PLAN Security Guide*

for PLAN version 3.2

Michael W. Hicks

July 25, 2001

1 Introduction

This document should serve as a tutorial and reference for the use of security features of PLAN 3.2. More detail on the individual services may be found in the PLAN Programmer's Guide [2]. Details about the technical rationale and the design of the security system may be found in [7]; this paper should be read first, as its knowledge will be assumed herein. Note that the per-packet CPU and memory restrictions described in [1] are not implemented in the present version of PLAN[1].

PLAN was designed to be expression-limited with the idea that 'pure' PLAN programs could run without the need for heavyweight security mechanisms while PLAN service routines, which are written in more general-purpose languages, may require additional restrictions. To make service calls safe, we have provided a system of trust management. Each node administrator creates a policy which restricts the use of certain 'unsafe' services to select parties. A series of routines are provided in this release to formulate such a policy and enforce it at runtime.

Service call restriction occurs through a process of *authorization*. This implies that a user must *authenticate* himself with the node, and then be checked with the node's security policy. Authentication must be secure so that no spoofing (pretending to be someone you're not) is possible. We ensure this by using cryptography. Even though many PLAN programs should be able to avoid accessing restricted services, we would still like authentication and authorization to be as lightweight as possible.

*This work was supported by DARPA under Contract N66001-96-C-852.

[1]PLAN version 3.2 is implemented in OCaml, version 2.02. While this document attempts to be as implementation-independent as possible, certain aspects are implementation dependent, and will refer to OCaml as necessary. An earlier, Java-based version of PLAN (2.2) is also available and has its own documentation suite. *PLAN 2.2 does not have any of the service-level security features described herein.*

Contents

2 Authentication

Authentication takes place in two phases. First, the application which is injecting PLAN programs into the network must authenticate itself with the node on which it requires secure access. This is done by an exchange of messages (a protocol) which allows both parties to derive a *shared secret*.

Secondly, after the authentication protocol has taken place, the user uses the shared secret to *sign* a *chunk* (a piece of code, see below) to be executed on the secured node. Before evaluating the chunk, the node *verifies* the signature (using the derived shared secret) as that of the purported identity of the code. If this checks out, then that code is evaluated *with the privilege of that user* – any additional services that are allowed to the represented user are *added* to the default environment, and any restricted services are *subtracted*. This technique is called *namespace-based security*. We'll talk about it first, and then discuss authentication.

2.1 `Autheval`

A chunk (or **c**ode **hunk**) may be thought of as a function that is waiting to be applied. In PLAN, chunks are first-class (that is, they are program values), and consist internally of some PLAN code (if necessary), a function name, and a list of values to be used as arguments during the application. A chunk is evaluated as part of normal packet evaluation, or by passing it to the `eval` service, which processes the definitions in the code, resolves the function name with the current environment, performs the application, and returns the result.

We've added an additional service called `authEval` which takes as arguments a chunk and a digital signature. The chunk is in marshalled form, represented as a PLAN value of type `blob` (meaning unstructured data), while the signature is a PLAN three-tuple of type $blob \times int \times int$. The first field of the three-tuple is the cryptographic signature, the second is the user's identifying *SPI* (Security Parameters Index) as negotiated in the authentication protocol, and the final field is a counter, to prevent replay attacks. `authEval` verifies the digital signature against the binary representation of the chunk. If successful, the chunk is evaluated as in `eval`; otherwise, the exception `AuthenticationFailed` is raised. During the evaluation of the chunk, the authentication code keeps track of the authenticated principal by associating it with the current thread id. When the chunk terminates, the association is removed. This way, if any service invoked as a result of chunk evaluation wishes to perform authorization, it may ask the authentication service for the current principal. Because a caller's thread id cannot be forged, this provides a safe way to track a principal without worry that some malicious service will change the associated principal after the authentication phase.

2.2 Authentication protocol

As mentioned, authentication is done in advance of any PLAN program exchange through a protocol which negotiates a secret shared between a user's application and a PLAN node. This protocol is a variant of the well-known Diffie-Hellman [4] exchange which makes use of public key cryptography to prevent man-in-the-middle attacks. We illustrate this protocol with an exchange initiated by an application to a node:

1. The user application requests authentication with a remote PLAN node. The application sends a PLAN program to the node with which it wants to authenticate. This program invokes the PLAN service `DHmessageOne` with two arguments: a *certificate*, and a signature of that certificate using the user's public/private key pair. In the current implementation, we use DSA [5] keys for authentication. All certificates used during the exchange are PLAN tuples which begin with the following four fields:

   - the signer's public key
   - a random number (a *cookie*)
   - the time at which certificate is valid
   - the time at which certificate expires

   The latter three fields are are essentially used to prevent replay attacks. Note that the duration of time fields implies the level of synchronization between the two nodes' clocks. This first certificate additionally contains:

   - the sender's *exchange id*
   - the receiver's address
   - the sender's address

   The exchange id is generated at the sender, and is used to identify this particular protocol exchange. At the completion of the protocol it will be used to establish the *SPI* (more later). For a node, the address is represented as a PLAN *host*, while an application's address is of type $host \times port$.

2. When the node receives the message, it verifies the signature on the certificate with the certificate signer's key. It then makes sure the certificate is active and has not expired, and that the receiver's address is the current node.

   If the current node wishes to negotiate with the sender, it creates a bit of state to keep track of the exchange. It stores the sender's exchange id, calculates its own exchange id, and additionally stores the sender's address and public key. It also calculates its local portion of the shared secret and the "public" value of this secret. The response certificate includes both exchange id's, the public value, the app's public key, and both addresses.

Since the response message is being sent to an application, rather than a node, it is packaged as a tuple, labeled by a string "DHmessageTwo," to be delivered to the application. This tuple also contains the certificate and its signature.

3. The application verifies the signature, looks up the exchange id to find the information stored about this exchange, and verifies that it is all correct. It then calculates its secret and corresponding public value, then combines it with the value in the message to produce the shared secret. The SPI identifying the secret is then calculated based on the two exchange id's. This SPI is used to identify the secret in later messages which have been signed using the secret. The application's public value is included in a message back to the node which is essentially a mirror of the message just received. As described earlier, this third message is actually a PLAN program that invokes the PLAN service `DHmessageThree` with two arguments: the certificate containing the described information, and a signature of that certificate using the user's public/private key pair.

4. The node receives the final message and repeats the actions taken by the application for the previous message. No response is sent; the protocol is complete.

Each principal in the exchange now has secret known only to the other principal. The shared secret is used to sign chunks used by `authEval`, as described. This is done by a technique called HMAC-SHA1 [6]. At this point you might be wondering, why go through all this trouble when we can simply sign and verify chunks using a principal's public key? The simple answer is that using HMAC-SHA1 is *much* faster, following the trend that secret cryptography is faster than public key cryptography.

In our implementation, the secret is stored in two tables; one table indexed by the peer's address (which includes other information about the protocol), and another indexed by SPI. The former table may be used by the application when it wants to send a message to the peer, the latter table is used to look up the SPI found in a signed chunk so that the signature may be verified. More about how the implementation may be used is found in the next section.

Note that this authentication exchange is not limited to application contacting a node—nodes may contact other nodes and applications may contact other applications. The latter is in common practice in the Internet today, and the former may be used, for example, to establish trust relations between administrative domains.

2.3 Authentication Tutorial

We'll present two tutorials for authentication in PLAN, presenting first application-node authentication, and then node-node authentication. For these tutorials to work, you need to compile PLAN to use QCM, and to have authentication/authorization debugging messages turned on. To do so, edit the Makefile as follows:

1. Change the line at the top of the Makefile

   ```
   # USE_QCM = true             ### Use QCM as the node policy manager
   ```

   to be uncommented:

   ```
   USE_QCM = true             ### Use QCM as the node policy manager
   ```

2. Add the `AUTH_DEBUG` flag to the compilation flags. That is, change

   ```
   CAMLP4FLAGS     = -DPLAN_UNSAFE -DPLANPORT_TCP
   ```

   to be

   ```
   CAMLP4FLAGS     = -DPLAN_UNSAFE -DAUTH_DEBUG -DPLANPORT_TCP
   ```

   After making these changes, do a `make clean depend` to regenerate the dependencies, and then `make` to rebuild the system.

2.3.1 Application-Node Authentication

A sample application and service have been provided as templates for writing code using the security services. This application will perform the authentication protocol as described and then invoke a user-defined function which may make use of the shared secret; by default this involves simply printing a message from the authenticated node.

To try out the application, first start an active router as described in the PLAN tutorial [3]. As in the tutorial, we'll assume that the router is started on a machine with address m on port 3324. Using this port is important because the router will use the keys provided with the plan distribution in the file `key.3324`. We also assume that all commands are invoked from the `plan` directory.

We need to start the application so it may negotiate with the router. The application is called `authapp`; it may be built by doing

```
make authapp
```

which deposits the executable in the `bin` directory. Usage of the program can be obtained by `bin/authapp -help`:

```
usage: bin/authapp [-hf host_file] [-p local_port] [-s socknum]
      [-server] dest_host key_file
  -hf Specify name resolution file (default=EXP_IP_ADDRS).
  -p Set injection port. (default=None)
  -server Set Server Mode (default=not set)
  -s Specify domain socket to connect to on receiving side. (default=3324)
```

As required arguments, the application takes the hostname of the router with which it wishes to communicate and the files that store its own public and private keys. Keys can be generated by another program provided with the distribution called `plan_keygen`, which may be built using the command `make plan_keygen`. This program takes a filename argument and generates DSA public and private keys storing a base-64 representation in the given file.

Standard options to `authapp` include `-hf` for the name of the file used to resolve domain-name addresses, and `-p` for the port number of the local plan interpreter; see the tutorial document for more on these. `authapp` is general enough that it may authenticate not only with an active node, but also with another application (and may itself be the server in that instance) or serve as the server side of a node-app authentication. In the case that the application is authenticating with another application, `authapp` would use the `-s` option to specify the PLAN port number on which the remote application is listening at the specified destination. To use `authapp` as the receiving participant in the protocol exchange (where the initiator is either another `authapp` or a PLAN router), use the `-server` option.

In our example, we'll assume that we're starting the `authapp` on the same machine as our router, m. From another xterm, we would type

```
bin/authapp -p 3324 m key
```

where `key` is the file that contains the application's public and private keys. This has been provided with the distribution (you don't need to generate it) in the `plan` directory.

The application sends the first message to the node. Printed on the application console are the following messages[2]:

```
main: got my port: <port:(127.0.0.1,3324),1>
Storing entry with key <Node: (127.0.0.1,3324) >
```

Essentially the application is storing in its protocol table that it has started a negotiation with the node at $(127.0.0.1, 3324)$ (i.e., m). Furthermore, the application is keeping track of the PLAN port it uses to communicate with its local router; this port contains the session id for this application (1).

In the node's log file, we find the following messages are printed after the receipt of the first message:

```
Authproto: [ DHmessageOne ] with cert (<blob:566>,<blob:38>,93051
9501,930520101,0,(127.0.0.1,3324),((127.0.0.1,3324),<port:(127.0.
0.1,3324),1>))
Storing entry with key <Application: ((127.0.0.1,3324), <port:(12
7.0.0.1,3324),1>)>
```

[2]The IP address of m is printed in the diagnostic messages. Here this address is printed as the loopback address, 127.0.0.1 for expositional purposes.

This indicates that the authentication request was received and the protocol was initiated for the application on node (127.0.0.1, 3324), which has session id 1. The node sends the response, which causes the app to print

```
Authproto: [ DHmessageTwo ] with cert (<blob:566>,<blob:39>,930519
505,930520105,0,0,<blob:309>,<blob:566>,((127.0.0.1,3324),<port:(1
27.0.0.1,3324),1>),(127.0.0.1,3324))
Finding entry with key <Node: (127.0.0.1,3324) >
Calculated shared secret |2125718927524411328493843923218892174119
922226269965517999906850471020703992323480943083011891723260923569
786035298658358526807900077466329250605511687325263329121557366026
123831338589432298164307582211644566693860112609214362571249075333
519598834822874461197794695987211289164577495723370914058267031267
2334|
sSPI=0
rSPI=0
Updating entry with key <Node: (127.0.0.1,3324) >
```

Authapp indicates it has received the message from the node that contains the node's public value. From this and its own secret (now generated), authapp creates the shared secret which it stores in a table. The shared secret is printed to the console, followed by the generated sender and receiver SPI's for the exchange (sSPI and rSPI, respectively). Finally, the node receives the final message from the application, logging:

```
Authproto: [ DHmessageThree ] with cert (<blob:566>,<blob:39>,9305
19512,930520112,0,0,<blob:308>,<blob:566>,(127.0.0.1,3324),((127.0
.0.1,3324),<port:(127.0.0.1,3324),1>))
Finding entry with key <Application: ((127.0.0.1,3324), <port:(127
.0.0.1,3324),1>)>
Calculated shared secret |2125718927524411328493843923218892174119
922226269965517999906850471020703992323480943083011891723260923569
786035298658358526807900077466329250605511687325263329121557366026
123831338589432298164307582211644566693860112609214362571249075333
519598834822874461197794695987211289164577495723370914058267031267
2334|
sSPI=0
rSPI=0
Updating entry with key <Application: ((127.0.0.1,3324), <port:(12
7.0.0.1,3324),1>)>
```

Here we see that the shared secret and the SPI's calculated are the same. When you run this code, the shared secret will not have the same value as the one shown here, but the SPI's should be the same (since they all start counting from 0). The node stores this information in its tables.

Finally, the application executes its user function. This function creates a chunk which simply prints a message "authenticated successfully." It signs this chunk and sends it the remote node, providing the chunk and the signature as arguments to `authEval`. If authentication succeeds, the message will be printed at the application, otherwise an exception will be raised. The application prints:

```
Finding entry with key <Node: (127.0.0.1,3324) >
Signing tuple |(<blob:43>,0,1)|
Data = |[8][0][0][0][3][6][0][0][0]+[13][0][4]doit[0][0][0][1][2][
0][4]doit[0][0][0][2][0][5]print[0][1][2][0][7]authSvc[0][0][1][0]
[0][0][0][1][0][0][0][1]|
 Sign = |[23]x@G[237]_[246]_[251][220]x[229]|
Sending test message
```

In the node log, we see the message

```
Verifying tuple |(<blob:43>,0,1)|
Data = |[8][0][0][0][3][6][0][0][0]+[13][0][4]doit[0][0][0][1][2][
0][4]doit[0][0][0][2][0][5]print[0][1][2][0][7]authSvc[0][0][1][0]
[0][0][0][1][0][0][0][1]|
 Calc |[23]x@G[237]_[246]_[251][220]x[229]| =?
 Sign |[23]x@G[237]_[246]_[251][220]x[229]|
```

Here we see that the node verifies that the provided signature matches the calculated one. Then, as a result, the application prints the message

```
127.0.0.1:3324 authenticated successfully
```

`Authapp` as a template The `authapp` program was written as a template as well as a demonstration program. It has been structured to allow easy modification for your own purposes.

Assuming that the first thing you want your application to do is to authenticate with a node, you need only modify the function `user_main` to perform the appropriate actions. This function is of type $Authproto.principal\_addr \times Activehost.activehost \times (Basis.active\_packet \rightarrow unit) \times (unit \rightarrow string)$. The first argument is the peer just authenticated with (either a node or an application), the second is address of the local PLAN interpreter, the third is the function used to send packets to the interpreter, and the fourth may be used to receive data from the interpreter (presented as strings). When the `user_main` function returns, the connection to the local interpreter is closed and the program exits.

It is slightly more complicated if you want to authenticate with multiple nodes and perhaps perform different actions with each of them. In this case, you have to pull the authentication code out of `main` into a separate function which may be repeated, perhaps on a per-thread basis.

2.3.2 Node-Node Authentication

The PLAN daemons are also equipped with the ability to authenticate with other nodes; this ability is particularly important to the firewall application that we will illustrate in Section 3.3.2.

When a node starts, it may be given a list of hostnames to its `-authlist` parameter to indicate the other nodes to authenticate with. For each address given in the list, it waits until an entry may be found in the RIP routing table for that node, and then initiates a connection to it. This allows a node-node connection to be possible for any node on the network, as opposed to just adjacent nodes (which could be achieved by looking in the ARP table for the requested recipient). However, this technique leverages the fact that the RIP routing implementation does not having “default” routing table entries. As an alternative, we could have simply attempted to send the initiation message, retrying if the destination was determined to be unreachable along the way.

We present the tutorial by starting two nodes, as in the PLAN tutorial [3], one on port 3324 and the other on port 3325. We start the one on port 3325 first, just as in the tutorial:

% bin/pland -l log3325 -ip 3325 *m*2

Then we start the one on 3324, nearly the same as in the tutorial, but we also specify *m*:3325 as an argument to `-authlist`:

% bin/pland -l log3324 -ip 3324 -authlist*m*:3325 *m*1

The initiating node (*m*:3324) first waits for the receiving node to become accessible:

```
trying to authenticate with node (127.0.0.1,3325)
currently unknown
trying to authenticate with node (127.0.0.1,3325)
```

Once the receiver is known, the initiation message is sent. The receiving node processes the initiation message, calculates its portion of the shared secret and sends the response to the initiating node:

```
Authproto: [ DHmessageOne ] with cert (<blob:566>,<blob:38>,930513
828,930514428,0,(127.0.0.1,3325),(127.0.0.1,3324))
Storing entry with key <Node: (127.0.0.1,3324) >
```

The initiating node receives the response, calculates its own portion of the shared secret, and using the information in the message received, calculates and stores the shared secret:

```
Storing entry with key <Node: (127.0.0.1,3325) >
Authproto: [ DHmessageTwo ] with cert (<blob:566>,<blob:39>,930513
832,930514432,0,0,<blob:308>,<blob:566>,(127.0.0.1,3324),(127.0.0.
1,3325))
Finding entry with key <Node: (127.0.0.1,3325) >
Calculated shared secret |2576912389757616131645963461487159443026
571643347028392844167107049555585165992701860118710028039629843832
243076311528753543566417243156174533504726150218095603906742182340
692675347036154605455462926931605367360169111711582680304704305625
```

```
472475762774815248463312144731495286620602337979104139886906037797
263|
sSPI=0
rSPI=0
Updating entry with key <Node: (127.0.0.1,3325) >
```

Finally, the receiving node receives the other piece of the shared secret, and calculates and stores the full secret.

```
Authproto: [ DHmessageThree ] with cert (<blob:566>,<blob:39>,9305
13839,930514439,0,0,<blob:308>,<blob:566>,(127.0.0.1,3325),(127.0.
0.1,3324))
Finding entry with key <Node: (127.0.0.1,3324) >
Calculated shared secret |2576912389757616131645963461487159443026
571643347028392844167107049555585165992701860118710028039629843832
243076311528753543566417243156174533504726150218095603906742182340
692675347036154605455462926931605367360169111711582680304704305625
472475762774815248463312144731495286620602337979104139886906037797
263|
sSPI=0
rSPI=0
Updating entry with key <Node: (127.0.0.1,3324) >
```

Multiple protocol exchanges may occur simulataneously and will be stored properly in the resulting tables.

2.4 Implementation Notes

The code used to implement the authentication protocol may be found in the `security` directory in the PLAN distribution. The file `authproto.ml` contains the guts of the code that does the protocol operations. The protocol is initiated by calling `initiate_dh_exchange`. Each of the exchanged messages is processed by `dh_message_one`, `dh_message_two`, and `dh_message_three`, respectively, and the certificate to be sent in response is returned. Each one of the protocol-processing functions is called by wrappers appropriate to the principal type – either node or application (recall that for the former, the protocol messages take the form of PLAN programs making service invocations, while the the latter are delivered tuples) – and sends the response. The PLAN services used in the protocol are defined in `authproto_svc_impl.ml`. The code for the node-node authentication may be found in `nodeauth.ml`.

The file `auth.ml` implements functions for authentication, both DSA-based for the protocol, and HMAC-SHA1 for subsequent authentications. It also provides services used by `authEval` for mapping a principal to a thread id, and a service for retrieving this mapping, used by the authorization functions. `Autheval` itself is defined with a cast of supporting functions in `crypto_svc_impl.ml`. The DSA implementation resides in the `crypto` directory.

The messages printed during the authentication are printed because the compilation flag `-DAUTH_DEBUG` is on by default. To eliminate these messages, recompile without this flag (remove it from the `CAMLP4FLAGS` variable in the Makefile).

Finally, the protocol is computationally intensive, and it is often the case that the initiator completes his portion of the calculation (and is thus ready to use the secret) before the receiver. To compensate, we put a timer in `authapp` to wait for the receiver to finish. You may find that you have to bump up this timer on slower machines. A more robust solution would be to have the receiver send a completion message.

3 Authorization

Once a principal has been authenticated, it may have *access* to and *usage* of services corresponding to its level of privilege. This implies that the node must be able to map some set of protected services to each principal–this is the security *policy* of the node–and keep track of how services may be used by that principal.

For access, we define a way to formulate such as policy based on sets in [7, 1]. Each level of privilege is defined by the set of principals which have the same privilege level, and each service is associated with some privilege level. Privilege levels may be overlapping. For instance, in the government, those with top secret clearance may look at all classified documents, those with secret clearance may see a subset of these, and finally those with confidential clearance may see a subset of these. In our model, top secret clearance would be defined by some set of users TS, secret clearance would be $TS \cup S$, and confidential clearance would be $TS \cup S \cup C$. This way, a service which is associated with top secret clearance may only be accessed by those in the set TS; however, a service with secret clearance may be accessed by those in S and TS. In this case, the sets are concentric, but other approaches are also possible.

For the latter, we leave it to the service to define sets indicating how the service may be used. This may be formed in a way similar to the access policy, but rather than storing which services may be access, we store parameters indicating how that service may be used. The form of these parameters depends on the implementation of the service.

Using this architecture implies that the node must be able to determine to which privilege set a user belongs, and the node must be able to determine the association between services (or usage parameters) and privilege sets. How these determinations are made is a matter of implementation. Authorization data for a node could be stored *monolithically*—that is, the node maintains all security information internally to itself—or, *hierarchically*—it shares the task of storing the information with other, trusted nodes—or, *ephemerally*—it doesn't store any information directly, but instead relies on *certificates* from trusted sources which describe privilege on a per-use basis. We have structured the PLAN implementation to be independent of the approach taken, or the means by which the approach is implemented.

In addition to access to privileged services, the *use* of those services may also be a matter of privilege. For example, a service which provides persistent storage on the node may be available to privileged users, and the amount of storage allowed to those users may also vary according to privilege.

To implement the security policy of the node, one must write some code that implements the `POLICY` signature (in the file `security/policy.ml`):

```
(* Stuff particular to an implementation *)
module type POLICY =
  sig
    (* Type for the user's key *)
    type key = Dsa.public_key
    type privkey = Dsa.private_key
    type cert = string

    (* For default policies *)
    val default_principal : key
    val default_principal_privkey : privkey

    (* Namespace authorization action *)
    val construct_all_service_lists : unit -> string list
    val construct_service_lists : key -> (cert list) option ->
      (string list * string list)

    (* Per-service authorization action *)
    val query_service_parameter: key -> string -> string ->
      (cert list) option -> string

    (* Initialization *)
    val init : string -> int -> key -> privkey -> unit
  end
```

The implementation of this signature provides privilege-based service access and use. The first three type definitions indicate that public and private keys are implemented using DSA, and that certificates are in string form. The next two definitions define the principal corresponding to the default (unauthenticated) user. The next three definitions indicate the authorization actions. The first two functions deal with service access management while the third is for per-usage queries. The first function, `construct_all_service_lists`, returns all services which are privileged beyond the default user; this is used by the node to prevent these services from being added to the default service symbol table at startup.[3] The second function, `construct_service_lists`, is called to alter

[3]In effect, this defines the default environment as all services registered by the node minus the ones indicated as privileged by the node policy. However, this implies that these restrictions may be known statically, which is true for the monolithic and hierarchical imple-

the namespace with the additional services allowed to the authenticated principal and those restricted from him (usually this will be one or the other, but not both). This function may optionally take a list of certificates to provide further information. The third function, `query_service_parameter`, is used to return service parameters: given a user's public key, service name, and service parameter name, it returns, in string form, the value of the specified parameter. Finally, the initialization function, `init`, initializes the node policy, given the policy file, port number, and public and private keys of the node.

In the PLAN distribution, we provide two implementations of this signature, one using a monolithic approach, and another allowing all possible policy storage strategies, using the Query Certificate Manager (QCM).

3.1 Monolithic Approach

By default, PLAN is compiled to use our monolithic, set-based approach to policy management. It is not expected that this approach scales well to real-world situations; it was designed as a testbed for ideas. In fact, we have also developed a translator to QCM for policy files written for the monolithic approach. We present a basic example here.

The form of the policy file is as a list of set definitions with key associations. For example:

```
p1 = ( print ) + { "public.key.3324" };
```

creates a set named `p1` that specifies that the `print` service routine is privileged and may be accessed by the prinicipal whose public key is stored in file `public.key.3324`, implying that it is unavailable by default. By contrast:

```
p1 = ( print ) - { "public.key.3324" };
```

indicates that the `print` service is *not* available to the given principal, implying that it is available by default. This is useful for our active firewall, discussed in Section 3.3.2. Finally, sets can be specified in relation to each other, as in

```
p1 = ( print ) + { "public.key" };
privileged = ( thisHost ) union p1 +
  { "public.key.3325", "public.key.3324" };
```

This indicates that the privilege afforded to prinicipals associated with set `privileged` is a superset of the privilege afforded to those associated with set `p1`.

This approach does not currently implement usage parameters; thus the function `query_service_parameter` is not implemented. This is, however, implemented with QCM as explained below.

mentations, but not for ephemeral ones. Thus, to allow for this possibility, we should alter our implementation to specify the default service environment explicitly rather than implicitly.

3.2 Query Certificate Manager

Also provided with the distribution is some code which implements the `POLICY` signature using the Query Certificate Manager (QCM). QCM is a natural choice for a policy implementation because it is based on sets, and supports all three implementation approaches described above (monolithic, hierarchical, and ephemeral). Please see the QCM documentation in the distribution for more specific information about QCM; it's in `plan/qcmlib/docs`. By default, `plan` is built with QCM support; if you wish to not use QCM, you should comment out the line

```
USE_QCM = true
```

in the Makefile before building plan.

The QCM policy checker is implemented in the file `security/policy_qcm.ml`. The implementation strategy for access is to define a QCM program with a set named `acl`. This set consists of a list of three-tuples consisting of

1. a set of keys indicating the principals this tuple applies to
2. a thicken set of privileged services to add to the principals' environment
3. a thin set of services to subtract from the principals' environment

Using this approach, the last example in the previous section could be encoded in QCM as:

```
online
{
  p1 = {
    Principal(<PublicKey="...">)
  };
  p10 = {"print"};

  privileged = {
    Principal(<PublicKey="...">)
  };
  privileged0 = {"thisHost"};

  acl = {
    ( p1, p10, {} ),
    ( privileged, union ( privileged0, p10 ), {} )
  };
}
```

Note that QCM programs may refer to QCM programs on other nodes to implement a distributed, hierarchical program. Details are omitted here; see the QCM documentation for more.

For usage-based security, you simply add set definitions such that the name of the set is the name of the service to parameterize. The elements of the set are two-tuples of the form

1. a set of principals (as in the acl)
2. a labeled record of length 1 with the label corresponding to the parameter name

To have multiple parameters per service, you add multiple records.[4]

As an example, the `resident` state package implements the ability for packets to store soft state on the router. PLAN 3.2 allows you to create a policy about how much particular users are allowed to store. For example, the above program might be extended as follows:

```
default = {
  Principal(<PublicKey="...">)
};
resident = { ( default, <amount=100> ),
             ( privileged, <amount=1000> ) };
```

This indicates that all of the principals in `default` and `privileged` are allowed to store 100 and 1000 words of information, respectively, at any one time. The value of the default key should match `Policy_impl.default_principal` (see `security/policy_qcm.ml` for the exact string value). This policy is enforced by the resident state implementation itself. If we look in the file `resident/resident_svc_impl.ml` at the implementation of the function `put`, we see[5]

```
exception TooMuch

(* check_put: 'a -> string *)
(* Code to implement restriction of service policy *)
let check_put v =
  let whokey =
      (match (Auth.get_principal()) with
        Some k -> k
      | None -> Policy_impl.default_principal) in
  let who = Dsa.string_of_public_key whokey in
  let max_amount = int_of_string
      (Policy_impl.query_service_parameter
        whokey "resident" "amount" None) in
  let current_amount = space_used who in
  let requested_amount = Obj.size (Obj.repr v) in
  if (current_amount + requested_amount) > max_amount then
    raise TooMuch
  else
    who
```

[4] This is an artifact of the fact that QCM does not allow *width-subtyping* in record pattern-matches. The QCM developers assure us that this feature will be added at some point, at which time we'll alter our policy implementation.

[5] The actual code contains `ifdef`'s that are needed for debugging and because usage-based security is not implemented without QCM; these are elided here for clarity.

```
let put key v timeout =
  let who = check_put v in
  (* Actual service code ... *)
```

The first thing that happens in `put` is that the function `check_put` is called to determine whether the action is permitted. It first checks who the current principal is from the `Auth.get_principal()` call; this value is stored after a packet does an `authEval()`. If no `authEval` has taken place, `get_principal` returns `None` and so the default principal is used. The next thing is that the policy is queried using `Policy_impl.query_service_parameter` to determine the maximum allowable amount for this prinicpal. Then, the code looks up in the state table how much this principal currently has stored and if that combined with the requested amount is less than the value dictated by policy, then the operation is permitted (returning the current principal to be used by `put`). Otherwise, an exception is raised.

In the next subsection we provide an example implementation and service for use with `authapp` to demonstrate the use of QCM in the monolithic setting.

3.3 Authorization Tutorial

We present three demonstrations of authorization. First, we present the use of privilege-expansion so that a privileged packet may have access to a greater number of services. Next we present the use of privilege-restriction to limit the default services available to a packet; this allows us to implement an active firewall. Finally, we present usage-based security using resident state, as described above.

3.3.1 Application Privilege Expansion

As a demonstration of namespace-expansion authorization, we have provided a sample QCM program in the file `expand.qcm` in the `plan` directory:

```
online
{
  enhancements = {
    Principal(<PublicKey="P: iuWJZ7KVoRbhoCDrDqUSo7pd/hsYFvs9E5rk2
JYB2W7M06F2ughbk8zaIzgiVriopSMhZziJXyvs6fla/6QzRIwMwhLtTUzAIQZYDUD
duJjClHZzU8lN7FM4jGNyq3wOn/uH/L9i3tJGm/V9ywfqqUFNFOO2amlxhM1s1JMci
kM=, Q: ledWdPQsmKTqyynTCBnOvWlhFeM=, ALPHA: D7CquhERwI22+bW0JfY9o
D+tLiV7+0p+5y12Mvx/RYsQgwmA1BUD6PqyZ54lXRh3hm6k/AM9zvYXbcG8Dx49uq6
jQXHdjQCK2c8rlztrjplJvPdG5bSMTZzbVJAY1oDM9sugFGmMJLvizqFsZfJsrE13P
pOTNc7LRS8b26NodUM=, Y: Ow++XUNYt5JnxuQ4WgjvESddl+CKV6vEE6SGSu+mbI
ubArSzYUV2OBYdMU6uqZSaPxX0lSIDrHC+Tvq4ckNU+U5LciYDydgfUxFQQXbLInwS
UI2iaus9jGVP5Px8gASIkhy0eKQg45qZHySalBpuA6AyDlHqRMIR7IK8zV2XKcQ=">)
  };
  enhancements0 = {"authSvc"};
```

```
  acl = {
    ( enhancements, enhancements0, {} )
  };
}
```

It defines a single set named `enhancements` with a single member (which is the pair of keys in the file `key` used for `authapp` in the last tutorial). Then this user authenticates on the node using this policy, the service `authSvc` will be added to his namespace. This service is defined in `security/crypto_svc_impl.ml` as the function `auth_svc`, and always returns the value `true`.

We can try this service out with a slight modification to `authapp`. In the file `security/authapp.ml`, on line 266 you'll find three blocks of PLAN code:

```
      (let chunk_code = Printf.sprintf
        "svc print : 'a -> unit
         svc toString : 'a -> string
         svc getHostByName : string -> host
         svc defaultRoute : host -> host * dev
         svc getRB : void -> int
         svc thisHost : void -> host list
         fun doit() =
           OnRemote(|print|(toString(hd thisHost()) ^
                            \" authenticated successfully\"),
                    getHostByName(\"%s\"),getRB(),defaultRoute)"
(*        "svc print : 'a -> unit
         svc getHostByName : string -> host
         svc defaultRoute : host -> host * dev
         svc getRB : void -> int
         svc authSvc : void -> bool
         fun doit() = OnRemote(|print|(authSvc()),
                               getHostByName(\"%s\"),getRB(),
                               defaultRoute)"
        "svc print : 'a -> unit
         svc toString : 'a -> string
         svc put : (string,key,'a,int) -> unit
         svc generateKey : void -> key
         svc thisHost : void -> host list
         svc getHostByName : string -> host
         svc defaultRoute : host -> host * dev
         svc getRB : void -> int
         fun store(i:int list) =
           let val session = generateKey()
               fun mark(a: int, f:unit) =
             put(\"mark\"^toString(a),session,true,20)
           in
             (foldr(mark,i,());
             OnRemote(|print|(toString(hd thisHost()) ^
```

```
                              \" stored data successfully\"),
                    getHostByName(\"%s\"),getRB(),defaultRoute))
         end
       fun doit() = store([1;2;3;4;5])" *)
           (Activehost.activeHostToString my_host_addr) in
```

Comment out the first block of PLAN code (a string) and uncomment the second:

```
      (let chunk_code = Printf.sprintf
(*        "svc print : 'a -> unit
         svc toString : 'a -> string
         svc getHostByName : string -> host
         svc defaultRoute : host -> host * dev
         svc getRB : void -> int
         svc thisHost : void -> host list
         fun doit() =
           OnRemote(|print|(toString(hd thisHost()) ^
                            \" authenticated successfully\"),
                   getHostByName(\"%s\"),getRB(),defaultRoute)" *)
        "svc print : 'a -> unit
         svc getHostByName : string -> host
         svc defaultRoute : host -> host * dev
         svc getRB : void -> int
         svc authSvc : void -> bool
         fun doit() = OnRemote(|print|(authSvc()),
                               getHostByName(\"%s\"),getRB(),
                               defaultRoute)"
(*        "svc print : 'a -> unit
         svc toString : 'a -> string
         svc put : (string,key,'a,int) -> unit
         svc generateKey : void -> key
         svc thisHost : void -> host list
         svc getHostByName : string -> host
         svc defaultRoute : host -> host * dev
         svc getRB : void -> int
         fun store(i:int list) =
           let val session = generateKey()
               fun mark(a: int, f:unit) =
             put(\"mark\"^toString(a),session,true,20)
           in
             (foldr(mark,i,());
             OnRemote(|print|(toString(hd thisHost()) ^
                              \" stored data successfully\"),
                     getHostByName(\"%s\"),getRB(),defaultRoute))
           end
         fun doit() = store([1;2;3;4;5])" *)
```

```
            (Activehost.activeHostToString my_host_addr) in
```

Then recompile authapp.

Now, we must start node $m1$ slightly differently, so that it will use expand.qcm. To start it, do:

bin/pland -l log3324 -ip 3324 -policy expand.qcm $m1$

Now start authapp as before. After it has authenticated with the remote node, rather than simply authenticating and printing the message "authenticated successfully," the program will access this service which requires privilege. If the principal authenticated with is in the privileged set, the message true will be printed as the last of authapp's output; this should happen if you specify key as the app's keys as in the last tutorial:

bin/authapp -p 3324 m key

To see the authorization fail, have authapp use the keys in key.3325, also present in the plan directory:

bin/authapp -p 3324 m key.3325

In this case, ERROR: will be printed in authapp, and the log file for $m1$ will contain:

```
Call: No such function authSvc
Eval thread 4 exiting with failure:
  Uncaught packet exception ServiceNotPresent
```

3.3.2 An Active Firewall

The active firewall is described in more detail in [7]. The basic idea is that it wraps packets coming in from a protected interface with a chunk that evaluates the original chunk associated with a *guest* principal. The local policy restricts the namespace of this guest principal, thus limiting the effect of external packets.

To demonstrate the use of the firewall, we need to set up a three-node network, as depicted in Figure 1. Node $m3$ in the untrusted network is started as normal:[6]

bin/pland -l log3326 -ip 3326 $m3$

where $m3$ contains:
1
ip0 ip m:3326 m:3327

[6] In all of these examples, m is the machine on which this example is run—i.e. your machine's hostname, as understood by the PLAN DNS service. See the tutorial or the FAQ if you run into problems.

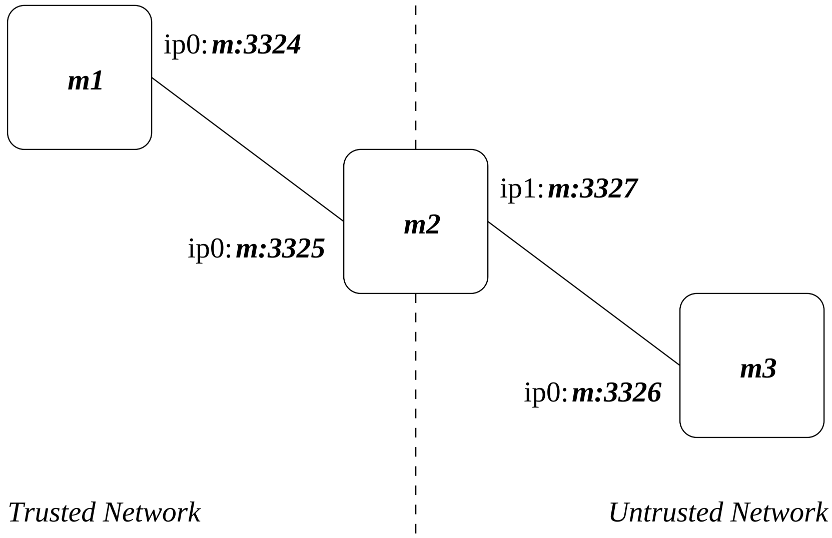

Figure 1: *Firewall Tutorial*: Shows three nodes: m1 is the source node, inside the trusted network; m2 is the firewall; and m3 is the unstrusted node. Each connection is labelled by the device name and that device's address, as specified in the interface configuration files for the three respective nodes.

The middle node, $m2$, serves as the firewall. It must be started with one additional parameter:

`bin/pland -l log3325 -ip 3325 -firewall ip0` $m2$

where $m2$ contains:
```
2
ip0 ip m:3325 m:3324
ip1 ip m:3327 m:3326
```

The `-firewall` parameter indicates that `pland` should function as a firewall rather than a router. The arguments given to this option specify which devices are being protected by the firewall. All packets that are emitted via these devices are first wrapped with a privilege-reducing chunk. Here we specify `ip0` as that is the device that leads to the trusted network (see the Figure).

Finally, the trusted node $m1$ is started as:

`bin/pland -l log3324 -ip 3324 -policy firewall.qcm -authlist` m`:3325` $m1$

where $m1$ contains:
```
1
ip0 ip m:3324 m:3325
```

There are two additional parameters. The first parameter `-policy` takes a policy file that specifies which services should be removed from the the guest environment. For this tutorial, we use the file `firewall.qcm`:

```
online
{
  restrictions = {
    Principal(<PublicKey="...">)
  };
  restrictions0 = {"DevNotLocal", "FlowNoRoute", "getToll", "setTo
ll", "flowSet", "flow", "RIPNoRoute", "RIPGetRoutes", "RIPRecv", "
getNeighbors", "RIPGetNeighbors", "RIP", "DHmessageThree", "DHmess
ageTwo", "DHmessageOne", "authSvc", "uncompress", "compress", "set
Percent", "setUncompDelay", "setCompDelay", "getHostByName", "Load
GenNotStarted", "completeFlow", "startFlow", "stopSender", "demoSh
owHop", "flowUseNewFlow", "flowNewFlowReady", "flowNewScout", "flo
wGo", "flowKeyReady", "flowStartToDemo", "flowStartToPort", "flowS
tart", "startSenderToDemo", "startSenderToPort", "startSender", "s
tartReceiverDemo", "NonLocalHost", "delete", "genKeyHost", "genera
teKey", "setGT", "setLT", "put", "get", "BindFailed", "NoBinding",
 "retrieveBinding", "bind", "openPort", "abort", "evalBlob", "eval
", "getMTU", "getDevs", "getSrcDev", "getStaticRoutes", "staticRou
te", "remove", "member", "length", "gettimeofday", "timestamp", "t
hisHost", "getSessionKey", "getSrc", "getSource", "logGC", "queryG
C", "exit"};
  acl = {
    ( restrictions, {}, restrictions0 ),
  };
}
```

This defines s set of principals called `restrictions`. The sole member of the `restrictions` set is the firewall's public key (as stored in the file `key.3325`. This set is stored in the ACL with the thin set as defined by `restrictions0`. To protect the trusted node using this policy, the firewall wraps incoming packets with an `authEval` that changes the identity to that of the firewall (serving as the *guest* identity). Packets running with this principal will have the thinned environment. In order for the firewall to sign incoming packets, the trusted host must authenticate with the firewall. This is done via the `-authlist` parameter, which takes a comma-separated list of nodes to be authenticated with. The firewall is specified in this list.

Once all of these nodes are started, $m1$ will authenticate with the $m2$, just as we saw in Section 2.3.2. Once this authentication is complete, the stage is set to try running some packets. To show how privilege is restricted, we send the `Helloworld` program from $m3$ to $m1$ through the firewall. The policy specified in `firewall.qcm` indicates that the `thisHost` service should be removed. Indeed, when we do:

bin/inject -p 3326 -ed m:3324 interp_tests/Helloworld.plan 10

```
doit();
```

The result is

```
1.1-1.6: Warning: service doit not declared.
ERROR: <chunk:
doit[]> raised ServiceNotPresent on (127.0.0.1,3324)
```

and at the end of the log file for $m1$ we see

```
Call: No such function thisHost
Eval thread 1 exiting with failure:
  Uncaught packet exception ServiceNotPresent
```

On the other hand, if we run a program, such as `ping`, that does not use any thinned services (e.g. `thisHost`) on $m1$, things should work OK. Indeed, if we run `bin/planping`:

```
bin/planping m:3326
PING    0       483.964920 ms
PING    1       21.622062 ms
PING    2       21.566033 ms
...
```

3.3.3 Privileged Usage

As described in the previous section, we have implemented privileged usage for the *resident state* package. In this section we demonstrate the how privileged service usage works using this service suite. The policy file that we use for $m1$ in this case is `usage.qcm`:

```
online
{
  privileged = {
    Principal(<PublicKey="P: iuWJZ7KVoRbhoCDrDqUSo7pd/hsYFvs9E5rk2
JYB2W7M06F2ughbk8zaIzgiVriopSMhZziJXyvs6fla/6QzRIwMwhLtTUzAIQZYDUD
duJjClHZzU8lN7FM4jGNyq3wOn/uH/L9i3tJGm/V9ywfqqUFNF0O2amlxhM1s1JMci
kM=, Q: ledWdPQsmKTqyynTCBn0vWlhFeM=, ALPHA: D7CquhERwI22+bW0JfY9o
D+tLiV7+0p+5y12Mvx/RYsQgwmA1BUD6PqyZ54lXRh3hm6k/AM9zvYXbcG8Dx49uq6
jQXHdjQCK2c8rlztrjplJvPdG5bSMTZzbVJAY1oDM9sugFGmMJLvizqFsZfJsrE13P
pOTNc7LRS8b26NodUM=, Y: Ow++XUNYt5JnxuQ4WgjvESddl+CKV6vEE6SGSu+mbI
ubArSzYUV2OBYdMU6uqZSaPxX0lSIDrHC+Tvq4ckNU+U5LciYDydgfUxFQQXbLInwS
UI2iaus9jGVP5Px8gASIkhy0eKQg45qZHySalBpuA6AyDlHqRMIR7IK8zV2XKcQ=">)
  };

  acl = {
    ( privileged, {}, {} )
  };
```

```
  # usage-based security
  # this key matches Policy_qcm.Policy_impl.default_principal
  default = {
    Principal(<PublicKey="P: 6e+1xVU3D5dHFrZo+Cd2YsR/CxvFVcDyAImn5
nESz17GjBurur3xrEVj7XV+wpS7N2XtZ0QPeNoUId1RRXAY2qDK8Cc7WyofsomqCzW
5Sdz4d4sa45J2ur/+pdiZtQEnfGoYd15mUPhShE5BtfaY0VrBzrfX7pqIPLhq476zo
bs=, Q: 2dfTnTvli0m4HzCh9rlIaKT2alk=, ALPHA: 0XmMrRFwHdRY2i/ya7d1m
urBUkhJwc+H4lJTpZ7LJNKc0s3+HolntEwtWi05B40wwS02Pl5B7azZxm23zFXUJVX
QATGEH13XsX+BEaxrIMj6Vh+d0w5B3286wjhAm/lgyglrSsl20BNqNeW42zTMCoYQe
Frs+vI4Z10Zx8yCMmY=, Y: D+4Mymn0id6KMo0gZjappc1Azd/qM3LI2A8aaTERym
YQe0bqULkLxX/SiYkGaLZNJ9LJ2Iy9EQGiG2NPEaC+nENHnYYDfSK0XjaaeC0aulRE
ky0QVK5qvvXql3lYAbinnBMRrJx5nZoL2Te1VAW/KY5JNM9riUSp6KdKKnT+dE4=">)
  };
  resident = { ( default, <amount=4> ),
               ( privileged, <amount=1000> ) };
}
```

In this file, two principal sets are defined: one contains the default principal (matching that defined in `Policy_qcm.Policy_impl.default_principal`) and the other contains a privileged principal. The policy is implemented by the `resident` set: it specifies that the default principal may store (at any one time) at most 4 words of soft state, while the privileged user may store 1000 words.[7]

To demonstrate the use of this policy, we've provided a PLAN program `interp_tests/store.plan` which stores the value `true` on the node 5 times, exceeding the policy for the default user, but well under it for the privileged one. This code has also been incorporated into `authapp`; you need to uncomment the last region of code:

```
      (let chunk_code = Printf.sprintf
(*        "svc print : 'a -> unit
         svc toString : 'a -> string
         svc getHostByName : string -> host
         svc defaultRoute : host -> host * dev
         svc getRB : void -> int
         svc thisHost : void -> host list
         fun doit() =
           OnRemote(|print|(toString(hd thisHost()) ^
                             \" authenticated successfully\"),
                    getHostByName(\"%s\"),getRB(),defaultRoute)" *)
        "svc print : 'a -> unit
         svc getHostByName : string -> host
         svc defaultRoute : host -> host * dev
         svc getRB : void -> int
```

[7]This is, in fact, not a very reasonable policy: the default user should get much more state, since it represents all untrusted users of the network. We use this policy for demonstration purposes.

```
     svc authSvc : void -> bool
     fun doit() = OnRemote(|print|(authSvc()),
                           getHostByName(\"%s\"),getRB(),
                           defaultRoute)" *)
    "svc print : 'a -> unit
     svc toString : 'a -> string
     svc put : (string,key,'a,int) -> unit
     svc generateKey : void -> key
     svc thisHost : void -> host list
     svc getHostByName : string -> host
     svc defaultRoute : host -> host * dev
     svc getRB : void -> int
     fun store(i:int list) =
       let val session = generateKey()
           fun mark(a: int, f:unit) =
         put(\"mark\"^toString(a),session,true,20)
       in
         (foldr(mark,i,());
         OnRemote(|print|(toString(hd thisHost()) ^
                         \" stored data successfully\"),
                  getHostByName(\"%s\"),getRB(),defaultRoute))
       end
     fun doit() = store([1;2;3;4;5])"
         (Activehost.activeHostToString my_host_addr) in
```

Let us first run this as the default user. Start up the node, using the `usage.qcm` policy:

`bin/pland -l log3324 -ip 3324 -policy usage.qcm` $m1$

Then, inject the `store.plan` program with initial invocation `doit`:

`bin/inject -p 3326 -ed` m`:3324 interp_tests/Helloworld.plan 10`
`doit();`

The result is

```
1.1-1.6: Warning: service doit not declared.
ERROR: <chunk:
doit[]> raised InsufficientPrivilege on (127.0.0.1,3324)
```

and at the end of the log file for $m1$ we see


```
Principal |P: 6e+1xVU3D5dHFrZo+Cd2YsR/CxvFVcDyAImn5nESz17GjBurur3x
rEVj7XV+wpS7N2XtZ0QPeNoUId1RRXAY2qDK8Cc7WyofsomqCzW5Sdz4d4sa45J2ur
/+pdiZtQEnfGoYd15mUPhShE5BtfaY0VrBzrfX7pqIPLhq476zobs=, Q: 2dfTnTv
li0m4HzCh9rlIaKT2alk=, ALPHA: 0XmMrRFwHdRY2i/ya7d1murBUkhJwc+H4lJT
pZ7LJNKc0s3+HolntEwtWi05B40wwS02Pl5B7azZxm23zFXUJVXQATGEH13XsX+BEa
xrIMj6Vh+d0w5B3286wjhAm/lgyglrSsl20BNqNeW42zTMCoYQeFrs+vI4Z10Zx8yC
```

```
MmY=, Y: D+4Mymn0id6KMo0gZjappc1Azd/qM3LI2A8aaTERymYQe0bqULkLxX/Si
YkGaLZNJ9LJ2Iy9EQGiG2NPEaC+nENHnYYDfSKOXjaaeCOaulREkyOQVK5qvvXql3l
YAbinnBMRrJx5nZoL2Te1VAW/KY5JNM9riUSp6KdKKnT+dE4=|
  Allowed=4
  Current=4
  Request=1
Eval thread 4 exiting with failure:
  Uncaught packet exception InsufficientPrivilege
```

Now let's try this program with greater privilege. Start `authapp` as before, using the keyfile `key`. Now, after it has authenticated with the remote node, it will run the store program, but it will complete successfully, printing on authapp:

```
127.0.0.1:3324 stored data successfully
```

3.4 Implementation Notes

To use QCM in a hierarchical fashion, it must be possible for multiple PLAN evaluations to execute concurrently. Due to overheads in the runtime system, this causes a significant performance hit, so this functionality is not enabled by default. To enable it, you need to recompile plan with the `-DMULTI_EVAL` flag added to the `CAMLP4FLAGS` variable in the Makefile.

QCM has its own notion of the node's identity as specified in the QCM program at startup. If a public/private key pair is specified in the QCM program, it will be overridden by any keys given explicitly to `pland` at startup.

4 Future Directions

We have established the base foundation for security services in PLAN. A number of possible implementation improvements could be made:

- Allow a signature to `authEval` consist instead of a public key in the case that privilege is *only* being restricted. In other words, if a principal asks for less than default privilege, it should be granted without question. This would improve the performance of the firewall, for example.
- Improve the methodolody for per-use security. The problem with the current approach is that security enforcement code must pollute the implementation of the service. For example, the `check_put` function had to be added to enforce the resident service policy. This could be improved in two ways:
    1. Switch to a policy manager that answers more expressive queries, such as Keynote [8]. In Keynote, most of the code in the `check_put` function could be encoded in the query as either boolean data or callbacks. This causes less intrusion in the service code base.

2. Automatic insertion of policy checking code. This, again, causes less intrusion into normal service code.

References

[1] M.W. Hicks "PLAN System Security," *University of Pennsylvania Technical Report # MS-CIS-98-25*, `http://www.cis.upenn.edu/~switchware/papers/plan_security.ps`, July 1998.

[2] J.T. Moore and M.W. Hicks, "PLAN Programmer's Guide," PLAN documentation.

[3] P. Kakkar and M. Hicks, "PLAN Tutorial," PLAN documentation.

[4] W. Diffie and M. E. Hellman. "New Directions in Cryptography." *IEEE Transactions on Information Theory.* vol IT-22, no. 6, pp. 664-654, November 1976.

[5] "Digital Signature Standard," Tech. Rep. FIPS-186, U.S. Department of Commerce, May 1994.

[6] H. Krawczyk, M. Bellare and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication." *IETF RFC 2104*, February 1997.

[7] M. Hicks and A.D. Keromytis, "A Secure PLAN" *First International Working conference on Active Networks*, Springer-Verlag, July 1999.

[8] Blaze, M., Feigenbaum, J., Ioannidis, J. and Keromytis, A. "The Role of Trust Management in Distributed Systems Security." In *Secure Internet Programming.*