NAME

systr_init_library, systr_cleanup_library, systr_run, systr_stop, systr_trace_syscall, systr_untrace_syscall, systr_get_pid, systr_get_param, systr_set_params, systr_is_entry, systr_pmem_read, systr_pmem_write, systr_pszmem_read − System Call Tracing Library support functions

SYNOPSIS

```
#include <sysctr.h>

int systr_init_library(void);
void systr_cleanup_library(void);
int systr_run(char **av);
int systr_stop(void);
int systr_trace_syscall(int syscall, int (*scfunc)(void *, trsyscall_t), void *priv);
int systr_untrace_syscall(int syscall);
int systr_get_pid(trsyscall_t tsc, unsigned long *pid);
int systr_get_param(trsyscall_t tsc, int param, unsigned long *pparam);
int systr_set_params(trsyscall_t tsc, ...);
int systr_is_entry(trsyscall_t tsc);
int systr_pmem_read(trsyscall_t tsc, int where, unsigned long addr, char *buf, int size);
int systr_pmem_write(trsyscall_t tsc, int where, unsigned long addr, char *buf, int size);
int systr_pszmem_read(trsyscall_t tsc, int where, unsigned long addr, char *buf, int bmax);
```

DESCRIPTION

LibSysCTr is a utility library that can be used to intercept system call functions on a Linux system. Process monitoring and sandboxing are just two of the potential usages of **LibSysCTr**. Internally the **LibSysCTr** library uses the **ptrace**(2) functionalities by monitoring and reporting events to the library caller. The **LibSysCTr** is callback driven, that means that the user initializes the library with **systr_init_library**(), registers the system calls he wants to monitor with **systr_trace_syscall**(), and calls **systr_run**() to start receiving events in the form of callback invocation. For each intercepted system call, two calls to the registered callback function are performed. One during the system call entry, before the system call iteself will be executed by the kernel, and one after the kernel has processed the system call (right before returning the userspace). Utility functions are supplied to, retrieve information about the process, get/set the system call parameters, and read/write the monitored process address space. The **LibSysCTr** library follows all threads and processes spawned by the traced task, by giving the caller the complete control over the whole monitored process hierarchy.

Functions

The following functions are defined:

int systr_init_library(void);

Initialize the library and makes it ready to accept other calls to library functions. It should be called only once at the beginning of the program. It returns 0 in case of success, and a value <0 in case of error.

void systr_cleanup_library(void);

Undo the operations done with the **systr_init_library**() call. It stop the monitoring process and kills all the processes spawned by the **systr_run**() call. It should be called after the return of the **systr_run**(), when the user has done using the **LibSysCTr** library.

int systr_run(char *****av*);**

It starts the monitoring activity on the system calls registered with **systr_trace_syscall**(). The *av* parameter is an arrary of character pointers that specify the process binary to be monitored. The *av*[0] array element is the path (or name) of the binary file to be executed, while the following elements are the parameters supplied to the binary. The *av* array is terminated by a *NULL* pointer. The function returns when no more monitored processes are available, or when a call to *systr_stop*() is done by the caller. In case of success the function returns 0, while in case errors happened during the process, a value <0 is returned.

int systr_stop(void);

The function stops the internal monitor loop triggered by a call to **systr_run**() and makes the function **systr_run**() to return soon. It is usually called from inside a user callback to stop the library event processing. It returns 0 in case of success, and a value <0 in case of error.

int systr_trace_syscall(int *syscall***, int (****scfunc***)(void *, trsyscall_t), void ****priv***);**

The function lets the user to register the system call *syscall* to be traced by the **LibSysCTr** library. The parameter *scfunc* specify a pointer to a callback function that will be invoked at every entry and exit from the *syscall* system call. The callback function template will look like:

```
int scfunc(void *priv, trsyscall_t tsc) {
        ...
}
```

The *priv* parameter will be passed back to the callback function, and it is treated transparently by the **LibSysCTr** library. The *tsc* parameter that the callback will receive, is a system call context handle that can be used to call other **LibSysCTr** utility functions. The callback function will return **SYSTR_RET_CONTINUE** if it wants to continue tracing the current process, or **SYSTR_RET_DETACH** if it does not want to receive any more notification from the process associated with the *tsc* context. The *systr_trace_syscall*() returns 0 in case of success, and a value <0 in case of error.

int systr_untrace_syscall(int *syscall***);**

Undo the effects of a *systr_trace_syscall*() function call, by unregistering the *syscall* from the list of the ones monitored by the **LibSysCTr** library. it returns 0 in case of success, and a value <0 in case of error.

int systr_get_pid(trsyscall_t *tsc***, unsigned long ****pid***);**

The function will be used to retrieve the process ID (pid) associated with the current system call context *tsc*. The process ID value will be stored in the location pointed by *pid*. The **systr_get_pid**() function returns 0 in case of success, and a value <0 in case of error.

int systr_get_param(trsyscall_t *tsc***, int** *param***, unsigned long ****pparam***);**

The function lets the caller to retrieve system call parameters (or registers) associated with the context *tsc*. The *param* value specify the system call parameter to be retrieved, and it can be one of the following values:

SYSTR_SYSCALL Returns the system call number.

SYSTR_PARAM_1 Returns the first system call parameter.

SYSTR_PARAM_2 Returns the second system call parameter.

SYSTR_PARAM_3 Returns the third system call parameter.

SYSTR_PARAM_4 Returns the fourth system call parameter.

SYSTR_PARAM_5 Returns the fifth system call parameter.

SYSTR_PARAM_6 Returns the sixth system call parameter.

SYSTR_PARAM_RCODE Returns the return code of the system call (valid only if the system call is exiting).

SYSTR_REG_SP Returns the stack pointer of the process that invoked the current system call.

SYSTR_REG_IP Returns the instruction pointer of the process that invoked the current system call.

The retrieved system call parameter will be stored in the location pointed by *pparam*. The function returns 0 in case of success, or a value <0 in case of error.

int systr_set_params(trsyscall_t *tsc*, **...);**

The function lets the caller to set system call parameters (or registers) associated with the context *tsc*. To optimize the process context writing, the **systr_set_params**() accepts multiple parameter-value cuples to be set at the same time. So, following the *tsc* parameter, there will be a list of *parameter*,*value* couples, terminated with a last *parameter* equal to −1. Example:

```
unsigned long param1, param2;

systr_set_params(tsc, SYSTR_PARAM_1, &param1, SYSTR_PARAM_2, &param2, -1);
```

The **systr_set_params**() function will return 0 in case of success, or a value <0 in case of error.

int systr_is_entry(trsyscall_t *tsc***);**

The **LibSysCTr** system call interception will trigger two callback invocations per each system call. One on system call entry, and one on exit. The **systr_is_entry**() function can be used to distinguish between the entry and the exit from the system call. It returns a value different from 0 in case it is an entry call, or 0 in case it is an exit.

int systr_pmem_read(trsyscall_t *tsc*, **int** *where*, **unsigned long** *addr*, **char ****buf*, **int** *size***);**

The function lets the caller to read the memory of the process associated with the context *tsc*. The *where* parameter is either:

SYSTR_DATA_SECT Read memory from the **DATA** section

SYSTR_TEXT_SECT Read memory from the **TEXT** section

The *addr* parameter specify the address, in the traced process address space, from where to start the read operation, and the *size* parameter specifies the size in bytes of the block to be read. The read data will be stored in the buffer pointed by *buf*. The function return the number of bytes read (usually *size*), or a number lower than *size* in case errors happened.

int systr_pmem_write(trsyscall_t *tsc***, int** *where***, unsigned long** *addr***, char *** *buf***, int** *size***);**

The function lets the caller to write the memory of the process associated with the context *tsc*. The *where* parameter is either:

SYSTR_DATA_SECT Write memory to the **DATA** section

SYSTR_TEXT_SECT Write memory to the **TEXT** section

The *addr* parameter specify the address, in the traced process address space, from where to start the write operation, and the *size* parameter specifies the size in bytes of the block to be written. The data will be read from the user buffer pointed by *buf* . The function return the number of bytes written (usually *size*), or a number lower than *size* in case errors happened.

int systr_pszmem_read(trsyscall_t *tsc***, int** *where***, unsigned long** *addr***, char *** *buf***, int** *bmax***);**

The function lets the caller to read the memory of the process associated with the context *tsc*. The data will be read as zero-terminated string, up to *bmax* bytes. The *where* parameter is either:

SYSTR_DATA_SECT Read memory from the **DATA** section

SYSTR_TEXT_SECT Read memory from the **TEXT** section

The *addr* parameter specify the address, in the traced process address space, from where to start the read operation. The read data will be stored in the buffer pointed by *buf* . The function return the number of bytes read, or −1 in case of error.

EXAMPLE

The follwoing example shows a few lines of C code that uses the **LibSysCTr** library to intercept a few system calls and print parameters during the monitored process life.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <linux/unistd.h>
#include <sysctr.h>

static int open_scfunc(void *priv, trsyscall_t tsc) {
        int entry;
        unsigned long pid, param, rcode;
        char buf[512];

        systr_get_pid(tsc, &pid);
        entry = systr_is_entry(tsc);
        systr_get_param(tsc, SYSTR_PARAM_1, &param);
        if (!entry)
                systr_get_param(tsc, SYSTR_PARAM_RCODE, &rcode);
        buf[0] = 0;
        systr_pszmem_read(tsc, SYSTR_DATA_SECT, param, buf, sizeof(buf) - 1);

        fprintf(stderr, "[%lu] %s open(%s)", pid, entry ? "E": "X", buf);

        if (entry)
```

```
                fprintf(stderr, " = ?0);
        else
                fprintf(stderr, " = %lu0, rcode);

        return SYSTR_RET_CONTINUE;
}

static int close_scfunc(void *priv, trsyscall_t tsc) {
        int entry;
        unsigned long pid, param;

        systr_get_pid(tsc, &pid);
        entry = systr_is_entry(tsc);
        systr_get_param(tsc, SYSTR_PARAM_1, &param);

        fprintf(stderr, "[%lu] %s close(%d)0, pid, entry ? "E": "X", param);

        return SYSTR_RET_CONTINUE;
}

static int exec_scfunc(void *priv, trsyscall_t tsc) {
        int entry;
        unsigned long pid, param;
        char buf[512];

        systr_get_pid(tsc, &pid);
        entry = systr_is_entry(tsc);
        systr_get_param(tsc, SYSTR_PARAM_1, &param);
        buf[0] = 0;
        if (entry)
                systr_pszmem_read(tsc, SYSTR_DATA_SECT, param, buf, sizeof(buf) - 1);

        fprintf(stderr, "[%lu] %s exec(%s)0, pid, entry ? "E": "X", buf);

        return SYSTR_RET_CONTINUE;
}

static int fork_scfunc(void *priv, trsyscall_t tsc) {
        int entry;
        unsigned long pid, cpid;

        systr_get_pid(tsc, &pid);
        entry = systr_is_entry(tsc);
        if (entry)
                fprintf(stderr, "[%lu] E fork()0, pid);
        else {
                systr_get_param(tsc, SYSTR_PARAM_RCODE, &cpid);
                fprintf(stderr, "[%lu] X fork() -> %lu0, pid, cpid);
        }

        return SYSTR_RET_CONTINUE;
}

static int wait_scfunc(void *priv, trsyscall_t tsc) {
        int entry;
        unsigned long pid, res, wpid, options;

        systr_get_pid(tsc, &pid);
        systr_get_param(tsc, SYSTR_PARAM_1, &wpid);
        systr_get_param(tsc, SYSTR_PARAM_3, &options);
        entry = systr_is_entry(tsc);
        if (entry)
                fprintf(stderr, "[%lu] E wait(%ld, %lu)0, pid, wpid, options);
        else {
                systr_get_param(tsc, SYSTR_PARAM_RCODE, &res);
                fprintf(stderr, "[%lu] X wait(%ld, %lu) = %ld0, pid, wpid, options, res);
        }

        return SYSTR_RET_CONTINUE;
}

int main(int ac, char **av) {

        if (systr_init_library() < 0)
                return 1;

        systr_trace_syscall(__NR_execve, exec_scfunc, NULL);
        systr_trace_syscall(__NR_open, open_scfunc, NULL);
        systr_trace_syscall(__NR_close, close_scfunc, NULL);
        systr_trace_syscall(__NR_fork, fork_scfunc, NULL);
        systr_trace_syscall(__NR_vfork, fork_scfunc, NULL);
        systr_trace_syscall(__NR_clone, fork_scfunc, NULL);
        systr_trace_syscall(__NR_waitpid, wait_scfunc, NULL);
        systr_trace_syscall(__NR_wait4, wait_scfunc, NULL);

        systr_run(&av[i]);

        systr_cleanup_library();

        return 0;
}
```

LIMITATIONS

The **init** process cannot be traced using the **LibSysCTr** library (ptrace limitation). Also, setuid binaries will be traced using the caller permissions, and not the suid ones (ptrace security constraint). The **strace**(1) command will not work when run from inside a shell monitored by **LibSysCTr**. Same thing for the **gdb**(1) debugger (and more in general for all debuggers using the **ptrace**(2) system call). Currently **LibSysCTr** supports only i386 CPUs, but it is easily extendible to other CPUs supported by the Linux OS (adding CPU support by extending the *sctr_linux.h* include file). If you do extend **LibSysCTr** support to other CPUs, please send *sctr_linux.h* patches to <**davidel@xmailserver.org**>.

LICENSE

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. A copy of the license is available at :

http://www.gnu.org/copyleft/lesser.html

AUTHOR

Developed by Davide Libenzi <**davidel@xmailserver.org**>

AVAILABILITY

The latest version of **LibSysCTr** can be found at :

http://www.xmailserver.org/sysctr-lib.html

BUGS

There are no known bugs. Bug reports and comments to Davide Libenzi <**davidel@xmailserver.org**>