USENIX Association

Proceedings of the 11th USENIX Security Symposium

San Francisco, California, USA
August 5-9, 2002

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

© 2002 by The USENIX Association All Rights Reserved For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738 Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer.
Permission is granted for noncommercial reproduction of the work for educational or research purposes.
This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

Security in Plan 9

Russ Cox, *MIT LCS*
Eric Grosse, *Bell Labs*
Rob Pike, *Bell Labs*
Dave Presotto, *Avaya Labs and Bell Labs*
Sean Quinlan, *Bell Labs*
`{rsc,ehg,rob,presotto,seanq}@plan9.bell-labs.com`

Abstract

The security architecture of the Plan 9™ operating system has recently been redesigned to address some technical shortcomings. This redesign provided an opportunity also to make the system more convenient to use securely. Plan 9 has thus improved in two ways not usually seen together: it has become more secure *and* easier to use.

The central component of the new architecture is a per-user self-contained agent called `factotum`. `Factotum` securely holds a copy of the user's keys and negotiates authentication protocols, on behalf of the user, with secure services around the network. Concentrating security code in a single program offers several advantages including: ease of update or repair to broken security software and protocols; the ability to run secure services at a lower privilege level; uniform management of keys for all services; and an opportunity to provide single sign on, even to unchanged legacy applications. `Factotum` has an unusual architecture: it is implemented as a Plan 9 file server.

1. Introduction

Secure computing systems face two challenges: first, they must employ sophisticated technology that is difficult to design and prove correct; and second, they must be easy for regular people to use. The question of ease of use is sometimes neglected, but it is essential: weak but easy-to-use security can be more effective than strong but difficult-to-use security if it is more likely to be used. People lock their front doors when they leave the house, knowing full well that a burglar is capable of picking the lock (or avoiding the door altogether); yet few would accept the cost and awkwardness of a bank vault door on the house even though that might reduce the probability of a robbery. A related point is that users need a clear model of how the security operates (if not how it actually provides security) in order to use it well; for example, the clarity of a lock icon on a web browser is offset by the confusing and typically insecure steps for installing X.509 certificates.

The security architecture of the Plan 9 operating system [11] has recently been redesigned to make it both more secure and easier to use. By *security* we mean three things: first, the business of authenticating users and services; second, the safe handling, deployment, and use of keys and other secret information; and third, the use of encryption and integrity checks to safeguard communications from prying eyes.

The old security architecture of Plan 9 had several engineering problems in common with other operating systems. First, it had an inadequate notion of security domain. Once a user provided a password to connect to a local file store, the system required that the same password be used to access all the other file stores. That is, the system treated all network services as belonging to the same security domain.

Second, the algorithms and protocols used in authentication, by nature tricky and difficult to get right, were compiled into the various applications, kernel modules, and file servers. Changes and fixes to a security protocol required that all components using that protocol needed to be recompiled, or at least relinked, and restarted.

Third, the file transport protocol, 9P [12], that forms the core of the Plan 9 system, had its authentication protocol embedded in its design. This meant that fixing or changing the authentication used by 9P required deep changes to the system. If someone were to find a way to break the protocol, the system would be wide open and very hard to fix.

These and a number of lesser problems, combined with a desire for more widespread use of encryption in the system, spurred us to rethink the entire security architecture of Plan 9.

The centerpiece of the new architecture is an agent, called `factotum`, that handles the user's keys and negotiates all security interactions with system services and applications. Like a trusted assistant with a copy of the owner's keys, `factotum` does all the negotiation for security and authentication. Programs no longer need to be compiled with cryptographic code; instead they communicate with `factotum` agents that represent distinct entities in the cryptographic exchange, such as a user and server of a secure service. If a security protocol needs to be added, deleted, or modified, only `factotum` needs to be updated for all system services to be kept secure.

Building on `factotum`, we modified secure services in the system to move user authentication code into `factotum`; made authentication a separable component of the file server protocol; deployed new security protocols; designed a secure file store, called `secstore`, to protect our keys but make them easy to get when they are needed; designed a new kernel module to support transparent use of Transport Layer Security (TLS) [3]; and began using encryption for all communications within the system. The overall architecture is illustrated in Figure 1a.

Secure protocols and algorithms are well understood and are usually not the weakest link in a system's security. In practice, most security problems arise from buggy servers, confusing software, or administrative oversights. It is these practical problems that we are addressing. Although this paper describes the algorithms and protocols we are using, they are included mainly for concreteness. Our main intent is to present a simple security architecture built upon a small trusted code base that is easy to verify (whether by manual or automatic means), easy to understand, and easy to use.

Although it is a subjective assessment, we believe we have achieved our goal of ease of use. That we have achieved our goal of improved security is supported by our plan to move our currently private computing environment onto the Internet outside the corporate firewall. The rest of this paper explains the architecture and how it is used, to explain why a system that is

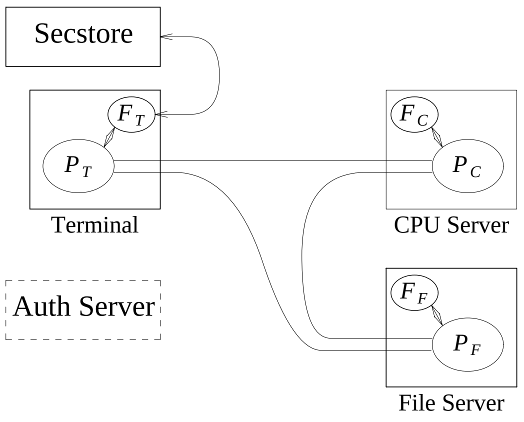

Figure 1a. Components of the security architecture. Each box is a (typically) separate machine; each ellipse a process. The ellipses labeled F_X are `factotum` processes; those labeled P_X are the pieces and proxies of a distributed program. The authentication server is one of several repositories for users' security information that `factotum` processes consult as required. `Secstore` is a shared resource for storing private information such as keys; `factotum` consults it for the user during bootstrap.

easy to use securely is also safe enough to run in the open network.

2. An Agent for Security

One of the primary reasons for the redesign of the Plan 9 security infrastructure was to remove the authentication method both from the applications and from the kernel. Cryptographic code is large and intricate, so it should be packaged as a separate component that can be repaired or modified without altering or even relinking applications and services that depend on it. If a security protocol is broken, it should be trivial to repair, disable, or replace it on the fly. Similarly, it should be possible for multiple programs to use a common security protocol without embedding it in each program.

Some systems use dynamically linked libraries (DLLs) to address these configuration issues. The problem with this approach is that it leaves security code in the same address space as the program using it. The interactions between the program and the DLL can therefore accidentally or deliberately violate the interface, weakening security. Also, a program using a library to implement secure services must run at a privilege level necessary to provide the service; separating the security to a different program makes it possible to run the services at a weaker privilege level, isolating the privileged code to a single, more trustworthy component.

Following the lead of the SSH agent [20], we give each user an agent process responsible for holding and using the user's keys. The agent program is called `factotum` because of its similarity to the proverbial servant with the power to act on behalf of his master because he holds the keys to all the master's possessions. It is essential that `factotum` keep the keys secret and use them only in the owner's interest. Later we'll discuss some changes to the kernel to reduce the possibility of `factotum` leaking information inadvertently.

`Factotum` is implemented, like most Plan 9 services, as a file server. It is conventionally mounted upon the directory `/mnt/factotum`, and the files it serves there are analogous to virtual devices that provide access to, and control of, the services of the `factotum`. The next few sections describe the design of `factotum` and how it operates with the other pieces of Plan 9 to provide security services.

2.1. Logging in

To make the discussions that follow more concrete, we begin with a couple of examples showing how the Plan 9 security architecture appears to the user. These examples both involve a user `gre` logging in after booting a local machine. The user may or may not have a secure store in which all his keys are kept. If he does, `factotum` will prompt him for the password to the secure store and obtain keys from it, prompting only when a key isn't found in the store. Otherwise, `factotum` must prompt for each key.

In the typescripts, `\n` represents a literal newline character typed to force a default response. User input is in italics, and long lines are folded and indented to fit.

This first example shows a user logging in without help from the secure store. First, `factotum` prompts for a user name that the local kernel will use:

```
user[none]: gre
```

(Default responses appear in square brackets.) The kernel then starts accessing local resources and requests, through `factotum`, a user/password pair to do so:

```
!Adding key: dom=cs.bell-labs.com
    proto=p9sk1
user[gre]: \n
password: ****
```

Now the user is logged in to the local system, and the mail client starts up:

```
!Adding key: proto=apop
    server=plan9.bell-labs.com
user[gre]: \n
password: ****
```

`Factotum` is doing all the prompting and the applications being started are not even touching the keys. Note that it's always clear which key is being requested.

Now consider the same login sequence, but in the case where `gre` has a secure store account:

```
user[none]: gre
secstore password: *********
STA PIN+SecurID: *********
```

That's the last `gre` will hear from `factotum` unless an attempt is made to contact a system for which no key is kept in the secure store.

2.2. The factotum

Each computer running Plan 9 has one user id that owns all the resources on that system — the scheduler, local disks, network interfaces, etc. That user, the *host owner*, is the closest analogue in Plan 9 to a Unix `root` account (although it is far weaker; rather than having special powers, as its name implies the host owner is just a regular user that happens to own the resources of the local machine). On a single-user system, which we call a terminal, the host owner is the id of the terminal's user. Shared servers such as CPU servers normally have a pseudo-user that initially owns all resources. At boot time, the Plan 9 kernel starts a `factotum` executing as, and therefore with the privileges of, the host owner.

New processes run as the same user as the process which created them. When a process must take on the identity of a new user, such as to provide a login shell on a shared CPU server, it does so by proving to the host owner's `factotum` that it is authorized to do so. This is done by running an authentication protocol with `factotum` to prove that the process has access to secret information which only the new user should possess. For example, consider the setup in Figure 1a. If a user on the terminal wants to log in to the CPU server using the Plan 9 `cpu` service [12], then P_T might be the `cpu` client program and P_C the `cpu` server. Neither P_C nor P_T knows the details of the authentication. They do need to be able to shuttle messages back and forth between the two `factotums`, but this is a generic function easily performed without knowing, or being able to extract, secrets in the messages. P_T will make a network connection to P_C. P_T and P_C will then relay messages between the `factotum` owned by the user, F_T, and the one owned by the CPU server, F_C, until mutual authentication has been established. Later sections describe the RPC between `factotum` and applications and the library functions to support proxy operations.

The kernel always uses a single local instance of `factotum`, running as the host owner, for its authentication purposes, but a regular user may start other `factotum` agents. In fact, the `factotum` representing the user need not be running on the same machine as its client. For instance, it is easy for a user on a CPU server, through standard Plan 9 operations, to replace the `/mnt/factotum` in the user's private file name space on the server with a connection to the `factotum` running on the terminal. (The usual file system permissions prevent interlopers from doing so maliciously.) This permits secure operations on the CPU server to be transparently validated by the user's own `factotum`, so secrets need never leave the user's terminal. The SSH agent [20] does much the same with special SSH protocol messages, but an advantage to making our agent a file system is that we need no new mechanism to access our remote agent; remote file access is sufficient.

Within `factotum`, each protocol is implemented as a state machine with a generic interface, so protocols are in essence pluggable modules, easy to add, modify, or drop. Writing a message to and reading a message from `factotum` each require a separate RPC and result in a single state transition. Therefore `factotum` always runs to completion on every RPC and never blocks waiting for input during any authentication. Moreover, the number of simultaneous authentications is limited only by the amount of memory we're willing to dedicate to representing the state machines.

Authentication protocols are implemented only within `factotum`, but adding and removing protocols does require relinking the binary, so `factotum` processes (but no others) need to be restarted in order to take advantage of new or repaired protocols.

At the time of writing, `factotum` contains authentication modules for the Plan 9 shared key protocol (p9sk1), SSH's RSA authentication, passwords in the clear, APOP, CRAM, PPP's CHAP, Microsoft PPP's MSCHAP, and VNC's challenge/response.

2.3. Local capabilities

A capability system, managed by the kernel, is used to empower `factotum` to grant permission to another process to change its user id. A kernel device driver implements two files, `/dev/caphash` and `/dev/capuse`. The write-only file `/dev/caphash` can be opened only by the host owner, and only once. `Factotum` opens this file immediately after booting.

To use the files, `factotum` creates a string of the form *userid1@userid2@random-string*, uses SHA1 HMAC to hash *userid1@userid2* with key *random-string*, and writes that hash to `/dev/caphash`. `Factotum` then passes the original string to another process on the same machine, running as user *userid1*, which writes the string to `/dev/capuse`. The kernel hashes the string and looks for a matching hash in its list. If it finds one, the writing process's user id changes from *userid1* to *userid2*. Once used, or if a timeout expires, the capability is discarded by the kernel.

The capabilities are local to the machine on which they are created. Hence a `factotum` running on one machine cannot pass capabilities to processes on another and expect them to work.

2.4. Keys

We define the word *key* to mean not only a secret, but also a description of the context in which that secret is to be used: the protocol, server, user, etc. to which it applies. That is, a key is a combination of secret and descriptive information used to authenticate the identities of parties transmitting or receiving information. The set of keys used in any authentication depends both on the protocol and on parameters passed by the program requesting the authentication.

Taking a tip from SDSI [15], which represents security information as textual S-expressions, keys in Plan 9 are represented as plain UTF-8 text. Text is easily understood and manipulated by users. By contrast, a binary or other cryptic format can actually reduce overall security. Binary formats are difficult for users to examine and can only be cracked by special tools, themselves poorly understood by most users. For example, very few people know or understand what's inside their X.509 certificates. Most don't even know where in the system to find them. Therefore, they have no idea what they are trusting, and why, and are powerless to change their trust relationships. Textual, centrally stored and managed keys are easier to use and safer.

Plan 9 has historically represented databases as attribute/value pairs, since they are a good foundation for selection and projection operations. `Factotum` therefore represents the keys in the format *attribute*=*value*, where *attribute* is an identifier, possibly with a single-character prefix, and *value* is an arbitrary quoted string. The pairs themselves are separated by white space. For example, a Plan 9 key and an APOP key might be represented like this:

```
dom=bell-labs.com proto=p9sk1 user=gre
      !password='don''t tell'
proto=apop server=x.y.com user=gre
      !password='open sesame'
```

If a value is empty or contains white space or single quotes, it must be quoted; quotes are represented by doubled single quotes. Attributes that begin with an exclamation mark (`!`) are considered *secret*. `Factotum` will never let a secret value escape its address space and will suppress keyboard echo when asking the user to type one.

A program requesting authentication selects a key by providing a *query*, a list of elements to be matched by the key. Each element in the list is either an *attribute*=*value* pair, which is satisfied by keys with exactly that pair; or an attribute followed by a question mark, *attribute*?, which is satisfied by keys with some pair specifying the attribute. A key matches a query if every element in the list is satisfied. For instance, to select the APOP key in the previous example, an APOP client process might specify the query

```
server=x.y.com proto=apop
```

Internally, `factotum`'s APOP module would add the requirements of having `user` and `!password` attributes, forming the query

```
server=x.y.com proto=apop user? !password?
```

when searching for an appropriate key.

`Factotum` modules expect keys to have some well-known attributes. For instance, the `proto` attribute specifies the protocol module responsible for using a particular key, and protocol modules may expect other well-known attributes (many expect keys to have `!password` attributes, for example). Additional attributes can be used as comments or for further discrimination without intervention by `factotum`; for example, the APOP and IMAP mail clients conventionally include a `server` attribute to select an appropriate key for authentication.

Unlike in SDSI, keys in Plan 9 have no nested structure. This design keeps the representation simple and straightforward. If necessary, we could add a nested attribute or, in the manner of relational databases, an attribute that selects another tuple, but so far the simple design has been sufficient.

A simple common structure for all keys makes them easy for users to administer, but the set of attributes and their interpretation is still protocol-specific and can be subtle. Users may still need to consult a manual to understand all details. Many attributes (`proto`, `user`, `password`, `server`) are self-explanatory and our short experience has not uncovered any particular difficulty in handling keys. Things will likely get messier, however, when we grapple with public keys and their myriad components.

2.5. Protecting keys

Secrets must be prevented from escaping `factotum`. There are a number of ways they could leak: another process might be able to debug the agent process, the agent might swap out to disk, or the process might willingly disclose the key. The last is the easiest to avoid: secret information in a key is marked as such, and whenever `factotum` prints keys or queries for new ones, it is careful to avoid displaying secret information. (The only exception to this is the ‘‘plaintext password’’ protocol, which consists of sending the values of the `user` and `!password` attributes. Only keys tagged with `proto=pass` can have their passwords disclosed by this mechanism.)

Preventing the first two forms of leakage requires help from the kernel. In Plan 9, every process is represented by a directory in the `/proc` file system. Using the files in this directory, other processes could (with appropriate access permission) examine `factotum`'s memory and registers. `Factotum` is protected from processes of other users by the default access bits of its `/proc` directory. However, we'd also like to protect the agent from other processes owned by the same user, both to avoid honest mistakes and to prevent an unattended terminal being exploited to discover secret passwords. To do this, we added a control message to `/proc` called `private`. Once the `factotum` process has written `private` to its `/proc/`*pid*`/ctl` file, no process can access `factotum`'s memory through `/proc`. (Plan 9 has no other mechanism, such as `/dev/kmem`, for accessing a process's memory.)

Similarly, the agent's address space should not be swapped out, to prevent discovering unencrypted keys on the swapping media. The `noswap` control message in `/proc` prevents this scenario. Neither `private` nor `noswap` is specific to `factotum`. User-level file servers such as `dossrv`, which interprets FAT file systems, could use `noswap` to keep their buffer caches from being swapped to disk.

Despite our precautions, attackers might still find a way to gain access to a process running as the host owner on a machine. Although they could not directly access the keys, attackers could use the local `factotum` to perform authentications for them. In the case of some keys, for example those locking bank accounts, we want a way to disable or at least detect such access. That is the role of the `confirm` attribute in a key. Whenever a key with a `confirm` attribute is accessed, the local user must confirm use of the key via a local GUI. The next section describes the actual mechanism.

We have not addressed leaks possible as a result of someone rebooting or resetting a machine running `factotum`. For example, someone could reset a machine and reboot it with a debugger instead of a kernel, allowing them to examine the contents of memory and find keys. We have not found a satisfactory solution to this problem.

2.6. Factotum transactions

External programs manage `factotum`'s internal key state through its file interface, writing textual `key` and `delkey` commands to the `/mnt/factotum/ctl` file. Both commands take a list of attributes as an argument. `Key` creates a key with the given attributes, replacing any extant key with an identical set of public attributes. `Delkey` deletes all keys that match the given set of attributes. Reading the `ctl` file returns a list of keys, one per line, displaying only public attributes. The following example illustrates these interactions.

```
% cd /mnt/factotum
% ls -l
-lrw------- gre gre 0 Jan 30 22:17 confirm
--rw------- gre gre 0 Jan 30 22:17 ctl
-lr-------- gre gre 0 Jan 30 22:17 log
-lrw------- gre gre 0 Jan 30 22:17 needkey
--r--r--r-- gre gre 0 Jan 30 22:17 proto
--rw-rw-rw- gre gre 0 Jan 30 22:17 rpc
% cat >ctl
key dom=bell-labs.com proto=p9sk1 user=gre
    !password='don''t tell'
key proto=apop server=x.y.com user=gre
    !password='bite me'
^D
% cat ctl
key dom=bell-labs.com proto=p9sk1 user=gre
key proto=apop server=x.y.com user=gre
% echo 'delkey proto=apop' >ctl
% cat ctl
key dom=bell-labs.com proto=p9sk1 user=gre
%
```

(A file with the `l` bit set can be opened by only one process at a time.)

The heart of the interface is the `rpc` file. Programs authenticate with `factotum` by writing a request to the `rpc` file and reading back the reply; this sequence is called an RPC *transaction*. Requests and replies have the same format: a textual verb possibly followed by arguments, which may be textual or binary. The most common reply verb is `ok`, indicating success. An RPC session begins with a `start` transaction; the argument is a key query as described earlier. Once started, an RPC conversation usually consists of a sequence of `read` and `write` transactions. If the conversation is successful, an `authinfo` transaction will return information about the identities learned during the transaction. The `attr` transaction returns a list of attributes for the current conversation; the list includes any attributes given in the `start` query as well as any public attributes from keys being used.

As an example of the `rpc` file in action, consider a mail client connecting to a mail server and authenticating using the POP3 protocol's APOP challenge-response command. There are four programs involved: the mail client P_C, the client `factotum` F_C, the mail server P_S, and the server `factotum` F_S. All authentication computations are handled by the `factotum` processes. The mail programs' role is just to relay messages.

At startup, the mail server at `x.y.com` begins an APOP conversation with its `factotum` to obtain the banner greeting, which includes a challenge:

$P_S \rightarrow F_S$: `start proto=apop role=server`
$F_S \rightarrow P_S$: `ok`
$P_S \rightarrow F_S$: `read`
$F_S \rightarrow P_S$: `ok +OK POP3` *challenge*

Having obtained the challenge, the server greets the client:

$P_S \rightarrow P_C$: `+OK POP3` *challenge*

The client then uses an APOP conversation with its `factotum` to obtain a response:

$P_C \rightarrow F_C$: `start proto=apop role=client`
`server=x.y.com`

$F_C \rightarrow P_C$: `ok`
$P_C \rightarrow F_C$: `write +OK POP3` *challenge*
$F_C \rightarrow P_C$: `ok`
$P_C \rightarrow F_C$: `read`
$F_C \rightarrow P_C$: `ok APOP gre` *response*

`Factotum` requires that `start` requests include a `proto` attribute, and the APOP module requires an additional `role` attribute, but the other attributes are optional and only restrict the key space. Before responding to the `start` transaction, the client `factotum` looks for a key to use for the rest of the conversation. Because of the arguments in the `start` request, the key must have public attributes `proto=apop` and `server=x.y.com`; as mentioned earlier, the APOP module additionally requires that the key have `user` and `!password` attributes. Now that the client has obtained a response from its `factotum`, it echoes that response to the server:

$P_C \rightarrow P_S$: `APOP gre` *response*

Similarly, the server passes this message to its `factotum` and obtains another to send back.

$P_S \rightarrow F_S$: `write APOP gre` *response*
$F_S \rightarrow P_S$: `ok`
$P_S \rightarrow F_S$: `read`
$F_S \rightarrow P_S$: `ok +OK welcome`

$P_S \rightarrow P_C$: `+OK welcome`

Now the authentication protocol is done, and the server can retrieve information about what the protocol established.

$P_S \rightarrow F_S$: `authinfo`
$F_S \rightarrow P_S$: `ok client=gre`
`capability=`*capability*

The `authinfo` data is a list of *attr*`=`*value* pairs, here a client user name and a capability. (Protocols that establish shared secrets or provide mutual authentication indicate this by adding appropriate *attr*`=`*value* pairs.) The capability can be used by the server to change its identity to that of the client, as described earlier. Once it has changed its identity, the server can access and serve the client's mailbox.

Two more files provide hooks for a graphical `factotum` control interface. The first, `confirm`, allows the user detailed control over the use of certain keys. If a key has a `confirm=` attribute, then the user must approve each use of the key. A separate program with a graphical interface reads from the `confirm` file to see when a confirmation is necessary. The read blocks until a key usage needs to be approved, whereupon it will return a line of the form

`confirm tag=1` *attributes*

requesting permission to use the key with those public attributes. The graphical interface then prompts the user for approval and writes back

```
tag=1 answer=yes
```

(or `answer=no`).

The second file, `needkey`, diverts key requests. In the APOP example, if a suitable key had not been found during the `start` transaction, `factotum` would have indicated failure by returning a response indicating what key was needed:

$F_C \rightarrow P_C$: `needkey proto=apop`
`server=x.y.com user? !password?`

A typical client would then prompt the user for the desired key information, create a new key via the `ctl` file, and then reissue the `start` request. If the `needkey` file is open, then instead of failing, the transaction will block, and the next read from the `/mnt/factotum/needkey` file will return a line of the form

`needkey tag=1` *attributes*

The graphical interface then prompts the user for the needed key information, creates the key via the `ctl` file, and writes back `tag=1` to resume the transaction.

The remaining files are informational and used for debugging. The `proto` file contains a list of supported protocols (to see what protocols the system supports, `cat /mnt/factotum/proto`), and the `log` file contains a log of operations and debugging output enabled by a `debug` control message.

The next few sections explain how `factotum` is used by system services.

3. Authentication in 9P

Plan 9 uses a remote file access protocol, 9P [12], to connect to resources such as the file server and remote processes. The original design for 9P included special messages at the start of a conversation to authenticate the user. Multiple users can share a single connection, such as when a CPU server runs processes for many users connected to a single file server, but each must authenticate separately. The authentication protocol, similar to that of Kerberos [18], used a sequence of messages passed between client, file server, and authentication server to verify the identities of the user, calling machine, and serving machine. One major drawback to the design was that the authentication method was defined by 9P itself and could not be changed. Moreover, there was no mechanism to relegate authentication to an external (trusted) agent, so a process implementing 9P needed, besides support for file service, a substantial body of cryptographic code to implement a handful of startup messages in the protocol.

A recent redesign of 9P addressed a number of file service issues outside the scope of this paper. On issues of authentication, there were two goals: first, to remove details about authentication from the protocol itself; second, to allow an external program to execute the authentication part of the protocol. In particular, we wanted a way to quickly incorporate ideas found in other systems such as SFS [8].

Since 9P is a file service protocol, the solution involved creating a new type of file to be served: an *authentication file*. Connections to a 9P service begin in a state that allows no general file access but permits the client to open an authentication file by sending a special message, generated by the new `fauth` system call:

```
    afd = fauth(int fd, char *servicename);
```

Here `fd` is the user's file descriptor for the established network connection to the 9P server and `servicename` is the name of the desired service offered on that server, typically the file subsystem to be accessed. The returned file descriptor, `afd`, is a unique handle representing the authentication file created for this connection to authenticate to this service; it is analogous to a capability. The authentication file represented by `afd` is not otherwise addressable on the server, such as through the file name hierarchy. In all other respects, it behaves like a regular file; most important, it accepts standard read and write operations.

To prove its identity, the user process (via `factotum`) executes the authentication protocol, described in the next section of this paper, over the `afd` file descriptor with ordinary reads and writes. When client and server have successfully negotiated, the authentication file changes state so it can be used as evidence of authority in `mount`.

Once identity is established, the process presents the (now verified) `afd` as proof of identity to the `mount` system call:

```
    mount(int fd, int afd, char *mountpoint,
          int flag, char *servicename)
```

If the `mount` succeeds, the user now has appropriate permissions for the file hierarchy made visible at the mount point.

This sequence of events has several advantages. First, the actual authentication protocol is implemented using regular reads and writes, not special 9P messages, so they can be processed, forwarded, proxied, and so on by any 9P agent without special arrangement. Second, the business of negotiating the authentication by reading and writing the authentication file can be delegated to an outside agent, in particular `factotum`; the programs that implement the client and server ends of a 9P conversation need no authentication or cryptographic code. Third, since the authentication protocol is not defined by 9P itself, it is easy to change and can even be negotiated dynamically. Finally, since `afd` acts like a capability, it can be treated like one: handed to another process to give it special permissions; kept around for later use when authentication is again required; or closed to make sure no other process can use it.

All these advantages stem from moving the authentication negotiation into reads and writes on a separate file. As is often the case in Plan 9, making a resource (here authentication) accessible with a file-like interface reduces *a priori* the need for special interfaces.

3.1. Plan 9 shared key protocol

In addition to the various standard protocols supported by `factotum`, we use a shared key protocol for native Plan 9 authentication. This protocol provides backward compatibility with older versions of the system. One reason for the new architecture is to let us replace such protocols in the near future with more cryptographically secure ones.

P9sk1 is a shared key protocol that uses tickets much like those in the original Kerberos. The difference is that we've replaced the expiration time in Kerberos tickets with a random nonce parameter and a counter. We summarize it here:

$$\begin{aligned}
C \rightarrow S:&\quad nonce_C \\
S \rightarrow C:&\quad nonce_S, uid_S, domain_S \\
\\
C \rightarrow A:&\quad nonce_S, uid_S, domain_S, uid_C, \\
&\quad factotum_C \\
A \rightarrow C:&\quad K_C\{nonce_S, uid_C, uid_S, K_n\}, \\
&\quad K_S\{nonce_S, uid_C, uid_S, K_n\} \\
\\
C \rightarrow S:&\quad K_S\{nonce_S, uid_C, uid_S, K_n\}, \\
&\quad K_n\{nonce_S, counter\} \\
S \rightarrow C:&\quad K_n\{nonce_C, counter\}
\end{aligned}$$

(Here $K\{x\}$ indicates x encrypted with DES key K.) The first two messages exchange nonces and server identification. After this initial exchange, the client contacts the authentication server to obtain a pair of encrypted tickets, one encrypted with the client key and one with the server key. The client relays the server ticket to the server. The server believes that the ticket is new because it contains $nonce_S$ and that the ticket is from the authentication server because it is encrypted in the server key K_S. The ticket is basically a statement from the authentication server that now uid_C and uid_S share a secret K_n. The authenticator $K_n\{nonce_S, counter\}$ convinces the server that the client knows K_n and thus must be uid_C. Similarly, authenticator $K_n\{nonce_C, counter\}$ convinces the client that the server knows K_n and thus must be uid_S. Tickets can be reused, without contacting the authentication server again, by incrementing the counter before each authenticator is generated.

In the future we hope to introduce a public key version of p9sk1, which would allow authentication even when the authentication server is not available.

3.2. The authentication server

Each Plan 9 security domain has an authentication server (AS) that all users trust to keep the complete set of shared keys. It also offers services for users and administrators to manage the keys, create and disable accounts, and so on. It typically runs on a standalone machine with few other services. The AS comprises two services, `keyfs` and `authsrv`.

`Keyfs` is a user-level file system that manages an encrypted database of user accounts. Each account is represented by a directory containing the files `key`, containing the Plan 9 key for p9sk1; `secret` for the challenge/response protocols (APOP, VNC, CHAP, MSCHAP, CRAM); `log` for authentication outcomes; `expire` for an expiration time; and `status`. If the expiration time passes, if the number of successive failed authentications exceeds 50, or if `disabled` is written to the status file, any attempt to access the `key` or `secret` files will fail.

`Authsrv` is a network service that brokers shared key authentications for the protocols p9sk1, APOP, VNC, CHAP, MSCHAP, and CRAM. Remote users can also call `authsrv` to change their passwords.

The p9sk1 protocol was described in the previous section. The challenge/response protocols differ in detail but all follow the general structure:

$C \rightarrow S$: $nonce_C$
$S \rightarrow C$: $nonce_S, uid_S, domain_S$
$C \rightarrow A$: $nonce_S, uid_S, domain_S, hostid_C, uid_C$
$A \rightarrow C$: $K_C\{nonce_S, uid_C, uid_S, K_n\}, K_S\{nonce_S, uid_C, uid_S, K_n\}$
$C \rightarrow S$: $K_S\{nonce_S, uid_C, uid_S, K_n\}, K_n\{nonce_S\}$
$S \rightarrow C$: $K_n\{nonce_C\}$

The password protocol is:

$C \rightarrow A$: uid_C
$A \rightarrow C$: $K_c\{K_n\}$
$C \rightarrow A$: $K_n\{password_{old}, password_{new}\}$
$A \rightarrow C$: OK

To avoid replay attacks, the pre-encryption clear text for each of the protocols (as well as for p9sk1) includes a tag indicating the encryption's role in the protocol. We elided them in these outlines.

3.3. Protocol negotiation

Rather than require particular protocols for particular services, we implemented a negotiation metaprotocol, *p9any*, which chooses the actual authentication protocol to use. P9any is used now by all native services on Plan 9.

The metaprotocol is simple. The callee sends a null-terminated string of the form:

`v.`*n* $proto_1$`@`$domain_1$ $proto_2$`@`$domain_2$ `...`

where *n* is a decimal version number, $proto_k$ is the name of a protocol for which the `factotum` has a key, and $domain_k$ is the name of the domain in which the key is valid. The caller then responds

proto`@`*domain*

indicating its choice. Finally the callee responds

`OK`

Any other string indicates failure. At this point the chosen protocol commences. The final fixed-length reply is used to make it easy to delimit the I/O stream should the chosen protocol require the caller rather than the callee to send the first message.

With this negotiation metaprotocol, the underlying authentication protocols used for Plan 9 services can be changed under any application just by changing the keys known by the `factotum` agents at each end.

P9any is vulnerable to man in the middle attacks to the extent that the attacker may constrain the possible choices by changing the stream. However, we believe this is acceptable since the attacker cannot force either side to choose algorithms that it is unwilling to use.

4. Library Interface to Factotum

Although programs can access factotum's services through its file system interface, it is more common to use a C library that packages the interaction. There are a number of routines in the library, not all of which are relevant here, but a few examples should give their flavor.

First, consider the problem of mounting a remote file server using 9P. An earlier discussion showed how the fauth and mount system calls use an authentication file, afd, as a capability, but not how factotum manages afd. The library contains a routine, amount (authenticated mount), that is used by most programs in preference to the raw fauth and mount calls. Amount engages factotum to validate afd; here is the complete code:

```
int
amount(int fd, char *mntpt,
   int flags, char *aname)
{
   int afd, ret;
   AuthInfo *ai;

   afd = fauth(fd, aname);
   if(afd >= 0){
      ai = auth_proxy(afd, amount_getkey,
         "proto=p9any role=client");
      if(ai != NULL)
         auth_freeAI(ai);
   }
   ret = mount(fd, afd, mntpt,
      flags, aname);
   if(afd >= 0)
      close(afd);
   return ret;
}
```

where parameter fd is a file descriptor returned by open or dial for a new connection to a file server. The conversation with factotum occurs in the call to auth_proxy, which specifies, as a key query, which authentication protocol to use (here the metaprotocol p9any) and the role being played (client). Auth_proxy will read and write the factotum files, and the authentication file descriptor afd, to validate the user's right to access the service. If the call is successful, any auxiliary data, held in an AuthInfo structure, is freed. In any case, the mount is then called with the (perhaps validated) afd. A 9P server can cause the fauth system call to fail, as an indication that authentication is not required to access the service.

The second argument to auth_proxy is a function, here amount_getkey, to be called if secret information such as a password or response to a challenge is required as part of the authentication. This function, of course, will provide this data to factotum as a key message on the /mnt/factotum/ctl file.

Although the final argument to auth_proxy in this example is a simple string, in general it can be a formatted-print specifier in the manner of printf, to enable the construction of more elaborate key queries.

As another example, consider the Plan 9 cpu service, which exports local devices to a shell process on a remote machine, typically to connect the local screen and keyboard to a more powerful computer. At heart, cpu is a superset of a service called exportfs [12], which allows one machine to see an arbitrary portion of the file name space of another machine, such as to export the network device to another machine for gatewaying. However, cpu is not just exportfs because it also delivers signals such as interrupt and negotiates the initial environment for the remote shell.

To authenticate an instance of cpu requires factotum processes on both ends: the local, client end running as the user on a terminal and the remote, server end running as the host owner of the server machine. Here is schematic code for the two ends:

```
/* client */
int
p9auth(int fd)
{
   AuthInfo *ai;

   ai = auth_proxy(fd, auth_getkey,
      "proto=p9any role=client");
   if(ai == NULL)
      return -1;

   /* start cpu protocol here */
}

/* server */
int
srvp9auth(int fd, char *user)
{
   AuthInfo *ai;

   ai = auth_proxy(fd, NULL,
      "proto=p9any role=server");
   if(ai == NULL)
      return -1;
   /* set user id for server process */
   if(auth_chuid(ai, NULL) < 0)
      return -1;

   /* start cpu protocol here */
}
```

Auth_chuid encapsulates the negotiation to change a user id using the caphash and capuse files of the (server) kernel. Note that although the client process may ask the user for new keys, using auth_getkey, the server machine, presumably a shared machine with a pseudo-user for the host owner, sets the key-getting function to NULL.

5. Secure Store

Factotum keeps its keys in volatile memory, which must somehow be initialized at boot time. Therefore, factotum must be supplemented by a persistent store, perhaps a floppy disk containing a key file of commands to be copied into /mnt/factotum/ctl during bootstrap. But removable media are a nuisance to carry and are vulnerable to theft. Keys could be stored encrypted on a shared file system, but only if those keys are not necessary for authenticating to the file system in the first place. Even if the keys are encrypted under a user password, a thief might well succeed with a dictionary attack. Other risks of local storage are loss of the contents through mechanical mishap or dead batteries. Thus for convenience and safety we provide a secstore (secure store) server in the network to hold each user's permanent list of keys, a *key file*.

Secstore is a file server for encrypted data, used only during bootstrapping. It must provide strong authentication and resistance to passive and active protocol attacks while assuming nothing more from the client than a password. Once factotum has loaded the key file, further encrypted or authenticated file storage can be accomplished by standard mechanisms.

The cryptographic technology that enables secstore is a form of encrypted key exchange called PAK [2], analogous to EKE [1], SRP [19], or SPEKE [5]. PAK was chosen because it comes with a proof of equivalence in strength to Diffie-Hellman; subtle flaws in some earlier encrypted key exchange protocols and implementations have encouraged us to take special care. In outline, the PAK protocol is:

$$\begin{aligned} C \rightarrow S&: \ C, g^x H \\ S \rightarrow C&: \ S, g^y, hash(g^{xy}, C, S) \\ C \rightarrow S&: \ hash(g^{xy}, S, C) \end{aligned}$$

where H is a preshared secret between client C and server S. There are several variants of PAK, all presented in papers mainly concerned with proofs of cryptographic properties. To aid implementers, we have distilled a description of the specific version we use into an Appendix to this paper. The Plan 9 open source license provides for use of Lucent's encrypted key exchange patents in this context.

As a further layer of defense against password theft, we provide (within the encrypted channel $C \rightarrow S$) information that is validated at a RADIUS server, such as the digits from a hardware token [14]. This provides two-factor authentication, which potentially requires tricking two independent administrators in any attack by social engineering.

The key file stored on the server is encrypted with AES (Rijndael) using CBC with a 10-byte initialization vector and trailing authentication padding. All this is invisible to the user of secstore. For that matter, it is invisible to the secstore server as well; if the AES Modes of Operation are standardized and a new encryption format designed, it can be implemented by a client without change to the server. The secstore is deliberately not backed up; the user is expected to use more than one secstore or save the key file on removable media and lock it away. The user's password is hashed to create the H used in the PAK protocol; a different hash of the password is used as the file encryption key. Finally, there is a command (inside the authenticated, encrypted channel between client and secstore) to change passwords by sending a new H; for consistency, the client process must at the same time fetch and re-encrypt all files.

When factotum starts, it dials the local secstore and checks whether the user has an account. If so, it prompts for the user's secstore password and fetches the key file. The PAK protocol ensures mutual authentication and prevents dictionary attacks on the password by passive wiretappers or active intermediaries. Passwords saved in the key file can be long random strings suitable for simpler challenge/response authentication protocols. Thus the user need only remember a single, weaker password to enable strong, ‘‘single sign on’’ authentication to unchanged legacy applications scattered across multiple authentication domains.

6. Transport Layer Security

Since the Plan 9 operating system is designed for use in network elements that must withstand direct attack, unguarded by firewall or VPN, we seek to ensure that all applications use channels with appropriate mutual authentication and encryption. A principal tool for this is TLS 1.0 [3]. (TLS 1.0 is nearly the same as SSL 3.0, and our software is designed to interoperate with implementations of either standard.)

TLS defines a record layer protocol for message integrity and privacy through the use of message digesting and encryption with shared secrets. We implement this service as a kernel device, though it could be performed at slightly higher cost by invoking a separate program. The library interface to the TLS kernel device is:

```
int pushtls(int fd, char *hashalg,
    char *cryptalg, int isclient,
    char *secret, char *dir);
```

Given a file descriptor, the names of message digest and encryption algorithms, and the shared secret, `pushtls` returns a new file descriptor for the encrypted connection. (The final argument `dir` receives the name of the directory in the TLS device that is associated with the new connection.) The function is named by analogy with the ‘‘push’’ operation supported by the stream I/O system of Research Unix and the first two editions of Plan 9. Because adding encryption is as simple as replacing one file descriptor with another, adding encryption to a particular network service is usually trivial.

The Plan 9 shared key authentication protocols establish a shared 56-bit secret as a side effect. Native Plan 9 network services such as `cpu` and `exportfs` use these protocols for authentication and then invoke `pushtls` with the shared secret.

Above the record layer, TLS specifies a handshake protocol using public keys to establish the session secret. This protocol is widely used with HTTP and IMAP4 to provide server authentication, though with client certificates it could provide mutual authentication. The library function

```
int tlsClient(int fd, TLSconn *conn)
```

handles the initial handshake and returns the result of `pushtls`. On return, it fills the `conn` structure with the session ID used and the X.509 certificate presented by the server, but makes no effort to verify the certificate. Although the original design intent of X.509 certificates expected that they would be used with a Public Key Infrastructure, reliable deployment has been so long delayed and problematic that we have adopted the simpler policy of just using the X.509 certificate as a representation of the public key, depending on a locally-administered directory of SHA1 thumbprints to allow applications to decide which public keys to trust for which purposes.

7. Related Work and Discussion

Kerberos, one of the earliest distributed authentication systems, keeps a set of authentication tickets in a temporary file called a ticket cache. The ticket cache is protected by Unix file permissions. An environment variable containing the file name of the ticket cache allows for different ticket caches in different simultaneous login sessions. A user logs in by typing his or her Kerberos password. The login program uses the Kerberos password to obtain a temporary ticket-granting ticket from the authentication server, initializes the ticket cache with the ticket-granting ticket, and then forgets the password. Other applications can use the ticket-granting ticket to sign tickets for themselves on behalf of the user during the login session. The ticket cache is removed when the user logs out [18]. The ticket cache relieves the user from typing a password every time authentication is needed.

The secure shell SSH develops this idea further, replacing the temporary file with a named Unix domain socket connected to a user-level program, called an agent. Once the SSH agent is started and initialized with one or more RSA private keys, SSH clients can employ it to perform RSA authentications on their behalf. In the absence of an agent, SSH typically uses RSA keys read from encrypted disk files or uses passphrase-based authentication, both of which would require prompting the user for a passphrase whenever authentication is needed [20]. The self-certifying file system SFS uses a similar agent [6], not only for moderating the use of client authentication keys but also for verifying server public keys [8].

`Factotum` is a logical continuation of this evolution, replacing the program-specific SSH or SFS agents with a general agent capable of serving a wide variety of programs. Having one agent for all programs removes the need to have one agent for each program. It also allows the programs themselves to be protocol-agnostic, so that, for example, one could build an SSH workalike capable of using any protocol supported by `factotum`, without that program knowing anything about the protocols. Traditionally each program needs to implement each authentication protocol for itself, an $O(n^2)$ coding problem that `factotum` reduces to $O(n)$.

Previous work on agents has concentrated on their use by clients authenticating to servers. Looking in the other direction, Sun Microsystem's pluggable authentication module (PAM) is one of the earliest attempts to provide a general authentication mechanism for Unix-like operating systems [17]. Without a central authority like PAM, system policy is tied up in the various implementations of network services. For example, on a typical Unix, if a system administrator decides not to allow plaintext passwords for authentication, the configuration files for a half dozen different servers — `rlogind`, `telnetd`, `ftpd`, `sshd`, and so on — need to be edited. PAM solves this problem by hiding the details of a given authentication mechanism behind a common library interface. Directed by a system-wide configuration file, an application selects a particular authentication mechanism by dynamically loading the appropriate shared library. PAM is widely used on Sun's Solaris and some Linux distributions.

`Factotum` achieves the same goals using the agent approach. `Factotum` is the only process that needs to create capabilities, so all the network servers can run as untrusted users (e.g., Plan 9's `none` or Unix's `nobody`), which greatly reduces the harm done if a server is buggy and is compromised. In fact, if `factotum` were implemented on Unix along with an analogue to the Plan 9 capability device, venerable programs like `su` and `login` would no longer need to be installed ''setuid root.''

Several other systems, such as Password Safe [16], store multiple passwords in an encrypted file, so that the user only needs to remember one password. Our `secstore` solution differs from these by placing the storage in a hardened location in the network, so that the encrypted file is less liable to be stolen for offline dictionary attack and so that it is available even when a user has several computers. In contrast, Microsoft's Passport system [9] keeps credentials in the network, but centralized at one extremely-high-value target. The important feature of Passport, setting up trust relationships with e-merchants, is outside our scope. The `secstore` architecture is almost identical to Perlman and Kaufman's [10] but with newer EKE technology. Like them, we chose to defend mainly against outside attacks on `secstore`; if additional defense of the files on the server itself is desired, one can use distributed techniques [4].

We made a conscious choice of placing encryption, message integrity, and key management at the application layer (TLS, just above layer 4) rather than at layer 3, as in IPsec. This leads to a simpler structure for the network stack, easier integration with applications and, most important, easier network administration since we can recognize which applications are misbehaving based on TCP port numbers. TLS does suffer (relative to IPsec) from the possibility of forged TCP Reset, but we feel that this is adequately dealt with by randomized TCP sequence numbers. In contrast with other TLS libraries, Plan 9 does not require the application to change `write` calls to `sslwrite` but simply to add a few lines of code at startup [13].

8. Conclusion

Writing safe code is difficult. Stack attacks, mistakes in logic, and bugs in compilers and operating systems can each make it possible for an attacker to subvert the intended execution sequence of a service. If the server process has the privileges of a powerful user, such as `root` on Unix, then so does the attacker. `Factotum` allows us to constrain the privileged execution to a single process whose core is a few thousand lines of code. Verifying such a process, both through manual and automatic means, is much easier and less error prone than requiring it of all servers.

An implementation of these ideas is in Plan 9 from Bell Labs, Fourth Edition, freely available from `http://plan9.bell-labs.com/plan9`.

Acknowledgments

William Josephson contributed to the implementation of password changing in `secstore`. We thank Phil MacKenzie and Martín Abadi for helpful comments on early parts of the design. Chuck Blake, Peter Bosch, Frans Kaashoek, Sape Mullender, and Lakshman Y. N., predominantly Dutchmen, gave helpful comments on the paper. Russ Cox is supported by a fellowship from the Fannie and John Hertz Foundation.

References

1. S.M. Bellovin and M. Merritt, ''Augmented Encrypted Key Exchange,'' Proceedings of the 1st ACM Conference on Computer and Communications Security, 1993, pp. 244 - 250.

2. Victor Boyko, Philip MacKenzie, and Sarvar Patel, ''Provably Secure Password-Authenticated Key Exchange using Diffie-Hellman,'' Eurocrypt 2000, 156−171.

3. T . Dierks and C. Allen, ‘‘The TLS Protocol, Version 1.0,’’ RFC 2246.

4. Warwick Ford and Burton S. Kaliski, Jr., ‘‘Server-Assisted Generation of a Strong Secret from a Password,’’ IEEE Fifth International Workshop on Enterprise Security, National Institute of Standards and Technology (NIST), Gaithersburg MD, June 14 - 16, 2000.

5. David P. Jablon, ‘‘Strong Password-Only Authenticated Key Exchange,’’ `http://integritysciences.com/speke97.html`.

6. Michael Kaminsky. ‘‘Flexible Key Management with SFS Agents,’’ Master’s Thesis, MIT, May 2000.

7. Philip MacKenzie, private communication.

8. David Mazières, Michael Kaminsky, M. Frans Kaashoek and Emmett Witchel, ‘‘Separating key management from file system security,’’ Symposium on Operating Systems Principles, 1999, pp. 124-139.

9. Microsoft Passport, `http://www.passport.com/`.

10. Radia Perlman and Charlie Kaufman, ‘‘Secure Password-Based Protocol for Downloading a Private Key,’’ Proc. 1999 Network and Distributed System Security Symposium, Internet Society, January 1999.

11. Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom, ‘‘Plan 9 from Bell Labs,’’ Computing Systems, **8**, 3, Summer 1995, pp. 221-254.

12. Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, Phil Winterbottom, ‘‘The Use of Name Spaces in Plan 9,’’ Operating Systems Review, **27**, 2, April 1993, pp. 72-76 (reprinted from Proceedings of the 5th ACM SIGOPS European Workshop, Mont Saint-Michel, 1992, Paper nº 34).

13. Eric Rescorla, ‘‘SSL and TLS: Designing and Building Secure Systems,’’ Addison-Wesley, 2001. ISBN 0-201-61598-3, p. 387.

14. C. Rigney, A. Rubens, W. Simpson, S. Willens, ‘‘Remote Authentication Dial In User Service (RADIUS),’’ RFC2138, April 1997.

15. Ronald L. Rivest and Butler Lampson, ‘‘SDSI—A Simple Distributed Security Infrastructure,’’ `http://theory.lcs.mit.edu/~rivest/sdsi10.ps`.

16. Bruce Schneier, Password Safe, `http://www.counterpane.com/passsafe.html`.

17. Vipin Samar, ‘‘Unified Login with Pluggable Authentication Modules (PAM),’’ Proceedings of the Third ACM Conference on Computer Communications and Security, March 1996, New Delhi, India.

18. Jennifer G. Steiner, Clifford Neumann, and Jeffrey I. Schiller, ‘‘*Kerberos*: An Authentication Service for Open Network Systems,’’ Proceedings of USENIX Winter Conference, Dallas, Texas, February 1988, pp. 191−202.

19. T. Wu, ‘‘The Secure Remote Password Protocol,’’ Proceedings of the 1998 Internet Society Network and Distributed System Security Symposium, San Diego, CA, March 1998, pp. 97-111.

20. Ylonen, T., ‘‘SSH—Secure Login Connections Over the Internet,’’ 6th USENIX Security Symposium, pp. 37-42. San Jose, CA, July 1996.

Appendix: Summary of the PAK protocol

Let $q > 2^{160}$ and $p > 2^{1024}$ be primes such that $p = rq + 1$ with r not a multiple of q. Take $h \in Z_p^*$ such that $g \equiv h^r$ is not 1. These parameters may be chosen by the NIST algorithm for DSA, and are public, fixed values. The client C knows a secret π and computes $H \equiv (H_1(C, \pi))^r$ and H^{-1}, where H_1 is a hash function yielding a random element of Z_p^*, and H^{-1} may be computed by gcd. (All arithmetic is modulo p.) The client gives H^{-1} to the server S ahead of time by a private channel. To start a new connection, the client generates a random value x, computes $m \equiv g^x H$, then calls the server and sends C and m. The server checks $m \neq 0$ mod p, generates random y, computes $\mu \equiv g^y$, $\sigma \equiv (mH^{-1})^y$, and sends S, μ, $k \equiv sha1(\text{"server"}, C, S, m, \mu, \sigma, H^{-1})$. Next the client computes $\sigma = \mu^x$, verifies k, and sends $k' \equiv sha1(\text{"client"}, C, S, m, \mu, \sigma, H^{-1})$. The server then verifies k' and both sides begin using session key $K \equiv sha1(\text{"session"}, C, S, m, \mu, \sigma, H^{-1})$. In the published version of PAK, the server name S is included in the initial hash H, but doing so is inconvenient in our application, as the server may be known by various equivalent names.

MacKenzie has shown [7] that the equivalence proof [2] can be adapted to cover our version.