ruby for penetration testers

When you're down deep reversing a protocol or picking apart a binary, getting up to speed quickly can be challenging in the best of circumstances. Over the past few years, we've figured out a tool that we can rely on every time: the Ruby programming language. We'd like to highlight our use of Ruby to solve the security testing problems we're faced with every day.

We use Ruby because it's easy, flexible, and powerful. It works for everything from reverse engineering firmware bus protocols to fuzzing file formats to static and dynamic binary analysis. We've used it to beat up web apps, and we've stuck with it all the way to attacking exotic proprietary hardware applications. Having a great set of tools available to meet your needs might be the difference between a successful result for your customer and updating your resume with the details of your former employer.

Not familiar with Ruby? None of us were either on that fateful day when Dino Dai Zovi declared Python "the language of over the hill hackers". But we were surprised at how easy Ruby was to pick up. So we'll lead off by illustrating why Ruby is so powerful, making a case for rapidly prototyping everything from reversing tools to hacked up network clients using our not-so-patented "bag-o-tricks" approach.

Then we dive into our real-world experiences using Ruby to quickly get up and running on a wide range of tasks, including:

- Ripping apart static binaries and bending them to your will
- Getting up close and personal with proprietary file formats
- Becoming the puppet-master of both native and Java applications atruntime
- Exposing the most intimate parts of exotic network services like JRMI and Web services
- Trimming the time you spend decoding proprietary protocols and cutting directly to fuzzing them

As if all that wasn't enough, we'll show you how to make Ruby mash-ups of the stuff you already love. Make the tools you already rely on new again by getting them to work together, harder and smarter. When you're asked to get twice as much done in half the time, smile confidently knowing you have a secret weapon and the job will get done.

WHY WE LIKE RUBY

You wouldn't be reading this white paper or attending our talk unless you already knew some kind of scripting language. So the easiest way to help you "get" Ruby is to compare it to other languages.

The language everyone compares Ruby to is Python. You can Bing "Ruby vs. Python" and find 1,000 good shootouts. Most of them are going to point out the most important fact: Ruby and Python are remarkably similar languages, to the point where you can readily port code between them. If you're a pentester, here are some of the big differences you'll care about:

- Ruby has "blocks", which are a notation for defining functions on the fly without naming them; you can stuff them into variables and pass them around. This is huge: it allows you to define domain-specific languages and new control structures, and it's absolutely killer for writing asynchronous network code.

- **Python is faster than Ruby.** Not a little bit faster. A lot faster.

- But Ruby has first-class regular expressions, using the /regex/ syntax borrowed from Perl. This means regexes are insanely easy to use in Ruby. You don't have to "import" them from a library or instantiate classes.

- Python has a huge, sprawling standard library. Ruby has a smaller, tighter standard library.

Yes, Ruby has some syntax borrowed from Perl. Yes, this is a scary idea. But you don't care: the regex syntax is good, and the rest of it you can pretend doesn't exist. Nobody writes Ruby code that looks like Perl.

Mike Tracy, god help him, came to Matasano from Tcl. Tcl and Ruby are surprisingly similar: **you can call Ruby "Japanese Tcl"** and defend that name long enough to upset a Rails programmer. Go ahead, try it! Ruby programmers use blocks for a lot of the same things that Tcl programmers use "uplevel" for, and the Ruby object model is very similar to [incr Tcl].

All these dynamic languages are flexible. Ruby allows us to rapidly prototype tools for vulnerability exploitation, protocol fuzzing, reverse engineering and everything in-between. Many of the tools we develop in Ruby are easily hooked into one another which can further speed up tool development and promotes code reuse.

Ruby has an answer to almost every situation where we would want to develop custom code to solve a problem:

- We can redefine portions of the library with "monkey patches", for instance to allow all Numeric types to render as bignums.

- We can call low-level C libraries with Ruby/DL, FFI, or Win32ole. Or we can wrap the library directly by extending the Ruby interpreter.

- We can even add Ruby into existing tools written in languages like C.

- Ruby allows us to easily create DSL (Domain Specific Language) frameworks like Ruckus, where defining complex structures is done in code, not complex configuration files.

WHO ELSE IS USING RUBY?

Ever hear of Metasploit? Metasploit may be one of the largest Ruby projects in existence and arguably in the most popular list of Ruby frameworks. Metasploit makes advanced exploitation of vulnerabilities possible through easy to use interfaces, payloads and tools. All of this great stuff is also supported on multiple platforms thanks to Ruby.

Metasm is another powerful Ruby framework for compiling, disassembling and debugging native code from Ruby. Metasm is included with the Metasploit framework as well.

Ronin is another Ruby framework written with security and data exploration in mind. Ronin provides convienence methods for an array of different protocols that penetration testers might find useful.

SCRIPTED PENETRATION TESTING

Your first question about whether a language is good for pentesting is, "how does it handle web work". Our answer: WWMD.

WWMD is a console for breaking web applications. It's like "pentesting Expect": it's something in between a programming environment and a console.

WWMD isn't intended to be just another of the myriad tools used to conduct web application security assessments. Its goal is to provide an easily accessible scripting framework that includes the basic elements of a web testing tool (transport and parsing) and combine them with convenience methods that make manual and automated testing tasks easier. Working either in IRB or from scripts, it's a snap to create powerful tools that take care of the time consuming and repetitive stuff and help you with the more subtle and advanced things you need to get done.

WWMD relies on Ruby and some great libraries for its base. Even if you're not going to use WWMD, you should know about:

- Curb, which provides libcurl bindings for Ruby, which we use for our raw HTTP transport.
- Nokogiri, for parsing HTML documents.

Curb and Nokogiri are extremely excellent libraries, each of them reason enough to spend some time learning Ruby.

To this, WWMD adds methods for everything from manipulating headers and application inputs to encodings. It also includes a patch to Curb to allow sending requests using arbitrary methods (OPTIONS, TRACE, RANDOM). All of the behaviors of the base Page object can be easily modified on a per-application basis using mixins and monkey patches that are specific to your engagement.

It also includes a ViewState (de)serializer that outputs to and reads in from XML. If you've never fuzzed ViewState before (working on one of the 4% of web applications out there that don't have EnableViewStateMac = true?) then this is your huckleberry. Another interesting use for the ViewState deserializer is to programatically base64 decode BinarySerialized() (custom serializations of objects like Telerik controls) that you'll find in many web applications. Before WWMD, I had to do all that work by hand.

A simple login example:

```
wwmd(main):003:0> page =
Page.new();nil
=> nil
wwmd(main):004:0> page.baseurl =
"http://www.example.com"
=> "http://www.example.com"
wwmd(main):005:0> page.get "http://
www.example.com/example/"
=> [200, 663]
wwmd(main):006:0> page.text
=> "Login:\nPassword:\n"
wwmd(main):007:0> form = page.getform
=> [["username", nil], ["password",
nil]]
wwmd(main):008:0> form['username'] =
"jqpublic"
=> "jqpublic"
wwmd(main):011:0> form['password'] =
"password"
=> "password"
wwmd(main):012:0> page.submit form
=> [200, 2117]
wwmd(main):013:0>
page.bodydata.match(/you are logged
in.*/)[0].striphtml
=> "you are logged in as jqpublic
[logout]"
wwmd(main):014:0>
```

Ever see a web form that takes an argument like:

```
args=key|value;key|value;key|value
```

Instead of just fuzzing the form variable, you can simply create a copy of the FormArray class that uses | and ; as delimiters and fuzz everything:

```
wwmd(main):006:0> form = FormArray.new
=> [] wwmd(main):007:0> cust =
FormArray.new => []
wwmd(main):008:0> cust.delimiter = ";"
=> ";"
wwmd(main):009:0> cust.equals = "|"
=> "|"
wwmd(main):010:0>
cust.fromarray([["key1","val1"],
["key2","val2"],["key3","val3"]])
=> [["key1", "val1"], ["key2",
"val2"], ["key3", "val3"]]
wwmd(main):011:0> cust.topost
=> "key1|val1;key2|val2;key3|val3"
wwmd(main):012:0> form['args'] =
cust.topost
=> "key1|val1;key2|val2;key3|val3"
wwmd(main):013:0> form['test'] =
"value"
=> "test"
wwmd(main):014:0> form.topost
=> "args=key1|val1;key2|val2;key3|
val3&test=value"
```

WWMD is available on github (http://github.com/miketracy/wwmd/tree/master) and remember, swiss army knives don't kill people but 15 different sharp things can't hurt.

REVERSING

Reverse engineering has taken a front seat in vulnerability research and penetration testing over the last few years. Often a penetration tester may be tasked with reversing proprietary network protocols or closed source binaries in a relatively short amount of time.

Ruby enables this kind of rapid tool development whether the goal is breaking open a custom network protocols header structure and de-obfuscating its payload or finding that backdoor in a compiled executable. We have developed tools to do both these kinds of things.

NETWORK PROTOCOLS

Being able to transparently intercept and modify network traffic is a great advantage to a penetration tester tasked with finding bugs in a proprietary network protocol. Not all operating systems have well defined support for this type of behavior. We have developed a few OS-indepedent inline proxy tools to help ease the process of attacking protocols in this way.

These tools are available in our 'Ruby BlackBag (rbkb)' distribution and are named 'blit', 'telson', 'plugsrv' and 'feed'. They work together to allow for inline network traffic modification and inspection.

- **blit**: a simple OOB (Out Of Band) IPC (Inter Process Communication)
  mechanism for sending messages to blit enabled tools.
- **telson**: is responsible for setting up network connections and
  listening for commands from blit enabled clients
-

- **plugsrv**: is a reverse TCP/UDP proxy between one or more connections
- **feed**: a blit capable tool that feeds files to blit enabled servers

Packet captures can be modified and replayed with ease by using a combination of blit and telson. Simply save your saved session, modify the desired bytes, setup a connection with telson and send the packets to blit manually or use feed to send all of the modified packets one at a time to telson. Using these tools seems a bit manual at first, but Ruby allows for their usage to be scripted easily and they often come in use for fuzzing network sessions inline or reversing tricky protocols.

BINARIES

Ruby is also effective in the area of static binary analysis.

Often when reverse engineering a closed source binary the penetration tester will be presented with embedded compressed images or obfuscated data segments. We can combine the usefulness of Ruckus with our many monkey patches to help de-obfuscate and extract these portions of applications. deezee is a tool included in Matasano's original black bag C implementation. It works by traversing a binary blob for compressed zlib images. Ruby has support for the Zlibc library by default so porting this tool to Ruby is trivial. This tool is often successful in extracting embedded file system blobs from firmware images or compressed data segments within an executable.

There are times when custom obfuscation is used to hide data segments of a binary on disk. Often this comes in the form of a simple xor or base64 encoding. This is when we use Ruby monkey patches to extract this data. A quick and easy String class monkey patch to xor bytes against a 'key' may look like this:

```
def xor(k)
  s=self
  out=StringIO.new ; i=0;
  s.each_byte do |x|
   out.write((x ^ (k[i] ||
k[i=0]) ).chr)
    i+=1
  end
  out.string
end
```

Extracting strings is often the first step to take when analyzing a foreign binary blob. We wrote a better 'strings' utility in Ruby called rstrings. rstrings has support for optional start and end offsets and different encoding types ascii and unicode and the ability to print at what offset in the blob the string was found.

```
$ rstrings -t ascii -l 10 /bin/ls
00001024:0000102f:a:"__PAGEZERO"
000012d8:000012e3:a:"__pointers"
0000131c:00001329:a:"__jump_table"
00001368:00001373:a:"__LINKEDIT"
```

Grabbing the strings from a binary can only take you so far, at some point its file format structure and code segments must be examined in detail. For this we use Ruckus and in the case of x86 executable, Frasm. Frasm is a Ruby extension to the Distorm64 disassembly library. Disassembling x86 code in Ruby has never been easier:

```
require 'frasm'
d = Frasm::DistormDecoder.new
d.decode("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
.each do |l|
    puts "#{l.mnem} #{l.size}
#{l.offset} #{l.raw}"
end
```

```
INC ECX 1 0 41
INC EDX 1 1 42
INC EBX 1 2 43
INC ESP 1 3 44
INC EBP 1 4 45
INC ESI 1 5 46
INC EDI 1 6 47
DEC EAX 1 7 48
...
```

RUNTIME ANALYSIS

For debugging native code we have developed a debugger named Ragweed. Ragweed uses Ruby/DL to wrap the native debug API on Win32, OS X and Linux. Ragweed is basically a scriptable debugger which allows us to automate every task from hit tracing to extracting data during execution.

FUZZING

Fuzzing is how you find bugs in binary attack surfaces. You take a message, jumble it up, and throw it at the target. Again and again. Eventually the target crashes. You find out why. The answer is a security advisory.

Every major language has a fuzzing framework. Probably the best-known is Peach, which is Python's fuzzer du jour. We have a Ruby fuzzing framework. It's called Ruckus. Ruckus will take the Pepsi Challenge against Peach any time.

The first thing you want from a fuzzer is the ability to define messages. So, you've got your DHCP header:

```
0..7    8..15   16..23  24..31
opcode  type    addr_len    hopcount
transaction id
num_seconds flags
client IP
your IP
server IP
gateway IP
client hardware address
(cont'd)  hostname
(cont'd)
...
(cont'd)  bootfile
```

And here it is in Ruckus:

```
class DHCPHeader < Ruckus::Structure
    byte :opcode, :value => 1
    byte :hwtype, :value => 6
    byte :hw_address_len, :value => 6
    byte :hopcount
    n32 :trans_id
    n16 :num_secs
    n16 :flags
    ipv4 :client_ip,
          :value => "0.0.0.0"
    ipv4 :your_ip
    ipv4 :server_ip,
          :value => "0.0.0.0"
    ipv4 :gateway_ip
    num :client_hw, :width => 48
    string :server_hostname,
          :size => 64,
          :value => ""
    string :boot_file,
          :size => 128,
          :value => "generic"
end
```

Some things to notice here:

- Ruckus messages types are Ruby classes, but we give you a DSL-style interface for defining the fields.
- We've got field types for everything you're going to see in a normal message. Byte-

sized fields. 32 bit network byte order fields. IP addresses. Strings.

- We do arbitrary numeric types. Got a 27 bit integer field? Done! Got a flag word? Define the flags bit by bit!
- Want a new field type? Every Ruckus message type is automatically a field type (lowercase the class name). It's turtles all the way down.
- Of course fields can take default values.

But wait! There's more!

Every field in a Ruckus message can relate to another field. For instance:

```
class Foo < Ruckus::Structure
    byte :len
    str :string

    relate_size :string,
       :to => :len
    relate_value :len,
       :to => :string,
       :through => :size
end
```

This is something that comes up in network protocols all the time: length delimited strings. The field "len" records the 8 bit length of the field "string". Ruckus takes care of this for you.

Ruckus works in both directions: in and out. If you define a working message type for sending messages, that same message type can parse raw byte strings back into messages. Why is this cool? Because it allows us to do template-based fuzzing; for instance, we can write a proxy for a network protocol, capture messages, and then replay them with subtle (or not-subtle) variations.

Here's where things with Ruckus start to go crazy-go-nuts. Ruckus is actually modeled in part after the HTML DOM.

Like we said earlier, "turtles all the way down"? Every field is itself a class. An integer is a Ruckus::Number. A string is a Ruckus::Str. If you want to wrap a DHCP header in a TCP message, you can do that with one field declaration.

Every field of every message is identified in two important ways:

- its class; Ruby is introspective: you can take any variable and gets its type with a single call.
- its optional "tag", which is the moral equivalent of an HTML DOM "id".

All the fields of a message, nested arbitrarily deep, form a tree. Just like in the HTML DOM. And you can ask that tree for, say, all the nodes that are of class "string". Or the node with the id "smbheaderbase". Or all strings in message components descended from the node marked "smbheaderbase". See where we're going with this? **Cascading fuzz sheets!**

Take an arbitrary message modeled with Ruckus, and you can mutate it using CSS style selectors. You can pick out all the strings under just a portion of the message, modify them in some evil way, and render the message back out, with all the associated length fields and doohickeys valid.

To actually mutate the fields, we use some Dino Dai Zovi code that leverages another Ruby feature: generators. A generator takes a loop and turns it into a vending machine that dispenses the loop results one at a time.

For instance, here's a loop that never ends, which generates random 10-character strings:

```
Loop { str = ""; 10.times { str <<
"A"[0] + rand(26) } }
```

This loop isn't very useful, because if you invoke it, your program freezes. But using Ruby Generators, we can make it useful:

```
g = Generator.new {|g|
    Loop {
        str = "";
        10.times {
             str << "A"[0] + rand(26)
        }
        g.yield(str)
    }
}
```

This is the same loop, but now each time we generate a string, we yield it to the Generator object. We can get each successive string using "g.next", any time we want a random string.

Ruckus uses DFuzz, which is Dino's Generator-driven fuzzer library. DFuzz::String will generate a long sequence of progressively longer, weirder strings. DFuzz::Int will generate integers.

ACTIVEX

ActiveX is an active area of vulnerability research and testing. A handful of general purpose ActiveX security testing tools already exist, each with their own strengths. The available ActiveX testing tools fall, generally, into two categories:

Browser-based Testing. Examples Include:
axman – by H.D. Moore
Dranzer – cert

Direct COM-based Interface Testing. Examples Include:
comraider – iDefense
Dranzer – cert

While the available tools have impressive track records for finding vulnerabilities in ActiveX controls, they can be of limited use for testing controls which have unique peculiarities such as specific initialization requirements or non-standard interfaces. In these cases, being able to quickly prototype and build custom COM or browser-based ActiveX tools to specification can be of immeasurable value.

Ruby lends itself well to the task of ActiveX research and vulnerability testing and brings benefits of rapid prototyping and testing. The windows version of Ruby ships with 'win32ole' as part of its standard library. The 'win32ole' library is designed to expose COM objects to Ruby in a manner not unlike VBScript. The library is implemented a native extension written in C/C++ and exposed to the ruby runtime.

The 'win32ole' library makes dynamic enumeration, testing, and fuzzing of ActiveX (or even other COM interfaces) a snap.

COM ENUMERATION

The code below demonstrates quickly identifying all the installed and registered OLE type libraries on the system including their name, GUID, description, and registered file location:

```
require ‘win32ole’
WIN32OLETYPELIB.typelibs.each do |lib|
begin
puts “Name: #{lib.libraryname}”,
“GUID: #{lib.guid}”,
“Path: #{lib.path}”,
“Desc: #{lib.name}\n\n”

rescue WIN32OLERuntimeError
# skip mis-registered TLB’s
next
end
end
```

Below is an example of a standalone Ruby program which will produce a list of visible methods for any COM interface installed on the system, accompanied by invocation type, return value, and typed arguments. This can be used to quickly identify the exposed methods for an ActiveX control:

```
require ‘win32ole’
obj = WIN32OLE.new( ARGV.shift ) rescue(exit(1))
obj.olemethods.select {|m| m.visible? }. each do |m|
puts “#{m.invokekind}: #{m.returntypedetail.join(‘ ’)} #{m.name}(” +
m.params.map {|p| “#{p.ole_type} #{p.name}”}.join(‘, ’) + “)”
end
```

ACTIVEX FUZZING

Using the same interfaces for enumeration, we can easily begin producing test cases based on the method interfaces. The example below is an extremely trivial test case which simply generates several html files, one for each argument per each method. The test-case checks for unexpected errors when a 10k string is supplied for each argument individually with null for all other arguments.

```
obj.olemethods.each do |m|
psz = m.sizeparams
pary = Array.new(psz, “null”)
0.upto(psz-1) do |idx|
args = pary.dup

tc = “testcase10kstr-argument#{m.name}#{idx}”
args[idx] = ‘“’ + "A”*10000 + ‘“’ # … really lame testcase

File.open(”#{tc}.html" % idx, “w”) do |f|
f.write <<EOF




EOF
end
end
end
```

We built a tool named ‘AxRub’ which takes in a CLSID as its argument and sets up a generic fake HTTP server in order to fuzz an ActiveX control in the browser automatically. AxRub is hooked into the DFuzz generator to fuzz the controls methods with a variety of strings and numeric values.

INTEGRATING RUBY

Most dynamic languages lend themselves to easy integration with existing platforms and toolsets, Ruby is no exception. Ruby can be extended using native C library, existing tools written in C or even bridged to other languages like Java.

WRAPPING LIBRARIES

Wrapping native libraries using Ruby is supported in two different ways. A native Ruby extension can be written in C which links with the library it is intended to expose to Ruby. This is a straight forward method that doesn't require any additional third party code to achieve, only what is absolutely necessary, only a C compiler, Ruby libraries, and the native library you intend to wrap.

Another, and increasingly more popular, way to wrap native libraries is the use of Ruby extensions such as DL and FFI . These extensions allow you to wrap a native library with nothing more then Ruby code. Ruby/DL acts as a basic extension of the dynamic linker, as such you must provide it with the location of your linker and it takes care of the rest. The advantage here is that no native code must be written and compiled. Our portable native code debugger, Ragweed, is written using Ruby/DL. It wraps the linker on Win32, OSX and Linux.

FRASM

One native library we wrapped recently is distorm64 , an x86 32 and 64 bit disassembly library written in C. Distorm already contains Python bindings and we wanted the ability to use it from Ruby. We wrapped the underlying distorm C library in 104 lines of C and now Ruby scripts can be written to disassemble x86 instructions.

```
require 'frasm'

d = Frasm::DistormDecoder.new


d.decode("ABCDEFGHIJKLMNOPQRSTUVWXYZ").each do |l|
    puts "#{l.mnem} #{l.size} #{l.offset} #{l.raw}"
end
```

```
INC ECX 1 0 41
  INC EDX 1 1 42
  INC EBX 1 2 43
  INC ESP 1 3 44
  INC EBP 1 4 45
  INC ESI 1 5 46
  INC EDI 1 6 47
  DEC EAX 1 7 48
  ...
```

The 'decode' method takes in a string of characters and passes them the distorm library for disassembly. An array of objects is returned which hold four class variables 'mnem', 'size', 'offset' and 'raw'.

EMBEDDING THE RUBY INTERPRETER

While wrapping native libraries seems like the most ideal situation for extending existing tools, it is not always an option. Another option for integrating Ruby into an existing tool is embedding the Ruby interpreter itself. The original Ruby interpreter is written in C and provides a convenient API for calling Ruby code from C. In certain cases we had older tools written in C that worked perfectly yet lacked the dynamic programmability that Ruby provides. Rewriting these tools in Ruby is an enormous task that goes against our philosophy of not reinventing the wheel. The basic steps for embedding an interpreter and sharing a string with a ruby script are below:
example.c

```
#include <stdio.h>
#include <ruby.h>

int main(int argc, char *argv[])
{
    ruby_init();                /* Initialize Ruby */

    VALUE str;                  /* Declare the string in C */

    str = rb_str_new2("Some String");   /* Assign the string a value */


rb_load_file("simple.rb");      /* Load the Ruby script we want to run */
```

```

        rb_define_variable("glbl",
&str);   /* Expose our string to our
script */

        ruby_exec();                  /*
Run the interpreter */


rb_eval_string("modify_str");       /*
Call the method 'modify_str' in our
script */

        printf("%s\n",
STR2CSTR(str));      /* Print the
string our Ruby script modified */

        ruby_finalize();              /*
We are now done with Ruby */

        return 0;
    }
```

example.rb

```
    def modify_str
        puts $glbl
        $glbl = "Hello from Ruby!"
    end
```

We can compile our example.c program using gcc, provided we have the right Ruby development libraries in place:

```
$ gcc -I/usr/lib/ruby/1.8/i486-linux/
-lruby1.8 -o example example.c
Running our program:
$ ./example
Hello from Ruby!
```

Our Ruby script, example.rb, was called and the 'modify_str' method modified the global string '$glbl'. Our C program, example.c, printed out the modified string using the STR2CSTR macro provided by ruby.h

Although somewhat cruder then wrapping a native library, embedding the Ruby interpreter is a viable way to add scripting capabilities to existing code bases where you don't wish to rewrite the project from scratch in Ruby.

QUERUB

An older existing project named QueFuzz uses the libnetfilterqueue libraries on Linux to create an inline network packet fuzzer. Writing scalable fuzzing code in C is a lot more difficult then writing it in Ruby. Despite its limitations QueFuzz works as intended, there was no reason to throw it out and start over. Instead we removed the C fuzzing code in favor of embedding the Ruby interpreter and passing the packet to be fuzzed to a Ruby script. This allows us to use all the built in methods Ruby provides when reversing or fuzzing the packets contents. While a Ruby wrapper around the libnetfilterqueue libraries would be ideal, this is an involved software development process that requires all aspects of the libraries functionality be taken into consideration. QueRub serves a specific purpose, to fuzz network packets inline using the dynamic nature of Ruby while utilizing an existing code base.

LEAFRUB

Leaf is another existing tool that was lacking a scripting component but was not eligible to be wrapped as an existing library. Leaf is an extendable ELF analysis and disassembly platform that has support for plugins written in C. A plugin called LeafRub was written to embed the Ruby interpreter and expose Leaf's internal API and data to Ruby scripts that mirror the design of a native C plugin. LeafRub works by creating constants for each x86 instruction type, plugin function arguments, and helpful functions in the Leaf API. As each plugin hook is called in C, its Ruby counter part is called. Just like QueRub, this allows Ruby based Leaf plugins to utilize all Ruby has to offer when disassembling ELF objects.

The following LeafRub Ruby script prints out each instruction and looks up each opcode against a list of known cross references and the ELF symbol table.

```
class Leaf
    def initialize
        puts "\n(LeafRub.rb
initialized)"
    end
    def leaf_code_output
        print sprintf("%s %x [%16s]
(%s) (%x %x %x)\n",
$state.segment_name, $state.offset,
            $instr.hex_string,
$instr.inst_string,
$instr.op_one_value,
            $instr.op_two_value,
$instr.op_three_value,
$instr.op_one_value)

        self.match_xref($state.offset,
$state.offset).each do |x| puts
"\t#{x}" end

self.match_xref($instr.op_one_value,
$state.offset).each do |x| puts
"\t#{x}" end

self.match_xref($instr.op_two_value,
$state.offset).each do |x| puts
"\t#{x}" end

self.match_xref($instr.op_three_value,
$state.offset).each do |x| puts
"\t#{x}" end


self.match_symbols($instr.op_one_value
).each do |x| puts "\t#{x}" end

self.match_symbols($instr.op_two_value
).each do |x| puts "\t#{x}" end

self.match_symbols($instr.op_three_val
ue).each do |x| puts "\t#{x}" end
    end
end
leaf = Leaf.new
```

The output of this script is shown below:

```
$leaf -f /bin/ls
[ LEAF - Leaf ELF Analysis Framework ]
[ Loading LEAF Plugins ... ]

-> LeafRub  [Version: 0.1]
(LeafRub.rb initialized)

.rel.plt 8049508 [             55 ]
(push %ebp) (0 0 0)
    (.rel.plt 0x8049508) @ [0x805aec4
call 0x8049508]
    0x8049508 = _init@GLIBC
.init 8049509 [           89e5 ] (mov
%esp, %ebp) (0 0 0)
.init 804950b [             53 ] (push
%ebx) (0 0 0)
.init 804950c [         83ec04 ] (sub
$0x4, %esp) (0 4 0)
.init 804950f [     e800000000 ] (call
0x8049514) (8049514 0 0)
    (.init 0x8049514) @ [0x804950f
call 0x8049514]
.init 8049514 [             5b ] (pop
%ebx) (0 0 0)
    (.init 0x8049514) @ [0x804950f
call 0x8049514]
...
```

OTHER LANGUAGES

We use JRuby extensively to bridge the gap between Java and Ruby. This is particularly useful to a pentester who runs into a lot of enterprise Java applications such as JRMI. In particular we have created Buby, a Jruby wrap of the Burp Java API.