3 presents an architecture for managing digital credentials. Section 4 explains how credentials are authenticated by establishing a chain of trust back to a trusted certification authority. Section 5 discusses generally how credential acceptance policies can be written as logic programs. Section 6 illustrates our programming methodology. Section 7 discusses related work. Section 8 outlines future work.

2 Verifiable, Unforgeable Digital Credentials

To be effective, credentials must be unforgeable and verifiable. Paper credentials incorporate signatures, official seals, holograms, and special materials in an attempt to make the cost of forging the credential prohibitive. Verifiability implies that it is possible to confirm that the person presenting a digital or paper credential is the person to whom it was issued. The owner can divulge a verifiable credential without fear of someone copying it and successfully using it with a diligent service provider. For instance, using paper credentials, an individual might present their driver's license to prove their identity. The recipient verifies the driver's license by comparing the photo on it with the individual's appearance. Digital credentials achieve unforgeability and verifiability more reliably than paper ones by utilizing public key encryption, which encrypts and decrypts by using a pair of keys, one of which is kept private by the key owner and one of which is made public. Cipher text that has been encrypted with one key can be decrypted successfully only by using the other key. Conversely, cipher text that has been successfully decrypted by using one key could only have been encrypted by using the other key.

The key owner can sign any text by using the private key to encrypt it. By having a credential issuer use her key to sign a credential, that credential is made unforgeable. Some part of the contents of the credential are included as clear text as well as encrypted text. Doing so facilitates credential management and enables one to confirm that decryption with the public key yields the original clear text.

Public key encryption supports verifiability as follows. Credentials can identify their owner by providing a public key for which the owner can demonstrate possession of the private-key mate. The ownership of the credential can be authenticated by challenging the bearer to use that private key to encrypt arbitrary, server-specified text. (Below, we often write simply that the credential contains a key that the client must demonstrate is in her possession. Expressing it this way blurs the distinction between the public and private keys, but greatly simplifies the text.)

For the server to be able to trust a credential, the server must know the public key corresponding to the private key used to sign it, either directly, as the public key belonging to a trusted certification authority, or indirectly, through a chain of credentials that leads back to a trusted *certification authority* (CA). Such a chain establishes a sequence of known credential issuers and the keys they each used to sign the next credential in the chain, in this way establishing a chain of trust -- or at least culpability. Thus, a client wishing to obtain services may often be required to attach several credentials to the request. This paper is about supporting the client and the server in managing those credentials and, in particular, reasoning about which credentials should accompany service requests.

The technological components needed for deployment of a digital credential infrastructure exist today. For instance Verisign, Inc. [12] issues X.509 certificates [6] and RSA Laboratories issues PKCS certificates [9]. X.509 version 3 certificates allow the issuer to attest to owner properties other than identity. It is possible to implement a digital credential with several separate certificates, each attesting to different properties. Such *modular credentials* allow the owner to control which information is released to each server by withholding some of the certificates that are included in the credential. Note, however, that the examples shown in this paper do not use modular credentials.

3 Managing Digital Credentials

Figure 1 illustrates an architecture to manage digital credentials based on existing web technology. A *personal security agent* (PSA) manages the client's credentials. The PSA examines each request and determines what credentials, if any, should be included with the request. A *server security agent* (SSA) is the (generic) front end of any server that uses credentials to determine whether and how to satisfy service requests.

In our demo, the PSA is implemented as a proxy server and the SSA is implemented as a CGI script. Other alternative implementations for the PSA would include making it part of the browser or, better, putting it on a smart card [8]. Making the PSA a part of the browser might support a nicer user interface, however it offers the same degree of security as a proxy implementation running on the same machine as the browser. The smart card implementation would have significant advantages in security and mobility.

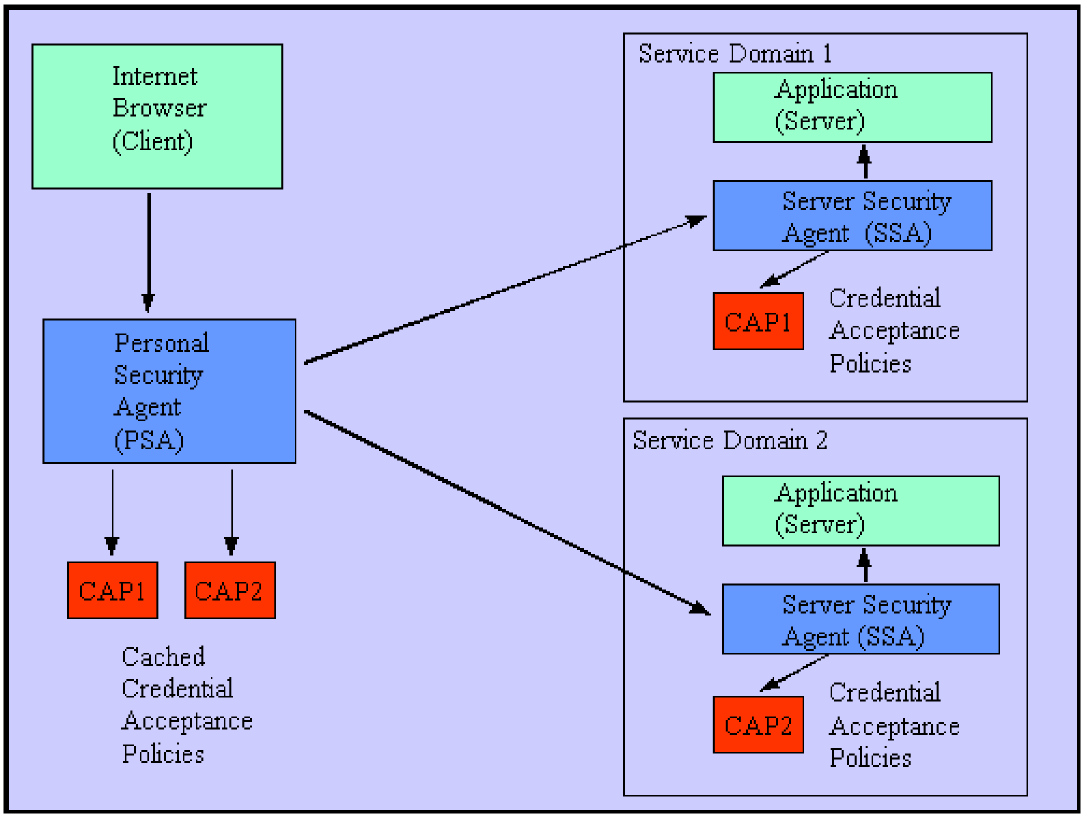

Figure 1. Architecture for management of digital credentials for use on the Web. The SSA has a credential acceptance policy (CAP) for each of the services it supports (the figure shows only one). The PSA caches policies from the various SSAs that its client has requested services from.

Each SSA has an associated credential-acceptance policy for each service the SSA assists. That policy dictates which combinations of credentials are acceptable to obtain the service. The SSA executes the policy, providing as input to it the credentials that the PSA has forwarded on the client's behalf. The PSA also uses the policy to determine which credentials it should send to the SSA. (The client does not want the PSA to divulge credentials needlessly. Nor does the service provider want the SSA to receive and have to reason about extraneous credentials from the PSA.) The PSA does not know a priori what the SSA's policy is. Rather, portions of the policy are transferred from the SSA to the PSA when the client first requests a service. The PSA then caches the policy for later use.

Each service may support clients interacting with it in several different roles. The credentials that a policy requires a client to present depend on the role she is requesting. The view of the service that she receives may depend on the client's role. Some services are available only to certain roles. For example, an electronic publishing service might offer different subscription rates to students than to purchasing agents of for-profit institutions.

Our concept of role differs from the one used, for instance, in [10]. We view a role as a category of client, rather than as being directly associated with service authorization. For services that have pre-existing relationships with their clients, authorization can be granted only when server-specific knowledge is added to the client properties captured by our concept of role. That server-specific knowledge typically should not be transmitted to the client. So when the client selects her role, she must do so without benefit of that knowledge. An example would be when a client must demonstrate her name, which the server will map to a pre-existing set of authorizations. In this case, the ‘‘role’’ that the client selects expresses the property of being the person with her name. In general, the role should express all properties of the client that the service needs to have the security mechanism check. At the abstract level, the role comprise all the information communicated from the CAP to the service. Successful execution of the CAP implies no further information, such as the authenticity of a particular submitted credential.

The PSA acts on behalf of the client, making much of the credential handling automated and seamless. When the client requests a service for the first time, the PSA will not attach credentials, knowing nothing about the credential acceptance policy, but will forward the request as is. Upon receiving the service request without attached credentials, the SSA will respond to the PSA, indicating that credentials are required and providing a copy of the credential acceptance policy. The PSA will then use the policy to determine what combinations of the client's credentials, if any, could be used to satisfy the policy. At the same time, the PSA finds out which collections of the client's credentials could be submitted to obtain authorization in each of those possible roles. The PSA returns a HTML form to the client browser, presenting those credential combinations and the corresponding roles that they would entitle the client to. The HTML form enables the client to choose a credential submission policy by selecting a role and a credential combination authorizing that role. The client indicates on the form whether the submission policy applies to this request, only, or should be cached at the PSA and applied to future requests, as well. This is how the client retains control over dissemination of her credentials. (There will also be a way to edit those choices later, when the client wishes.) The PSA will not be visible to the client when she requests services for which there is a cached submission policy.

Research project

The research reported here is part of an on-going project at Transarc Corporation (see http://www.transarc.com/~winsboro/DigitalCredentials/) to revise and extend the design and deployment of digital credential management tools that have been developed at the University of Illinois at Urbana-Champaign (see http://drl.cs.uiuc.edu/security/). We are designing and deploying a demo based on electronic, peer-reviewed publishing. The demo will include a server, a PSA, a SSA, and an acceptance policy. The PSA included in the demo will run at the server site, which would not be the case in a production deployment.

4 Authenticating Credentials and Verifying Ownership

This section describes how to construct chains of digital credentials that can be authenticated and their ownership, verified. Credential authentication is performed by using a trusted public key to successfully decrypt the cipher-text portion of the credential to obtain the clear-text portion. Often the key used is itself obtained from another credential that, like the first one, is presented by the client. Figuring out which keys have to be used to decrypt which credentials involves reasoning about many of the same issues as the acceptance policy must address. Once an acceptable collection of credentials has been identified by the policy, it is straightforward for the policy also to organize authentication of each credential with the appropriate key. Thus, our methodology organizes credential authentication to be invoked by the policy. (See Figure 2.) Also after an acceptable combination of credentials has been identified, the policy has easy access to the keys that the client must be challenged to demonstrate are in her possession. However, because they involve communication with the client, the challenges are handled by the SSA, by using client authentication routines, also pictured in Figure 2. The policy identifies the keys that must be challenged and passes them back to the SSA when it returns.

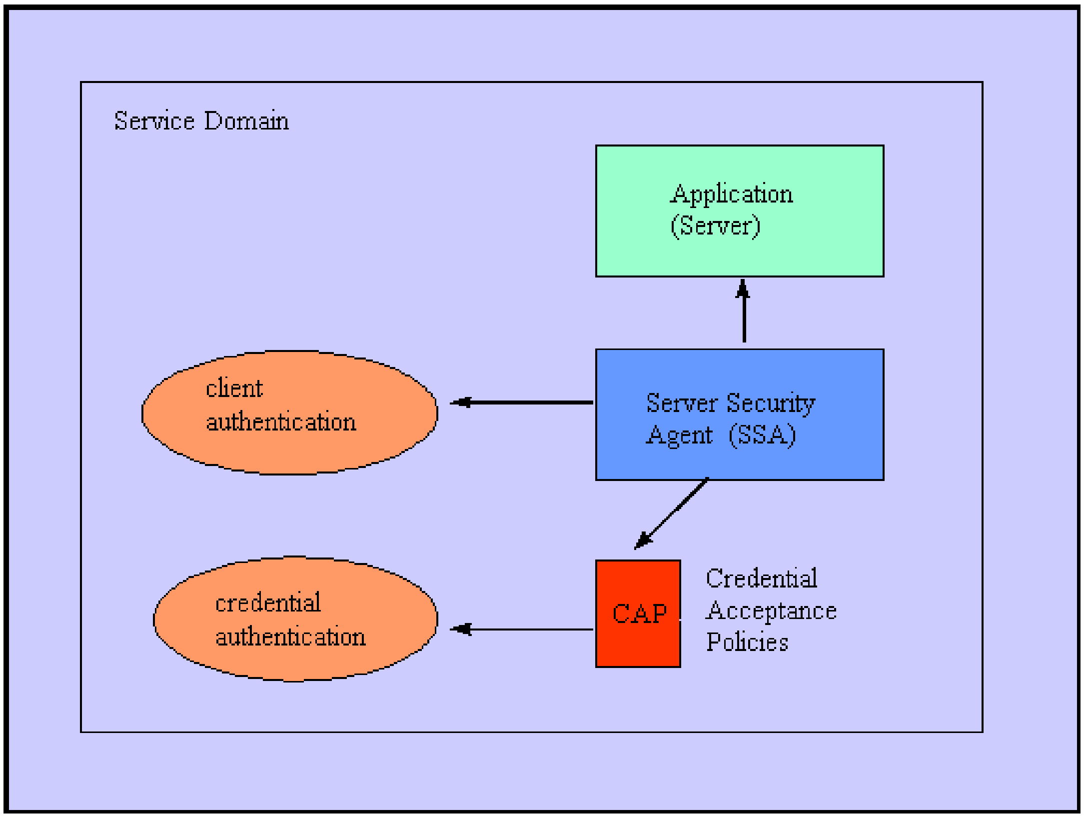

Figure 2. Enlargement of the server side of the credential management architecture, showing authentication functionality. Credential and client authentication are both controlled by the server acceptance policy. However, client authentication entails communication to challenge the client's keys. The SSA manages that communication.

Credential Chains: Authentication and Complex Properties
In order to authenticate a credential, such as a student ID, it is often necessary to have a supporting chain of other credentials. A student ID must be signed with the private key of the university, so authentication

requires knowing the university's public key. A separate credential may be required to authenticate that public key. For instance, this supporting credential could be signed by a certification authority and state that the university's public key belongs to the university. (Several corporations provide such a certification service today.) Assuming the SSA knows the certification authority's public key, the SSA can authenticate the ownership of the university's public key, which in turn can be used to authenticate the student ID.

It may be unrealistic to presume that the SSA's acceptance policy knows all the universities whose students it wants to accept. In the event that the SSA wants to accept a student ID from any accredited university, it would probably be inappropriate for that policy to include an enumeration of all the accredited universities. Better would be to enumerate the relatively few accrediting agencies and to require that the chain of credentials include a certification of accreditation for the university, signed with the private key of a known accrediting agency. A client who must demonstrate that she is a student at an accredited university must present a chain of three credentials, as illustrated by the following example credentials, shown here in a Prolog format that abstracts the details of their representation.

```
certifies( ca_all_knowing,
           belongs_to( acme_key, acme_accrediting_agency ),
           certification_signature ),
accredited_school( acme_accrediting_agency, wam_u, wam_key,
                   acme_accrediting_signature ),
student_id( wam_u, rocky_squirrel, student_key,
            16/8/1996, 15/5/1997, wam_signature )
```

In the example, a chain of credentials demonstrates that Rocky Squirrel is a student at WAM University during the period from 16 August 1996 to 15 May 1997. To authenticate the chain, the public key of the All Knowing certification authority must be known and must successfully decrypt the `certification_signature`, yielding `belongs_to( acme_key, acme_accrediting_agency )`. The key, `acme_key`, must successfully decrypt `acme_accrediting_signature`, yielding the data structure, `accredited_school( acme_accrediting_agency, wam_u, wam_key )`. The key `wam_key` must successfully decrypt `wam_signature`. Finally, the client must successfully answer challenges to verify that the client possesses the key, `student_key`. We further discuss client challenges presently.

Notice that the name of the university, `wam_u`, included in the accreditation credential, is not important for authenticating Rocky as a student at an accredited university. Instead, the university name aids the policy in its search for the credential that contains the key used to sign the student id. For authentication, the important information is the university's key, `wam_key`, which can be used to authenticate Rocky's student ID.

Rocky's student ID can be used to demonstrate that Rocky attends a school with many different properties. This is because the university's key, which was used to sign Rocky's ID can be included in several different credentials belonging to the university. Rocky could combine his student ID with one of these to show, for instance, that he attends a school that is in the National Collegiate Athletic Association. The university can have a large number of credentials all containing the key used to sign Rocky's student ID. This scenario implies that the university chooses which key the credential issuer includes in the credential. This arrangement is probably better than having the credential issuer construct the public and private keys, incorporate the public one in the credential and give the private one to the credential owner. The latter method gives the issuer both work and knowledge that the issuer should not have.

Client Authentication
If authentication of the credential chain starts by knowing the public key of a trusted certification authority, which can be used to authenticate the first credential in the chain, then it concludes by verifying that the client possesses the private-key mate of the public key included in last credential of the chain. The SSA can use that public key to verify that each terminal credential was issued to the client, and not to someone else. The SSA requires the client to respond to challenges that verify the client possesses the private key. These challenges ask the client to use the private key to encrypt certain SSA-specified text that will, upon decryption with the public key, enable the SSA to verify that the client making the request is the owner of the credential.

The challenge text might include, for instance, the service request, the return URL, and a unique number, such as the concatenation of the SSA's current time and a sequence number. The point is to make the client encrypt something that is extremely unlikely to have been encrypted before. Another important issue, albeit beyond the current scope, is to ensure that the communication with the client required to complete the challenge has the necessary security properties. For instance, the SSA must be sure that the challenge response is not hijacked and used by someone else to gain improper access. This would be a possible motive for including the initial service request in the challenge text. However, this and many other communication security problems can be solved by using a secure protocol, such as the secure socket layer (SSL -- supported by Netscape and Microsoft web browsers), which the service provider may also wish to require be used for the service itself, to keep the service contents private.

Delegation
Digital credentials support delegation. For example, an electronic publishing server might offer a discount subscription to accredited universities. The server will be contacted by an agent of the university to whom purchasing authority has been delegated. An example of the credentials that would be required to document this delegation are as follows.

```
certifies( ca_all_knowing,
           belongs_to( acme_key, acme_accrediting_agency ),
           certification_signature ),
accredited_school( acme_accrediting_agency, wam_u,
                   wam_key, acme_accrediting_signature ),
purchasing_agent( wam_u, bullwinkle, bullwinkles_key,
                  1/1/1997, 12/31/1997, wam_signature )
```

Here the university key is supported by the same chain of credentials as in the student ID case. The university key is then used to authenticate the appointment of Bullwinkle as a purchasing agent. From the policy's point of view, the important point is that Bullwinkle's key enables Bullwinkle to prove that the university has delegated purchasing authority to him.

5 Writing Policies as Logic Programs

Logic programming is a very natural paradigm for expressing the logical requirements of a credential acceptance policy. We are currently using XSB Prolog [11], available from The State University of New York at Stony Brook at http://www.cs.sunysb.edu/~sbprolog/xsb-page.html/The acceptance policy of a given server will be provided to the PSA by the SSA when the PSA contacts the SSA for the service.

Credentials are Treated as Data by the Policy
The CAP works with a collection of *available credentials* , which in the PSA are the credentials owned

by the client and in the SSA are the credentials submitted with a given request. The available credentials are passed into the policy as a list of logical terms. Although, it would also be feasible to include the credentials into the program as Prolog facts, this is not done, because of the relatively high cost of asserting dynamic code in Prolog, as well as the danger of opening a security hole through which a client might corrupt the program that is the acceptance policy.

Policy Interface

The Prolog predicate defining a service's credential acceptance policy is as follows. An input parameter provides a list of the credentials available to support the service request. A parameter that can be either input or output describes the role that the client will be authorized to have with the service. An output parameter returns a list of the input credentials used by the policy in authorizing the service. The PSA can use this list to determine which of the credentials that it possesses should be submitted to the SSA. The same list is also used to return keys that the SSA should challenge the PSA to demonstrate are in its possession.

Use of Definite Clause Grammars

Many of the internal predicates used in our policy methodology are defined using *definite clause grammars* (DCGs). A DCG is essentially a Prolog program in which each predicate includes two implicit parameters, customarily used to provide an input terminal string to be parsed. Both of the implicit parameters are used to represent the list as a difference list. The first parameter is bound on input to a string from which the clause (or production) recognizes a prefix and binds the second parameter to the remaining input string. We take advantage of the bidirectional nature of unification by using these implicit parameters not for input, but for output of the list of credentials used by the policy.

The reason we use the DCG form is to simplify predicate argument lists. In addition to needing to output the credentials used, the policy also needs to output a list of the keys that the SSA should challenge the client to demonstrate are in her possession. Thus far, we have included those challenge keys in the same output list as the credentials used, despite the mismatch in significance. This choice is again motivated by a desire to simplify predicate argument lists.

6 Programming Methodology: Example Code

This section uses a few excerpts from our electronic publishing demo to illustrate our methodology for writing CAPs by using Prolog. The credentials it works with are of the form presented in section 4.

In order to submit a new paper to our electronic publishing service, the server requires a client to prove what her name is. That information enables the service provider to reliably identify those who submit libelous or plagiarized materials. This section presents predicates that support checking credentials to verify the client's name. Let us emphasize that many other properties would be appropriate to check by using credentials. We assume that the server has no prior relationship with the client and must rely on credentials issued by others.

The predicate `submit`/5 is used only for determining authorization to the submission service.

```
submit( CredsProvided, SubmittingAuthor, CredsUsed ) :-
  submit_internal( CredsProvided, SubmittingAuthor, CredsUsed, [] ).
```

It takes the list of available credentials (all the client's credentials if executing under the PSA, the submitted credentials if executing under the SSA). The role is given by `SubmittingAuthor`. `CredsUsed` is an output list of credentials and keys that the client must be challenged to demonstrate are in her possession. The former are used at the PSA for assisting the client in selecting which credentials to attach. The latter are used at the SSA to authenticate the client as the credentials' owner. The predicate `submit_internal`/6 is used simply to provide the extra parameter needed for the difference list used by the DCG. In the definition of that predicate, the two last parameters are implicit:

```
submit_internal( CredsProvided, SubmittingAuthor ) -->
  check_name( CredsProvided, SubmittingAuthor ).
```

There are many ways of proving your name. Standard predicates can be used and reused to check various credentials documenting such properties of clients. Here is a DCG production defining one way to document the client's name. Other productions would be added to handle the other credentials that could be used for this.

```
check_name( Creds, OwnerName ) -->
  authenticate_credit_card( Creds, _CardCompany, _IssuingCompany,
                            OwnerName, _OwnerAddress, OwnerKey,
                            _CardNumber, IssueDate, ExpirationDate ),
  { get_date( CurrentDate ),
    date_in_range( IssueDate, CurrentDate, ExpirationDate )
  }.
```

The predicate `authenticate_credit_card`/11 (there are two implicit DCG parameters in the call shown) is used to find and authenticate a credit card among the clients credentials. It is a standard predicate that can be used in many different policies for many different purposes. In its use here, many of the arguments' values it provides are ignored, as indicated by variable names that begin with underscore. Simple CAPs can be written by using only stock predicates for widely-available credentials. The curly braces delimit calls to predicates that do not have the implicit DCG parameters. The code shown here checks dates to be sure that the credit card is current. However, this may or may not be important for using it simply to check the client's name.

The predicate `authenticate_credit_card`/11 can be defined according to a pattern that is standard across many credentials.

```
authenticate_credit_card( Creds, CardCompany, IssuingCompany,
                          OwnerName, OwnerAddress, OwnerKey,
                          CardNumber, IssueDate, ExpirationDate ) -->
  [challenge( OwnerKey )],
  credit_card( Creds, CardCompany, IssuingCompany,
               OwnerName, OwnerAddress, OwnerKey, CardNumber,
               IssueDate, ExpirationDate, CCSignature ),
  authenticate_credit_card_issuer( Creds, CardCompany,
                                   IssuingCompany, IssuerKey,
                                   IssuerAuthDate, IssuerExpDate ),
  authenticate_contents( CCSignature, IssuerKey,
                         credit_card( CardCompany, IssuingCompany,
                                      OwnerName, OwnerAddress,
                                      OwnerKey, CardNumber,
                                      IssueDate, ExpirationDate ) ),
  { date_in_range( IssuerAuthDate, IssueDate, IssuerExpDate )
  }.
```

This clause adds to the output list of keys to be challenged the key of the owner of the credit card. The predicate, credit_card/12 is used to find a credit card among the credentials in Creds. It adds that credential to the output list of credentials used. To authenticate the credit card it is necessary to find a credential belonging to a credit card issuer, like a bank, and issued by a credit card company, like Visa or Mastercard. The problem of finding and authenticating that credential is solved by authenticate_credit_card_issuer/8, which returns IssuerKey. IssuerKey is then used in the call to authenticate_contents/5 to authenticate the contents of the client's credit card. That call uses to issuer key to decrypt the contents of CCSignature and to verify that it does in fact agree with the cleartext values in the credit_card/10 data structure, also passed into the call. The call to authenticate_credit_card_issuer/8 also returns authorization and expiration dates for the credential of the credit card issuer, which are checked against the credit card issue date. The arguments CardCompany and IssuingCompany are include as input to the call to support searching Creds. The definition of authenticate_credit_card_issuer/8 would be as follows.

```
authenticate_credit_card_issuer( Creds, CardCompany,
                                 IssuingCompany, IssuerKey,
                                 IssuerAuthDate, IssuerExpDate ) -->
  credit_card_issuer( Creds, CardCompany, IssuingCompany, IssuerKey,
                      IssuerAuthDate,
                      IssuerExpDate, CCISignature ),
  authenticate_credit_card_company( Creds, CardCompanyKey,
                                    CardCompany ),
  authenticate_contents( CCISignature, CardCompanyKey,
                         credit_card_issuer( CardCompany,
                                             IssuingCompany,
                                             IssuerAuthDate,
                                             IssuerExpDate ) ).
```

As with credit_card/12, credit_card_issuer/8 performs the search for the credit card issuer credential and presents its contents. authenticate_credit_card_company/5 handles the subproblem of finding and authenticating proof that CardCompanyKey belongs to the credit card company. That key is then used to authenticate the credit card issuer credential. Finally, authenticate_credit_card_company/5 can be implemented as follows.

```
authenticate_credit_card_company( Creds, CardCompanyKey, CardCompany ) -->
  { known_credit_card_company( CardCompany ) },
  certified( Creds, belongs_to( CardCompanyKey, CardCompany ) ).
```

This predicate must complete the chain of trust. Here, known_credit_card_company/1 holds for credit card companies recognized by the service. The call to certified/5 finds and authenticates a credential issued by a trusted certification authority stating that the key given by CardCompanyKey belongs to the credit card company, CardCompany.

A major credit card company, like Visa or Mastercard, and a trusted certification authority, like Verisign, would be a typical root for a chain of trust -- at least in the near future. However, as credentials proliferate, the roots of these chains will depend entirely on the service provider, reflecting increasingly diverse kinds of social connections.

The principle in writing predicates that find and authenticate a credential is that they are responsible for tracing a chain of credentials back to a trusted credential issuer whose key the SSA knows. These predicates return the key contained in the start credential, like the owner key of the credit card in authenticate_credit_card/11, which can be used either to authenticate another credential, or to establish a property of the client, provided she can answer challenges to that key. We believe that this strategy, augmented with suitable policy-writer support tools and a library of predefined authentication predicates, will make the task of writing policies manageable.

Note that credential-checking predicates are specific to the property of the key owner that they are intended to prove. For instance a different predicate would be needed to check a student ID when the desired property is that the student attends an accredited university than when the property is that she attends a university in the NCAA.

7 Related Work

The current work extends research by Winslett et al. at the University of Illinois at Urbana-Champaign [4,7,13]. A demo and several papers from that initial work are available at http://drl.cs.uiuc.edu/security/. The demo uses a policy written as a Prolog program, albeit a rudimentary one. For instance, credentials are loaded into the interpreter as facts, making them part of the program, rather than data. All data structures are constants, making the policy a Datalog program. This means that there is no support for determining which credentials were used to satisfy the policy.

Blaze, et al., [1] describe a prototype system, PolicyMaker, that accepts policies, credentials, and a request as input and determines whether the proposed request is authorized. They have designed a new high-level, interpreted language in which credentials, policies, and trust relationships can be specified. Like our system, theirs is targeted for credentials that describe properties beyond simple identity. They do not address the problem of selecting combinations of credentials, therefore they provide the client no assistance in selecting which credentials to submit to gain authorization. We see the thrust of their work as complementary to ours: they have made policies easier to write and we have made the client's credential submission choices manageable.

A credentialing scheme with many of the properties that we desire has been proposed by Chaum and associates [2,3] (distinct from their proposals for digital cash). The most interesting feature of these credentials is that they allow *complete* anonymity on the part of the presenter. For example, suppose the client has one Chaum-style credential issued by the state driver's license facility, and a Chaum-style student ID issued by a local university. Under Chaum's approach, the client can replace the identity referred to in those credentials by a new pseudonym chosen just for the current transaction. The recipient can verify that the credentials were indeed issued by the purported issuer and do belong to the identity presenting them, but cannot trace them back to the identity used at the issuing source. From our point of view, the drawbacks of Chaum's scheme are that, first, the cryptographic magic used to provide these guarantees is very expensive, thus making the credentials quite costly to deploy. Also, his scheme as proposed does not scale well, being limited to a predetermined number of credential types and thus not suited for wide-scale deployment. Third, many of the servers that would accept our kind of credentials may be uncomfortable with complete untraceability of credentials. Our proposed scheme allows traceability if multiple information sources collude to share what they know about the different identities possessed by individuals. More exploration of the tradeoffs between different kinds of credentialing schemes can be found in [7].

8 Future Work

To be successful, digital credentials must be easy to use. There is a danger of bewildering the client when a service request is denied due to inadequate credentials. A gap in the current design that we hope to fill in a future project is the need for the PSA to explain to the client what credentials, if added, would enable acceptance. We expect that this is functionality for which logic programming will be well suited. By adding clauses at the bottom of predicate definitions, it should be possible to trap policy failures, to generate diagnostics, and to reason about combining those messages into a coherent explanation of policy components that cannot be satisfied by using the client's current set of credentials.

To manage credentials effectively, the PSA should be able to help the client find credentials that authenticate or augment her credentials. Credentials should include information that supports this. For instance, while it might be reasonable to issue each student a copy of a credential certifying that the key that decrypts their student ID belongs to the particular university, it probably would not be reasonable for each student to keep a complete copy of every credential that characterizes the university, such as its accreditation. Instead, it should be clear where the student can obtain such information. Each student ID issued should include a URL at which supporting university credentials could be found.

To enable SSAs to be deployed easily, it will be necessary to construct tools to assist in policy writing. Service providers could not all be expected to develop the Prolog writing skills that are required to produce sound policies.

Acknowledgements

We thank Donald Beaver at Transarc for enlightening discussions on security and encryption technologies. We are also grateful to the reviewers, whose extensive comments and suggestions enabled us to improve the paper significantly.
This research was supported by ARPA and US Army CERL under DACA-88-94-0029.

References

1. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. IEEE Symposium on Security and Privacy, Oakland CA, May 1996.
2. D. Chaum. Showing credentials without identification: Transferring signatures between unconditionally unlinkable pseudonyms. In *Auscrypt '90*, pages 246--264, Berlin, 1990. Springer-Verlag.
3. D. Chaum and J.-H. Evertse. A secure and privacy-protecting protocol for transmitting personal information between organizations. In *Advances in Cryptology--CRYPTO '86 Proceedings*, pages 118-167, Berlin, 1986. Springer-Verlag.
4. N. Ching, V. Jones, and M. Winslett. Authorization in the digital library: Secure access to services across enterprise boundaries. In *Proceedings of ADL '96 -- Forum on Research and Technology Advances in Digital Libraries*, May 1996.
5. A. O. Frier, P. Karlton, and P. C. Kocher. The SSL protocol. Submitted to the Internet Engineering Task Force as a draft standard, and available on line at ftp://ietf.cnri.reston.va.us/internet-drafts/draft-freier-ssl-version3-01.txt
6. International Telephone and Telegraph Consultative Committee (CCITT). CCITT recommendation X.509: The directory-authentication framework, November 1988.
7. V. Jones, N. Ching, and M. Winslett. Credentials for privacy and interoperation. In *Proceedings of the Workshop on New Security Paradigms*, pages 92--100. IEEE Computer Society Press, 1995.
8. M. Krajewski, J. C. Chipchak, D. A. Chodorow, and J. T. Trostle. Applicability of smart cards to network user authentication. Computing Systmes, 7(1):75-89, Winter 1994

9. RSA Laboratories. PKCS #6: Extended-certificate syntax standard, technical note version 1.5. Available on line at ftp://ftp.rsa.com/pub/pkcs/ascii/pkcs-6.asc, Nov. 1993.
10. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. IEEE Computer, February, 1996.
11. K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, H. Davulcu, S. Dawson. The XSB programmer's manual, Version 1.7. http://www.cs.sunysb.edu/~sbprolog/manual/manual.html, March 1997.
12. Verisign, 1390 Shorebird Way, Mountain View, CA, 94043. http://www.verisign.com/.
13. M. Winslett, N. Ching, V. Jones and I. Slepchin. Assuring security and privacy for digital library transactions on the web: client and server security policies. Proceedings of ADL '97 --- Forum on Research and Technology Advances in Digital Libraries, Washington, DC, May 1997.