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** plugin provides audit, security policy and I/O logging capabilities. Via the plugin API, **sudo** can be configured to use alternate 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[], const char **errstr);
    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[], const char **errstr);
    int (*list)(int argc, char * const argv[], int verbose,
        const char *list_user, const char **errstr);
    int (*validate)(const char **errstr);
    void (*invalidate)(int remove);
    int (*init_session)(struct passwd *pwd, char **user_env[],
        const char **errstr);
```

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

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[], const char **errstr);
```

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 *Conversation API* for details). 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 *Conversation API* for details). 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 options the user specified when running **sudo**. As such, they will only be present when the corresponding option 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.

The following values may be set by **sudo**:

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

closefrom=number
If specified, the user has requested via the **-C** option 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.

cmnd_chroot=string
The root directory (see chroot(2)) to run the command in, as specified by the user via the **-R** option. The plugin may ignore or restrict the user's ability to specify a new root directory. Only available starting with API version 1.16.

cmnd_cwd=string
The working directory to run the command in, as specified by the user via the **-D** option. The plugin may ignore or restrict the user's ability to specify a new working directory. Only available starting with API version 1.16.

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.

ignore_ticket=bool
: Set to true if the user specified the **-k** option 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** option.

login_shell=bool
: Set to true if the user specified the **-i** option, 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** option, 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** option, indicating that the user wishes to preserve the environment.

preserve_groups=bool
: Set to true if the user specified the **-P** option, 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** option.

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** option, indicating that the user wishes to run a shell.

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

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

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

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

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

sudoedit=bool
Set to true when the **-e** option 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.

timeout=string
Command timeout specified by the user via the **-T** option. Not all plugins support command timeouts and the ability of the user to set a timeout may be restricted by policy. The format of the timeout string is plugin-specific.

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.

The following values may be set by **sudo**:

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.

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

rlimit_as=soft,hard
: The maximum size to which the process’s address space may grow (in bytes), if supported by the operating system. The soft and hard limits are separated by a comma. A value of "infinity" indicates that there is no limit. Only available starting with API version 1.16.

rlimit_core=soft,hard
: The largest size core dump file that may be created (in bytes). The soft and hard limits are separated by a comma. A value of "infinity" indicates that there is no limit. Only available starting with API version 1.16.

rlimit_cpu=soft,hard
: The maximum amount of CPU time that the process may use (in seconds). The soft and hard limits are separated by a comma. A value of "infinity" indicates that there is no limit. Only available starting with API version 1.16.

rlimit_data=soft,hard
: The maximum size of the data segment for the process (in bytes). The soft and hard limits are separated by a comma. A value of "infinity" indicates that there is no limit. Only available starting with API version 1.16.

rlimit_fsize=soft,hard
: The largest size file that the process may create (in bytes). The soft and hard limits are separated by a comma. A value of "infinity" indicates that there is no limit. Only available starting with API version 1.16.

rlimit_locks=soft,hard
: The maximum number of locks that the process may establish, if supported by the operating system. The soft and hard limits are separated by a comma. A value of "infinity" indicates that there is no limit. Only available starting with API version 1.16.

rlimit_memlock=soft,hard
: The maximum size that the process may lock in memory (in bytes), if supported by the operating system. The soft and hard limits are separated by a comma. A value of "infinity" indicates that there is no limit. Only available starting with API version 1.16.

rlimit_nofile=soft,hard
: The maximum number of files that the process may have open. The soft and hard limits are separated by a comma. A value of "infinity" indicates that there is no limit. Only available starting with API version 1.16.

rlimit_nproc=soft,hard
: The maximum number of processes that the user may run simultaneously. The soft and hard limits are separated by a comma. A value of "infinity" indicates that there is no limit. Only available starting with API version 1.16.

rlimit_rss=soft,hard
: The maximum size to which the process’s resident set size may grow (in bytes). The soft and hard limits are separated by a comma. A value of "infinity" indicates that

there is no limit. Only available starting with API version 1.16.

rlimit_stack=soft,hard
: The maximum size to which the process's stack may grow (in bytes). The soft and hard limits are separated by a comma. A value of "infinity" indicates that there is no limit. Only available starting with API version 1.16.

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 0 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**.

umask=octal
: The invoking user's file creation mask. Only available starting with API version 1.10.

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.

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.

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.

errstr

If the **open**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

close

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

The **close**() function is called when **sudo** is finished, shortly before it exits. Starting with API version 1.15, **close**() is called regardless of whether or not a command was actually executed. This makes it possible for plugins to perform cleanup even when a command was not run. It is not possible to tell whether a command was run based solely on the arguments passed to the **close**() function. To determine if a command was actually run, the plugin must keep track of whether or not the **check_policy**() function returned successfully.

The function arguments are as follows:

exit_status

The command's exit status, as returned by the wait(2) system call, or zero if no command was run. 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 zero.

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.

Returns 1 on success, 0 on failure, -1 if a general error occurred, or -2 if there was a usage error, although the return value is currently ignored.

check_policy

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

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* (environment variables may include command line options). 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. If **sudo** is unable to change to the new working directory, the command will not be run unless *cwd_optional* is also set (see below).

cwd_optional=bool
If enabled, **sudo** will treat an inability to change to the new working directory as a non-fatal error. This setting has no effect unless *cwd* is also set.

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 pseudo-terminal (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 pseudo-terminal.

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, macOS fails to restart the **tcgetattr**() and **tcsetattr**() system calls (this is a bug in macOS). 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.

execfd=number
: If specified, **sudo** will use the fexecve(2) system call to execute the command instead of execve(2). The specified *number* must refer to an open file descriptor.

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_group=string
: The group that will own newly created I/O log files and directories. This is a hint to the I/O logging plugin which may choose to ignore it.

iolog_mode=octal
: The file permission mode to use when creating I/O log files and directories. This is a hint to the I/O logging plugin which may choose to ignore it.

iolog_user=string
: The user that will own newly created I/O log files and directories. 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.

rlimit_as=soft,hard
: The maximum size to which the process's address space may grow (in bytes), if supported by the operating system. The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user's resource limit to be preserved. A value of "default" will cause the target user's default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

rlimit_core=soft,hard
: The largest size core dump file that may be created (in bytes). The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user's resource limit to be preserved. A value of "default" will cause the target user's default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

rlimit_cpu=soft,hard
: The maximum amount of CPU time that the process may use (in seconds). The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user's resource limit to be preserved. A value of "default" will cause the target user's default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

rlimit_data=soft,hard
: The maximum size of the data segment for the process (in bytes). The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user’s resource limit to be preserved. A value of "default" will cause the target user’s default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

rlimit_fsize=soft,hard
: The largest size file that the process may create (in bytes). The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user’s resource limit to be preserved. A value of "default" will cause the target user’s default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

rlimit_locks=soft,hard
: The maximum number of locks that the process may establish, if supported by the operating system. The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user’s resource limit to be preserved. A value of "default" will cause the target user’s default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

rlimit_memlock=soft,hard
: The maximum size that the process may lock in memory (in bytes), if supported by the operating system. The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user’s resource limit to be preserved. A value of "default" will cause the target user’s default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

rlimit_nofile=soft,hard
: The maximum number of files that the process may have open. The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user’s resource limit to be preserved. A value of

"default" will cause the target user's default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

rlimit_nproc=soft,hard
: The maximum number of processes that the user may run simultaneously. The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user's resource limit to be preserved. A value of "default" will cause the target user's default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

rlimit_rss=soft,hard
: The maximum size to which the process's resident set size may grow (in bytes). The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user's resource limit to be preserved. A value of "default" will cause the target user's default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

rlimit_stack=soft,hard
: The maximum size to which the process's stack may grow (in bytes). The soft and hard limits are separated by a comma. If only a single value is specified, both the hard and soft limits are set. A value of "infinity" indicates that there is no limit. A value of "user" will cause the invoking user's resource limit to be preserved. A value of "default" will cause the target user's default resource limit to be used on systems that allow per-user resource limits to be configured. Only available starting with API version 1.17.

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_group=string
: The name of the group the command will run as, if it is different from the *runas_user*'s default group. This value is provided for auditing purposes only, the **sudo** front-end uses *runas_egid* and *runas_gid* when executing the command.

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.

runas_user=string
: The name of the user the command will run as, which should correspond to *runas_euid* (or *runas_uid* if *runas_euid* is not set). This value is provided for auditing purposes only, the **sudo** front-end uses *runas_euid* and *runas_uid* when executing the command.

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-terminal 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.

sudoedit_checkdir=bool
: Set to false to disable directory writability checks in **sudoedit**. By default, **sudoedit** 1.8.16 and higher will check all directory components of the path to be edited for writability by the invoking user. Symbolic links will not be followed in writable

directories and **sudoedit** will refuse to edit a file located in a writable directory. These restrictions are not enforced when **sudoedit** is run by root. The *sudoedit_follow* option can be set to false to disable this check. Only available starting with API version 1.8.

sudoedit_follow=bool
: Set to true to allow **sudoedit** to edit files that are symbolic links. By default, **sudoedit** 1.8.15 and higher will refuse to open a symbolic link. The *sudoedit_follow* option can be used to restore the older behavior and allow **sudoedit** to open symbolic links. Only available starting with API version 1.8.

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. This value may be overridden by PAM or login.conf on some systems unless the *umask_override* option is also set.

umask_override=bool
: Force the value specified by the *umask* option to override any umask set by PAM or login.conf.

use_pty=bool
: Allocate a pseudo-terminal 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 pseudo-terminal 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.

errstr
: If the **check_policy**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

  The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

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

  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.

  The function arguments are as follows:

  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.

  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.
  ```

errstr

If the **list**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

validate

int (*validate)(const char **errstr);

The **validate**() function is called when **sudo** is run with the **-v** option. 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.

The function arguments are as follows:

errstr

If the **validate**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

invalidate

void (*invalidate)(int remove);

The **invalidate**() function is called when **sudo** is run with the **-k** or **-K** option. 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_env_out[]);

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 user-ID or group-ID 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 user-ID the command will run as was found in the password database, otherwise it will be NULL.

The *user_env_out* 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_out*. The expected use case is to merge the contents of the PAM environment (if any) with the contents of *user_env_out*. The *user_env_out* 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_out*. 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.

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.

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.

event_alloc

```
struct sudo_plugin_event * (*event_alloc)(void);
```

The **event_alloc**() function is used to allocate a struct sudo_plugin_event which provides access to the main **sudo** event loop. Unlike the other fields, the **event_alloc**() pointer is filled in by the **sudo** front-end, not by the plugin.

See the *Event API* section below for more information about events.

The **event_alloc**() function is only available starting with API version 1.15. If the **sudo** front-end doesn't support API version 1.15 or higher, **event_alloc**() will not be set.

errstr

If the **init_session**() function returns a value other than 1, the plugin may store a message

describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

Policy Plugin Version Macros

```
/* Plugin API version major/minor. */
#define SUDO_API_VERSION_MAJOR 1
#define SUDO_API_VERSION_MINOR 13
#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_API_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[], const char **errstr);
    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,
```

```
        const char **errstr);
    int (*log_ttyout)(const char *buf, unsigned int len,
        const char **errstr);
    int (*log_stdin)(const char *buf, unsigned int len,
        const char **errstr);
    int (*log_stdout)(const char *buf, unsigned int len,
        const char **errstr);
    int (*log_stderr)(const char *buf, unsigned int len,
        const char **errstr);
    void (*register_hooks)(int version,
      int (*register_hook)(struct sudo_hook *hook));
    void (*deregister_hooks)(int version,
      int (*deregister_hook)(struct sudo_hook *hook));
    int (*change_winsize)(unsigned int lines, unsigned int cols,
        const char **errstr);
    int (*log_suspend)(int signo, const char **errstr);
    struct sudo_plugin_event * (*event_alloc)(void);
};
```

When an I/O plugin is loaded, **sudo** runs the command in a pseudo-terminal. 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 (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-terminal 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 audit_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[], char * const command_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**(), **log_suspend**(), **change_winsize**(), 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 options the user specified when running **sudo**. As such, they will only be present when the corresponding option 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.

command_info
: A vector of information describing the command being run in the form of "name=value" strings. The vector is terminated by a NULL pointer.

  When parsing *command_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. It can be zero, when **sudo** is called with **-V**.

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.

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.

errstr
If the **open**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

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

The **close**() function is called when **sudo** is finished, shortly before it exits.

The function arguments are as follows:

exit_status
The command's exit status, as returned by the wait(2) system call, or zero if no command was run. 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 zero.

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.

Returns 1 on success, 0 on failure, -1 if a general error occurred, or -2 if there was a usage error, although the return value is currently ignored.

log_ttyin

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

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.

errstr

If the **log_ttyin**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

log_ttyout

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

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.

errstr

If the **log_ttyout**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

log_stdin

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

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.

errstr

If the **log_stdin**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

log_stdout

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

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.

errstr

If the **log_stdout**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

log_stderr

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

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.

errstr
If the **log_stderr**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

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.

change_winsize
```
int (*change_winsize)(unsigned int lines, unsigned int cols,
    const char **errstr);
```

The **change_winsize**() function is called whenever the window size of the terminal changes from the initial values specified in the user_info list. Returns -1 if an error occurred, in which case no further calls to **change_winsize**() will be made,

The function arguments are as follows:

lines The number of lines (rows) in the re-sized terminal.

cols The number of columns in the re-sized terminal.

errstr
If the **change_winsize**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the

plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

log_suspend

int (*log_suspend)(int signo, const char **errstr);

The **log_suspend**() function is called whenever a command is suspended or resumed. Logging this information makes it possible to skip the period of time when the command was suspended during playback of a session. Returns -1 if an error occurred, in which case no further calls to **log_suspend**() will be made,

The function arguments are as follows:

signo

The signal that caused the command to be suspended, or SIGCONT if the command was resumed.

errstr

If the **log_suspend**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

The *errstr* parameter is only available starting with API version 1.15. A plugin **must** check the API version specified by the **sudo** front-end before using *errstr*. Failure to do so may result in a crash.

event_alloc

struct sudo_plugin_event * (*event_alloc)(void);

The **event_alloc**() function is used to allocate a struct sudo_plugin_event which provides access to the main **sudo** event loop. Unlike the other fields, the **event_alloc**() pointer is filled in by the **sudo** front-end, not by the plugin.

See the *Event API* section below for more information about events.

The **event_alloc**() function is only available starting with API version 1.15. If the **sudo**

front-end doesn't support API version 1.15 or higher, **event_alloc**() will not be set.

I/O Plugin Version Macros

Same as for the *Policy plugin API*.

Audit plugin API

```
/* Audit plugin close function status types. */
#define SUDO_PLUGIN_NO_STATUS          0
#define SUDO_PLUGIN_WAIT_STATUS        1
#define SUDO_PLUGIN_EXEC_ERROR         2
#define SUDO_PLUGIN_SUDO_ERROR         3

#define SUDO_AUDIT_PLUGIN 3
struct audit_plugin {
    unsigned int type; /* always SUDO_AUDIT_PLUGIN */
    unsigned int version; /* always SUDO_API_VERSION */
    int (*open)(unsigned int version, sudo_conv_t conversation,
        sudo_printf_t sudo_printf, char * const settings[],
        char * const user_info[], int submit_optind,
        char * const submit_argv[], char * const submit_envp[],
        char * const plugin_options[], const char **errstr);
    void (*close)(int status_type, int status);
    int (*accept)(const char *plugin_name,
        unsigned int plugin_type, char * const command_info[],
        char * const run_argv[], char * const run_envp[],
        const char **errstr);
    int (*reject)(const char *plugin_name, unsigned int plugin_type,
        const char *audit_msg, char * const command_info[],
        const char **errstr);
    int (*error)(const char *plugin_name, unsigned int plugin_type,
        const char *audit_msg, char * const command_info[],
        const char **errstr);
    int (*show_version)(int verbose);
    void (*register_hooks)(int version,
        int (*register_hook)(struct sudo_hook *hook));
    void (*deregister_hooks)(int version,
        int (*deregister_hook)(struct sudo_hook *hook));
    struct sudo_plugin_event * (*event_alloc)(void);
}
```

An audit plugin can be used to log successful and unsuccessful attempts to run **sudo** independent of the policy or any I/O plugins. Multiple audit plugins may be specified in sudo.conf(5).

The audit_plugin struct has the following fields:

type The type field should always be set to SUDO_AUDIT_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 sudo_printf, char * const settings[],
  char * const user_info[], int submit_optind,
  char * const submit_argv[], char * const submit_envp[],
  char * const plugin_options[], const char **errstr);
```

The audit **open**() function is run before any other **sudo** plugin API functions. This makes it possible to audit failures in the other plugins. 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 options the user specified when running **sudo**. As such, they will only be present when the corresponding option 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.

submit_optind
: The index into *submit_argv* that corresponds to the first entry that is not a command line option. If *submit_argv* only consists of options, which may be the case with the **-l** or **-v** options, submit_argv[submit_optind] will evaluate to the NULL pointer.

submit_argv
: The argument vector **sudo** was invoked with, including all command line options. The *submit_optind* argument can be used to determine the end of the command line options.

submit_envp
: The invoking user's environment in the form of a NULL-terminated vector of "name=value" strings.

  When parsing *submit_envp*, 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.

errstr

If the **open**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

close

```
void (*close)(int status_type, int status);
```

The **close**() function is called when **sudo** is finished, shortly before it exits.

The function arguments are as follows:

status_type

The type of status being passed. One of SUDO_PLUGIN_NO_STATUS, SUDO_PLUGIN_WAIT_STATUS, SUDO_PLUGIN_EXEC_ERROR or SUDO_PLUGIN_SUDO_ERROR.

status

Depending on the value of *status_type*, this value is either ignored, the command's exit status as returned by the wait(2) system call, the value of errno set by the execve(2) system call, or the value of errno resulting from an error in the **sudo** front-end.

accept

```
int (*accept)(const char *plugin_name, unsigned int plugin_type,
    char * const command_info[], char * const run_argv[],
    char * const run_envp[], const char **errstr);
```

The **accept**() function is called when a command or action is accepted by a policy or approval plugin. The function arguments are as follows:

plugin_name

The name of the plugin that accepted the command or "sudo" for the **sudo** front-end.

plugin_type
: The type of plugin that accepted the command, currently either SUDO_POLICY_PLUGIN, SUDO_POLICY_APPROVAL, or SUDO_FRONT_END. The **accept**() function is called multiple times--once for each policy or approval plugin that succeeds and once for the sudo front-end. When called on behalf of the sudo front-end, *command_info* may include information from an I/O logging plugin as well.

  Typically, an audit plugin is interested in either the accept status from the **sudo** front-end or from the various policy and approval plugins, but not both. It is possible for the policy plugin to accept a command that is later rejected by an approval plugin, in which case the audit plugin's **accept**() and **reject**() functions will *both* be called.

command_info
: An optional vector of information describing the command being run in the form of "name=value" strings. The vector is terminated by a NULL pointer.

  When parsing *command_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.

run_argv
: A NULL-terminated argument vector describing a command that will be run in the same form as what would be passed to the execve(2) system call.

run_envp
: The environment the command will be run with in the form of a NULL-terminated vector of "name=value" strings.

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

errstr
: If the **accept**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

reject
: int (*reject)(const char *plugin_name, unsigned int plugin_type,

```
const char *audit_msg, char * const command_info[],
const char **errstr);
```

The **reject**() function is called when a command or action is rejected by a plugin. The function arguments are as follows:

plugin_name
: The name of the plugin that rejected the command.

plugin_type
: The type of plugin that rejected the command, currently either SUDO_POLICY_PLUGIN, SUDO_APPROVAL_PLUGIN, or SUDO_IO_PLUGIN.

  Unlike the **accept**() function, the **reject**() function is not called on behalf of the **sudo** front-end.

audit_msg
: An optional string describing the reason the command was rejected by the plugin. If the plugin did not provide a reason, *audit_msg* will be the NULL pointer.

command_info
: An optional vector of information describing the command being run in the form of "name=value" strings. The vector is terminated by a NULL pointer.

  When parsing *command_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.

errstr
: If the **reject**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

error

```
int (*error)(const char *plugin_name, unsigned int plugin_type,
const char *audit_msg, char * const command_info[],
const char **errstr);
```

The **error**() function is called when a plugin or the **sudo** front-end returns an error. The function arguments are as follows:

plugin_name
: The name of the plugin that generated the error or "sudo" for the **sudo** front-end.

plugin_type
: The type of plugin that generated the error, or SUDO_FRONT_END for the **sudo** front-end.

audit_msg
: An optional string describing the plugin error. If the plugin did not provide a description, *audit_msg* will be the NULL pointer.

command_info
: An optional vector of information describing the command being run in the form of "name=value" strings. The vector is terminated by a NULL pointer.

  When parsing *command_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.

errstr
: If the **error**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

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.

  Returns 1 on success, 0 on failure, -1 if a general error occurred, or -2 if there was a usage error, although the return value is currently ignored.

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.

event_alloc
struct sudo_plugin_event * (*event_alloc)(void);

The **event_alloc**() function is used to allocate a struct sudo_plugin_event which provides access to the main **sudo** event loop. Unlike the other fields, the **event_alloc**() pointer is filled in by the **sudo** front-end, not by the plugin.

See the *Event API* section below for more information about events.

The **event_alloc**() function is only available starting with API version 1.17. If the **sudo** front-end doesn't support API version 1.17 or higher, **event_alloc**() will not be set.

Approval plugin API

```
struct approval_plugin {
#define SUDO_APPROVAL_PLUGIN 4
    unsigned int type; /* always SUDO_APPROVAL_PLUGIN */
    unsigned int version; /* always SUDO_API_VERSION */
    int (*open)(unsigned int version, sudo_conv_t conversation,
        sudo_printf_t sudo_printf, char * const settings[],
        char * const user_info[], int submit_optind,
        char * const submit_argv[], char * const submit_envp[],
        char * const plugin_options[], const char **errstr);
    void (*close)(void);
    int (*check)(char * const command_info[], char * const run_argv[],
        char * const run_envp[], const char **errstr);
    int (*show_version)(int verbose);
};
```

An approval plugin can be used to apply extra constraints after a command has been accepted by the policy plugin. Unlike the other plugin types, it does not remain open until the command completes. The plugin is opened before a call to **check**() or **show_version**() and closed shortly thereafter (audit plugin functions must be called before the plugin is closed). Multiple approval plugins may be specified in sudo.conf(5).

The approval_plugin struct has the following fields:

type The type field should always be set to SUDO_APPROVAL_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 sudo_printf, char * const settings[],
  char * const user_info[], int submit_optind,
  char * const submit_argv[], char * const submit_envp[],
  char * const plugin_options[], const char **errstr);
```

The approval **open**() function is run immediately before a call to the plugin's **check**() or **show_version**() functions. 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 can be used by the plugin to interact with the user (see *Conversation API* for details). 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 *Conversation API* for details). 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 options the user specified

when running **sudo**. As such, they will only be present when the corresponding option 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.

submit_optind
: The index into *submit_argv* that corresponds to the first entry that is not a command line option. If *submit_argv* only consists of options, which may be the case with the **-l** or **-v** options, submit_argv[submit_optind] will evaluate to the NULL pointer.

submit_argv
: The argument vector **sudo** was invoked with, including all command line options. The *submit_optind* argument can be used to determine the end of the command line options.

submit_envp
: The invoking user's environment in the form of a NULL-terminated vector of "name=value" strings.

When parsing *submit_envp*, 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.

errstr

If the **open**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

close

```
void (*close)(void);
```

The **close**() function is called after the approval plugin's **check**() or **show_version**() functions have been called. It takes no arguments. The **close**() function is typically used to perform plugin-specific cleanup, such as the freeing of memory objects allocated by the plugin. If the plugin does not need to perform any cleanup, **close**() may be set to the NULL pointer.

check

```
int (*check)(char * const command_info[], char * const run_argv[],
    char * const run_envp[], const char **errstr);
```

The approval **check**() function is run after the policy plugin **check_policy**() function and before any I/O logging plugins. If multiple approval plugins are loaded, they must all succeed for the command to be allowed. 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:

command_info

A vector of information describing the command being run in the form of "name=value" strings. The vector is terminated by a NULL pointer.

When parsing *command_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.

run_argv

A NULL-terminated argument vector describing a command that will be run in the same form as what would be passed to the execve(2) system call.

run_envp

The environment the command will be run with in the form of a NULL-terminated vector of "name=value" strings.

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

errstr

If the **open**() function returns a value other than 1, the plugin may store a message describing the failure or error in *errstr*. The **sudo** front-end will then pass this value to any registered audit plugins. The string stored in *errstr* must remain valid until the plugin's **close**() function is called.

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.

Returns 1 on success, 0 on failure, -1 if a general error occurred, or -2 if there was a usage error, although the return value is currently ignored.

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
- SIGPIPE
- 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.

An exception to this is SIGPIPE, which is ignored until the command is executed.

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 {
    unsigned int hook_version;
    unsigned 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 system 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 system implementation. For example, a setenv hook that operates on a private copy of the environment but leaves environ unchanged.

Care must be taken when hooking C library functions, it is very easy to create an infinite loop. 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_VERSION SUDO_API_MKVERSION(SUDO_HOOK_VERSION_MAJOR,\
                                SUDO_HOOK_VERSION_MINOR)
```

For getters and setters see the *Policy plugin API*.

Event API

When **sudo** runs a command, it uses an event loop to service signals and I/O. Events may be triggered based on time, a file or socket descriptor becoming ready, or due to receipt of a signal. Starting with API version 1.15, it is possible for a plugin to participate in this event loop by calling the **event_alloc**() function.

Event structure

Events are described by the following structure:

```
typedef void (*sudo_plugin_ev_callback_t)(int fd, int what, void *closure);

struct sudo_plugin_event {
    int (*set)(struct sudo_plugin_event *pev, int fd, int events,
        sudo_plugin_ev_callback_t callback, void *closure);
    int (*add)(struct sudo_plugin_event *pev, struct timespec *timeout);
    int (*del)(struct sudo_plugin_event *pev);
    int (*pending)(struct sudo_plugin_event *pev, int events,
        struct timespec *ts);
    int (*fd)(struct sudo_plugin_event *pev);
    void (*setbase)(struct sudo_plugin_event *pev, void *base);
    void (*loopbreak)(struct sudo_plugin_event *pev);
    void (*free)(struct sudo_plugin_event *pev);
};
```

The sudo_plugin_event struct contains the following function pointers:

set()

```
int (*set)(struct sudo_plugin_event *pev, int fd, int events,
    sudo_plugin_ev_callback_t callback, void *closure);
```

The **set**() function takes the following arguments:

struct sudo_plugin_event **pev*
: A pointer to the struct sudo_plugin_event itself.

fd
: The file or socket descriptor for I/O-based events or the signal number for signal events. For time-based events, *fd* must be -1.

events
: The following values determine what will trigger the event callback:

SUDO_PLUGIN_EV_TIMEOUT
callback is run after the specified timeout expires

SUDO_PLUGIN_EV_READ
callback is run when the file descriptor is readable

SUDO_PLUGIN_EV_WRITE
callback is run when the file descriptor is writable

SUDO_PLUGIN_EV_PERSIST
event is persistent and remains enabled until explicitly deleted

SUDO_PLUGIN_EV_SIGNAL
callback is run when the specified signal is received

The SUDO_PLUGIN_EV_PERSIST flag may be ORed with any of the event types. It is also possible to OR SUDO_PLUGIN_EV_READ and SUDO_PLUGIN_EV_WRITE together to run the callback when a descriptor is ready to be either read from or written to. All other event values are mutually exclusive.

sudo_plugin_ev_callback_t *callback*
typedef void (*sudo_plugin_ev_callback_t)(int fd, int what,
void *closure);

The function to call when an event is triggered. The **callback**() function is run with the following arguments:

fd The file or socket descriptor for I/O-based events or the signal number for signal events.

what The event type that triggered that callback. For events that have multiple event types (for example SUDO_PLUGIN_EV_READ and SUDO_PLUGIN_EV_WRITE) or have an associated timeout, *what* can be used to determine why the callback was run.

closure
The generic pointer that was specified in the **set**() function.

closure
A generic pointer that will be passed to the callback function.

The **set**() function returns 1 on success, and -1 if a error occurred.

add()

int (*add)(struct sudo_plugin_event *pev, struct timespec *timeout);

The **add**() function adds the event *pev* to **sudo**'s event loop. The event must have previously been initialized via the **set**() function. If the *timeout* argument is not NULL, it should specify a (relative) timeout after which the event will be triggered if the main event criteria has not been met. This is often used to implement an I/O timeout where the event will fire if a descriptor is not ready within a certain time period. If the event is already present in the event loop, its *timeout* will be adjusted to match the new value, if any.

The **add**() function returns 1 on success, and -1 if a error occurred.

del()

int (*del)(struct sudo_plugin_event *pev);

The **del**() function deletes the event *pev* from **sudo**'s event loop. Deleted events can be added back via the **add**() function.

The **del**() function returns 1 on success, and -1 if a error occurred.

pending()

int (*pending)(struct sudo_plugin_event *pev, int events,
struct timespec *ts);

The **pending**() function can be used to determine whether one or more events is pending. The *events* argument specifies which events to check for. See the **set**() function for a list of valid event types. If SUDO_PLUGIN_EV_TIMEOUT is specified in events, the event has an associated timeout and the *ts* pointer is non-NULL, it will be filled in with the remaining time.

fd()

int (*fd)(struct sudo_plugin_event *pev);

The **fd**() function returns the descriptor or signal number associated with the event *pev*.

setbase()

void (*setbase)(struct sudo_plugin_event *pev, void *base);

The **setbase**() function sets the underlying event *base* for *pev* to the specified value. This can be

used to move an event created via **event_alloc**() to a new event loop allocated by sudo's event subsystem. If *base* is NULL, *pev*'s event base is reset to the default value, which corresponds to **sudo**'s main event loop. Using this function requires linking the plugin with the sudo_util library. It is unlikely to be used outside of the **sudoers** plugin.

loopbreak()

void (*loopbreak)(struct sudo_plugin_event *pev);

The **loopbreak**() function causes **sudo**'s event loop to exit immediately and the running command to be terminated.

free()

void (*free)(struct sudo_plugin_event *pev);

The **free**() function deletes the event *pev* from the event loop and frees the memory associated with it.

Remote command execution

The **sudo** front-end does not support 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.

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.

Conversation function structures

The conversation function takes as arguments pointers to the following structures:

```
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 */
#define SUDO_CONV_PREFER_TTY       0x2000 /* flag: use tty if possible */
    int msg_type;
    int timeout;
    const char *msg;
};

#define SUDO_CONV_REPL_MAX      1023

struct sudo_conv_reply {
    char *reply;
};

typedef int (*sudo_conv_callback_fn_t)(int signo, void *closure);
struct sudo_conv_callback {
    unsigned int version;
    void *closure;
    sudo_conv_callback_fn_t on_suspend;
    sudo_conv_callback_fn_t on_resume;
};
```

Pointers to the **conversation**() and **printf**()-style functions are passed in to the plugin's **open**() function when the plugin is initialized. The following type definitions can be used in the declaration of the **open**() function:

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

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

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, that is, both arrays must have the same number of elements. Each struct sudo_conv_reply must have its *reply* member initialized to NULL. The struct sudo_conv_callback pointer, if not NULL, should contain function pointers to be called when the **sudo** process is suspended and/or resumed during conversation input. The *on_suspend* and *on_resume* functions are called with the signal that caused **sudo** to be suspended and the *closure* pointer from the struct sudo_conv_callback. These functions should return 0 on success and -1 on error. On error, the conversation will end and the conversation function will return a value of -1. The intended use is to allow the plugin to release resources, such as locks, that should not be held indefinitely while suspended and then reacquire them when the process is resumed. The functions are not actually invoked from within a signal handler.

The *msg_type* must be set to one of the following values:

SUDO_CONV_PROMPT_ECHO_OFF
: Prompt the user for input with echo disabled; this is generally used for passwords. The reply will be stored in the *replies* array, and it will never be NULL.

SUDO_CONV_PROMPT_ECHO_ON
: Prompt the user for input with echo enabled. The reply will be stored in the *replies* array, and it will never be NULL.

SUDO_CONV_ERROR_MSG
: Display an error message. The message is written to the standard error unless the SUDO_CONV_PREFER_TTY flag is set, in which case it is written to the user’s terminal if possible.

SUDO_CONV_INFO_MSG
: Display a message. The message is written to the standard output unless the SUDO_CONV_PREFER_TTY flag is set, in which case it is written to the user’s terminal if possible.

SUDO_CONV_PROMPT_MASK
: Prompt the user for input but echo an asterisk character for each character read. The reply will be stored in the *replies* array, and it will never be NULL. This can be used to provide visual feedback to the user while reading sensitive information that should not be displayed.

In addition to the above values, the following flag bits may also be set:

SUDO_CONV_PROMPT_ECHO_OK
: Allow input to be read when echo cannot be disabled when the message type is SUDO_CONV_PROMPT_ECHO_OFF or SUDO_CONV_PROMPT_MASK. By default, **sudo** will refuse to read input if the echo cannot be disabled for those message types.

SUDO_CONV_PREFER_TTY
: When displaying a message via SUDO_CONV_ERROR_MSG or SUDO_CONV_INFO_MSG, try to write the message to the user's terminal. If the terminal is unavailable, the standard error or standard output will be used, depending upon whether SUDO_CONV_ERROR_MSG or SUDO_CONV_INFO_MSG was used. The user's terminal is always used when possible for input, this flag is only used for output.

The *timeout* in seconds until the prompt will wait for no more input. A zero value implies an infinite timeout.

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.

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.

Plugin invocation order

As of **sudo** 1.9.0, the plugin **open**() and **close**() functions are called in the following order:

1. audit open
2. policy open
3. approval open
4. approval close
5. I/O log open

6. command runs

7. command exits

8. I/O log close

9. policy close

10. audit close

11. sudo exits

Prior to **sudo** 1.9.0, the I/O log **close**() function was called *after* the policy **close**() function.

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)
```

For getters and setters see the *Policy plugin API*.

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.

Version 1.8 (sudo 1.8.15)

The *sudoedit_checkdir* and *sudoedit_follow* entries were added to the command_info list. The default value of *sudoedit_checkdir* was changed to true in sudo 1.8.16.

The sudo *conversation* function now takes a pointer to a struct sudo_conv_callback as its fourth argument. The sudo_conv_t definition has been updated to match. The plugin must specify that it supports plugin API version 1.8 or higher to receive a conversation function pointer that supports this argument.

Version 1.9 (sudo 1.8.16)

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

Version 1.10 (sudo 1.8.19)

The *umask* entry was added to the user_info list. The *iolog_group*, *iolog_mode*, and *iolog_user*

entries were added to the command_info list.

Version 1.11 (sudo 1.8.20)
The *timeout* entry was added to the settings list.

Version 1.12 (sudo 1.8.21)
The change_winsize field was added to the io_plugin struct.

Version 1.13 (sudo 1.8.26)
The log_suspend field was added to the io_plugin struct.

Version 1.14 (sudo 1.8.29)
The *umask_override* entry was added to the command_info list.

Version 1.15 (sudo 1.9.0)
The *cwd_optional* entry was added to the command_info list.

The *event_alloc* field was added to the policy_plugin and io_plugin structs.

The *errstr* argument was added to the policy and I/O plugin functions which the plugin function can use to return an error string. This string may be used by the audit plugin to report failure or error conditions set by the other plugins.

The **close**() function is now is called regardless of whether or not a command was actually executed. This makes it possible for plugins to perform cleanup even when a command was not run.

SUDO_CONV_REPL_MAX has increased from 255 to 1023 bytes.

Support for audit and approval plugins was added.

Version 1.16 (sudo 1.9.3)
Initial resource limit values were added to the user_info list.

The *cmnd_chroot* and *cmnd_cwd* enties were added to the settings list.

Version 1.17 (sudo 1.9.4)
The *event_alloc* field was added to the audit_plugin and approval_plugin structs.

Version 1.18 (sudo 1.9.9)

The policy may now set resource limit values in the command_info list.

SEE ALSO

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

AUTHORS

Many people have worked on **sudo** over the years; this version consists of code written primarily by:

Todd C. Miller

See the CONTRIBUTORS.md file in the **sudo** distribution (https://www.sudo.ws/about/contributors/) for an exhaustive list of people who have contributed to **sudo**.

BUGS

If you believe you have found a bug in **sudo**, you can 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