Instrument and Find Out

Writing Parasitic Tracers for High(-Level) Languages

Jeff Dileo
@chaosdatumz

DEF CON 29

nccgroup

```
call_usermodehelper("/bin/sh",(char*[]){"/bin/sh","-c","whoami",NULL},NULL,5)
```

- @chaosdatumz
- Agent of chaos
- Unix aficionado
- Technical Director / Research Director @ NCC Group
- I like to do terrible things to/with/in:
  - programs
  - languages
  - runtimes
  - memory
  - kernels
  - packets
  - bytes
  - ...

nccgroup

NOTICE

By viewing this presentation you agree to indemnify and hold harmless the presenter in the event that you decide to take any of his advice and find yourself unable to sleep at 4am in the morning due to “language demons.”

Outline

- Background
- Parasitic Tracers
- Designing Parasitic Tracers (for high-level languages)
- Case Study: Ruby
- Conclusion

nccgroup

Background — Tracing and Me

- I've done a lot of work with dynamic instrumentation and tracing
  - Java Bytecode
  - Android (Frida, Xposed)
  - Linux (eBPF, Frida)

- Generally, to aid in reversing (e.g. dump process state) or to util-ize existing things

Background — Tracing and Me

- I’ve done a lot of work with dynamic instrumentation and tracing
  - Java Bytecode
  - Android (Frida, Xposed)
  - Linux (eBPF, Frida)
- Generally, to aid in reversing (e.g. dump process state) or to util-ize existing things
- ”Dynamic Instrumentation”
  - Function hooking
  - Instruction instrumentation (assembly, bytecode, etc.)
- ”Dynamic Tracing”
  - Dynamically enabling/disabling existing logging functionality
  - Dynamically adding enhanced logging functionality that wasn’t there before

nccgroup

Background — Tracing and Me

- I’ve done a lot of work with dynamic instrumentation and tracing
  - Java Bytecode
  - Android (Frida, Xposed)
  - Linux (eBPF, Frida)
  - Ruby (Frida)
- Generally, to aid in reversing (e.g. dump process state) or to util-ize existing things
- ”Dynamic Instrumentation”
  - Function hooking
  - Instruction instrumentation (assembly, bytecode, etc.)
- ”Dynamic Tracing”
  - Dynamically enabling/disabling existing logging functionality
  - Dynamically adding enhanced logging functionality that wasn’t there before

Background — Ruby and Me

- Ruby bytecode transformation
  - I was doing blackbox testing of a Rails web app
  - That loaded from pre-compiled Ruby bytecode, which I was able to extract
  - Unfortunately, it used a current-ish Ruby and the bytecode format was incompatible with exiting Ruby bytecode decompilers.
  - So I diff'd the instruction listings between versions and de-optimized the new opcodes into equivalent versions supported by the decompiler

[1] https://research.nccgroup.com/2020/12/15/an-adventure-in-contingency-debugging-ruby-ioread-iowrite-considered-harmful/

Background — Ruby and Me

- Ruby bytecode transformation
  - I was doing blackbox testing of a Rails web app
  - That loaded from pre-compiled Ruby bytecode, which I was able to extract
  - Unfortunately, it used a current-ish Ruby and the bytecode format was incompatible with exiting Ruby bytecode decompilers.
  - So I diff'd the instruction listings between versions and de-optimized the new opcodes into equivalent versions supported by the decompiler
- Socket Adventure 2 Battle[1]
  - A colleague and I were looking into Ruby's dRuby protocol
  - We were writing a scanner for it in Ruby
  - *"when I run it in a Docker container it hangs on read ...in a VM it works"*
  - I then spent an obscene amount of time delving into Ruby's internals to ultimately discover that you should never use `IO#read` on a `Socket` object

[1] https://research.nccgroup.com/2020/12/15/an-adventure-in-contingency-debugging-ruby-ioread-iowrite-considered-harmful/

Background — Ruby and Me

- Ruby bytecode transformation
  - I was doing blackbox testing of a Rails web app
  - That loaded from pre-compiled Ruby bytecode, which I was able to extract
  - Unfortunately, it used a current-ish Ruby and the bytecode format was incompatible with exiting Ruby bytecode decompilers.
  - So I diff'd the instruction listings between versions and de-optimized the new opcodes into equivalent versions supported by the decompiler
- Socket Adventure 2 Battle[1]
  - A colleague and I were looking into Ruby's dRuby protocol
  - We were writing a scanner for it in Ruby
  - *"when I run it in a Docker container it hangs on read ...in a VM it works"*
  - I then spent an obscene amount of time delving into Ruby's internals to ultimately discover that you should never use `IO#read` on a `Socket` object
  - This led me to start writing a (parasitic) low-level tracer for Ruby

[1] https://research.nccgroup.com/2020/12/15/an-adventure-in-contingency-debugging-ruby-ioread-iowrite-considered-harmful/

Parasitic Tracers — What?

- A ”tracer” is basically an enhanced logger that dumps everything you might want about program state

nccgroup

Parasitic Tracers — What?

- A ”tracer” is basically an enhanced logger that dumps everything you might want about program state
- A ”parasite” is a highly-specialized (unwanted) organism that symbiotically lives on or inside another organism that it is completely adapted to

nccgroup

Parasitic Tracers — What?

- A ”tracer” is basically an enhanced logger that dumps everything you might want about program state
- A ”parasite” is a highly-specialized (unwanted) organism that symbiotically lives on or inside another organism that it is completely adapted to
- So a ”parasitic tracer” is tracer specially adapted to a target process that it attaches to and injects itself into and makes use of internal functionality that is not intended to be exposed

nccgroup

Parasitic Tracers — What?

- A ”tracer” is basically an enhanced logger that dumps everything you might want about program state
- A ”parasite” is a highly-specialized (unwanted) organism that symbiotically lives on or inside another organism that it is completely adapted to
- So a ”parasitic tracer” is tracer specially adapted to a target process that it attaches to and injects itself into and makes use of internal functionality that is not intended to be exposed
- The tracing is the ”goal” (one of many); the parasitism is an implementation detail
  - Have you ever injected code via LD_PRELOAD

nccgroup

Parasitic Tracers — Why?

- Bridging the gap between high-level abstractions and low-level implementations aids:
  - Reversing
  - Performance analysis
  - Debugging
- To prototype tracing infrastructure that could be implemented within the target itself

nccgroup

Parasitic Tracers — Examples

- Frida's Java bridge is arguably two parasitic tracers
  - One for Android
  - One for the JVM (HotSpot/JVMTI)
  - Both provide anchor points for tapping functionality with hooks targeting higher-level Java operations in ways that are not intended by either platform
- However, a vanilla Java agent (`java.lang.instrument`/JVMTI) would not qualify as those are public instrumentation APIs
- If you're just crawling around the memory of a process or intercepting its syscalls, you may have a tracer
- But if you're hooking functions inside the process, or, more importantly, *calling* functions from inside the process, then you probably are doing some parasitic tracing

Designing Parasitic Tracers (for high-level languages)

Designing Parasitic Tracers — Prerequisites

- Some means to hook code or instrument it
  - Ideally, one that supports dynamically adding/removing such hooks/instrumentation at runtime
    - E.g. with a debugger or an instrumentation toolkit like Frida
- A way to invoke existing functionality
  - Ideally with ”stable” native APIs
    1. Public APIs (preferred)
    2. Internal APIs with symbols
    3. Internal APIs without symbols for which handles can be reliably obtained
    4. Reimplementation
    5. ...
    ∞. Gadgets (last resort)

nccgroup

Designing Parasitic Tracers — Concepts — Reconnaissance

- Reverse engineering
  - To understand the internals
  - You may even have source, but you need to understand what happens at the native level (or the level your modifications run at/apply to)
    - Optimizations can elide out functions or even result uncleared registers being passed to calls since they are never (normally) used
    - Often, programming language runtimes will be written in very implementation-defined C or C++
- Identifying relevant target functionality
  - To extract information from
  - To wrap with hooks
    - To get whatever you might need by snatching it out of function calls
    - Sometimes it makes sense to hook lower-level/smaller operations to spy on or mess with higher-level ones that they form or themselves invoke

Designing Parasitic Tracers — Concepts — Instrumentation

- Hook the relevant functionality
- Extract program state at runtime
- Invoke internal operations as needed to obtain more information or configure program state

nccgroup

Designing Parasitic Tracers — Concepts — Puppeteering

- Take control from your hooks or injected threads/tasks/etc.
- Build interfaces to invoke functionality and translate inputs and outputs between the target and the instrumentation
- Enhance those interfaces to enable first-class interop

nccgroup

Designing Parasitic Tracers — Concepts — Composition

- Start small, build big
- You're essentially starting with a fully functioning program
- So focus on making your footholds strong to build a foundation for deeper hooking/instrumentation/analysis/etc.
  - This is more or less how Frida itself is designed
  - The Java instrumentation APIs are implemented using lower-level hooks into the inner machinery of the target
- Layer abstractions such that lower-level version-specific logic can be swapped in or out as needed to support multiple versions of the target
  - Two primary methods, per-version implementations and version-based switches/ifdefs

nccgroup

Let’s talk about Ruby

Ruby — A high-level language for the high-minded

- Ruby is an ”interpreted, high-level, general-purpose,” dynamically typed programming language that ”supports multiple programming paradigms, including procedural, object-oriented, and functional programming”

Ruby — A high-level language for the high-minded

- Ruby is an ”interpreted, high-level, general-purpose,” dynamically typed programming language that ”supports multiple programming paradigms, including procedural, object-oriented, and functional programming”
- tl;dr Ruby is a scripting language
- It’s notable features are:
  - Everything is an object
  - Everything else is a method, which are also objects
  - Optional parentheses, so every ”field access” is a method call
  - Internally, methods get called by sending messages, but you can __send__ messages too
  - Method swizzling
  - Object-centric programming
  - Duck hunt typing
  - Perl-inspired super globals
  - Having at least 3 ways to do everything

Ruby — For want of a tracer

- Ruby has 3 kitchen sinks
- But it doesn't have good low-level introspection/tracing capabilities
  - Its `TracePoint` API leaves much to be desired
    - It can't intercept Ruby method arguments or native function parameters
    - It can't provide information on bytecode execution
    - It provides limited information on Ruby-to-native transitions
- This potentially stems from Ruby's nature as a multi-implementation language.
  - The reference implementation is "CRuby" (AKA YARV, formerly MRI)
  - The current CRuby implements a custom bytecode virtual machine, but this is an implementation detail
  - In essence, Ruby (language) on CRuby (interpreter) is similar to Java on the JVM, except that it has none of the analysis and tooling support around its bytecode ISA, which changes slightly from version to version

nccgroup

ruby-trace — A tracer, for Ruby

- A Frida-based CLI tool for instrumenting and dumping execution information from Ruby programs running on Linux
  - Node.js CLI interface
  - `frida-compile` (webpack for Frida) used to support modular design of injected agent JS payload

ruby-trace — Features

- Hooks the following (among other things):
  - All Ruby VM opcode implementations through the opcode instruction table
    - The common build pattern on Linux is a goto-based state machine, where each label is stored in an array (`insns_address_table`)
  - Ruby method call and Ruby's dynamic C function execution mechanisms
    - `send`, `opt_send_without_block`, `invokesuper` opcodes
    - `rb_vm_call_cfunc`
    - `vm_call_cfunc(_with_frame)`
    - `rb_vm_call0`
    - `vm_search_method_fastpath`
    - `vm_sendish`
    - `rb_iterate0`
    - `rb_define_method`
    - `rb_define_module_function`
  - Ruby exception handling mechanisms
    - `throw` opcode
    - `rb_throw_obj`
    - related methods and native implementations

nccgroup

ruby-trace — Features

- Extracts (among other things):
  - Opcode arguments, both ”register”-based and ”stack”-based
    - Will also `inspect` most values, falling back to several alternate stringification mechanisms when it is unsafe to invoke method sending or even directly `inspect` an object
    - Will disassemble bytecode (”ISEQ”/instruction sequence) objects, like methods and blocks
  - Opcode and native function return values
  - Relevant arguments
  - Method call metadata
    - e.g. branches, other control flow changes
  - Bytecode metadata
    - To make certain runtime values human-readable

nccgroup

ruby-trace — Features

- Ruby 2.6-3.0+ support
  - Uses generic shared implementations of hooks with version-specific overrides
  - Reuses C structs from each supported version of Ruby to generically extract struct fields from the correct offsets

nccgroup

ruby-trace — Other cool™ things

- Supports using the TracePoint API as an execution context to enable tracing. This enables fine-grained tracing of Ruby execution.
- Extensive series of test cases mapping to bytecode sequences
  - Arguably covers more than Ruby's own internal VM opcode test suite
- Implements support for dead Ruby opcodes that can't even be compiled
  - i.e. `reput`, a cursed instruction dating back to a 1995 paper on "Prolog, Forth and APL"
- ruby-trace is essentially a CRuby bytecode interpreter as it re-implements a large portion of the bytecode handler logic itself to determine how a given opcode will execute

nccgroup

DEMOs

nccgroup

ruby-trace — Future Work

- Support for Ractors (Ruby's new actor model concurrency feature) is upcoming
- Keeping ruby-trace up-to-date with new Ruby versions as they come out (or as I need to support them)

nccgroup

ruby-trace — Where?

`https://github.com/nccgroup/ruby-trace`

(shortly after this presentation airs)

nccgroup

Conclusion

- This has been a pretty fun exercise in reversing
  - It was also maddening between Ruby's more "magical" opcodes and darker facets
- I think it's pretty reasonable to apply these techniques to getting deeper insights into code running on higher level language interpreters/runtimes
  - There are a lot of applicable languages lacking good tooling (other than Ruby):
    - Python
    - Node/V8
    - Golang
    - Haskell
- More people should try building these things

nccgroup

Conclusion — Go Forth and Write Parasitic Tracers

You know, if one person, just one person does it, they may think he's really sick.
And three people do it, three, can you imagine three people writing parasitic tracers?
They may think it's an organization.
And can you, can you imagine fifty people, I said fifty people, writing these tracers?
Friends, they may think it's a movement.

~Arlo Guthrie

nccgroup

Greetz

- Addison Amiri

nccgroup

You can’t hide ~~secrets~~
from the future ~~using math~~

Questions?

jeff.dileo@nccgroup.com
@ChaosDatumz

nccgroup

Instrument and Find Out

Writing Parasitic Tracers for High(-Level) Languages

Jeff Dileo
@chaosdatumz

DEF CON 29

nccgroup