Ptrace (Process Trace)

Helali Bhuiyan

Linux provides `ptrace` as a process tracing tool, which can intercept system calls at their entry and exit points, made from another process. `ptrace` provides a mechanism by which a parent process may observe and control the execution of another process. It can examine and change a child process's core image and registers, and is used primarily to implement breakpoint debugging and system call tracing.

`ptrace` takes the following arguments,

```
long ptrace(enum __ptrace_request request,
            pid_t pid,
            void *addr,
            void *data);
```

where,
`request` = type of behavior of `ptrace`. For example, we can attach or detach from a process, read/write registers, read/write code segment and data segment.
`pid` = process id of the traced process
`addr` = address
`data` = data

A process can be traced by two ways:
1. The traced application can be run as a child, executing `fork()` within the parent process, or the tracer application. In this case, the traced application needs to call `ptrace` with the following parameters

```
ptrace(PTRACE_TRACEME, 0, NULL, NULL);
```

This means, we need to modify the source code of the traced application to add this line of code.

2. If an application is already running and we want to trace it, then the tracer application can use the following format of `ptrace`

```
ptrace(PTRACE_ATTACH, pid_of_traced_process, NULL, NULL);
```

In this case, the traced application needs not to add any code. In both cases, all we need is the process id or `pid` to trace an application. The `pid` of a running process is obtained by executing the `ps` command in Linux. Once being traced, the traced application becomes a child process of the tracer application.

Once a process is being traced, whenever the traced process executes a system call or returns from a system call, the control of execution is handed over to the tracer application. Then the tracer application can check the arguments of the system call or do other things, such as looking into the registers, modifying register values, inject codes

into the code segment. Also values returned from the system call can be accessed and modified in a similar way. Once the tracer application is done examining the system call, the traced application can continue with the system call.

Example:
This section demonstrates the idea of `ptrace` with some examples. Help on how to use `ptrace` can be found in the following two websites

http://www.linuxjournal.com/article/6100
http://www.linuxjournal.com/node/6210/print

In order to demonstrate the usage of `ptrace`, I wrote a small `server` and a `client` application. The `client` application connects to the `server` by opening a TCP socket. Then the `client` prompts the user for an input string to send, which is later transmitted to the `server` application. In this example, the `server` application is run on host `zelda4` and the `client` is run on host `zelda1`,

Server

```
[amb6fp@zelda4 tracer]$ ./server 5001
#CC algorithm:      bic
Here is the message: Hello World
```

Client

```
[amb6fp@zelda1 tracer]$ ./client 198.124.42.17 5001
CLIENT: socket fd: 3
CLIENT: buffer size: 8192000
CLIENT: #CC algorithm:      bic
CLIENT: Please enter the message: Hello World
CLIENT: I got your message
```

Once the `client` connects to the `server`, it prints some basic information of the socket. For example, it prints the buffer size and the congestion-control algorithm being used for this connection. Then the `client` asks for the message from the user, which is later transmitted to the `server` and printed. In order to distinguish between outputs printed by the `client` and the `ptrace` application (`interceptor`), which is explained later, the print commands executed from the `client` start with `CLIENT:` . Similarly outputs of the interceptor application start with `INTERCEPTOR:` .

Before explaining how `interceptor` works, the following example shows the output when the `client` application is traced by the `interceptor`,

```
[amb6fp@zelda1]$ ./interceptor ./client 198.124.42.17 5001
Number of input 4
INTERCEPTOR: This is a SOCKET call
INTERCEPTOR: Family:2 Type:1 Protocol:0
CLIENT: socket fd: 3
```

```
CLIENT: buffer size: 8192000
INTERCEPTOR: This is a CONNECT call
INTERCEPTOR: IP: 198.124.42.17, Port: 5001, Family: 2
CLIENT: #CC algorithm:      bic
CLIENT: Please enter the message: Hello World
CLIENT: I got your message
INTERCEPTOR: Exited
```

In this example, the `client` application is passed as an argument (with `client`'s own arguments) to the `interceptor` application. As displayed, the `interceptor` application traces the `client` application and traps the socket system call. It also traps the connect system call, and prints the destination IP address and the port number, which were passed as arguments to the connect call.

The following code segment of the `interceptor` application describes the idea on how to trace an application,

```
char *cmd[10];
for(i = 0; i < argc-1; i++){
        cmd[i] = argv[i+1];
}
cmd[i] = (char *)0;

pid_t processid = fork();
if(processid == 0){
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execvp(argv[1], cmd);
}
```

At first, arguments passed to the `interceptor` are copied in an array. Then a child process is created by executing the `fork` command. At this point, the child process is just an image of the parent process (`interceptor`), but it does not do anything meaningful. Before actually running the `client` application (which is passed in the command list), tracing of the child process is started by calling the `ptrace` function call. Therefore, the `interceptor` will trap every system call from the beginning that will be made from this child process. Once the tracing is started, the `client` application is executed by the `execvp` command.

The `interceptor` application then enters into a loop and waits for any system call that will be made from the `client` application. Once a system call is trapped, using the `ptrace` function call with appropriate arguments, parameters passed to that system call can be viewed and also be modified.

The `interceptor` application can also be used if a circuit-setup decision needs to be made. If the destination IP address that has been passed to the connect system call matches with the desired IP address, before letting the connect call to continue, a circuit-

setup procedure can be executed. Once the circuit is set up, the traced application continues with the connect call and transmits data over the circuit. Attached source code has some example codes that explain this idea.