

Abolish Root Daemons!

Carol Hurwitz

Scott McPeak

{hurwitz,smcpeak}@cs.berkeley.edu

February 12, 2001

Abstract

Network service software (daemons) must perform tasks that affect system security. Most existing Unix systems address this by only allowing privileged processes to do these tasks. Moreover, they often must grant *all* system privileges to processes which perform any such tasks. The problem with granting all privileges is that subversion of the daemon (e.g. by buffer overflow) immediately leads to total compromise of the system.

Our solution to this problem is to change the operating system primitives, to allow daemons to complete their tasks while running with fewer (if any) special privileges. We use three techniques: an atomic primitive for authentication, direct support for sandboxing, and support for granting limited privileges such as binding a privileged port. These techniques are not entirely new; the main contribution of this paper is their application to a legacy Unix environment.

We argue that by reducing the privileges granted to network services, we lengthen the path to complete compromise, and thereby provide substantial security benefits.

1 Introduction

Network service software (daemons) perform tasks that affect system security. Chief among these are authenticating and acting on behalf of remote users. These services may also choose to run in a self-imposed limited-access environment, to reduce the impact of compromise. Finally, network daemon software may need to perform explicitly privileged

operations, such as binding low-numbered ports.

Most existing Unix systems only allow processes with elevated privilege to perform security-relevant activities. Further, there is often no way to grant a process just the required privileges. This makes network daemons an attractive target for attackers, because their elevated privilege is then inherited by an attacker if she can successfully gain control of such a daemon.

Some new systems address these issues, but the large installed base of Unix systems presents many attractive targets for attackers. The challenge is to retrofit the entire Unix system, both kernel and network daemon software, to incorporate ways to limit the impact of daemon compromise. This is difficult because the software was not designed with these ideas in mind, yet we want to keep the required changes to an absolute minimum so that they will be accepted by the users and maintainers of this software.

In this paper, we discuss ways to allow network daemons to perform their intended tasks without giving them undue privilege. One technique we use is to create new primitives which directly implement the privileged tasks daemons need to perform. An example is authenticating a user by checking her password, and then gaining authority to access that user's files: we combine these two tasks into a single atomic operation, which is then secure for unprivileged, untrusted processes to use. We move the activity previously done by the daemon software itself, buried inside all of the other complex functionality, to a small, separate, trusted subsystem.

We also identify those tasks which inherently require some level of *a priori* authorization, and provide mechanisms to let the system administrator grant specific authority to do those tasks. We refine and extend the traditional Unix group mechanism, so

that groups become the carriers of privilege. While applying the Principle of Least Privilege [6] is not new, it is particularly effective in combination with the other techniques presented.

Throughout, the changes to the operating system are intended to minimize changes to legacy code. In most cases, no change to the daemon software itself is required; instead, configuration files or file system permissions are all that must be changed. For those cases where changes have to be made, the changes fortunately are small and easy to understand.

We argue that by reducing the special privileges possessed by network daemon processes, we make it harder for an attacker to leverage a software weakness into complete control of the system. While this does not by itself provide absolute security, in combination with other prudent measures, this is a powerful element of a layered security system.

2 Tasks performed by daemons

In this section we identify several tasks network daemons need to perform, but which in traditional Unix systems require `root` (superuser) privileges to do. For each task, we provide some alternative to running as `root` that still lets the daemon do its work.

2.1 Authentication

Many network services must perform authentication. Among these services are the popular FTP (File Transfer Protocol) and SSH (Secure Shell). FTP daemons in particular have been notorious for buffer overflow vulnerabilities (e.g. [2]). Currently, both `ftpd(8)`¹ and `sshd(8)` (the respective daemons) must run as `root` in order to authenticate the user who is trying to establish a connection.²

Authentication by username and password usually requires access to database files such as `/etc/passwd` and `/etc/shadow`. To access the latter file requires `root` privileges in modern installations. Following a

¹Using standard Unix manual page notation: (1) means user command, (2) means system call, (8) means administrator command.

²Another security problem with FTP is that it sends passwords over the network in the clear. This concern is orthogonal to those addressed in this paper.

successful check of the username and password, the daemon then changes the user id of the process to that of the user requesting the service. To change the user id, a process calls `setuid(2)`, again an operation that must be done by a process running as `root`.

To provide a mechanism by which a service can authenticate a username and password without running as `root`, we build a trusted subsystem through which the daemon can request authentication. The subsystem is accessed via an `ioctl(2)` call to a virtual device, which forwards the request to a trusted process.

The trusted activity, formerly handled by `ftpd` itself, is now handled by a small trusted program, called `asp(8)` (Authentication Server Process). `asp`, running as `root`, waits to receive a request for authentication. Upon receipt of a request, it accesses the files required to check the username and password, namely, `/etc/passwd` and `/etc/shadow`. If the username and password match, `asp` simply answers “yes”; the virtual device then will make `setuid` calls to change the user and group id of the service to that of the authenticated user, and return. If the username and password do not match, then `asp` simply returns “no”.

More precisely, the effective user and group id are changed to that of the authenticated user, while the real user and group id remain unchanged. At that point the process has both credentials, so existing software (which assumes it has `root` privileges) is likely to still work. Additionally, because the software has elevated credentials, it is not possible for an ordinary user process to send it signals, `ptrace(2)` it, etc., so it can safely destroy security tokens such as passwords before dropping its original credentials.

We provide two measures to protect the interface from brute-force abuse. First, on every reply, `asp` can specify a delay time in microseconds; the kernel will hold the reply for this period of time (*before* switching credentials, if the answer is “yes”). Second, `asp` has the option to reply “too many tries”, which it will use if it receives too many attempts to authenticate a given user in some time period (this defeats an attacker attempting to circumvent the delay time by forking many processes).

A key feature of this design is that, while the kernel is involved in mediating the authentication ex-

change, it does not need to know what the exact authentication method is. For example, `asp` could check credentials either in a local password database or in a distributed system such as NIS (Network Information Service). Preventing the kernel from knowing the authentication method means that system administrators are free to use whichever methods are appropriate to their sites.

More abstractly, we can think of the login process as an authentication step followed by an authorization step. Checking that the username and password match is authentication: it establishes the identity of the principal supplying this information. Changing the process' user id establishes the limits of what it is authorized to do. Traditionally, these are separate, independent tasks as far the operating system kernel is concerned, which is exactly why `root` must do them: we expect successful authentication to precede authorization, but nothing in the kernel enforces this. However, when we combine these operations by putting both into a trusted subsystem, the atomic combination is secure for unprivileged, untrusted processes to use directly.

To take advantage of the new mechanism, a daemon must be changed so that it no longer does its own authentication but instead requests it from the trusted subsystem. For example, in the case of `ftpd`, we replaced the code that directly accessed `/etc/shadow` with our own code, which calls `ioctl`.

However, many daemons have been augmented to use the PAM (Pluggable Authentication Modules) [7] interface for authentication. We have implemented a PAM module which does the necessary `ioctl` call internally; thus, a PAM-aware daemon need not be changed at all.

2.2 Sandboxing

Prudent network daemons put themselves into *sandboxes*. A sandbox is an environment in which a process has fewer privileges than it would ordinarily. This is done to limit the impact of compromising the daemon, which is in general consistent with the theme of this paper.

Two types of (crude) sandboxing mechanisms are typically available in Unix systems: unprivileged users, and `chroot`. We treat each in turn.

2.2.1 Sandboxing with users

When an FTP connection is requested by an anonymous user, `ftpd` wants to run as an unprivileged user, `ftp`, for the duration of the anonymous session. This is a form of sandboxing; if `ftpd` is subverted during an anonymous session, it will be unable to interfere with named (as opposed to anonymous) FTP sessions.

We generalize this to the notion of a user hierarchy, which we illustrate in Figure 1. Users `scott` and `carol` are ordinary unprivileged users. `initftp` can become `ftp` at will, and `scott` can become `sandboxscott`. `carol` and `sshd` cannot become anyone else. `root` can become anyone. In the case of `ftpd`, we let it run initially as `initftp`; `ftpd` can then make the switch whenever an anonymous user requests service.

To implement this, we augment the authentication interface: we add to `asp` the alternative of authorization via the user hierarchy, specified in a system configuration file, `/etc/sandbox_users`. The system administrator creates and maintains this file. Entries in this file are of the form "A -> B", which means A can become B.

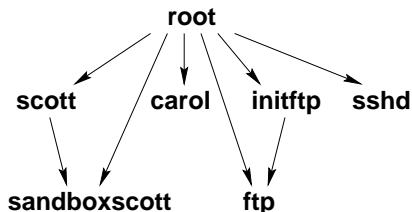
2.2.2 Sandboxing with chroot

The `chroot(2)` mechanism, available in most Unix systems, redefines the meaning of `/`, the root directory of the file system. For example, if a process calls `chroot("/tmp/foo")`, then `open("/bar/baz")`, the file actually opened is `/tmp/foo/bar/baz`. Network services can use this to limit the potential damage of a compromise: upon compromise, the attacker is effectively trapped in the `chroot` "jail", unable to access files outside the subtree rooted at the new `/`.

Naturally, the mechanism has basic escape countermeasures, such as disallowing `chdir("../")` to move above `/`. However, if a process can become `root` while inside the jail, there are many ways to escape; for example, it can use `mknod(2)` to create device entry points, such as `/dev/kmem` (kernel memory).

Presently, `chroot` is privileged (only `root` can do it). Naïvely, this might seem unnecessary – after all, `chroot` simply reduces the set of files a process can access, right? Wrong! In addition to limiting *which* files a process can access, it also changes the *names*

Figure 1: Example of a User Hierarchy.



```

# /etc/sandbox_users file

initftp  ->  ftp
scott    ->  sandboxscott
  
```

of files. E.g., consider that after `chroot("/tmp")`, the name `/etc/shadow` actually refers to the file `/tmp/etc/shadow`. The former is a trusted name with trusted contents; the latter is not. If ordinary users were allowed to use `chroot` arbitrarily, they could subvert programs that trust filenames, such as `su(1)` (“become superuser”).³

Furthermore, it turns out that `chroot` itself could be used to escape the jail if it were unprivileged. On most Unix systems, the code in Figure 2 can be used to escape if run as `root`.

Figure 2: Using `chroot` to escape a `chroot` jail.

```

int fd, i;
fd = open("/", O_RDONLY);
mkdir("tmpdir");
chroot("tmpdir");
fchdir(fd);
for (i=0; i<100; i++) {
    chdir(".");
}
chroot(".");
execl("/bin/sh", "/bin/sh", NULL);
/* unjailed shell */
  
```

Since `chroot` jails are a useful confinement mechanism for daemons, we choose to allow non-`root` users to call `chroot`. However, to ensure that this is safe, we place two restrictions on what a process can do once it’s inside the jail.

First, under the new system, jailed processes can not execute `setuid` binaries. This means processes can never elevate their privileges once inside the jail, except by using the new authentication interface described above. This defeats the renaming attack, and generally provides greater assurance that jailed

processes cannot escape.

We choose to allow jailed processes to access the authentication mechanism because it is safe to do so. `asp`, which checks the password, is not vulnerable to the `chroot` renaming attack the way `su` is, because it is not a child process of any (untrusted) user process. Since our interface cannot be abused, there is no reason to deny access to a jailed process. Moreover, an administrator can choose to deny access to the authentication interface simply by not including its virtual device file in the jail.

Second, jailed processes cannot call `chroot` recursively. This defeats the `chroot` escape. It is somewhat unfortunate that recursive `chroots` cannot be allowed, since there is a certain conceptual elegance to it, but fixing this would require substantial kernel changes.

There is a practical problem with disallowing recursive `chroot`, namely that it breaks certain uses of `chroot` in software testing and root-disk creation. These uses are different from ones discussed already, in that they use `chroot` *not* for its sandboxing effect, but *purely* for its renaming effect (there is no trust boundary). The problem arises because inside these `chroot` environments, the software being tested needs to use `chroot` (recursively). We address this problem by creating a new system call, `rename_chroot`, which has the same semantics as traditional `chroot` (specifically: it can only be called by `root`, and recursive calls are allowed). This is a fail-safe approach: software that expects sandboxing semantics will get it automatically, while software that wants renaming semantics and the ability to make recursive calls must be modified to use the new name. Note that it *is* possible to escape from `rename_chroot` by using (our) `chroot`; do not use it for sandboxing!

³For example, an attacker could install a known `root` password in `/tmp/etc/shadow`, which `su` would then trust.

2.3 Privileged operations

Certain daemon tasks simply require privilege. For these tasks, we create a mechanism through which we can designate certain processes as possessing the privilege necessary to perform that task.

We chose to use the existing Unix “group” mechanism as a way to name privileges, and to identify processes which have those privileges. Groups have many attractive qualities: groups are ubiquitous; they are easy (for the administrator) to create; executables can be marked to grant group privilege (set-group-id); and a process can be a member of several groups at once (so-called “supplementary” groups).

2.3.1 Binding a privileged port

To listen for incoming connections, a network daemon must use `bind(2)` to bind a socket to a well-known port, so that clients can know where to make contact. BSD introduced the notion of *privileged* ports, which could only be bound by processes running as `root`: a privileged port is any with numerical value between 1 and 1023, inclusive. The intent was to provide a form of authentication: if you connect to a privileged port, only an authorized process can receive that connection.

To allow a non-root daemon to bind such ports, we define a privilege (a group) with the right to bind those ports. We call this group `cbpp`: “Can bind privileged ports”. When the daemon is spawned by `root`, it will be granted membership in this group, and hence the `bind` privilege.

To implement the semantics of the `cbpp` group, we modified the kernel to recognize membership in this special group as constituting authorization to bind any port. Rather than defining a particular group id number as special, we let the administrator define the id of the `cbpp` group at run time, via the Linux `sysctl(2)` interface (which is also accessible in `/proc/sys`).

2.3.2 Login audit databases

Most Unix systems have databases to track user logins. `/var/run/utmp`⁴ has one record for each user currently logged in, and `/var/log/wtmp` is a log of every login and logout.

Naturally, these files need to be protected from modification by arbitrary users. Thus, they are typically installed such that only `root` can modify them. However, this means that any program wishing to add entries to the databases must run as `root`. An extreme example is `xterm(1)`, which modifies `utmp`. `xterm` is therefore usually `setuid-root`⁵, which should make anyone familiar with its source code cringe.

The solution is again to grant a privilege at a finer grain. We create two new user groups, `utmp` and `wtmp`, and mark the login databases as writable by the respective groups. Then, network services (and local programs such as `xterm`) that need to access the databases are run with membership in whichever group they need. We separated `utmp` and `wtmp` because most programs that access either do not access both, and in keeping with the Principle of Least Privilege, we wanted to grant only what was needed.

Since the `wtmp` login audit database is rather sensitive, an attractive alternative approach would be to define a trusted subsystem for interaction with `wtmp`. Clients of this service could add records for login and logout, but not remove anything (due to its internal format, it’s usually not possible to simply make `wtmp` append-only). However, this would expand the need for making changes to legacy source, so we did not do this.

2.4 Generalizing the group mechanism

There are two obstacles to the use of groups as a general privilege mechanism. First, it is not possible to mark an executable as set-group-id for multiple groups. Second, there is no way for an unprivileged user to manipulate group membership (e.g. drop a supplementary group).

⁴File names in this paper are taken from a specific Linux distribution; they vary from `system` to `system`.

⁵An executable marked “setuid” runs with the credentials of the executable file’s owner, rather than those of the user who runs the program. Thus, a `setuid-root` program will run with full `root` privileges.

To illustrate the first problem, consider a terminal emulator that wants to update both `utmp` and `wtmp`. As a first attempt, we could create a user, `uwtmp`, who is a member of both groups. Then the executable can be marked `set-user-id` to `uwtmp`.

Unfortunately, this only yields the effective *user* id; the group membership is ignored by the kernel. So, the first thing the executable must do is call `asp`, to authenticate itself as `uwtmp`. This succeeds, as a degenerate case of the `sandbox_users` mechanism. But in addition to simply saying “yes”, `asp` makes the process a member of all of the supplementary groups that `uwtmp` is. The effective user id `uwtmp` can (and should) then be dropped.

The terminal emulator now has the permissions necessary to update `utmp` and `wtmp`. But it usually does not want to give these permissions to the program that will run “inside” the terminal (typically a shell)! So, within the forked child process, the extra group memberships are dropped, and then it is safe to `exec` (say) the shell.

However, one of the nice features of `set-group-id` is that an administrator can change the privilege status of an executable without modifying its source. We can achieve something similar for `set-multiple-group-id` by using a wrapper program. The wrapper makes the call to `asp` to acquire the group privileges, and drops the effective user id. It then `execs` the real (unprivileged) binary using its enhanced credentials.

A subtle point to using `asp` is that we must decide whether the new group memberships *replace*, or simply *augment*, whatever supplementary group memberships the process may already have. For the purpose outlined above, we need augmentation. But in most authentication scenarios (such as what happens inside `ftpd` or `sshd`), the daemon expects replacement. We reconcile this with an extra flag passed to `asp`, indicating which semantics are desired.

As indicated above, for all of this to work, it is essential that a process be able to manipulate group membership. Most Unix systems implement the `setgroups(2)` system call to change the supplementary groups, but it usually can only be called by `root`. Further, some systems support `setresgid(2)` to move group ids between real, effective, and saved, but do not let the process save any of the supplementary groups in this way.

To allow these rudimentary privilege manipulations, we extend the `setgroups` and `setresgid` calls to permit arbitrary permutation of the group memberships among real, effective, saved, and supplementary groups. We also let `setgroups` be used to drop privileges (by requesting a subset of the current groups).

With these extensions in place, the Unix group mechanism comes closer to a reasonable privilege system. Privileges can be passed or not passed from parent to child, at the parent’s discretion, and can be granted to binaries (albeit with a somewhat convoluted mechanism for granting multiple privileges). Further, unlike Posix privilege bits (see Section 5.1), it’s easy to add new groups for new privileges. It would be reasonable to have one group for each of the Posix privilege bits, to emulate that system; but one can go further, dividing `root`’s privileges differently (with kernel changes), and by using groups to protect files in the usual way.

3 Authorization spectrum

In this section we attempt to put our work into a more abstract framework, to better understand what we have done and what more could be done.

3.1 Levels of required authorization

The techniques presented in this paper can be seen as the application of one general principle, namely, to reduce the level of authorization needed to use a primitive (a primitive is a system call, basically). Each primitive lies at one of four possible positions on the spectrum of required authorization: `root`-only, privileged, mediated access, or benign. Table 1 lays the primitives discussed above along various points of this spectrum.

A primitive lying at the leftmost of these positions requires `root` privileges: only processes running as `root` can use it. Existing Unix systems tend to require this level of authorization for every security-related primitive.

If a primitive allows authorization to be granted as a *privilege*, it is possible to do so on a per-process basis. The Unix `games` group is an example of this

Table 1: Tasks, and the authorization spectrum.

Daemon task	← Req'd authorization spectrum →			
	root	Privileged	Mediated	Benign
Authenticate (check user/pass)				*
Become authenticated user		(1)	*	
Become sandbox user			*	
Bind privileged port		*		
Update login databases		*		(2)
Create a <code>chroot</code> jail				*
Start network daemons (<code>inetd</code>)	*			

(1) Granting the ability to `setuid` to arbitrary users would amount to granting full root privileges.

(2) It would be possible to implement a benign primitive for this purpose, but we did not do so.

in existing systems: any game that has a high-score file is marked set-group-id, and owned by the group `games`. The game can then write to the high-score file, while ordinary users cannot.

If a primitive requires *mediated* authorization, then in order to use it, a process must provide proof that it is authorized to do the operation at the *same time* it requests the operation. In other words, it does the two atomically. The atomicity is important; among other things, it avoids time-of-check to time-of-use errors. An example of a mediated primitive in existing systems is the `read(2)` system call, in which the process supplies a file descriptor as proof of authorization. In capability systems, this is the primary authorization mechanism.

A *benign* primitive does operations that don't need to be restricted because they can't be used to violate the security policy. A trivial example is `getpid(2)` which retrieves the current process id.

Generally, *adding* requirements (moving the dot to the left) is always possible, while remaining consistent with the system's security policies⁶, though it reduces the availability of the primitive. But, since it increases the burden of proof on a process wishing to use the primitive, if a process can meet that burden of proof, so can an attacker who gains control of that process.

Reducing requirements (moving the dot to the right) decreases the need for a process to be special, thereby reducing the impact of compromise. However, it isn't always possible; for example, if we allow any process to bind a privileged port, those ports are no longer "privileged".

⁶We do not here specify what those policies are; but generally, we assume the usual provisions for privacy, integrity, isolation where appropriate, etc.

Most Unix systems have all of the security-relevant primitives at the left edge (root-only) of the authorization spectrum. We therefore observe that the objective of the work presented here is to relax the requirements for each system call, without violating the security policy.

3.2 Techniques to make primitives more available

The simplest way to make a primitive available to non-root processes is to define a privilege for that primitive. This is effective when the primitive in question can't be readily leveraged to gain further access. By granting only limited, self-contained privileges to daemons, the impact of a compromise is reduced.

However, granting access to privilege has its limits. First, it's not a complete solution: depending on operating system details, it may be possible to leverage such a privilege to gain unauthorized access (consider, for example, running a trojan FTP daemon and collecting passwords). Second, for some primitives (such as `setuid`), unrestricted access to the primitive is tantamount to full root access, so it doesn't make any sense to grant such access as a privilege.

Instead of merely controlling access to dangerous primitives, one way to improve the situation is to combine multiple existing primitives into a single atomic call, which can be made safe for ordinary users to use. This is the essence of the new authentication interface described above: a process simultaneously checks a submitted password *and*, upon success, gains the authenticated user's credentials.

Another approach to providing primitives for non-root processes is to redefine the semantics of a primitive such that it cannot be abused. Existing implementations of `chroot` have the property that they allow trusted files to be renamed. While we do not change this behavior, we prevent it from being exploited, by disallowing the execution of setuid binaries (in general, we prevent the gaining of privilege).

Finally, in some cases we can create completely new operations to allow daemons to accomplish their tasks. The `sandbox_users` mechanism, above, is an example. In general, it is safe to allow a process to sandbox itself in an unprivileged user. However, the only mechanism traditionally available was `setuid` itself, which of course is not safe to be used arbitrarily. Our solution relies on understanding what the end objective is, and to provide a primitive that accomplishes exactly that, and no more.

4 Security analysis

In this section we attempt to justify the proposed changes in light of the effort of making them.

In this paper, we assume our primary threat is an anonymous attacker somewhere on the network, wishing to gain root access to our machine. We further assume it is likely that the network daemons on this machine contain exploitable vulnerabilities that allow an attacker to gain control of the daemon process.

Given that we do not consider the network daemon software entirely trustworthy, our approach is to fall back on local host security. We assume (and hope!) that our system's host security is such that non-root access has less potential to be destructive than root access.

At the very least, by taking root privileges away from network daemons, we raise the bar for a successful remote root exploit. Existing attack scripts will fail if they assume they have root once the daemon is subverted. Future attacks will require a two-stage approach, where the second stage involves a search for a local exploit. Such attacks, especially if the second stage is not automated, may be significantly easier to detect.

In some cases our techniques reduce the possibility

of finding a local exploit. For example, if a process calls `chroot("/")`, this has the effect of preventing all future setuid `execs`. This is useful if a process has no need to use setuid executables; an example might be a web server. If it is subverted by an attacker, the inability to use setuid executables will make it much more difficult to find a local exploit (though not impossible since, e.g., the kernel could have vulnerabilities).

Further, by introducing some delay (again, especially in the case of an interactive attacker) between the initial subversion and the final root compromise, we have introduced the possibility of an early-warning system. A properly designed auditing service could record the initial attack and alert the system administrator. For example, in current systems, an effective attack technique is to kill `syslogd(8)` (which sends system log reports to the disk, and optionally to dedicated logging hosts) as soon as root is obtained. By doing so, there is a good chance `syslogd` will die before the log reports make it to durable storage. With our changes, the log of the initial attack is more likely to survive.

One possible objection to our scheme is that some attackers don't need root to make the attack worth their while. These attackers may simply want to launder network connections, run IRC relays, etc. The key observation here, however, is that a non-root compromise is much easier to detect (since auditing is intact), much easier to *clean up* after, less able to compromise user data, etc.

Finally, we note that taking root privileges away from daemons means that local host security can play a larger role. However, many systems' host security is very poor, and we can ask whether this renders our approach ineffective. First, host security can be improved independently; prior to this work, improving local host security did not improve network application security, so the former received relatively less attention. But, once daemons are less privileged, enhancements to host security effectively improve network security as well. Second, our techniques can be applied to improve host security directly, by reducing the need for setuid-root programs. A good example is `xlock(1)`, which locks a workstation, displaying some random graphics, until a password is provided (which requires root privileges to check, if shadow passwords are used). With our scheme in place, this large and complex package need not be trusted.

5 Related work

5.1 Granting of privileges

The Debian Linux distribution implemented a `utmp` group some time ago.⁷ One difference, however, is that Debian's `utmp` can write to `wtmp` as well, whereas we separate these privileges (in accordance with the Principle of Least Privilege).

The Linux kernel has an implementation of Posix 1003.1e “capabilities” [4]⁸. Posix capabilities are per-process privilege bits, where each bit corresponds to some subset of `root`'s traditional privileges. This mechanism is intended as a way to allow services to run with only the minimum required privileges. We attempted to use Linux capabilities to implement some of our system changes. The first problem is expressiveness: the existing bits do not let us *take away* privileges from ordinary users, but that is precisely what is required to implement the safe `chroot`. This can be addressed simply by adding more bits and giving them appropriate semantics. Worse, the Linux implementation unfortunately ties capabilities and user ids in a way that renders both systems hopelessly tangled. We are currently attempting to convince the Linux developers to repair this.

VMS and Windows NT define a variety of privilege bits, similar to the Posix capability bits. However, like Posix capabilities, they happen not to have the privilege bits for some of the services we need to restrict or grant. An NT or VMS implementation would therefore require kernel changes, too.

5.2 OS primitives

The BSD `jail(2)` mechanism is designed to be a more sophisticated sandbox than `chroot`. In particular, in addition to limiting file access, it limits network access. However, like `chroot`, `jail` can only be used by `root`. We do not see any reason why similar techniques could not be applied to `jail`, but we have not attempted to do so.

One of the classic difficulties with writing terminal emulators under Unix is the need to allocate a pseu-

doterminal⁹ and then use `chown(2)` to prevent other local users from eavesdropping on it. Older Unix systems required that a process have `root` privileges to `chown` the pseudoterminal, so terminal emulators would have to be `setuid-root`. Unix98 [5] standardizes the (now common) solution, which is to define a new system call, `grantpt(2)`¹⁰, which is essentially `chown` specialized to this situation. This is another example of changing the system primitives to allow non-`root` processes to securely manipulate security state.

NT provides a kernel system call, `LogonUser`, to do authentication. We feel it is better to perform authentication in user-space, rather than in the kernel, for reasons outlined in section 2.1; however, this is a detail.

An alternative approach to designing primitives is to attempt to limit what `root`, itself, can do [9]. The problem with this is that it is very difficult to confine `root` in a Unix-like system. Throughout its history, Unix has regarded `root` as all-powerful; reversing this fundamental design decision is nearly as hard as rewriting the system from scratch, since it requires understanding every assumption made about `root` and its privileges. In general, it better to enumerate the *allowed* actions, rather than the *disallowed* actions. Hence, our approach is to make non-`root` users potentially more capable, instead of making `root` less powerful.

5.3 Sandboxing

Janus [3] is a general-purpose sandboxing system, originally designed for use with web browser “plugins”, and implemented by intercepting system calls via `ptrace`. Janus is clearly a much more general and expressive sandboxing environment than `chroot` and unprivileged users. However, few (if any) existing applications have been written with the expectation of using Janus as their sandboxing mechanism. Further, it is not clear how to use Janus to achieve some of our goals; e.g., should every `xterm` run in Janus, just so it can safely access `utmp`? We do not use a separate sandboxing system because our goal is to minimize changes to legacy

⁹A pseudoterminal is a virtual device with two sides. One side, the slave, looks like a serial line connected to a physical terminal. The other side, the master, is an interface to simulate a terminal.

¹⁰It's permitted to implement this as a library call and a `setuid-root` helper.

⁷See <http://lwn.net/1999/0610/a/debian-policy.html>

⁸Posix 1003.1e is not a standard; it has been withdrawn.

code and performance degradation, while preserving whatever sandboxing the application currently employs.

Qmail [1] is a Mail Transport Agent developed by Dan Bernstein, intended to replace Sendmail. Fundamental to its design is the use of six unprivileged, mutually untrusting Unix users, plus a small piece that runs as `root`. This approach is based on the idea that code running as one (unprivileged) user cannot interfere with code running as another user. Our work similarly pursues security through the use of unprivileged users wherever possible.

5.4 OS “Hardening”

Industry marketing literature has recently been using the term “operating system hardening”. This term refers to the process of modifying a system’s configuration files, typically to disable services that are not needed and therefore present an unnecessary risk of attack. Our efforts are orthogonal: if a service is not needed, it should be disabled; but if it is, then our techniques apply.

6 Implementation status

All of the changes described above have been implemented, including kernel modifications, under Linux 2.2.17. We modified `BSD-ftp-0.3.2-5`; all FTP functions work as expected, including anonymous logins. Similarly, we modified `sshd-1.2.30` to support password authentication. The SSH protocol also allows for authentication by RSA private key; this requires expanding our `ioctl` interface to support interactive challenge-response, which we plan to do.

We added about 20 lines in the kernel, and wrote a 500 line virtual device driver. In user-space, we added or changed 15 lines of daemon code and wrote the 300 line `asp`. We found the implementation surprisingly easy. The only significant difficulties arose from our failed attempt to use the Linux implementation of Posix capabilities.

Overall, the kinds of changes we made all tend to be small, easy to understand, and (conceptually) portable. We see no reason why someone couldn’t

implement analogous changes for other Unix-like operating systems besides Linux.

Patches to daemons and kernel, and the `asp` daemon, are publicly available on the authors’ website: <http://www.cs.berkeley.edu/~smcpeak/root-daemon/>.

7 Conclusion

We have demonstrated that important network daemons can be made to accomplish their tasks, on a legacy Unix system, without ever running as `root`. We identify a spectrum of required authorization for security primitives, which can be used to evaluate the access policy tradeoffs, and point the way towards better primitive design. Finally, we argue that by reducing the privileges that must be granted to a network daemon, we limit the damage an attacker can do if such a daemon is compromised by an attack, thereby significantly improving overall system security.

8 Acknowledgments

We would like to thank David Wagner for many helpful discussions, advice, and support.

References

- [1] Bernstein, D. qmail: a replacement for sendmail, <http://www.qmail.org/>
- [2] CERT, “Advisory CA-1999-13 Multiple Vulnerabilities in WU-FTPD,” <http://www.cert.org/advisories/CA-1999-13.html>
- [3] Goldberg, I., Wagner, D., Thomas, R., Brewer, E. A secure environment for untrusted helper applications. In Proceedings of the 6th Usenix Security Symposium, San Jose, CA, July 1996.
- [4] Information technology - Portable Operating System Interface (POSIX) - Part 1: system Application Program Interface (API) - Amendment #: Protection, Audit, and Control Interfaces [C language]. Inst. Electr. & Electron. Eng., New York, NY, USA, October 1997

- [5] The Open Group, “The Single UNIX Specification, Version 2 and UNIX 98,” <http://www.UNIX-systems.org/unix98.html>
- [6] Saltzer, J., Schroeder, M. The Protection of Information in Computer Systems. Proceedings of the IEEE, 63(9):1278–1308, September 1975.
- [7] Samar, V., “Unified login with pluggable authentication modules (PAM).” *Proceedings of 3rd ACM Conference on Computer and Communications Security*, New Delhi, India, March 1996
- [8] Simmons, S., “Life without root (system administration),” *Proceedings of the Fourth Large Installation System Administrator’s Conference*, Colorado Springs, CO, USA, Oct. 17-19, 1990.
- [9] Walker, K.M., Sterne, D.F., Badger, L., Oostendorp, K.A., Petkac, M.J., Sherman, D.L. “Confining root programs with Domain and Type Enforcement (DTE).” *6th USENIX UNIX Security Symposium*, San Jose, CA, July 1996, pp. 21-36