Frida

2017年7月20日 14:31

Frida 原理分析

主目录下面主要关注build，frida-core ，frida-gum 和 frida-python，v
这么几个， 当然frida-node也可以关注一下。
目录里有个frida-clr ，大概是windows上的一个界面程序，因为用了c#，另外一个src目录放了一些stub吧，实际应该是生成一个dll给gui程序用的
有一部分是用vala写的，这个可以转成c语言，然后编译成二进制

官网有一张架构图，还有一些博客可以研究一下。
Frida-core
Frida 核心模块 ，根据官网描述，这是最要的注入代码，从这个core，我们可以注入一个进程，创建v8线程，
打开目录主要关注inject， src， lib 等模块
Inject 目录 有一个inject-glue.c ，主要是负责状态初始化和结束，初始化这里，如果是android的话，还需要patch selinux 。如果把系统的selinux 关掉的话，这部分其实可以没有的
Selinux patch 实际是根据这个表更新的

```
static const FridaSELinuxRule frida_selinux_rules[] =
{
{ { "domain", NULL }, "domain", "process", { "execmem", NULL } },
{ { "domain", NULL }, "frida_file", "dir", { "search", NULL } },
{ { "domain", NULL }, "frida_file", "fifo_file", { "open", "write",
NULL } },
{ { "domain", NULL }, "frida_file", "file", { "open", "read",
"getattr", "execute", NULL } },
{ { "domain", NULL }, "frida_file", "sock_file", { "write",
NULL } },
{ { "domain", NULL }, "shell_data_file", "dir", { "search",
NULL } },
{ { "domain", NULL }, "zygote_exec", "file", { "execute", NULL } },
```

```
{ { "zygote", NULL }, "zygote", "capability", { "sys_ptrace",
NULL } },
{ { "zygote", NULL }, "shell", "process", { "sigchld", NULL } },
};
```

整个流程都比较容易理解

比较重要的是vala 这个文件，这个文件会被翻译成c，然后进行编译，

inject.vala 有个frida.inject 的namespace,仔细看inject-glue.c发现，这两个函数其实是Environment

namespace里的两个函数，相当于一个桥梁了。

Inject.vala的主要功能就是解析命令行，然后调用application.run

```
Application 类包含一下一些成员
        private DeviceManager device_manager;
        private int pid;
        private string script_file;
        private bool enable_jit;
        private bool enable_development;
        private ScriptRunner script_runner;
```

这个类有一个start和stop用于开始和结束hook

Start 函数 创建一个DeviceManager ， 并attach 上一个pid ，返回一个session，然后new一个ScriptRunner 开始跑脚本

```
        private async void start () throws Error {
            device_manager = new DeviceManager ();
            var device = yield device_manager.get_device_by_type
(DeviceType.LOCAL);
            var session = yield device.attach (pid);
            var r = new ScriptRunner (session, script_file,
enable_jit, enable_development);
            try {
                yield r.start ();
                script_runner = r;
            } catch (Error e) {
                printerr ("Failed to load script: " + e.message +
"\n");
            }
        }
```

Inject.vala 中最重要的就是这个类了。

ScriptRunner 主要是一个session 成员，和一个script——file ，session 成员控制是否开启jit

```
        public ScriptRunner (Session session, string script_file,
bool enable_jit, bool enable_development) {
```

```
boot enable_jit, boot enable_development) {
            this.session = session;
            this.script_file = script_file;
            this.enable_development = enable_development;
            if (enable_jit)
                session.enable_jit.begin ();
        }
```

application中的run主要调用scriptRunner的start，这个start内部又调用了load，后面是开发相关的代码，可以自己选择看不看

接下来看load函数

```
        private async void load () throws Error {
            load_in_progress = true;
            try {
                var name = Path.get_basename (script_file).split
(".", 2)[0];
                string source;
                try {
                    FileUtils.get_contents (script_file, out
source);
                } catch (FileError e) {
                    throw new Error.INVALID_ARGUMENT (e.message);
                }
                var s = yield session.create_script (name, source);
                if (script != null) {
                    try {
                        yield call ("dispose", new Json.Node[] {});
                    } catch (Error e) {
                    }
                    yield script.unload ();
                    script = null;
                }
                script = s;
                script.message.connect (on_message);
                yield script.load ();
                try {
                    yield call ("init", new Json.Node[] {});
                } catch (Error e) {
                }
            } finally {
                load_in_progress = false;
            }
        }
```

它主要做了一下几件事，把js文件读到内存，然后通过session对象创建一个script对象，接着这个script对象和一个 消息处理函数做一个连接，然后调用script.load()函数加载。中间有几次调用了这个call函数，看代码发现是一个向rpc发送请求的代码，这个服务端收到这个请求后会调用其中的函数，那么问题来了，这个rpc的服务端是谁，如果能搞清楚是谁，或许我们也可以假装

客户端向服务端通信了。

最后就是一个on_message函数，用于解析jsondata，并输出，没什么好说的。

这个文件留下的最大疑问就是session这个成员内部是咋样的？

Lib 目录

Lib 目录下文件也是众多。先看agent ，这个文件貌似是注入到目标文件中的。

Lib 目录中的agent 目录

这个目录包含了两个Agent ，

```
    public void main (string pipe_address, ref bool stay_resident,
Gum.MemoryRange? mapped_range) {
        Environment._init ();
        {
            var agent_range = memory_range (mapped_range);
            Gum.Cloak.add_range (agent_range);
            Gum.Cloak.add_thread (Gum.Process.get_current_thread_id
());
            Gum.MemoryRange stack;
            if (Gum.Thread.try_get_range (out stack))
                Gum.Cloak.add_range (stack);
            var interceptor = Gum.Interceptor.obtain ();
            interceptor.ignore_current_thread ();
            var exceptor = Gum.Exceptor.obtain ();
            var server = new AgentServer (pipe_address,
agent_range);
            try {
                server.run ();
            } catch (Error e) {
                printerr ("Unable to start agent server: %s\n",
e.message);
            }
            exceptor = null;
        }
        Environment._deinit ();
    }
```

Main 函数调用了gum相关的函数，这个后面再分析，后面就是开了一个AgentServer

AgentServer 比较重要的是注册连接，创建脚本引擎和打开jit

还有一个AgentClient，是在AgentServer open的时候创建的，这个Client

会从server获取js引擎，然后调用引擎相关的函数。

这个Agent获取了 脚本引擎，终于 看到了v8的影子

```
GumScriptBackend *
_frida_agent_environment_obtain_script_backend (gboolean
jit_enabled)
{
GumScriptBackend * backend = NULL;
#ifdef HAVE_DIET
backend = gum_script_backend_obtain_duk ();
#else
if (jit_enabled)
backend = gum_script_backend_obtain_v8 ();
if (backend == NULL)
backend = gum_script_backend_obtain_duk ();
#endif
return backend;
}
```

gadget目录主要是gadget.vala和script-engine.vala两个文件

Script-engine.vala 主要是绑定了AgentScriptId	和Gum.Script 最重要的是这个Gum.Script,用处挺大的，其他的一些函数，都是调用v8 进行操作。

Interfaces 目录定义了一些host 使用的函数 和agent使用的函数，可以看到agent几乎和脚本编译相关

```
        public abstract async AgentScriptId create_script (string
name, string source) throws GLib.Error;
        public abstract async AgentScriptId create_script_from_bytes
(uint8[] bytes) throws GLib.Error;
        public abstract async uint8[] compile_script (string name,
string source) throws GLib.Error;
        public abstract async void destroy_script (AgentScriptId
sid) throws GLib.Error;
        public abstract async void load_script (AgentScriptId sid)
throws GLib.Error;
        public abstract async void post_to_script (AgentScriptId
sid, string message, bool has_data, uint8[] data) throws GLib.Error;
        public signal void message_from_script (AgentScriptId sid,
string message, bool has_data, uint8[] data);
```

Host 主要的功能是注入

```
        public abstract async InjectorPayloadId inject_library_file
(uint pid, string path, string entrypoint, string data) throws
GLib.Error;
        public abstract async InjectorPayloadId inject_library_blob
(uint pid, uint8[] blob, string entrypoint, string data) throws
GLib.Error;
```

```
GLib.Error;
```

接着是loader目录，看到loader，想到cydia这个工具，可想到这个loader的主要作用就是把动态链接库注入到应用中

Loader主要代码如下

```
static void
frida_loader_init (void)
{
gpointer libc;
GumAttachReturn attach_ret;
frida_loader_prevent_unload ();
gum_init_embedded ();
libc = dlopen ("libc.so", RTLD_GLOBAL | RTLD_LAZY);
fork_impl = dlsym (libc, "fork");
dlclose (libc);
interceptor = gum_interceptor_obtain ();
monitor = g_object_new (FRIDA_TYPE_ZYGOTE_MONITOR, NULL);
attach_ret = gum_interceptor_attach_listener (interceptor,
fork_impl, GUM_INVOCATION_LISTENER (monitor), fork_impl);
g_assert_cmpint (attach_ret, ==, GUM_ATTACH_OK);
}
```

另外代码里有一些关于 zygote相关的代码，暂时不明确是干啥，

相关的函数 JNI_GetCreatedJavaVMs ，还有就是 android/os/Process ，setArgV0 ,这个以后可以分析一下

Channel-unix.c 是在用socket进行读写，那就有可能是frida-agent开了端口，等待server把编译好的代码传给agent，agent 再进行hook等操作，这个当然只是猜测

Server目录

直接看server.vala,涉及到一些mach_port相关的东西，这个当作android 系统服务里的bpbinder就好

```
    private const string DEFAULT_LISTEN_ADDRESS = "127.0.0.1";
    private const uint16 DEFAULT_LISTEN_PORT = 27042;
```

开头定义了ip，port，看代码貌似没有指定端口的地方，都是直接用了默认端口

main函数一直往下走，又调用了一个application.run() .这个application初始化的时候会创建一个re.frida.server 临时目录，用于存放文件，我看了一下发现只有一个frida-helper-32，64这两个文件，然后就是创建

各种Session 用于和pc通信

```
        construct {
            TemporaryDirectory.always_use ("re.frida.server");
#if WINDOWS
            host_session = new WindowsHostSession ();
#endif
#if DARWIN
            host_session = new DarwinHostSession (new
DarwinHelperBackend (), new TemporaryDirectory ());
#endif
#if LINUX
            host_session = new LinuxHostSession ();
#endif
#if QNX
            host_session = new QnxHostSession ();
#endif
            host_session.agent_session_opened.connect
(on_agent_session_opened);
            host_session.agent_session_closed.connect
(on_agent_session_closed);
        }

        public void run (string listen_uri) throws Error {
            try {
                server = new DBusServer.sync (listen_uri,
DBusServerFlags.AUTHENTICATION_ALLOW_ANONYMOUS, DBus.generate_guid
());
            } catch (GLib.Error listen_error) {
                throw new Error.ADDRESS_IN_USE
(listen_error.message);
            }
            server.new_connection.connect (on_connection_opened);
            server.start ();
            loop = new MainLoop ();
            loop.run ();
        }
```

Run 函数就是监听事件了，然后处理连接，可以看到代码里new 了一个Client对象，用来代表一个客户端，这里的agent_sessions 应该是用来存放session的

```
        private bool on_connection_opened (DBusConnection
    connection) {
            connection.closed.connect (on_connection_closed);
            var client = new Client (connection);
            client.register_host_session (host_session);
            foreach (var entry in agent_sessions.entries)
                client.register_agent_session (AgentSessionId
    (entry.key), entry.value);
            clients.set (connection, client);
```

```
            return true;
        }
```

这个Client 有两个函数，一个是register_host_session，是pc和手机之间的一个会话，register_agent_session暂时不明确
另外几个文件，和jit 有关，这个是和ios相关的，大概是搞了一个系统服务，用于提供jit服务，android 的没有看到，这个应该是编译进frida-server里的，这样编译完的代码就可以直接通过frida-agent 插到 应用里了

Src 目录

Agent-container.vala 主要去找frida_agent_main 这个接口，并且运行
System.vala 工具类
Frida.vala
Injector类 初始化不同系统的注入类 ，可以看到各个目录包含了对不同系统的操作
Linux 这个目录主要是用于注入用的，包含了Loader，Injector ，包含了32/64的处理主要逻辑在这里

Frida-gum
代码量比较大，看着比较蒙蔽

他是一个底层的注入框架，
目录下有个bindings ，和v8有一定的关系
bindings下面，一个是gumjs
js里有一个runtime，这里有很多js文件，是写js脚本时用到的一些导出函数
Core.js 包含了很多函数，另外还有一堆v8和duk的文件，内容比较重复
一个gumpp 不是很懂在干啥
看了一下vapi，看到了很多熟悉的namespace。
Gum 目录下包含了多个架构和一些基础文件，这里暂时只考虑arm和arm64

看一下arch-arm里的gumarmwirter ，里面包含了一些初始化的函数，和一些xxx_put_xxx函数

```
struct _GumArmWriter
{
volatile gint ref_count;
GumOS target_os;
guint32 * base;
guint32 * code;
GumAddress pc;
GumArmLiteralRef * literal_refs;
guint literal_refs_len;
};
```

自己带了一个引用计数， base是基址.code放的是代码， pc是pc。。。。

看了一下，代码，发现这些put 最终是会组合成最终的指令，比如arm就是组合成固定指令| 寄存器值或其他一些址，相当于是把指令组合好，然后调用 gum_arm_writer_put_instruction 把指令放到code里去，pc+对应的值。知道啦这个之后，其他几个架构的代码其实也差不多的。也就是现在解决了指令如何生成的问题。接下来的问题就是 如何找到我要修改的函数地址（ native层） 。 现在可以想象的就是，我们的代码是运行在了一块申请的内存中， 这块内存里包含了我们的js代码，这些代码由v8来执行。

还有一个是backend ，很大部分的内存操作都在这个函数里定义。

Frida 使用了 capstone 的c 版本

在backend-arm64 中， 可以看到，这里初始化了一些东西， 一个wirter，一个relocator，

```
GumInterceptorBackend *
_gum_interceptor_backend_create (GumCodeAllocator * allocator)
{
GumInterceptorBackend * backend;
backend = g_slice_new (GumInterceptorBackend);
backend->allocator = allocator;
gum_arm64_writer_init (&backend->writer, NULL);
gum_arm64_relocator_init (&backend->relocator, NULL, &backend->
writer);
gum_interceptor_backend_create_thunks (backend);
return backend;
}
```

Frida-server 启动的时候会创建一个re.frida.server的文件夹，里面有个so，一个helper，当注入的时候，会生产一个injecotr和pipe 文件

注入的时候直接往app里注入了一个frida-agent-xx.so

注入的时候直接往app里注入了一个frida-agent-xx.so

Frida-python

包含了一个src目录，这个目录采用了meson build ，然后代码是在frida这个目录里。

Application.py主要包含了一个ConsoleApplication,这个用语解析命令行，设置参数

__init__.py 包含了一些有用的函数，比如枚举设备，注入文件

其中有个device_manager,用处比较大，可以看到多个函数都会用到这个manager

```
global _device_manager
_device_manager = None
def get_device_manager():
global _device_manager
if _device_manager is None:
from . import core
_device_manager = core.DeviceManager(_frida.DeviceManager())
return _device_manager
```

Kill.py 继承了ConsoleApplication,调用了父类的run函数，最终会进入一个循环

Lsd.py 用于打印当前连接的设备

Ps.py 也比较简单，跟lsd.py有点像，主要是打印安装安装的应用

Discoverer.py

主线是通过Discoverer这个类以及一个js脚本完成的,这个功能主要是发现二进制文件中的函数，

Core.py

核心模块，放了一些重要的结构提供给其他几个py用，

Repl.py

主要是一个实时的控制台

Tracer.py

TracerProfileBuilder 用于构建参数

主体和其他几个py一样， py文件主要是架构，功能放在了js脚本中

包含了一个examples ，里面就是我们经常编写的fridapython脚本

包含了一个tests目录， 这个目录应该是用于开发测试的，从目录文件中可以看到，主要是测试了4个项目，test_core.py

看到，主要是测试了4个项目，test_core.py，
test_discoverer.py ,test_rpc.py test_tracer.py