The Design and Implementation of syscalltrack

http://syscalltrack.sf.net

Muli Ben-Yehuda

`mulix@mulix.org`

IBM Haifa Research Labs

TOC

Topics (Cont.)

- The Tree Parser
- Handling Errors
- Interesting Bugs and Technical Issues
- The Problems With Hijacking System Calls
- Handling Structure Parameters
- System Call Multiplexing
- The Future
- The Authors

General Overview - What Is syscalltrack?

- syscalltrack is a free software project, composed of Kernel modules and user utilities to allow filtering, logging and altering the invocation of system calls.
- syscalltrack was created to answer the question “which process did that?”, complementing strace’s “what did that process do?”.
- Currently supports filtering, logging, suspending or killing a process and failing a system call.
- Kernel 2.4.X fully supported, support for 2.6.X is a work in progress.

Architecture Overview

- 2 kernel modules:
  - A module to hijack system calls (sct_hijack).
  - A module to perform the actual filtering, and communicating with user space (sct_rules).
- A rules library (sct_rules), composing the filtering and action logic.
- A communications library (sct_ctrl_lib) allows user-processes to configure the module.
- User-mode utility parses the configuration file, validates the rules, and then deletes all existing rules in the module, and injects the new ones.
- Lots of tests - stress, regression, functionality.
- Various useful utilities (sctlog, sctrace, etc).

syscalltrack’s Kernel Modules

- syscalltrack contains 2 kernel modules - ’sct_hijack.o’ and ’sct_rules.o’.
- The former handles system call ’hijacking’ (that is, replacing a system call with another function). The latter does the filtering and takes appropriate action, and handles communication with user space.
- The split into two modules is done to to seperate mechanism (system call hijacking) from specifics (syscalltrack’s filtering and actions) and to avoid race conditions inherent to the use of modules in this capacity. More about this later.

The Hijacker Module

- This module exports syscall hijacking functions:

```
int hijack_syscall_before(int syscall_id, func_ptr function);
int hijack_syscall_after(int syscall_id, func_ptr function);

int release_syscall_before(int syscall_id);
int release_syscall_after(int syscall_id);
```

- Pointers to hijack stubs are inserted into the kernel’s system call table, ’sys_call_table’, instead of the pointer to the original system call. .
- Once a hijacked system call is called, the hijack stub will call a ’before hook’, the original syscall, and then an ’after hook’.
- For kernel 2.6, the hijack module is no longer a seperate module but rather built into the kernel.

The Filtering Module

- This module (sct_rules.o) accepts user-control messages, to add rules to a given syscall, delete a rule, print the rules, etc.
- When a system call is invoked, the before or after hook for this syscall is executed. This function matches the call's parameter and environment against all rules for this syscall, and if a match is found, an action is performed.

The Filtering Module (Cont.)

- For each system call, 2 sets of rules are kept - ’before’ rules, and ’after’ rules.
- ’before’ rules are checked right before invoking the system call. Thus, they could be used to disallow the syscall from being executed, or even alter parameters sent to the system call.
- ’after’ rules are checked right after the syscall returns, and before returning to the user. They allow checking and logging the syscall’s return value, altering this return value, and so on.

Communication With User-Space

- Communications with user-space is currently done by reading and writing from a special device file. Each supported command has its own ’constant’, and a single function then accepts these commands and handles them.
- We implemented our own protocol for these messages, including serialization of binary data and other fun stuff. We are (lazily) evaluating other options that do not involve reinventing a wheel, such as netlink or relayfs.

Code Auto-Generation

- Many functions in both kernel modules look very similar, and vary mostly by name/id of the system call they handle, and the parameters this system call receives.
- Instead of writing a lot of similar functions using copy/paste, a perl script generates the code for these functions. It does that by a combination of template files with macros, data types mappings and hard-coded constructs.
- This approach made sense in the early days of the project, where we were tasked with writing (and modifying!) over 200 such stubs for the different syscalls. Nowdays, with the integration of the hijack module into kernel 2.6, we are evaluating other options.

Kernel Fun - Module Unload Race

- When a module has one of its functions executed, or in the execution stack of a process/interrupt handler, unloading the module could crash the system - its code page is still in use, and yet might be re-allocated by the kernel for other purposes.
- Thus, a module writer must make sure that no active invocations of its functions exist when the module is unloaded, using for example MOD_INC_USE_COUNT.
- However, doing this from within the module is inherently racy.
- Solution? integrate the hijack module into the core kernel.

Kernel Fun - SMP And Re-Entrancy

- A process executing a system call might go to sleep, allowing another process to execute this system call. In SMP systems, even if a process does not go to sleep, its code might be executed in parallel in another process.
- To avoid races (and data structures corruption) inherent to such situations, data structures must be carefully protected, using semaphores (around sections that might sleep), spin-locks (around non-sleeping sections, to handle SMP machines) etc.

Kernel Fun - Locking

- However, over-use of locks would slow the kernel: either by forcing serial execution in cases where an SMP machine could actually work in parallel, or by introducing redundant overhead (locking and unlocking adds extra overhead).
- Due to “let’s do it simple now and optimize later” mentality, syscalltrack has a huge semaphore, “tracker_sem”, that serializes the filtering of different system calls. Getting rid of it is a prequisite before the 1.0 release (but requires a redesign of some of the most sensitive code...).

syscalltrack's Configuration - sct_config

- 'sct_config' is a C++ program that allows configuring the module. It is made of a parser (to read and understand a config file) and a commands generator (to generate data for commands for the kernel module).
- 'sct_config' is written using a combination of top-down and recursive-decent parsing.
- 'sct_config' may be used to perform other control operations - deleting all active rules, or printing them to the system's log file.

The Configuration File

The configuration file contains a list of rules. A rule might look like this:

```
rule
{
        syscall_name = unlink
        rule_name = passwd_unlink_rule
        filter_expression {PARAMS[1] == "passwd" }
        action {
                type = LOG
                log_format {syscall:%pid[%comm]:%sid_%sname(%params)
                            (rule %ruleid)}
        }
}
```

The Tree Parser

- ’sct_config’ reads the configuration file to memory and passes it to a parser.
- The parser parses the configuration file and for each keyword it recognizes, such as ’rule’, ’syscall_name’ or ’filter_expression’, it calls a parser for that keyword’s value.
- The value can be a simple token, such as ’before’, or a complex token such as the entire rule’s contents or a filter expression, like

  ```
  PID != 1 && PARAMS[1] in ("boo", "bee", "bah")
  ```

- The parser should one day be written using compiler construction tools, such as lex and yacc. It was fun to write by hand, but it’s a pain in the neck to maintain and extend.

Error Handling

- Since 'sct_config' is our user interface, it is imperative that it reports errors to the user clearly and concisely and not drown him in useless debugging data.
- Each function that encounters an error throws an exception with as much context information as it has. What line of the file was the error on, what was the token that caused the error, etc.
- Once thrown, an exception travels upwards in the call chain until a suitable exception handler is found. An exception handler could add information only it has to the exception and re-throw it, or it could report it to the user.

Interesting Bugs and Technical Issues

- Development of kernel modules (and even user-mode code) tends to expose various system bugs and "design obstacles". syscalltrack, being a non-standard project, seems to reveal quite a few of those.
- We'll illustrate a few of the more interesting/annoying ones here, to give one an impression.
- There were probably quite a few others, which we thankfully managed to suppress.

Hijacking System Calls

- There is no mechanism in Linux's kernel to hijack system calls, due to political reasons. Thus, hijacking them is done using various less-than-elegant techniques.
- The method we use means you locate the system call table - a table of pointers to all syscalls, mapped by syscall ID - copy a pointer to your own table, and replace the original pointer with a pointer to your "hijack stub" function.
- In 2.4, the system call table (sys_call_table) is exporter for modules to use, so there was no need for searching the kernel's memory directly. For 2.6, we refused to go searching in kernel memory, and instead moved the hijack module into the kernel, neatly sidestepping the propblem.

Hijacking System Calls (Cont.)

Of-course, you eventually should invoke the original system call, with the original parameters, or else the system breaks in most peculiar manners. It did...

Handling Structure Parameters

- Some syscalls accept pointers to structures as parameters. Since they are complex parameters, we needed a method to filter based on a specific field in a struct.
- The simplest way to do this was translating a struct into a vector, with each struct field as an item in that vector. ’sct_config’ translates struct type + field name into an index.
- The kernel module translates the struct into a vector, and uses that vector for later matching operations.
- This approach makes type-casting impossible. The debate is still out on how to support type-casting.

System Call Multiplexing

- Some "system calls" are actually multiplexers for several functions. For example, ’socketcall’ is a multiplexer for all socket-related functions: socket, accept, connect, listen, shutdown, recv, send.
- Handling those required special handling, since we wanted to expose the muxed functions (not the muxing syscall) to users.
- Our solution was to encode the function ID inside the syscall ID, and have the kernel module break the number down, and apply rules to sub-functions, rather then to the syscall itself.
- For that to work, we needed to copy the kernel’s mux syscall and make our own copy, that calls our functions, instead of the original system call.

The Future...

- Syscalltrack could divert to quite a few directions in the future, as our hearts desire.
- For instance, we intend to add support for altering the contents of parameters before invoking a syscall. This has quite a few usages, e.g. "fixing" programs for which we don’t have the source, or injecting faults into programs and seeing how they cope with them, etc.
- We could write an API that allows other modules to register rules that invoke callbacks in their functions - though this would definitely cause political wars in the kernel - unless we explicitly state this interface is GPLed - so we will.

The Future (Cont.)

- Perhaps even externalising system call activation into user-space, to allow for easier development of various types of features, and debugging them in user-space.
- Muli is playing with the idea (well, not his - its "stolen") of writing a module that sits on top of syscalltrack, learns patterns of use of the system, and later alerts if these patterns change. It has been done, but it's still a neat idea.

CREDITS

By alphabetical last name order:

- Orna Agmon <agmon@tx.technion.ac.il>
- Muli Ben-Yehuda <mulix@mulix.org>
- Gilad Ben-Yossef <gilad@benyossef.com>
- Shlomi Fish <shlomif@vipe.technion.ac.il>
- guy keren <choo@actcom.co.il>
- Itai Segall <spapuk@t2.technion.ac.il>
- Amir Shalem <amir@boom.org.il>
- Eli Shemer <apparitio@linuxmafia.org>
- Lior Ronen <LiorR@radlan.com>
- Ori Fine <Ori.Fine@comverse.com>
- David Landsberg <d_landsberg@yahoo.com>