NAME

sudo_plugin - Sudo Plugin API

DESCRIPTION

Starting with version 1.8, **sudo** supports a plugin API for policy and session logging. Plugins may be compiled as dynamic shared objects (the default on systems that support them) or compiled statically into the **sudo** binary itself. By default, the **sudoers** policy plugin and an associated I/O logging plugin are used. Via the plugin API, **sudo** can be configured to use alternate policy and/or I/O logging plugins provided by third parties. The plugins to be used are specified in the sudo.conf(5) file.

The API is versioned with a major and minor number. The minor version number is incremented when additions are made. The major number is incremented when incompatible changes are made. A plugin should be check the version passed to it and make sure that the major version matches.

The plugin API is defined by the sudo_plugin.h header file.

Policy plugin API

A policy plugin must declare and populate a policy_plugin struct in the global scope. This structure contains pointers to the functions that implement the **sudo** policy checks. The name of the symbol should be specified in sudo.conf(5) along with a path to the plugin so that **sudo** can load it.

```
struct policy_plugin {
#define SUDO_POLICY_PLUGIN     1
    unsigned int type; /* always SUDO_POLICY_PLUGIN */
    unsigned int version; /* always SUDO_API_VERSION */
    int (*open)(unsigned int version, sudo_conv_t conversation,
            sudo_printf_t plugin_printf, char * const settings[],
            char * const user_info[], char * const user_env[],
            char * const plugin_options[]);
    void (*close)(int exit_status, int error);
    int (*show_version)(int verbose);
    int (*check_policy)(int argc, char * const argv[],
                    char *env_add[], char **command_info[],
                    char **argv_out[], char **user_env_out[]);
    int (*list)(int argc, char * const argv[], int verbose,
            const char *list_user);
    int (*validate)(void);
    void (*invalidate)(int remove);
    int (*init_session)(struct passwd *pwd, char **user_env[]);
    void (*register_hooks)(int version,
```

```
    int (*register_hook)(struct sudo_hook *hook));
  void (*deregister_hooks)(int version,
    int (*deregister_hook)(struct sudo_hook *hook));
};
```

The policy_plugin struct has the following fields:

type The type field should always be set to SUDO_POLICY_PLUGIN.

version

The version field should be set to SUDO_API_VERSION.

This allows **sudo** to determine the API version the plugin was built against.

open

```
int (*open)(unsigned int version, sudo_conv_t conversation,
    sudo_printf_t plugin_printf, char * const settings[],
    char * const user_info[], char * const user_env[],
    char * const plugin_options[]);
```

Returns 1 on success, 0 on failure, -1 if a general error occurred, or -2 if there was a usage error. In the latter case, **sudo** will print a usage message before it exits. If an error occurs, the plugin may optionally call the **conversation**() or **plugin_printf**() function with SUDO_CONF_ERROR_MSG to present additional error information to the user.

The function arguments are as follows:

version

The version passed in by **sudo** allows the plugin to determine the major and minor version number of the plugin API supported by **sudo**.

conversation

A pointer to the **conversation**() function that can be used by the plugin to interact with the user (see below). Returns 0 on success and -1 on failure.

plugin_printf

A pointer to a **printf**()-style function that may be used to display informational or error messages (see below). Returns the number of characters printed on success and -1 on failure.

settings

A vector of user-supplied **sudo** settings in the form of ‘‘name=value’’ strings. The vector is terminated by a NULL pointer. These settings correspond to flags the user specified when running **sudo**. As such, they will only be present when the corresponding flag has been specified on the command line.

When parsing *settings*, the plugin should split on the **first** equal sign (‘=’) since the *name* field will never include one itself but the *value* might.

bsdauth_type=string

Authentication type, if specified by the **-a** flag, to use on systems where BSD authentication is supported.

closefrom=number

If specified, the user has requested via the **-C** flag that **sudo** close all files descriptors with a value of *number* or higher. The plugin may optionally pass this, or another value, back in the *command_info* list.

debug_flags=string

A debug file path name followed by a space and a comma-separated list of debug flags that correspond to the plugin’s Debug entry in sudo.conf(5), if there is one. The flags are passed to the plugin exactly as they appear in sudo.conf(5). The syntax used by **sudo** and the **sudoers** plugin is *subsystem@priority* but a plugin is free to use a different format so long as it does not include a comma (‘,’). Prior to **sudo** 1.8.12, there was no way to specify plugin-specific *debug_flags* so the value was always the same as that used by the **sudo** front end and did not include a path name, only the flags themselves. As of version 1.7 of the plugin interface, **sudo** will only pass *debug_flags* if sudo.conf(5) contains a plugin-specific Debug entry.

debug_level=number

This setting has been deprecated in favor of *debug_flags*.

ignore_ticket=bool

Set to true if the user specified the **-k** flag along with a command, indicating that the user wishes to ignore any cached authentication credentials. *implied_shell* to true. This allows **sudo** with no arguments to be used similarly to su(1). If the plugin does not to support this usage, it may return a value of -2 from the **check_policy**() function, which will cause **sudo** to print a usage message and exit.

implied_shell=bool

If the user does not specify a program on the command line, **sudo** will pass the plugin the path to the user's shell and set

login_class=string
: BSD login class to use when setting resource limits and nice value, if specified by the **-c** flag.

login_shell=bool
: Set to true if the user specified the **-i** flag, indicating that the user wishes to run a login shell.

max_groups=int
: The maximum number of groups a user may belong to. This will only be present if there is a corresponding setting in sudo.conf(5).

network_addrs=list
: A space-separated list of IP network addresses and netmasks in the form ''addr/netmask'', e.g. ''192.168.1.2/255.255.255.0''. The address and netmask pairs may be either IPv4 or IPv6, depending on what the operating system supports. If the address contains a colon (':'), it is an IPv6 address, else it is IPv4.

noninteractive=bool
: Set to true if the user specified the **-n** flag, indicating that **sudo** should operate in non-interactive mode. The plugin may reject a command run in non-interactive mode if user interaction is required.

plugin_dir=string
: The default plugin directory used by the **sudo** front end. This is the default directory set at compile time and may not correspond to the directory the running plugin was loaded from. It may be used by a plugin to locate support files.

plugin_path=string
: The path name of plugin loaded by the **sudo** front end. The path name will be a fully-qualified unless the plugin was statically compiled into **sudo**.

preserve_environment=bool
: Set to true if the user specified the **-E** flag, indicating that the user wishes to preserve the environment.

preserve_groups=bool

Set to true if the user specified the **-P** flag, indicating that the user wishes to preserve the group vector instead of setting it based on the runas user.

progname=string
: The command name that sudo was run as, typically ‘‘sudo’’ or ‘‘sudoedit’’.

prompt=string
: The prompt to use when requesting a password, if specified via the **-p** flag.

remote_host=string
: The name of the remote host to run the command on, if specified via the **-h** option. Support for running the command on a remote host is meant to be implemented via a helper program that is executed in place of the user-specified command. The **sudo** front end is only capable of executing commands on the local host. Only available starting with API version 1.4.

run_shell=bool
: Set to true if the user specified the **-s** flag, indicating that the user wishes to run a shell.

runas_group=string
: The group name or gid to run the command as, if specified via the **-g** flag.

runas_user=string
: The user name or uid to run the command as, if specified via the **-u** flag.

selinux_role=string
: SELinux role to use when executing the command, if specified by the **-r** flag.

selinux_type=string
: SELinux type to use when executing the command, if specified by the **-t** flag.

set_home=bool
: Set to true if the user specified the **-H** flag. If true, set the HOME environment variable to the target user’s home directory.

sudoedit=bool
: Set to true when the **-e** flag is is specified or if invoked as **sudoedit**. The plugin shall substitute an editor into *argv* in the **check_policy**() function or return -2 with a usage error if the plugin does not support *sudoedit*. For more information, see the

check_policy section.

Additional settings may be added in the future so the plugin should silently ignore settings that it does not recognize.

user_info
A vector of information about the user running the command in the form of ‘‘name=value’’ strings. The vector is terminated by a NULL pointer.

When parsing *user_info*, the plugin should split on the **first** equal sign (‘=’) since the *name* field will never include one itself but the *value* might.

cols=int
The number of columns the user’s terminal supports. If there is no terminal device available, a default value of 80 is used.

cwd=string
The user’s current working directory.

egid=gid_t
The effective group ID of the user invoking **sudo**.

euid=uid_t
The effective user ID of the user invoking **sudo**.

gid=gid_t
The real group ID of the user invoking **sudo**.

groups=list
The user’s supplementary group list formatted as a string of comma-separated group IDs.

host=string
The local machine’s hostname as returned by the gethostname(2) system call.

lines=int
The number of lines the user’s terminal supports. If there is no terminal device available, a default value of 24 is used.

pgid=int

The ID of the process group that the running **sudo** process is a member of. Only available starting with API version 1.2.

pid=int
: The process ID of the running **sudo** process. Only available starting with API version 1.2.

plugin_options
: Any (non-comment) strings immediately after the plugin path are passed as arguments to the plugin. These arguments are split on a white space boundary and are passed to the plugin in the form of a NULL-terminated array of strings. If no arguments were specified, *plugin_options* will be the NULL pointer.

  NOTE: the *plugin_options* parameter is only available starting with API version 1.2. A plugin **must** check the API version specified by the **sudo** front end before using *plugin_options*. Failure to do so may result in a crash.

ppid=int
: The parent process ID of the running **sudo** process. Only available starting with API version 1.2.

sid=int
: The session ID of the running **sudo** process or 0 if **sudo** is not part of a POSIX job control session. Only available starting with API version 1.2.

tcpgid=int
: The ID of the foreground process group associated with the terminal device associated with the **sudo** process or -1 if there is no terminal present. Only available starting with API version 1.2.

tty=string
: The path to the user's terminal device. If the user has no terminal device associated with the session, the value will be empty, as in ''tty=''.

uid=uid_t
: The real user ID of the user invoking **sudo**.

user=string
: The name of the user invoking **sudo**.

user_env
The user's environment in the form of a NULL-terminated vector of ''name=value'' strings.

When parsing *user_env*, the plugin should split on the **first** equal sign ('=') since the *name* field will never include one itself but the *value* might.

close
void (*close)(int exit_status, int error);

The **close**() function is called when the command being run by **sudo** finishes.

The function arguments are as follows:

exit_status
The command's exit status, as returned by the wait(2) system call. The value of exit_status is undefined if error is non-zero.

error If the command could not be executed, this is set to the value of errno set by the execve(2) system call. The plugin is responsible for displaying error information via the **conversation**() or **plugin_printf**() function. If the command was successfully executed, the value of error is 0.

If no **close**() function is defined, no I/O logging plugins are loaded, and neither the *timeout* not *use_pty* options are set in the command_info list, the **sudo** front end may execute the command directly instead of running it as a child process.

show_version
int (*show_version)(int verbose);

The **show_version**() function is called by **sudo** when the user specifies the **-V** option. The plugin may display its version information to the user via the **conversation**() or **plugin_printf**() function using SUDO_CONV_INFO_MSG. If the user requests detailed version information, the verbose flag will be set.

check_policy
int (*check_policy)(int argc, char * const argv[]
char *env_add[], char **command_info[],
char **argv_out[], char **user_env_out[]);

The **check_policy**() function is called by **sudo** to determine whether the user is allowed to run the

specified commands.

If the *sudoedit* option was enabled in the *settings* array passed to the **open**() function, the user has requested *sudoedit* mode. *sudoedit* is a mechanism for editing one or more files where an editor is run with the user's credentials instead of with elevated privileges. **sudo** achieves this by creating user-writable temporary copies of the files to be edited and then overwriting the originals with the temporary copies after editing is complete. If the plugin supports *sudoedit*, it should choose the editor to be used, potentially from a variable in the user's environment, such as EDITOR, and include it in *argv_out* (note that environment variables may include command line flags). The files to be edited should be copied from *argv* into *argv_out*, separated from the editor and its arguments by a ''--'' element. The ''--'' will be removed by **sudo** before the editor is executed. The plugin should also set *sudoedit=true* in the *command_info* list.

The **check_policy**() function returns 1 if the command is allowed, 0 if not allowed, -1 for a general error, or -2 for a usage error or if *sudoedit* was specified but is unsupported by the plugin. In the latter case, **sudo** will print a usage message before it exits. If an error occurs, the plugin may optionally call the **conversation**() or **plugin_printf**() function with SUDO_CONF_ERROR_MSG to present additional error information to the user.

The function arguments are as follows:

argc The number of elements in *argv*, not counting the final NULL pointer.

argv The argument vector describing the command the user wishes to run, in the same form as what would be passed to the execve(2) system call. The vector is terminated by a NULL pointer.

env_add
Additional environment variables specified by the user on the command line in the form of a NULL-terminated vector of ''name=value'' strings. The plugin may reject the command if one or more variables are not allowed to be set, or it may silently ignore such variables.

When parsing *env_add*, the plugin should split on the **first** equal sign ('=') since the *name* field will never include one itself but the *value* might.

command_info
Information about the command being run in the form of ''name=value'' strings. These values are used by **sudo** to set the execution environment when running a command. The plugin is responsible for creating and populating the vector, which must be terminated with a NULL pointer. The following values are recognized by **sudo**:

chroot=string
: The root directory to use when running the command.

closefrom=number
: If specified, **sudo** will close all files descriptors with a value of *number* or higher.

command=string
: Fully qualified path to the command to be executed.

cwd=string
: The current working directory to change to when executing the command.

exec_background=bool
: By default, **sudo** runs a command as the foreground process as long as **sudo** itself is running in the foreground. When *exec_background* is enabled and the command is being run in a pty (due to I/O logging or the *use_pty* setting), the command will be run as a background process. Attempts to read from the controlling terminal (or to change terminal settings) will result in the command being suspended with the SIGTTIN signal (or SIGTTOU in the case of terminal settings). If this happens when **sudo** is a foreground process, the command will be granted the controlling terminal and resumed in the foreground with no user intervention required. The advantage of initially running the command in the background is that **sudo** need not read from the terminal unless the command explicitly requests it. Otherwise, any terminal input must be passed to the command, whether it has required it or not (the kernel buffers terminals so it is not possible to tell whether the command really wants the input). This is different from historic *sudo* behavior or when the command is not being run in a pty.

  For this to work seamlessly, the operating system must support the automatic restarting of system calls. Unfortunately, not all operating systems do this by default, and even those that do may have bugs. For example, Mac OS X fails to restart the **tcgetattr**() and **tcsetattr**() system calls (this is a bug in Mac OS X). Furthermore, because this behavior depends on the command stopping with the SIGTTIN or SIGTTOU signals, programs that catch these signals and suspend themselves with a different signal (usually SIGTOP) will not be automatically foregrounded. Some versions of the linux su(1) command behave this way. Because of this, a plugin should not set *exec_background* unless it is explicitly enabled by the administrator and there should be a way to enabled or disable it on a per-command basis.

  This setting has no effect unless I/O logging is enabled or *use_pty* is enabled.

iolog_compress=bool
: Set to true if the I/O logging plugins, if any, should compress the log data. This is a hint to the I/O logging plugin which may choose to ignore it.

iolog_path=string
: Fully qualified path to the file or directory in which I/O log is to be stored. This is a hint to the I/O logging plugin which may choose to ignore it. If no I/O logging plugin is loaded, this setting has no effect.

iolog_stdin=bool
: Set to true if the I/O logging plugins, if any, should log the standard input if it is not connected to a terminal device. This is a hint to the I/O logging plugin which may choose to ignore it.

iolog_stdout=bool
: Set to true if the I/O logging plugins, if any, should log the standard output if it is not connected to a terminal device. This is a hint to the I/O logging plugin which may choose to ignore it.

iolog_stderr=bool
: Set to true if the I/O logging plugins, if any, should log the standard error if it is not connected to a terminal device. This is a hint to the I/O logging plugin which may choose to ignore it.

iolog_ttyin=bool
: Set to true if the I/O logging plugins, if any, should log all terminal input. This only includes input typed by the user and not from a pipe or redirected from a file. This is a hint to the I/O logging plugin which may choose to ignore it.

iolog_ttyout=bool
: Set to true if the I/O logging plugins, if any, should log all terminal output. This only includes output to the screen, not output to a pipe or file. This is a hint to the I/O logging plugin which may choose to ignore it.

login_class=string
: BSD login class to use when setting resource limits and nice value (optional). This option is only set on systems that support login classes.

nice=int
: Nice value (priority) to use when executing the command. The nice value, if

specified, overrides the priority associated with the *login_class* on BSD systems.

noexec=bool
: If set, prevent the command from executing other programs.

preserve_fds=list
: A comma-separated list of file descriptors that should be preserved, regardless of the value of the *closefrom* setting. Only available starting with API version 1.5.

preserve_groups=bool
: If set, **sudo** will preserve the user's group vector instead of initializing the group vector based on runas_user.

runas_egid=gid
: Effective group ID to run the command as. If not specified, the value of *runas_gid* is used.

runas_euid=uid
: Effective user ID to run the command as. If not specified, the value of *runas_uid* is used.

runas_gid=gid
: Group ID to run the command as.

runas_groups=list
: The supplementary group vector to use for the command in the form of a comma-separated list of group IDs. If *preserve_groups* is set, this option is ignored.

runas_uid=uid
: User ID to run the command as.

selinux_role=string
: SELinux role to use when executing the command.

selinux_type=string
: SELinux type to use when executing the command.

set_utmp=bool
: Create a utmp (or utmpx) entry when a pseudo-tty is allocated. By default, the new entry will be a copy of the user's existing utmp entry (if any), with the tty, time, type

and pid fields updated.

sudoedit=bool
Set to true when in *sudoedit* mode. The plugin may enable *sudoedit* mode even if **sudo** was not invoked as **sudoedit**. This allows the plugin to perform command substitution and transparently enable *sudoedit* when the user attempts to run an editor.

timeout=int
Command timeout. If non-zero then when the timeout expires the command will be killed.

umask=octal
The file creation mask to use when executing the command.

use_pty=bool
Allocate a pseudo-tty to run the command in, regardless of whether or not I/O logging is in use. By default, **sudo** will only run the command in a pty when an I/O log plugin is loaded.

utmp_user=string
User name to use when constructing a new utmp (or utmpx) entry when *set_utmp* is enabled. This option can be used to set the user field in the utmp entry to the user the command runs as rather than the invoking user. If not set, **sudo** will base the new entry on the invoking user's existing entry.

Unsupported values will be ignored.

argv_out
The NULL-terminated argument vector to pass to the execve(2) system call when executing the command. The plugin is responsible for allocating and populating the vector.

user_env_out
The NULL-terminated environment vector to use when executing the command. The plugin is responsible for allocating and populating the vector.

list

```
int (*list)(int verbose, const char *list_user,
    int argc, char * const argv[]);
```

List available privileges for the invoking user. Returns 1 on success, 0 on failure and -1 on error.

On error, the plugin may optionally call the **conversation**() or **plugin_printf**() function with SUDO_CONF_ERROR_MSG to present additional error information to the user.

Privileges should be output via the **conversation**() or **plugin_printf**() function using SUDO_CONV_INFO_MSG,

verbose
: Flag indicating whether to list in verbose mode or not.

list_user
: The name of a different user to list privileges for if the policy allows it. If NULL, the plugin should list the privileges of the invoking user.

argc
: The number of elements in *argv*, not counting the final NULL pointer.

argv
: If non-NULL, an argument vector describing a command the user wishes to check against the policy in the same form as what would be passed to the execve(2) system call. If the command is permitted by the policy, the fully-qualified path to the command should be displayed along with any command line arguments.

validate
: int (*validate)(void);

The **validate**() function is called when **sudo** is run with the **-v** flag. For policy plugins such as **sudoers** that cache authentication credentials, this function will validate and cache the credentials.

The **validate**() function should be NULL if the plugin does not support credential caching.

Returns 1 on success, 0 on failure and -1 on error. On error, the plugin may optionally call the **conversation**() or **plugin_printf**() function with SUDO_CONF_ERROR_MSG to present additional error information to the user.

invalidate
: void (*invalidate)(int remove);

The **invalidate**() function is called when **sudo** is called with the **-k** or **-K** flag. For policy plugins such as **sudoers** that cache authentication credentials, this function will invalidate the credentials. If the *remove* flag is set, the plugin may remove the credentials instead of simply invalidating them.

The **invalidate**() function should be NULL if the plugin does not support credential caching.

init_session

int (*init_session)(struct passwd *pwd, char **user_envp[];

The **init_session**() function is called before **sudo** sets up the execution environment for the command. It is run in the parent **sudo** process and before any uid or gid changes. This can be used to perform session setup that is not supported by *command_info*, such as opening the PAM session. The **close**() function can be used to tear down the session that was opened by init_session.

The *pwd* argument points to a passwd struct for the user the command will be run as if the uid the command will run as was found in the password database, otherwise it will be NULL.

The *user_env* argument points to the environment the command will run in, in the form of a NULL-terminated vector of ‘‘name=value’’ strings. This is the same string passed back to the front end via the Policy Plugin’s *user_env_out* parameter. If the **init_session**() function needs to modify the user environment, it should update the pointer stored in *user_env*. The expected use case is to merge the contents of the PAM environment (if any) with the contents of *user_env*. NOTE: the *user_env* parameter is only available starting with API version 1.2. A plugin **must** check the API version specified by the **sudo** front end before using *user_env*. Failure to do so may result in a crash.

Returns 1 on success, 0 on failure and -1 on error. On error, the plugin may optionally call the **conversation**() or **plugin_printf**() function with SUDO_CONF_ERROR_MSG to present additional error information to the user.

register_hooks

void (*register_hooks)(int version,
int (*register_hook)(struct sudo_hook *hook));

The **register_hooks**() function is called by the sudo front end to register any hooks the plugin needs. If the plugin does not support hooks, register_hooks should be set to the NULL pointer.

The *version* argument describes the version of the hooks API supported by the **sudo** front end.

The **register_hook**() function should be used to register any supported hooks the plugin needs. It returns 0 on success, 1 if the hook type is not supported and -1 if the major version in struct hook does not match the front end’s major hook API version.

See the *Hook function API* section below for more information about hooks.

NOTE: the **register_hooks**() function is only available starting with API version 1.2. If the **sudo** front end doesn't support API version 1.2 or higher, register_hooks will not be called.

deregister_hooks

```
void (*deregister_hooks)(int version,
  int (*deregister_hook)(struct sudo_hook *hook));
```

The **deregister_hooks**() function is called by the sudo front end to deregister any hooks the plugin has registered. If the plugin does not support hooks, deregister_hooks should be set to the NULL pointer.

The *version* argument describes the version of the hooks API supported by the **sudo** front end.

The **deregister_hook**() function should be used to deregister any hooks that were put in place by the **register_hook**() function. If the plugin tries to deregister a hook that the front end does not support, deregister_hook will return an error.

See the *Hook function API* section below for more information about hooks.

NOTE: the **deregister_hooks**() function is only available starting with API version 1.2. If the **sudo** front end doesn't support API version 1.2 or higher, deregister_hooks will not be called.

Policy Plugin Version Macros

```
/* Plugin API version major/minor. */
#define SUDO_API_VERSION_MAJOR 1
#define SUDO_API_VERSION_MINOR 2
#define SUDO_API_MKVERSION(x, y) ((x << 16) | y)
#define SUDO_API_VERSION SUDO_API_MKVERSION(SUDO_API_VERSION_MAJOR,\
                                SUDO_API_VERSION_MINOR)

/* Getters and setters for API version */
#define SUDO_API_VERSION_GET_MAJOR(v) ((v) >> 16)
#define SUDO_API_VERSION_GET_MINOR(v) ((v) & 0xffff)
#define SUDO_API_VERSION_SET_MAJOR(vp, n) do { \
    *(vp) = (*(vp) & 0x0000ffff) | ((n) << 16); \
} while(0)
#define SUDO_VERSION_SET_MINOR(vp, n) do { \
```

```
    *(vp) = (*(vp) & 0xffff0000) | (n); \
} while(0)
```

I/O plugin API

```
struct io_plugin {
#define SUDO_IO_PLUGIN 2
    unsigned int type; /* always SUDO_IO_PLUGIN */
    unsigned int version; /* always SUDO_API_VERSION */
    int (*open)(unsigned int version, sudo_conv_t conversation,
            sudo_printf_t plugin_printf, char * const settings[],
            char * const user_info[], char * const command_info[],
            int argc, char * const argv[], char * const user_env[],
            char * const plugin_options[]);
    void (*close)(int exit_status, int error); /* wait status or error */
    int (*show_version)(int verbose);
    int (*log_ttyin)(const char *buf, unsigned int len);
    int (*log_ttyout)(const char *buf, unsigned int len);
    int (*log_stdin)(const char *buf, unsigned int len);
    int (*log_stdout)(const char *buf, unsigned int len);
    int (*log_stderr)(const char *buf, unsigned int len);
    void (*register_hooks)(int version,
      int (*register_hook)(struct sudo_hook *hook));
    void (*deregister_hooks)(int version,
      int (*deregister_hook)(struct sudo_hook *hook));
};
```

When an I/O plugin is loaded, **sudo** runs the command in a pseudo-tty. This makes it possible to log the input and output from the user's session. If any of the standard input, standard output or standard error do not correspond to a tty, **sudo** will open a pipe to capture the I/O for logging before passing it on.

The log_ttyin function receives the raw user input from the terminal device (note that this will include input even when echo is disabled, such as when a password is read). The log_ttyout function receives output from the pseudo-tty that is suitable for replaying the user's session at a later time. The **log_stdin**(), **log_stdout**() and **log_stderr**() functions are only called if the standard input, standard output or standard error respectively correspond to something other than a tty.

Any of the logging functions may be set to the NULL pointer if no logging is to be performed. If the open function returns 0, no I/O will be sent to the plugin.

If a logging function returns an error (-1), the running command will be terminated and all of the

plugin's logging functions will be disabled. Other I/O logging plugins will still receive any remaining input or output that has not yet been processed.

If an input logging function rejects the data by returning 0, the command will be terminated and the data will not be passed to the command, though it will still be sent to any other I/O logging plugins. If an output logging function rejects the data by returning 0, the command will be terminated and the data will not be written to the terminal, though it will still be sent to any other I/O logging plugins.

The io_plugin struct has the following fields:

type The type field should always be set to SUDO_IO_PLUGIN.

version
The version field should be set to SUDO_API_VERSION.

This allows **sudo** to determine the API version the plugin was built against.

open
```
int (*open)(unsigned int version, sudo_conv_t conversation,
    sudo_printf_t plugin_printf, char * const settings[],
    char * const user_info[], int argc, char * const argv[],
    char * const user_env[], char * const plugin_options[]);
```

The **open**() function is run before the **log_ttyin**(), **log_ttyout**(), **log_stdin**(), **log_stdout**(), **log_stderr**(), or **show_version**() functions are called. It is only called if the version is being requested or if the policy plugin's **check_policy**() function has returned successfully. It returns 1 on success, 0 on failure, -1 if a general error occurred, or -2 if there was a usage error. In the latter case, **sudo** will print a usage message before it exits. If an error occurs, the plugin may optionally call the **conversation**() or **plugin_printf**() function with SUDO_CONF_ERROR_MSG to present additional error information to the user.

The function arguments are as follows:

version
The version passed in by **sudo** allows the plugin to determine the major and minor version number of the plugin API supported by **sudo**.

conversation
A pointer to the **conversation**() function that may be used by the **show_version**() function to display version information (see **show_version**() below). The **conversation**() function may

also be used to display additional error message to the user. The **conversation**() function returns 0 on success and -1 on failure.

plugin_printf
A pointer to a **printf**()-style function that may be used by the **show_version**() function to display version information (see show_version below). The **plugin_printf**() function may also be used to display additional error message to the user. The **plugin_printf**() function returns number of characters printed on success and -1 on failure.

settings
A vector of user-supplied **sudo** settings in the form of ‘‘name=value’’ strings. The vector is terminated by a NULL pointer. These settings correspond to flags the user specified when running **sudo**. As such, they will only be present when the corresponding flag has been specified on the command line.

When parsing *settings*, the plugin should split on the **first** equal sign (‘=’) since the *name* field will never include one itself but the *value* might.

See the *Policy plugin API* section for a list of all possible settings.

user_info
A vector of information about the user running the command in the form of ‘‘name=value’’ strings. The vector is terminated by a NULL pointer.

When parsing *user_info*, the plugin should split on the **first** equal sign (‘=’) since the *name* field will never include one itself but the *value* might.

See the *Policy plugin API* section for a list of all possible strings.

argc The number of elements in *argv*, not counting the final NULL pointer.

argv If non-NULL, an argument vector describing a command the user wishes to run in the same form as what would be passed to the execve(2) system call.

user_env
The user’s environment in the form of a NULL-terminated vector of ‘‘name=value’’ strings.

When parsing *user_env*, the plugin should split on the **first** equal sign (‘=’) since the *name* field will never include one itself but the *value* might.

plugin_options

Any (non-comment) strings immediately after the plugin path are treated as arguments to the plugin. These arguments are split on a white space boundary and are passed to the plugin in the form of a NULL-terminated array of strings. If no arguments were specified, *plugin_options* will be the NULL pointer.

NOTE: the *plugin_options* parameter is only available starting with API version 1.2. A plugin **must** check the API version specified by the **sudo** front end before using *plugin_options*. Failure to do so may result in a crash.

close

void (*close)(int exit_status, int error);

The **close**() function is called when the command being run by **sudo** finishes.

The function arguments are as follows:

exit_status

The command's exit status, as returned by the wait(2) system call. The value of exit_status is undefined if error is non-zero.

error If the command could not be executed, this is set to the value of errno set by the execve(2) system call. If the command was successfully executed, the value of error is 0.

show_version

int (*show_version)(int verbose);

The **show_version**() function is called by **sudo** when the user specifies the **-V** option. The plugin may display its version information to the user via the **conversation**() or **plugin_printf**() function using SUDO_CONV_INFO_MSG. If the user requests detailed version information, the verbose flag will be set.

log_ttyin

int (*log_ttyin)(const char *buf, unsigned int len);

The **log_ttyin**() function is called whenever data can be read from the user but before it is passed to the running command. This allows the plugin to reject data if it chooses to (for instance if the input contains banned content). Returns 1 if the data should be passed to the command, 0 if the data is rejected (which will terminate the running command) or -1 if an error occurred.

The function arguments are as follows:

buf The buffer containing user input.

len The length of *buf* in bytes.

log_ttyout

int (*log_ttyout)(const char *buf, unsigned int len);

The **log_ttyout**() function is called whenever data can be read from the command but before it is written to the user's terminal. This allows the plugin to reject data if it chooses to (for instance if the output contains banned content). Returns 1 if the data should be passed to the user, 0 if the data is rejected (which will terminate the running command) or -1 if an error occurred.

The function arguments are as follows:

buf The buffer containing command output.

len The length of *buf* in bytes.

log_stdin

int (*log_stdin)(const char *buf, unsigned int len);

The **log_stdin**() function is only used if the standard input does not correspond to a tty device. It is called whenever data can be read from the standard input but before it is passed to the running command. This allows the plugin to reject data if it chooses to (for instance if the input contains banned content). Returns 1 if the data should be passed to the command, 0 if the data is rejected (which will terminate the running command) or -1 if an error occurred.

The function arguments are as follows:

buf The buffer containing user input.

len The length of *buf* in bytes.

log_stdout

int (*log_stdout)(const char *buf, unsigned int len);

The **log_stdout**() function is only used if the standard output does not correspond to a tty device. It is called whenever data can be read from the command but before it is written to the standard

output. This allows the plugin to reject data if it chooses to (for instance if the output contains banned content). Returns 1 if the data should be passed to the user, 0 if the data is rejected (which will terminate the running command) or -1 if an error occurred.

The function arguments are as follows:

buf The buffer containing command output.

len The length of *buf* in bytes.

log_stderr

int (*log_stderr)(const char *buf, unsigned int len);

The **log_stderr**() function is only used if the standard error does not correspond to a tty device. It is called whenever data can be read from the command but before it is written to the standard error. This allows the plugin to reject data if it chooses to (for instance if the output contains banned content). Returns 1 if the data should be passed to the user, 0 if the data is rejected (which will terminate the running command) or -1 if an error occurred.

The function arguments are as follows:

buf The buffer containing command output.

len The length of *buf* in bytes.

register_hooks

See the *Policy plugin API* section for a description of register_hooks.

deregister_hooks

See the *Policy plugin API* section for a description of deregister_hooks.

I/O Plugin Version Macros

Same as for the *Policy plugin API*.

Signal handlers

The **sudo** front end installs default signal handlers to trap common signals while the plugin functions are run. The following signals are trapped by default before the command is executed:

- SIGALRM

- SIGHUP
- SIGINT
- SIGQUIT
- SIGTERM
- SIGTSTP
- SIGUSR1
- SIGUSR2

If a fatal signal is received before the command is executed, **sudo** will call the plugin's **close**() function with an exit status of 128 plus the value of the signal that was received. This allows for consistent logging of commands killed by a signal for plugins that log such information in their **close**() function.

A plugin may temporarily install its own signal handlers but must restore the original handler before the plugin function returns.

Hook function API

Beginning with plugin API version 1.2, it is possible to install hooks for certain functions called by the **sudo** front end.

Currently, the only supported hooks relate to the handling of environment variables. Hooks can be used to intercept attempts to get, set, or remove environment variables so that these changes can be reflected in the version of the environment that is used to execute a command. A future version of the API will support hooking internal **sudo** front end functions as well.

Hook structure

Hooks in **sudo** are described by the following structure:

```
typedef int (*sudo_hook_fn_t)();

struct sudo_hook {
    int hook_version;
    int hook_type;
    sudo_hook_fn_t hook_fn;
    void *closure;
};
```

The sudo_hook structure has the following fields:

hook_version

The hook_version field should be set to SUDO_HOOK_VERSION.

hook_type

The hook_type field may be one of the following supported hook types:

SUDO_HOOK_SETENV

The C library setenv(3) function. Any registered hooks will run before the C library implementation. The hook_fn field should be a function that matches the following typedef:

```
typedef int (*sudo_hook_fn_setenv_t)(const char *name,
  const char *value, int overwrite, void *closure);
```

If the registered hook does not match the typedef the results are unspecified.

SUDO_HOOK_UNSETENV

The C library unsetenv(3) function. Any registered hooks will run before the C library implementation. The hook_fn field should be a function that matches the following typedef:

```
typedef int (*sudo_hook_fn_unsetenv_t)(const char *name,
  void *closure);
```

SUDO_HOOK_GETENV

The C library getenv(3) function. Any registered hooks will run before the C library implementation. The hook_fn field should be a function that matches the following typedef:

```
typedef int (*sudo_hook_fn_getenv_t)(const char *name,
  char **value, void *closure);
```

If the registered hook does not match the typedef the results are unspecified.

SUDO_HOOK_PUTENV

The C library putenv(3) function. Any registered hooks will run before the C library implementation. The hook_fn field should be a function that matches the following typedef:

```
typedef int (*sudo_hook_fn_putenv_t)(char *string,
  void *closure);
```

If the registered hook does not match the typedef the results are unspecified.

hook_fn

sudo_hook_fn_t hook_fn;

The hook_fn field should be set to the plugin's hook implementation. The actual function arguments will vary depending on the hook_type (see hook_type above). In all cases, the closure field of struct sudo_hook is passed as the last function parameter. This can be used to pass arbitrary data to the plugin's hook implementation.

The function return value may be one of the following:

SUDO_HOOK_RET_ERROR

The hook function encountered an error.

SUDO_HOOK_RET_NEXT

The hook completed without error, go on to the next hook (including the native implementation if applicable). For example, a getenv(3) hook might return SUDO_HOOK_RET_NEXT if the specified variable was not found in the private copy of the environment.

SUDO_HOOK_RET_STOP

The hook completed without error, stop processing hooks for this invocation. This can be used to replace the native implementation. For example, a setenv hook that operates on a private copy of the environment but leaves environ unchanged.

Note that it is very easy to create an infinite loop when hooking C library functions. For example, a getenv(3) hook that calls the snprintf(3) function may create a loop if the snprintf(3) implementation calls getenv(3) to check the locale. To prevent this, you may wish to use a static variable in the hook function to guard against nested calls. For example:

```
static int in_progress = 0; /* avoid recursion */
if (in_progress)
   return SUDO_HOOK_RET_NEXT;
in_progress = 1;
...
in_progress = 0;
return SUDO_HOOK_RET_STOP;
```

Hook API Version Macros

```
/* Hook API version major/minor */
#define SUDO_HOOK_VERSION_MAJOR 1
#define SUDO_HOOK_VERSION_MINOR 0
#define SUDO_HOOK_MKVERSION(x, y) ((x << 16) | y)
#define SUDO_HOOK_VERSION SUDO_HOOK_MKVERSION(SUDO_HOOK_VERSION_MAJOR,\
                                SUDO_HOOK_VERSION_MINOR)

/* Getters and setters for hook API version */
#define SUDO_HOOK_VERSION_GET_MAJOR(v) ((v) >> 16)
#define SUDO_HOOK_VERSION_GET_MINOR(v) ((v) & 0xffff)
#define SUDO_HOOK_VERSION_SET_MAJOR(vp, n) do { \
    *(vp) = (*(vp) & 0x0000ffff) | ((n) << 16); \
} while(0)
#define SUDO_HOOK_VERSION_SET_MINOR(vp, n) do { \
    *(vp) = (*(vp) & 0xffff0000) | (n); \
} while(0)
```

Remote command execution

The **sudo** front end does not have native support for running remote commands. However, starting with **sudo** 1.8.8, the **-h** option may be used to specify a remote host that is passed to the policy plugin. A plugin may also accept a *runas_user* in the form of ‘‘user@hostname’’ which will work with older versions of **sudo**. It is anticipated that remote commands will be supported by executing a ‘‘helper’’ program. The policy plugin should setup the execution environment such that the **sudo** front end will run the helper which, in turn, will connect to the remote host and run the command.

For example, the policy plugin could utilize **ssh** to perform remote command execution. The helper program would be responsible for running **ssh** with the proper options to use a private key or certificate that the remote host will accept and run a program on the remote host that would setup the execution environment accordingly.

Note that remote **sudoedit** functionality must be handled by the policy plugin, not **sudo** itself as the front end has no knowledge that a remote command is being executed. This may be addressed in a future revision of the plugin API.

Conversation API

If the plugin needs to interact with the user, it may do so via the **conversation**() function. A plugin should not attempt to read directly from the standard input or the user’s tty (neither of which are guaranteed to exist). The caller must include a trailing newline in msg if one is to be printed.

A **printf**()-style function is also available that can be used to display informational or error messages to

the user, which is usually more convenient for simple messages where no use input is required.

```
struct sudo_conv_message {
#define SUDO_CONV_PROMPT_ECHO_OFF  0x0001 /* do not echo user input */
#define SUDO_CONV_PROMPT_ECHO_ON   0x0002 /* echo user input */
#define SUDO_CONV_ERROR_MSG        0x0003 /* error message */
#define SUDO_CONV_INFO_MSG         0x0004 /* informational message */
#define SUDO_CONV_PROMPT_MASK      0x0005 /* mask user input */
#define SUDO_CONV_PROMPT_ECHO_OK   0x1000 /* flag: allow echo if no tty */
    int msg_type;
    int timeout;
    const char *msg;
};

#define SUDO_CONV_REPL_MAX      255

struct sudo_conv_reply {
    char *reply;
};

typedef int (*sudo_conv_t)(int num_msgs,
            const struct sudo_conv_message msgs[],
            struct sudo_conv_reply replies[]);

typedef int (*sudo_printf_t)(int msg_type, const char *fmt, ...);
```

Pointers to the **conversation**() and **printf**()-style functions are passed in to the plugin's **open**() function when the plugin is initialized.

To use the **conversation**() function, the plugin must pass an array of sudo_conv_message and sudo_conv_reply structures. There must be a struct sudo_conv_message and struct sudo_conv_reply for each message in the conversation. The plugin is responsible for freeing the reply buffer located in each struct sudo_conv_reply, if it is not NULL. SUDO_CONV_REPL_MAX represents the maximum length of the reply buffer (not including the trailing NUL character). In practical terms, this is the longest password **sudo** will support. It is also useful as a maximum value for the **memset_s**() function when clearing passwords filled in by the conversation function.

The **printf**()-style function uses the same underlying mechanism as the **conversation**() function but only supports SUDO_CONV_INFO_MSG and SUDO_CONV_ERROR_MSG for the *msg_type* parameter. It can be more convenient than using the **conversation**() function if no user reply is needed and supports

standard **printf**() escape sequences.

See the sample plugin for an example of the **conversation**() function usage.

Sudoers group plugin API

The **sudoers** plugin supports its own plugin interface to allow non-Unix group lookups. This can be used to query a group source other than the standard Unix group database. Two sample group plugins are bundled with **sudo**, *group_file* and *system_group*, are detailed in sudoers(5). Third party group plugins include a QAS AD plugin available from Quest Software.

A group plugin must declare and populate a sudoers_group_plugin struct in the global scope. This structure contains pointers to the functions that implement plugin initialization, cleanup and group lookup.

```
struct sudoers_group_plugin {
  unsigned int version;
  int (*init)(int version, sudo_printf_t sudo_printf,
          char *const argv[]);
  void (*cleanup)(void);
  int (*query)(const char *user, const char *group,
           const struct passwd *pwd);
};
```

The sudoers_group_plugin struct has the following fields:

version

The version field should be set to GROUP_API_VERSION.

This allows **sudoers** to determine the API version the group plugin was built against.

init

```
int (*init)(int version, sudo_printf_t plugin_printf,
        char *const argv[]);
```

The **init**() function is called after *sudoers* has been parsed but before any policy checks. It returns 1 on success, 0 on failure (or if the plugin is not configured), and -1 if a error occurred. If an error occurs, the plugin may call the **plugin_printf**() function with SUDO_CONF_ERROR_MSG to present additional error information to the user.

The function arguments are as follows:

version
: The version passed in by **sudoers** allows the plugin to determine the major and minor version number of the group plugin API supported by **sudoers**.

plugin_printf
: A pointer to a **printf**()-style function that may be used to display informational or error message to the user. Returns the number of characters printed on success and -1 on failure.

argv
: A NULL-terminated array of arguments generated from the *group_plugin* option in *sudoers*. If no arguments were given, *argv* will be NULL.

cleanup

```
void (*cleanup)();
```

The **cleanup**() function is called when **sudoers** has finished its group checks. The plugin should free any memory it has allocated and close open file handles.

query

```
int (*query)(const char *user, const char *group,
    const struct passwd *pwd);
```

The **query**() function is used to ask the group plugin whether *user* is a member of *group*.

The function arguments are as follows:

user
: The name of the user being looked up in the external group database.

group
: The name of the group being queried.

pwd
: The password database entry for *user*, if any. If *user* is not present in the password database, *pwd* will be NULL.

Group API Version Macros

```
/* Sudoers group plugin version major/minor */
#define GROUP_API_VERSION_MAJOR 1
#define GROUP_API_VERSION_MINOR 0
#define GROUP_API_VERSION ((GROUP_API_VERSION_MAJOR << 16) | \
                GROUP_API_VERSION_MINOR)
```

```
/* Getters and setters for group version */
#define GROUP_API_VERSION_GET_MAJOR(v) ((v) >> 16)
#define GROUP_API_VERSION_GET_MINOR(v) ((v) & 0xffff)
#define GROUP_API_VERSION_SET_MAJOR(vp, n) do { \
    *(vp) = (*(vp) & 0x0000ffff) | ((n) << 16); \
} while(0)
#define GROUP_API_VERSION_SET_MINOR(vp, n) do { \
    *(vp) = (*(vp) & 0xffff0000) | (n); \
} while(0)
```

PLUGIN API CHANGELOG

The following revisions have been made to the Sudo Plugin API.

Version 1.0

Initial API version.

Version 1.1 (sudo 1.8.0)

The I/O logging plugin's **open**() function was modified to take the command_info list as an argument.

Version 1.2 (sudo 1.8.5)

The Policy and I/O logging plugins' **open**() functions are now passed a list of plugin parameters if any are specified in sudo.conf(5).

A simple hooks API has been introduced to allow plugins to hook in to the system's environment handling functions.

The init_session Policy plugin function is now passed a pointer to the user environment which can be updated as needed. This can be used to merge in environment variables stored in the PAM handle before a command is run.

Version 1.3 (sudo 1.8.7)

Support for the *exec_background* entry has been added to the command_info list.

The *max_groups* and *plugin_dir* entries were added to the settings list.

The **version**() and **close**() functions are now optional. Previously, a missing **version**() or **close**() function would result in a crash. If no policy plugin **close**() function is defined, a default **close**() function will be provided by the **sudo** front end that displays a warning if the command could not be executed.

The **sudo** front end now installs default signal handlers to trap common signals while the plugin functions are run.

Version 1.4 (sudo 1.8.8)

The *remote_host* entry was added to the settings list.

Version 1.5 (sudo 1.8.9)

The *preserve_fds* entry was added to the command_info list.

Version 1.6 (sudo 1.8.11)

The behavior when an I/O logging plugin returns an error (-1) has changed. Previously, the **sudo** front end took no action when the **log_ttyin**(), **log_ttyout**(), **log_stdin**(), **log_stdout**(), or **log_stderr**() function returned an error.

The behavior when an I/O logging plugin returns 0 has changed. Previously, output from the command would be displayed to the terminal even if an output logging function returned 0.

Version 1.7 (sudo 1.8.12)

The *plugin_path* entry was added to the settings list.

The *debug_flags* entry now starts with a debug file path name and may occur multiple times if there are multiple plugin-specific Debug lines in the sudo.conf(5) file.

SEE ALSO

sudo.conf(5), sudoers(5), sudo(8)

BUGS

If you feel you have found a bug in **sudo**, please submit a bug report at https://bugzilla.sudo.ws/

SUPPORT

Limited free support is available via the sudo-users mailing list, see https://www.sudo.ws/mailman/listinfo/sudo-users to subscribe or search the archives.

DISCLAIMER