

基于 FRIIDA 的全平台逆向分析

caisi.zz@alipay.com

关于

- 蚂蚁金服光年实验室高级安全工程师
- 从事多种平台客户端漏洞攻防研究
- BlackHat, XDEF 等国内外会议演讲者
- 知名 iOS App 审计工具 passionfruit 开发者
- ~~frida~~ 非官方布道师



提纲

- frida 上手
- 自动化测试
- Javascript 进阶
- 操纵本地代码、Java 和 Objective C 运行时
- 多平台案例分析
- frida 高级编程技巧

本课程所有示例代码请访问

<https://github.com/ChiChou/gossip-summer-school-2018>



什么是 FRIDA

什么是 frida

- <https://www.frida.re/> *Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers*
- 使用 javascript 脚本注入进程，执行二进制插桩。可以与调试器共存
- 在脚本和原生函数中实现了双向的 bridge。既可 hook 函数调用并修改参数，也可使用脚本调用原生函数实现复杂功能
- 除本地代码外，内置对 Java 和 Objective C 运行时的支持
- 跨平台支持 Windows, macOS, GNU/Linux, iOS, Android 和 QNX(*)

常见运行时插桩

- 硬件支持（如 Intel Pin）
- 特定运行时
- 调试器
 - 硬件断点
 - 软件断点
- Method Swizzling
- ART 虚拟机
- 函数指针（如导入表，Objective C isa 指针）重绑定
- Inline hook

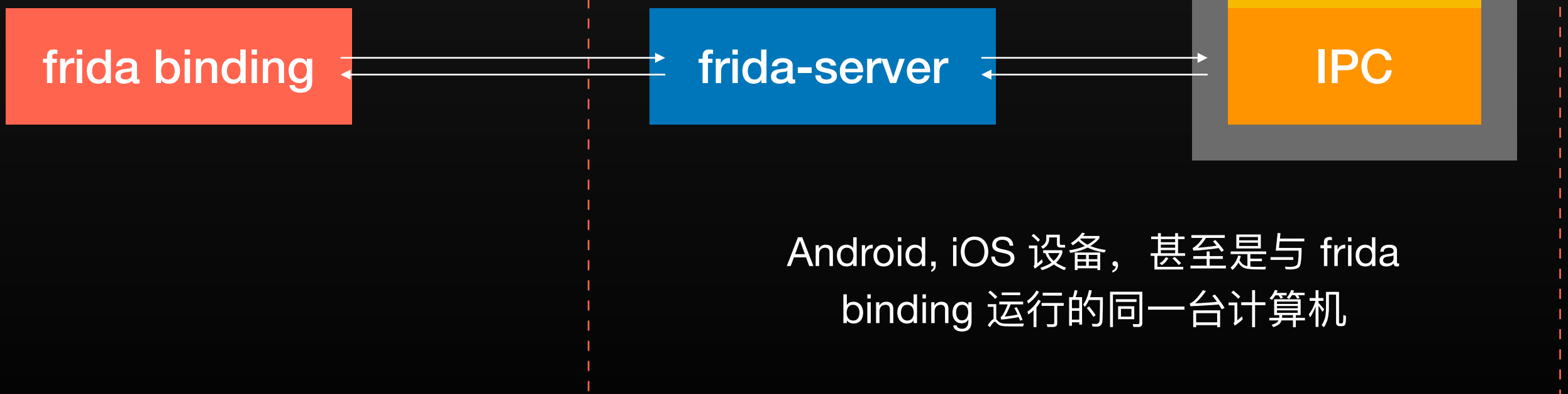
frida 插桩实现

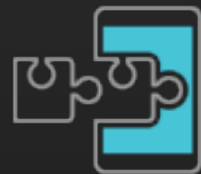
- 主要是 inline hook
 - 在 iOS 上禁止 RWX 内存页，使用修改临时文件再 mmap 的方式
- Objective C (github: frida/frida-objc)
 - implementation setter: Method Swizzling
 - Interceptor.attach: 直接对 selector 指向的函数指针进行 inline hook
- Android ART / Dalvik (github: frida/frida-java)
 - 动态生成 JNI stub, 修改结构体中的函数指针

架构

frida 提供多语言绑定:

- **frida-python** (含 cli)
- **frida-node**
- frida-qml
- frida-swift
- frida-clr





FRIDA

- 需要 root 和刷机 (*注)
 - 热更新支持较差，每次修改代码需要重启设备 (*注)
 - 以上问题可用 VirtualXposed 解决
 - 原生 Java 语法，开发体验顺畅
 - 不内置原生函数（如 JNI）的修改，需结合其他框架实现
 - 适合开发生产环境下使用的插件
- 无需 root，重打包植入 gadget 库仍然可用
 - 支持 JNI 函数的 hook 和调用
 - 需要转译 Java 到 js
 - 更新无需重启，适合快速迭代的脚本开发

cycrypt

- 内置 Substrate 引擎，支持 hook
- 模仿 Objective C 和 Javascript 的混合语法，体验非常顺滑
- 甚至支持直接使用 RTLD_DFAULT 这样的常量
- iOS 11 后更新缓慢，有（可解决的）兼容性问题
- 除语法之外，两者 REPL 的使用体验非常相似

FRIDA

- 由于 iOS 强制代码签名限制，由 v8 改为 duktape 解释器。以 ECMAScript5 为主，只支持极少的 ES6 语法
- 移植现有 Objective C 代码需要手写翻译到 js，实现复杂功能非常别扭
- 宏定义常量需要自行查找头文件获得实际的值
- 支持到最新的 Electra 越狱（iOS 11），开箱即用

Why Javascript

- 优势：世界上最流行的脚本语言
 - 学习成本简单
 - 庞大的生态系统
- 缺点：作为插桩 DSL 局限性明显
 - 使用弱类型语言操作强类型语言（C、Java、Objective C）
 - 实现同样功能，语法比原生代码冗长
 - 操作二进制结构体非常麻烦

安装部署

- 桌面端命令行工具: `pip install frida-tools` (可能需使用代理)
- Android: <https://github.com/frida/frida/releases>
 - 有 root: 下载 `frida-server`, adb 推入后以 root 执行。默认监听 TCP 27042 端口, 可通过 `adb forward` 转发到计算机上; 或使用 `frida-server -l 0.0.0.0` 监听在局域网
 - 无 root: 重打包 apk 植入 `FridaGadget.so`
 - 反编译, 修改 smali 添加 `System.loadLibrary` 调用
 - 或使用 LIEF 等工具, 附加依赖项到已有的 elf:
https://lief.quarkslab.com/doc/latest/tutorials/09_frida_lief.html
 - 需要确保 `AndroidManifest.xml` 中开启网络访问权限

安装部署

- iOS: <https://github.com/frida/frida/releases>
- 已越狱: Cydia 市场中添加源 <https://build.frida.re>, 安装 Frida (或 Frida for 32-bit devices)
- 未越狱:
 - 对于具有原始工程的项目: 将 FridaGadget.dylib 添加到链接库。需要对 FridaGadget 手动添加开发者签名, 以及关闭 Build Options 中的 Enable Bitcode 选项
 - AppStore 的安装包具有加密壳。需从已越狱设备, 或第三方市场中获取解密后的 ipa 安装包, 然后使用 MonkeyDev 集成: 非越狱App集成

安装部署

- 篇幅限制，完整的步骤参考文档
 - <https://www.frida.re/docs/ios/>
 - <https://www.frida.re/docs/android/>
- macOS 默认启用 SIP，无法附加系统自带进程。如有需求需关闭（具有一定安全风险）
- 未越狱 / root 环境下的 frida 会有部分功能受限
- macOS, Linux 和 Windows 既可直接安装 frida-tools 测试本机进程，也可使用 frida-server 通过 TCP 协议远程测试
- 不推荐使用局域网方式连接 Android / iOS 设备，稳定性不如 USB，且任何人都可以连接设备远程执行任意代码 ✨

cli 工具

frida	类似 python 等脚本解释器的 REPL
frida-ps	列出可附加的进程 / App 列表
frida-trace	根据 glob 匹配符号并自动生成 hook 代码框架 修改 __handlers__ 中的脚本后会自动重新载入
frida-ls-devices	列出可用的设备
frida-kill	杀进程
frida-discover	记录一段时间内各线程调用的函数和符号名

→ /tmp frida-ls-devices

Id	Type	Name
-----	-----	-----
local	local	Local System
***	usb	iPhone
tcp	remote	Local TCP

→ /tmp frida-ps -U

PID	Name
-----	-----
9367	InCallService
13295	Mail
13077	AppleIDAuthAgent
13300	AssetCacheLocatorService
13792	CacheDeleteAppContainerCaches
13012	CloudKeychainProxy
4128	CommCenter

→ /tmp frida-trace -i open iTunes

Instrumenting functions...

open: Loaded handler at "/private/tmp/__handlers__/libsystem_kernel.dylib/open.js"

Started tracing 1 function. Press Ctrl+C to stop.

```
      /* TID 0x12e07 */
43527 ms  open(path="/Users/codecolorist/Music/iTunes/iTunes Library.itl", oflag=0x26)
43527 ms  open(path="/Users/codecolorist/Music/iTunes/Temp File.tmp", oflag=0xa00)
43528 ms  open(path="/Users/codecolorist/Music/iTunes/Temp File.tmp", oflag=0x26)
      /* TID 0x11d17 */
43560 ms  open(path="/Users/codecolorist/Music/iTunes/iTunes Library Genius.itdb-journal",
oflag=0x1000202)
43562 ms  open(path="/Users/codecolorist/Music/iTunes/iTunes Library Extras.itdb-journal",
oflag=0x1000202)
```

REPL

➔ ~ frida Calculator

```
-----
/ _ |   Frida 11.0.13 - A world-class dynamic instrumentation toolkit
| (_| |
> _ |   Commands:
/_/ |_ |   help      -> Displays the help system
. . . .   object?    -> Display information about 'object'
. . . .   exit/quit  -> Exit
. . . .
. . . .   More info at http://www.frida.re/docs/home/
```

```
[Local::Calculator]-> const POINTER_WIDTH = Process.pointerSize * 2;
function formatPointer(p) {
  const hex = p.toString().slice(2);
  const padding = hex.length < POINTER_WIDTH ?
    '0'.repeat(POINTER_WIDTH - hex.length) : '';
  return '0x' + padding + hex;
}
Process.enumerateRangesSync('').forEach(function(range) {
  console.log(
    range.protection,
    formatPointer(range.base), '-', formatPointer(range.base.add(range.size)),
    range.file ? range.file.path : '');
});
```

```
r-x 0x00000000106d83000 - 0x00000000106da1000 /Applications/Calculator.app/Contents/MacOS/Calculator
rw- 0x00000000106da1000 - 0x00000000106dae000 /Applications/Calculator.app/Contents/MacOS/Calculator
r-- 0x00000000106dae000 - 0x00000000106db4000 /Applications/Calculator.app/Contents/MacOS/Calculator
rw- 0x00000000106db4000 - 0x00000000106db6000
r-- 0x00000000106db6000 - 0x00000000106db7000
rw- 0x00000000106db7000 - 0x00000000106db8000
```

REPL

- 支持 tab 自动补全
- 特殊命令
 - `%load` / `%unload`: 载入 / 卸载文件中的 js
 - `%reload`: 修改外部 js 之后重新载入, 且重置之前的 hook
 - `%resume`: 继续执行以 spawn 模式启动的进程
- 使用 `quit` / `exit` 或 Ctrl + D 退出

语言绑定

- 官方提供的 binding:
 - frida-gum: 没有 js 引擎, 单纯的 hook 框架
 - frida-core: 具有 js 引擎的 C/C++ binding
 - 其他语言: frida-python (python) 、 frida-node (node.js) 、 frida-qml (Qt) 、 frida-swift (swift)
- 缺少文档, 建议直接参考源码
 - frida-python: <https://github.com/frida/frida-python/blob/master/src/frida/core.py>
 - JavaScript: <https://github.com/frida/frida-gum/blob/master/bindings/gumjs/types/frida-gum/frida-gum.d.ts>

自动化测试任务

以 python binding 为例



设备 api

获得设备

```
all_devies = frida.enumerate_devices()
local = frida.get_local_device()
usb = frida.get_usb_device()
remote = frida.get_device_manager().add_remote_device(ip)
```

设备事件处理

```
device_manager = frida.get_device_manager()
device_manager.on('changed', on_changed) # listen
device_manager.off('changed', on_changed) # remove listener
```

监听设备插拔

```
device_manager.on('add', on_changed)
device_manager.on('changed', on_changed)
device_manager.on('remove', on_removed)
```

进程管理

```
pid = device.spawn('com.apple.mobilesafari')
device.resume(pid)
device.kill(pid)
```

App 信息

```
device.enumerate_applications()
```

附加进程 / App

- 启动新的实例：`device.spawn('path or bundle id')`
 - 可指定启动参数
 - 支持在进程初始化之前执行一些操作
 - iOS 上如果已经 App 运行（包括后台休眠）会导致失败
- 附加到现有进程：`device.attach(pid)`
 - 可能会错过 hook 时机
 - `spawn` 在移动设备容易出现不稳定现象，可使用 `attach` 模式

spawn 选项

frida >= 11.0 支持的 spawn 参数

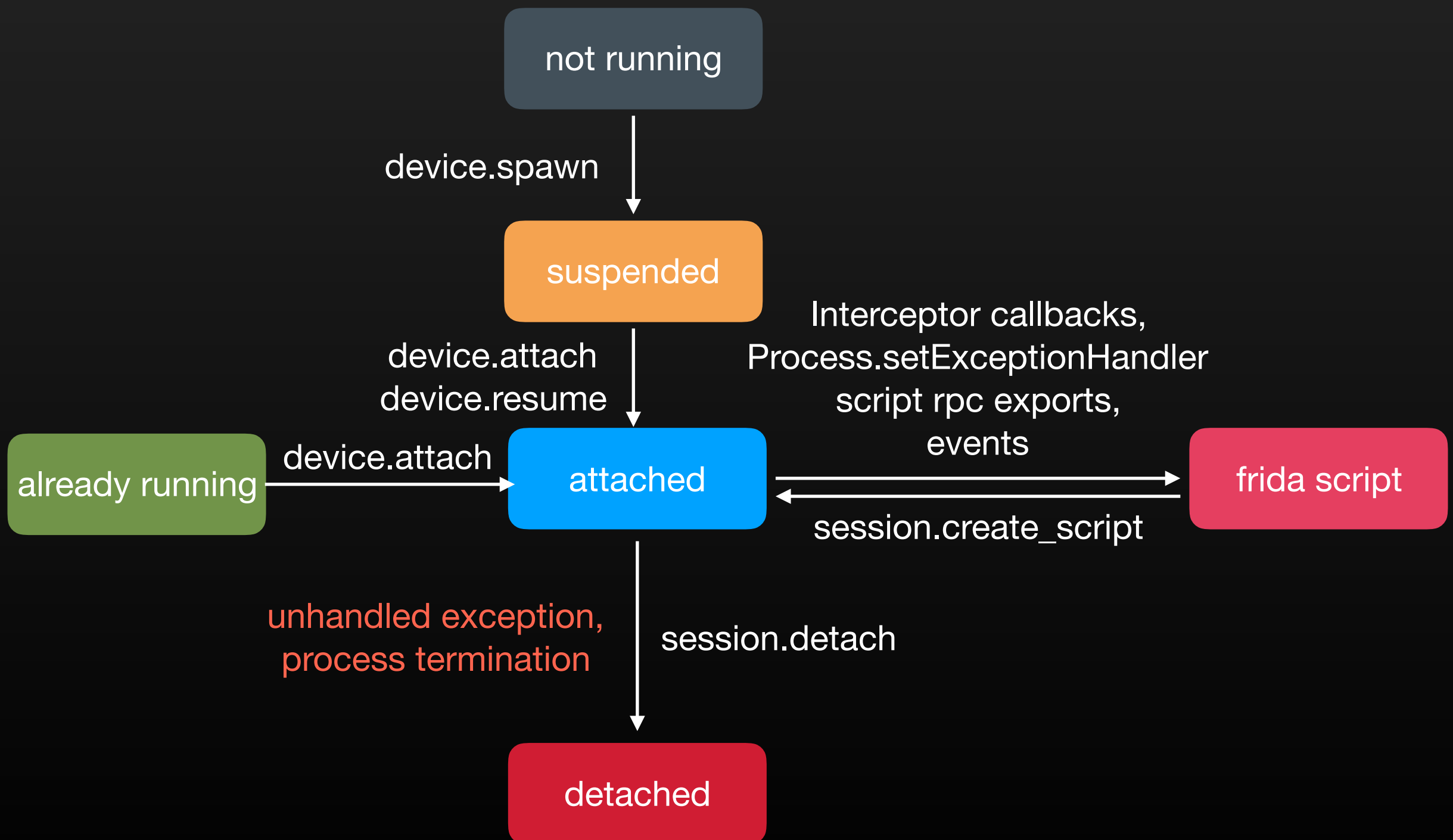
- argv: 命令行
- cwd: 当前目录
- envp: 替换整个环境变量
- env: 添加额外环境变量
- stdio: 重定向 stdio
- aux: 平台特定参数 (可合并到 kwargs)

```
public class SpawnOptions : GLib.Object {  
    public string[]? argv { get; set; }  
    public string[]? envp { get; set; }  
    public string[]? env { get; set; }  
    public string? cwd { get; set; }  
    public Frida.Stdio stdio { get; set; }  
    public GLib.VariantDict aux { get; }  
  
    public SpawnOptions ();  
}
```

```
device.spawn("/bin/busybox", argv=["/bin/cat", "/etc/passwd"])  
device.spawn("com.apple.mobilesafari") # by bundle id, for Android / iOS only  
device.spawn("/bin/ls", envp={ "CLICOLOR": "1" }) # replace original env  
device.spawn("/bin/ls", env={ "CLICOLOR": "1" }) # extends original env  
device.spawn("/bin/ls", stdio="pipe")
```

```
device.spawn("com.apple.mobilesafari", url="https://frida.re") # invoke url scheme  
device.spawn("com.android.settings", activity=".SecuritySettings") # specific activity  
device.spawn("/bin/ls", aslr="disable") # disable ASLR
```


session 生命周期



session 对象方法

- on / off: 添加 / 删除事件监听回调
 - detached 事件: 会话断开 (进程终止等)
- create_script: 从 js 代码创建 Script 对象
- compile_script / create_script_from_bytes: 将 js 编译为字节码, 然后创建 Script
- enable_debugger / disable_debugger: 启用 / 禁用外部调试器
- enable_jit: 切换到支持 JIT 的 v8 脚本引擎 (不支持 iOS)
- enable_child_gating / disable_spawn_gating: 启用 / 禁用子进程收集

事件和异常处理

- 脚本使用 `send` 和 `recv` 与 python 绑定进行双向通信
- `recv` 返回的对象提供 `wait` 方法，可阻塞等待 python 端返回
- 在 js 的回调中产生的异常，会生成一个 type: “error” 的消息
- 除了消息之外，python 还可调用 js 导出的 `rpc.exports` 对象中的方法。通过返回 `Promise` 对象来支持异步任务（详见后续章节*）
- `rpc` 接口产生的异常会直接抛出到 python，而不是交给 `on(‘message’)` 的回调函数
- 示例代码： `hello-frida/rpc.py`

Javascript 进阶

JS

frida 的 Javascript 引擎

- 由于 iOS 的 JIT 限制，以及嵌入式设备的内存压力，新版将默认脚本引擎从 V8 迁移至 Duktape (<http://duktape.org/>)
- 在 Android 等支持 v8 的平台上仍然可以使用 enable-jit 选项切换回 v8
- Duktape 比 v8 缺失了非常多 ECMAScript 6 特性，如箭头表达式、let 关键字 <http://wiki.duktape.org/PostEs5Features.html>
- frida --debug 启用调试需使用 Duktape，不兼容 v8-inspector

箭头函数

- ECMAScript 6 引入的书写匿名函数的特性
- 需要启用 JIT，或 frida-compile 转译才可在 frida 中使用
- 比 function 表达式简洁。适合编写逻辑较短的回调函数
- 语义并非完全等价。箭头函数中的 this 指向父作用域中的上下文；而 function 可以通过 Function.prototype.bind 方法指定上下文
- 以下两行代码等价

```
Process.enumerateModulesSync().filter(function(module) { return  
module.path.startsWith('/Applications') })
```

```
Process.enumerateModulesSync().filter(module => module.path.startsWith('/  
Applications'))
```

generator 函数

- generator（生成器）是一种具有“暂停”功能的特殊函数。用于生成集合、数列等场景
- 语法上比普通函数多了一个 `*`，在函数内部使用 `yield` 关键字产生一个输出。调用后获得一个 generator 对象，generator 对象的 `next` 方法执行到第一个 `yield`，暂停 generator 函数
- 需要 v8 引擎支持；或使用 `frida-compile` 以及 `transform-regenerator` 插件

```
function *gen() {  
  yield 1;  
  yield 2;  
  yield 3;  
}
```

← 暂停执行并返回，直到调用下一次 `next` 方法

```
let list = gen();  
console.log(list.next());  
console.log(list.next());  
console.log(list.next());
```

```
→ ~ node generator.js  
{ value: 1, done: false }  
{ value: 2, done: false }  
{ value: 3, done: false }
```

generator 函数

- 浏览器和 node.js 中的 Javascript 使用单线程（注*），使用阻塞接口会导致界面锁死或影响性能。AJAX 等接口设计成异步回调，嵌套使用会出现 callback hell
- 因为生成器函数暂停执行而不阻塞事件循环的特点，被 js 社区用来管理异步控制流

```
const then = Date.now();
setTimeout(function() {
  console.log(Date.now() - then);
  setTimeout(function() {
    setTimeout(function() {
      // do something
    }, 2000);
  }, 1000);
}, 500);
```

```
const co = require('co');
function sleep(ms) {
  return function (cb) {
    setTimeout(cb, ms);
  };
}

co(function* () {
  var now = Date.now();
  yield sleep(500);
  console.log(Date.now() - now);
  yield sleep(1000);
  yield sleep(2000);
});
```


async / await

- 调用一个 async 函数会返回一个 Promise 对象。当这个 async 函数返回一个值时，Promise 的 resolve 方法会负责传递这个值；当 async 函数抛出异常时，Promise 的 reject 方法也会传递这个异常值

```
async function name([param[, param[, ... param]]]) { statements }
```

- async 函数中可能会有 await 表达式，这会使 async 函数暂停执行，等待表达式中的 Promise 解析完成后继续执行 async 函数并返回解决结果
- await 关键字仅仅在 async function 中有效。如果在 async function 函数体外使用 await，会得到一个语法错误 (SyntaxError)

async / await

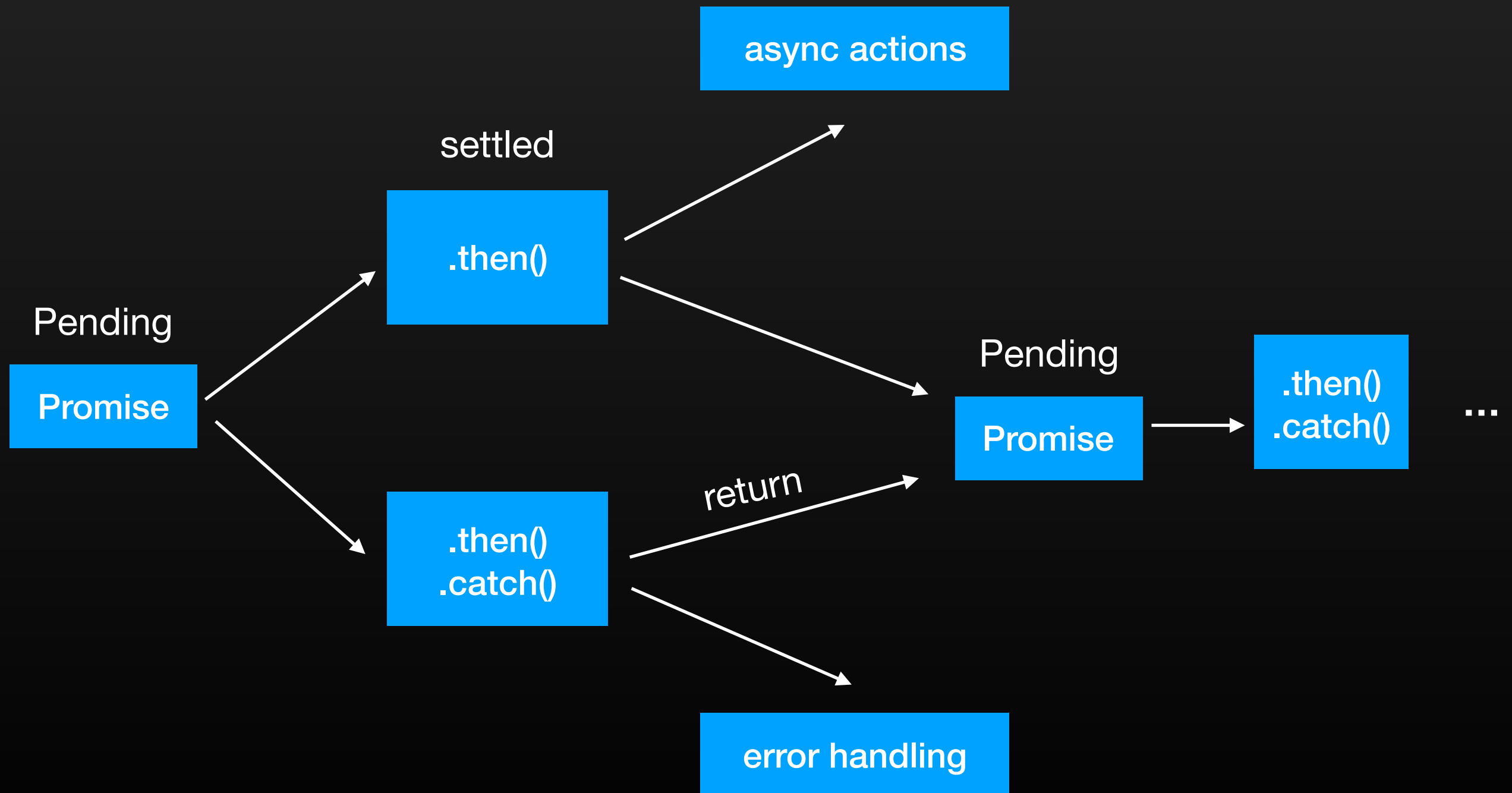
```
step1(function(value1) {  
  // do something  
  step2(value1, function(value2) {  
    // do some other thing  
    step3(value2, function(value3) {  
      console.log('the final result', value3);  
    });  
  });  
});
```

```
const step1 = () =>  
  new Promise((resolve, reject) => {  
    // do something  
    resolve(result);  
  });  
  
const value1 = await step1();  
const value2 = await step2(value1);  
const value3 = await step2(value2);  
console.log('the final result',  
value3);
```

Promise

- Promise 对象是一个代理对象（代理一个值），被代理的值在 Promise 对象创建时可能是未知的。它允许你为异步操作的成功和失败分别绑定相应的处理方法（handlers）。这让异步方法可以像同步方法那样返回值，但并不是立即返回最终执行结果，而是一个能代表未来出现的结果的 promise 对象
- 一个 Promise 有以下几种状态：
 - pending: 初始状态，既不是成功，也不是失败状态。
 - fulfilled: 意味着操作成功完成。
 - rejected: 意味着操作失败。
- 因为 `Promise.prototype.then` 和 `Promise.prototype.catch` 方法返回 promise 对象，所以它们可以被链式调用。

Promise 状态转换



使用 Promise

- Duktape 原生支持 Promise 规范，但 `async/await` 或 `yield` 需要使用 `frida-compile` 转译回 ES5
- `rpc.exports` 接口支持 Promise，可实现等待回调函数返回。示例代码：`hello-frida/rpc.py`
- `frida` 内置与 I/O 相关的接口 `Socket`, `SocketListener`, `IOStream` 及其子类均使用 Promise 的接口。结合 `Stream` 可实现大文件传输等异步任务

frida-compile

- 需求
 - 默认使用的 Duktape 不支持最新的 ECMAScript 特性
 - 单个 js 文件，难以管理大型项目
- 可将 TypeScript 或 ES6 转译成 Duktape 可用的 ES5 语法
- 支持 Browserify 的打包，支持 ES6 modules、source map 和 uglify 代码压缩。甚至可生成 Duktape 字节码
- 支持使用 `require` 或 `es6 module` 引用第三方 npm 包
- frida-compile 是 npm 包，需要 node.js 运行环境。与 frida-python 不冲突，可同时安装使用

frida-compile

- frida-compile 使用解语法糖和 polyfill 实现 ECMAScript 5 之后的特性
 - babel <https://babeljs.io/docs/en/plugins/>
 - typescript <https://www.typescriptlang.org/docs/home.html>
- 使用 TypeScript 可享受到类型系统带来的大型项目管理便利
- Babel 添加插件支持高级的语法特性 (generator / async-await)

frida-compile

npm 创建目录结构、安装依赖，在 package.json 中添加构建脚本

```
1 {
2   "name": "frida-ipa-agent",
3   "version": "0.0.1",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1",
8     "build": "frida-compile src -o dist.js",
9     "watch": "frida-compile src -o dist.js --watch"
10  },
11  "browserify": {
12    "transform": [
13      [
14        "babelify",
15        {
16          "presets": [
17            [
18              "es2015",
19              {
20                "loose": true
21              }
22            ]
23          ],
24          "plugins": [
25            "transform-runtime"
26          ]
27        }
28      ]
29    ]
30  }
31 }
```

frida-compile 命令参数

- o 输出文件名
- w 监视模式。源文件改动后立即编译
- c 开启 uglify 脚本压缩
- b 输出字节码
- h 查看完整命令行用法
- x, --no-babelify 关闭 babel 转译

browserify 中可深度配置编译参数，通过加载 plugins 支持各种 babel 扩展

复用 npm 包

- 使用 frida-compile 之后可以引用第三方 npm 包
- 配置好 frida-compile 之后在相同目录 npm install, 然后直接引用

```
const macho = require('macho')  
import macho from 'macho'
```

- node.js 与 frida 的 Duktape 运行环境存在差异, 无法完全兼容
 - 缺少 node.js 内置核心模块导致依赖缺失
 - 不支持 node C++ 扩展
 - frida-compile 默认提供一些兼容 node.js 的内置包
<https://github.com/frida/frida-compile/blob/master/index.js#L19>

复用 npm 包

模拟了部分 node.js 中的 process 对象、
socket、文件系统、http、大整数和 Buffer
api

```
const fridaBuiltins = Object.assign({}, require('browserify/lib/builtins'), {  
  '_process': require.resolve('frida-process'),  
  'buffer': require.resolve('frida-buffer'),  
  'fs': require.resolve('frida-fs'),  
  'net': require.resolve('frida-net'),  
  'http': require.resolve('frida-http'),  
  'bignum': require.resolve('bignumber.js'),  
  'any-promise': require.resolve('frida-any-promise'),  
});
```

IOStream

- 模仿 node.js 中的流式处理接口。InputStream 对应 ReadableStream, OutputStream 对应 WritableStream
- 优点：每次只读 / 写指定大小的缓冲区，节省内存使用
- 缺点：不支持文件随机访问
- 场景：大文件传输、流式的 parser、Socket 编程
- frida 内置的流对象：IOStream, OutputStream, InputStream, UnixInputStream, UnixOutputStream, Win32InputStream, Win32OutputStream, SocketConnection

IOStream

read

InputStream

UnixInputStream

fd

Win32InputStream

HANDLE

write

OutputStream

UnixOutputStream

fd

Win32OutputStream

HANDLE

IOStream

R/W

SocketConnection

{Unix,Win32}{Input,Output}Stream

- 构造器使用文件描述符 / 句柄而不是路径。根据不同平台的功能，使用文件句柄可以访问除文件系统之外的其他资源（如驱动抽象的设备）
 - `new UnixInputStream(fd[, options])`
 - `new UnixOutputStream(fd[, options])`
 - `new Win32InputStream(handle[, options])`
 - `new Win32OutputStream(handle[, options])`
- frida 目前没有提供获取 fd / HANDLE 的接口，需要自行封装 NativeFunction / SystemFunction 调用对应平台的接口（libc!open / user32!OpenFileW）获取
- 传输大文件的示例代码：stream-adb-pull/

与本地代码交互



指针和内存管理

- NativePointer: 表示 C 中的指针, 可指向任意 (包括非法) 地址
- frida 数据指针、函数指针、代码页地址、基地址等均依赖 NativePointer 接口
- 提供 add, sub, and, or, xor 和算术左右移运算。详见文档
- 可用 Memory.alloc /.allocUtf8String / .allocAnsiString / .allocUtf16String 分配
- Memory.alloc* 分配的内存, 在变量作用域之外会被释放

```
function alloc() {  
    return Memory.alloc(8);  
}
```

```
const p = alloc(); // dangling pointer
```

指针和内存管理

frida	libc
Memory.alloc(size)	malloc
Memory.scan(address, size, pattern, callbacks) Memory.scanSync(address, size, pattern)	memmem
Memory.copy(dst, src, n)	memcpy
Memory.protect(address, size, protection)	mprotect

内存读写

- 任意地址读写。实现解引用等 C 语言才有的功能
- Memory.write* 和 Memory.read* 系列函数，类型包括 S8, U8, S16, U16, S32, U32, Short, UShort, Int, UInt, Float, Double
- 其中 Memory.{read,write}{S64,U64,Long,ULong} 由于 Javascript 引擎默认对数字精度有限制（不支持 64 位大整数），需要使用 frida 的 Int64 或 UInt64 大整数类

内存读写

- 字符串函数
 - 分配 `Memory.alloc{Ansi,Utf8,Utf16}String`
 - 读取 `Memory.read{C,Utf8,Utf16,Ansi}String` (注意 `CString` 只提供了 `read`)
 - 覆写 `Memory.write{Ansi,Utf8,Utf16}String`
- 读写一块连续内存: `Memory.{read,write}ByteArray` (想想为什么没有 `writeCString` 和 `allocCString`)

Unicode 和 ANSI

Windows 平台为兼容 16 位系统，以 A/W 后缀区分宽字符，用 TCHAR 宏处理

BOOL CreateProcessW(

LPCWSTR	lpApplicationName,
LPWSTR	lpCommandLine,
LPSECURITY_ATTRIBUTES	lpProcessAttributes,
LPSECURITY_ATTRIBUTES	lpThreadAttributes,
BOOL	bInheritHandles,
DWORD	dwCreationFlags,
LPVOID	lpEnvironment,
LPCWSTR	lpCurrentDirectory,
LPSTARTUPINFOFOW	lpStartupInfo,
LPPROCESS_INFORMATION	lpProcessInformation

);

wchar_t *

Memory.readUtf16String

BOOL CreateProcessA(

LPCSTR	lpApplicationName,
LPSTR	lpCommandLine,
LPSECURITY_ATTRIBUTES	lpProcessAttributes,
LPSECURITY_ATTRIBUTES	lpThreadAttributes,
BOOL	bInheritHandles,
DWORD	dwCreationFlags,
LPVOID	lpEnvironment,
LPCSTR	lpCurrentDirectory,
LPSTARTUPINFOA	lpStartupInfo,
LPPROCESS_INFORMATION	lpProcessInformation

);

char *

Memory.readAnsiString

wchar_t 并未规定宽字符的实际编码，以上只对 Windows API 适用

Windows 字符编码示例

```
const buf = Memory.alloc(1024);

GetWindowTextA(hWnd, buf, 1024);
console.log('ansi: ', Memory.readAnsiString(buf));
console.log('c string: ', Memory.readCString(buf));

GetWindowTextW(hWnd, buf, 1024);
console.log('unicode16: ', Memory.readUtf16String(buf));
```

```
//
ansi:  电池指示器?
c string:  ????????
unicode16:  电池指示器
undefined
[Local::explorer.exe]->
```

- ANSI: 使用 GetWindowTextA 返回的中文正常显示, 但终止符出现乱码
- CString: Windows 上不支持中文
- unicode16: 使用 GetWindowTextW 返回的中文完美显示
- 使用 utf8 可能会抛出编码异常

分析模块和内存页

- Process 对象
 - 进程基本信息: arch, platform, pageSize, pointerSize 等
 - 枚举模块、线程、内存页: enumerateThreads(Sync), enumerateModules(Sync), enumerateRanges(Sync)
 - 查找模块: findModuleByAddress, findModuleByName
 - 查找内存页:

分析模块和内存页

- frida 内置了导入导出表的解析
 - `Module.enumerateImports(Sync)`
 - `Module.enumerateExports(Sync)`
- 确保模块初始化完成 `Module.ensureInitialized`
 - 例：等待 Objective C 运行时初始化：
`Module.ensureInitialized('Foundation')`

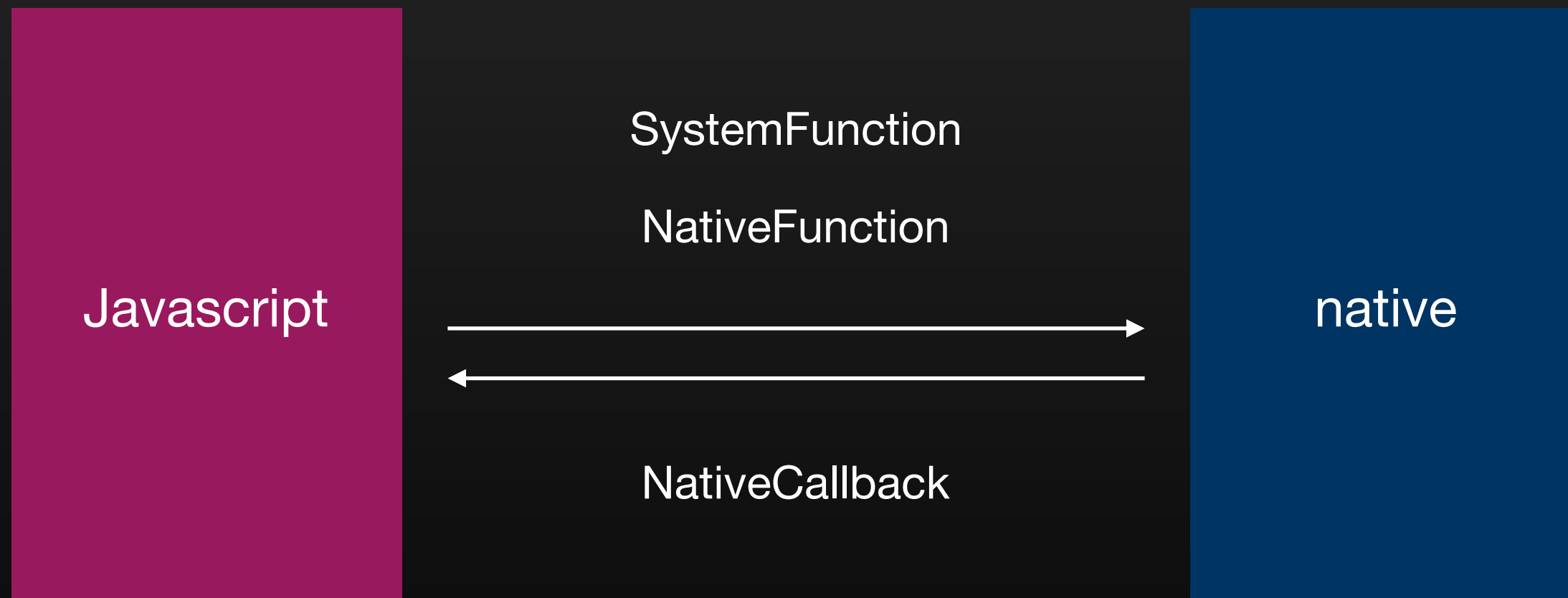
enumerate*Sync?

- frida 的 API 设计中存在大量具有 Sync 后缀的 api
 - 带 Sync 后缀：直接返回整个列表
 - 不带 Sync 后缀：传入一个 js 对象，其 onMatch 和 onComplete 属性分别对应迭代的每一个元素的回调，和结束时的回调函数。与 IO 不同的是这些回调函数都是同步执行的。onMatch 函数可以返回 'stop' 字符串终止迭代
- 猜测应是 Duktape 没有原生支持 js 迭代器的原因

解析函数地址

- 导入 / 导出的符号：
 - `Module.findExportByName(null, 'open')`
 - `Module.enumerateExports(Sync) / enumerateImports(Sync)`
- 内嵌调试符号（未 strip）
 - `DebugSymbol.getFunctionByName("")`
 - `DebugSymbol.findFunctionsNamed()`
 - `DebugSymbol.findFunctionsMatching`
- 无符号：使用模块基地址 + 偏移
 - `Process.findModuleByName('Calculator').base.add(0x3200)`

ffi bridge



frida 中的 javascript 函数
可使用 NativeCallback 封
装成一个有效的机器码
stub

反之 javascript 引擎封装
了可以动态指定参数和返
回值类型的 foreign
function interface 给
javascript

ffi bridge

- NativeFunction: 本地代码函数接口
 - 由于缺少类型信息, 需要指定函数指针、原型, 包括参数列表和返回值类型, 调用约定 (可选)
`new NativeFunction(address, returnType, argTypes[, abi])`
- SystemFunction: 类似 NativeFunction, 但返回字典, 根据平台不同可访问 `errno` (POSIX) 或者 `lastError` 来获得错误信息。函数返回值在 `value` 属性

```
[Local::Finder]-> const openPtr = Module.findExportByName(null, 'open');  
                  const nonExistFile = Memory.allocUtf8String('/aaa');  
                  console.log(JSON.stringify(new SystemFunction(openPtr, 'int',  
['pointer', 'int'])(nonExistFile, 0)));  
                  console.log(new NativeFunction(openPtr, 'int', ['pointer', 'int'])(  
nonExistFile, 0));  
{"value":-1,"errno":2}  
-1
```

ffi bridge

- NativeCallback
 - 返回一个 NativePointer, 指向一个包装好的 javascript 回调。本地代码执行到 NativePointer 的指针时将调用到 javascript

```
const handler = new NativeCallback(function(sig) {  
    console.log('signal:', sig);  
}, 'void', ['int']);  
  
const signal = new NativeFunction(Module.findExportByName(null, 'signal'),  
    'int', ['int', 'pointer']);  
const SIGINT = 2;  
signal(SIGINT, handler);
```

ffi bridge

- 返回值和参数列表支持的类型： void, pointer, int, uint, long, ulong, char, uchar, float, double, int8, uint8, int16, uint16, int32, uint32, int64。
- 不同的头文件、编译环境可能会有会 typedef 别名
- 结构体参数（注意不是结构体指针）可使用嵌套的 Array 表示
- 可变参数可用 ‘...’ 处理

可变参数

```
const NSLog = new NativeFunction(Module.findExportByName('Foundation',  
'NSLog'), 'void', ['pointer', '...', 'pointer']);  
const format = ObjC.classes.NSString.stringWithString_('hello %@!');  
const param = ObjC.classes.NSString.stringWithString_('world');  
NSLog(format, param);
```

`['pointer', '...', 'pointer']`

...

格式串的类型 表示可变参数 剩余参数的对应类型

事实上无法像 NSLog 那样完全支持动态参数个数，需要
预先知道每个实际格式化的类型

结构体参数

C 语言支持直接将结构体作为参数传递，调用时即使调用约定为使用寄存器传参，结构体也将整个由堆栈传递

例如 iOS 中的 CGSize

```
struct CGSize {  
    CGFloat width;  
    CGFloat height;  
};
```

将所有的成员均列到数组中（缺一不可），支持嵌套

```
const CGFloat = (Process.pointerSize === 4) ? 'float' : 'double';  
const CGSize = [CGFloat, CGFloat];  
  
const UIGraphicsBeginImageContextWithOptions = new NativeFunction(  
    Module.findExportByName('UIKit', 'UIGraphicsBeginImageContextWithOptions'),  
    'void', [CGSize, 'bool', CGFloat]);
```

ABI

- default
- Windows 32-bit:
 - sysv
 - stdcall
 - thiscall
 - fastcall
 - mscdecl
- Windows 64-bit:
 - win64
- UNIX x86:
 - sysv
 - unix64
- UNIX ARM:
 - sysv
 - vfp

default: frida 根据系统和 CPU 自动决定

cdecl: 最常见的 C 语言调用约定, 使用堆栈传参

stdcall: Win32 API 使用调用约定

thiscall: MSVC 编译的 32 位 x86 C++ 程序, 特点是使用 ecx 传递 this 指针

win64: 64 位系统微软统一了调用约定, 使用寄存器和堆栈结合的方式

unix64: 与 win64 类似, 但使用不同的寄存器

Interceptor

- Interceptor 对指定函数指针设置 inline hook
 - Interceptor.attach: 在进入函数之前, 函数返回后分别调用 onEnter 和 onLeave 回调函数。可以对函数参数和返回值进行修改, 但原始函数一定会被调用
 - Interceptor.replace: 整个替换掉函数调用。可以在 js callback 里继续调用原始函数, 也可以完全屏蔽掉
- 在 js 回调中可以访问 this 获得上下文信息, 如常用寄存器、thread id 等; 此外 this 还可以在 onEnter 保存额外的参数传递给 onLeave

Interceptor

args: 以下标函数参数，默认均为
NativePointer。可用 toInt32 转换为整型

```
Interceptor.attach(Module.findExportByName("libc.so", "open"), {  
  onEnter: function (args) {  
    console.log(Memory.readUtf8String(args[0]));  
    this.fd = args[1].toInt32();  
  },  
  onLeave: function (retval) {  
    if (retval.toInt32() == -1) {  
      /* do something with this.fd */  
    }  
  }  
});
```

retVal.replace 可整个替换掉返回值

调用堆栈

- 获取寄存器上下文
 - 插桩回调中访问 `this.context`
 - `Process.enumerateThreadsSync()` 枚举线程信息
- `Thread.backtrace` 可根据上下文回溯出调用堆栈的地址
- `DebugSymbol.fromAddress` 进一步对地址符号化

```
console.log('\tBacktrace:\n\t' + Thread.backtrace(this.context,  
    Backtracer.ACCURATE).map(DebugSymbol.fromAddress).join('\n\t'));
```

与 ~~C++~~ C with classes 交互

- 与 C 函数交互类似，区别在于
 - this 指针传递的调用约定
 - 对象内存的分配和成员
 - 返回一个对象
- 不同编译器可能存在实现差异，以反汇编为准

从反编译入手

```
38 Cat::Cat(&v16, 1LL, 2LL);
39 Cat::toString((Cat *)&v15);
40 v23 = std::__1::operator<<<char,std::__1::char_traits<char>,std::__1::allocator<char>>(&std::__1::cout, &v15);
41 v22 = std::__1::endl<char,std::__1::char_traits<char>>;
42 std::__1::endl<char,std::__1::char_traits<char>>(v23, &v15);
43 std::__1::basic_string<char,std::__1::char_traits<char>,std::__1::allocator<char>>::~~basic_string(&v15);
44 Dog::Dog((Dog *)&v14, 6);
45 Dog::talk((Dog *)&v14);
46 Dog::toString((Dog *)&v13);
47 v25 = std::__1::operator<<<char,std::__1::char_traits<char>,std::__1::allocator<char>>(&std::__1::cout, &v13);
48 v24 = std::__1::endl<char,std::__1::char_traits<char>>;
49 std::__1::endl<char,std::__1::char_traits<char>>(v25, &v13);
50 std::__1::basic_string<char,std::__1::char_traits<char>,std::__1::allocator<char>>::~~basic_string(&v13);
51 v9 = (Cat *)operator new(0x10uLL);
52 Cat::Cat(v9, 10);
53 v12 = v9;
54 Cat::printDescription(v9);
55 Cat::toString((Cat *)&v11);
56 v27 = std::__1::operator<<<char,std::__1::char_traits<char>,std::__1::allocator<char>>(&std::__1::cout, &v11);
57 v26 = std::__1::endl<char,std::__1::char_traits<char>>;
58 std::__1::endl<char,std::__1::char_traits<char>>(v27, &v11);
59 std::__1::basic_string<char,std::__1::char_traits<char>,std::__1::allocator<char>>::~~basic_string(&v11);
60 if ( v12 )
61     operator delete((void *)v12);
```

反编译伪代码 (clang)

分配和传参

- 两种分配方式
 - 局部变量：栈上直接分配好 `sizeof(Class)`，用起始地址（`this` 指针）作为第一个参数调用 `constructor`
 - 动态分配：`new` 运算符被链接到库函数，同样也是分配 `sizeof(Class)` 大小，调用构造器方式相同
- 构造器和成员函数的调用约定
 - x86(_64) 上的 `clang` 和 `gcc`，均是将 `this` 指针作为第一个参数，使用寄存器和堆栈传递参数的行为与 C 语言一致，使用 `frida` 的 `default ABI` 参数即可
 - 而 `MSVC` 使用 `ecx / rcx` 传递 `this`，只有 x86 的 `thiscall` 被 `frida` 直接支持

frida 构造 C++ 对象

```
class Cat : public Animal {  
    int m_weight;  
  
public:  
    Cat(int age) : Animal(age), m_weight(5);  
    Cat(int age, Color color) : Animal(age);  
    void printDescription();  
};
```

```
Cat cat(1, orange);  
cat.print();
```

```
const ctor = new NativeFunction(DebugSymbol.getFunctionByName('Cat::Cat(int,  
Color)'), 'pointer', ['pointer', 'int', 'int']);  
const print = new  
NativeFunction(DebugSymbol.getFunctionByName('Cat::printDescription()'), 'void',  
['pointer']);  
const instance = Memory.alloc(16); // sizeof(Cat)  
  
ctor(instance, 3, 2); // 3 year old orange cat  
print(instance); // call instance method
```

完整示例见 [cxx-clang-mac/](#)

frida 调用 C++ 对象

- 实际遇到的 binary 可能没有调试符号
 - 如果导出了 C++ 符号，可以使用 name mangling 过后的名字 `findExportByName`
 - 如果没有符号，逆向获取偏移量从运行时基地址计算
- 根据逆向结果判断 `sizeof(Class)`，使用 `Memory.alloc` 或者 `new` 运算符分配内存。注意 `Memory.alloc` 没有对应的手动释放函数，处理析构有麻烦
- 可以绕过成员函数对结构体中的数据直接进行 `patch`，实现修改私有成员等高级功能

篡改结构体

clang 从源码打印 class 的结构

```
→ cxx-clang-mac git:(master) x clang -cc1 -fdump-record-layouts main.cpp
*** Dumping AST Record Layout
    0 | class Cat
    0 |   class Animal (primary base)
    0 |     (Animal vtable pointer)
    8 |     int m_age
   12 |     int m_weight
      | [sizeof=16, dsize=16, align=8,
      |   nvszie=16, nvalign=8]
```

根据 this 和 offset 篡改类成员

```
const vtable = Memory.readPointer(instance);
console.log('relative addr:', vtable.sub(base));
console.log(DebugSymbol.fromAddress(vtable));

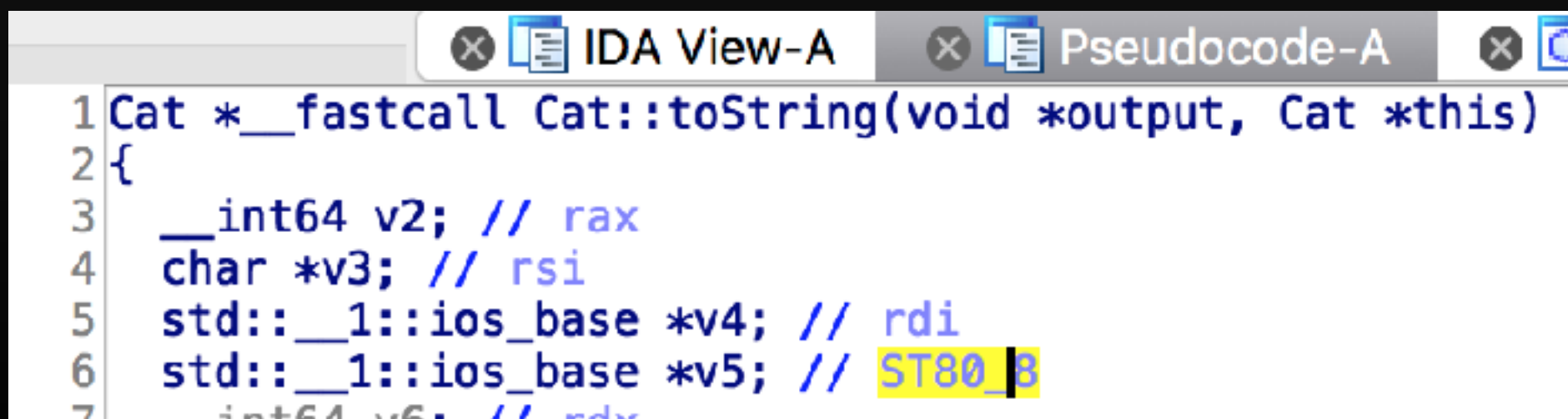
console.log('age', Memory.readInt(instance.add(8)));
console.log('weight', Memory.readInt(instance.add(12)));

Memory.writeInt(instance.add(8), 1);
// patched data
print(instance);
```


返回值

- C++ 成员函数如果返回的不是 primitive value, 那么第一个参数应为调用者开辟的, 用来保存返回值对象的地址

```
string toString() {  
    std::ostringstream out("");  
    out << "<Cat kind=" << kind() << " weight=" << m_weight << "kg"  
        << ">";  
    return out.str();  
}
```



```
1 Cat *__fastcall Cat::toString(void *output, Cat *this)  
2 {  
3     __int64 v2; // rax  
4     char *v3; // rsi  
5     std::__1::ios_base *v4; // rdi  
6     std::__1::ios_base *v5; // ST80_8  
7     int64 v6; // rdx
```

C++ 符号还原

为区分同名的重载函数，将符号按照规则变换 (name mangling)

源码 / 调试符号

```
android::AndroidRuntime::start(char const*,  
android::Vector<android::String8> const&, bool)
```

编译器

demangle

可执行文件

```
_ZN7android14AndroidRuntime5startEPKcRKNS_6VectorINS_7String8EEEb
```

C++ 符号还原

Compiler	<code>void h(int)</code>	<code>void h(int, char)</code>	<code>void h(void)</code>
Intel C++ 8.0 for Linux	<code>_Z1hi</code>	<code>_Z1hic</code>	<code>_Z1hv</code>
HP aC++ A.05.55 IA-64			
IAR EWARM C++ 5.4 ARM			
GCC 3.x and higher			
Clang 1.x and higher			
IAR EWARM C++ 7.4 ARM	<code>_Z<number>hi</code>	<code>_Z<number>hic</code>	<code>_Z<number>hv</code>
GCC 2.9x	<code>h__Fi</code>	<code>h__Fic</code>	<code>h__Fv</code>
HP aC++ A.03.45 PA-RISC			
Microsoft Visual C++ v6-v10 (mangling details)	<code>?h@@YAXH@Z</code>	<code>?h@@YAXHD@Z</code>	<code>?h@@YAXXZ</code>
Digital Mars C++			
Borland C++ v3.1	<code>@h\$qi</code>	<code>@h\$qizc</code>	<code>@h\$qv</code>
OpenVMS C++ V6.5 (ARM mode)	<code>H__XI</code>	<code>H__XIC</code>	<code>H__XV</code>
OpenVMS C++ V6.5 (ANSI mode)		<code>CXX\$__7H__FIC26CDH77</code>	<code>CXX\$__7H__FV2CB06E8</code>
OpenVMS C++ X7.1 IA-64	<code>CXX\$_Z1HI2DSQ26A</code>	<code>CXX\$_Z1HIC2NP3LI4</code>	<code>CXX\$_Z1HV0BCA19V</code>
SunPro CC	<code>__1cBh6Fi_v__</code>	<code>__1cBh6Fic_v__</code>	<code>__1cBh6F_v__</code>
Tru64 C++ V6.5 (ARM mode)	<code>h__Xi</code>	<code>h__Xic</code>	<code>h__Xv</code>
Tru64 C++ V6.5 (ANSI mode)	<code>__7h__Fi</code>	<code>__7h__Fic</code>	<code>__7h__Fv</code>
Watcom C++ 10.6	<code>W?h\$(i)v</code>	<code>W?h\$(ia)v</code>	<code>W?h\$()v</code>

https://en.wikipedia.org/wiki/Name_mangling#How_different_compilers_mangle_the_same_functions

C++ 符号还原

- 使用 C++ abi `abi::__cxa_demangle`
- 实际编译后在不同平台上的实现：
 - Windows + MSVC:
`dbghelp.dll!UnDecorateSymbolName(PCSTR name, PSTR outputString, DWORD maxStringLength, DWORD flags)`
 - clang / gcc:
`__cxa_demangle(const char *mangled_name, char *output_buffer, size_t *length, int *status)`
- 支持 mac/iOS, Android, Windows 和 Linux 的实例代码
`cxx-demangle/agent.js`

swift 符号还原

swift 存在类似 C++ 的 mangling 机制，但规则不同

```
Foundation._MutableHandle<__ObjC.NSMutableURLRequest>
```

```
_TtGC10Foundation14_MutableHandleCSo19NSMutableURLRequest_
```

swift 支持 unicode 函数名（甚至 emoji 🐸）

```
→ ~ xcrun swift-demangle __TF4testX4GrIhFTSiSi_Si  
_TF4testX4GrIhFTSiSi_Si ---> test.💖(Swift.Int, Swift.Int) -> Swift.Int
```

版本较新的 macOS 和 iOS 自带了运行时 demangle 的库：

/System/Library/PrivateFrameworks/Swift/libswiftDemangle.dylib

导出函数 `swift_demangle_getDemangledName`

swift ABI 目前仍不稳定，以官方文档为准 <https://github.com/apple/swift/blob/master/docs/ABI/Mangling.rst>

近期的一个 breaking change 就是将全局符号的前缀 `_T` 改为 `_S`

swift 符号还原

→ `gossip-summer-school-2018 git:(master) x frida -U Music -l swift-demangle/demangle.js`

```
----
/ _ |   Frida 12.0.3 - A world-class dynamic instrumentation toolkit
| (| |
> _ |   Commands:
/_/ |_|   help      -> Displays the help system
. . . .   object?    -> Display information about 'object'
. . . .   exit/quit  -> Exit
. . . .
. . . .   More info at http://www.frida.re/docs/home/
Attaching...
classes
_TtCCV5Music4Text7Drawing5CacheP33_BF7DEC98CE203AC9ACE0BF189B2A64B714ContextWrapper >>
Music.Text.Drawing.Cache.(ContextWrapper in _BF7DEC98CE203AC9ACE0BF189B2A64B7)
_TtGC10FoundationP33_2D7761BAEB66DCEF0A109CF42C1440A718_MutablePairHandleCSo10NSIndexSetCSo17NSMutableIndexSet_
>> Foundation.(_MutablePairHandle in _2D7761BAEB66DCEF0A109CF42C1440A7)<__ObjC.NSIndexSet,
__ObjC.NSMutableIndexSet>
_TtCV5Music21CloudLibraryUtilities22LibraryUpdatesNotifier >> Music.CloudLibraryUtilities.LibraryUpdatesNotifier
_TtC5MusicP33_1491B7AA0651257DFE189F6D76BD86A331ScrollViewContentOffsetObserver >> Music.
(ScrollViewContentOffsetObserver in _1491B7AA0651257DFE189F6D76BD86A3)
_TtCV5Music7Artwork24LoadingStatusCoordinator >> Music.Artwork.LoadingStatusCoordinator
_TtC5MusicP33_F29266F010DB9192D9DBD2C311376C0A20ClassicalWorkSection >> Music.(ClassicalWorkSection in
_F29266F010DB9192D9DBD2C311376C0A)
_TtCV5Music7Artwork9Component >> Music.Artwork.Component
```

示例代码: `swift-demangle/`

Stalker

- Stalker 提供了指令级和代码块级的插桩
- 可用作覆盖率测试，追踪一段时间内所有可能的函数等
- frida-discover 即使用 Stalker 实现
- 示例代码：misc/stalker.js

Stalker

```
0x7fff2db0be60 JavaScriptCore!WTF::ThreadSpecific<WTF::RefPtr<WTF::(anonymous namespace)::ThreadData, WTF::DumbPtrTraits<WTF::(anonymous namespace)::ThreadData> >, (WTF::CanBeGCThread)1>::destroy(void*)
0x7fff2cefea60 JavaScriptCore!WTF::fastFree(void*)
0x7fff2d082f90 JavaScriptCore!WTF::ThreadCondition::~~ThreadCondition()
0x7fff2db26ee0 JavaScriptCore!bmalloc::PerThread<bmalloc::PerHeapKind<bmalloc::Cache> >::destructor(void*)
0x7fff2cefefe0 JavaScriptCore!WTF::Mutex::lock()
0x7fff2db2d8e6 JavaScriptCore!DYLD-STUB$$std::__1::mutex::~~mutex()
0x7fff2db25dc0 JavaScriptCore!bmalloc::Allocator::~~Allocator()
0x7fff2db03a30 JavaScriptCore!WTF::ThreadSpecific<std::optional<WTF::GCThreadType>, (WTF::CanBeGCThread)1>::destroy(void*)
0x7fff2db24090 JavaScriptCore!bmalloc::mapToActiveHeapKind(bmalloc::HeapKind)
0x7fff2cf00a60 JavaScriptCore!WTF::monotonicallyIncreasingTime()
0x7fff2daf00e0 JavaScriptCore!WTF::AutomaticThread::threadIsStopping(WTF::AbstractLocker const&)
0x7fff2cefef0 JavaScriptCore!WTF::Mutex::unlock()
...
```


反汇编和指令 patch

- `Instruction.parse()`
- `{MIPS,Thumb,Arm,Arm64,X86}Relocator`
- `{MIPS,Thumb,Arm,Arm64,X86}Writer`

使用 SQLite

- 调用对应平台上的 SQLite 库函数
- 使用 frida 内置的 SQLiteDatabase 和 SQLiteStatement

```
const db = SQLiteDatabase.open('/path/to/people.db');
const smt = db.prepare('SELECT name, bio FROM people WHERE age = ?');
console.log('People whose age is 42:');
smt.bindInteger(1, 42);
var row = null;
while ((row = smt.step()) !== null) {
    console.log('Name:', row[0], 'Bio:', row[1]);
}
smt.reset();
```

libclang 生成代码

- 从头文件中查找常量、typedef 和手写 NativeFunction 体验非常糟糕，可以利用 clang 节省一部分工作量
- 打印源码中的结构体和偏移：
`clang -cc1 -fdump-record-layouts main.cpp`
- 使用 libclang AST 生成代码
- 示例见：[libclang/](#)

libclang 生成代码

- 遍历 AST 查找 CursorKind.CALL_EXPR
 - 遍历函数参数列表，映射 clang AST 中的类型到 frida 的类型
 - 生成 new NativeFunction 语句
- 由于 #define 属于预处理 pass，AST 阶段处理只能保留一部分 token 信息，结果不够准确

libclang 生成代码

```
44 void checkport(pid_t pid) {
45     LOG();
46     LOG("list network connections on pid: %d", pid);
47     int buf_size = proc_pidinfo(pid, PROC_PIDLISTFDS, 0, 0, 0);
48     LOG("buf size: %d", buf_size);
49
50     REQUIRES(buf_size != -1, "unable to get process fd");
51     struct proc_fdinfo *info_array = (struct proc_fdinfo *)malloc(buf_size);
52     REQUIRES(info_array, "out of memory? must be kidding me");
53     proc_pidinfo(pid, PROC_PIDLISTFDS, 0, info_array, buf_size);
54     int n_fd = buf_size / PROC_PIDLISTFD_SIZE;
55 }
```

→ libclang git:(master) python3 codegen.py

```
const __LP64__ = 1;
const PROC_PIDLISTFDS = 1;
const PROC_PIDLISTFDS = 1;
const PROX_FDTYPE_VNODE = 1;
const PROC_PIDFDEVNODEPATHINFO = 2;
const macho_section = section_64;
const macho_section = section_64;
const = new NativeFunction(Module.findExportByName(null, ''), 'int', ['pointer', 'pointer']);
/* [info] function fprintf detected, try `console.log()` or OutputStream */
const proc_pidinfo = new NativeFunction(Module.findExportByName(null, 'proc_pidinfo'), 'int', ['int', 'int', 'uint64', 'pointer', 'int']);
const exit = new NativeFunction(Module.findExportByName(null, 'exit'), 'void', ['int']);
/* [info] function malloc detected, try `Memory.alloc()` */
const proc_pidfdinfo = new NativeFunction(Module.findExportByName(null, 'proc_pidfdinfo'), 'int', ['int', 'int', 'int', 'pointer', 'int']);
/* [info] function _dyld_get_image_header detected, try `Process.enumerateModulesSync()[index].base` */
```

frida on Android

frida-java

- frida-java 是 frida 内置库，即 Java 命名空间下的函数。可对 ART 和 Dalvik 运行时插桩
- 源代码 [github/frida/frida-java](https://github.com/frida/frida-java)
- 在 frida 框架基础上完全由 javascript 实现。frida-gum 只实现了通用的二进制插桩，而 frida-java 借助 jni 打通了 Java 和 Javascript 的世界（有兴趣研究可阅读 `/lib/class-factory.js`）

Java api

- available: 判断 Java 环境可用
- enumerateLoadedClasses(Sync): 枚举所有已加载的 class
- perform: 执行任意 Java 操作都需要使用此函数
- use: 根据完整类名查找类的句柄
- scheduleOnMainThread: 在 JVM 主线程执行一段函数
- choose: 内存中搜索类的实例
- cast: 类型强转

Why Java.perform?

- ART 和 Dalvik 都按照 JVM 的规范实现: <https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/invocation.html>
- frida 的 js 脚本引擎使用了（非主线程）的其他线程，需要使用 `javaVM->AttachCurrentThread`。而对应为了释放资源，完成任务后需 `DetachCurrentThread`
- 为了保证关联和释放，所有涉及 JVM 的操作都需要放在 `Java.perform` 回调中执行

操作对象

- frida 既可以 new 对象实例，也可以搜索已有的对象
- 在 class-factory.js 可以看到一些文档上未标注的，\$ 开头的成员
 - `$new`: `new` 运算符，初始化新对象。注意与 `$init` 区分
 - `$alloc`: 分配内存，但不初始化
 - `$init`: 构造器方法，用来 hook 而不是给 js 调用
 - `$dispose`: 析构函数
 - `$isSameObject`: 是否与另一个 Java 对象相同
 - `$className`: 类名

操作对象

```
if (!Java.available)
    throw new Error('requires Android');

Java.perform(function() {
    const JavaString = Java.use('java.lang.String');
    var exampleString1 = JavaString.$new('Hello World, this is an
example string in Java. ');
    console.log('[+] exampleString1: ' + exampleString1);
    console.log('[+] exampleString1.length(): ' +
exampleString1.length());
});
```

```
[+] exampleString1: Hello World, this is an example string in Java.
[+] exampleString1.length(): 47
```

操作对象

- frida 访问 / 修改对象成员
 - `instance.field.value`
 - `instance.field.value = newValue`
 - 这种方式不区分成员可见性，即使是私有成员同样可以直接访问
 - 除 `value` 的 `setter` 和 `getter` 之外，`fieldType` 和 `fieldReturnType` 获取类型信息
- frida 对数组做了封装，直接取下标即可访问

操作对象

- 注意 instance 和 Class 的区别
- Java.choose 找到实例后查询字段的类型

```
Java.perform(function () {  
    var MainActivity =  
    Java.use('com.example.seccon2015.rock_paper_scissors.MainActivity');  
    Java.choose(MainActivity.$className, {  
        onMatch: function(instance) {  
            console.log(JSON.stringify(instance.P.fieldReturnType));  
        },  
        onComplete: function() {}  
    });  
})
```

```
{"className": "android.widget.Button", "name": "Landroid/widget/Button;", "type": "pointer", "size": 1}
```

(注) APK 来源: <https://github.com/ctfs/write-ups-2015/tree/master/seccon-quals-ctf-2015/binary/reverse-engineering-android-apk-1>

插桩

- Java 层的插桩
 - `Java.use().method.implementation = hookCallback`
 - 由于 Java 支持同名方法重载，需要用 `.overload` 确定具体的方法

```
Java.use('java.lang.String').$new.overload('[B', 'java.nio.charset.Charset')
```
- JNI 层插桩
 - JNI 实现在 so 中，且符号必然是导出函数，照常使用 `Interceptor` 即可

获取调用堆栈

- Android 提供了工具函数可以打印 Exception 的堆栈。此方式等价于 `Log.getStackTraceString(new Exception)`

```
Java.perform(function () {
    const Log = Java.use('android.util.Log');
    const Exception = Java.use('java.lang.Exception');
    const MainActivity = Java.use('com.example.secon2015.rock_paper_scissors.MainActivity');
    MainActivity.onClick.implementation = function(v) {
        this.onClick(v);
        console.log(Log.getStackTraceString(Exception.$new()));
    };
});
```

frida on macOS/iOS

frida-objc

- 对应 Java, ObjC api 是 frida 的另一个“一等公民”
- 源代码 [github/frida/frida-objc](https://github.com/frida/frida-objc)
- 与 JVM 类似, Objective C 也提供了 runtime api: [https://developer.apple.com/documentation/objectivec/objective_c_runtime?changes= 3](https://developer.apple.com/documentation/objectivec/objective_c_runtime?changes=3)
- frida 将 Objective C 的部分 runtime api 提供到 ObjC.api 中

frida-objc

- 与 Java 显著不同，frida-objc 将所有 class 信息保存到 ObjC.classes 中。直接对其 for in 遍历 key 即可
 - Objective C 实现：[NSString stringWithString:@"Hello World"]
 - 对应 frida：var NSString = ObjC.classes.NSString;
NSString.stringWithString_("Hello World");
- new ObjC.Object 可以将指针转换为 Objective C 对象。如果指针不是合法的对象或合法的地址，将抛出异常或导致未定义行为

hook Objective C

- `ObjC.classes.Class.method`, 以及 `ObjC.Block` 都提供了一个 `.implementation` 的 setter 来 hook 方法实现。实际上就是 iOS 开发者熟悉的 Method Swizzling
- 另一种方式是使用 `Interceptor.attach(ObjC.classes.Class.method.implementation)`, 看上去很相似, 但实现原理是对 selector 指向的代码进行 inline hook
- Proxy 也是 Objective C 当中的一种 hook 方式。frida 提供了 `ObjC.registerClass` 来创建 Proxy

跨平台示例

Unix 信号

- 使用 `signal` 捕获进程信号
- 要点： `signal` 接收一个函数指针，用 `NativeCallback` 实现
- 示例代码： `misc/signals.js`

iOS / macOS XPC 通信

- XPC 是 macOS / iOS 上常见的进程间通信机制，类似 C/S
- 接收端： `_xpc_connection_call_event_handler`
- 发送端： `xpc_connection_send_message(_with_reply(_sync))`
- `xpc_object_t` 事实上继承自 `NSObject` 对象，可用 frida 直接打印其内容

iOS / macOS 定位伪造

- iOS 和 macOS 定位使用统一 API: CLLocationManager
- 需指定一个 delegate 实现如下回调方法获取相应事件:
 - -(void)locationManager:(CLLocationManager *)manager didUpdateLocations:(NSArray *)locations
 - - (void)locationManager:(CLLocationManager *)manager didUpdateToLocation:(CLLocation *)newLocation fromLocation:(CLLocation *)oldLocation;
- 使用如下方法开始定位
 - - (void)requestLocation;
 - - (void)startUpdatingLocation;

iOS / macOS 定位伪造

- 先处理 `requestLocation` 等方法拿到 `delegate` 的指针
- 在 `delegate` 上查找对应回调方法是否存在，逐个 hook
- `CLLocation` 的经纬度是只读属性，需要创建新的副本。为了对抗时间戳等特征检测，最好把正确的 `CLLocation` 除经纬度之外所有的属性复制上去
- 示例代码：`ios-macOS-fake-location/fake.js`

Windows 服务端抓包

- 目标: `Ws2_32.dll` 导出的 `WSARecv` 和 `WSAAccept`
- 要点: `WSARecv` 在执行完毕之后缓冲区才有有效数据, 但缓冲区个数和长度需要在函数进入之前的参数来确定
- 示例代码: `windows-socket/server.js`

参考

- frida.re
- glider菜鸟 - frida 源码阅读之 frida-java