Outline

1. Getting started  3
2. Tracing child processes  10
3. Filtering strace output  14
4. System call tampering  20
5. Further strace options  24
1 Getting started
2 Tracing child processes
3 Filtering strace output
4 System call tampering
5 Further strace options
strace(1)

- A tool to trace system calls made by a user-space process
  - Implemented via ptrace(2)
- Or: a debugging tool for tracing complete conversation between application and kernel
  - Application source code is not required
- Answer questions like:
  - What system calls are employed by application?
  - Which files does application touch?
  - What arguments are being passed to each system call?
  - Which system calls are failing, and why (errno)?
- There is also a loosely related ltrace(1) command
  - Trace library function calls in dynamic shared objects (e.g., libc)
  - We won’t cover this tool

©2018, Michael Kerrisk System Call Tracing with strace 4 §1.1
Log information is provided in **symbolic form**
- **System call names** are shown
- We see **signal names** (not numbers)
- **Strings** printed as characters (up to 32 bytes, by default)
- **Bit-mask arguments displayed symbolically**, using corresponding bit flag names ORed together
- **Structures** displayed with **labeled fields**
- **errno values** displayed symbolically + matching error text
- “large” arguments and structures are abbreviated by default

```
fstat(3, {st_dev=makedev(8, 2), st_ino=401567, 
  st_mode=S_IFREG|0755, st_nlink=1, st_uid=0, st_gid=0, 
  st_blksize=4096, st_blocks=280, st_size=142136, 
  st_atime=2015/02/17-17:17:25, st_mtime=2013/12/27-22:19:58, 
  st_ctime=2014/04/07-21:44:17}) = 0

open("/lib64/liblzma.so.5", O_RDONLY|O_CLOEXEC) = 3
```
Simple usage: tracing a command at the command line

- A very simple C program:
  ```c
  int main(int argc, char *argv[]) {
    #define STR "Hello world\n"
    write(STDOUT_FILENO, STR, strlen(STR));
    exit(EXIT_SUCCESS);
  }
  ```

- Run `strace(1)`, directing logging output (`-o`) to a file:
  ```bash
  $ strace -o strace.log ./hello_world
  Hello world
  ```
  (By default, trace output goes to standard error)

-⚠️ On some systems, may first need to:
  ```bash
  # echo 0 > /proc/sys/kernel/yama/ptrace_scope
  ```
  Yama LSM disables `ptrace(2)` to prevent attack escalation; see man page
Simple usage: tracing a command at the command line

$ cat strace.log
execve("./hello_world", ["./hello_world"], [/* 110 vars */]) = 0
... access("/etc/ld.so.preload", R_OK) = -1 ENOENT
(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY | O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=160311, ...}) = 0
mmap(NULL, 160311, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fa5ecfc0000
close(3) = 0
open("/lib64/libc.so.6", O_RDONLY | O_CLOEXEC) = 3
... write(1, "Hello world\n", 12) = 12
exit_group(0) = ?
+++ exited with 0 +++

- Even simple programs make lots of system calls!
  - 25 in this case (many have been edited from above output)
- Most output in this trace relates to finding and loading shared libraries
  - First call (execve()) was used by shell to load our program
  - Only last two system calls were made by our program
Simple usage: tracing a command at the command line

```
$ cat strace.log
execve("./hello_world", ["./hello_world"], [/* 110 vars */]) = 0
...  
access("/etc/ld.so.preload", R_OK) = -1 ENOENT  
(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY | O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=160311, ...}) = 0
mmap(NULL, 160311, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fa5ecfc0000
close(3) = 0
open("/lib64/libc.so.6", O_RDONLY | O_CLOEXEC) = 3
...  
write(1, "Hello world\n", 12) = 12
exit_group(0) = ?
+++ exited with 0 +++
```

For each system call, we see:

- Name of system call
- Values passed in/returned via arguments
- System call return value
- Symbolic `errno` value (+ explanatory text) on syscall failures
A gotcha...

The last call in our program was:

```c
exit(EXIT_SUCCESS);
```

But `strace` showed us:

```c
exit_group(0) = ?
```

Some detective work:
- We “know” `exit(3)` is a library function that calls `_exit(2)`
- But where did `exit_group()` come from?
- `_exit(2)` man page tells us:

```
$ man 2 _exit
...
C library/kernel differences
In glibc up to version 2.3, the _exit() wrapper function invoked the kernel system call of the same name. Since glibc 2.3, the wrapper function invokes exit_group(2), in order to terminate all of the threads in a process.
```

⇒ may need to dig deeper to understand `strace(1)` output
Outline

1  Getting started  
2  Tracing child processes  
3  Filtering strace output  
4  System call tampering  
5  Further strace options
Tracing child processes

- By default, `strace` does not trace children of traced process
- `-f` option causes children to be traced
  - Each trace line is prefixed by PID
  - In a program that employs POSIX threads, each line shows
    kernel thread ID (`gettid()`)

©2018, Michael Kerrisk System Call Tracing with strace 11 §1.2
int main(int argc, char *argv[]) {
    pid_t childPid;
    char *newEnv[] = {"ONE=1", "TWO=2", NULL};

    printf("PID of parent: %ld\n", (long) getpid());
    childPid = fork();
    if (childPid == 0) {     /* Child */
        printf("PID of child: %ld\n", (long) getpid());
        if (argc > 1) {
            execve(argv[1], &argv[1], newEnv);
            errExit("execve");
        }
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_SUCCESS);
}

wait(NULL);     /* Parent waits for child */
exit(EXIT_SUCCESS);

$ strace -f -o strace.log ./fork_exec
PID of parent: 1939
PID of child: 1940
Tracing child processes: strace/fork_exec.c

Each line of trace output is prefixed with corresponding PID

- Inside glibc, `fork()` is actually a wrapper that calls `clone(2)`
- `wait()` is a wrapper that calls `wait4(2)`
- We see two lines of output for `wait4()` because call blocks and then resumes
- `strace` shows us that parent received a `SIGCHLD` signal
Outline

1 Getting started 3
2 Tracing child processes 10
3 Filtering strace output 14
4 System call tampering 20
5 Further strace options 24
Selecting system calls to be traced

- `strace -e` can be used to select system calls to be traced
  - `-e trace=<syscall>[,<syscall>]...`
    - Specify system call(s) that should be traced
    - Other system calls are ignored

  ```
  $ strace -o strace.log -e trace=open,close ls
  ```

- `-e trace=!<syscall>[,<syscall>]...`
  - **Exclude** specified system call(s) from tracing
    - Some applications do bizarre things (e.g., calling `gettimeofday()` 1000s of times/sec.)
    - △ “!” needs to be quoted to avoid shell interpretation

- `-e trace=/<regexp>`
  - Trace syscalls whose names match regular expression
    - April 2017; expression will probably need to be quoted...
Selecting system calls by category

- `–e trace=<syscall-category>` trace a category of syscalls

Categories include:
- `%file`: trace all syscalls that take a filename as argument
  - `open()`, `stat()`, `truncate()`, `chmod()`, `setxattr()`, `link()`...
- `%desc`: trace file-descriptor-related syscalls
  - `read()`, `write()`, `open()`, `close()`, `fsetxattr()`, `poll()`, `select()`, `pipe()`, `fcntl()`, `epoll_create()`, `epoll_wait()`...
- `%process`: trace process management syscalls
  - `fork()`, `clone()`, `exit_group()`, `execve()`, `wait4()`, `unshare()`...
- `%network`: trace network-related syscalls
  - `socket()`, `bind()`, `listen()`, `connect()`, `sendmsg()`...
- `%signal`: trace signal-related syscalls
  - `kill()`, `rt_sigaction()`, `rt_sigprocmask()`, `rt_sigqueueinfo()`...
- `%memory`: trace memory-mapping-related syscalls
  - `mmap()`, `mprotect()`, `mlock()`...
Filtering signals

- `strace --e signal=set`
  - Trace only specified set of signals
  - “sig” prefix in names is optional; following are equivalent:
    - `strace -o strace.log --e signal=sigio,int ls > /dev/null`
    - `strace -o strace.log --e signal=io,int ls > /dev/null`

- `strace --e signal=!set`
  - Exclude specified signals from tracing
Filtering by pathname

- **`strace -P pathname`**: trace only system calls that access file at *pathname*
  - Specify multiple `-P` options to trace multiple paths

**Example:**

```bash
$ strace -o strace.log -P /lib64/libc.so.6 ls > /dev/null
Requested path '/lib64/libc.so.6' resolved into '/usr/lib64/libc-2.18.so'

$ cat strace.log
open("/lib64/libc.so.6", O_RDONLY | O_CLOEXEC) = 3
read(3, "\177 ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0 >\0\1\0\0\0\0\0\0\0\0\0\0\0...", 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2093096, ...}) = 0
mmap(NULL, 3920480, PROT_READ|PROT_EXEC,
    MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f8511fa3000
mmap(0x7f8512356000, 24576, PROT_READ|PROT_WRITE,
    MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1b3000 )
    = 0x7f8512356000
close(3)                                = 0
+++ exited with 0 +++
```

- **`strace`** noticed that the specified file was opened on FD 3, and also traced operations on that FD
Mapping file descriptors to pathnames

- \( -y \) option causes `strace` to display pathnames corresponding to each file descriptor
- Useful info is also displayed for other types of file descriptors, such as pipes and sockets

```
$ strace -y cat greet
...
openat(AT_FDCWD, "greet", O_RDONLY) = 3</home/mtk/greet>
fstat(3</home/mtk/greet>, {st_mode=S_IFREG|0644, ...
read(3</home/mtk/greet>, "hello world\n", 131072) = 12
write(1</dev/pts/11>, "hello world\n", 12) = 12
read(3</home/mtk/greet>, "", 131072) = 0
close(3</home/mtk/tlpi/code/greet>) = 0
...
```

- \( -yy \) is as for \( -y \) but shows additional protocol-specific info for sockets

```
write(3<TCP:[10.0.20.135:33522->213.131.240.174:80]>,
"GET / HTTP/1.1\r\nUser-Agent: Wget\r\n...", 135) = 135
read(3<TCP:[10.0.20.135:33522->213.131.240.174:80]>,
"HTTP/1.1 200 OK\r\nDate: Thu, 19 J...", 253) = 253
```
Outline

1. Getting started 3
2. Tracing child processes 10
3. Filtering strace output 14
4. System call tampering 20
5. Further strace options 24
**System call tampering**

- *strace* can be used to **modify** behavior of selected syscall(s)
  - Initial feature implementation completed in early 2017

- Various possible effects:
  - Inject delay before/after syscall
  - Generate a signal on syscall
  - Bypass execution of syscall, making it return a “success” value or fail with specified value in *errno*
  - (Limited) ability to choose which invocation of syscall will be modified

- Syntax: `strace -e inject=<syscall-set>[[: <option>]]`...
  - *syscall-set* is set of syscalls whose behavior will be modified
strace -e inject options

- :error=errnum: syscall is not executed; returns failure status with errno set as specified
- :retval=value: syscall is not executed; returns specified “success” value
  - Can’t specify both :retval and :errno together
- :signal=sig: deliver specified signal on entry to syscall
- :delay_enter=usecs, :delay_exit=usecs: delay for usecs microseconds on entry to/return from syscall
- :when=expr: specify which invocation(s) to tamper with
  - :when=N: tamper with invocation N
  - :when=N+: tamper starting at Nth invocation
  - :when=N+S: tamper with invocation N, and then every S invocations
- Range of N and S is 1..65535
Example

```
$ strace -y -e close \n   -e inject=close:error=22:when=3 /bin/ls > d
close(3</etc/ld.so.cache>) = 0
close(3</usr/lib64/libselinux.so.1>) = 0
close(3</usr/lib64/libcap.so.2.25>) = -1 EINVAL
  (Invalid argument) (INJECTED)
close(3</usr/lib64/libcap.so.2.25>) = 0
/bin/ls: error while loading shared libraries: libcap.so.2:
cannot close file descriptor: Invalid argument
+++ exited with 127 +++
```

- Use 
  
- Inject error 22 (EINVAL) on third call to `close()`
- Third `close()` was not executed; an error return was injected
  - (After that, `ls` got sad)
<table>
<thead>
<tr>
<th>Outline</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Getting started</td>
<td>3</td>
</tr>
<tr>
<td>2 Tracing child processes</td>
<td>10</td>
</tr>
<tr>
<td>3 Filtering strace output</td>
<td>14</td>
</tr>
<tr>
<td>4 System call tampering</td>
<td>20</td>
</tr>
<tr>
<td>5 Further strace options</td>
<td>24</td>
</tr>
</tbody>
</table>
Obtaining a system call summary

- `strace -c` counts time, calls, and errors for each system call and reports a summary on program exit

```
$ strace -c who > /dev/null

%  time    seconds  usecs/call    calls    errors  syscall
---------  -----------  -----------  -------  -------  --------------
 21.77    0.000648      9           72     alarm
 14.42    0.000429      9           48  rt_sigaction
 13.34    0.000397      8           48    fcntl
  8.84    0.000263      5           48    read
  7.29    0.000217     13           17     2 kill
  6.79    0.000202      6           33    1 stat
  5.41    0.000161      5           31    mmap
  4.44    0.000132      4           31    6 open
  2.89    0.000086      3           29    close
  2.86    0.000085     43           2   socket
  2.82    0.000084     42           2   connect

...  

---------  -----------  -------  -------  --------------
100.00    0.002976    442     13 total
```

- Treat time measurements as indicative only, since `strace` adds overhead to each syscall
Tracing live processes

- `–p PID`: trace running process with specified PID
  - Type `Control-C` to cease tracing
  - To trace multiple processes, specify `–p` multiple times
  - Can trace only processes you own
  - ⚠️⚠️ tracing a process can heavily affect performance
    - E.g., up to two orders of magnitude slow-down in syscalls
    - ⚠️ Think twice before using in a production environment

- `–p PID -f`: will trace all threads in specified process
Further *strace* options

- **–v**: don’t abbreviate arguments (structures, etc.)
  - Output can be quite verbose...

- **–s *strsize***: maximum number of bytes to display for strings
  - Default is 32 characters
  - Pathnames are always printed in full

- Various options show start time or duration of system calls
  - **–t, –tt**: prefix each trace line with wall-clock time
    - **–tt** also adds microseconds
  - **–T**: show time spent in syscall
    - But treat as indications only, since *strace* causes overhead on syscalls

- **–i**: print value of instruction pointer on each system call
Thanks!

Michael Kerrisk  mtk@man7.org  @mkerrisk

Slides at http://man7.org/conf/
Source code at http://man7.org/tlpi/code/

Training: Linux system programming, security and isolation APIs, and more; http://man7.org/training/