radare2
//rooted

---

pancake
pancake@nopcode.org
nibble
nibble.ds@gmail.com

Overview

radare2 is a rewrite of radare (r1) focusing on:

- API (refactor, clean)
- Portability (osx,linux,bsd,w32)
- Modularity (~40 modules)
- Scripting and bindings (valaswig)

Status of 0.4

- Aiming to be as compatible as possible with r1
- Some command and concepts has been redefined
- Runtime >10x faster
- Smart and cleaner code (40% of LOCs)
- Refactoring never ends **-:)**

radare2 // 0.4 release

Download sources:

`http://www.radare.org/get/radare2-0.4.tar.gz`

Debian packages:

`http://www.radare.org/get/r2deb`

Chiptune session: (Thanks neuroflip!)

`http://www.radare.org/get/r2-0.4.mp3`

6 months from 0.3 and ~300 commits

Language bindings

* C is fun, but people love to loose CPU cycles..
  - Automatic bindings generated by valaswig
  - Vala and Genie by default
  - Python, Perl, Lua and Ruby (more will come)
  - Access to full internal API
  - Binded code can use native instances and viceversa
  - Transparent access to generics, collections, iterators, classes, enums, structures, arrays, basic types..
* Valaswig is a .vapi to .i translator

```
$ hg clone http://hg.youterm.com/valaswig
$ wget http://radare.org/get/valaswig-0.1.tar.gz
```

Scripting demo

```
$ python
>>> import libr
>>> core = libr.RCore()
>>> core.loadlibs()
>>> file = core.file_open("dbg:///bin/ls", False)
>>> core.dbg.use("native")
>>> core.cmd0("dp=%d"%file.fd)
```

```
$ lua
> require "r_bin"
> file = arg[1] or "/bin/ls"
> b = r_bin.RBin ()
> b:load (file, "")
> baddr = b:get_baddr ()
> s = b:get_sections ()
> for i=0,s:size()-1 do
>   print (string.format ('0x%08x va=0x%08x size=%05i %s',
      s[i].offset, baddr+s[i].rva, s[i].size, s[i].name))
> end
```

Scripting demo (2)

```
$ ruby <<EOF
require 'libr'
core = Libr::RCore.new
core.file_open("/bin/ls", 0);
print core.cmd_str("pd 20");
EOF
```

```
$ perl <<EOF
require "r2/r_asm.pm";
sub disasm {
  my ($a, $arch, $op) = @_;
  $a->use ($arch);
  my $code = $a->massemble ($op);
  if (defined($code)) {
    my $buf = r_asmc::RAsmCode_buf_hex_get ($code);
    print "$op | $arch | $buf\n";
  }
}
my $a = new r_asm::RAsm();
disasm ($a, 'x86.olly', 'mov eax, 33');
disasm ($a, 'java', 'bipush 33');
EOF
```

r2w

Aims to be a web frontend for radare2

- Written in python (no dependencies)
- jQuery and CSS hardly simplifies the design of the gui
- At the moment it is just a PoC
- Assembler/disassembler, debugger, hasher demos

```
$ python main.py
Process with PID 20951 started...
URL=http://127.0.0.1:8080/
ROOT=/home/pancake/prg/r2w/www
```

```
$ surf http://127.0.0.1:8080
 ...
```

(demo)

Searching bytes

* One of the very basic features of r1 has been rewritten in order to offer a clean API to search keywords with binary masks, patterns, regular expressions and strings.

```
/* Genie example search patterns */
uses
    Radare.RSearch

init
    var s = new RSearch (Mode.KEYWORD)
    s.kw_add ("lib", "")
    s.begin ()

    var str = "foo is pure lib"
    s.update_i (0, str, str.len ())
```

Debugging

* Several APIs affected: (debug, reg, bp, io)
  - No os/arch specific stuff
  - Same code works on w32, OSX, BSD and GNU/Linux
  - Basics on x86-32/64, PowerPC, MIPS and ARM
  - Not all functionalities of r1 implemented (work in progress)
  - Debugger is no longer an IO backend
  - Program transplant between different backends
  - Some basics on backtrace, process childs and threads
  - Memory management (user/system memory maps)
  - Only software breakpoints atm
  - Traptracing, and software stepping implemented

Demo

Sample debugging session

```
$ r2 -V
radare2 0.4 @ linux-lil-x86
```

```
$ r2 -d ls
[0x080498a0]> ds   # step one instruction
[0x080498a0]> dsl  # step source line
[0x080498a0]> dr=  # display registers
 eip 0xb7883812   oeax 0xffffffff    eax 0xbfd89800
 ecx 0x00000000    edx 0x00000000    esp 0xbfd89800
 esi 0x00000000    edi 0x00000000 eflags 0x00000292
[0x080498a0]> dcu sym.main  # continue until sym.main
[0x080498a0]> dpt  # display process threads
 6064 s (current)
 6064 s thread_0
[0x080498a0]> dbt  # display backtrace
```

NOTE: Debugger commands no longer relay on IO backend '!'

r2rc the relocatable code compiler

* Simple and minimal compiler for x86 32/64
  - arm and powerpc support will follow
  - C-like syntax, with low-level hints
  - Allows to generate assembly code ready to be injected
  - Used as interface for native and crossplatform injection
* Accessible thru shell and API

```
# r_sys_cmd_str -> r_asm_massemble -> r_debug_inject
$ r2rc main.r > main.asm
$ rasm2 -f main.asm > main.hex
$ r2 -d ls
[0x08048594]> wF main.hex @ eip  # write hexpairs
[0x08048594]> dc                 # continue execution
```

r2rc code example

```
main@global(128) {
        .var80 = "argc = %d\n";                         # arguments
        printf (.var80, .arg0);
        .var80 = "0x%08x : argv[%02d] = %s\n";
        .var0 = 0;
        .var4 = *.arg1;
        while (.var0 <= .arg0) {
                printf (.var80, .var4, .var0, .var4);
                .var0 += 1;                             # increment counter
                .arg1 += 4;                             # increment pointer
                .var4 = *.arg1;                         # get next argument
        }
        .var80 = "0x%08x : envp[%02d] = %s\n";          # environ
        .var0 = 0;
        .var4 = *.arg2;
        { printf (.var80, .var4, .var0, .var4);
                .var0 += 1;                             # increment counter
                .arg2 += 4;                             # increment pointer
                .var4 = *.arg2;                         # get next environ
        } while (.var4);
        0;
}
```

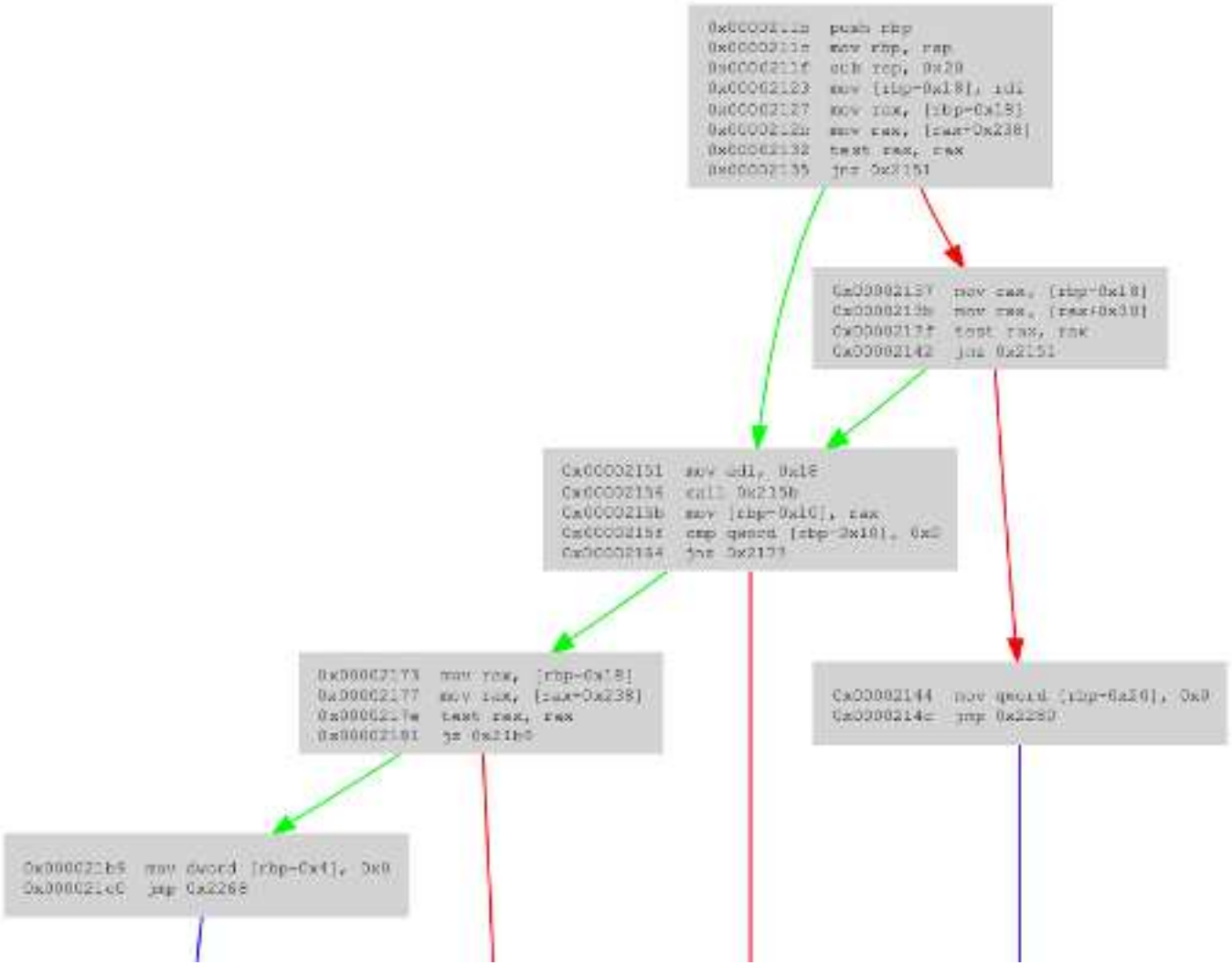
RAnal

* Data and code analysis
* Analyzed data is accessible from opcode level to function level (opcode, BB, functions, vars, xrefs...)
* Combine data is very quickly
  Eg.: Filter bb by function, graph bb hierarchy, analyze references...
* Graph output in graphviz format (dot)

Demo

* Code & Data analysis
* Graph generation
  - Full
  - Partial
* Source code graph

RAnal


0x00002...a push rbp
0x00002...b mov rbp, rsp
0x00002...f sub rsp, 0x20
0x00002123 mov [rbp-0x18], rdi
0x00002127 mov rax, [rbp-0x18]
0x0000212b mov rax, [rax+0x238]
0x00002132 test rax, rax
0x00002135 jns 0x2151
0x00002137 mov rax, [rbp-0x18]
0x0000213b mov rax, [rax+0x30]
0x0000213f test rax, rax
0x00002142 jns 0x2151
0x00002151 mov edi, 0x18
0x00002156 call 0x215b
0x0000215b mov [rbp-0x10], rax
0x0000215f cmp qword [rbp-0x10], 0x0
0x00002164 jns 0x2177
0x00002173 mov rax, [rbp-0x18]
0x00002177 mov rax, [rax+0x238]
0x0000217e test rax, rax
0x00002181 js 0x21b0
0x00002144 mov qword [rbp-0x10], 0x0
0x0000214c jmp 0x2280
0x000021b5 mov dword [rbp-0x4], 0x0
0x000021c0 jmp 0x2268

RBin

* Header analysis
* Supports:
  ELF32, ELF64, PE32, PE32+, MACH-O,
  MACH-O64, CLASS...
* Format-Agnostic API
* All sub-libs have been written from scratch
* All sub-libs offer a complete API for working
  with specific formats
* Keeps reversing (and minimalism) in mind

RBin

* Read support
  - Imports
  - Symbols (Exports)
  - Sections
  - Linked libraries
  - Strings
  - Binary info
    object type
    endianness
    debug data/stripped
    static/dynamic...

RBin

* Write support (*)
  - Add/Remove/Resize {sections, imports, symbols}
  - Edit header fields
* Metadata support (*)

(*) = Work in progress

Demo

* Format-agnostic API

```
$ python imports.py ls
$ python imports.py user32.dll
$ python imports.py osx-ls.1
```

```
$ cat imports.py
#!/usr/bin/python
from libr import *
import sys

if (len (sys.argv) == 2):
        file = sys.argv[1]
else:
        file = "/bin/ls"
        b = RBin ()
        b.load(file, None)
        baddr= b.get_baddr()
        print '-> Imports'
        for i in b.get_imports ():
                print 'offset=0x%08x va=0x%08x %s' % (
                                i.offset, baddr+i.rva, i.name)
```

RAsm

* (Dis)Assembly library
* Supports x86, x86-64, PPC, MIPS, ARM,
  SPARC, m68k, psosvm...
* Uses:
  - (Dis)Assembly backed
  - Compile inline code in order to be injected
  - Assembly backend of rcc
* All parameters (arch, wordsize...) can be modified
  in runtine, so generic injection are easy to implement

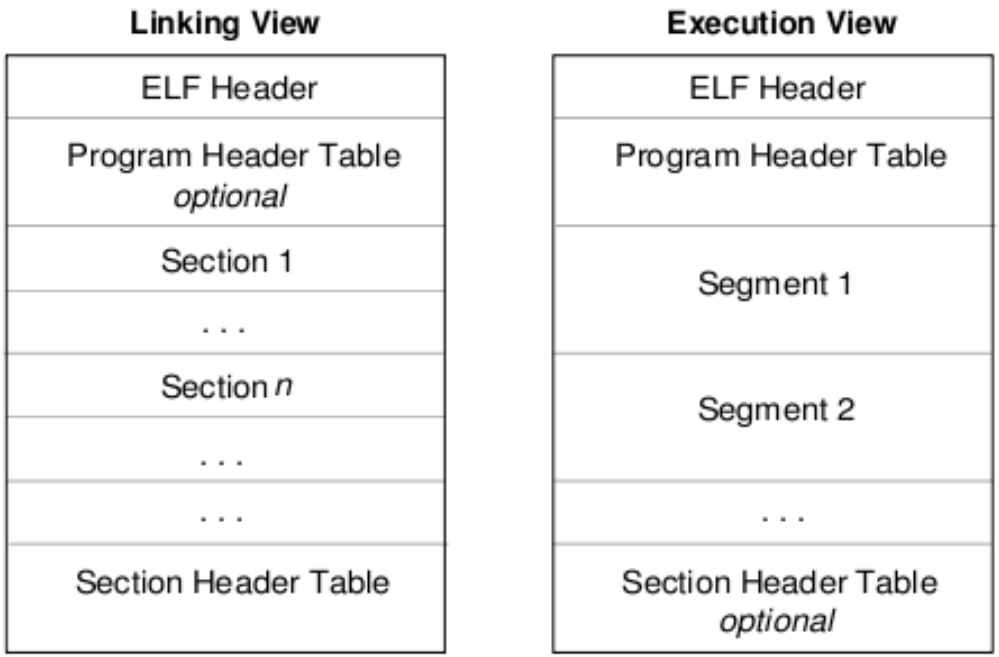
Demo

* Interactive disassembler

```
$ ./widget-asm
```

Demo

* XorPacker
  - ELF structure



Demo (XorPacker)

```
$ rabin2 -S test | cut -d ’ ’ -f 2,6-7
[...]
address=0x08048340 privileges=-r-x name=.text
address=0x080484fc privileges=-r-x name=.fini
address=0x08048518 privileges=-r-- name=.rodata
[...]
```

Demo (XorPacker)

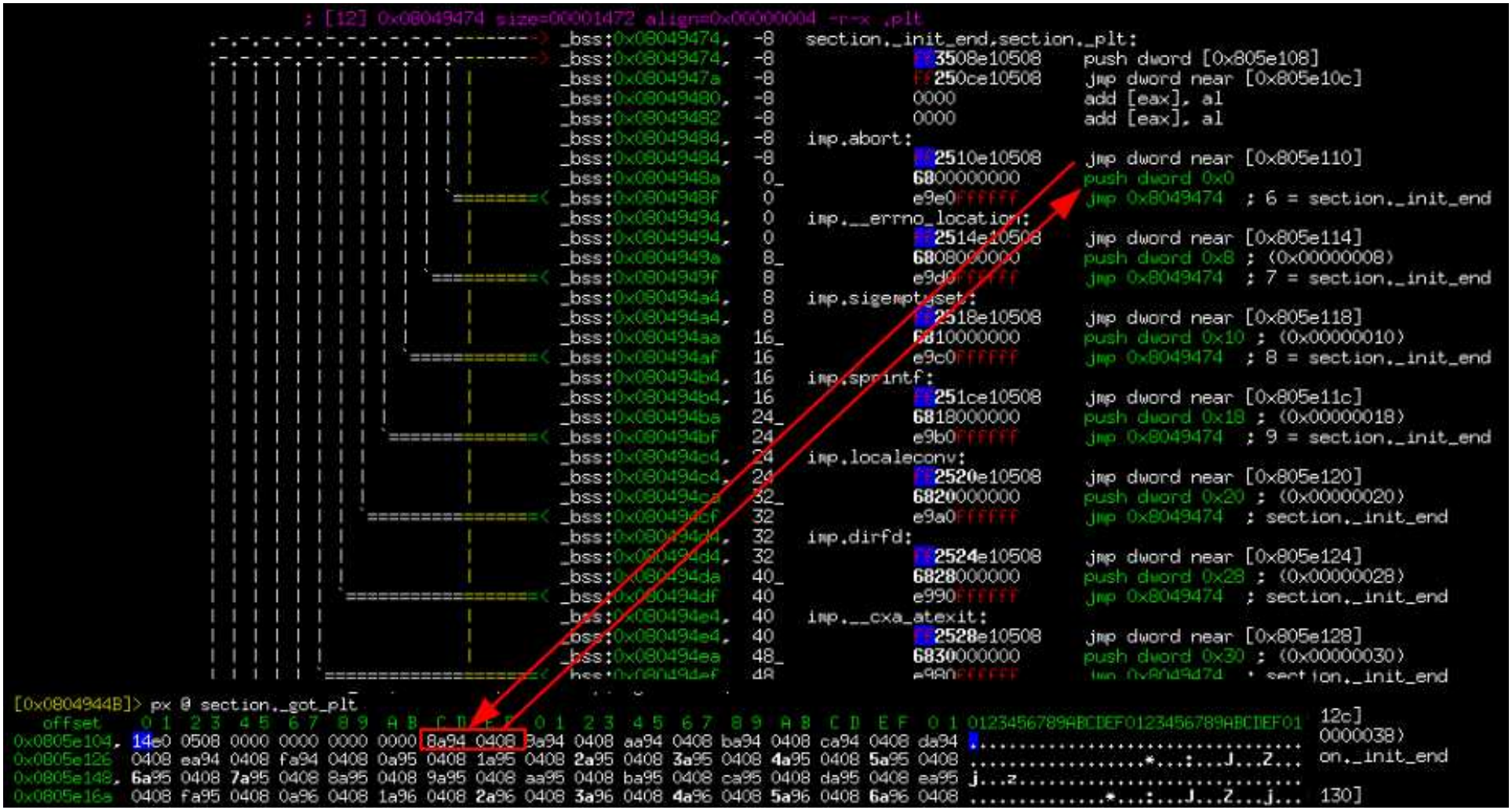
- Xor from .text to .rodata
- Execution flow
  Entrypoint -> Init -> main
- Analyze entrypoint
  Get init address
- Overwrite init with the packer payload
  Change page permissions with mprotect
  Xor from .text to .data (take care of payload code)

Demo (XorPacker)

```
$ rabin2 -z test | grep "section=.rodata"
  | cut -d ' ' -f 1,5-6
address=0x08048520 section=.rodata string=passw0rd
address=0x08048529 section=.rodata string=ROOTED!
address=0x08048531 section=.rodata string=Ooops
$ rabin2 -z a.out | grep "section=.rodata"
  | cut -d ' ' -f 1,5-6
address=0x08048518 section=.rodata string=jiiihiki
address=0x08048528 section=.rodata string=i;&&=,-Hi&
$ ./a.out foo
Ooops
$ ./a.out passw0rd
ROOTED!
```

Demo

* ITrace


; [12] 0x08049474 size=00001472 align=0x00000004 -r-x .plt
section._init_end,section._plt:
push dword [0x805e108]
jmp dword near [0x805e10c]
imp.abort:
jmp dword near [0x805e110]
push dword 0x0
jmp 0x8049474 ; 6 = section._init_end
imp.__errno_location:
jmp dword near [0x805e114]
push dword 0x8 ; (0x00000008)
jmp 0x8049474 ; 7 = section._init_end
imp.sigemptyset:
jmp dword near [0x805e118]
push dword 0x10 ; (0x00000010)
jmp 0x8049474 ; 8 = section._init_end
imp.sprintf:
jmp dword near [0x805e11c]
push dword 0x18 ; (0x00000018)
jmp 0x8049474 ; 9 = section._init_end
imp.localeconv:
jmp dword near [0x805e120]
push dword 0x20 ; (0x00000020)
jmp 0x8049474 ; section._init_end
imp.dirfd:
jmp dword near [0x805e124]
push dword 0x28 ; (0x00000028)
jmp 0x8049474 ; section._init_end
imp.__cxa_atexit:
jmp dword near [0x805e128]
push dword 0x30 ; (0x00000030)
jmp 0x8049474 ; section._init_end
[0x0804944B]> px @ section._got_plt
offset
0x0805e104 14e0 0508 0000 0000 0000 0000 8a94 0408 3a94 0408 aa94 0408 ba94 0408 ca94 0408 da94
0x0805e126 0408 ea94 0408 fa94 0408 0a95 0408 1a95 0408 2a95 0408 3a95 0408 4a95 0408 5a95 0408
0x0805e148 6a95 0408 7a95 0408 8a95 0408 9a95 0408 aa95 0408 ba95 0408 ca95 0408 da95 0408 ea95
0x0805e16a 0408 fa95 0408 0a96 0408 1a96 0408 2a96 0408 3a96 0408 4a96 0408 5a96 0408 6a96 0408

Demo (ITrace)

- Edit all plt entries but hijacked import
- Analyze entrypoin
  Get init address
- Write Hook code into init
  Push interesting parameters
  Call hijacked import
  Fix stack
  jump to the first PLT entry
- LD_PRELOAD library containing hijacked import

Demo (ITrace)

```
$ LD_PRELOAD=./preload.so ./a.out
Fake sleep call from import 0x8 @ 0x804830c
Fake sleep call from import 0x18 @ 0x804832c
ROOTED!
Fake sleep call from import 0x18 @ 0x804832c
ROOTED!
Fake sleep call from import 0x18 @ 0x804832c
ROOTED!
^C
```

So...

EOF

- Ideas, questions?

Thanks for listening!