Unprivileged login daemons in Linux

Serge Hallyn
IBM
serge@hallyn.com

Jonathan T. Beard
GeekNet / Sourceforge
jbeard@geek.net

Abstract

Login daemons require the ability to switch to the userid of any user who may legitimately log in. Linux provides neither a fine-grained setuid privilege which can be targeted at a particular userid, nor the ability for one privileged task to grant another task the setuid privilege. A login service must therefore always run with the ability to switch to any userid.

Plan 9 is a distributed operating system designed at Bell Labs to be a next generation improvement over Unix. While it is most famous for its central design principle - everything is a file - it is also known for simpler userid handling. It provides the ability to pass a setuid capability - a token which may be used by a task owned by one userid to switch to a particular new userid only once - through the /dev/caphash and /dev/capuse files. Ashwin Ganti has previously implemented these files in Linux. His p9auth device driver was available for a time as a staging driver. We have modified the concepts explored in his initial driver to better match Linux userid and groups semantics. We provide sample code for a p9auth server and a fully unprivileged login daemon. We also present a biased view of the pros and cons of the p9auth filesystem.

1 Introduction

For the past fifteen years, the Computer Security Institute has surveyed security practitioners and government and private institutions regarding security breaches and cybercrime [14]. In 2008, they surveyed 522 respondents with an average annual loss report just under $300,000. From 2004 through 2008, the number of respondents reporting unauthorized access (including privilege escalation) as a component of their reported attacks remained approximately a third. Similarly, in the mobile computing and console gaming arenas, jailbreaks through privilege escalation remain one of the leading security concerns for these platforms. As Linux forms the core operating system for a growing number of these devices [5], a solution that greatly reduces or eliminates opportunities for privilege escalation would represent a potential savings worth tens or hundreds of millions – at least from a comparison of potential losses due privilege escalation as described above.

One avenue to privilege escalation is the exploitation of flaws in privileged programs. For instance, a buffer overflow in the ping program might allow a regular unprivileged user to pass specific data as an argument to ping, invoking a shellcode resulting in a root shell. Therefore, it is a laudable goal to reduce the number of programs which need to run privileged.

We begin by reviewing the current design of login servers in Linux. We present two ways in which Linux could be extended to support making these login servers unprivileged. For the first, we provide sample code for both the privileged and unprivileged servers. We discuss the pros and cons of both approaches, in the hopes of raising a discussion on the merits of supporting one of the designs upstream.

2 Login services in Linux

Throughout this paper, familiarity with the user [11] and privilege [7] mechanisms of Linux is assumed. Tasks are owned by a numerical user ID and one or more numerical group IDs. These numerical IDs are known in userspace by textual names. Tasks also carry three sets of privileges which for historical reasons are called "POSIX capabilities" or just "capabilities". We usually say a task has some privilege if that privilege is in the task's "effective" set, in which case the task is allowed to escape certain access checks or perform sensitive tasks. For instance, the `CAP_DAC_OVERRIDE` privilege allows a task to read other users' private files and `CAP_SYS_BOOT` allows a task to reboot the system. Usually, user IDs and privileges are linked, and user ID

0 is special in that its privilege sets are full, while privilege sets for other user IDs are by default empty.

Tasks in Linux can switch user IDs in two ways. One is to execute a file with the "setuid bit" set. This bit indicates that when the file is executed, the "effective" user ID should be set to that of the file owner. The "real" user ID remains unchanged. The other way is to use the `setuid` family of system calls. This can be used by unprivileged tasks to switch values between the effective and real (and another, called "saved") user IDs. To set a user ID to an entirely new value requires the `CAP_SETUID` privilege. The changing of primary group IDs is analogous. Changing the set of auxiliary group IDs (for which there is no analogy with user IDs) always requires the `CAP_SETGID` privilege.

A simple login service in Linux can be structured as in Figure 1. The login daemon, running with process ID (PID) 300 waits for a username to be entered on its terminal, `/dev/tty`. Generally it then asks for a password on the same terminal. It can verify the validity of the password by checking hashed entries in `/etc/passwd` or `/etc/shadow`. If the password is valid, then the login task finds the user's default auxiliary groups, primary group ID, and user ID, switches to these credentials, and executes the user's default shell. The user now has a shell running with his credentials on `/dev/tty`.

Figure 1 also shows communication between login and PAM [15]. PAM is a set of libraries which offer authentication and session management services and ease system-wide configuration through a single set of configuration files. While these are great advantages over the alternative of each service implementing their own authentication and session management, library functions execute with the credentials of their caller. Therefore PAM does not help with reducing the amount of privilege required for login daemons to perform their authentication and to switch user IDs.

The principle of privilege separation [13] advises that a privileged program be structured so that privileged operations are pushed out into small, easier to verify, privileged helpers, each serving precisely one purpose. Then the bulk of the program can run without privilege. Figure 2 shows how the privilege-separate OpenSSH server manages to prevent the user from directly interacting, at any time, with a privileged process. A privileged parent process communicates with an unprivileged child running as an unused user ID (usually "ssh"). The child passes authentication communications to the parent and, if they are correct, then the parent ends the first unprivileged child and forks a new child which calls setresgid() and setresuid() to drop privilege and assume the authenticated user's identity.

While this can make successful privilege escalation attacks far less likely by reducing the types of messages which an attacker can cause to be sent to the privileged task, it does not completely eliminate attacks. While privilege is kept further away from the unprivileged user, the number of programs running with privilege are not reduced. The implementation of each login service is also greatly complicated.

3 Credentials passing

Unix domain sockets in Linux support the ancillary messages `SCM_RIGHTS` and `SCM_CREDENTIALS`. The first allows passing an open file descriptor to a target task. The second allows passing the sending task's credentials (user ID, group ID and PID) so the receiving task can verify the sender's identity. Alan Cox has proposed a new ancillary message type, which we call `SCM_AUTH`, which any task may use to convey the right for the recipient to switch to its own credentials. Such a feature could provide interesting new features, such as one unprivileged user granting another unprivileged user the temporary credentials needed to debug an environment problem for the first. Even a system service which is partially privileged (but without `CAP_SETUID`) or unprivileged could make use of this feature to act on behalf of its clients.

This proposal also elicited discussion about a related, less dangerous ancillary message for passing only audit credentials. These could be used to augment audit messages with the credentials of a client on whose behalf a back-end server was acting. This would make the audit messages more informative than if they carry only the server's credentials.

3.1 Unprivileged login based upon SCM_AUTH

The `SCM_AUTH` ancillary message type would facilitate the implementation of unprivileged login clients in Linux. A simple architecture for such a system is shown in Figure 3. A single privileged daemon can serve all unprivileged login servers. The login servers

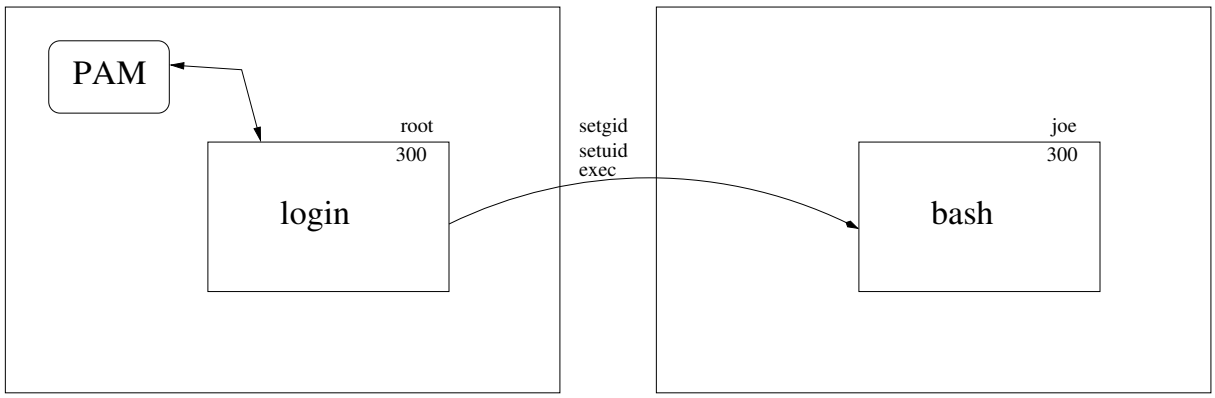

Figure 1: Simple login design using setuid.

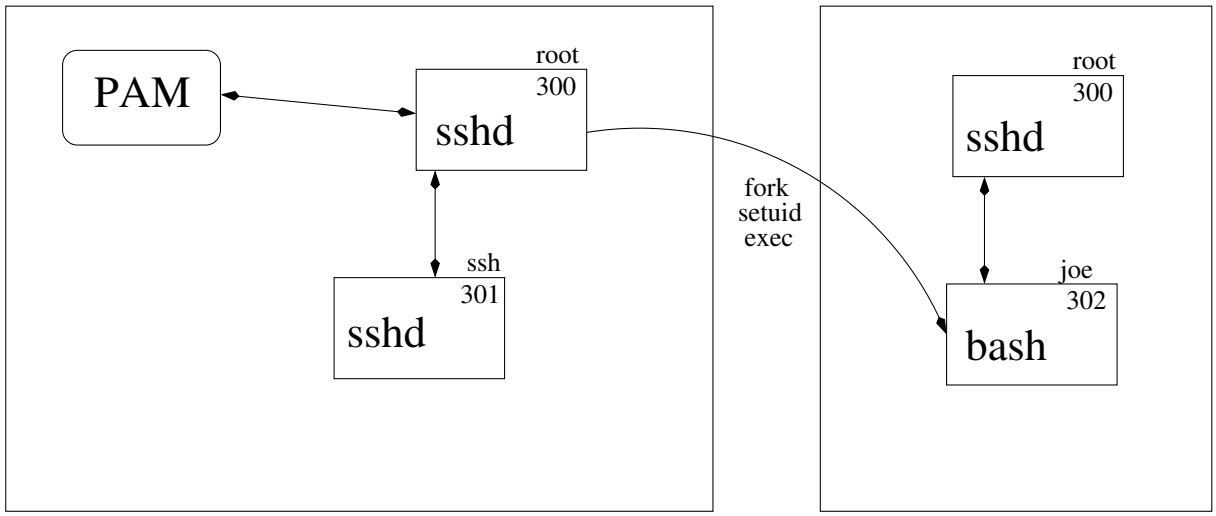

Figure 2: Login using privilege-separated OpenSSH.

act as a proxy to facilitate the communication allowing a user to authenticate his identity to the privileged server. Once satisfied of the user's identity, the privileged server forks a new task which establishes for itself the user's desired credentials using the standard `setresgid`, `setgroups` and `setresuid` system calls. It then passes a `SCM_AUTH` message to the login daemon, which uses a new `accept_id` system call to assume these received credentials. The login process can now proceed calling the user's login shell as it normally would after having explicitly called `setuid` if it had been running with privilege.

3.2 Security Considerations

When POSIX capabilities were first introduced, file capabilities remained unimplemented. In a POSIX capability system, file execution causes a task's permitted capability set to be recalculated. Any capabilities which end up in that set must be active in one of the executable file's capability sets. Therefore, without file capabilities, no task can have permitted capabilities after executing a file. As a way to achieve non-root partially privileged tasks, `capsetp` was implemented as a Linux-specific way to grant capabilities to another task [9]. Doing so required the `CAP_SETPCAP` privilege. However, the ability to grant privileges to another task was deemed so unsafe that the `CAP_SETPCAP` capability was placed in the system-wide capability bounding set, so that effectively no one was ever able to employ it.

Likewise, passing credentials over a UNIX socket could be seen as too dangerous a way to spread privileges and access rights. It particularly spreads the risk of any pro-

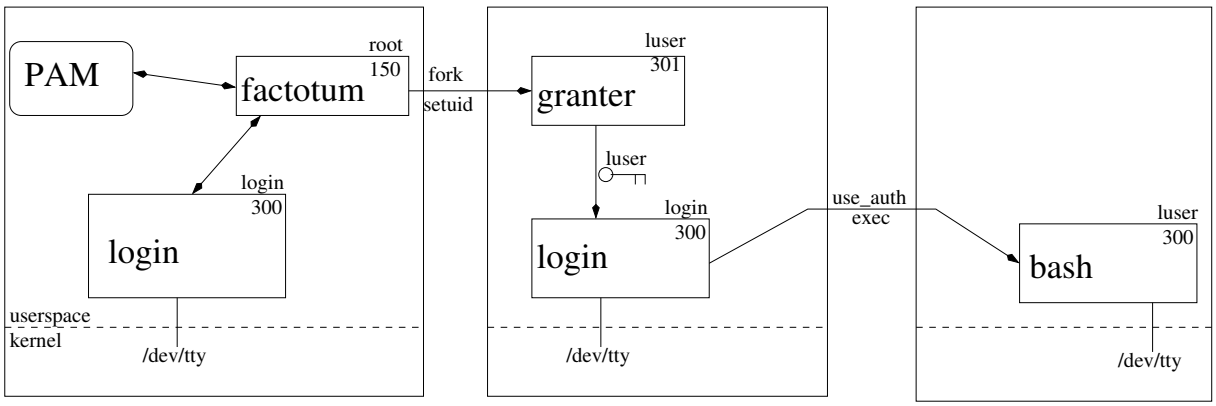

Figure 3: Login design using SCM_AUTH.

gram running with `CAP_SETUID`, since any task which may arbitrarily set its user ID can then pass the resulting credentials along to any other task. It also makes a Trojan even more dangerous. The Trojan now does not need to immediately do its damage, open a back door, release the information it targets, or hide a task which it creates with the victim's credentials. Instead, it can simply transfer the conquered credentials to the attacker who can stash them away for later. This could be undetectable.

One part of a defense against such a situation is notion of a timeout on the passed credentials. This could help prevent the tokens being too liberally passed around, accidentally inherited by an untrusted child task, or simply stashed away as a long term privilege token for later use. Analysis of used and stashed tokens could also help mitigate the dangers. Both the sending and the use of tokens should obviously be audited. Additionally, a list of sent but unused tokens and details on the sending and receiving tasks should be available, possibly through a `/proc` interface.

4 The p9auth filesystem

Ashwin Ganti implemented [6] a device driver for Linux which implemented the Plan 9 "capability [1] device" [2]. Briefly, an unprivileged login client persuades a privileged daemon that it is authorized to change to a particular user ID. The privileged daemon writes to the `/dev/caphash` device a token consisting of the client's current user ID and the authorized new user ID joined together with a special separating character ('@') and hashed with a random string. This token is then appended by the kernel to a list of such hashes. The daemon communicates the random string to the client, which can then use the token by writing the unhashed values (old user ID, new user ID, and random value, separated by '@') to `/dev/capuse`. This causes the client's real and effective user IDs to be changed. Since this was a proof of concept, some shortcuts were taken. For instance, the saved user ID is not updated [2], the primary and auxiliary groups remain unchanged, and the filesystem user ID (fsuid), which is used for filesystem accesses and by convention mirrors all effective user ID changes, is not updated.

We have changed the Linux p9auth semantics to better suit the Linux user and group behavior and simplify usage. The p9auth setuid capability now contains the original user ID, new user ID, a new primary group, and a list of auxiliary groups. Upon presenting a valid capability, a task's real, saved, effective, and filesystem user and group IDs are all assigned to the new values. While different effective and real user IDs can be useful for applications to get things done, a newly logged-in process needs a simple, sane, and useful initial set of credentials. We have also changed from a device interface to a more standard one using a custom filesystem. We have also implemented a timeout on setuid capabilities, placed a limit on the number of unused capabilities stored in the kernel, and made the implementation user namespace aware as will be discusssed later.

Figure 4 shows how login services can be structured

[1] Capability here is used in its classical sense, not as the unfortunately chosen pseudonym for a privilege.

[2] This may be intended as a feature to increase flexibility at the cost of more complicated users.

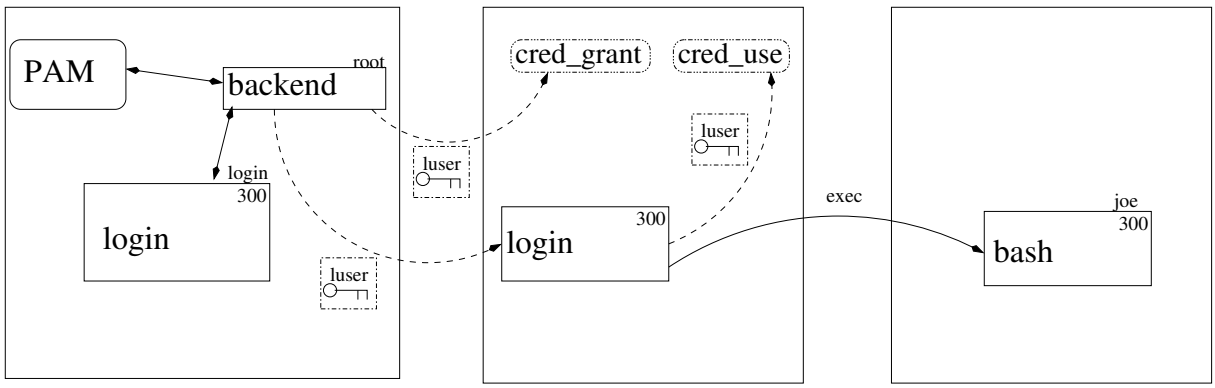

Figure 4: Login design using p9auth setuid tokens.

with the new p9auth filesystem. A single privileged backend service, called p9auth, can serve all login clients, and need never interact directly with users. An unprivileged login client, perhaps running with user ID "login", interacts with the user and passes the communication for an authentication protocol between the terminal and the privileged p9auth service. This service itself uses PAM for the actual authentication. Assuming authentication succeeds, p9auth looks up the initial credentials for the new user and creates a random string (`XYZ`) and a p9auth capability token which looks something like `121@1001@1001@0` In this example user ID 121 may switch to user ID 1001, primary group 1001, and no auxiliary groups. P9auth hashes the capability token with the random string and writes that to the `cred_grant` file in the p9auth filesystem. It also passes the token and the random string to the user. The user then passes `121@1001@1001@0@XYZ` to the `cred_use` file to effect the change of credentials.

From a higher level, we see that, as with `SCM_AUTH` ancillary messages, an unprivileged login client was made possible by providing a way for the ability to switch user IDs to be sent between tasks. In this case, one isolated privileged task can send to unprivileged console or network-facing tasks tokens representing the ability to switch to a particular user ID.

4.1 A proof of concept p9auth service

Our goal is for a single p9auth service to serve all unprivileged login servers. For simplicity, it will communicate with them over a Unix file socket. Given that secrets like passwords will be sent over this socket, the communication protocol should begin with the p9auth service proving its own identity to the unprivileged login daemon. Details of such an algorithm are outside the scope of this paper, but could be based on a certificate or public key pair for the p9auth daemon.

A proof of concept (POC) implementation of a p9auth server and an unprivileged login client can be found at github [**?**] [3]. It is based upon the p9auth filesystem patchset which as of this writing was out of tree, and can be found at [**?**]. Since the userspace code is POC and does not implement server authentication, the client, called `frontend` and shown as pseudo code in Figure 5, simply connects to the Unix socket `/var/run/factotum` and assumes it is talking to the valid backend server. It writes its current user ID and desired new username, then serves as a PAM relay until it receives the desired setuid token in a messages beginning with "FINAL: ". It then writes that token to the `cred_use` file in the `p9auth` filesystem. That write triggers a callback in the kernel which effects the group and user ID changes. Finally, it executes the user's specified shell to complete the login.

The privileged server, called `backend` and shown in Figure 6, listens on the Unix socket for requests from login clients. It uses the `sock_conv()` conversation function (based on an example by Andrew Morgan [**?**]) to allow PAM, called with privilege by `backend` and guided by the system's PAM configuration file `/etc/pam.d/factotum`, to perform the actual authentication.

Assuming the PAM authentication succeeds, `backend`

[3] Note that this is purely a proof of concept, and not intended to be used as-is. For instance, the client does not currently clear the environment before executing the login shell.

```
void client(int sfd)
{
    sprintf(buf, "olduid %d\nusername %s\n", getuid(), username);
    write(sfd, buf, strlen(buf)+1);
    while (1) {
        read(sfd, buf, MAXLINE);
        if (strncmp(buf, "FINAL: ", 7) == 0) {
            break;
        ...
                } else if (strncmp(buf, "ECHO: ", 6) == 0 ||
                        strncmp(buf, "NOECHO: ", 8) == 0) {
                        /* get_input reads a user's response */
                        get_input(buf, buf[0] == ,E,);
                        write(sfd, buf, strlen(buf)+1);
                }
        }
        cfd = open("/mnt/p9auth/cred_use", O_WRONLY);
        p = buf + 7;
        write(cfd, p, strlen(p));

    shell = getpwnam(username)->pw_shell;
    execl(shell, shell, NULL);
}
```

Figure 5: Pseudo code for unprivileged p9auth login client.

generates a random string and a login token as described in Section 4, writing the encrypted hash of the login token with the random string to the `cred_grant` file and passing the concatenated token and random string to the client.

The p9auth filesystem is user-namespace-aware. Any setuid tokens granted by a particular privileged p9auth server are tied to that server's user namespace. Therefore, a p9auth server in a container cannot be used to bypass the host's p9auth server. At the same time, since each container will have its own `/var/run` directory and therefore its own `/var/run/factotum` socket, any properly configured container will be able to offer its own privileged p9auth server for use within the container.

4.2 Security Considerations

As with credentials passing, we must consider whether the ability to grant the ability to switch to new credentials with the p9auth concept should raise the same concerns as did transferring POSIX capabilities using `capsetp`. Just as with granting capabilities to third parties, we essentially have a privileged task which is allowed to grant to other tasks the privilege to switch to new user IDs. This is especially true since on most systems, by default, switching to the root user ID also raises all capabilities.

However, this concern is predicated on the risk of spreading privileges because p9auth can authorize a change of the user ID for an existing process. In fact, the opposite case is true and is described in the example below. First, assume that some unprivileged process, PID 3451, improperly manages to get p9auth to authorize its setuid to 0. This is no worse than if the same process tricks a privileged login service into forking a new process with user ID 0, running a binary specified by the malicious process. Additionally, since a single privileged p9auth service allows us to remove the `CAP_SETUID` capability from all privileged login services, like su, sudo, login, sshd, ftp, etc. By running those services completely unprivileged, this approach actually greatly decreases the amount of potentially vulnerable privileged code.

```
void handle_client(int clientfd)
{
    read(clientfd, buf, MAXLINE);
    sscanf(buf, "olduid %d\nusername %s\n", &olduid, username);
    /*
     * validate() performs PAM authentication and returns
     * the password entry (struct passwd *pe) for username.
     */
    struct passwd *pe = validate(clientfd, username);

    /*
     * readgroups() places all username's auxiliary groups
     * into gid_t *groups
     */
    numgroups = readgroups(username, pw->pw_gid, &groups);

    /*
     * make_token places the string
     *      olduid@newuid@newgrp@numgroups@grp1@...@grpn
     * into char *token
     */
    make_token(token, olduid, pe, numgroups, groups);

    /* generate a hash of the token and hand that to the kernel */
    len = generate_hash(token, hash, randstr);
    capfd = open("/mnt/p9auth/cred_grant", O_RDWR);
    write(capfd, hash, len);

    /* write the token and the random string to the client */
    sprintf(clientstr, "FINAL: %s@%s", token, randstr);
    write(clientfd, clientstr, strlen(clientstr)+1);
}
```

Figure 6: Pseudo code for p9auth backend.

Another advantage of the approach is that in a completely converted system we may have the init daemon fork off the p9auth service early and then drop CAP_SETUID from its capability bounding set, so that all other privileged daemons will not be able to expose the system to CAP_SETUID empowered rootkit exploits.

5 Other privilege needs

While the above designs, and the POC implementation of the p9auth-based unprivileged login daemon suffice for simple cases, some sites require additional setup at login. That setup itself may require privilege. For instance, in order to provide polyinstantiated directories, login session to an LSPP [3] system may require a private mounts namespace and custom directories to be mounted according to their privilege level. User joe logged in as "secret" sees a different /tmp than the same user logged in as "unclassified". Other sites may require a Smack [16] label to be specified at login, which requires the CAP_MAC_ADMIN privilege. Still other sites may wish to set up an initial inheritable capability set [7], which requires the login daemon to have CAP_SETPCAP.

These obstacles ought not be insurmountable. For in-

stance, it it is hoped that both creating private namespaces and directory mounting will eventually be allowed for unprivileged users. The specification of Smack labels and initial capabilities could be added as extensions to the p9auth token specification, while the `SCM_AUTH` ancillary message could well represent full credentials, including security labels. Finally, while these designs remove the need for `CAP_SETUID` and `CAP_SETGID` from login daemons, other file capabilities can be added to the login daemon binaries if needed. Upon the execution of the login shell, the effective and permitted capabilities will be recalculated, so while it would be preferable to have the login daemon entirely unprivileged, some capabilities could be enabled if needed.

More generally, a reliable way must be found to set appropriate initial LSM (security) contexts upon login. Proper behavior will differ per LSM. TOMOYO contexts are meant to purely reflect the history of executed files, so a TOMOYO context should likely not be passable with `SCM_AUTH` or p9auth. SELinux contexts are the results of domain transitions initiated by execution of carefully designated "entry points". The login daemons can remain unprivileged, but their execution can trigger entry into a domain which can select an initial security context for the user's session. For POSIX capabilities, it would seem desirable to avoid the need to grant `CAP_SETPCAP` to an unprivileged login daemon by allowing the privileged backend daemon to specify an inheritable capability set. Passing permitted and effective capabilities may be found to have uses for non-login applications of `SCM_AUTH`, but is not useful for login daemons. We therefore may well want only the inheritable set to be specified. Smack contexts are generally set by userspace, but require the `CAP_MAC_ADMIN` capability to set, and therefore should not be trivially changeable, although allowing tasks with `CAP_MAC_ADMIN` to transfer their Smack label along with their credentials may be desirable.

Since LSMs may vary in how they treat the security context passed through `SCM_AUTH` or granted through p9auth, it seems prudent to provide a new LSM [**?**] hook which processes a set of initial and intended credentials, and produces the final LSM context for the login session.

6 Conclusion

Exploitation of bugs in privileged programs can allow an unprivileged user to illegitimately escalate privilege, and can serve as the second step in an outsider attack which begins with hijacking of a local unprivileged account. To reduce the potential for such attacks, it is desirable to reduce the amount of code which must run with privilege. We have presented two ways in which Linux can be extended to support unprivileged login daemons.

Both the `SCM_AUTH` and p9auth designs remove the need for login services to run with `CAP_SETUID` and `CAP_SETGID` privileges, concentrating privilege instead in a single system-wide authentication service. Without this support, each service needing to switch user IDs needs to be installed with the privilege to switch to any user ID. In that case, it requires constant vigilance and multiple redundancy to limit the privilege granted to the immediate user-facing terminal program – with every single instance being a potential oversight leading to exploit. With this support, the number of programs requiring privilege is reduced, and with it the gross odds of a successful privilege escalation attack.

7 Legal Statement

8 Acknowledgments

The authors would like to thank Ashwin Ganti for writing the original p9auth capability device, Greg Kroah-Hartman for the drivers staging tree and for hosting the p9auth driver, and Alan Cox for once again providing an interesting new insight into a proposed design.

References

[1] Alan Cox. *Regarding add p9auth driver,* http://lkml.org/lkml/2010/4/21/88, Apr 21 2010.

[2] Russ Cox, Eric Grosse, Rob Pike, Dave PResotto, Sean Quinlan, "Security in Plan 9", Proceedings of the 2002 Usenix Security Symposium, San Francisco.

[3] Janak Desai, George Wilson and Chad Sellers. *Extending SELinux to meet LSPP data import/export requirements,* Proceedings of the 2006 Security Enhanced Linux Symposium, March 2006.

[4] Chris Friedhoff. *POSIX Capabilities and File POSIX Capabilities,* http://www.friedhoff.org/posixfilecaps.html.

[5] C. Gallen. *Linux to Be the Fastest-Growing Smartphone OS over the Next 5 Years* ABI Research. August 2007. http://www.abiresearch.com/press/922.

[6] Ashwin Ganti. *Plan 9 authentication in Linux*, Operating Systems Review 42(5): 27-33 (2008).

[7] Serge E. Hallyn and Andrew G. Morgan. *Linux Capabilities: making them work,* Proceedings of the Ottawa Linux Symposium, July 2008.

[8] Linux man-pages project. *Capabilities.7 man page,* http://linux.die.net/man/7/capabilities.

[9] Andrew Morgan. *capsetp(3) manpage,* http://linux.die.net/man/3/capsetp.

[10] C. Ozancin. *Securing the linux environment: Programs that need root access.* pages 42âĂŞ45, March/April 2001.

[11] Dan Tsafrir and Dilma Da Silva and David Wagner. *The Murky Issue of Changing Process Identity: Revising "Setuid Demysitified.",* Linux Journal.

[12] J. Saltzer and M. Schroeder. *The protection of information in computer systems.* Fourth ACM Symposium on Operating System Principles (October 1973).

[13] N. Provos, M. Friedl, and P. Honeyman. *Preventing Privilege Escalation* Security '03 Paper, USENIX Security '03 Technical Program, 2003.

[14] R. Richardson. *2008 CSI Computer Crime & Security Survey*. Computer Security Institute. 2008.

[15] 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.

[16] Casey Schaufler. *Smack and the Application Ecosystem,* http://linuxplumbersconf.org/2009/slides/Casey-SmackPlumbers2010.pdf.

[17] Ylonen, T., "SSH-Secure Login Connections Over the Internet", 6th USENIX Security Symposium, pp 37-42. San Jose, CA, July 1996.

Proceedings of the Linux Symposium

July 13th–16th, 2010
Ottawa, Ontario
Canada

Conference Organizers

Programme Committee

Proceedings Committee