Transparent Internet E-mail Security

Raph Levien Lewis McCarthy Matt Blaze
AT&T Laboratories
Murray Hill, NJ 07974

DRAFT – August 9, 1996 – DRAFT

Abstract

This paper describes the design and prototype implementation of a comprehensive system for securing Internet e-mail transparently, so that the only user intervention required is the initial setup and specification of a trust policy. Our system uses the PolicyMaker trust management engine for evaluating the trustworthiness of keys, in particular whether the given binding between key and name is valid. In this approach, user policies and credentials are written as predicates in a safe programming language. These predicates can examine the graph of trust relationships among all the credentials presented. Thus, credentials can express higher-order policies that depend upon global properties of the trust graph or that impose specific conditions under which keys are considered trusted. "Standard" certificates, such as PGP and X.509, are automatically translated into simple PolicyMaker credentials that indicate that the certifier trusts a binding between a key and a name and address, and certifiers can also issue more sophisticated credentials written directly in the PolicyMaker language.

Our system does not assume any particular public key, certificate, or message format. Our prototype implementation, which runs under most versions of Unix, accepts PGP key certificates as well as our own credentials, and uses standard PGP message formats. Thus, our system interoperates with the existing infrastructure of secure e-mail applications while providing additional flexibility at those sites where the system is used. We plan also to support S/MIME and other message formats, X.509 certificates, and Win32-based platforms.

1 Introduction

This paper discusses the design and implementation of what seems to be the most straightforward application of public-key cryptography possible: transparently securing electronic mail messages on the Internet. All secure e-mail implementations aim for user transparency and flexibility, but the architecture of existing tools and standards, such as PEM, PGP, or S/MIME, forces the designer to sacrifice or compromise at least one of these goals. Very flexible systems, such as those that implement PGP-style "web-of-trust," require special user intervention for each new correspondant and special mail user-interface software, while completely transparent systems are typically limited to a well-defined, hierarchical model of trust supported by a widely-deployed, highly-available, trusted infrastructure of certification authorities.

Part of the reason for this unsatisfactory state of affairs, we argue, is that current secure e-mail systems lack a flexible mechanism for the specification of *trust* and the ability to evaluate credentials issued under many different trust assumptions. Previous work in automatic key distribution has

generally been restricted to a single-rooted trust hierarchy, which does not scale well to the Internet. Conversely, previous work in decentralized key distribution has not fully automated the process, requiring a certain degree of user involvement for each new correspondent. General e-mail on the scale of the Internet requires both flexibility and transparency, and yet these two goals seem in fundamental conflict with one another. Internet e-mail, it appears, is a harder problem than one might first expect.

1.1 The "e-mail security layer"

The standard model for e-mail systems is a stack of layers. The top layer is the mail user agent (MUA), which presents a user interface, but does not handle the actual delivery of the mail. The Unix world provides a variety of MUA programs (/BIN/MAIL, MH, ELM, etc.). Mail delivery is the function of the mail transport agent (MTA), which is generally responsible for the routing and queuing of messages. The Unix SENDMAIL program is the most common example of an MTA. In turn, a mail-delivery network protocol such as SMTP is the layer below the MTA. This protocol is layered on top of network protocols such as TCP and IP.

For unencrypted e-mail, this model works well, but adding security poses a fundamental question: where in this model does the cryptography belong? Most secure e-mail systems simply add the cryptography to the MUA, generally as a feature to be selected from the user interface.

This approach does not facilitate the goals of transparent secure e-mail. Placing security functions in the MUA exports the complexity to the user interface. In particular, most existing secure e-mail systems require user intervention for each message in order to apply the security transforms. Even those that do not require user action for each message still require some user action for each correspondent.

Ideally, security services would exist below the MUA, perhaps in the MTA layer itself. Since the MTA layer does not interact directly with the user (except by delivering or "bouncing" messages), the security model must support automation of the security tasks performed by the user in an MUA-based system. Consider the problems:

1. Collection and recovery of keys and credentials. Highly-available X.500-based directory services and X.509-based keys would make it easy to automate this process since every entity has a well-defined path of credentials between the user's trust root and the entity's key. Unfortunately, the Internet is not based on hierarchically-structured X.500 directory services, and highly-available services simply do not exist on the Internet (a problem made even more acute in e-mail systems that support off-line operation). PGP-style certification, in which the trust path is not unambiguously specified, makes this problem even harder.

2. Specification of the user's desires and policies. Here the tradeoff between flexibility and transparency is at its most acute. In a strictly hierarchical system, trust is completely specified in terms of the roots trusted by the local system; if a path exists between an entity and the root, a key is trusted, otherwise it is not. In a free-form scheme such as that used by PGP, whether a key is trustworthy is based on a judgment made on a case-by-case basis by the user. Such systems assume that the user is available every time a decision must be made about the keys in the trust path of each message.

3. The mapping between the layer boundaries and the trust boundaries. The MUA is almost by

definition trusted; the MTA may not be. In general, transparency is facilitated by pushing security services *away* from the user, while security is facilitated by keeping the security services under the user's direct control.

A better approach to layering is to add an e-mail security layer between the MUA and MTA layers. Such an e-mail security layer requires little or no direct interaction with the user. Instead, a user-configured *policy*, combined with *credentials* automatically retrieved from the net, directs the application of security transforms. For example, a typical policy would be to encrypt all outgoing e-mail for which a trusted public key can be found for the recipient, and for which the recipient has signed a statement that indicates the desire to receive encrypted e-mail. The policy also specifies the exact parameters of the keys to be considered trusted.

The selection of various security transforms and how to handle unanticipated conditions is difficult. Traditional approaches to secure e-mail systems (based in the MUA) can simply pass the question to the user. However, in a decentralized environment such as the Internet, it is difficult for users to know which security transforms are most appropriate. As such, most users simply forgo security.

The design of protocols such as PGP and S/MIME assumes that the most important problem is the formatting, encryption, decryption, signing, and verification of the message itself. Unfortunately, the message formatting is only the easiest part of the problem – even if these services are integrated across the great variety of widely-deployed e-mail clients, the other problems inherent in secure e-mail still remain to be solved.

In some cases, specifying a policy is quite straightforward. For example, within a centralized organization, trusting a certification hierarchy to bind keys to identities would be a reasonable policy. In such cases, making a secure e-mail system transparent is not as difficult, and some successes have been noted. Most of these systems have been based on hierarchical X.509-style certificates and are implemented in the MUA. For example, the US DoD Defense Message System supports transparent message security based on hardware key tokens.

However, certificate hierarchies lack the flexibility to bind arbitrary Internet e-mail addresses with their appropriate keys and assume a widely-deployed infrastructure for certifying keys and distributing certificates. More flexible decentralized schemes have been proposed, e.g. the PGP "web-of-trust." However, even though PGP does support binding of arbitrary e-mail addresses, such bindings cannot always be established by the PGP certification mechanism alone. The existence of a PGP "certification path" is only advisory; there is no mechanism for specifying a policy for deciding which paths will be trusted. As such, the PGP implementation usually requires user intervention to establish that a given key/name binding will be trusted. (It is only possible to specify that the key/name bindings directly signed by a given key will be trusted locally, but this does not expand the space of trusted keys by a significant amount since most paths include keys not known directly to the local user.)

Thus, existing systems either support transparency without flexibility (as in a strictly hierarchical X.509-based system) or flexibility without transparency (as in a fully-general PGP system). It would seem that these two goals are in conflict.

We argue that the most important missing piece is a semantically rich mechanism for describing credentials and user policies.

2 A language for describing trust

This section is a brief overview of the PolicyMaker trust management system, previously described in [BFL96].

The goal of *trust management* is to evaluate whether a particular statement or proposed action is consistent with local policy. In the PolicyMaker framework, the action under consideration is represented by an *action string.* The trust assumptions are divided into a *policy,* which is under local control, and *credentials,* which generally represent statements by others. The basic problem solved by PolicyMaker is to evaluate an action string based on the local policy and received credentials (collectively called *assertions*), and return to the application a boolean answer indicating whether the action encoded by the action string is trusted according to the policy.

In the general PolicyMaker framework, action strings are arbitrary strings, the semantics of which are determined by the application, bound to a list of one or more *requester-ids.* In an e-mail application (and, in fact, in the e-mail application described in the next section), the action strings might be based on the message headers and the requester-id might be the public key with which the message is associated. Policies and credentials are programs, written in a safe language, that examine the binding between an action string, the list of entities and its associated requester-id.

A credential is a statement giving a criterion for what action strings its *authorizer-id* accepts. Like requester-ids, authorizer-ids correspond directly to public keys, at least in the context of e-mail. Credentials are specified as programs, written in a safe programming language, that take as input an action string bound to a list of requester-ids and authorizer-ids with which it is associated. If the credential *trusts* the IDs to perform the action encoded in the action string, it can *accept* the action by adding its own authorizer-id to the list. As a simple example, an *unconditional delegation credential* states that its authorizer-id will accept all action strings that are accepted by another specific authorizer-id. Aside from the fact that they are written as arbitrary programs rather than simple attribute value pairs, PolicyMaker credentials differ in two important ways from the *certificates* found in X.509 (called *signed keys* in PGP). First, certificates are digitally signed, while PolicyMaker credentials are simple statements of the form "A allows B." Second, credentials are written the PolicyMaker syntax, while certificates may be represented in some other syntax, for example X.509 or PGP.

Consequently, the PolicyMaker model includes the notion of *translating* certificates into credentials. The translation step includes, at the same time, verifying the signature to make sure it was really signed by the authorizer-id, and translating the statement into the PolicyMaker credential syntax.

PolicyMaker policies are analogous to credentials. In fact, they are a special case of a credential, with an authorizer-id of "POLICY." Action strings accepted by POLICY are considered to be locally acceptable (Credentials generated from the translation certificates can never have an authorizer-id of "POLICY.") We use the term *assertion* to refer to the set of credentials and policy. Policies are locally generated and trusted, and form the "trust root" under which all queries are evaluated.

Assertions can also modify the action string that they accept. *Annotations* allow credentials to add additional information to the action strings they accept, rather than being limited to the boolean choice of accepting the action string or not. As such, annotations can be thought of as a communication mechanism between assertions, as well as communication between the application and the credentials. In our e-mail application, we use the annotation mechanism for representing

the trust graph, and for computing functions on it.

The basic PolicyMaker model suggests the sequence of steps for an application to use PolicyMaker to evaluate trust:

1. Generate an action string for the action to be considered.
2. Find the certificates (generally through remote queries) that are needed to support the action string.
3. Translate the certificates into PolicyMaker credentials, checking signatures where appropriate.
4. Present the action string, the policy, and the credentials to the PolicyMaker interpreter.
5. Use the result of the PolicyMaker evaluation to perform or not perform the action.

3 A transparent secure e-mail system for the Internet

We describe here the design of an e-mail security layer that is easily integrated into existing Internet infrastructure. In our system, user policies and e-mail certificates are specified as (or translated into) PolicyMaker assertions. The function of the security layer is to collect the appropriate certificates and user policy and form a PolicyMaker query for each processed message, the result of which determines how the message is treated.

The purpose of the security layer is to examine each incoming and outgoing message, apply the user's policy to the message to determine which, if any, security transforms should be applied to the message, and process the message appropriately. There are four possible transforms, two that can be applied to incoming messages and two that can be applied to outgoing messages.

The encryption transform protects the message body from eavesdropping attacks. It requires the least user interaction of the four transforms. Once a trusted key is found for the recipients, encryption is performed silently.

The signing transform adds a digital signature to the message, thereby authenticating the sender. Because this transform requires the secret key, it may cause a window to pop up requesting a passphrase. This passphrase is stored until an explicit "logout" operation, reducing the number of times it need be typed. Because signed messages are readable by all users, including those that do not have the ability to verify the signatures, one reasonable policy is to sign all outgoing mail.

The decryption transform does not require any evaluation of key trust, but does require the secret key, which is handled in the same way as signing.

Finally, the signature verification transform verifies and removes the signature. It then evaluates whether the signer's key is trusted, and reports on the trust in a header field which is added to the message and may be displayed by the MUA.

3.1 Using the e-mail security layer

Once the policy has been written and the MUA has been configured to communicate with the security layer, sending and receiving E-mail proceeds much as before. The "standard" policy distributed with the package will sign all outbound messages and encrypt "opportunistically"; if a trusted key is found for the recipient's address, it is used to encrypt the message. The standard policy

also interprets keywords on the subject line that override the default behavior. For example, the keyword `confidential` indicates that the message should bounce if no recipient key is found, and the keyword `clear` indicates that no security transform should be applied. Policies are written in a general programming language and can therefore be made arbitrarily more sophisticated if needed.

There are two ways that the security layer may find candidate keys and the certificates used to evaluate the trust in those keys. First, they may be stored in a local database. Second, they may be retrieved as the result of a remote database query. Thus, the local database might be considered to be a cache of the remote one.

For example, if `raph@cs.Berkeley.EDU` has configured his mailer to use the default policy, he can send a signed, encrypted message to `mab@research.att.com` by invoking the mailer in the ordinary way:

```
$ Mail mab@research.att.com
Subject: confidential: Business opportunity

I'll pay you $1,000,000 for the right to market
your Cold Fusion invention on the planet Mars.
.
```

If no trusted key is found for `mab@research.att.com`, the message bounces. Otherwise it is signed and encrypted and passed to the MTA for delivery.

Upon receipt of the message, invoking the mailer automatically filters the mailbox through the security layer, which applies the appropriate security transforms automatically. The results of any signature verifications are indicated with special `X-Premail-` header fields. The message above would be presented as follows:

```
To: mab@research.att.com
From: raph@cs.Berkeley.EDU
Subject: confidential: Business opportunity
X-Premail-Auth: Decrypted for mab; Trusted signature from user
   "Raphael L. Levien <raph@cs.Berkeley.EDU>".

I'll pay you $1,000,000 for the right to market
your Cold Fusion invention on the planet Mars.
```

Exceptions to the security policy (unsigned messages or bogus signatures) are indicated with similar `X-Premail-` headers.

3.2 A fully transparent e-mail system for Unix

We used the freely available PREMAIL package [Lev] as the starting point for our implementation. This software integrates encryption services with existing Unix e-mail clients. In particular, the integration with Netscape 3.0's built-in mailer is quite smooth.

While a full description of premail is beyond the scope of this document, it is perhaps worthwhile to recap the basic architecture of premail. It is installed as an additional layer between the e-mail client and the mail transport agent. From the perspective of the client, premail masquerades as

the mail transport interface (i.e. exports those interfaces), while from the point of view of the mail transport agent, premail appears to be a client.

When a user sends an outgoing message, the mail client hands the message to premail instead of the mail transport agent. At this point, premail performs whatever processing is requested (including encryption, signing, etc.). The resulting encrypted message is then passed to the mail transport agent for actual transmission across the network.

Conversely, for incoming mail, we are assuming an e-mail client which issues a command to retrieve messages from the mail folder. This command is reconfigured to point to a program which first actually retrieves the e-mail, then performs any necessary decryption and signature verification before handing the messages to the actual client.

The 0.44 release of premail automated message encryption and signing, but left the key management to PGP. As a result, some amount of manual configuration was required for each correspondent.

To this version of premail, we added a PolicyMaker-based system for finding trusted keys, thus making a fully transparent, secure e-mail system, including automatic verification of incoming signed e-mail, and automatic encryption of outgoing mail.

The modules added include:

- A database for storing candidate keys for a given e-mail address, in which the trust in the key/name binding is not known.
- A database for storing certificates, both PGP and PolicyMaker.
- A search engine for the database which, given a search root, a search target, and a search depth, finds the subgraph containing all paths from source to target bounded by the depth.
- Translators from the certificates into PolicyMaker credentials.
- "Glue code" to carry out the security transforms specified by the policy.

Because the database is not trusted, there is no reason for it to reside on the same machine as the e-mail client. We envision a simple TCP/IP protocol for querying the database, although our present implementation is entirely local. It is, however, efficient enough to handle searches over the MIT keyring, containing roughly 26,000 keys.

During normal operation, the only interaction between premail and the user is to query the user for a passphrase, if one is needed for signing or decryption. The passphrase stays "logged in" until explicitly logged out, so this interaction need only be typed once per e-mail session or never if the user trusts the local file system to store the passphrase permanently.

3.3 Integrating PolicyMaker

PolicyMaker is not itself a secure e-mail system; in fact, PolicyMaker does not interpret the semantics of action strings at all. There are a number of design decisions that must be made to adapt PolicyMaker for any particular application. This section describes how we use PolicyMaker as the basis for determining trust in Internet e-mail.

The first task is the design of the language for specifying action strings. The language must capture the security semantics of the application. In our e-mail application, whether a message is

considered "secure" depends on whether the information in the message header is reliable. Our action-string language, then, is based on RFC-822 [Cro82] message headers, with a few added fields that indicate specific security processing. A typical action string is as follows:

```
To: Raphael L. Levien <raph@cs.Berkeley.EDU>
Key: pgp.bb47cbda9285dd7be08fcb8101933bc2cc9b8da5
Subject: Test of PolicyMaker e-mail
Date: 19960805202407
```

The `To:` field indicates that this is an outgoing message and identifies the recipient (for verifying the signature of incoming mail, the field would have read `From:` instead). The `Key:` field specifies a candidate key that may be associated with the e-mail address. This field corresponds to the concept of "requestor-id" as discussed in section 2. However, for technical reasons, it is actually implemented as a component of the action string. The subject line of the message is included so that policies and credentials may be sensitive to specific subjects. For example, the policy may specify that, if certain keywords are present, the mail *must* be encrypted. Finally, the `Date:` field contains the present date, to allow certificates and policies to expire or otherwise be sensitive.

The overall meaning of this action string should be clear: if it is accepted, then the specified key can be used to encrypt mail with the specified subject list to the specified recipient on the specified date.

The next design detail is the translation of certificates into credentials. Our prototype supports two formats for certificates: standard PGP signed keys, and a "native PolicyMaker" certificate format which simply adds a signature wrapper around a PolicyMaker credential.

A PGP signed key is translated into a PolicyMaker credential similar to this one:

```
c57b34c6_signs_bb47cbda_pgp.9585d16830569410eeb8364d1b9f9f03264486a5.t:
  pgp.c57b34c6fece7e8a134d1f9fb5689537da29ffe8 ALLOWS
  {
    if ((lookup("To") == "Raphael L. Levien <raph@cs.Berkeley.EDU>" ||
 lookup("To") == "raph@cs.berkeley.edu") &&
        annotate_require("Key", "pgp.bb47cbda9285dd7be08fcb8101933bc2cc9b8da5"))
      accept()
  }
```

The first line of this credential contains an assertion-id, which must be unique for each assertion. Consequently, it contains a cryptographic hash of the original PGP certificate. The second line specifies the authorizer-id of the certificate issuer. The body specifies the actual content of the certificate predicate, which, in this case, is quite simple—action strings containing this key/name binding are accepted.[1]

A typical policy might resemble the following:

```
policy_1:
  POLICY ALLOWS
```

[1]The syntax may appear a bit awkward at first; PolicyMaker assertions are written in a "safe" version of the AWK programming language supported by a library for reading and manipulating action string data structures.

```
{
  check_auth("pgp.bb47cbda9285dd7be08fcb8101933bc2cc9b8da5")
  check_auth_depth("pgp.c57b34c6fece7e8a134d1f9fb5689537da29ffe8", 3)
  if (lookup("To" ~ /.*@(.*\.)?mit\.edu(>|$)/)) {
    check_auth("pgp.cad80f109f9ec5c5734f1c59cf7f5eb8946abb10")
  }
  if (!(lookup("Subject" ~ /nuclear/) && lookup("Key" == "none"))) {
    atleast(1)
  }
}
```

This policy expresses the following points:

- Full authority is delegated to key pgp.bb47...
- Authority is delegated to pgp.c57b..., but only for trust paths of length 3 or less.
- Authority is delegated to pgp.cad8..., but only for e-mail addresses within the mit.edu domain.
- If the subject field contains the word "nuclear," encryption is required (i.e. the key "none" will not be accepted), no matter what authority is granted by the other credentials.

Simpler policies are also possible. For example, here is a policy that allows sending unencrypted mail:

```
policy_opportunistic:
  POLICY ALLOWS
  {
    if (lookup("Key") == "none") {
      accept()
    }
  }
```

In conjunction with other policies, the effect of this policy is to encrypt whenever possible, and otherwise send the message unencrypted. The encryption process first tries all candidate keys, and if none succeed, tries the dummy key named "none." If this does not succeed, then an error message is displayed.

Alternatively, the policy could require that all outgoing e-mail is encrypted. These policies resemble the following:

```
policy_strict:
  POLICY ALLOWS
  {
    check_auth...
    if (lookup("Key") != "none") {
      atleast(1)
    }
  }
```

Since PGP certificates can only specify a binding between name and key, we also designed and implemented a format for native PolicyMaker certificates. These translations allow access to the full generality of PolicyMaker credentials.

The native PolicyMaker certificate is a signed tuple comprising the PolicyMaker assertion and metadata which is not used in the PolicyMaker evaluator, but can be used in a database search. The metadata is in the form of Key: Value pairs, in standard RFC822 syntax. The body of the certificate comprises the metadata and the PolicyMaker assertion, separated by a blank line. This body is tagged with a MIME header indicating a content-type of "application/x-pm-credential". Finally, this MIME object is signed using any of the signature protocols that can operate on MIME objects (including PGP/MIME and S/MIME).

Here is a simple example of such a certificate, in which the holder of key pgp.bb47... delegates authority fully to key pgp.09b8...:

```
Content-Type: multipart/signed; boundary="QXWgQ0FHoKs5w9";
   protocol="application/pgp-signature"; micalg=pgp-md5

--QXWgQ0FHoKs5w9
Content-Type: application/x-pm-credential

Domain: e-mail
Dependents: pgp.c57b34c6fece7e8a134d1f9fb5689537da29ffe8

bb47cbda_signs_c57b34c6_pgp.d8421c4d2989db17f3c3a03119828aca2d148d78.d:
  pgp.bb47cbda9285dd7be08fcb8101933bc2cc9b8da5 ALLOWS
  {
    delegate("pgp.c57b34c6fece7e8a134d1f9fb5689537da29ffe8")
  }

--QXWgQ0FHoKs5w9
Content-Type: application/pgp-signature

-----BEGIN PGP SIGNATURE-----
Version: 2.6.2

iQBFAwUBMgIbBGRuKj5D5x2JAQF05gF9E6255BF037aaRhkQffJ8WRLdSjuuBqDa
9+1522LSlE3tI0TgsFWj5W3xfzB1n/H2
=VQE2
-----END PGP SIGNATURE-----

--QXWgQ0FHoKs5w9--
```

3.4 Adding rich trust to existing e-mail infrastructure

An important goal of our system is that it work with existing infrastructure, rather than defining a completely new, non-interoperable set of message formats and certificate formats. Interoperability

is an issue on several levels.

First, since PolicyMaker expresses trust relationships about arbitrary objects, it is easy to evaluate the trust on existing keys, including PGP keys as well as keys for other encryption protocols.

Second, existing credentials can be used, by verifying and translating them into native PolicyMaker format. This opens the possibility of mixing credentials that were translated from several different certificate formats.

Third, even native PolicyMaker certificates use existing, standard signature protocols for the actual cryptography. In this way, we decouple the specification of trust from the encryption techniques. In addition, we avoid the problems of creating yet another signature protocol.

To demonstrate the benefits of this approach, we developed a prototype of a simple utility, RING O' TRUST, that interprets the key certificates on a PGP public keyring according to a specified PolicyMaker policy. Ring o' Trust translates ordinary PGP output[2] into native PolicyMaker assertions, derived from the signatures on key/user-id bindings. For each key on the keyring, Ring o' Trust formulates a PolicyMaker query requesting approval of that key for some purpose by the keyring policy. Then PolicyMaker evaluates each query in the context of the local keyring policy and the translated assertions. Finally, Ring o' Trust calls PGP to build a new keyring (the "ring of trust") consisting solely of keys approved by the policy. Minimal design effort is required to produce utilities such as Ring o' Trust that integrate cryptographic facilities with the PolicyMaker trust manager. While less transparent than our fully integrated system, it can run on a wider variety of platforms.

3.5 Deficiencies of PGP certificates

We found that PGP's certification model contains a number of limitations which render them inadequate for our application.

The most significant limitation with respect to automatic trust processing is that PGP certificate semantics are incompletely specified. It is not clear how, exactly, certificates should be interpreted when they are in the middle of a long chain of trust. While not stated as such in the PGP manual [Zim94], the best known formulation of PGP's trust policy is that "the web-of-trust is not transitive" [Fin96]. More specifically, the "standard" semantics of PGP signatures encode only a binding between key and name. There is no concept in PGP for a signed certificate that delegates trust to others. Thus, in the standard PGP model, the only key/name bindings that are approved are those for which the user *directly* trusts the certifier. This interpretation leaves most users in the "web" unable to communicate securely with one another.

For the time being, we get around this limitation by allowing the user to configure the manner in which PGP certificates are interpreted. The most conservative option allows only the strict semantics of the standard PGP model. A less restrictive option treats each PGP certificate as if it also conferred delegation. Binding and delegation are quite different concepts, and there are certainly cases where someone would want to vouch for one but not the other. However, by configuring our system

[2]Naive reliance on output from the PGP 2.6.2 user interface exposes the system to several known security weaknesses. Chief among these are the short length of key-ids, and the ability to create user-ids containing newlines. The former property can lead to confusion between keys with matching key IDs, while the latter allows spoofing of arbitrary keys and signatures in some PGP keyring output formats. A release version of Ring o' Trust should incorporate measures to compensate for these shortcomings in the PGP keyring management interface.

to interpret binding certificates as also conferring delegation, we are able to experiment with an infrastructure in which delegation certificates exist.

A second problem is that PGP does not specify how keys are to be used; a key is simply bound to an e-mail address and/or name. There is no way to distinguish between, *e.g.,* keys used for signature and keys used for encryption.

Finally, the PGP expiration and revocation model puts control in the hands of the entity being certified rather than the entity doing the certifying. That is, a user may revoke his or her own key, but a certifier cannot revoke a signature without the cooperation of the key holder being revoked. This is contrary to intuitive notions of certificate semantics.

3.6 The need for autocerts

One requirement for a useful public key infrastructure is an implementation of an "autocert" (terminology borrowed from [RL]). In the e-mail domain, this is a simple signed object stating a policy that the *recipient* of e-mail would like enforced.

In the PGP world, the most pressing need for the autocert is to indicate whether the recipient prefers mail to be PGP encrypted by default. There are good reasons for this policy to be set either way. For example, I may prefer all my incoming mail to be encrypted because I have an e-mail client which can easily handle the decryption. On the other hand, many have inadequate clients, or wish to be able to read their e-mail from a variety of machines, not all of which are trusted for holding the encryption key.

In the near future, another compelling motivation for the autocert will come to light. The current version of PGP, 2.6.2, does not implement algorithm choice—the algorithms for hashing, symmetric encryption, and public key encryption are all fixed. However, support for more algorithms is promised for the next major release. Currently, there is no mechanism for negotiating the algorithm choice. Thus, users must either configure algorithm selection by hand (which is unacceptable in the ubiquitous e-mail encryption model), or rely on defaults, which may be inadequate (for example, the MD5 hash algorithm, which is the algorithm used in PGP 2.6.2, is partially broken[Dob96], and will very possibly be completely broken soon).

If the autocert infrastructure is in place, then there are other recipient policy items that fit nicely into the structure. For example, the recipient may indicate the ability to accept MIME objects in general, or specific content-types. It is worth noting that it is becoming increasingly common to find information of this nature in sigs, .plan files, and Web pages. The autocert mechanism would unify and formalize these information paths.

Other possible uses for the autocert include: advertising e-mail based servers associated with the account (this could be used, among other things, to negotiate return receipts), pointers to personal Web pages, v-cards, and other such information.

Here is an autocert in the contemplated format:

```
Content-Type: multipart/signed; boundary="jAN0mrMZICOYuL";
   protocol="application/pgp-signature"; micalg=pgp-md5

--jAN0mrMZICOYuL
Content-Type: application/x-pm-autocert
```

```
Autocert-Owner: pgp.bb47cbda9285dd7be08fcb8101933bc2cc9b8da5
Encryption-Preferences: pgp-2.6.2, none
E-mail-address: raph@cs.berkeley.edu
MIME-Accepted: text/plain, text/html, application/postscript, image/*,
 */*
WWW: http://www.cs.berkeley.edu/~raph/

--jANOmrMZICOYuL
Content-Type: application/pgp-signature

-----BEGIN PGP SIGNATURE-----
Version: 2.6.2

iiQBFAwUBMgIbK2RuKj5D5x2JAQFcCAF+KAWAHRHEWaeIq82sDWF6SpfzLw6zIFYc
hPty/k6zD1ntvuTQsjgmhkfXV5CLe7gw
=MOoZ
-----END PGP SIGNATURE-----

--jANOmrMZICOYuL--
```

In the S/MIME world, the need for autocerts is equally pressing, if not more so. In particular, the current S/MIME draft [Inc95] specifies a default of 40-bit RC4 encryption, which is completely insecure [BDR+96]. Since S/MIME clients will use these defaults unless manually overridden, if this problem is not fixed, the vast majority of S/MIME messages in practice will be insecure. The S/MIME community is considering a number of solutions to this problem, but no such solution is clearly headed for standardization.

4 Discussion

Our prototype implementation demonstrates that transparent, secure e-mail on the Internet is a realistic goal. The central component is a rich language for specifying trust.

The true test of such a trust management system is whether it expresses trust relationships that are truly useful, and which cannot easily be expressed in simpler mechanisms.

Our system clearly outperforms standard PGP certificates. However, this is not surprising given the severe limitations of PGP. The lack of delegation certificates is perhaps the most serious problem.

A more interesting comparison is that with X.509. The initial version of X.509 depended on a X.500-style hierarchical directory and did not work well on the Internet, as made clear by the recent reclassification of the PEM RFC's as "historical." More recent versions of X.509 attempt to address Internet more directly. In particular, X.509v3 does explicitly allow local configuration of trust roots [II96]. Additionally, X.509v3 includes the notion of "certification path constraints," which are analogous to our policies. However, new types of policy require direct support in each application which uses X.509v3 certificates. As a consequence, it is likely that only policies specified directly by X.509v3 will be widely supported. These are path length bounds and subtree constraints on distinguished names. The latter type is of limited use on the Internet, since distinguished names are not in widespread use.

By contrast, PolicyMaker can express a significantly richer set of policies, especially certificates defined in terms of the entire trust graph, rather than a linear trust chain. For example, it is quite feasible to write a policy that verifies the result of a PathServer database query [RS96a]. The PathServer search engine finds a maximal number of vertex-disjoint paths from source to target. A policy that requires at least k such vertex-disjoint paths will not accept an invalid key if there are less than k forged certificates. Such a policy can increase the confidence in trusted name/key bindings even when "certification authorities" are hosted on untrusted machines. Our implementation of an assertion to check this property was about 80 lines of code.

It is important that these arbitrarily complex policies can be encapsulated into certificates in addition to local policy. For example, the "e-mail security officer" of a company may create certificates containing useful policies. Then, individual users can specify a policy which simply delegates to the e-mail security officer.

5 Future work

We are in the process of releasing our tools for general distribution. We expect to have them available in Fall 1996 on the `research.att.com` web site.

While our prototype demonstrated that transparently adding security to Internet e-mail is feasible, considerable work remains to make this a broadly useful tool.

The most serious limitations exist in the database. At present, there is no facility for remote queries. We envision that the local database would be used as a cache for certificates. A remote query would be generated only if the trust in the key/name binding cannot be determined using certificates contained in the cache. Also, our prototype only supports automatic retrieval of PGP certificates. Adding automatic retrieval of PolicyMaker certificates is obviously important.

The remote database need not necessarily be based on the same technology as the local one. In fact, we hope to interface our system to the PathServer search engine, which is available on the Web [RS96b].

Another obvious direction is to incorporate S/MIME message formats, so as to interoperate with other S/MIME clients, and to demonstrate the decoupling of our trust management framework from the underlying cryptographic engine.

An issue with S/MIME integration is the fact that existing S/MIME clients require X.509 certificates, rather than plain RSA keys. One solution would be to create a local certification authority which creates certificates for each trusted key/name binding.

Our prototype was built in the Unix operating system for ease of development, but to attract a wider user community, a Win32 version is essential. In the Unix world, it is straightforward to add an e-mail security layer between the MUA and the MTA, because the MTA is simply invoked on a shell command. On most Win32 machines, the connection between the MUA and the MTA is through net-based protocols such as SMTP and POP3. One promising approach is to install the e-mail security layer as a proxy server running on the user's machine. The MUA is then configured to connect to the local machine rather than a remote mail server. Since the e-mail security layer is capable of decrypting and signing messages, it needs to authenticate clients as they connect. On a multiuser machine, this presents problems. However, on a single-user setup, it suffices to check that the IP address of the client matches that of the server.

6 Acknowledgments

This paper was helped immeasurably by discussions with Joan Feigenbaum, Ron Rivest, Jack Lacy, Carl Ellison, and Mike Reiter. Thanks to Heather Levien for editing drafts of the paper. The PolicyMaker system was co-developed by the third author, Joan Feigenbaum and Jack Lacy. The original design of premail, including the layered approach, was inspired by PGPsendmail [Goo94].

References

[BDR+96] Matt Blaze, Whitfield Diffie, Ronald L. Rivest, Bruce Schneier, Tsutomu Shimomura, Eric Thompson, and Michael Wiener. Minimal key lengths for symmetric ciphers to provide adequate commercial security. ftp://research.att.com/dist/mab/keylength.ps, January 1996.

[BFL96] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proc. 1996 IEEE Symposium on Security and Privacy*, 1996.

[Cro82] David Crocker. Standard for the format of ARPA Internet text messages. RFC 822, August 1982.

[Dob96] Hans Dobbertin. Cryptanalysis of MD5 compress. Eurocrypt 1996 rump session, also http://www.cs.ucsd.edu/users/bsy/dobbertin.ps, 1996.

[Fin96] Hal Finney. Transitive trust and MLM. post to cypherpunks mailing list, archived at http://infinity.nus.sg/cypherpunks/dir.archive-96.05.02-96.05.08/0415.html, May 1996.

[Goo94] Richard Gooch. PGPSendmail package, July 1994.

[II96] ISO/IEC JTC 1/SC 21/WG 4 and ITU-T Q15/7. Final text of draft amendments DAM 4 to ISO/IEC 9594-2, DAM 2 to ISO/IEC 9694-6, dam 1 to ISO/IEC 9594-8 on certificate extensions. Final draft (June 30). To be distributed by the SC21 Secretariat., June 1996.

[Inc95] RSA Data Security Inc. S/MIME implementation guide and interoperability profile, version 1. http://www.rsa.com/pub/S-MIME/smimeimp.txt, August 1995.

[Lev] Raph Levien. premail web page. http://www.c2.net/~raph/premail.html.

[RL] Ron Rivest and Butler Lampson. SDSI—a simple distributed security infrastructure. http://theory.lcs.mit.edu/~rivest/. working draft.

[RS96a] M. K. Reiter and S. G. Stubblebine. Path independence for authentication in large-scale systems. Technical Report Research Technical Report Number 96.8.1, AT&T Laboratories, August 1996.

[RS96b] M. K. Reiter and S. G. Stubblebine. Pathserver web interface. http://www.research.att.com/~reiter/PathServer/, July 1996.

[Zim94] Philip Zimmermann. Pgp user's guide, October 1994.

Appendix A: Implementation of PolicyMaker

We implemented a simple version of PolicyMaker [BFL96]. The description of the implementation in [BFL96] is not fully satisfying, especially the implementation of annotations, so we briefly recap it here.

In this implementation, the input comprises a query and a set of assertions (including policy assertions and assertions derived from credential). The result is a set of *acceptance records*, as well as a status code indicating success or failure.

The interpreter loop is conceptually quite simple: it iterates a *state* consisting of a set of acceptance records until fixpoint is reached. Initially, the state is seeded from the query. On each iteration, the state is presented to each of the assertions. Whenever a credential produces a new action string in response to the state, that new action string is placed into an acceptance record and added to the state.

Acceptance records are 4-tuples consisting of: authorizer-id of the credential issuer, assertion-id, record-id, and action string.

An assertion is a 3-tuple of: assertion-id, authorizer-id, and a *filter program*. If the authorizer-id is `POLICY`, then the assertion is considered to be a *policy*. Otherwise, it is considered to be a *credential*. In general, credentials generated as a result of translating some signed credential into the PolicyMaker language will have authorizer-ids corresponding to the public key which signed the original credential.

Each filter program accepts a list of acceptance records as input, and produces a set of action strings as output. To convert these action strings to acceptance records, the PolicyMaker interpreter simply adds the authorizer-id and assertion-id from the assertion itself, and generates a new unique record-id, assuming that the (assertion-id, authorizer-id, action string)-tuple is unique. If that tuple already exists in the state it is discarded.