dump 法从入门到熟练（三）

一、引言

本篇文章是对上篇文章的回顾和解释。

二、拦截目标时机

处理的第一步就是拦截我们想要中段执行的时机点，在上文里，我们通过 Frida inline hook 实现这一需求。

```
function hookGenericNt3(){
    Java.perform(function() {
        var base_addr = Module.findBaseAddress("libtiger_tally.so");
        var real_addr = base_addr.add(0x25a9d);
        var obj = Java.use("java.lang.Object");
        var ByteString = Java.use("com.android.okhttp.okio.ByteString");
        Interceptor.attach(real_addr, {
            onEnter: function (args) {
                console.log("call GenericNt3")
                console.log(JSON.stringify(this.context))
                console.log("arg2:", args[2])
                console.log("arg3:", ByteString.of(Java.array('byte', Java.cas
            }
        });
    })
}
```

这一步看起来普普通通，但其实暗藏玄机。回想一下，在 Android Native 上有哪些拦截执行时机的办法？最常见的是以下四种方式。

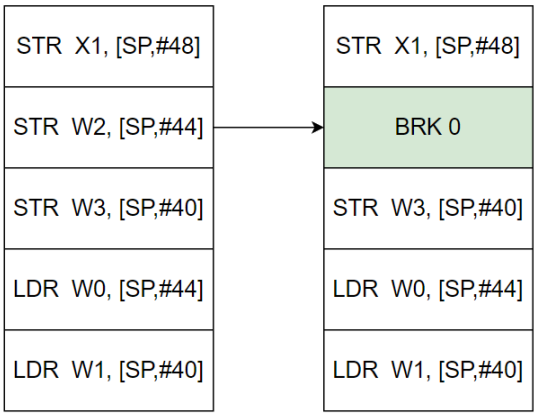
- 软件断点
- 硬件断点
- inline Hook
- plt/got Hook

plt/got Hook 只能处理有符号函数，我们的中段执行需要在任意地址做拦截的能力，所以它不符合要求。接下来讨论其余三种 Hook。

软件断点和 inline Hook 都是侵入式的处理方案，以下面的 ARM64 代码片段举例。

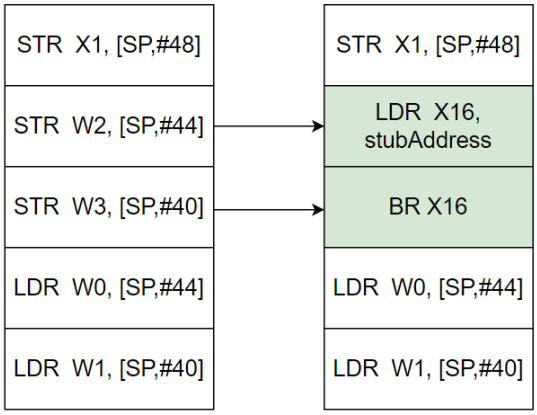
```
.text:00000000000009F0 E1 1B 00 F9                 STR             X1, [SP,#4
.text:00000000000009F4 E2 2F 00 B9                 STR             W2, [SP,#4
.text:00000000000009F8 E3 2B 00 B9                 STR             W3, [SP,#4
.text:00000000000009FC E0 2F 40 B9                 LDR             W0, [SP,#4
.text:0000000000000A00 E1 2B 40 B9                 LDR             W1, [SP,#4
.text:0000000000000A04 E8 07 00 F9                 STR             X8, [SP,#8
```

如果我们通过软件断点拦截 0x9F4 地址，那么 0x9F4 指向的这条指令会被修改为 ARM64 上的软件断点指令 `brk #imm`，比如 `brk 0`。



关于 ARM 架构上软件断点的详细介绍可以看我之前的《简述 IDA Debugger》<https://www.yuque.com/lilac-2hqvv/eighpm/hz0vuydtyxxnib1f?singleDoc# 《简述 IDA Debugger》> 这篇文章。

如果我们通过 inline hook 做拦截，比如 Frida Hook，那么影响的范围会扩大到数条指令，它需要跳到自定义的跳板函数，最后处理完毕再返回。



这两个方案都会修改 Hook 区域的代码，在短暂的处理后再恢复到原先指令。

硬件断点是更隐蔽的方案，它的原理也很简单——将需要拦截和中断的地址写到某个架构相关的特殊寄存器里，当实际执行到这个地址时，CPU 就会中断下来。

在实际场景里，调试器主要基于软件断点，Frida 等工具则基于 inline Hook。

为什么要谈这几种 Hook 对内存的修改呢？这涉及到 dump 的问题。

```
function dumpSo(name){
    var libSO = Process.getModuleByName(name);
    var file_path = savePath + libSO.base + "_" + libSO.base.add(libSO.size) -
    var file_handle = new File(file_path, "wb");
    if (file_handle && file_handle != null) {
        Memory.protect(ptr(libSO.base), libSO.size, 'rwx');
        var libso_buffer = ptr(libSO.base).readByteArray(libSO.size);
        file_handle.write(libso_buffer);
        file_handle.flush();
        file_handle.close();
        console.log("[dump]:", file_path);
    }
}
```

我们后续使用 Frida dump 目标 SO，因此 inline Hook 对目标地址处的修改，也一并被 dump 了下来，但事实上我们想要执行的是原指令，这是一个糟糕的问题。

在调试器上我们就不会遇到这种问题，在对某个位置下断点时，尽管这个地址处的实际内容变为了断点指令所对应的机器码。但是在调试器的汇编界面、内存界面，又或是在调试器里发起内存 dump，目标地址处的内容都始终是原字节。即调试器都做了专门的处理，让用户看到、得到的始终都是断点所在位置本来的内容。

为了避免使用 Frida dump 时遇到这个尴尬的问题，在后续 Unidbg 模拟执行的逻辑里，我将 dump 下来的从中断位置起始的 16 字节替换为正常 SO 对应位置的内容。

```
// patch SO
byte[] originCode = emulator.getBackend().mem_read(module.base + offset, 0x10)
emulator.getBackend().mem_write(moduleBase + offset, originCode);
```

三、暂停程序

拦截到目标时机点后，我们需要让它停下来，固定住此时的内存等环境，dump 可不是一瞬间能完成的事。回顾一下上文里我们的代码。

```
function hookGenericNt3(){
    Java.perform(function() {
        var base_addr = Module.findBaseAddress(targetSO);
        var real_addr = base_addr.add(0x25a9d);
        var obj = Java.use("java.lang.Object");
        var ByteString = Java.use("com.android.okhttp.okio.ByteString");
        var raise = new NativeFunction(Module.findExportByName("libc.so", "ra
        var SIGSTOP = 19
        Interceptor.attach(real_addr, {
            onEnter: function (args) {
                console.log("call GenericNt3")
                console.log("arg2:", args[2])
                console.log("arg3:", ByteString.of(Java.array('byte', Java.cas
                // regs
                console.log(JSON.stringify(this.context))
                // symbols
                dumpSymbol("libc.so")
                // target SO
                dumpSo(targetSO)
                // wait
                Thread.sleep(10)
                // stop process
                raise(SIGSTOP)
            }
        });
    })
}
```

上文通过 `raise(19)` 发出 SIGSTOP 信号，将整个进程挂起。这个处理办法其实不是特别好。

请读者回忆一下调试器断点，当断下的时候，整个进程似乎也被暂停了，但与此同时，我们又能在调试器里查看内存、寄存器或其他信息。但是在上面的 `raise(19)` 操作后，我们却没法通过 Frida 获取各种信息了。事实上，这种差异是因为 `raise(19)` 是将整个进程挂起，而调试器是将除了自身以外的所有线程挂起。

调试器都自带这样的处理机制，不管是 LLDB 还是 IDA Debugger，但 Frida 这类工具上不提供直接实现这件事的机制或 API，所以为了简单起见，我们仅 dump 了目标 SO 以及一个 symbols.log，然后就匆匆将整个进程挂起，此时 Frida 没法操纵进程了，接下来要使用别的工具去做完整内存的 dump。

有的朋友可能会意识到，调试器去处理整个拦截+中断+dump，似乎比用 Frida 方便很多，不用绕这些弯弯。其实确实如此，但调试器操作起来太拧巴了，很多人都不喜欢用调试器，包括我自己。

四、保存上下文

我们之前聊过，上下文在理论上包括内存、寄存器、进程信息、文件描述符等一堆东西，但在一般的情况里，所需仅仅是少量的内存罢了，甚至寄存器都不重要。

在 dump 内存这件事上，我们可以将整个进程的潜在内存一次性都 dump 下来，然后在本地慢慢用；也可以后续缺什么再单独 dump 出来，反正进程被挂起来了，不用担心内存发生改变。在上篇文章里我们选择的后一种方式，这出于两个原因。

一是如果采用第一个方案，把可能用到的内存都 dump 出来，这会很慢很耗时，要半小时甚至更久。为了避免这个问题，相关方案都会做很多优化，比如设置黑白名单（即只 dump 最可能用到的

内存区块，或不 dump 很少用到的内存区块），比如对内存做一些压缩等等。这些处理逻辑多少有些复杂，我不喜欢。用多少取多少的策略，就不用担心整体 dump 太耗时这个问题。

二是在 dump 时采用的是 adb shell 的 dd 命令，先 dump 到 sdcard，再 pull 到电脑上。

```
def dumpRange(start, end):
    dumpName = hex(start).replace("0x", "")+"_"+hex(end).replace("0x", "")+".b
    filePath = f"/sdcard/{pid}/{dumpName}"
    command = f"adb shell su -c dd if=/proc/{pid}/mem of={filePath} skip={start
    os.system(command)
    os.system(f'adb pull /sdcard/{pid}/{dumpName} {pid}')
```

dd 是 Linux 上的一个常见命令，用于读取和转储数据，不熟悉的朋友可以看《Linux dd 命令》<https://www.runoob.com/linux/linux-comm-dd.html> 。有它的话，我们就不需要引入第三方工具去做内存的 dump，比如 MemDumper <https://github.com/kp7742/MemDumper> 。但 dd 的 dump 速度相比较其他工具真挺慢，经不住我们大批量 dump。

五、恢复到 Unidbg 环境

来审视一下我们最初的 Unidbg 代码

```java
package com.wtoken;

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Emulator;
import com.github.unidbg.Module;
import com.github.unidbg.arm.backend.Unicorn2Factory;
import com.github.unidbg.file.FileResult;
import com.github.unidbg.file.IOResolver;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
import com.github.unidbg.linux.android.dvm.AbstractJni;
import com.github.unidbg.linux.android.dvm.DalvikModule;
import com.github.unidbg.linux.android.dvm.VM;
import com.github.unidbg.memory.Memory;

import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;

public class TigerDump extends AbstractJni implements IOResolver {
    @Override
    public FileResult resolve(Emulator emulator, String pathname, int oflag
        System.out.println("File open:"+pathname);
        return null;
    }

    private final AndroidEmulator emulator;
    private final VM vm;
    private Module module;
    public final String dumpPath = "unidbg-android/src/test/resources/wtoke
    public long moduleBase = 0xb99d5000L;
    public long offset = 0x25a9C;


    TigerDump() {
        emulator = AndroidEmulatorBuilder.for32Bit()
                .addBackendFactory(new Unicorn2Factory(false))
                .setProcessName("com.nike.omega")
                .build();
        Memory memory = emulator.getMemory();
        memory.setLibraryResolver(new AndroidResolver(23));
        emulator.getSyscallHandler().setEnableThreadDispatcher(true);
        emulator.getSyscallHandler().addIOResolver(this);
        emulator.getBackend().registerEmuCountHook(100000); // 设置执行多少条
        vm = emulator.createDalvikVM();
        vm.setJni(this);
        vm.setVerbose(true);
        DalvikModule dm = vm.loadLibrary(new File("unidbg-android/src/test/
        module = dm.getModule();
        initContext();
        patch();
    }

    public void patch(){
        // patch SO
        byte[] originCode = emulator.getBackend().mem_read(module.base + of
        emulator.getBackend().mem_write(moduleBase + offset, originCode);

    }

    public void initContext(){
        File file = new File(dumpPath);
        File[] fs = file.listFiles();
        assert fs != null;
        for(File f:fs) {
            if (!f.isDirectory()){
                String name = f.getName();
                mapAndWrite(name);
            }
        }
    }

    public byte[] restoreBinary(String name){
        byte[] bytes = new byte[0];
        try {
```

```
            bytes = Files.readAllBytes(Paths.get(dumpPath+name));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return bytes;
    }

    public void mapAndWrite(String name){
        byte[] data = restoreBinary(name);
        String[] base_and_end = name.replaceAll(".bin", "").replaceAll("0x"
        long start = Long.parseLong(base_and_end[0], 16);
        long end = Long.parseLong(base_and_end[1], 16);
        emulator.getBackend().mem_map(start, end - start, 7);
        emulator.getBackend().mem_write(start, data);
    }

    public static void main(String[] args) {
        TigerDump tigerDump = new TigerDump();
    }
}
```

initContext 函数会遍历 dumpPath 文件夹，将其中所有的文件加载到 Unidbg 虚拟内存里。

接下来看调用函数

```
private void callTarget() {
    List<Object> list = new ArrayList<>(10);
    list.add(vm.getJNIEnv());
    DvmObject<?> thiz = vm.resolveClass("com/aliyun/TigerTally/TigerTallyAPI")
    list.add(vm.addLocalObject(thiz));
    list.add(1);
    ByteArray barr = new ByteArray(vm, "da965a94-97da-4730-b7d3-3d16e4061489"
    list.add(vm.addLocalObject(barr));
    //  开始模拟执行
    Number result = Module.emulateFunction(emulator, moduleBase + offset + 1,
    String ret = vm.getObject(result.intValue()).getValue().toString();
    System.out.println("result:"+ret);
}
```

这么做调用很不优雅，读者可能会怀念 Unidbg 封装的调用函数，像下面这样。

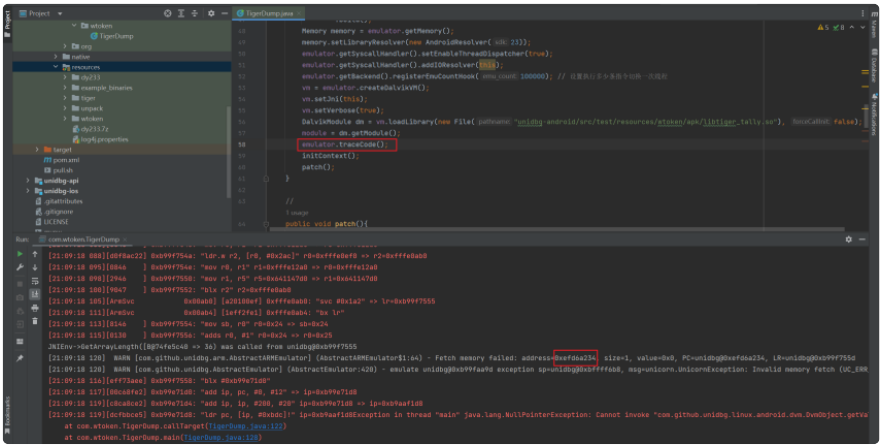
```
public void callTarget(){
    DvmObject<?> dvmObject = TigerTallyAPI.callStaticJniMethodObject(emulator,
    String wtoken = dvmObject.getValue().toString();
    System.out.println("result:"+wtoken);
}
```

这种调用方式依赖于函数和地址的绑定关系，通过 dump 构建的运行环境里没有这些信息，所以只能用稍微复杂一些的 `emulateFunction` API。

在后续的处理中，主要是处理两类内存问题

1 是内存读写，这是数据类问题，我们需要 dump 对应的内存放到 dumpPath 里。

2 是内存访问，即 fetch memory，执行流试图跳转到这个地址上，这是代码类问题。



理论上第二类问题，同样可以 dump 内存然后放 Unidbg 里执行，但事实上我们没有这么做，这其实也有考量。

这些代码是你 Android 测试机上相关逻辑，比如 libc.so、libart.so 等等，Unidbg 没有办法很好的处理这些逻辑，因为它们的版本很高。

举个例子，我们调用 gettimeofday 函数，在正常的 Unidbg 环境里，它会走到 `src/main/resources/android/sdk23/lib(64)/libc.so` 里，这其实就是 Android 6 上的标准库，因为版本比较老，逻辑也相对简单，所以Unidbg 处理它不会有什么问题。但我们的测试机版本一般比较高，Unidbg 在处理高版本的 libc.so 时可能会遇到各种各样奇怪的问题，很多很痛苦。

因此我们做了移花接木，当访问外部函数时，就通过 Hook 把控制流转到 Unidbg 可以稳定且正常处理的 Android 6 的基本库里，反正函数的语义和功能没差别。

在 Frida hook 的那一步里，dump 了 libc.so 的导出函数表，是因为最频繁使用的外部库就是 libc。

我们的 Unidbg 代码里加载了原 SO，也不是什么无用的代码，既是因为第二节末尾提到的原因，也是为了让 Unidbg 把目标 SO 的依赖库加载到内存里，方便我们 Hook 转接上去。

六、尾声

对基于调试器的 dump 和处理方案感兴趣的可以看脸哥的《dump内存与模拟执行》<https://blog.seeflower.dev/archives/165/> 系列文章。下篇我们基于 Unidbg 的这个 dump 环境做一定的算法分析。