

基本开发环境配置



工欲善其事，必先利其器。本章将会介绍 Android 应用安全技术所需的环境配置，包括主机和测试机的基础环境。

一个良好的系统能给工作人员带来很多便利，不必因为环境问题焦头烂额。建议 Android 应用安全学习的新手，准备好与本书相同的环境，这样更有利于之后的学习，以免因环境配置问题而导致实验无法复现的可能性。

1.1 虚拟机环境搭建

推荐使用虚拟机而不是真机。

首先，虚拟机自带“系统时光机”功能——“快照”，虚拟机的这个特性让用户能够随时得到一个全新的真机，避免由于一个配置失误导致系统崩溃，最终只能选择重装系统而懊恼。图 1-1 所示为笔者在日常工作过程中开发 FART 脱壳机时创建的诸多虚拟机快照。

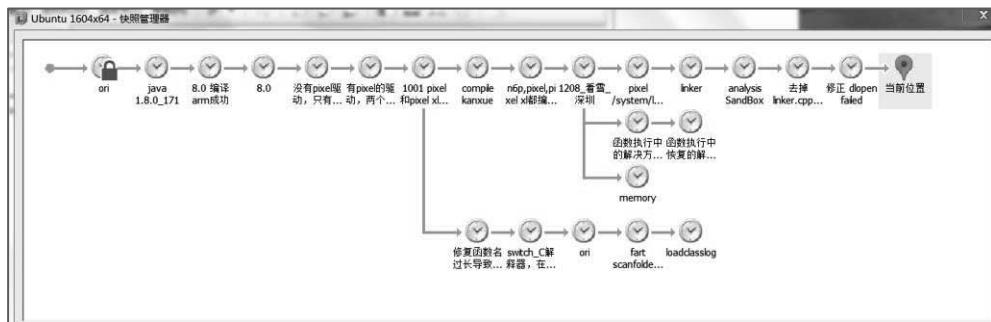


图 1-1 带快照功能的虚拟机

其次，虚拟机在工作环境中具有良好的隔离特性，可以确保在实验过程中不会“污染”真机，

是测试全新功能的天然“沙盘”环境。推荐使用 VMware 公司出品的虚拟机软件系列。VMware 具有良好的跨平台特性，不同系统的虚拟机文件都能做到复制即用。

对于虚拟机环境的选择，笔者推荐使用 Ubuntu 系列的 Linux 操作系统。经过笔者测试，不论是编译 Android 源码，还是使用 Frida、GDB、Ollvm 等重要工具，这个操作系统总是表现出更少受系统环境影响的特性。

在笔者的工作中，主要使用的是 Kali Linux 操作系统，Kali Linux 是基于 Debian 的 Linux 发行版，与 Ubuntu 师出同门，专门设计用于数字取证。Kali Linux 预装了许多渗透测试软件，包括 Metasploit、Burp Suite、SQLmap、Nmap 等，提供了一整套开箱即用的专业渗透测试工具。

Kali Linux 自带 VMware 镜像版本，下载并解压该镜像文件，然后双击打开.vmx 文件即可启动虚拟机。

图 1-2 为 Kali Linux 的界面。

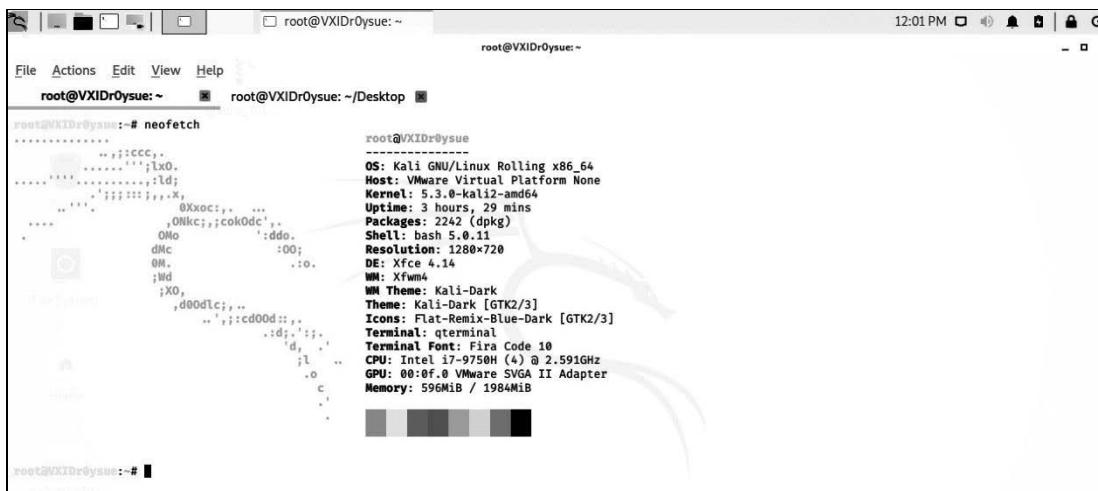


图 1-2 Kali Linux 的界面

因为虚拟机本身的时间可能与东八区时间不一致，所以需要进行时区设置。在启动虚拟机后，首先打开终端 Terminal，然后使用如下命令设置时区：

```
root@VXIDr0ysue:~# dpkg-reconfigure tzdata

Current default time zone: 'Asia/Shanghai'
Local time is now:      Sat Nov 14 12:11:05 CST 2020.
Universal Time is now:  Sat Nov 14 04:11:05 UTC 2020.
```

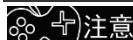
在弹出的窗口中选择 Asia→Shanghai 即可，读者可以根据自己所在的国家或地区进行选择。

另外，Kali Linux 默认不带中文支持，如果需要在中文网页上浏览或者进行抓包时解析包含中文的数据包，则还需要对 Kali 系统进行中文支持的配置，具体可执行如下命令：

```
root@VXIDr0ysue:~# apt update
```

```
root@VXIDr0ysue:~# apt install xfonts-intl-chinese
```

```
root@VXIDr0ysue:~# apt install ttf-wqy-microhei
```

 注意 一定不要把系统切换为纯中文环境，切换为纯中文环境会出现意想不到的问题。

1.2 逆向环境搭建

在准备好虚拟机环境之后，为了进行后面的逆向开发工作，还需要安装一些基础的开发工具。本节将介绍 Android Studio、ADB 工具、Python/Frida 环境以及 Objection 的安装配置和使用方法。

1.2.1 Android Studio 安装 NDK 编译套件

作为 Android 逆向开发人员，Android Studio 是一款必不可少的开发工具。在 Eclipse 退出 Android 开发历史舞台后，作为 Google 官方的 Android 应用开发 IDE，笔者首先推荐的就是这款软件。在虚拟机中，从官网下载和解压 Android Studio 之后，切换到 android-studio/bin 目录下，运行当前目录下的 studio.sh 即可启动 Android Studio。

首次启动 Android Studio 时，该软件会自动下载一些插件，比如 Android SDK 等工具，这些配置是后续开发所必需的，因此保持默认设置一直单击 Next 按钮即可。在插件下载完毕后，Android Studio 的界面将如图 1-3 所示。



图 1-3 Android Studio 主界面

单击 Create New Project 选项，创建新工程并选择模板 Native C++，如图 1-4 所示。

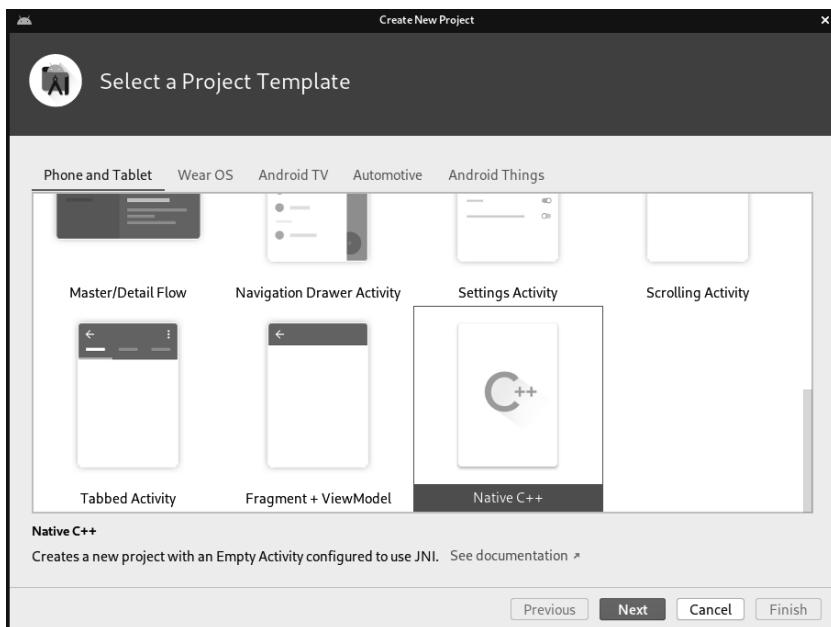


图 1-4 Native C++模板

下一步配置项目。项目的名称、包名以及保存位置可根据个人喜好自定义，语言选择 Java，最小 SDK 版本保持默认即可。单击 Next 按钮进入下一步，如图 1-5 所示。

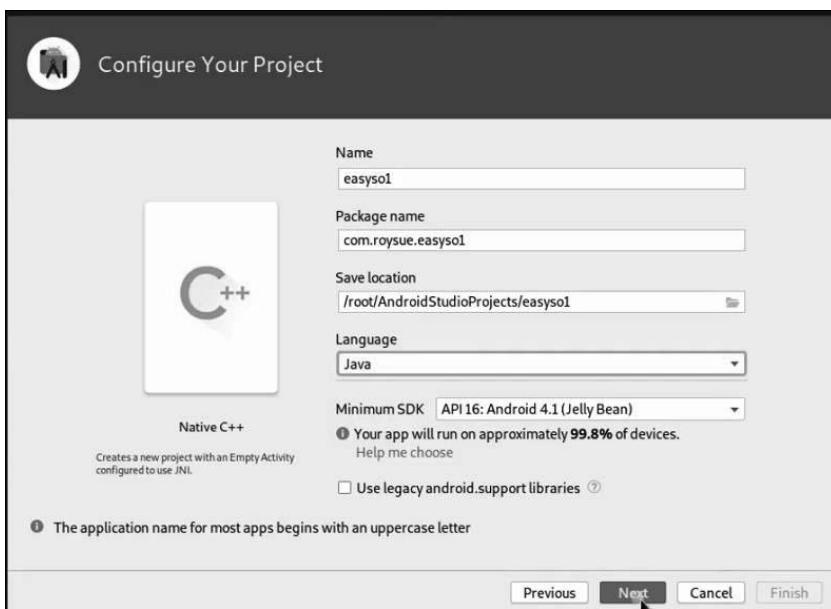


图 1-5 项目配置

在配置好这些选项后，单击 Finish 按钮，等待 Android Studio 完成配置和依赖的同步。在第一次创建 Project 时，Android Studio 会下载编译系统，下载完成之后进行编译时会发现没有 NDK（Native Development Kit），如图 1-6 所示。单击 Install latest NDK and sync project 安装最新版 NDK 即可。同步环节通常耗时比较久，这个时候只需要去喝杯茶静静等待即可。

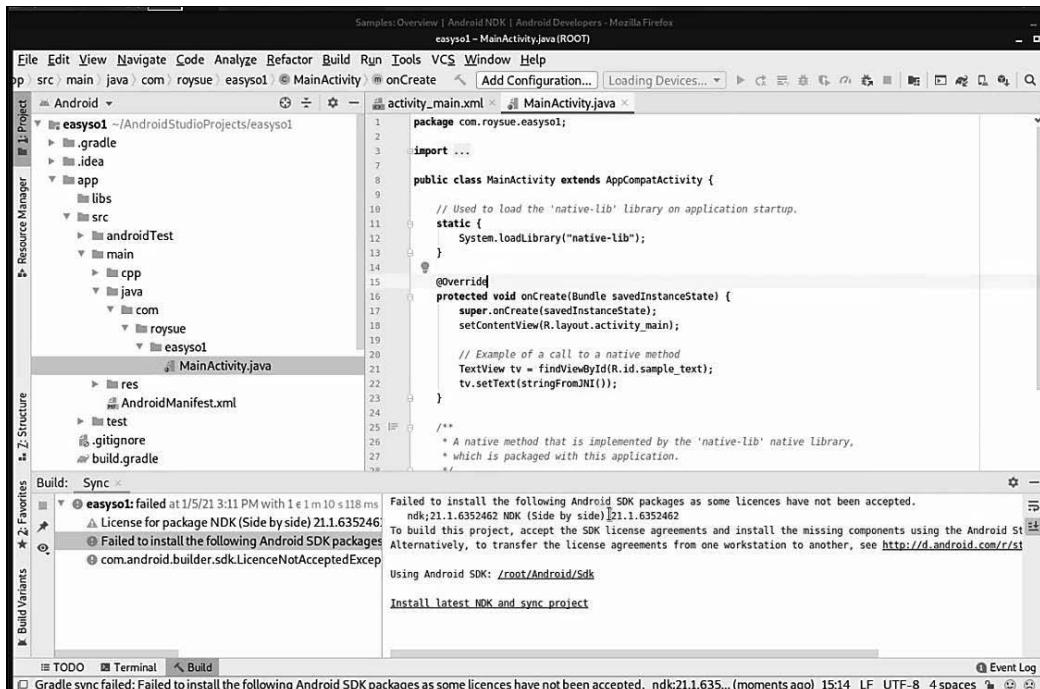


图 1-6 单击 Install latest NDK and sync project 以安装最新版 NDK

1.2.2 ADB 的配置和使用

ADB（Android Debug Bridge，Android 调试桥）是一种功能多样的命令行工具，用于与移动设备进行通信。由于 Android Studio 预先安装了 ADB，因此我们可以通过以下命令进入 ADB 的安装目录并查看其路径：

```

root@VXIDr0ysue:~# cd Android/Sdk/platform-tools
root@VXIDr0ysue:~/Android/Sdk/platform-tools# pwd
/root/Android/Sdk/platform-tools
  
```

这里使用 nano 命令将 ADB 工具所在目录加入环境变量，以便在任意目录下都能执行 adb 命令。

```

root@VXIDr0ysue:~/Android/Sdk/platform-tools# nano ~/.zshrc
  
```

文件打开后，将光标移动到文件末尾，添加一行代码：

```
export PATH="/root/Android/Sdk/platform-tools:$PATH"
```

读者可参考终端窗口最下方的提示，保存修改并退出文件。在将 adb 命令加入环境变量后，为了使得设置生效，需要重新启动终端 Terminal，然后再次执行 adb 命令，结果如下：

```
root@VXIDr0ysue:~# adb shell
* daemon not running; starting now at tcp:5037
* daemon started successfully
adb: no devices/emulators found
```

使用 ADB 连接设备有两种方法：USB 连接和网络连接。前者较为简单，用手机数据线接上主机 USB 接口即可。后者需要先使用 USB 数据线连接设备，再在命令行执行以下操作：

```
root@VXIDr0ysue:~# adb tcpip 5555

root@VXIDr0ysue:~# adb connect 10.203.171.25:5555
failed to authenticate to 10.203.171.25:5555

root@VXIDr0ysue:~# adb connect 10.203.171.25:5555
already connected to 10.203.171.25:5555
```

第一次尝试连接的时候，在手机上单击“允许调试”，再次尝试即可连接成功。需要注意的是，使用网络连接应保证主机与测试机在同一局域网。

1.2.3 Python 版本管理

笔者推荐使用 Python 版本的包管理软件 pyenv。通过 pyenv，我们可以安装和管理不同的 Python 版本，每个由 pyenv 包管理软件安装的 Python 版本都是相互隔离的。换句话说，无论在这个 Python 中安装了多少依赖包，对于另一个 Python 版本都是不可见的。如果你觉得自己的 Python 环境不纯或者需要一个新的环境，随时可以安装一个新的纯净的 Python，这也可以说是一种特殊的虚拟机环境。

需要注意的是，在安装 pyenv 之前，建议读者先对虚拟机进行一次快照。这样做是为了防止在安装 pyenv 的最后一步依赖时，可能导致整个系统无法进入桌面环境。下面是安装 pyenv 的具体过程：

```
root@VXIDr0ysue:~# git clone https://github.com/pyenv/pyenv.git ~/.pyenv
Cloning into '/root/.pyenv'

done.
Resolving deltas: 100% (12507/12507), done.

root@VXIDr0ysue:~# echo 'export PYENV_ROOT="$HOME/.pyenv"' >> ~/.zshrc
```

```

root@VXIDr0ysue:~# echo 'export PATH="$PYENV_ROOT/bin:$PATH"' >> ~/.zshrc

root@VXIDr0ysue:~# echo -e 'if command -v pyenv 1>/dev/null 2>&1; then\n  eval "$(pyenv init -)"\nfi' >> ~/.zshrc

root@VXIDr0ysue:~# exec "$SHELL"

root@VXIDr0ysue:~# sudo apt-get update; sudo apt-get install --no-install-recommends
make build-essential libssl-dev zlib1g-dev libbz2-dev libreadline-dev libsqlite3-dev wget
curl llvm libncurses5-dev xz-utils tk-dev libxml2-dev libxmlsec1-dev libffi-dev liblzma-dev

```

如果安装后重启能够正常进入桌面环境，接下来就可以方便地使用 `pyenv install` 命令安装不同版本的 Python 了，在安装完毕后还需要运行 `pyenv local` 命令切换到对应的版本。例如安装 Python 3.8.5 的过程如下：

```

root@VXIDr0ysue:~# pyenv install 3.8.5

root@VXIDr0ysue:~# pyenv local 3.8.5

root@VXIDr0ysue:~# python -V
Python 3.8.5

```

使用以上方法，可以再安装一个 3.8.0 版本的 Python，在之后的实验过程中，我们将会用到这两个版本的 Python 环境。

1.2.4 移动设备环境准备

在讲解后续虚拟机上的逆向环境配置之前，我们先准备移动设备。对于手机环境的搭建，建议参考以下文章，文章内容从硬件准备到刷机流程都讲解得非常详细，本书的实验环境也是按照下述文章进行配置的，参考网址 1（参见配书资源文件）。

另外，笔者推荐安装投屏软件 QtScrcpy，参考网址 2（参见配书资源文件）安装使用。

1.2.5 Frida 版本管理

移动设备环境准备好以后，就可以安装我们的主角——重要的逆向工具 Frida。在本书的实验中，Frida 安装了两个版本：一个是最新版，另一个是 12.8.0 版，后者运行一些相对较旧的代码比较管用。官网安装 Frida 的命令如下：

```
root@VXIDr0ysue:~# pip install frida-tools
```

安装成功后，可以在终端 Terminal 看到 Frida 安装的版本号（笔者在进行实验时，Frida 最新版本为 14.2.13）。在网址 3（参见配书资源文件）的 Frida 网站找到与之对应的版本号的 `frida-server` 进行下载。例如，安装版本号为 14.2.13 的 Frida，其对应的 `frida-server` 是 `frida-server-14.2.13-android-arm64.xz`。

下载 frida-server 后，使用 7z 解压并通过 adb push 命令将解压后的 frida-server 文件上传到 /data/local/tmp 目录下。接着，进入手机 Shell 并切换到 root 用户下的 /data/local/tmp 目录，也就是 frida-server 文件所在的位置，在该目录下，为 frida-server 文件赋予执行权限并运行它。

注意，在运行 14.2.13 版本的 frida-server 之前，应保证此时虚拟机中的 Python 环境为 3.8.5。具体过程如下：

```
root@VXIDr0ysue:~/Desktop # 7z x frida-server-14.2.13-android-arm64.xz
```

```
root@VXIDr0ysue:~/Desktop # adb push frida-server-14.2.13-android-arm64
/data/local/tmp
```

```
root@VXIDr0ysue:~/Desktop # adb shell
bullhead:/ $ su
bullhead:/ # cd /data/local/tmp/
bullhead:/ # chmod 777 frida-server-14.2.13-android-arm64
bullhead:/ # ./frida-server-14.2.13-android-arm64
```

另外，可以使用以下命令查看手机的进程：

```
root@VXIDr0ysue:~/Desktop # frida-ps -U
```

1.2.6 Objection 的安装和使用

Objection 是基于 Frida 的命令行 Hook 工具，可以不写代码，简单地使用命令就可以对 Java 函数进行 Hook。安装 Objection 的命令如下：

```
root@VXIDr0ysue:~# pip install objection
```

如果使用基于特定 Frida 版本的 Objection，则需要先安装特定版本的 Frida 和 frida-tools。然后，在 Objection 的 Releases 页面中找到在该 Frida 版本之后发布的 Objection 版本。在本文的实验中，使用的是 12.8.0 版本的 Frida。首先切换到 Python 环境 3.8.0，再执行如下指令：

```
root@VXIDr0ysue:~/Desktop # pyenv local 3.8.0
```

```
root@VXIDr0ysue:~/Desktop # pip install frida==12.8.0
```

```
root@VXIDr0ysue:~/Desktop # pip install frida-tools==5.3.0
```

```
root@VXIDr0ysue:~/Desktop # pip install objection==1.8.4
```

按照这个顺序安装 Objection 时，系统会直接显示 Requirement already satisfied(需求已满足)，不会再下载新的 Frida 来安装了。完成后，下载 12.8.0 版本的 frida-server，具体过程可参照 1.2.5 节的步骤。接下来，在手机 Shell 中运行 12.8.0 版本的 frida-server，并使用 Objection 打开 Settings，如图 1-7 所示。



图 1-7 使用 Objection 打开 Settings

```
root@VXIDr0ysue:~/Desktop # pyenv local 3.8.0

root@VXIDr0ysue:~/Desktop # objection -g com.android.settings explore
Using USB device `LGE Nexus 5X`
Agent injected and responds ok!

| | | | | | | |
| . | . | | - | | | | | | . | | |
|__|__| |__|__|_|_|_|_|_|_|_|_|_|_
|__| (object) inject(ion) v1.8.4

Runtime Mobile Exploration
by: @leonjza from @sensepost

[tab] for command suggestions
com.android.settings on (google: 8.1.0) [usb] #
```

可以看到手机的设置打开了。

查看所有的类:

```
com.android.settings on (google: 8.1.0) [usb] # android hooking list classes
```

同理，如果使用 14.2.13 版本的 Frida，则在手机上运行 14.2.13 版本的 frida-server，还要将虚拟机上的 Python 环境切换至 3.8.5 版本，否则会报错。之后，再次使用 Objection 打开 Settings，我们将看到不同的 Objection 版本号。

1.3 Frida 基本源码开发环境搭建

本节继续讲解 Frida 基本源码开发环境的搭建。在这个过程中，我们需要用到带有包管理器的 Node.js。

打开网址 4（参见配书资源文件），选择 Installation instructions，然后执行以下命令：

```
root@VXIDr0ysue:~/Desktop# curl -fsSL https://deb.nodesource.com/setup_15.x | bash -  
  
root@VXIDr0ysue:~/Desktop# apt-get install -y nodejs  
  
root@VXIDr0ysue:~/Desktop# node -V  
v15.12.0
```

安装好 Node 后，就可以开发项目了：

```
root@VXIDr0ysue:~# cd Desktop  
  
root@VXIDr0ysue:~/Desktop# mkdir 20210105
```

启动 VS Code → 20210105 文件夹，创建文件 demo.js。此时，还没有代码提示功能，我们需要安装 Gum，以便在任意工程中获得 Frida 代码提示、补全和 API 查看功能。

```
root@VXIDr0ysue:~/Desktop# npm install --save @types/frida-gum
```

重启 VS Code，即可获得代码提示。VS Code 是指 Visual Studio Code，是微软公司向开发者们提供的一款跨平台源代码编辑器。

接下来，在 demo.js 中编写代码实现基本的 Hook 项目 easysol 中的 onCreate 函数，内容见代码清单 1-1。

代码清单 1-1 demo.js

```
setImmediate(function() {  
    Java.perform(function() {  
        Java.use("com.r0ysue.easysol.MainActivity").onCreate.implementation =  
function(x) {  
            console.log("Entering onCreate!");  
            return this.onCreate(x);  
        }  
    })  
})
```

打开终端 Terminal，使用以下指令运行脚本：

```
root@VXIDr0ysue:~/Desktop/20210105 # frida -U -f com.roysue.easysol -l demo.js
--no-pause
```

结果如图 1-8 所示，说明我们 Hook 上了。

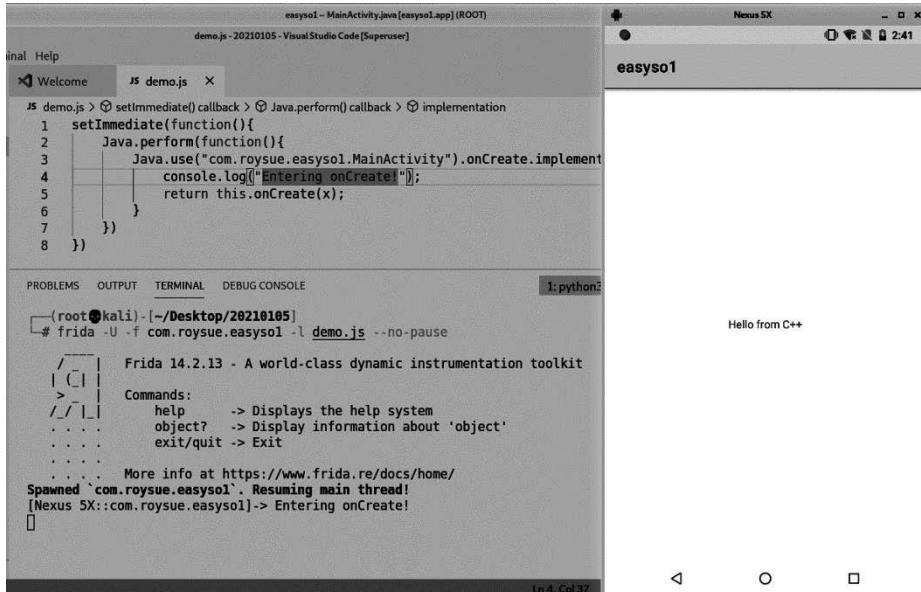


图 1-8 Hook onCreate 函数

继续尝试 Hook 项目 easysol 中的 stringFromJNI 方法，内容见代码清单 1-2。

代码清单 1-2 demo.js

```
setImmediate(function(){
    Java.perform(function(){
        Java.use("com.r0ysue.easysol.MainActivity").onCreate.implementation =
function(x){
            console.log("Entering onCreate!");
            return this.onCreate(x);
        }
        Java.use("com.r0ysue.easysol.MainActivity").stringFromJNI.implementation =
function(){
            var result = this.stringFromJNI();
            console.log("Return value of stringFromJNI is => ", result);
            return result;
        }
    })
})
```

结果如图 1-9 所示，说明我们 Hook 成功了。

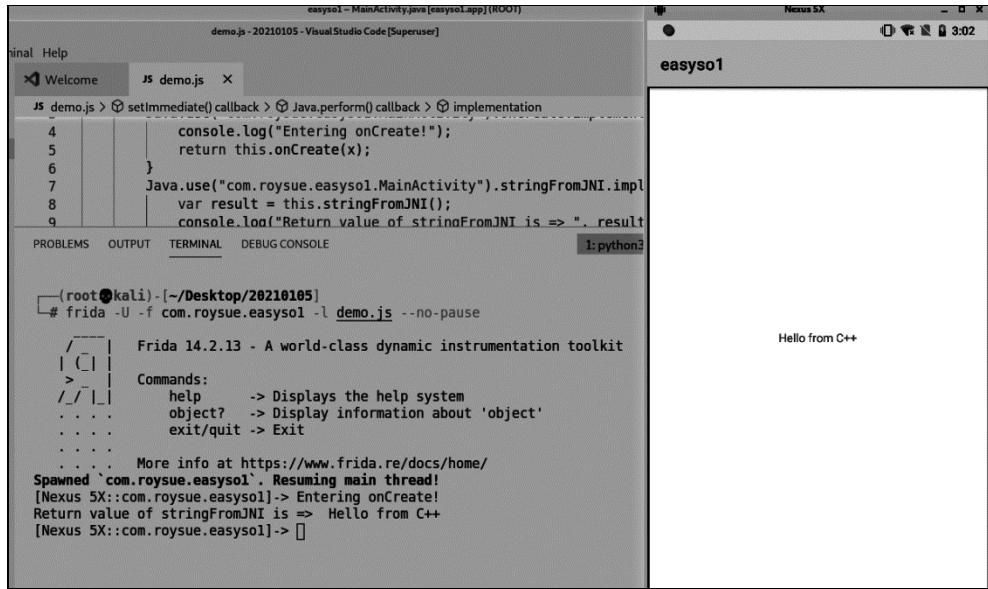


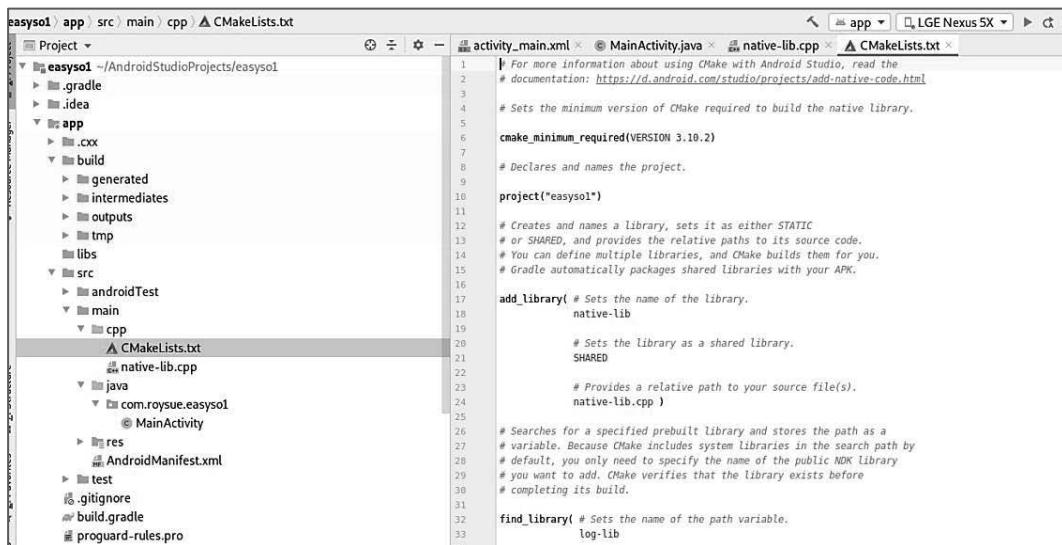
图 1-9 Hook stringFromJNI 方法

1.4 初识 NDK

笔者在这里推荐网址 5（参见配书资源文件）的 NDK 入门指南：读者可以参阅此文档来了解 NDK 开发的相关知识。

NDK 开发使用编译型语言 C++，相较于脚本型语言 Java，并不适合初学者。C/C++代码被编译成 CPU 代码，可以直接在 CPU 上运行，而 Java 代码会被编译成中间语言，在解释器中进行解释和执行。很明显，编译型语言执行起来效率更高，而脚本型语言的优势在于代码只需要编译一次，无须修改便可在任意平台上运行。因此，为了提高性能、保护程序的逻辑和代码，可以使用 C++ 来开发，C++ 的语言特性更加难以被反编译和分析。

在 NDK 入门指南文档中，比较关键的内容是“创建 CMake 构建脚本”。对于 CMake 如何对原生源文件进行编译，由 CMake 的构建脚本中的一些重要选项来指定。如果原生库已有 CMakeLists.txt 构建脚本，则可直接使用；否则需要自己创建一个编译脚本。CMakeLists.txt 脚本文件在项目的 app/src/main/cpp 路径下，如图 1-10 所示。



```

easysol > app > src > main > cpp > CMakeLists.txt
Project ④ ☰ ⌂ ⌂ ⌂ - activity_main.xml ✎ app LGE Nexus S5 ▶ ⌂
easysol ~ /AndroidStudioProjects/easysol
  ▷ gradle
  ▷ .idea
  ▷ app
    ▷ .cxx
  ▷ build
    ▷ generated
    ▷ intermediates
    ▷ outputs
    ▷ tmp
  ▷ libs
  ▷ src
    ▷ androidTest
    ▷ main
      ▷ cpp
        ▷ CMakeLists.txt
          ▷ native-lib.cpp
        ▷ java
          ▷ com.royse.easysol
            ▷ MainActivity
        ▷ res
        ▷ AndroidManifest.xml
      ▷ test
      ▷ .gitignore
      ▷ build.gradle
      ▷ proguard-rules.pro
  ▷ .gitignore
  ▷ build.gradle
  ▷ proguard-rules.pro

1  # For more information about using CMake with Android Studio, read the
2  # documentation: https://d.android.com/studio/projects/add-native-code.html
3
4  # Sets the minimum version of CMake required to build the native library.
5
6  cmake_minimum_required(VERSION 3.10.2)
7
8  # Declares and names the project.
9
10 project("easysol")
11
12 # Creates and names a library, sets it as either STATIC
13 # or SHARED, and provides the relative paths to its source code.
14 # You can define multiple libraries, and CMake builds them for you.
15 # Gradle automatically packages shared libraries with your APK.
16
17 add_library( # Sets the name of the library.
18   native-lib
19
20   # Sets the library as a shared library.
21   SHARED
22
23   # Provides a relative path to your source file(s).
24   native-lib.cpp
25
26   # Searches for a specified prebuilt library and stores the path as a
27   # variable. Because CMake includes system libraries in the search path by
28   # default, you only need to specify the name of the public NDK library
29   # you want to add. CMake verifies that the library exists before
30   # completing its build.
31
32   find_library( # Sets the name of the path variable.
33     log-lib
34

```

图 1-10 CMakeLists.txt 所在的路径

由于不同设备使用不同的 CPU，不同的 CPU 支持的指令集也不同，因此需要提供对应的二进制接口交互规则才能进行交互。Android ABI（Application Binary Interface，应用程序二进制接口）相当于 CPU 指令集的一种模式和调用规范。读者可参阅文档了解 Android ABI，在后续的章节中，我们还会进一步学习 Android ABI 的知识。了解 CPU 和架构对于开发非常重要。因为代码是直接在 CPU 上运行的，如果不了解 CPU 和架构，就无法理解不同的寄存器和指令代表的含义。不同的 CPU 架构所支持的 ABI 如图 1-11 所示。

简介 开始使用 概念 JNI 提示	支持的 ABI																	
构建您的项目 简介 ► ndk-build CMake 将 NDK 与其他构建系统配合使用 独立工具链	表 1. ABI 和支持的指令集。 <table border="1"> <thead> <tr> <th>ABI</th> <th>支持的指令集</th> <th>备注</th> </tr> </thead> <tbody> <tr> <td>armeabi-v7a</td> <td> <ul style="list-style-type: none"> armeabi Thumb-2 VFPv3-D16 </td> <td>与 ARMv5/v6 设备不兼容。</td> </tr> <tr> <td>arm64-v8a</td> <td> <ul style="list-style-type: none"> AArch64 </td> <td></td> </tr> <tr> <td>x86</td> <td> <ul style="list-style-type: none"> x86 (IA-32) MMX SSE/2/3 SSSE3 </td> <td>不支持 MOVBE 或 SSE4。</td> </tr> <tr> <td>x86_64</td> <td> <ul style="list-style-type: none"> x86-64 MMX SSE/2/3 SSSE3 SSE4.1、4.2 POPCNT </td> <td></td> </tr> </tbody> </table> <p>★ 注意：NDK 以前支持 ARMv5 (armeabi) 以及 32 位和 64 位 MIPS，但 NDK r17 已不再支持。</p>			ABI	支持的指令集	备注	armeabi-v7a	<ul style="list-style-type: none"> armeabi Thumb-2 VFPv3-D16 	与 ARMv5/v6 设备不兼容。	arm64-v8a	<ul style="list-style-type: none"> AArch64 		x86	<ul style="list-style-type: none"> x86 (IA-32) MMX SSE/2/3 SSSE3 	不支持 MOVBE 或 SSE4。	x86_64	<ul style="list-style-type: none"> x86-64 MMX SSE/2/3 SSSE3 SSE4.1、4.2 POPCNT 	
ABI	支持的指令集	备注																
armeabi-v7a	<ul style="list-style-type: none"> armeabi Thumb-2 VFPv3-D16 	与 ARMv5/v6 设备不兼容。																
arm64-v8a	<ul style="list-style-type: none"> AArch64 																	
x86	<ul style="list-style-type: none"> x86 (IA-32) MMX SSE/2/3 SSSE3 	不支持 MOVBE 或 SSE4。																
x86_64	<ul style="list-style-type: none"> x86-64 MMX SSE/2/3 SSSE3 SSE4.1、4.2 POPCNT 																	
架构和 CPU 简介 Android ABI CPU 功能 Neon 支持																		
编写 C/C++ 代码 简介 Android SDK 版本属性 C++ 支持 原生 API																		
调试和性能剖析 简介 通过 Android Studio 调试																		

图 1-11 不同的 CPU 架构支持的 ABI

通常情况下，我们从 APK 中解压出文件夹时，文件夹中包含的子文件夹会出现如图 1-11 所示的文件名，文件夹的命名表示对应的 ABI 架构。默认情况下，Gradle 会针对手机支持的 ABI 进行构建。我们可以从终端 Terminal 进入 app-debug.apk 文件所在的位置，使用 7z 解压文件，再从文件管理器打开 app-debug.apk 文件所在的目录，进入 lib 文件夹，可以看到为特定 ABI 生成的代码文件夹，其命名即为手机支持的 ABI 版本，如图 1-12 所示。

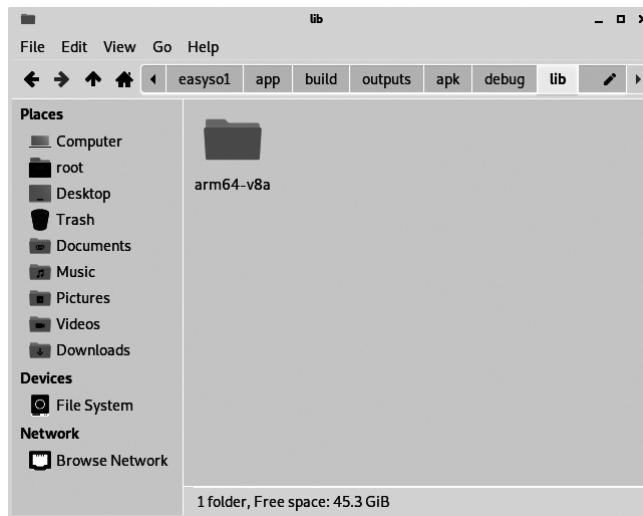


图 1-12 手机支持的 ABI 版本

如果需要为特定 ABI 生成代码，可通过在项目的 Gradle 中更改配置来针对不同的 ABI 进行构建，如图 1-13 所示。

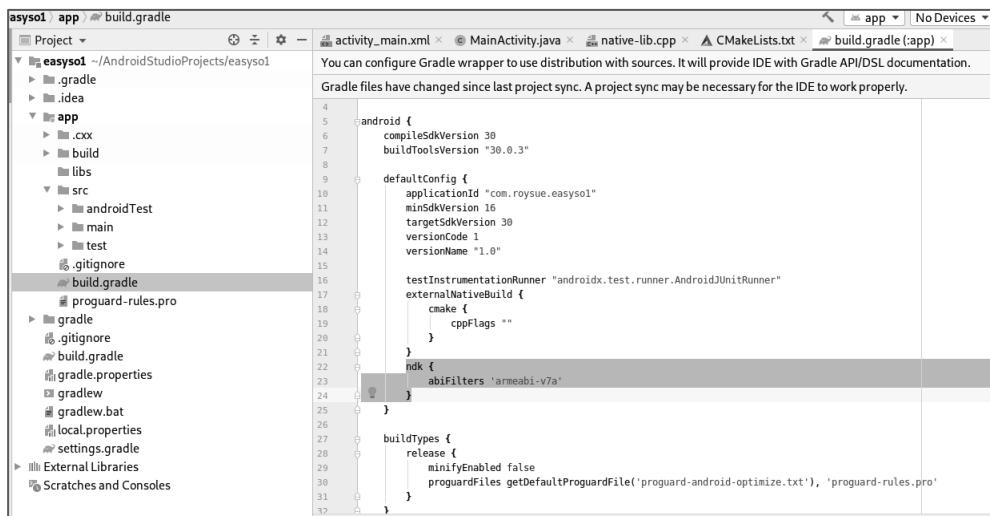


图 1-13 修改 Gradle 配置

在修改完成后，保存文件并重新运行 App，解压 app-debug.apk 文件后，打开 lib 文件夹，我们可以看到为特定 ABI 生成的代码文件夹的命名发生了变化，如图 1-14 所示。

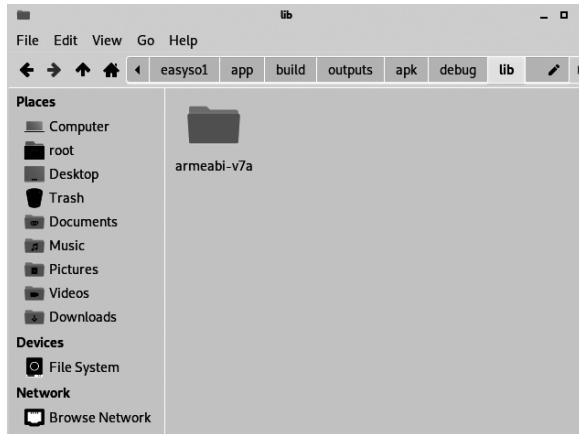


图 1-14 armeabi-v7a

查看系统主要的 ABI 接口需要用到 Objection。首先，安装 Wallbreaker 插件，启动 Objection：

```
root@VXIDr0ysue:~/Desktop # git clone git clone https://github.com/hluwa/Wallbreaker.git

root@VXIDr0ysue:~/Desktop# objection -g com.android.settings explore

com.android.settings on (google: 8.1.0) [usb] # plugin load /root/Desktop/Wallbreaker
Loaded plugin: wallbreaker
com.android.settings on (google: 8.1.0) [usb] # plugin wallbreaker classdump
android.os.Build
```

执行上述命令后，终端显示出系统主要的 ABI 接口，如图 1-15 所示。

<code>static String[] SUPPORTED_32_BIT_ABIS; => armeabi-v7a,armeabi</code>
<code>static String[] SUPPORTED_64_BIT_ABIS; => arm64-v8a</code>
<code>static String[] SUPPORTED_ABIS; => arm64-v8a,armeabi-v7a,armeabi</code>

图 1-15 系统主要的 ABI 接口

接下来，将 arm64-v8a 添加到 Gradle 配置文件的 NDK 中，运行 App，这时优先以 64 位运行（因为当前系统本来就是 64 位的）。

再次启动 Objection：

```
root@VXIDr0ysue:~/Desktop # objection -g com.roysue.easysol explore

com.roysue.easysol on (google: 8.1.0) [usb] # frida
-----
Frida Version 14.2.13
```

```

Process Architecture arm64
Process Platform linux
Debugger Attached False
Script Runtime QJS
Script Filename /script1.js
Frida Heap Size 7.5 MiB
-----
```

从 Process Architecture 表项可以看到进程架构是 64 位。此外，我们也可以把 App 强制安装成 32 位。卸载 App，并重新解压 app-debug.apk 文件，可以发现 lib 中有两个架构，如图 1-16 所示。

使用 ADB 强制安装指定架构：

```

root@VXIDr0ysue:~/.../build/outputs/apk/debug # adb install -r -t --abi armeabi-v7a
app-debug.apk
Performing Streamed Install Success
```

然后用 Objection 查看。如果 Objection 报错，则可切换 Python 和 Frida 版本再次尝试。如图 1-17 所示，我们可以看到 Process Architecture 变成了 32 位的。

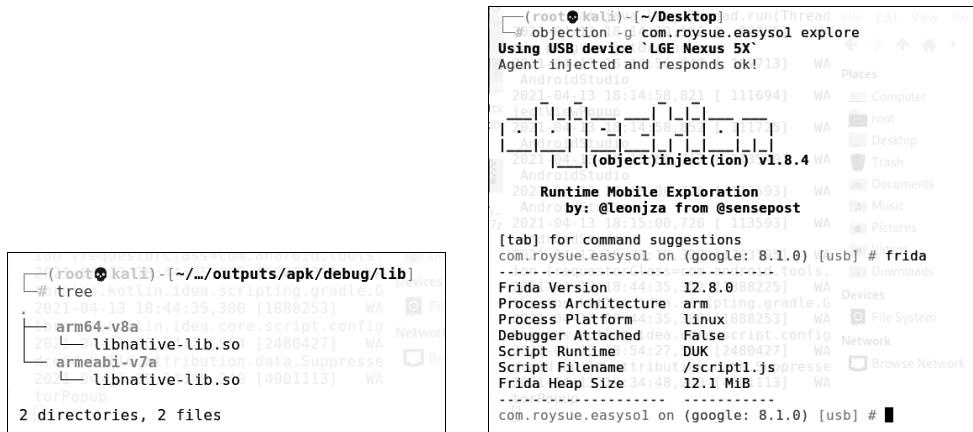


图 1-16 tree 查看两个架构

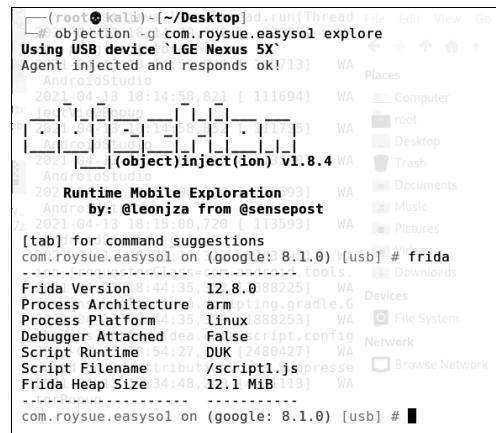


图 1-17 强制安装成 32 位架构

1.5 其他工具

至此，我们基本完成了环境准备工作。另外，笔者还要推荐一些在日常工作中使用的小工具。尽管这些工具可能不会直接对工作产生影响，但一旦拥有了这些工具，用户在日常工作中可以更准确地掌握自己的工具环境。

首先，推荐使用 htop 这款加强版 top 工具。htop 可以动态地查看当前活跃的、占用资源较多的进程，如图 1-18 所示。这一点在编译 Android 源码时非常好用，当我们执行 make 命令之后，可以肉眼可见的速度看到内存 Mem 跑到底之后，开始侵占 Swp 的进度条。Uptime 表示系统的运行时间，

Load average 表示平均负载, 比如我们的系统有一个四核 CPU, 在平均负载跑到4时说明系统满载了。有关 htop 的其他操作教程, 读者可以在网上搜索。

另外, 还要推荐一款实时查看系统网络负载的工具 jnettop。在安装和使用软件(比如 Frida)的过程中, jnettop 可以实时查看其下载和安装进度。甚至在 AOSP 编译过程中, 仍然可以观察到它连接到国外服务器进行依赖包下载等操作。除此之外, 在抓包时打开这个工具往往会有奇效, 比如实时查看对方的 IP 等信息。jnettop 界面如图 1-19 所示, 我们可以清楚地看到主机连接的远程 IP、端口、速率以及协议等信息。

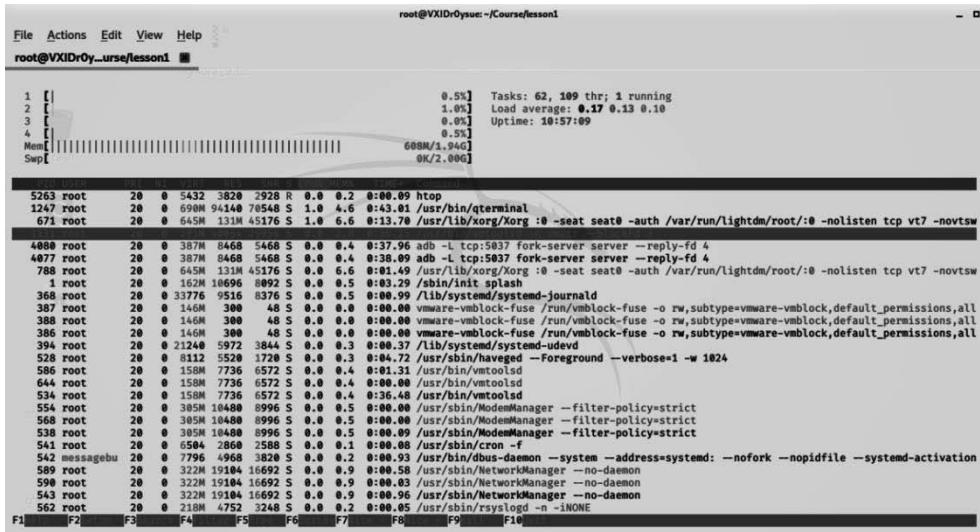


图 1-18 htop 界面

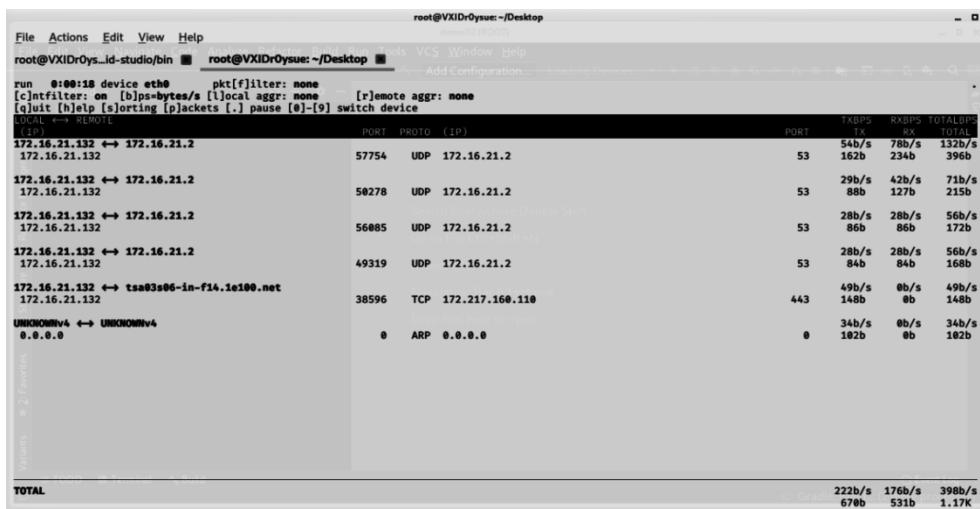


图 1-19 jnettop 界面

1.6 本 章 小 结

本章主要介绍了 Android 应用安全技术中常用的一些基础环境配置，虽然这一章没有过多的技术内容，但它却是整本书的基石。良好的环境配置能够节省学习过程中的时间，从而大大提高学习效率。

Android SO 动态调试入门



通过第 1 章的学习，我们完成了基本的环境配置。在本章中，我们将介绍 NDK 程序的动态调试、交叉编译以及汇编/反汇编工具的简单应用。

2.1 Android SO 基本动态分析调试

SO (Shared Object, 共享库) 是机器可以直接运行的二进制代码，它是 Android 上的动态链接库，类似于 Windows 上的 DLL。Android SDK 编译出来的 SO 文件有好几种，以适配不同的 CPU 架构。一般人并不需要了解 SO 文件，但如果要做逆向分析工作，就必须了解 SO 文件，并且具备对 SO 文件进行分析和调试的能力。

2.1.1 第一个 NDK 程序

ndk-samples 中提供了非常多的案例，我们选择其中最简单的 hello-jni，打开它来安装一下，结果如图 2-1 所示。

在 hello-jni.c 文件中，以下代码决定了显示内容：

```
return (*env)->NewStringUTF(env, "Hello from JNI !\nCompiled with ABI \" ABI \".");
```

其中，ABI 是预定义的。如果想要更改显示的内容，则需要在项目的 build.gradle 文件中添加如图 2-2 所示的代码。

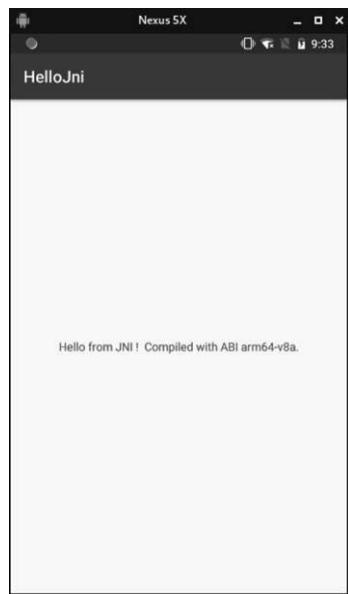


图 2-1 hello.jni 运行结果



什么是 ABI?

每一种 CPU 架构都定义了一种 ABI (Application Binary Interface, 应用程序二进制接口), ABI 定义了对应的 CPU 架构能够执行的二进制文件(如 SO 文件)的格式规范, 决定了二进制文件如何与系统进行交互。

The screenshot shows the Android Studio interface. On the left, the project structure for 'hello-jni' is visible, including .google, .gradle, .idea, app (containing .cxx, build, src, .gitignore, build.gradle, proguard-rules.pro), build (containing build, kotlin, gradle, build.gradle), and a README.md file. On the right, the build.gradle (app) file is open in the code editor. The code defines the application's configuration, including the minSdkVersion (23), targetSdkVersion (29), versionCode (1), and versionName ("1.0"). It also specifies the NDK configuration with abiFilters set to 'armeabi-v7a'.

```

Gradle files have changed since last project sync. A project sync may be required for external tool support.
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 29
    ndkVersion '21.2.6472646'

    defaultConfig {
        applicationId 'com.example.hellojni'
        minSdkVersion 23
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"
        ndk {
            abiFilters 'armeabi-v7a'
        }
    }
}

```

图 2-2 改变 hello-jni 显示的内容

删除 productFlavors 中的 arm64-v8a, 保留 armeabi-v7a, 以免配置被覆盖, 如图 2-3 所示。

The screenshot shows the productFlavors section of the build.gradle file. It contains two entries: arm8 and x86_64. The arm8 entry includes a dimension 'cpuArch' and an NDK configuration with abiFilters set to 'armeabi-v7a'. The x86_64 entry includes a dimension 'cpuArch' and an NDK configuration with abiFilters set to 'x86_64', 'x86'.

```

productFlavors {
    arm8 {
        dimension 'cpuArch'
        ndk {
            abiFilters 'armeabi-v7a'
        }
    }
    x86_64 {
        dimension 'cpuArch'
        ndk {
            abiFilters 'x86_64', 'x86'
        }
    }
}

```

图 2-3 保留 armeabi-v7a

在弹出的提示中单击 Sync Now, 再次进行编译并运行, 可以看到应用界面显示出 armeabi-v7a。

2.1.2 动态调试 NDK 程序

通过运行 hello-jni, 我们对 NDK 开发有了一个初步的印象。接下来, 将通过编写一个简单的 C 语言程序来学习 SO 的基本动态分析调试方法。

首先, 在 app/src/main/cpp 路径下新建一个文件 roysue.c, 右击 cpp 文件夹选择 New→C/C++ Source File。我们可以复制 ndk-samples 中的程序 hello-jni, 并在此基础之上进行改动, 修改后的代码如下:

```
#include <string.h>
#include <jni.h>
```

```
/* This is a trivial JNI example where we use a native method
 * to return a new VM String. See the corresponding Java source
 * file located at:
 *
 *   hello-jni/app/src/main/java/com/example/hellojni/HelloJni.java
 */
JNIEXPORT jstring JNICALL
Java_com_roysue_easysol_MainActivity_stringFromJNI( JNIEnv* env, jobject thiz )
{
    #if defined( __arm__ )
        #if defined( __ARM_ARCH_7A__ )
            #if defined( __ARM_NEON__ )
                #if defined( __ARM_PCS_VFP__ )
                    #define ABI "armeabi-v7a/NEON (hard-float)"
                #else
                    #define ABI "armeabi-v7a/NEON"
                #endif
            #else
                #if defined( __ARM_PCS_VFP__ )
                    #define ABI "armeabi-v7a (hard-float)"
                #else
                    #define ABI "armeabi-v7a"
                #endif
            #endif
        #else
            #define ABI "armeabi"
        #endif
    #else
        #if defined( __i386__ )
            #define ABI "x86"
        #elif defined( __x86_64__ )
            #define ABI "x86_64"
        #elif defined( __mips64__ ) /* mips64el-* toolchain defines __mips__ too */
            #define ABI "mips64"
        #elif defined( __mips__ )
            #define ABI "mips"
        #elif defined( __AArch64__ )
            #define ABI "arm64-v8a"
        #else
            #define ABI "unknown"
        #endif
    #endif

    return (*env)->NewStringUTF(env, "Hello from JNI ! Compiled with ABI " ABI ".");
}
```

此时，我们会在代码区域上方看到一行提示，如图 2-4 所示。

This file is not part of the project. Please include it in the appropriate build file (build.gradle, CMakeLists.txt or Android.mk etc.) and sync the project.

```

1 // Created by root on 4/19/21.
2
3
4
5 #include <string.h>
6 #include <jni.h>
7
8 /* This is a trivial JNI example where we use a native method
9 * to return a new VM String. See the corresponding Java source
10 * file located at:
11 *
12 *   hello-jni/app/src/main/java/com/example/hellojni/HelloJni.java
13 */
14 JNIEXPORT jstring JNICALL
15 Java_com_roysue_easysol_MainActivity_stringFromJNI( JNIEnv* env, jobject thiz )
16 {
    ...
}

```

图 2-4 Sync 提示

根据提示，打开 build.gradle，在 externalNativeBuild 中可以看到项目使用 CMake 的编译系统生成 CMakeLists.txt。因此，当我们新建文件之后，需要把它加入 CMakeLists.txt 中，否则编译系统无法识别新建的文件。打开 CMakeLists.txt，将 native-lib 全部替换为 roysue，再将 add_library 中的 roysue.cpp 改为 roysue.c，然后把不需要的 native-lib.cpp 文件删掉。

回到 roysue.c，在提示中单击 Sync Now 进行同步。除此之外，还需要在 MainActivity 中修改加载模块为 roysue，如图 2-5 所示。

```

1 package com.roysue.easysol;
2
3 import ...
4
5 public class MainActivity extends AppCompatActivity {
6
7     // Used to load the 'native-lib' library on application startup.
8     static {
9         System.loadLibrary("roysue");
10    }
11
12    @Override
13
14
15

```

图 2-5 修改加载模块

这时就可以运行项目了。

接下来，我们调试一个简单的 C 语言程序来实现数字求和，并以 info 的日志权限打印出结果。

首先，在 roysue.c 中包含头文件<android/log.h>，在函数体中添加一段 C 循环代码如下：

```

int i=1, sum=0;
while(i<=10) {
    sum+=i;
    i++;
    __android_log_print(ANDROID_LOG_INFO, "roysue", "now sum is %d", sum);
}

```

运行程序，可以在日志输出中看到迭代求和结果，如图 2-6 所示。

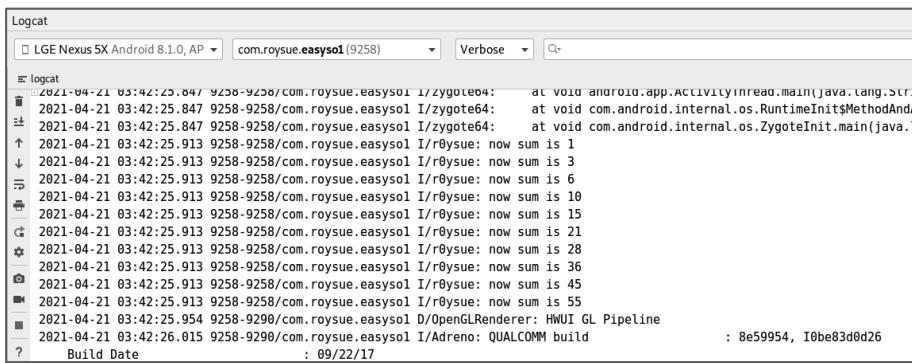


图 2-6 以 info 的日志权限打印出求和结果

对于使用 Android Studio 调试应用的具体流程，网址 6（参见配书资源文件）的 Android 官方文档提供了详细说明。

这里，我们以 roysue.c 为例进行演示。

首先，在“`sum+=1;`”一行处加上断点，单击调试键开始调试，程序遇到断点后，进入 Debugger，就可以开始调试(Debug)了。调试主要使用图 2-7 中框出的前 4 个按钮，从左至右依次是：Step Over、Step Into、Force Step Into、Step Out，如图 2-7 所示。

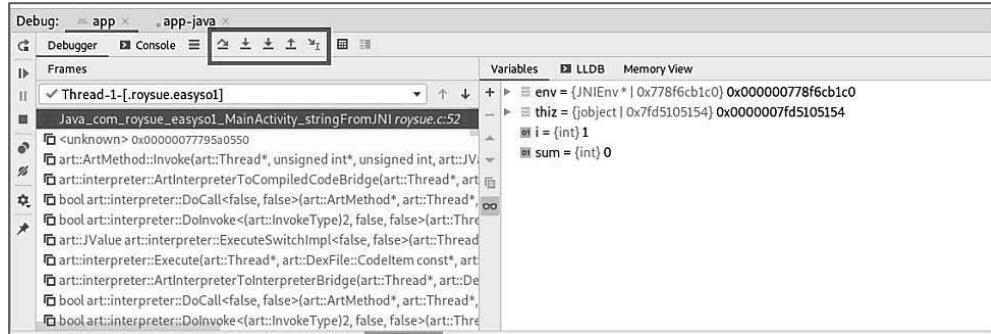


图 2-7 Debug 的 5 个按钮（常用前 4 个）

- **Step Over:** 前进到代码中的下一行，如果下一行是方法调用，则不进入方法。
- **Step Into:** 前进到代码中的下一行，如果下一行是方法调用，则进入方法（仅限于自定义的方法）。
- **Force Step Into:** 前进到代码中的下一行，如果下一行是方法调用，则进入方法（强制进入，包括系统方法）。
- **Step Out:** 前进到当前方法之外的下一行（跳出方法）。

主要就是通过操作上述 4 个按钮来进行调试，观察图 2-7 右边 Variables 区域所显示的变量信息的变化，以便对程序进行修改。

从图 2-7 中可以看到，除了 Variables 标签页以外，还有一个标签页 LLDB，进入后可以看到 LLDB 的命令行。在命令行中，输入 LLDB 命令可以实现很多强大的功能，比如打印变量、寻址、调用堆栈等，通过这些命令可以有效地帮助我们调试 NDK 程序。

例如，在 LLDB 的命令行输入 dis (disassemble) 执行反汇编操作，如图 2-8 所示。

```

Variables LLDB Memory View
(lldb) script exec("if libstdcxx_printers_available: import printers")
(lldb) script exec("if libstdcxx_printers_available: printers.register_libstdcxx_printers(None)")
(lldb) script exec("if lldb.debugger.GetCategory('libstdc++-v6').IsValid(): lldb.debugger.GetCategory('gnu-libstdc++')")
(lldb) dis
libroysue.so`Java_com_royse_easyso_MainActivity_stringFromJNI:
    0x777935f63c <+0>: sub    sp, sp, #0x30          ; =0x30
    0x777935f640 <+4>: stp    x29, x30, [sp, #0x20]
    0x777935f644 <+8>: add    x29, sp, #0x20          ; =0x20
    0x777935f648 <+12>: mov    w8, #0x1
    0x777935f64c <+16>: stur   x0, [x29, #-0x8]
    0x777935f650 <+20>: str    x1, [sp, #0x10]
    0x777935f654 <+24>: str    w8, [sp, #0xc]
    0x777935f658 <+28>: str    wzr, [sp, #0x8]
    0x777935f65c <+32>: ldr    w8, [sp, #0xc]
    0x777935f660 <+36>: cmp    w8, #0xa             ; =0xa
    0x777935f664 <+40>: cset   w8, gt
    0x777935f668 <+44>: tbnz  w8, #0x0, 0x777935f6a8  ; <+108> at roysue.c:57
-> 0x777935f66c <+48>: ldr    w8, [sp, #0xc]
    0x777935f670 <+52>: ldr    w9, [sp, #0x8]

```

图 2-8 LLDB 反汇编

2.1.3 交叉编译

通过 2.1.2 节的学习，我们基本了解了如何在 Android Studio 上调试一个 NDK 程序。然而在实际应用中，如何将一个 C 语言程序移植到已有的 NDK 工程中呢？接下来，我们将通过演示一个稍微复杂一些的案例来介绍交叉编译。

以 MD5 算法（消息摘要算法）为例，MD5 是计算机安全领域广泛使用的一种散列函数，用以提供消息的完整性保护，可作为数字签名入门算法。首先，在 GitHub 上找一个 C 语言实现的 MD5 算法，复制到本地：

```

root@VXIDr0ysue:~/Desktop # mkdir 20210110
root@VXIDr0ysue:~/Desktop # cd 20210110
root@VXIDr0ysue:~/Desktop/20210110 # git clone https://github.com/pod32g/MD5.git

```

用 VS Code 打开 md5.c，可以在“扩展”搜索 C/C++ 插件并进行安装，以便查看函数实现。

安装好插件后，按住 Ctrl 键，单击函数即可查看函数实现。编译并运行 md5.c：

```

root@VXIDr0ysue:~/Desktop/20210110/md5 # gcc -o md5 md5.c
root@VXIDr0ysue:~/Desktop/20210110/md5 # ls
md5  md5.c  README.md

```

```
root@VXIDr0ysue:~/Desktop/20210110/md5 # ./md5 r0ysue
8e2b0b38f557578f578ef21b33e4df0d
```

通过访问网址 7(参见配书资源文件), 将计算结果在 CyberChef 网站上与标准结果进行比对, 看是否相同, 若相同, 则说明这个程序是一个标准的 MD5 算法的实现。

查看编译后的 MD5 可执行文件格式:

```
root@VXIDr0ysue:~/Desktop/20210110/md5 # file md5
md5: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=14a627a2ddebaa8f4b86b02bb9b259922ed34255, for GNU/Linux 3.2.0, not stripped
```

如果我们要把它编译到手机上, 该如何操作呢? 由于手机与虚拟机指令级架构不同, 因此需要借助 NDK 提供的工具链对现有的 C 语言代码进行构建, 即进行交叉编译。关于交叉编译的具体用法, 可参考网址 8 (参见配书资源文件) 文档。

下面, 我们以 md5.c 为例进行交叉编译的演示。

首先, 进入 ndk 目录, 选择最新版本:

```
root@VXIDr0ysue:~/Desktop/20210110/md5 # cd ~/Android/Sdk/ndk

root@VXIDr0ysue:~/Android/Sdk/ndk # ls
21.1.6352462 21.2.6472646 23.0.7123448

root@VXIDr0ysue:~/Android/Sdk/ndk # cd 23.0.7123448

root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448 # cd toolchains

root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448/toolchains # cd llvm

root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448/toolchains/llvm # cd prebuilt

root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448/toolchains/llvm/prebuilt # cd
linux-x86_64

root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448/toolchains/llvm/prebuilt/linux-x86_
64 # cd bin

root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448/toolchains/llvm/prebuilt/linux-x86_
64/bin # pwd
/root/Android/Sdk/ndk/23.0.7123448/toolchains/llvm/prebuilt/linux-x86_64/bin
```

复制 pwd 返回的路径, 回到 VS Code, 打开终端 Terminal, 执行以下命令:

```
root@VXIDr0ysue:~/Desktop/20210110/md5 #
/root/Android/Sdk/ndk/23.0.7123448/toolchains/llvm/prebuilt/linux-x86_64/bin/clang
-target AArch64-linux-android21 md5.c
```

```
root@VXIDr0ysue:~/Desktop/20210110/md5 # ls
a.out  md5  md5.c  README.md

root@VXIDr0ysue:~/Desktop/20210110/md5 # file a.out
a.out: ELF 64-bit LSB pie executable, ARM AArch64, version 1 (SYSV), dynamically linked,
interpreter /system/bin/linker64, not stripped
```

可以看到，生成的 a.out 文件是在 ARM 架构的 CPU 上运行而生成的。接下来，将 a.out 文件推送到手机上：

```
root@VXIDr0ysue:~/Desktop/20210110/md5 # adb push a.out /data/local/tmp
a.out: 1 file pushed, 0 skipped. 0.3 MB/s (8304 bytes in 0.029s)
```

在手机中查看 a.out 文件，并加上权限：

```
root@VXIDr0ysue:~# adb shell
bullhead:/ $ su
bullhead:/ # cd /data/local/tmp
bullhead:/data/local/tmp # chmod 777 a.out
bullhead:/data/local/tmp # ./a.out r0ysue
8e2b0b38f557578f578ef21b33e4df0d
```

以上就是交叉编译到手机上直接运行的流程。可以将之前复制的 clang 路径加入 zsh 的路径，这样使用的时候更方便一些：

```
root@VXIDr0ysue:~ # nano ~/.zshrc
```

在打开的文件末尾添加：

```
export PATH="/root/Android/Sdk/ndk/23.0.7123448/toolchains/llvm/prebuilt/
linux-x86_64/bin:$PATH"
```

验证：

```
root@VXIDr0ysue:~ # clang --version
Debian clang version 11.0.1-2
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

再把它移植到我们开发的工程中。

首先在 roysue.c 中添加 md5.c 的头文件，再将除 main 函数之外的其他部分代码复制到 roysue.c 中。

按照原 md5.c 的算法逻辑，在 Java_com_roysue_easyso1_MainActivity_stringFromJNI 中添加以下代码实现 MD5 算法并在日志（log）中打印出结果。

```
char* msg = "r0ysue";
size_t len = strlen(msg);
uint8_t result[16];
```

```

char* r = (char*)malloc(16);
memset(r, 0x00, sizeof(char) *16);

char* final = (char*)malloc(16);
memset(final, 0x00, sizeof(char) *16);

md5((uint8 t*)msg, len, result);

for (int i = 0; i < 16; i++) {
    sprintf(r, "%2.2x", result[i]);
    android log print(ANDROID LOG INFO, "r0ysue", "Hi,now i is %s", r);
    sprintf(final, "%s%s", final, r);
}

jstring jresult = (*env)->NewStringUTF(env, final);
return jresult;

```

编写 MD5 算法的代码时，需要注意将输出结果作为一个完整的字符串返回给前端显示，而不能按照原程序逐个输出。此外，还要遵循 C 语言的编写规范。例如，调用 memset()方法将存放字符串的数组清空后再赋值。

修改配置文件 build.gradle，在 NDK 中只保留 32 位的 ABI，即 armeabi-v7a，然后运行即可。

2.2 LLDB 动态调试（三方）Android SO

LLDB 调试第三方 App 可以使用以下两个方法之一：

- App 处于调试（Debug）模式、APK 包中的 debuggable==true（需要重新打包，或者 Xposed/Frida 去 Hook）。
- 手机是 AOSP 系统，编译成 userdebug 模式（N5X、Sailfish）。

第一种方法容易失败，笔者在这里推荐第二种方法，使用第二种方法可以调试手机中所有的 App。

对于 LLDB 调试工具的使用可参考网址 9 和网址 10 中的文章（参见配书资源文件）。

在 2.1 节中，我们使用了 Android Studio 中自带的 LLDB 调试。因为是首次调试，lldb-server 会被自动推送（push）到手机/data/local/tmp 目录下，Android Studio 调试控制台记录如图 2-9 所示。

```

05/04 19:28:17: Launching 'app' on LG Nexus 5X.
App restart successful without requiring a re-install.
$ adb shell am start -n "com.roysue.easysol/com.roysue.easysol.MainActivity" -a android.intent.action.MAIN -c android.intent.category.LAUNCHER -D
Waiting for application to come online: com.roysue.easysol | com.roysue.easysol.test
Connected to process 10596 on device 'lg-nexus_5x-0bd7d2df3758b2f9'.
Connecting to com.roysue.easysol
Capturing system logcat messages from application. This behavior can be disabled in the "Logcat output" section of the "Debugger" settings page.
W/ActivityThread: Application com.roysue.easysol is waiting for the debugger on port 8100..
I/System.out: Sending WAIT chunk
Now Launching Native Debug Session
$ adb shell cat /data/local/tmp/lldb-server | run-as com.roysue.easysol sh -c 'cat > /data/data/com.roysue.easysol/lldb/bin/lldb-server && chmod 700 /data/data/com.roysue.easysol/lldb/bin/lldb-server'
$ adb shell cat /data/local/tmp/start_lldb_server.sh | run-as com.roysue.easysol sh -c 'cat > /data/data/com.roysue.easysol/lldb/bin/start_lldb_server.sh && chmod 700 /data/data/com.roysue.easysol/lldb/bin/start_lldb_server.sh'
Starting LLDB server: /data/data/com.roysue.easysol/lldb/bin/start_lldb_server.sh /data/data/com.roysue.easysol/lldb unix-abstract /com.roysue.easysol-0 platform-1620127700
Debugger attached to process 10593

```

图 2-9 ADB push lldb-server

如果没有调试过，则可通过以下途径查看 lldb-server 的位置：

```

root@VXIDr0ysue:~ # cd ~/Android/Sdk/ndk

root@VXIDr0ysue:~/Android/Sdk/ndk # cd 23.0.7123448

root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448 # tree -NCfh | grep -i lldb-server
[ 28M] ./toolchains/llvm/prebuilt/linux-x86_64/lib64/clang/11.0.5/lib/linux/AArch64/ll
db-server
[ 25M] ./toolchains/llvm/prebuilt/linux-x86_64/lib64/clang/11.0.5/lib/linux/arm/ll
db-server
[ 26M] ./toolchains/llvm/prebuilt/linux-x86_64/lib64/clang/11.0.5/lib/linux/i386/ll
db-server
[ 29M] ./toolchains/llvm/prebuilt/linux-x86_64/lib64/clang/11.0.5/lib/linux/x86_64/ll
db-server

```

我们把 64 位的和 32 位的 lldb-server 复制出来：

```

root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448 # cp
toolchains/llvm/prebuilt/linux-x86_64/lib64/clang/11.0.5/lib/linux/AArch64/lldb-server
~/Desktop/20210110/ls64

root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448 # cp
toolchains/llvm/prebuilt/linux-x86_64/lib64/clang/11.0.5/lib/linux/arm/lldb-server
~/Desktop/20210110/ls

```

64 位和 32 位的 lldb-server 并不是混用的，至于如何选择，需要与进程相匹配。将 ls 与 ls64 用 ADB 推送（push）到 /data/local/tmp 目录下：

```

root@VXIDr0ysue:~ # cd Desktop/20210110

root@VXIDr0ysue:~ # ls
ls  ls64  MD5

```

```
root@VXIDr0ysue:~ # adb push ls* /data/local/tmp
ls: 1 file pushed, 0 skipped. 12.5 MB/s (26343620 bytes in 2.018s)
ls64: 1 file pushed, 0 skipped. 14.4 MB/s (28965712 bytes in 1.924s)
2 files pushed, 0 skipped. 13.0 MB/s (55309332 bytes in 4.062s)
```

进入手机目录给 ls 和 ls64 添加权限，然后启动 ls64：

```
bullhead:/data/local/tmp # ls
bullhead:/data/local/tmp # chmod 777 * ls*
bullhead:/data/local/tmp # ./ls64 platform --listen "0.0.0.0:10086" --server
```

查看 lldb 命令在哪里：

```
root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448 # tree -NCfh | grep -i lldb
— [ 91] .. /ndk-lldb
|   |   |   [229K] ./toolchains/llvm/prebuilt/linux-x86_64/bin/lldb
|   |   |
[126K] ./toolchains/llvm/prebuilt/linux-x86_64/bin/lldb-argdumper
|   |   [ 98] ./toolchains/llvm/prebuilt/linux-x86_64/bin/lldb.sh

root@VXIDr0ysue:~/Android/Sdk/ndk/23.0.7123448 # cd

root@VXIDr0ysue:~ # lldb
(lldb) platform select remote-android
Platform: remote-android
Connected: no
(lldb) platform connect connect://00d7d2df3758b2f9:10086
Platform: remote-android
Triple: AArch64-unknown-linux-android
OS Version: 27 (3.10.73-g89fd15db99aa)
Hostname: localhost
Connected: yes
WorkingDir: /data/local/tmp
Kernel: #1 SMP PREEMPT Thu Oct 11 19:31:31 UTC 2018
(lldb)
```

进入 platform 使用 connect 连接时，如果手机是使用 USB 连接的，则要在 connect://后输入 adb devices 显示的设备名称；如果是使用网络连接的，则需要输入手机的 IP 地址。

然后，根据需要执行以下操作。

查看 easyso 进程号：

```
bullhead:/ # ps -e |grep easy
u0_a102 16289 569 4310276 63632 SyS_epoll_wait 78120c83f8 S com.roysue.easysol
```

回到 LLDB，attach easyso 进程：

```
(lldb) attach -p 16289
```

```
Process 16289 stopped
```

查看当前线程：

```
(lldb) thread list
```

查看当前执行的函数：

```
(lldb) dis
libc.so`__epoll_pwait:
```

关于调试更具体的内容，我们会在后续章节展开讲解。

2.3 Capstone/Keystone/Unicorn（反）汇编器

在之前的内容中，我们了解到使用 LLVM 可以进行一定的汇编与反汇编。本节介绍 Capstone/Keystone（反）汇编器。

首先，我们简单回顾一些基本概念。

- 汇编（编译）：指将源码转换为机器码的过程，编译的过程用的是 LLVM。
- 反汇编（反编译）：指将机器码转换为汇编语言代码的操作，用的是 LLVM 的 LLDB 调试器的 Disassembler 功能，将当前的机器码反汇编成更容易理解的指令。

汇编与反汇编是互逆的过程。

我们可以用 Capstone、Keystone 和 Unicorn 实现同样的效果。为什么使用这些工具呢？因为 LLVM 工具链使用起来并不是很方便，尤其是当我们对 SO 文件进行自定义修改的时候。

接下来，我们将对 Capstone、Keystone 以及 Unicorn 进行简单的介绍。这 3 款工具是由同一个团队开发的，其中，最先发布的是 Capstone，而 Unicorn 将是我们课程的重点。模拟执行是指对于一个 SO 文件，我们可以在不用知道其实现细节的情况下直接运行它并获取结果。

Capstone 和 Keystone 的实现基于 LLVM。LLVM 项目是一个模块化、可重用的编译器和工具链技术集合。例如，LLVM 将我们编写的 C 语言代码编译成中间状态代码 LLVM IR，然后通过后端将中间代码编译成各个平台的机器码，从而可以在不同的目标机器上运行。LLVM 的架构如图 2-10 所示。

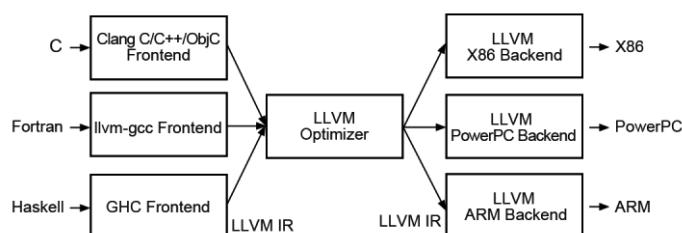


图 2-10 LLVM 的架构

02

反编译器 Capstone 是一个精准裁剪的 LLVM 的子模块，可以将指定的 SO 文件中的二进制数据转换成指令并返回结果。相对应的是汇编器 Keystone，可以将一个或一组指令转换成二进制数据并返回结果。

Keystone 相比于 LLVM，是非常轻量化的中间件。一个基于 Keystone 的工具 Keypatch，其操作界面如图 2-11 所示，可以通过输入指令得到相应的机器码。



图 2-11 Keypatch 工具的操作界面

我们可以用 Keypatch 来进行硬编码，即打开 SO 文件并直接修改其中的二进制数据。与 Capstone 和 Keystone 不同，Unicorn 是经过精准裁剪的 QEMU。QEMU 本身是一个完整的模拟器。Android Studio 自带的模拟器也是基于 QEMU 实现的。Unicorn 是一个基于 QEMU 的 CPU 模拟器，它可以直接运行机器码并忽略设备之间的差异。在某些场景下，我们可能只需要模拟代码的执行，而不需要一个真实的 CPU 来执行这些操作。例如，我们当前使用的这个虚拟机是 x86 架构的，如图 2-12 所示，我们可以在此虚拟机上使用 Unicorn 运行 ARM 架构的二进制文件。

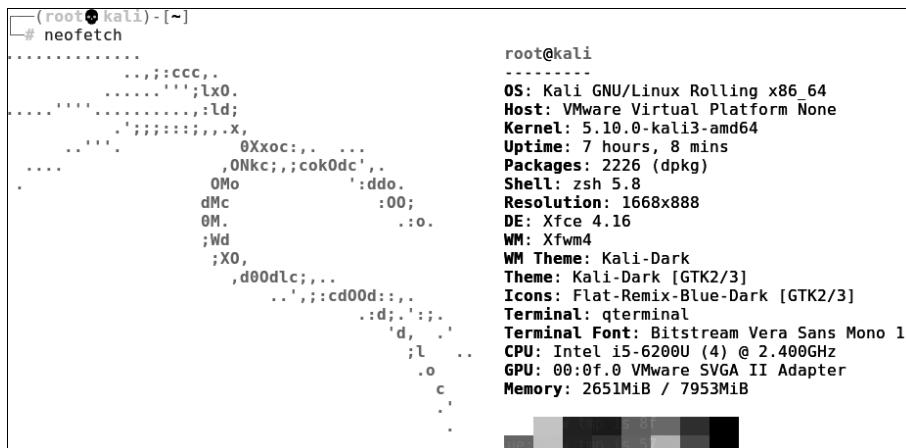


图 2-12 使用 neofetch 查看系统信息

2.4 Frida 动态调试 Android Native 部分

本节将介绍一款调试工具 Dwarf，读者简要了解即可。

Dwarf 是一个基于 Frida 开发的调试器，它将 Capstone、Keystone 和 Unicorn 集成到自己的框架中。在使用 Dwarf 调试器之前，需要先启动 frida-server：

```
bullhead:/data/local/tmp # ./frida-server-12.8.0-android-arm64
```

安装 Dwarf：

```
root@VXIDr0ysue:~ # cd Desktop/20210110
root@VXIDr0ysue:~/Desktop/20210110 # git clone https://github.com/iGio90/Dwarf
root@VXIDr0ysue:~/Desktop/20210110 # cd Dwarf
root@VXIDr0ysue:~/Desktop/20210110/Dwarf # pip3 install -r requirements.txt
root@VXIDr0ysue:~/Desktop/20210110/Dwarf # python3 dwarf.py
```

本质上，Dwarf 使用了 Frida 断点功能，然后使用 Capstone 将当前的指令进行反编译，再用 Keystone 将用户修改的内容写入内存区域。

我们也可以直接使用 Frida 的 Capstone 引擎，因为该引擎已经内置在 Frida 中。

2.5 Frida Instruction 模块动态反汇编

Frida 的 Instruction 模块也是通过集成 Capstone 实现对目标内存地址中指令的解析。本节将演示 Frida Instruction 模块的用法。

首先，在手机上运行 12.8.0 版本的 frida-server，并切换到相对应的 Python 环境 3.8.0。然后，在~/Desktop/20210110 目录下新建文件 instruction.js。使用 Instruction 模块时，可以通过 Objection 找到想要查看的地址：

```
root@VXIDr0ysue:~ # objection -g com.roysue.easysol explore
com.roysue.easysol on (google: 8.1.0) [usb] # memory list modules
com.roysue.easysol on (google: 8.1.0) [usb] # memory list exports libroysue.so
```

libroysue.so 导出的信息如图 2-13 所示。

com.roysue.easysol on (google: 8.1.0) [usb] # memory list exports libroysue.so		
Type	Name	Address
variable	r	0x7776beb580
function	to_bytes	0x7776beb0c
variable	k	0x7776beb480
function	to_int32	0x7776beb5c
function	md5	0x7776beaba4
function	Java_com_roysue_easysol_MainActivity_stringFromJNI	0x7776beb064

图 2-13 libroysue.so 导出的信息

以 stringFromJNI 函数为例编写如下代码：

```
setImmediate(function() {
    var stringFromJniaddr =
Module.findExportByName("libroysue.so","Java com roysue easysol MainActivity stringFrom
JNI")
    var ins = Instruction.parse(stringFromJniaddr);
    console.log("ins =>", ins.toString())
})
```

保存并运行，可以看到打印出的指令：

```
root@VXIDr0ysue:~/Desktop/20210110 # frida -UF -l instruction.js

ins => sub sp, sp, #0x130
[LGE Nexus 5X::easysol]->
```

修改程序，打印 10 条指令：

```
function dis(address, number) {
    for(var i = 0; i < number; i++) {
        var ins = Instruction.parse(address);
        console.log("address:" + address + "--dis:" + ins.toString());
        address = ins.next;
    }
}

setImmediate(function(){
    var stringFromJniaddr = Module.findExportByName("libroysue.so","Java_com_roysue_
easysol_MainActivity_stringFromJNI")
    dis(stringFromJniaddr, 10);
})
```

打印结果如图 2-14 所示。

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
ins=> str x28, [sp, #0x110]
ins=> str x28, [sp, #0x110] I
ins=> str x28, [sp, #0x110]
ins=> str x28, [sp, #0x110]
address:0x7c82d2d064--dis:sub sp, sp, #0x130
address:0x7c82d2d068--dis:str x28, [sp, #0x110]
address:0x7c82d2d06c--dis:stp x29, x30, [sp, #0x120]
address:0x7c82d2d070--dis:add x29, sp, #0x120
address:0x7c82d2d074--dis:mrs x8, tpidr_el0
address:0x7c82d2d078--dis:ldr x8, [x8, #0x28]
address:0x7c82d2d07c--dis:stur x8, [x29, #-0x18]
address:0x7c82d2d080--dis:str x0, [sp, #0x70]
address:0x7c82d2d084--dis:str x1, [sp, #0x68]
address:0x7c82d2d088--dis:mov w9, wzr

```

图 2-14 打印结果

结果与 LLDB 的调试结果相同，如图 2-15 所示。

```

Variables LLDB Memory View
(lldb) script exec("if lldb.debugger.GetCategory('libstdc++-v6').IsValid(): lldb.debugger.GetProcess().GetSelectedThread().Stop();")
(lldb) dis
librosue.so`Java_com_royosue_easysol.MainActivity_stringFromJSON:
0x7c82b35064 <+0>: sub    sp, sp, #0x130           ; =0x130
0x7c82b35068 <+4>: str    x28, [sp, #0x110]
0x7c82b3506c <+8>: stp    x29, x30, [sp, #0x120]
0x7c82b35070 <+12>: add    x29, sp, #0x120          ; =0x120
0x7c82b35074 <+16>: mrs    x8, TPIDR_EL0
0x7c82b35078 <+20>: ldr    x8, [x8, #0x28]
0x7c82b3507c <+24>: stur   x8, [x29, #-0x18]
0x7c82b35080 <+28>: str    x0, [sp, #0x70]
0x7c82b35084 <+32>: str    x1, [sp, #0x68]
0x7c82b35088 <+36>: mov    w9, wzr

```

图 2-15 LLDB 的调试结果

2.6 本章小结

本章主要介绍了在 Android Studio 中使用 Debugger 进行调试的方法，以及 LLDB 动态调试 NDK 程序的过程。通过一个简单的 MD5 案例，我们学习了交叉编译的过程，即如何将一个 C 语言程序移植到现有的 NDK 项目中。之后介绍了反汇编器 Capstone 和汇编器 Keystone，并演示了集成 Capstone 的 Frida Instruction 模块的基本用法。在后续的章节中，我们将更加深入地学习应用调试的技术并介绍 Unicorn CPU 模拟器的使用。