

# function hooking for osx and linux

joe damato  
@joedamato  
[timetobleed.com](http://timetobleed.com)

slides are on  
[timetobleed.com](http://timetobleed.com)

SOPHSEC INTRUSION LABS



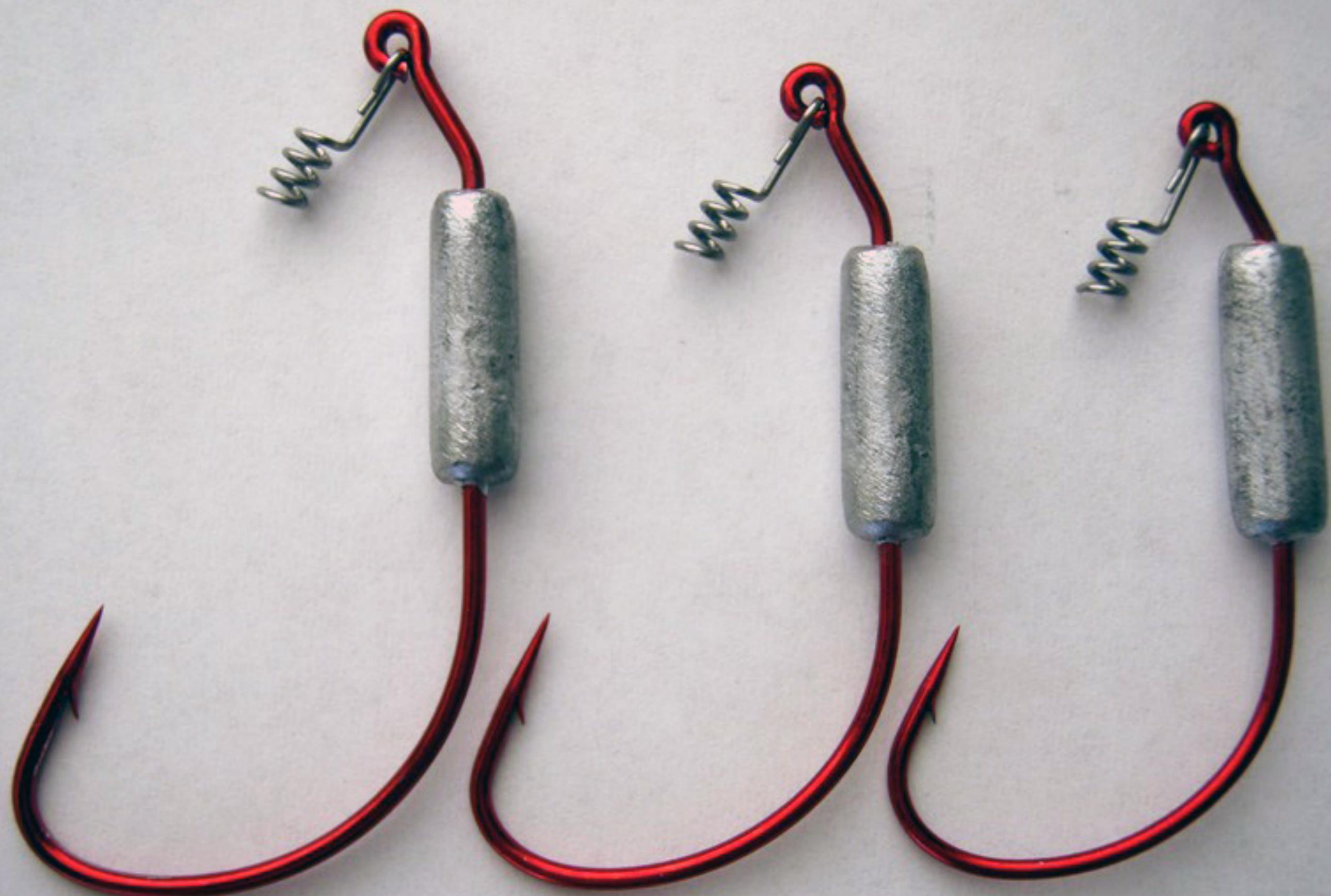
(free jmpesp)

**i'm not a security**  
**researcher.**

call me a script kiddie:  
**@joedamato**









assembly is in att syntax

at&t

**WTF is an ABI ?**

WTF is an Application  
Binary  
Interface ?

# alignment



# calling convention

# object file and library formats



hierarchy of specs

# HIERARCHY OF BEARDS.



SystemV ABI (271 pages)

SystemV ABI AMD64 Architecture Processor  
Supplement (128 pages)

SystemV ABI Intel386 Architecture Processor  
Supplement (377 pages)

MIPS, ARM, PPC, and IA-64 too!

mac osx x86-64 calling convention

based on

SystemV ABI AMD64 Architecture  
Processor Supplement

**BUFFALO TRACE**  
DISTILLERY

**Beer Still**

**Capacity**

**60,000 gallons**

# alignment



end of argument area must be aligned on a 16byte boundary.

and \$0xffffffffffff0, %rsp

# calling convention

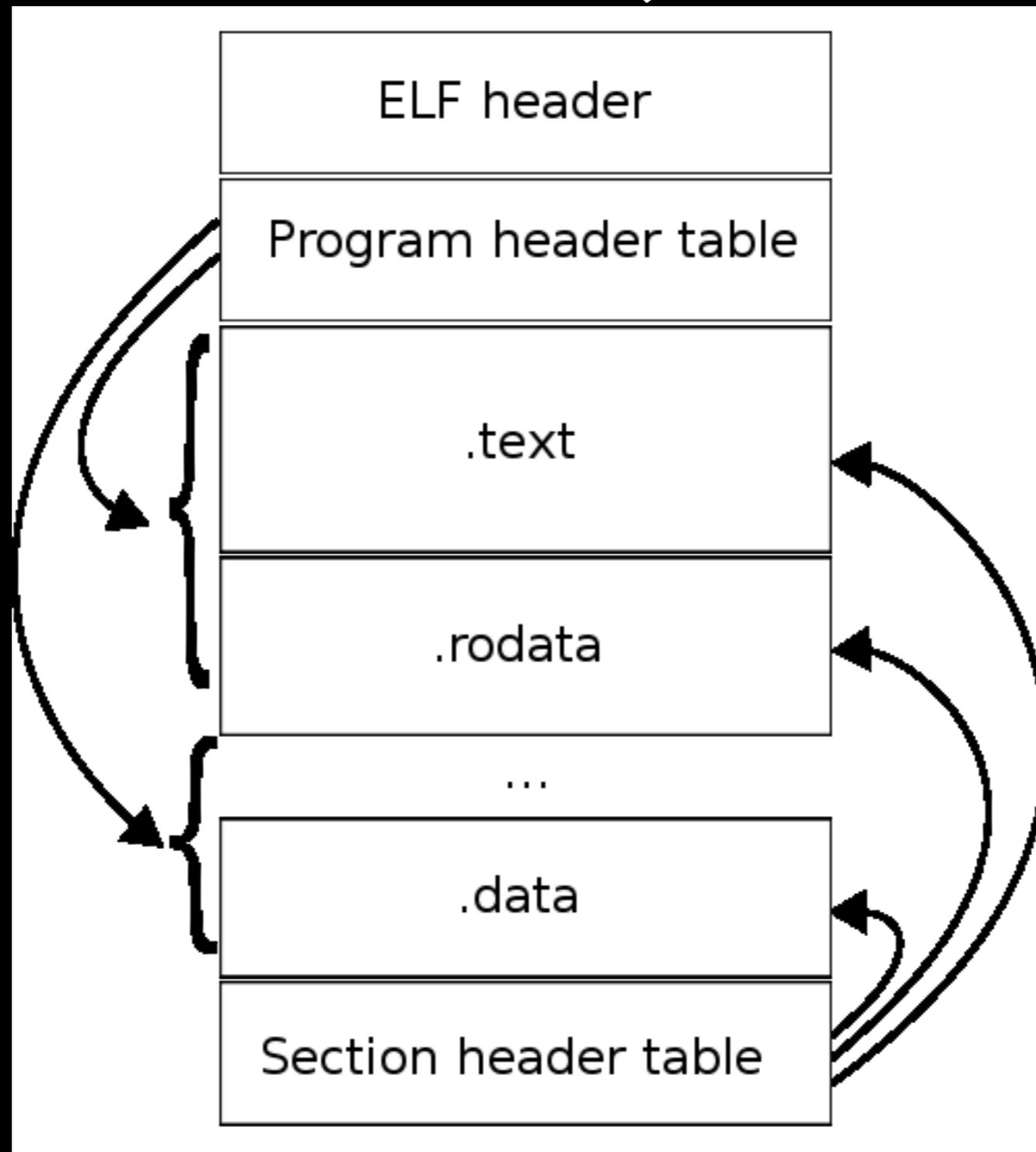
- function arguments from left to right live in:  
%rdi, %rsi, %rdx, %rcx, %r8, %r9
- that's for INTEGER class items.
- Other stuff gets passed on the stack (like on i386).
- registers are either caller or callee save

# object file and library formats





# ELF Objects



# ELF Objects

- ELF objects have headers
  - elf header (describes the elf object)
  - program headers (describes segments)
  - section headers (describes sections)
- libelf is useful for wandering the elf object extracting information.
- the executable and each .so has its own set of data

# ELF Object sections

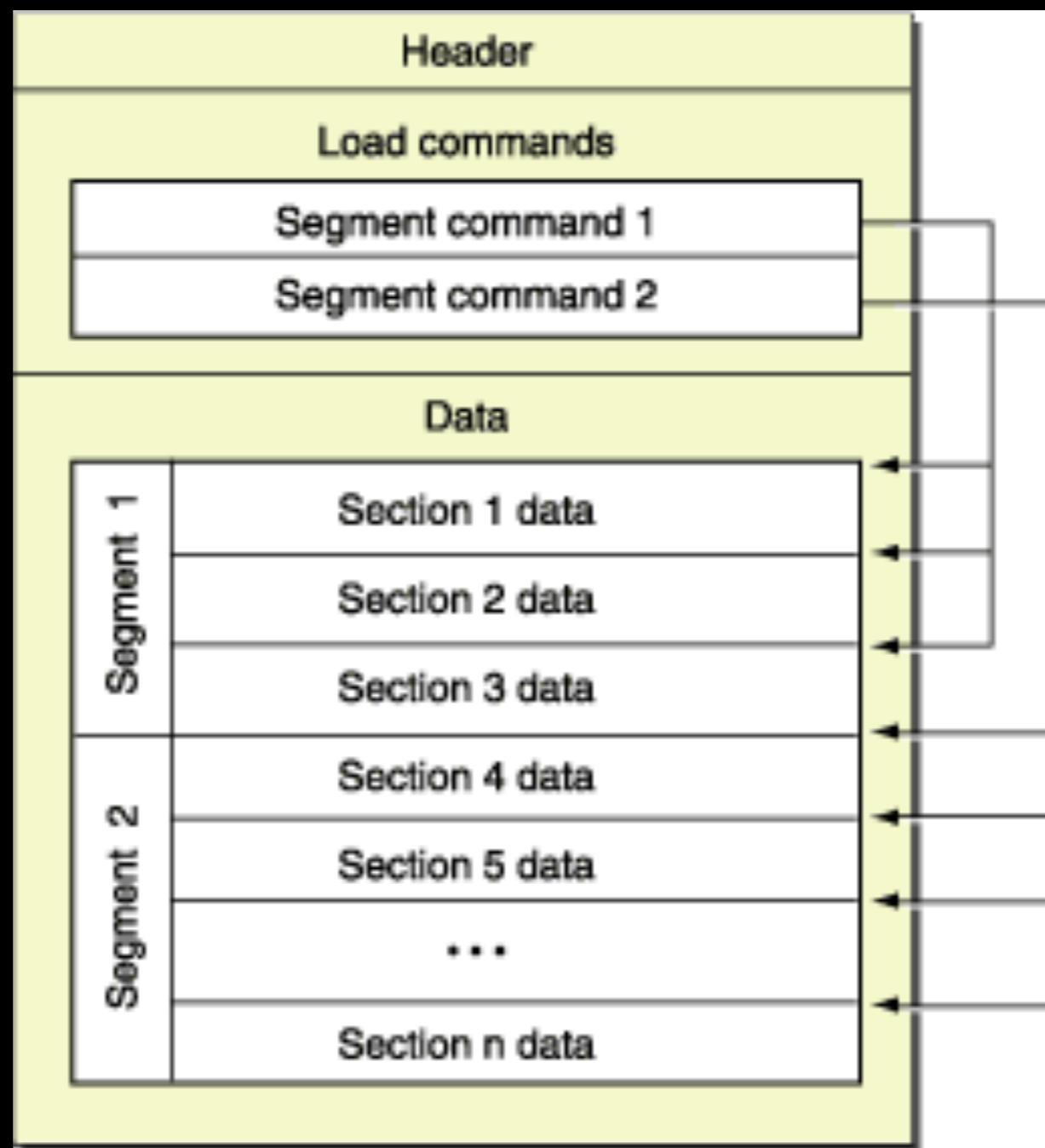
- `.text` - code lives here
- `.plt` - stub code that helps to “resolve” absolute function addresses.
- `.got.plt` - absolute function addresses; used by `.plt` entries.
- `.debug_info` - debugging information
- `.gnu_debuglink` - checksum and filename for debug info

# ELF Object sections

- **.dynsym** - maps *exported symbol names* to offsets
- **.dynstr** - stores *exported symbol name strings*
- **.symtab** - maps *symbol names* to offsets
- **.strtab** - *symbol name strings*
- more sections for other stuff.



# Mach-O Objects



# Mach-O Objects

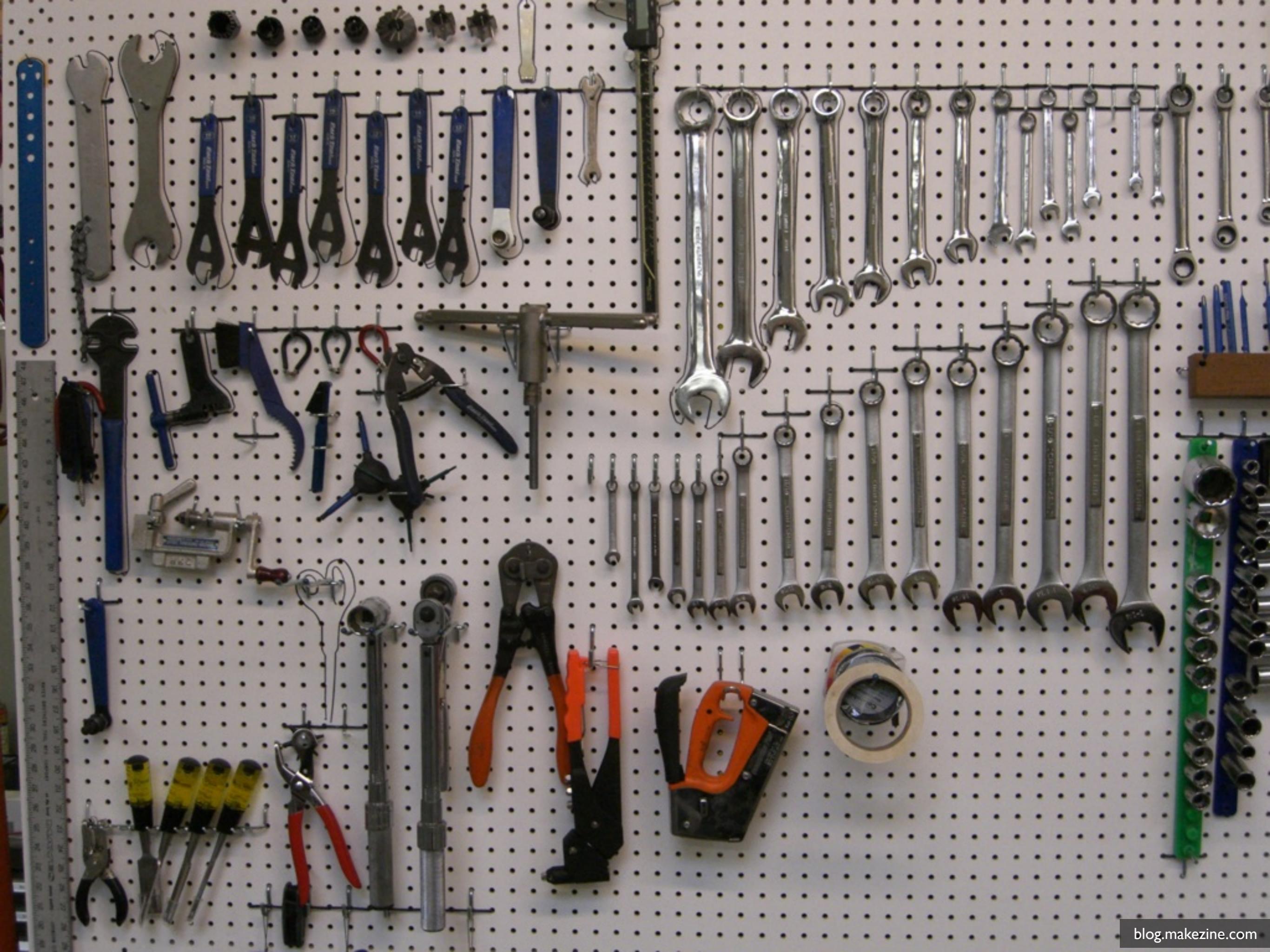
- Mach-O objects have load commands
  - header (describes the mach-o object)
  - load commands (describe layout and linkage info)
  - segment commands (describes sections)
- dyld(3) describes some apis for touching mach-o objects
- the executable and each dylib/bundle has its own set of data

# Mach-O sections

- `__text` - code lives here
- `__symbol_stubl` - list of jmpq instructions for runtime dynamic linking
- `__stub_helper` - stub code that helps to “resolve” absolute function addresses.
- `__la_symbol_ptr` - absolute function addresses; used by symbol stub

# Mach-O sections

- **syms** do not live in a segment, they have their own load commands.
- **LC\_SYMTAB** - holds offsets for symbol table and string table.
- **LC\_DYSYMTAB** - a list of 32bit offsets into **LC\_SYMTAB** for dynamic symbols.



# nm

```
% nm /usr/bin/ruby
```

symbol  
“value”

000000000048ac90	t	Balloc
0000000000491270	T	Init_Array
0000000000497520	T	Init_Bignum
000000000041dc80	T	Init_Binding
000000000049d9b0	T	Init_Comparable
000000000049de30	T	Init_Dir
00000000004a1080	T	Init_Enumerable
00000000004a3720	T	Init_Enumerator
00000000004a4f30	T	Init_Exception
000000000042c2d0	T	Init_File
0000000000434b90	T	Init_GC

symbol names

# objdump

% objdump -D /usr/bin/ruby

00000000000434860 <rb\_newobj>:

434860:	48 83 ec 08
434864:	8b 05 82 12 2c 00
43486a:	85 c0
43486c:	75 6b
43486e:	48 83 3d 3a 85 2a 00
434875:	00
434876:	74 58
434878:	48 83 3d 20 12 2c 00
43487f:	00
434880:	74 4e
434882:	48 8b 05 17 12 2c 00
434889:	48 8b 50 08
43488d:	48 89 15 0c 12 2c 00
434894:	48 c7 00 00 00 00 00
43489b:	48 c7 40 08 00 00 00
4348a2:	00

offsets

434860:	48 83 ec 08
434864:	8b 05 82 12 2c 00
43486a:	85 c0
43486c:	75 6b
43486e:	48 83 3d 3a 85 2a 00
434875:	00
434876:	74 58
434878:	48 83 3d 20 12 2c 00
43487f:	00
434880:	74 4e
434882:	48 8b 05 17 12 2c 00
434889:	48 8b 50 08
43488d:	48 89 15 0c 12 2c 00
434894:	48 c7 00 00 00 00 00
43489b:	48 c7 40 08 00 00 00
4348a2:	00

opcodes

sub \$0x8,%rsp
mov 0x2c1282(%rip),%eax
test %eax,%eax
jne 4348d9 <rb_newobj+0x79>
cmpq \$0x0,0x2a853a(%rip)
je 4348d0 <rb_newobj+0x70>
cmpq \$0x0,0x2c1220(%rip)
je 4348d0 <rb_newobj+0x70>
mov 0x2c1217(%rip),%rax
mov 0x8(%rax),%rdx
mov %rdx,0x2c120c(%rip)
movq \$0x0,(%rax)
movq \$0x0,0x8(%rax)

instructions

# 6f5aec <during\_gc>  
# 6dcdb0 <malloc\_limit>  
# 6f5aa0 <freelist>  
# 6f5aa0 <freelist>  
# 6f5aa0 <freelist>

helpful metadata

# readelf

% readelf -a /usr/bin/ruby

[ 6]	.dynstr	STRTAB	000000000040a270	0000a270		
	000000000003815	0000000000000000	A	0	0	1
[ 7]	.gnu.version	VERSYM	000000000040da86	0000da86		
	00000000000086e	0000000000000002	A	5	0	2
[ 8]	.gnu.version_r	VERNEED	000000000040e2f8	0000e2f8		
	0000000000000c0	0000000000000000	A	6	5	8
[ 9]	.rela.dyn	RELA	000000000040e3b8	0000e3b8		
	000000000000078	000000000000018	A	5	0	8
[10]	.rela.plt	RELA	000000000040e430	0000e430		
	000000000001248	000000000000018	A	5	12	8
[11]	.init	PROGBITS	000000000040f678	0000f678		
	000000000000018	0000000000000000	AX	0	0	4
[12]	.plt	PROGBITS	000000000040f690	0000f690		
	0000000000000c40	000000000000010	AX	0	0	4
[13]	.text	PROGBITS	00000000004102d0	000102d0		
	0000000000096988	0000000000000000	AX	0	0	16

This is a \*tiny\* subset of the data available

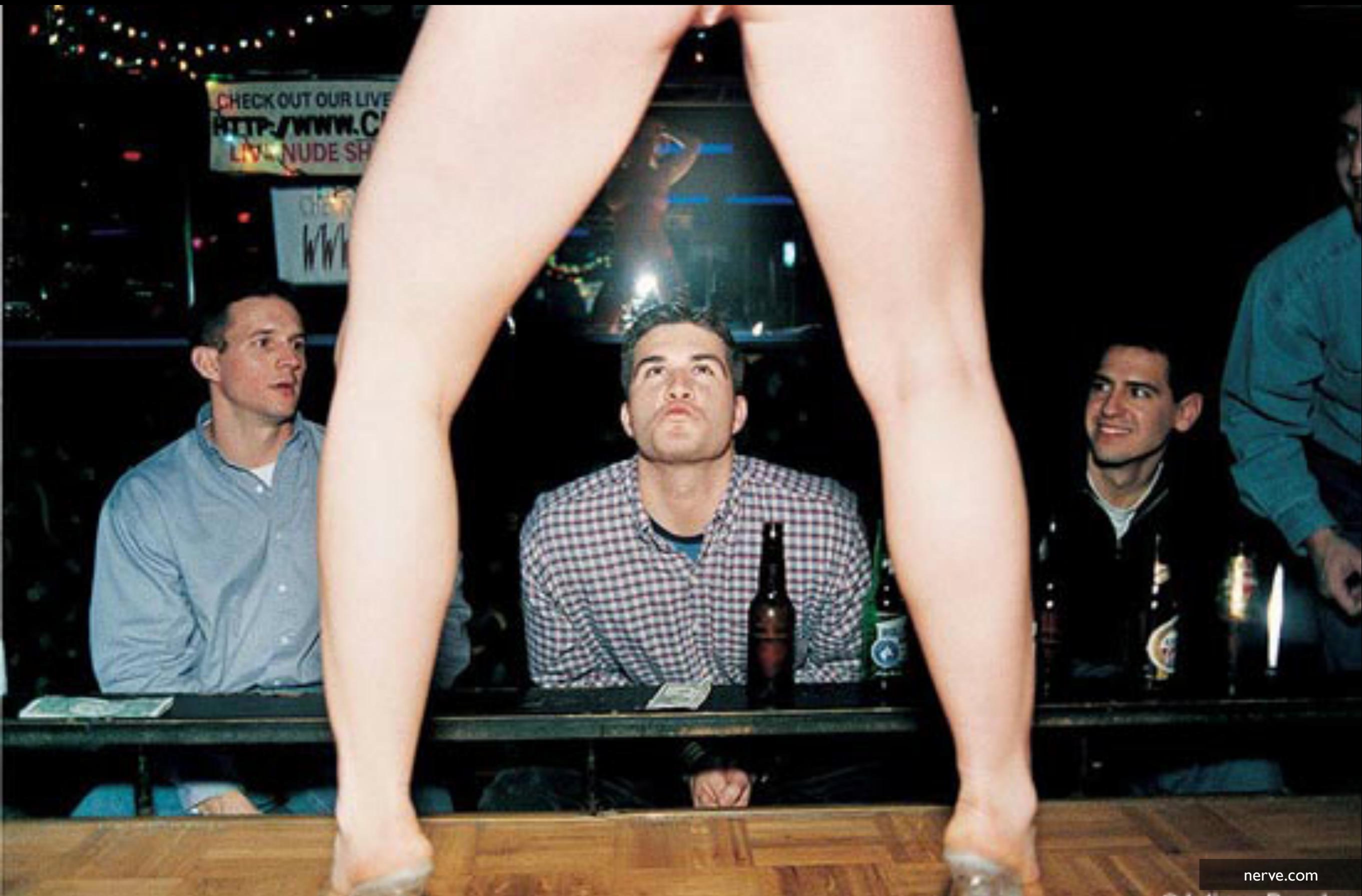
# otool

```
% otool -l /usr/bin/ruby
```

---

```
Load command 0
    cmd LC_SEGMENT_64
    cmdsize 72
    segname __PAGEZERO
    vmaddr 0x0000000000000000
    vmsize 0x0000000100000000
    fileoff 0
    filesize 0
    maxprot 0x00000000
    initprot 0x00000000
    nsects 0
    flags 0x0
Load command 1
    cmd LC_SEGMENT_64
    cmdsize 632
    segname __TEXT
    vmaddr 0x0000000100000000
    vmsize 0x00000000000d6000
    fileoff 0
    filesize 876544
    maxprot 0x00000007
    initprot 0x00000005
    nsects 7
    flags 0x0
```

This is a \*tiny\* subset of the data available



# strip

- You can strip out whatever sections you want....
- but your binary may not run.
- you need to leave the dynamic symbol/string tables intact or dynamic linking will not work.



# Calling functions

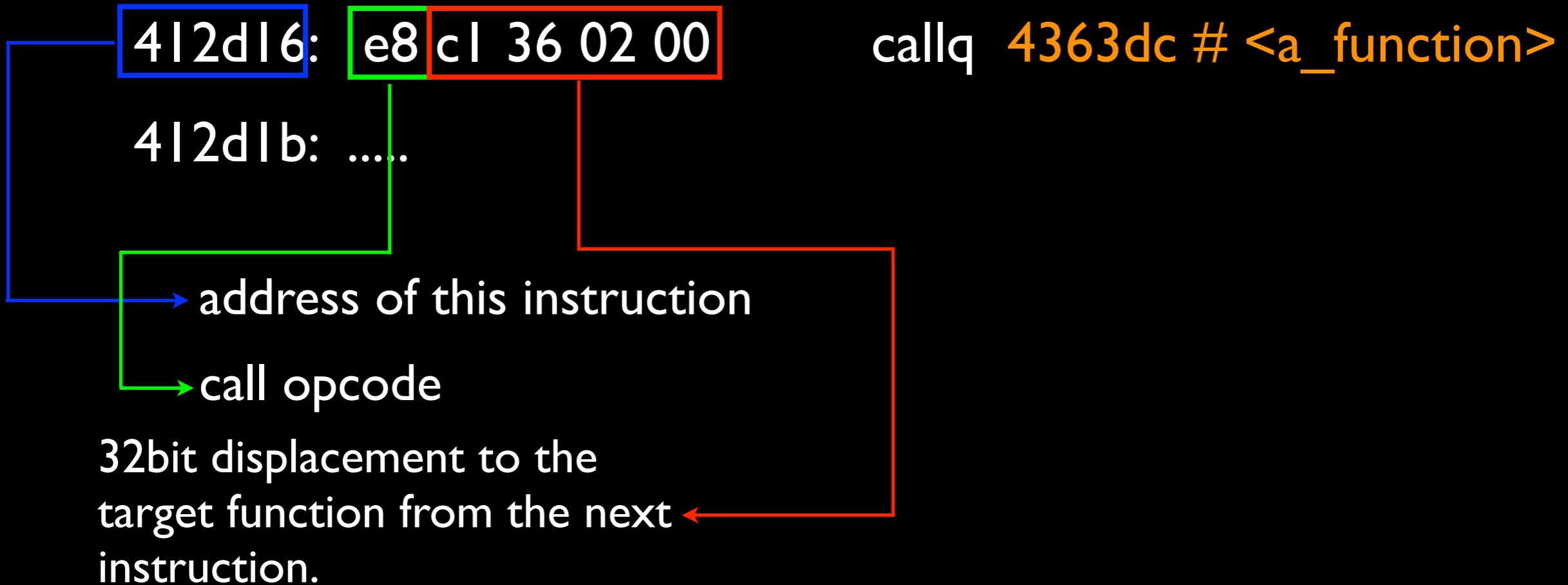
callq \*%rbx

callq 0xdeadbeef

other ways, too...

# anatomy of a call

(objdump output)



# anatomy of a call

(objdump output)



# Hook a\_function

Overwrite the **displacement** so that all calls to a\_function actually call a different function instead.

It may look like this:

```
int other_function()
{
    /* do something good/bad */

    /* be sure to call a_function! */
    return a_function();
}
```

# codez are easy

```
/* CHILL, it's fucking psuedo code */

while (are_moar_bytes()) {
    curr_ins = next_ins;
    next_ins = get_next_ins();
    if (curr_ins->type == INSN_CALL) {
        if ((hook_me - next_ins) == curr_ins->displacement) {
            /* found a call hook_me!*/
            rewrite(curr_ins->displacement, (replacement_fn - next_ins));
            return 0;
        }
    }
}
```

... right?.....



**NEAR MACHINERY**

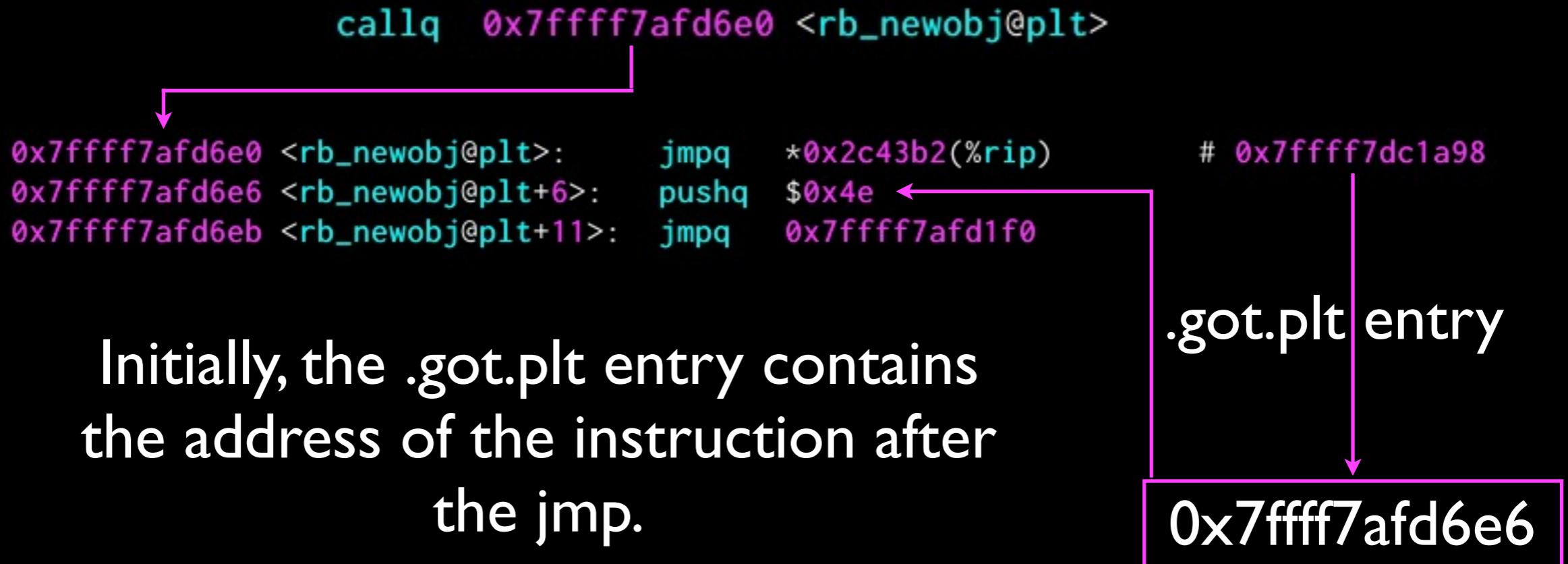
# 32bit displacement

- overwriting an existing call with another call
- stack will be aligned
- args are good to go
- can't redirect to code that is outside of:
  - $[\text{rip} + 32\text{bit displacement}]$
- you can scan the address space looking for an available page with `mmap`, though...

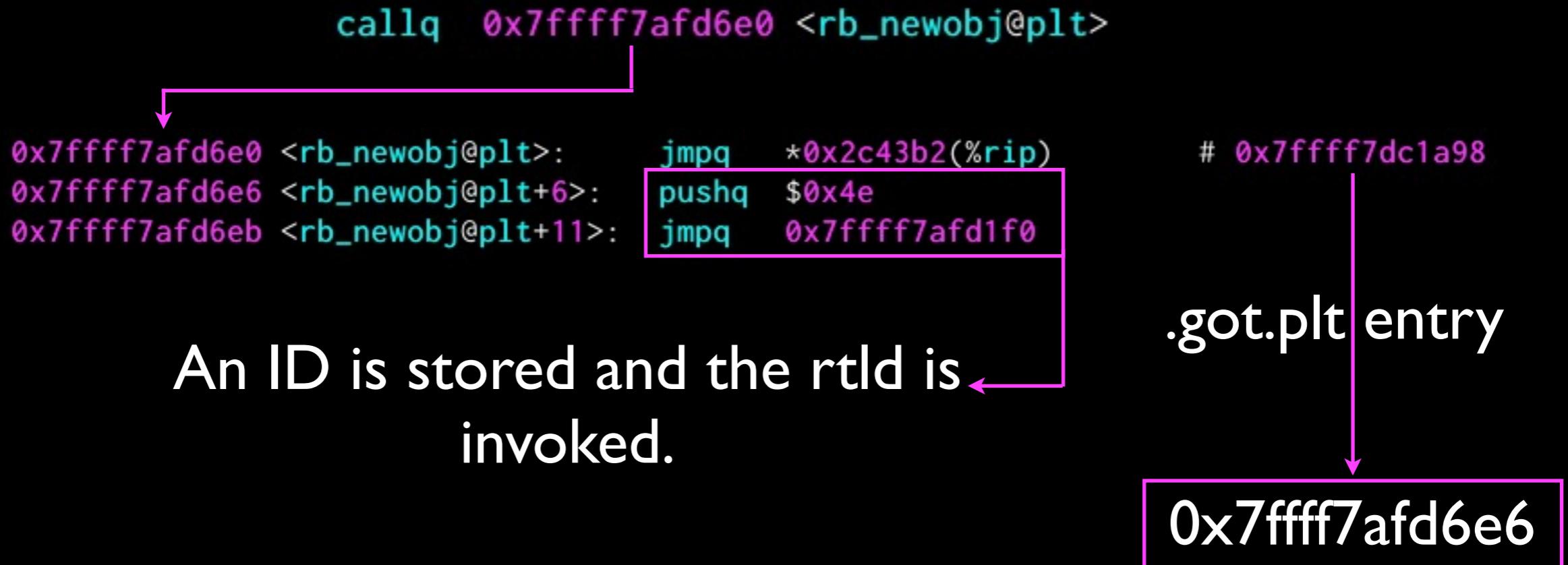
# Doesn't work for all

calling a function that is exported by a  
dynamic library **works differently.**

# How runtime dynamic linking works (elf)



# How runtime dynamic linking works (elf)



# How runtime dynamic linking works (elf)

```
callq 0xfffff7afd6e0 <rb_newobj@plt>
```

```
0xfffff7afd6e0 <rb_newobj@plt>: jmpq *0x2c43b2(%rip) # 0xfffff7dc1a98
0xfffff7afd6e6 <rb_newobj@plt+6>: pushq $0x4e
0xfffff7afd6eb <rb_newobj@plt+11>: jmpq 0xfffff7afd1f0
```

rtld writes the address of `rb_newobj` to the `.got.plt` entry.

`.got.plt` entry

**0xfffff7b34ac0**

# How runtime dynamic linking works (elf)

```
callq 0xfffff7afd6e0 <rb_newobj@plt>
```

```
0xfffff7afd6e0 <rb_newobj@plt>: jmpq *0x2c43b2(%rip) # 0xfffff7dc1a98
0xfffff7afd6e6 <rb_newobj@plt+6>: pushq $0x4e
0xfffff7afd6eb <rb_newobj@plt+11>: jmpq 0xfffff7afd1f0
```

rtld writes the address of rb\_newobj to the .got.plt entry.

.got.plt entry

0xfffff7b34ac0

calls to the PLT entry jump immediately to rb\_newobj now that .got.plt is filled in.

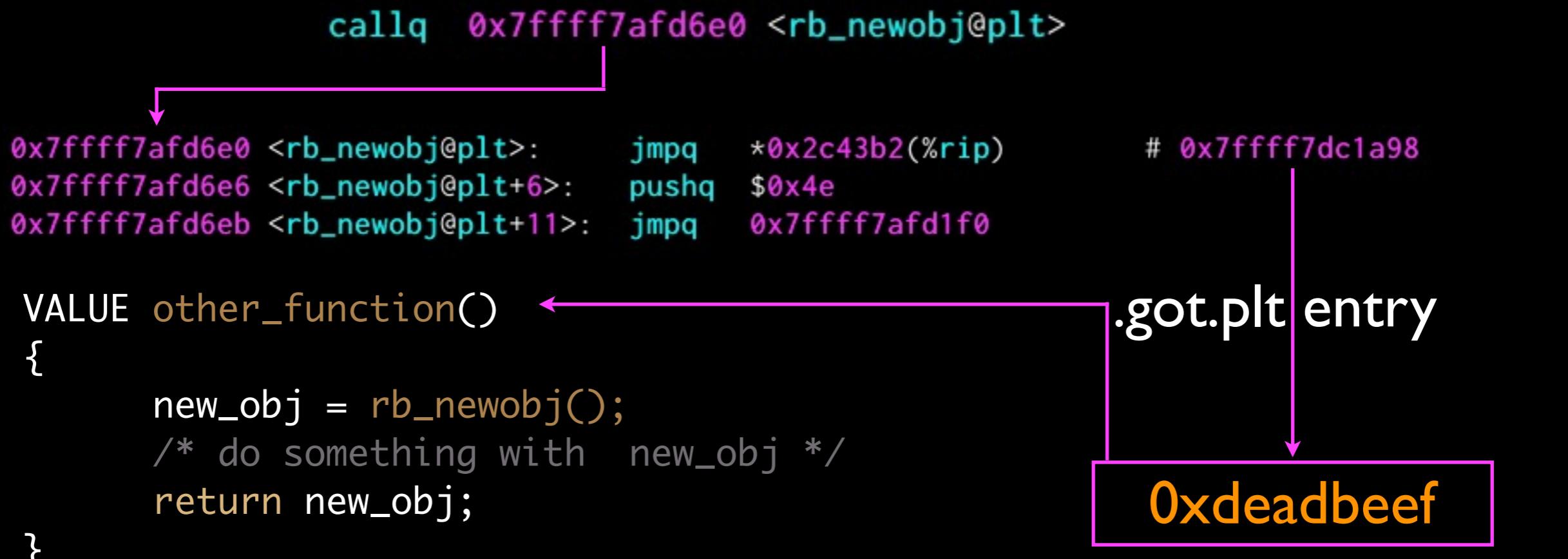
```
0x00007ffff7b34ac0 <rb_newobj+0>: sub    $0x8,%rsp ←
0x00007ffff7b34ac4 <rb_newobj+4>: mov    0x2a840a(%rip),%eax # 0xfffff7ddced4 <during_gc>
0x00007ffff7b34aca <rb_newobj+10>: test   %eax,%eax
```



# Hook the GOT

Redirect execution by overwriting all the .got.plt entries for `rb_newobj` in each DSO with a handler function instead.

# Hook the GOT

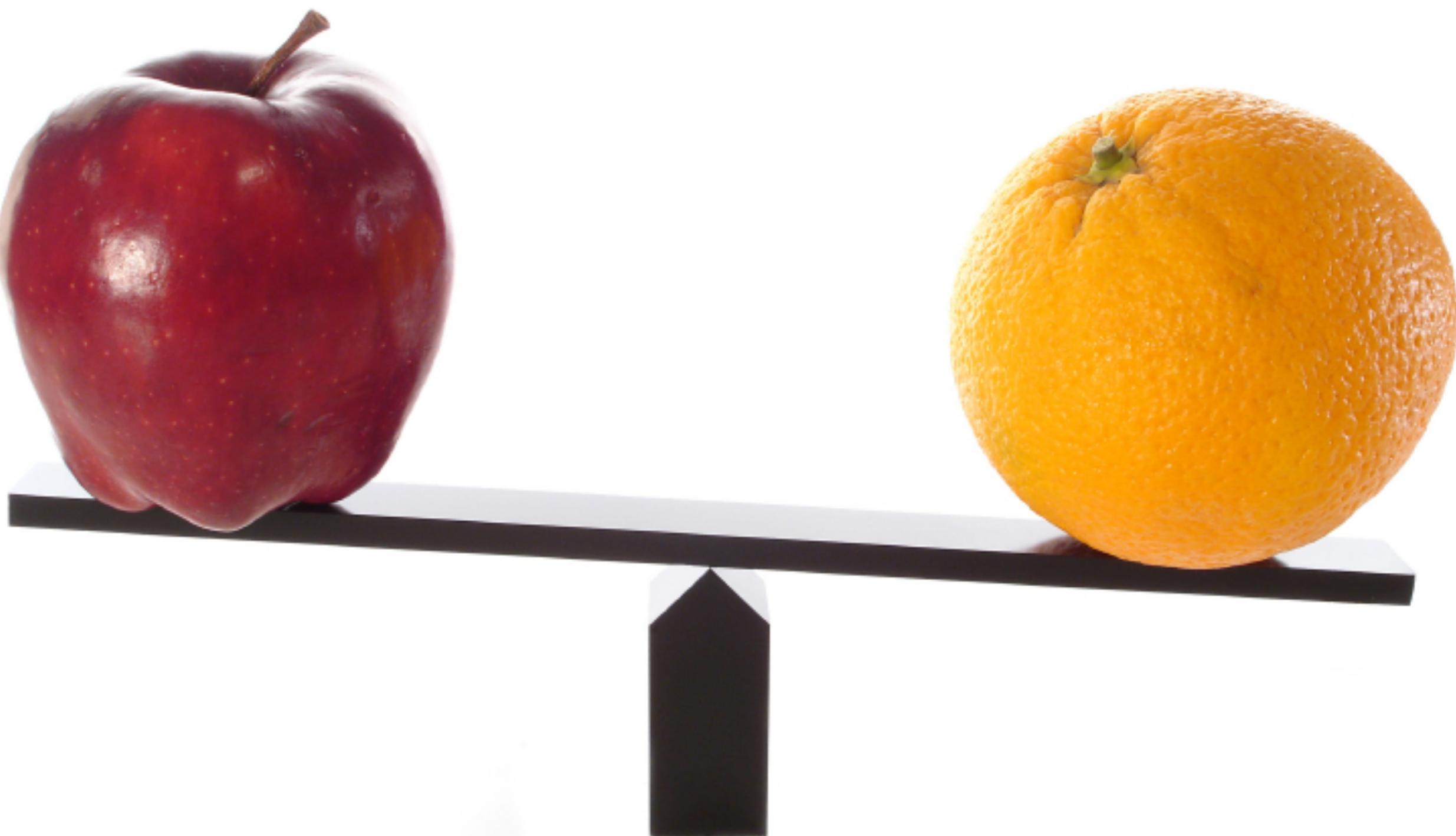


WAIT... `other_function()` calls `rb_newobj()` isn't that an infinite loop?

NO, it isn't. `other_function()` lives in its own DSO, so its calls to `rb_newobj()` use the `.plt/.got.plt` in its own DSO.

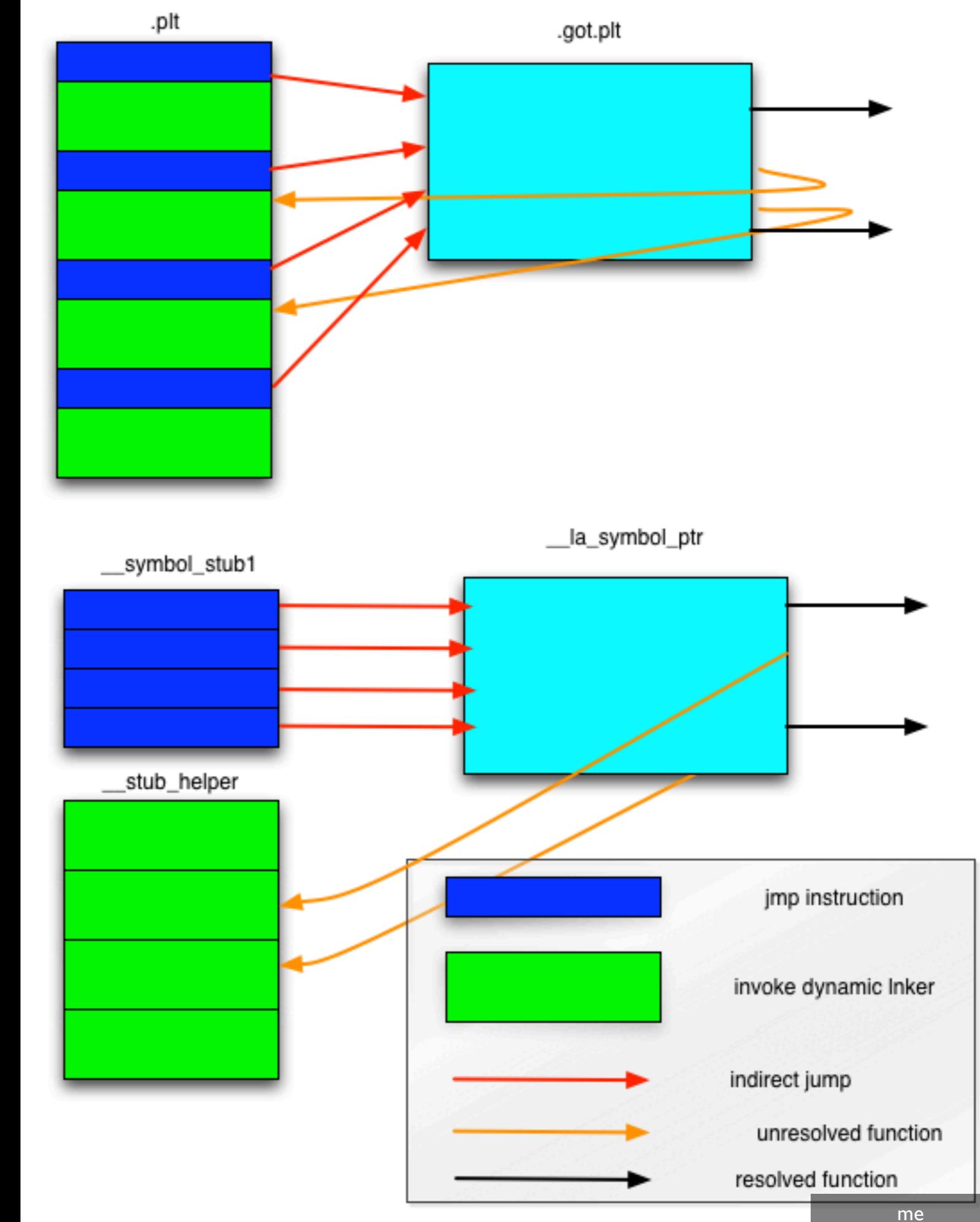
As long as we leave `other_function()`'s DSO unmodified, we'll avoid an infinite loop.





# mach-o

# elf



what else is left?

inline functions.

# add\_freelist

- Can't hook because add\_freelist is *inlined*:

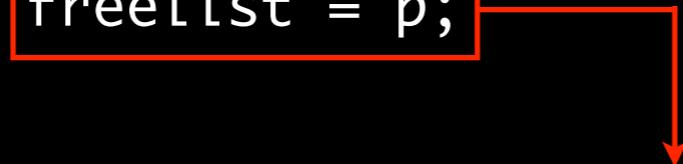
```
static inline void  
add_freelist(p)  
    RVALUE *p  
{  
    p->as.free.flags = 0;  
    p->as.free.next = freelist;  
    freelist = p;  
}
```

- The compiler has the option of inserting the instructions of this function directly into the callers.
- If this happens, you won't see any calls.

# So... what now?

- Look carefully at the code:

```
static inline void  
add_freelist(p)  
    RVALUE *p  
{  
    p->as.free.flags = 0;  
    p->as.free.next = freelist;  
    freelist = p;  
}
```



- Notice that freelist gets updated.
- freelist has file level scope.
- hmmmm.....

# A (stupid) crazy idea

- freelist has file level scope and lives at some static address.
- add\_freelist updates freelist, so...
- Why not search the binary for *mov instructions that have freelist as the target!*
- Overwrite that mov instruction with a call to our code!
- But... we have a problem.
- The system isn't ready for a call instruction.

# alignment



# calling convention

# Isn't ready? What?

- The 64bit ABI says that the stack must be aligned to a 16byte boundary after any/all arguments have been arranged.
- Since the overwrite is just some random mov, no way to guarantee that the stack is aligned.
- If we just plop in a call instruction, we won't be able to arrange for arguments to get put in the right registers.
- So now what?

# jmp

- Can use a jmp instruction.
- Transfer execution to an assembly stub  
**generated at runtime.**
  - recreate the overwritten instruction
  - set the system up to call a function
- do something good/bad
- jmp back when done to resume execution



# checklist

- save and restore caller/callee saved registers.
- align the stack.
- recreate what was overwritten.
- arrange for any arguments your replacement function needs to end up in registers.
- invoke your code.
- resume execution as if nothing happened.

this instruction updates the freelist and comes from  
add\_freelist:

```
48 89 1d 1a 1a 2c 00      mov    %rbx,0x2c1a1a(%rip)      # 0x6f5aa0 <freelist>
```

Can't overwrite it with a call instruction because the state of the system is not ready for a function call.

```
e9 e3 8d bc 3f          jmpq   0x40000800
90
90
nop
nop
```

The jmp instruction and its offset are 5 bytes wide. Can't grow or shrink the binary, so insert 2 one byte NOPs.

# shortened assembly

## stub

```
mov    %rbx,-0x3f8eaa6f(%rip)          # recreate overwritten instruction
push   %rax                           # save %rax incase the handler destroys it
push   %rdi                           # save %rdi, we need it to pass arg 1
mov    -0x3f8eaa77(%rip),%rdi         # mov top of freelist to rdi (arg 1 to handler)
push   %rbx                           # save rbx
push   %rbp                           # save rbp
mov    %rsp,%rbp                      # set base pointer to current stack pointer
and    $0xfffffffffffffff0,%rsp        # align stack to conform with 64bit ABI
mov    $0x7fff6a479b4,%rbx            # mov the handler address into %rbx
callq  *%rbx                          # call handler via %rbx
leaveq
pop    %rbx                           # mov rbp, rsp; pop rbp
pop    %rdi                           # restore rdi
pop    %rax                           # restore rax
jmpq   0x437a1f <gc_sweep+1096>      # continue execution
```

# shortened assembly

## stub

```
mov    %rbx,-0x3f8eaa6f(%rip)          # recreate overwritten instruction
push   %rax                           # save %rax incase the handler destroys it
push   %rdi                           # save %rdi, we need it to pass arg 1
mov    -0x3f8eaa77(%rip),%rdi         # mov top of freelist to rdi (arg 1 to handler)
push   %rbx                           # save rbx
push   %rbp                           # save rbp
mov    %rsp,%rbp                      # set base pointer to current stack pointer
and    $0xfffffffffffffff0,%rsp        # align stack to conform with 64bit ABI
mov    $0x7fff6a479b4,%rbx            # mov the handler address into %rbx
callq  *%rbx                          # call handler via %rbx
leaveq
pop    %rbx                           # mov rbp, rsp; pop rbp
pop    %rdi                           # restore rdi
pop    %rax                           # restore rax
jmpq   0x437a1f <gc_sweep+1096>      # continue execution
```

# shortened assembly

## stub

```
mov    %rbx,-0x3f8eaa6f(%rip)          # recreate overwritten instruction
push   %rax                           # save %rax incase the handler destroys it
push   %rdi                           # save %rdi, we need it to pass arg 1
mov    -0x3f8eaa77(%rip),%rdi         # mov top of freelist to rdi (arg 1 to handler)
push   %rbx                           # save rbx
push   %rbp                           # save rbp
mov    %rsp,%rbp                      # set base pointer to current stack pointer
and    $0xfffffffffffffff0,%rsp        # align stack to conform with 64bit ABI
mov    $0x7fff6a479b4,%rbx            # mov the handler address into %rbx
callq  *%rbx                          # call handler via %rbx
leaveq
pop    %rbx                           # mov rbp, rsp; pop rbp
pop    %rdi                           # restore rdi
pop    %rax                           # restore rxax
jmpq   0x437a1f <gc_sweep+1096>      # continue execution
```

# shortened assembly

## stub

```
mov    %rbx,-0x3f8eaa6f(%rip)          # recreate overwritten instruction
push   %rax                           # save %rax incase the handler destroys it
push   %rdi                           # save %rdi, we need it to pass arg 1
mov    -0x3f8eaa77(%rip),%rdi        # mov top of freelist to rdi (arg 1 to handler)
push   %rbx                           # save rbx
push   %rbp                           # save rbp
mov    %rsp,%rbp                      # set base pointer to current stack pointer
and    $0xfffffffffffffff0,%rsp       # align stack to conform with 64bit ABI
mov    $0x7fff6a479b4,%rbx           # mov the handler address into %rbx
callq  *%rbx                          # call handler via %rbx
leaveq
pop    %rbx                           # mov rbp, rsp; pop rbp
pop    %rdi                           # restore rdi
pop    %rax                           # restore rxax
jmpq   0x437a1f <gc_sweep+1096>     # continue execution
```

# shortened assembly

## stub

```
mov    %rbx,-0x3f8eaa6f(%rip)          # recreate overwritten instruction
push   %rax                           # save %rax incase the handler destroys it
push   %rdi                           # save %rdi, we need it to pass arg 1
mov    -0x3f8eaa77(%rip),%rdi         # mov top of freelist to rdi (arg 1 to handler)
push   %rbx                           # save rbx
push   %rbp                           # save rbp
mov    %rsp,%rbp                      # set base pointer to current stack pointer
and    $0xfffffffffffffff0,%rsp        # align stack to conform with 64bit ABI
mov    $0x7fff6a479b4,%rbx            # mov the handler address into %rbx
callq  *%rbx                          # call handler via %rbx
leaveq
pop    %rbx                           # mov rbp, rsp; pop rbp
pop    %rdi                           # restore rdi
pop    %rax                           # restore rax
jmpq   0x437a1f <gc_sweep+1096>      # continue execution
```

# shortened assembly stub

```
mov    %rbx,-0x3f8eaa6f(%rip)          # recreate overwritten instruction
push   %rax                           # save %rax incase the handler destroys it
push   %rdi                           # save %rdi, we need it to pass arg 1
mov    -0x3f8eaa77(%rip),%rdi         # mov top of freelist to rdi (arg 1 to handler)
push   %rbx                           # save rbx
push   %rbp                           # save rbp
mov    %rsp,%rbp                      # set base pointer to current stack pointer
and    $0xfffffffffffff0,%rsp          # align stack to conform with 64bit ABI
mov    $0x7fff6a479b4,%rbx            # mov the handler address into %rbx
callq  *%rbx                          # call handler via %rbx
leaveq
pop    %rbx                           # restore rbx
pop    %rdi                           # restore rdi
pop    %rax                           # restore rax
jmpq   0x437a1f <gc_sweep+1096>      # continue execution
```

# shortened assembly

## stub

```
mov    %rbx,-0x3f8eaa6f(%rip)          # recreate overwritten instruction
push   %rax                           # save %rax incase the handler destroys it
push   %rdi                           # save %rdi, we need it to pass arg 1
mov    -0x3f8eaa77(%rip),%rdi         # mov top of freelist to rdi (arg 1 to handler)
push   %rbx                           # save rbx
push   %rbp                           # save rbp
mov    %rsp,%rbp                      # set base pointer to current stack pointer
and    $0xfffffffffffff0,%rsp          # align stack to conform with 64bit ABI
mov    $0x7ffff6a479b4,%rbx          # mov the handler address into %rbx
callq  *%rbx                          # call handler via %rbx
leaveq
pop    %rbx                           # restore rbx
pop    %rdi                           # restore rdi
pop    %rax                           # restore rax
jmpq   0x437a1f <gc_sweep+1096>      # continue execution
```

# shortened assembly

## stub

```
mov    %rbx,-0x3f8eaa6f(%rip)          # recreate overwritten instruction
push   %rax
push   %rdi
mov    -0x3f8eaa77(%rip),%rdi        # mov top of freelist to rdi (arg 1 to handler)
push   %rbx
push   %rbp
mov    %rsp,%rbp
and    $0xfffffffffffffff0,%rsp       # align stack to conform with 64bit ABI
mov    $0x7ffff6a479b4.%rbx          # mov the handler address into %rbx
callq  *%rbx                         # call handler via %rbx
leaveq
pop    %rbx
pop    %rdi
pop    %rax
jmpq  0x437a1f <gc_sweep+1096>     # continue execution
```

```
void handler(VALUE freed_object)
{
    mark_object_freed(freed_object);
    return;
}
```

# shortened assembly

## stub

```
mov    %rbx,-0x3f8eaa6f(%rip)          # recreate overwritten instruction
push   %rax                           # save %rax incase the handler destroys it
push   %rdi                           # save %rdi, we need it to pass arg 1
mov    -0x3f8eaa77(%rip),%rdi         # mov top of freelist to rdi (arg 1 to handler)
push   %rbx                           # save rbx
push   %rbp                           # save rbp
mov    %rsp,%rbp                      # set base pointer to current stack pointer
and    $0xfffffffffffffff0,%rsp        # align stack to conform with 64bit ABI
mov    $0x7ffff6a479b4,%rbx           # mov the handler address into %rbx
callq  *%rbx                          # call handler via %rbx
leaveq
pop    %rbx                           # mov rbp, rsp; pop rbp
pop    %rdi                           # restore rdi
pop    %rax                           # restore rax
jmpq   0x437a1f <gc_sweep+1096>      # continue execution
```

and it actually works.

gem install memprof

<http://github.com/ice799/memprof>



# Sample Output

```
require 'memprof'  
Memprof.start  
require "stringio"  
StringIO new  
Memprof.stats
```



```
108 /custom/ree/lib/ruby/1.8/x86_64-linux/stringio.so:0:__node__  
14 test2.rb:3:String  
2 /custom/ree/lib/ruby/1.8/x86_64-linux/stringio.so:0:Class  
1 test2.rb:4:StringIO  
1 test2.rb:4:String  
1 test2.rb:3:Array  
1 /custom/ree/lib/ruby/1.8/x86_64-linux/stringio.so:0:Enumerable
```

# memprof.com

a web-based heap visualizer and leak analyzer

**new rails3-beta application** by **tmm1** about a month ago

## [ruby-1.8.7-p249/bin/ruby](#)

- ruby 1.8.7 (2010-01-10 patchlevel 249) [i686-darwin10.2.0]
- executing `./script/rails`
- compiled with `-g -O2 -fno-common -pipe -fno-common $(cflags)`
- memory usage is 97156 bytes
- working directory is `test/code/newapp`
- 6 IO objects and 10 file descriptors
- 20 shared libraries

## [404869 objects](#)

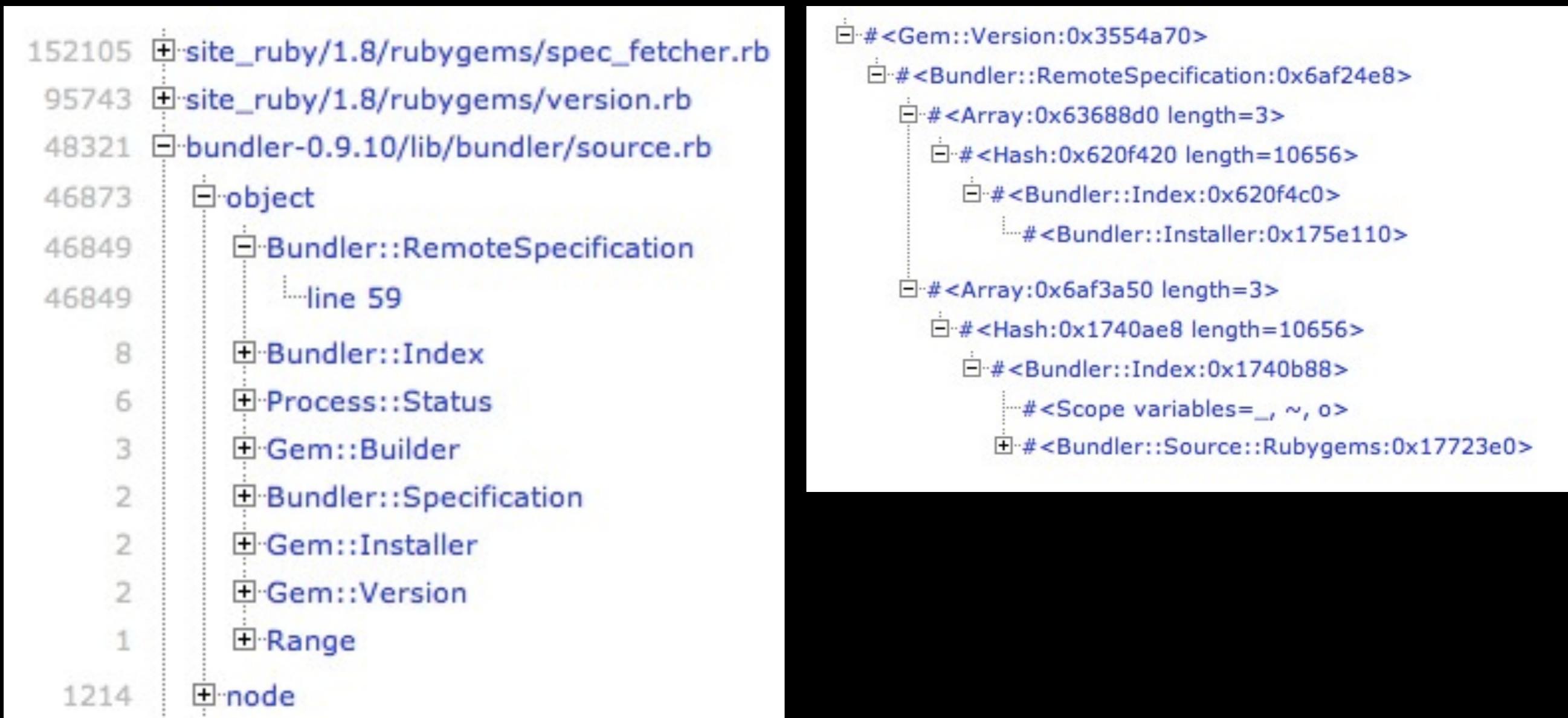
- 78 global variables
- 213 constants inside Object
- objects grouped by age
- [objects grouped by type](#)
- objects with most outbound references

## [2428 classes and 695 modules](#)

- namespace hierarchy
- class hierarchy
- instances per class
- duplicate classes by name

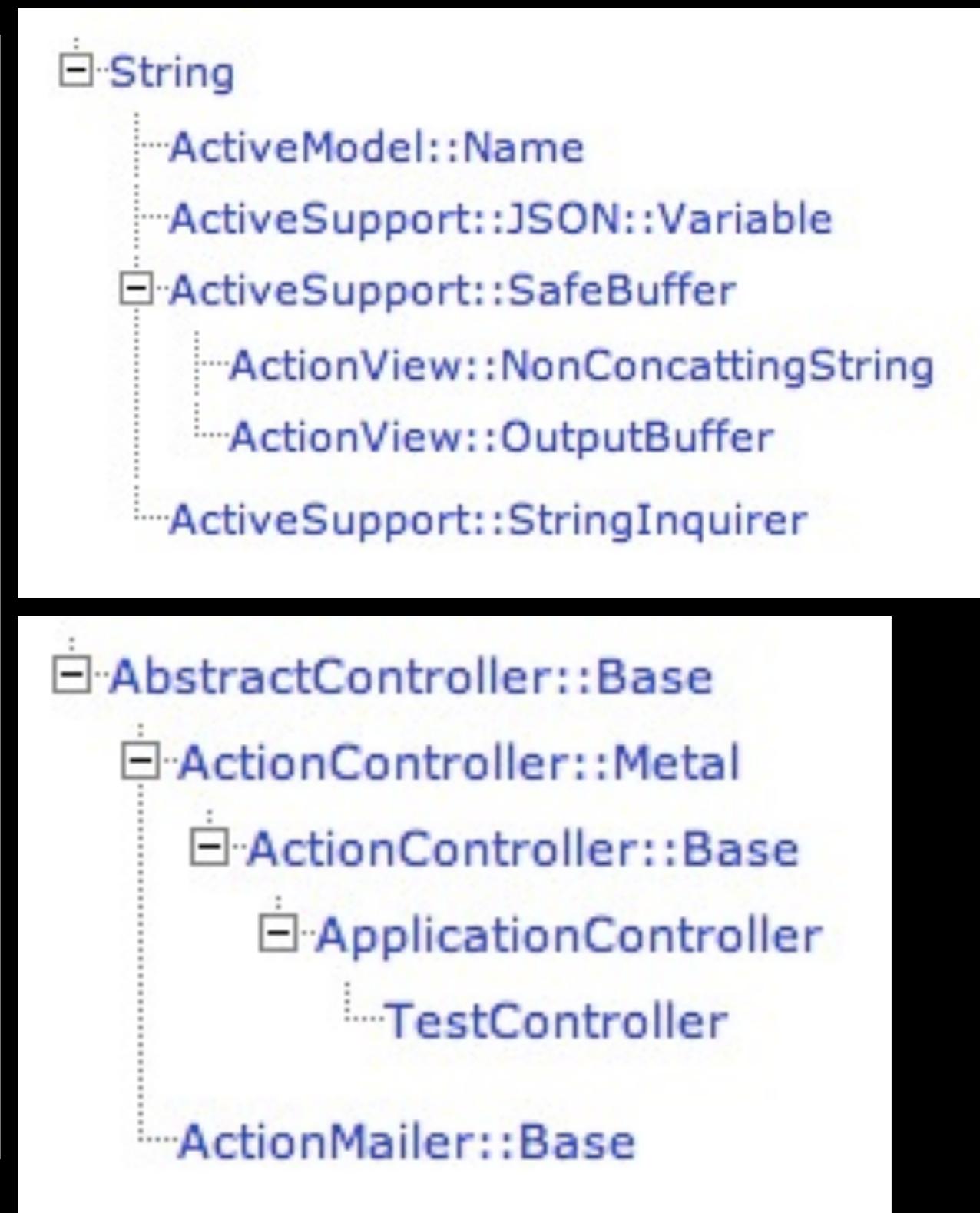
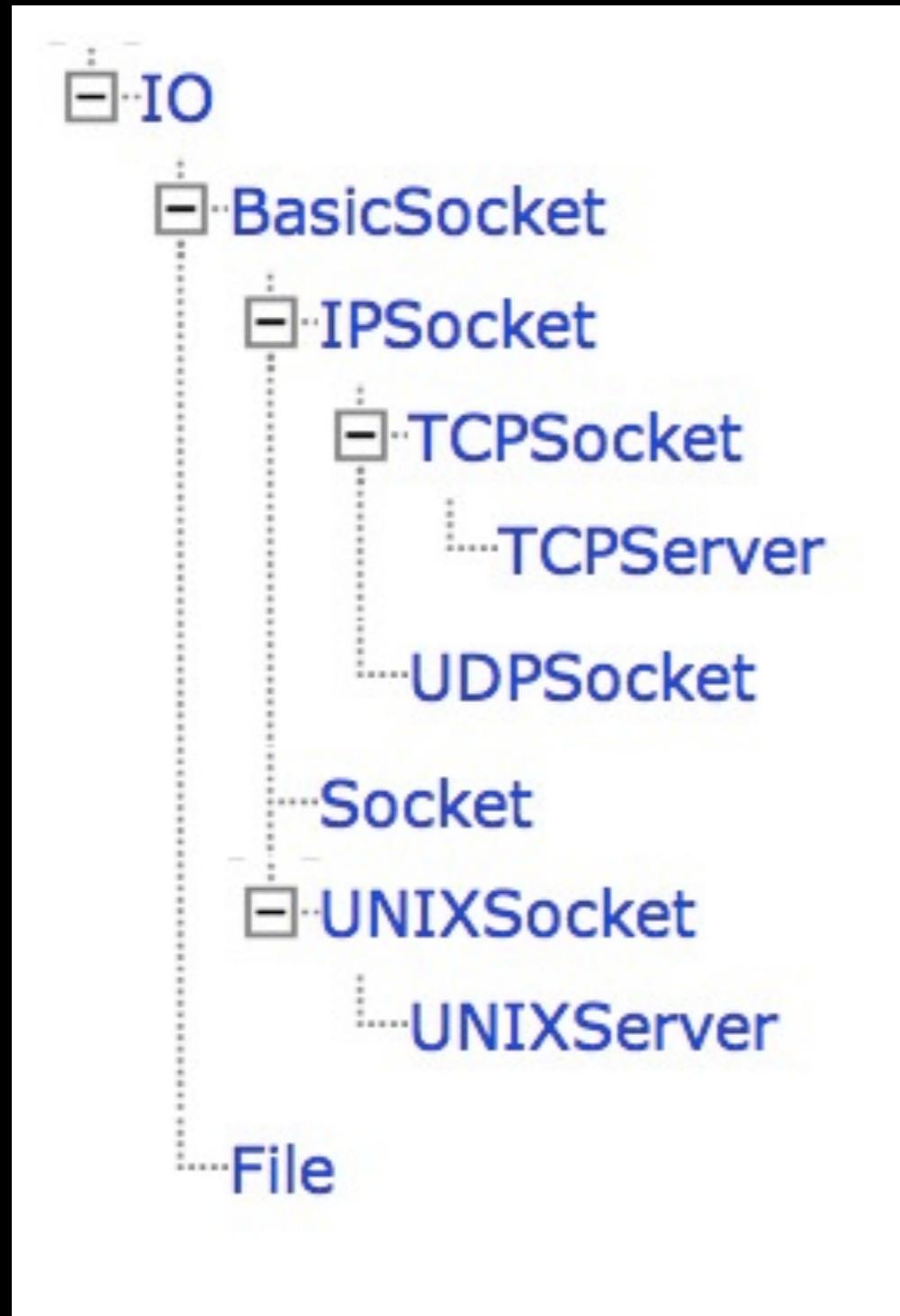
# memprof.com

a web-based heap visualizer and leak analyzer



# memprof.com

a web-based heap visualizer and leak analyzer



# memprof.com

a web-based heap visualizer and leak analyzer

```
address #<Array:0x6279948 length=4096>
file gems/googlecharts-1.3.6/lib/gchart.rb
line 15
type array
class Array
length 4096
```

0 "AA"

1 "AB"

2 "AC"

3 "AD"

4 "AE"

```
def self.simple_chars
  @simple_chars ||= ('A'..'Z').to_a + ('a'..'z').to_a + ('0'..'9').to_a
```

6 "AG"

```
end
```

7 "AH"

```
def self.chars
```

8 "AI"

```
  @chars ||= simple_chars + ['-','.']}
```

9 "AJ"

```
end
```

10 "AK"

```
def self.ext_pairs
```

11 "AL"

```
  @ext_pairs ||= chars.map { |char_1| chars.map { |char_2| char_1 + char_2 } }.flatten
end
```

# memprof.com

a web-based heap visualizer and leak analyzer

```
address node:WHITE
  type node
node_type WHILE
  file lib/ruby/1.8/singleton.rb
  line 147

  n1 node:CALL
  n2 node:BLOCK
  n3 0

while false.equal?(@__instance__)
  do
    Thread.critical = false
    sleep(nil)
    Thread.critical = true
end
```

```
address node:OP_ASZN2
  type node
node_type OP_ASZN2
  file ruby/1.8/date/format.rb
  line 551

  n1 node:LVAR
  n2 node:IF
  n3 node:OP_ASZN2
```

```
e._cent ||= (val >= 69) ? (19) : (20)
```

```
{"_id": "0x35da08"}
```

1 object [detail](#) [references](#)

```
address node:DEFN
  type node
node_type DEFN
  file lib/ruby/1.8/delegate.rb
  line 267

  n1 true
  n2 :method_missing
  n3 node:SCOPE

def method_missing(m, *args, &block)
  super(m, *args, &block) unless @_dc_obj.respond_to?(m)
  @_dc_obj.__send__(m, *args, &block)
end
```

# memprof.com

a web-based heap visualizer and leak analyzer

```
{"type": "file"}
```

4 objects [list](#) [group](#)

(REG:txt) i686-darwin10.2.0/digest/sha1.bundle  
(REG:txt) 1.8/i686-darwin10.2.0/digest.bundle  
(REG:txt) 1.8/i686-darwin10.2.0/strscan.bundle  
(REG:txt) 1.8/i686-darwin10.2.0/fcntl.bundle  
(REG:txt) i686-darwin10.2.0/racc/cparse.bundle  
(REG:txt) 1.8/i686-darwin10.2.0/zlib.bundle  
(REG:txt) 1.8/i686-darwin10.2.0/socket.bundle  
(REG:txt) 1.8/i686-darwin10.2.0/openssl.bundle  
(REG:txt) 1.8/i686-darwin10.2.0/nkf.bundle  
(REG:txt) eventmachine-0.12.10/lib/rubyeventmachine.bundle

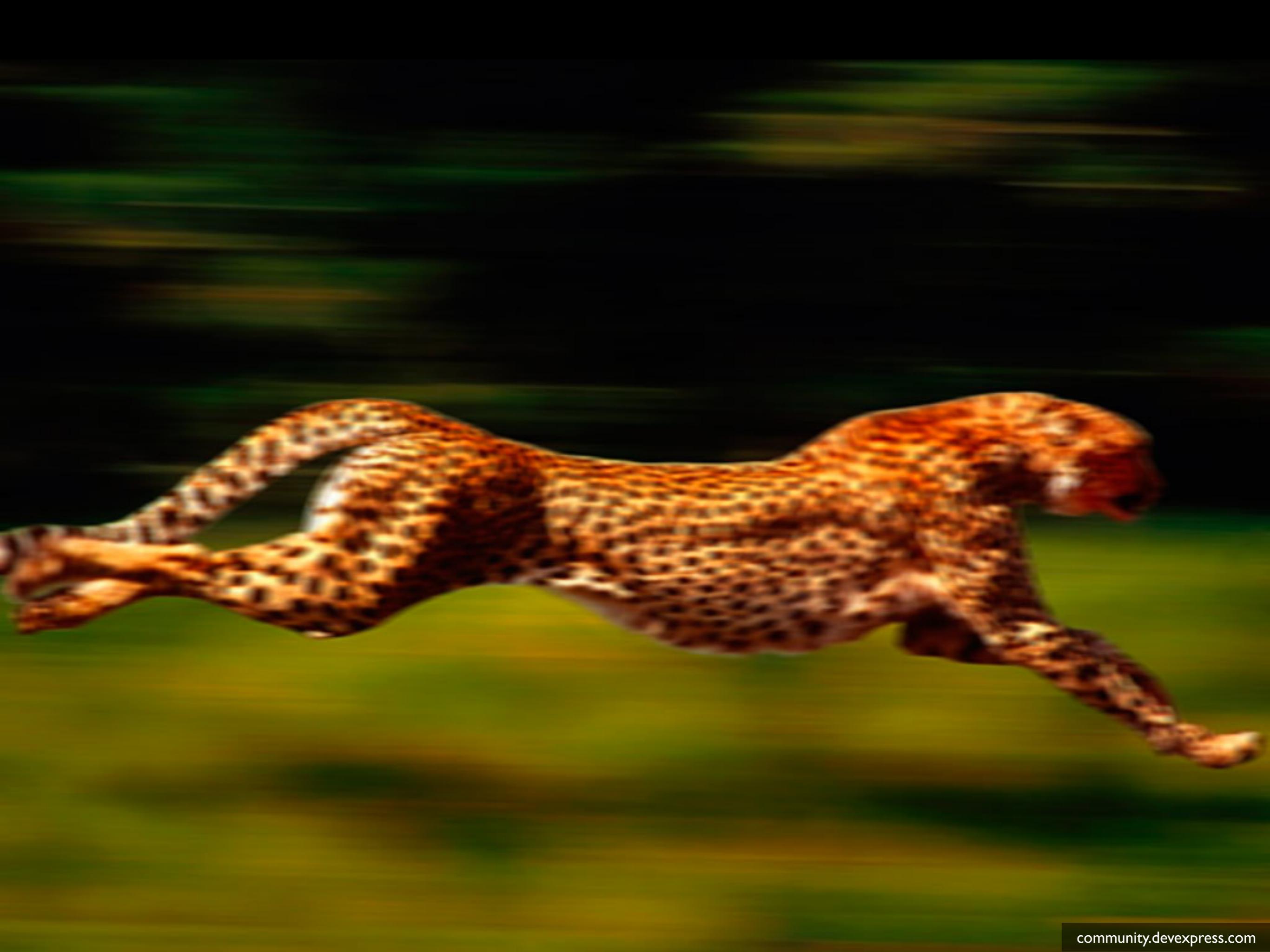
0x1a6ae8 #<TCPSocket:0x1a6ae8>

[IO:0x1b5b88>](#)  
[IO:0x1b5bb0>](#)  
[IO:0x1b5bd8>](#)

```
{"_id": "0x1a6ae8"}
```

1 object [detail](#) [references](#)

**address** #<TCPSocket:0x1a6ae8>  
**file** -e  
**line** 1  
**time** 1269746382129610  
**type** file  
**class** TCPSocket  
**fileno** (IPv4:3u) 192.168.1.138:54337 -> 74.125.19.105:http (ESTABLISHED)  
**mode** readable  
writable  
readwrite  
sync



```
config.middleware.use(Memprof::Tracer)

{
  "time": 4.3442,           ← total time for request
  "rails": {
    "controller": "test",
    "action": "index"
  },
  "request": {               ← request env info
    "REQUEST_PATH": "/test",
    "REQUEST_METHOD": "GET"
  },
}
```

```
config.middleware.use(Memprof::Tracer)
```

```
"mysql": {  
    "queries": 3,           ← 3 mysql queries  
    "time": 0.00109302  
},
```

```
"gc": {  
    "calls": 8,            ← 8 calls to GC  
    "time": 2.04925        ← 2 secs spent in GC  
},
```

```
config.middleware.use(Memprof::Tracer)

"objects": {
  "created": 3911103, ← 3 million objs created
  "types": {
    "none": 1168831, ← 1 million method calls
    "object": 1127, ← object instances
    "float": 627,
    "string": 1334637, ← lots of strings
    "array": 609313, ← lots of arrays
    "hash": 3676,
    "match": 70211 ← regexp matches
  }
}
}
```





## SOCIAL ENGINEERING

Verify requests for sensitive information and

# evil lives

<http://github.com/ice799/memprof/tree/dnw>

- makes ruby faster! ||!!|
- hooks read syscall
- looks for magic cookie (JOE)
- turns off GC
- Ruby is fast.

it makes ruby faster!!!

look a bullshit  
benchmark!

# it makes ruby faster!!!

#NORMAL RUBY!!!!!!

```
[joe@mawu:/Users/joe/code/defcon/memprof/ext]% ab -c 10 -n 200 http://blah:  
4567/hi/JOE
```

Benchmarking blah (be patient)

Completed 100 requests

Completed 200 requests

Finished 200 requests

Concurrency Level: 10

Time taken for tests: 7.462 seconds

Complete requests: 200

Failed requests: 0

Write errors: 0

Requests per second: 26.80 [#/sec] (mean)

Time per request: 373.108 [ms] (mean)

Time per request: 37.311 [ms] (mean, across all concurrent requests)

# it makes ruby faster!!!

```
# fast0r RUBY!!!|||!!!
```

```
[joe@mawu:/Users/joe/code/defcon]% ab -c 10 -n 200 http://blah:4567/hi/JOE
```

Benchmarking blah (be patient)

Completed 100 requests

Completed 200 requests

Finished 200 requests

Concurrency Level: 10

Time taken for tests: 6.594 seconds

Complete requests: 200

Failed requests: 0

Write errors: 0

Requests per second: 30.33 [#/sec] (mean)

Time per request: 329.708 [ms] (mean)

Time per request: 32.971 [ms] (mean, across all concurrent requests)

# you can do anything

- this example is stupid, but you can do anything.
- hook read/write and phone home with data.
- fork a backdoor when a specific cookie is seen
- whatever



BREAK  
[break.com](http://break.com)



# injectso

- written by Shaun Clowes
- injects libraries into *running* processes using ptrace(2).
- super clever hack!

**FUCKING MAGNETS**

**HOW DO THEY  
WORK**

# injecting live processes

- `ptrace(2)`
- allows you to view and modify the register set and address space of another process
- permissions on memory are ignored

# fucking injectso, how does it work?

- attach to target process using ptrace
- save a copy of a small piece of the program stack.
- save a copy of the register set
- create a fake stack frame with a saved return address of 0

# fucking injectso, how does it work?

- set register set to point at `dlopen`
  - `rip` = `&dlopen`
  - `rdi` = dso name
  - `rsi` = mode
- let er `rip`, `waitpid` and it'll segfault on return to 0.
- restore stack, register set, resume as normal.

# ptrace

- remote allocating memory is a pain in the ass.
- generating segfaults in running processes might be bad (core dumps, etc).
- binary patching is hard, doing it with ptrace is harder.

# evil dso

- getting the user to use your library might be hard.
- already running processes will need to be killed first.
- need to poison each time app is started.
- binary patching is hard.



# combine 'em

- use injectso hack to load an evil dso
- evil dso will take it from there

# 64bit injectso port

- ported by Stealth
- <http://c-skills.blogspot.com/2007/05/injectso.html>
- i did some trivial cleanup and put the codez on github
- <http://github.com/ice799/injectso64>
- tested it on 64bit ubuntu VM, works.

injectso

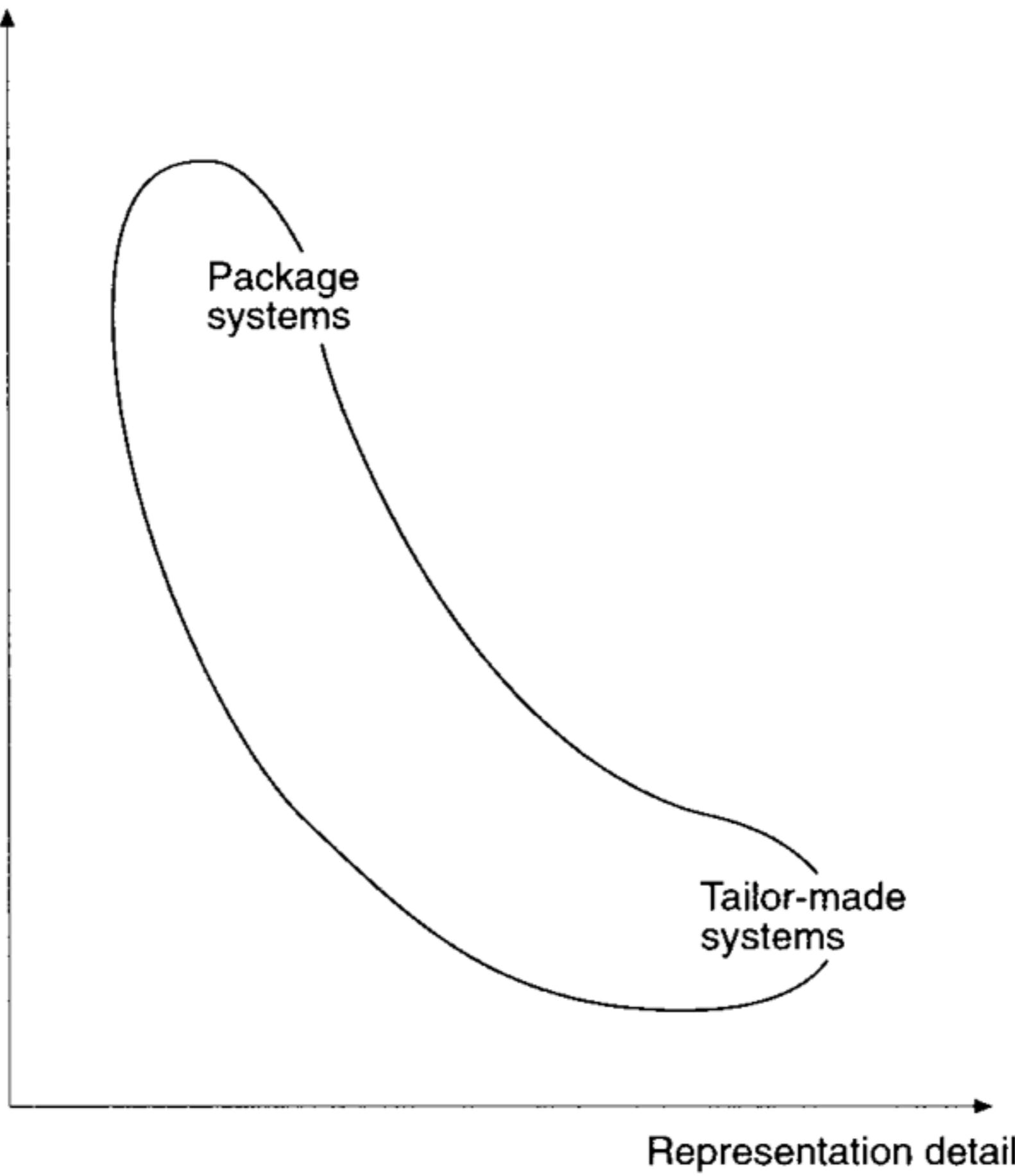
+

evil-binary-patching-dso





Speed of implementation

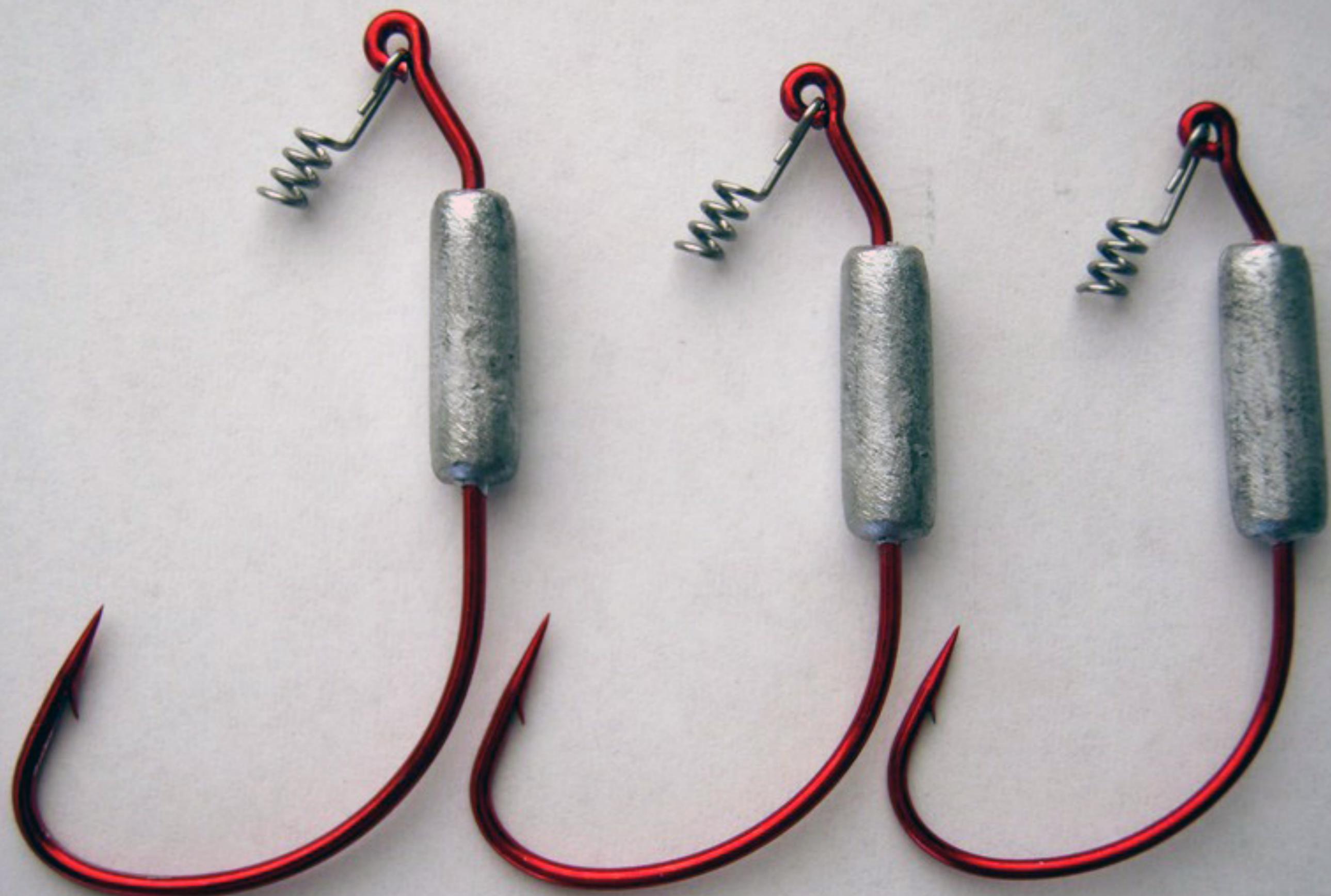


# how to defend against it

- NX bit
  - call mprotect
- strip debug information
  - mostly prebuilt binaries
- statically link everything
  - extremely large binaries
- put all .text code in ROM
  - maybe?
- don't load DSOs at runtime
  - no plugins, though
- disable ptrace
  - no gdb/strace.
- check /proc/<pid>/maps
  - word.



my future research:  
exploring alternative  
binary formats.





**Before**



**After**

# alignment



# calling convention

# object file and library formats



# questions?

joe damato

@joedamato

[timetobleed.com](http://timetobleed.com)

<http://timetobleed.com/string-together-global-offset-tables-to-build-a-ruby-memory-profiler/>

[http://timetobleed.com/hot-patching-inlined-functions-with-x86\\_64-asm-metaprogramming/](http://timetobleed.com/hot-patching-inlined-functions-with-x86_64-asm-metaprogramming/)

<http://timetobleed.com/rewrite-your-ruby-vm-at-runtime-to-hot-patch-useful-features/>

<http://timetobleed.com/dynamic-linking-elf-vs-mach-o/>

<http://timetobleed.com/dynamic-symbol-table-duel-elf-vs-mach-o-round-2/>



Worth<sup>1000</sup>.com

tallteacher.files.wordpress.com

# “Interesting Behavior of OS X”

- Steven Edwards (winehacker@gmail.com)
- november 29 2007
- <http://www.winehq.org/pipermail/wine-devel/2007-November/060846.html>

# leopard has a pe loader?

```
handle = dlopen("./procexp.exe", RTLD_NOW | RTLD_FIRST );
```

```
steven-edwardss-imac:temp sedwards$ ./a.out
dlopen("./procexp.exe", 258): Library not loaded: WS2_32.dll
Referenced from: /Users/sedwards/Library/Application
Support/CrossOver/Bottles/winetest/drive_c/windows/temp/
procexp.exe
Reason: image not found
```

