function hooking for osx and linux

joe damato
@joedamato
timetobleed.com
slides are on timetobleed.com
i’m not a security researcher.
call me a script kiddie: @joedamato
assembly is in att syntax
WTF is an ABI ?
WTF is an Application Binary Interface?
alignment
calling convention
object file and library formats
hierarchy of specs
System V ABI (271 pages)

System V ABI AMD64 Architecture Processor Supplement (128 pages)

System V ABI Intel386 Architecture Processor Supplement (377 pages)

MIPS, ARM, PPC, and IA-64 too!
mac osx x86-64 calling convention

based on

System V ABI AMD64 Architecture Processor Supplement
alignment
end of argument area must be aligned on a 16-byte boundary.

and $0xfffffffffffffff0, %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 saved
object file and library formats
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

Header

Load commands

Segment command 1
Segment command 2

Data

Segment 1
Section 1 data
Section 2 data
Section 3 data

Segment 2
Section 4 data
Section 5 data
...
Section n data
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_stub1 - 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

- Symtabs 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 /usr/bin/ruby

```
000000000004ac90 T Balloc
00000000000491270 T Init_Array
00000000000497520 T Init_Bignum
0000000000041dc80 T Init_Binding
0000000000049d9b0 T Init_Comparable
0000000000049de30 T Init_Dir
000000000004a1080 T Init_Enumerable
000000000004a3720 T Init_Enumerator
000000000004a4f30 T Init_Exception
0000000000042c2d0 T Init_File
00000000000434b90 T Init_GC
```

symbol "value"

symbol names
**objdump**

% objdump -D /usr/bin/ruby

<table>
<thead>
<tr>
<th>offsets</th>
<th>opcodes</th>
<th>instructions</th>
<th>helpful metadata</th>
</tr>
</thead>
<tbody>
<tr>
<td>434860:</td>
<td>48 83 ec 08</td>
<td>sub $0x8,%rsp</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>434864:</td>
<td>8b 05 82 12 2c 00</td>
<td>mov 0x2c1282(%rip),%eax</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>43486a:</td>
<td>85 c0</td>
<td>test %eax,%eax</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>43486c:</td>
<td>75 6b</td>
<td>jne 4348d9 &lt;rb_newobj+0x79&gt;</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>43486e:</td>
<td>48 83 3d 3a 85 2a 00</td>
<td>cmpq $0x0,0x2a853a(%rip)</td>
<td># 6df5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>434875:</td>
<td>00</td>
<td>je 4348d0 &lt;rb_newobj+0x70&gt;</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>434876:</td>
<td>74 58</td>
<td>cmpq $0x0,0x2c1220(%rip)</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>434878:</td>
<td>48 83 3d 20 12 2c 00</td>
<td>je 4348d0 &lt;rb_newobj+0x70&gt;</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>43487f:</td>
<td>00</td>
<td>mov 0x2c1217(%rip),%rax</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>434880:</td>
<td>74 4e</td>
<td>mov 0x8(%rax),%rdx</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>434882:</td>
<td>48 8b 05 17 12 2c 00</td>
<td>mov %rdx,0x2c120c(%rip)</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>434889:</td>
<td>48 8b 50 08</td>
<td>movq $0x0,0x2c120c(%rip)</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>43488d:</td>
<td>48 89 15 0c 12 2c 00</td>
<td>movq $0x0,%rax</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>434894:</td>
<td>48 c7 00 00 00 00 00</td>
<td>movq $0x0,0x8(%rax)</td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>43489b:</td>
<td>48 c7 00 00 00 00 00</td>
<td></td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
<tr>
<td>4348a2:</td>
<td>00</td>
<td></td>
<td># 6f5a0 &lt;freelist&gt;</td>
</tr>
</tbody>
</table>
### readelf

```bash
% readelf -a /usr/bin/ruby

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>.dynstr</td>
<td>STRTAB</td>
<td>000000000000003815</td>
<td>0000000000000000</td>
<td>A 0 0 1</td>
</tr>
<tr>
<td>7</td>
<td>.gnu.version</td>
<td>VERSYM</td>
<td>0000000000000086e</td>
<td>0000000000000002</td>
<td>A 5 0 2</td>
</tr>
<tr>
<td>8</td>
<td>.gnu.version_r</td>
<td>VERNEED</td>
<td>00000000000000c0</td>
<td>0000000000000000</td>
<td>A 6 5 8</td>
</tr>
<tr>
<td>9</td>
<td>.rela.dyn</td>
<td>RELA</td>
<td>0000000000000078</td>
<td>0000000000000000</td>
<td>A 5 0 8</td>
</tr>
<tr>
<td>10</td>
<td>.rela.plt</td>
<td>RELA</td>
<td>0000000000001248</td>
<td>0000000000000000</td>
<td>A 5 12 8</td>
</tr>
<tr>
<td>11</td>
<td>.init</td>
<td>PROGBITS</td>
<td>0000000000000018</td>
<td>0000000000000000</td>
<td>AX 0 0 4</td>
</tr>
<tr>
<td>12</td>
<td>.plt</td>
<td>PROGBITS</td>
<td>000000000000c40</td>
<td>0000000000000010</td>
<td>AX 0 0 4</td>
</tr>
<tr>
<td>13</td>
<td>.text</td>
<td>PROGBITS</td>
<td>00000000000096988</td>
<td>0000000000000000</td>
<td>AX 0 0 16</td>
</tr>
</tbody>
</table>
```

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 0x0000000001000000
    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
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)

```
412d16:   e8 c1 36 02 00          callq  4363dc # <a_function>
412d1b:  .....
```

address of this instruction
call opcode
32bit displacement to the target function from the next instruction.
anatomy of a call

(objdump output)

\[ \text{callq } 4363dc \# <a\_function> \]

(x86 is little endian)

\[ 412d16: \text{ e8 cl 36 02 00 } \]

\[ 412d1b: \text{ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_}\]

\[ 412d1b: \text{ } + \text{ } 000236c1 \text{ } = \text{ } 4363dc \]
Hook a_function

Overwrite the displacement so that all calls to a_function actually call a different function instead.

It may look like this:

```c
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?.....
BF CAREFUL
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:
  - [rip + 32bit 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)

callq 0x7fffff7afd6e0 <rb_newobj@plt>

0x7fffff7afd6e0 <rb_newobj@plt>:     jmpq *0x2c43b2(%rip)
0x7fffff7afd6e6 <rb_newobj@plt+6>:   pushq $0x4e
0x7fffff7afd6eb <rb_newobj@plt+11>:  jmpq 0x7fffff7afd1f0

# 0x7fffff7dc1a98

.got.plt entry

0x7fffff7afd6e6
How runtime dynamic linking works (elf)

Initially, the .got.plt entry contains the address of the instruction after the jmp.

```
callq 0x7fffff7afd6e0 <rb_newobj@plt>
0x7fffff7afd6e0 <rb_newobj@plt>: jmpq *0x2c43b2(%rip)
0x7fffff7afd6e6 <rb_newobj@plt+6>: pushq $0x4e
0x7fffff7afd6eb <rb_newobj@plt+11>: jmpq 0x7fffff7afd1f0
```
How runtime dynamic linking works (elf)

An ID is stored and the rtld is invoked.
How runtime dynamic linking works (elf)

```
callq 0x7ffffff7afd6e0 <rb_newobj@plt>
```

```
0x7ffffff7afd6e0 <rb_newobj@plt>:     jmpq  *0x2c43b2(%rip)
0x7ffffff7afd6e6 <rb_newobj@plt+6>:   pushq  $0x4e
0x7ffffff7afd6eb <rb_newobj@plt+11>:  jmpq  0x7ffffff7afd1f0
```

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

```
# 0x7fffffff7dc1a98
```

0x7fffffff7b34ac0
How runtime dynamic linking works (elf)

```
callq 0x7fffffff7afd6e0 <rb_newobj@plt>
```

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

calls to the PLT entry jump immediately to rb_newobj now that .got.plt is filled in.

```
0x7fffffff7afd6e0 <rb_newobj@plt>:    jmpq *0x2c43b2(%rip)
0x7fffffff7afd6e6 <rb_newobj@plt+6>:  pushq $0x4e
0x7fffffff7afd6eb <rb_newobj@plt+11>: jmpq 0x7fffffff7afd1f0
```
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

```
callq 0x7ffffff7afd6e0 <rb_newobj@plt>
```

```
0x7ffffff7afd6e0 <rb_newobj@plt>:     jmpq *0x2c43b2(%rip)           # 0x7ffffff7dc1a98
0x7ffffff7afd6e6 <rb_newobj@plt+6>:  pushq $0x4e
0x7ffffff7afd6eb <rb_newobj@plt+11>: jmpq 0x7ffffff7afd1f0
```

VALUE `other_function()`
```
{
  new_obj = rb_newobj();
  /* do something with new_obj */
  return new_obj;
}
```

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

NO, it isn’t. `other_function()` lives in it’s 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.
what else is left?

inline functions.
add_freelist

- Can’t hook because add_freelist is *inline*:

  ```c
  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:

```c
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.
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:

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

The jmp instruction and its offset are 5 bytes wide. Can’t grow or shrink the binary, so insert 2 one byte NOPs.
this instruction updates the freelist and comes from add_freelist:

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

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
push %rdi
mov  -0x3f8eaa77(%rip),%rdi  # save %rax incase the handler destroys it
push %rbx
push %rbp
mov  %rsp,%rbp  # save %rdi, we need it to pass arg 1
and  $0xffffffffffffffff,%rsp
mov  $0x7ffffff6a479b4,%rbx
movq  *%rbx  # mov top of freelist to rdi (arg 1 to handler)
callq  *%rbx
leaveq
pop  %rbx
pop  %rdi
pop  %rax
jmpq  0x437a1f <gc_sweep+1096>  # call handler via %rbx
                # mov the handler address into %rbx
                # call handler via %rbx
                # restore rbx
                # restore rdi
                # restore rax
                # continue execution
shortened assembly stub

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

```
mov    %rbx,-0x3f8ea6f(%rip)
push   %rax
push   %rdi
mov    -0x3f8ea77(%rip),%rdi
push   %rbx
push   %rbp
mov    %rsp,%rbp
and    0xfffffffffffffff0,%rsp
mov    $0x7fff6a479b4,%rbx
    # recreate overwritten instruction
    # save %rax incase the handler destroys it
    # save %rdi, we need it to pass arg 1
    # mov top of freelist to rdi (arg 1 to handler)
    # save rbx
    # save rbp
    # set base pointer to current stack pointer
    # align stack to conform with 64bit ABI
    # mov the handler address into %rbx
    callq *%rbx
    # call handler via %rbx
leaveq
pop    %rbx
pop    %rdi
pop    %rax
jmpq   0x437a1f <gc_sweep+1096>    # mov rbp, rsp; pop rbp
    # restore rbx
    # restore rdi
    # restore rax
    # continue execution
```
and it actually works.

gem install memprof

http://github.com/ice799/memprof
Sample Output

```ruby
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

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

def self.chars
  @chars ||= simple_chars + ['-', '.']
end

def self.ext_pairs
  @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
memprof.com
a web-based heap visualizer and leak analyzer
config.middleware.use(Memprof::Tracer)
{
  "time": 4.3442, ← total time for request
  "rails":
    "controller": "test",
    "action": "index"
},

  "request": {
    "REQUEST_PATH": "/test,
    "REQUEST_METHOD": "GET"
  },
}
config.middleware.use(Memprof::Tracer)

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

"gc": {
  "calls": 8,
  "time": 2.04925
},

3 mysql queries
8 calls to GC
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,      // lots of strings
        "string": 1334637, // lots of arrays
        "array": 609313,   // regexp matches
        "hash": 3676,
        "match": 70211
    }
}
```
Verify requests for sensitive information and be suspicious of anything unusual.
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!! I!

look a bullshit benchmark!
it makes ruby faster!!11!!

#NORMAL RUBY!!!!11!!

[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!!1!1!

# fast0r RUBY!!!!11111
[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
injectsso

- 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 inject.so, 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 inject.so, 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](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](http://github.com/ice799/injectso64)

• tested it on 64bit ubuntu VM, works.
injectso + evil-binary-patching-dso
how to defend against it

- NX bit
- strip debug information
- statically link everything
- put all .text code in ROM
- don’t load DSOs at runtime
- disable ptrace
- check /proc/<pid>/maps

- call mprotect
- mostly prebuilt binaries
- extremely large binaries
- maybe?
- no plugins, though
- no gdb/strace.
- word.
my future research: exploring alternative binary formats.
alignment
calling convention
object file and library formats
questions?

joe damato

@joedamato
timetobleed.com

http://timetobleed.com/rewrite-your-ruby-vm-at-runtime-to-hot-patch-useful-features/
http://timetobleed.com/dynamic-linking-elf-vs-mach-o/
“Interesting Behavior of OS X”

- Steven Edwards (<winehacker@gmail.com>)
- November 29, 2007
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