Playing with ptrace, Part I

Using ptrace allows you to set up system call interception and modification at the user level.

by Pradeep Padala

Have you ever wondered how system calls can be intercepted? Have you ever tried fooling the kernel by changing system call arguments? Have you ever wondered how debuggers stop a running process and let you take control of the process?

If you are thinking of using complex kernel programming to accomplish tasks, think again. Linux provides an elegant mechanism to achieve all of these things: the ptrace (Process Trace) system call. **ptrace** provides a mechanism by which a parent process may observe and control the execution of another process. It can examine and change its core image and registers and is used primarily to implement breakpoint debugging and system call tracing.

In this article, we learn how to intercept a system call and change its arguments. In Part II of the article we will study advanced techniques--setting breakpoints and injecting code into a running program. We will peek into the child process' registers and data segment and modify the contents. We will also describe a way to inject code so the process can be stopped and execute arbitrary instructions.

Basics

Operating systems offer services through a standard mechanism called system calls. They provide a standard API for accessing the underlying hardware and low-level services, such as the filesystems. When a process wants to invoke a system call, it puts the arguments to system calls in registers and calls soft interrupt 0x80. This soft interrupt is like a gate to the kernel mode, and the kernel will execute the system call after examining the arguments.

On the i386 architecture (all the code in this article is i386-specific), the system call number is put in the register %eax. The arguments to this system call are put into registers %ebx, %ecx, %edx, %esi and %edi, in that order. For example, the call:

```
write(2, "Hello", 5)
```

roughly would translate into

```
movl   $4, %eax
movl   $2, %ebx
movl   $hello,%ecx
movl   $5, %edx
int    $0x80
```

where $hello points to a literal string "Hello".

So where does ptrace come into picture? Before executing the system call, the kernel checks whether the process is being traced. If it is, the kernel stops the process and gives control to the tracking process so it can examine and modify the traced process' registers.

Let's clarify this explanation with an example of how the process works:

```
#include <sys/ptrace.h>
#include <sys/types.h>
```

```
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>   /* For constants
                                ORIG_EAX etc */

int main()
{   pid_t child;
    long orig_eax;

    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    }
    else {
        wait(NULL);
        orig_eax = ptrace(PTRACE_PEEKUSER,
                          child, 4 * ORIG_EAX,
                          NULL);
        printf("The child made a "
               "system call %ld\n", orig_eax);
        ptrace(PTRACE_CONT, child, NULL, NULL);
    }
    return 0;
}
```

When run, this program prints:

```
The child made a system call 11
```

along with the output of ls. System call number 11 is execve, and it's the first system call executed by the child. For reference, system call numbers can be found in /usr/include/asm/unistd.h.

As you can see in the example, a process forks a child and the child executes the process we want to trace. Before running `exec`, the child calls ptrace with the first argument, equal to PTRACE_TRACEME. This tells the kernel that the process is being traced, and when the child executes the execve system call, it hands over control to its parent. The parent waits for notification from the kernel with a wait() call. Then the parent can check the arguments of the system call or do other things, such as looking into the registers.

When the system call occurs, the kernel saves the original contents of the eax register, which contains the system call number. We can read this value from child's USER segment by calling ptrace with the first argument PTRACE_PEEKUSER, shown as above.

After we are done examining the system call, the child can continue with a call to ptrace with the first argument PTRACE_CONT, which lets the system call continue.

ptrace Parameters

ptrace is called with four arguments:

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

The first argument determines the behaviour of ptrace and how other arguments are used. The value of request should be one of PTRACE_TRACEME, PTRACE_PEEKTEXT, PTRACE_PEEKDATA, PTRACE_PEEKUSER, PTRACE_POKETEXT, PTRACE_POKEDATA, PTRACE_POKEUSER, PTRACE_GETREGS, PTRACE_GETFPREGS, PTRACE_SETREGS, PTRACE_SETFPREGS, PTRACE_CONT, PTRACE_SYSCALL, PTRACE_SINGLESTEP, PTRACE_DETACH. The significance

of each of these requests will be explained in the rest of the article.

Reading System Call Parameters

By calling ptrace with PTRACE_PEEKUSER as the first argument, we can examine the contents of the USER area where register contents and other information is stored. The kernel stores the contents of registers in this area for the parent process to examine through ptrace.

Let's show this with an example:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>
#include <sys/syscall.h>   /* For SYS_write etc */

int main()
{   pid_t child;
    long orig_eax, eax;
    long params[3];
    int status;
    int insyscall = 0;

    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("/bin/ls", "ls", NULL);
    }
    else {
       while(1) {
          wait(&status);
          if(WIFEXITED(status))
              break;
          orig_eax = ptrace(PTRACE_PEEKUSER,
                    child, 4 * ORIG_EAX, NULL);
          if(orig_eax == SYS_write) {
             if(insyscall == 0) {
                /* Syscall entry */
                insyscall = 1;
                params[0] = ptrace(PTRACE_PEEKUSER,
                                   child, 4 * EBX,
                                   NULL);
                params[1] = ptrace(PTRACE_PEEKUSER,
                                   child, 4 * ECX,
                                   NULL);
                params[2] = ptrace(PTRACE_PEEKUSER,
                                   child, 4 * EDX,
                                   NULL);

                printf("Write called with "
                       "%ld, %ld, %ld\n",
                       params[0], params[1],
                       params[2]);
                }
          else { /* Syscall exit */
                eax = ptrace(PTRACE_PEEKUSER,
                             child, 4 * EAX, NULL);
                printf("Write returned "
                       "with %ld\n", eax);
                insyscall = 0;
                }
            }
            ptrace(PTRACE_SYSCALL,
                   child, NULL, NULL);
```

```
        }
    }
    return 0;
}
```

This program should print an output similar to the following:

```
ppadala@linux:~/ptrace > ls
a.out        dummy.s      ptrace.txt
libgpm.html  registers.c  syscallparams.c
dummy        ptrace.html  simple.c

ppadala@linux:~/ptrace > ./a.out
Write called with 1, 1075154944, 48
a.out        dummy.s      ptrace.txt
Write returned with 48
Write called with 1, 1075154944, 59
libgpm.html  registers.c  syscallparams.c
Write returned with 59
Write called with 1, 1075154944, 30
dummy        ptrace.html  simple.c
Write returned with 30
```

Here we are tracing the write system calls, and `ls` makes three write system calls. The call to ptrace, with a first argument of PTRACE_SYSCALL, makes the kernel stop the child process whenever a system call entry or exit is made. It's equivalent to doing a PTRACE_CONT and stopping at the next system call entry/exit.

In the previous example, we used PTRACE_PEEKUSER to look into the arguments of the write system call. When a system call returns, the return value is placed in %eax, and it can be read as shown in that example.

The status variable in the wait call is used to check whether the child has exited. This is the typical way to check whether the child has been stopped by ptrace or was able to exit. For more details on macros like WIFEXITED, see the wait(2) man page.

Reading Register Values

If you want to read register values at the time of a syscall entry or exit, the procedure shown above can be cumbersome. Calling ptrace with a first argument of PTRACE_GETREGS will place all the registers in a single call.

The code to fetch register values looks like this:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>
#include <sys/syscall.h>

int main()
{   pid_t child;
    long orig_eax, eax;
    long params[3];
    int status;
    int insyscall = 0;
    struct user_regs_struct regs;

    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
```

```
        execl("/bin/ls", "ls", NULL);
    }
    else {
       while(1) {
          wait(&status);
          if(WIFEXITED(status))
              break;
          orig_eax = ptrace(PTRACE_PEEKUSER,
                            child, 4 * ORIG_EAX,
                            NULL);
          if(orig_eax == SYS_write) {
              if(insyscall == 0) {
                 /* Syscall entry */
                 insyscall = 1;
                 ptrace(PTRACE_GETREGS, child,
                        NULL, &regs);
                 printf("Write called with "
                        "%ld, %ld, %ld\n",
                        regs.ebx, regs.ecx,
                        regs.edx);
              }
              else { /* Syscall exit */
                  eax = ptrace(PTRACE_PEEKUSER,
                               child, 4 * EAX,
                               NULL);
                  printf("Write returned "
                         "with %ld\n", eax);
                  insyscall = 0;
              }
          }
          ptrace(PTRACE_SYSCALL, child,
                 NULL, NULL);
       }
   }
   return 0;
}
```

This code is similar to the previous example except for the call to ptrace with PTRACE_GETREGS. Here we have made use of the user_regs_struct defined in <linux/user.h> to read the register values.

Doing Funny Things

Now it's time for some fun. In the following example, we will reverse the string passed to the write system call:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>
#include <sys/syscall.h>

const int long_size = sizeof(long);

void reverse(char *str)
{   int i, j;
    char temp;

    for(i = 0, j = strlen(str) - 2;
        i <= j; ++i, --j) {
        temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}
```

```
void getdata(pid_t child, long addr,
             char *str, int len)
{   char *laddr;
    int i, j;
    union u {
            long val;
            char chars[long_size];
    }data;

    i = 0;
    j = len / long_size;
    laddr = str;
    while(i < j) {
        data.val = ptrace(PTRACE_PEEKDATA,
                          child, addr + i * 4,
                          NULL);
        memcpy(laddr, data.chars, long_size);
        ++i;
        laddr += long_size;
    }

    j = len % long_size;
    if(j != 0) {
        data.val = ptrace(PTRACE_PEEKDATA,
                          child, addr + i * 4,
                          NULL);
        memcpy(laddr, data.chars, j);
    }
    str[len] = '\0';
}

void putdata(pid_t child, long addr,
             char *str, int len)
{   char *laddr;
    int i, j;
    union u {
            long val;
            char chars[long_size];
    }data;

    i = 0;
    j = len / long_size;
    laddr = str;
    while(i < j) {
        memcpy(data.chars, laddr, long_size);
        ptrace(PTRACE_POKEDATA, child,
               addr + i * 4, data.val);
        ++i;
        laddr += long_size;
    }

    j = len % long_size;
    if(j != 0) {
        memcpy(data.chars, laddr, j);
        ptrace(PTRACE_POKEDATA, child,
               addr + i * 4, data.val);
    }
}

int main()
{
   pid_t child;

   child = fork();
   if(child == 0) {
      ptrace(PTRACE_TRACEME, 0, NULL, NULL);
      execl("/bin/ls", "ls", NULL);
   }
```

```
   else {
      long orig_eax;
      long params[3];
      int status;
      char *str, *laddr;
      int toggle = 0;

      while(1) {
         wait(&status);
         if(WIFEXITED(status))
             break;
         orig_eax = ptrace(PTRACE_PEEKUSER,
                           child, 4 * ORIG_EAX,
                           NULL);

         if(orig_eax == SYS_write) {
            if(toggle == 0) {
               toggle = 1;

               params[0] = ptrace(PTRACE_PEEKUSER,
                                  child, 4 * EBX,
                                  NULL);
               params[1] = ptrace(PTRACE_PEEKUSER,
                                  child, 4 * ECX,
                                  NULL);
               params[2] = ptrace(PTRACE_PEEKUSER,
                                  child, 4 * EDX,
                                  NULL);

               str = (char *)calloc((params[2]+1)
                                    * sizeof(char));
               getdata(child, params[1], str,
                       params[2]);
               reverse(str);
               putdata(child, params[1], str,
                       params[2]);
            }
            else {
               toggle = 0;
            }
         }
      ptrace(PTRACE_SYSCALL, child, NULL, NULL);
      }
   }
   return 0;
}
```

The output looks like this:

```
ppadala@linux:~/ptrace > ls
a.out        dummy.s      ptrace.txt
libgpm.html  registers.c  syscallparams.c
dummy        ptrace.html  simple.c
ppadala@linux:~/ptrace > ./a.out
txt.ecartp      s.ymmud      tuo.a
c.sretsiger     lmth.mpgbil  c.llacys_egnahc
c.elpmis        lmth.ecartp  ymmud
```

This example makes use of all the concepts previously discussed, plus a few more. In it, we use calls to ptrace with PTRACE_POKEDATA to change the data values. It works exactly the same way as PTRACE_PEEKDATA, except it both reads and writes the data thatt the child passes in arguments to the system call whereas PEEKDATA only reads the data.

Single-Stepping

ptrace provides features to single-step through the child's code. The call to ptrace(PTRACE_SINGLESTEP,..) tells the kernel to stop the child at each instruction and let the parent take control. The following example shows a way of reading the instruction being executed when a system call is executed. I have created a small dummy executable for you to understand what is happening instead of bothering with the calls made by libc.

Here's the listing for dummy1.s. It's written in assembly language and compiled as gcc -o dummy1 dummy1.s:

```
.data
hello:
    .string "hello world\n"

.globl  main
main:
    movl    $4, %eax
    movl    $2, %ebx
    movl    $hello, %ecx
    movl    $12, %edx
    int     $0x80
    movl    $1, %eax
    xorl    %ebx, %ebx
    int     $0x80
    ret
```

The example program that single-steps through the above code is:

```
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <linux/user.h>
#include <sys/syscall.h>

int main()
{   pid_t child;
    const int long_size = sizeof(long);

    child = fork();
    if(child == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("./dummy1", "dummy1", NULL);
    }
    else {
        int status;
        union u {
            long val;
            char chars[long_size];
        }data;
        struct user_regs_struct regs;
        int start = 0;
        long ins;

        while(1) {
            wait(&status);
            if(WIFEXITED(status))
                break;
            ptrace(PTRACE_GETREGS,
                   child, NULL, &regs);
            if(start == 1) {
                ins = ptrace(PTRACE_PEEKTEXT,
                             child, regs.eip,
                             NULL);
                printf("EIP: %lx Instruction "
```

```
                        "executed: %lx\n",
                        regs.eip, ins);
            }

            if(regs.orig_eax == SYS_write) {
                start = 1;
                ptrace(PTRACE_SINGLESTEP, child,
                        NULL, NULL);
            }
            else
                ptrace(PTRACE_SYSCALL, child,
                        NULL, NULL);
        }
    }
    return 0;
}
```

This program prints:

```
hello world
EIP: 8049478 Instruction executed: 80cddb31
EIP: 804947c Instruction executed: c3
```

You might have to look at Intel's manuals to make sense out of those instruction bytes. Using single stepping for more complex processes, such as setting breakpoints, requires careful design and more complex code.

In Part II, we will see how breakpoints can be inserted and code can be injected into a running program.

All of the example code from this article and from Part II (which will be printed in next month's issue) is available as a tar archive on the *Linux Journal* FTP site [ftp.ssc.com/pub/lj/listings/issue103/6011.tgz].

Pradeep Padala *is currently working on his Master's degree at the University of Florida. His research interests include Grid and distributed systems. He can be reached via e-mail at p_padala@yahoo.com or through his web site (www.cise.ufl.edu/~ppadala).*