strace and Lua

Viktor Krapivenskiy

Moscow Institute of Physics and Technology

23rd of September, 2017

Previously in strace

- strace is a useful diagnostic, instructional, and debugging tool. It intercepts and records the system calls which are called by a process and the signals which are received by a process.

Previously in strace

- strace is a useful diagnostic, instructional, and debugging tool. It intercepts and records the system calls which are called by a process and the signals which are received by a process.
- Within a GSoC 2016 project, strace was extended with the tampering capability, allowing the user to inject fake syscall results.

Previously in strace

- strace is a useful diagnostic, instructional, and debugging tool. It intercepts and records the system calls which are called by a process and the signals which are received by a process.
- Within a GSoC 2016 project, strace was extended with the tampering capability, allowing the user to inject fake syscall results.
- Tampering could be performed on a given set of syscalls, or only on those accessing a given set of paths; either for each syscall, only N-th one, or N-th one and then each K-th one.

Previously in strace

- strace is a useful diagnostic, instructional, and debugging tool. It intercepts and records the system calls which are called by a process and the signals which are received by a process.
- Within a GSoC 2016 project, strace was extended with the tampering capability, allowing the user to inject fake syscall results.
- Tampering could be performed on a given set of syscalls, or only on those accessing a given set of paths; either for each syscall, only N-th one, or N-th one and then each K-th one.
- Complex filtering logic or sematics-preserving success injection is impossible.

Can we do better?

- Lua is a powerful, efficient, lightweight, embeddable scripting language.

Can we do better?

- Lua is a powerful, efficient, lightweight, embeddable scripting language.
- LuaJIT is a Just-In-Time compiler for Lua that is considered to be "one of the fastest dynamic language implementations".

Can we do better?

- Lua is a powerful, efficient, lightweight, embeddable scripting language.
- LuaJIT is a Just-In-Time compiler for Lua that is considered to be "one of the fastest dynamic language implementations".
- LuaJIT comes with the FFI (foreign function interface) library that can parse plain C declarations (almost compatible with C99)!

Can we do better?

- Lua is a powerful, efficient, lightweight, embeddable scripting language.
- LuaJIT is a Just-In-Time compiler for Lua that is considered to be "one of the fastest dynamic language implementations".
- LuaJIT comes with the FFI (foreign function interface) library that can parse plain C declarations (almost compatible with C99)!
- It can also create and manipulate boxed C objects of known types.

Can we do better?

- Lua is a powerful, efficient, lightweight, embeddable scripting language.
- LuaJIT is a Just-In-Time compiler for Lua that is considered to be "one of the fastest dynamic language implementations".
- LuaJIT comes with the FFI (foreign function interface) library that can parse plain C declarations (almost compatible with C99)!
- It can also create and manipulate boxed C objects of known types.
- Functions like **typeof**, **sizeof**, **alignof**, **offsetof**, **istype** etc; implicit conversion between native Lua types and boxed C values.
- **No hand-holding!**

FFI?

```
ffi = require 'ffi'
ffi.cdef[[
// available as ffi.C.printf
int printf(const char *fmt, ...);

// a boxed object can be created with, e.g.,
// ffi.new('struct my_struct')
struct my_struct {
    int a;
    uint64_t b; // a number of types are pre-defined
};

// available as ffi.C.MY_CONSTANT
enum { MY_CONSTANT = 42 };

// available as ffi.C.ANOTHER_CONSTANT
const static int ANOTHER_CONSTANT = 84;
]]
```

The mechanism (1/3)

```
/* typedefs for kernel_[u]long_t are provided to FFI,
   as well as definitions for some other structures */

struct tcb {
    int flags; /* Not documented as a part of the
                * interface, but used by helper library */
    int pid; /* Tracee's PID */
    int qual_flg; /* Just like the ::flags field */
    unsigned long u_error; /* Error code */
    kernel_ulong_t scno; /* Syscall number */
    /* MAX_ARGS gets expanded before feeding it to FFI */
    kernel_ulong_t u_arg[MAX_ARGS]; /* Syscall args */
    kernel_long_t u_rval; /* Syscall return value */

/* That's it for FFI's definition of struct tcb, but not
 * for strace's once! */
```

The mechanism (2/3)

- strace now maintains two additional (to the tracing options sets) sets per personality: a set of syscalls that need to be returned to Lua on syscall entry, and those that need to be returned on syscall exit.
- Initially, both of them are empty.

The mechanism (2/3)

- strace now maintains two additional (to the tracing options sets) sets per personality: a set of syscalls that need to be returned to Lua on syscall entry, and those that need to be returned on syscall exit.
- Initially, both of them are empty.
- `bool monitor(unsigned scno, unsigned pers, bool on_entry, bool on_exit)` — marks syscall with number `scno` on personality `pers` as to be returned from `next_sc`;
    - Exposed as `strace.C.monitor`.

The mechanism (2/3)

- strace now maintains two additional (to the tracing options sets) sets per personality: a set of syscalls that need to be returned to Lua on syscall entry, and those that need to be returned on syscall exit.
- Initially, both of them are empty.
- `bool monitor(unsigned scno, unsigned pers, bool on_entry, bool on_exit)` — marks syscall with number `scno` on personality `pers` as to be returned from `next_sc`;
    - Exposed as `strace.C.monitor`.
- `void monitor_all(bool on_entry, bool on_exit)` — marks all syscalls as to be returned from `next_sc`;
    - Exposed as `strace.C.monitor_all`.

The mechanism (2/3)

- strace now maintains two additional (to the tracing options sets) sets per personality: a set of syscalls that need to be returned to Lua on syscall entry, and those that need to be returned on syscall exit.
- Initially, both of them are empty.
- `bool monitor(unsigned scno, unsigned pers, bool on_entry, bool on_exit)` — marks syscall with number `scno` on personality `pers` as to be returned from `next_sc`;
  - Exposed as `strace.C.monitor`.
- `void monitor_all(bool on_entry, bool on_exit)` — marks all syscalls as to be returned from `next_sc`;
  - Exposed as `strace.C.monitor_all`.
- `struct tcb * next_sc(void)` — returns either a pointer to the trace control block of the next syscall event being monitored, or a null pointer if strace needs to be terminated.

The mechanism (2/3)

- strace now maintains two additional (to the tracing options sets) sets per personality: a set of syscalls that need to be returned to Lua on syscall entry, and those that need to be returned on syscall exit.
- Initially, both of them are empty.
- `bool monitor(unsigned scno, unsigned pers, bool on_entry, bool on_exit)` — marks syscall with number `scno` on personality `pers` as to be returned from `next_sc`;
  - Exposed as `strace.C.monitor`.
- `void monitor_all(bool on_entry, bool on_exit)` — marks all syscalls as to be returned from `next_sc`;
  - Exposed as `strace.C.monitor_all`.
- `struct tcb * next_sc(void)` — returns either a pointer to the trace control block of the next syscall event being monitored, or a null pointer if strace needs to be terminated.
  - Not exposed directly; `strace.next_sc` is a (thin) wrapper that saves the result to a library-local variable; and returns `nil` if it was a null pointer.
  - To protect the user from dereferencing a null pointer.

The mechanism (3/3)

- More C functions (for performing injection, reading and writing memory, using strace's path-matching facilities) are exposed through the `strace.C` namespace.
- C constants (sets of syscall info entries, signals, errors and `ioctl` entries per personality) are also exposed through the `strace.C` namespace.

The mechanism (3/3)

- More C functions (for performing injection, reading and writing memory, using strace's path-matching facilities) are exposed through the `strace.C` namespace.
- C constants (sets of syscall info entries, signals, errors and `ioctl` entries per personality) are also exposed through the `strace.C` namespace.
- **The helper library** written in Lua provides convenience wrappers around the low-level C interface, as well as a push-style hooking API.

The helper library

Note: all the functions work on the current syscall event.

- `strace.entering()`, `strace.exiting()` — is this a syscall entry, or an exit?

The helper library

Note: all the functions work on the current syscall event.

- `strace.entering()`, `strace.exiting()` — is this a syscall entry, or an exit?
- `strace.trace([flag])`, `strace.abbrev([flag])`, `strace.verbose([flag])`, `strace.raw([flag])` — filtering and altering trace options (`flag` defaults to true).

The helper library

Note: all the functions work on the current syscall event.

- `strace.entering()`, `strace.exiting()` — is this a syscall entry, or an exit?
- `strace.trace([flag])`, `strace.abbrev([flag])`, `strace.verbose([flag])`, `strace.raw([flag])` — filtering and altering trace options (`flag` defaults to true).
- Get syscall/signal/error/`ioctl` request number by its name, or vice versa.

The helper library

Note: all the functions work on the current syscall event.

- `strace.entering()`, `strace.exiting()` — is this a syscall entry, or an exit?
- `strace.trace([flag])`, `strace.abbrev([flag])`, `strace.verbose([flag])`, `strace.raw([flag])` — filtering and altering trace options (`flag` defaults to true).
- Get syscall/signal/error/`ioctl` request number by its name, or vice versa.
- `strace.inject_signal(sig)`, `strace.inject_error(err)` — inject a signal or an error by its name or number.

The helper library

Note: all the functions work on the current syscall event.

- `strace.entering()`, `strace.exiting()` — is this a syscall entry, or an exit?
- `strace.trace([flag])`, `strace.abbrev([flag])`, `strace.verbose([flag])`, `strace.raw([flag])` — filtering and altering trace options (`flag` defaults to true).
- Get syscall/signal/error/`ioctl` request number by its name, or vice versa.
- `strace.inject_signal(sig)`, `strace.inject_error(err)` — inject a signal or an error by its name or number.
- `strace.read_obj(addr, ct[, nlelem])`, `strace.write_obj(addr, obj)` — read or write a FFI object from/to the tracee's memory at the given address (`ct[, nelem]` define a C type to read).

The helper library

Note: all the functions work on the current syscall event.

- `strace.entering()`, `strace.exiting()` — is this a syscall entry, or an exit?
- `strace.trace([flag])`, `strace.abbrev([flag])`, `strace.verbose([flag])`, `strace.raw([flag])` — filtering and altering trace options (`flag` defaults to true).
- Get syscall/signal/error/`ioctl` request number by its name, or vice versa.
- `strace.inject_signal(sig)`, `strace.inject_error(err)` — inject a signal or an error by its name or number.
- `strace.read_obj(addr, ct[, nlelem])`, `strace.write_obj(addr, obj)` — read or write a FFI object from/to the tracee's memory at the given address (`ct[, nelem]` define a C type to read).
- `strace.read_str(addr[, maxsz[, bufsz]])`, `strace.read_path(addr)` — read a C string or a path C string.

The helper library

Note: all the functions work on the current syscall event.

- `strace.entering()`, `strace.exiting()` — is this a syscall entry, or an exit?
- `strace.trace([flag])`, `strace.abbrev([flag])`, `strace.verbose([flag])`, `strace.raw([flag])` — filtering and altering trace options (`flag` defaults to true).
- Get syscall/signal/error/`ioctl` request number by its name, or vice versa.
- `strace.inject_signal(sig)`, `strace.inject_error(err)` — inject a signal or an error by its name or number.
- `strace.read_obj(addr, ct[, nlelem])`, `strace.write_obj(addr, obj)` — read or write a FFI object from/to the tracee's memory at the given address (`ct[, nelem]` define a C type to read).
- `strace.read_str(addr[, maxsz[, bufsz]])`, `strace.read_path(addr)` — read a C string or a path C string.
- **Hooks.**

Hooks

Note: `when` argument is either `"entering"`, `"exiting"`, or `"both"`.

- `strace.hook(scname, when, callback)` — by syscall name.

Hooks

Note: `when` argument is either `"entering"`, `"exiting"`, or `"both"`.

- `strace.hook(scname, when, callback)` — by syscall name.
- `strace.hook_class(clsname, when, callback)` — by syscall class.

Hooks

Note: `when` argument is either `"entering"`, `"exiting"`, or `"both"`.

- `strace.hook(scname, when, callback)` — by syscall name.
- `strace.hook_class(clsname, when, callback)` — by syscall class.
- `strace.hook_scno(scno, pers, when, callback)` – by syscall number and personality number.

Hooks

Note: `when` argument is either `"entering"`, `"exiting"`, or `"both"`.

- `strace.hook(scname, when, callback)` — by syscall name.
- `strace.hook_class(clsname, when, callback)` — by syscall class.
- `strace.hook_scno(scno, pers, when, callback)` – by syscall number and personality number.
- `strace.at_exit(callback)` – at exit.

Example: counting number of processes spawned

```
n = 0
assert(strace.hook({'clone', 'fork', 'vfork'}, 'exiting',
function(tcp)
    if tcp.u_rval ~= -1 then
        n = n + 1
    end
end))
strace.at_exit(function() print('Processes spawned:', n) end)
```

Example: using external preprocessor (1/2)

```
ffi = require 'ffi'
f = assert(io.popen([[cpp - <<EOF | grep -v '^#'

#define _GNU_SOURCE
#include <fcntl.h>
enum { f_setpipe_sz = F_SETPIPE_SZ };

EOF]], 'r'))
ffi.cdef(f:read('*a'))
f:close()

assert(strace.hook({'fcntl', 'fcntl64'}, 'entering',
function(tcp)
    if tcp.u_arg[1] == ffi.C.f_setpipe_sz then
        assert(strace.inject_error('EPERM'))
    end
end))
```

Example: using external preprocessor (2/2)

```
ffi = require 'ffi'
f = assert(io.popen([[cpp - <<EOF | grep -v '^#'
#include <sys/utsname.h>
EOF]], 'r'))
ffi.cdef(f:read('*a'))
f:close()

assert(strace.hook('uname', 'exiting', function(tcp)
    if tcp.u_rval == -1 then
        return
    end

    local u = assert(strace.read_obj(tcp.u_arg[0], 'struct utsname'))

    local s = 'Windows'
    assert(ffi.sizeof(u.sysname) >= #s + 1)
    ffi.copy(u.sysname, s)

    assert(strace.write_obj(tcp.u_arg[0], u))
end))
```

Example: using external preprocessor (2/2)

```
$ uname
Linux
$ strace -l pretend-win.lua -e none uname
Windows
+++ exited with 0 +++
```

Example: using ffiex library

```
ffiex = require 'ffiex'
ffiex.cdef('#include <sys/wait.h>')
function is_truthy(x) return x and x ~= 0 end
stats = {}
assert(strace.hook({'waitpid', 'wait4', 'osf_wait4'}, 'exiting',
function(tcp)
    if tcp.u_rval == -1 or tcp.u_rval == 0 or tcp.u_arg[1] == 0 then
        return
    end
    local status = tonumber(assert(strace.read_obj(tcp.u_arg[1],
        'int')))
    if is_truthy(ffiex.defs.WIFEXITED(status)) then
        local c = ffiex.defs.WEXITSTATUS(status)
        stats[c] = (stats[c] or 0) + 1
    end
end))
strace.at_exit(function()
    print('Exit codes:')
    for k, v in pairs(stats) do print(k .. ':', v) end
end)
```

Project status

Not merged yet.