

Frida官方手册中文版

目的

- 给自己一个认真阅读文档的机会
- 方便英文恐惧症的同学^..^

信息来源

- 原文信息地址：<https://www.frida.re/docs/home/>

翻译说明

- 部分名词部分，找不到比较合适的中文表达，直接使用原英文名词
- 翻译的时候，自己不完全确定的部分进行了标注，请大家参考原文
- 如果阅读的时候，发现中文表述的难以理解，也可参考原文，很多时候原版英文表述更容易理解

译者

- 看雪ID: freakish
- 微信:freakishfox(敌法)
- 发现问题的地方欢迎交流~~

2017-11-13

FЯIDA

Frida官方手册中文版

目的

- 给自己一个认真阅读文档的机会
- 方便英文恐惧症的同学^..^

信息来源

- 原文信息地址：<https://www.frida.re/docs/home/>

翻译说明

- 部分名词部分，找不到比较合适的中文表达，直接使用原英文名词
- 翻译的时候，自己不完全确定的部分进行了标注，请大家参考原文
- 如果阅读的时候，发现中文表述的难以理解，也可参考原文，很多时候原版英文表述更容易理解

译者

- 看雪ID: freakish
- 微信:freakishfox(敌法)
- 发现问题的地方欢迎联系交流~~
- 20180616经过原作者同意调整（主要是我的kindle对里面的图片不是很友好，所以添加了**highlight Javascript** 替换了图片）
52pojie_ID:shutd0wn，如果有错误你可以联系我~

2017-11-13

欢迎使用

- 这篇文章的主要目的是对**Frida**做一个概要的阐述。主要涵盖如下几个方面：
 1. 命令行模式下，交互式函数跟踪
 2. 基于**Frida API**编写你自己的实用工具
 3. 关于**Frida**开发的一些建议

Frida到底是什么？

- 通俗一点讲，**Frida**是为**Native**应用开发出来的一把瑞士军刀。从技术的角度讲，**Frida**是一个动态代码执行工具包。通过**Frida**你可以把一段**JavaScript**注入到一个进程中去，或者把一个动态库加载到另一个进程中去，并且**Frida**是跨平台的，也就是说，你可以对**Windows, macOS, GNU/Linux, iOS, Android**以及**QNX** 系统上的进程进行上述操作。
- 为了使用方便，**Frida**工具包在基于自身**API**的基础上也提供了一些简单的小工具， 你可以根据自己的需求直接拿来使用，或者可以作为你后续进行**Frida**脚本开发的参考示例。

Frida能干啥呢？

- 这个问题很实际， 我们来看几个实际的使用场景你就大致明白了：
 - 假如说现在市面上出了很火的**App**, 目前只有 **iOS**版本的， 现在假如你对这个**App**比较感兴趣想了解下大致的实现逻辑。通过**WireShark**这样的抓包工具发现，网络协议是加密的， 看不到有效的信息，这个抓包的方案就不行了。这个时候，你就可以考虑用**Frida**进行 **API** 跟踪的方式来达到你的目的。

- * 假如你开发了一个桌面应用并且已经部署到用户那里，现在用户反馈过来一个问题，但是你会发现你打的日志信息又不足以定位问题， 那你只能说再加一些日志，然后再给
- * 你还可以使用**Frida**给**WireShark**快速的做一个经过加密的协议嗅探器，更进一步，为了满足你的实验需求，你甚至可以主动发起函数调用来模拟不同的网络状态。
- * 如果你有一个内部**App**需要做黑盒测试，如果你用**Frida**的话，你就可以用完全不侵染原有业务逻辑的基础上进行测试，尤其在逻辑异常流程测试的时候，效果很好。

为什么**Frida**使用**Python**提供**API**, 又用**JavaScript**来调试程序逻辑呢？

- **Frida**核心引擎是用**C**写的， 并且集成了**google** 的**JavaScript**引擎**V8**, 把包含了**V8**的代码注入到目标进程之后，通过一个专门的信息通道，目标进程就可以和这个注入进来的**JavaScript**引擎进行通信了， 这样通过**JavaScript**代码就可以在目标进程中做很多事情了，比如：内存访问、函数**Hook**甚至调用目标进程中的原生函数都可以。
- 通过**Python**和**JS**代码， 你可以快速的开发出无风险的**Frida**脚本，因为如果你的**JS**代码里面错误或者异常的话，**Frida**都会给你捕获到，不会让目标进程**Crash**的，所以叫无风险。
- 如果你非得不想使用**Python**进行开发， 你也可以直接用**C**进行开发， 并且**Frida**还提供了 **Node.js**、 **Swift**、 **.net**、 **Qml**等语言的接口 封装。

建议、提醒

- 这里提供了一些建议、提醒的信息链接，抽空可以去看看
- 具体信息请参考（<https://www.frida.re/docs/home/>）最下面， 我就不贴图了。

快速入门

- 对于急性子的朋友是不是想赶紧看下Frida怎么用呢？这里先给出一个例子吧：

```
~ $ sudo pip install frida
~ $ frida-trace -i "recv*" -i "read*" *twitter*
recv: Auto-generated handler: .../recv.js
# (snip)
recvfrom: Auto-generated handler: .../recvfrom.js
Started tracing 21 functions. Press Ctrl+C to stop.
    39 ms      recv()
   112 ms      recvfrom()
   128 ms      recvfrom()
   129 ms      recvfrom()
```

- 在这个例子中，Frida把自己注入到Twitter，然后枚举进程中已经加载的模块，查找函数名称能匹配到**recv**和**read**的所有函数，然后对这些函数进行Hook。Frida框架会自动生成Hook回调处理脚本。你可以在这个脚本上面随意修改以满足你对需求，修改保存之后，Frida会自动重新加载修改之后对脚本。Frida生成的默认脚本只是打印了函数名，从上图中的输出也可以看的出来。
- 下面我们来看下框架生成的默认脚本**recvfrom.js**

```
/*
 * Auto-generated by Frida. Please modify to match the
 * signature of recvfrom.
 *
 * This stub is somewhat dumb. Future versions of Frida
 * could auto-generate based on OS API references, manpages,
 * etc. (Pull-requests appreciated!)
 *
 * For full API reference, see:
 * http://www.frida.re/docs/javascript-api/
 */

{
  /**
   * Called synchronously when about to call recvfrom.
   *
   * @this {object} - Object allowing you to store state for
   * use in onLeave.
   * @param {function} log - Call this function with a string
   * to be presented to the user.
   * @param {array} args - Function arguments represented as
   * an array of NativePointer objects.
   * For example use Memory.readUtf8String(args[0]) if the
   * first argument is a pointer to a C string encoded as
   * UTF-8.
   * It is also possible to modify arguments by assigning a
   * NativePointer object to an element of this array.
   * @param {object} state - Object allowing you to keep
   * state across function calls.
   * Only one JavaScript function will execute at a time, so
   * do not worry about race-conditions. However, do not use
   * this to store function arguments across onEnter/onLeave,
   * but instead use "this" which is an object for keeping
   * state local to an invocation.
   */
  onEnter: function onEnter(log, args, state) {
    log("recvfrom()");
  },

  /**
   * Called synchronously when about to return from recvfrom.
   *
   * See onEnter for details.
   *
   * @this {object} - Object allowing you to access state
   * stored in onEnter.
   * @param {function} log - Call this function with a string
   * to be presented to the user.
   * @param {NativePointer} retval - Return value represented
   * as a NativePointer object.
   */
}
```

```
    * @param {object} state - Object allowing you to keep
    * state across function calls.
    */
    onLeave: function onLeave(log, retval, state) {
    }
}
```

- 现在，把 **log()** 这一行用下面的代码代替：

```
log("recvfrom(socket=" + args[0].toInt32()
+ ", buffer=" + args[1]
+ ", length=" + args[2].toInt32()
+ ", flags=" + args[3]
+ ", address=" + args[4]
+ ", address_len=" + Memory.readPointer(args[5]).toInt32()
+ ")");
```

- 保存这个文件（Frida会自动加载修改之后的文件），然后去操作一下 **Twitter** 来触发网络访问，你就会看到一些输出了，大概应该是下面这样的结果：

```
8098 ms      recvfrom(socket=70,
                      buffer=0x32cc018, length=65536,
                      flags=0x0,
                      address=0xb0420bd8, address_len=16)
```

- 其实这些还不算啥，Frida最厉害的地方是你可以基于Frida API开发出各种场景下的实用工具，就像Frida-trace一样。

安装Frida

- 从零开始安装使用Frida大概只需要几分钟的时间就可以搞定。如果你安装的过程中发现什么问题，可以告诉我们（请参考原文链接<https://www.frida.re/docs/installation>），看看我们是否有比较好的解决方案。

安装环境要求

- 虽然安装Frida很简单，但是还是需要一点环境上的准备，开始安装之前请确认一下你的系统环境：
 1. Python - 建议使用最新的 3.x 版本
 2. Windows, MacOS, 或者 GNU/Linux

使用Pip进行安装

- 安装Frida最好的方式就是使用PyPi。在命令提示符界面输入下面的命令即可开始安装：

```
$ sudo pip install frida
```

- 这条安装命令，会自动安装Frida并自动解决安装过程中的各种依赖包，如果你在安装过程中遇到了问题，请告诉我们（联系链接请参考原文<https://www.frida.re/docs/installation>），可以帮助后来的人少遇到这样的安装坑。

手动安装

- 你也可以从官网下载已经编译好的安装包进行安装，下载地址：<https://build.frida.re/frida/>

测试一下效果

- 启动一个可以被注入的进程，比如我启动 **cat** 程序，命令如下：

```
$ cat
```

- 让程序就这样处于等待输入状态，如果你是用Windows测试的，你可以启动记事本程序 **notepad.exe** 来代替。
- 需要注意的是，我的这个例子可能不适合高于macOS El Caption的系统，因为这些系统禁止从命令行直接启动系统程序。所以在这些系统下面，你可以把程序先拷贝到一个临时目录再来启动，比如你把 **cat** 程序拷贝到 **/tmp/cat** 然后再启动，类似下面的操作：

```
$ cp /bin/cat /tmp/cat
$ /tmp/cat
```

- 接着启动另外一个命令行，创建一个包含如下内容的文件，命名为 **example.py**:

```
import frida
session = frida.attach("cat")
print([x.name for x in session.enumerate_modules()])
```

- 如果你是 GNU/Linux的系统，你可能还需要执行一下这个命令来允许进行非父子进程之间的**ptrace**操作：

```
$ sudo sysctl kernel.yama.ptrace_scope=0
```

- 现在一切都准备好了，我们来执行 **example.py** 看看效果吧：

```
$ python example.py
```

- 如果不出意外的话，脚本应该有这样的输出结果（根据你的测试平台和库版本不同而不同）：

```
[u'cat', ..., u'ld-2.15.so']
```

基础用法

- Frida的Python API接口 是一种底层接口 的封装，而且功能也是相当的有限，你可以把这些接口 当作是一种底层核心接口 封装的示例代码来看待。强烈建议大家看下 **frida/core.py** 和 **frida/tracer.py** 的代码来学习下底层的具体细节。

模块枚举

- **enumerate_modules()** 函数枚举当前进程中所有加载的模块。
- 执行如下代码：

```
print(s.enumerate_modules())
```

- 应该会看到下面的输出结果：

```
[Module(name="cat", base_address=0x400000, size=20480, path="/bin/cat"), ...]
```

- 其中，**base_address** 是这个模块的加载基地址。

枚举内存块

- **enumerate_ranges(mask)** 函数枚举当前进程中已经映射到进程地址空间中的所有内存块。
- 执行如下代码：

```
print(s.enumerate_ranges('rw-'))
```

- 应该能看到如下输出：

```
[Range(base_address=0x2d4160a06000, size=1019904, protection='rwx'), ...]
```

- 其中，**base_address** 是这个内存块的起始地址。**enumerate_ranges()** 的mask参数表示你要枚举的内存块的保护属性类型，支持 **rwx**，也可以直接填写 - 表示任意类型

内存读写

- **read_bytes(address, n)** 函数在目标进程中从 **address** 这个地址开始，连续读取 **n** 个字节。
- **write_bytes(address, n)** 函数在目标进程中从 **address**这个地址开始，连续写入 **n** 个字节。
- 比如执行如下代码：

```
print(s.read_bytes(49758817247232, 10).encode("hex"))
```

- 执行完毕之后，应该能看到如下结果：

```
454c4602010100000000
```

- 执行如下代码：

```
s.write_bytes(49758817247232, "frida")
```

- 执行完毕之后， 相应的内存块的值会被更新成 **frida** 这个字符串的值

使用姿势

- Frida的动态代码执行功能，主要是在它的核心引擎Gum中用C语言来实现的。一般大家使用Frida进行开发的时候，只需要使用脚本就行了，使用脚本来开发就可以大大缩短开发周期了。比如说GumJS，只需要很少的几行C代码，就可以拉起一个包含JavaScript运行环境的执行环境，这个环境中，你可以访问Gum API，也可以Hook 函数、枚举模块、内存枚举，甚至是调用导出函数。

本篇内容

1. 注入模式
2. 嵌入模式
3. 预加载模式

注入模式

- 大部分情况下，我们都是附加到一个已经运行到进程，或者是在程序启动到时候进行劫持，然后再在目标进程中运行我们的代码逻辑。这种方式就是Frida最常用的使用方式，因此我们的文档也把大部分的篇幅集中在这种方式上。
- 注入模式的大致实现思路是这样的，带有GumJS的Frida核心引擎被打包成一个动态连接库，然后把这个动态连接库注入到目标进程中，同时提供了一个双向通信通道，这样你的控制端就可以和注入的模块进行通信了，在不需要的时候，还可以在目标进程中把这个注入的模块给卸载掉。
- 除了上述功能，Frida还提供了枚举已经安装的App列表，运行的进程列表已经已经连接的设备列表，这里所说的设备列表通常就是frida-server所在的设备。frida-server 是一个守护进程，通过TCP和Frida核心引擎通信，默认的监听端口 是27042。

嵌入模式

- 在实际使用的过程中，你会发现在没有 root 过的iOS Android设备上你是没有办法对进程进行注入的，也就是说第一种注入模式失效了，这个时候嵌入模式就派上用场了。Frida提供了一个动态连接库组件 frida-gadget， 你可以把这个动态库集成到你程序里面来使用Frida的动态执行功能。一旦你集成了gadget，你就可以和你的程序使用Frida进行交互，并且使用 frida-trace 这样的功能，同时也支持从文件自动加载Js文件执行JS逻辑。
- 关于 Gadget 更多功能，请参考原文链接（<https://www.frida.re/docs/modes>）

预加载模式

- 大家应该熟悉 LD_PRELOAD，或者 DYLD_INSERT_LIBRARIES吧， 如果有 JS_PRELOAD 这种东西是不是更爽了呢！事实上对于我们上面讨论的 Gadget 是支持这种模式的， 这才是Gadget 的强大之处，通过这种模式你就可以达到自动加载Js文件并执行的目的了。
- 译者注：这种模式我没测试，我猜测主要目的是说可以执行那些在程序入口 点执行之前就需要执行的JS逻辑，而注入模式是在入口 点跑过之后才能执行JS逻辑的，这个是最大的区别吧，大家最好再看下原文，避免被我误导）

Functions

- 下面向你展示怎么使用**frida**修改参数，进程中调用函数

环境准备

- 创建文件 **hello.c**

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

void
f (int n)
{
    printf ("Number: %d\n", n);
}

int
main (int argc,
      char * argv[])
{
    int i = 0;

    printf ("f() is at %p\n", f);

    while (1)
    {
        f (i++);
        sleep (1);
    }
}
```

- 使用如下命令编译：

```
$ gcc -Wall hello.c -o hello
```

- 然后启动程序，并记录下函数 **f()** 的地址（在这个例子中是**0x400544**）：

```
f() is at 0x400544
Number: 0
Number: 1
Number: 2
...
```

HOOK程序

- 下面的脚本想你展示了注入目标进程获取参数，创建**hook.py**文件：

```
from __future__ import print_function
import frida
import sys

session = frida.attach("hello")
script = session.create_script("""
Interceptor.attach(ptr("%s"), {
    onEnter: function(args) {
        send(args[0].toInt32());
    }
});
""")
print("% int(sys.argv[1], 16))")
def on_message(message, data):
    print(message)
script.on('message', on_message)
script.load()
sys.stdin.read()
```

- 运行脚本，添加参数（如上图输出**0x400544**）：

```
$ python hook.py 0x400544
```

- 每秒输出一条新消息：

```
{u'type': u'send', u'payload': 531}
{u'type': u'send', u'payload': 532}
...
```

修改参数

- 修改进程输入参数，创建**modify.py**脚本：

```
import frida
import sys

session = frida.attach("hello")
script = session.create_script("""
Interceptor.attach(ptr("%s"), {
    onEnter: function(args) {
        args[0] = ptr("1337");
    }
});
""")
script.load()
sys.stdin.read()
```

- 运行脚本（程序运行中....）

```
$ python modify.py 0x400544
```

- 脚本运行起来后，终端停止打印自增，开始输出1337，直到你**Ctrl-D**停止脚本

```
Number: 1281
Number: 1282
Number: 1337
Number: 1337
Number: 1337
Number: 1337
Number: 1337
Number: 1296
Number: 1297
Number: 1298
...
```

调用程序

- 使用**frida**目标进程中的func，创建**call.py**脚本：

```
import frida
import sys

session = frida.attach("hello")
script = session.create_script("""
var f = new NativeFunction(ptr("%s"), 'void', ['int']);
f(1911);
f(1911);
f(1911);
""")
script.load()
```

- 运行脚本：

```
$ python call.py 0x400544
```

- 我们来看看终端输出：

```
Number: 1879
Number: 1911
Number: 1911
```

```
Number: 1911
Number: 1880
...
```

实验2：注入字符串调用程序

- 上面我们测试了注入int，现在我们开始测试注入string，关于其他类型你可以自己尝试。
- 创建**hi.c**

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

int
f (const char * s)
{
    printf ("String: %s\n", s);
    return 0;
}

int
main (int argc,
      char * argv[])
{
    const char * s = "Testing!";

    printf ("f() is at %p\n", f);
    printf ("s is at %p\n", s);

    while (1)
    {
        f (s);
        sleep (1);
    }
}
```

- 创建脚本**stringhook.py**使用Frida将string注入内存，调用f()方法：

```
from __future__ import print_function
import frida
import sys

session = frida.attach("hi")
script = session.create_script("""
var st = Memory.allocUtf8String("TESTMEPLZ!");
var f = new NativeFunction(ptr("%s"), 'int', ['pointer']);
    // In NativeFunction param 2 is the return value type,
    // and param 3 is an array of input types
f(st);
""") % int(sys.argv[1], 16))
def on_message(message, data):
    print(message)
script.on('message', on_message)
script.load()
```

- 看看终端输出

```
...
String: Testing!
String: Testing!
String: TESTMEPLZ!
String: Testing!
String: Testing!
...
```

- 使用类似的方法，比如 **Memory.alloc()**和**Memory.protect()**很容就能操作目标进程的内存，可以创建Python的 **ctypes** 和其他内存结构比如 **structs**，然后以字节数组的方式传递给目标函数。

注入指定格式的内存对象 - 例子：sockaddr_in结构

- 有过网络编程经验的人应该都熟悉这个最常见的C结构。下面给出一个示例程序，程序中创建了一个socket，然后通过5000端口连接上服务器，然后通过这条连接发送了一个字符串 **"Hello there!"**，代码如下：

```
#include <arpa/inet.h>
#include <errno.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>

int
main (int argc,
      char * argv[])
{
    int sock_fd, i, n;
    struct sockaddr_in serv_addr;
    unsigned char * b;
    const char * message;
    char recv_buf[1024];

    if (argc != 2)
    {
        fprintf (stderr, "Usage: %s <ip of server>\n", argv[0]);
        return 1;
    }

    printf ("connect() is at: %p\n", connect);

    if ((sock_fd = socket (AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror ("Unable to create socket");
        return 1;
    }

    bzero (&serv_addr, sizeof (serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons (5000);

    if (inet_pton (AF_INET, argv[1], &serv_addr.sin_addr) <= 0)
    {
        fprintf (stderr, "Unable to parse IP address\n");
        return 1;
    }
    printf ("\nHere's the serv_addr buffer:\n");
    b = (unsigned char *) &serv_addr;
    for (i = 0; i != sizeof (serv_addr); i++)
        printf ("%s%02x", (i != 0) ? " " : "", b[i]);

    printf ("\n\nPress ENTER key to Continue\n");
    while (getchar () == EOF && ferror (stdin) && errno == EINTR)
        ;

    if (connect (sock_fd, (struct sockaddr *) &serv_addr, sizeof (serv_addr)) < 0)
    {
        perror ("Unable to connect");
        return 1;
    }

    message = "Hello there!";
    if (send (sock_fd, message, strlen (message), 0) < 0)
    {
        perror ("Unable to send");
        return 1;
    }

    while (1)
    {
```

```

    n = recv (sock_fd, recv_buf, sizeof (recv_buf) - 1, 0);
    if (n == -1 && errno == EINTR)
        continue;
    else if (n <= 0)
        break;
    recv_buf[n] = 0;

    fputs (recv_buf, stdout);
}

if (n < 0)
{
    perror ("Unable to read");
}

return 0;
}

```

- 这基本上是一个比较标准的网络程序，使用第一个参数作为目标IP地址进行连接。打开一个命令，执行 **nc -l 5000** 这条命令，然后再打开另外一个命令行，执行如下命令： **./client 127.0.0.1**，这个时候你应该就能看到执行**nc**命令的那个窗口 开始显示消息了，你也可以在**nc**窗口 里面发送字符串到**client**程序去。
- 现在我们来玩点好玩的吧，根据前面的描述，我们可以往目标进程中注入字符串以及内存指针，我们也可以用同样的方式来操作 **sockaddr_in** 来达到我们的目的，现在我们来运行程序，看到如下输出：

```

$ ./client 127.0.0.1
connect() is at: 0x400780

Here's the serv_addr buffer:
02 00 13 88 7f 00 00 01 30 30 30 30 30 30 30
Press ENTER key to Continue

```

- 如果你还不熟悉 **sockaddr_in** 这个结构，可以到网上去查找相关的资料，相关资料还是很多的。这里我们重点关注 **0x1388**，也就是10进制的5000，这个就是我们的端口号，如果我们把这个数据改成 **0x1389**，我们就能把客户端的连接重定向到另外一个端口去了，如果我们把接下来的4字节数据也修改的话，那就能把客户端重定向到另外一个IP去了!
- 下面我们来看这个脚本，这个脚本注入了一个指定格式的内存结构，然后劫持了libc.so中的 **connect()** 函数，在劫持的函数中用我们构造的结构体，替换 **connect()** 函数的第一个参数。创建文件 **struct_mod.py**，内容如下：

```

from __future__ import print_function
import frida
import sys

session = frida.attach("client")
script = session.create_script("""
// First, let's give ourselves a bit of memory to put our struct in:
send("Allocating memory and writing bytes...");
var st = Memory.alloc(16);
// Now we need to fill it - this is a bit blunt, but works...
Memory.writeByteArray(st,[0x02, 0x00, 0x13, 0x89, 0x7F, 0x00, 0x00, 0x01, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30]);
// Module.findExportByName() can find functions without knowing the source
// module, but it's slower, especially over large binaries! YMMV...
Interceptor.attach(Module.findExportByName(null, "connect"), {
    onEnter: function(args) {
        send("Injecting malicious byte array:");
        args[1] = st;
    }
    //, onLeave: function(retval) {
    //    retval.replace(0); // Use this to manipulate the return value
    //}
});
""")

# Here's some message handling..
# [ It's a little bit more meaningful to read as output :-D
#   Errors get [!] and messages get [i] prefixes. ]
def on_message(message, data):
    if message['type'] == 'error':
        print("[!] " + message['stack'])
    elif message['type'] == 'send':
        print("[i] " + message['payload'])
    else:
        print(message)

```

```
script.on('message', on_message)
script.load()
sys.stdin.read()
```

- 这个脚本里同时还提到，我们可以使用 **Module.findExportByName()** 这个API来在目标进程中的指定模块中查找指定的导出函数，尤其是在比较大的可执行文件里面，但是在我们的这里例子中，这不是重点。
- 现在使用命令 **./client 127.0.0.1** 把程序执行起来，然后在另外一个命令行中执行 **nc -l 5001**，在第三个命令行中执行 **./struct_mod.py**，一旦我们的脚本执行起来，在 **client** 的那个命令行窗口 里面和 **nc** 命令行窗口 里面就能看到字符串消息了。
- 通过上面这个实验，我们成功的劫持了网络应用程序，并且通过**Frida**修改了程序的原始行为达到了我们不可告人的目的。
- 这个实验证明了**Frida**的真正强大之处，那就是：
 1. 无代码修改
 2. 不需要复杂的逆向
 3. 不需要花大量的时间去反汇编（译者注：感觉跟第2点是同一个东西呢）
- 这里再给出一个关于这篇文章内容的视频演示地址：
 - <https://www.youtube.com/watch?v=cTcM7R872Ls> (得翻墙看)

消息

- 下面我们将向你展示如何发送消息。

环境准备

- 创建文件 **hello.c**

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

void
f (int n)
{
    printf ("Number: %d\n", n);
}

int
main (int argc,
      char * argv[])
{
    int i = 0;

    printf ("f() is at %p\n", f);

    while (1)
    {
        f (i++);
        sleep (1);
    }
}
```

- 编译:

```
$ gcc -Wall hello.c -o hello
```

- 然后启动程序，并记录下函数 **f()** 的地址（在这个例子中是**0x400544**）:

```
f() is at 0x400544
Number: 0
Number: 1
Number: 2
...
```

发送消息

- 下面的脚本展示了如何发送消息到py进程，你可以发送任何JSON格式化的值。

- 创建**send.py**脚本

```
from __future__ import print_function
import frida
import sys

session = frida.attach("hello")
script = session.create_script("send(1337);")
def on_message(message, data):
    print(message)
script.on('message', on_message)
script.load()
sys.stdin.read()
```

- 运行脚本

```
$ python send.py
```

- 终端打印消息如下:


```
{u'type': u'send', u'payload': 1337}
```

- 这就意味着JavaScript中send(1337)在hello已经执行了，使用**Ctrl-D**终端脚本。

处理JavaScript中的错误

- 如果JavaScript在执行中报错，会传递错误信息到Python脚本中。上面的例子中如果把**send(1337)**修改为**send(a)**，Python脚本接受到的错误消息：

```
{u'type': u'error', u'description': u'ReferenceError: a is not defined', u'lineNumber': 1}
```

- 注意类型（error VS send）

接收消息

- 我们来演示一下向JavaScript中发送消息
- 创建文件 **pingpong.py**

```
from __future__ import print_function
import frida
import sys

session = frida.attach("hello")
script = session.create_script("""
    recv('poke', function onMessage(pokeMessage) { send('pokeBack'); });
""")
def on_message(message, data):
    print(message)
script.on('message', on_message)
script.load()
script.post({"type": "poke"})
sys.stdin.read()
```

- 运行脚本

```
$ python pingpong.py
```

- 输出：

```
{u'type': u'send', u'payload': u'pokeBack'}
```

The mechanics of recv()

The recv() method itself is async (non-blocking). The registered callback (onMessage) will receive exactly one message. To receive the next message, the callback must be reregistered with recv().

在目标进程中以阻塞方式接收消息

- 在目标进程中的JavaScript代码可以用阻塞的方式接收来自主控端的消息，下面给一个例子 **rpc.py**:

```
from __future__ import print_function
import frida
import sys

session = frida.attach("hello")
script = session.create_script("""
Interceptor.attach(ptr("%s"), {
    onEnter: function(args) {
        send(args[0].toString());
        var op = recv('input', function(value) {
            args[0] = ptr(value.payload);
        });
        op.wait();
    }
});
"" % int(sys.argv[1], 16))
def on_message(message, data):
    print(message)
```

```
val = int(message['payload'], 16)
script.post({'type': 'input', 'payload': str(val * 2)})
script.on('message', on_message)
script.load()
sys.stdin.read()
```

- 先把 **hello** 这个程序执行起来，然后记录下打印出来的函数地址（比如：**0x400544**）

```
$ python rpc.py 0x400544
```

- 然后观察 **hello** 命令行输出，大致应该如下：

```
Number: 3
Number: 8
Number: 10
Number: 12
Number: 14
Number: 16
Number: 18
Number: 20
Number: 22
Number: 24
Number: 26
Number: 14
```

- **hello**这个程序界面应该一直输出你输入的数据的2倍的值，直到你按下 **Ctrl-D** 结束。

在iOS上使用Frida

- 在iOS设备上，Frida支持两种使用模式，具体使用哪种模式要看你的iOS设备是否已经越狱。

使用场景

- 已越狱机器
- 未越狱机器

已越狱机器

- 在越狱的环境下，是用户权限最大的场景，在这样的环境下你可以很轻松的调用系统服务和基础组件。
- 在这篇教程中，我们来看下如何在iOS设备上进行函数追踪。

设置iOS设备

- 启动 **Cydia** 然后通过 **Manage -> Sources -> Edit -> Add** 这个操作步骤把 **https://build.frida.re** 这个代码仓库加入进去。然后你就可以在 **Cydia** 里面找到 **Frida** 的安装包了，然后你就可以把你的iOS设备插入电脑，并可以开始使用 **Frida** 了，但是现在还没有必要马上插到电脑上。

快速的冒烟测试

- 现在在你的主控端电脑上（Windows、macOS）执行如下命令，确保Frida可以正常工作：

```
$ frida-ps -U
```

Using a Linux-based OS?

As of Frida 6.0.9 there's now usbmuxd integration, so -U works. For earlier Frida versions you can use WiFi and set up an SSH tunnel between localhost:27042 on both ends, and then use -R instead of -U.

- 如果你还没有把你的iOS设备插入到电脑里面（或者插到电脑但是没有被正常识别），那应该会像下面这样提示：

```
Waiting for USB device to appear...
```

- 如果iOS设备已经正常连接了，那应该会看到设备上的进程列表了，大致如下：

```
PID NAME
488 Clock
116 Facebook
312 IRCCloud
1711 LinkedIn
...
```

- 如果到了这一步没有问题，那就可以很开心的继续往下走了。

跟踪Twitter中的加密函数

- OK，现在我们来开始搞点好玩的。在你的设备上启动**Twitter**，然后让它持续的保持在前台，并确保你的机器不会进入睡眠状态。现在在你的主控端的机器上执行如下命令：

```
$ frida-trace -U -i "CCCryptorCreate*" Twitter
Uploading data...
CCCryptorCreate: Auto-generated handler .../CCCryptorCreate.js
CCCryptorCreateFromData: Auto-generated handler .../CCCryptorCreateFromData.js
CCCryptorCreateWithMode: Auto-generated handler .../CCCryptorCreateWithMode.js
CCCryptorCreateFromDataWithMode: Auto-generated handler .../CCCryptorCreateFromDataWithMode.js
Started tracing 4 functions. Press Ctrl+C to stop.
```

- 目前，很多App的加密、解密、哈希算法基本上都是使用 **CCryptorCreate** 和相关的一组加密函数。
- 现在，开始尝试在App里面触发一些网络操作，然后就应该能看到一些输出了，比如我的输出是下面这样的：

```
3979 ms CCCryptorCreate()
3982 ms CCCryptorCreateWithMode()
3983 ms CCCryptorCreate()
3983 ms CCCryptorCreateWithMode()
```

- 现在，你还可以实时的修改JavaScript脚本，然后继续在App里面深挖各种功能。

没有越狱的iOS设备

- 为了让一个App能使用Frida，必须想办法让它加载一个 **.dylib**，就是一个 **Gadget** 模块。
- 在这篇教程里面我们需要配置一下 **xcode** 的编译配置来让你的App可以集成Frida。当然也可以使用相关的工具来修改一个已经编译好的App，比如 **insert_dylib** 这样的工具。

定制你的xCode工程

- 给iOS设备下载最新的 **FridaGadget.dylib** 库，然后给这个库签名：

```
$ mkdir Frameworks
$ cd Frameworks
$ frida_version=x.y.z
$ curl https://github.com/frida/frida/releases/download\
/$frida_version/frida-gadget-$frida_version-ios-univers\
al.dylib.xz | xz -d > FridaGadget.dylib
$ security find-identity -p codesigning -v
  1) A30E15162B3EB979D2572783BF3... "Developer ID Application: ..."
  2) E18BA16DF86318F0ECA4BE17C03... "iPhone Developer: ..."
    2 valid identities found
$ codesign -f -s E18BA16DF86318F0ECA4BE17C03... FridaGadget.dylib
FridaGadget.dylib: replacing existing signature
```

- 在xCode里面打开你的工程，然后把 **Frameworks** 文件夹拖动到 **AppDelegate** 旁边，注意一定要拖动整个文件夹，而不是文件，xCode会提示你 **Choose options for adding these files:**，然后选择 **Copy items if needed** 选项，并且勾选 **Create folder references**，然后点击 **完成**。然后选中项目，切换到 **Build Phases** 页面，展开 **Frameworks** 文件夹，然后把 **FridaGadget.dylib**拖进 **Link Binary With Libraries** 一节，并确保 **Frameworks** 文件夹被加入了 **Copy Bundle Resours** 一节。

快速的冒烟测试

- 在xCode里面启动App，然后就应该能看到下面的输出：

```
Frida: Listening on TCP port 27042
```

- 现在你会发现App处于挂起状态，这是因为集成进来的Frida起作用了，Frida正在等待你来执行任何你感兴趣的API 或者你可以选择直接让程序继续运行。
- 现在Frida正在等待我们操作，并且集成到App内部的**Gadget**和**frida-server**提供的是一样的接口，现在我们尝试枚举一下进程列表试试：

```
$ frida-ps -U
PID NAME
892 Gadget
$
```

- 不同于 **frida-server**，我们只能枚举到一个进程，就是这个App本身。
- 我们还可以使用下面的命令来看下可以运行哪些App：

```
$ frida-ps -Uai
PID  Name      Identifier
---  -
892  Gadget    re.frida.Gadget
$
```

- 到目前为止还不错，现在如果我们调用 **attach()** 函数，目标App就会结束等待状态，继续运行。但是如果一开始使用 **spawn(["re.frida.Gadget"])** 启动App的话，我们这个时候再 **attach()** 的话，这个时候目标App也不会直接运行的，除非我们主动调用 **resume()**，也就是说使用前一种方式我们的代码执行时机晚一点，后一种方式可以在更早的时机执行我们的代码。

跟踪libc函数

- 现在假如你使用xCode启动的程序，现在App处于挂起状态，现在我们尝试开始和Frida交互吧，看下面的例子：

```
$ frida-trace -U -f re.frida.Gadget -i "open*"
Instrumenting functions...
openlog: Auto-generated handler at .../openlog.js
opendev: Auto-generated handler at .../opendev.js
opendir: Auto-generated handler at .../opendir.js
openpty: Auto-generated handler at .../openpty.js
```

```
openx_np: Auto-generated handler at .../openx_np.js
open: Auto-generated handler at .../open.js
open$NOCANCEL: Auto-generated handler at .../open_NOCANCEL.js
open_dprotected_np: Auto-generated handler at .../open_dprotected_np.js
openat: Auto-generated handler at .../openat.js
openat$NOCANCEL: Auto-generated handler at .../openat_NOCANCEL.js
openbyid_np: Auto-generated handler at .../openbyid_np.js
Started tracing 11 functions. Press Ctrl+C to stop.
    /* TID 0xb07 */
    193 ms  open(path=0x1988c4669, oflag=0x0, ...)
    194 ms  open(path=0x16fdeebc6, oflag=0x0, ...)
    195 ms  opendir()
    195 ms    |  open$NOCANCEL()
    195 ms  opendir()
    196 ms    |  open$NOCANCEL()
```

- 现在你可以实时的编辑JavaScript脚本了，然后继续在iOS的App里面深挖。

使用模拟器

- 现在在模拟器中进行测试，就要把上面的命令行中的 **-U** 替换成 **-R**，这样一来底层的内部调用也从 **getusbdevice()** 变成 **getremotedevice()**。

打造自己的工具

- 像是 Frida, Frida-trace 等这些工具确实很有用，但是有时候你会发现你还是需要定制自己更加个性化的功能，那就最好去读一下 **Functions** 和 **Messages** 这两章，比如当你使用到 **frida.attach()** 的时候，其实底层调用的就是 **frida.getusbdevice().attach()**。

在Android上使用Frida

- 在这篇教程里面，我们来演示一下如何在Android设备上进行函数跟踪。

设置你的Android设备

- 在我们开始之前，请确保你的Android设备已经完成**root**操作。我们大部分的实验操作都是在Android4.4版本上进行的，但是Frida本身是支持从4.2到6.0的版本的，但是目前来说对**Art**的支持还是有限的，所以我们建议最后还是用使用Dalvik虚拟机的系统设备或者模拟器来进行尝试。
- 除了一台root设备之外，还需要安装Android SDK工具，因为我们要用到它的 **adb** 工具。
- 首先，下载最新的Android设备上的 **frida-server** （下载地址：<https://github.com/frida/frida/releases>）,然后把这个文件在手机上运行起来：

```
$ adb root # might be required
$ adb push frida-server /data/local/tmp/
$ adb shell "chmod 755 /data/local/tmp/frida-server"
$ adb shell "/data/local/tmp/frida-server &"
```

- 最后要注意的是，确保你的 **frida-server** 是以root权限运行的，在已经root过设备上，先运行**su**，然后再运行程序，即可确保是root权限运行的。
- 接下来，使用**adb**看看设备是否连接正常：

```
$ adb devices -l
```

- 这个操作主要是为了确保**adb**守护进程在你的主控端电脑上运行正常，也同时保证Frida可以正常和设备进行通信。

快速冒烟测试

- 现在，一切环境准备完毕，为了测试Frida是否准备正常，执行如下命令：

```
$ frida-ps -U
```

- 这条命令应该会显示一个进程列表：

```
PID NAME
1590 com.facebook.katana
13194 com.facebook.katana:providers
12326 com.facebook.orca
13282 com.twitter.android
...
```

- 有了以上这些准备，我们就可以顺畅的往下走了。

跟踪Chrome里的Open()函数

- 现在我们就开始吧，在你的Android机器上启动Chrome App然后在你主控端电脑上运行如下命令：

```
$ frida-trace -U -i open com.android.chrome
Uploading data...
open: Auto-generated handler .../linker/open.js
open: Auto-generated handler .../libc.so/open.js
Started tracing 2 functions. Press Ctrl+C to stop.
```

- 现在你可以任意的在Chrome App里面任意玩耍啦，现在可以看到主控端的命令行关于 **open()** 的输出了：

```
1392 ms open()
1403 ms open()
1420 ms open()
```

- 现在可以实时的修改JavaScript脚本，并且在Android的App里面深挖了。

构建自己的工具

- 像是 Frida, Frida-trace 等这些工具确实很有用，但是有时候你会发现你还是需要定制自己更加个性化的功能，那就最好去读一下 **Functions** 和 **Messages** 这两章，比如当你使用到 **frida.attach()** 的时候，其实底层调用的就是 **frida.getusbdevice().attach()**。

Android上示例一则

Android CTF例子

- 在这个示例中，强烈建议大家使用Android4.4 x86的模拟器，这个示例工具脚本是基于 **SECTION Quals CTF 2015 APK1**的，APK下载地址可以参考原文（<https://www.frida.re/docs/examples/android/>）
- 创建 **ctf.py** 然后执行如下命令**python ctf.py**:

```
import frida, sys

def on_message(message, data):
    if message['type'] == 'send':
        print("[*] {}".format(message['payload']))
    else:
        print(message)

jscode = """
Java.perform(function () {
    // Function to hook is defined here
    var MainActivity = Java.use('com.example.seccon2015.rock_paper_scissors.MainActivity');

    // Whenever button is clicked
    MainActivity.onClick.implementation = function (v) {
        // Show a message to know that the function got called
        send('onClick');

        // Call the original onClick handler
        this.onClick(v);

        // Set our values after running the original onClick handler
        this.m.value = 0;
        this.n.value = 1;
        this.cnt.value = 999;

        // Log to the console that it's done, and we should have the flag!
        console.log('Done:' + JSON.stringify(this.cnt));
    };
});
"""

process = frida.get_usb_device().attach('com.example.seccon2015.rock_paper_scissors')
script = process.create_script(jscode)
script.on('message', on_message)
print('[*] Running CTF')
script.load()
sys.stdin.read()
```

JavaScript API

目录

1. Global
2. console
3. rpc
4. Frida
5. Process
6. Module
7. ModuleMap
8. Memory
9. MemoryAccessMonitor
10. Thread
11. Int64
12. UInt64
13. NativePointer
14. NativeFunction
15. NativeCallback
16. SystemFunction
17. Socket
18. SocketListener
19. SocketConnection
20. IOStream
21. InputStream
22. OutputStream
23. UnixInputStream
24. UnixOutputStream
25. Win32InputStream
26. Win32OutputStream
27. File
28. SQLiteDatabase
29. SqliteStatement
30. Interceptor
31. Stalker
32. ApiResolver
33. DebugSymbol
34. Instruction
35. ObjC
36. Java
37. WeakRef
38. X86Writer
39. X86Relocator
40. X86enumtypes
41. ArmWriter
42. ArmRelocation
43. ThumbWriter
44. ThumbRelocator
45. ARMenumtypes
46. Arm64Writer
47. Arm64Relocator
48. AArch64enumtypes
49. MipsWriter
50. MipsRelocator
51. Mipsenumtypes

Global

- **hexdump(target[, options]):** 把一个 **ArrayBuffer** 或者 **NativePointer** 的target变量，附加一些 **options** 属性，按照指定格式进行输出，比如：

```
var libc = Module.findBaseAddress('libc.so');
var buf = Memory.readByteArray(libc, 64);
console.log(hexdump(buf, {
```




```
offset: 0,
length: 64,
header: true,
ansi: true
}});
```

```

      0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
00000000  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF.....
00000010  03 00 28 00 01 00 00 00 00 00 00 00 34 00 00 00  ..(.....4...
00000020  34 a8 04 00 00 00 00 05 34 00 20 00 08 00 28 00  4.....4. ...(.
00000030  1e 00 1d 00 06 00 00 00 34 00 00 00 34 00 00 00  .....4...4...
```

- **int64(v): new Int64(v)** 的缩写格式
- **uint64(v): new UInt64(v)** 的缩写格式
- **ptr(s): new NativePointer(s)** 的缩写格式
- **NULL ptr("0")** 的缩写格式
- **recv([type,]callback):** 注册一个回调，当下次有消息到来的时候会收到回调消息，可选参数 **type** 相当于一个过滤器，表示只接收这种类型的消息。需要注意的一点是，这个消息回调是一次性的，收到一个消息之后，如果需要继续接收消息，那就需要重新调用一个 **recv**
- **send(message[, data]):** 从目标进程中往主控端发 **message**（必须是可以序列换成Json的），如果你还有二进制数据需要附带发送（比如使用 **Memory.readByteArray** 拷贝的内存数据），就把这个附加数据填入 **data** 参数，但是有个要求，就是 **data** 参数必须是一个 **ArrayBuffer** 或者是一个整形数组（数值是 0-255）

 **Performance considerations**

While *send()* is asynchronous, the total overhead of sending a single message is not optimized for high frequencies, so that means Frida leaves it up to you to batch multiple values into a single *send()*-call, based on whether low delay or high throughput is desired.

<http://blog.csdn.net/freakishfox>

- **setTimeout(fn, delay):** 在延迟 **delay** 毫秒之后，调用 **fn**，这个调用会返回一个ID，这个ID可以传递给 **clearTimeout** 用来进行调用取消。
- **clearTimeout(id):** 取消通过 **setTimeout** 发起的延迟调用
- **setInterval(fn, delay):** 每隔 **delay** 毫秒调用一次 **fn**，返回一个ID，这个ID可以传给 **clearInterval** 进行调用取消。
- **clearInterval(id):** 取消通过 **setInterval** 发起的调用

console

- **console.log(line), console.warn(line), console.error(line):** 向标准输入输出界面写入 **line** 字符串。 比如：使用 **Frida-Python** 的时候就输出到 **stdout** 或者 **stderr**，使用 **frida-qml** 的时候则输出到 **qDebug**，如果输出的是一个ArrayBuffer，会以默认参数自动调用 **hexdump** 进行格式化输出。

rpc

- **rpc.exports:** 可以在你的程序中导出一些 **RPC-Style API**函数，**Key**指定导出的名称，**Value**指定导出的函数，函数可以直接返回一个值，也可以是异步方式以 **Promise** 的方式返回，举个例子：

```
'use strict';

rpc.exports = {
  add: function (a, b) {
    return a + b;
  },
  sub: function (a, b) {
    return new Promise(resolve => {
      setTimeout(() => {
        resolve(a - b);
      }, 100);
    });
  }
};
```

- 对于Python主控端可以使用下面这样的脚本使用导出的函数：

```
import codecs
import frida

def on_message(message, data):
    if message['type'] == 'send':
        print(message['payload'])
    elif message['type'] == 'error':
        print(message['stack'])

session = frida.attach('iTunes')
with codecs.open('./agent.js', 'r', 'utf-8') as f:
    source = f.read()
script = session.create_script(source)
script.on('message', on_message)
script.load()
print(script.exports.add(2, 3))
print(script.exports.sub(5, 3))
session.detach()
```

- 在上面这个例子里面，我们使用 **script.on('message', on_message)** 来监控任何来自目标进程的消息，消息监控可以来自 **script** 和 **session** 两个方面，比如，如果你想要监控目标进程的退出，可以使用下面这个语句 **session.on('detached', my_function)**

Frida

- **Frida.version**: 包含当前Frida的版本信息

Process

- **Process.arch**: CPU架构信息，取值范围：**ia32**、**x64**、**arm**、**arm64**
- **Process.platform**: 平台信息，取值范围：**windows**、**darwin**、**linux**、**qnx**
- **Process.pageSize**: 虚拟内存页面大小，主要用来辅助增加脚本可移植性
- **Process.pointerSize**: 指针占用的内存大小，主要用来辅助增加脚本可移植性
- **Process.codeSigningPolicy**: 取值范围是 **optional** 或者 **required**，后者表示Frida会尽力避免修改内存中的代码，并且不会执行未签名的代码。默认值是 **optional**，除非是在 **Gadget** 模式下通过配置文件来使用 **required**，通过这个属性可以确定 **Interceptor API** 是否有限制，确定代码修改或者执行未签名代码是否安全。(译者注：这个目前没有实验清楚，可以参考原文)
- **Process.isDebuggerAttached()**: 确定当前是否有调试器附加
- **Process.getCurrentThreadId()**: 获取当前线程ID
- **Process.enumerateThreads(callbacks)**: 枚举所有线程，每次枚举到一个线程就执行回调类callbacks:
 - **onMatch: function(thread)**: 当枚举到一个线程的时候，就调用这个函数，其中**thread**参数包含：
 1. id, 线程ID
 2. state, 线程状态，取之范围是 **running**, **stopped**, **waiting**, **uninterruptible**, **halted**
 3. context, 包含 **pc**, **sp**，分别代表 EIP/RIP/PC 和 ESP/RSP/SP，分别对应于 ia32/x64/arm平台，其他的寄存器也都有，比如 **eax**, **rax**, **r0**, **x0** 等。
 4. 函数可以直接返回 **stop** 来停止枚举。
 - **onComplete: function()**: 当所有的线程枚举都完成的时候调用。
- **Process.enumerateThreadSync()**: **enumerateThreads()**的同步版本，返回线程对象数组
- **Process.findModuleByAddress(address)**, **Process.getModuleByAddress(address)**, **Process.findModuleByName(name)**, **Process.getModuleByName(name)**: 根据地址或者名称来查找模块，如果找不到这样的模块，**find**开头的函数返回 **null**，**get**开头的函数会抛出异常。
- **Process.enumerateModules(callbacks)**: 枚举已经加载的模块，枚举到模块之后调用回调对象:
 - **onMatch: function(module)**: 枚举到一个模块的时候调用，**module**对象包含如下字段:
 1. name, 模块名
 2. base, 基地址
 3. size, 模块大小
 4. path, 模块路径
 5. 函数可以返回 **stop** 来停止枚举。
 - **onComplete: function()**: 当所有的模块枚举完成的时候调用。
- **Process.enumerateModulesSync()**: **enumerateModules()** 函数的同步版本，返回模块对象数组
- **Process.findRangeByAddress(address)**, **Process.getRangeByAddress(address)**: 返回一个内存块对象，如果在这个**address**找不到内存块对象，那么 **findRangeByAddress()** 返回 **null** 而 **getRangeByAddress** 则抛出异常。
- **Process.numerateRanges(protection | specifier, callbacks)**: 枚举指定 **protection** 类型的内存块，以指定形式的字符串给出：**rwX**，而 **rw-** 表示最少是可读可写，也可以用分类符，里面包含 **protection** 这个Key，取值就是前面提到的**rwX**，还有一个 **coalesce** 这个Key，表示是否要把位置相邻并且属性相同的内存块合并给出结果，枚举过程中回调 **callbacks** 对象:
 - **onMatch: function(range)**: 每次枚举到一个内存块都回调回来，其中**Range**对象包含如下属性:
 1. base: 基地址

- 2. **size**: 内存块大小
- 3. **protection**: 保护属性
- 4. **file**: （如果有的话）内存映射文件: 4.1 **path**, 文件路径 4.2 **offset**, 文件内偏移
- 5. 如果要停止枚举过程, 直接返回 **stop** 即可
 - **onComplete: function()**: 所有内存块枚举完成之后会回调
- **Process.enumerateRangesSync(protection | specifier): enumerateRanges()** 函数的同步版本, 返回内存块数组
- **Process.enumerateMallocRanges(callbacks)**: 用于枚举在系统堆上申请的内存块
- **Process.enumerateMallocRangesSync(protection): Process.enumerateMallocRanges()** 的同步版本
- **Process.setExceptionHandler(callback)**: 在进程内安装一个异常处理函数 (Native Exception), 回调函数会在目标进程本身的异常处理函数之前调用, 回调函数只有一个参数 **details**, 包含以下几个属性:
 - **type**, 取值为下列之一:
 1. abort
 2. access-violation
 3. guard-page
 4. illegal-instruction
 5. stack-overflow
 6. arithmetic
 7. breakpoint
 8. single-step
 9. system
 - **address**, 异常发生的地址, NativePointer
 - **memory**, 如果这个对象不为空, 则会包含下面这些属性:
 1. operation: 引发一场的操作类型, 取值范围是 read, write 或者 execute
 2. address: 操作发生异常的地址, NativePointer
 - **context**, 包含 **pc** 和 **sp** 的NativePointer, 分别代表指令指针和堆栈指针
 - **nativeContext**, 基于操作系统定义的异常上下文信息的NativePointer, 在 **context** 里面的信息不够用的时候, 可以考虑用这个指针, 但是一般不建议使用 (译者注: 估计是考虑到可移植性或者稳定性)
 - 捕获到异常之后, 怎么使用就看你自己了, 比如可以把异常信息写到日志里面, 然后发送个信息给主控端, 然后同步等待主控端的响应之后处理, 或者直接修改异常信息里面包含的寄存器的值, 尝试恢复掉异常, 继续执行。如果你处理了异常信息, 那么这个异常回调里面你要返回 **true**, Frida会把异常交给进程异常处理函数处理, 如果到最后都没人去处理这个异常, 就直接结束目标进程。

Module

- **Module.enumerateImports(name, callbacks)**: 枚举模块 **name** 的导入表, 枚举到一个导入项的时候回调callbacks, callbacks包含下面2个回调:
 - **onMatch: function(imp)**: 枚举到一个导入项到时候会被调用, **imp**包含如下的字段:
 1. **type**, 导入项的类型, 取值范围是 **function**或者**variable**
 2. **name**, 导入项的名称
 3. **module**, 模块名称
 4. **address**, 导入项的绝对地址
 5. 以上所有的属性字段, 只有 **name** 字段是一定会有, 剩余的其他字段不能保证都有, 底层会尽量保证每个字段都能给出数据, 但是不能保证一定能拿到数据, onMatch函数可以返回字符串 **stop** 表示要停止枚举。
 - **onComplete: function()**: 当所有的导入表项都枚举完成的时候会回调
- **Module.enumerateImportsSync(name): enumerateImports()** 的同步版本
- **Module.enumerateExports(name, callbacks)**: 枚举指定模块 **name** 的导出表项, 结果用 **callbacks** 进行回调:
 - **onMatch: function(exp)**: 其中 exp 代表枚举到的一个导出项, 包含如下几个字段:
 1. **type**, 导出项类型, 取值范围是 **function**或者**variable**
 2. **name**, 导出项名称
 3. **address**, 导出项的绝对地址, NativePointer
 4. 函数返回 **stop** 的时候表示停止枚举过程
 - **onComplete: function()**: 枚举完成回调
- **Module.enumerateExportsSync(): Module.enumerateExports()** 的同步版本
- **Module.enumerateSymbols(name, callbacks)**: 枚举指定模块中包含的符号, 枚举结果通过回调进行通知:
 - **onMatch: function(sym)**: 其中 **sym** 包含下面几个字段:
 - **isGlobal**, 布尔值, 表示符号是否全局可见
 - **type**, 符号的类型, 取值是下面其中一种:
 1. unknown
 2. undefined
 3. absolute
 4. section
 5. prebound-undefined
 6. indirect
 - **section**, 如果这个字段不为空的话, 那这个字段包含下面几个属性:

- 1. **id**, 小节序号, 段名, 节名
- 2. **protection**, 保护属性类型, **rwX**这样的属性
 - **name**, 符号名称
 - **address**, 符号的绝对地址, NativePointer
 - 这个函数返回 **stop** 的时候, 表示要结束枚举过程
- **Module.enumerateSymbolsSync(name)**: **Module.enumerateSymbols()** 的同步版本



Module.enumerateSymbols() is only available on i/macOS for now

We would love to support this on the other platforms too, so if you find this useful and would like to help out, please get in touch. You may also find the DebugSymbol API adequate, depending on your use-case.

<http://blog.csdn.net/freakishfox>

- **Module.enumerateRanges(name, protection, callbacks)**: 功能基本等同于 **Process.enumerateRanges()**, 只不过多了一个模块名限定了枚举的范围
- **Module.enumerateRangesSync(name, protection)**: **Module.enumerateRanges()** 的同步版本
- **Module.findBaseAddress(name)**: 获取指定模块的基地址
- **Module.findExportByName(module | null, exp)**: 返回模块**module** 内的导出项的绝对地址, 如果模块名不确定, 第一个参数传入 null, 这种情况下会增大查找开销, 尽量不要使用。

ModuleMap

- **new ModuleMap([filter])**: 可以理解为内存模块快照, 主要目的是可以作为一个模块速查表, 比如你可以用这个快照来快速定位一个具体的地址是属于哪个模块。创建**ModuleMap**的时候, 就是对目标进程当前加载的模块的信息作一个快照, 后续想要更新这个快照信息的时候, 可以使用 **update** 进行更新。这个 **filter** 参数是可选的, 主要是用来过滤你关心的模块, 可以用来缩小快照的范围（注意：**filter**是过滤函数, 不是字符串参数），为了让模块进入这个快照里, 过滤函数的返回值要设置为true, 反之设为false, 如果后续内存中的模块加载信息更新了, 还会继续调用这个filter函数。
- **has(address)**: 检查 **address** 这个地址是不是包含在ModuleMap里面, 返回bool值
- **find(address), get(address)**: 返回 **address** 地址所指向的模块对象详细信息, 如果不存在 **find** 返回null, **get** 直接会抛出异常, 具体的返回的对象的详细信息, 可以参考 **Process.enumerateModules()**
- **findName(address), getName(address), findPath(address), getPath(address)**: 功能跟 **find()**, **get()** 类似, 但是只返回 **name** 或者 **path** 字段, 可以省点开销
- **update()**: 更新ModuleMap信息, 如果有模块加载或者卸载, 最好调用一次, 免得使用旧数据。

Memory

- **Memory.scan(address, size, pattern, callbacks)**: 在 **address** 开始的地址, **size** 大小的内存范围内以 **pattern** 这个模式进行匹配查找, 查找到一个内存块就回调callbacks, 各个参数详细如下:
 - **pattern** 比如使用**13 37 ?? ff**来匹配0x13开头, 然后跟着0x37, 然后是任意字节内容, 接着是0xff这样的内存块
 - **callbacks** 是扫描函数回调对象:
 1. **onMatch: function(address, size)**: 扫描到一个内存块, 起始地址是**address**, 大小**size**的内存块, 返回 **stop** 表示停止扫描
 2. **onError: function(reason)**: 扫描内存的时候出现内存访问异常的时候回调
 3. **onComplete: function()**: 内存扫描完毕的时候调用
- **Memory.scanSync(address, size, pattern)**: 内存扫描 **scan()** 的同步版本
- **Memory.alloc(size)**: 在目标进程中的堆上申请**size**大小的内存, 并且会按照**Process.pageSize**对齐, 返回一个NativePointer, 并且申请的内存如果在JavaScript里面没有对这个内存的使用的时候会自动释放的。也就是说, 如果你不想要这个内存被释放, 你需要自己保存一份对这个内存块的引用。
- **Memory.copy(dust, src, n)**: 就像是memcpy
- **Memory.dup(address, size)**: 等价于 **Memory.alloc()**和**Memory.copy()**的组合。
- **Memory.protect(address, size, protection)**: 更新address开始, size大小的内存块的保护属性, **protection** 的取值参考 **Process.enumerateRanges()**, 比如: **Memory.protect(ptr("0x123", 4096, 'rw-'))**;
- **Memory.patchCode(address, size, apply)**: **apply**是一个回调函数, 这个函数是用来在 **address** 开始的地址和 **size** 大小的地方开始Patch的时候调用, 回调参数是一个NativePointer的可写指针, 需要在**apply**回调函数里面要完成patch代码的写入, 注意, 这个可写的指针地址不一定和上面的**address**是同一个地址, 因为在有的系统上是不允许直接写入代码段的, 需要先写入到一个临时的地方, 然后在影射到响应代码段上, （比如iOS上, 会引发进程丢失 CS_VALID 状态），比如:

```
var getLivesLeft = Module.findExportByName('game-engine.so', 'get_lives_left');
var maxPatchSize = 64; // Do not write out of bounds, may be a temporary buffer!
Memory.patchCode(getLivesLeft, maxPatchSize, function (code) {
```

```
var cw = new X86Writer(code, { pc: getLivesLeft });
cw.putMovRegU32('eax', 9000);
cw.putRet();
cw.flush();
});
```

- 下面是接着是一些数据类型读写：
 1. Memory.readPointer(address)
 2. Memory.writePointer(address, ptr)
 3. Memory.readS8, Memory.readU8
 4. ...

MemoryAccessMonitor



MemoryAccessMonitor is only available on Windows for now

We would love to support this on the other platforms too, so if you find this useful and would like to help out, please get in touch.

<http://blog.csdn.net/freakishfox>

- **MemoryAccessMonitor.enable(ranges, callbacks):** 监控一个或多个内存块的访问，在触发到内存访问的时候发出通知。**ranges** 要么是一个单独的内存块，要么是一个内存块数组，每个内存块包含如下属性：
 - **base:** 触发内存访问的NativePointer地址
 - **size:** 被触发访问的内存块的大小
 - **callbacks:** 回调对象结构：
 - **onAccess: function(details):** 发生访问的时候同步调用这个函数，**details**对象包含如下属性：
 - **operation:** 触发内存访问的操作类型，取值范围是 **read**, **write** 或者 **execute**
 - **from:** 触发内存访问的指令地址，NativePointer
 - **address:** 被访问的内存地址
 - **rangeIndex:** 被访问的内存块的索引，就是调用**MemoryAccessMonitor.enable()**的时候指定的内存块序号
 - **pageIndex:** 在被监控内存块范围内的页面序号
 - **pagesCompleted:** 到目前为止已经发生过内存访问的页面的个数（已经发生过内存访问的页面将不再进行监控）
 - **pagesTotal:** 初始指定的需要监控的内存页面总数
- **MemoryAccessMonitor.disable():** 停止监控页面访问操作

Thread

- **Thread.backtrace([context, backtracer]):** 抓取当前线程的调用堆栈，并以 **NativePointer** 指针数组的形式返回。
 1. 如果你是在 **Interceptor.onEnter**或者**Interceptor.onLeave** 中调用这个函数的话，那就必须要把 **this.context** 作为参数传入，这样就能拿到更佳精准的堆栈调用信息，如果省略这个参数不传，那就意味着从当前堆栈的位置开始抓取，这样的抓取效果可能不会很好，因为有不少V8引擎的栈帧的干扰。
 2. 第二个可选参数 **backtracer**，表示使用哪种类型的堆栈抓取算法，目前的取值范围是 **Backtracer.FUZZY** 和 **Backtracer.ACCURATE**，目前后者是默认模式。精确抓取模式下，如果如果程序是调试器友好（比如是标准编译器编译的结果，没有什么反调试技巧）或者有符号表的支持，抓取效果是最好的，而模糊抓取模式下，抓取器会在堆栈上尝试抓取，并且会猜测里面包含的返回地址，也就是说中间可能包含一些错误的信息，但是这种模式基本能在任何二进制程序里面工作：

```
var f = Module.findExportByName("libcommonCrypto.dylib",
    "CCCryptorCreate");
Interceptor.attach(f, {
    onEnter: function (args) {
        console.log("CCCryptorCreate called from:\n" +
            Thread.backtrace(this.context, Backtracer.ACCURATE)
                .map(DebugSymbol.fromAddress).join("\n") + "\n");
    }
});
```

- **Thread.sleep(delay):** 线程暂停 **delay** 秒执行

下篇继续

- 一篇文章写太长发现浏览器容易崩溃，所以下篇继续

JavaScript API

Int64

- **new Int64(v)**: 以v为参数，创建一个Int64对象，v可以是一个数值，也可以是一个字符串形式的数值表示，也可以使用 **Int64(v)** 这种简单的方式。
- **add(rhs), sub(rhs), and(rhs), or(rhs), xor(rhs)**: Int64相关的加减乘除。
- **shr(n), shl(n)**: Int64相关的左移、右移操作
- **compare(rhs)**: Int64的比较操作，有点类似 String.localCompare()
- **toNumber()**: 把Int64转换成一个实数
- **toString([radix = 10])**: 按照一定的数值进制把Int64转成字符串，默认是十进制

UInt64

- 可以直接参考Int64

NativePointer

- 可以直接参考Int64

NativeFunction

- **new NativeFunction(address, returnType, argTypes[, abi])**: 在**address**（使用**NativePointer**的格式）地址上创建一个NativeFunction对象来进行函数调用，**returnType** 指定函数返回类型，**argTypes** 指定函数的参数类型，如果不是系统默认类型，也可以选择性的指定 **abi** 参数，对于可变类型的函数，在固定参数之后使用 **"..."** 来表示。
- 类和结构体
 - 在函数调用的过程中，类和结构体是按值传递的，传递的方式是使用一个数组来分别指定类和结构体的各个字段，理论上为了和需要的数组对应起来，这个数组是可以支持无限嵌套的，结构体和类构造完成之后，使用**NativePointer**的形式返回的，因此也可以传递给 **Interceptor.attach()** 调用。
 - 需要注意的是， 传递的数组一定要和需要的参数结构体严格吻合，比如一个函数的参数是一个3个整形的结构体，那参数传递的时候一定要是 **['int', 'int', 'int']**，对于一个拥有虚函数的类来说，调用的时候，第一个参数一定是虚表指针。

- **Supported Types**

- void
- pointer
- int
- uint
- long
- ulong
- char
- uchar
- float
- double
- int8
- uint8
- int16
- uint16
- int32
- uint32
- int64
- uint64

- **Supported ABIs**

- default
- Windows 32-bit:
 - sysv
 - stdcall
 - thiscall
 - fastcall
 - mscdecl
- Windows 64-bit:

- win64
- UNIX x86:
 - sysv
 - unix64
- UNIX ARM:
 - sysv
 - vfp

NativeCallback

- **new NativeCallback(func, returnType, argTypes[, abi]):** 使用JavaScript函数 **func** 来创建一个Native函数，其中**returnType**和**argTypes**分别指定函数的返回类型和参数类型数组。如果不想使用系统默认的 **abi** 类型，则可以指定 **abi** 这个参数。关于**argTypes**和**abi**类型，可以查看 **NativeFunction**来了解详细信息，这个对象的返回类型也是**NativePointer**类型，因此可以作为 **Interceptor.replace** 的参数使用。

SystemFunction

- **new SystemFunction(address, returnType, argTypes[, abi]):** 功能基本和**NativeFunction**一致，但是使用这个对象可以获取到调用线程的last error状态，返回值是对平台相关的数值的一层封装，为**value**对象，比如是对这两个值的封装， **errno**(UNIX) 或者 **lastError**(Windows)。

Socket

SocketListener

SocketConnection

IOStream

InputStream

OutputStream

UnixInputStream

UnixOutputStream

Win32InputStream

Win32OutputStream

File

SqliteDatabase

SqliteStatement

Interceptor

- **Interceptor.attach(target, callbacks):** 在target指定的位置进行函数调用拦截，**target**是一个NativePointer参数，用来指定你想要拦截的函数的地址，有一点需要注意，在32位ARM机型上，**ARM**函数地址末位一定是0（2字节对齐），**Thumb**函数地址末位一定1（单字节对齐），如果使用的函数地址是用Frida API获取的话，那么API内部会自动处理这个细节（比如：**Module.findExportByName()**）。其中**callbacks**参数是一个对象，大致结构如下：
 - **onEnter: function(args):** 被拦截函数调用之前回调，其中原始函数的参数使用**args**数组（NativePointer对象数组）来表示，可以在这里修改函数的调用参数。
 - **onLeave: function(retval):** 被拦截函数调用之后回调，其中**retval**表示原始函数的返回值，**retval**是从NativePointer继承来的，是对原始返回值的一个封装，你可以使用**retval.replace(1337)**调用来修改返回值的内容。需要注意的一点是，**retval**对象只在 **onLeave**函数作用域范围内有效，因此如果你要保存这个对象以备后续使用的话，一定要使用深拷贝来保存对象，比如：**ptr(retval.toString())**。
- 事实上Frida可以在代码的任意位置进行拦截，但是这样一来 **callbacks** 回调的时候，因为回调位置有可能不在函数的开头，这样**onEnter**这样的回调参数Frida只能尽量的保证（比如拦截的位置前面的代码没有修改过传入的参数），不能像在函数头那样可以确保正确。
- 拦截器的**attach**调用返回一个监听对象，后续取消拦截的时候，可以作为 **Interceptor.detach()** 的参数使用。
- 还有一个比较方便的地方，那就是在回调函数里面，包含了一个隐藏的 **this** 的线程**tls**对象，方便在回调函数中存储变量，比如可以在 **onEnter** 中保存值，然后在 **onLeave** 中使用，看一个例子：


```

Interceptor.attach(Module.findExportByName("libc.so", "read"), {
  onEnter: function (args) {
    this.fileDescriptor = args[0].toInt32();
  },
  onLeave: function (retval) {
    if (retval.toInt32() > 0) {
      /* do something with this.fileDescriptor */
    }
  }
});

```

- 另外，**this** 对象还包含了一些额外的比较有用的属性：
 - **returnAddress**: 返回NativePointer类型的 **address** 对象
 - **context**: 包含 **pc**, **sp**, 以及相关寄存器比如 **eax**, **ebx**等, 可以在回调函数中直接修改
 - **errno**: (UNIX) 当前线程的错误值
 - **lastError**: (Windows) 当前线程的错误值
 - **threadId**: 操作系统线程Id
 - **depth**: 函数调用层次深度
 - 看个例子:

```

Interceptor.attach(Module.findExportByName(null, 'read'), {
  onEnter: function (args) {
    console.log('Context information:');
    console.log('Context   : ' + JSON.stringify(this.context));
    console.log('Return    : ' + this.returnAddress);
    console.log('ThreadId  : ' + this.threadId);
    console.log('Depth     : ' + this.depth);
    console.log('Errornr   : ' + this.err);

    // Save arguments for processing in onLeave.
    this.fd = args[0].toInt32();
    this.buf = args[1];
    this.count = args[2].toInt32();
  },
  onLeave: function (result) {
    console.log('-----')
    // Show argument 1 (buf), saved during onEnter.
    numBytes = result.toInt32();
    if (numBytes > 0) {
      console.log(hexdump(this.buf, { length: numBytes, ansi: true }));
    }
    console.log('Result    : ' + numBytes);
  }
});

```



Performance considerations

The callbacks provided have a significant impact on performance. If you only need to inspect arguments but do not care about the return value, or the other way around, make sure you omit the callback that you don't need; i.e. avoid putting your logic in *onEnter* and leaving *onLeave* in there as an empty callback.

On an iPhone 5S the base overhead when providing just *onEnter* might be something like 6 microseconds, and 11 microseconds with both *onEnter* and *onLeave* provided.

Also be careful about intercepting calls to functions that are called a bazillion times per second; while *send()* is asynchronous, the total overhead of sending a single message is not optimized for high frequencies, so that means Frida leaves it up to you to batch multiple values into a single *send()*-call, based on whether low delay or high throughput is desired.

<http://blog.csdn.net/freakishfox>

- **Interceptor.detachAll():** 取消之前所有的拦截调用
- **Interceptor.replace(target, replacement):** 函数实现代码替换，这种情况主要是你想要完全替换掉一个原有函数的实现的时候来使用，注意 replacement 参数使用 JavaScript 形式的一个 NativeCallback 来实现，后续如果想要取消这个替换效果，可以使用 **Interceptor.revert** 调用来实现，如果你还想在你自己的替换函数里面继续调用原始的函数，可以使用以 target 为参数的 NativeFunction 对象来调用，来看一个例子：

```
var openPtr = Module.findExportByName("libc.so", "open");
var open = new NativeFunction(openPtr, 'int', ['pointer', 'int']);
Interceptor.replace(openPtr, new NativeCallback(function (pathPtr, flags) {
    var path = Memory.readUtf8String(pathPtr);
    log("Opening '" + path + "'");
    var fd = open(pathPtr, flags);
    log("Got fd: " + fd);
    return fd;
}, 'int', ['pointer', 'int']));
```

- **Interceptor.revert(target):** 还原函数的原始实现逻辑，即取消前面的 **Interceptor.replace** 调用
- **Interceptor.flush():** 确保之前的内存修改操作都执行完毕，并切已经在内存中发生作用，只要少数几种情况需要这个调用，比如你刚执行了 **attach()** 或者 **replace()** 调用，然后接着想要使用 NativeFunction 对象对函数进行调用，这种情况就需要调用 flush。正常情况下，缓存的调用操作会在当前线程即将离开 JavaScript 运行时环境或者调用 **send()** 的时候自动进行 flush 操作，也包括那些底层会调用 **send()** 操作的函数，比如 RPC 函数，或者任何的 console API

Stalker

- **Stalker.follow([threadId, options]):** 开始监视线程ID为 threadId（如果是本线程，可以省略）的线程事件，举个例子：

```
Stalker.follow(Process.getCurrentThreadId(), {
  events: {
    call: true, // CALL instructions: yes please

    // Other events:
    ret: false, // RET instructions
    exec: false, // all instructions: not recommended as it's
                // a lot of data
    block: false, // block executed: coarse execution trace
    compile: false // block compiled: useful for coverage
  },

  //
```

```

// Only specify one of the two following callbacks.
// (See note below.)
//

//
// onReceive: Called with `events` containing a binary blob
//             comprised of one or more GumEvent structs.
//             See `gumevent.h` for details about the
//             format. Use `Stalker.parse()` to examine the
//             data.
//
//onReceive: function (events) {
//},
//

//
// onCallSummary: Called with `summary` being a key-value
//                  mapping of call target to number of
//                  calls, in the current time window. You
//                  would typically implement this instead of
//                  `onReceive()` for efficiency, i.e. when
//                  you only want to know which targets were
//                  called and how many times, but don't care
//                  about the order that the calls happened
//                  in.
//
onCallSummary: function (summary) {
},

//
// Advanced users: This is how you can plug in your own
//                  StalkerTransformer, where the provided
//                  function is called synchronously
//                  whenever Stalker wants to recompile
//                  a basic block of the code that's about
//                  to be executed by the stalked thread.
//
//transform: function (iterator) {
//  var instruction = iterator.next();
//
//  var startAddress = instruction.address;
//  var isAppCode = startAddress.compare(appStart) >= 0 &&
//    startAddress.compare(appEnd) === -1;
//
//  do {
//    if (isAppCode && instruction.mnemonic === 'ret') {
//      iterator.putCmpRegI32('eax', 60);
//      iterator.putJccShortLabel('jb', 'nope', 'no-hint');
//
//      iterator.putCmpRegI32('eax', 90);
//      iterator.putJccShortLabel('ja', 'nope', 'no-hint');
//
//      iterator.putCallout(onMatch);
//
//      iterator.putLabel('nope');
//    }
//
//    iterator.keep();
//  } while ((instruction = iterator.next()) !== null);
//},
//
// The default implementation is just:
//
//  while (iterator.next() !== null)
//    iterator.keep();
//
// The example above shows how you can insert your own code
// just before every `ret` instruction across any code
// executed by the stalked thread inside the app's own
// memory range. It inserts code that checks if the `eax`
// register contains a value between 60 and 90, and inserts
// a synchronous callout back into JavaScript whenever that
// is the case. The callback receives a single argument

```

```
// that gives it access to the CPU registers, and it is
// also able to modify them.
//
// function onMatch (context) {
//   console.log('Match! pc=' + context.pc +
//     ' rax=' + context.rax.toInt32());
// }
//
// Note that not calling keep() will result in the
// instruction getting dropped, which makes it possible
// for your transform to fully replace certain instructions
// when this is desirable.
//
});
```



Performance considerations

The callbacks provided have a significant impact on performance. If you only need periodic call summaries but do not care about the raw events, or the other way around, make sure you omit the callback that you don't need; i.e. avoid putting your logic in *onCallSummary* and leaving *onReceive* in there as an empty callback.

<http://blog.csdn.net/freakishfox>

- **Stalker.unfollow([threadId]):** 停止监控线程事件，如果是当前线程，则可以省略 **threadId** 参数
- **Stalker.parse(events[, options]):** 按照指定格式介些 **GumEvent** 二进制数据块，按照 **options** 的要求格式化输出，举个例子：

```
onReceive: function (events) {
  console.log(Stalker.parse(events, {
    annotate: true, // to display the type of event
    stringify: true
    // to format pointer values as strings instead of `NativePointer`
    // values, i.e. less overhead if you're just going to `send()` the
    // thing not actually parse the data agent-side
  }));
},
```

- **Stalker.garbageCollect():** 在调用 **Stalker.unfollow()** 之后，在一个合适的时候，释放对应的内存，可以避免多线程竞态条件下的内存释放问题。
- **Stalker.addCallProbe(address, callback):** 当 **address** 地址处的函数被调用的时候，调用 **callback** 对象（对象类型和 **Interceptor.attach.onEnter** 一致），返回一个 **Id**，可以给后面的 **Stalker.removeCallProbe** 使用
- **Stalker.removeCallProbe():** 移除前面的 **addCallProbe** 调用效果。
- **Stalker.trustThreshold:** 指定一个整型 **x**，表示可以确保一段代码在执行 **x** 次之后，代码才可以认为是可靠的稳定的，**-1** 表示不信任，**0** 表示持续信任，**N** 表示执行 **N** 次之后才是可靠的，稳定的，默认值是 **1**。
- **Stalker.queueCapacity:** 指定事件队列的长度，默认长度是 **16384**
- **Stalker.queueDrainInterval:** 事件队列查询派发时间间隔，默认是 **250ms**，也就是说 **1** 秒钟事件队列会轮询 **4** 次

ApiResolver

- **new ApiResolver(type):** 创建指定类型 **type** 的 API 查找器，可以根据函数名称快速定位到函数地址，根据当前进程环境不同，可用的 **ApiResolver** 类型也不同，到目前为止，可用的类型有：
 - **Module:** 枚举当前进程中已经加载的动态链接库的导入导出函数名称。
 - **objc:** 定位已经加载进来的 Object-C 类方法，在 macOS 和 iOS 进程中可用，可以使用 **Objc.available** 来进行运行时判断，或者在 **try-catch** 块中使用 **new ApiResolver('objc')** 来尝试创建。
 - 解析器在创建的时候，会加载最小的数据，后续使用懒加载的方式来持续加载剩余的数据，因此最好是一次相关的批量调用使用同一个 **resolver** 对象，然后下次的相关操作，重新创建一个 **resolver** 对象，避免使用上个 **resolver** 的老数据。
- **enumerateMatches(query, callbacks):** 执行函数查找过程，按照参数 **query** 来查找，查找结果调用 **callbacks** 来回调通知：
 - **onMatch: function(match):** 每次枚举到一个函数，调用一次，回调参数 **match** 包含 **name** 和 **address** 两个属性。
 - **onComplete: function():** 整个枚举过程完成之后调用。
 - 举个例子：

```
var resolver = new ApiResolver('module');
resolver.enumerateMatches('exports:*!open*', {
  onMatch: function (match) {
    /*
     * Where `match` contains an object like this one:
     *
     * {
     *   name: '/usr/lib/libSystem.B.dylib!opendir$INODE64',
     *   address: ptr('0x7fff870135c9')
     * }
     */
  },
  onComplete: function () {
  }
});
```

```
var resolver = new ApiResolver('objc');
resolver.enumerateMatches('-[NSURL* *HTTP*]', {
  onMatch: function (match) {
    /*
     * Where `match` contains an object like this one:
     *
     * {
     *   name: '-[NSURLRequest valueForKeyForHTTPHeaderField:]',
     *   address: ptr('0x7fff94183e22')
     * }
     */
  },
  onComplete: function () {
  }
});
```

- **enumerateMatchesSync(query): enumerateMatches()** 的同步版本，直接返回所有结果的数组形式

DebugSymbol

- **DebugSymbol.fromAddress(address), DebugSymbol.fromName(name):** 在指定地址或者指定名称查找符号信息，返回的符号信息对象包含下面的属性：
 - **address:** 当前符号的地址，NativePointer
 - **name:** 当前符号的名称，字符串形式
 - **moduleName:** 符号所在的模块名称
 - **fileName:** 符号所在的文件名
 - **lineNumber:** 符号所在的文件内的行号
 - 为了方便使用，也可以在这个对象上直接使用 **toString()**，输出信息的时候比较有用，比如和 **Thread.backtrace** 配合使用，举个例子来看：

```
var f = Module.findExportByName("libcommonCrypto.dylib",
  "CCCryptorCreate");
Interceptor.attach(f, {
  onEnter: function (args) {
    console.log("CCCryptorCreate called from:\n" +
      Thread.backtrace(this.context, Backtracer.ACCURATE)
        .map(DebugSymbol.fromAddress).join("\n") + "\n");
  }
});
```

- **DebugSymbol.getFunctionByName(name), DebugSymbol.findFunctionsNamed(name), DebugSymbol.findFunctionsMatching(glob):** 这三个函数，都是根据符号信息来查找函数，结果返回 **NativePointer** 对象。

Instruction

- **Instruction.parse(target):** 在 **target** 指定的地址处解析指令，其中target是一个NativePointer。注意，在32位ARM上，ARM函数地址需要是2字节对齐的，Thumb函数地址是1字节对齐的，如果你是使用Frida本身的函数来获取的target地址，Frida会自动处理掉这个细节，parse函数返回的对象包含如下属性：
 - **address:** 当前指令的EIP，NativePointer类型
 - **next:** 下条指令的地址，可以继续使用parse函数
 - **size:** 当前指令大小
 - **mnemonic:** 指令助记符

- **opStr**: 字符串格式显示操作数
- **operands**: 操作数数组，每个操作数对象包含**type**和**value**两个属性，根据平台不同，有可能还包含一些额外属性
- **regsRead**: 这条指令显式进行读取的寄存器数组
- **regsWritten**: 这条指令显式的写入的寄存器数组
- **groups**: 该条指令所属的指令分组
- **toString()**: 把指令格式化成一条人比较容易读懂的字符串形式
- 关于**operands**和**groups**的细节，请参考**CapStone**文档

ObjC

Java

- **Java.available**: 布尔型取值，表示当前进程中是否存在完整可用的**Java**虚拟机环境，**Dalvik**或者**Art**，建议在使用**Java**方法之前，使用这个变量来确保环境正常。
- **Java.enumerateLoadedClasses(callbacks)**: 枚举当前进程中已经加载的类，每次枚举到加载的类回调**callbacks**:
 - **onMatch: function(className)**: 枚举到一个类，以类名称进行回调，这个类名称后续可以作为 **Java.use()** 的参数来获取该类的一个引用对象。
 - **onComplete: function()**: 所有的类枚举完毕之后调用
- **Java.enumerateLoadedClassesSync()**: 同步枚举所有已经加载的类
- **Java.use(fn)**: 把当前线程附加到**Java VM**环境中去，并且执行**Java**函数**fn**（如果已经在**Java**函数的回调中，则不需要再附加到**VM**），举个例子：

```
Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
    Activity.onResume.implementation = function () {
        send("onResume() got called! Let's call the original implementation");
        this.onResume();
    };
});
```

- **Java.use(className)**: 对指定的类名动态的获取这个类的**JavaScript**引用，后续可以使用**\\$new()**来调用类的构造函数进行类对象的创建，后续可以主动调用 **\\$dispose()** 来调用类的析构函数来进行对象清理（或者等待**Java**的垃圾回收，再或者是**JavaScript**脚本卸载的时候），静态和非静态成员函数在**JavaScript**脚本里面也都是可见的， 你可以替换**Java**类中的方法，甚至可以在里面抛出异常，比如：

```
Java.perform(function () {
    var Activity = Java.use("android.app.Activity");
    var Exception = Java.use("java.lang.Exception");
    Activity.onResume.implementation = function () {
        throw Exception.$new("Oh noes!");
    };
});
```

- **Java.scheduleOnMainThread(fn)**: 在虚拟机主线程上执行函数**fn**
- **Java.choose(className, callbacks)**: 在**Java**的内存堆上扫描指定类名称的**Java**对象，每次扫描到一个对象，则回调**callbacks**:
 - **onMatch: function(instance)**: 每次扫描到一个实例对象，调用一次，函数返回**stop**结束扫描的过程
 - **onComplete: function()**: 当所有的对象都扫描完毕之后进行回调
- **Java.cast(handle, klass)**: 使用对象句柄**handle**按照**klass**（**Java.use**方法返回）的类型创建一个对象的**JavaScript**引用，这个对象引用包含一个**class**属性来获取当前对象的类，也包含一个**\\$className**属性来获取类名称字符串，比如：

```
var Activity = Java.use("android.app.Activity");
var activity = Java.cast(ptr("0x1234"), Activity);
```

单篇儿太长，下篇继续

JavaScript API

WeakRef

- **WeakRef.bind(value, fn):** 监控**value**对象，当被监控的对象即将被垃圾回收或者脚本即将被卸载的时候，调用回调函数**fn**，**bind**返回一个唯一ID，后续可以使用这个ID进行 **WeakRef.unbind()**调用来取消前面的监控。这个API还是很有用处的，比如你想要在JavaScript的某个对象销毁的时候跟着销毁一些本地资源，这种情况下，这个机制就比较有用了。
- **WeakRef.unbind(id):** 停止上述的对象监控，并且会立即调用一次**f n**

x86Writer

- **new X86Writer(codeAddress[, {pc: ptr('0x1234')}]):** 创建一个x86机器码生成器，并且在codeAddress指向的内存进行写入，codeAddress是NativePointer类型，第二个参数是可选参数，用来指定程序的初始EIP。在iOS系统上，使用**Memory.patchCode()**的时候，指定初始EIP是必须的，因为内存写入是先写入到一个临时的位置，然后再映射到指定位置的内存
- **reset(codeAddress[, { pc: ptr('0x1234')}]):** 取消codeAddress位置的上次的代码写入
- **dispose():** 立即进行X86相关的内存修改清理
- **flush():** 代码中标签引用的解析，操作缓存立即应用到内存中去。在实际的应用中，当生成一段代码片段的时候，就应该调用一次这个函数。多个相关联的函数片段在一起使用的时候，也应该调用一次，尤其是要在一起协同运行的几个函数片段。
- **base:** 输出结果的第一个字节码的内存位置，NativePointer类型
- **code:** 输出结果的下一个字节码的内存位置，NativePointer类型
- **pc:** 输出结果的指令指针的内存位置，NativePointer类型
- **offset:** 当前的偏移（JavaScript数值）
- **putLabel(id):** 在当前位置插入一个标签，标签用字符串**id**表示
- **putCallAddressWithArguments(fund, args):** 准备好一个调用C函数的上下文环境，其中args表示被调用函数的参数数组（JavaScript数组），数组里面可以是字符串形式指定的寄存器，可以是一个数值，也可以是一个指向立即数的NativePointer
- **putCallAddressWithAlignedArguments(func, args):** 跟上面一个函数差不多，但是参数数组是16字节对齐的
- **putCallRegWithArguments(reg, args):** 准备好一个调用C函数的上下文环境，其中args表示被调用函数的参数数组（JavaScript数组），数组里面可以是字符串形式指定的寄存器，可以是一个数值，也可以是一个指向立即数的NativePointer
- **putCallRegWithAlignedArguments(reg, args):** 参数数组16字节对齐
- **putCallRegOffsetPtrWithArguments(reg, offset, args):** 准备好一个调用C函数的上下文环境，其中args表示被调用函数的参数数组（JavaScript数组），数组里面可以是字符串形式指定的寄存器，可以是一个数值，也可以是一个指向立即数的NativePointer
- **putCallAddress(address):** 写入一个Call指令
- **putCallReg(reg):** 写入一个Call指令
- **putCallRegOffsetPtr(reg, offset):** 写入一个Call指令
- **putCallIndirect(addr):** 写入一个Call指令
- **putCallNearLabel(labelId):** 在前面定义的Label处创建一个Call 指令
- **putLeave():** 创建一个 LEAVE 指令
- **putRet():** 创建一个 RET 指令
- **putRetImm(immValue):** 创建一个RET指令
- **putJumpShortLabel(labelId):** 创建一个JMP指令，跳转到labelId标志的位置
- **putJumpNearLabel(labelId):** 创建一个JMP指令，跳转到labelId标志的位置
- **putJumpReg(reg):** 创建一个JMP指令
- **putJumpRegPtr(reg):** 创建一个JMP指令
- **putJumpRegOffsetPtr(reg, offset):** 创建一个JMP指令
- **putJumpNearPtr(address):** 创建一个JMP指令
- **putJccShort(labelId, target, hint):** 创建一个JCC指令
- **putJccNear(labelId, target, hint):** 在labelId处创建一个JCC指令
- **putAddRegImm(reg, immValue)**
- **putAddRegReg**
- **putAddRegNearPtr(dstReg, srcAddress)**
- **putSubRegImm(reg, immValue)**
- **putSubRegReg(dstReg, srcReg)**
- **putSubRegNearPtr(dstReg, srcAddress)**
- **putIncReg(reg)**
- **putDecReg(reg)**
- **putIncRegPtr(target, reg)**
- **putDecRegPtr(target, reg)**
- **putLockXaddRegPtrReg(dstReg, srcReg)**
- **putLockInclImm32Ptr(target)**
- **putLockDecImm32Ptr(target)**
- **putAddRegReg(dstReg, srcReg)**
- **putAddRegU32(reg, immValue)**

- putShlRegU8(reg, immValue)
- putShrRegU8(reg, immValue)
- putXorRegReg(dstReg, srcReg)
- putMovRegReg(dstReg, srcReg)
- putMovRegU32(dstReg, immValue)
- putMovRegU64(dstReg, immValue)
- putMovRegAddress(dstReg, immValue)
- putMovRegPtrU32(dstReg, immValue)
- putMovRegOffsetPtrU32(dstReg, dstOffset, immValue)
- putMovRegPtrReg(dstReg, srcReg)
- putMovRegOffsetPtrReg(dstReg, dstOffset, srcReg)
- putMovRegRegPtr(dstReg, srcReg)
- putMovRegRegOffsetPtr(dstReg, srcReg, srcOffset)
- putMovRegBaseIndexScaleOffsetPtr(dstReg, baseReg, indexReg, scale, offset)
- putMovRegNearPtr(dstReg, srcAddress)
- putMovNearPtrReg(dstAddress, srcReg)
- putMovFsU32PtrReg(fsOffset, srcReg)
- putMovRegFsU32Ptr(dstReg, fsOffset)
- putMovGsU32PtrReg(fsOffset, srcReg)
- putMovqXmm0EspOffsetPtr(offset)
- putMovqEaxOffsetPtrXmm0(offset)
- putMovdquXmm0EspOffsetPtr(offset)
- putMovdquEaxOffsetPtr(offset)
- putLeaRegRegOffset(dstReg, srcReg, srcOffset)
- putXchgRegRegPtr(leftReg, rightReg)
- putPushU32(immValue)
- putPushNearPtr(address)
- putPushReg(reg)
- putPopReg(reg)
- putPushImmPtr(immPtr)
- putPushax()
- putPopax()
- putPushfx()
- putPopfx()
- putTestRegReg(regA, regB)
- putTestRegU32(reg, immValue)
- putCmpRegI32(reg, immValue)
- putCmpRegOffsetPtrReg(regA, offset, regB)
- putCmplImmPtrImmU32(immPtr, immValue)
- putCmpRegReg(regA, regB)
- putClc()
- putStc()
- putCld()
- putStd()
- putCpuid()
- putLfence()
- putRdtsc()
- putPause()
- putNop()
- putBreakpoint()
- putPadding(n)
- putNopPadding(n)
- putU8(value)
- putS8(value)
- putBytes(data) 从ArrayBuffer中拷贝原始数据

X86Relocator

- **new X86Relocator(inputCode, output):** 创建一个代码重定位器，用以进行代码从一个位置拷贝到另一个位置的时候进行代码重定位处理，源地址是 **inputCode**的NativePointer，**output**表示结果地址，可以用X86Writer对象来指向目的内存地址
- **reset(inputCode, output):** 回收上述的X86Relocator对象
- **dispose():** 内存清理
- **input:** 最后一次读取的指令，一开始是null，每次调用readOne()会自动改变这个属性

- **eob**: 表示当前是否抵达了块结尾，比如是否遇到了下列任何一个指令：CALL, JMP, BL, RET
- **eoi**: 表示input代表的属性是否结束，比如可能当前遇到了下列的指令：JMP, B, RET，这些指令之后可能没有有效的指令了
- **readOne()**: 把一条指令读入relocator的内部缓存，返回目前已经读入缓存的总字节数，可以持续调用readOne函数来缓存指令，或者立即调用writeOne()或者skipOne()，也可以一直缓存到指定的点，然后一次性调用writeAll()。如果已经到了eoi，则函数返回0，此时**eoi**属性也是true
- **peekNextWriteInsn()**: peek一条指令出来，以备写入或者略过
- **peekNextWriteSource()**: 在指定地址peek一条指令出来，以备写入或者略过
- **skipOne()**: 忽略下一条即将写入的指令
- **skipOneNoLabel()**: 忽略下一条即将写入的指令，如果遇到内部使用的Label则不忽略，这个函数是对skipOne的优化，可以让重定位范围覆盖的更全面
- **writeOne()**: 写入下条缓存指令
- **writeOneNoLabel()**
- **writeAll()**: 写入所有缓存的指令

x86枚举类型

- 寄存器：xar, xcx, xdx, xbx, tsp, xbp, xsi, xdi, sax, ecx, edx, ebx, esp, ebx, esi, edi, rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15, r8d, r9d, r10d, r11d, r12d, r13d, r14d, r15d, xip, eip, rip
- 跳转指令：jo, jno, jb, jae, je, jne, jbe, ja, js, jns, jp, jnp, jl, jge, jle, jg, jcxz, jecxz, jrcxz
- 分支提示：no-hint, likely, unlikely
- 指针类型：byte, sword, qword

ArmWriter(参考X86Writer)

- new ArmWriter(codeAddress[, {pc: ptr('0x1234')}])
- reset(codeAddress[, {pc: ptr('0x1234')}])
- dispose()
- flush()
- base
- code
- pc
- offset
- skip(nBytes)
- putBlmm(target)
- putLdrRegAddress(reg, address)
- putLdrRegU32(reg, val)
- putAddRegRegImm(dstReg, srcReg, immVal)
- putLdrRegRegImm(dstReg, srcReg, immVal)
- putNop()
- putBreakpoint()
- putInstruction(insn)
- putBytes(data)

ArmRelocator(参考X86Relocator)

ThumbRelocator(参考X86Relocator)

Arm enum types

- 寄存器：r0~r15, sp, lr, sb, sl, fp, ip, pc
- 条件码：eq, ne, hs, lo, mi, pl, vs, vc, hi, ls, ge, lt, gt, le, al

Arm64Writer(参考X86Writer)

Arm64Relocator(参考X86Relocator)

AArch64 enum types

- 寄存器：x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 x20 x21 x22 x23 x24 x25 x26 x27 x28 x29 x30 w0 w1 w2 w3 w4 w5 w6 w7 w8 w9 w10 w11 w12 w13 w14 w15 w16 w17 w18 w19 w20 w21 w22 w23 w24 w25 w26 w27 w28 w29 w30 sp lr fp wsp wzr xzr nzcv ip0 ip1 s0 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s19 s20 s21 s22 s23 s24 s25 s26 s27 s28 s29 s30 s31 d0 d1 d2 d3 d4 d5 d6 d7 d8 d9 d10 d11 d12 d13 d14 d15 d16 d17 d18 d19 d20 d21 d22 d23 d24 d25 d26 d27 d28 d29 d30 d31 q0 q1 q2 q3 q4 q5 q6 q7 q8 q9 q10 q11 q12 q13 q14 q15 q16 q17 q18 q19 q20 q21 q22 q23 q24 q25 q26 q27 q28 q29 q30 q31
- 条件码：eq ne hs lo mi pl vs vc hi ls ge lt gt le al nv

- 索引模式: post-adjust signed-offset pre-adjust

MipsWriter(参考X86Writer)

MipsRelocator(参考X86Relocator)

MIPS enum types

- 寄存器: v0 v1 a0 a1 a2 a3 t0 t1 t2 t3 t4 t5 t6 t7 s0 s1 s2 s3 s4 s5 s6 s7 t8 t9 k0 k1 gp sp fp s8 ra hi lo zero at 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

全部完～～

目录

Frida官方手册中文版	2
目的	2
信息来源	2
翻译说明	2
译者	2
欢迎使用	3
Frida到底是什么？	3
Frida能干啥呢？	3
为什么Frida使用Python提供API, 又用JavaScript来调试程序逻辑呢？	3
建议、提醒	3
快速入门	4
安装Frida	6
安装环境要求	6
使用Pip进行安装	6
手动安装	6
测试一下效果	6
基础用法	7
模块枚举	7
枚举内存块	7
内存读写	7
使用姿势	8
本篇内容	8
注入模式	8
嵌入模式	8
预加载模式	8
消息发送	9
环境准备	9
从目标进程中发消息	10
处理JavaScript中的运行时错误	9
在目标进程中接收消息	9
在目标进程中以阻塞方式接收消息	9
在iOS上使用Frida	18
使用场景	18
已越狱机器	18
设置iOS设备	18
快速的冒烟测试	18
跟踪Twitter中的加密函数	18
没有越狱的iOS设备	19
定制你的xCode工程	19
快速的冒烟测试	19
跟踪libc函数	19
使用模拟器	20
打造自己的工具	20
在Android上使用Frida	21
设置你的Android设备	21
快速冒烟测试	21
跟踪Chrome里的Open()函数	21

构建自己的工具	21
Android上示例一则	22
Android CTF例子	22
JavaScript API	23
目录	23
Global	23
console	24
rpc	24
Frida	25
Process	25
Module	26
ModuleMap	27
Memory	27
MemoryAccessMonitor	28
Thread	28
下篇继续	28
JavaScript API	30
Int64	30
UInt64	30
NativePointer	30
NativeFunction	30
NativeCallback	31
SystemFunction	31
Socket	31
SocketListener	31
SocketConnection	31
IOStream	31
InputStream	31
OutputStream	31
UnixInputStream	31
UnixOutputStream	31
Win32InputStream	31
Win32OutputStream	31
File	31
SqliteDatabase	31
SqliteStatement	31
Interceptor	31
Stalker	33
ApiResolver	35
DebugSymbol	36
Instruction	36
ObjC	37
Java	37
单篇儿太长，下篇继续	37
JavaScript API	38
WeakRef	38
x86Writer	38
X86Relocator	39
x86枚举类型	40
ArmWriter(参考X86Writer)	40
ArmRelocator(参考X86Relocator)	40

ThumbRelocator(参考X86Relocator)	40
Arm enum types	40
Arm64Writer(参考X86Writer)	40
Arm64Relocator(参考X86Relocator)	40
AArch64 enum types	40
MipsWriter(参考X86Writer)	41
MipsRelocator(参考X86Relocator)	41
MIPS enum types	41
全部完～～	41