CUJOAI

Scripting Linux system calls with Lua

Lua Workshop 2018
Pedro Tammela
CUJO AI

Scripting system calls

- Customizing system calls at the kernel level
- Why bother?
- “Through scripts, users can adapt the operating system behavior to their demands, defining appropriate policies and mechanisms.” (Vieira et al. 2014)

Existing solutions

- eBPF
  - “One of the more interesting features in this cycle is the ability to attach eBPF programs (user-defined, sandboxed bytecode executed by the kernel) to kprobes. This allows user-defined instrumentation on a live kernel image that can never crash, hang or interfere with the kernel negatively.” (Ingo Molnar, 2015)

Existing solutions

- eBPF took a broader approach (BPF Compiler Collection)
  - "Any" programming language to eBPF byte-code
- “The universal in-kernel virtual machine” - LWN.net
- eBPF is extremely popular in tracing applications

Existing solutions

- Lunatik (Lua in Kernel for Linux)
  - Lua in Kernel is actually older than eBPF! (2010-2011)
- Common use cases:
  - Packet filtering, Tracing
- Lua in Kernel and eBPF are long lost siblings

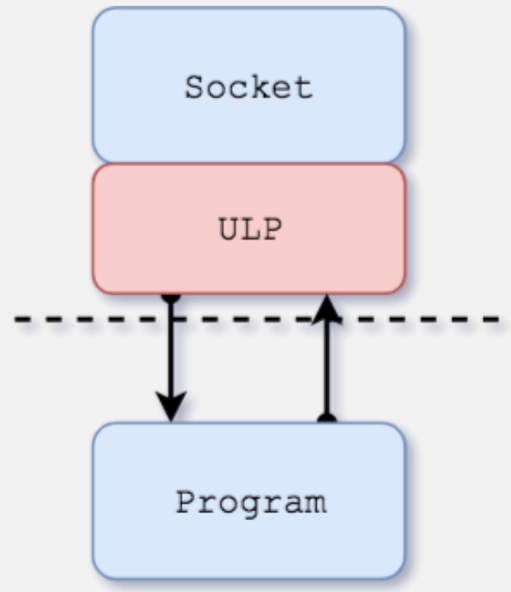
Why Lua?

- Already ported to various kernels (NetBSD, Linux…)
- Lua C API
- Simple but a complete language
- Making a kernel scriptable with a single kernel module

Extending

- Linux provides an Upper Layer Protocol architecture for extending network system calls
- Created for the TLS in kernel feature
- Supports only TCP (officially)

The Upper Layer Protocol



- Write your own socket system calls
- "Raw access" to the socket internal structure
- What would be interesting to do?
  - HTTP header analysis (CRLF injection, spurious fields...)
  - Layer 4 pre-processing (TLS)
  - Cached responses

Lua as an ULP

- Activated and controlled via *setsockopt()*
- Lua scripts are transferred to the kernel using *setsockopt()*
- Every internal socket structure has it’s own Lua state

Initializing

```c
setsockopt(sock, SOL_TCP, TCP_ULP, "lua", sizeof("lua"));
```

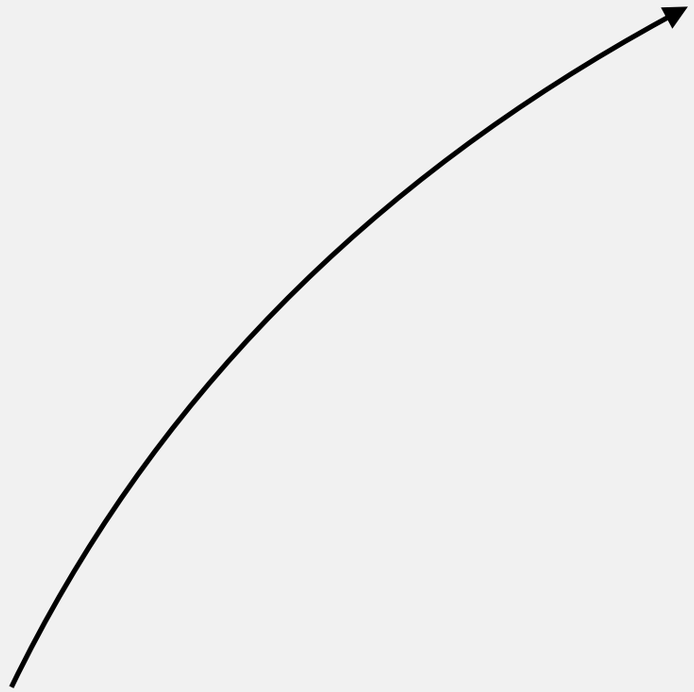
```c
static int ss_tcp_init(struct sock *sk)
{
   /* … */
   sys = sk->sk_prot;
   if (sk->sk_family == AF_INET)
      sk->sk_prot = &tcpssv4;
   else
      sk->sk_prot = &tcpssv6;

   return 0;
}

static struct tcp_ulp_ops ss_tcpulp_ops
__read_mostly = {
   .name           = "lua",
   .uid            = TCP_ULP_LUA,
   .user_visible   = true,
   .owner          = THIS_MODULE,
   .init           = ss_tcp_init
};

static int __init ss_tcp_register(void)
{
   /* … */
   tcp_register_ulp(&ss_tcpulp_ops);
   return 0;
}

static void __exit ss_tcp_unregister(void)
{
   tcp_unregister_ulp(&ss_tcpulp_ops);
}

module_init(ss_tcp_register);
module_exit(ss_tcp_unregister);
```

Initializing

```
setsockopt(sock, SOL_TCP, TCP_ULP, "lua", sizeof("lua"));
```

```
static int ss_tcp_init(struct sock *sk)
{
    /* … */
    sys = sk->sk_prot;
    if (sk->sk_family == AF_INET)
        sk->sk_prot = &tcpssv4;
    else
        sk->sk_prot = &tcpssv6;

    return 0;
}

static struct tcp_ulp_ops ss_tcpulp_ops
__read_mostly = {
    .name           = "lua",
    .uid            = TCP_ULP_LUA,
    .user_visible   = true,
    .owner          = THIS_MODULE,
    .init           = ss_tcp_init
};

static int __init ss_tcp_register(void)
{
    /* … */
    tcp_register_ulp(&ss_tcpulp_ops);
    return 0;
}

static void __exit ss_tcp_unregister(void)
{
    tcp_unregister_ulp(&ss_tcpulp_ops);
}

module_init(ss_tcp_register);
module_exit(ss_tcp_unregister);
```

Loading Scripts



```
setsockopt(sock, SOL_LUA, SS_LUA_LOADSCRIPT, buff, sz);
```

```
static int ss_setsockopt(struct sock *sk, int level, int
optname,
      char __user *optval, unsigned int optlen)
{
   /* ... */
   if (level != SOL_LUA)
      return sys->setsockopt(sk, level, optname, optval,
optlen);

   switch (optname) {
      case SS_LUA_LOADSCRIPT: {
         lua_State *L = SS_LUA_STATE(sk);
         int stack = lua_gettop(L);
         char *script;

         if (!optval || optlen > SS_SCRIPTSZ)
            return -EINVAL;

         script = kmalloc(optlen, GFP_KERNEL);
         if (script == NULL)
            return -ENOMEM;

         err = copy_from_user(script, optval, optlen);
         if (unlikely(err))
            return -EFAULT;

         if (luaL_loadbufferx(L, script, optlen, "lua", "t")
               || lua_pcall(L, 0, 0, 0)) {
            pr_err("%s\n", lua_tostring(L, -1));
            lua_settop(L, stack);
            return -EINVAL;
         }

         break;
      }
   }
   /* ... */
   return 0;
}
```

Loading Scripts

Copies the script from user space

```
setsockopt(sock, SOL_LUA, SS_LUA_LOADSCRIPT, buff, sz);
```

```
static int ss_setsockopt(struct sock *sk, int level, int
optname,
      char __user *optval, unsigned int optlen)
{
   /* ... */
   if (level != SOL_LUA)
      return sys->setsockopt(sk, level, optname, optval,
optlen);

   switch (optname) {
      case SS_LUA_LOADSCRIPT: {
         lua_State *L = SS_LUA_STATE(sk);
         int stack = lua_gettop(L);
         char *script;

         if (!optval || optlen > SS_SCRIPTSZ)
            return -EINVAL;

         script = kmalloc(optlen, GFP_KERNEL);
         if (script == NULL)
            return -ENOMEM;

         err = copy_from_user(script, optval, optlen);
         if (unlikely(err))
            return -EFAULT;

         if (luaL_loadbufferx(L, script, optlen, "lua", "t")
               || lua_pcall(L, 0, 0, 0)) {
            pr_err("%s\n", lua_tostring(L, -1));
            lua_settop(L, stack);
            return -EINVAL;
         }

         break;
      }
   }
   /* ... */
   return 0;
}
```

Loading Scripts

```
setsockopt(sock, SOL_LUA, SS_LUA_LOADSCRIPT, buff, sz);
```

Loads the script in a Lua state

```
static int ss_setsockopt(struct sock *sk, int level, int
optname,
      char __user *optval, unsigned int optlen)
{
   /* ... */
   if (level != SOL_LUA)
      return sys->setsockopt(sk, level, optname, optval,
optlen);

   switch (optname) {
      case SS_LUA_LOADSCRIPT: {
         lua_State *L = SS_LUA_STATE(sk);
         int stack = lua_gettop(L);
         char *script;

         if (!optval || optlen > SS_SCRIPTSZ)
            return -EINVAL;

         script = kmalloc(optlen, GFP_KERNEL);
         if (script == NULL)
            return -ENOMEM;

         err = copy_from_user(script, optval, optlen);
         if (unlikely(err))
            return -EFAULT;

         if (luaL_loadbufferx(L, script, optlen, "lua", "t")
               || lua_pcall(L, 0, 0, 0)) {
            pr_err("%s\n", lua_tostring(L, -1));
            lua_settop(L, stack);
            return -EINVAL;
         }

         break;
      }
   }
   /* ... */
   return 0;
}
```

Lua as an ULP

- Messages are preprocessed by the kernel using Lua

- The *recvmsg()* system call uses an Lua entry point defined by the user application

Socket messages

```c
size_t msgsz = recv(sock, msg, 8192, 0);
```

```c
static int ss_recvmsg(struct sock *sk, struct msghdr *msg,
size_t len, int nonblock, int flags, int *addr_len)
{
   /* … */
   err = sys->recvmsg(sk, msg, len, nonblock, flags, addr_len);
   if (err < 0)
      goto out;

   /* … */

   /* skip Lua processing */
   if (ctx->entry[0] == '\0')
      goto out;

   lock_sock(sk);
   /* … */
   baseref = ldata_newref(L, ubuff, size);
   lua_pushinteger(L, (lua_Integer) size);
   lua_pushboolean(L, nonblock);
   perr = lua_pcall(L, 3, 1, 0);
   ldata_unref(L, baseref);
   if (perr) {
      pr_err("%s\n", lua_tostring(L, -1));
      goto outlua;
   }

   trash = lua_toboolean(L, -1);
   if (trash) {
      err = 0;
      copy_to_user(ubuff, &err, sizeof(int));
   }

outlua:
   release_sock(sk);
   lua_settop(L, stack);
out:
   return err;
}
```

Socket messages

Calls the system's recvmsg

```c
size_t msgsz = recv(sock, msg, 8192, 0);
```

```c
static int ss_recvmsg(struct sock *sk, struct msghdr *msg,
size_t len, int nonblock, int flags, int *addr_len)
{
   /* … */
   err = sys->recvmsg(sk, msg, len, nonblock, flags, addr_len);
   if (err < 0)
      goto out;

   /* … */

   /* skip Lua processing */
   if (ctx->entry[0] == '\0')
      goto out;

   lock_sock(sk);
   /* … */
   baseref = ldata_newref(L, ubuff, size);
   lua_pushinteger(L, (lua_Integer) size);
   lua_pushboolean(L, nonblock);
   perr = lua_pcall(L, 3, 1, 0);
   ldata_unref(L, baseref);
   if (perr) {
      pr_err("%s\n", lua_tostring(L, -1));
      goto outlua;
   }

   trash = lua_toboolean(L, -1);
   if (trash) {
      err = 0;
      copy_to_user(ubuff, &err, sizeof(int));
   }

outlua:
   release_sock(sk);
   lua_settop(L, stack);
out:
   return err;
}
```

Socket messages

Does processing with Lua

```
size_t msgsz = recv(sock, msg, 8192, 0);
```

```
static int ss_recvmsg(struct sock *sk, struct msghdr *msg,
size_t len, int nonblock, int flags, int *addr_len)
{
   /* … */
   err = sys->recvmsg(sk, msg, len, nonblock, flags, addr_len);
   if (err < 0)
      goto out;

   /* … */

   /* skip Lua processing */
   if (ctx->entry[0] == '\0')
      goto out;

   lock_sock(sk);
   /* … */
   baseref = ldata_newref(L, ubuff, size);
   lua_pushinteger(L, (lua_Integer) size);
   lua_pushboolean(L, nonblock);
   perr = lua_pcall(L, 3, 1, 0);
   ldata_unref(L, baseref);
   if (perr) {
      pr_err("%s\n", lua_tostring(L, -1));
      goto outlua;
   }

   trash = lua_toboolean(L, -1);
   if (trash) {
      err = 0;
      copy_to_user(ubuff, &err, sizeof(int));
   }

outlua:
   release_sock(sk);
   lua_settop(L, stack);
out:
   return err;
}
```

Final Remarks

- A step closer to a customizable OS Kernel in run time
  - eBPF, Lua in Kernel, etc...
- An old idea (Lampson 1969)
- Some questions yet to be studied:
  - What about the other system calls?
  - How much does it cost?
  - eBPF and Lua in Kernel, which path?

Thank you!

Pedro Tammela

https://www.pedrotammela.com