Plan 9 Authentication in Linux

Ashwin Ganti [*]
Google Inc.
aganti@google.com

ABSTRACT
In Linux, applications like `su` and `login` currently run as root in order to access authentication information and set or alter the identity of the process. In such cases, if the application is compromised while running as a privileged user, the entire system can become vulnerable. An alternative approach is taken by the Plan 9 operating system from Bell Labs, which runs such applications as a non-privileged user and relies on a kernel-based capability device working in coordination with an authentication server to provide the same services. This avoids the risk of an application vulnerability becoming a system vulnerability.

This paper discusses the extension of Linux authentication mechanisms to allow the use of the Plan 9 approach with existing Linux applications in order to reduce the security risks mentioned earlier. It describes the port of the Plan 9 capability device as a character device driver for the Linux kernel. It also describes the port of the Plan 9 authentication server and the implementation of a PAM module which allows the use of these new facilities. *It is now possible to restrain processes like login and su from the uncontrolled setuid bit and make them run on behalf of an unprivileged user in Linux.*

Categories and Subject Descriptors
D.4.6 [**Operating Systems**]: Security and Protection—*Authentication*

General Terms
Authentication

1. INTRODUCTION
Authentication is any process by which a system verifies that someone is who they claim to be. This usually involves a user name and password but it is also done using hardware tokens, biometrics etc. The anatomy of authentication involves two aspects, one is proving who you say you are and the other part is to change the owner of the process to the authenticated user.

Historically, Linux authentication was based on the passwd and shadow files kept in */etc* which contained both a hashed password and various information about the user (such as her home directory and default shell). Applications heavily depended on these files which led to less extensible code. Moreover if a new authentication mechanism was introduced it required all the applications (like login, su etc.) that use the authentication information be rewritten to support it.

`Pluggable Authentication Modules (PAM)` [10] [2] were created to provide more effective ways to authenticate users. PAM provides a generic framework enabling uniform authentication of user applications. It enables user applications to perform authentication without actually knowing the implementation details of the underlying mechanisms. It is usually done through a password based authentication mechanism but also supports a challenge response interaction. In order to use a particular authentication scheme (ex: Kerberos) to authenticate a user, an application can dynamically link a PAM module that implements that authentication scheme. Any PAM module which is written using the PAM framework exposes a generic set of API/functions to the applications. Applications simply call the functions defined in the module passing in the credentials of the user. The advantage of this framework is that any change that occurs in the authentication mechanism does not require the applications to be retrofitted to support it.

While PAM is a convenient way of authenticating the users from the perspective of an application, it results in having the authentication code run in the same address space as the application and might be circumvented. Since it is just a library, an application using PAM must operate at the capability level which is necessary to provide the service. It does not have any increased level of privilege over the application using it and hence the onus of protecting the environment in which PAM operates is on the application.

As mentioned earlier, the second part of authentication is the changing of the process ownership to the authenticated user. Historically, the ability to change the user id associated with a process was among a group of actions (such as mounting file systems, raw device access, or shutting down the system) which could only be performed by the super user. Applications (such as `login`, `ping`, and `mount`) which required these facilities used a special permission bit, named setuid, which would always execute the application with super user privileges even when started by a normal user. The danger in this approach was that flaws in these applications could enable privilege escalation which would allow normal

[*]Ashwin Ganti was earlier with the Department of Computer Science, University of Illinois at Chicago, IL, USA where the current work has been done. He is now with Google Inc., Mountain View, CA 94043 USA.

users to become the super user or otherwise compromise the system.

The Linux capability system was introduced to subdivide the actions typically associated with the super user to limit the security risk of applications which required a particular privileged operation. Instead of granting applications such as `ping` and `mount` super-user privileges via the setuid bit, they are only given the capabilities they need to perform their function. This should prevent an application such as `ping`, which only requires the super user capability to directly access network devices, from being subverted to change the user identity of the process. However, this approach does not reduce the risk for applications such as `su` and `login` which require the capability to change the user identity. *The existing capability system does not govern which users a process may change its identity to.* In other words, the same vulnerability described earlier exists for all applications involved with authentication such as `su` and `login`.

Having an application run with the capability to change ownership to any user does not achieve necessary privilege separation [9] and if the application has been compromised then an attacker can get super user access to the system. A new mechanism is required which isolates authentication from applications and provides more granular control over critical system capabilities such as the ability to set the user and group identity of a process.

The Plan 9 research operating system from Bell Labs [8] solves most of the above security issues in a very graceful and modular way. Authentication in Plan 9 is centered around a per user agent called *factotum* [7] following the lead of the SSH *agent* [11]. *Factotum* is a trusted process that holds the secure keys of the user and negotiates authentication protocols on behalf of the user. Factotum is implemented as a file server in Plan 9 and applications that need authentication services communicate with factotum using the usual file system calls (read, write etc.) The applications need not be compiled with the authentication or cryptographic code and can remain agnostic of the underlying mechanism which is understood by factotum. Moreover the applications need not be retrofitted for the changes in the authentication protocols. Isolating the privileged code to a single more trustworthy component makes it possible to run the applications at a weaker privilege level.

Plan 9 avoids privilege escalation in applications such as `su` and `login` through a granular capability system which only provides the ability to switch identity to a specific user instead of granting these applications the ability to switch to any user. When a process such as the `login` program needs to change its identity it proves to the *host owner's* factotum that it has the required credentials to run as that identity by running an authentication protocol. *Host owner* is a user in Plan 9 that basically owns the local resources of a machine like the local disks, network interfaces, etc. The *Host owner*'s factotum and the `login` program then interact with a kernel implemented capability system to authorize `login` to change its identity to the specific user authenticated by the protocol. Because the capability is specific to the authenticated identity, the `login` application can not be compromised to obtain super user (or even *Host owner*) privileges. This is a more secure way of handling things when compared to Linux, which allows applications such as `login` to switch their identity to any user.

Our effort here is to achieve similar results for Linux by adopting the authentication mechanisms and Plan 9 notion of capabilities. *This work is going to be very useful in getting closer to a goal of getting a Linux machine to boot with no processes running as root.*

There are numerous advantages:

- Applications like `su` and `login` need not run as the super user.
- The capability system is managed by the kernel which is inherently trusted and more secure.
- Capabilities to change identity are restricted to specific authenticated identities by the authentication system.
- Protocols used to communicate with the authentication server have been extensively researched and are proven to be more secure than the traditional protocols used for the communication of the authentication information. They are not subject to the "man in the middle" attacks that the other protocols suffer from.
- This is a cluster/networked solution in the sense that the same authentication mechanism which works for a stand alone machine can be scaled up to be used to perform authentication over a clustered environment.

We have implemented the Plan 9 capability device for the Linux kernel, porting the authentication server as part of this effort. We wrote a PAM module to perform the authentication with the host owner's factotum.

The rest of the paper delineates the components needed for Linux to make use if this new mechanism while not being disruptive to the existing applications' code, explains the implementation details of the port and concludes by mentioning some future improvements and related work.

2. COMPONENTS NEEDED IN LINUX

This section delineates the various components required to make the existing applications in Linux use the new authentication mechanism of Plan 9. The implementation details of each of the components needed is explained in Section 4.

2.1 Plan 9 authentication server for Linux

Each security domain in Plan 9 has a trusted authentication server where all the shared keys of the users are maintained. It also offers services for users and administrators to create and disable accounts, manage keys, etc. It is comprised of two services, *authsrv* and *keyfs*. *authsrv* is a network service which is similar to the Key Distribution Center (KDC) in Kerberos. It brokers the network authentication sessions. The *passwd* utility can be used to change account passwords on the authentication server. *keyfs* is a user level file system that manages an encrypted database of user accounts. We have ported *authsrv* and *keyfs* for Linux using

the pre-existing Plan 9 from User Space [6] package which already included a Linux port of *factotum*. We also ported *changeuser* which is a utility to create and manage user accounts in the authentication server. A side effect of using these Plan 9 tools is that a copy of the user's account information must be maintained in the authentication server separate from the typical Linux locations, but this can be avoided in the future by building a Name Service Module configurable through *nsswitch.conf* that retrieves the user's credentials from the Plan 9 authentication server.

2.2 Linux Capability Device

In Plan 9, a process that wants to change its identity authenticates with the host owner's factotum. The host owner's factotum on successful authentication creates a capability, gives a hash of it to the kernel and passes the capability to the process that requested the change of identity. Once the process receives the capability it proves to the kernel that it has a valid capability and the kernel changes the uid of the process. Further details on the semantics and working of the cap device for the Linux kernel can be found in Section 4.1.

We have implemented the Plan 9 capability device for the Linux kernel as a character device driver. This is perhaps the most important part of the authentication scheme. The capability device managed by the kernel is used to allow factotum to grant permission to a process to change the user id. It exposes two device files which are used by the factotum to grant the capability and the process to use that capability.

2.3 Pluggable Authentication Module

We have implemented a Pluggable Authentication Module *(pam_devcap)* that is an interface for the process to authenticate against the host owner's factotum. With regard to the security limitations of PAM discussed in the earlier sections, it should be noted that PAM does not perform the actual authentication. It is just used as an interface to interact with the factotum which performs the actual authentication. In this manner, all authentication code runs in a separate address space from the application instead of being linked in as a shared library. By using the PAM framework, the use of factotum to authenticate users is transparent to applications using the existing PAM API.

The user process passes the target user's credentials to *pam_devcap* which authenticates them against the host owner's factotum. *pam_devcap* uses the *p9cr* authentication protocol to talk to the host owner's factotum. It gets back from the factotum a capability in the form of a string which it stores for later use. Whenever the user id needs to be changed, the application requests the PAM module to do so, which retrieves the stored capability and writes it to `/dev/capuse` after which the process runs on behalf of the new user.

3. OVERVIEW OF THE PORT

Figure 1 shows a sequence diagram of the overall flow between a user process that wants to change a user id, the PAM module and the host owner's factotum. The user process contacts *pam_devcap* initially to authenticate the target user's credentials. The PAM module contacts the host owner's factotum and talks to it using the *p9cr* protocol. *p9cr* is a textual challenge-response protocol which is typically done between a factotum and a local program and not between two factotums as in the case of other authentication protocols in Plan 9. The protocol with factotum is textual where in the client writes a user name, server responds with a challenge, client writes a response, server responds with `ok` or `bad`. Typically this information that is being exchanged is wrapped in other protocols like *p9sk1* by the local programs before being sent over the network.

The factotum using the information from the authentication server validates the target user's information. If the credentials (user name and password) are valid then it creates a capability, hashes it and writes it to `/dev/caphash` which is a write only file opened by the host owner's factotum at boot time. It also passes this back to the PAM module. Now the PAM module stores this capability for later use and sends back PAM_SUCCESS to the user process informing the successful authentication of the target user. When the process wants to change the user id it contacts the PAM module again which basically writes the earlier saved capability to `/dev/capuse` and returns PAM_SUCCESS. The application now runs on behalf of the target user.

4. IMPLEMENTATION

We shall now describe the implementation details of each of the components that are needed in Linux to use the Plan 9 authentication mechanisms.

4.1 Capability Device for Linux

Figure 2 shows the flow of control in the cap device.

1. A process running on behalf of *user1* sends a message to the the host owner's factotum requesting a capability to change its user id from *user1* to *user2*.
2. The host owner's factoum performs an authentication protocol(*p9cr*) with the requesting process to authenticate the target user's credentials provided by the requesting process.
3. Once the host owner's factotum knows that the requesting process gets the valid credentials of the *user2* it creates a capability of the form *user1@user2@randomString* and writes a HMAC SHA1 hash of it to `/dev/caphash`.
4. The kernel internally maintains a linked list of these hashes written by the host owner's factotum. It adds the newly created hash to the list.
5. The requesting process get the capability string and simply writes it to `/dev/capuse`.
6. Once the capability is written to `/dev/capuse` the kernel checks whether the process requesting the uid change is actually running on behalf of *user1*. If the process is not, then the kernel throws an error and the write operation fails.
7. The kernel now splits the capability string and creates a HMAC SHA1 hash of *user1@user2* with the *randomString*.

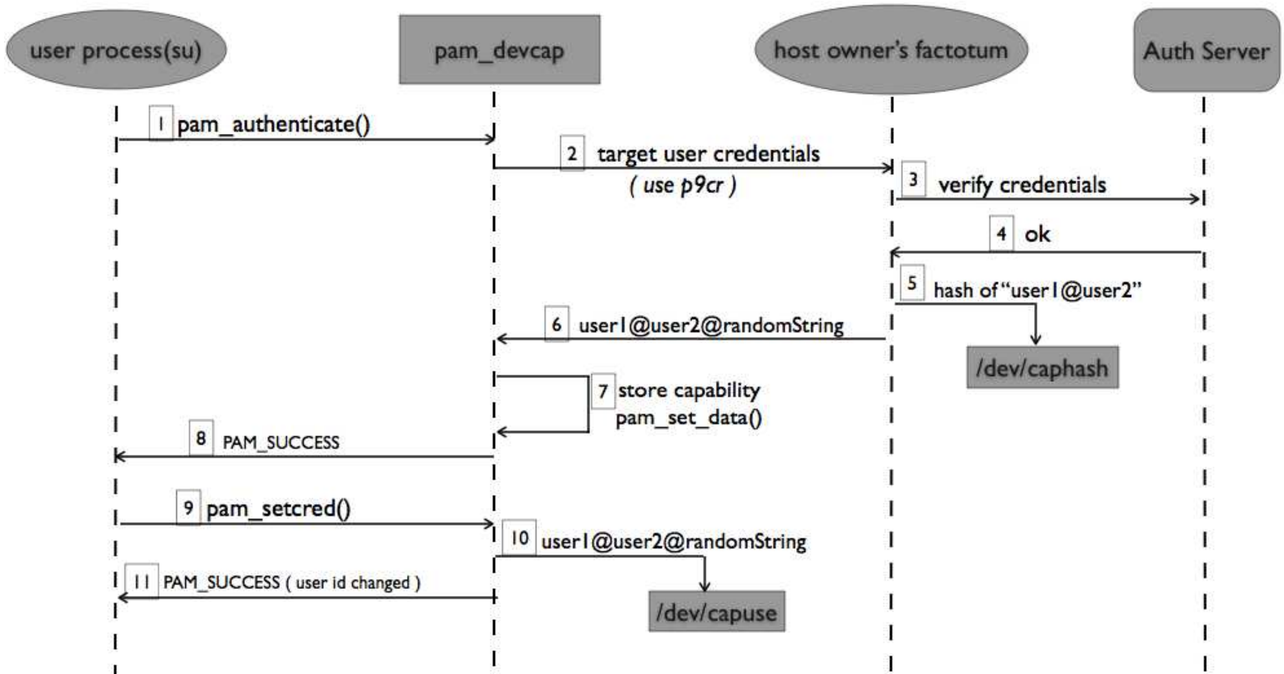

Figure 1: Overview of the port

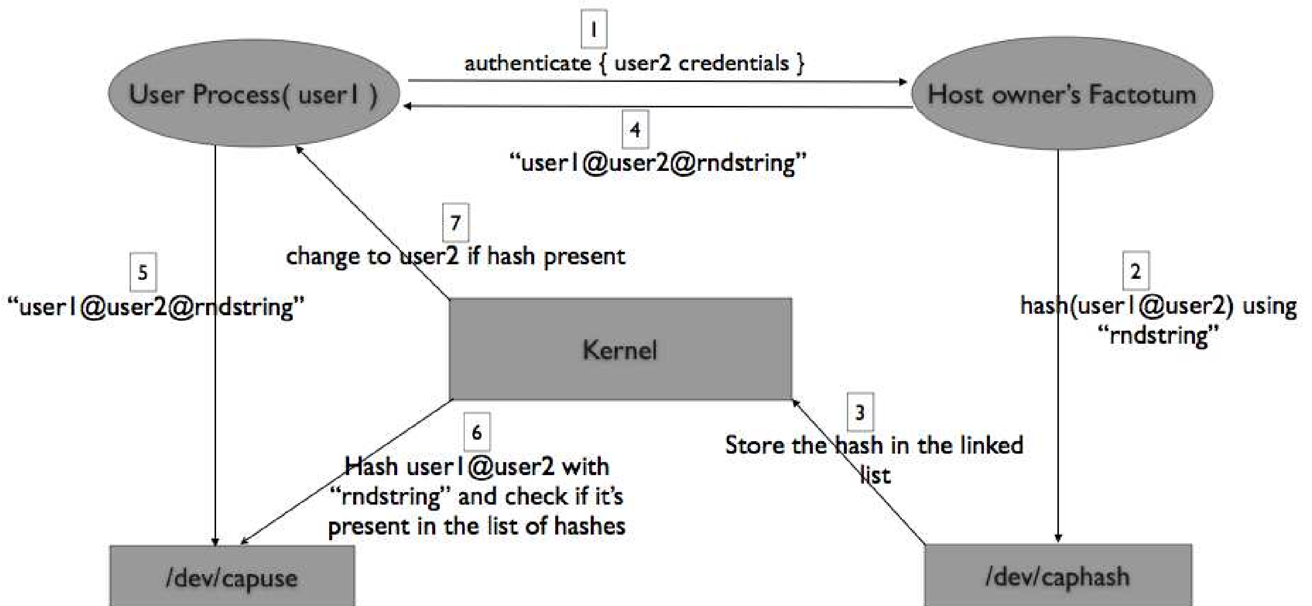

Figure 2: Flow of control in the cap device

8. It does a linear search amongst the existing list of hashes for this hash and if it finds a match it changes the effective user id of the process to *user2*.

9. Once used the capability is discarded by the kernel, i.e. it is removed from the list of hashes.

10. If there is no match found, or if there is a time out on the capability then the kernel returns an error and the write operation to `/dev/capuse` fails.

4.2 Authentication server

We have made minimal changes to the existing authentication server code in Plan 9 and ported it to Linux using the libraries provided by *Plan 9 from User Space* in Linux. The authentication server, as of now, needs to be set up using xinetd or the like. This is only a temporary solution. We plan to avoid xinetd and setup the auth server in a more graceful manner in the future.

4.2.1 Steps to setup the authentication server under xinetd

1. Create the /etc/xinetd.d/authsrv file containing the following:

```
service authsrv
{
  socket_type = stream
  protocol = tcp
  wait = no
  user = root
  server = /usr/local/plan9/bin/authsrv
  server_args = -d
}
```

2. Create an entry in /etc/services

```
authsrv  567/tcp  #Plan 9 auth server
```

3. Make an entry in $PLAN9/ndb/local for the auth server

```
 authdomain = <domain of the machine
running the auth server>
   auth = <host name of the machine running
   auth server> port = 567
```

4. Make sure the `$NAMESPACE` environment variable is set. This is essential for the authentication server to run. Typically it defaults to `/tmp/ns.$USER.$DISPLAY`. Set it manually if not set by default using the following command :

```
export NAMESPACE=<directory path>
```

5. Restart xinetd

```
sudo /etc/init.d/xinetd restart
```

4.2.2 User Account Setup

The authentication information of Linux users is generally handled by storing them in /etc/ files or some kind of naming service (configurable through *nsswitch.conf*). But the authentication protocol(*p9cr*) that happens between the host owner's factotum and the process is brokered by the authentication server. The authentication server in Plan 9 stores the shared keys of the users including the account information. Since the Plan 9 shared key protocols(*p9cr*) are being used to authenticate the users to the host owner's factotum it is necessary for the existing user's accounts to be created again in the authentication server. We have ported the Plan 9 *changeuser* utility to help add user accounts to the auth server.

```
changeuser <userid>
```

Users are represented by integer uids in the Linux kernel unlike the Plan 9 case where the users are represented as string identifiers. The capability that is written to `/dev/caphash` and `/dev/capuse` is of the form *user1@user2@randomstring* where *user1* and *user2* are the string values of the user identifiers in the case of Plan 9.

NFS solves a similar problem by having a user space mapping daemon that is contacted by the kernel to get the integer user ids for the string names that are supplied. The easiest way to solve this problem in our case was to simply create the user names in the authentication server by the string equivalents of the uids (for example: *changeuser 1000*). Now when the capability is created by host owner's factotum it would be the actual integer userids in string form that the kernel can understand. Whenever the capability is written to the kernel it is converted to an integer. Since the kernel understands the integer equivalent of this it simply changes the userid of the process to the target user id thereby avoiding the requirement of a user space mapping daemon.

Please remember that this is a temporary solution. It would be preferable to have a user space mapping daemon in the long run.

4.3 Pluggable Authentication Module

The PAM module *pam_devcap* authenticates against the host owner's factotum, retrieves the capability and whenever the user space application wants to change the user credentials, writes the capability to `/dev/capuse`.

4.3.1 pam_devcap configuration with PAM framework

1. Copy the module's shared object file `pam_devcap.so` to `/lib/security`.

2. Include the following line in the beginning of the PAM configuration file for the application using this module - `su` in our example.

```
auth  requisite  pam_devcap.so
```

`auth` signifies the PAM module's interface type. Modules with this interface type authenticate the user by

a password. The module also can change the user's credentials such as Kerberos tickets or group memberships.

`requisite` is the control flag for the *pam_devcap* module. It tells PAM what to do with the result of the module(pass/fail). Since PAM modules can be stacked in a particular order, control flags decide in what way does the result of the current module affect the final authentication outcome of the user. The `requisite` flag tells PAM that the module has to return success for the authentication to continue. The user is notified immediately if the module fails.

`pam_devcap.so` is the name of the module that implements the PAM interface.

3. `pam_devcap.so` implements the service module's equivalent methods for the `auth` interface type i.e. *pam_sm_authenticate()* and *pam_sm_setcred()*. In *pam_sm_authenticate()* the PAM module authenticates against the host owner's factotum and retrieves the capability for the user. The capability is used to change the user id in *pam_sm_setcred()*.

4.3.2 Logical Flow of authentication using PAM

Refer to Figure 1 for the overall flow.

1 The application makes a call to the *pam_authenticate()* supplying the target user's credentials to the PAM module.

2 The PAM framework internally routes it the service module's (i.e. *pam_devcap*) implementation of the pam_authenticate() method which is *pam_sm_authenticate()*.

3 The authentication happens in the *pam_sm_authenticate()* method of the PAM module.

4 The target user's credentials are passed over to the host owner's factotum which validates them by contacting the authentication server. Upon successful authentication it returns the capability as a string to the PAM framework.

5 The PAM module saves this capability using *pam_set_data()* and returns PAM_SUCCESS to the user application.

6 When the application wants to change the user id it makes a call to the *pam_setcred()* function.

7 The service module equivalent of the *pam_setcred()* is the *pam_sm_setcred()* function which is internally called by the PAM framework. The PAM module retrieves the capability that it saved earlier using the *pam_get_data()* function. It writes this capability to `/dev/capuse` which results in the kernel changing the user id of the process to the target user id of the process.

4.3.3 Zero Application Code Change

While working on the PAM module we discovered that there might be no need to make *any* changes to the application's code in order to use this new authentication mechanism. The application calls *pam_authenticate()* to authenticate the user and then does a *setuid()* to the authenticated user. Since the purpose of this new authentication mechanism is to eliminate the need for applications to run as root and *setuid()*, we might expect that at the minimum we need to modify the code to remove the *setuid()* system calls. But we can avoid this and practically achieve a zero application code change.

Basically *pam_authenticate()* uses the *pam_devcap* PAM module which internally writes to the cap device to change the user id. Hence, by the time *pam_authenticate()* returns the user is already running on the behalf of the new user. So the subsequent *setuid()* call would effectively be a no-op. This means that it requires no code changes to user space applications to use the cap device as long as they use PAM to authenticate users.

Unfortunately the PAM module's implementation is not complete at this point to demonstrate this.

5. FUTURE WORK

We focused on implementing the capability device to the Linux kernel and tried to demonstrate its use through the PAM framework. Additional improvements on the port could make it more secure and help applications to seamlessly use this authentication mechanism.

- The Plan 9 kernel starts a factotum at boottime which runs on behalf of the host owner. In order to do this the kernel needs a credential to start the factotum. In a similar case, the authentication server requires a credential in order to start. Plan 9 uses a secret password stored in nvram which the kernel prevents anyone from reading and uses it start the authentication server and the host owner's factotum. A similar mechanism needs to be developed for use with Linux in order to maximize security.
- The authentication server that is currently ported does not have the *ndb* database file that implements the speaks-for relationship for the host id. If an application needs to use the speaks-for relation then it would be useful if ndb support is provided.
- The authentication server is currently setup to run with xinetd or the like. It would be useful to make it run as an independent service if more security is desired.
- If scalability is required then we recommend a user space mapping daemon that maps the string user names and the integer user ids. This daemon would be contacted by the kernel to resolve the user names it gets after parsing the capabilities written to the device files (by the process and the host owner's factotum).
- Linux uses Naming Services to retrieve the user's login information. Applications that are used to the *getpwnam()* calls to retrieve the user account informa-

tion might find an NSS module (configurable through nsswitch.conf) that contacts the Plan 9 authentication server useful.

6. RELATED WORK

Privilege separation by running a separate process for each user is widely used. `qmail` [5] forks off a new process(and then does a setuid) for each user to deliver email.

Linux implements the concept of Compartmented Mode Workstations that have split up root privileges into about 30 separate capabilities [4], including a *setuid* capability. The programs do not run as root and assume privileges when needed and drop them when they don't thus implementing least privilege.

SELinux [3] provides protection by limiting the privileges of a process based on the user on behalf of which it runs and also the process's executable.

In comparison to these methods the *cap device* provides fine-grain one-time use capability to limit the setuid privileges further by allowing only the host owner's factotum to create the capability and managing the capability system in the kernel thereby providing better security.

7. CONCLUSION

Applications like `su` and `login` can now run on behalf of an unprivileged user instead of being setuid to root. This proves that the capability device for the Linux kernel is a very useful authentication mechanism considering the existing alternatives.

Since these applications run as root, an attacker can get root access to the system by exploiting any kind of security flaw in the application (buffer overflows etc.). By implementing this new authentication mechanism, even if these applications are compromised we do not give away much when compared to the former case. This achieves better privilege separation thereby providing better security.

We have completed the implementation of the capability device for the kernel (about 300 lines of kernel code) and the port of the authentication server and Key File System of Plan 9 to Linux. Both the authentication server and keyfs use the framework provided by p9p (Plan 9 from User Space).

We still need to complete the PAM module's implementation in order for it to be used by user space applications like `su`. We are currently in the process of debugging the module after which it will be made available at [1]

We believe that the Plan 9 authentication and capability mechanisms can enhance the existing Linux security model, and hope that our prototype implementation can be used as a foundation for future improvements

8. ACKNOWLEDGMENTS

We would like to thank

- *Eric Van Hensbergen*[1] for his continual support throughout the entire work.
- my adviser *Prof. Jon Solworth*[2] for providing invaluable guidance throughout this research.
- *Latchesar Ionkov*[3] for giving me useful feedback and comments on the capability device driver's code.

This work was supported in part by the National Science Foundation under Grants No. 0627586 and 0551660. Any opinions, findings and conclusions or recommendations expressed in this paper are those of the author and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

[1] Implementation code url. http://code.google.com/p/p9authlinux/source/browse.

[2] Linux-pam. http://www.kernel.org/pub/linux/libs/pam.

[3] Security enhanced linux. http://www.nsa.gov/selinux/.

[4] J. L. Berger, J. Picciotto, J. P. L. Woodward, and P. T. Cummings. Compartmented mode workstation: Prototype highlights. *IEEE Trans. Softw. Eng.*, 16(6):608–618, 1990.

[5] D. Bernstein. Qmail. http://cr.yp.to/qmail.html.

[6] R. Cox. Plan 9 from user space. http://swtch.com/plan9port.

[7] R. Cox, E. Grosse, R. Pike, D. L. Presotto, and S. Quinlan. Security in plan 9. In *Proceedings of the 11th USENIX Security Symposium*, pages 3–16, Berkeley, CA, USA, 2002. USENIX Association.

[8] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[9] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. *12th USENIX Security Symposium*, August 2002.

[10] V. Samar. Unified login with pluggable authentication modules(pam). In *CCS '96: Proceedings of the 3rd ACM conference on Computer and communications security*, pages 1–10, New York, NY, USA, 1996. ACM Press.

[11] T. Ylonen. SSH - secure login connections over the internet. Proceedings of the 6th Security Symposium) (USENIX Association: Berkeley, CA):37, 1996.

[1] Research Staff Member in the Novel Systems Architecture group at IBM's Austin Research Lab

[2] http://www.rites.uic.edu/∼solworth/index.html

[3] Technical Staff Member at Los Alamos National Laboratory