SO Hook 技术汇总

ThomasKing --2014.12.16

HOOK 早已不是什么新鲜名词，在 windows 平台 HOOK 技术已经用得很烂，以至于 win7 x64 及其之后，MS 还搞了 PatchGuard 来防止 HOOK。当然，在 Android 平台的 HOOK 的基本原理和 windows 平台是相同的。但具体到实现，还是有许多差异，特别是由于 ARM 指令集的缘故，inline hook 的实现比较繁琐。为了 SO hook 体系体系的完整性，下面将依次介绍 ELF 导入表 HOOK(即 GOT 表 HOOK)，inline hook(ARM 32-bit 平台)以及 Android 平台基于 linker 特性的导出表 HOOK。

一、导入表(GOT 表 HOOK)

熟悉 ELF 结构的读者都知道，SO 引用外部函数的时候，在编译时会将外部函数的地址以 Stub 的形式存放在.GOT 表中，加载时 linker 再进行重定位，即将真实的外部函数写到此 stub 中。HOOK 的思路就是：替换.GOT 表中的外部函数地址。具体流程：

1. 注入进程
2. 可能有读者想到马上就是读取并解析 SO 的结构，找到外部函数对应在 GOT 表中的存放地址。在 http://bbs.pediy.com/showthread.php?t=194053 中已经讨论 dlopen 返回的是 solist，已经包含 SO 信息。(直接通过 SOLIST 实现替换 HOOK，代码量就很小了)

导入表 HOOK 的实现是最简单的了，但也不难看出，导入表的 HOOK 功能是很有限的。例举两点：1. 导入表 HOOK 对进程通过 dlopen 动态获得并调用外部符号是无效的。2. 导入表 HOOK 只能影响被注入进程。

二、ARM inline hook – 32bit

Inline hook 的基本原理网上已经很多，这里就不赘述了。Inline hook 的基本流程如图 1 所示：

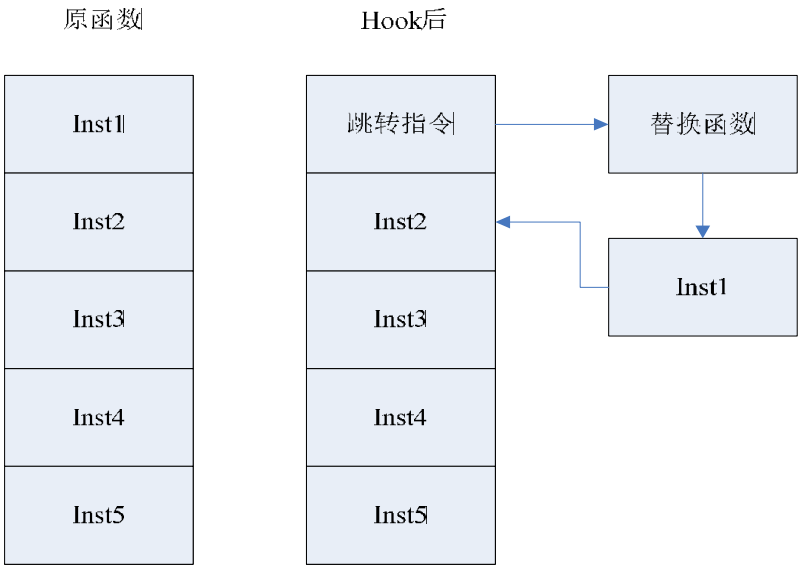

图 1

在 Windows 平台，无条件跳转指令(jmp 地址)一共需要 5 个字节。故实现时，构造 jmp 指令跳转到替换函数。而在基于 ARM 的 Android/IOS 平台，由于存在 ARM 指令集和 Thumb(嵌入 Thumb2)指令集，实现时必须谨慎考虑跳转指令的构造和 Inst1 的修正。下面分别讨论这种指令集的实现，涉及 opcode，故需读者自行下载 ARM 32bit 手册(Read the fucking ARM Architecture Reference Manual)。

2.1 ARM 指令集

2.1.1 跳转指令构造

借鉴 windows 平台实现的思路，在 ARM 平台也有相应的无条件指令 B。稍微了解 ARM 指令集的读者知道，ARM 指令集的指令都是 4 字节，故也是 4 字节对齐的。那么问题来了(不是挖掘机…)，32bit 地址占 4 字节，B 指令跳转范围是受限的，如何跳转任意地址？其实，ELF 中已经存在类似例子，即.plt 结构，如图 2 所示：

```
.plt:00000C88
.plt:00000C88 __cxa_atexit                            ; CODE XREF: .text:00000D18↓j
.plt:00000C88                 ADR     R12, 0xC90
.plt:00000C8C                 ADD     R12, R12, #0x3000
.plt:00000C90                 LDR     PC, [R12,#(__cxa_atexit_ptr - 0x3C90)]! ; __imp___cxa_atexit
.plt:00000C90 ; End of function __cxa_atexit
```

图 2

第三条指令，LDR PC, [偏移地址](此地址指向 GOT 表)。借鉴此思路，跳转指令的构造即为：Ldr PC, [下一条指令的偏移]，将 HOOK 函数地址写到下一条指令。如图 3、4 所示：

```
ARM.so:80B00C24                 Java_com_example_substratetest_MainActivity_ARMCALL
ARM.so:80B00C24 38 40 2D E9 STMFD           SP!, {R3-R5,LR}
ARM.so:80B00C28 00 40 A0 E1 MOV             R4, R0
ARM.so:80B00C2C 01 50 A0 E1 MOV             R5, R1
ARM.so:80B00C30 95 04 04 E0 MUL             R4, R5, R4
ARM.so:80B00C34 F6 FF FF EB BL              unk_80B00C14
ARM.so:80B00C38 04 0B A0 E1 MOV             R0, R4,LSL#22
ARM.so:80B00C3C 20 0B A0 E1 MOV             R0, R0,LSR#22
ARM.so:80B00C40 38 80 BD E8 LDMFD           SP!, {R3-R5,PC}
ARM.so:80B00C40                 ; End of function Java_com_example_substratetest_Mai
```

图 3

```
ARM.so:80B00C24                 Java_com_example_substratetest_MainActivity_ARMCALL
ARM.so:80B00C24 04 F0 1F E5 LDR             PC, loc_80B00C28
ARM.so:80B00C28
ARM.so:80B00C28                 loc_80B00C28
ARM.so:80B00C28 24 0F E0 80 RSCHI           R0, R0, R4,LSR#30
ARM.so:80B00C2C 01 50 A0 E1 MOV             R5, R1
ARM.so:80B00C30 95 04 04 E0 MUL             R4, R5, R4
ARM.so:80B00C34 F6 FF FF EB BL              unk_80B00C14
ARM.so:80B00C38 04 0B A0 E1 MOV             R0, R4,LSL#22
ARM.so:80B00C3C 20 0B A0 E1 MOV             R0, R0,LSR#22
ARM.so:80B00C40 38 80 BD E8 LDMFD           SP!, {R3-R5,PC}
ARM.so:80B00C40                 ; End of function Java_com_example_substratetest_Ma
```

图 4

注: 图 4 第二条不是指令，是 HOOK 函数地址：0x80e00f24。

2.1.2 Inst1 指令的修正

不管是 ARM 指令还是 Thumb 指令，都存在相对 PC 的指令，手册中称为：literal。另外还有一种隐式的 literal(额，我这么叫。。。)即 B 系列指令。由于替换之后，指令执行时，PC 的值已经改变，原指令中相对 PC 的偏移值就必须修正。具体来说就是：

1. 解析指令，判定是否为 literal
2. 如果是，则解析基于 PC 的 offset
3. 1)若是 LDR 系列指令，将原 offset 的值复制到修正指令的后面，修正 LDR 指令的 offset；

2)若是 ADD literal 指令，则还需计算原 PC 的值，生成如下指令：

push Rx,

LDR Rx, #offset

Add Rd, Rx(替换 PC), Rn or Rn, #offset(寄存器和寄存器移位，详见手册)

Pop Rx

Rx 选择为非 Rd, Rn 即可

3)若是 B 系列指令，则转换为 LDR 指令。即计算出 B 系列指令跳转到的绝对地址，再构造 LDR:

LDR Rx, #offset

有了跳转指令替换和 Inst1 指令的修正思路，剩下就是扒手册了。那是不是就 OK 了？Inst1 可没有返回 Inst2 的地址。为了解决 offset 的访问问题，返回指令仍然采用 LDR PC,#offset 来返回 Inst2。另外，注意构造存放 offset 的位置，因为 LDR(literal)寻址范围是有限的。修正后的如图 5、6 所示：

```
ARM_PC.so:80A00C9C                TK_puts                                 ; CODE XREF: Jav
ARM_PC.so:80A00C9C 04 00 9F E5 LDR             R0, =(aArm_pc_call - 0x80A00CA8)
ARM_PC.so:80A00CA0 00 00 8F E0 ADD             R0, PC, R0              ; "ARM_PC_CALL"
ARM_PC.so:80A00CA4 DD FF FF EA B               sub_80A00C20
ARM_PC.so:80A00CA4                ; End of function TK_puts
ARM_PC.so:80A00CA4
```

图 5 原函数

```
debug084:4664E000                CODE32
debug084:4664E000 F8 03 9F E5 LDR              R0, =0x80A00CA4
debug084:4664E004 04 00 90 E5 LDR              R0, [R0,#4]
debug084:4664E008 04 10 2D E5 STR              R1, [SP,#-4]!
debug084:4664E00C F0 13 9F E5 LDR              R1, =off_80A00CA8
debug084:4664E010 00 00 81 E0 ADD              R0, R1, R0
debug084:4664E014 04 10 9D E4 LDR              R1, [SP],#4
debug084:4664E018 E8 F3 9F E5 LDR              PC, =loc_80A00CA4
debug084:4664E018                ; ---------------------------------------
```

图 6 修正原 LDR、ADD 地址(看粗红线)

这里需要稍微解释下，第一条指令修正时，我采用的方式是：解析原 PC 指令的值，将原 PC 的值 LDR 到 Rx 中，然后通过寄存器间值得到原偏移的值。这里可以减少内存占用。但 Thumb 模式由于指令集的缘故，是不能采用这种方式。第二个粗红线框中的指令和上述讨论相同，由于 PUSH、POP 其实是 STR SP， LDR SP 的特殊形式，IDA 统一按 STR 识别了。其中 Rx 为 R1。最后一条即为返回指令，通过 LDR PC，#offset 实现。(说个题外话，Substrate inline hook ARM 指令集时，对 B 系列和 ADR 指令修正是不对的，一旦起始 8 字节这两种指令，运行时会出错的)

2.2 Thumb 指令集

Thumb 指令集相对 ARM 指令集的指令数目要少很多，但也由于其指令的限制，2 字节对齐和与 Thumb-2 的混用，inline hook 实现起来是颇为麻烦，需要相当小心。在讨论之前，先说说 Thumb-2 指令集。

2.2.1 Thumb-2 指令集

T2 指令集与 T1 指令集是可以混用的，都是 2 字节对齐，但 T2 指令占 4 个字节。另外，T2 指令虽然占 4 个字节，但其是按 2 个字节构成的。如图下 7 所示：

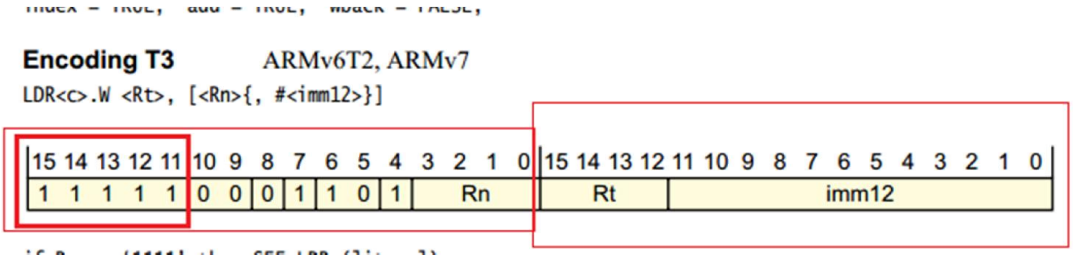

图 7 T2 指令集

图中红色部分用于区分 T2 和 ARM 指令集。通过指令的前 2 个字节的高 5 位(11101, 11110, 11111)这三种组合来标识其为 T2 指令(SO 破解修改指令时需要注意区分是 T2 还是 ARM)。

2.2.2 跳转指令构造

借鉴 ARM 指令集的思路，想利用 Thumb 指令中的 LDR 来构造跳转。遗憾的是，Thumb 模式并没有能表跳转任意地址的指令，只能切换到 ARM 状态，再进行跳转。为了尽可能的减少替换的指令数，状态切换应尽快，这里采用 BX PC。另外，这里还需注意 2 点：1、ARM 指令是 4 字节对齐，BX PC 状态切换时，必须保证跳转到的地址为 4 字节对齐。2、由于 T2 指令占 4 字节，如果被替换指令的最后一条为 T2 指令，且 T2 指令的前 2 字节处于被替换指令中，而后 2 字节未处于其中时，也是需要将后 2 字节归入被替换的指令中作为一个整体。有了上面的分析，指令的替换流程如下：

1) 判定其实地址是否 4 字节对齐，如果不为 4 字节对齐，则 BX PC 之前构造 NOP 指令
2) 由于预取 2 条指令，BX PC 之后 2 字节填充 NOP
3) 构造 ARM LDR 指令，占 4 字节。其后 4 字节存放跳转绝对地址
4) 判定被替换指令最后 2 字节是否为 T2 指令的前 2 两字节，如果是，则还需把之后 2 字节加入替换指令中，后 2 字节用 Thumb NOP 填充。

具体如图 8、9 所示：

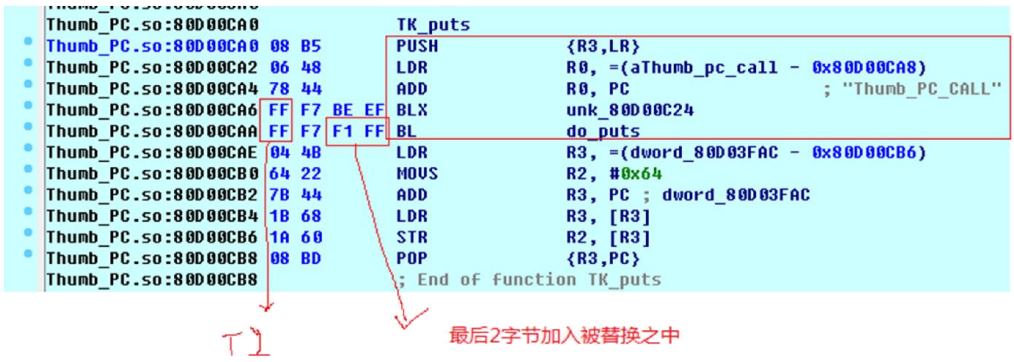

图 8 Thumb 替换之前

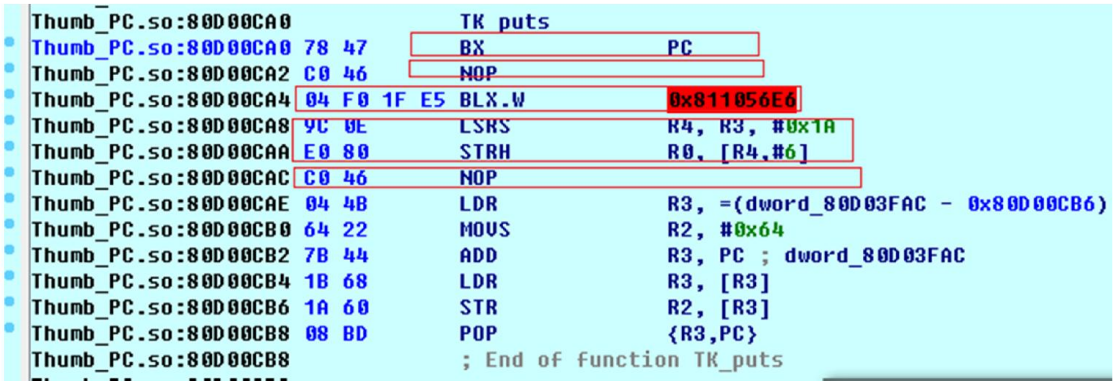

图 9 Thumb 替换之后

解释下图 9 中的指令：

1. 由于起始 0x80d00ca0 已 4 字节对齐，故不需要补 THUMB NOP，直接 BX PC
2. 由于预取，补 THUMB NOP
3. 由于 BX PC 已经切换了状态，此时为 ARM 状态，IDA 识别错误了。前面已经说明，T2 的高 5 为必须是(11101, 11110, 11111)之一，但其 0xe51ff004 高 5 位(11100)，也可以说明不是 T2 指令。
4. 第 4 个红框中为跳转地址：0x80e00e9c
5. 此时已经占用了 12 字节，即图 8 中 0x80d00caa 前 2 字节。而这条指令为 T2 指令，故也需要把后 2 字节归入 Inst1 中。故最后补了 2 字节 THUMB NOP。

2.2.3 Inst1 指令的修正

Inst1 指令的修正和 ARM 的雷同，只是使用的是 Thumb 指令的，限制颇多。值得注意的是，Thumb 的 literal 指令的中的 PC 是 Align(PC)，计算时需要小心。另外，Inst1 末尾构造的返回指令也是通过 ARM LDR 实现。故也需要注意地址对齐。具体如图 10 所示：

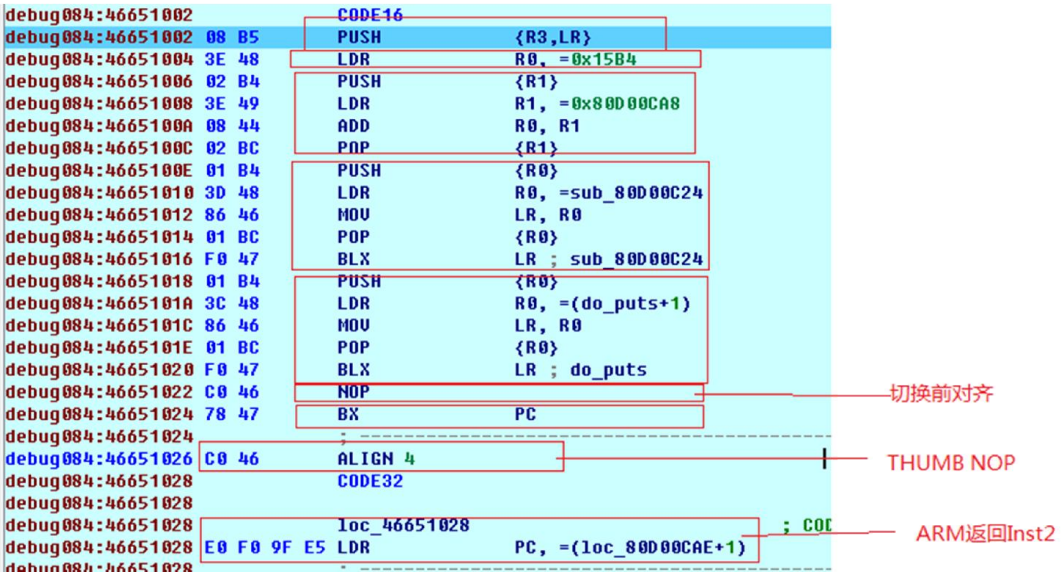

图 10 修正 Inst1

当然，ARM Inline 的实现还需要注意一些细枝末节的地方，这些都可以在 ARM 手册上找到，限于篇幅，就不一一列出了。Inline hook 的优势之处很多，这里例举两点劣势：1. 实现较为复杂，与硬件平台相关。2. ARM 的 inline 替换字节为 8-14 字节，远远高于 x86 平台的 5 字节。当函数比较简短时，是无法发挥作用的。

三、基于 Android linker 的导出表 HOOK

熟悉 ELF 格式的读者应该知道，ELF 并没有类似 PE 格式的显式导出表。但 ELF 文件的 symtab 表说明了符号是导入符号还是导出符号，这点可以在 linker.c 的源码中可以看到。要实现导出表 HOOK，先来分析下 linker 是如何帮助 SO 获得外部符号的地址的。

在 linker 源码中可以看到，linker 将 SO 加载到内存之后，最后阶段最主要的是对符号进行重定位。在重定位过程中，如果发现符号为外部符号，就会去解析 NEEDED SO，获取外部符号的地址。具体一点来说，就是通过 NEEDED SO，根据外部符号的名字，找到对应的 Elf32_Sym，从这个 Elf32_Sym 中的 st_value 字段得到函数的虚地址。那么修改 NEEDED SO 中的这个符号 st_value 字段，即可实现导出表 HOOK。

具体实现流程：以 libc.so 中的 unlink 函数为例

1. 注入 zygote 进程
2. dlopen libc.so，找到 unlink 符号
3. 解析此符号，得到其 st_value 地址
4. 修改此地址的值为：NewFunc – BaseAddr(libc.so 加载的基地址)。因为符号的绝对地址是通过 base + st_value 计算，即 st_value 保存的本身是偏移地址

从上述流程可以看出，SO 的导出表的 HOOK 实现复杂度和调入表差不多，但确能起到很好的效果。当然，拦截效果不如 inline 这种方式。由于 inline 受到函数字节书的轻微限制，导出表 HOOK 也可视为一种对 Inline 的补充 HOOK 方式。不难看出，导出表 HOOK 只能 HOOK 导出的符号。

上述三种 HOOK 技术，我已经写好上传到 github。如发现任何 BUG，请发送 Email: ThomasKingNew@hotmail.com 告知我，非常感谢。
https://github.com/ThomasKing2014/ELF-ARM-HOOK-Library

四、参考文献

http://bbs.pediy.com/showthread.php?t=194053
http://bbs.pediy.com/showthread.php?t=180918&highlight=inline+hook
ARM Architecture Reference Manual
Linker.c 源码