Dynamic Instrumentation and Fuzzing with Frida

Sean Deaton, 21 May 2025

About

- Former Army Officer, keyboard warrior
  - Unix Access Crew Lead
- Went to work on Linux kernel security, lots of fuzzing
- Switched over to iOS
- Go Army, Beat Navy

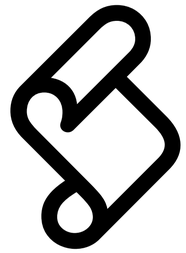
Agenda

- Instrumentation and Frida
- Injection by Example
- API Docs
- Frida Stalker
- Fuzzing with fpicker
- Using Frida as an iOS researcher

Instrumentation

Running Code

Our Tooling



Inject

```
int main() {
    /* Set up code. */

    char buf[128];
    int len = recv(c, buf, sizeof(buf) - 1, 0);
    if (len <= 0) return EXIT_FAILURE;
    buf[len] = '\0';

    printf(buf); // Bonk

    /* Tear down code. */
```

Output

- Memory
- Function Calls
- Trace Execution
- Modify State
- Coverage

Frida

- Dynamic instrumentation toolkit
- Works with GNU/Linux, Windows, *OS, Android, etc.
  - Portable to custom archs
- Inject scripts of Javascript into the program
  - Also has C, Swift, and Go bindings[1]
- No recompilation necessary

[1]With worse API documentation.

Modes of Operation

- Injected
  - Spawn or attach to a program
- Embedded
  - Where injected is not possible (jailed *OS and Android)
  - Inject a frida-core
- Preloaded
  - LD_PRELOAD
  - DYLD_INSERT_LIBRARY
    - Not possible on iOS/iPadOS/tvOS/visionOS

Injection by Example

```c
int main() {
    int s = socket(AF_INET, SOCK_STREAM, 0);
    struct sockaddr_in addr = {AF_INET, htons(4444), INADDR_ANY};
    bind(s, (struct sockaddr *)&addr, sizeof(addr));
    listen(s, 1);
    int c = accept(s, NULL, NULL);

    char buf[128];
    int len = recv(c, buf, sizeof(buf) - 1, 0);
    if (len <= 0) return 1;
    buf[len] = '\0';

    log_message(buf); // bonk
    ...
}
```

clang -Wno-format-security -o main main.c

```c
void log_message(char * message) {
    fprintf(stdout, message); //bonk
    return;
}
```

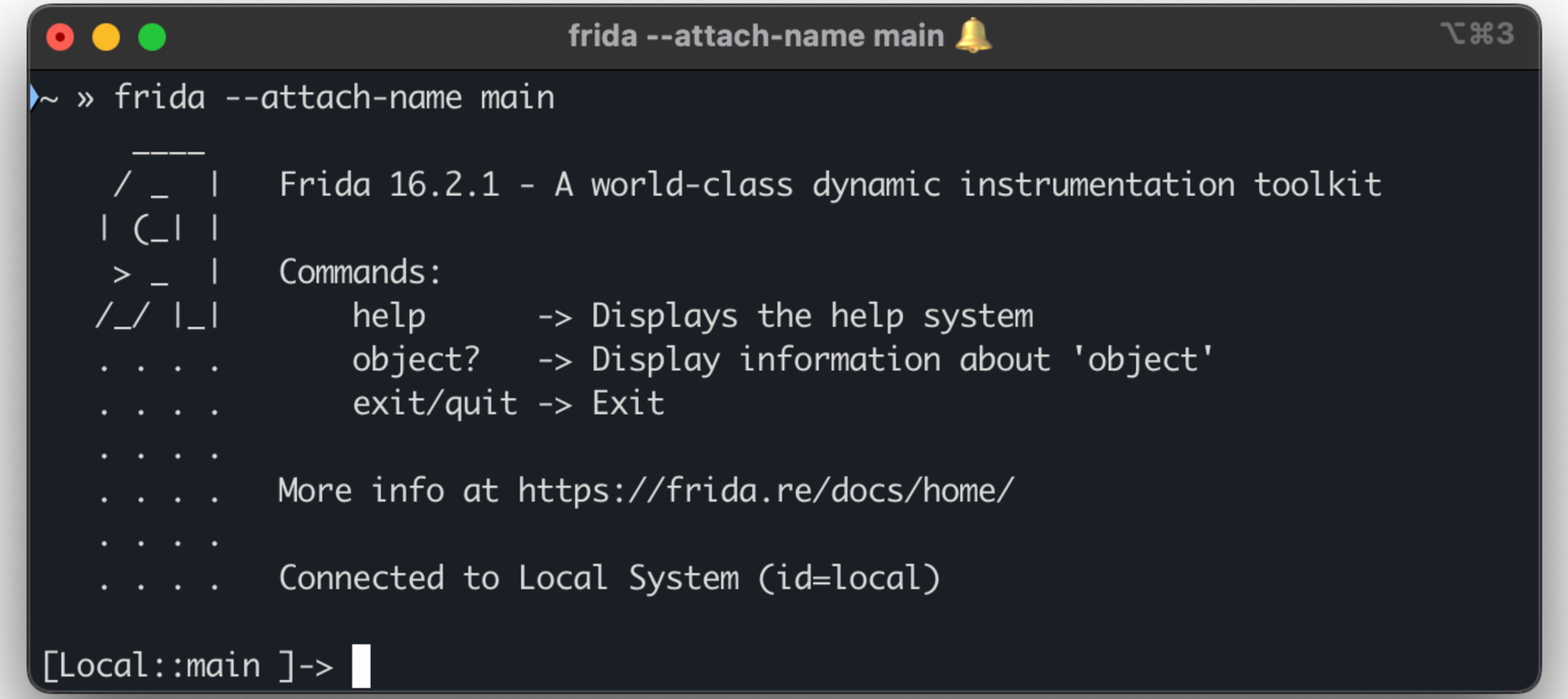
clang -Wno-format-security -o main main.c

```
./main
```

Run the program

```
frida -attach-name main
```

Attach



OVERVIEW DOCS NEWS CODE CONTACT

Getting Started

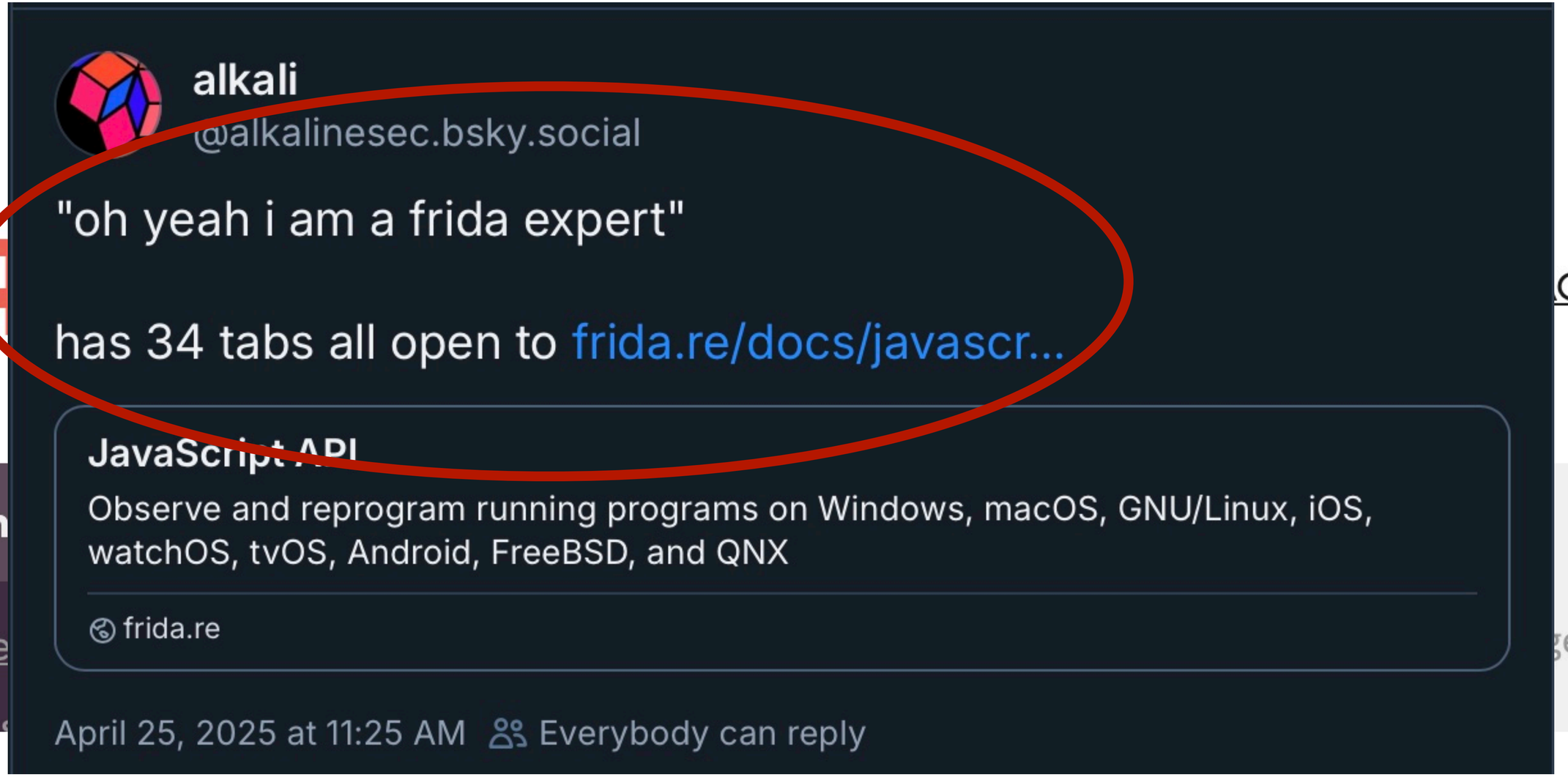
Welcome

Quick-start guide

JavaScript API

Improve this page

Now we can open 100 tabs, each open to different part of the API.

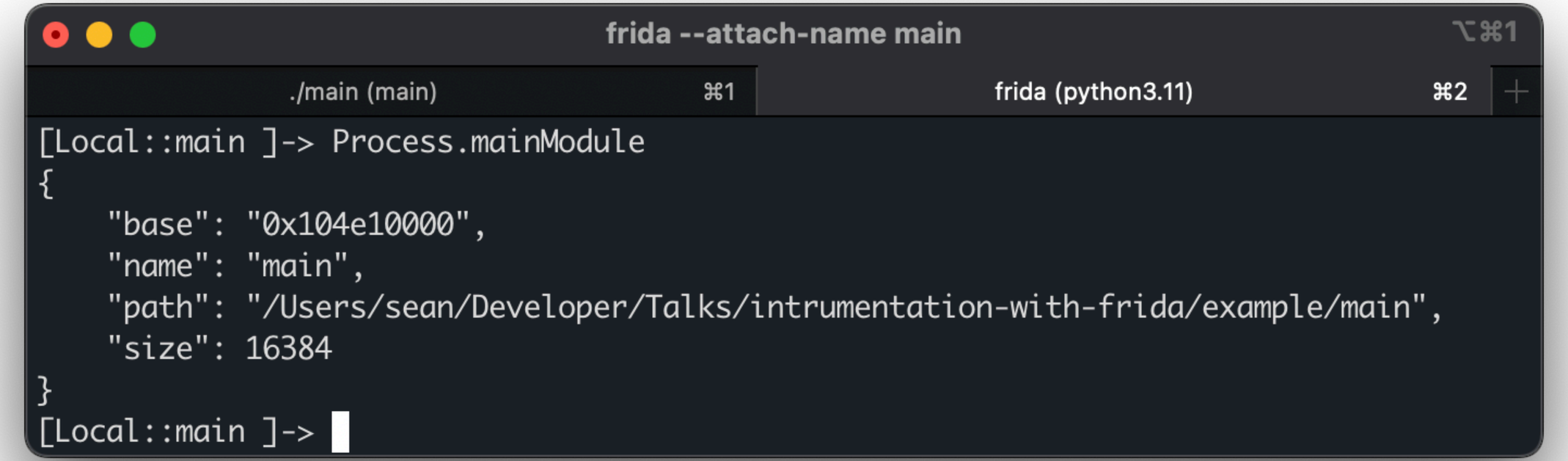


Now we can open 100 tabs, each open to different part of the API.

A Few Important Classes…

Process

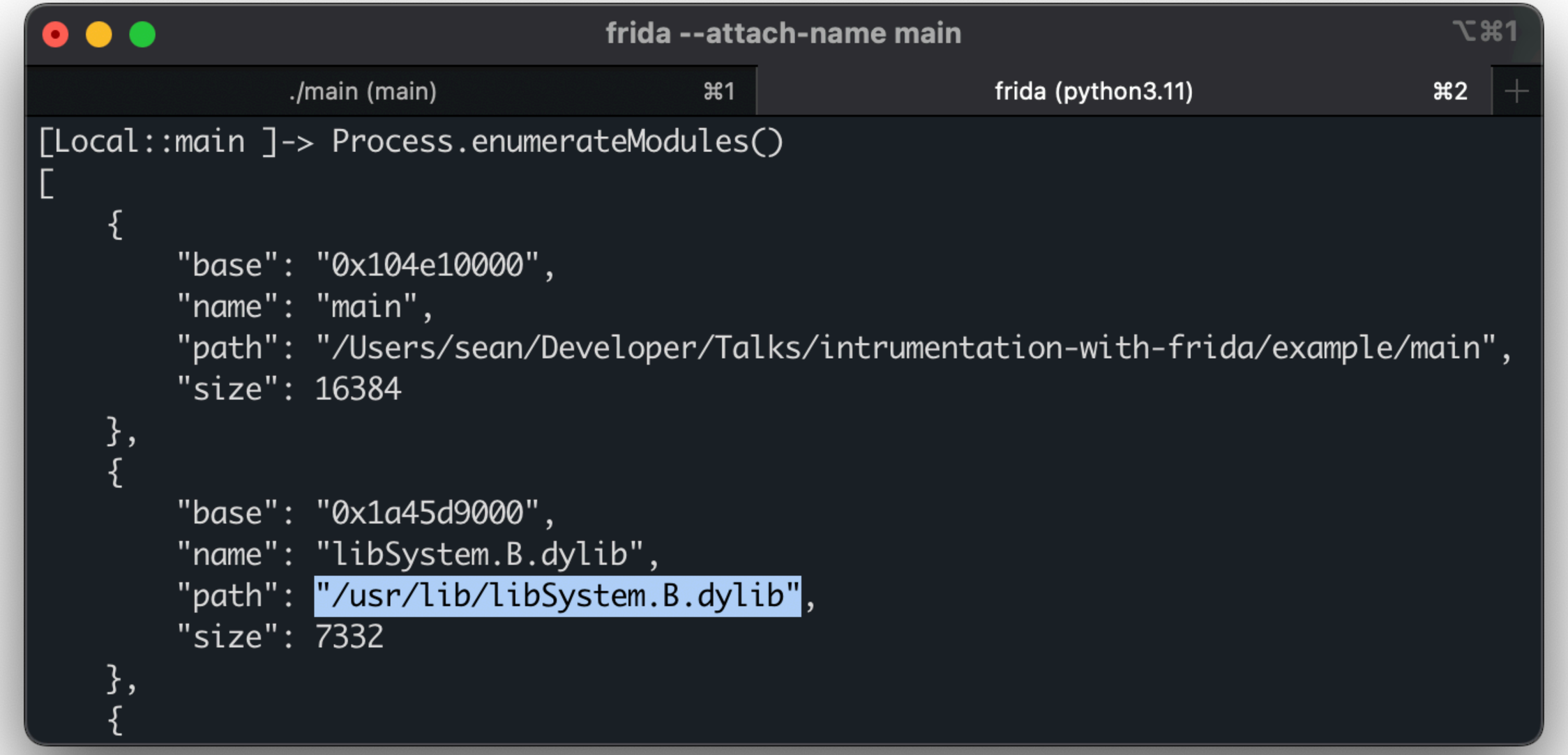
- `Process.id`: property containing the PID as a number
- `Process.arch`: property containing the string `ia32`, `x64`, `arm` or `arm64`
- `Process.platform`: property containing the string `windows`, `darwin`, `linux`, `freebsd`, `qnx`, or `barebone`
- `Process.pageSize`: property containing the size of a virtual memory page (in bytes) as a number. This is used to make your scripts more portable.
- `Process.pointerSize`: property containing the size of a pointer (in bytes) as a number. This is used to make your scripts more portable.
- `Process.codeSigningPolicy`: property containing the string `optional`


frida --attach-name main
⌥⌘1
./main (main)
⌘1
frida (python3.11)
⌘2
[Local::main]-> Process.mainModule
{
"base": "0x104e10000",
"name": "main",
"path": "/Users/sean/Developer/Talks/intrumentation-with-frida/example/main",
"size": 16384
}
[Local::main]->

Module

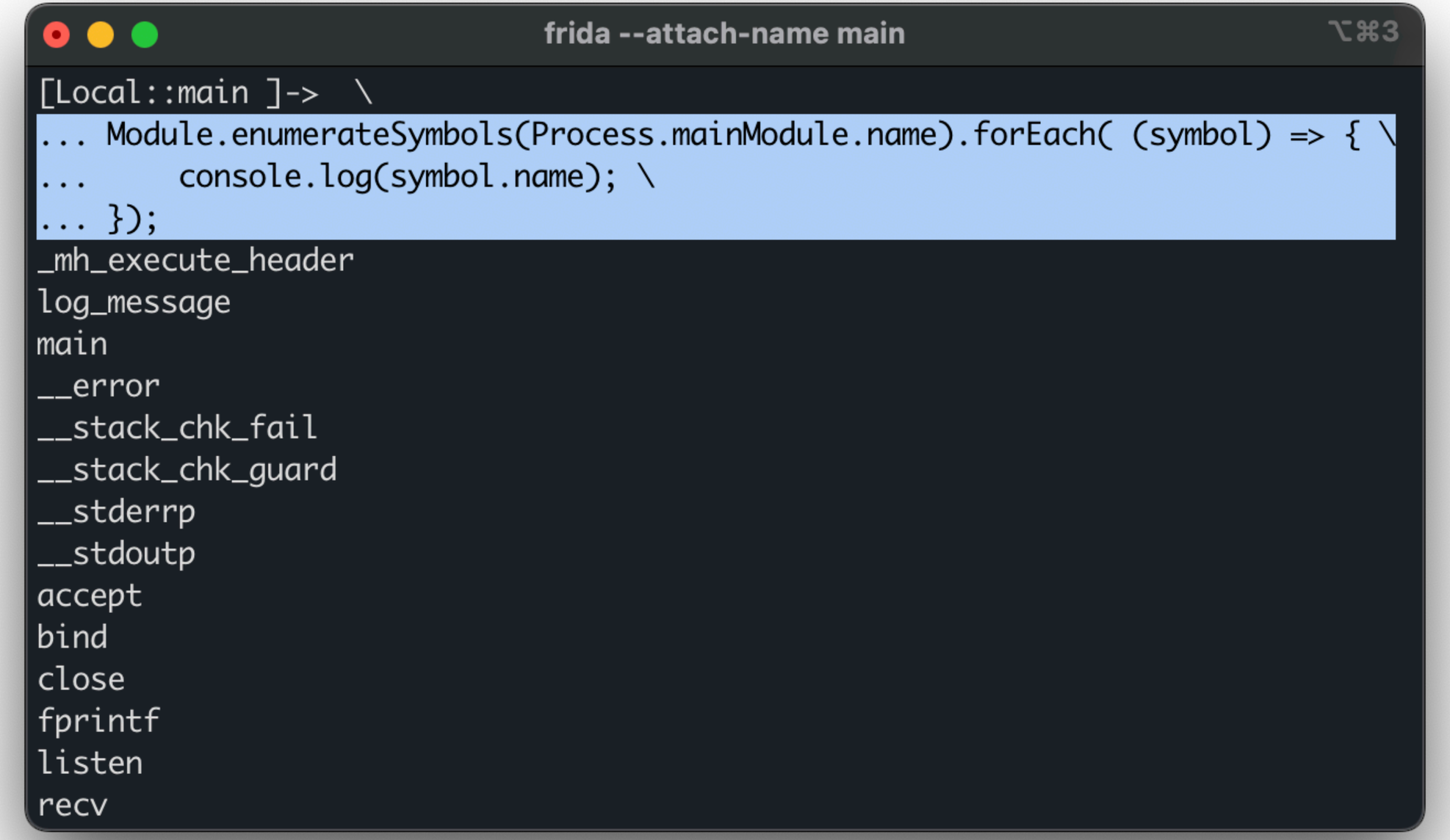
Objects returned by e.g. `Module.load()` and `Process.enumerateModules()`.

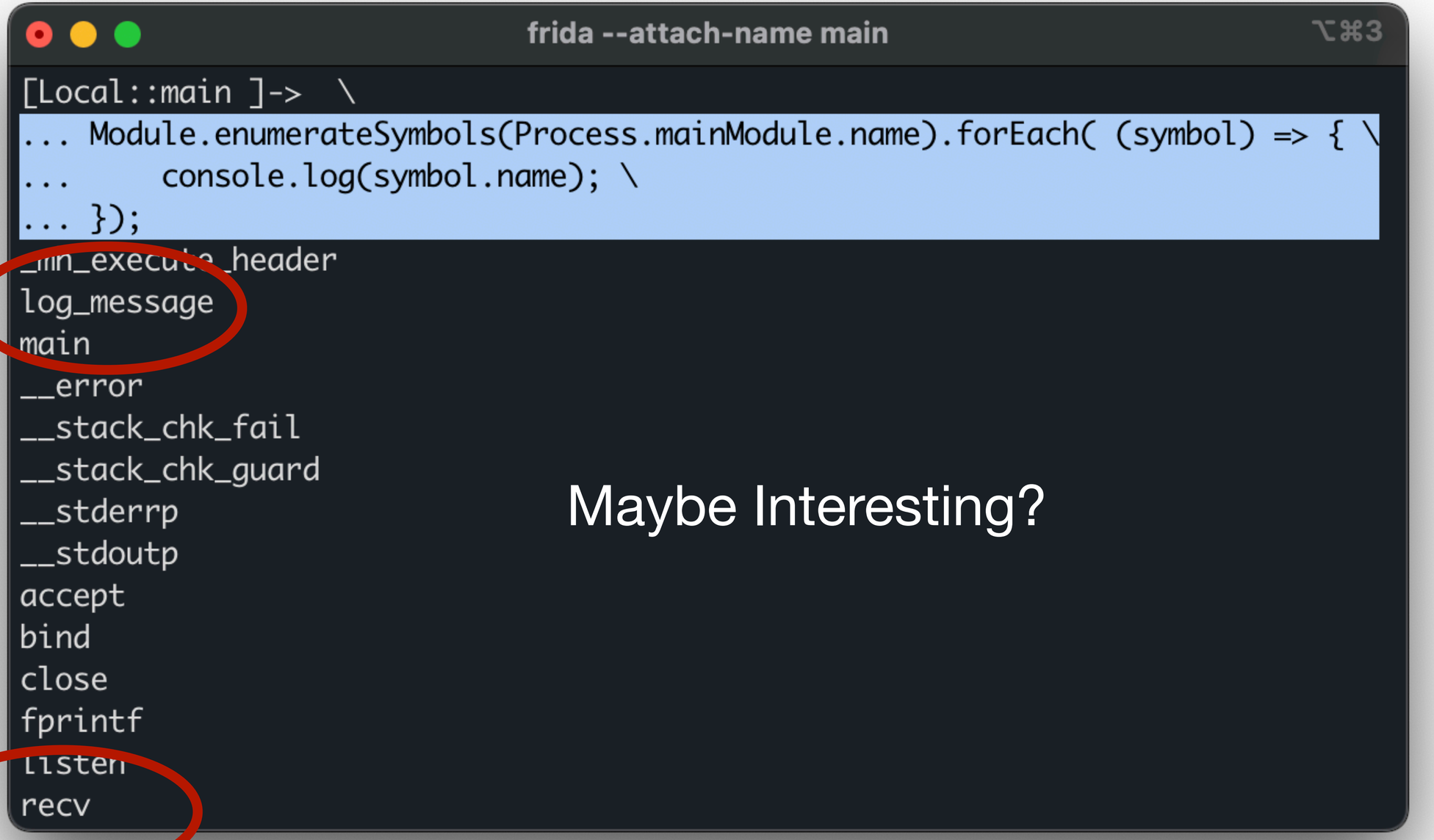
- `name`: canonical module name as a string
- `base`: base address as a `NativePointer`
- `size`: size in bytes
- `path`: full filesystem path as a string


frida --attach-name main
⌥⌘1
./main (main)
⌘1
frida (python3.11)
⌘2
[Local::main]-> Process.enumerateModules()
[
{
"base": "0x104e10000",
"name": "main",
"path": "/Users/sean/Developer/Talks/intrumentation-with-frida/example/main",
"size": 16384
},
{
"base": "0x1a45d9000",
"name": "libSystem.B.dylib",
"path": "/usr/lib/libSystem.B.dylib",
"size": 7332
},
{

- `enumerateSymbols()`: enumerates symbols of module, returning an array of objects containing the following properties:
    - `isGlobal`: boolean specifying whether symbol is globally visible
    - `type`: string specifying one of:


```
{
    "address": "0x0",
    "isGlobal": true,
    "name": "printf",
    "type": "undefined"
}
```

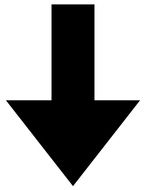

frida --attach-name main
⌥⌘3
[Local::main]-> \
... Module.enumerateSymbols(Process.mainModule.name).forEach((symbol) => { \
...   console.log(symbol.name); \
... });
_mh_execute_header
log_message
main
__error
__stack_chk_fail
__stack_chk_guard
__stderrp
__stdoutp
accept
bind
close
fprintf
listen
recv


frida --attach-name main
⌥⌘3
[Local::main]-> \
... Module.enumerateSymbols(Process.mainModule.name).forEach((symbol) => { \
...   console.log(symbol.name); \
... });
_mh_execute_header
log_message
main
__error
__stack_chk_fail
__stack_chk_guard
__stderrp
__stdoutp
accept
bind
close
fprintf
listen
recv
Maybe Interesting?

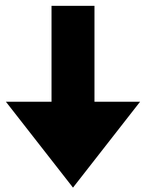
NativeFunction

- `new NativeFunction(address, returnType, argTypes[, abi])`: create a new NativeFunction to call the function at `address` (specified with a `NativePointer`), where `returnType` specifies the return type, and the `argTypes` array specifies the argument types. You may optionally also

```
void log_message(char * message)
```



```
const log_message = new NativeFunction(
    log_message_addr,
    "void", ["pointer"], {
});
```



```
log_message(ptr(0xdeadbeef));
```

Let’s Get Hooking

(also called interception)

What is a Hook?

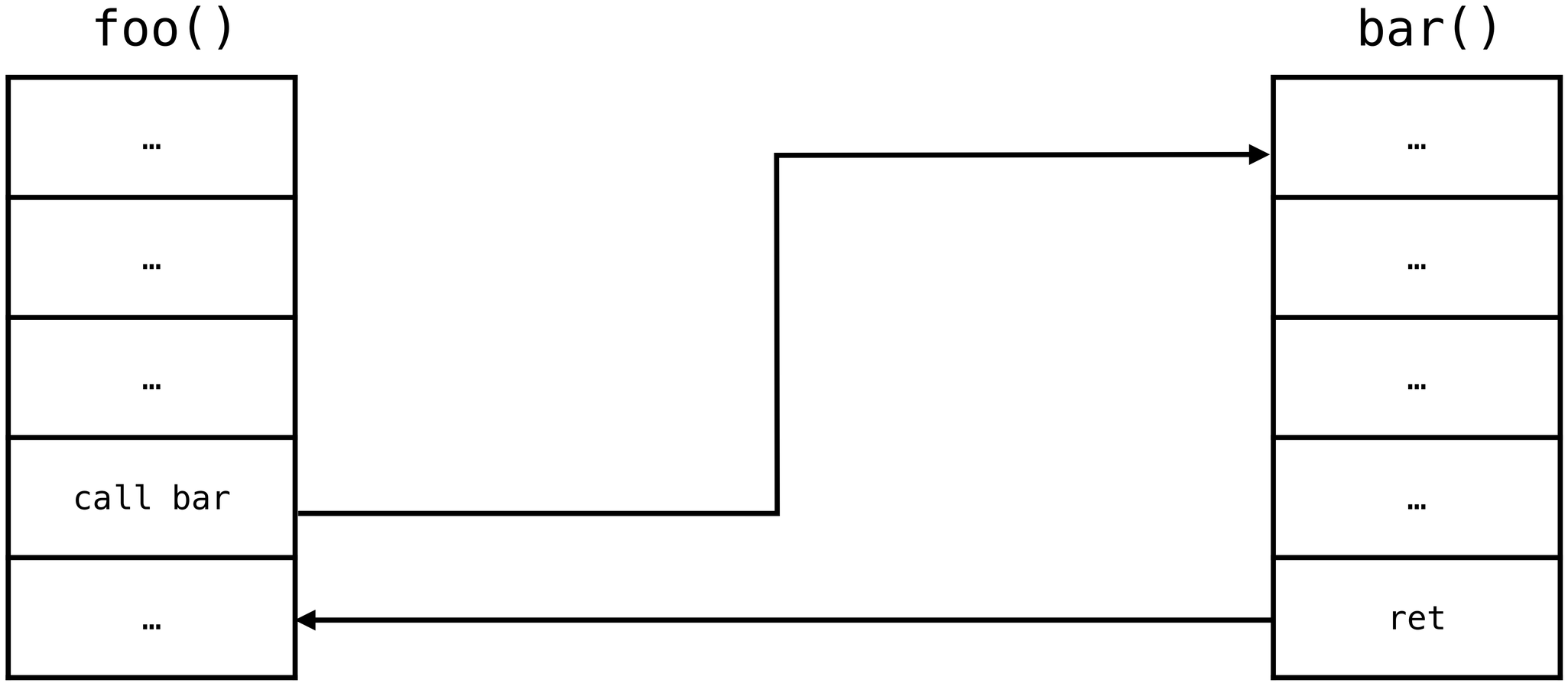


Diagram recreated from material by
Ole André Vadla Ravnås
University of Oslo, OSDC 2015

What is a Hook?

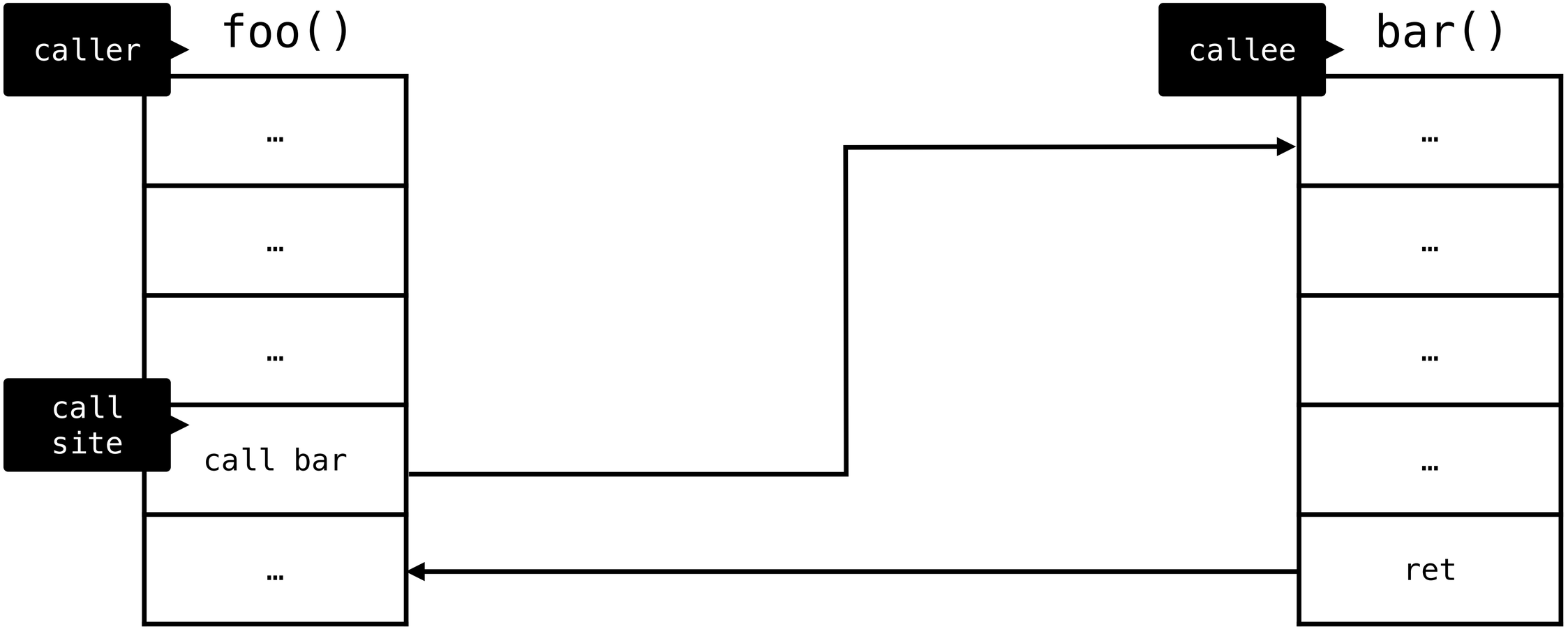


Diagram recreated from material by
Ole André Vadla Ravnås
University of Oslo, OSDC 2015

What is a Hook?

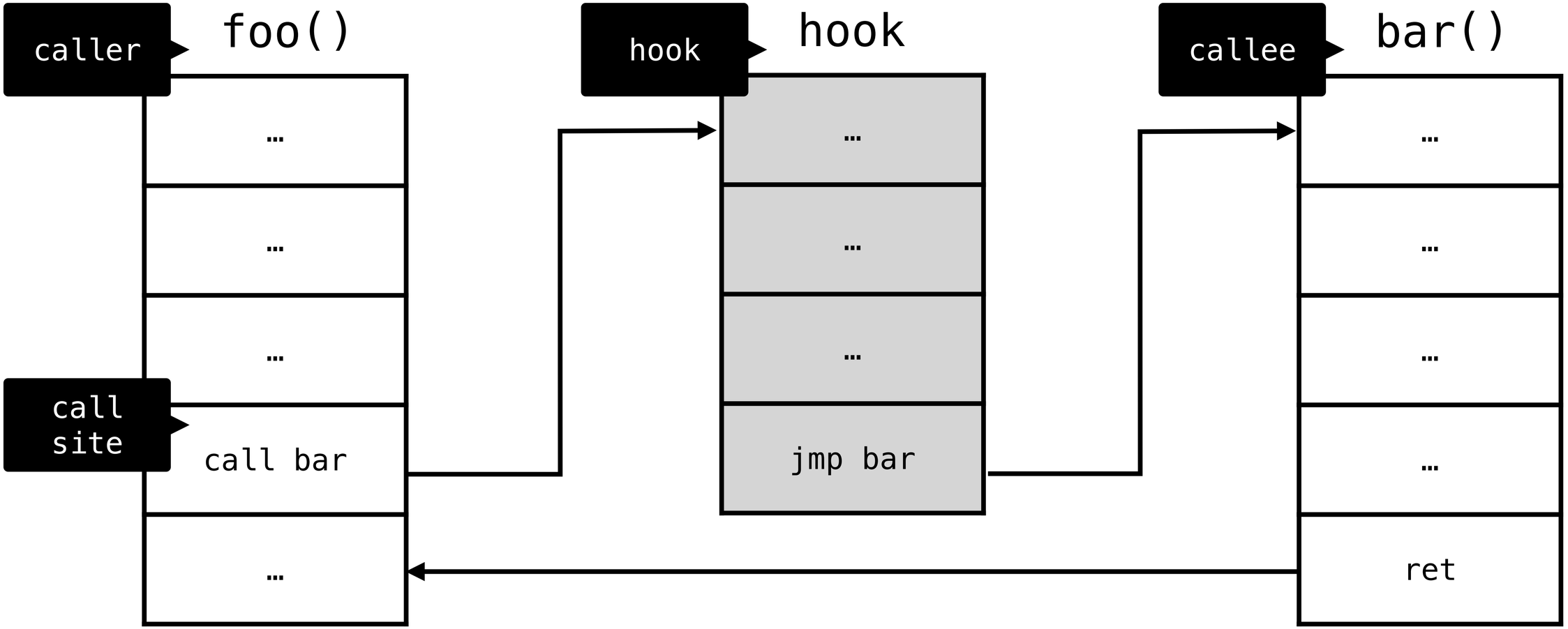
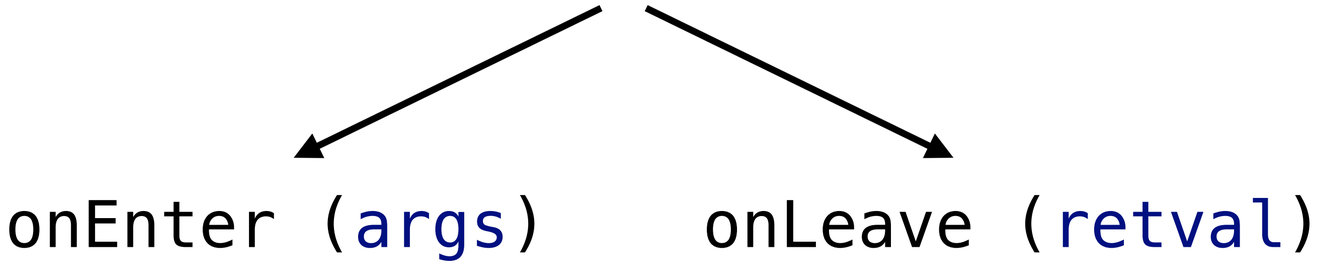


Diagram recreated from material by
Ole André Vadla Ravnås
University of Oslo, OSDC 2015

Game Plan

- Create a new TypeScript project
- Write a routine, using the `Interceptor` API to `attach` to `log_message`
  - Dump the argument passed to the function.

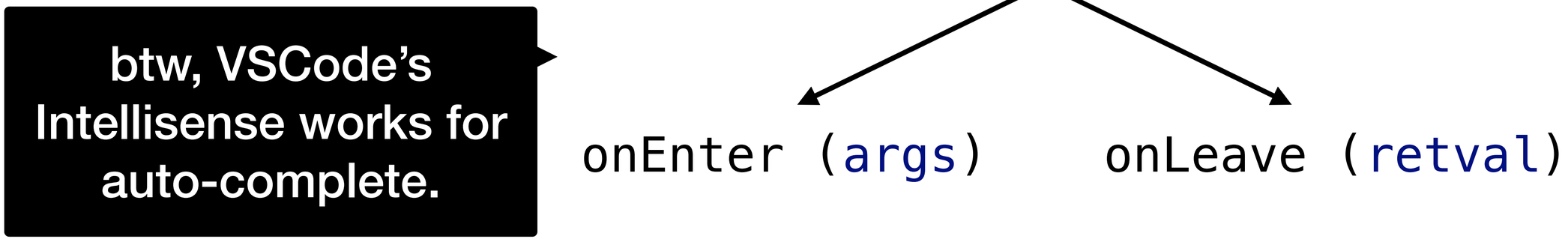
```
Interceptor.attach(target, callbacks[, data])
```



Game Plan

- Create a new TypeScript project
- Write a routine, using the `Interceptor` API to `attach` to `log_message`.
  - Dump the argument passed to the function.

```
Interceptor.attach(target, callbacks[, data])
```




```
npm install --save-dev @types/frida-gum
```

```
const m = Process.mainModule;
if (m) {
    console.log(`Main module: ${m.name}`);




}
```

Get the main module.

```
const m = Process.mainModule;
if (m) {
    console.log(`Main module: ${m.name}`);
    const log_message = m.findSymbolByName("log_message");
    if (log_message) {
        console.log(`Found log_message @ ${log_message}`);

    }
}
```

Find the log_message symbol.
This is an absolute address.

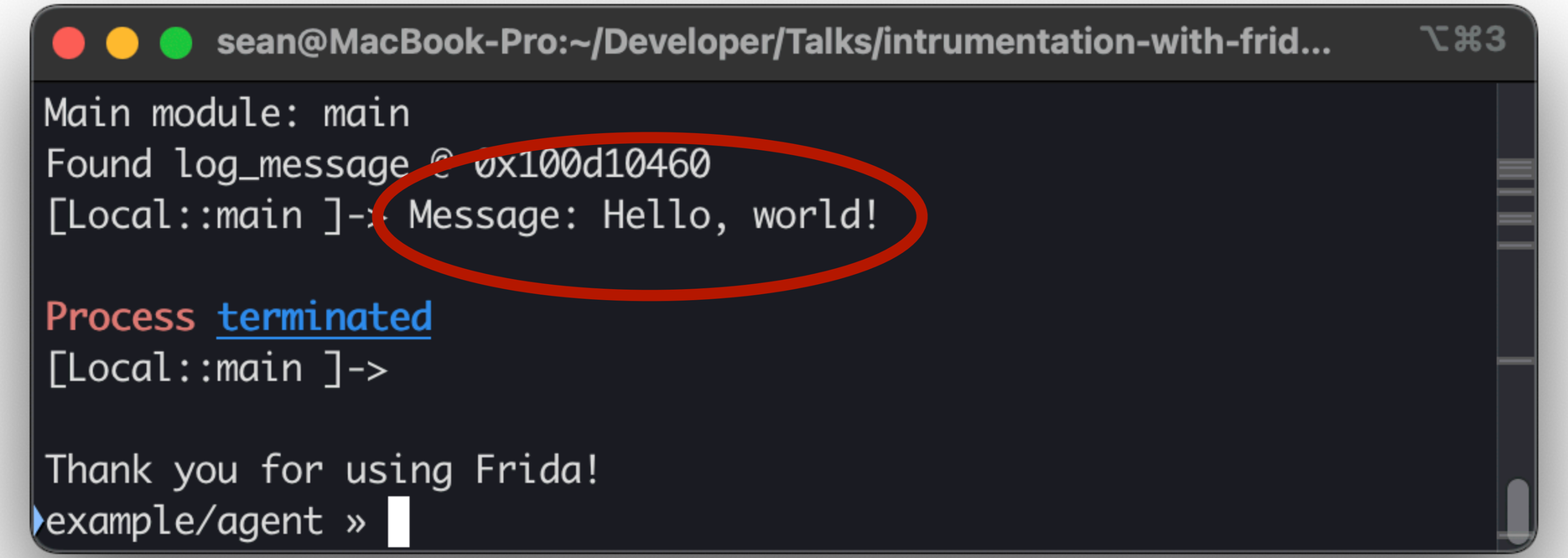
```
const m = Process.mainModule;
if (m) {
    console.log(`Main module: ${m.name}`);
    const log_message = m.findSymbolByName("log_message");
    if (log_message) {
        console.log(`Found log_message @ ${log_message}`);

        Interceptor.attach(log_message, {



        });
    }
}
```

Create an Interceptor. Attach to the address.

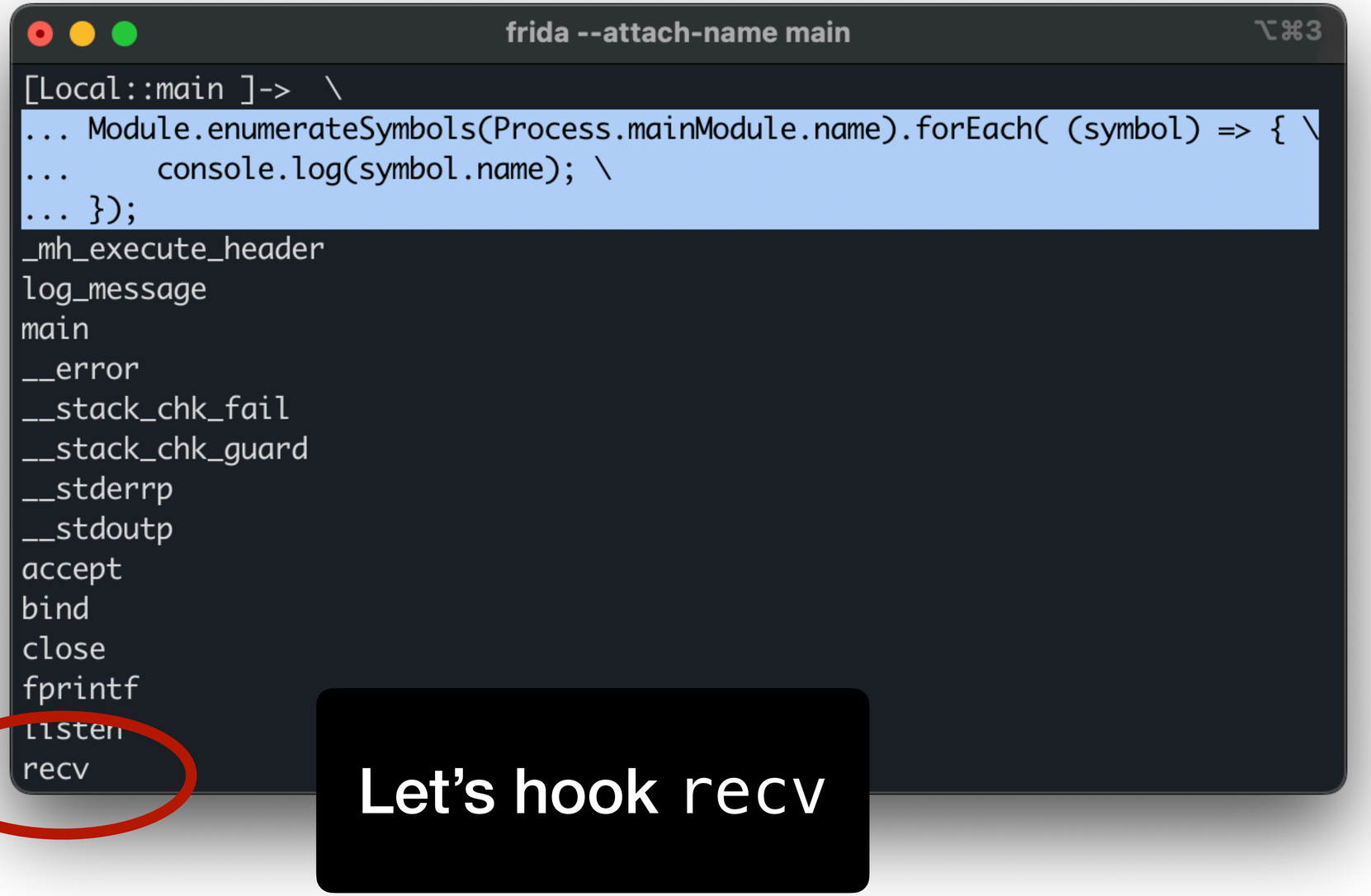
```
const m = Process.mainModule;
if (m) {
    console.log(`Main module: ${m.name}`);
    const log_message = m.findSymbolByName("log_message");
    if (log_message) {
        console.log(`Found log_message @ ${log_message}`);

        Interceptor.attach(log_message, {
            onEnter: function (args) {
                const message = args[0].readCString();
                console.log(`Message: ${message}`);
            },
        });
    }
}
```

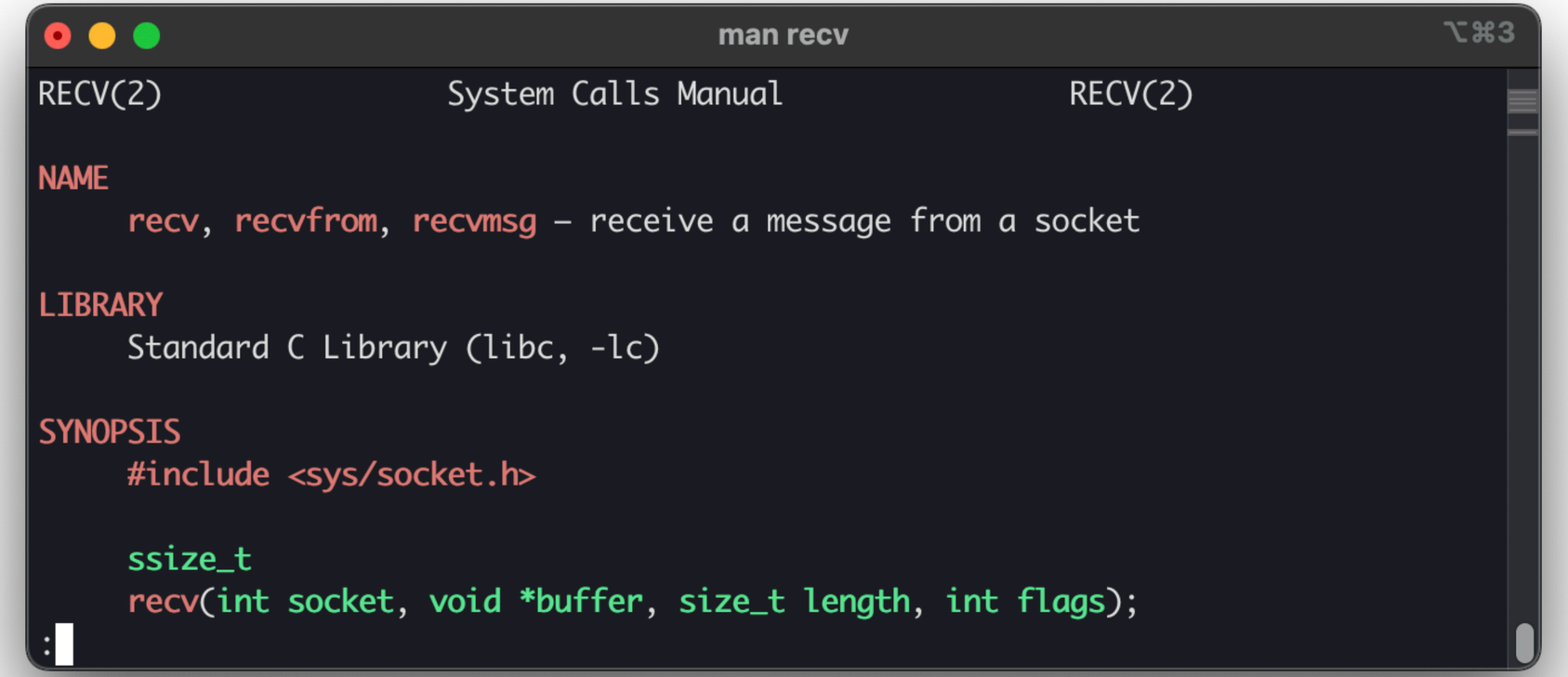
Create an anonymous function, onEnter.
Treat the first arg as char *.
Log it.

```
$ ./main
$ frida --attach-name main --load ./_agent.js
$ echo Hello, world | nc localhost 4444
```



Lame example, you say?


frida --attach-name main
⌥⌘3
[Local::main]-> \
... Module.enumerateSymbols(Process.mainModule.name).forEach((symbol) => { \
...   console.log(symbol.name); \
... });
_mh_execute_header
log_message
main
__error
__stack_chk_fail
__stack_chk_guard
__stderrp
__stdoutp
accept
bind
close
fprintf
listen
recv
Let's hook recv



The same approach won’t work.

Game Plan

- Find the exported symbol `recv`.
- Store the address passed as the `buffer` parameter.
- Log the contents after the function returns.

```
const recv = m.findExportByName("recv");
```

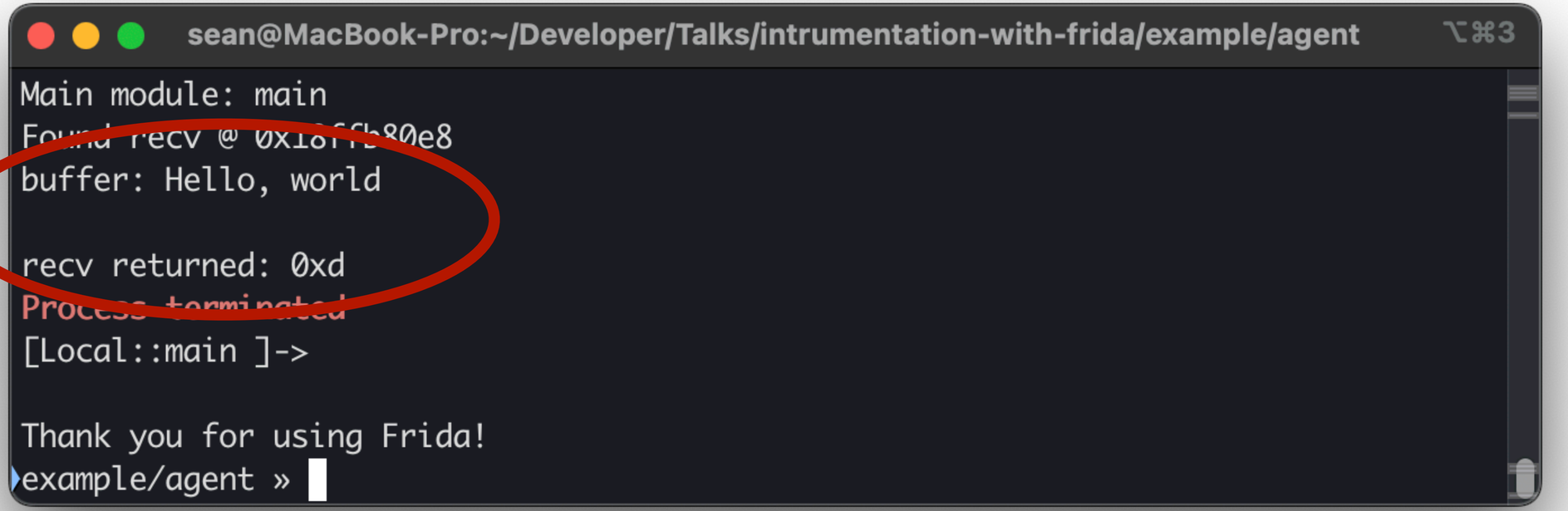
Find the exported symbol `recv`.

```
const recv = m.findExportByName("recv");

Interceptor.attach(recv, {
    onEnter: function (args) {
        this.buffer = args[1];
    },



});
```

Save the address passed as the buffer parameter.

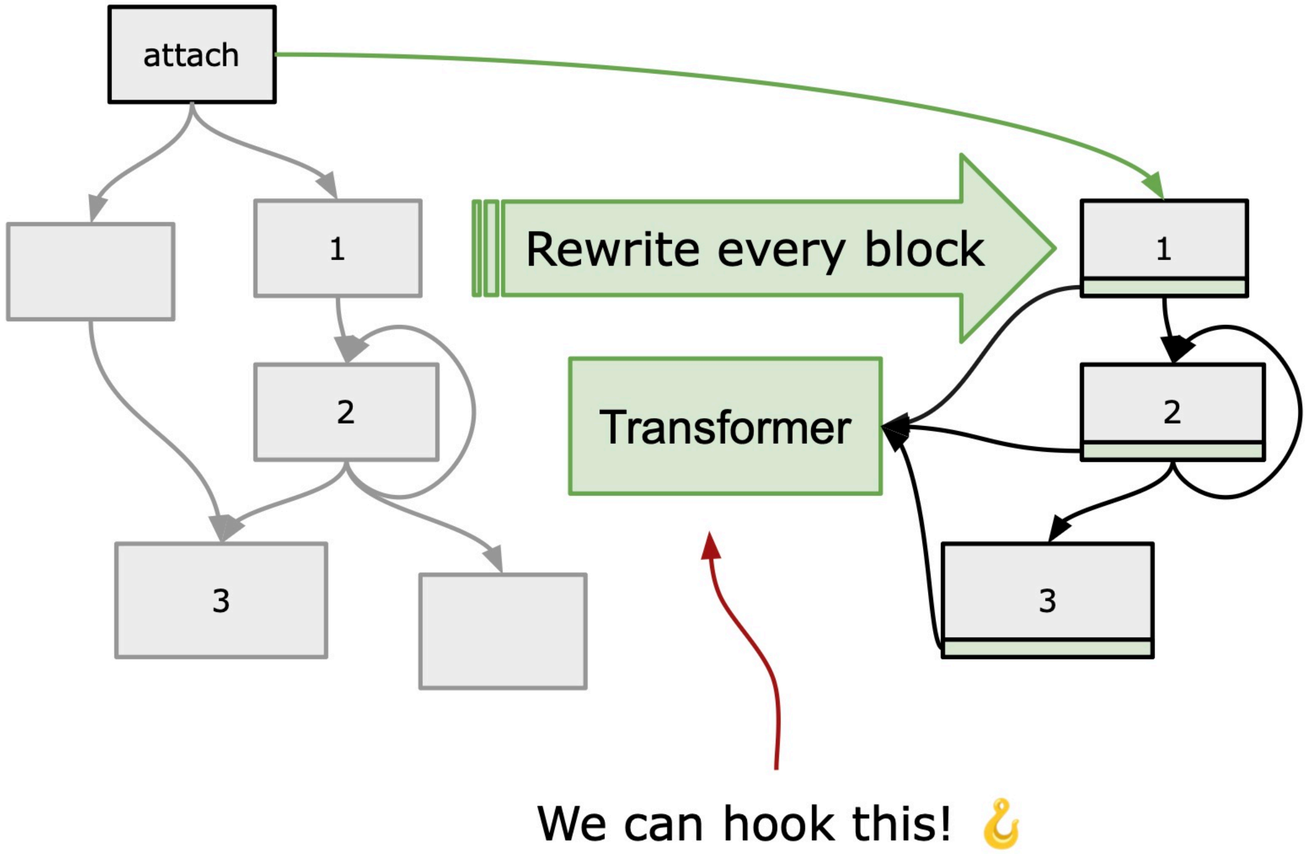
```
const recv = m.findExportByName("recv");

Interceptor.attach(recv, {
    onEnter: function (args) {
        this.buffer = args[1];
    },
    onLeave: function (retval) {
        console.log(`buffer: ${this.buffer.readCString()}`);
        console.log(`recv returned: ${retval}`);
    }
});
```

After the function returns, read the buffer.
Log the return value because why not.


sean@MacBook-Pro:~/Developer/Talks/intrumentation-with-frida/example/agent
⌥⌘3
Main module: main
buffer: Hello, world
recv returned: 0xd
[Local::main]->
Thank you for using Frida!
example/agent »

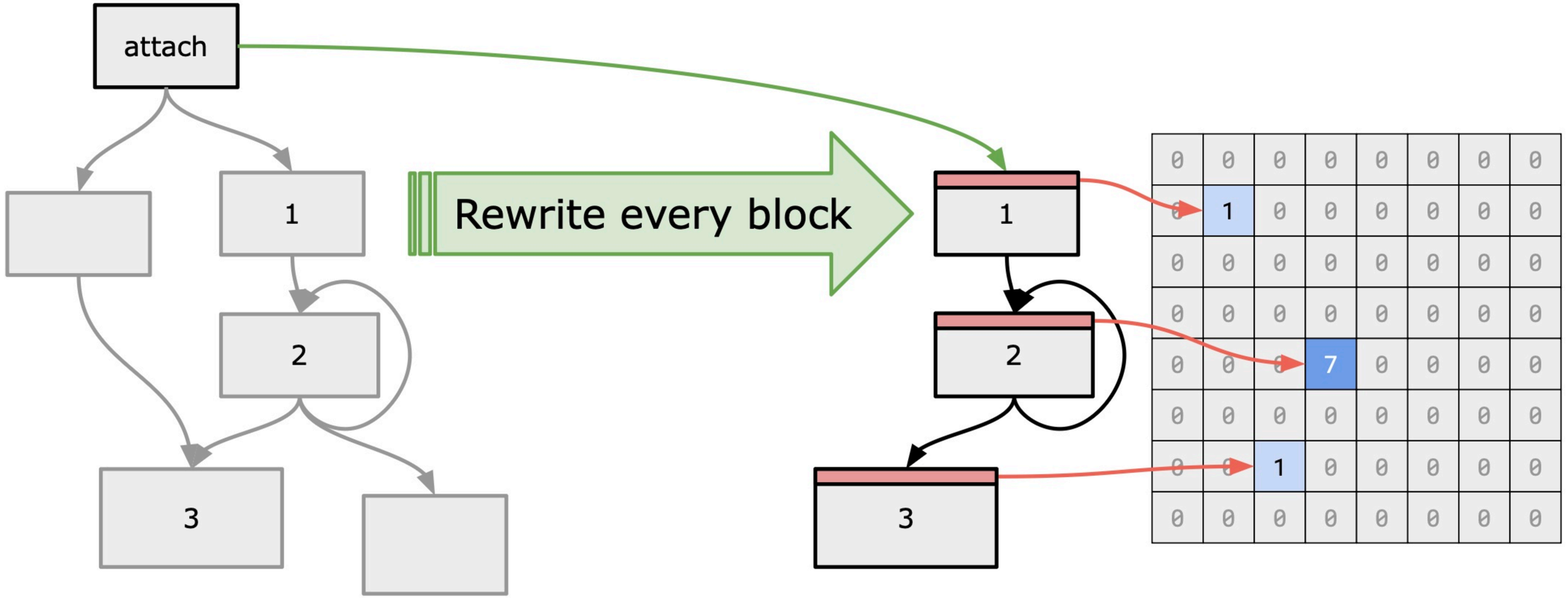
Frida Stalker

Frida Stalker Code Coverage Basics



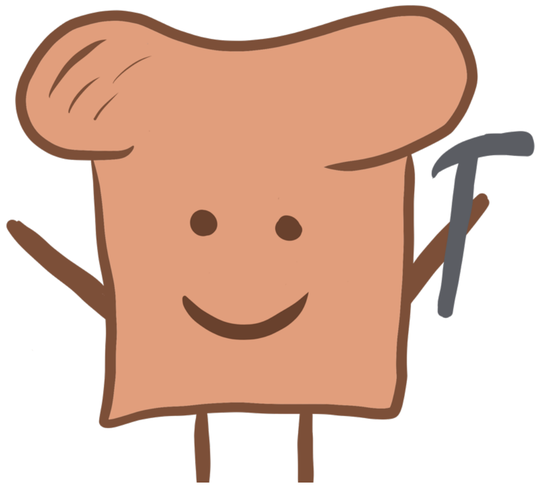
Graphic courtesy of Jiska Classen & Alexander Heinrich, *Practical iOS App, User-, Kernel-Space and Firmware Reverse Engineering*, Objective by the Sea, Maui, HI, 2024

Frida Stalker Code Coverage Basics



Graphic courtesy of Jiska Classen & Alexander Heinrich, *Practical iOS App, User-, Kernel-Space and Firmware Reverse Engineering*, Objective by the Sea, Maui, HI, 2024

Fuzzing with fpicker



fpicker

- Frida-based coverage-guided fuzzer
- Utilizes Frida Stalker
- AFL++ proxy mode for blackbox binaries
- Started off as a Master’s Thesis
- Great way to get a very quick, if maybe slower, fuzzing campaign going before moving to something like Jackalope (iOS/macOS)

```
class TestFuzzer extends Fuzzer {
    constructor() {...}
    prepare() {...}
    fuzz() {...}
}
```

```
class TestFuzzer extends Fuzzer {
    constructor() {

    }
}
```

Subclass the Fuzzer class that fpicker creates for you.

```
class TestFuzzer extends Fuzzer {
    constructor() {
        const log_message_addr = Module.getExportByName(null,
                                                        "log_message");









    }
}
```

Get the address of the function that we want to fuzz.

```
class TestFuzzer extends Fuzzer {
    constructor() {
        const log_message_addr = Module.getExportByName(null,
                                                        "log_message");
        const log_message = new NativeFunction(
            log_message_addr,
            "void", ["pointer"], {
        });









    }
}
```

Create a new NativeFunction. This informs Frida what the signature of the function is.

```
class TestFuzzer extends Fuzzer {
    constructor() {
        const log_message_addr = Module.getExportByName(null,
                                                        "log_message");
        const log_message = new NativeFunction(
            log_message_addr,
            "void", ["pointer"], {
        });


    }
}
```

The implementation of the function is found at the address of log_message_addr

return

arguments:
pointer to a C string.

```
class TestFuzzer extends Fuzzer {
    constructor() {
        const log_message_addr = Module.getExportByName(null,
                                                        "log_message");
        const log_message = new NativeFunction(
            log_message_addr,
            "void", ["pointer"], {
        });

        // The constructor needs:
        //       - the module name
        //       - the address of the targeted function
        //       - the NativeFunction of the targeted function
        super("main", log_message_addr, log_message);
    }
}
```

```
class TestFuzzer extends Fuzzer {
    // If we needed to set up some global state.
    // Like initialize the bluetoothd daemon...
    prepare() {
    }
}
```

```
class TestFuzzer extends Fuzzer {

    fuzz(payload, len) {

    }

}
```

Input from the corpus or mutation and length of the input.

```
class TestFuzzer extends Fuzzer {

    fuzz(payload, len) {
        this.debug_log(payload, len);
        this.target_function(payload);
    }

}
```

Since we have a dumb simple example, we don’t need to do anything with the input. Just pass it to the function we want to fuzz. Don’t even need the length.

```
frida-compile \
        log-message-fuzzer.js \
        log-message-harness.js
```

```
./fpicker \
        --fuzzer-mode active
        -e attach
        -p main
        -o ./out
        -i ./in
        -f ./log-message-harness.js
        --standalone-mutator cmd
        --mutator-command "radamsa"
```

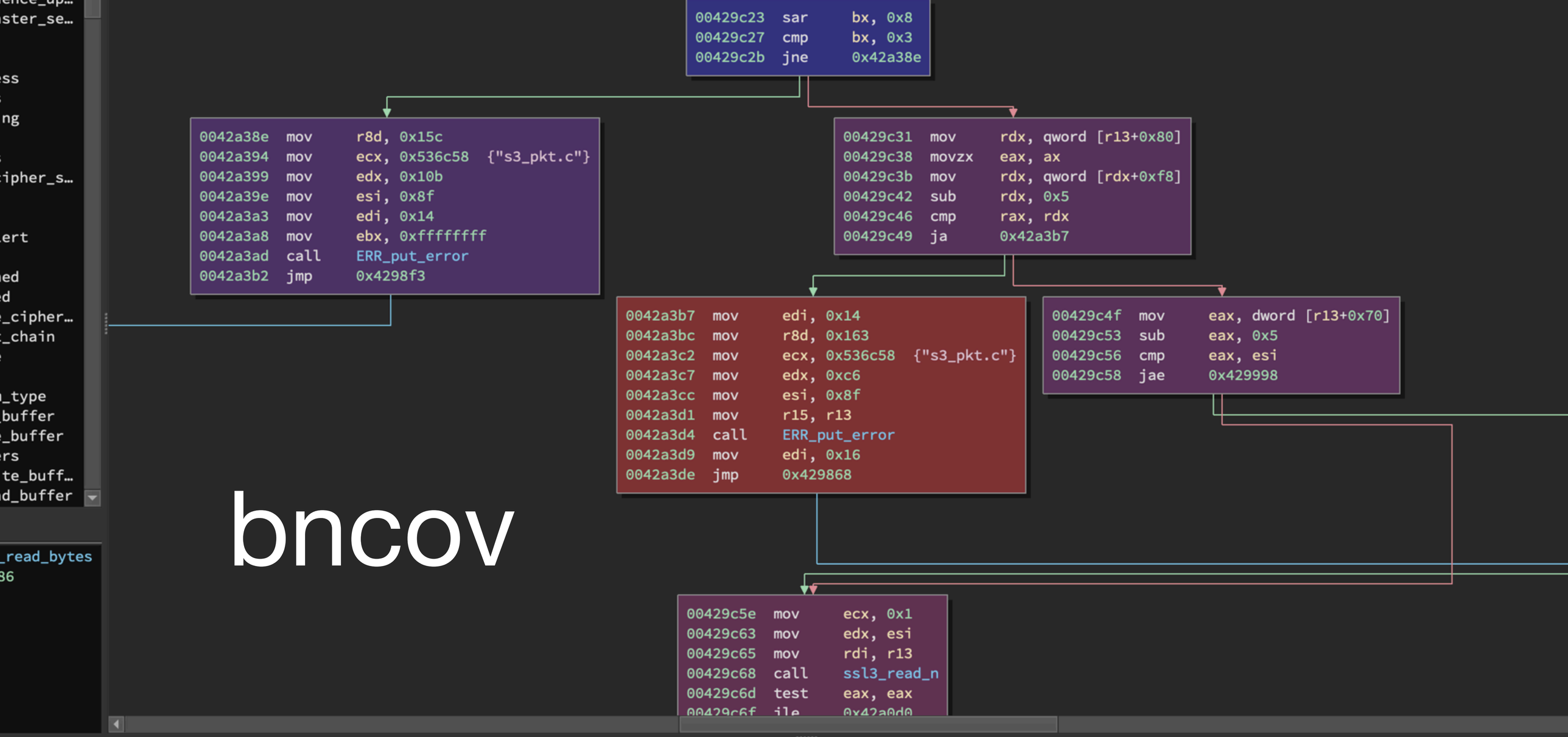
Note: we're attaching, the process has to already be running.

We got a crash!

```
%n$!!%n$(xcalc)%p$'%n`xcalc`%n%n$`NaN$PATH;xcalc\x347a%n$!!%n$
(xcalc)%p$'%n`xcalc`%n%n$`NaN$PATH;xcalc\x0a
```

```
$ ./main
[1]    66386 abort      ./main
```

```
$ cat payload | nc localhost 4444
```


00429c23 sar bx, 0x8
00429c27 cmp bx, 0x3
00429c2b jne 0x42a38e
0042a38e mov r8d, 0x15c
0042a394 mov ecx, 0x536c58 {"s3_pkt.c"}
0042a399 mov edx, 0x10b
0042a39e mov esi, 0x8f
0042a3a3 mov edi, 0x14
0042a3a8 mov ebx, 0xffffffff
0042a3ad call ERR_put_error
0042a3b2 jmp 0x4298f3
00429c31 mov rdx, qword [r13+0x80]
00429c38 movzx eax, ax
00429c3b mov rdx, qword [rdx+0xf8]
00429c42 sub rdx, 0x5
00429c46 cmp rax, rdx
00429c49 ja 0x42a3b7
0042a3b7 mov edi, 0x14
0042a3bc mov r8d, 0x163
0042a3c2 mov ecx, 0x536c58 {"s3_pkt.c"}
0042a3c7 mov edx, 0xc6
0042a3cc mov esi, 0x8f
0042a3d1 mov r15, r13
0042a3d4 call ERR_put_error
0042a3d9 mov edi, 0x16
0042a3de jmp 0x429868
00429c4f mov eax, dword [r13+0x70]
00429c53 sub eax, 0x5
00429c56 cmp eax, esi
00429c58 jae 0x429998
00429c5e mov ecx, 0x1
00429c63 mov edx, esi
00429c65 mov rdi, r13
00429c68 call ssl3_read_n
00429c6d test eax, eax
bncov

How I Use Frida, Daily

Start with a Trace

- Automatically creates Frida Interceptor hooks, dumps arguments.
- Works for Objective-C, Swift, regular functions, etc.
- Then just interact with the device as normal. Does it get hit?

```
frida-trace -await com.apple.apsd -include-objc-method "-[APSProtocolParser *]”

killall apsd
```

Find Anything Interesting?

- Improve the Interceptor
- Modify the arguments
- Improve Binary Ninja markup

```
-[APSProtocolParser copyFilterMessageWithEnabledHashes:
                    ignoredHashes:
                    opportunisticHashes:
                    nonWakingHashes:
                    pausedHashes:
                    token:]
```

Create the Objective-C Object

```
const cls = new ObjC.Object.APSProtocolParser;
console.log(cls.$ivars);
console.log(cls.$ownMethods);

const sharedInstance: ObjC.Object = cls["- sharedInstance"];
```

Attach to an iPhone or Android

```
frida --usb
```

Liked This Presentation?

Maybe you'd like more

- Bypassing iOS anti-jailbreak heuristics with Frida
- *OS security protections
  - Entitlements, SIP, AMFI, PAC, sandboxing, code signatures, notarization
- Apple Push Notifications (APNs)
- Reversing Objective-C (and some Swift) with Binary Ninja
- Improving C++ markup in Binary Ninja

References

- Ole André Vadla Ravnås, *Frida: The engineering behind the reverse-engineering*, University of Oslo, OSDC 2015.
- Jiska Classen & Alexander Heinrich, *Practical iOS App, User-, Kernel-Space and Firmware*, Reverse Engineering, Objective by the Sea, Maui, HI, 2024