Recently, I needed to investigate the write activity of a Linux process and was surprised to discover that CPAN had a Ptrace module. Ptrace is a technology that roots in the Linux kernel, adding the ability to step through processes and retrieve information on the data they use. Debuggers such as GDB leverage this technology and build a user interface on it.

To find out which files a process opens for writing over the course of its lifetime, you can pass the PTRACE_SYSCALL parameter to ptrace to make the process stop whenever it issues a system call. Filtering out calls coming from libc’s open() function in write mode then reveals the desired list of files. Invoking objdump -d /lib/libc.so.6 tells you what libc does to open the specified file and return a file descriptor (Figure 1).

To most of us, disassembler output is incomprehensible at first glance. The x86 assembler code in Figure 1 picks up the function parameters for open() from the stack (%esp) and uses the mov (move) instruction to store them in the processor registers EBX, ECX, and EDX (assembler code prepends a percent sign). From the include file adm/unistd.h (Figure 2), you can see that the kernel refers to the open() system call internally as 5, and libc calls mov $0x5,%eax to write the value to the processor’s EAX register.

The int $0x80 call lets the kernel take control. The call triggers an interrupt, and the kernel switches to privileged mode and processes the system call on.

Linux lets users watch the kernel at work with a little help from Ptrace, a tool that both debuggers and malicious process kidnappers use. A CPAN module introduces this technology to Perl and, if this is not enough, C extensions add functionality. BY MICHAEL SCHILLI

Perl script uses Ptrace for process tracing

PROCESS SPY
the other side of the wall in kernel land. It picks up the parameters from the processor registers where they were stored previously by libc.

The `open()` function expects up to three parameters: `int open(const char *pathname, int flags, mode_t mode)`.

The string that specifies the path will obviously not fit in a 32-bit register. Therefore, the EBX register only holds the memory address at which the string can be found.

To find out whether a system call picked up at random is an `open()` with write option, the monitoring code must check to see whether EAX contains the value 5 (the code for `open()`) and whether an AND operation of the ECX register and the `O_WRONLY` constant defined in `sys/fcntl.h` results in a true value. A file could also be opened for writing with `O_RDWR` (read/write access) or `O_APPEND` (append to file), but I will ignore this to keep things simple. Incidentally, it makes no difference which higher level language was used to write the code — C, Perl, Java, Ruby, etc. All of them use the `open()` call from libc.

Listing 1 shows the Perl code that helps a script trace system calls in a process and eavesdrop on it for occurrences of `open()` requests with write intention. After the `fork()`, the new child process issues the Ptrace `PTRACE_TRACEME` command and then launches the surveyed program with `exec()`. The parent process waits (waitpid()) for the kernel to stop the child process right after it has started its payload. The parent process then reactivates the child process by issuing `PTRACE_SYSCALL`, which tells the kernel to stop the child again the next time it issues a system call. The next time the child is then stopped, the parent process can investigate which system call has been issued with which

### Listing 1: WriteTracer.pm

```perl
package WriteTracer;
use strict;
use POSIX;
use Inline "C";
use Fcntl;

sub run {
    my ($prg, @params) = @_; 
    my @files = (); 
    my %files = (); 
    
    if ((my $pid = fork()) < 0) {
        die "fork failed";
    } elsif ($pid == 0) { 
        # child
        ptrace(PTRACE_TRACEME, $$, 0, 0);
        exec($prg, @params);
    } else { 
        # parent
        
        my $rc = waitpid($pid, 0);
        last if $rc < 0;
        if (WIFSTOPPED($?)) {
            my ($eax, $orig_eax, $ebx, $ ecx, $edx) = ptrace_getregs($pid);
            if ($eax == -ENOSYS()) {
                if ($orig_eax == 5 and $ecx & O_WRONLY) {
                    my $str = ptrace_string_read($pid, $ebx);
                    push @files, $str unless $files{$str}++;
                }
            }
            
        } 
        ptrace(PTRACE_SYSCALL, $pid, undef, undef);
        redo;
    } 
    return @files;
}
```

Figure 2: Excerpt from unistd.h.

Figure 3: Parent and child process interacting during Ptrace tracing.
parameters with the use of other Ptrace commands.

Normally, the kernel would call the appropriate system call handler without any delay after receiving a system call request. If the kernel notices that Ptrace is monitoring the process, it instead jumps to the `tracesys` kernel function that
- stops the process and notifies the parent process of the imminent system call and
- stops again after completing the system call and notifies the parent process of the results.

To allow the tracer to distinguish between these two cases, the kernel sets the EAX register to `-ENOSYS` for the first stop. As I mentioned previously, the EAX register normally contains the number of the system call to be executed. `-ENOSYS` is the kernel’s error message if it encounters a non-existent system call number. Because this is an impossible value for a system call, the tracing process knows that the subject of the trace is about to issue a system call, whose number the kernel stores in ORIG_EAX for safekeeping.

Line 39 in `WriteTracer.pm` uses the `WIFSTOPPED()` macro and Perl’s status variable `?` to check to see whether the child process stopped or whether `wait‑pid()` alerted because the child crashed. Line 44 verifies that the EAX register read by the `ptrace_getargs()` function does contain a value of `-ENOSYS`.

If so, the next if condition checks to see whether ORIG_EAX is set to 5 (the

### Listing 1: WriteTracer.pm (continued)

```perl
073  sv_2mortal(newSViv(x))); 108  char *aligned_addr; 143  Inline_Stack_Vars;
074  109  long word; 144
075  /* ------------------------ */ 110  void *ptr; 145  pv = newSVpv( 146     (const char *)"");
076  void ptrace_getregs( 111  147
077       int pid) { 112  aligned_addr = (char *) ( 148     while(1) {
078       int rc; 113  (long)addr & 149
079       struct user_regs_struct 114  - (sizeof(long) - 1)); 150  rc =
080        registers; 115  151  ptrace_aligned_word_read_c( 152     ptrace_aligned_word_read_c 153     word = ptrace( 154     PTRACE_PEEKDATA, pid,
081        registers; 116  155     aligned_addr, NULL); 156     aligned_addr, NULL);
082  117  PTRACE_PEEKDATA, pid,
083  ptr = 0, &registers); 118  word; 157     word = ptrace(PTRACE_PEEKDATA, pid,
084  119  158     aligned_addr, NULL); 159     *len = sizeof(long) - ( 160     ptr = &word;
085  120  if(word == -1) { 121  (long) addr - 161     memcpyp(buf, ptr, *len);
086  122  return -1; 123  } 162     } 163     )
087  124  *len = sizeof(long) - ( 125     ptr = 0;
088  126  (long) aligned_addr ); 127     &word;
089  127  ptr +=(sizeof(long) - *len); 128     ptr +=(sizeof(long) - *len);
090  if( registers.eax == 129     memcpyp(buf, ptr, *len);
091    -ENOSYS ) { 130     } 164     (const char *)
092  Inline_Stack_Reset; 131     return 0;
093  IVPUSH(registers.eax); 132     )
094  IVPUSH(registers.orig_eax); 133
095  IVPUSH(registers ebx); 134  /* ------------------------ */
096  IVPUSH(registers ecx); 135  void ptrace_string_read( 136  int pid, void *addr) {
097  IVPUSH(registers edx); 137  138  char word_buf[sizeof(long)];
098  Inline_Stack_Done; 139  int word_len;
099  | 140  5V *pv;
100  | 141  int rc;
101  */ ------------------------ */ 142  int i;
102  int 135  void ptrace_string_read( 136  int pid, void *addr) {
103  ptrace_aligned_word_read_c( 137  138  char word_buf[sizeof(long)];
104  int pid, void *addr, 139  int word_len;
105  140  5V *pv;
```

Figure 4: The tracer identifies the files opened for reading by the Perl script.
open() system call number) and whether an AND operation with O_WRONLY and the ECX register returns a true value. If all of these conditions are fulfilled, the ptrace_string_read() function reads the string at the memory address stored in the EBX register and stores the returned Perl scalar in the @files array. A hash %files ensures that this happens exactly once per file name.

After this, WriteTracer.pm issues a ptrace command with the PTRACE_SYS‑CALL parameter, which revives the child. The redo instruction in line 57 of the parent process jumps to waitpid(), which waits for the next child process state change. Listing 2 shows an application for the tracer and expects a command along with its command-line parameters to pass to WriteTracer.pm. Figure 4 shows a Perl program that opens two files along with the correct output of the tracer monitoring the process.

The Sys::PTRACE Perl module from CPAN, which I used for the Ptrace commands, is not complete. To work around this, WriteTracer.pm uses Inline::C to define a few C extensions. The functions called by the Perl code, ptrace_getregs() and ptrace_string_read(), are defined in the __DATA__ area following the Perl code. Inline::C compiles them first time that WriteTracer.pm is executed.

The ptrace_getregs() function expects the child process number because the __DATA__ area following the Perl code. The child process number because the __DATA__ area following the Perl code.

listings

Listing 2: write‑tracer

```perl
01 #!/usr/bin/perl -w
02 #########################################################################
03 # write‑tracer
04 # Mike Schilli, 2008
05 # (m@perlmeister.com)
06 #########################################################################
07 use strict;
08 use WriteTracer;
09
die "usage: $0 program"
11 unless @ARGV;
12
13 my @files =
14 WriteTracer::run(@ARGV);
15
16 print "Written files: ",
17 join(" ", @files), "\n";
```

ptrace(PTRACE_GETREGS,...) function requires you to specify the process whose registers you want it to query. The register values are stored in a user_regs_struct type C structure, which is defined in the asm/user.h kernel header. The __DATA__ area following the Perl code. Inline::C compiles them first time that WriteTracer.pm is executed.

At first, the Perl scalar created by ptrace_getregs() inline C Perl function to return a list of register values to Perl land.

The values prepared by su_2mortal(newSViv(s)) are temporary scalars that Perl’s garbage collector cleans up when the referencing Perl variables disappear from their scope.

The ptrace_string_read() function defined in lines 135ff of Listing 1 uses the Ptrace TRACE_PEEKDATA command to read a C string at a known memory address, but it does not have to deal with the peculiarities of alignment in Linux memory. As Figure 5 shows, strings can start at arbitrary memory addresses but can only be retrieved at 4-byte word boundaries. The ptrace_aligned_word_read_c() C function defined in lines 104ff handles this; it expects a PID and a memory address and returns a buffer along with its length as buf and len. If the address lies on a word boundary, the first snippet has a length of 4 bytes; the length is shorter for uneven addresses.

At first, the Perl scalar created by newSVpv() to hold the file name string is empty, and su_cattsv() appends each new byte it finds. If the function encounters a null byte, it has found the end of the string in memory and uses goto to jump out of the twin loop to the FINISH label.

Restrictions

If the program traced by Ptrace invokes further processes, it is impossible to trace them. Because make does not execute the installation commands within the same process (instead, it launches new ones for each of them), you can’t simply trace what make does by running write‑tracer make install.

To work around this restriction, tracers such as installwatch [2] and checkinstall [3] adopt a different approach. They set the LD_PRELOAD environmental variable, which injects a shared library with system call wrappers and which the subprocesses inherit from make. The wrapper library defines new entries for all popular file functions in libc and tricks the traced program into thinking that these are the real thing.

The wrapper functions only log the proceedings before calling the appropriate libc function, which does all the work. But even this approach fails if a Perl script issues the system(“cp a b”) command, because LD_PRELOAD is not inherited in this case, and installwatch or checkinstall don’t notice the copy.

Ptrace is not only useful for legitimate applications. Black hats love to use the technology to hijack active processes to do their dastardly deeds [4].

If you are interested in more advanced debugging and process tracing techniques besides Ptrace, read Self‑Service Linux [5], which was a big help to me in writing this article.

Ptrace’s biggest customer is the tracer [6] command‑line tool, which traces – and can latch onto – active processes.

INFO

[1] Listings for this article: http://www.linux‑magazine.com/resources/article_code