安卓逆向之自动化JNI静态分析

原创

有价值炮灰 于 2020-10-07 15:04:13 发布 451 收藏 1

分类专栏： 信息安全 随笔 文章标签： android python

本文链接：https://blog.csdn.net/u010206565/article/details/108929149

国庆几天在家写了个用于辅助 JNI 接口逆向分析的工具，同时支持 Ghidra、IDA、Radare2。本文即是对这个工具的简单介绍，以及一些编写插件的体验记录。

前言

平时进行安卓逆向时，一个常见的场景就是目标把关键逻辑放到 Native 代码中，使用 JNI 接口进行实现。进行分析一般是把对应的动态库 `so` 拖进逆向工具中，然后定位具体的 Native 实现，再对参数类型、JNI 调用等逻辑进行一些优化方便对反汇编/反编译代码的理解。比如对于大家常用的 IDA-Pro 就是 `Parse C Header` 加载 `jni_all.h`，然后修改 JNI 接口的函数原型和变量结构。

这对于少量代码来说不是大问题，但显然是一种重复性的劳动，因此我们可以对这个过程进行一定的自动化。目前已经有了一些优秀的项目，比如 aryx 的 JNIAnalyzer，这是一个 Ghidra 插件，支持从 apk 中提取 JNI 接口并批量修改动态库的函数原型。但是这些项目都存在一些问题，而且缺乏拓展性，所以为了方便自己使用就构建了 JNI Helper 项目，支持各类日常使用的逆向工具。

JNI Helper

该项目的详细介绍可以参考 Github，其主要特性有下面这些:

- 基于 Jadx api 提供一个独立的 Java 可执行程序 `JadxFindJNI.jar`，用来分析 apk 并提取其中的 JNI 接口信息，并产出一个 JSON 文件；
- 同时支持 C 和 C++ 类型的 JNI 接口实现；
- 支持重载的 JNI 函数；
- 分别实现了 Ghidra、IDA 和 Radare2 的插件用来加载 `JadxFindJNI.jar` 生成的 JSON 文件；

JadxFindJNI.jar

得益于 Jadx 项目整洁的函数接口，我们可以很方便在其基础上实现自己的功能，如下所示:

```
JadxArgs jadxArgs = new JadxArgs();
jadxArgs.setDebugInfo(false);
jadxArgs.setSkipResources(true);
jadxArgs.getInputFiles().add(new File(args[0]));
JadxDecompiler jadx = new JadxDecompiler(jadxArgs);
jadx.load();
```

我们需要实现的主要功能就是调用 Jadx 分析目标 apk，然后迭代每个类和其中的 **native** 方法。只是有一些需要注意的点，比如对于重载的 JNI 函数。根据 Oracle 的文档，JNI native 函数的命名由以下方式组成:

- 以 `Java_` 为前缀；
- 格式化的类名完整路径；
- 下划线分隔符 `_`；
- 格式化的函数名称；
- 对于重载函数，需要加双下划线 `__` 并跟着参数签名；
- 对于特殊的字符需要进行转义，比如 `_`、`;` 和 `]` 需要分别转义成 `_1`、`_2` 和 `_3` 等；

对于使用者而言，无需关心内部细节:

```
$ java -jar JadxFindJNI/JadxFindJNI.jar
Usage: JadxFindJNI.jar <file.apk> <output.json>
```

我在仓库中上传了一个编译好的 demo/app-debug.apk，所生成的 JNI 签名信息如下:

```
{
  "Java_com_evilpan_demojni_MainActivity_testOverload__I": {
    "argumentSignature": "I",
    "argumentTypes": [
      "jint"
    ],
    "returnType": "jint",
    "isStatic": false
  },
  "Java_com_evilpan_demojni_MainActivity_testStatic": {
    "argumentSignature": "I",
    "argumentTypes": [
      "jint"
    ],
    "returnType": "jint",
    "isStatic": true
  },
  "Java_com_evilpan_demojni_MainActivity_stringFromJNI": {
    "argumentSignature": "",
    "argumentTypes": [],
    "returnType": "jstring",
    "isStatic": false
  },
  "Java_com_evilpan_demojni_MainActivity_c_1testArray": {
    "argumentSignature": "[I",
    "argumentTypes": [
      "jintArray"
    ],
    "returnType": "void",
    "isStatic": false
  },
  ...
}
```


该 JSON 函数签名文件以 JNI 函数的 native 名称为键，可以方便地在各种语言中反序列化为哈希表，从而方便函数的查找。

`JadxFindJNI.jar` 可以自己编译，也可以使用打包好的版本。

实现效果

只要有了函数签名信息，就很方便在各种逆向工具中进行自动化处理了，这里选取的是我比较常用的几个逆向工具，Ghidra、IDA 和 Radare2。这几个工具的插件都是使用 Python 编写的，感兴趣可以直接查看源码，优化前后的反编译代码如下图所示。

Ghidra 优化前:

Decompile: Java_com_evilpan_demojni_MainActivity_c_1testClass - (libnative-c.so)

```
undefined4
Java_com_evilpan_demojni_MainActivity_c_1testClass
          (int *param_1,undefined4 param_2,undefined4 param_3)

{
  undefined4 uVar1;
  undefined4 uVar2;
  undefined4 uVar3;
  undefined4 uVar4;
  code *pcVar5;
  undefined4 local_14;

  uVar1 = (**(code **)(*param_1 + 0x7c))(param_1,param_2);
  uVar2 = (**(code **)(*param_1 + 0x7c))(param_1,param_3);
  uVar3 = (**(code **)(*param_1 + 0x1c4))(param_1,uVar1,"static_callback","(Ljava/lang/String;)V");
  uVar2 = (**(code **)(*param_1 + 0x84))(param_1,uVar2,"getDataDir","()Ljava/io/File;");
  uVar2 = (**(code **)(*param_1 + 0x88))(param_1,param_3,uVar2);
  uVar4 = (**(code **)(*param_1 + 0x18))(param_1,"java/io/File");
  pcVar5 = *(code **)(*param_1 + 0x88);
  uVar4 = (**(code **)(*param_1 + 0x84))(param_1,uVar4,"getAbsolutePath","()Ljava/lang/String;");
  uVar2 = (*pcVar5)(param_1,uVar2,uVar4);
  (**(code **)(*param_1 + 0x234))(param_1,uVar1,uVar3,uVar2);
  return local_14;
}
```

Ghidra 优化后:

Decompile: Java_com_evilpan_demojni_MainActivity_c_1testClass - (libnative-c.so)

```
jint Java_com_evilpan_demojni_MainActivity_c_1testClass(JNIEnv *env,jobject thiz,jobject a1)

{
  jclass clazz;
  jclass clazz_00;
  jmethodID methodID;
  jmethodID methodID_00;
  jobject obj;
  _func_270 *p_Var1;
  jint local_14;

  clazz = (*(*env)->GetObjectClass)((JNIEnv *)env,thiz);
  clazz_00 = (*(*env)->GetObjectClass)((JNIEnv *)env,a1);
  methodID = (*(*env)->GetStaticMethodID)
                       ((JNIEnv *)env,clazz,"static_callback","(Ljava/lang/String;)V");
  methodID_00 = (*(*env)->GetMethodID)((JNIEnv *)env,clazz_00,"getDataDir","()Ljava/io/File;");
  obj = (*(*env)->CallObjectMethod)((JNIEnv *)env,a1,methodID_00);
  clazz_00 = (*(*env)->FindClass)((JNIEnv *)env,"java/io/File");
  p_Var1 = (*env)->CallObjectMethod;
  methodID_00 = (*(*env)->GetMethodID)
                          ((JNIEnv *)env,clazz_00,"getAbsolutePath","()Ljava/lang/String;");
  obj = (*p_Var1)((JNIEnv *)env,obj,methodID_00);
  (*(*env)->CallStaticVoidMethod)((JNIEnv *)env,clazz,methodID,obj);
  return local_14;
}
```

IDA-Pro 优化前:

IDA View-A | Pseudocode-A

```
1 int __fastcall Java_com_evilpan_demojni_MainActivity_c_1testClass(int a1)
2 {
3   int v1; // ST58_4
4   int v2; // ST3C_4
5   void (__fastcall *v3)(int, int, int); // ST14_4
6   int v4; // ST08_4
7   int v6; // [sp+5Ch] [bp-Ch]
8
9   v1 = a1;
10  (*(void (**)(void))(*(_DWORD *)a1 + 124))();
11  (*(void (**)(void))(*(_DWORD *)v1 + 124))();
12  (*(void (**)(void))(*(_DWORD *)v1 + 452))();
13  (*(void (**)(void))(*(_DWORD *)v1 + 132))();
14  v2 = (*(int (**)(void))(*(_DWORD *)v1 + 136))();
15  (*(void (**)(void))(*(_DWORD *)v1 + 24))();
16  v3 = *(void (__fastcall **)(int, int, int))(*(_DWORD *)v1 + 136);
17  v4 = (*(int (**)(void))(*(_DWORD *)v1 + 132))();
18  v3(v1, v2, v4);
19  (*(void (**)(void))(*(_DWORD *)v1 + 564))();
20  return v6;
21 }
```

IDA-Pro 优化后:

IDA View-A | Pseudocode-A

```
1 jint __cdecl Java_com_evilpan_demojni_MainActivity_c_1testClass(JNIEnv *env, jobject thiz, jobject arg1)
2 {
3   JNIEnv *v3; // ST58_4
4   jobject v4; // ST50_4
5   jclass v5; // ST4C_4
6   jclass v6; // ST48_4
7   _jmethodID *v7; // ST44_4
8   _jmethodID *v8; // r0
9   jobject v9; // ST3C_4
10  jclass v10; // r0
11  jobject (*v11)(JNIEnv *, jobject, jmethodID, ...); // ST14_4
12  _jmethodID *v12; // r0
13  jobject v13; // r0
14  jint v15; // [sp+5Ch] [bp-Ch]
15
16  v3 = env;
17  v4 = arg1;
18  v5 = (*env)->GetObjectClass(env, thiz);
19  v6 = (*v3)->GetObjectClass(v3, v4);
20  v7 = (*v3)->GetStaticMethodID(v3, v5, "static_callback", "(Ljava/lang/String;)V");
21  v8 = (*v3)->GetMethodID(v3, v6, "getDataDir", "()Ljava/io/File;");
22  v9 = (*v3)->CallObjectMethod(v3, v4, v8);
23  v10 = (*v3)->FindClass(v3, "java/io/File");
24  v11 = (*v3)->CallObjectMethod;
25  v12 = (*v3)->GetMethodID(v3, v10, "getAbsolutePath", "()Ljava/lang/String;");
26  v13 = v11(v3, v9, v12);
27  (*v3)->CallStaticVoidMethod(v3, v5, v7, v13);
28  return v15;
29 }
```

详见 https://github.com/evilpan/jni_helper 。

插件编写体验

在实现 JNI Helper 的过程中，摸索了一遍不同逆向工具的拓展功能，所以这里谈谈编写过程中的一些感受，正好也可以作为一次横向对比。

Ghidra

Ghidra 作为 NSA 出品的工具，拥有丰富的内部资料，开发文档非常整洁规范。虽然说是机缘巧合之下泄露出来才开源的版本，但其质量可以和许多商业工具相媲美，甚至在很多方面还稍胜一筹。使用下来发现 Ghidra 有很多在线文档和资料，可以很方便地实现指定功能，这个跟后面的两个工具可以说形成鲜明对比。

由于 Ghidra 是使用 Java 进行开发的，因此一个明显的问题是运行分析速度相对较慢。这个问题我的一个解决办法是通过 `headlessAnalyzer` 在性能较好的云服务器后台对目标先进行分析，导出 `.gzf` 项目文件再在本地导入。本地的 Ghidra 插件通常使用 Java 进行开发，但可以通过 JPython 使用 Python 脚本编写。

得益于开源软件的特性，开源社区中对于 Ghidra 插件和资料的贡献一直呈爆发式增长。当然这也是因为软件本身投入了很多国家经费被打磨过很长时间，另外一个开源逆向工具 Radare2 就相形见绌了，后面会说到。

一些比较有用的 Ghidra 相关参考资料如下:

- https://ghidra.re/ghidra_docs/api/
- https://ghidra.re/online-courses/
- https://github.com/cetfor/GhidraSnippets
- https://github.com/AllsafeCyberSecurity/awesome-ghidra

IDA Pro

老牌逆向工具，在 Ghidra 出来之前已经制霸了十几年，因此也拥有丰富的插件生态。从使用者的角度来说，IDA 无疑是一个优秀的选择，其反编译器可以给出非常接近源码的高层伪代码，极大减少了逆向工程师的工作量。缺点也就一个: 太贵。每年几千美刀的授权费用对于个人逆向爱好者而言并不是一个小数目，而且还分别针对不同的指令集架构进行收费。之前还满心期待家庭版的 IDA Home，365美元一年，但是没有 decompiler、没有 SDK、还不能商业使用，于是我又默默关闭页面并掏出了 Binary Ninja。

因此要有一个舒适的逆向体验，还是需要一个 `IDA Pro`。从插件开发者的角度来说，IDA Pro 本身确实提供了接口文档，而且由于年代久远也积累了许多插件生态，但是实际使用下来还是有些卡壳。举例来说，在写插件的过程中遇到一个需求，比如判断 `jni_all.h` 这个头文件是否已经加载过，即某个结构体是否已经定义，在文档中说是使用下面的 idapython 接口:

```
idaapi.get_struc_id('JNIInvokeInterface_') != idaapi.BADADDR
```

可是实际上这样即便 JNI 接口已经定义，返回也是 BADADDR 。最终实现还是通过一种比较取巧的方法，即根据下面的语句是否抛出异常来判断:

```
idc.parse_decl('JNIInvokeInterface_ *if', idc.PT_SILENT)
```

类似的问题还有不少，而且很多技巧需要通过阅读其他人的插件代码去了解，对于不熟悉的开发者来说体验并不是太好。下面是一些 IDA 开发相关的资料:

- https://www.hex-rays.com/products/ida/support/
- https://www.hex-rays.com/products/ida/support/tutorials/
- https://www.hex-rays.com/products/ida/support/sdkdoc/index.html
- https://gist.github.com/icecr4ck/7a7af3277787c794c66965517199fc9c

Radare2

作为一个骨灰级命令行爱好者，接触 Radare2(r2) 也是顺应自然的选择。最初使用 Radare2 的一个原因是因为其开源，可以在各个平台中使用；另外一个原因是支持 VI 快捷键，省去了我很大的学习成本。虽然一开始以命令行逆向工具为卖点，但也可以配合 Cutter 使用。

虽然每次我都狂热推荐使用 r2，但实际上他也有明显的局限性，其中一个是 decompiler 的缺乏。这可以使用一些三方的 decompiler 实现，比如:

- Ghidra Decompiler
- RetDec Decompiler
- r2dec

作为插件开发者，使用的主要是 r2pipe 接口，这个接口实际上是 r2 的命令行参数的管道，所以写插件本质上还是需要通过 r2 的命令实现。说到 r2 的命令，内置的帮助信息可以方便日常使用的查询，用户只需要记得大致的命令类目，比如分析类命令是`a`开头，可以使用`a?`查看所有分析相关的命令，进而可以使用`af?`查看所有和函数分析相关的命令。

虽然和 Ghidra 一样是开源工具，但由于 r2 主要是爱好者去维护，因此投入上的差距也导致了开发进度的差距。在 Github 中可以看到开发者们经常在努力修 bug 以及更新优化代码，但关闭 issue 的速度远跟不上新开 issue 的速度。举一个遇到的例子，我想要修改某个地址对应函数的签名为对应 JNI 函数签名，搜索后发现有几种方法: 一是使用 `afvr/avfn/afvt` 修改变量；二是使用 `tl` (type link)去连接类型；三是使用 `afs` 去直接修改签名信息。可测试下来这几种方式都有问题。其中 `afs` 是最接近我需求的命令了，但是却不支持自定义的类型，而解决这个问题还需要等 r2 项目组将内置 parser 从 `Tcc` 迁移到 `tree-sitter` (见 issue#17432)。诸如此类的问题也是比较打击插件开发者积极性的，毕竟谁也不想为了一个简单的功能找半天文档发现无法实现。

尽管如此，我还是非常看好 Radare2 的未来，他年岁尚浅，但有很大的拓展性，值得持续保持关注。同时也应该给开发者更多的耐心和支持，如果可以的话，为其添砖加瓦。下面是一些 r2 相关的资料:

- https://github.com/radareorg/radare2/blob/master/doc/intro.md
- https://radare.gitbooks.io/radare2book/content/
- https://book.rada.re/

后记

本文主要是分享 JNI Helper 这个辅助自动化静态逆向分析 JNI 接口的工具，可以在日常安卓逆向时候减少一些重复的劳动。俗话说工欲善其事，必先利其器，在日常中打磨自己的武器库值得投入必要时间，但也要避免陷入盲目造轮子的冲动。另一方面，也记录一下在编写各个逆向工具插件过程中的一些感受。总的来说 Ghidra 的插件编写体验最好，文档丰富且完善，不愧为 NSA 的产品；其次是 IDAPython，虽然有部分文档，但很多都只是一句话带过，想查找某些特定功能需要花费额外时间；而对于 Radare2，虽然是使用管道来进行批处理，但基本的交互都可以支持，只是部分功能实现尚不完善，需要给开发者更多的支持和耐心，毕竟大家都是花费着自己的业余时间和精力去维护和贡献。希望能有更多人投入到 Radare2 的社区中，哪怕只是贡献一点文档，写一写 writeup，对开源社区的帮助也是很大的。

参考链接

- JNI 文档
- skylot/jadx
- Ayrx/JNIAnalyzer
- https://github.com/evilpan/jni_helper
- https://evilpan.com/2020/10/07/jni-helper/