

# frida-boot

a binary instrumentation workshop, using Frida, for beginners

```
let whoami = (w) ⇒ {  
  w.name      = Leon Jacobs;  
  w.hacks_at  = {  
    company: SensePost,  
    part_of: Orange Cyberdefense  
  };  
  w.twitter   = @leonjza;  
}
```

```
let disclaimer = (me) ⇒ {  
    me.noob      = True;  
    me.learning  = True;  
}
```

```
let objectives = () => {  
  return [  
    hands on frida 101,  
    explore some internals,  
    !mobile (but relevant),  
    build tools & explore  
  ];  
}
```

```
let applies_to = (p)  $\Rightarrow$  {  
    return [windows, macos, linux,  
            android, ios].includes(p)  
}
```

```
while(workshop) {  
    follow_by_doing();  
}
```

# toc

- 0x0 environment setup
- 0x1 hooking with LD\_PRELOAD
- 0x2 hooking with Frida
- 0x3 frida tools, agents and modes

# overview - chapter 0x1

## 0x1 hooking with LD\_PRELOAD

- 0x1 setup a sample C program
- 0x2 LD\_PRELOAD basics
- 0x3 building a shared library
- 0x4 hooking with a shared library
- 0x5 hooking internals with gdb



# overview - chapter 0x2

## 0x2 hooking with frida

- 0x1 basic components
- 0x2 porting the LD\_PRELOAD hook
- 0x3 hooking arguments & return values
- 0x4 repurposing existing code

overview - chapter 0x3

0x3 frida tools, agents and modes

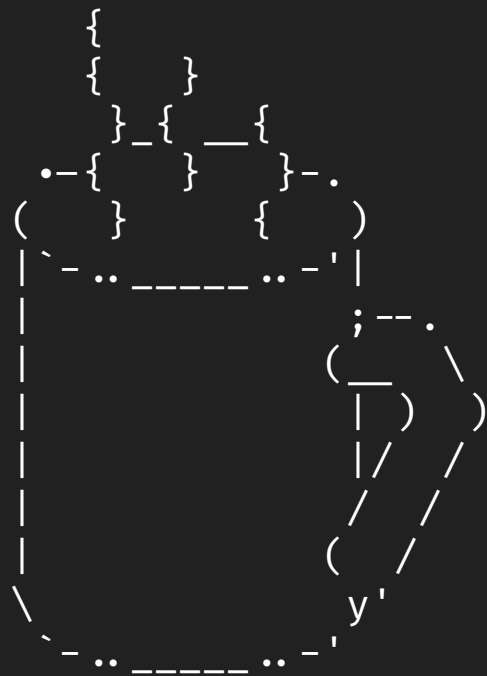
0x1 building python tools

0x2 channels vs frida rpc

0x3 typescript and documentation

0x4 execution modes

0xdeadbeef bonus chapter



environment setup

- everything you need is pre-installed in the `docker` container
- you can pull the container from `docker hub` or build it yourself (i suggest you just pull)

```
git clone github.com/leonjza/frida-boot
```

a helper script to **pull**, **run** and get new **shells** in the container can be found in the **frida-boot** repository

```
./docker.sh pull
```

```
./docker.sh run
```

```
./docker.sh shell # container has tmux
```

`code/` storage on your computer

running the container created a `code/` directory. if the container dies, your code will be safe



## shells in the container

- you may need multiple sessions
- `tmux` is installed in the container
- otherwise, run `docker exec -it frida-boot /bin/bash` in a new terminal (or use the shell script)

chapter 0x1

part 0x1

we'll start small. copy `pew.c` to your `code/` folder.

```
~$ cp software/pew.c code/
```

```
~$ cd code/
```

```
~/code$ ls
```

```
pew.c
```

reduced `pew.c` source code

```
int main() {  
  
    while(1) {  
        d = rand_range(1, 5);  
        sleep(d);  
    }  
}
```

now, compile `pew.c` with `gcc`

```
~/code$ gcc pew.c -o pew
```

```
~/code$ ls
```

```
pew    pew.c
```

next, run pew

```
~/code$ ./pew
```

```
[+] Starting up!
```

```
[+] Sleeping for 5 seconds
```

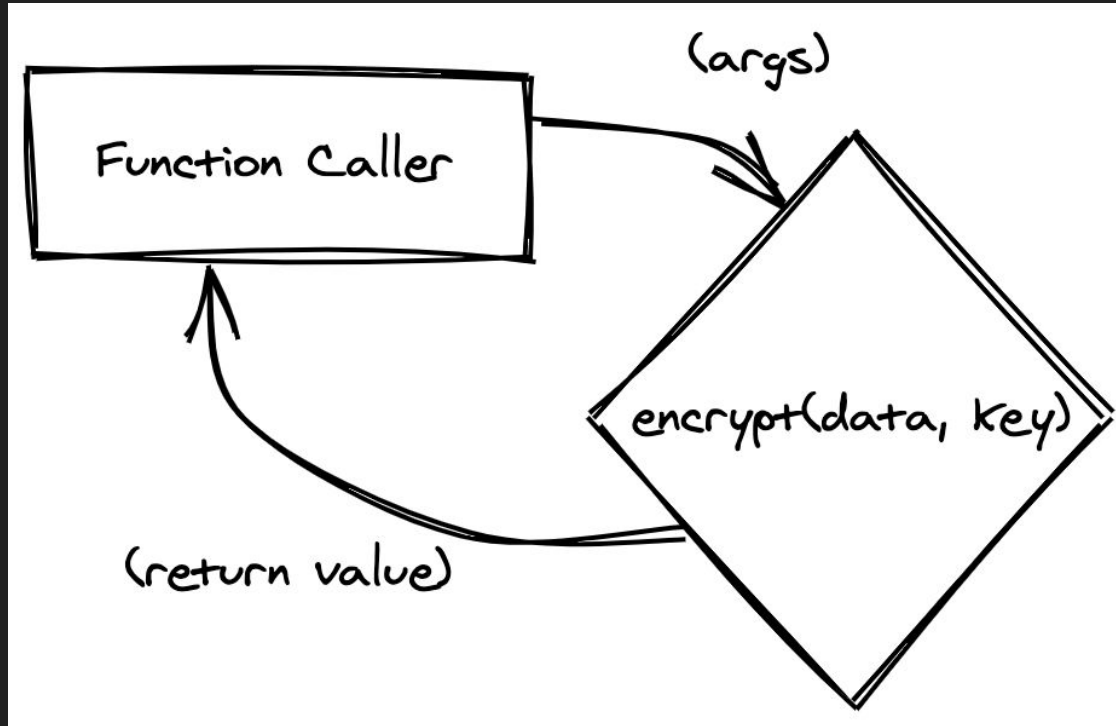
```
[+] Sleeping for 3 seconds
```

```
^C
```

# hooking in general

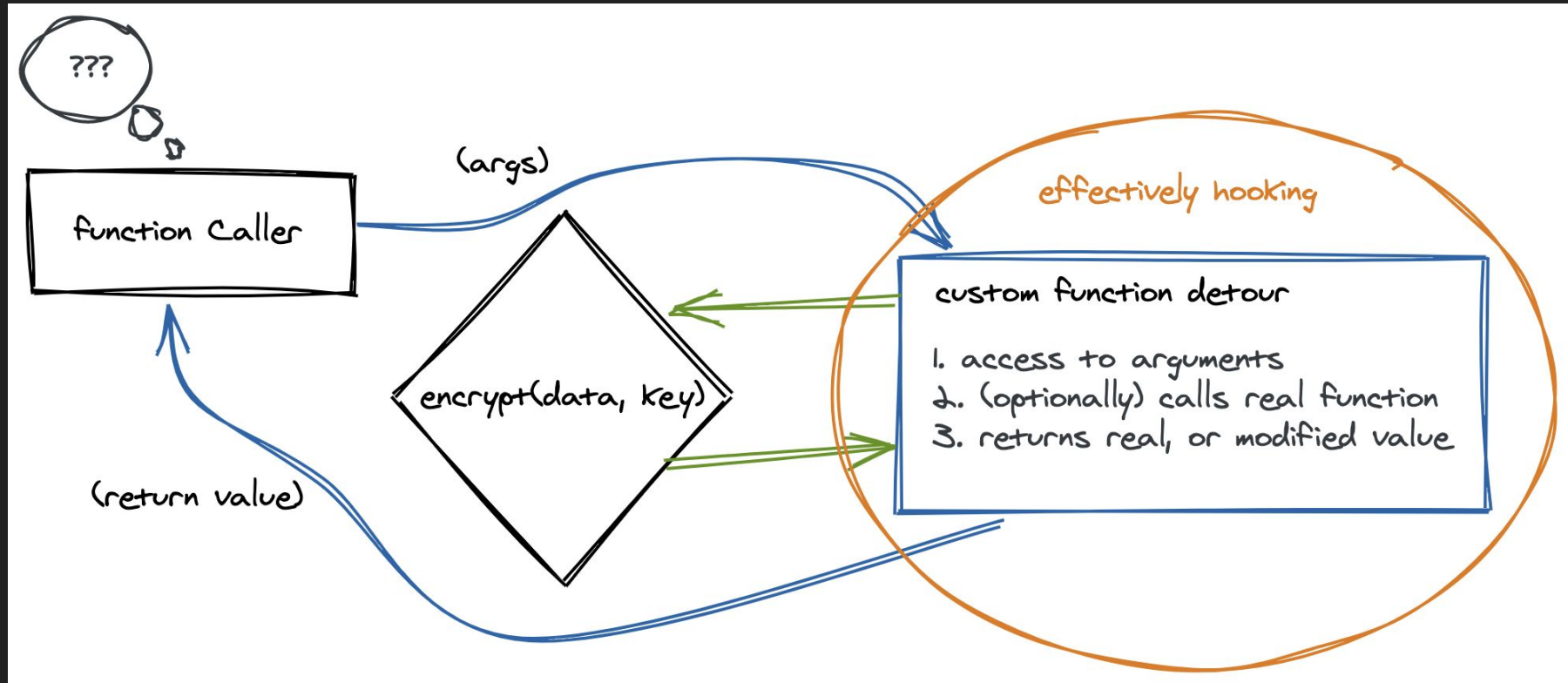
```
void encrypt(int *data, char *key);
```

```
char * getKey();
```





# a hooked function call



## LD\_PRELOAD (man 8 ld.so)

A list of **additional**, **user**-specified, ELF **shared objects** to be loaded **before** all others. This feature can be used to selectively **override functions** in other shared objects.

# shared libraries in `pew`.

```
~/code$ ldd pew  
    linux-vdso.so.1 (0x00007ffd5b1f5000)  
    libc.so.6 ⇒ /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8b4ae0e000)  
    /lib64/ld-linux-x86-64.so.2 (0x00007f8b4afdc000)
```

shared libraries in a static bin.

```
~/code$ gcc -static pew.c -o pew-static
```

```
~/code$ ldd pew-static
```

*not a dynamic executable*

our `pew` program makes use of some standard libc functions.

- `sleep()`
- `printf()`

## NAME

sleep - sleep for a specified number of seconds

## SYNOPSIS

```
#include <unistd.h>
```

```
unsigned int sleep(unsigned int seconds);
```

## DESCRIPTION

sleep() causes the calling thread to sleep either until the number of real-time seconds specified in seconds have elapsed or until a signal arrives which is not ignored.

real source code for libc `sleep()`  
... more than 100 LoC ...

<https://github.com/sgallagher/glibc/blob/master/sysdeps/unix/sysv/linux/sleep.c>

our own implementation for `sleep()` saved  
to `fake_sleep.c`

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

```
unsigned int sleep(unsigned int seconds) {  
    printf("[-] sleep goes brrr\n");  
    return 0;  
}
```



get the first step for `fake_sleep.c` into  
your code folder

```
~/code$ cp ../software/fake_sleep.c.1  
fake_sleep.c  
~/code$ ls  
fake_sleep.c  pew  pew.c
```

compile the `fake_sleep` shared library

```
~/code$ gcc -fPIC -shared fake_sleep.c  
-o fake_sleep.so
```

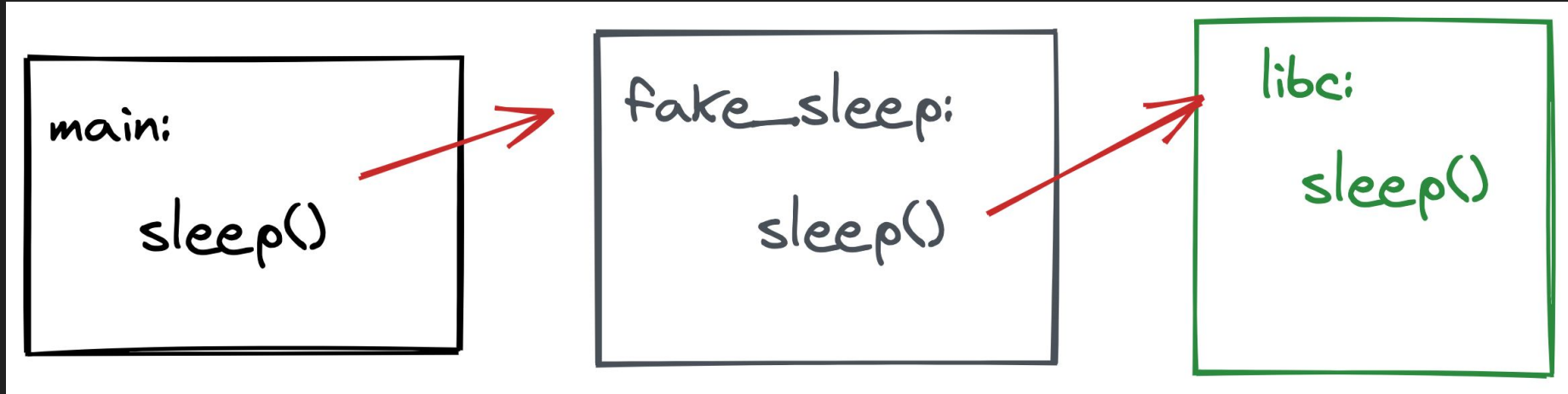
```
~/code$ ls
```

```
fake_sleep.c  fake_sleep.so  pew  pew.c
```

run pew loading our new shared library  
it's going to fly past, we not sleeping anymore...

```
~/code$ LD_PRELOAD=./fake_sleep.so ./pew
[+] Starting up!
[+] Sleeping for 1 seconds
[-] sleep goes brrr
[+] Sleeping for 5 seconds
[-] sleep goes brrr
^C
```

# building a `sleep()` “proxy”



# finding the real `sleep()`

DLSYM(3)      Linux Programmer's Manual      DLSYM(3)

## NAME

`dlsym`, `dlvsym` - obtain address of a symbol in a shared object or executable

## SYNOPSIS

```
#include <dlfcn.h>
```

```
void *dlsym(void *handle, const char *symbol);
```

get `fake_sleep.c.2` into your code folder  
now

```
~/code$ cp ../software/fake_sleep.c.2  
fake_sleep.c  
~/code$ ls  
fake_sleep.c  pew  pew.c
```

```
unsigned int sleep(unsigned int seconds) {  
    seconds = 1;  
  
    unsigned int (*original_sleep)(unsigned int);  
    original_sleep = dlsym(RTLD_NEXT, "sleep");  
  
    return (original_sleep)(seconds);  
}
```

let's compile a new `fake_sleep` to link  
`ld-linux` for `dlsym`

```
~/code$ gcc -fPIC -shared fake_sleep.c  
-o fake_sleep.so -ldl
```

```
~/code$ ls
```

```
fake_sleep.c  fake_sleep.so  pew  pew.c
```



```
~/code$ LD_PRELOAD=./fake_sleep.so ./pew
[+] Starting up!
[+] Sleeping for 4 seconds
[-] sleep goes brrr
[+] Sleeping for 1 seconds
[-] sleep goes brrr
[+] Sleeping for 5 seconds
[-] sleep goes brrr
^C
```

## recap - hooking with LD\_PRELOAD

- shared libraries can **override** functions.
- overriding functions lets you **spy** on and alter arguments and return values
- the real, **intended** function can be called in the function “proxy”

chapter 0x1

part 0x2

enumerate `pew` with `nm`, luke

```
~/code$ nm -D pew  
      U __libc_start_main  
      U printf  
      U puts  
      U rand  
      U sleep  
      U srand  
      U time
```

under the hood of LD\_PRELOAD using `gdb`

- `q` - `quit` `gdb`
- `r` - `run` the program
- `b` - `set` a breakpoint
- `s` - step `over` an instruction
- `si` - step `into` an instruction

under the hood of LD\_PRELOAD using `gdb`

<code>info func</code>	- show available <code>functions</code>
<code>info break</code>	- show available <code>breakpoints</code>
<code>del &lt;index&gt;</code>	- <code>delete</code> a breakpoint

start debugging `pew` with `gdb`

```
~/code$ gdb -q ./pew
```

```
frida-boot:~/code$ gdb -q ./pew
GEF for linux ready, type `gef' to start, `gef config' to configure
75 commands loaded for GDB 9.1 using Python engine 3.8
[*] 5 commands could not be loaded, run `gef missing` to know why.
Reading symbols from ./pew...
(No debugging symbols found in ./pew)
gef> |
```



get known **function** information in gdb

```
gef> info func
```

All defined functions:

Non-debugging symbols:

...

0x00000000000000001000	<b>_init</b>
0x00000000000000001070	<b>sleep@plt</b>
0x00000000000000001080	<b>rand@plt</b>

...

disassemble `main`, look for `call`'s

```
gef> disas main
```

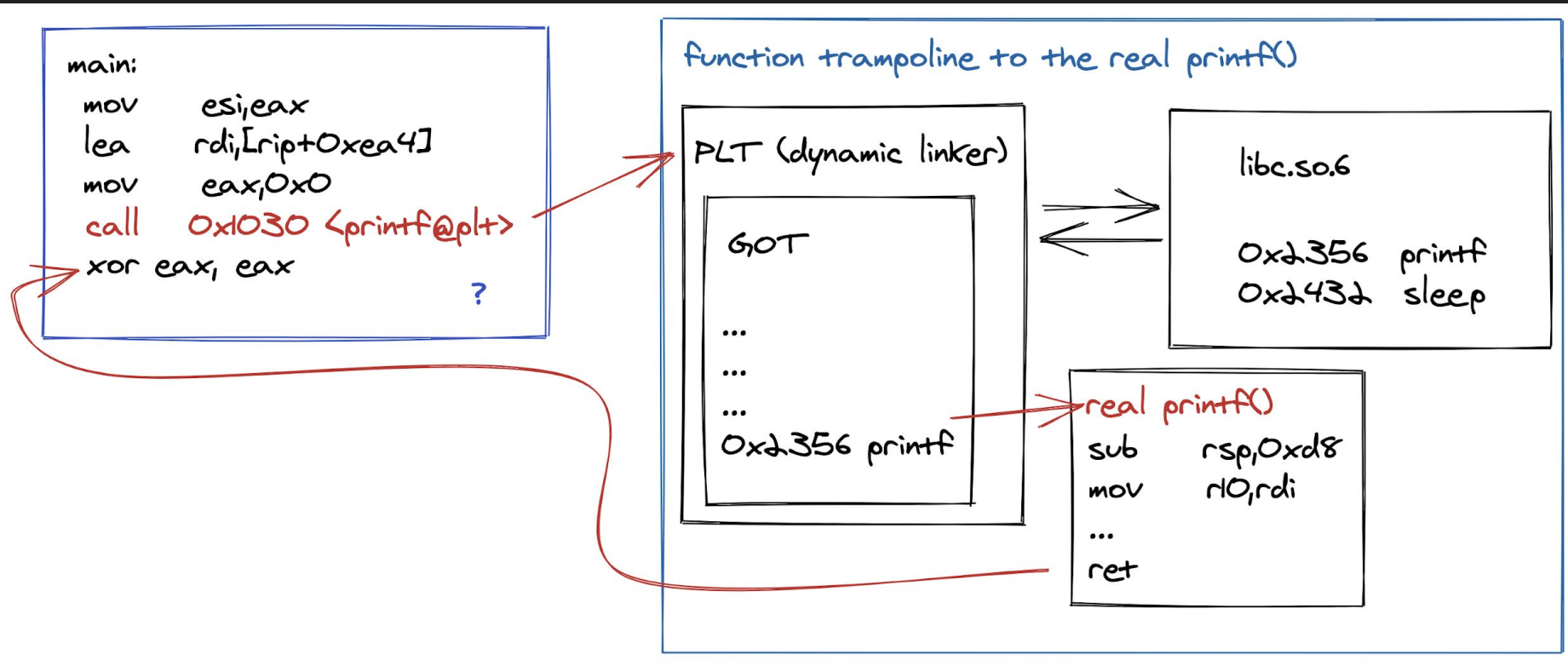
Dump of assembler code for function main:

```
[ ... ]
```

```
0x000000000000011f5 <+72>: call    0x1040 <printf@plt>
0x000000000000011fa <+77>: mov    eax,DWORD PTR [rbp-0x4]
0x000000000000011fd <+80>: mov    edi,eax
0x000000000000011ff <+82>: call    0x1070 <sleep@plt>
0x00000000000001204 <+87>: jmp    0x11d2 <main+37>
```

End of assembler dump.

# the `plt` && `got`



run `pew` in gdb, breaking on `*main`

gef> `b *main`

Breakpoint 1 at 0x11ad

gef> `info br`

Num	Type	Disp	Enb	Address
What				
1	breakpoint	keep	y	0x11ad <main>

gef> `r`

```
gef- r
Starting program: /root/code/pew
warning: Error disabling address space randomization: Operation not permitted
```

```
Breakpoint 1, 0x0000555e9e1a31ad in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
```

registers

```
$rax : 0x0000555e9e1a31ad → <main+0> push rbp
$rbx : 0x0
$rcx : 0x00007fafbc6d7718 → 0x00007fafbc6d9980 → 0x0000000000000000
$rdx : 0x00007ffc857f9488 → 0x00007ffc857f9ecd → "LANGUAGE=en_US:en"
$rsp : 0x00007ffc857f9398 → 0x00007fafbc541e0b → <__libc_start_main+235> mov edi, eax
$rbp : 0x0000555e9e1a3210 → <__libc_csu_init+0> push r15
$rsi : 0x00007ffc857f9478 → 0x00007ffc857f9ebe → "/root/code/pew"
$rdi : 0x1
$rip : 0x0000555e9e1a31ad → <main+0> push rbp
$r8 : 0x0
$r9 : 0x00007fafbc6f3530 → push rbp
$r10 : 0x0
$r11 : 0x0
$r12 : 0x0000555e9e1a30a0 → <_start+0> xor ebp, ebp
$r13 : 0x00007ffc857f9470 → 0x0000000000000001
$r14 : 0x0
$r15 : 0x0
```

```
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
```

```
$cs: 0x0033 $ss: 0x002b $ds: 0x0000 $es: 0x0000 $fs: 0x0000 $gs: 0x0000
```

stack

```
0x00007ffc857f9398 +0x0000: 0x00007fafbc541e0b → <__libc_start_main+235> mov edi, eax - $rsp
0x00007ffc857f93a0 +0x0008: 0x0000000000000000
0x00007ffc857f93a8 +0x0010: 0x00007ffc857f9478 → 0x00007ffc857f9ebe → "/root/code/pew"
0x00007ffc857f93b0 +0x0018: 0x0000000010004000
0x00007ffc857f93b8 +0x0020: 0x0000555e9e1a31ad → <main+0> push rbp
0x00007ffc857f93c0 +0x0028: 0x0000000000000000
0x00007ffc857f93c8 +0x0030: 0xd42f69660b9cf43a
0x00007ffc857f93d0 +0x0038: 0x0000555e9e1a30a0 → <_start+0> xor ebp, ebp
```

we only care about the **code** and **stack** regions for now

```
code:x86:64
0x555e9e1a31a9 <rand_range+36> add    eax, edx
0x555e9e1a31ab <rand_range+38> leave
0x555e9e1a31ac <rand_range+39> ret
→ 0x555e9e1a31ad <main+0>      push   rbp
0x555e9e1a31ae <main+1>      mov    rbp, rsp
0x555e9e1a31b1 <main+4>      sub    rsp, 0x10
0x555e9e1a31b5 <main+8>      lea    rdi, [rip+0xe48]      # 0x555e9e1a4004
0x555e9e1a31bc <main+15>     call   0x555e9e1a3030 <puts@plt>
0x555e9e1a31c1 <main+20>     mov    edi, 0x0

threads
[#0] Id 1, Name: "pew", stopped 0x555e9e1a31ad in main (), reason: BREAKPOINT

trace
[#0] 0x555e9e1a31ad → main()

gef> |
```

get your hands dirty debugging **pew**

- step an instruction with **si**. you should see the code section move on by one line.
- hit **enter** after issuing the **si** command. gdb will rerun the previous command when pressing **enter**.
- retrieve the context view again with the **context** command.

step instructions with `si` until after  
the call to `puts@plt` (entering it)

```
# code region
```

```
...
```

```
> 0x8c030 <puts@plt+0>    jmp     QWORD PTR [rip+0x2fe2] # 0x8f018 <puts@got.plt>
    0x8c036 <puts@plt+6>    push    0x0
    0x8c03b <puts@plt+11>   jmp     0x55702578c020
    0x8c040 <printf@plt+0>  jmp     QWORD PTR [rip+0x2fda] # 0x8f020 <printf@got.plt>
    0x8c046 <printf@plt+6>  push    0x1
    0x8c04b <printf@plt+11> jmp     0x55702578c020
```

```
...
```

```
# trace region
```

```
[#0] 0x55702578c030 → puts@plt()
[#1] 0x55702578c1c1 → main()
```



just **step** to the next instruction with **s**  
for now, taking note of the trace region

```
# code region
```

```
...
```

```
> 0x2d030 <puts+0>      push    r14
   0x2d032 <puts+2>      push    r13
   0x2d034 <puts+4>      push    r12
   0x2d036 <puts+6>      mov     r12, rdi
   0x2d039 <puts+9>      push    rbp
   0x2d03a <puts+10>     push    rbx
```

```
...
```

```
# trace region
```

```
[#0] 0x7f2298a2d030 → puts()
[#1] 0x55702578c1c1 → main()
```

notice how it's no  
longer **puts@plt** in  
the trace region

the `dynamic linker` resolved the real location to `puts` in `libc`.

these locations are recorded in the `global offset table`.

the next call to `puts` will `jmp` to the location in the `got` and not invoke the linker.

```
gef> got
```

```
GOT protection: Partial RelRO | GOT functions: 6
```

```
[0x55702578f018] puts@GLIBC_2.2.5 → 0x7f2298a2d030  
[0x55702578f020] printf@GLIBC_2.2.5 → 0x55702578c046  
[0x55702578f028] srand@GLIBC_2.2.5 → 0x55702578c056  
[0x55702578f030] time@GLIBC_2.2.5 → 0x55702578c066  
[0x55702578f038] sleep@GLIBC_2.2.5 → 0x55702578c076  
[0x55702578f040] rand@GLIBC_2.2.5 → 0x55702578c086
```

using the `got` as reference, we can ask gdb to tell us `where` a symbol is located

```
gef> info symbol 0x7f2298a2d030  
puts in section .text of /lib/x86_64-linux-gnu/libc.so.6
```

```
gef> info symbol 0x55702578c046  
printf@plt + 6 in section .plt of /root/code/pew
```

follow the lookup for `printf` a bit closer.

- disassemble the main function with `disas main`.
- find the address where the first call to `printf@plt` occurs.
- set a breakpoint on the address where this happens with `b *addr`.

continue execution with `c` until the new breakpoint is hit

...

```
0x8c1f0 <main+67>      mov     eax, 0x0
> 0x8c1f5 <main+72>      call    0x8c040 <printf@plt>
> 0x8c040 <printf@plt+0> jmp     QWORD PTR [rip+0x2fda] # 0x8f020 <printf@got.plt>
0x8c046 <printf@plt+6>  push    0x1
0x8c04b <printf@plt+11> jmp     0x8c020
0x8c050 <srand@plt+0>   jmp     QWORD PTR [rip+0x2fd2] # 0x8f028 <srand@got.plt>
0x8c056 <srand@plt+6>   push    0x2
0x8c05b <srand@plt+11>  jmp     0x8c020
```

...

now, just `si` a few times and watch the call trace grow

`# trace section`

```
[#0] 0x7efe8320fde1 → cmp eax, 0x30
[#1] 0x7efe831fe665 → test eax, eax
[#2] 0x7efe831feb3b → add rsp, 0x30
[#3] 0x7efe831ff3f1 → add rsp, 0x30
[#4] 0x7efe83203af3 → mov r8, rax
[#5] 0x7efe8320a44a → mov r11, rax
[#6] 0x55a63774e1fa → main()
```

generate a backtrace with `bt` to get an idea of where those stack frames are from.

```
gef> bt
#0  0x00007efe8320fde1 in ?? () from /lib64/ld-linux-x86-64.so.2
#1  0x00007efe831fe665 in ?? () from /lib64/ld-linux-x86-64.so.2
#2  0x00007efe831feb3b in ?? () from /lib64/ld-linux-x86-64.so.2
#3  0x00007efe831ff3f1 in ?? () from /lib64/ld-linux-x86-64.so.2
#4  0x00007efe83203af3 in ?? () from /lib64/ld-linux-x86-64.so.2
#5  0x00007efe8320a44a in ?? () from /lib64/ld-linux-x86-64.so.2
#6  0x000055a63774e1fa in main ()
```



knowing a bit about the dynamic linker  
should help in understanding how  
`LD_PRELOAD` behaves.

debugging `pew` with `LD_PRELOAD` set:

- start a fresh gdb session with `gdb -q ./pew`
- set a breakpoint on the main function with `b *main`
- set the `LD_PRELOAD` variable with `set environment LD_PRELOAD ./fake_sleep.so`
- run with `r`

confirm `fake_sleep.so` was loaded using  
`vmmmap`

gef> `vmmmap`

[ Legend: Code | Heap | Stack ]

Start	End	Offset	Perm	Path
-------	-----	--------	------	------

...

0x00007f121ec42000	0x00007f121ec43000	0x0000000000000000	r--	/root/code/ <code>fake_sleep.so</code>
0x00007f121ec43000	0x00007f121ec44000	0x0000000000000100	r-x	/root/code/ <code>fake_sleep.so</code>
0x00007f121ec44000	0x00007f121ec45000	0x0000000000000200	r--	/root/code/ <code>fake_sleep.so</code>
0x00007f121ec45000	0x00007f121ec46000	0x0000000000000200	r--	/root/code/ <code>fake_sleep.so</code>
0x00007f121ec46000	0x00007f121ec47000	0x0000000000000300	rw-	/root/code/ <code>fake_sleep.so</code>

...

`break` after the call to `sleep` so we can inspect the `got`

- disassemble the main function with `disas main`
- break on the instruction after the call to `sleep@plt` (should be a `jmp` call because of our infinite loop)
- continue the programs execution with `c`

check where the symbol for `sleep()` now points according to the `got`

```
gef> got
```

GOT protection: Partial RelRO | GOT functions: 6

```
[0x55d5c325d018] puts@GLIBC_2.2.5 → 0x7f121eaf1030
[0x55d5c325d020] printf@GLIBC_2.2.5 → 0x7f121ead1470
[0x55d5c325d028] srand@GLIBC_2.2.5 → 0x7f121eab9a10
[0x55d5c325d030] time@GLIBC_2.2.5 → 0x7ffded6dfee0
[0x55d5c325d038] sleep@GLIBC_2.2.5 → 0x7f121ec43115
[0x55d5c325d040] rand@GLIBC_2.2.5 → 0x7f121eaba110
```

```
gef> info symbol 0x7f121ec43115
sleep in section .text of ./fake_sleep.so
```

# what about the call to the *real* sleep()

```
gef> disas sleep
```

```
Dump of assembler code for function sleep:
```

```
0x43115 <+0>:  push    rbp
0x43116 <+1>:  mov     rbp, rsp
0x43119 <+4>:  sub     rsp, 0x20
0x4311d <+8>:  mov     DWORD PTR [rbp-0x14], edi
0x43120 <+11>: lea     rdi, [rip+0xed9]
0x43127 <+18>: call    0x7f121ec43030 <puts@plt>
0x4312c <+23>: mov     DWORD PTR [rbp-0x14], 0x1
0x43133 <+30>: lea     rsi, [rip+0xeda]
0x4313a <+37>: mov     rdi, 0xffffffffffffffff
0x43141 <+44>: call    0x7f121ec43040 <dlsym@plt>
0x43146 <+49>: mov     QWORD PTR [rbp-0x8], rax
0x4314a <+53>: mov     eax, DWORD PTR [rbp-0x14]
0x4314d <+56>: mov     rdx, QWORD PTR [rbp-0x8]
0x43151 <+60>: mov     edi, eax
0x43153 <+62>: call    rdx
0x43155 <+64>: leave
0x43156 <+65>: ret
```

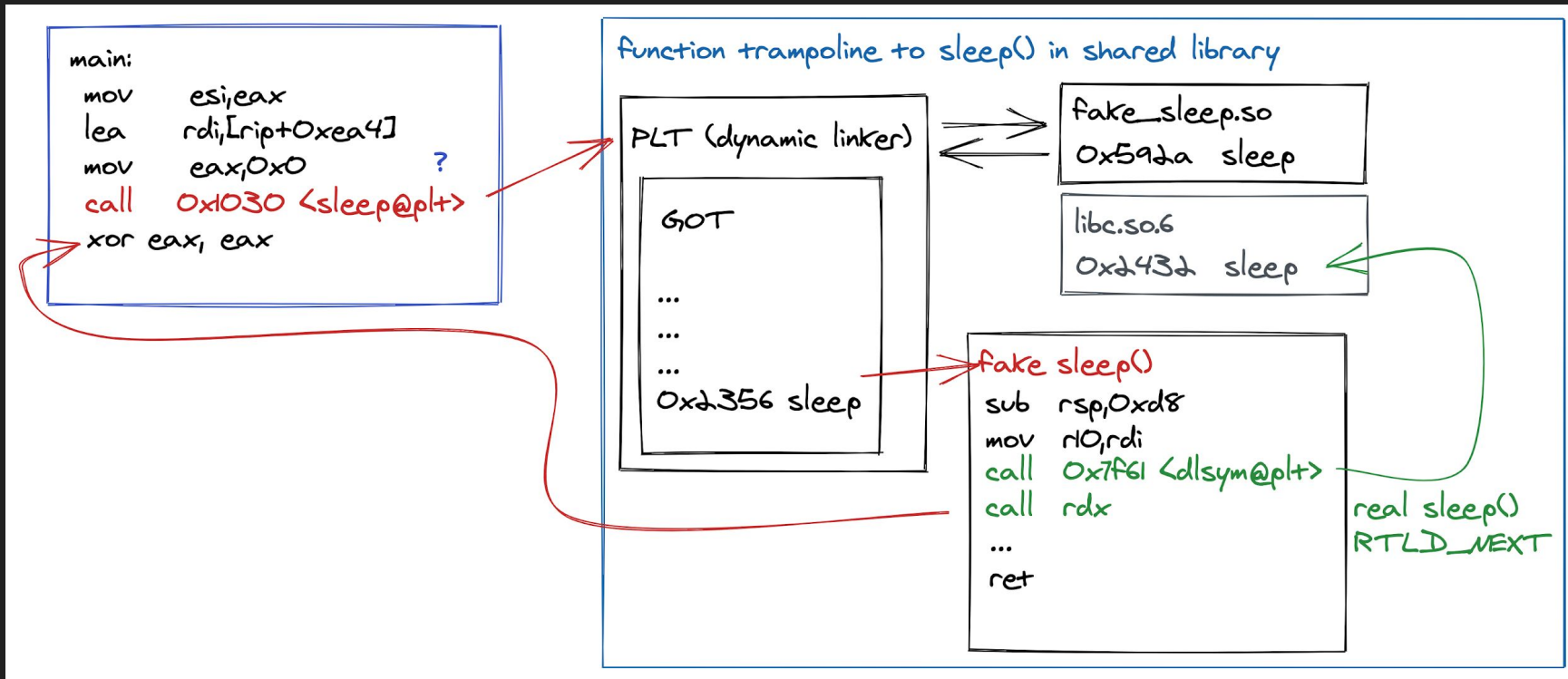
```
End of assembler dump.
```

`break` on the instruction that calls the  
`RDX` register

`continue` execution until the breakpoint  
is hit

`inspect` the address that is in `RDX`

# LD\_PRELOAD-ed sleep()





## recap - LD\_PRELOAD under the hood

- the dynamic linker resolves function locations at runtime, storing results in the `got`.
- libraries specified with LD\_PRELOAD get `preference`.
- debuggers are `not` as scary as they seem to be :)

chapter 0x2

part 0x1

# FRIIDA

# multiple modes of operation

- **injected**
  - on the same computer using bindings
  - remote device via frida-server
- **embedded**
  - running local or remote
  - completely autonomous

# components

- core (injector and all of the glue)
- language bindings (C, Python, Node, .NET, etc.)
- runtime bridges for ObjC & Java (focussed on mobile platforms)
- cli tools; frida, frida-ps, frida-trace, etc.

our tool

```
script = session.create_script(source)
script.load()
```

DBUS ↓ ↑

target application

frida-agent session

```
var p = Module.  
findExportByName(  
  "libc.so.6", "puts");
```

**\*\* basically magic \*\***

```
push rbp  
mov rbp, rsp  
sub rsp, 0x10  
lea rdi, [rip+0xea0]  
call 0x1030 <puts@plt>  
ret
```

0x457a puts libc.so.6

the difference between `spawning` and `attaching` to a target process

- `spawning` with Frida will launch and `pause` the target application, great for `early instrumentation`.
- `attaching` will `inject` Frida to a running process (using `PTRACE`), launching a new `thread` to use for instrumentation.

# spawning an application with frida

```
~/code$ frida ./pew
```

```
  /_/_/_/_|
 | ( _ | |
  > _ |
/_/_/_/_|_|
. . . .
. . . .
. . . .
. . . .
. . . .
More info at https://www.frida.re/docs/home/
```

```
Spawned `./pew`. Use %resume to let the main thread start executing!
[Local::pew]→
```



# attaching to an application with frida

```
~/code$ frida pew
```

```

/_____|
| ( _ |
> _ |
/_/_|_
. . . .
. . . .
. . . .
. . . .
More info at https://www.frida.re/docs/home/

```

Frida 12.9.2 - A world-class dynamic instrumentation toolkit

Commands:

- help → Displays the help system
- object? → Display information about 'object'
- exit/quit → Exit

[Local :: pew] →

## examples of launching the Frida REPL

- `frida ./pew` # spawn's the app
- `frida pew` # attaches to the app
- `frida -p 23`
- `frida -p $(pidof pew)`

instrumentation logic is written in  
JavaScript.

frida exposes its own API on top of the  
JavaScript standard library.

<https://www.frida.re/docs/javascript-api/>

# the frida REPL

```
frida-boot:~/code$ frida pew
```

```

/-----|
| ( _ |
| >
| / _ |
. . . .
. . . .
. . . .
. . . .
More info at https://www.frida.re/docs/home/

```

Frida 12.9.2 - A world-class dynamic instrumentation toolkit

Commands:

- help -> Displays the help system
- object? -> Display information about 'object'
- exit/quit -> Exit

```
[Local::pew]->
```

```
[Local::pew]-> Process.enumerate
```

```
_enumerateMallocRanges
_enumerateModules
_enumerateRanges
_enumerateThreads
enumerateMallocRanges
enumerateMallocRangesSync
enumerateModules
```

# external scripts with auto reload

```
~/code$ cat index.js  
console.log(Frida.version);
```

```
~/code$ frida pew -l index.js  
12.9.3
```

use the REPL for quick prototyping

use a script for long term development

chapter 0x2

part 0x2



```
Interceptor.attach(target, callbacks);
```

figuring out the `target` to `attach` to

`offsets` || `symbols`

resolving `sleep()` statically

```
~/code$ nm pew | grep sleep  
U sleep@@GLIBC_2.2.5
```

# resolving `sleep()` statically

```
~/code$ ldd pew
linux-vdso.so.1 (0x00007ffc9b5f4000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f552ca13000)
/lib64/ld-linux-x86-64.so.2 (0x00007f552cbe1000)
```

# resolving `sleep()` statically

```
nm -D /lib/x86_64-linux-gnu/libc.so.6 | grep sleep
0000000000010a3c0 T __clock_nanosleep
0000000000010a3c0 W clock_nanosleep
000000000000cae80 T __nanosleep
000000000000cae80 W nanosleep
000000000000f35d0 T __nanosleep_nocancel
000000000000cad90 W sleep
00000000000084d70 T thrd_sleep
000000000000f5870 T usleep
```

`sleep @ 0xcad90`

confirming the offset of `sleep()` in `libc`

- run a new debugging session of `pew`
- break on `main` and run with `r`
- check `info proc map` or `vmmap` for `libc`
- take that mapped offset and add `0xcad90` in the `info symbol` command

```
gef> info proc map
process 33
Mapped address spaces:
```

Start Addr	End Addr	Size	Offset	objfile
[ ... ]				
0x564cc8854000	0x564cc8855000	0x1000	0x0	/root/code/pew
0x564cc8855000	0x564cc8856000	0x1000	0x1000	/root/code/pew
0x564cc8856000	0x564cc8857000	0x1000	0x2000	/root/code/pew
0x564cc8857000	0x564cc8858000	0x1000	0x2000	/root/code/pew
<b>0x7f06f680b000</b>	<b>0x7f06f6830000</b>	<b>0x25000</b>	<b>0x0</b>	<b>/lib/x86_64-linux-gnu/libc-2.30.so</b>
0x7f06f6830000	0x7f06f697a000	0x14a000	0x25000	/lib/x86_64-linux-gnu/libc-2.30.so
0x7f06f697a000	0x7f06f69c4000	0x4a000	0x16f000	/lib/x86_64-linux-gnu/libc-2.30.so
0x7f06f69c4000	0x7f06f69c7000	0x3000	0x1b8000	/lib/x86_64-linux-gnu/libc-2.30.so

```
gef> info symbol 0x7f06f680b000 + 0xcad90
sleep in section .text of /lib/x86_64-linux-gnu/libc.so.6
```

options to resolve `sleep()` with Frida

(time to learn some Frida API's)

- calculate the offset and `add()` it to a `Module.getBaseAddress()` address
- use `Module.getExportByName()`
- use `DebugSymbol.getFunctionByName()`



# sleep() with frida and an offset

```
[Local::pew]→ Process.enumerateModulesSync();  
[  
  {  
    "base": "0x7fc2edb78000",  
    "name": "libc-2.30.so",  
    "path": "/lib/x86_64-linux-gnu/libc-2.30.so",  
    "size": 1830912  
  },  
  {  
    "base": "0x7fc2edd41000",  
    "name": "ld-2.30.so",  
    "path": "/lib/x86_64-linux-gnu/ld-2.30.so",  
    "size": 172032  
  },  
  [ ... ]  
]
```

## sleep() with frida and an offset

```
[Local::pew]→ Process.getModuleByName("libc-2.30.so");  
{  
    "base": "0x7fc2edb78000",  
    "name": "libc-2.30.so",  
    "path": "/lib/x86_64-linux-gnu/libc-2.30.so",  
    "size": 1830912  
}
```

# sleep() with frida and an offset

```
[Local::pew]→ Process.getModuleByName("libc-2.30.so").base;  
"0x7fc2edb78000"
```

```
[Local::pew]→ Module.getBaseAddress("libc-2.30.so");  
"0x7fc2edb78000"
```

`sleep()` with frida and an offset

```
Module.getBaseAddress("libc-2.30.so").add("0xcad90");  
"0x7fc2edc42d90"
```

# sleep() with frida and an export

```
[Local::pew]→ Process.getModuleByName("libc-2.30.so").enumerateExports();  
[  
  {  
    "address": "0x7fc2edc71ef0",  
    "name": "vwarn",  
    "type": "function"  
  },  
  {  
    "address": "0x7fc2edc6a370",  
    "name": "fts64_close",  
    "type": "function"  
  },  
  [ ... ]  
]
```

`sleep()` with frida and an export

```
[Local::pew]→ Module.getExportByName(null, "sleep");  
"0x7fc2edc42d90"
```

`sleep()` with frida and a debug symbol

```
[Local::pew] → DebugSymbol.getFunctionByName("sleep");  
"0x7fc2edc42d90"
```

we have the `target`, what about the  
`callbacks`?

```
Interceptor.attach(target, callbacks);
```



```
Interceptor.attach(sleepPtr, {  
    onEnter: function(args) {},  
    onLeave: function(retval) {}  
});
```

```
var sleep = Module.getExportByName(null, "sleep");

Interceptor.attach(sleep, {

    onEnter: function(args) {
        console.log("[*] Sleep from Frida!");
    },

    onLeave: function(retval) {
        console.log("[*] Done sleeping from Frida!");
    }

});
```

attach to `sleep()` in `pew`

```
~/code$ cp  
../software/interceptor-attach.js.1  
interceptor-attach.js
```

```
~/code$ frida pew -l  
interceptor-attach.js
```

what did attach do to `sleep`?

- attach gdb to the running instance of pew while the instrumentation is running with `gdb -q -p $(pidof pew)`
- disassemble the `sleep` function and have a look at the start with `disas sleep`

```
# with Interceptor.attach()
```

```
gef> disas sleep
```

```
Dump of assembler code for function sleep:
```

```
0x00007fc2edc42d90 <+0>:      jmp      0x7fc2ed197708  
0x00007fc2edc42d95 <+5>:      nop
```

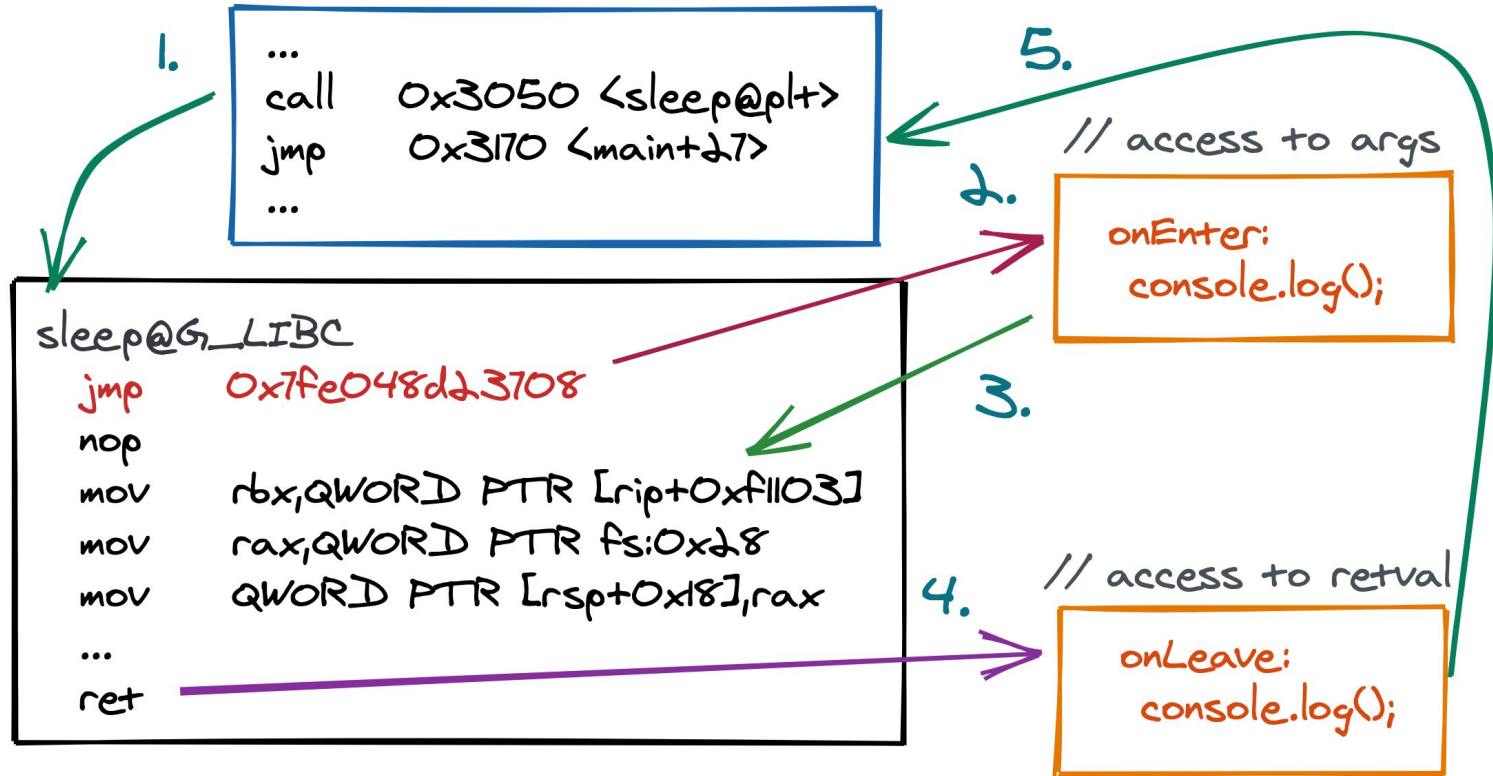
```
# without Interceptor.attach()
```

```
gef> disas sleep
```

```
Dump of assembler code for function sleep:
```

```
0x00007fc2edc42d90 <+0>:      push      rbp  
0x00007fc2edc42d91 <+1>:      push      rbx
```

## Interceptor.attach()



chapter 0x2

part 0x3

overriding arguments



interceptor arguments

```
onEnter: function(args);
```

# interceptor arguments

- does **not** know **how many** arguments there are, by design
- **args[0]** is the first argument
- args are of type **NativePointer**

```
args = [  
    NativePointer("0x01"),      # args[0]  
    NativePointer("0x0fe4795"), # args[1]  
    ...  
];
```

*len(args) + 1 of the real function is possible, but is *\*not\** an argument*

# interceptor arguments

```
onEnter: function(args) {  
    console.log("[*] Argument for sleep() ⇒ " +  
                parseInt(args[0]));  
  
    console.log("[*] Sleep from Frida!");  
}
```

```
~/code$ cp  
../software/interceptor-attach.js.2  
interceptor-attach.js
```

```
~/code$ frida pew -l  
interceptor-attach.js
```

now modify the argument to `sleep()`

```
args[0] = ptr("0x01");
```

# or

```
args[0] = new NativePointer("0x01");
```

integers are easy, what about strings?

- allocate a new character array with `Memory.allocUtf8String` and save the pointer
- assign the relevant `arg` to the value of the new pointer
- profit?

override the argument to `printf()`,  
you know enough to do this yourself. if  
not ...

```
~/code$ cp  
../software/interceptor-attach.js.3  
interceptor-attach.js
```

```
~/code$ frida pew -l  
interceptor-attach.js
```



you have `register` access here too..

```
var printf = Module.getExportByName(null, "printf");

Interceptor.attach(printf, {
  onEnter: function(args) {
    console.log(JSON.stringify(this.context, null, 4));
  }
});
```

overriding return values

interceptor return values

```
onLeave: function(retval);
```

let's find our target, `rand_range`

which api will you use to get it's  
address?

```
[Local::pew]→ DebugSymbol.getFunctionByName("rand_range");  
"0x55a5ef21d185"
```

```
var rand_range = DebugSymbol
    .getFunctionByName("rand_range");

Interceptor.attach(rand_range, {
    onLeave: function(retval) {

        console.log(retval);
    }
});
```

```
~/code$ cp  
../software/interceptor-attach.js.4  
interceptor-attach.js
```

```
~/code$ frida pew -l  
interceptor-attach.js
```

to replace a return value, we call  
`replace()` on the return value

```
retval.replace(ptr("0x01"));
```

your turn ...



databinding between `onEnter` and `onLeave`

```
onEnter: function(a) {  
    this.value = ptr("0x01");  
},
```

# **this** is in scope

```
onLeave: function(r) {  
    r.replace(this.value);  
}
```

```
~/code$ cp  
.. /software/interceptor-attach.js.5  
interceptor-attach.js
```

```
~/code$ frida pew -l  
interceptor-attach.js
```

chapter 0x2

part 0x4

**reusing existing code**

we are done with `pew`. enter `crypt`.

*try not to peek at the source code, but  
get and compile `crypt` with:*

```
~/code$ cp ../software/crypt.c .
```

```
~/code$ gcc crypt.c -o crypt
```

run `crypt` to get a feel for how it works

```
~/code$ ./crypt
```

```
Pin: 1234
```

```
Pin:
```

imagine you don't have the source code for crypt (which will most often be the case), so let's analyse it.

<https://cloud.binary.ninja/>

## takeaways from analysing `crypt`

- success condition will print `Pwnd!!`
- `test_pin` should return **`true`** (`0x1`) to be successful
- `test_pin` tests a value against `0xd64`



static analysis aside, let's do *dynamic*  
analysis of `test_pin`!

use `interceptor.attach()` on `test_pin`

what are the arguments and return  
values?

```
~/code$ cp ../software/crypt.js.1  
crypt.js
```

```
~/code$ frida crypt -l crypt.js
```

```
[Local::crypt]→  
test_pin(0x7fff33742ac2)  
⇒ ret: 0x0  
test_pin(0x7fff33742ac2)  
⇒ ret: 0x0  
test_pin(0x7fff33742ac2)  
⇒ ret: 0x0
```

`hexdump` the address the first argument  
(`args[0]`) points to

```
onEnter: function(args) {  
    console.log(hexdump(args[0]));  
}
```

```
[Local::crypt]→ test_pin(0x7fff33742ac2)
```

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
7fff33742ac2	31	32	33	34	0a	00	00	00	00	00	00	00	00	00	10	a2	1234.....
7fff33742ad2	54	e1	e4	55	00	00	0b	4e	25	a7	96	7f	00	00	00	00	T..U...N%.....

```
# we can print the string too
```

```
onEnter: function(args) {  
    console.log("test_pin(" +  
        args[0].readCString().trim() +  
        ")");  
}
```

so what will be the easy win here?

swap the return value to 0x1!

give it a try now.

```
var testPin = DebugSymbol.getFunctionByName("test_pin");  
  
Interceptor.attach(testPin, {  
    onLeave: function(retval) {  
        console.log(" ⇒ ret: " + retval);  
  
        retval.replace(ptr("0x1")); // ←  
    }  
});
```



```
~/code$ cp ../software/crypt.js.2  
crypt.js
```

```
~/code$ frida crypt -l crypt.js
```

Pin: 1

Pwnd !!

**so that's cool 'n all, but what's the  
real pin?**

what if **we** could call `test_pin()`  
ourselves? we know the argument and the  
return type ...

```
new NativeFunction(address, returnType,  
                    argTypes[, abi]);
```

```
var testPinPtr = DebugSymbol.getFunctionByName("test_pin");

var testPin = new NativeFunction(
    testPinPtr, "int", ["pointer"]);

// Try a PIN of 1111

var pin = Memory.allocUtf8String("1111");
var r = testPin(pin);

console.log(r);
```

```
~/code$ cp ../software/crypt.js.3  
crypt.js
```

```
~/code$ frida crypt -l crypt.js
```

so let's loop that and brute the pin?

```
for (var i = 0; i < 9999; i++) {  
  
    var pin = Memory.allocUtf8String(i.toString());  
    var r = testPin(pin);  
  
    if (r == 1) {  
        console.log("Pin is: " + i.toString());  
        break;  
    }  
}
```

```
~/code$ cp ../software/crypt.js.4  
crypt.js
```

```
~/code$ frida crypt -l crypt.js
```



[ ... ]

Trying: 3422

Trying: 3423

Trying: 3424

Trying: 3425

Trying: 3426

Trying: 3427

Trying: 3428

Pin is: 3428

**Pin: 3428**

**Pwnd !!**

0\_o

o\_0

chapter 0x3

part 0x1

<https://codeshare.frida.re/>

- use python bindings to attach/spawn to targets
- inject frida scripts
- accept input

```
import frida
import sys

session = frida.attach("crypt")
script = session.create_script("""
    console.log("ver: " + Frida.version);
""")
script.load()
```

```
~/code$ cp ../software/tool.py.1 tool.py
```

```
~/code$ python3 tool.py
```

replace the `script` with our `pin` brute  
forcer now

```
[ ... ]
```

```
script = session.create_script("""  
    var testPinPtr =  
        DebugSymbol.getFunctionByName("test_pin");
```

```
    [ ... ]  
    """)
```



```
~/code$ cp ../software/tool.py.2 tool.py
```

```
~/code$ python3 tool.py
```

two languages in the same file is not bad, but it's not great either so let's refactor again

```
import frida
import sys

# read the agent source
with open("index.js", "r") as f:
    agent = f.read()

session = frida.attach("crypt")
script = session.create_script(agent)
script.load()
```

```
~/code$ cp ../software/tool.py.3 tool.py
```

```
~/code$ cp ../software/index.js.1  
index.js
```

```
~/code$ python3 tool.py
```

chapter 0x3

part 0x2

# the `script` loading lifecycle

`script.load()`

~30 seconds

```
var testPinPtr = DebugSymbol  
    .getFunctionByName("test_pin");  
  
for (var i = 0; i < 9999; i++) {  
    doStuff();  
}
```

- so far we abused the `startup` to let our script finish
- longer running scripts (like those using the `Interceptor`) will need a `blocking function` so we don't just `exit`
- in python we can do this with `sys.stdin.read()` to wait for input

```
import sys
```

```
[ ... ]
```

```
session = frida.attach("crypt")  
script = session.create_script(agent)  
script.load()
```

```
# block so that the program does not quit.  
sys.stdin.read()
```



`send()` and `recv()`

```
// javascript                                send() example
var answer = 42;
send(answer);
```

```
# python
def incoming(message, data):
    print(message)

script.on("message", incoming)
```

- update your agent script to `send()` a message when the brute forcer starts and when it is finished
- update the python tool to handle `incoming` messages

```
~/code$ cp ../software/tool.py.4 tool.py
```

```
~/code$ cp ../software/index.js.2  
index.js
```

```
~/code$ python3 tool.py
```

```
// javascript                                recv() example
recv(function(m) {
    console.log("message: " + m);
});
```

```
# python
script.on("message", incoming)
script.load()
script.post("test")
```

update your agent to **receive** a message  
from the python environment and print it

```
~/code$ cp ../software/tool.py.5 tool.py
```

```
~/code$ cp ../software/index.js.3  
index.js
```

```
~/code$ python3 tool.py
```

# frida **RPC** interface

(the bindings to agent glue)



- `call` functions defined in the JavaScript agent, `from python`
- pass along `arguments` from python, and get `return values` from JavaScript
- `don't` have to use `send()` / `recv()`
- does not block `script.load()`

- functions in the agent should be exported in the global `rpc.exports` object in the agent
- exports are accessed from the `script.exports` property in python
- the function `testPin()` defined in the agent is called `test_pin()` in python

# javascript `exports` definition

```
rpc.exports = {  
    brute: function() {  
        console.log("Brute function");  
    }  
}
```

accessing `exports` in python

```
script.exports.brute()
```

let's refactor once more, this time  
implementing the brute force logic in an  
agent with a `rpc` export

```
~/code$ cp ../software/tool.py.6 tool.py
```

```
~/code$ cp ../software/index.js.4  
index.js
```

```
~/code$ python3 tool.py
```

one more refactor where we implement the pin check loop in `python` and simplify or agent

```
# python
```

```
for x in range(0, 9999):  
    res = test_pin(str(x))
```

```
[ ... ]
```



```
~/code$ cp ../software/tool.py.7 tool.py
```

```
~/code$ cp ../software/index.js.5  
index.js
```

```
~/code$ python3 tool.py
```

from `python`, you just called a  
`javascript` method that called a `c` method  
and returned some `results` back...

(•\_•)

( •\_• ) > r■-■

( r■\_■ )

chapter 0x3

part 0x3

typescript && frida = <3

# crash course for us mere mortals

- `typescript` is a superset of `javascript`
- valid `javascript` is `also` valid `typescript`
- `frida-compile` can take `typescript` and `transpile` to any target (important for `duktape/v8` language support)

`frida-compile` exposes the entire NPM ecosystem to use inside of agents

what do you need?

- vscode
- node
- `oleavr/frida-agent-example` repository

# what does it look like?

11

12

`Interceptor.at`

 `attach`

 `detachAll`

`function Interceptor.attach(target: NativePointerValue, callbacksOrProbe: ScriptInvocationListenerCallbacks | NativeInvocationListenerCallbacks | InstructionProbeCallback, data?: NativePointerValue): InvocationListener`

Intercepts calls to function/instruction at `target`. It is important to specify a `InstructionProbeCallback` if

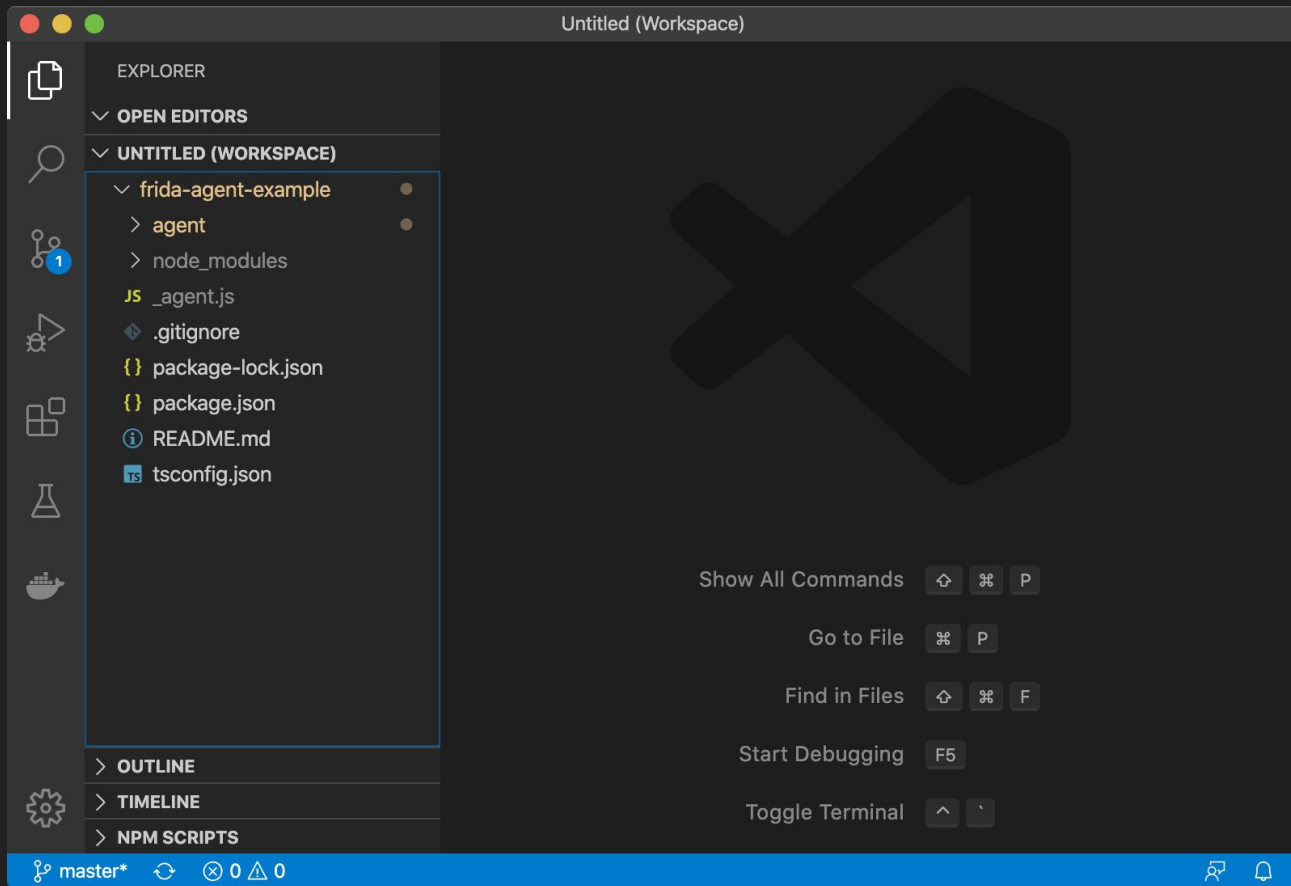
- a-mazing inline documentation
- autocompletion
- benefits of strictly typed TS



# setup for this workshop

- already have `node` & the repo cloned and all `dependencies` installed
- copy the repo to your `code/` folder
- add the `frida-agent-example` folder to `vscode`
- play with the autocompletion

```
~/code$ cp -R ~/frida-agent-example/ .
```



npm run `watch` vs npm run `build`

- `build` will perform a single shot build of the agent
- `watch` will monitoring for changes and automatically rebuild
- both result in a new `_agent.js` file

implement `testPin` in the example `index.ts` file or copy the example from the software folder

```
~/code/frida-agent-example$ cp  
  ~/software/index.ts.1 agent/index.ts
```

finally, update the tool.py to read the generated `_agent.js` file

```
with open("frida-agent-example/_agent.js", "r") as f:
```

a few moving parts but now you're ready

- run `./crypt`
- run `npm run watch`
- run `python3 tool.py`
- edit `index.ts` and watch the recompile

what about starting up a web server in  
the `target process` and try and guess the  
pin using `http requests`?



```
~/code/frida-agent-example$ cp  
  ~/software/index.ts.2 agent/index.ts
```

```
~/code$ cp ../software/tool.py.8 tool.py
```

```
~$ curl -v localhost:1337/3428
```

( ° Д ° )

chapter 0x3

part 0x4

tools part of the `frida-tools` pip package

- `frida`
- `frida-discover`
- `frida-kill`
- `frida-ls-devices`
- `frida-ps`
- `frida-trace`

`frida-trace`, a tool for quick function call tracing

- generate `interceptor-like` hooks
- wildcard symbol `resolution`
- `man` page reference for function args
- dumps hooks in `__handlers__` folder

try it

```
$ frida-trace crypt -i "ato*"
```

chapter 0x3

part 0x5



you've made it this far, let's cover  
operating modes quickly

- frida-server & frida-gadget opens a socket that clients to
- frida-server usually runs standalone
- frida-gadget run embedded / preloaded
  - LD\_PRELOAD
  - patching the gadget to load
  - can be configured

using `frida-server`

the docker container has `frida-server` in the `$PATH` already. run it with  
`frida-server -l 0.0.0.0:1337`

next, `frida-ps -H localhost:1337`

using the `frida-ps` output, you can now tell frida which target to `attach` to

```
$ frida -H localhost:1337 crypt
$ frida -H localhost:1337 -p 24
```

in this mode, `frida-server` does the heavy lifting (injection etc.), client tools just communicate with it over the socket

using `frida-gadget`

- load with `LD_PRELOAD`
- load with `patchelf`

apps launched with the `Gadget` are `paused` until a `client connects`. this can be changed with a `gadget config`.

frida-gadget using LD\_PRELOAD

```
~$ LD_PRELOAD=./frida-gadget.so  
code/crypt
```

```
[Frida INFO] Listening on 127.0.0.1 TCP  
port 27042
```

```
~$ frida-ps -R
```

```
PID  Name
```

```
--  -----
```

```
78  Gadget
```

```
# or run frida-ps -H localhost
```

# run the frida REPL connecting to the remote gadget

# failed invocation

```
~$ frida -q -R crypt -e "console.log(Process.id);"  
Failed to spawn: unable to find process with name 'crypt'
```

# successful invocation

```
~$ frida -q -R Gadget -e "console.log(Process.id);"  
12
```



notice how when the client connected,  
the **Pin** prompt displayed. this is  
because the application **resumed** when the  
client connected

frida-gadget by patching (patchelf)

```
~/code$ patchelf --add-needed  
../frida-gadget.so crypt
```

```
~/code$ ./crypt  
[Frida INFO] Listening on 127.0.0.1 TCP  
port 27042
```

```
~/code$ ldd crypt
linux-vdso.so.1 (0×00007ffdc7962000)
../frida-gadget.so (0×00007fceeeca4b000)
libc.so.6 ⇒ /lib/x86_64-linux-gnu/libc.so.6 (0×00007fceeec884000)
libdl.so.2 ⇒ /lib/x86_64-linux-gnu/libdl.so.2 (0×00007fceeec87f000)

[ ... ]
```

# configuring the gadget

- a small `json` file `next` to the gadget
- same name with the `.config` extension

```
{  
  "interaction": {  
    "type"      : "listen",  
    "address"   : "127.0.0.1",  
    "port"      : 27042,  
    "on_load"   : "wait"  
  }  
}
```

add a gadget configuration, changing the default `wait` behaviour to `resume`.

```
~$ cp software/frida-gadget.config.1  
    frida-gadget.config
```

```
{  
  "interaction": {  
    "type": "script",  
    "path": "/root/code/agent.js"  
  }  
}
```

```
{  
  "interaction": {  
    "type": "script-directory",  
    "path": "/usr/local/frida/scripts"  
  }  
}
```



add a gadget configuration, embedding an agent to print the crypt `pin`.

```
~$ cp software/frida-gadget.config.2  
    frida-gadget.config
```

```
~$ cp software/embedded-agent.js  
    code/embedded-agent.js
```

thank you!

<https://t.me/fridadotre>

want to work with us? see our other  
training?

<https://sensepost.com/services/education>

<https://sensepost.com/contact>

<https://twitter.com/sensepost>

# bonus chapter

```
new CModule()
```

compile C source code, in memory, from  
your JavaScript (using TinyCC)

```
const cm = new CModule(`  
#include <stdio.h>  
  
void init() {  
    printf("hello from CModule\n");  
}  
`);
```

limited available headers, by design

- glib.h
- stdio.h
- stdlib.h
- etc ...

<https://github.com/frida/frida-gum/tree/master/bindings/gumjs/runtime/cmodule>



```
const cm = new CModule(`  
int value() {  
    return 42;  
}  
`);
```

```
const v = new NativeFunction(cm.value,  
    'int', []); // no args for our func
```

```
v(); // 42
```

```
const p = Memory.alloc(4);  
const cm = new CModule(`  
#include <glib.h>  
#include <stdio.h>  
  
extern volatile gpointer p;  
  
int value() {  
    printf("%p", p);  
}  
`, { p }); // ← pass a symbol
```

frida REPL can load `.c` files  
(with the same magical auto-reload  
goodies you are used to)

```
$ frida crypt -l index.js -C test.c
```

```
~/code$ cp ../software/tool.py.9 tool.py
~/code$ cp ../software/index.js.6
index.js
```

`test_pin`, from `python`, calling  
`JavaScript`, calling a `CModule`, calling  
the real `C` implementation

.fini