VSys: A Privilege Allocation Tool

Sapan Bhatia
Princeton University

September 16, 2008

Abstract

Privilege allocation is an essential feature of Operating Systems that prevents users from abusing system resources and from bypassing admission control. There are several methods to implement such allocation: using programs such as *sudo* and *Proper*; through OS virtualization; etc. Vsys provides yet another means to this end and can function in conjunction with existing tools. Using Vsys, an administrator can deploy scripts that arbitrate access to specific system resources and data based on the requirements of one or more users. Users access these scripts (and hence the resources they arbitrate) through FIFO pipes. The advantages of using Vsys include the ability to develop scripts in an arbitrary programming language, the ability to deploy scripts dynamically and the ability to restrict access at a fine grain, e.g. giving a user partial access to a file. Furthermore, Vsys scripts can be accessed through simple UNIX tools such as cat, echo and grep. This document describes the design and implementation of VSys and its use on and off PlanetLab.

1 Using Vsys

This Section provides a brief but comprehensive summary of how to deploy and use Vsys. Your interaction with Vsys depends on your role in a system: whether you are a user, an administrator or a script developer. These roles and the corresponding usage of Vsys are described below.

1.1 Users

A user is an individual who would like to access a restricted resource (e.g. RAW sockets, privileged TCP and UDP ports, restricted files, restricted memory etc.). If you are a user, then you need not bother yourself with the

installation and configuration of Vsys, or the development of Vsys scripts. You should get the specification of your Vsys setup from your administrator. It will typically encapsulate a *VSys frontend*, which is a directory; and a set of Vsys entries, which appear as FIFO pipes in the VSys frontend. Here's an example.

```
bash-3.2# ls /vsys/
local_dcookie.in  local_dcookie.out  local_login.in  local_login.out  pfmount.in
pfmount.out
bash-3.2#
```

In this example, we see a VSys entry named *local_login* that in this example gives a user access to a restricted file, `/var/log/secure`. This file contains logs of the user's `ssh` sessions with the system. The script gives a user partial access to the file, making available the set of IP addresses from which the user has logged in in the past while hiding other information including logs pertaining to other users on the system.

Each Vsys entry consists of two items: an input pipe and an output pipe. All data written to the input pipe (in this case, `local_login.in`) is sent to the corresponding VSys script as input, and the output of a script can be read via the output pipe (in this case, `local_login.out`). A user can perform both the read and the write operations at the command line. For example:

```
bash-3.2# cat /vsys/local_login.out &
[1] 9426
bash-3.2# echo 1 > /vsys/local_login.in
Sep 14 04:05:17 planetlab-7 sshd[17271]: Accepted publickey for
pl_netflow from 128.112.139.45 port 37085 ssh2
Sep 14 04:53:03 planetlab-7 sshd[5952]: Postponed publickey for
pl_netflow from 128.112.139.45 port 33324 ssh2
Sep 14 04:53:03 planetlab-7 sshd[5951]: Accepted publickey for
pl_netflow from 128.112.139.45 port 33324 ssh2
Sep 14 05:41:43 planetlab-7 sshd[28020]: Postponed publickey for
pl_netflow from 128.112.139.45 port 39399 ssh2
```

Here, we redirect the output pipe to the terminal (thereby reading the output of the script) and write into the input pipe using the `echo` command. Note that these operations must be carried out in this order for the output of the pipe to appear in time for the user to read it. A Vsys script is launched as soon as it is sent input. Any output issued between this time and the time at which the output pipe is opened for reading will be lost (though it will be logged). If the output pipe is not open while a script is active, its output is optionally logged in the Vsys log file.

An alternative method of accessing a vsys script is to use `vsyssh`, as follows. We specify the vsys entry to connect to as an argument to `vsyssh`.

All subsequent input from the user is sent to the script, and output from the script displayed on the screen.

```
bash-3.2# vsyssh /vsys/local_login
vsyssh>1
Sep 14 04:05:17 planetlab-7 sshd[17271]: Accepted publickey for pl_netflow from
128.112.139.45 port 37085 ssh2
Sep 14 04:53:03 planetlab-7 sshd[5952]: Postponed publickey for pl_netflow from
128.112.139.45 port 33324 ssh2
Sep 14 04:53:03 planetlab-7 sshd[5951]: Accepted publickey for pl_netflow from
128.112.139.45 port 33324 ssh2
Sep 14 05:41:43 planetlab-7 sshd[28020]: Postponed publickey for pl_netflow from
128.112.139.45 port 39399 ssh2
vsyssh>
```

1.2 Administrators

If you are an administrator, then you are set with the responsible for installing and maintaining Vsys and associated scripts. You can install Vsys from its source code [**?**] or from a pre-built binary package [**?**]. If you use the source code, then decompress it and run the following commands as root:

```
make
make install
```

The installation might fail because of missing dependencies. If so, make sure you have the following packages:

```
ocaml
ocaml-doc
inotify-tools
inotify-tools-devel
kernel-headers
```

Next, edit the configuration file for vsys: `vsys.conf` to declare the set of users that are to be given access to vsys.

```
/vservers/pl_netflow/vsys pl_netflow
/vservers/princeton_slicestat/vsys princeton_slicestat
/vservers/princeton_comon/vsys princeton_comon
/vservers/imperial_worm/vsys imperial_worm
```

Each line in the configuration file declares a user (e.g. pl_netflow) and a directory to which the user has access, in which the vsys frontend will be populated. Although vsys will create the the frontend directory if it does not exist, it is recommended that it be created separately independent of vsys.

Vsys can be invoked in two ways: through its initscript, and directly by executing its binary. Have a look at its initscript to see the options available. The main configuration option is the backend, which is the location in which you drop vsys scripts and Access Control Lists (ACL) files, so that they become available to users.

Here is an example of executing vsys:

```
bash-3.2# /usr/bin/vsys -failsafe -backend /vsys -conffile /etc/vsys.conf -daemon
```

The `failsafe` option instructs vsys to never crash. This feature was designed for PlanetLab, on which many critical services require vsys to function. With this option, vsys will try to recover, however serious the error might be. The *backend* specifies the location at which scripts are dropped (explained below).

Once vsys has been started, the next step is to install the vsys scripts that have been developed for your environment. You may find common vsys scripts on the vsys homepage or elsewhere. Please review the next section for instructions on how vsys scripts are developed. Deploying a new vsys script happens in two steps:

1. First, an ACL file is created in the backend directory to specify the set of users that have access to the script. The name of this ACL file must be in the form <name>.acl, where name is the name of the script to be deployed.

   ```
   bash-3.2# cat > /vsys/newscript.acl
   pl_netflow
   some_slice
   ```

   Here, the users *pl_netflow* and *some_slice* are being given access to the script, newscript, which is to be installed. Next, we actually install the script:

   ```
   bash-3.2# cp <location of newscript> /vsys/newscript
   bash-3.2# ls /vservers/pl_netflow/vsys
   newscript.in    newscript.out
   ```

   Thus, the newly deployed script has become available in the frontend of the user. Vsys reacts dynamically to such events. To remove a script, you can simply delete the script from the backend. Vsys will remove the corresponding entries from the front ends automatically. Similarly, you can organize the scripts in a directory heirarchy.

   ```
   bash-3.2# mkdir /vsys/common
   bash-3.2# ls /vservers/pl_netflow/vsys
   common/    newscript.in    newscript.out
   bash-3.2#
   ```

If you are administrating a deployment of PlanetLab (MyPLC), then the process is somewhat different. In particular:

- Vsys will come pre-installed as part of the node software, and with a set of common scripts. You might want to check that the latest version of vsys is being used. The latest version can be found at the vsys homepage [**?**].

- The /etc/vsys.conf file will be set up and populated by the Node Manager, based on a set of *vsys slice attributes*. More information on this process can be found in the Node Manager documentation. The Node Manager will also create and manage all required ACLs. A slice attribute can be specified in the PLC gui. Select "Add a slice attribute" for the slice that you would like to be able to access vsys. Set the name of the new attribute to 'vsus' and the script to the vsys script.

- The frontend and backend are already defined. The frontend can be found in the /vsys directory of each slice.

  When using vsys on PlanetLab, all scripts are managed by the Node Manager, except for scripts prefixed with the string *local_*. These scripts are meant for local installation and can be used to give users temporary access to a resource.

1.3 Script authors

A script author develops vsys scripts. You might want to introduce a new facility on the system for your users (e.g. the ability to bind to a particular privileged port), or to develop a script for your own use if you are a slice user on PlanetLab. In the latter case, naturally, your script will be reviewed by the administrators of the PlanetLab deployment in question.

Here is the source code of the `login` script mentioned previously.

```
\#!/bin/sh
SLICENAME=$1
grep SLICENAME /var/log/secure
```

Vsys scripts receive as their first argument the name of the client slice. In this case, the script outputs the lines in the file /var/log/secure that contain the name of the slice, indicating login events involving the slice user. This output is sent to the slice via the corresponding output entry (login.out).

Input is received via STDIN. Consider the following script, written in C, for fetching pathnames based on an in-kernel tag called a *directory cookie*, used to temporarily save a filename and to restore it afterwards.

```
int main(int argc,char *argv[]) {
        char path_buf[16384],dcookie_buf[sizeof(INT64_MAXSZ)];
        path_buf[0]='\0';
        while (fgets(dcookie_buf, sizeof(dcookie_buf),stdin)) {
                if (lookup_dcookie(atoll(dcookie_buf), path_buf, sizeof(path_buf))>0) {
                        printf(``%s\n'',path_buf);
                }
                else {
                        printf(``% Not found\n'');
                }
        }
}
```

The script reads the values of the cookies via standard input, looks up the corresponding pathnames and sends them to the slice via standard output.

VSys scripts can be of two types: stateless and stateful. The example scripts defined thus far are stateless, in that they are instantiated anew for each request and do not retain any state between invocations. Scripts can also be persistent if need be. Writing a persistent script is easy – run in a loop and do not terminate. Instead, go to sleep, waiting for subsequent input. For such scripts, VSys keeps track of the pid of the first instance of the script and reuses this instance for subsequent interactions. Stateful scripts need to be prefixed with the string *daem_* to indicate to Vsys that they are stateful. Here is an example of a stateful script ("daem_counter"):

```
\#!/usr/bin/perl

use strict;

my $counter=0;

while (<>) {
        $counter++;
        print ``Current value of counter: $counter\n'';
}
```

In order to deploy such scripts, please make sure that the `PERSISTENT` flag is set in the Makefile at compilation time.

2 Programmatic invocation of Vsys scripts

Invoking Vsys programmatically is more complex than invoking it at the command line. This is because of the blocking semantics of Vsys. While most programmers prefer interacting with files in blocking mode, Vsys requires that entries be opened in non-blocking mode. Once the entries have been opened, there are two ways to proceed. The first is to continue to use them in non-blocking mode and to use event-driven I/O to transfer data.

If this method is too laborious to implement, then the second approach is to switch the entries back into blocking mode. The following example demonstrates how the latter is implemented:

```
FILE *fp = NULL, *fp_in = NULL;
FILE *out_fp = NULL, *diff_fp = NULL;
const char* top_out_file = "fe/test.out";
const char* top_in_file = "fe/test.in";
int fd_in = -1, fd_out;
int res;
int flag,flag2;
int count = 1;
struct timeval tv={.tv_sec=5,.tv_usec=0};
fd_set readSet;
int res;
int nlines=0;
if ((fd_out = open(top_out_file, O_RDONLY | O_NONBLOCK)) < 0) {
        fprintf(stderr, "error opening entry\n");
    exit(-1);
}
while ((fd_in = open(top_in_file, O_WRONLY| O_NONBLOCK)) < 0) {
    fprintf(stderr, "Waiting for %s (%s)\n", top_in_file,strerror(errno));
        usleep (50);
}
if ((flag = fcntl(fd_out, F_GETFL)) == -1) {
    printf("fcntl get failed\n");
    exit(-1);
}
if ((flag2 = fcntl(fd_in, F_GETFL)) == -1) {
    printf("fcntl get failed\n");
    exit(-1);
}
FD_ZERO(&readSet);
FD_SET(fd_out, &readSet);
/* We are waiting for the fd to become available */
res = select(fd_out + 1, &readSet, NULL, NULL, NULL);
fprintf(stderr,"select failed errno=%d errstr=%s\n", errno, strerror(errno));
exit(-1);
if (fcntl(fd_out, F_SETFL, flag & ~O_NONBLOCK) == -1) {
        printf("fcntl set failed\n");
    exit(-1);
}
if ((flag = fcntl(fd_out, F_GETFL)) == -1) {
    printf("fcntl get failed\n");
    exit(-1);
}
if (fcntl(fd_in, F_SETFL, flag2 & ~O_NONBLOCK) == -1) {
    printf("fcntl set failed\n");
    exit(-1);
}
if ((flag2 = fcntl(fd_in, F_GETFL)) == -1) {
    printf("fcntl get failed\n");
    exit(-1);
}
if (flag & O_NONBLOCK == 0) {
```

```
    printf("fd_out still nonblocking\n");
    exit(-1);
}
if (flag & O_NONBLOCK == 0) {
    printf("fd_in still nonblocking\n");
    exit(-1);
}
if ((fp = fdopen(fd_out, "r")) == NULL) {
    printf("fdopen failed\n");
    exit(-1);
}
while (fgets(buf, sizeof(buf), fp) != NULL) {
        nlines++;
}
fclose(fp);
close(fd_in);
close(fd_out);
```

As you can see, the entries are opened first in NON-BLOCKING mode, and then switched over to BLOCKING mode so that data can be read and written conveniently.

3 Design and implementation

Vsys is written as an