userv - an alternative to setuid

`http://www.chiark.greenend.org.uk/~ian/userv/`

Ian Jackson
Debian Project
ian@chiark.greenend.org.uk

27th June 1998
UKUUG Linux Conference
Manchester

Talk overview

Problem description

What is a security boundary.

Security-boundary crossing programs. Examples.
Current techniques and problems (setuid, IPC, etc.)

Proposed solution - userv ('you-serve')

Safe program call across privilege boundary.

Example applications

man program: safe, correct, with caching.
cron subsystem with least privilege.
lpr: with delayed copy, safe from hostile documents.

Implications and conclusions

Notes for designing and programming with userv.
When not to use userv.
Current status, where to get program and more info.

What is a security boundary

Question of trust: security boundary is the line between things that are trusted (and are expected to be able to do bad things) and things that are not (and are expected not to be able to even if they try).

E.g.: ‘login’ forms part of the external security boundary of a Linux system. It is exposed to untrusted external people, and is itself trusted to decide whether to let them in.

Principle of least privilege: trusted parts should be as small as possible. Security boundaries should be ‘strong’ and not ‘messy’.

Security-boundary crossing programs

Subsystems which deal with more than one security domain.

Either user calls service provided by system which requires privilege (of some kind), or system calls user back.

Examples

cron (and at)
caching man program
WWW CGI scripts
mail sending and delivery
lpr
mount
finger (.plan or scripts)

Current techniques — setuid

Very hard to do right.

Many inherited properties of processes (fds, signal handlers, controlling tty, LD_*, PATH and other environment variables, ulimit, umask, child processes, …

Hard or impossible to use scripting languages (which would otherwise make it easier to get things right).

Caller must trust called program absolutely.

Leads to many security holes (buffer overruns, argument (un)parsing problems, inherited-property attacks).

Current techniques — IPC or network

Often lots of work to implement.

Needs one permanently-running daemon for every subsystem.

Needs crypto if done over the network.

How to identify ‘client’ end of IPC ?

Cumbersome to pass data (marshalling, etc.).

Current techniques — Facility runs as root

No problem running programs ‘as user’ when required.

But, hard to correctly drop privilege (inherited fds, etc).

Successful attack on subsystem breaks system security completely.

Often combined with setuid to do user->system calls.

Current techniques — Problem ignored

I.e., no security boundary implemented (eg, some man programs, finger):

Easy, but …

Globally-writeable directories (missing **`O_EXCL`** attacks, etc).

Globally-readable files and directories (eg, home a+x for finger).

Problems with devices, pipes, etc.

Cannot trust results from many of these subsystems (eg man).

Only even possible for some subsystems.

What to do about this problem ?

Considering requirements of the various examples: cron, man, CGI, mail, lpr, finger, mount, etc.

Need controlled way for program to gain or lose privilege.

Needs to work if neither account completely trusts the other.

Must be flexible and general enough to allow secure implementation of all these (and others not yet thought of).

Should have strong security boundary, and make it easy (or easier) to write correct, secure programs.

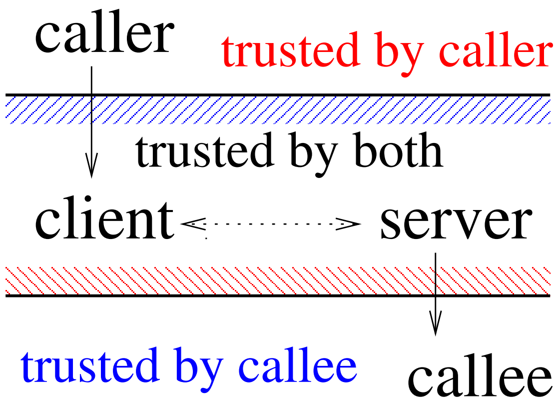
We want to allow **program call for programs with different privileges.** Program call is a natural division in UN*X, well-understood. Heavyweight, so strong separation but slow. No language restrictions.

Implementation

In order to get clean environment for called program, must fork it from a trusted daemon with known-good environment.

Need to tell daemon what program to run, so have client program. Client program must be setuid so daemon can rely on its info about caller, and to safeguard filesystem area used for IPC.

Result looks like a program call, is actually IPC.



Design details

AF_UNIX sockets and pipes in a well-known (system-wide) but access-protected directory.

All information transfer mediated by client/server, including cross-boundary I/O. Therefore, pipes.

Services have abstract names — caller does not directly specify specify program to run. Called user configures which programs to run to provide which services.

Called programs often specially written (but can often be simple scripts).

Server reads configuration files (provided by called user and system) to determine what to do for each request.

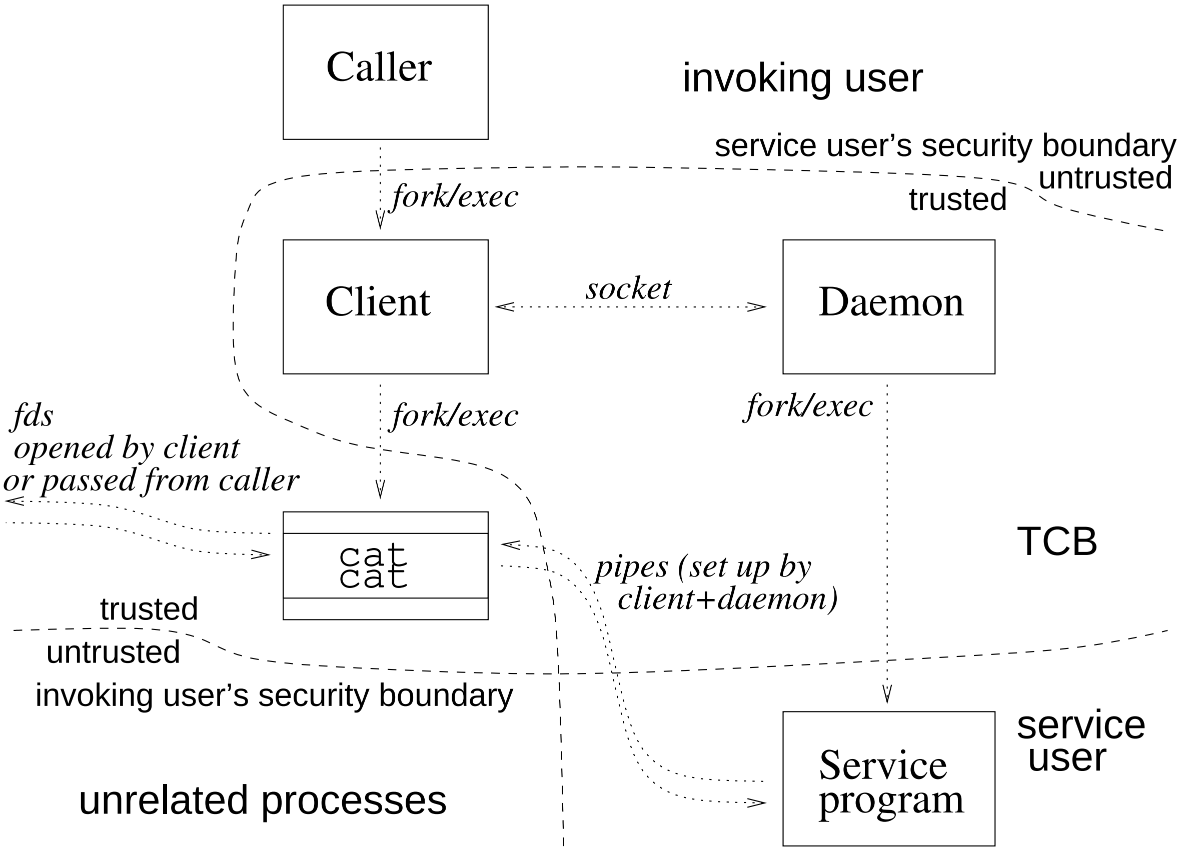
Who specifies what ?

Caller specifies
- service name and arguments
- what stdin, stdout, stderr, etc. are
- timeout, exit status handling

Service configuration specifies
- whether to provide service
- meaning of service name(s)
- which actual program to run
- which fd’s are expected to be input, which output
- send SIGHUP if client times out, is killed, etc ?
- pass arguments through ?

Other information passed through
- Calling user (and groups)
- Caller’s cwd (may be hidden by caller)
- -Dxxx=yyy arguments to client program

Implementation diagram



Service-side configuration

Server reads system and user configuration files.

System configuration files for system services (cron, finger, etc.) — including when system-installed software uses ‘service’ provided by user account.

User configuration files for programs they provide, or to override system configuration.

Configuration files must explicitly allow each service.

Configuration language is reasonably simple. Not Turing-powerful, but hopefully sufficiently complete for most purposes.

Information available in configuration language also available to service program (ie, called program), so can have wrapper with more fancy decisionmaking if required.

Configuration directives

```
reject
execute <program> [<argument> …]
execute-from-directory <directory> [<argument> …]
execute-from-path
execute-builtin <builtin-service> [<argument> …]
```

```
cd <directory>
eof
quit
error <message>
```

```
[no-]set-environment
[no-]suppress-args
[no-]disconnect-hup
reset
```

```
require-fd <fd-range> read|write
allow-fd <fd-range> [read|write]
null-fd <fd-range> [read|write]
reject-fd <fd-range>
ignore-fd <fd-range>
```

```
include <file>
include-ifexist <file>
include-lookup <param> <directory>
include-directory <directory>

user-rcfile <file>

errors-push
srorre
catch-quit
hctac

errors-to-stderr
errors-to-file <file>
errors-to-syslog [<facility> [<level>]]
```

```
if glob <param> <pattern> …
if range <param> <min> <max>
if grep <param> <file>
if ! <condition>
if ( <condition>
   & <condition>
   )
if ( <condition>
   | <condition>
   )
elif <condition>
else
fi
```

manpage service — requirements

Caching manpage program.

Cached manpages must be reliable.

User-supplied manpages must be formatted by their ‘owner’ (in case of dangerous troff, etc.).

Breaking ‘man’ security should not allow anything but corrupting manpages.

manpage service — design

Central cache managed by ‘man’ user.

‘man’ program invokes userv to get cached copy or format page.

userv service returns from cache, formats system manpage, or invokes userv again to format user-provided page

usman.conf specifies which manpages we export (ie, are willing to format).

usman.conf and/or MANPATH specify which manpages we wish to import (ie, want to read)

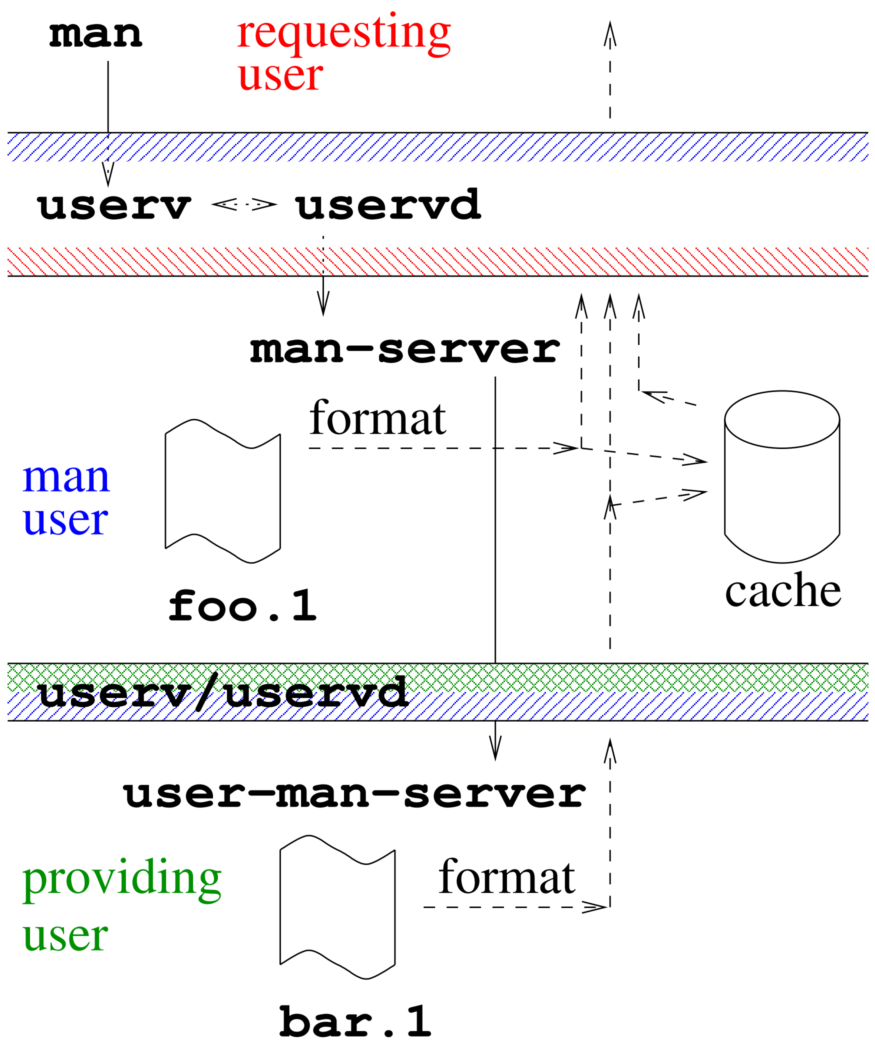
manpage service — details

system-wide userv configuration lets man-server service be provided by user ‘man’.

system-wide userv configuration provides user-man-server service to ‘man’ from all users.

Implementation by Richard Kettlewell *richard@greenend.org.uk* currently in development.

manpage service — diagram



manpage service — userv configuration

```
if ( glob service man-server
   & glob service-user man
   )
        reset
        disconnect-hup
        require-fd 1 write
        allow-fd 2 write
        reject-fd 3-
        no-suppress-args
        execute /usr/local/lib/usman/man-server
fi
if ( glob service user-man-server
   & glob calling-user man
   & grep service-user-shell /etc/shells
   )
        reset
        disconnect-hup
        require-fd 1 write
        allow-fd 2 write
        reject-fd 3-
        no-suppress-args
        execute /usr/local/lib/usman/user-man-server
fi
```

cron service

cron daemon should do scheduling, callbacks, but as little as possible.

cron daemon should not have equivalent power to root — just ability to cause jobs to happen at wrong times.

User-side programs should interpret crontab.

cron daemon maintains list of times when users wish to be called back.

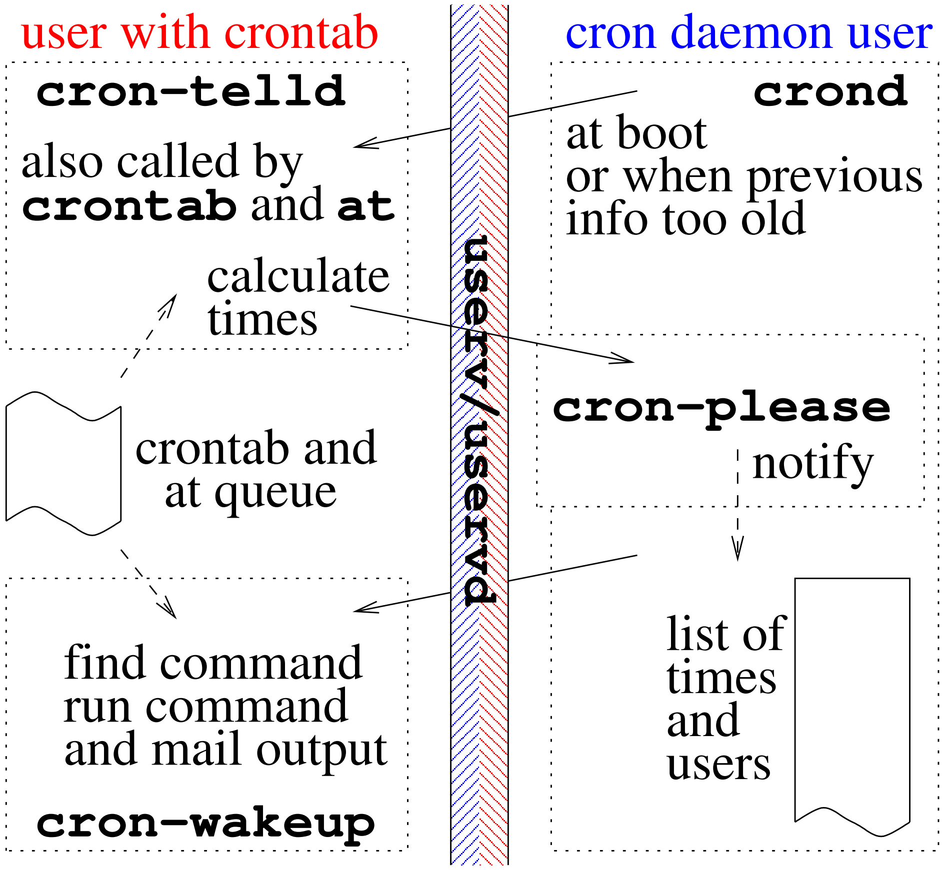
Callback just specifies ‘this is your 10pm wakeup call’.

User-side crontab program parses crontab, notifies cron daemon of desired wakeups via userv.

User-provided userv service runs command(s) from crontab.

Implementation currently in detailed design / early implementation.

cron service — diagram



cron service — userv configuration

```
if ( glob service cron/user/*
   & glob calling-user cron
   & grep service-user-shell /etc/shells
   )
        reset
        no-disconnect-hup
        null-fd 0 read
        null-fd 1-2 write
        no-suppress-args
        execute-from-directory /usr/local/lib/uscron/us/
        # us contains telld and wakeup
fi
if ( glob service cron/daemon/please
   & glob service-user cron
   & grep calling-user-shell /etc/shells
   )
        reset
        no-disconnect-hup
        require-fd 0 read
        null-fd 1-2 write
        reject-fd 3-
        suppress-args
        execute /usr/local/lib/uscron/please
fi
```

lpr service

lpr with deferred copy and safe format conversion.

Daemon needs to read file to be printed at time of its choice.

Daemon may want to format/convert file in user's account.

Keep list of files and their types in user's filespace.

lpr program records file and type, notifies daemon via userv.

When daemon wants to convert or print, calls user back via userv.

Callback service checks list of files and types to make sure file was intended for printing and that type is right.

If deferred copy is not required, lpr program makes copy in user's filespace, or daemon could make copy.

Programming with userv

userv cannot protect against bad programmers (though it can make life easier).

Must still get argument parsing/unparsing right. Make sure to use `--` end-of-options argument, proper quoting (in sh), not `eval` or ` `` ` (sh, Perl, etc.), etc.

If writing in C, can have all the usual buffer overrun problems.

Correct placement of security boundary essential. Minimise data flow and trust across the boundary. Minimise processing of data on wrong side of boundary.

Naive design can end up with userv-using facility having access equivalent to root.

When not to use userv ?

userv prevents access to caller's tty and fd's. So is no good for e.g. rsh, utmp/pty helper programs, etc.

If intent is to give away full access to called account, userv is overly restrictive. (e.g., not good for allow-everything sudo/really.)

Heavyweight and slow: do not use for very common operations which must run quickly.

Not very good against denial-of-service from local user. Resource accounting problems.

Implementation and status

userv itself is currently in beta. GPL.

Mainly C; also uses m4, flex, and autoconf. Intended for any sensible UN*X, currently only ported to Linux.

userv-based caching man program in development by Richard Kettlewell *richard@greenend.org.uk*.

userv WWW CGI wrappers in internal use; may be released shortly.

userv-based cron in early stage of development.

Addresses

`http://www.chiark.greenend.org.uk/~ian/userv/`

Mailing lists via *majordomo@chiark.greenend.org.uk*

```
subscribe userv-announce
subscribe userv-discuss
```

Contact author at *userv-maint@chiark.greenend.org.uk*

Summary

Security boundary crossing programs:

- Hard to do right with existing mechanisms (setuid, IPC). setuid used often, but very hard, and cannot choose language for task.
- Therefore often done wrong (holes, absence of security boundary, facility runs as root).

userv can help solve this:

- Secure program call across security boundary.
- Insulates called program.
- OK even in the case of mutual distrust.
- userv in late beta now; userv-using applications in development. Help write more !