用高级语言实现指令自修改

窦路
alexdou@e28.com

[摘要] 所谓指令自修改即程序在运行时产生后面即将执行的指令的技术。自修改指令在诸如软件加密，程序装载等方面均有应用。本文举例说明如何用高级语言实现针对 C++程序的自修改指令。

1. 基本方法

在输入法所依赖的开发库中，中文成语的封装库对其所支持语言采用了缺省为简体中文的硬编码方式，因此无法通过 C++方法调用来设置期望支持的语言，例如繁体中文。而相关的负责工程师暂时没有精力来增加这个设置语言的接口，于是我们考虑是否可以通过运行时修改指令的方式来实现在现有二进制库的基础上动态设置所期望支持的语言。

通过分析源代码及反汇编代码发现

```
                        ; CODE XREF: _ZN18YLPhraseCandidates10GetPhrasesEPt+20↑j
BL      _ZN18YLPhraseCandidates17ClearResultBufferEv ; PIC mode

MOV     R4, R6
ADD     R1, R5, #0x22C
STRH    R4, [R1]
LDR     R0, [R5,#0x228] ; <suspicious>
MOV     LR, #0x100 ; <suspicious>
ADD     R2, R5, #0x214
MOV     R1, R6
MOV     R3, R7
STR     LR, [SP,#0x1C+var_1C]
STR     R4, [SP,#0x1C+var_18]
BL      PPFRAS_getword  ; PIC mode
```

在 ***YLPhraseCandidates::GetPhrases()*** 方法中，***mov*** 指令来设置了语言 ID。

通过查阅 ARM 指令格式和分析二进制库的十六进制数据了解到，***mov*** 指令的最低 8 位为立即数，本例中，立即数为语言的 ID。也就是说，只要能设置该指令的低 8 位，就能设置所期望的语言 ID。

指令在内存中的地址可以通过计算其与函数首地址的偏移量来确定，而函数地址可以通过其函数名作为符号来获取。本例中，指令偏移量为 ***0x3a0-0x370***。只要知道了指令的地址，就可以通过简单地赋值操作来修改它。

基于此，动态修改指令的流程可以用如下伪代码表示：

```
void dynamicInstruction() {
```

```
    methodAddress     = getMethodAddress(methodName);
    InstructionAddress = getInstructionAddress(methodAddress, offset);
    changeInstruction(InstructionAddress);
}
```

2. 获得 C++方法地址和指令地址

正如上面提到的，在计算指令地址，必须先获得函数地址，本例中，函数为 C++的方法。通过使用指向 C++方法的指针，我们可以得到该方法的首地址；另外，由于 C++编译器最终产生的了 C 风格的函数符号，因此使用一般的函数指针也能获得方法地址，只不过这种情况下需要知道编译后的 C++方法的符号名。下面分别介绍这两种方法。

2.1 使用 C++方法指针

C++方法指针的用法与 C 函数指针稍有不同，***C++ Primer*** 一书中进行了详细介绍。

在本例中，通过方法指针获得方法地址的代码如下

```
    // Function pointer points to method to change
    int (YLPhraseCandidates::*methodPointer)(unsigned short*) = NULL;
    // Get method address
    methodPointer = &YLPhraseCandidates::GetPhrases;
    // Seek target instruction by method address
    instruction    =    (unsigned    char*)(_phraseEngine->*methodPointer)    +
Instructionoffset;
```

在上面代码中，首先声明一个方法指针，然后将该指针与指定的方法关联，最后通过计算指令与方法首地址的偏移量获得指令地址。

需要注意的是，使用方法指针在获得方法地址时与 C 语言函数指针颇有些不同，因为方法指针实际上被 g++编译器作为结构体来处理。

使用 C++方法指针的好处主要是不用知道额外的其他信息，处理起来比较“干净”，但是方法指针用起来没有函数指针方便。

2.2 使用 C 风格的函数符号

使用 C 语言的函数指针也能获得 C++方法地址，这是因为通常情况下，C++方法最后都被编译器处理成了 C 函数的形式，只是名字符号和参数形式会有些改变，但这并不影响获得其地址。

通过反汇编，我们发现 ***YLPhraseCandidates::GetPhrases()*** 方法的最终符号是 ***_ZN18YLPhraseCandidates10GetPhrasesEPt***，为了能使用该符号，我们必须先将此符号声明为 ***extern "C"***。然后再使用一个指向该函数的指针，即可得到该符号的地址。

请看如下代码

```
// declare symbol of method to be update in run-time
extern "C" {

//#ifndef EXT_RELEASE
    // extern function symbol, we will change an instruction in this method later.
    int _ZN18YLPhraseCandidates10GetPhrasesEPt(unsigned short*);
//#endif //EXT_RELEASE

}

    // Function pointer points to method to change
    int (*methodPointer)(unsigned short*)       = NULL;
    // Get method pointer
    methodPointer = _ZN18YLPhraseCandidates10GetPhrasesEPt;
    // Seek target instruction by method address
    instruction = (unsigned char*)(methodPointer) + Instructionoffset;
```

C 语言的函数指针与 C++方法指针相比，使用起来更直接，更明了，更符合“指针”的思维。

3. 修改内存访问权限

在缺省情况下，代码段所对应的内存页的访问权限是可读，可执行，任何的写操作都会对应产生访问违例。因此，在修改指令前，必须先将指令所在内存页的访问权限设置为可写。这个任务可以由 ***mprotect()*** 函数来完成，通过使用该函数，应用程序员可以修改指定内存页的访问权限。

不过需要特别注意的是，***mprotect()*** 可能会失败，因此必须检查其返回值，只有当权限修改成功的情况下才能修改指令，否则会导致程序崩溃。

另外，***mprotect()*** 可接受的地址必须是按页对齐的，因此要把指令和方法地址转换为相应的页对齐地址。

```
// align the instruction address to page size
    pagePointer = instruction;
    pagePointer = (unsigned char*)(((int)pagePointer + PAGESIZE-1) &
~(PAGESIZE-1));
    pagePointer = pagePointer - PAGESIZE;
```

```
    // in default case, text region is read-only, so we need to change the page
to writable
    int mprotectRes = mprotect(pagePointer, PAGESIZE, PROT_READ | PROT_WRITE |
PROT_EXEC);
```

上面代码中，首先计算页对齐地址，然后修改了相应页面的访问权限，如果返回值为 0，则说明成功。只有这种情况下我们才能修改代码段的数据。

4. 修改指令

当确定了指令地址并且将该地址设置为可写以后，修改指令的工作仅仅剩下是一个很简单的赋值了。

需要特别注意的是，在修改指令前，必须先确认要修改的指令确实就是目标指令，因为如果二进制库更新以后，指令偏移量可能发生变化，如果不加判断地直接修改，其结果多半就是崩溃。

另外，在判断目标指令时还需要了解的是，包含地址信息的指令通常不适合作为判断依据，因为地址可能会在重定位时或者 ***prelink*** 时被修改。

本例中，首先对目标指令以及其后续指令进行了确认，确认无误的情况下，再修改目标指令。

```
        // We must check target instruction carefully to avoid any crash
        // If seeked instruction is target, change language ID in the instruction.
The first byte of the
        // instruction is language id.
        if((*(instruction + 1) == targetInstruction[1]) && (*(instruction + 2)
== targetInstruction[2])
                && (*(instruction + 3) == targetInstruction[3])) {

            //Check next instruction, length of ARM instruction is 4 bytes
            nextInstructionPointer = instruction + 4;
            if((*(nextInstructionPointer + 0) == nextInstruction[0]) &&
(*(nextInstructionPointer + 1) == nextInstruction[1])
                && (*(nextInstructionPointer + 2) == nextInstruction[2]) &&
(*(nextInstructionPointer + 3) == nextInstruction[3])) {

                // Both target instruction and next instruction is OK, update
target instruction here
                // Change the instruction by language
                if(region == IDictionary::CHINESE_MAINLAND) {
```

```
                *instruction = PENPOWER_SIMPLIFIED;
            }
            else {
                *instruction = PENPOWER_TRADITIONAL;
            }
        }
    }
```

5. 同步 Cache 与主存

至此，从编程语言的视角来看，已经完成了修改指令的全部工作，但是事实上还有一个更底层的细节需要关注，就是 Cache 与主存的同步。

由于 Cache 的存在，修改很可能没有立即被反应到主存，尤其是在 ARM9 这类哈佛体系结构的处理器中，指令和数据使用不同的 Cache，情况就更加复杂，因此必须确保所修改的指令在执行前已经被正确同步到主存，而不是仅仅存放在数据 Cache 中。

对于 ARM 来说，可以使用协处理器指令来同步主存与 Cache，但这意味着需要在高级语言中嵌入汇编代码，而由此引起的另一个更大的麻烦是，不同的 ARM 核的 Cache 清理指令略有不同，这使得汇编级代码不利于移植。

不过幸运的是，我们发现了一个 ***mprotect()*** 的有益的副作用，即清理 Cache，使其将修改后的脏数据回写主存

调用 ***mprotect()*** 时，如果指定内存页将要设置的权限与当前权限不同，则会触发 Cache 与主存的同步。与使用协处理器指令相比，这种方法不需要嵌入汇编，因而也不需要为不同的 ARM 核编写不同的汇编代码。

6. 结束语

本例中，通过运行时修改指令，实现了在现有二进制库的基础上查询简体或者繁体中文词语的功能。事实上，在其他平台上，动态指令修改早已被广泛地使用于例如防火墙，病毒，蠕虫，加密等诸多应用中。不过，仍然需要强调的是，动态修改指令会产生了潜在风险，如果在不正确的地址产生了不正确的指令，则会导致无效指令，访问违例等崩溃，因此需要特别谨慎。