How to Use Strace - A Linux Debugging Utility

Strace is a Linux Utility which lists all the system calls and any signals, of any executable running on Linux Operating System. Strace generally comes along with the Linux installation.

If one notices, its name "strace" has come from system-tracing i.e. tracing the system calls and signals. One can trace system calls while a process is being running, and also by running the executable using strace. Hence, this utility is a really helpful in debugging linux kernel space programs as system calls are traced as they are called with their parameters and return values, and even errors. We shall be discussing more on debugging in the forthcoming sections of this article.

Here is an usage example:
Lets try to trace the system calls for the most common command in Linux - 'ls'

Code:

```
rupali@ubuntux:~/aprograms$ strace ls test.txt
execve("/bin/ls", ["ls", "test.txt"], [/* 36 vars */]) = 0
brk(0)                                  = 0x96c9000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb772a000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=73332, ...}) = 0
mmap2(NULL, 73332, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7718000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
open("/lib/i386-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0@A\0\0004\0\0\0"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
statfs64("/selinux", 84, {f_type="EXT2_SUPER_MAGIC", f_bsize=4096,
f_blocks=64713465, f_bfree=60337807, f_bavail=57098179, f_files=16203776,
f_ffree=15911537, f_fsid={-252760809, 122935572}, f_namelen=255,
f_frsize=4096}) = 0
brk(0)                                  = 0x96c9000
close(3)                                = 0
ioctl(1, SNDCTL_TMR_TIMEBASE or TCGETS, {B38400 opost isig icanon echo ...})
= 0
```

```
ioctl(1, TIOCGWINSZ, {ws_row=24, ws_col=80, ws_xpixel=0, ws_ypixel=0}) = 0
stat64("test.txt", {st_mode=S_IFREG|0664, st_size=18, ...}) = 0
lstat64("test.txt", {st_mode=S_IFREG|0664, st_size=18, ...}) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7728000
write(1, "test.txt\n", 9test.txt
)               = 9
close(1)                                = 0
munmap(0xb7728000, 4096)                = 0
close(2)                                = 0
exit_group(0)                           = ?
```

Note: This is a truncated snippet of output shown above.

One can see, there are hell lots of system calls for a simple list command. The kernel is doing so many opens, reads, writes and mmaps. We will not get into the details of understanding all these system calls as to retain the focus of the article on the 'strace' utility.

Using STRACE

For running strace to debug your executable, here is how we run

Code:

```
strace ./executable-name <arguments-if-any>
```

For already running process, the way we use strace is

Code:

```
strace -p <PID>
```

Strace provides various options to know what time each system call took, any errors, printing instruction pointers, etc

To know more on various available options and usage front, you can refer to the man page of strace. Type

Code:

```
 man strace
```

or go to Strace Man Page

Debugging with STRACE

Strace can leverage in debugging Linux applications in a great deal in tracing which system call failed and what signals were received by the kernel during the execution. It is a also helpful in scenarios when any process has hanged or in a freeze state, we can check what is actually going at a particular point of time.

To get a hands-on, lets take an example error-ed program and debug it with the help of strace.

The Program:

Code:

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    char str[30];
    FILE *fp = fopen("any.txt", "r");
    fgets(str, 5, fp);
    fclose(fp);

    return 0;
}
```

Now in the above code, we are trying to a open a file in read-only mode. However, this file does not exist on the disk. Hence, there should be an error.

Lets build and run it:

Code:

```
rupali@ubuntu:~/programs/strace$ gcc -Wall st.c -o st
rupali@ubuntu:~/programs/strace$ ./st
Segmentation fault (core dumped)
```

There it is as we expected - a segmentation fault for opening a file which does not exist. However, lets see what strace has to say about it.

Code:

```
rupali@ubuntu:~/programs/strace$strace ./ferr
```

Here is the output of strace, which can be analysed to determine the problem.

Code:

```
'execve("./st", ["./st"], [/* 51 vars */]) = 0
brk(0)                                  = 0x8a28000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7861000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=77018, ...}) = 0
mmap2(NULL, 77018, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb784e000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or
directory)
open("/lib/i386-linux-gnu/libc.so.6", O_RDONLY) = 3
read(3,
"\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\220o\1\0004\0\0\0"..., 512)
= 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1442372, ...}) = 0
mmap2(NULL, 1448456, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0xb76ec000
mmap2(0xb7848000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x15c) = 0xb7848000
mmap2(0xb784b000, 10760, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb784b000
close(3)                                = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb76eb000
set_thread_area({entry_number:-1 -> 6, base_addr:0xb76eb8d0, limit:1048575,
seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1,
seg_not_present:0, useable:1}) = 0
mprotect(0xb7848000, 8192, PROT_READ)   = 0
mprotect(0x8049000, 4096, PROT_READ)    = 0
mprotect(0xb7880000, 4096, PROT_READ)   = 0
munmap(0xb784e000, 77018)               = 0
brk(0)                                  = 0x8a28000
brk(0x8a49000)                          = 0x8a49000
open("any.txt", O_RDONLY)               = -1 ENOENT (No such file or
directory)
--- SIGSEGV (Segmentation fault) @ 0 (0) ---
+++ killed by SIGSEGV (core dumped) +++
Segmentation fault (core dumped)
```

For a simple program, there are lots of system call records, however we need to focus on the segmentation fault i.e. the SIGSEGV signal received as one can see in the strace output at the end.

What happened one call above is an open() call for "any.txt" in read only mode. Look at what open() returned. Its a negative value with error message "No such file or directory". This is what exactly happened, right? System calls return negative value on failures. Through strace, we were able to understand, what failure happened, why and what signal caused the crash.

Suppose this code was just a part of a magnificent source code base. Strace would had been our first and foremost tool to identify the bug.

Conclusion

So, now we know an interesting, easy to use tool to debug our programs. Although yes, GDB is the next step debugging tool helpful for complicated debugging with much intensive functionalities. However, strace can be used in various cases where there are segmentation faults or some file operation errors.

Strace is very helpful specially for kernel modules programming and system programming. At times, knowing how things work at system level does help debugging issues at upper levels. Therefore, strace is highly recommended tool by Linux.

Source: http://www.go4expert.com/articles/strace-linux-debugging-utility-t29091/