前言

Native逆向大概是每个逆向人都敬畏又心存挑战的存在，但是又不知道从何下手，得益于汇编的繁琐，简单的高级语言在反编译成汇编指令之后分析起来复杂膨胀了无数倍，再加上对IDA Pro的一无所知，反编译阅读就是安卓ARM逆向的第一道难题，因此决定一步一步从《IDA Pro权威指南》开始积累知识点和学习经验，相信很多朋友也有一样的想法，但又苦于完整的阅读一本书，而且基本网上也没有现成的读书笔记可以快速索引和预览，因此决定将自己每天的阅读收获记录下来，整理其中关键的知识点，并且结合自己的收获心得，以每天一个小技巧的粒度方便大家快速学习，并且如果对其中感兴趣的知识点可以单独更细致的深入学习研究。

前五章基本都是基础知识，IDA如何打开使用等，无需记录，因此直接从第六章开始，聚焦干货和不会的或者有意义的知识点。

IDA中的符号和符号生成规则

- IDA在反汇编时为了便于区分大量的程序位置，会自动分配符号名称，例如常见的loc_xxxx、var_x、sub_xxx代表代码块、局部变量、函数等，每个符号都可以双击跳转到对应地址。
- 局部变量名称以var_为前缀，后面跟一个表示变量与被保存的帧指针之间距离（以字节为单位）的十六进制后缀。
- 函数参数名则以arg_为前缀，后面跟一个表示其与最顶端的参数之间的相对距离的十六进制后缀。
- IDA只会为那些在函数中直接引用的栈变量自动生成名称。

栈帧

栈帧是一个程序运行时内存布局的概念，通常指每个函数调用时所创建的用于记录返回地址、参数信息等的数据结构，栈帧是在程序的运行时栈中所分配的内存块。

栈帧存在的意义在于：

- 一，函数的调用方可能希望以参数（实参）的方式向该函数传递信息，这些参数需要存储到某个位置使得函数能够找到它们的位置。
- 二，在执行任务的过程中，函数可能需要临时的存储空间，程序员通常会通过

- 二，在执行任务的过程中，函数可能需要临时的存储空间。程序员通常会通过声明局部变量来分配这类临时空间，这些变量将在函数内部使用，完成函数调用以后，就无法再访问它们。

在将控制权转交给函数之前，编译器会插入代码，将函数参数放入栈帧内，并分配足够的内存，以保存函数的局部变量。鉴于栈帧的结构，该函数的返回地址也存储在新的栈帧内。使用栈帧使得递归成为可能，因为每个递归函数调用都有它自己的栈帧，这恰好将当前调用与前一次调用分隔开来。**这也是每个函数的汇编代码之前通常都有一段不属于方法反编译出来的用于分配栈空间的指令的原因。**

下面是调用一个函数时的详细操作步骤。

- （1）调用方将被调用函数所需的任何参数放入到该函数所采用的 **调用约定(知识点)** 指定的位置。如果参数被放到运行时栈上，该操作可能导致程序的栈指针发生改变。
- （2）调用方将控制权转交给被调用的函数，然后，返回地址被保存到程序栈或CPU寄存器中。
- （3）（函数的序言）如有必要，被调用的函数会配置一个栈指针，并保存调用方希望保持不变的任何寄存器值。
- （4）（函数的序言）被调用的函数为它可能需要的任何局部变量分配空间。一般，通过**调整程序栈指针在运行时栈上保留空间**来完成这一任务。
- （5）（函数的主体）被调用的函数执行其操作，可能生成一个结果。在执行操作的过程中，被调用的函数可能会访问调用函数传递给它的参数。如果函数返回一个结果，此结果通常被放置到一个特定的寄存器中，或者放置到函数返回后调用方可立即访问的寄存器中。
- （6）（函数的尾声）函数完成其操作后，任何为局部变量保留的栈空间将被释放。通常，逆向执行第（4）步中的操作，即可完成这个任务。
- （7）（函数的尾声）如果某个寄存器的值还为调用方保存（第（3）步）着，那么将其恢复到原始值。这包括恢复调用方的帧指针寄存器。
- （8）（函数的尾声）被调用的函数将控制权返还给调用方。实现这一操作的主要指令包括x86RET和MIPS JR。根据所使用的调用约定，这一操作可能还会从程序栈中清除一个或多个参数。
- （9）调用方一旦重新获得控制权，它可能需要删除程序栈中的参数。这时可能需要对栈进行调整，以将程序栈指针恢复到第（1）步以前的值。

总结一下栈帧的重要知识点：

- 通过调用函数将函数参数存入栈中。调用函数必须存储被调用函数所需的参数，否则可能导致严重的问题。各个函数会选择并遵照某一特定的调用约定，以表明它们希望以何种方式接收参数。

- 调用约定指定调用方放置函数所需参数的具体位置。调用约定可能要求将参数放置在特定的寄存器、程序栈、或者寄存器和栈中
- 被调用函数完成其操作后，由谁负责从栈中删除这些参数。一些调用约定规定，由调用方负责删除它放置在栈中的参数，而另一些调用约定则要求被调用函数负责删除栈中的参数。

可以看出上述这些操作都涉及到 程序栈指针，理解维护程序栈指针stack pointer的完整性对于还原被模糊的IDA 反汇编代码有极大帮助。

常用函数调用约定（规定参数传递和返回值）

C调用约定(cdecl)

cdecl调用约定规定：调用方按从右到左的顺序将函数参数放入栈中，在被调用的函数完成其操作时，调用方（而不是被调用方）负责从栈中清除参数。

常见于x86体系中。要求调用函数从栈中删除参数，意味着你将经常看到：指令在由被调用的函数返回后，会立即对程序栈指针进行调整。如果函数能够接受数量可变的参数，则调用方非常适于进行这种调整，因为它清楚地知道，它向函数传递了多少个参数，因而能够轻松做出正确的调整。而被调用的函数事先无法知道自己会收到多少个参数，因而很难对栈做出必要的调整。

标准调用约定(_stdcall)

和cdecl调用约定一样，stdcall调用约定按从右到左的顺序将函数参数放在程序栈上。使用stdcall调用约定的区别在于：函数结束执行时，应由被调用的函数负责删除栈中的函数参数。

对被调用的函数而言，要完成这个任务，它必须清楚知道栈中有多少个参数，这只有在函数接受的参数数量固定不变时才有可能。因此，printf这种接受数量可变的参数的函数不能使用stdcall调用约定。

使用stdcall的主要优点在于，在每次函数调用之后，不需要通过代码从栈中清除参数，因而能够生成体积稍小、速度稍快的程序。

x86 fastcall约定

fastcall约定是stdcall约定的一个变体，它向CPU寄存器（而非程序栈）最多传递两个参数。传递给函数的前两个参数将分别位于ECX和EDX寄存器中。剩余的其他参数则以类似于stdcall约定的方式从右到左放入栈上。同样与stdcall约定类似的是、

在返回其调用方时，fastcall函数负责从栈中删除参数,注意只需要清除栈中的参数，寄存器中的参数无需处理。

c++调用约定

C++类中的非静态成员函数与标准函数不同，它们需要使用this指针，该指针指向用于调用函数的对象。用于调用函数的对象的地址必须由调用方提供，因此，它在调用非静态成员函数时作为参数提供。

Microsoft Visual C++提供thiscall调用约定，它将this传递到ECX寄存器中，并且和在stdcall中一样，它要求非静态成员函数清除栈中的参数。

GNU g编译器将this看成是任何非静态成员函数的第一个隐含参数，而在所有其他方面与使用cdecl约定相同。因此，对使用g编译的代码来说，在调用非静态成员函数之前，this被放置到栈顶，且调用方负责在函数返回时删除栈中的参数（至少有一个参数）。

其他调用约定

起这个标题的意义在于提醒大家并非只有以上几种标准约定，以及如果程序员不怕麻烦，完全可以通过汇编语言控制如何向他们创建的函数传递参数，因此在分析自定义汇编代码时请格外小心。

系统调用也有一种特殊的调用约定，在linux中可以使用int 0x80(对应的就是svc #0x80)或者sysenter启动系统调用，系统调用的传参方式是通过寄存器保存系统调用号来让内核找到对应调用的处理函数。

局部变量布局

存在规定如何向函数传递参数的调用约定，但不存在规定函数的局部变量布局的约定。

编译器的第一个任务是，计算出函数的局部变量所需的空间。编译器的第二个任务，则是确定这些变量是否可在CPU寄存器中分配，或者它们是否必须在程序栈上分配。至于具体的分配方式，既与函数的调用方无关，也与被调用的函数无关。

例如某个函数中声明了一个int变量和一个长度为64的char数组，那么这个函数的局部变量可以计算出至少需要76字节的栈空间，体现在汇编指令中，可能就是一个sub esp 76;

无论函数使用何种调用约定，它们的栈帧都完全相同。在栈帧中通过栈指针来定位变量在栈帧中的位置，因此栈指针的频繁变化代表着栈帧的频繁读取和压栈等操作。同时，不同汇编指令对栈帧的操作给栈指针所带来的影响也是不同，需要具体分析。

IDA中的栈帧视图

虽然栈帧是一个运行时概念，没有栈和运行中的程序，栈帧就不可能存在，但是二进制文件中包含配置每个函数的栈帧所需的全部代码。通过仔细分析这段代码，我们可以深入了解任何函数的栈帧的结构，即使这个函数并未运行。

IDA中的一些最复杂的分析，就是为了专门确定IDA反汇编的每个函数的栈帧的布局。在初始分析过程中，IDA会记住每一项push或pop操作，以及其他任何可能改变栈指针的算术运算，如增加或减去常量，尽其所能去监控栈指针在函数执行过程中的行为。

来简单实战分析一下应用刚才所学的知识：

```
void demo_stackframe(int a, int b, int c) {
  int x = c;
```

```
    char buffer[64];
    int y = b;
    int z = 10;
    buffer[0] = 'A';
    bar(z, y);
}
```

```
.text:00401090 ; ========= S U B R O U T I N E ==========================
.text:00401090
.text:00401090 ; Attributes: ❶bp-based frame
.text:00401090
.text:00401090 demo_stackframe proc near      ; CODE XREF: sub_4010C1+41↓p
.text:00401090
❹ .text:00401090 var_60          = dword ptr -60h
.text:00401090 var_5C          = dword ptr -5Ch
.text:00401090 var_58          = byte ptr -58h
.text:00401090 var_C           = dword ptr -0Ch
.text:00401090 arg_4           = dword ptr  0Ch
.text:00401090 arg_8           = dword ptr  10h
.text:00401090
.text:00401090                 push    ebp
.text:00401091                 mov     ebp, esp
.text:00401093                 sub     esp, ❷78h
.text:00401096                 mov     eax, [ebp+❺arg_8]
.text:00401099                ❻mov     [ebp+var_C], eax
.text:0040109C                ❼mov     eax, [ebp+arg_4]
.text:0040109F                ❼mov     [ebp+var_5C], eax
.text:004010A2                ❽mov     [ebp+var_60], 0Ah
.text:004010A9                ❾mov     [ebp+var_58], 41h
.text:004010AD                 mov     eax, [ebp+var_5C]
.text:004010B0                ❸mov     [esp+4], eax
.text:004010B4                 mov     eax, [ebp+var_60]
.text:004010B7                ❸mov     [esp], eax
.text:004010BA                 call    bar
.text:004010BF                 leave
.text:004010C0                 retn
.text:004010C0 demo_stackframe endp
```

这是一段x86的汇编指令，虽然不是arm，但是不妨碍学习。

从上到下，bp-based frame表示这个函数使用EBP寄存器作为栈指针。

图中的4即0x00401090部分就是刚才所说的IDA所提供的摘要栈帧视图，列出了栈帧内被直接引用的每一个变量，以及变量的大小和它们与帧指针的偏移距离。

0x00401090到0x00401093部分的指令即是函数的序言中分配栈帧空间，位置2可以发现在栈帧中分配了0x78字节的局部变量空间，编译器有时会用额外的字节填补局部变量空间，以确保栈帧内的特殊对齐方式（这里又涉及到了内存对齐。。）

局部变量位于被保存的返回地址之上，而函数参数则位于被保存的返回地址之下。

在本例中，局部变量var_C是一个4字节（dword）(此外还有db，dd等大小)变量，它位于所保存的帧指针之上，距离为12字节（[ebp-oCh]）。

可以发现在反汇编出的汇编指令中，并没有通过数字常量偏移量的栈指针例如ebp-12类似的内存引用，而是通过ebp+var_C这样的符号名称引用。IDA已经用与栈视图中的符号对应的符号名称，以及它们与栈帧指针的相对偏移量替代了所有常量偏移量。这样做是为了确保IDA生成更高级的反汇编代码。与处理数字常量相比，处理符号名称更容易一些。实际上，为方便我们记忆栈变量的名称，IDA允许任意修改任何栈变量的名称。例如，在反汇编代码清单中出现内存引用[ebp+arg_8]的地方，可以使用[ebp+10h]或[ebp+16]代替，可以在IDA中右键这些变量自由选择显示格式。

对照源代码分析IDA反汇编窗口的汇编指令，可以发现：

1. demo_stackframe使用了3个参数：a、b和c。它们分别与变量arg_0、arg_4和arg_8对应（尽管arg_0因没有被引用而被反汇编代码清单忽略了）。
2. 局部变量x由参数c初始化。因此，var_C与x对应，因为x由❻处的arg_8初始化。
3. 同样，局部变量y由参数b初始化。因此，var_5C与y对应，因为y由❼处的arg_4初始化。
4. 局部变量z与var_60对应，因为它由❽处的值10初始化。
5. 64字节的字符数组buffer从var_58处开始，因为buffer[0]由❾处的A（ASCII0x41）初始化。
6. 调用bar的两个变量被转移到❸处的栈中，而非压入栈。这是当前版本（3.4及更高版本）的gcc的典型做法。IDA认可这一约定并选择不为栈帧顶部的两项创建局部变量引用。

IDA搜索

IDA提供了Search Text文本搜索和Binary Search二进制搜索，文本搜索相当于对反汇编列表窗口进行子字符串搜索，如果需要搜索特定的二进制内容，如已知的字节序列，可以使用二进制搜索。

要搜索一个十六进制字节序列，应将搜索字符串指定为以空格分隔的两位十六进制值组成的列表，如CA FE BA BE，这与搜索ca fe ba be的结果相同，无论你是否选中Case-sensitive（区分大小写）选项都是如此。

要搜索内嵌的字符串数据（有效搜索十六进制窗口中的ASCII字符串），你必须将搜索字符串用引号括起来。使用Unicode Strings选项可以搜索你所搜索的字符串的Unicode版本。

Case-sensitive选项可能会引起混淆。在搜索字符串时，它的作用相当简单。如果没有选中Case-sensitive选项，则搜索hello时会出现HELLO。但是，在进行十六进制搜索时，如果没有选中Case-sensitive选项，情况会有所不同。如果对E9 41C3进

行不区分大小写的搜索，你会惊奇地发现，E9 61 C3出现在了搜索结果中。**这是因为，0x41对应于字符A，而0x61则对应于字符a，所以IDA认为这两个字符串相互匹配。**所以，即使你指定了进行十六进制搜索，但0x41却等同于0x61，因为你并没有指定进行区分大小写的搜索。在进行十六进制搜索时，如果希望将搜索限定为完全匹配，你必须选中Case-sensitive选项。在你搜索特定的操作码序列而非ASCII文本时，这点尤为重要。