bash, howto, redirection, sudo, tee

Confusion With File Redirection And sudo

Local copy of:
http://giedrius.blog/2017/10/31/confusion-with-file-redirection-and-sudo/

Introduction

I have noticed that newbies sometimes fail to understand why, for example, `sudo echo “hi” > /tmp/test` will create a file */tmp/test* with whatever effective *user ID* and effective *group ID* the command is ran with. They fall into the trap thinking that *root:root* will be the owner of */tmp/test*. However, that is certainly not true. This blog post will try to clarify why this happens and what the user can do to avoid this issue. This occurs due to peculiar parsing rules defined in POSIX. We will examine the standards which detail this behaviour, the source code of the popular shells **bash/zsh**, and some methods on how to avoid this problem with, for example, tee(1).

Why it happens

You, my dear reader, have to understand first of all that *sudo* is just a command, a binary just like any else. It is a special binary, though, because it runs whatever command you pass as arguments. However, the redirection part (> /tmp/test) is **not** part of the command. This special file redirection syntax is interpreted by the shell, not passed to *sudo* for it to be executed later. So the shell that you are running gets special instructions to run that command with *file descriptor number 1* which will be opened to write to a newly created file */tmp/test*.

Notice that at this point command has **not** been run yet. The shell, before starting the command, begins to prepare the first fd. The shell does this with whatever *EUID/EGID* that shell is running with so it creates that new file with not the root:root rights but with whatever *EUID/EGID* the shell is running with. Most shells probably prepare the *file descriptor 1* with the dup*(2) family of functions or fcntl(2), and open(2) before doing a fork(2) and exec*(3) just after it.

This functionality is mandated by a section of POSIX: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_07. All POSIX-compatible shells follow that section. Obviously, some shells are not compatible with POSIX. This article targets POSIX-compatible shells, however. In either case, to find out all of the gory details of how file redirection works, either refer to that section, read the fine manual, and/or read the source code of the shell that you are using. The latter option gives you the most detail and is the most trustworthy whereas the former one is more accessible. I will not go on a tangent here and I will redirect you to this article that I wrote before if you want to find more about this.

Next we will review the code of two popular shells: GNU/Bash and the Z shell, to see how they realize this functionality in code.

How do popular shells implement this

GNU/Bash uses the GNU/Bison project for specifying the input command syntax. The syntax file is written in the yacc format. You can find the special syntax described in the *parse.y* file, in the root directory of GNU/Bash source. There are two special types defined: redirection and redirection_list. Here is an excerpt from *parse.y* file which defines how redirecting **stdout** to a file works or how passing a file to **stdin** works:

```
redirection:    '>' WORD
            {
              source.dest = 1;
              redir.filename = $2;
              $$ = make_redirection (source, r_output_direction, redir, 0);
            }
        |       '<' WORD
            {
              source.dest = 0;
              redir.filename = $2;
              $$ = make_redirection (source, r_input_direction, redir, 0);
            }
        | ...
```

And here is the excerpt which shows that the redirection list goes after a command:

```
command:    simple_command
            { $$ = clean_simple_command ($1); }
    |    shell_command
            { $$ = $1; }
    |    shell_command redirection_list
            {
              ...
            }
    |    ...
```

My point is that you can see that GNU/Bash really has a special syntax for file redirection, it is **not** passed to sudo. The shell command is separate from the redirection list. With the Z shell it is a bit different because there is no one single place where the grammar is defined. Instead, it has a big file dedicated for the hand-rolled parsing engine that is written in C. Thus, it will not be possible to present the whole file but here are the excerpts from the file which prove to you once again that the file redirection feature is indeed based on a special syntax, it is not passed to the actual binary as arguments. In the Z shell source code, *Src/parse.c* we can find this comment:

```

/*
 * cmd : { redir } ( for | case | if | while | repeat |
 * subsh | funcdef | time | dinbrack | dinpar | simple ) { redir }
 *
 * zsh_construct is passed through to par_subsh(), q.v.
 */
```

Indeed, redirection is parsed later in the function that is responsible for parsing the command:

```

static int
par_cmd(int *cmplx, int zsh_construct)
{
    int r, nr = 0;

    r = ecused;
    if (IS_REDIROP(tok)) {
        *cmplx = 1;
        while (IS_REDIROP(tok)) {
            nr += par_redir(&r, NULL);
        }
    }

    switch (tok) {
    case FOR:
          ...
    case FOREACH:
          ...
    case SELECT:
          ...
    case CASE:
          ...
    case IF:
          ...
    case WHILE:
          ...
    case UNTIL:
          ...
    case REPEAT:
          ...
    ...
    }

    if (IS_REDIROP(tok)) {
        *cmplx = 1;
        while (IS_REDIROP(tok))
            (void)par_redir(&r, NULL);
    }

    ...
    return 1;
}
```

As you can see, the redirections may be at either the end or the beginning of the command. Let me repeat again: file redirection is **special syntax** and it is not passed to the actual thing being run. The GNU/Bash shell supports the file redirection syntax at the beginning of the line as well but I have just decided to not include it for brevity.

So how to actually redirect output to a file as root?

I guess the most popular solution to this is to simply use a pipe to redirect output to a file. I am not sure but probably the program, tee(1), was made for this purpose. Or it was made as an extension of the tee system call but still it is the perfect tool to solve our issue. So, the solution to this problem would be:

```
echo "hi" | sudo tee /tmp/test >&-
```

This will at first (probably, depending on your set up) ask for your password. Then, "hi" will be written to a file descriptor which will get passed on to tee(1). Then, tee(1) will dump everything into */tmp/test*. The `>&- (or >/dev/null)` is needed so that tee(1) would not output anything. tee(1), unfortunately, copies the content from standard input to standard output as well.

Another way to solve this is to pass more commands to run via root either by using the **-s** *sudo* option. For example, this works:

```
sudo -s <<EOF
exec >/tmp/test
echo "hi"
EOF
```

Or, run another shell using *sudo* which will redirect everything to */tmp/test*:

```
sudo /bin/bash -c 'exec >/tmp/test; echo "hi"'
```

Or, run a script with *sudo* which will internally redirect **stdout** to */tmp/test*:

```
cat >./script.sh <<EOF
exec >/tmp/test
echo "hi"
EOF
chmod +x ./script.sh
sudo ./script.sh
```

Obviously, there are more (complex) ways how you could achieve this but these are the main methods. You are free to adopt these examples to your own case. Please comment if you find any errors or if you want to add anything about this topic.

~~DISCUSSION~~