Binding to YARA with LuaJIT
$ whoami

- Team Lead @ Kong
- OpenResty/ModSecurity contributor
- Previous Dreamhost, Zenedge
YARA Overview

- Pattern-matching Swiss Army knife
- Used in malware research
  - More generically, used in pattern description
- Multi platform
- Data-driven design, simple, extensible DSL
YARA Overview

• Rules consist of a set of strings and a boolean expression

• Flexible match expressions and modeling
  – Case-insensitive strings
  – Wildcards
  – Regular expressions
  – Custom match modules
  – Textual or binary data
YARA Rules

- Human readable DSL compiles down to binary
- Text or hex strings
- Internal regular expression engine
  - No capture groups or POSIX character classes
  - No backtracking (no lookarounds)
- Condition clauses determines match state
YARA Rules

• Metadata
  – Tags
  – Namespaces
  – Arbitrary identifiers

• Compiled down with `yarac` CLI tool
  – Uses the YARA C API under the hood
  – Multiple rulesets can compile to a single binary ruleset
YARA Rules

- Miscellaneous match conditions
  - File size
  - String repetition, match count
  - Executable entry point (PE, ELF)
  - MIME type ($ file)
  - Byte offset
  - Rule reference

- Module development API
Example rules

/*
   Finds PHP code in JP(E)Gs, GIFs, PNGs.
   Magic numbers via Wikipedia.
*/
rule php_in_image
{
  meta:
    author = "Vlad https://github.com/vlad-s"
    date  = "2016/07/18"
    description = "Finds image files w/ PHP code in images"
  strings:
    $gif = /^GIF8[79]a/ 
    $jfif = { ff d8 ff e? 00 10 4a 46 49 46 } 
    $png = { 89 50 4e 47 0d 0a 1a 0a }

    $php_tag = "<?php"
  condition:
    (($gif at 0) or
    ($jfif at 0) or
    ($png at 0)) and

    $php_tag
}
rule Linux_DirtyCow_Exploit {
    meta:
        description = "Detects Linux Dirty Cow Exploit - CVE-2012-0056 and CVE-2016-5195"
        author = "Florian Roth"
        reference = "http://dirtycow.ninja/"
        date = "2016-10-21"
    strings:
        $a1 = { 48 89 D6 41 B9 00 00 00 00 41 89 C0 B9 02 00 00 00 BA 01 00 00 BF 00 00 00 00 }
        $b1 = { E8 ?? FC FF FF 48 8B 45 E8 BE 00 00 00 00 48 89 C7 E8 ?? FC FF FF 48 8B 45 F0 BE 00 00 00 00 48 89 } 
        $b2 = { E8 ?? FC FF FF B8 00 00 00 00 }
        $source1 = "madvise(map,100,MADV_DONTNEED);"
        $source2 = "=open("/proc/self/mem",O_RDWR);"
        $source3 = ",map SEEK_SET);"
        $source_printf1 = "mmap %x"
        $source_printf2 = "procselfmem %d"
        $source_printf3 = "madvise %d"
        $source_printf4 = "[-] failed to patch payload"
        $source_printf5 = "[-] failed to win race condition..."
        $source_printf6 = "[*] waiting for reverse connect shell..."
        $s1 = "/proc/self/mem"
        $s2 = "/proc/%d/mem"
        $s3 = "/proc/self/map"
        $s4 = "/proc/%d/map"
        $p1 = "pthread_create" fullword ascii
        $p2 = "pthread_join" fullword ascii
    condition:
        ( uint16(0) == 0x457f and $a1 ) or
        all of ($b*) or
        3 of ($source*) or
        ( uint16(0) == 0x457f and 1 of ($s*) and all of ($p*) and filesize < 20KB )
}
YARA Rules

• Additional match functionality
  – Iterating over string occurrences
  – Rule references
  – External data references
rule Occurrences
{
  strings:
    $a = "dummy1"
    $b = "dummy2"

  condition:
    for all i in (1,2,3) : ( @a[i] + 10 == @b[i] )
}
rule Rule1
{
    strings:
        $a = "dummy1"

    condition:
        $a
}

rule Rule2
{
    strings:
        $a = "dummy2"

    condition:
        $a and Rule1
}
rule ExternalVariableExample1
{
    condition:
    ext_var == 10
}

rule ExternalVariableExample2
{
    condition:
    bool_ext_var or filesize < int_ext_var
}

rule ExternalVariableExample3
{
    condition:
    string_ext_var contains "text"
}

rule ExternalVariableExample4
{
    condition:
    string_ext_var matches /^[a-z]+$/
}
YARA Usage

• Command line tooling
  
  $ yara /path/to/rules /path/to/target

• Can use precompiled or textual rulesets

• CLI options
  – Recursive scanning
  – Filter target matches
  – Import custom module data/metadata
YARA Usage

• API Bindings
  – Python
  – Ruby
  – Go
  – Java
  – Lua! :D
Use Case

• In-line reverse proxy scanning
  – File uploads
  – Scanning large in-memory request bodies

• Leverage existing research datasets

• Typically virus scanning is out of band
  – Expensive
  – Passive detection
Usage Design Goals

• Leverage existing edge tier platform
  – OpenResty/LuaJIT
  – Existing development/module environment
  – Integrate C API via FFI

• Maximize performance
  – Use pre-compiled rules
  – Benchmark with existing OR tooling (stapp++ and friends)
  – Avoid buffering to disk when possible
Usage Design Goals

• Use efficient rule design
  – Regular expressions are slow; flat strings are faster (byte offsets even better)
  – Case insensitive string definitions
  – Avoid short (<6 byte) strings
  – Base designs on existing rule sets

• Expose useful results to SOC/logging
C API

- YARA is an API, not a command line tool
  - Initialization
  - Compiling rules
  - Scanning data

- Multiple ways to scan
  - Memory buffer
  - On-disk file
  - File descriptor
C API

int yr_rules_scan_mem(YR_RULES* rules,
                        uint8_t* buffer,
                        size_t buffer_size,
                        int flags,
                        YR_CALLBACK_FUNC callback,
                        void* user_data,
                        int timeout)
C API

- Problematic design
  - Every rule execution generates a callback
  - Push-style API

```c
int callback_function(
    int message, // flag
    void* message_data,
    void* user_data);
```
LuaJIT FFI Callbacks

Callback performance

Callbacks are slow! First, the C to Lua transition itself has an unavoidable cost, similar to a `lua_call()` or `lua_pcall()`. Argument and result marshalling add to that cost. And finally, neither the C compiler nor LuaJIT can inline or optimize across the language barrier and hoist repeated computations out of a callback function.

Do not use callbacks for performance-sensitive work: e.g. consider a numerical integration routine which takes a user-defined function to integrate over. It's a bad idea to call a user-defined Lua function from C code millions of times. The callback overhead will be absolutely detrimental for performance.
C API

• Design crossroads
  – Use the API in a hot path
  – Can’t (shouldn’t) use callbacks in hot paths
  – API usage demands callbacks

• Solution: don’t use callbacks!
Wrapping the YARA API

- Obscure the C ↔ Lua translation

- C wrapper for YARA C API
  - Handle callbacks internally
  - Statically compile in YARA symbols
  - Single call from Lua → C
  - No more push-style design!
Wrapping the YARA API

- C functions to wrap the whole YARA API
  - YARA initialization
  - Load rulesets
  - Scan mem/file path
  - Finalize
  - Handle each rule execution callback
  - Pass state/match data back to caller
Wrapping the YARA API

typedef struct yawrap_match_s {
    char *msg;
    struct yawrap_match_s* next;
} yawrap_match_t;

typedef struct yarawrap_user_data_s {
    unsigned int count:7;
    unsigned int multi_cap:1;
    yawrap_match_t* head;
} yawrap_user_data_t;
Calling the Wrapper

- Integrate into OpenResty
  - Simple Lua interface
  - Return match status, matched rules name(s)
  - Handle disk-buffered request bodies?
    - Integrating into existing infrastructure
    - Tuning request buffering becomes complex
access_by_lua_block {  
   local yara = require "yara"

   local messages = {} -- table to hold matched rule data
   local rulepath = "/path/to/compiled/yara.rules"

   ngx.req.read_body()
   local body = ngx.req.get_body_file()

   local matched = yara.inspect_file(body, rulepath, messages)

   if matched then
      for i, rule_name in ipairs(matches) do
         ngx.say(rule_name)
      end
   end
}

Calling the Wrapper
Design Improvements

• Expand returned metadata
  – Tags
  – Rule metadata (arbitrary values)
  – Execution timing

• Preallocate user data buffer
  – On-demand malloc() of each yawrap_match_t
  – Requires free() of each elt as well
    • Currently done in Lua :(
-- match_type = ffi.typeof("yawrap_match_t[1]")

-- we had a match, walk the list, logging and freeing each match
if user_data.count > 0 then
    matched = true

    local match_ptr = user_data.head

    -- tmp pointer so we can free each element in the list
    local tmp_ref = ffi.new(match_type)
    local tmp = tmp_ref[0]

    while true do
        table.insert(matches, ffi.string(match_ptr.msg))

        -- free the msg ptr in the match, then free the match itself
        ffi.C.free(match_ptr.msg)
        tmp = match_ptr
        match_ptr = match_ptr.next
        ffi.C.free(tmp)

        if match_ptr == nil then break end
    end
end
Design Improvements

- Filter match execution
  - Tags
  - Metadata
  - Import external variables

- Additional CLI feature parity
  - SCAN_FLAGS_FAST_MODE
  - Configurable stack size
  - Custom module data
$ yara -t Packer /path/to/rules bazfile

$ yara -d mybool=true \ 
   -d myint=5 \ 
   -d mystring="my string" \ 
/foo/bar/rules bazfile
Design Improvements

- Wrapper to scan file descriptor
- Wrappers/bindings for rule compilation
  - Current design calls for rule compilation to be done out-of-band, deployed as separate part of pipeline
Questions?